ABSTRACT
This paper explores the need for ‘history-aware’ searches, by experimentally showing a development process that includes code fragments which disappear at a revision and appear again at a later revision. Some of these code re-appearances are not a result of a revert command of a version control system, but a result of a developer who copied a code fragment from old source files.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]: [Restructuring, reverse engineering, and reengineering]

General Terms
Measurement, Experimentation

Keywords
Code Search, Mining Repositories, Code Clone

1. INTRODUCTION
Developers have been spending more and more time trying to understand the structure of a software product and searching APIs or usages. The goal of the Remics project [8] is to provide a toolkit, which enables a developer to rapidly develop search/mining tools on source code and other artifacts (e.g. configuration[4]), by mixing programming tools (e.g. parser, static analyzer) algorithms (e.g. diff, tree/graph manipulation, frequent itemset mining[5]) and visualizations (e.g. dot-plotting[1], power graph[5], dendrogram[5]), depending on an individual special needs in his/her own development process.

As an example of such a customizable search tool, the Remics project provides a ‘history-aware’ grep tool named hag[8]. (Some studies of code searches, such as CodeDepot[2] by Yunwen Ye, are moving in this direction.) The history-aware search tool searches in a VCS(version control system) repository. The search results include ‘old’ code fragments that ordinal search tools miss, when the code fragments have been removed from the latest revision through various development tasks such as enhancement of functionalities, evolution of the product, and refactoring.

This paper reveals why these kinds of history-aware searches can be useful in the software development process, in comparison with normal search tools that scan just a current revision of source code.

2. QUESTIONS
In this paper, when a code fragment appears in all revisions, \(R, R + 1, ..., S\), define the code fragments to be continuous through rev. \(R\) to \(S\). When a code fragment appears in rev. \(R\) and \(S\), but does not appear in some of \(R + 1, ..., S - 1\), that is, the code fragment is not continuous, define skipped revisions to be the revisions (between \(R\) and \(S\)) that do not include the code fragment.

The research questions are follows:

EXISTENCE Do non-continuous code fragments exist?

CAUSE Are such non-continuous code fragments generated by a developer’s copying-and-pasting from the older revisions to the current revision?

If both of the answers are positive, a code search in a historical code body can be helpful to the developers engaged in such a development process because the developers will need to search for some code fragments not in the current revision, but in the historical code body.

3. DETECTING NON-CONTINUOUS CODE
To detect non-continuous code and skipped revisions of the code, use the algorithm shown in the following pseudo code.

```pseudo
def mark(line, revision, d):
    SC[revision][d] = SC[revision][d] U {line}
def find(new_lines, new_revision, previous_lines, previous_revision):
    added_lines = new_lines - previous_lines
    unchanged_lines = previous_lines
    for line in added_lines:
        if line in unchanged_lines:
            mark(line, new_revision, 0)
        else:
            for d in range(1, new_revision):
                if line in previous_lines[:d]:
                    mark(line, new_revision, d)
    return SC[new_revision]
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
4. PRELIMINARY EXPERIMENT

The above detection method was implemented as a program and applied to a repository of FreeBSD[7], which includes over 190,000 revisions. The target files are C files (.c and .h) in the repository. The implementation follows the above algorithm, except for putting a limitation on skipped revisions up to 10,000 as a relaxation of space complexity. That is, the implementation works as if line 9 of the above algorithm has been replaced by line `for (d in 1 .. R - 10000)'.

To visualize the detected skips over revisions as a comprehensive graph, prepare a metric to measure the amount of lines of non-continuous code fragments, weighted by skipped revisions. The metric is defined as:

$$ cd(R) = \sum_{d=2}^{R-1} SC_{R,d} $$

where $SC_{R,d}$ represents the count of lines in $SC[R][d]$. Each line of $SC[R][d]$ (d ≥ 2) is skipping $d - 1$ revisions, that is, the line appears in rev. $R - d$, is skipping $R - d + 1, ..., R - 1$, and re-appears in $R$. Note that the values of $SC[R][0]$ and $SC[R][1]$ are not counted in the metric $cd(R)$, because these values do not mean that some revisions are being skipped.

Figure 1 shows the metric $cd(R)$ value and the count of `svn revert' commands found in a change log (#revert). For comprehensibility, each value of $cd(R)$ and #revert is summarized for 1000 revisions. This graph also shows which revision is tagged as a major or minor release.

First, the series of revisions looks divided into a first half and a second half, somewhere before version 5.0.0. As for the first half, $cd(R)$ has more peaks than the second half and $cd(R)$ does not seem to have any relation with the release. As for the second half, peaks of $cd(R)$ frequently occur at the same time of releases, except for the large peak at rev. 180K.

A consideration of the assumptions:

EXISTENCE With the graph, we can safely conclude that non-continuous code fragments exist.

CAUSE If these non-continuous code fragments are the results of revert changes (dispose of the current revision and commit to an old revision as a newer one), then the curve of $cd(R)$ will likely become similar to the curve of #revert. However, the graph does not seem to show this. A more reasonable cause of the non-continuous code fragments is: experimental code was removed once from the main branch. After a major version is released, such experimental code is again introduced to the code base.

These causes are not validated in any sense, currently, but will be by collecting further data, or with another analysis technique [3, 6].

5. ACKNOWLEDGMENTS

I am deeply grateful to the MSR conference organizers for providing useful repository data, which is used in the preliminary experiment in Section 4. This work was supported by Grant-in-Aid for challenging Exploratory Research, KAKENHI (70415660).

6. REFERENCES