AN EMPIRICAL COMPARISON OF PROGRAM AURALIZATION TECHNIQUES

By

ANDREAS MIKAL STEFIK

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2005

ACKNOWLEDGEMENT

Thanks goes out to Kelly Fitz, who put up with my constant revisions, often before he could read the previous draft.

AN EMPIRICAL COMPARISON OF PROGRAM AURALIZATION TECHNIQUES

Abstract

by Andreas Mikal Stefik, M.S.
Washington State University
December 2005

Chair: Kelly Fitz

This thesis presents a new approach to using music for human computer interaction, layered program auralization. I use layers of musical structure to represent the state and behavior of a computer program while it is running, taking advantage of metaphorical relationships between musical structure and programming constructs. Layers overlap one another, and can intelligently collaborate to create meaningful mappings from program state or behavior to music. I describe three possible layers in this new system. A dynamically controlled tonal structure changes the harmony while a computer program is running. Program state is represented by changes in the orchestration during execution. Lyrics add semantic information that is difficult to represent with music alone.

One possible application of layered program auralization is in debugging runtime behavior of computer programs. Three programs, with faults strategically added, were written to test the effectiveness of layered program auralization. The three programs created included a roulette game, a bank automatic teller machine, and an address book. An empirical study was conducted comparing the effectiveness of three groups of participants while debugging these programs. The first group of participants were given no auralizations, the second strictly musical auralizations, and the third musical auralizations with additional lyrics. Three sessions of experiments were run, the first of which without training into how the auralizations work. In the last two sessions of the experiment, participants in the music and music plus voice groups were given training in the auralizations.

Results indicate that layered program auralization was effective in the music group for the bank example, but may not work for every type of computer program, using the current auralization design. Interestingly, users debugging control flow aspects of programs found more errors than other types of programs, like linked structures. In addition, in the music plus voice group, subjects were found to debug less effectively than the control group if they were given no training, although this effect was not seen with the music only group.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Dedication

To my wife.

CHAPTER 1

INTRODUCTION

Typical software does not go from design to implementation without error. Errors of design, errors of requirements, and implementation errors can be a serious cause of confusion and frustration. In this thesis, I propose the use of sound during the debugging process to help find faults while computer programs are executing. I use layers of musical structure to represent the state and behavior of a computer program while it is running, taking advantage of metaphorical relationships between musical structure and programming constructs. Musical layers overlap one another, often similar to viewgraphs on an overhead projector.

1.1 The Layered Program Auralization approach

Musical layers are mapped to program constructs, like IF or WHILE. A layer is, “A method of generating music, or properties of music, from behavior or data.” While a computer program runs, program constructs are aurally enumerated, meaning sounds are emitted from the computer. Sounds emitted relate to the computer code. Creating this mapping is non-trivial, and a general process for creating such a mapping does not exist in the literature. In addition, not all parts of a mapping from a running program to sound are equally difficult. Some program constructs, like an IF, can be mapped to metaphorically equivalent sounds, or even music. Other characteristics, even those as simple as representing an arbitrary string or number, have no known, general case, metaphorical mapping. Mapping sounds to computer programs may benefit computer users, for example those trying to debug computer programs.

This thesis introduces two concepts unique to the program auralization literature. The first is the application of layers to sound mappings [16]. Mappings are divided into layers that work well for any given part of a mapping, and these layers are transparently combined during auralization. The second is the use of lyrics to represent program constructs where no metaphorical mapping exists.

For the purpose of debugging, I define three main types of layers, the cadential layer, the orchestration layer, and the lyrics layer. In the cadential layer, program structure is encoded into musical cadential patterns, a common pattern of chords often used in classical music to indicate the end of a phrase. The orchestral layer uses the number of instruments currently playing to represent the state of a computer program. The lyrics layer is used to represent semantic information with no obvious metaphorical mapping, like numbers and text.

1.2 Experimentation

I conducted an empirical study in order to test the effectiveness of the layered auralization approach. Novice programmers in an advanced data structures course, sophomore year, were asked to debug auralized computer programs. There were three experimental groups, those with no auralizations, and two groups with different types of auralizations.

The first non-control group was given auralizations with the cadential layer and the orchestral layer. This group had no sounds mapped to semantic data. The second group was given the cadential layer, the orchestral layer, and the lyrics layer. The audio for this group was extremely similar to the first, but the lyrics were recorded, by a musician, and played back as an additional layer while debugging.

Lastly, it is possible that the subjects will require training to use layered program auralization techniques effectively. Techniques may or may not be effective without training, and may become more effective with additional training. The experiment was designed to test participants effectiveness without training, a second test with some training, and a third test with additional training.

CHAPTER 2

BACKGROUND RESEARCH

In program auralization, sound is mapped to behavior or data. In my case, this means mapping running computer programs to sound, and especially music. Other researchers have used sound for a variety of purposes. For example, sound is used to help users perform a task like debugging a program [48], is used to help the blind navigate hierarchical tree structures [38], and is used to help users understand normal computing events, like copying a file [17]

One important concept for program auralization was first defined by Gaver when working on the SonicFinder. Gaver defines two types of sound mappings, conceptual and perceptual [17, 16]. A conceptual mapping maps parts of the computer, transistors, gates, data structures, to a model world. This can be thought of as a metaphor for the way the computer works, similar to thinking of the computer as a desktop, with files and folders. A perceptual mapping maps the model world into something a user can perceive, like a file or a menu. There are three types of perceptual mappings, symbolic, metaphorical, and iconic or nomic.

A symbolic mapping has meaning only by convention, say a beep at an arbitrary frequency. A metaphorical mapping makes use of a non-literal relationship between an object and its representation. An example is filling up a water glass representing copying a file. An iconic, or nomic, representation looks or sounds like the thing it is trying to represent. For example, a nomic mapping of deleting a file could be represented by a crash sound, a crunching sound, or an explosion.

Gaver later worked on representing “dimensional” information in sounds [18], and proposes using sound synthesis techniques to parameterize the sounds. He presents this approach to give auditory icons more parameterization capabilities. For example, a file could sound large if it is large, or could sound small if it is small [18].

Sonnenwald describes an architecture for InfoSound, a program auralizer [40]. Sonnenwald used several types of auralizations, including speech, everyday sounds like a telephone ringing, in addition to musical events. These sounds corresponded to various events, where a telephone ringing could, for the most obvious presentation, represent someone calling on a phone. Music can, instead, represent events that are more difficult to define with a telephone ringing, or other every day sounds. In the case of InfoSound, InfoSound used music to define abstract events in parallel processing.

Earcons are defined as "... nonverbal audio messages used in the user-computer interface to provide information to the user about some computer object, operation, or interaction" [27]. In this context, Blattner considers computer objects to be things like files or menus and operations to be things like editing or compiling. Editing a file would be an example of an interaction that could be auralized with an earcon.

The first empirical studies conducted on earcons were completed by Brewster [8]. Brewster determined that earcons were more effective for portraying information than unstructured sound. Two experiments were conducted, and with that data, general guidelines for creating earcons were developed.

Knuth gives an interesting, and humorous, account of the complexity of song. In [23], reprinted in [24], Knuth discusses the length of songs in relation to the length of the text for those songs. For example, if a song has a refrain, and then m verse refrain pairs, the total space complexity of the song is cm, where $c < 1$. Other songs have further reductions in space complexity. Old Macdonald has a complexity of $(20 + \lambda + \alpha)\sqrt{n/(30 + 2\lambda)} + O(1)$, the 12 days of christmas has a complexity of $\sqrt{n/\log n}$, and n bottles of beer on the wall has a complexity of $O(\log n)$ [24].

Boardman created the language Listen, now reimplemented in Java and called JListen [6]. The original motivation for Listen was to create a tool for describing how computer source code can be auralized. This work included the creation of the LSL, Listen specification language, that allows auralization code to be put into other computer programs. This specification tells the Listen system how to interpret computer code it receives, and what to auralize in that code. So, in Listen, the focus was on inputting code and outputting auralized code, but not on the actual sounds themselves.

New work into JListen has included the creation of an auralized web server [32] and an intrusion detection system [19].

Francioni et al describe an auralization of parallel programs [14, 12, 13]. In these papers, a parallel program's behavior is defined as a series of events that can be auralized, or played, during program execution. Other work by Francioni in program auralization was intended to help users with disabilities, especially non-sighted users. This work included an auralizing Java programming tool, called JavaSpeak, which was created for visually impaired computer science students [39, 15, 38].

In 2005, Rigas and Alty present aural information about a 40 by 40 grid to visually impaired users [34]. The results of the experiment indicated that, even without visual aid, users were able understand the graphical information presented. Similarly, Alty used structured music stimuli to help blind users understand shapes, which in turn formed diagrams [3, 2].

Bonar et al conducted empirical studies of novice programmers in an attempt to understand why programming is difficult for novices [7]. They conducted video taped interviews with subjects to try and understand the reasons why novices were making certain types of faults. Spohrer and Solloway discuss several of the most common, or high frequency, faults, although they call them bugs, created in novice programs [42]. Data from several universities was collected on a series of computer science problems. Each problem involved typical program constructs, like file I/O, loops, if-else like constructs, and several others. Faults were then categorized by the frequency in which they occur in novice computer programs.

Chmeil [9] gave training to novice debuggers in an attempt to increase their debugging skills in a CS1 course. They found, given debugging training, that students required significantly less time to debug programs than students who were not given debugging training. Ahmadzadeh created a two phase debugging study, where they first tested student programs for common compilation errors, and second tested for logic errors [1]. Results indicated that subjects who were considered good programmers, according to the author's criteria, were not necessarily good debuggers. They also found that weak programmers did not tend to use a desktop debugging strategy, like using print statements to indicate output at points in the program.

Alty discusses the importance of audio in computer programs, and gives examples of its potential use to help non-sighted users with computers [4]. Rigas used diatonic structures, specifically those from the major scale, stereophony, and multiple timbres in an attempt to communicate information to subjects about a sorting algorithm [33]. The results of this study on sorting algorithms indicate that subjects can determine information about single entities or sets of information over execution, like the ordering of elements during a bubble sort [33].

Paul Vickers uses tonal musical structures to design his auralizations [44, 47, 46, 48]. With his program, CAITLIN, Vickers describes his method of program auralization for the Pascal programming language. His auralizations are loosely based on tonal theory models, and considered the use of characteristics like harmony and melody. One of the most important aspects of his work is that Vickers ran empirical experiments on his auralizations. In these experiments, he showed a positive correlation between use of the auralizations and debugging toy Pascal computer programs.

One interesting concept in auralization to appear in Vickers is the so called "Point of Interest," or (POI). Vickers describes this concept as:

> "A point of interest is a feature of a construct the details of which are of interest to the programmer at execution." [44]

Since Vickers used Pascal in his CAITLIN sonification environment, he considered the points of interest that are encompassed by that language. For example, a major chord could indicate a true in an IF statement, and a minor chord could indicate a false [48]. In a point of interest, musical or sonic events correspond to execution time events in the code.

An interesting topic in relation to program auralization techniques is the concept of aesthetics. A recent topic of interest, for example in Scheirer [36], has been user's emotions when using an auralization, and its role in the larger topic of human computer interaction. Vickers discusses aesthetic computing [45], a topic of importance in auralization, as uninteresting auralizations may prevent users from adopting an auralized system.

Leplatre and McGregor present a case study is designed around an email notification system [25]. Several different types of auralizations are categorized, and the goal of the experiment was to discover aesthetic differences within the different sounds brought forth. The authors determined, through experiment, that the type of task that is analyzed, or what the user is asked to do, effects the aesthetic judgment of the participants significantly.

Several studies attempt to analyze the emotional impact of user interfaces in human computer interaction. Tractinsky discusses the general difficulties of considering aesthetics in HCI [43]. Light discusses the use of emotions in network media, and the ethics of manipulating those emotions [26]. In addition, Light consider's the use of text in different styles, for example a passive or corporate sounding [26] "voice" and then asks forty subjects of their impressions. Her work relates a great deal to that of Klein [22, 21], Gilbert [10], and Picard [31].

Scheirer presents a computer system designed to intentionally frustrate the users and gives a physiological method for determining whether a user is frustrated [36]. The key result of this paper was to use a physiological method, as opposed to something akin to a survey, to measure a user's affective responses to a computer system.

CHAPTER 3

AURALIZATION DESIGN

3.1 Layered Program Auralization

In this section, I describe the layered design of the sounds in my program auralizer. I use layers to represent behavior with sound. I chose three different categories of program comprehension to represent: control flow, state, and semantic data [30]. Previous work, like Vickers, is concerned only with control flow [48]. I use music to represent computer code for the purpose of debugging runtime behavior in that code. The layered approach generates a musical score, which can be performed by a music rendering system, like MIDI.

I define a layer as, “A method of generating music, or properties of music, from behavior or data.” A layer is a method because it is a process for creating either music or musical properties. A layer may or may not create music itself, depending on its layer type. Layers may, instead of generating music, generate only properties of music. I interpret the definition of the term *music* broadly, and do not limit it to a particular time period, genre, or compositional style.

Layers are either performing layers, non-performing layers, or layers that perform the functions of both. I call layers that perform both functions heterogeneous layers. Performing layers literally create sound, and take, as input, some behavior or property. The output of a performing layer is sound or music.

Non-performing layers and heterogeneous layers, are similar. To understand these two layer types, layer properties must first be described. A layer property is a characteristic of sound. Simple examples of a layer property would be high, low, sparse, dense, or a chord progression. A heterogeneous layer outputs music and properties. A non-performing layer outputs only properties and does not generate music itself. The concept of a heterogeneous layer is important, as the creation of layer properties allows for layers to map sounds to other layers properties, allowing layers

Layer Type	Input	Output
Performing Layer	Behavior and/or Properties	Music or Sound
Heterogeneous Layer	Behavior and/or Properties	Properties and Sound
Non-Performing Layer	Behavior and/or Properties	Properties

Table 3.1: Different types of layers and their properties.

Responses to two musical properties, respectively requesting a sparse and dense sound
Make the resulting sound sparse
Make the resulting sound dense
Make the resulting sound somewhere between sparse and dense

Table 3.2: Possible Responses to two conflicting properties. Notice that no matter the solution, the original properties are not fulfilled, and a user may not correctly perceive the appropriate behavior through the given mapping.

to cooperate. For example, one layer can map a behavior to a chord progression. Another layer can detect properties and "sing along" in the same key, or add instruments to the musical score. Table 3.1 several layer types and their properties.

Properties of a layer can be either conflicting or non-conflicting. In most cases, properties are non-conflicting. A conflicting property is a single participant in a conflicting relationship between two or more properties, where the output loses information. Properties can only conflict when they attempt to modify musical parameters that would cause another property's request to go unmet.

