Software Evolution Sonification

Pedro O. Raimundo
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
pedrooraimundo@ifba.edu.br

Sandro S. Andrade
Instituto Federal de Educação, Ciência e Tecnologia da Bahia
sandroandrade@ifba.edu.br

ABSTRACT

Program comprehension is one of the most challenging tasks undertaken by software developers. Achieving a firm grasp on the software's structure, behavior and evolution directly from its development artifacts is usually a time-consuming and challenging task. Software visualization tools have effectively been used to assist developers on these tasks, motivated by the use of images as outstanding medium for knowledge dissemination. Under such perspective, software sonification tools emerge as a novel approach to convey temporal and concurrent streams of information and have been proven to perform remarkably well due to the their inherently temporal nature. In this work, we describe how software evolution information can be effectively conveyed by audio streams, how music, software architecture concepts and techniques come together to achieve such means, elaborate on the possibilities for future endeavors in this research area and finally propose a framework for sonification of extracted evolutionary metrics from software repositories.

Categories and Subject Descriptors

H.5.2 [**INFORMATION INTERFACES AND PRESENTATION**]: User Interfaces—*Auditory (non-speech) feedback, Theory and methods*; H.5.5 [**INFORMATION INTERFACES AND PRESENTATION**]: Sound and Music Computing—*Methodologies and techniques, Signal analysis, synthesis, and processing*; D.2.8 [**SOFTWARE ENGINEERING**]: Metrics—*Complexity measures,Product metrics,Process metrics*; D.2.m [**SOFTWARE ENGINEERING**]: Miscellaneous—*Software Evolution*

General Terms

Software Sonification, Auditory Display, Software Evolution, Program Comprehension, Sound and Music Computing

Keywords

program comprehension, auditory display, software evolution, sound and music computing

1. INTRODUCTION

Comprehending computer programs is a notoriously difficult task that involves gathering information from diverse sources (source code, documentation, runtime behavior, version history, just to mention a few) and gets progressively harder as the program's size and complexity grow [24]. Synthesizing that information to tackle the development process effectively and efficiently is an endeavor that requires time, experience and, more often than not, peer support.

Regardless of the abstraction level (code, design, architecture) at which the developer is going to address the problem at hand, tools are usually employed in order to help the decision making and understanding. Such tools range from built-in Integrated Development Environment (IDE) helpers and code metric viewers to complex software visualization solutions, focusing on conveying information to the user through the computer screen using tables, charts, drawings or animations due to their higher apprehension, if compared with a naive representation of the same data. While such approaches are helpful in the comprehension process, aural representations of software structure and behavior have been shown to excel at representing structural [26], relational [4] and parallel or rapidly changing behavioral data [23] in an non-invasive and uncluttered fashion with high apprehension rate even for individuals without extensive musical background [4, 28, 27].

Software Evolution, specifically, adds another dimension to the problem, since it forces the subject to add another layer of understanding to the structural aspects of a software program; the temporal layer. Common analysis requires, for example, information about the relationship between components of a system, and lower-level information such as lines of code and fan-in/fan-out values per component. Evolutionary analysis can, for example, consume the results of the regular analysis procedures to yield higher-level information such as architectural drift in a given time slice, through the use of the project's meta-data (number of commits, bugs reported, time between releases, etc.). Evolutionary analysis can also provide environment-related information which is particularly useful to manage Large Program Evolution and Continuing Change [16] in computer systems.

In order to convey such information through the means of aural representations it is possible to use any kind of sound. Vickers and Alty proposed that these constructs are more effective if they follow culturally-appropriate styles, conceived

analogously to words, which carry meaning to an individual native to or familiar with a specific language, and also that western musical forms (based on the seven-note diatonic scale) are more readily recognized around the world [28].

This work presents a novel approach to transmit information about software evolution by exploiting sound's uniquely temporal nature and aural events such as melody, harmony, rhythm and noise. Different events are used because each event has a distinct impact on the listener and they can be mixed and matched together to convey different streams of information, as long as not being confusing or intensely overwhelming.

The key contributions of this work are as follows: first, we propose the foundations for a sonification framework in which the evolutionary aspects of a piece of software can be represented as sound streams, in an unobtrusive and noninvasive manner, building upon the existing research on the fields of auditory display and software comprehension through sonification. Second, we categorized and surveyed the academic production in the multiple disciplines that come together in this field. Third, we explore a research field that, to the best of our knowledge, has yet many research challenges, raising awareness to this particular field and possibly creating a new forum for idea exchange with a lot of untapped multidisciplinary potential.

The remainder of this publication is organized as follows: **Section 2** elaborates on the theoretical foundations for this work and gives some historical background, while also explaining the fundamental concepts upon which it builds to answer the research questions brought forth, **Section 3** details the proposed approach to implementing a software solution that helps answer these questions, **Section 4** discusses the implementation choices and development process undertaken to build the framework, **Section 5** contextualizes the usage of sonifications for program comprehension, and details the results of a survey on previous work in the field of software evolution sonification, **Section 6** presents the validation strategy utilized to evaluate the framework, along with some findings and **Section 7** summarizes the content and purpose of this paper, restates the research questions brought forth and hints at further possibilities in this line of investigation.

2. FOUNDATIONS

The present work lies within the intersection between the fields of Sound and Music Computing, Auditory Display and Software Evolution, while all these three areas have been object of numerous studies, the efforts to try and bring all three together have been comparatively modest.

2.1 Sound and Music Computing

Sound and Music Computing (SMC) is the research field that deals, generally speaking, with the relationship between music and computers, a bond that can have its roots traced back to the 1950's when composers and engineers worked together to compose music from the beeps and clicks from their machines. Nowadays, the field has consolidated and broadened itself to take on the challenges of understanding, modeling and producing sound and music through computational means with artistic, scientific and technological aspects being taken into consideration [5].

SMC research is inherently multi-disciplinary because the researchers in this field aimed from the very beginning to bridge the gap between disciplines as diverse as physics, psychology and engineering. Music was the common ground that allowed them to do so, in [5], Bernadini and de Poli define music in the SMC area as follows:

> Music, as the name itself indicates, provides the core investigation area for SMC. It supplies both an endless material (both in scope and depth) for analytical investigation (Musicology) and the requirements to extend expressive means of creation (Music Composition), while Music Performance yields interest on both sides (analytical and expressive).

That said, in this work we deal with SMC as a cohesive and consolidated unit, without going in detail about the building blocks that compose it, but rather looking into the main studies of its literature and applying the underlying music theory in our musical auralizations.

2.2 Auditory Display

Auditory Display is another well-established research field that tackles challenges involving sound and technology, not necessarily dealing with musical, artistic or psychological aspects, this field is more technology-oriented and focuses heavily on how to transmit information from a computer to the user through sound. While also having its roots in early experiments with computers and sound synthesis, the Auditory Display field matured much more quickly in the 80's, when Sara Bly's works [6, 7] demonstrated that it was feasible to display data using non-speech sound, laying the keystone in the field. In 1992, the International Conference on Auditory Display (ICAD) was established, holding an annual event for researchers in the field to share their ideas and findings.

At the end 90's and well into the 2000's, developments in the auditory display field allowed researchers to sonify various software aspects and start working towards leveraging auditory display's potential in order to aid program comprehension. One of the earliest implementations of software sonification was *InfoSound* [23], a tool that aimed to assist professional coders follow rapidly-occurring, concurrent or otherwise hard to visualize software behavior. In 1999, Vickers presented *CAITLIN* [26] as part of his doctorate thesis. His tool extracted information about program's control flow and structure to help novice pascal developers track bugs and achieve a firmer grasp of the code they just wrote.

