The Last Line Effect

Beller, Moritz; Zaidman, Andy; Karpov, Andrey

DOI
10.1109/ICPC.2015.34

Publication date
2015

Document Version
Accepted author manuscript

Published in
Proceedings - 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015

The Last Line Effect

Moritz Beller, Andy Zaidman
Delft University of Technology,
The Netherlands
{m.m.beller,a.e.zaidman}@tudelft.nl

Andrey Karpov
OOO “Program Verification Systems,”
Tula, Russian Federation
karpov@viva64.com

***Abstract*—Micro-clones are tiny duplicated pieces of code; they typically comprise only a few statements or lines. In this paper, we expose the “last line effect,” the phenomenon that the last line or statement in a micro-clone is much more likely to contain an error than the previous lines or statements. We do this by analyzing 208 open source projects and reporting on 202 faulty micro-clones.**

I. Introduction

Software developers oft need to repeat one particular line of code several times in succession with only small alterations, like in this example from TrinityCore:[1]

Example 1. TrinityCore

```
1 x += other.x;
2 y += other.y;
3 z += other.y;
```

The 3D-coordinates of the `other` object are added onto the member variables representing the coordinates `x, y, z`. However, the last line in this block of three similar lines contains an error, as it adds the `y` coordinate onto the `z` coordinate. Instead, the last line should be

```
3 z += other.z;
```

Another example from the popular web browser Chromium[2] shows that this effect also occurs in similar statements within one single line:

Example 2. Chromium

```
1 std::string host = ...;
2 std::string port_str = ...;
3 if (host != buzz::STR_EMPTY &&
      host != buzz::STR_EMPTY)
```

Instead of comparing twice that `host` does not equal the empty string, in the last position, `port_str` should have been compared:

```
3 if (host != buzz::STR_EMPTY &&
      port_str != buzz::STR_EMPTY)
```

Lines 1–3 from Example 1 are similar to each other, as are the statements in the if clause in line 3 from Example 2. We call such an extremely short block of almost identically looking, repeated lines or statements a *micro-clone*. Through our experience as software engineers and software quality consultants, we had the intuition that *the last line or statement in a micro-clone is much more likely to contain an error than the previous lines or statements*. The aim of this paper is to verify whether our intuition is indeed true, leading to two research questions:

RQ 1 Is the last line in a multi-line micro-clone more likely to contain an error?

RQ 2 Is the last statement in a single-line micro-clone more likely to contain an error?

As recurring micro-clones are common to most programming languages, the presence of a last line effect can impact almost any programmer. If we can prove that the last of a series of similar statements is more likely to be faulty, programmers and code reviewers know which code segments to give extra attention to. This can increase software quality by reducing the amount of errors in a program. When we posted a popular science blog entry[3] about the last line effect, it was picked up quickly and excitedly in Internet fora.[4] Many programmers agreed to our observation, often assuming a psychological reason to cause the effect, namely that programmers think they finished a change one step too early.

A natural way to come up with code like in Examples 1 and 2 is to write the first part, copy-and-paste it the necessary number of times, and then adapt the pasted fragments. Copy-and-paste is one of the most widely used idioms in the development of software [1]. It is easy, fast, hence cheap, and the copied code is already known to work. Though often considered harmful, sometimes (micro-)cloning is in fact the only way to achieve a certain program behavior, like in the examples above.

A number of clone detection tools have been developed to find and possibly remove code clones [2], [3]. While these automated clone detection tools have produced very strong results down to the method level, they are ill-suited for recognizing micro-clones in practice because of an abundance of false-positives. Therefore, we currently have no understanding of the error proneness of a single line or statement in a micro-clone, and, to the best of our knowledge, no similar study has ever been conducted.

Our paper closes this gap by

- introducing the term micro-clone.
- introducing the static analysis tool PVS-Studio that is able to reliably detect faulty micro-clones, which cannot be found with traditional clone detection.

[1]TrinityCore is a popular opensource framework for the creation of Massively Multiplayer Online Games (MMOGs), www.trinitycore.org.

[2]Chromium is the open-source part of Google Chrome, www.chromium.org

[3]www.viva64.com/en/b/0260

[4]www.reddit.com/r/programming/comments/270orx/the_last_line_effect

TABLE I
ERROR TYPES FROM PVS-STUDIO AND THEIR DISTRIBUTION.