For example, two properties requiring that a sound be both sparse and dense would be conflicting. If the property *dense* states that it wants the sound produced to be that of a full orchestra, and *sparse* says to create the sound of just a piano, a sparse timbre, then these two properties conflict. If the resulting sound has a sparse texture, then the *dense* property is not fulfilled. Likewise, if the texture is dense, *sparse* is not fulfilled. No matter the resolution between this conflict, there is information loss. Table 3.2 gives possible responses to two conflicting properties. Notice that no matter the solution, the at least one properties request is not fulfilled.

For an example of constructing a layer using layered program auralization, consider a layered auralization approach to copying a file. Suppose that this file had a size of x. For n seconds, 1/x of

Attribute	Example Mapping
File Size	Full orchestra for a “large” file and a single guitar for a “small” file.
Time Required	Voice saying, “You have x minutes remaining” every thirty seconds
Time since start	Slowly raising pitch

Table 3.3: Layered mappings from copying a file to sound.

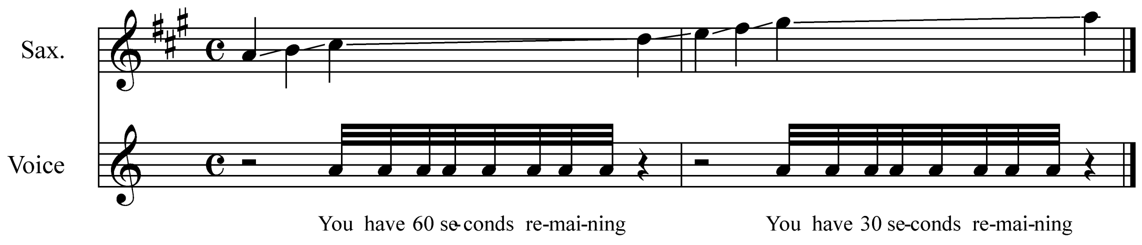

Sax.
Voice
You have 60 se-conds re-mai-ning
You have 30 se-conds re-mai-ning

Figure 3.1: This is an auralization for copying a file. Assume that the quarter notes in this example take up the appropriate amount of time. Since this auralization includes only a saxophone, the file size must be small. To remind the reader, the lines between notes indicate a glissando, or a continuing increase in pitch between notes.

this file is copied from one location to another. The operation of copying a file has several attributes which may or not be important to a user of a system. For example, the following may be important to some users: the size of the file to be copied, the time required to copy the file, and how long the file has been copying for.

Suppose we want to create a layered program auralization design of the attributes of this operation. In layered program auralization, each characteristic is granted its own layer. Table 3.3 contains an example layering from the behavior of copying a file. Figure 3.1 shows a corresponding musical score for this auralization.

Thus, to look at Figure 3.1, we can see that the concepts of timbre, vocals and pitch are used. The timbre of the sounds do not effect the vocals. For example, a very deep voice can say, “You have x minutes remaining,” but so can a “tinny” voice. Similarly, pitch does not effect the words that can be sung. Singers can sing a given text high or low, subject to the limitations of physiology. Thus, each element of this operation can be auralized.

Layered Program Auralization	Parameterized Icons [18]
Layers can communicate	Parameters do not communicate
Uses musical structure to create layers	Uses sound synthesis techniques with parameters
Easily adjustable in real time through addition or subtraction of layers	Certain parameters are easier to adjust than others. Synthesis techniques do not always allow for the adjustment of arbitrary parameters.
Layers have sets of properties	Parameters do not have sets of properties
Layers may represent sound abstractly	Parameters imply a literal representation of sound

Table 3.4: Differences between parameterized icons and layered program auralization.

Layered program auralization is a technique for letting multiple sound mappings play simultaneously and cooperatively. Layers should not be confused with Gaver's parameterized icons [18]. In a parameterized icon, sounds are given parameters that correspond to a property of an operation. For example, if a file is large, that file should sound like it is large. Thus, parameterized icons may give information regarding a parameter of an operation, like size. Gaver does not consider combinations of parameters. Table 3.4 enumerates several key differences between layered program auralization and parameterized icons.

3.2 Communication between layers

Layers can be, potentially, independent sources of aural information, but this is not required. In some cases, it becomes convenient for there to be a communication model for layers. Non-performing layers would have no purpose if they could not communicate their properties to other layers.

Allowing for communication to exist between layers can be useful. A non-performing layer does not generate sound, only properties. For these properties to be rendered musically, they must be communicated to another layer. Layers interpret properties, and then, depending on layer type, may generate sound. Layers may need, however, to allow for two way communication. One

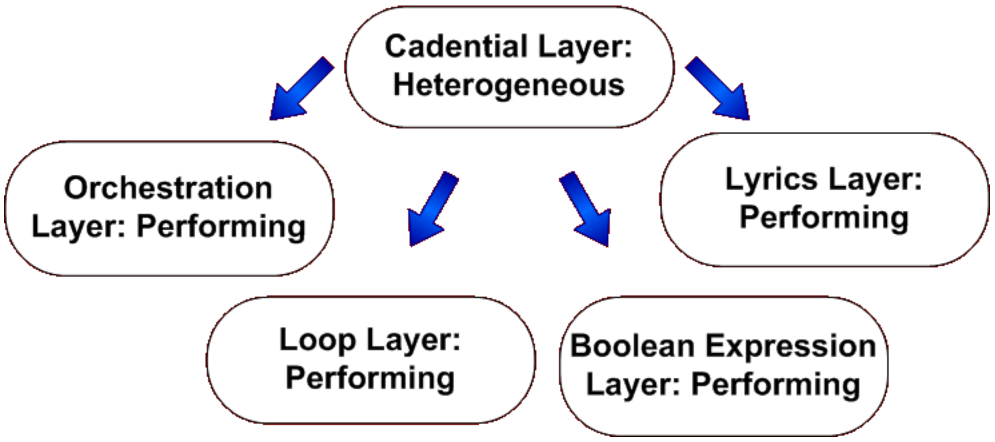

Cadential Layer: Heterogeneous
Orchestration Layer: Performing
Lyrics Layer: Performing
Loop Layer: Performing
Boolean Expression Layer: Performing

Figure 3.2: The communication model for the three layers used in this thesis, Cadential, Orchestral, and Lyrics.

layer may create a set of properties from behavior. These properties are then communicated to another layer, which in turn creates more properties. These second properties may be useful to the first layer, or require a change in the first layer's properties, and thus two way communication is allowed.

The auralization design in this thesis consists of three primary layers. The design is a tree based, top down, model of communication. The root note is a heterogeneous layer. This layer creates music and properties of music. In my system, the root layer, called the cadential layer, generates chord progressions over time. These chord progressions are communicated to lower layers, which read these properties and create sound on those properties. I chose for all lower level layers to be performing layers, they do not create properties.

3.3 When music meets code, the basic layers

To auralize C++ computer code, I designed a system to handle types of program constructs, like looping, conditional statements, semantic data like numbers or text, and a system for memory allocation and deallocation. The key to my system is mapping program constructs to like musical structures by considering their fundamental properties metaphorically. However, metaphorical mappings [17] fail in certain instances, especially in regards to semantic data, like numbers or text.

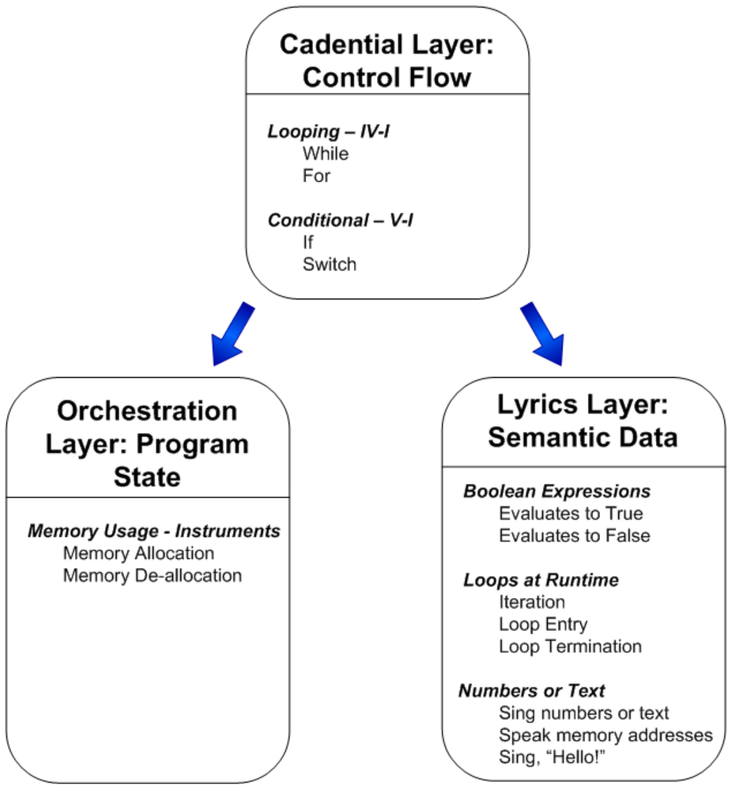


Figure 3.3: Overview of the auralization system

I have an alternate strategy for dealing with this data.

Elements of musical metaphor are powerful, however. The cadence, a chord progression often used at the end of a musical phrase, is a common occurrence in music from a multitude of time periods. The cadence is, in fact, so common, and used in so many works by so many composers, that many common patterns were given special names. Two common examples are plagal, an amen in a hymn, or perfect authentic, which is the traditional ending to a phrase in the classical period of music. Cadences are defined, among other things, by giving a strong pull to the end of a phrase or piece of music. The strong metaphor for *the end* was useful for mapping to program constructs, which also have a well defined beginning and ending.

The three layers in Figure 3.3 are cadential patterns, orchestration, and lyrics. Cadential patterns were used for auralizing control flow aspects of a program. Orchestration was used to represent elements of a program that sustain over time, or state, and so they add the layer of timbre to the existing cadential structure. The timbres add instruments which automatically integrate themselves into the existing cadential structure via one way communication from the cadential layer. Lyrics can be added as another layer to the auralization, and they also receive one way communication from the cadential layer. Thus, in other words, the cadential layer, in this case, serves as a root layer that sends messages to all other layers, allowing them to coordinate or change in real time.

The three layers in Figure 3.3 are not, however, the only layers that were generated during the course of the auralization design. Two other layers, the boolean expression layer, and the loop layer were created. These two layers are performing layers, and can be swapped out with the lyrics layer to represent information without the use of lyrics. The purpose of these two layers is to create a music only description of control flow. For example, the lyrics layer might literally sing the word "true" if an IF statement results in true. If, however, the boolean expression layer is used instead, a music only description of true is used, namely a major third, a common interval in tonal music. Thus, the cadential layer generates chords that represent control flow, but it does not represent the value control flow takes at runtime. If neither the lyrics layer were used, nor the boolean expression layer, the user could identify aurally that an IF statement occurred, but whether that statement was true or false could not be determined. Figure 3.4 shows the auralization system with the lyrics layer removed, and with both the boolean expression layer and the loop layer added.

3.4 Cadential Layer - Control Flow

Cadential patterns were chosen to represent control flow because of their likeness to computer code structures, and their recognizability as part of a musical structure. This metaphorical mapping may give the listener an opportunity to guess what the music they are listening to means. The goals of using metaphor is to reduce training time in a given auralziation system. Any computer statement,

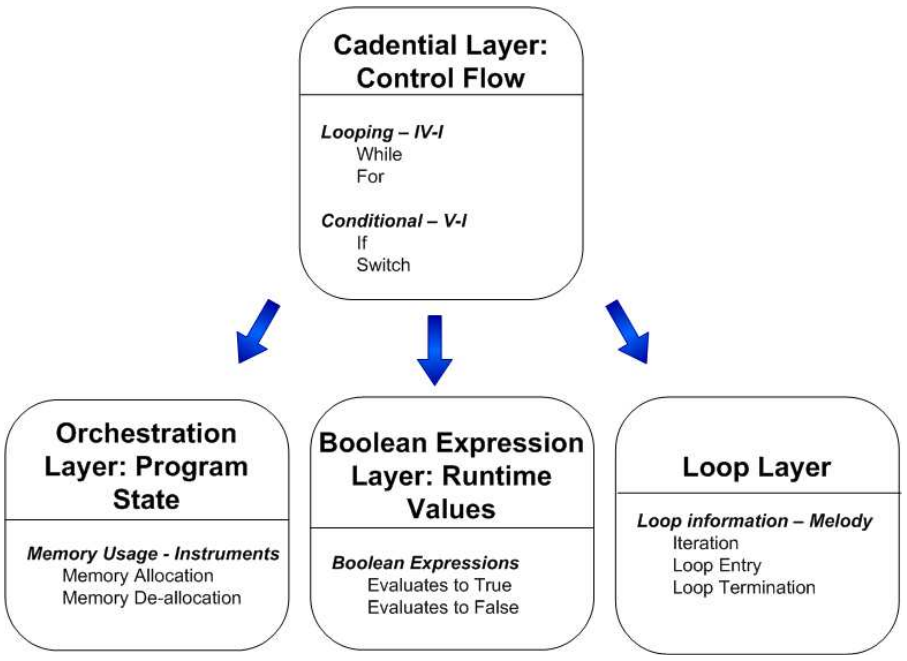


Figure 3.4: Overview of the auralization system, but with the lyrics layer removed, and with both the boolean expression layer and the loop layer added.

syntactically, has a beginning and ending point. Even in unbounded, infinite, loops, there is a starting and ending point for each iteration of the loop. Cadential patterns are similar, in that they have a clear beginning and ending. Since the beginning and ending points are well defined, in terms of voice leading, creating points in between the cadential patterns allowed me to create build ups to the cadences. Suppose I choose a cadential pattern of I-V-I for some program construct. To remind the reader, I-V-I is numeric representation of chords that are key neutral. I could allow repeated instances of that construct to add notes in between the cadential pattern, giving perhaps I-IV-V-I, I-IV-ii-V-I, or I-vi-IV-ii-V-I.

To keep my auralizations short, I chose to use two chord cadential patterns, like V-I, for many of my program constructs. Figure 3.5 shows code, and Figures 3.6 and 3.7 an auralization, for an IF statement in C++. These two figures show the control flow layer with the boolean expression layer. Later, I will build upon this and other examples, and show how adding program state or semantic information does not alter this design, except by the well defined method of one way

```
if(d < 10) {
  //statements to be executed
  //when d is less than 10
}
```

Figure 3.5: An IF statement

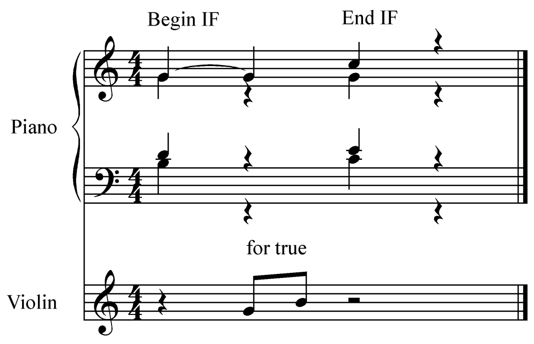


Figure 3.6: Auralization for a true IF statement, using the cadential layer and the boolean expression layer. The chord progressions in the piano are created by the cadential layer, and the violin notes are created by the boolean expression layer.

communication. In addition, future examples will swap out the boolean expression layer with the lyrics layer.

