An Empirical Study of End-user Programmers in the Computer Music Community

Gregory Burlet
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
gburlet@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
abram.hindle@ualberta.ca

***Abstract*—Computer musicians are a community of end-user programmers who often use visual programming languages such as Max/MSP or Pure Data to realize their musical compositions. This research study conducts a multifaceted analysis of the software development practices of computer musicians when programming in these visual music-oriented languages. A statistical analysis of project metadata harvested from software repositories hosted on GitHub reveals that in comparison to the general population of software developers, computer musicians' repositories have less commits, less frequent commits, more commits on weekends, yet similar numbers of bug reports and similar numbers of contributing authors. Analysis of source code in these repositories reveals that the vast majority of code can be reconstructed from duplicate fragments. Finally, these results are corroborated by a survey of computer musicians and interviews with individuals in this end-user community. Based on this analysis and feedback from computer musicians we find that there are many avenues where software engineering can be applied to help aid this community of end-user programmers.**

I. Introduction

Musicians currently have a multitude of tools at their disposal for creating music. Before the digital age, music was created by physically manipulating a conventional musical instrument to produce sound. With the advent of synthesizers, samplers, and sequencers came a rapid paradigm shift in the music creation process that increasingly challenged the definition of "instrument", the role of musicians, and their technical proficiency. Many musicians have embraced the technical challenges arising from the changing landscape of the music creation process, forming a relatively small [1] but tight-knit community of individuals invested in developing music-making applications on computers or mobile devices.

Computer musicians are end-user programmers who "face software engineering challenges that are similar to their professional counterparts" [2]. As end users, computer musicians still have to be mindful of data flow, data structures, debugging strategies, testing, and calls to application programming interfaces [3]. As a consequence of having to learn a variety of skills from computing science, music theory, electrical engineering, and software engineering, the technical aptitude of computer musicians is quite varied. Novice programmers in specialized domains, such as computer music, who are not formally trained in software engineering tend to gravitate towards visual programming languages [4].

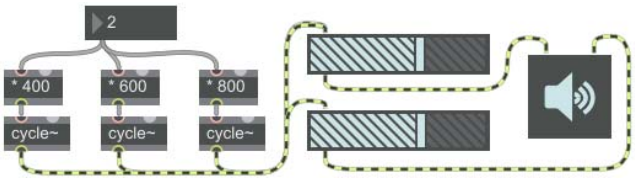


Fig. 1: Max/MSP patch consisting of a triad of oscillators with stereo volume control.

Visual music programming languages allow users to programmatically generate sounds and respond to human-computer interaction devices by spatially arranging rectangular objects on the screen and connecting them with lines called *patchcords*. Objects represent functions whose purpose is to manipulate audio signals or other data structures. Objects have inlets and outlets, which correspond with parameters and return values, respectively. A *patch* is a file containing a collection of connected objects that perform some function that is typically musical in nature. Computer music applications typically consist of several interconnected patches. There are several proprietary and opensource visual music programming languages that are used by the community of computer musicians including, but not limited to, *Max/MSP* [5], *Pure Data*, *AudioMulch*, and *OpenMusic*. This paper will focus on the analysis of Max/MSP and Pure Data patches. Pure Data is an opensource alternative to Max/MSP, having similar objects and functionality as its proprietary counterpart. Figure 1 displays a Max/MSP patch consisting of three `cycle~` objects referred to as oscillators, which each produce a periodic sine wave.

We currently know little about how computer musicians program, share, and collaborate on patches developed in visual music programming languages such as Pure Data and Max/MSP. Furthermore, we are unaware if computer musicians use software engineering tools such as source control repositories or bug trackers and where they seek help with programming syntax and semantics. We also lack understanding of issues that computer musicians face while developing musical applications, if they potentially lack development and software engineering tools, and the extent to which current software engineering practices and tools could positively impact the development process of computer musicians.

978-0-7695-5594-2/15 $31.00 © 2015 Crown Copyright
DOI 10.1109/MSR.2015.34

In response to these issues, the objective of this research is to analyze the software development practices of computer musicians when creating musical applications in the Max/MSP and Pure Data visual programming languages, and investigate to what extent their development practices differ from the general population of software developers. The intent of this analysis is to reveal software engineering tools that could aid the development practices of computer musicians, and more broadly, other end-user communities using visual programming languages. To this end, Section III presents a statistical analysis of project metadata harvested from computer musicians' source control repositories hosted on *GitHub* [6]. In Section IV, clone detection is performed on Max/MSP and Pure Data patches to reveal which code structures are often used by computer musicians. Finally, a survey of computer musicians is conducted in Section V and is supplemented with computer musician interviews to provide qualitative evidence to support the quantitative results in the paper.

II. Related Work

A. End-user Visual Programming

End users constitute a large demographic of programmers [7]. The study of end-user programmers and their communities is an active research topic, which aims to create tools to aid development and improve software quality by studying how end users program [3]. There has recently been a fundamental shift in the perception of end users as consumers to being participators [8], demanding software frameworks and programming environments that are easily extensible and malleable to their needs [9]. Ko *et al.* [3] state that end-user programmers have different goals than professional programmers who are paid to develop, test, deploy, and maintain software over a period of time; they often develop programs using special-purpose languages to achieve a personal goal in their domain of expertise [10]. Considering these definitions, computer musicians fall into the category of *end users* because they often use specialized music-oriented programming languages for their personal creative musical endeavours.

Although specialized audio libraries exist for languages such as C or Java, these are general-purpose programming languages, which are arguably ill-suited for the specific needs of computer musicians [11] who require flexibility in the combination of concepts and tools for their creative compositions [12], [13]. Among the specialized music-oriented programming languages used by computer musicians are visual programming languages such as Max/MSP or Pure Data, which "represent the programmable world as graphical metaphors" [14] to lower the learning curve. These real-time programming environments provide immediate visual and auditory feedback to the programmer, allowing them to test for and hypothetically eradicate bugs at run-time [4]. Burnett [4] also noted that although some users of these visual languages are professional programmers, many are end users with no formal training in software engineering methods. Furthermore, they often face several barriers to entry when learning to program in these languages [15], [16].

In an effort to understand how end-user programmers create and share software artifacts, Stolee *et al.* [2] analyzed the Yahoo! Pipes end-user community and Ko *et al.* [3] investigated a variety of other end-user communities such as children, system administrators, web designers, and so forth. Bogart *et al.* [17] analyzed the reuse and extension of software artifacts created by end-user programmers developing in CoScripter, a web-browser macro recording language, and found that end users often reused similar scripts shared by the community for their own endeavours. An exceptional amount of effort has also been directed towards the analysis of end user interactions with spreadsheet applications [18] and the development of tools to aid these users [19], [20]. However, no empirical end-user studies have been conducted on the community of computer musicians and their software artifacts.