PVS Error Code	Description[5]	#multiline warnings	#within line warnings	Σ#
V501	There are identical sub-expressions to the left and to the right of the 'foo' operator.	93	88	181
V519	The 'x' variable is assigned values twice successively. Perhaps this is a mistake.	10	0	10
V537	Consider reviewing the correctness of 'X' item's usage.	8	0	8
V524	It is odd that the body of 'Foo_1' function is fully equivalent to the body of 'Foo_2'.	3	0	3
V525	The code containing the collection of similar blocks. Check items X, Y, Z, ... in lines N1, N2, N3, ...	0	1	1

- manually investigating the error proneness of 202 micro-clones in 208 well-known open-source systems (OSS).

Our findings show that in micro-clones similar to Examples 1 and 2, the last line or statement is significantly more likely to contain an error than any other preceding line or statement.

II. METHODS

In this section, we shortly describe traditional clone detection, why it is not suitable for the recognition of micro-clones, and how we circumvented this problem.

As Examples 1 and 2 demonstrate, the code blocks that we study in this paper contain "syntactically identical cop[ies]; only variable, type, or function identifiers have [...] changed." [4] This makes them *extremely short type-2 clones* (usually shorter than 5 lines or statements, see tables II and III), which we refer to as *micro-clones*.

A. Why Current Clone Detectors Are Not Suitable

Traditional code clone detection works with a token-, line-, abstract syntax tree (AST), or graph-based comparison [4]. However, to reduce the number of false-positives, all approaches are in need of specifying a minimal code clone length for their unit of measurement (be it tokens, statements, lines or AST nodes) when applied in practice. This minimal clone length is usually in the range of 5-10 units [2], [5], which makes it too long to detect our micro-clones of length 2 to 4 units (see tables II and III).

B. How We Found Faulty Micro-Clones Instead

As clone detection is not able to reliably detect micro-clones in practice, we devised a different strategy to find them. Our RQs aim not at finding all possible micro-clones, but only the ones which are faulty. With this additional constraint, we could design and implement a handful of powerful analyses based on simple textual identity. These are able to find faulty code that is very likely the result of a micro-clone. Table I describes the five analyses that found micro-clones in our study. For example, the analysis V501 simply evaluates whether there are identical expressions next to certain logical operators. If so, these are at best redundant and therefore cause a maintenance problem, or at worst present a real bug in the system.

[5]For a more detailed description, refer to www.viva64.com/en/d

III. STUDY SETUP

In this section, we describe how we set-up our empirical study and which study objects we selected.

A. Study Design

Our study consists of four easily replicable steps.

1) Run the static analysis checker PVS-Studio on our study objects, with all checks enabled. PVS-Studio is a commercial static analysis tool developed by OOO Program Verification System and incorporates dozens of static analyses from clone detection to anti-usage-patterns of C-specific library functions. For replication purposes, a free trial of PVS-Studio is publicly available.[6]
2) Inspect the corpus of warnings from PVS-Studio and remove false-positives and warnings not related to micro-cloning.
3) For each faulty micro-clone, count the total number of lines (RQ1) or statements (RQ2) and denote which lines or statements are faulty.
4) Naively, we assume each line has the same likelihood $1/n$ of containing an error (H_0), independent of its position in an n-line long faulty micro-clone. For example, lines 1 and 2 in a 2-liner clone each have a 0.5 probability of containing an error. However, if we can show that the error distribution per line from step (3) significantly differs from such a uniform distribution on a $\sigma = 0.05$ significance level, we reject H_0 and assume a non-uniform error distribution. For each micro-clone length n, we use Pearson's χ^2 test with $n - 1$ degrees of freedom to compare our empirical distribution's goodness-of-fit to a $1/n$-equipartition.

B. Study Objects

To ensure the replicability and feasibility of our study, we focused on well-known open-source systems. Among the 208 OSS, we found instances of defective micro clones in such renowned projects as the music editing software Audacity (2 findings), the web browsers Chromium (9) and Firefox (9), the XML library libxml (1), the databases MySQL (1) and MongoDB (1), the C compiler clang (14), the egoshooter Quake III (3), the rendering software Blender (4), the 3D visualization toolkit VTK (7), the network protocols Samba (3) and OpenSSL (1), the video editor VirtualDub (3), and the programming language PHP (1).

IV. RESULTS

In this section, we deepen our understanding of faulty micro-clones by example and statistical evaluation.

A. In-Depth Investigation of Findings

We ran the complete suite of all PVS-Studio analyses on our 208 OSS from mid-2011 to December 2014. Andrey Karpov, a professional software consultant, manually sorted-out false positives, so that 1,683 potentially interesting warnings with 148 different error codes remained.[7] We manually investigated

[6]www.viva64.com/en/pvs-studio-download
[7]Our raw and analyzed data: dx.doi.org/10.6084/m9.figshare.1313697.

all 1,683 and found that 202 warnings with five distinct error codes were related to micro-cloning. Table II shows the error-per-line distribution of our 108 micro-clones consisting of several lines, and table III that of our 94 micro-clones within one single line. Cells with gray background are non-sensible (i.e. for a micro-clone of 2 lines, no error can occur in line 3). The yellow diagonal highlights errors in the last line or statement.