In addition, if multiple IF statements occur one after the other, in a typical IF-ELSE construct, such as the one shown in Figure 3.8, the listener should be able to determine which block of the code is about to be executed. Figure 3.9 gives a second example with several IF-ELSE combinations. This second auralization combines several IF-ELSE blocks into a short set of cadential patterns. The music represented in Figure 3.9 is what would play if, in the running computer program, the value of the variable *d* were, for example, 59.

Looping constructs, WHILE and FOR, are represented with a different set of cadential patterns, in this case plagal cadences, or IV-I. Each iteration of the loop, at runtime, changes the chords. In Figure 3.10, a FOR loop with ten iterations is shown, now with the cadential layer and the loop layer, which plays the violin melody underneath the chords. Figure 3.11 shows its corresponding auralization. The first chord is the beginning of the auralization. Each successive chord indicates

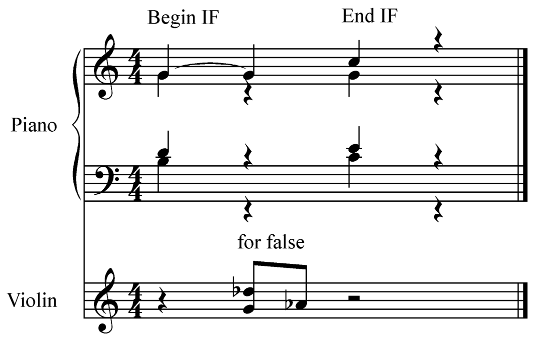


Figure 3.7: Auralization for a false IF statement, using the cadential layer and the boolean expression layer. The chord progressions in the piano are created by the cadential layer, and the violin notes are created by the boolean expression layer.

```
if(d < 10) { //1
   //statements to be executed
   //when d is less than 10
}
else if(d > 10 && d <= 30) { //2
   //when d is greater than 10
   //and less than or equal to 30
}
else if(d > 30 && d < =50) { //3
    //when d is greater than 30
    //and less than or equal to 50
}
else { //4
   //when d is none of the above
} //5
```

Figure 3.8: This is an IF-ELSE combination. The music represented here is what would play if, in the running computer program, the value of the variable d were, for example, 59.

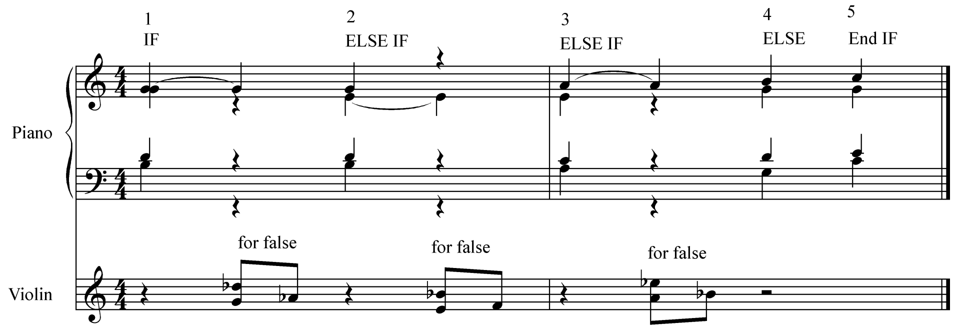


Figure 3.9: Auralization for an IF-ELSE combination

```
for(int i = 0; i < 10; ++i) {
   //this loop would execute
   //10 times
}
```

Figure 3.10: A for loop

another iteration of the loop. The last two chords indicate the end of the FOR loop.

For auralizations to indicate where in the code a particular structure is nested, elements must sound different. I use key changes to indicate nested structures. Each key change uses the same cadential relationships, but shifted. As control flow leaves the nested region, the key changes back to the original, eventually leading to cadences in the original key. Figure 3.12 gives an example of a nested IF statement and Figure 3.13 gives its corresponding auralization.

3.5 Orchestration Layer - Program State

Program state is "the connections between execution of an action and the state of all aspects of the program that are necessarily true at that point in time" [29]. Good examples of program state in a music notation editor would be the number of notes in a score, the name of a score, and each note itself. In my auralizations, I use orchestration to represent dynamic memory allocation in the implementation of a linked list data structure. As I added elements to the list, I added instruments to the auralization, and as I removed elements from the list, I removed instruments from the auralization. These instruments follow the harmonic structure of the existing cadential

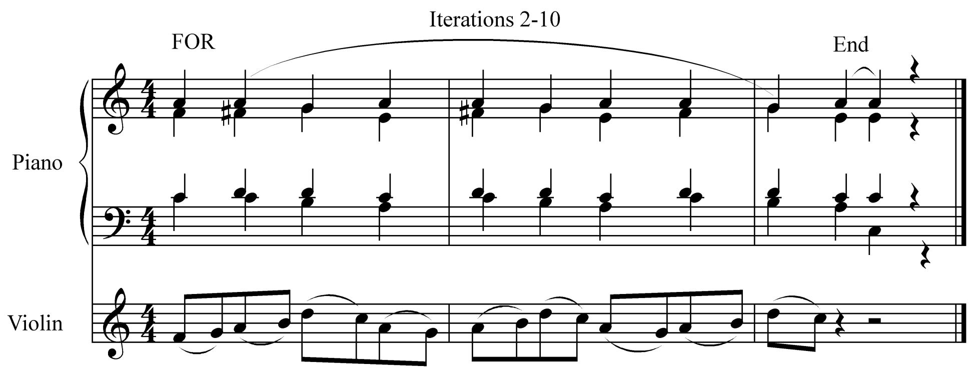


Figure 3.11: This is an auralization for a for loop. The chords in the piano are created by the cadential layer, and the violin melody is created by the loop layer.

```
if(d < 10) { //1
   //executes if d is less than 10
   if(q < 15) { //2
      //executes if d is less than 10
      //and q is less than 15
   }
   else { //3
       //executes if d is less than 10
       //and q is greater than 15
   }//4
}//5
```

Figure 3.12: A nested if statement

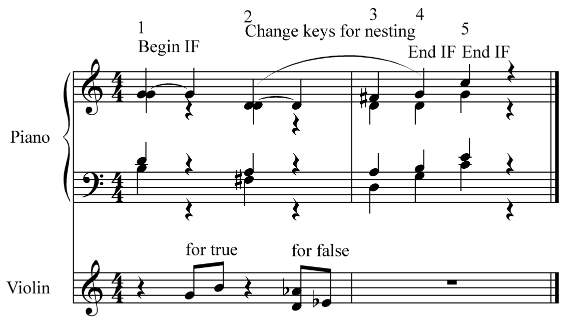


Figure 3.13: This is an auralization for a nested IF statement. In this example, the value of d would be less than 5 and q would be 15 or greater.

```
void LinkedList::clear() {
   Iterator it(this);
   while(it.hasNext()) {
            //Cadential layer

      ListNode* node = it.next();
      Object* ob = node->getObject();
            //Lyrics layer

     delete node;
     delete ob;
            //Orchestration layer

  }
   head = 0;
   numNodes = 0;
            //Lyrics layer
}
```

Figure 3.14: The clear method in a linked list, annotated to identify the layer corresponding to each auralized construct.

patterns.

Figure 3.14 shows the program code for the *clear* method, which removes all the elements from the list, and releases the associated memory. Figure 3.15 shows the auralization of the *clear* method as it deletes four nodes. In this figure, the piano part is created by the cadential layer. The part labeled tenor is created by the lyrics layer, this time with the boolean expression layer and loop layer removed. The last five staves are created by the orchestration layer.

The calls to *it.next()* and *node→getObject()* are not auralized. The *delete* operation is also not auralized directly, but, instead, the orchestration changes from this operation occurring. Note that the orchestration of the passage “thins out” as the memory is deleted. This operation works well when you do not need to know the exact number of objects removed. The thicker the orchestration, the more memory is consumed.

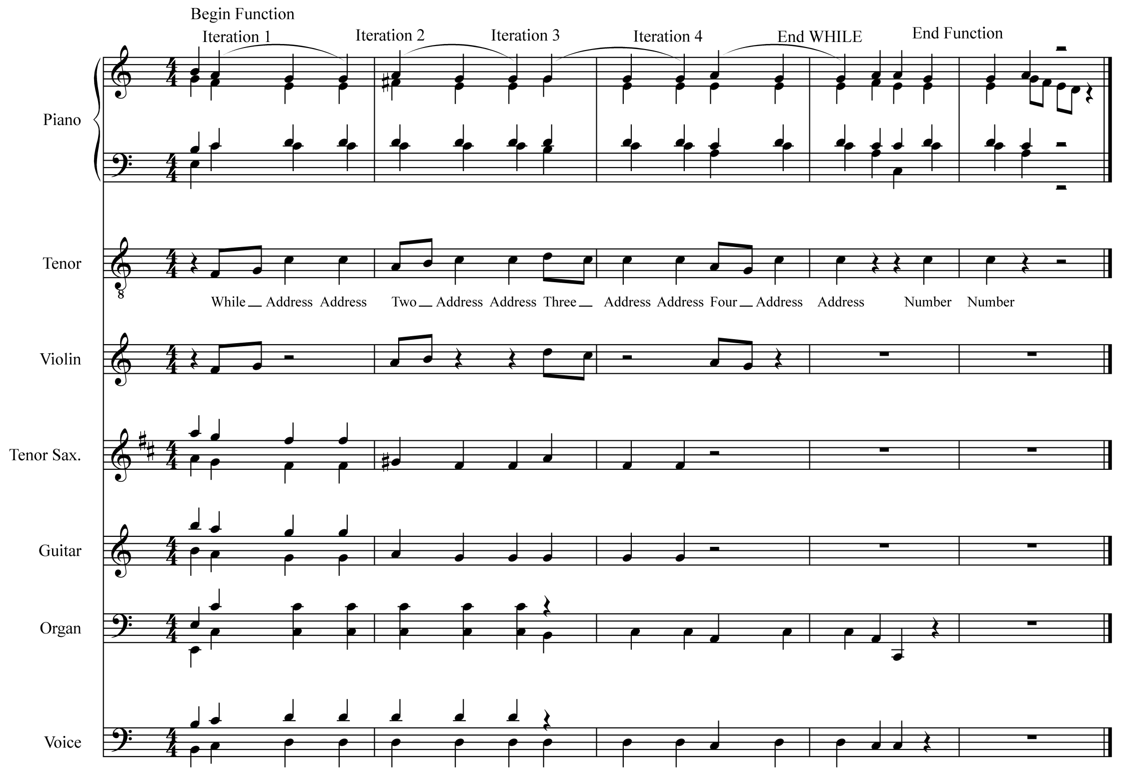


Figure 3.15: This score shows dynamic memory allocation with the orchestration layer and the lyrics layer. The piano part is created by the cadential layer. The part labeled tenor is created by the lyrics layer, this time with the boolean expression layer and loop layer removed. The last five staves are created by the orchestration layer.

3.6 Lyrics Layer - Semantic Data

Previous work in program auralization included only musical information [44]. However, in this approach, it is difficult to represent numbers and text. Further, while musical elements may be created with good reason or a strong sense of logic, they must be memorized to be understood by the listener.

In the previous musical examples, I can see several difficulties in determining the location of a running program in code from cadential sounds alone. The largest difficulty is making the cadential structures sufficiently different, so that an untrained listener can tell them apart. To solve this problem, words can be used to help the listener understand the current location in an executing

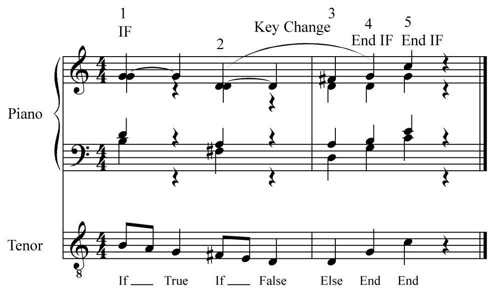


Figure 3.16: Auralization for a nested if statement with the boolean expression layer removed and the lyrics layer added.

program. Figure 3.16 shows the nested IF statement from before, but this time with lyrics.

It is impractical to map numeric data into pitch numbers in the general case, and an obvious intuitive mapping from music into text does not exist. As an example, consider trying to represent the text “Hello, how are you Sally?” To assign auralizations to this text, I could define melodies for each character, melodies for each word, cadential patterns, or other techniques, but as the number of text strings the user tries to remember grows in size, it becomes increasingly difficult to determine an appropriate mapping. In addition, complex text strings that do not correspond to spoken language, like a regular expression, are even more difficult to represent in an obvious way. Since a metaphorical mapping does not exist for this type of information, a different solution was necessary.

Adding a lyrics layer simplifies representing strings and numbers by making their audio representation iconic [17]. Figure 3.17 and 3.18 show an example of program constructs causing the auralizer, using the lyrics layer, to auralize a memory address. With music alone, representing the address is difficult, but using the lyrics layer, the representation becomes non-ambiguous.