It should be noted that these early tools required special MIDI (*Musical Instrument Digital Interface*) Synthesis hardware and familiarity with musical structures, since the user had to create his or her own motifs for the application, these were merely limitations of the technologies of storage, processing and synthesis available at that time. The experimentations ran in the studies that lead to the development of

these applications were also great contribution to the field, allowing researchers to have an initial understanding of what is effective and what isn't when it comes to software sonification.

As technological advances facilitated the access to sound synthesis hardware, and higher level sonification technologies such as *CSound* [25], *MP4-SA* [21] and *MAX-MSP* [3] emerged, the proposed software solutions became increasingly more sophisticated. Tools such as the *Sonified Omniscient Debugger* [24], *CodeDrummer* [15] and *CocoViz* [8] are all good examples of the technological advancements in the area, and each of these authors proposed additions to the tested and proven sonification frameworks, such as the inclusion of noise events to the auralizations, deeper utilization of rhythmic structures or the revival of software sonification as a means to provide accessibility to visually impaired users.

Out of the software sonification tools that were unearthed in our initial research efforts, the only one that attempted to bring temporal aspects to the table was the latest version of the *CocoViz* [8] tool. While this is a first effort to bring evolutionary aspects to a software sonification tool, we find that there's much more that can be done by tapping into that area's potential.

2.3 Software Evolution

Software Evolution is elaborated on by Lehman [16] with the following:

> (...)Evolution is an intrinsic, feedback driven, property of software. The meta-system within which a program evolves contains many more feedback relationships than those identified (...) Primitive instincts of survival and growth result in the evolution of stabilizing mechanisms in response to needs, events and changing objectives.

This quotation carries two defining qualities of the process of software evolution: 1) The existence of a meta-system, implying that a system is never alone in its production environment but rather interacting, stimulating and receiving feedback from other systems; 2) The evolution of stabilizing mechanisms to make sure the system maintains its correctitude after deployment even if the requirements change.

Aware of that, it becomes clear that evolutionary aspects go far beyond tracking amount of source change between revisions of source code file, the fact that evolution concerns itself with the nature of the software at a meta-systemic level shows that there are metrics at the architectural level that should be extracted, analyzed and displayed. This high level of abstraction, the large number of components involved in these interactions and the temporal nature of evolution itself make audio stand out as an efficient medium to convey this information to the user.

3. PROPOSED APPROACH

This paper proposes a sonification framework that sonifies software evolution by following three main steps: source code retrieval, data extraction and sound synthesis. The solution's general architecture and the sonification process are detailed in the following sections.

3.1 Proposed Architecture

Figure 1 depicts the structural (component-and-connector) architectural view of the proposed framework, presenting three specialized software modules and the interactions between them and their shared working set. This approach allows each module to evolve and adapt independently, should its requirements change, as long as the format of their shared data and their communication interfaces (both module to module and module to working set).

Another advantage of the modular approach is the possibility of reusing each of the modules for their specific purposes in an uncoupled manner. For example, as long as the user knows the data format used by sound renderer this module can be used independently to auralize any given data. Similarly, the metrics extractor can retrieve evolutionary metrics from a local, previously downloaded, source code repository for use in a completely independent fashion.

An important artifact of this architecture is the direct connection between the repository browser and the metrics extractor modules, this was done to allow repository metadata to be collected *while* the source code repository is transversed – since such data can't be retrieved from source code itself. Given the importance of contextual and environmental data in evolutionary software analysis this decision is adequate, despite making it harder to independently evolve the repository browser module, which now has to maintain the compatibility of one additional interface.

3.2 Source Code Retrieval

Version control systems (VCS) create repositories that store an initial version of the source code and then successive versions of the files by using delta compression, saving only discrete files that contain the differences between files rather than the entire data. This allows for efficient storage and seamless switching between different snapshots of the versioned project, each such snapshot is often called a *revision*.

When a developer send his proposed changes for integration in a branch, his version control client prepares a package of the proposed changes along with contextual information such as author name, date, time and the author's comments on the changes. This package is called a *commit*[1].

Software evolution is an inherently temporal phenomenon, as such it's fundamental to track the changes in a program's structure and code-metrics across a period of time to achieve a proper representation of it. This involves retrieving snapshots of the software's source code at different revisions, version control systems greatly simplify this task since repositories created by such tools can be systematically transversed, processed and compared.

In order to extract meaningful data from software repositories and generate interesting sonifications, the metrics extractor module uses filters. While the framework includes

[1]Not to be confused with the concept of commits for transactional database systems.

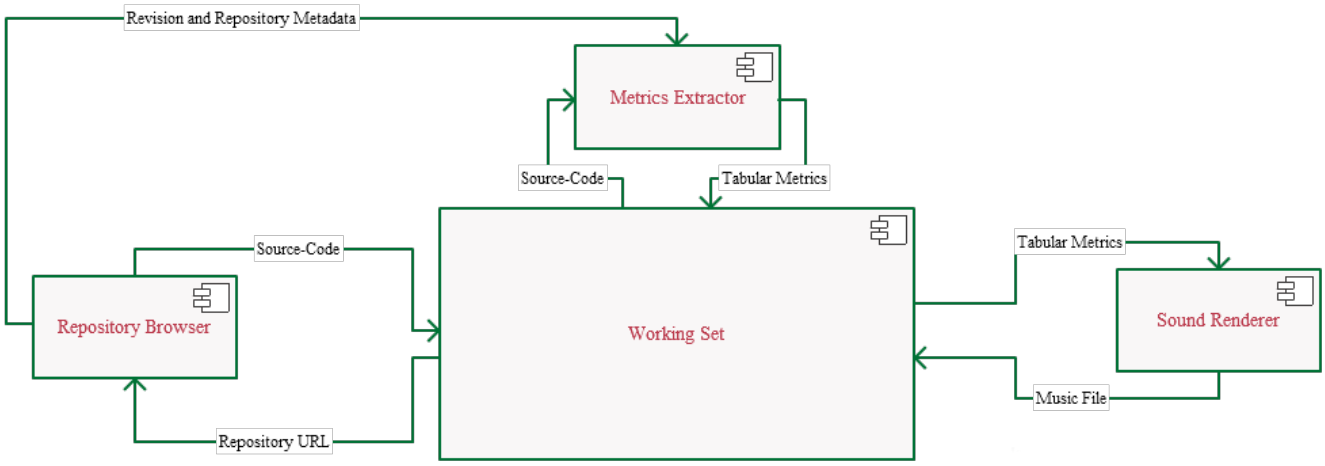

Figure 1: Component and connector view of the proposed framework.

some filters to demonstrate its capabilities, the ability to easily write and combine new filters is the highlight that allows the user to transverse source code repositories in order to achieve virtually any goal.

3.3 Data Extraction

The broad field of investigations that generally deal with extracting data from software repositories in order to uncover trends, relationships and extract pertinent information is called Mining Software Repositories (MSR), *software repositories* refer, in this context, to the entirety of development artifacts that are produced during the process of software development and evolution, including the data in VCS repositories, bug-tracking systems and communication archives such as email and memos.

The content of these aggregated data sources exists for the entirety of the software project life cycle and carry a wealth of information that includes, but is not limited to: The versions that the system has gone through, the changes, metadata about the revisions of the software (as seen in 3.2), the rationale for the projects architectural choices and discussions between the projects members.

Mining data and metadata from software repositories typically serves one out of two purposes, as outlined by Kagdi, Collard and Maletic [14]. The first purpose is described as follows:

> The first is the market-basket question (MBQ) formulated as: if A occurs then what else occurs on a regular basis? The answer is a set of rules or guidelines describing situations of trends or relationships. For example, if A occurs then B and C happen X amount of the time.

Later in their paper, the second purpose is clarified:

> The second type of MSR purpose relates to prevalence questions (PQ). Instances include metric and boolean queries. For example, was a particular function added/deleted/modified? Or how many and which of the functions are reused? The questions asked indicate the purpose of the mining approach.