B. Mining Software Repositories

One facet of analysis for computer music end-user programmers is how they use software engineering tools such as source control repositories and bug trackers. Significant effort has been devoted to mining *Git* software repositories in order to analyze software artifacts, calculate project development metrics, and study authorship and collaboration tendencies. Hosting over 6.8 million public Git repositories, *GitHub* is among the most popular collections of publicly available software projects on the internet [21]. However, Bird *et al.* [22] warn that mining the wealth of information in Git repositories on GitHub comes with its own set of perils to be wary of. For example, Git allows commit *rebasing*, which obfuscates the true software development history. Moreover, an analysis of Git repositories hosted on GitHub revealed that most repositories have very few commits, are relatively inactive, and do not necessarily contain software artifacts [21]. Despite these perils, the mining software repositories (MSR) research community has gone through great lengths to harvest the publicly available software repository data hosted on GitHub and publish the resulting dataset called *GHTorrent* [23], [24].

C. Clone Detection

After mining a set of computer music repositories, it is worthwhile to analyze the software artifacts produced by this end-user community. One facet of analysis is clone detection. Software clones are duplicates or near-duplicates of code entities. The detection of clones in a software system can promote code reuse, refer novice programmers to existing related code, as well as locate software entities that may benefit from refactoring. Several clone detection algorithms have been proposed in the literature [25]–[27] and operate by first setting the granularity of detected clones. For example, one might be interested in clones that are exact replicas of other code entities, or clones that are identical except for changes in literal values, identifier names, layout, and comments. Next, the relevant information is extracted from each code entity under analysis and a suffix tree [28], [29], dynamic pattern matching [30], or hash comparison [31]–[33] algorithm is used to detect matches.

Focusing on clone detection in visual programming languages, several research studies have adapted conventional clone detection algorithms to locate clones in Matlab Simulink models. Simulink is a visual programming language used to mathematically model systems in which blocks (objects) represent mathematical functions that are connected together with lines to form a directed graph. Clones are detected in these models using graph search algorithms with heuristics to improve computational complexity [33]–[37].

Matlab Simulink is conceptually similar to the Max/MSP or Pure Data visual music programming languages. Contrary to Simulink models, the spatial arrangement of objects in Max/MSP or Pure Data patches potentially affects the precedence of operations. Taking this into consideration, Gold *et al.* [38] propose a clone taxonomy and use pairwise comparison of Max/MSP patch subgraphs to locate clones. This clone detection algorithm was run on 68 preprocessed Max/MSP tutorial patches supplied with the software and found that 86% of connected objects were clones in the lowest level of granularity. Gold *et al.* [38] did not consider Pure Data patches or patches developed by the computer music community.

III. Mining Git Repositories

In an effort to understand if computer musicians develop software differently than the general population of programmers, a statistical analysis of project metadata harvested from Git repositories hosted on GitHub has been performed. From this analysis we hope to uncover how computer musicians' interactions with source control repositories and bug trackers differ from the general population of developers on GitHub.

A. Datasets

The GHTorrent database of extracted Git repositories [23], [24] is queried to compile two datasets. The first dataset consists of 819 computer music repositories and was formed by querying the *language* field in the GHTorrent MySQL database to retrieve repositories that predominantly contain Max/MSP or Pure Data files. Table I provides an overview of the scale of the compiled dataset. Notably, Pure Data projects are over-represented in the compiled dataset; there are almost four times as many repositories containing predominantly Pure Data patches as there are Max/MSP patches on GitHub. The second dataset consists of 819 general software repositories collected by random sampling using the following methodology: 819 random GHTorrent project identifiers were generated; if a repository was unable to be cloned due to deletion or renaming, a new project identifier was resampled. The resulting dataset represents a random sample of GitHub repositories from the general population of software developers. No computer music repositories were present in the random sample dataset. The randomly sampled dataset has an average of 831 files per repository. Projects in this dataset are written in a variety of programming languages including Java, JavaScript, Perl, Lua, Matlab, R, Python, Ruby, C, and derivatives of C. These code repositories serve a variety of purposes such as vim plugins, online games, low-level data structure implementations, and jQuery plugins. For each extracted Git repository, several attributes of interest are calculated: number of commits, number of weekend or weekday commits, frequency of commits, number of issues, and the number of unique authors.

TABLE I: Computer music dataset metrics

	Max/MSP	Pure Data	Total
Repositories	168	651	**819**
Patches	15,016	103,465	**118,481**
Objects	565,705	2,521,573	**3,087,278**
Comment Objects	86,127	419,109	**505,236**
Mean Objects per Patch	37.67	24.37	**26.06**
Patchcords	508,295	1,973,871	**2,482,166**

B. Hypotheses

The following hypotheses regarding the software development practices of this end-user community were posed as open questions to 15 computer musicians who agreed to participate in an interview. Section V presents the interview recruitment strategy in detail. Relative to the general population of software developers we hypothesize that:

1) Computer musicians make less commits.

Responses: 7 computer musicians agreed with this hypothesis, 1 disagreed, and 7 were uncertain.

Rationale: Advocates of the hypothesis argued that:

- "Computer musicians are used to working solo ... there's less incentive to keep source control repositories up to date."
- "The community [of computer musicians] resembles a musical community more than a development community. The culture of sharing ideas is different."
- "It is really hard to differentiate minor changes from actual, structural changes in your code. After a while, it gets pedantic to commit changes like 'Changed parameter X so it sounds more like a guitar'."

The computer musician who disagreed argued that music projects are no different than general opensource projects.

2) Computer musicians commit more on the weekend.

Responses: 4 computer musicians agreed with this hypothesis, 2 disagreed, and 9 were uncertain.

Rationale: Advocates of the hypothesis argued that "many others have day-jobs unrelated to computer music, meaning their projects are more hobbyist in nature, which may mean the weekend is the only opportunity they have to make solid contributions". Those that disagreed argued that the number of weekend contributions would differ "only if computer music is your hobby".

3) Computer musicians make commits less frequently.

Responses: 5 computer musicians agreed with this hypothesis, 1 disagreed, and 9 were uncertain.