More work is required to design a program auralization system that can account for any type of program construct. In addition, communication between layers is an open problem. What is the best communication method for layered program auralization? Do particular communication methods have consequences for design or for the listener? Other open questions include deciding

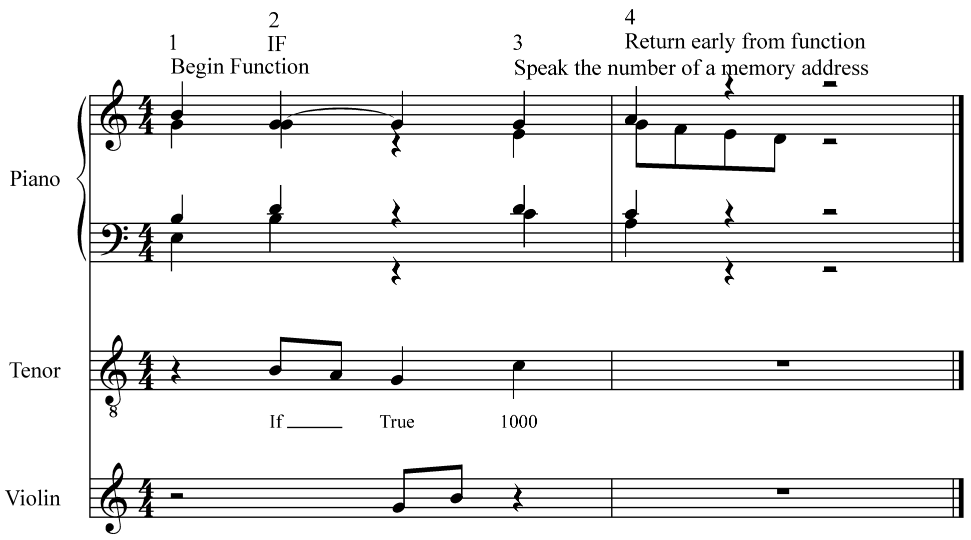
```
void LinkedList::add(ListNode* node) //1
{
   if(NULL == head) //2
   { //Cadential layer
      head = node; //3
            //Lyrics layer

      // ERROR here:
      // should increment numNodes
   }
   else
   {  //Cadential layer

      node->setNext(head);

      ++numNodes;
      head = node;
            //Lyrics layer
   }
} //4
```

Figure 3.17: Program fragment for adding a node to a linked list, annotated to identify the layer corresponding to each auralized construct.



1
Begin Function
2
IF
3
Speak the number of a memory address
4
Return early from function
Piano
Tenor
If
True
1000
Violin

Figure 3.18: Auralization for the faulty add method. Notice that in this auralization, the lyrics layer and the boolean expression layer are added. One benefit of layered program auralization is that layers can combined without effecting the other layers.

how to map these structures to other languages, like Java or Smalltalk. Other layers will need to be created to handle different elements of program state, and work will need to be done to determine the most useful elements of program state for a listener.

CHAPTER 4

EXPERIMENTAL DESIGN

4.1 Introduction

In Chapter 3, I discussed a new technique for mapping sounds to attributes or data, layered program auralization. This system of auralization was used to create auralizations for C++ computer code, for use in debugging computer programs. This section describes a formal empirical study testing the effectiveness of the auralizations.

I conducted this empirical study in the fall of 2005, and used data from this study to determine whether my auralizations are effective tools for debugging computer code and whether my auralizations are effective without training. In the course of this chapter I explain the motivation for this study and the procedures used to help ensure correctness.

4.1.1 Motivation

There is little research that has been conducted into program auralization, and virtually no research that includes formal empirical studies. There are, however, a few exceptions. Rigas and Alty have conducted numerous studies into using audio for representing graphical information [34, 35, 3, 2]. Francioni has done work in using sound for parallel programs [14, 12, 13] and assistive technologies [39, 15, 38]. Francioni helped lay the groundwork for future auralization research, but her experiments were often, at least to a degree, informal. Vickers did the most extensive testing on program auralization [47], but Vicker's only scratched the surface of the empirical testing required to verify that auralization works. Further, Vicker's did not give a systematic approach, as I do with layered program auralization, for creating new auralizations.

Thus, data on whether auralization works for any task at all is limited, and data regarding auralization for debugging is virtually non-existent, with the exception of Vickers. One question this study answers is whether auralization can work without training. It may seem an intuitively obvious hypothesis that the use of auralization would be less effective without training. However, this question is unexplored in the auralization literature.

If auralization researchers find that program auralization is a useful tool for debugging, but requires a large amount of training to use effectively, it may not be adopted. Doing empirical work on training in auralization could be beneficial, as it will give empirical data concerning how much training each auralization takes. Further, not all auralizations are created equal, and it may be the case that some auralizations take less training than others. In a sense, studying this element of auralization could give users of an auralized system a better ”out of the box” experience.

Previous work on debugging with auralization by Vickers used Pascal as the programming language, and did not use layered program auralization. Thus, the current work intends tests various layers I generated using the layered program auralization concept, and to test see if these auralizations increased the number of bugs found by participants. When Vickers did his empirical experiments, he created computer programs with only one error in them, and subjects either found the error or they did not. In my experiment, participants were given much larger computer programs, each with eight faults. There was no empirical reason why eight faults were chosen, other than the number seemed about right for the size of programs.

The experimental goal is to verify a method for repeated testing with various types of auralizations. In this sense, the experiment should lay the groundwork for a way to effectively judge a set of auralizations. This would allow for future experiments to be conducted by swapping auralization types, but not adjusting the experiment or required criteria. I use the goal question metric approach to codifying these goals [5].

Goal

1. To determine whether particular auralizations increase or decrease the ability to find bugs.

2. To determine how effective a particular auralization is with no training, or the effectiveness “out of the box.”

The first goal is important because it gives a baseline for how effective an auralization is with its intended task, which is to help people debug. In this context, and for the purpose of this experiment, the effectiveness of an auralization is defined as:

> The ability of one particular auralization to increase the number of faults found in the context of reading source code for faults.

The second goal is how effective an auralization technique is without training. A likely hypothesis is that music-based auralizations will have lower level of effectiveness without training than speech-based auralizations. In addition, it may be the case, that certain musical, or speech-based auralizations require less training then others of the same type. For example, it might be discovered that jazz music works better as an auralization technique without training than baroque music, or the opposite may be true. Likewise, it may be found that preferred music, music preferred by the listener, has a significant effect on the understandability of auralizations.

This goal is important, as the amount of training required is a significant usability question, and not all auralizations will, necessarily, require the same amount of training. The most fundamental element of an auralization is its effectiveness, but its effectiveness without training is a reasonable secondary condition to consider. In other words, all else being equal, if auralization A has a smaller training time required than auralization B, then auralization A is superior to B. For this reason, the effectiveness of an auralization without training should be analyzed. The questions and metrics used in this experiment are enumerated below:

Question

1. Do the auralizations I created increase the ability to find bugs?

2. How effective was each auralization with no training?

Dependant Variable	Definition
μ_{Nno}	The mean number of faults found without auralizations.
μ_{Nms}	The mean number of faults found using music as an auralization.
μ_{Nsp}	The mean number of faults found using speech as an auralization.

Table 4.1: Variables used in this experiment.

Metric

1. Measure the number of faults found after completing a debugging task with or without an auralization.

2. Measure the effectiveness of each auralization technique with respect to first use, and without training.

4.1.2 Hypothesis and variables selection

Table 4.1 shows several variables used throughout the experiment.

In addition to these variables, I define any μ_{Nmsx}, where $x = \{1, 2, 3\}$ to be an instance of the experiment. Thus, μ_{Nms1} means the first of three runs for testing the number of faults using music based auralization techniques.

Null and alternative Hypotheses:

Music Hypotheses The null and alternative hypotheses involving music.

1. **Null:** $H_{0Mx} : \mu_{nx} = \mu_{mx}$, **Alternative:** $H_{aMx} : \mu_{nx} \neq \mu_{mx}$.

Speech Hypotheses The null and alternative hypotheses involving speech.

1. **Null:** $H_{0Vx} : \mu_{nx} = \mu_{vx}$, **Alternative:** $H_{aA} : \mu_{nx} \neq \mu_{vx}$.

Independent Variables

- Type of Auralization: None, music, or speech

- Debugging tasks

Dependent Variables

- The number of faults found.

4.2 Method

This experiment was conducted using student programmers in their second year. All programs used were toy programs, and as such, may be too specific to generalize into industry. While this study may not generalize to debugging large computer programs, future work may allow for such tests.

4.2.1 Participants

Participants were selected from a sophomore level course on data structures in C++. There were 32 students in the course, 29 of whom chose to participate in the study. Twenty students participated in all three sessions. The number of participants in the first session was greater than the number in the last session, due to mortality. Since comparisons between groups are only made for individual sessions, not between sessions, the change in the number of participants between sessions is not a threat to validity. Since participants in this study were all students, I encouraged participation in the study by allowing students to drop their lowest homework grade for participating in all three sessions of the experiment. This extra credit amounted to about 3% extra credit in the course. If a subject attended only one or two experimental sessions, they received no extra credit.

Subjects were blocked and balanced into experimental groups according to GPA. There were three experimental groups a single subject could be assigned to: the control group with no auralizations, the music group with only musical auralizations, and the music-plus-voice group. These groups were labeled N, M, and V. To place subjects into groups, subjects were ranked and put into subgroups of three subjects. To rank participants, overall GPA and computer science GPA was used. Participants were first ranked by GPA, and in the case of a tie, participants were ranked by computer science GPA.

Group V	**Makeup**	**GPA**	**CS GPA**	**Ear Training Score**
1V	yes	3.7	3.88	7
2V		3.3	4	x
3V	yes	3.2	3.7	6
4V		3	3.5	7
5V	yes	3	3.5	3
6V		2.95	2.15	5
7V		2.8	3.85	x
8V		2.8	3.35	6
9V		2.5	2.475	x
Average:		3.03	3.38	5.67
Std dev:		0.34	0.64	1.51

Table 4.2: Details for the subjects in group V. An x marks a participant that did did not show up to any experimental sessions, which prohibited that participant from taking the ear training exam.

Of the top three students, each individual was randomly put into a group, one in M, one in N, and one in V. Of the next three students, each was, again, randomly selected into a group. Tables 4.2, 4.3, and 4.4 show information about the groups. Groups were blocked by GPA, and not experience with a debugger, as we could not guarantee subjects experience to be accurate.

Of note in these tables is the column labeled makeup. When the experiment was originally run, attendance was lower than anticipated. A second set of three sessions was run with additional subjects. This makeup column indicates whether or not that subject participated in the original experiment or the experimental makeup sessions. A subject is marked with a yes if they were a member of the second set of three sessions. In addition, notice that participant 11M, does not fit with the balancing and blocking scheme. This subject was originally scheduled to participate in group V, but due to scheduling restraints could not attend this session. We moved this subject to group M so that he/she could still participate in the study. Allowing these makeup sessions was better then having next to no subjects, but may have effected the validity of the conclusions, as isomorphism between experimental sessions cannot be guaranteed.

Lastly, subjects were given an ear training test, or a test to determine whether they could tell the difference between basic musical constructs. Since the scores were nearly identical between

Group N	**Makeup**	**GPA**	**CS GPA**	**Ear Training Score**
1N		3.65	3.7	6
2N		3.32	3.5	5
3N		3.2	3.77	7
4N		3	3.75	6
5N		3	3.35	5
6N		3	2.75	7
7N		2.75	3	5
8N		2.5	2.9	6
9N		2.4	3	4
Average:		2.98	3.30	5.67
Std dev:		0.39	0.40	1

Table 4.3: Details for the subjects in group N. An x marks a participant that did did not show up to any experimental sessions, which prohibited that participant from taking the ear training exam.

Group M	**Makeup**	**GPA**	**CS GPA**	**Ear Training Score**
1M	yes	3.5	3.65	7
2M		3.5	3.5	3
3M		3.2	3.2	x
4M	yes	3	3.85	6
5M	yes	3	3.15	6
6M	yes	3	2.7	6
7M		2.9	3.2	6
8M		2.8	3.07	6
9M		2.5	2.75	5
10M		2.25	3.4	5
11M	yes	2.7	2	7
Average:		2.94	3.13	5.7
Std dev:		0.38	0.51	1.16

Table 4.4: Details for the subjects in group M. An x marks a participant that did did not show up to any experimental sessions, which prohibited that participant from taking the ear training exam.

all three groups, I determined that the musical aptitude between groups is at least roughly similar. Vickers [47] showed no correlation between musical knowledge and debugging performance, so this attribute would likely not have had an effect even if there were a significant difference between groups.

4.2.2 Materials and tasks

This was a multiple phase experiment. In each phase, a different test was administered to subjects, and at each phase, various amounts of training were given. There were three phases to the experiment. In the first phase, groups were administered an ear training test to test their ability to perceive music. The averages for all three groups were similar. The purpose of this test was to detect any significant differences between groups in regards to musical perception. No significant differences were detected, and thus these results were not used for any further purpose. See Appendix B for the details of this test. In the second and third tests, groups M and V were given training in the auralizations. Chmeil showed that giving users training in debugging tasks increased their ability to debug [9], and thus no debugging training was given to any group. Figure 4.1 shows the phases of the experiment.

The following is outline of the experiment.

1. Administer pre-qualification survey.
2. Break subjects into three blocked and balanced groups.
    (a) Control Group, no auralization provided, source code only.
    (b) Use of musical sound as an auralization in addition to source code.
    (c) Use of speech plus music as an auralization in addition to source code.
3. Run three tests. Each group is given the same source code example, with or without auralizations.

4. In the first test, no training was given in how to use the auralizations. In the second and third sessions, the non-control group received training into how to use the auralizations, but not training in debugging.

   (a) Test 1: Debugging a roulette game with or without auralizations.

   (b) Test 2: Debugging a bank program with or without auralizations.

   (c) Test 3: Debugging an address book with or without auralizations.

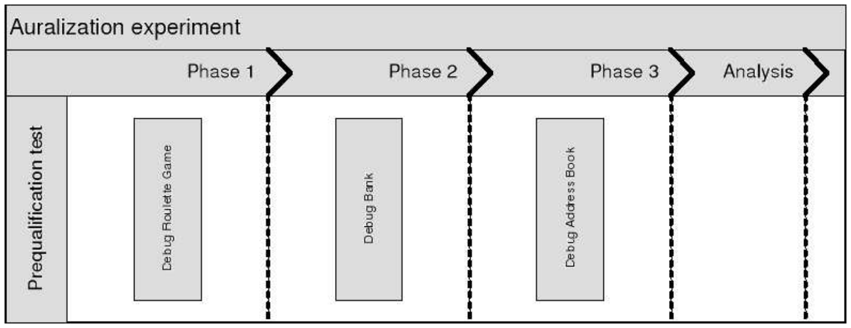


Figure 4.1: Experiment overview

In this section, all debugging tasks used in the experiment are discussed. For a complete listing of the code, or associated faults, for any of this software, see Appendix D.There are three debugging tasks that were used: a game of roulette, a bank automatic teller machine, and an address book. An attempt was made to make these tasks as similar as possible. Each task contained eight faults, faults were similar in nature, and programs were of approximately the same length. The last program, the address book, was slightly longer than the other two programs, and thus in this test, participants were informed that all faults were in either the AddressBook.cpp, AddressBoook.h, LinkedList.cpp, or LinkedList.h file. This narrowed the region participants had to look for faults down to a region of code similar to the other two experiments.

In addition to this, auralizations for all three debugging tasks were as similar as possible. All three sets of auralizations, both musical and musical with lyrics, had eight tracks. Participants were given a compact disc with all eight auralizations on them when they were asked to debug a program. Not all tracks illuminated a fault. Participants then used Windows Media Player to listen to the tracks while debugging the program.