Given the code base and the purpose of the MSR being undertaken two main strategies to answer the proposed questions are detailed by Kagdi: One coined interested in *changes to properties* dealing mostly with high-level aspects of the software evolution and metrics computed across versions, and a second approach that focuses on *changes to artifacts* that focus on the specific differences between versions to measure the evolutionary aspects, rather than consolidated metrics or indexes [14].

In this proposed approach to software evolution sonification the focus is not answering questions through MSR but rather using it to display the software evolutionary aspects in a timely and unobtrusive way, while mantaining a high apprehension rate. The strategy we elected for processing the data from the software repositories focuses on the changes to properties (through metric analysis) rather than internal measuring in order to grasp the system's evolutionary state, internal measuring can still be utilized if the existing are be unable to provide one or more necessary aspects for the desired sonification.

It is important to note that this approach doesn't contemplate the specialized software evolution metrics developed by Lehman and Ramil in [18], the rationale for that is that those metrics focus heavily on cost-estimation and applied aspects of project management, whereas this work focuses on program comprehension.

3.4 Sound Synthesis

Martin Russ [20] defines Sound Synthesis as:

> (...) the process of producing sound. It can reuse existing sounds by processing them, or it can generate sound electronically or mechanically. It

> may use mathematics, physics or even biology; and it brings together art and science in a mix of musical skill and technical expertise(...)

Which is a broad, but sufficient definition for the purposes of this work, in which the ultimate goal is to electronically generate sounds that are both meaningful and musical.

4. IMPLEMENTATION

The proposed framework was developed using the *Java* programming language, chosen due to its good balance of productivity, debuggability and the pre-existence of various libraries and components necessary for this implementation, the *Eclipse* [11] IDE for the development process because of its good community and amount of extensions available.

The following subsections elaborate on the development decisions, processes and technologies employed in each individual module, Figures 2 to 6 depict the implementation of the framework's modules in order to conduct the validation experiments detailed in Section 6, several relationships and packages were hidden from these figures to streamline the diagrams.

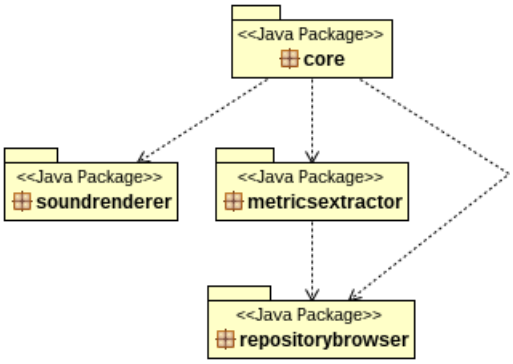

Figure 2: Package diagram for the framework.

4.1 Repository Browser

The repository browser is responsible for downloading a code repository, and provides interfaces for transversing it, in addition to reading and writing the repository's data and metadata.

While the tool's architecture allows for VCS flexibility, *Git* was chosen for the initial implementation of this work due to its widespread adoption in open-source software (OSS) projects and its ability to work with repository clones locally, eliminating the overhead that is caused by having to *continuously* retrieve version information over the network from a centralized VCS such as *Subversion*. While *Subversion* is better documented and more mature, the performance improvements and decentralized nature of *Git* weighted in on the final choice.

In order to embed *Git* functionality in the framework, the *JGit* [22] library was utilized. *JGit* was selected as the *Git* provider due to its lightweight, good performance, low number of dependencies, good documentation and successful deployment in large-scale projects such as *EGit* [1] and *Gerrit* [2].

The highlight of the current repository browser implementation is the ability to browse the commits in different ways through custom filtering. New filters can be created by extending from a abstract class in the framework and writing the custom filtering logic in the abstract methods. Once created, filters can be mixed and matched to meet the most diverse requirements in terms of detail, levels of representation and performance.

All the code pertaining to the repository browser implementation is in the project's `repositorybrowser` Java package, depicted in detail on Fig. 3 with its internal classes, one of which is the `Commit` class, which wraps around *JGit's* `RevCommit` class to provide some utility accessors.

The `GitRepositoryHandler` class is responsible for loading a *Git* repository (either local or remote), with the added ability to download remote repositories into local clones. The class provides the `getRevisions` and `getAllRevisions` methods to retrieve revisions from the loaded repository with or without filtering, respectively.

The `CommitFilter` abstract class provides the basis for the creation of custom commit filters. It does most of the generic filtering logic in itself, does some safety checks (for instance, makes sure that the child classes are calling it's constructor correctly) and declares the abstract method `doFilterCommit` which custom filters must use to implement their custom filtering logic, the custom filtering logic for one such filter is shown in Code Listing 1.

Code Listing 1: Custom filtering logic for the `StepCommitFilter` class.

```
public void doFilterCommit() {
      RevCommitList<RevCommit> filteredCommits = new
           RevCommitList<RevCommit>();
      RevCommitList<RevCommit> commits = this.
           getCommits();
      Integer step = (Integer) this.getArgs()[0];

      if (step > commits.size()) {
         filteredCommits.add(commits.get(0));
         filteredCommits.add(commits.get(commits.size
              () - 1));
      } else {
         for (int i = 0; i < commits.size(); i += step
              ) {
            RevCommit revCommit = commits.get(i);
            filteredCommits.add(revCommit);
         }
         if (commits.size() % step != 0) {
            filteredCommits.add(commits.get(commits.
                 size() - 1));
         }
      }

      this.setCommits(filteredCommits);
   }
```

From this listing, two important implementation details can be noted: first, the filter receives the arguments as an ar-

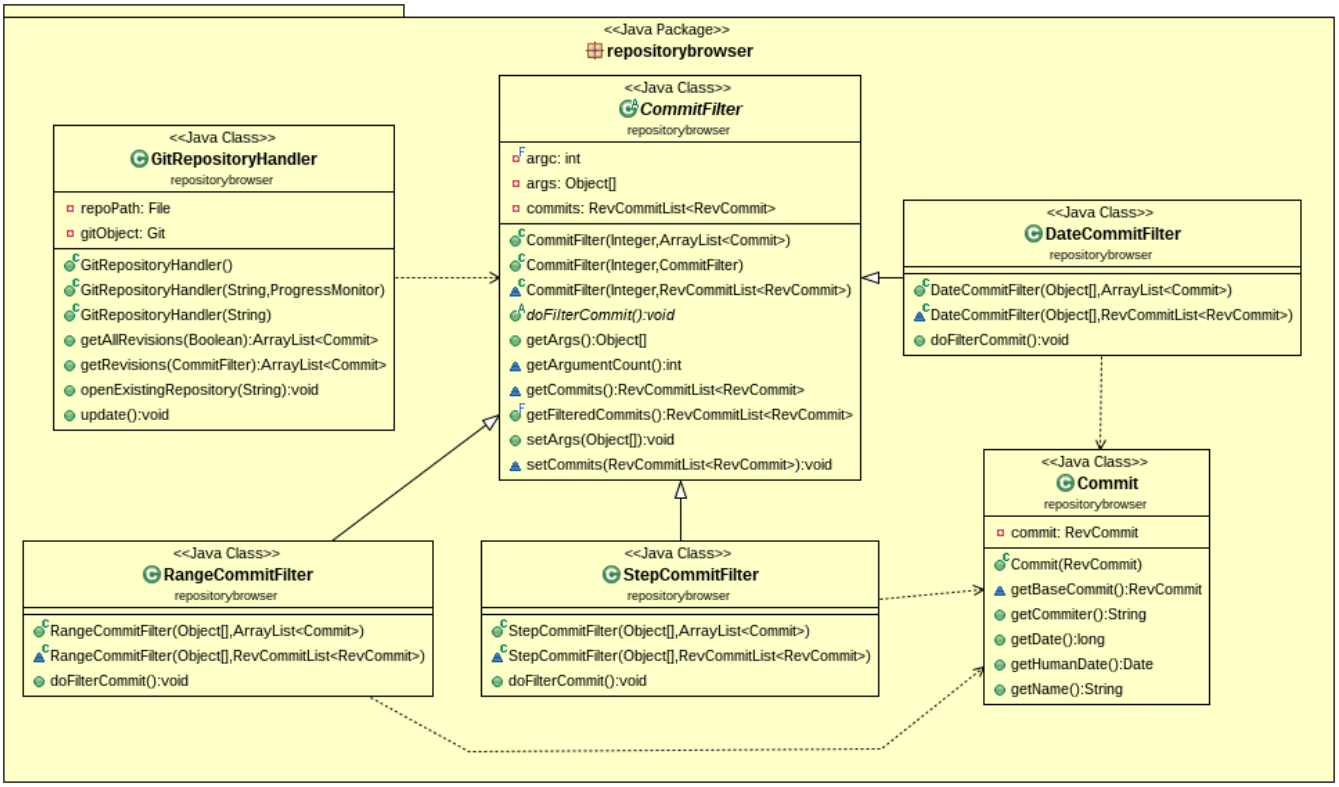

Figure 3: Diagram for the `repositorybrowser` package and its classes.