Rationale: Computer musicians advocating this hypothesis argued that since computer music software development is

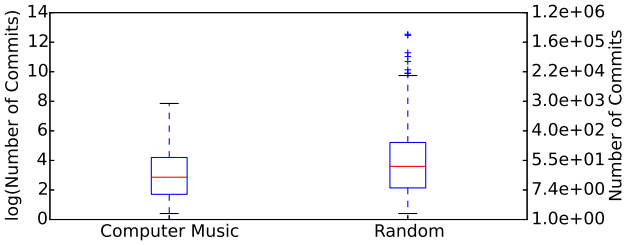


Fig. 2: Box plots of the log commit counts for the sample of computer music repositories (left) and the random sample of repositories (right).

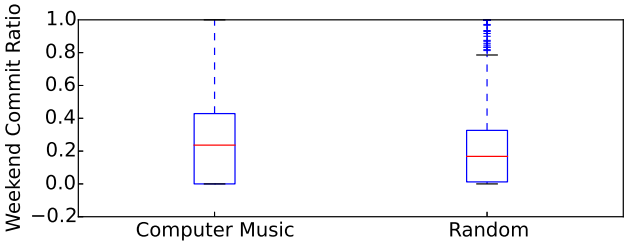


Fig. 3: Box plots of the proportion of commits that occur on weekends for the sample of computer music repositories (left) and the random sample of repositories (right).

often a hobby, commits to repositories are less frequent: "Computer music does tend to be a field in which our output is not particularly commercial (making it a free-time activity for a lot of us)." The computer musician who disagreed argued that music-oriented projects are no different than general opensource software projects.

4) Computer musicians report less issues (bugs).

Responses: 7 computer musicians agreed with this hypothesis, 3 disagreed, and 5 were uncertain.

Rationale: Advocates of the hypothesis argued that:

- "It is hard to describe some bugs in words. How do we solve 'my synthesizer sounds too bright ... ?'"
- "Many work solo, and the incentive to learn and update bug trackers is not so urgent."
- "So many computer music people are not initiated into the world of software development. It is a kind of amateurish community when it comes to technical stuff (notable exceptions abound, of course)."

Those that disagreed argued that music projects have similar numbers of bugs to report as general opensource projects.

5) Computer musicians' repositories have less unique contributing authors.

Responses: 8 computer musicians agreed with this hypothesis, 1 disagreed, and 6 were uncertain.

Rationale: Advocates of the hypothesis argued that:

- "Music projects are generally going to be more creative, and they might only reflect the vision of one developer."
- "The people who slave away at this wonderful software often work alone. Such is often the case, I am afraid, for audio people in general."

The computer musician who disagreed argued that there exists large-scale music projects with several contributors.

C. Methodology

The previously presented hypotheses are quantitatively confirmed or refuted using the Wilcoxon rank sum test. In this research study, the first population sample is the 819 computer music repositories queried from GHTorrent and the second population sample is the 819 repositories randomly sampled from GHTorrent. For each statistical test performed in the following section, both the z-value, p-value, and Cliff's Delta effect size are reported.

D. Results

Hypothesis 1: Number of repository commits

H_0: Computer musicians and general software developers make the same number of commits.

H_a: Computer musicians make less commits than general software developers.

Figure 2 displays a side-by-side box plot of commit counts for the compiled computer music dataset and the random sample dataset. To properly display the results, the commit counts were transformed into the logarithmic domain.[1] The median number of commits per computer music repository is 17, while the median for general software developers is 36 commits per repository. The Wilcoxon rank sum test reports a z-value of -7.332 and a p-value of $1.133\text{e}{-13}$. The value of Cliff's Delta effect size is small (-0.209 ± 0.052 with 95% confidence). At $\alpha = 0.01$ there is strong evidence to reject the null hypothesis and conclude that *computer musicians make less commits than the general population of software developers*. As several computer musicians noted during interviews, less commits may be made because of the community's culture of sharing intellectual property or the inability of computer musicians to identify significant structural changes in code.

Hypothesis 2: Number of weekend commits

H_0: Computer musicians and general software developers make equal numbers of weekend commits.

H_a: Computer musicians make more weekend commits than general software developers.

Before performing the statistical test, the data first needs to be preprocessed. Each repository has zero or more commits and each commit has an associated timestamp, which was processed to control for global time zones. If the commit occurred on a weekday it is assigned a value of zero, otherwise it is assigned a value of one. The average of these values are calculated for each repository and the result is the proportion of commits that occur on weekends. Figure 3 displays a side-by-side box plot of the proportion of commits occurring on weekends for the compiled computer music dataset and the random sample dataset. The Wilcoxon rank sum test reports a z-value of 3.805 and a p-value of $7.091\text{e}{-5}$. The value of

[1]For the display of box plots: to avoid taking the logarithm of elements with the value of zero, 0.5 was added to each measurement.

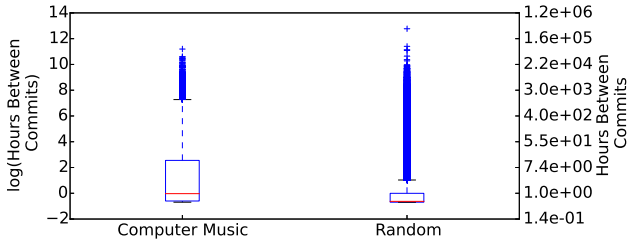


Fig. 4: Box plots of the log number of hours between commits for repositories in the computer music dataset (left) and the general software developers dataset (right).

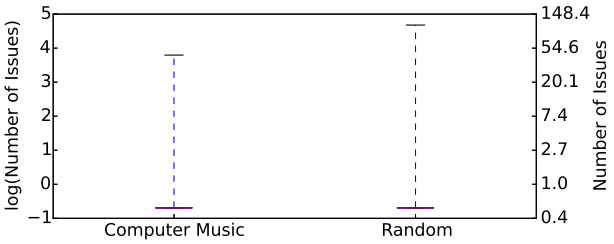


Fig. 5: Box plots of the log issue counts (left) and the number of unique authors (right) contributing to the repositories in the computer music dataset and the general developer dataset.

Cliff's Delta effect size is small (0.109 ± 0.056 with 95% confidence). At $\alpha = 0.01$ there is evidence to reject the null hypothesis and conclude that *computer musicians make more weekend commits than general software developers*. However, it is incorrect to reach the conjecture that computer musicians typically operate on the weekend; the box plot in Figure 3 shows that the median proportion of weekend commits is 23.6% for the sample of computer musicians' repositories.

Hypothesis 3: Commit frequency

H_0: Computer musicians and general software developers commit with the same frequency.

H_a: Computer musicians commit less frequently than general software developers.