For each track during a debugging task, an input to the program was given. For example, an input for the address book might be “5.” This would mean that audio for that track starts at the beginning, or first line of, the printAll function in AddressBook.cpp. Essentially, this tells the participant the state of the running computer program at the time the auralization begins. For a complete list of code and tracks see Appendix C. This approach is analogous to setting a breakpoint in a traditional program, except that I set the breakpoints for the participants and they listen to what sounds would be created at that breakpoint. This information was not given to the control group, which is a threat to validity.

In the roulette game test, users were asked to debug a command line version of the game of roulette. This version of roulette had only two methods of betting, color and number, and thus is a simplified version of the American casino game. In the color game, the user bets on either black or red. A ball is then spun on a roulette table. This table has sockets where the ball can land after it stops spinning. The ball will land on either a numbered red space, black space, or green space.

If the player bets on a color, and the ball lands on what the player chose, that person wins twice the amount they bet. If the ball lands on either green or the opposite color of what the player chose, they lose their bet. Having green slots in the table is, of course, how the house always wins, over time. If the ball lands on a green spot, the player always loses. The other type of bet is on a number. Each slot on the roulette table is numbered, green slots being 0 and 00. If the user guesses the correct number, that person wins 35 times the amount they bet. Like before, if the ball lands on 0 or 00, the player loses.

In this command line version of roulette, the user is given $1000 dollars to begin playing the game. The player is then asked how much they want to bet, and they then place their corresponding bet on a color or number. The wheel is virtually spun, and the user either wins or loses. The money is then tabulated, and the player is asked whether he/she would like to play another game.

The bank debugging task is a model of an automatic teller machine. The command line interface to this program has a menu, which has the following options:

1. Login

2. Deposit

3. Withdraw

4. Transfer

5. Check funds

6. Logout

Essentially, these functions simulate the behavior of a simple bank automatic teller machine. Since this is a bank system, the most obvious of security protocols should be implemented, namely that you must log in before you can withdraw, transfer, or deposit money.

The address book program is also a menu driven command line program for adding friends to a virtual address book. This third program was chosen, especially, because it includes the implementation for a linked list. Linked structures can have several obvious faults like memory leaks, garbage pointers, nodes not being properly instantiated, and others. This program had several common address book like functions, which are enumerated as follows:

1. Add a friend

2. Delete a friend

3. Edit information for a friend

4. Search for a friend

5. List all friends

6. Delete all friends

Fault insertion process

A list of the eleven most common faults in novice computer programs was created by Spohrer and Solloway [42]. These eleven common faults were categorized into what Spohrer and Solloway call ”bug types” . Not all eleven bug types were used. Adding too many faults to a small program may make the faults obvious, or may saturate the program with faults. To ensure there were not too few or too many faults, a pilot test was run, and no ceiling or floor effect was discovered from that data, meaning the number of faults in the program was about right. The following ”bug types” [42] were identified as the six most common in novice programs:

- Off-by-one bug: A bug involving boundary conditions in the program.
- Output fragmentation bug: This type of bug occurs when the program prints out an uninitialized variable.
- Or-for-and bug: A bug when an OR statement is mistakingly used for an AND, or vice versa.
- Incorrect Constant bug: A bug that occurs when a constant in a program has the wrong value. An example would be accidentally typing SOME_CONSTANT = 9999 instead of, perhaps, 99999 [42]. This occurs most often when a value is dropped at the end of a number with repeating digits.
- Incorrect formula bug: This occurs when, literally, an incorrect formula or calculation is used.
- Missing parentheses bug: Occurs when parentheses are left out of a calculation. This type of bug can be avoided with a thorough understanding of operator precedence, or likewise by the use of parentheses.

For each debugging task, computer programs were created, modeling a game of roulette, a bank, and an address book. Working versions of these programs were created, before adding in faults. While I can certainly make no guarantee the programs were fault free, the programs went through an extensive testing process. To test the programs I used simple unit testing techniques, conducted several code reading sessions, and performed manual testing on the computer. After testing, faults from the above categories, and other similar faults were inserted into the code in appropriate places. Since Spohrer and Solloway suggest these faults are the most common in novice programmers, and my participants for my experiment are novices, it is reasonable to assume there will not be a ceiling effect in the study. However, a pilot test was run to be sure, and no ceiling or floor effects were found. The goal was to add faults that are sufficiently difficult to find.

Pilot study

In order to test the initial correctness of the programs and auralizations further, I conducted a small pilot study. In this study, I tracked the number of faults found for each group and found no ceiling or floor effect. In addition, during this process, several errors and problems in the auralizations, the code, and the study handbook given to participants were found and corrected before implementation of the study itself.

Most importantly, running the initial pilot study helped determine initial problems with the auralizations in terms of perception. For example, the largest problem I corrected before implementation was in a set of auralizations that use only music, no voice. I discovered that participants had difficulty determining the difference between true and false on conditional statements. Before the pilot, this was done using major and minor chords. This was helpful feedback, as it made it clear that I needed to change the true and false in conditional statements to more aurally different constructs. In the end, I chose a major chord, in key, for true, and for false I chose a I^{b5b9} chord, a chord that is strikingly different from a major chord.

4.3 Procedure

The debugging experiment was designed to test the effectiveness, in this case the number of bugs found, of the auralizations for debugging computer programs. Several hypotheses were tested, including whether the users in groups M, the music group, and V, the music and voice group, found more errors than group N, the control group, without auralizations. This was done in three separate experimental sessions, the first of which subjects were given no training into how to use the auralizations.

4.3.1 Implementation issues

In this first session, the no-show rate in the experiment was 70% in the V group, 40% in the M group, and 0% in the N group. I attempted to minimize the high no show rate in groups M and V by holding makeup sessions for those that did not show up. Two makeup sessions were held, at different times, for group V, and one makeup session was held for group M. The number of participants, if you include those that made up the sessions, was five in group V, six in group M, and nine in group N, at this point. Unfortunately, this introduces a threat to validity, as it is not possible to make sessions 100% isomorphic. However, running makeup sessions, with the same packet, same code to debug, same procedure, same training, but at a different time, is certainly better than having no participants at all.

In the second session, participants in groups M and V were given a 25 minute training session into use of the auralizations. Since it has been shown that giving participants training into debugging techniques does impact how effective they are at debugging, no training into debugging techniques was given to any group [9]. The training examples were chosen carefully, as to not highlight bugs, but instead to highlight what sounds different program constructs made. For example, participants learn what an IF statement sounds like in the auralizations, but do not learn how to use that information to debug an IF statement that is exhibiting incorrect behavior. Thus, groups M and V were trained in the auralizations, not debugging, giving them no unfair advantage over group N.

No-show rates were similarly bad in the second session. Since only participants that went to the first session, or did a makeup, could attend the second session, no-show rates were worse than in the first session. In group V, two people attended the study, and only one person attended a makeup session. In group M, similarly, only four people attended, and no students attended a makeup session. For session three, no-show rates were similar to previous weeks. Most participants, except for one, that attended in week two, also attended in week three.

4.3.2 Makeup sessions

Overall, no-show rates were so bad, that a second complete set of sessions was completed to increase the overall experimental validity. To do this, a second round of all three sessions of the experiment was run with only participants that missed the initial session. Participants were kept in their original groups, but asked, again, to participate. Many more participants attended the makeup sessions. While the makeup sessions were at different times and days, great care was taken to ensure the sessions were as close as possible to the previous ones.

4.4 Results

In this section we enumerate the data and results from the experimental study created to test the effectiveness of the auralizations. Groups were blocked and balanced, originally, but because of the the no shows, groups ended up with an uneven number of participants. Statistics were run on each category separately, using one way ANOVA tests.

To test my hypotheses, an analysis of variance test was used [28, 20]. ANOVA assumes a normal distribution, though even without a normal distribution the test is considered robust [28]. To test for normality, a Shapiro-Wilk test was used [37, 11]. Shapiro-Wilk is typically used for sample sizes up to 50 [37]. Table 4.5 gives the results of the Shapiro-Wilk test. The null hypothesis of the Shapiro-Wilk test is that the data is normally distributed. Notice that all data but one fails to reject the null hypothesis. However, the mean number of bugs found in the session that for the

Session and group	**Shapiro-Wilk Statistic**	**df**	**Sig.**
Session 1V	0.876137791	6	0.251786
Session 1N	0.976489567	6	0.932837
Session 1M	0.821615562	6	0.091135
Session 2V	0.681970284	6	0.003969
Session 2N	0.826904886	6	0.101171
Session 2M	0.915458969	6	0.473271
Session 3V	0.889821884	6	0.317256
Session 3N	0.912375133	6	0.452205
Session 3M	0.91290618	6	0.45579

Table 4.5: Shapiro-Wilk test for normality on all for all sessions. Notice that all sessions but one cannot reject the null hypothesis, and thus it is fair to assume the distributions are normal.

Shapiro-Wilk the null hypothesis can be rejected is so close to the mean of the control group, that running a test for this comparison is inconsequential anyway.

4.4.1 Session 1

In the first session, participants were given no training in how to use the auralizations. In turn, this lack of training had a clear effect on the results for this section. The two results of note are that group V actually performed worse than group N, and group M seemed to perform only marginally better. In order to give further insight into the effectiveness, or potential lack thereof, of the auralizations bugs are tracked by number. This way, I can analyze the frequency of which individual bugs were found, and compare them to the groups. Tables 4.6, 4.7, and 4.8 show the data for the first session of the experiment.

4.4.2 Session 2

In this session, group M clearly outperformed the control group, $p-value = 0.003$, indicating that M group significantly found more errors than N group. Group V, however, did not reach statistical significance. Tables 4.9, 4.10, and 4.11 show the data for the second session of the experiment.

Group V	**Bugs found**	**Bug 1**	**Bug 2**	**Bug 3**	**Bug 4**	**Bug 5**	**Bug 6**	**Bug 7**	**Bug 8**
1V	0	0	0	0	0	0	0	0	0
3V	2	1	1	0	0	0	0	0	0
4V	3	0	1	1	1	0	0	0	0
5V	0	0	0	0	0	0	0	0	0
6V	3	1	0	0	0	1	0	0	1
8V	4	1	1	1	0	0	1	0	0
Average:	2	0.5	0.5	0.5	0.25	0.25	0.25	0	0.25
Std dev:	1.67	0.55	0.55	0.52	0.41	0.41	0.41	0	0.41

Table 4.6: Session 1 results for group V. Participants that did not show up to this session are not included in this table.

Group N	**Bugs found**	**Bug 1**	**Bug 2**	**Bug 3**	**Bug 4**	**Bug 5**	**Bug 6**	**Bug 7**	**Bug 8**
1N	5	0	0	1	1	1	0	1	1
2N	1	1	0	0	0	0	0	0	0
3N	4	1	0	1	1	0	0	0	1
4N	4	0	1	1	1	0	1	0	0
5N	3	1	0	1	1	0	0	0	0
6N	7	1	0	1	1	1	1	1	1
7N	5	1	0	1	1	0	1	0	1
8N	6	1	0	1	1	1	0	1	1
9N	4	1	0	1	1	0	0	0	1
Average:	4.33	0.78	0.11	0.89	0.89	0.33	0.33	0.33	0.67
Std dev:	1.73	0.44	0.33	0.33	0.33	0.5	0.5	0.5	0.5

Table 4.7: Session 1 results for group N. All participants attended this session.

Group M	Bugs found	Bug 1	Bug 2	Bug 3	Bug 4	Bug 5	Bug 6	Bug 7	Bug 8
1M	5	0	1	1	1	0	0	1	1
2M	5	1	1	0	0	1	0	1	1
4M	4	1	0	0	1	0	0	1	1
5M	4	1	1	0	1	1	0	0	0
6M	5	0	1	1	0	1	0	1	1
7M	3	0	1	0	0	0	1	0	1
8M	3	0	0	1	1	0	0	0	1
9M	7	1	1	1		1	1	1	1
10M	5	1	1	1	1	0	1	0	0
11M	7	1	1	1	1	1	1	1	0
Average:	4.78	0.67	0.78	0.56	0.63	0.56	0.44	0.56	0.67
Std dev:	1.40	0.52	0.42	0.52	0.5	0.53	0.52	0.52	0.48

Table 4.8: Session 1 results for group M. Participants that did not show up to this session are not included in this table.

Group V	Bugs found	Bug 1	Bug 2	Bug 3	Bug 4	Bug 5	Bug 6	Bug 7	Bug 8
1V	4	1	1	1	0	0	1	0	0
3V	3	0	0	1	0	1	0	0	1
4V	4	1	1	1	0	0	0	0	1
5V	3	1	1	1	0	0	0	0	0
6V	3	0	0	1	0	0	1	0	1
8V	8	1	1	1	1	1	1	1	1
Average:	4.17	0.75	0.75	1	0.25	0.25	0.5	0.25	0.75
Std dev:	1.94	0.52	0.52	0	0.41	0.52	0.55	0.41	0.52

Table 4.9: Session 2 results for group V. Participants that did not show up to this session are not included in this table.

Group N	**Bugs found**	**Bug 1**	**Bug 2**	**Bug 3**	**Bug 4**	**Bug 5**	**Bug 6**	**Bug 7**	**Bug 8**
1N	5	0	1	1	1	0	1	0	1
2N	3	0	0	1	0	0	1	0	1
3N	3	0	1	1	0	0	1	0	0
4N	5	1	0	1	1	0	1	0	1
6N	2	0	0	1	0	0	1	0	0
7N	3	0	0	1	0	0	1	0	1
8N	6	0	1	1	1	0	1	1	1
9N	3	0	0	1	1	0	1	0	0
Average:	3.75	0.13	0.38	1	0.5	0	1	0.13	0.63
Std dev:	1.39	0.35	0.52	0	0.53	0	0	0.35	0.52

Table 4.10: Session 2 results for group N. Participants that did not show up to this session are not included in this table.

Group M	**Bugs found**	**Bug 1**	**Bug 2**	**Bug 3**	**Bug 4**	**Bug 5**	**Bug 6**	**Bug 7**	**Bug 8**
1M	7	1	1	1	1	0	1	1	1
2M	6	1	1	1	1	0	1	0	1
4M	4	1	1	1	0	0	0	0	1
5M	6	1	1	1	1	0	1	0	1
6M	6	0	1	1	1	1	1	0	1
7M	5	1	1	1	1	0	0	0	1
9M	8	1	1	1	1	1	1	1	1
10M	5	1	1	1	1	0	1	0	0
11M	8	1	1	1	1	1	1	1	1
Average:	6.11	0.88	1	1	0.88	0.38	0.75	0.25	0.88
Std dev:	1.36	0.33	0	0	0.33	0.5	0.44	0.5	0.33

Table 4.11: Session 2 results for group M. Participants that did not show up to this session are not included in this table.