ray of `Object` (declared by the base class in order to allow flexibility) and therefore, its up to the programmer to type-check and document the number and type of the arguments that his custom filter takes as input; second, the method `setCommits` must be called at the end of the filtering process, in order to set the instance's `commits` property with the filtered commits and maintain the `getFilteredCommits` method as the sole outlet for the filtering operations.

4.2 Metrics Extractor

The metrics extractor retrieves information from a given source code revision and collects metadata from the repository and source code *while* the repository browser goes through the repository.

In the preliminary studies, both *GCC-XML* [13] and *Clang* [10] were considered as potential tools for metrics extraction. Further investigation revealed that, given the diversity and heterogeneity of software metrics that can be collected, its more productive to have the custom logic for each desired metric in its own class, and have the framework deploy the implemented metrics through inversion of control.

Custom logic for metrics dealing with metadata and contextual information can be written in terms of the language's constructs, while metrics that deal with static source code analysis should be retrieved through the invocation third party tools, and turned into evolutionary metrics through the addition of a temporal aspect. The variation for a given metric can be calculated across different source code revisions in order to entail evolutionary insight.

As seen in Figure 4, the framework includes two concrete classes that implement the `IMetricsExtractor` interface in order to extract two evolutionary metrics from the software repository's metadata: `CommittersPerMonthExtractor` retrieves the number of committers active per month while `CommitsPerMonthExtractor` calculates the total number of commits per month, these two metrics combined provide a good grasp of a software project's overall activity and tendency. Depending on what analysis the user wants to perform, the extracted metrics can be processed either raw to show disparities in projects' magnitudes or interpolated to place give both projects under a similar perspective. The complete source code for the `CommittersPerMonthExtractor` class is included in Appendix A.

The metrics extractor classes return, through the `getMetrics` method, a collection of key-value pairs with string identifiers and integer values, the `getOutputName` method, in turn, returns a string with the name the map should ba called by in the sonification templates (discussed in the next subsection).

4.3 Sound Renderer

After extracting the chosen metrics from the filtered revisions, a comprehensive mapping can be elaborated between the software metrics collected and the various aural events that will be defined according to the guidelines already present in the software sonification literature. Both musicality [27] and comprehensibility [24] of the auditory are taken into consideration.

The sound rendering module associates one or more sets

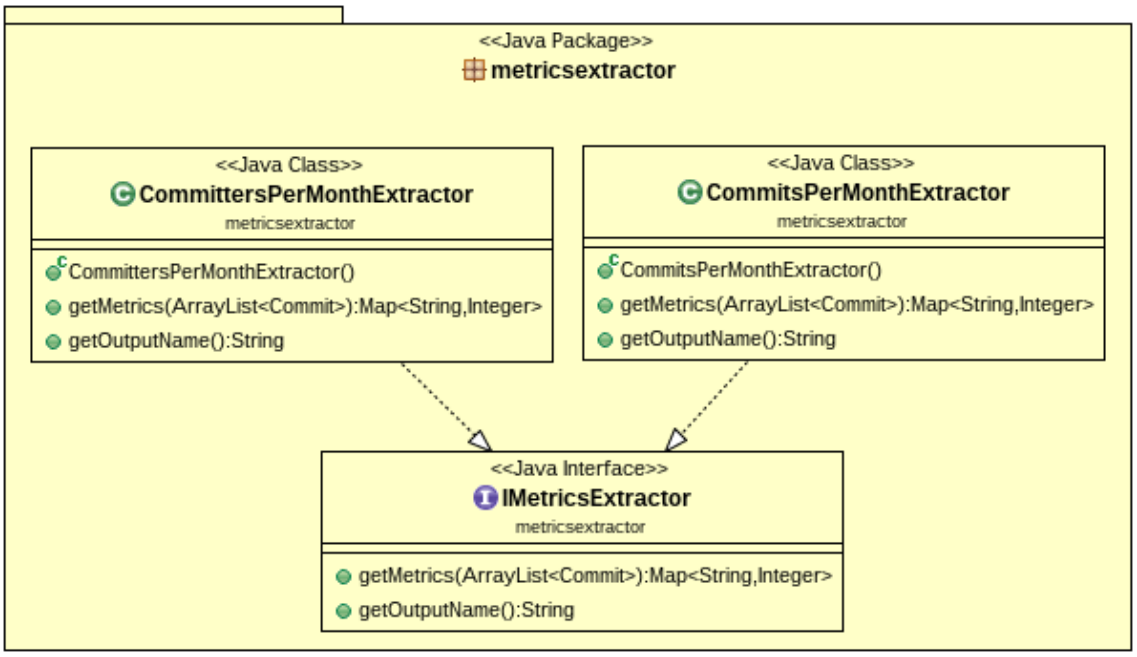

Figure 4: Diagram for the `metricsextractor` package and its classes.

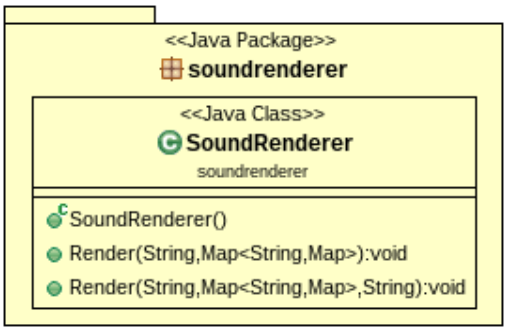

Figure 5: Diagram for the `soundrenderer` package and its sole class.

of consolidated metrics with one or more auralization templates to generate a MIDI music file that can be read by regular media players, two additional tools are used in this process: the *Freemarker* template engine [19] and the *Lilypond* music notation tool [17].

Freemarker processes both the metrics and the template (.ftl) files, outputting a *Lilypond* (.ly) that defines what sounds should be rendered. This file contains musical notes and additional information such as tempo, pitch and dynamics. *Lilypond* further processes the expanded .ly file and generates a MIDI sound file in addition to a PDF file with the corresponding music sheet.

The `soundrenderer` package (Figure 5) contains one single `SoundRenderer` class that generates the sonification through the `Render` method. Given the name of the template file, a map *of maps* containing the extracted metrics and, optionally, a filename for the output files, the class writes PDF, *Lilypond* and MIDI files in it's working directory. This implementation itself is very straightforward and the its most notable implementation detail is that it depends on *Lilypond* being installed and set up on the system's path as shown in Code Listing 2.