For each software repository, the difference in hours between subsequent commits is calculated and concatenated into an array of 49,762 commit delays for the computer music dataset and 1,207,413 commit delays for the random sample dataset. Figure 4 displays a side-by-side box plot of logarithmic commit delays for the sample of computer music repositories (mean of 131.688 hours, median of 0.471 hours, between commits) and the random sample of repositories (mean of 8.220 hours, median of 0.034 hours between commits). The Wilcoxon rank sum test reports a z-value of 167.323 and a p-value of ≈ 0. The value of Cliff's Delta effect size is moderate (0.442 ± 0.004 with 95% confidence). At $\alpha = 0.01$ there is extremely strong evidence to reject the null hypothesis and conclude that *computer musicians commit less frequently than general software developers*. This result makes sense given that computer musicians make significantly more commits on weekends in relation to the general population of software developers, yielding longer delays between subsequent commits.

Hypothesis 4: Number of issues (bug reports)

H_0: Computer musicians and general software developers create the same number of issues.

H_a: Computer musicians create less issues than general software developers.

Figure 5 displays a side-by-side box plot of issue counts for the computer music dataset and the random sample dataset. The median number of issues for both datasets is zero, meaning that the general population of developers also create few issues when contributing to software repositories on GitHub. The Wilcoxon rank sum test reports a z-value of -0.792 and a p-value of 0.214. The value of Cliff's Delta effect size is negligible (-0.023 ± 0.024 with 95% confidence). At $\alpha = 0.01$ there is insignificant evidence to reject the null hypothesis and we conclude that *computer musicians create the same number of issues as the general population of software developers*, which refutes our hypothesis and the intuitions of many computer musicians.

Hypothesis 5: Number of unique authors

H_0: The number of unique authors contributing to computer musicians' and general software developers' repositories are equal.

H_a: Computer musicians' repositories have less unique authors than general software developers' repositories.

The number of distinct commit authors is calculated for each software repository. Figure 6 displays a side-by-side box plot of unique author counts for the computer music and random sample dataset of Git repositories. The median number of distinct contributing authors for both population samples is one, meaning that both computer musicians and the general population of developers contributing to software repositories on GitHub tend to work alone. The Wilcoxon rank sum test reports a z-value of -0.082 and a p-value of 0.4673. z-value of -0.792 and a p-value of 0.214. The value of Cliff's Delta effect size is negligible (-0.002 ± 0.003 with 95% confidence). At $\alpha = 0.01$ there is insignificant evidence to reject the null hypothesis and we conclude that both *computer musicians' and general software developers' repositories have similar numbers of distinct authors*, which refutes our hypothesis as well as several computer musicians' intuitions.

Summarizing the statistically significant results, computer musicians make less commits to software repositories, more commits on weekends, and less frequent commits in comparison to the general population of developers on GitHub. Our interviews with computer musicians revealed several possible causes for these differences: the culture of sharing intellectual property may differ; the tendency of computer musicians to work alone may influence how interactions with software engineering tools occur; and the varied programming skill of computer musicians indicates that education of software

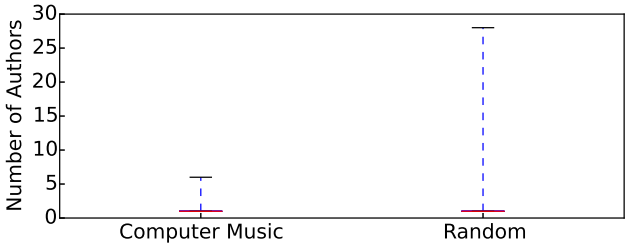


Fig. 6: Box plots of the number of unique authors contributing to repositories in the computer music dataset (left) and to repositories in the general software developers dataset (right).

engineering methodology and tools may be beneficial to this end-user community.

IV. Clone Detection

After mining computer music repositories to analyze the software development practices of this end-user community, we focus on analyzing the software artifacts within these repositories. Clone detection is an effective analysis tool for investigating the extent of code reuse in these visual programming languages. A revisitation of the previous Max/MSP clone detection study by Gold *et al.* [39] is performed in order to scale the clone detection's $O(n^2)$ pairwise matching algorithm to a $O(n)$ hash-based matching algorithm capable of processing a large quantity of patch files. Furthermore, instead of analyzing tutorial patches covering a wide variety of topics and concepts in the language, the following clone detection algorithm is used to locate duplicates or near duplicates of object structures in the abundance of both Max/MSP and Pure Data patches developed by computer musicians on GitHub.

A. Clone Detection Algorithm

The developed Max/MSP and Pure Data clone detection algorithm operates with two levels of granularity, locating DF1 and DF2 type clones in patch subgraphs. Recall that a patch is a directed graph consisting of *objects* (vertices) and *patchcords* (edges). The clone taxonomy is as follows:

- *DF2 clone:* two subgraphs with the same object types connected by patchcords to the same inlets and outlets.
- *DF1 clone:* two subgraphs that are a DF2 clone and, further, the corresponding objects in each subgraph possess the same literal values as default parameters.

Note that this clone taxonomy differs from Gold *et al.* [39] in that the absolute and relative positions of objects are not considered. In this research study we are solely interested in which objects computer musicians interface with other objects and are not concerned with situations where object position affects operational precedence.

The clone detection algorithm, presented in Algorithm 1, begins by parsing and translating Max/MSP and Pure Data files into a graph data structure consisting of nodes and edges. Using the resulting patch graphs as input, the proposed clone detection algorithm begins by setting the granularity of detected clones to either DF1 or DF2 clones. For each vertex in each directed graph representing a Max/MSP or Pure Data patch, the graph is traversed in a depth-first fashion. With each traversal, the attributes of objects (type, parameters, number of inlets, number of outlets) and patchcords (source object, outlet number, sink object, inlet number) along the path from the root vertex to the current vertex is compiled. Considering the size of the dataset of Max/MSP and Pure Data patches, the depth of paths considered by the clone detection algorithm is limited to eight. Depending on the granularity of clone detection, the gathered object and patchcord attributes are filtered accordingly. For example, when searching for DF1 clones all of the patchcord attributes are necessary but the *parameters* attribute of all objects should be discarded. The list of objects, patchcords, and their attributes are stored in a JSON data structure that is converted to text prior to hashing. Similar to the hash-based matching algorithm proposed by Hummel *et al.* [32], [33], this textual representation of the patch subgraph is transformed using the MD5 hash algorithm. If the hash is not unique, the subgraph is a clone.