Group V	**Bugs found**	**Bug 1**	**Bug 2**	**Bug 3**	**Bug 4**	**Bug 5**	**Bug 6**	**Bug 7**	**Bug 8**
1V	6	1	0	1	1	1	1	1	0
3V	1	0	0	1	0	0	0	0	0
4V	2	1	0	0	1	0	0	0	0
5V	0	0	0	0	0	0	0	0	0
6V	1	0	0	0	1	0	0	0	0
8V	3	1	0	1	1	0	0	0	0
Average:	2.17	0.5	0	0.25	0.75	0	0	0	0
Std dev:	2.12	0.55	0	0.55	0.52	0.41	0.41	0.41	0

Table 4.12: Session 3 results for group V. Participants that did not show up to this session are not included in this table.

Group N	**Bugs found**	**Bug 1**	**Bug 2**	**Bug 3**	**Bug 4**	**Bug 5**	**Bug 6**	**Bug 7**	**Bug 8**
1N	5	1	0	1	1	1	0	1	0
2N	0	0	0	0	0	0	0	0	0
3N	3	0	0	1	1	0	0	0	0
4N	3	1	0	0	1	0	0	1	0
6N	2	1	1	0	0	0	0	0	0
7N	0	0	0	0	0	0	0	0	0
8N	0	0	0	0	0	0	0	0	0
Average:	1.86	0.43	0.14	0.29	0.43	0.14	0	0.29	0
Std dev:	1.95	0.53	0.38	0.49	0.53	0.38	0	0.49	0

Table 4.13: Session 3 results for group N. Participants that did not show up to this session are not included in this table.

4.4.3 Session 3

In the third session, unfortunately, the number of of participants in group M dropped again, leaving only 7 participants. In this session, again, group V did not reach statistical significance, and group M, while it looked as if they performed better than N, had a $p-value = 0.16$. Thus, no significant effects were found. However, it is unclear if significance would have been reached had more participants been available for this session. Tables 4.12, 4.14, and 4.13 show the experimental data for session three, and Table 4.15 shows the results of the ANOVA test.

Using a one factor ANOVA test for each hypothesis we found that in session 1, V group did statistically significantly worse than N group, and that in session 2 M group did statistically significantly better than N group. Session three yielded no significant results, although M group was

Group M	**Bugs found**	**Bug 1**	**Bug 2**	**Bug 3**	**Bug 4**	**Bug 5**	**Bug 6**	**Bug 7**	**Bug 8**
1M	4	1	0	1	0	1	0	1	0
2M	5	1	0	1	1	1	0	1	0
4M	2	1	0	0	0	0	0	1	0
5M	4	1	0	1	0	1	0	1	0
9M	2	1	0	0	0	0	0	1	0
10M	1	1	0	0	0	0	0	0	0
11M	5	1	0	1	1	1	0	1	0
Average:	3.29	1	0	0.5	0.33	0.5	0	0.83	0
Std dev:	1.60	0	0	0.53	0.49	0.53	0	0.38	0

Table 4.14: Session 3 results for group M. Participants that did not show up to this session are not included in this table.

Null Hypothesis	Session	F-value	P-value	F-crit
$H_{0M1} : \mu_{n1} = \mu_{m1}$	Roulette	0.421558704	0.524836218	4.451321691
$H_{0M2} : \mu_{n2} = \mu_{m2}$	Bank	12.47553816	0.003018224	4.543077123
$H_{0M3} : \mu_{n3} = \mu_{m3}$	Address Book	2.23880597	0.16041536	4.747225336
$H_{0V1} : \mu_{n1} = \mu_{v1}$	Roulette	6.705263158	0.022450692	4.667192714
$H_{0V2} : \mu_{n2} = \mu_{v2}$	Bank	0.220913108	0.646771682	4.747225336
$H_{0V3} : \mu_{n3} = \mu_{v3}$	Address Book	0.074517978	0.789925598	4.844335669

Table 4.15: Experimental results for each hypothesis, using a single factor anova.

not terribly far off from statistical significance, and the number of participants in this session was lower than desired.

4.5 Analysis

In the first sessions, participants in the V and M group expressed, nearly universally, confusion over the task they were attempting to accomplish with the auralizations. Several students in V group wrote comments on their debugging forms indicating how they were interpreting the auralizations. For example, by far the most common writing in V group was “4123456789”. This was usually written near where the wheel is spun in the roulette wheel code. No one in M or N group made this marking. In the first test, the roulette game, participants heard a spinning roulette wheel, which is a FOR loop that displays characters on the command line.

The numbers participants wrote on the computer code shed light on the interpretation of the auralizations in two ways. The first is that participants interpreted the first word used in a FOR loop, which is “for”, to mean the number four. The first word in the auralization was intended to give the program construct, or the beginning of the FOR loop, and was not intended to mean the number four. This may have been confusing, and this problem would not have occurred had the loop been a while. Future auralizations of the FOR and WHILE program constructs may be switched to say ”loop”, as the meaning of the word loop is unambiguous. The second important aspect is that participants were focusing on what the auralizations were telling them, the fact that the computer was audibly counting. This was done instead of focusing on whether a bug could possibly exist in a for loop that only prints characters to the command line! In other words, since the music was likely foreign, at least without training, participants may have latched onto the lyrics, in an attempt to understand them. This occurred to the detriment of debugging.

Interestingly, the reaction to the auralizations was mixed in the second session. All three students in V group indicated verbally that they understood what the auralizations were telling them. In group M, however, participants gave little aural feedback about what they were doing in the training session. During the actual debugging work, however, one subject became particularly frustrated, stood up, walked to the proctor of the session and said that the auralizations were, “... stupid and not helping me.” While this does not indicate that the auralizations were or were not helping, it certainly indicates a negative reaction to the auralizations for that subject. Interestingly, this session achieved statistical significance in the positive direction, indicating the auralizations were, indeed, helping at least some participants.

While it is unclear, for certain, why group M did not, again, reach statistical significance in session three, there are several possibilities. It may be the case that certain debugging tasks require different techniques. It may also be the case that the auralizations, inadvertently, tended to favor finding one type of error over another. In addition, in task three, performance amongst all groups was worse, perhaps indicating that the debugging task itself was harder. It might be enlightening, in future work, to give participants debugging tasks that are ranked in difficulty, then see how the auralizations fare in regard to their level of difficulty. In other words, not reaching statistical significance in this task points out several important pieces of information that are still needed to increase the effectiveness of the auralizations.

In interviews with group V after the third session, most participants claimed that the voice on the auralizations was distracting. In addition, on participants debugging forms there were several common traits amongst all of V group. By far, similar to session 1, the most common was for their to be numbers written on the code. These numbers were clearly taken from the auralization compact disc, which gives semantic information about memory addresses, variable values, or other data. V group may have been focusing too closely on the low level details, and not focusing on other issues, like control flow. Another interesting experiment, based on this result, would be to remove any numbers indicating semantic information about memory addresses, and other such items, and keep the lyrics only for control flow.

However, there are a number of reasons why group V could have outperformed M group. It could be the case that the human voice is invariably distracting when debugging code. However, there are numerous factors with the voice, quality of performance, gender of singer, melody choice, word choice for program constructs, enunciation, and even choice of the singer. Further testing should attempt to incorporate these other possible factors, to discover the most important elements of the vocal performance or auralization design. Another possibility is that music with lyrics is only helpful for certain classes of debugging problems. It might be the case that for control flow lyrics are helpful, but for debugging linked structures, lyrics are distracting. Another possibility is that V groups required a different type of training than M group. Auralizations with vocals might require learning how to appropriately listen, else they are not helpful. For now, the best way to incorporate the voice is unclear.

4.6 Threats to Validity [49]

There were several problems effecting the validity in this experiment. The categories chosen to describe the threats to validity are described by Wohlin [49]. These categories are conclusion validity, construct validity, internal validity, and external validity.

4.6.1 Conclusion Validity

One problem I faced was a very high no-show rate between sessions and between groups. For example, in the first experimental sessions, group V had three members show up, group M had five members, and group N had nine members show up. This puts V group at a 70% no-show rate, group M at a 40% no-show rate, and group N at a 0% no-show rate. This no-show rate occurred despite carefully scheduling the experimental times so that no subject had a time conflict and giving extra credit to the students that participated. It is unclear whether these makeup sessions had an effect on the outcome.

The great disparity between groups could cause statistical problems in the analysis, and thus we decided to run extra experimental makeup sessions to give those that missed an opportunity to participate. These makeup sessions did not equalize the experimental group sizes, but they did make them closer in size. While makeup sessions were identical in content to the original sessions, they were done at different times, and some students did these makeup sessions alone, instead of in the original group.

In addition to the above threats, the room used for the experiment occasionally had neighbors making noise. There was no way to predict when other people from the university would be near the lab during testing time. Overall, however, neighbors were quite and unobtrusive, but since this is a sound based experiment, it is a threat. Subjects would wear headphones and make judgments about the sounds they were hearing in relation to the code. Any outside sound may have effected the results. While this is a threat, it is unlikely, because the typical amount of sound from neighbors was extremely small, for example, an individual walking by the laboratory.

Lastly, the control group in this experiment did not receive the same inputs to the computer programs that the two other groups received. Group M and V received inputs to each track, that would cause the sound output to occur. When designing the experiment, it seemed to me that the control group would only be confused by the extra inputs, since it related to only the sound output, and this sequence of inputs did not necessarily illuminate a bug. It is unclear whether this omission from the control group had an effect on the results, and it is an additional threat to validity.

4.6.2 Construct Validity

The interaction of testing and treatment and hypothesis guessing [49] may have had an effect on the construct validity of this experiment. Participants were aware that they were involved in a debugging study and probably inferred that the number of bugs was a measure, since it is the most obvious way to grade a study such as this. Subject's may also have "decided" that a musical treatment was or was not effective. Subject 8M, for instance, indicated in the second session, to the proctor, that his treatment was, "stupid, and not useful." This subject may have adjusted his/her behavior to accommodate his/her opinion, or what he/she thought was being tested about that opinion. However, other subjects may have felt similarly to subject 8M's response, but were not verbal about their opinions. Thus, there is no way to know whether this was a typical or atypical response.

Experimenter expectancies may also have been a threat to this experiment. The proctor of this experiment was also the designer of the auralization system. This was necessary as subjects could not have received training from another proctor, other than the designer, without giving that individual significant training into how the auralizations were designed and created. However, in defense, I had virtually no way of predicting the outcome of the experiment beforehand, as there is little empirical precedent for such experiments on program auralization. Thus, my own expectations were, at best, limited, and I essentially had no idea what the results would show.

4.6.3 Internal Validity

Since the no-show rate was so high, makeup sessions were held to increase participation in the study. An attempt was made to have the amount of time between each experimental session be exactly one week, although, for logistical reasons with subjects schedules, this was not always possible. Notably, in the second set of three makeup sessions, the first session was conducted only two days before the second session. While this situation was unfortunate, subject's schedules prevented a different approach. The second and third, of the makeup sessions were conducted exactly one week apart, like the previous sessions.

Maturation effects may also have been an issue. In the second session, one subject, subject 6V, reported having "Slept throughout the session." This subject, it turns out, was not literally sleeping, but that subject may have been taking a mental hiatus. This may indicate this subject was tired during that particular session, or that he/she did not honestly participate. Interestingly, this subject had the worst score in the second session, although the score was not significantly worse than other participants. It is unclear whether other subjects took similar mental breaks, but were not vocal about it.

In addition to no-show rates, mortality was also an issue in this experiment. While the mortality rate was relatively low, a few subjects dropped out of the experiment after attending only one, or two, sessions. There was no obvious pattern to the mortality rates, and thus this is likely not a significant threat to validity. For example, it was not the case that all of the high GPA or low GPA students dropped from the experiment. The dropouts seemed to be unrelated to a particular factor.

4.6.4 External Validity

The participants used in this experiment were taken from a second year course in data structures. My results may hold only for beginning programmers, and they may not scale to industrial practice. In addition, the programs the participants debugged were toy programs, and were only several hundred lines long. The results may not apply to large programs.

In industrial practice, debugging is often done in a debugger. In this experiment subjects were asked to listen to compact discs while debugging source code. This is highly unusual, and inconsistent with industrial practice. However, because creating a sonified debugger is a significant task, which may take several years to complete, it was a necessary intermediary step for mitigating risk into this line of research. In other words, this step was taken to determine whether using sound in a debugger could have a positive effect at all. Thus, while this is certainly different from industrial practice, a positive correlation between listening to auralizations and debugging could indicate that creating a sonified debugger is worth the effort. The current study's use of a compact disc, if anything, likely made it more difficult for participants to debug. With a compact disc, subjects cannot visually see what elements of code are being executed, like can be done in a debugger.

CHAPTER 5

CONCLUSION

I have discussed a new system of using musical layers for auralization of program code. I used this system of auralization to write music that potentially helps programmers debug computer programs more effectively, and ran an empirical study to test this hypothesis. The results of this study indicate that in some cases, the auralizations worked, and in others, they did not. Future research into auralization techniques should begin to classify, empirically, auralization techniques. The approach I will take with future work falls into two categories, iteration and automation.

The most obvious, and important, element of this research is to iterate the designs of the auralizations. The next iteration of the design should minimize the most problematic areas of the system. For example, a study of what semantic data would be useful to participants at runtime may be helpful in determining why the lyrics layer was unsuccessful. Future studies should, then, determine which elements of a computer program's runtime behavior are the most important, and then map sound or music directly to those elements.

Lyrics posed a problem for the participants in this study. Initial empirical results indicated that participants were either hurt, without training, or not helped, with training, when using lyrics based auralizations. However, to be clear, these initial results do not imply that lyrics are not useful, but that they were not useful in the current iteration of the design. In the next iteration, a two way communication model is extremely important, as it will allow more time to be given to longer words. Recall that in the current auralization design, words are spoken very quickly. The speed at which words are spoken may have had an effect on the participants interpretation of the sounds. In addition, elements of semantic data may not be useful to participants, and future iterations of the design should try to determine which elements are helpful, which are not, and most importantly, why.

The orchestration layer in this experiment aurally indicated information about the state of memory usage in the program, but did not indicate other elements of program state. Other elements, besides memory usage, may be useful to a participant debugging a program, but research is needed to determine the specifics. In regards to memory and the orchestration layer, saturation may occur if too much memory is used. In other words, if a computer program has no dynamic memory usage at a given point during execution, and then it, at some point in time, increases the memory usage dramatically, this will likely be obvious to a listener. Subtle changes in memory usage are, however, probably much more difficult to detect. Future iterations of the design should allow for subtle changes in memory usage to be obvious, as well as not so subtle changes.