B. Warning Types By Example

In the following, we report on the details of 202 PVS-Studio-generated warnings by providing representative examples to convey a better intuitive understanding of our findings.

1) V501: As table I shows, the majority of micro-clone warnings are of type V501. A prime example for a V501-type warning comes from Chromium:

Example 3. Chromium

```
return
  !profile.GetFieldText(AutofillType(
      NAME_FIRST)).empty() ||
  !profile.GetFieldText(AutofillType(
      NAME_MIDDLE)).empty() ||
  !profile.GetFieldText(AutofillType(
      NAME_MIDDLE)).empty();
```

In this one-liner micro-clone the second and third cloned statement are lexicographically identical but connected with the logical OR-operator (`||`), thus representing a tautology. Instead, the boolean expression misses to take into account the surname (`NAME_LAST`), an example of the last statement effect in this tricolon.

2) V519: When setting the value of a variable twice in succession, this is typically either unnecessary – and therefore a maintenance problem because it makes the code harder to understand as the first assignment is not effective –, or an outright bug. In this V519 example from MTASA, `m_ucRed` is assigned twice, but the developers forgot to set `m_ucBlue`.

Example 4. MTASA

```
m_ucRed = ucRed; m_ucGreen = ucGreen; m_ucRed =
    ucRed;
```

3) V537: An illustrative example for a V537 finding comes from Quake III, where PVS-Studio alerts us to review the use of `rectf.X`:

Example 5. Quake III

```
rect->X = roundr(rectf.X);
rect->Y = roundr(rectf.X);
```

Indeed, the rectangle is falsely assigned the rounded value of `rectf.X` in its `y` coordinate in the second (i.e. last) line of this micro-clone.

4) Counterexample: However, not in all instances of an erroneous micro-clone does the problem lie in the last line or statement. Take this rare counterexample from Chromium, which we counted to the 7 instances of an error in line 2 of a three-liner micro-clone (see table II):

Example 6. Chromium

```
1 if (std::abs(data_[M01] - data_[M10]) > epsilon ||
2     std::abs(data_[M02] - data_[M02]) > epsilon ||
3     std::abs(data_[M12] - data_[M21]) > epsilon)
```

TABLE II
ERROR DISTRIBUTION FOR MICRO-CLONES WITH > 1 LINE.

		#total lines									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**>10**
#errors in line 1		5	0	0	2	0	0	0	0	0	...
2		*47*	7	3	1	1	0	0	0	0	...
3			*12*	3	0	1	1	0	0	0	...
4				*9*	0	0	0	0	0	0	...
5					*3*	0	0	1	0	0	...
6						*2*	0	0	0	0	...
7							*1*	0	0	0	...
8								*0*	1	0	...
9									*2*	1	...
10										*0*	...
Σ	0	52	19	15	6	4	2	1	3	1	5
$\Sigma\Sigma$				108							
χ^2		33.9	11.5	11.4	5.7						
p		5×10^{-9}	0.003	0.009	0.225						

TABLE III
ERROR DISTRIBUTION FOR MICRO-CLONES WITHIN ONE LINE.

		#total statements				
	1	**2**	**3**	**4**	**5**	**> 5**
#errors in statement 1		0	0	0	0	0
2		*67*	3	2	0	0
3			*15*	0	0	0
4				*7*	0	0
5					*0*	0
Σ	0	67	18	9	0	0
$\Sigma\Sigma$			94			
χ^2		67	21	15		
p		2×10^{-16}	2×10^{-5}	0.002		

In line 2, the engineers deducted `data_[M02]` from itself. Instead, they meant to write:

```
2     std::abs(data_[M02] - data_[M20]) > epsilon ||
```

C. Statistical Evaluation

For each column in tables II and III, we performed a Pearson's χ^2 test on a $p = 0.05$ significance level to show whether the individual distributions are non-uniform. Results, reported in the last two rows, are only meaningful for micro-clone lengths with enough empirical observations, which are columns 2, 3 and 4 in both tables.

For RQ1 and RQ2, we got significant p-values for micro-clones consisting of 2, 3, and 4 lines or statements, respectively ($p < 0.05$). This means that we can reject the null hypothesis that errors are uniformly distributed across statements or lines. Instead, the distribution is significantly skewed towards the last line or statement. We would expect similar findings for longer micro-clones, but these were too rare to derive statistically valid information.