Algorithm 1 Max/MSP and Pure Data clone detection

```
1:  cloneLevel ← 1 or 2
2:  hashes ← ∅
3:  clones ← [ ]
4:  i ← 0
5:  for all patch graphs do
6:    for all objects in patch do
7:      paths ← depth-first traversal from object
8:      for all paths such that depth(path) ≤ 8 do
9:        // gather attributes of path for clone granularity
10:       attributes ← filter(path, cloneLevel)
11:       hash ← MD5(attributes)
12:       if hash ∩ hashes = ∅ then
13:         hashes ← hashes ∪ hash
14:       else
15:         clones[i] ← path
16:         i ← i + 1
17:       end if
18:     end for
19:   end for
20: end for
21: return clones
```

To validate our implementation, the proposed clone detection algorithm was run on the dataset of 68 preprocessed Max/MSP tutorial patches used by Gold *et al.* [39] and received results similar to their algorithm: 31.8% of paths are DF1 clones and 88.2% of paths are DF2 clones, while Gold *et al.* report that 22.7% of paths are DF1 clones and 86.2% of paths are DF2 clones. Note that the slight increase in clone counts reported by our algorithm is likely due to the relaxed criteria for clone detection that disregards object position information.

TABLE II: Clone counts in Max/MSP and Pure Data patches.

TYPE	CLONE COUNTS	PATHS	CLONE PROPORTION
DF1	9,798,031	10,985,064	89.2%
DF2	10,462,725	10,985,064	95.2%

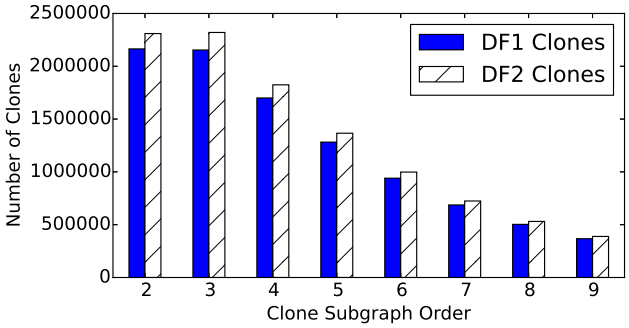


Fig. 7: Clone counts for different Max/MSP and Pure Data subgraph orders.

B. Experiment Methodology

The compiled dataset of 819 computer music repositories used in Section III is also used as the dataset for the clone detection experiment outlined in this section. This dataset of software repositories contain approximately 118,000 Max/MSP and Pure Data patches, 3 million objects, and 2.5 million patchcords (Table I). The proposed clone detection algorithm is run on the patches in the dataset once to locate DF1 clones and once more to locate DF2 clones.

C. Results

The number of DF1 and DF2 clones detected by the proposed algorithm is presented in Table II. Approximately 9.8 million DF1 clones and 10.5 million DF2 clones were detected out of the roughly 11 million paths traversed by the algorithm. From this analysis we note that 89.2% of connected object subgraphs in Max/MSP and Pure Data patches programmed by computer musicians are DF1 clones and 95.2% of connected object subgraphs are DF2 clones. These clone proportions are significantly higher than those reported by our algorithm on the validation dataset of preprocessed Max tutorial patches. This result is expected given the intentional variety of concepts and object connections explored in the tutorial patches, whereas patches created by computer musicians do not necessarily utilize all the features of these visual programming languages.

The distribution of clone counts over the order of path subgraphs is presented in Figure 7. The order of a graph $G = \{V, E\}$ is $|V|$, the cardinality of the set of vertices in the graph. For example, Figure 7 reports that approximately $50,000$ of the DF1 clones detected in Max/MSP and Pure Data patches created by computer musicians are composed of eight connected objects. The resulting distribution of clone counts reveals that as the order of subgraphs increase, the number of clones decrease. Moreover, as the criteria for clone detection becomes more relaxed—for example, as we move from DF1 to DF2 clone detection—the number of clones increase.

Among the millions of DF1 and DF2 clones detected within the Max/MSP and Pure Data patches gathered from GitHub, several commonly occurring clone structures (Figure 8) stand out that emphasize common practices of computer musicians and highlight idiosyncrasies of these visual programming languages. The clone depicted in Figure 8 (a) is a Pure Data envelope follower object, which outputs the amplitude in decibels of an input audio signal. However, an output of 1 is normalized to 100 decibels, and so many computer musicians subtract 100 to reverse the normalization. Figure 8 (b) displays a frequently occurring clone involving the `loadbang` object, which fires a *bang* when the patch loads. The bang message acts as a trigger for connected objects to start processing. This clone triggers two bangs instead of one when the patch starts, which suggests that the `loadbang` object should have a parameter indicating the number of bang messages to output. The following Pure Data clone shown in Figure 8 (c) outputs the number 1 when the patch loads. In Max/MSP, an object called `loadmess` exists to accomplish this task; however it is not implemented in Pure Data. The clone depicted in Figure 8 (d) is the identity function that simply outputs its input. A possible explanation for the frequency of this clone is that computer musicians create an identity function with the intent to add functionality later but forgot. The clone shown in Figure 8 (e) provides commentary on the point in which some computer musicians choose to abstract code fragments into functions. In this case, the clone is an overly simplistic function that attenuates the amplitude of the input signal. The clones in Figure 8 (f) demonstrate that computer musicians often choose default parameters that have no effect on the output—for example, multiplying a value by one—or choose default parameters that *zero* a value or *silence* an audio signal until an event occurs that changes the default parameters. The clones in Figure 8 (g) display two methods that computer musicians use to perform calculations: either as a one-line expression using the `expr` object, or as a daisy chain of mathematical operations. The clone displayed in Figure 8 (h) is a high-pass filter—responsible for passing high frequencies in audio signals—that is immediately followed by a low-pass filter, which passes low frequencies in audio signals. This configuration of objects is essentially a band-pass filter, which already exists as a stand-alone object in both Max/MSP and Pure Data. The clone shown in Figure 8 (i) demonstrates that computer musicians often use external objects, such as this crossfade object in a popular Pure Data library, to simplify common musical functions like fading out one audio signal while fading in another. Finally, the clones in Figure 8 (j) demonstrate that computer musicians often use magic numbers such as 127—the highest value encoded in the musical instrument digital interface (MIDI) protocol—or even divisions of the mathematical constant π.

These clones are a mere subset of examples that offer insight into the refactorings one could apply to Max/MSP or Pure Data patches and the development tools that could

be adopted to ease end-user implementation. For instance, with advice from a critic program [40] similar to Microsoft's *Clippy* [41], [42], computer musicians could be prompted to leverage already existing functionality within the language. Moreover, the detected clones could be used to create a corpus-based code completion and search tool that suggests similar object connections made by other computer musicians. Furthermore, the high number of clones within patches (Table II) is indicative of copy and paste programming techniques, emphasizing the finding that source code is often repetitive in structure [43], and reinforcing the importance of tools that suggest similar code to copy and modify. These tools could be generalized to other visual programming languages, such as Matlab Simulink, to aid other end-user communities.