However, by far the biggest problem with the auralization design, at this time, is the process of creating them. Creating an auralization, using current technology, requires the creation of computer code, generating inputs to that code, hand parsing the code, entering the results into a music notation editor, editing and recording sound files, and transferring the recordings to the appropriate medium such as compact disc. Any error that occurs in a stage X_i requires that X_{i+1} through X_{i+n} stages are reworked from scratch. In addition, all stages must occur for each and every sound file created, and with multi-group experiments, creating the auralizations takes months.

For example, in the current experiment, after a few months of work on the auralization creation, I discovered that a two way communication model between the lyrics layer and the cadential layer would allow the lyrics layer to have more time to say long numbers. However, a small change like changing the communication model would have required me to throw away months of work, for a change that would only potentially make the lyrics layer better. Since I had no empirical evidence suggesting one communication model was superior to another, reworking the design may have been wasted effort. If, however, a tool existed to automate the auralization creation process, iterations on the design could be created, and tested, far more efficiently.

Thus, in my opinion, automating the auralization creation process is by far the most important short term element of this research, as it will encourage rapid creation and testing of new auralization ideas. Unfortunately, not every element of auralization is trivial to automate. Automating the auralization process will require several key components, a parser, a music notation file writer, and perhaps teaching a computer program to sing lyrics. Creating these components may take significant effort, but not creating them may take more.

Program auralization is in its infancy. A nominal amount of empirical work has been done, virtually no sonified debuggers exist, and the auralization creation process itself is not well understood. The next generation of research will need to create structure for generating these auralizations, create standard empirical tests for testing their effectiveness, and determine which elements of runtime behavior programmers actually need to hear.

APPENDIX

APPENDIX A

CRASH COURSE IN MUSIC THEORY

This section enumerates the basic terminology used in music theory. This section is provided as a crash course in common musical concepts, used for the design of the current auralization system. Subjects do not necessarily need to understand any of the theory described here to potentially benefit from the auralization system created, similarly to a driver not needing to understand the engineering behind a car to drive it. However, for the purpose of enumerating the design, and design decisions, an understanding of music theory terminology is helpful. A comprehensive overview of music theory is beyond the scope of this work.

A.1 Notes and Intervals

The simplest possible idea in music theory is probably the note. Notes are graphically written onto a musical staff, and these notes indicate pitch in a score. Notations are attached to notes which indicate rhythms, and these rhythms indicate what frequencies are played over time, essentially creating music. Notes can be combined, creating intervallic relationships between other notes. While there are a huge variety of useful intervals, the most common are listed in Table A.1, along with an example pair of notes for that interval.

One of the easiest ways to think of particular interval is in analogy to a particular song. For example, an octave is the “Somewhere” in “Somewhere over the Rainbow,” from the Wizard of Oz, and a minor second is heard in the music to “Jaws” repeatedly. Of course, it much easier to understand intervals if they can also be heard.

A.2 Chords

Intervals are a useful construct because they give one building block for creating larger musical structures. Attempting to understand a concept like counterpoint without a firm knowledge of

Interval Name	Example Notes
Perfect Unison	C to C (The same note)
Minor second	C to Db
Major second	C to D
Minor third	C to Eb
Major third	C to E
Perfect fourth	C to F
diminished fifth	C to Gb
Perfect fifth	C to G
minor sixth	C to Ab
Major sixth	C to A
Minor seventh	C to Bb
Major seventh	C to B
Octave	C to C (The frequency of the original C * 2)

Table A.1: The basic intervals

intervals is, at best, difficult. The idea of intervals builds further into the idea of chords. Chords build up combinations of intervals. The most common chords are major and minor, which consist respectively of a perfect fifth and major third, and a perfect fifth and a minor third. Table A.2 gives examples of some of the most common chords, although the reader should be aware that there are thousands of others with fully qualified names. Further, depending on the musical genre, different notations and naming conventions exist. For example, classical musicians tend to use names like I^7, pronounced one-seven, whereas jazz players tend to use names like C Major 7 or $C^{\triangle}$.

A.3 Harmony

Chords can, again, be built up into larger conglomerates of notes over time, harmony. I limit the discussion here, again, to the most common chord progressions in classical music. Contemporary developments in harmony, although incredibly interesting in and of themselves, are far beyond the scope of this thesis. Harmony is typically written in classical music as numbers related to a key. A key is a focal point, like the note C, which all chords are considered relative too. For example, a C Major chord in the key of C Major is called a I, pronounced one, chord. If the key were C Minor,

Chord Name	Intervals involved	Example Notes
Major chord	Major third and Perfect fifth	C-E-G
Minor chord	Minor third and Perfect fifth	C-Eb-G
Major Seventh chord	Major third, Perfect fifth, and Major seventh	C-E-G-B
Minor Seventh chord	Minor third, Perfect fifth, and Minor seventh	C-Eb-G-Bb
Fully diminished seventh	Minor third, diminished fifth, and Diminished seventh	C-Eb-Gb-Bbb
Half diminished seventh	Minor third, diminished fifth, and Minor seventh	C-Eb-Gb-Bb

Table A.2: A selection of basic chords

Chord Number	Chord Name	Example Notes
I	C Major	C-E-G
ii	D Minor	D-F-A
iii	E Minor	E-G-B
IV	F Major	F-A-C
V	G Major	G-B-D
vi	A Minor	A-C-E
vii^o	B diminished	B-D-F

Table A.3: The chords in C Major

the chord C minor would also be considered a i chord, but the letter i would be lower case, as the chord in that key is minor. Note that i exists only in a minor key, not a major key. Table A.3 shows all of the typical chords considered to be in C Major in classical music. Notice that these chords only contain notes that are in the key of C Major, and no others.

Using numbers to represent harmony is useful, as the exact position and relation of notes can be abstracted. For example, the phrase $I - V - I$, in the key of C, would be the chord C Major followed by G Major, followed by C Major again. The phrase $I - IV - V - I$ is similar, but with an F Major chord after the initial I chord. Notice that if a key change occurs, perhaps to G, the phrase $I - IV - V - I$ will sound essentially the same, except for a transposition. Using numbers in this way is useful in an analogous way to using variables in mathematics. In the phrase $I - IV - V - I$ the exact key is an unknown, but the relationships between chords are important, similarly to $y = mx + b$. In the last equation, the numbers are unknown, but it is the relationship between the numbers that is what is important.

Leading further into harmony, secondary dominants are typically encountered. A secondary dominant can be considered a temporary key change, which will be discussed in more detail next. A secondary dominant is a V chord that is borrowed from another key. The easiest example of a secondary dominant would have the notation V/V. For example, if a piece of music was in the key of C, the V chord would have the notes G, B, and D. A secondary dominant for this V chord temporarily changes the key to this chord, or G major. thus the V/V in the key of C has the notes D, F#, and A. A typical chord progression using a secondary dominant would be $I - V/V - V - I$, taking note that the V/V chord leads to the V chord, the most common case.

The next topic of interest is a cadential pattern. A cadential pattern, or cadence, is a well defined, common, use of specific chords. These chords are used as endings to a phrase or piece of music. These endings are used so often, and in so many composer's works, that they are given special names to indicate their importance. The most important cadences are the authentic cadence and the plagal cadence. An authentic cadence is a $V - I$ progression, by far considered to be the most common cadential pattern in classical music. A plagal cadence, or amen cadence, is an ending with a $IV - I$ pattern, and is often heard in hymns.

Lastly, key changes are used in the current program auralization system I created, and thus should be discussed. A key change is when the tonal center of a series of chord progressions changes. For example, if I have the chord progression $I - V - I$ in the key of C, the chords are C-G and C respectively. If, however, I change keys to the key of G, the chord progression will be G-D-G. When classical music is written in 12-tone equal temperament, the typical tuning system used on modern pianos, every key sounds the same. Keep in mind, however, that historically this was not always the case, including in such famous pieces as The Well Tempered Clavier, by J.S. Bach.

APPENDIX B

SURVEYS

B.1 Pre-qualification survey: Part 1

1. Name: ____

2. What is your current GPA? ____________________________

3. What is your current GPA in computer science? __________

4. List all previously taken computer science courses, and grade received:

____	Grade:	____________
____	Grade:	____________
____	Grade:	____________
____	Grade:	____________
____	Grade:	____________
____	Grade:	____________
____	Grade:	____________
____	Grade:	____________

5. Have you ever used a debugger, if so, which ones?

__________________________	How many years?	____________
__________________________	How many years?	____________
__________________________	How many years?	____________
__________________________	How many years?	____________

B.2 Pre-qualification survey: Part 2, Ear Training Exam

In this portion of the test you will be asked to identify the differences between specific musical sounds. In each instance two sounds will be played, one after the other, and you will be asked to circle the word you feel most appropriately defines your answer.

1. Is the second note higher or lower than the first note? Higher/Lower/Don't know
2. Is the second note higher or lower than the first note? Higher/Lower/Don't know
3. Is the second note higher or lower than the first note? Higher/Lower/Don't know
4. Is the second note higher or lower than the first note? Higher/Lower/Don't know
5. Is the second note higher or lower than the first note? Higher/Lower/Don't know
6. Is the first melody exactly the same as the second melody? The same/Not the same/Don't know
7. Is the first melody exactly the same as the second melody? The same/Not the same/Don't know

B.3 Pre-qualification survey: Part 2, Ear Training musical examples

Previously shown was the survey sheet given to participants. The following are the musical examples played to participants. Note that the musical notation shown here was not shown to any subjects.

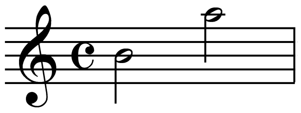


Figure B.1: Pre-Qualification survey: Part 2, Ear Training, Question 1

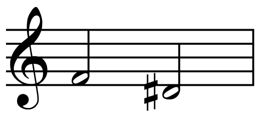


Figure B.2: Pre-Qualification survey: Part 2, Ear Training, Question 2

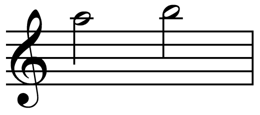


Figure B.3: Pre-Qualification survey: Part 2, Ear Training, Question 3

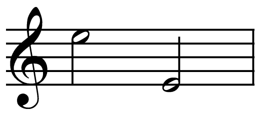


Figure B.4: Pre-Qualification survey: Part 2, Ear Training, Question 4

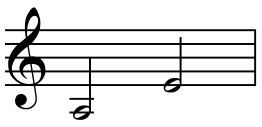


Figure B.5: Pre-Qualification survey: Part 2, Ear Training, Question 5

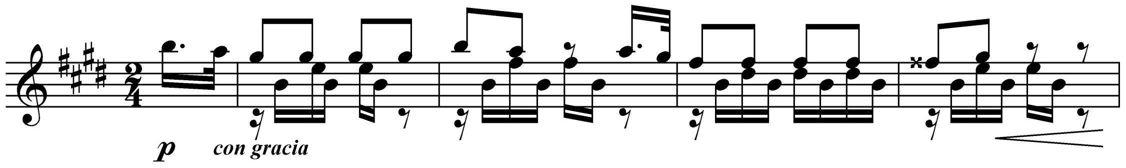


Figure B.6: Pre-Qualification survey: Part 2, Ear Training, Question 6 [41]

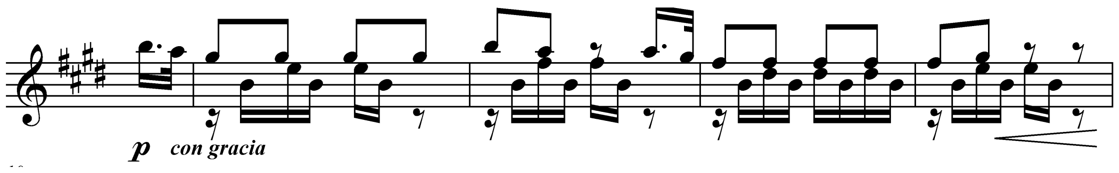


Figure B.7: Pre-Qualification survey: Part 2, Ear Training, Question 6, variation on [41]

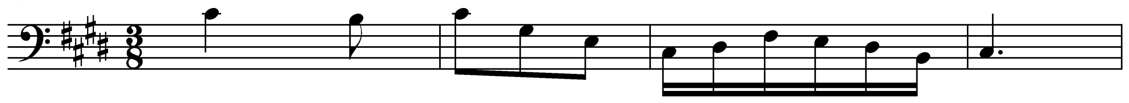


Figure B.8: Pre-Qualification survey: Part 2, Ear Training, Question 7

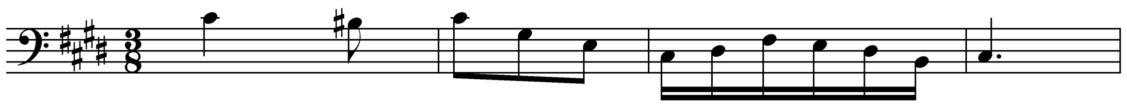


Figure B.9: Pre-Qualification survey: Part 2, Ear Training, Question 7

APPENDIX C

GUIDEBOOK GIVEN TO SUBJECTS

In this section we give the guide students receive when beginning the experimental study. Note that the introduction for this section was given in all three experimental sessions. In this thesis, to save space, the introduction is not repeated. In the last test, the address book, the subjects were told that bugs existed only in the AddressBook.cpp, AddressBook.h, LinkedList.cpp, and LinkedList.h files. Because the address book was so much larger, this was done to keep the amount of code they were actually looking through to a manageable level, and to keep the amount of code similar to the other tests.

C.1 Introduction

The following is the study description given to the participants.

Study Description:

Welcome to the CS223 study. In the following study, you will be asked to debug a computer program. Please do not turn the page until you are instructed to do so. You will be given 45 minutes to complete this task.

Task description:

Imagine you are developing software for a startup company. You just joined the team and are told that the latest version of this software needs to be checked for errors. No one on your development team knows how many errors exist in the code, if there are any at all. Your job is to read the code, from a printout your manager gives you, and find as many errors as you can.

Method for notating a bug:

Please clearly mark any errors you find directly on the code handout you are given. In addition, if you find an error, please write one or two short sentences as to why you think it is an error. Write these sentences, again, directly on the code handout.

Participation note:

Since this debugging study is based on participation over time, our results depend on your active participation. Reminder: You may drop an assignment in this course if and only if you attend all 3 debugging sessions, and we strongly encourage you to do so.

If you drop the course:

We ask that if you drop the CS223 course that you continue to attend all three debugging sessions. While you are by no means obligated to do so, our data depends on your consistent participation, and we would be grateful if you would continue for the 3 sessions anyway.

Again, thank you for your willingness to participate in this study, we appreciate it a great deal!

Please do not turn the page until you are instructed to do so.