Code Listing 2: `SoundRenderer`'s `Render` method.

```
public void Render(String templateName, Map<String,
    Map> data,
     String fileName) throws IOException,
         TemplateException {
  Configuration cfg = new Configuration(Configuration
      .VERSION_2_3_20);
  cfg.setDirectoryForTemplateLoading(new File("./
      templates"));
  cfg.setDefaultEncoding("UTF-8");
  cfg.setTemplateExceptionHandler(
      TemplateExceptionHandler.RETHROW_HANDLER);

  Template temp = cfg.getTemplate(templateName + ".
      ftl");
  FileWriter fr = new FileWriter("./" + fileName + ".
      ly", false);
  temp.process(data, fr);
  Runtime.getRuntime().exec("lilypond " + fileName +
      ".ly");
}
```

The sonification template bundled with the framework maps the default metrics, namely: the number of commits per month and number of committers per month to, respectively, the pitch and the number of notes that correspond to each month in the rendered sonification.

4.4 Application Core

The application core is responsible for communicating the specialized modules of the framework and providing an entry point for users that aren't particularly interested in extending the framework's capabilities, allowing them to work with whatever filters and metrics extractors are already implemented.

The `WorkingSet` class performs the data transfer operations needed by the framework. It takes user-provided information such as the source code repository's url and passes that on to the appropriate classes, provides methods to set-up filters, extractors and parameters and also provides an interface for the user to access all the individual modules and parameters involved in the sonification process.

The *Maestro* class does the actual *execution* of the sonification process, it *knows* the sonification procedure, while the *WorkingSet* class *holds* the assets and data necessary to undertake the task.

Each `Maestro` instance is initialized with a `WorkingSet` instance and provides no accessors to its internal fields (Figure 6), encouraging users to properly configure the working sets *before* passing them for sonification.

The `makeMusic` method in the `Maestro` class is the single entry point for the sonification process in the entire framework, while maintaining a very simple and efficient implementation (shown in Code Listing 3), this was achieved through meticulous encapsulation and consideration of the single responsibility principle throughout the framework.

Code Listing 3: `Maestro` class `makeMusic` method.

```
public void makeMusic() throws NoHeadException,
     GitAPIException,
        IOException, TemplateException {
      GitRepositoryHandler handler = this.ws.
           getRepoHandler();
      ArrayList<Commit> commits = new ArrayList<Commit
           >();
      commits = handler.getRevisions(this.ws.
           getFilters());

      Map<String, Map> data = new TreeMap<String, Map
           >();

      for (IMetricsExtractor extractor : this.ws.
           getExtractors()) {
         data.put(extractor.getOutputName(), extractor
              .getMetrics(commits));
      }

      // This a set that maps numbers to musical notes
      data.put("noteMap", NoteMap.getInstance());

      if (this.ws.getFileName() != null) {
         this.ws.getSoundrenderer().Render(this.ws.
              getTemplateName(), data,
               this.ws.getFileName());
      } else {
         this.ws.getSoundrenderer().Render(this.ws.
              getTemplateName(), data);
      }
   }
```

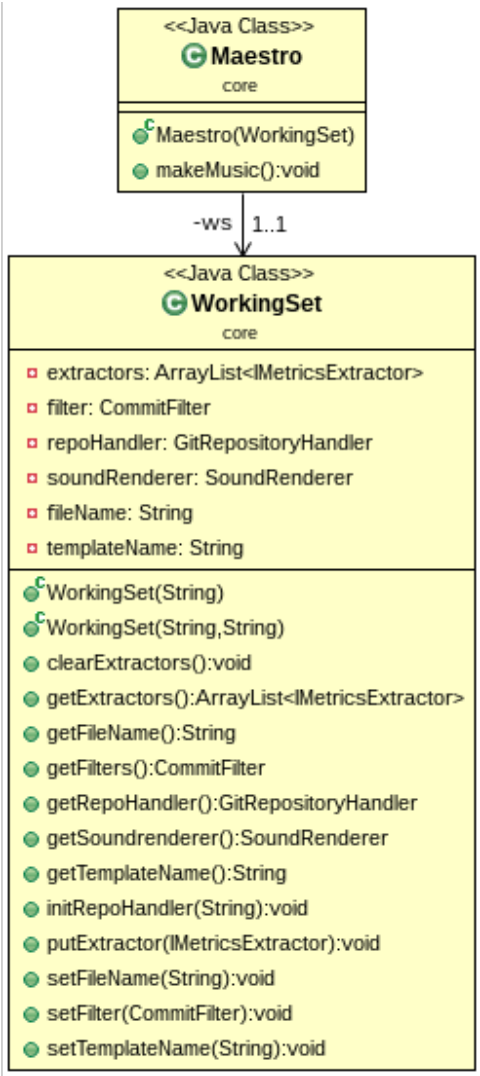

Figure 6: Detail of the classes contained by the `core` package.

5. RELATED WORK

Works that attempt to employ and evaluate sonifications in order to help specifically with program comprehension date back as far as 1990, as shown in Table B.5, this section presents next a brief timeline considering some of such works, followed by a brief discussion of their confluence and their differences to the current work.

Sonnenwald et al. proposed, in 1990, the *InfoSound* [23] tool in order to help subjects understand rapidly succeeding events in software behavior though sonifications, and provided enough validation in their studies to support the tool; Francioni and Jackson, in 1993, furthered the studies in behavioral software sonification by focusing on programs with parallel execution [12], and determined that users could correctly identify entities through sounds alone.

From 1998 to 2002, Vickers and Alty published works that sonified code structure as well as it's behavior, ran experiments to evaluate the tools didactically and musically and

also worked towards organizing some principles for musical program auralizations. Vickers also proposed, on his 1996 PhD thesis [26], the CAITLIN tool for auralization of software behavior and structure, validated by a controlled experiment performed with novice pascal developers.

While there were significant studies all throughout, the first tool to include evolutionary aspects in software sonifications was *CocoViz*, and only moderately so. Bocuzzo and Gall, in their 2008 publication [9], explain that *CocoViz* tracks the percentile change of an entity in a period of time and uses auditory cues to alert when it changes more than a defined threshold, while this may look like a simple sonification effort, controlled experiments in the same study highlighted the efficacy of these auditory cues.

All of these publications, along with the additional works presented in Table B.5 contributed to our effort by laying the foundations necessary to begin sonification work and providing initial insights on what works and what doesn't when trying to convey technical information to the user, with both theoretical and practical arguments to corroborate their claims.

Despite all the considerable efforts in the field, the evolutionary aspects of computer programs is severely less explored if compared to behavioral and structural aspects. The present work attempts to build upon the existing bibliography by bringing more emphasis to the evolution of software projects, presenting several methodological guidelines for evolutionary sonification and a lean and flexible sonification framework.

6. VALIDATION

In order to evaluate the proposed framework, a case study was conducted. The experimental procedure, their goals and our findings will be detailed in the following subsections.

6.1 Case Study

6.1.1 Goals

The case study was undertaken in order to assert whether or not the sonifications rendered from evolutionary data from source-code repositories are meaningful and easily understood.

Additionally, this case study helps determine whether or not the basic implementation of the framework has enough assets to render useful sonifications by itself.

6.1.2 Procedure

For this case study, two *open-source* office suites were selected as subjects: The *KOffice* suite – whose developed ceased in 2013 and the *Libreoffice* suite – whose development is still well underway and is largely utilized by the open-source community.

Evolutionary sonifications were rendered for both suites factoring in the code revisions in the period between *01/01/2010* and *31/04/2013*, inclusively. The finish date for the sonification period corresponds to the month *before* the last non-automatic commit of the *KOffice* project, because *KOffice*'s activity ceased midway through the month, whereas *Libreoffice* continued all throughout.

A simple runner class with an associated GUI (Graphical User Interface) was put together to conduct this study, the interface consists of an editable text field and a button that invokes the sonification process (Code Listing 4).

Figure 7: Graphical User Interface of the runner class.