V. Computer Musician Survey and Interviews

A survey of 175 computer musicians and interviews with 15 computer musicians was conducted to gather more information about this end-user community. Computer musicians were recruited using relevant forums on www.reddit.com, the Max/MSP and Pure Data forum boards, and several mailing lists. Authors of the 819 computer music GitHub repositories outlined in Section III were not used as the survey population due to sampling bias, since all respondents would report using version control. The surveyed computer musicians were invited to respond to a series of questions regarding their experience, motivations, programming methodology and use of software engineering tools, as well as the support they seek in the development process.[2] As supplementary commentary, those surveyed optionally participated in an interview over e-mail or in person to provide a more detailed analysis of themes investigated in the survey. No compensation was provided for participating in the survey or interview.

A. Experience

The first series of questions prompted computer musicians to reflect on how long they have been programming in the context of computer music and introspectively assess their skill level. Figure 9a presents the distribution of experience in years of computer musicians. Of these computer musicians, 15 considered themselves beginner programmers, 84 considered themselves intermediate programmers, 74 considered themselves advanced programmers, and 2 did not respond. According to these responses, the computer music community varies in experience, but predominantly consists of self-reported intermediate and advanced programmers.

B. Motivations

The next portion of the survey prompted computer musicians to reflect on their motivations for programming musical applications. Of the computer musicians who responded to the survey, 65% program musical applications as a hobby, while the remaining portion of computer musicians write musical code as a main source of income. If the majority of computer musicians program in their free time, one would expect their commits to occur more on weekends and be less frequent than professional software developers. Indeed, the results procured by the significance tests performed in Section III support this claim. In addition, only 40% of computer musicians who responded to the survey write music software for other individuals or companies. This response was resonated in the interviews with computer musicians, where the majority of interviewees advocated that music projects tend to be highly personal and follow the creative vision of one musician.

C. Programming Methodology and Source Control

The following portion of the survey prompted computer musicians to reflect on how they program. First, computer musicians were asked which programming languages they typically use for music application development. Figure 9b presents the top five languages used by computer musicians who responded to the survey. According to the responses, Max/MSP and Pure Data are the top two programming languages used by computer musicians.

Second, computer musicians were asked if they write tests for their musical code, to which 30% responded yes. Tests

[2]http://cs.ualberta.ca/ gburlet/musiccoders_survey.html

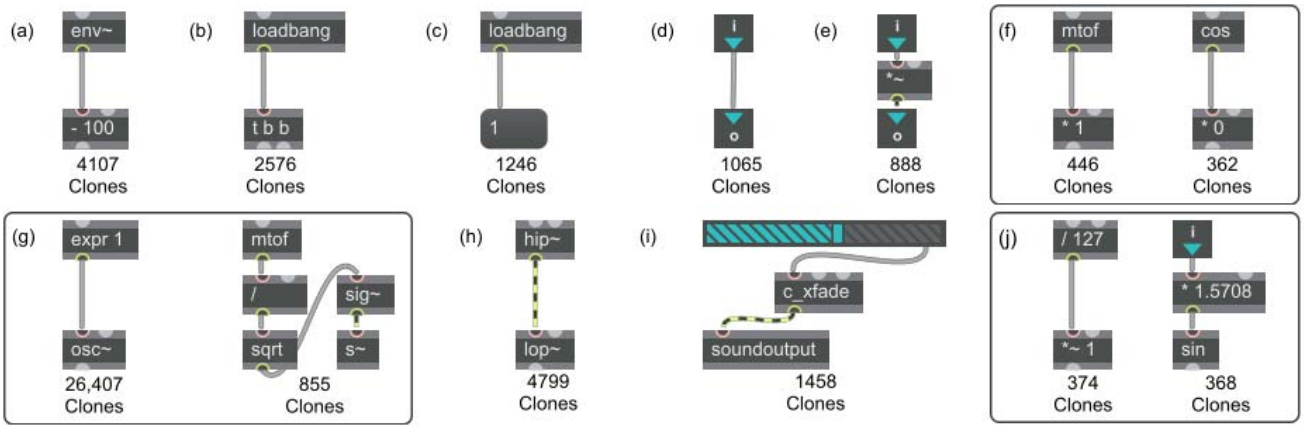
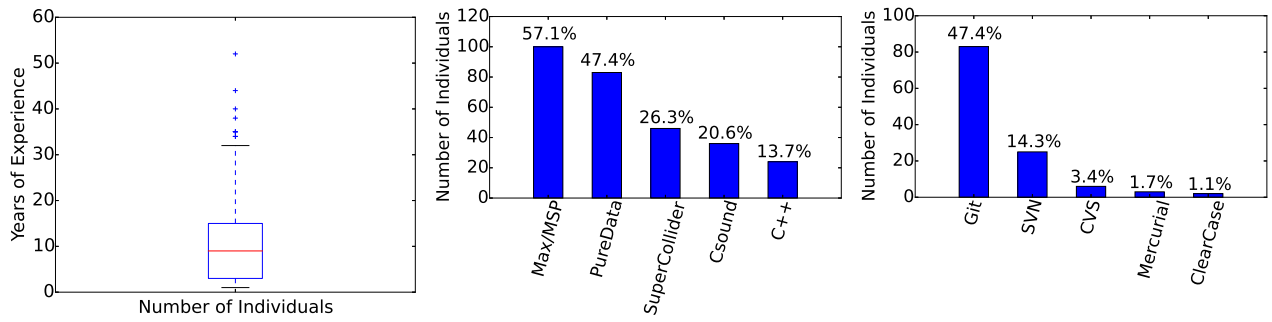


Fig. 8: Thirteen noteworthy clones detected in Max/MSP and Pure Data patches programmed by computer musicians.



(a) Distribution of years of experience of computer musicians.

(b) Top 5 out of 69 programming languages used by computer musicians.

(c) Top 5 out of 15 source control repositories used by computer musicians.

Fig. 9: Computer musicians' responses to selected survey questions.

may be written for user or device interfaces, infrastructure and logic code, or attributes of the sounds produced. A possible explanation for the large proportion of computer musicians who do not write tests is that it may be difficult to express desired qualities of a sound as a quantifiable property that can be asserted in the context of a test.

Next, computer musicians were asked if they comment their music-related code. Of the surveyed computer musicians, 34 barely comment their code, 94 provide comments on key functions, 44 extensively comment their code, and 3 did not respond. Within the dataset of 819 repositories containing Max/MSP and Pure Data patches, there exists $505,236$ comment objects out of the $3,087,278$ objects in the dataset (see Table I), which provides empirical evidence that computer musicians do comment their musical code.