We can summarize the results across micro-clone lengths into the two events "error not in last line or statement" and "error in last line or statement", shown in table IV. Our absolute counts show that in micro-clones similar to Example 1, the last line is more than twice as likely to contain a fault than all previous lines taken together. When looking at the individual line length in table II, the last line effect is even as high as a nine-fold increased error-proneness for the oft-appearing clone lengths 2 and 4. The results for cloned

TABLE IV
SUMMARIZED RESULTS.

	multi-line micro-clone	one-line micro-clone
#errors in last line/stmt.	*76*	*89*
#errors not in last line/stmt.	32	5
Σ	108	94

statements in micro-clones within one line, like Example 2, are stronger still: We found the last statement to be 17 times more faulty than all other statements taken together. In fact, for the 67 micro-clones consisting of two statements, the last statement was always the faulty one.

In total, our findings confirm the presence of a strong last line and last statement effect, accepting both RQ1 and RQ2. We assume the effect to be caused by and large through copy-and-pasting code, and that developers have a psychological tendency to think changes of similar code blocks are finished earlier than they really are. This way, they miss one critical last modification.

D. Usefulness of Results

Having unveiled a large number of potential bugs in OSS, we wanted to (a) help the OSS community and (b) see if our findings represented bugs that would be worth fixing in reality. We approached the OSS projects by creating issues with our findings in their bug trackers. Many of our bug reports lead to quality improvements in the projects, like fixing the validation bug from Example 2 in Chromium.[8] The search query `pvs-studio bug | issue`[9] delivers numerous successful bug fixes in Firefox, libxml, MySQL, Clang and many other projects.

V. Threats to Validity

In this section, we show internal and external threats to the validity of our results, and how we mitigated them.

One internal threat concerns how to determine in which line the error lies. Given Example 2, any of the two statements could be counted as the one containing the duplication. However, reading and writing source code typically happens from top to bottom and from left to right [6]. Therefore, the only natural assessment is to count the line as problematic according to this strict left-right/top-down-reading order: In Example 2, only when we read the second statement do we know it is a duplicate of the first. We hence count the second statement as the one containing the error. Moreover, in many cases, as in Example 2, the program context around the micro-clone (here the definition order of the variables `host` first and then `port_str`) imposes a natural logical order for the remainder of the program (first check `host`, then `port_str` in line 3). Because counting lines is a well-defined task under these circumstances, we are not concerned about interrater reliability.

[8]codereview.chromium.org/7031055
[9]www.google.com/search?q=pvs-studio+bug+|+issue

An external factor that threatens the generalizability is that PVS-Studio is specific to C(++). C is one of the most commonly used languages [7]. Therefore, even if our results were not generalizable, they would at least be valuable to the large C community. However, our findings typically contain language features common to most programming languages, like the variable assignments, if clauses, boolean expressions and array use in Examples 1 to 5. Almost all programming languages have these constructs. Thus, we expect to see analogous results in at least imperative languages such as Java, JavaScript, C#, PHP, Ruby, or Python. While our overall corpus of findings is quite large, the average number of ~1 warning per project is rather small. This could be because PVS-Studio's analyses for defective micro-clones are not exhaustive, and that the projects we studied are stable, production systems with a mature code base containing relatively little trivial errors.

VI. Future Work & Conclusion

In this section, we describe possible extensions of our study and summarize our initial results.

Because our study focuses on faulty micro-clones, we cannot make predictions about how many of all micro-clones are affected by the last line effect. A promising future research direction is to develop a clone detector that can reliably detect all micro-clones, and then to see how many are actually defective. This gives an indication of the scale of the problem at hand. Moreover, while our investigation provides a first theory about the last line effect, the assumed psychological reasons for its existence open up another vast and fruitful field for further study.

In 208 open source projects, we found 202 faulty micro-clones. Our analysis shows that there is a strong tendency for the last line, and an even stronger for the last statement to be faulty, called the last line effect. Because of it, we advise programmers be extra-careful when reading, modifying, code-reviewing, or creating the last line and statement of a micro-clone, especially when they copy-and-pasted it.

References

[1] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Proc. Int'l Symp. on Empirical Software Engineering (ISESE)*. IEEE, 2004, pp. 83–92.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[3] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[4] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, 2007.

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 485–495.

[6] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proc. Int'l Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 378–389.

[7] L. Meyerovich and A. Rabkin, "Empirical analysis of programming language adoption," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 1–18.