Code Listing 4: Excerpt of the method invoked by the GUI after a click event is registered.

```
SimpleDateFormat dateFormat = (SimpleDateFormat)
    DateFormat
     .getDateInstance();
dateFormat.applyPattern("dd/MM/yyyy");

Object[] dates = { dateFormat.parse("01/01/2013"),
     dateFormat.parse("31/12/2014") };

WorkingSet ws = new WorkingSet("template_bassline");
ws.initRepoHandler(txtRepoURL.getText());
ws.setFilter(new DateCommitFilter(dates, ws
     .getRepoHandler().getAllRevisions(true)));

ws.putExtractor(new CommitsPerMonthExtractor());
ws.putExtractor(new CommittersPerMonthExtractor());

Maestro vivaldi = new Maestro(ws);
vivaldi.makeMusic();
```

The monthly number of commits in each project was mapped to the pitch representing that specific month, while the number of committers for a given month corresponded to the number of notes it lasts for, as specified by the default template file. The extracted data was interpolated to make sure the minimum and maximum number of commits correspond to the pitches of C2 (two octaves below the middle C) and C8 (four octaves above the middle C), while the minimum and maximum number of committers correspond to, respectively, 2 and 8 notes; such interpolation strategy was deployed in the extractor classes to represent both projects in a similar perspective, even through their numbers were in slightly different orders of magnitude.

The finished sonifications were qualitatively analyzed in order to determine if the desired mappings were correctly reproduced in the sonification, if both projects were represented in a similar fashion, and if it's possible to get an insight on the project's health and activity through sonification alone. The raw data – extracted in the second part of the sonification process – can be found in Tables B.1, B.2, B.3 and B.4, all in Appendix B.

6.1.3 Results

Despite the project's numbers being in different orders of magnitude (*Libreoffice* has always had a lot more activ-

ity than *KOffice*), the sonifications were able to represent both projects in a comparable scale, the musical scores corresponding to each of the rendered sonifications can be seen in Appendix C, Figures C.1 and C.2, as a printed alternative to the sonification experience.

Through analysis of the sonifications, some conclusions were drawn: first, it is possible to coherently map data to aural events; second, through sonification it's possible to analyze large and small software projects under a similar perspective, preserving the evolutionary trends of each project and investing roughly the same amount of effort for each project.

From an evolutionary standpoint, the sonifications evidenced what could be an important pattern. In *KOffice*'s sonification there were mostly extreme frequencies, meaning that there were many moments of heavy development and long periods of very modest development. In contrast, *Libreoffice*'s sonification had mostly moderate frequencies, with the pitches varying in a roughly wavelike pattern, even in its most extreme moments, meaning that development sprints were cyclical and well-defined.

7. CONCLUSION

Visualization solutions to aid software developers comprehend computer programs suffer from the same scalability problem as program comprehension itself, their representations get exponentially more confusing and complex as the project's complexity and size increase. Given the room for improvement left by the current tools, their considerable success in aiding developers and their shortcomings, it's reasonable to prospect other mediums to aid in program comprehension.

This work draws from studies in the established areas of SMC, Auditory Display and Software Evolution in order to propose and detail a methodology and a extensible framework to convey evolutionary software information through sound alone.

The validation employed shows that sonifications are not only able to transmit evolutionary information to the user, but also escalate very well with the size and complexity of the software projects involved, theoretically requiring the same amount of training, time and effort regardless of the software being analyzed.

The results obtained here hint at the potential of this approach. Future publications planned in this field of research include a systematic literature review of software sonification for program comprehension, which was performed in tandem with this article and experimental studies to further validate the solution.

8. REFERENCES

[1] Egit - eclipse team provider for git. http://eclipse.org/egit/.

[2] Gerrit code review. https://code.google.com/p/gerrit/.

[3] Max-msp. http://cycling74.com/products/max/. Accessed: 16/07/2014, 2014.

[4] Berman, L. *Program Comprehension Through Sonification*. PhD thesis, Durham University, 2011.

[5] Bernardini, N., and de Poli, G. The sound and music computing field: Present and future. *Journal of New Music Research 36*, 3 (2007), 143–148.

[6] Bly, S. Presenting information in sound. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems* (New York, NY, USA, 1982), CHI '82, ACM, pp. 371–375.

[7] Bly, S. *Sound and Computer Information Presentation*. PhD thesis, 1982. AAI8220127.

[8] Boccuzzo, S., and Gall, H. C. Cocoviz with ambient audio software exploration. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 571–574.

[9] Bocuzzo, S., and Gall, H. Software visualization with audio supported cognitive glyphs. In *2008 IEEE International Conference on Software Maintenance* (2008).

[10] C language family frontend for llvm. http://clang.llvm.org/index.html. Accessed: 08/04/2014, 2003-2013.

[11] Foundation, E. Eclipse. http://eclipse.org/.

[12] Francioni, J., and Jackson, J. Breaking the silence: Auralization of parallel program behavior. *Journal of Parallel and Distributed Computing 18*, 2 (1993), 181 – 194.

[13] Xml output for gcc. http://gccxml.github.io/HTML/Index.html. Accessed: 08/04/2014, 2002-2012.

[14] Kagdi, H., Collard, M. L., and Maletic, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution, 2007.

[15] Kazuya, S., Shigeyuki, H., Kazutaka, M., and Minoru, T. Codedrummer: Sonification methods of function calls in program execution. *IPSJ SIG Notes 2011*, 14 (feb 2011), 1–6.

[16] Lehman, M. M. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Softw. 1* (Sept. 1984), 213–221.

[17] Nienhuys, H.-W., and Nieuwenhuizen, J. Gnu lilypond. http://lilypond.org/, 1996–2015.

[18] Ramil, J., and Lehman, M. Metrics of software evolution as effort predictors - a case study. In *Software Maintenance, 2000. Proceedings. International Conference on* (2000), pp. 163–172.

[19] Revusky, J., Szegedi, A., and Dékány, D. Freemarker. http://freemarker.org/, 2002–2015.

[20] Russ, M. Chapter 1 - background. In *Sound Synthesis and Sampling (Third Edition)*, M. Russ, Ed., third edition ed., Music Technology. Focal Press, Oxford, 2009, pp. 3 – 86.

[21] Scheirer, E. D. The mpeg-4 structured audio standard, 1998.

[22] Sohn, M., and Pearce, S. Jgit. https://eclipse.org/jgit/, 2010–2015.

[23] Sonnenwald, D. H., Gopinath, B., Haberman, G. ., Keese, W. M., and Myers, J. S. Infosound: An audio aid to program comprehension. In *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on* (1990).

[24] Stefik, A., Hundhausen, C., and Patterson, R.

An empirical investigation into the design of auditory cues to enhance computer program comprehension. *International Journal of Human-Computer Studies* (2011).

[25] Vercoe, B. The canonical csound reference manual, 1992.

[26] Vickers, P. *CAITLIN : implementation of a musical program auralisation system to study the effects on debugging tasks as performed by novice Pascal programmers*. PhD thesis, Loughborough University, 1999.

[27] Vickers, P., and Alty, J. Towards some organising principles for musical program auralisations. In *Proceedings of the Fifth International Conference on Auditory Display* (1998).

[28] Vickers, P., and Alty, J. L. Musical program auralisation: a structured approach to motif design. *Interacting with Computers 14*, 5 (2002), 457 – 485.