Computer musicians were then asked if they use source control repositories, to which 54% responded yes. The top 5 source control systems used by computer musicians are outlined in Figure 9c, which reveals that nearly all computer musicians who use source control use Git. With roughly half of computer musicians not using source control repositories, we turn to interview responses for possible explanations:

- "I'm still trying to figure out how best to work with it."
- "It hasn't really seemed necessary; lack of backup for previous versions hasn't really caused me any significant problems."
- "No one else really uses my code, so versioning isn't a priority for me."

These responses indicate that source control systems may be avoided by the computer music community due to a lack of technical understanding of the tool, a lack of understanding of the merits of the tool, or because collaboration is unnecessary in their computer music endeavours.

D. Development Support

The final portion of the survey prompted computer musicians to report the knowledge sources they consult for development support. Of the surveyed computer musicians, only 26% use www.stackoverflow.com, a popular question and answer website for software development. If the majority of computer musicians don't use www.stackoverflow.com, which knowledge sources do they consult for aid? Interviews with computer musicians revealed that search engines are first used before consulting their community through mailing lists. Indeed, 54% of the surveyed computer musicians subscribe to mailing lists. These results indicate that the community of computer musicians could benefit from more knowledge sources for support, such as a question and answer website dedicated to computer musicians.

In summary, the survey and interviews revealed that the community of computer musicians consists of predominantly intermediate and advanced programmers who typically program musical applications in Max/MSP and Pure Data as a hobby for their own musical works. These end-user programmers comment their code, infrequently write tests, and roughly half use source control systems. Furthermore, the computer music community is more reliant on mailing lists than sites such as www.stackoverflow.com for support.

VI. Threats to Validity

A possible threat to validity of the statistical tests in Section III is the sampling methodology. Only public data observable on GitHub was collected with the assumption that computer music repositories on GitHub are assumed to be a representative sample of the development practices and software artifacts of computer musicians. Although computer musicians who contribute to source control repositories may exhibit more proficiency in software engineering tools than other community members, the conclusion that they are also more technically proficient is unwarranted. Moreover, the assumption is made that developers who use GitHub are representative of the greater population of developers and that repositories unrelated to music are assumed to be a representative sample of the practices of general software developers. Furthermore, Kalliamvakou *et al.* [21] estimate that a third of repositories hosted on GitHub are not necessarily software projects but rather experimental sandboxes, academic projects, or file stores. Our random sample of GitHub repositories is not filtered to remove such repositories; analogously, the collected

computer music repositories are not filtered to remove patches that appear experimental or academic in nature.

A possible threat to validity of the clone detection experiments conducted in Section IV arises from mixing Max/MSP and Pure Data patches: a handful of objects have the same functionality but have different names in Max/MSP and Pure Data. For example, in Max/MSP the oscillator object is called `cycle~`, while in Pure Data the same object is named `osc~`. Although clones involving these different objects should be detected because they are semantically equivalent, they will remain undetected due to the different object names; thus, clone counts are likely underestimated.

The survey and interview recruitment strategy also invites a possible threat to validity since the sampling of certain computer music sub-communities may bias results.

VII. Future Directions

This research provides a glimpse into the operations and development practices of this end-user community and provides a foundation to develop and tailor existing software engineering tools for visual programming environments. Currently there are no software development tools explicitly tailored towards visual programming in languages such as Max/MSP or Pure Data, apart from external code libraries. Computer music end-user programmers could benefit from tools such as code completion, where object insertions in patches are intelligently suggested based on the context of the current patch or other music patches; code critics, which assess the semantics of recently connected objects and suggest alternative refactorings or abstractions; code search, which could provide references to similar object structures within their repository or other computer musicians' repositories; the adaptation of text-based code highlighting tools [44] to highlight important object structures to aid patch navigation; visual code clustering, which could group objects based on their abstract musical function and modify the layout of the patch accordingly; and music software testing frameworks to facilitate any quantitative tests necessary within musical patches. Further mining of computer music repositories containing textual source code could reveal other software development tools that are unavailable to visual music programmers. The aforementioned tools could be generalized to benefit other end-user communities using visual programming languages.

This research study also revealed interesting facts about the collaboration tendencies of computer music end-user programmers. The conducted survey and interviews showed that only half of computer musicians use source control systems for various reasons. In response to these findings, we encourage future research on adapting current revision control systems and source code difference tools to better handle visual patch-based source code by focusing on what the patch encoding represents rather than the patch encoding itself. It is our hope that adapting these software engineering tools to satisfy the needs of computer musicians will facilitate collaboration within this end-user community.

Finally, this research study indicates that the education of computer musicians about software engineering tools and methodologies is an important next step. The interviews indicate that a subset of this end-user community are either unfamiliar with currently available software engineering tools, lack the technical proficiency to use them, or do not acknowledge the merits of introducing such tools into their software development practices. Thus the software engineering research community can contribute much to this end-user community.

VIII. Conclusion

A multifaceted study of computer music programmers has been conducted to gain insight into how this community of end users develop music patches written in the Max/MSP and Pure Data visual programming languages.

The first facet of analysis was a comparison of the software development practices of computer musicians and the general population of software developers. Using a dataset of Git repositories hosted on GitHub, a series of statistical tests established that in comparison to the general population of software developers, computer musicians' repositories have less commits, less frequent commits, more commits on the weekend, yet similar numbers of bug reports and contributing authors. These differences are attributed to the cultural differences in sharing intellectual property or the lack of education surrounding the merits or use of version control systems.

The second facet of analysis was an investigation of cloned code and repeated object structures in visual music programming languages. When run on $118,481$ Max/MSP and Pure Data patches, the algorithm detected that 89% of connected objects are DF1 clones (object types, parameters, and connections are equivalent) and 95% of connected objects are DF2 clones (object types and connections are equivalent). Several clones discovered in source code programmed by computer musicians were re-implementations of already existing objects in Max/MSP and Pure Data.

The final facet of analysis investigates, via surveys and interviews, how computer musicians build their software and which software engineering tools they use. The surveys reinforced that computer musicians do not necessarily use source control repositories or bug trackers. Furthermore, computer musicians lack a dedicated support website for posing questions and answers to the entire computer music community and instead subscribe to mailing lists for support.

Now that an empirical study of computer music programmers has been conducted, more work can be done to educate computer musicians and develop software engineering tools for this end-user community and others using visual programming languages. For more details about the results and analyses presented in this paper, please see the technical report [45].