APPENDIX

A. CODE LISTINGS

Code Listing A.1: Source code for the CommittersPerMonthExtractor class.

```
public class CommittersPerMonthExtractor implements IMetricsExtractor {

   public CommittersPerMonthExtractor() {
   }

   @Override
   public Map<String, Integer> getMetrics(ArrayList<Commit> commits) {
      TreeMap<String, Integer> committersMonth = new TreeMap<>();
      TreeMap<String, LinkedHashSet<String>> committersMonthTemp = new TreeMap<>();

      for (Commit commit : commits) {Concrete
         @SuppressWarnings("deprecation")
         String key = ""
               + (commit.getHumanDate().getYear() + 1900)
               + "/"
               + Integer
                     .toHexString((commit.getHumanDate().getMonth() + 1));
         if (committersMonthTemp.containsKey(key)) {
            committersMonthTemp.get(key).add(commit.getCommiter());
         } else {
            LinkedHashSet<String> temp = new LinkedHashSet<String>();
            temp.add(commit.getCommiter());
            committersMonthTemp.put(key, temp);
         }
      }

      Iterator<Entry<String, LinkedHashSet<String>>> iteTemp = committersMonthTemp
            .entrySet().iterator();

      // Setting the actual committers/month value
      while (iteTemp.hasNext()) {
         Entry<String, LinkedHashSet<String>> atual = iteTemp.next();
         committersMonth.put(atual.getKey(), atual.getValue().size());
      }

      Iterator<Entry<String, Integer>> ite = committersMonth.entrySet()
            .iterator();

      // Ppm = People per month
      int minPpm = Integer.MAX_VALUE;
      int maxPpm = Integer.MIN_VALUE;

      while (ite.hasNext()) {
         Entry<String, Integer> atual = ite.next();
         int atualInt = atual.getValue();
         if (atualInt < minPpm) {
            minPpm = atual.getValue();
         }
         if (atualInt > maxPpm) {
            maxPpm = atual.getValue();
         }
      }

      Interpolator interpolator = new Interpolator(2, 8, minPpm, maxPpm);

      ite = committersMonth.entrySet().iterator();
      while (ite.hasNext()) {
         Entry<String, Integer> current = ite.next();
         current.setValue(interpolator.interpolate(current.getValue()));
      }
      return committersMonth;
   }

   @Override
   public String getOutputName() {
      return "committersMonth";
   }
}
```

B. TABLES

Table B.1: KOffice – Committers per month.

Committers per month			
2010/1	32	2011/9	2
2010/2	30	2011/10	1
2010/3	34	2011/11	1
2010/4	25	2011/12	3
2010/5	35	2012/1	3
2010/6	31	2012/2	2
2010/7	32	2012/3	1
2010/8	32	2012/4	1
2010/9	30	2012/5	1
2010/10	27	2012/6	1
2010/11	24	2012/7	1
2010/12	10	2012/8	3
2011/1	6	2012/9	1
2011/2	3	2012/10	2
2011/3	4	2012/11	1
2011/4	2	2012/12	1
2011/5	4	2013/1	1
2011/6	3	2013/2	1
2011/7	3	2013/3	1
2011/8	4	2013/4	2

Table B.2: KOffice – Commits per month.

Commits per month			
2010/1	602	2011/9	5
2010/2	455	2011/10	7
2010/3	474	2011/11	4
2010/4	504	2011/12	59
2010/5	331	2012/1	9
2010/6	447	2012/2	7
2010/7	388	2012/3	13
2010/8	543	2012/4	1
2010/9	498	2012/5	3
2010/10	299	2012/6	6
2010/11	264	2012/7	1
2010/12	156	2012/8	5
2011/1	228	2012/9	1
2011/2	284	2012/10	3
2011/3	168	2012/11	1
2011/4	40	2012/12	2
2011/5	172	2013/1	3
2011/6	383	2013/2	1
2011/7	113	2013/3	4
2011/8	106	2013/4	4

Table B.3: Libreoffice – Committers per month.

Committers per month			
2010/1	64	2011/9	38
2010/2	62	2011/10	34
2010/3	62	2011/11	40
2010/4	60	2011/12	40
2010/5	55	2012/1	42
2010/6	63	2012/2	44
2010/7	51	2012/3	45
2010/8	47	2012/4	39
2010/9	59	2012/5	45
2010/10	65	2012/6	46
2010/11	73	2012/7	41
2010/12	69	2012/8	47
2011/1	73	2012/9	42
2011/2	73	2012/10	42
2011/3	71	2012/11	50
2011/4	37	2012/12	48
2011/5	38	2013/1	41
2011/6	34	2013/2	45
2011/7	35	2013/3	53
2011/8	35	2013/4	55

Table B.4: Libreoffice – Commits per month.

Commits per month			
2010/1	1623	2011/9	1319
2010/2	1134	2011/10	1158
2010/3	1610	2011/11	1865
2010/4	1409	2011/12	1392
2010/5	1225	2012/1	1590
2010/6	1592	2012/2	1563
2010/7	1054	2012/3	1688
2010/8	938	2012/4	1497
2010/9	1553	2012/5	1431
2010/10	2516	2012/6	1627
2010/11	2958	2012/7	1799
2010/12	2161	2012/8	1549
2011/1	3178	2012/9	1623
2011/2	2426	2012/10	1645
2011/3	2636	2012/11	1959
2011/4	1485	2012/12	1331
2011/5	1768	2013/1	1310
2011/6	2167	2013/2	1762
2011/7	1626	2013/3	2459
2011/8	1497	2013/4	2088

Table B.5: Extracted data from works unearthed during the literature review process

Title	Author(s)	Year	Sonification Goal	Sonified Aspect
InfoSound: an audio aid to program comprehension	Sonnenwald, D. Haberman, O. Keese, W. Myers J. S.	1990	Program Comprehension	Behavior
Breaking the Silence: Auralization of Parallel Program Behavior	Francioni, J. Jackson, J.	1993	Program Comprehension	Behavior
Towards some Organising Principles for Musical Program Auralisations	Vickers, P. Alty, J.	1998	Debugging Assistance Program Comprehension	Structure Behavior
CAITLIN: implementation of a musical program auralisation system to study the effects on debugging tasks as performed by novice Pascal programmers	Vickers, P.	1999	Debugging Assistance Program Comprehension	Structure Behavior
Musical Program Auralisation: Empirical Studies	Vickers, P. Alty, J.	2000	Debugging Assistance Program Comprehension	Structure Behavior
Using music to communicate computing information	Vickers, P. Alty, J.	2002	Program Comprehension	Structure Behavior
An Empirical Comparison Of Program Auralization Techniques	Stefik, A.	2005	Debugging Assistance	Behavior
A Tool For Auralized Debugging	Chen, Y.	2005	Debugging Assistance IDE Extension	Behavior
Listening to Program Slices	Berman, L. Gallhager, K.	2006	Program Comprehension	Structure
The Well-tempered Compiler? The Aesthetics of Program Auralization.	Vickers, P. Alty, J.	2006	Aesthetics	Structure Behavior
On the role of senses in education	Kátai, Z. Juhász, K. Adorjáni, A.	2008	Teaching	Structure
Software visualization with audio supported cognitive glyphs	Bocuzzo, S. Gall, H.	2008	Program Comprehension	Structure Evolution
CocoViz with ambient audio software exploration	Bocuzzo, S. Gall, H.	2009	Program Comprehension	Structure
Using Sound to Understand Software Architecture	Berman, L. Gallhager, K.	2009	Program Comprehension IDE Extension	Structure
Sonification Design Guidelines to Enhance Program Comprehension	Hussein, K. Tilevich, E. Bukvic, I. Kim, S.	2009	Program Comprehension IDE Extension	Structure
Sound as an Aid in Understanding Low-Level Program Architecture	Berman, L.	2010	Program Comprehension	Structure
Program Comprehension Through Sonification	Berman, L.	2011	Program Comprehension IDE Extension	Structure
An empirical investigation into the design of auditory cues to enhance computer program comprehension	Stefik, A. Hundhausen, C. Patterson, R.	2011	Debugging Assistance Program Comprehension IDE Extension	Structure Behavior
CodeDrummer: Program audible system in which attention is focused on the rhythm (machine translated title)	Kazuya, S. Shigeyuki, H. Kazutaka, M. Minoru, T.	2011	Program Comprehension Entertainment	Behavior

C. MUSICAL SCORES

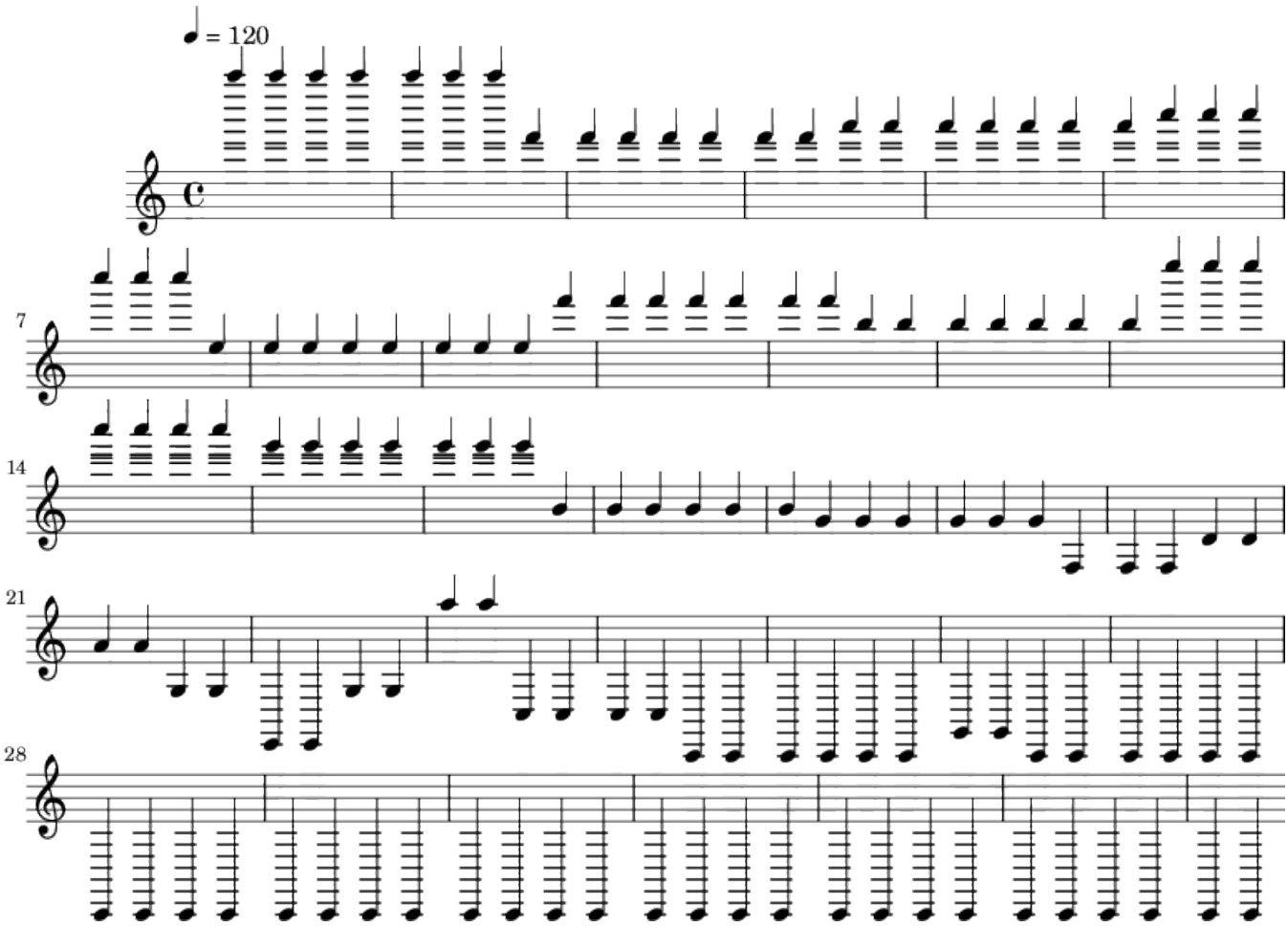

Figure C.1: Musical score for KOffice's evolutionary sonification.

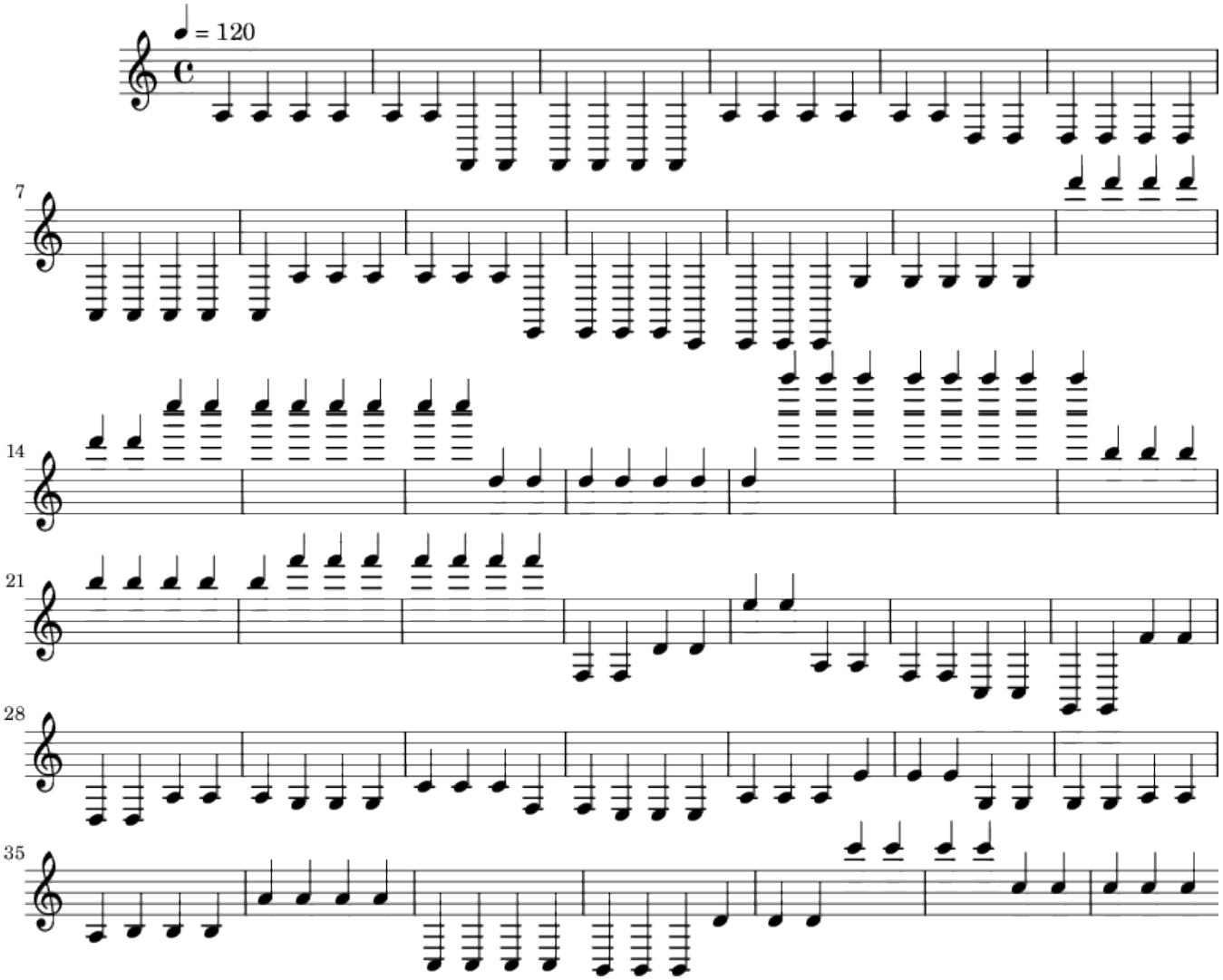

Figure C.2: Musical score for Libreoffice's evolutionary sonification.