Acknowledgements

Special thanks are owed to the computer musicians who participated in the survey and interviews.

REFERENCES

[1] J. Chadabe, "Remarks on computer music culture," *Computer Music Journal*, vol. 24, no. 4, pp. 9–11, 2000.

[2] K. T. Stolee, S. Elbaum, and A. Sarma, "End-user programmers and their communities: An artifact-based analysis," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 147–156.

[3] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, J. Lawrance, C. Scaffidi, H. Lieberman, B. A. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Widenbeck, "The state of the art in end-user software engineering," *ACM Computing Surveys*, vol. 43, no. 3, pp. 1–60, 2011.

[4] M. Burnett, *Software engineering for visual programming languages*. World Scientific Publishing Company, 2001, vol. 2, pp. 1–12.

[5] M. Puckette, "Max at seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[6] Github. [Online]. Available: https://github.com

[7] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the numbers of end users and end user programmers," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 207–214.

[8] G. Gischer and E. Giaccardi, "Meta-design: A framework for the future of end-user development," in *End User Development: Empowering People to Flexibly Employ Advanced Information and Communication Technology*, H. Lieberman, F. Paternò, and V. Wulf, Eds. Kluwer Academic Publishers, 2004, pp. 427–457.

[9] G. Fischer, "End-user development and meta-design: Foundations for cultures of participation," *Organizational and End User Computing*, vol. 22, no. 1, pp. 52–82, 2010.

[10] B. A. Myers, A. J. Ko, and M. Burnett, "Invited research overview: End-user programming," in *Extended Abstracts of the Conference on Human Factors in Computing Systems*, 2006, pp. 75–80.

[11] C. A. McLean, "Artist-programmers and programming languages for the arts," Ph.D. dissertation, Goldsmiths University of London, 2011.

[12] T. Coughlan and P. Johnson, "Constrain yourselves: Exploring end user development in support for musical creativity," in *ACM Conference on Creativity and Cognition*, 2007, pp. 247–248.

[13] ——, "An exploration of constraints for end user development in environments for creative tasks," *Human-computer Interaction*, vol. 24, no. 5, pp. 444–459, 2008.

[14] G. Fischer, E. Giaccardi, Y. Ye, A. G. Sutcliffe, and N. Mehandjiev, "Meta-design: a manifesto for end-user development," *Communications of the ACM*, vol. 47, no. 9, pp. 33–37, 2004.

[15] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A survey of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2, pp. 83–137, 2003.

[16] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, 2004, pp. 199–206.

[17] C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi, "End-user programming in the wild: A field study of coscripter scripts," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2008, pp. 39–46.

[18] D. G. Hendry and T. R. G. Green, "Creating, comprehending, and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model," *Human Computer Studies*, vol. 40, no. 6, pp. 1033–1065, 1994.

[19] R. Abraham and M. Erwig, "Goaldebug: A spreadsheet debugger for end users," in *Proceedings of the International Conference on Software Engineering*, 2007, pp. 251–260.

[20] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace, "End-user software engineering with assertions in the spreadsheet paradigm," in *Proceedings of the International Conference on Software Engineering*, 2003, pp. 93–103.

[21] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the Working Conference on Mining Software Repositories*, 2014, pp. 92–101.

[22] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining Git," in *Proceedings of the International Working Conference on Mining Software Repositories*, 2009, pp. 1–10.

[23] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *Proceedings of the Working Conference on Mining Software Repositories*, 2012, pp. 12–21.

[24] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the Working Conference on Mining Software Repositories*, 2013, pp. 233–236.

[25] 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.

[26] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," in *Proceedings of the International Conference on Program Comprehension*, 2008, pp. 153–162.

[27] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 4, pp. 470–495, 2009.

[28] R. Falke, P. Frenzel, and R. Koschke, "Clone detection using abstract syntax suffix trees," in *Proceedings of the Working Conference on Reverse Engineering*, 2006, pp. 253–62.

[29] E. Juergens, F. Deissenboeck, and B. Hummel, "CloneDetective: A workbench for clone detection research," in *Proceedings of the International Conference on Software Engineering*, 2009, pp. 603–606.

[30] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1–2, pp. 77–108, 2009.

[31] J. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1993, pp. 171–183.

[32] B. Hummel and E. Juergens, "Index-based code clone detection: Incremental, distributed, scalable," in *Proceedings of the International Conference on Software Maitenance*, 2010, pp. 1–9.

[33] B. Hummel, E. Juergens, and D. Steidl, "Index-based model clone detection," in *Proceedings of the International Workshop on Software Clones*, 2011, pp. 21–27.

[34] F. Deissenboeck, B. Hummel, E. Juergens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert, " Clone detection in automotive model-based development," in *Proceedings of the International Conference on Software Engineering*, 2008, pp. 603–612.

[35] F. Deissenboeck, B. Hummel, and B. Schätz, "Model clone detection in practice," in *Proceedings of the International Workshop on Software Clones*, 2010, pp. 57–64.

[36] N. Pham, H. Nguyen, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Complete and accurate clone detection in graph-based models," in *Proceedings of the International Conference on Software Engineering*, 2009, pp. 276–286.

[37] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for Simulink models," in *Proceedings of the International Conference on Software Maitenance*, 2012, pp. 295–304.

[38] N. Gold, J. Krinke, and M. Harman, "Issues in clone classification for dataflow languages," in *Proceedings of the International Workshop on Software Clones*, 2010, pp. 83–84.

[39] N. Gold, J. Krinke, M. Harman, and D. Binkley, "Cloning in Max/MSP patches," in *Proceedings of the International Computer Music Conference*, 2011, pp. 1–4.

[40] M. G. Armentano and A. A. Amandi, "Personalized detection of user intentions," *Knowledge-based Systems*, vol. 24, no. 8, pp. 1169–1180, 2011.

[41] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse, "The lumiŕe project: Bayesian user modeling for inferring the goals and needs of software users," in *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 1998, pp. 256–265.

[42] S. Hedberg, "Executive Insight: Is AI going mainstream at last? A look inside Microsoft Research," *IEEE Intelligent Systems*, vol. 13, no. 2, pp. 21–25, 1998.

[43] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *Proceedings of the International Conference on Software Engineering*, 2012, pp. 837–847.

[44] B. Athreya and C. Scaffidi, "Towards aiding within-patch information foraging by end-user programmers," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2014, pp. 13–20.

[45] G. Burlet and A. Hindle, "Patches and patchcords: An analysis of how computer music end-user programmers develop musical code," University of Alberta, Tech. Rep., 2014, http://webdocs.cs.ualberta.ca/~hindle1/2014/musiccoders.