C.2 Roulette

C.2.1 Roulette Game description

Roulette:

In game of roulette a ball is put onto a spinning table. Participants in the game place bets on where the ball will land. The version of Roulette you are about to analyze is simplified from the American casino game. In this version only two types of betting are allowed.

Bet type 1: Red-Black

Description: The user decides whether to place a bet on a red square or a black square. How to win: The participant wins if they chose the correct color. If you win: You receive 2 times the bet amount. If you had 500 dollars and you bet 50, you would now have 550 dollars. 500 - 50 + (50 * 2).

Bet type 2: Number

Description: The user chooses a number to bet on, between 1 and 35. How to win: The participant wins if they chose the correct number. Payout: You receive 35 times the bet amount. If you had 500 dollars and you bet 50 you would now have 500 - 50 + (35 * 50), or 2200 dollars.

C.2.2 Track Listings for V and M group

This list presents a series of program inputs. These inputs indicate the current state of the computer program at a particular point during execution. In addition to input, track numbers are listed, which correspond to the accompanying CD. Each track contains audio that relates to the program you are attempting to debug, although not all audio will illuminate a bug. Each track/Input pair has the following format:

******************** Example ***********************

Track Number: 4

Input: 362, c

Starting position for audio: spinWheel

******************** Example ***********************

In this example, this means that for track 4, the program was given an initial input of 362, c and that the audio on track 4 begins at the top line of the spinWheel() method. Not all methods have sound associated with them. Specifically, only custom written methods have sound, not system calls like cin, cout, or rand().

Track Number: 1

Input: 500, c, 5

Starting position for audio: spinWheel

Track Number: 2

Input: 361, c, 9

Starting position for audio: spinWheel

Track Number: 3

Input: 100, c, 5,

Starting position for audio: playNumber

Track Number: 4

Input: 600, c, 5,

Starting position for audio: playNumber

Track Number: 5

Input: 600, c, 0,

Starting position for audio: playNumber

Track Number: 6

Input: 600, n, r,

Starting position for audio: playRedBlack

Track Number: 7

Input: 600, q,

Starting position for audio: play

Track Number: 8

Input: 600, c, 5,

Starting position for audio: play

C.3 Bank

C.3.1 Bank Description

Bank:

The class you are about it analyze is a simple model of a standard bank's automatic teller machine, or ATM. There are several characteristics about this ATM machine which you should be aware to correctly debug this code.

1. To login the user must enter a username and password.

2. A blank name, the empty string, signifies that no one is logged in.

3. Any password is acceptable, real security was not implemented, and this is not considered a bug.

4. Bank users are allowed to withdraw a maximum of 300 dollars per day.

5. Bank users are only allowed to transfer money to three people, Andy, Melissa, or Fitz.

C.3.2 Track Listings for V and M group

This list presents a series of program inputs. These inputs indicate the current state of the computer program at a particular point during execution. In addition to input, track numbers are listed, which correspond to the accompanying CD. Each track contains audio that relates to the program you are attempting to debug, although not all audio will illuminate a bug. Each track/Input pair has the following format:

******************** Example ***********************

Track Number: 4

Input: 6

Starting position for audio: logout

******************** Example ***********************

In this example, this means that for track 4, the program was given an initial input of 6 and that the audio on track 4 begins at the top line of the logout() function. Not all methods have sound associated with them. Specifically, only member functions of the Bank class have sound associated with them.

Track Number: 1

Input: 6

Starting position for audio: logout

Track Number: 2

Input: 1

Bill

password

2

500

Starting position for audio: deposit

Track Number: 3

Input: 1

Bill

password

2

500

6

1

Starting position for audio: login

Note: Audio for track starts on the SECOND call to login.

Track Number: 4

Input: 4

Starting position for audio: withdraw

Track Number: 5

Input: 1

Bill

password

2

500

4

400

yes

Starting position for audio: withdraw

Track Number: 6

Input: 1

Bill

password

2

500

4

5134

Starting position for audio: withdraw

Track Number: 7

Input: 1

Bill

password

2

500

3

Andy

900

Starting position for audio: transfer

Track Number: 8

Input: 5

Starting position for audio: checkFunds

C.4 AddressBook

C.4.1 Address Book Description

Address Book:

The class you are about it analyze is a simple model of an address book. There are a few traits of this address book you should be aware of before debugging:

1. Only the AddressBook.cpp, LinkedList.cpp, AddressBook.h, and LinkedList.h files potentially have bugs in them.

2. All other files besides those listed in 1 are supplied only for your reference, they **DO NOT** contain bugs.

C.4.2 Track Listings for V and M group

This list presents a series of program inputs. These inputs indicate the current state of the computer program at a particular point during execution. In addition to input, track numbers are listed, which correspond to the accompanying CD. Each track contains audio that relates to the program you are attempting to debug, although not all audio will illuminate a bug. Each track/Input pair has the following format:

******************** Example **********************

Track Number: 4

Input: 5

Starting position for audio: AddressBook.printAll

******************** Example **********************

In this example, this means that for track 4, the program was given an initial input of 5 and that the audio on track 4 begins at the top line of the printAll() function. Not all methods have sound associated with them. Specifically, only member functions in the AddressBook and LinkedList class have sound associated with them.

Track Number: 1

Input: 1

Bill

122 N. Walnut St.

509-226-3476

Starting position for audio: LinkedList.add

Track Number: 2

Input: 1

Bill

122 N. Walnut St.

509-226-3476

2

Bill

Starting position for audio: LinkedList.remove

Track Number: 3

Input: 1

Bill

122 N. Walnut St.

509-226-3476

1

John

Blurft St.

555-555-5555

2

Bill

Starting position for audio: LinkedList.remove

Track Number: 4

Input: 1

Bill

122 N. Walnut St.

509-226-3476

1

John

Blurft St.

555-555-5555

2

John

Starting position for audio: LinkedList.remove

Track Number: 5

Input: 1

Bill

122 N. Walnut St.

509-226-3476

1

John

Blurft St.

555-555-5555

1

Adam

HappyTown Rd.

645-938-3635

1

Andy

MustBeAStud St.

938-490-3749

6

Starting position for audio: LinkedList.clear

Track Number: 6

Input: 1

Bill

122 N. Walnut St.

509-226-3476

4

John

Starting position for audio: AddressBook.findFriend

Track Number: 7

Input: 1

Bill

122 N. Walnut St.

509-226-3476

2

Bill

Starting position for audio: AddressBook.removeFriend

Track Number: 8

Input: 1

Bill

122 N. Walnut St.

509-226-3476

5

Starting position for audio: AddressBook.printAll

APPENDIX D

COMPLETE CODE

In this section we give the full code for all debugging examples, without bugs, and the bugs themselves are made explicit. If any bug has an obvious classification, when compared to [42], the classification is given.

D.1 Roulette

D.1.1 Bugs for the Roulette game

1. Method Roulette::play() Line: 35, 38, playRedBlack() and playNumber() should be swapped.

2. Method Roulette::play() Line: 42, If the user enters neither 'c' nor 'n', the program will display an uninitialized variable. This is an output fragmentation bug.

3. Method Roulette::play() Line: 88, The boolean should be set to true, else the program will never quit.

4. Method Roulette::play() Line: 78-81, The $>$ and $<$ signs should be swapped, else they display the reverse amount of money earned or gained.

5. Method Roulette::playRedBlack() Line: 103, This line should be an AND, not an OR, else the user will always choose black, regardless of whether they selected red. This bug is similar to an Or-for-and bug.

6. Method Roulette::playRedBlack() Line: 117, This line should be wonLost -= currentBid. If it were the other way, then the user would gain money when he/she was supposed to lose. This bug is similar to an Incorrect formula bug.

7. Method Roulette::playNumber() Line: 132, This line should be an OR, not an AND, else the user will never have an error condition, regardless of whether they choose an incorrect number. This bug is similar to an Or-for-and bug.

8. Method Roulette::playNumber() Line: 138, the *cout* statement should have the variable *answer* output to the console, not wonLost. The way it is currently written, 0 is always output.

D.1.2 Roulette code

```
#include ”Roulette.h”

/**
This program simulates a very simple version of roulette. Only two types of bets
are allowed in this simple program, namely black/red and a number.
*/
int main(int argc, int *argv[]) {
        Roulette roulette;
        roulette.play();
        return 0;
}
```

```
#ifndef ROULETTE_H
#define ROULETTE_H

typedef struct spinValue {
        int number;
        char color;
}SpinValue;


class Roulette {
public:
        Roulette() {}
        ~Roulette() {}

        void play();
private:
        static const int STARTING_DOLLARS = 1000;
        static const int RED_BLACK_BID = 0;
        static const int NUMBER_BID = 1;
        static const int NUM_ROULETTE_NUMBERS = 38;
        int currentMoney;
        int currentBid;
        int bidType;

        void init();
        int getBid();
        char getColorFromNumber(int num);
```

```cpp
    int playRedBlack();
    int playNumber();
    SpinValue spinWheel();
};

#endif
```

```cpp
#include "Roulette.h"

#include <iostream>
#include <windows.h>
#include <winbase.h>
#include <stdlib.h>
#include <time.h>

using namespace std;

void Roulette::init() {
    cout << "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n " <<
        "**************************************************\n";
    cout << "Welcome to Roulette. The house has given you" <<
        " $1000 with which to try your luck. All bet amounts" <<
        " must be in whole dollars. To play, enter your bet" <<
        " amount and then choose how you want to bet." << endl;

    cout << "**************************************************\n\n";
    currentMoney = STARTING_DOLLARS;
}

void Roulette::play() {
    init();
    bool quit = false;
    char answer;

    while(!quit) {
        int amountWon;
        currentBid = getBid();
        cout << "Would you like to bet on a color or a number? (Press c for color and n for number)";
        cin >> answer;

        if(answer == 'c') {
            amountWon = playNumber();
        }
        else if (answer == 'n') {
            amountWon = playRedBlack();
        }

        cout << "\n**************************************************\n";
        if(amountWon < 0) {
            cout << "You lost " << amountWon * -1 << " dollars.";
        }
        else {
            cout << "You won " << amountWon << " dollars.";
        }
        cout << "\n**************************************************\n";

        cout << "\nWould you like to play again? (Press y for yes and n for no)";
        cin >> answer;
        if(answer == 'y') {
            cout << "\n";

            quit = false;

            if(currentMoney == 0) {
                cout << "Sorry, you can't continue to play, you're broke.\n";
                    quit = true;
            }
            else if(currentMoney < STARTING_DOLLARS) {
                cout << "Ok, you currently have " << currentMoney << " dollars, having lost " <<
                    "a total of " << STARTING_DOLLARS - currentMoney << " dollars.";
            }
            else if ( currentMoney > STARTING_DOLLARS) {
                cout << "Ok, you currently have " << currentMoney << " dollars, having won " <<
                    "a total of " << currentMoney - STARTING_DOLLARS << " dollars.";
            }
            else {
                cout << "Ok, you currently have " << currentMoney << " dollars." <<
                    " So far you have broken even.";
            }
        }
        else {
            if(currentMoney == 0) {
                cout << "Looks like you lost your shorts ...\n";
            }
            if(currentMoney > STARTING_DOLLARS) {
                cout << "Ok, you lost a total of " << STARTING_DOLLARS - currentMoney << " dollars.";
            }
            else if ( currentMoney < STARTING_DOLLARS) {
                cout << "Ok, you won a total of " << currentMoney - STARTING_DOLLARS  << " dollars.";
            }
            else {
                cout << "Ok, looks like you broke even today.";
            }

            quit = false;
        }
        cout << "\n";
    }
}

/** Returns the amount of money won or lost in this game.*/
int Roulette::playRedBlack() {
    char answer;
    int wonLost = 0;

```

```
        bidType = RED_BLACK_BID;
        cout << "Would you like to bet on Red or Black? (Press r for red and b for black)";
        cin >> answer;

        if(answer != 'b' || answer != 'r') {
                cout << "Strange, choice, how about black instead." << endl;
                answer = 'b';
        }

        cout << "\nYou chose " << answer << ".\n" << endl;

        SpinValue val = spinWheel();
        if(val.color == answer) {
                currentMoney += currentBid;
                wonLost = currentBid;
        }
        else {
                currentMoney -= currentBid;
                wonLost = currentBid;
        }

        return wonLost;
}

/** Returns the amount of money won or lost in this game.*/
int Roulette::playNumber() {
        int answer;
        int wonLost = 0;

        bidType = NUMBER_BID;
        cout << "What number do you wish to bet on? (Between 1 and 35)";
        cin >> answer;

        if(answer < 1 && answer > 35) {
                cout << "Strange, choice, how about 17 instead." << endl;

                answer = 17;
        }

        cout << "\nYou chose " << wonLost << ".\n" << endl;

        SpinValue val = spinWheel();
        if(val.number == answer) {
                currentMoney -= currentBid;
                currentBid *= 35;
                currentMoney += currentBid;
                wonLost = currentBid;
        }
        else {
                currentMoney -= currentBid;
                wonLost = currentBid * -1;
        }
        return wonLost;
}

SpinValue Roulette::spinWheel() {
        SpinValue val;

        cout << "\n**************************************************\n";
        cout << "Spinning wheel";
        for(int i = 0; i < 10; ++i) {
                cout << ".";
                Sleep(100);
        }
        cout << "\n**************************************************\n\n";

        srand( (unsigned)time( NULL ) );

    int num = rand() % NUM_ROULETTE_NUMBERS;
        val.color = getColorFromNumber(num);
        if(num == 37) {
                num = 0;
        }
        val.number = num;

        if(val.color == 'g') {
                cout << "The wheel landed on " << val.number << ", green." << endl;
        }
        else if (val.color == 'r'){
                cout << "The wheel landed on " << val.number << ", red." << endl;
        }
        else if (val.color == 'b'){
                cout << "The wheel landed on " << val.number << ", black." << endl;
        }
        else {
                cout << "Oops, the ball fell off the table." << endl;
        }

        cout << "\n";

        return val;
}

char Roulette::getColorFromNumber(int num) {
        char answer;
        if(num == 1 || num == 3 || num == 5 || num == 7 || num == 9 || num == 12 ||
           num == 14 || num == 16 || num == 18 || num == 19 || num == 21 ||
           num == 23 || num == 25 || num == 27 || num == 30 || num == 32 ||
           num == 34 || num == 36 ) {
                        answer = 'r';
                        return answer;
        }
    else if(num == 2 || num == 4 || num == 6 || num == 8 || num == 10 || num == 11 ||
           num == 13 || num == 15 || num == 17 || num == 20 || num == 22 || num == 24 ||
              num == 26 || num == 28 || num == 29 || num == 31 || num == 33 || num == 35 ) {
                        answer = 'b';
                        return answer;
        }
    else if(num == 0 || num == 37) {
                        answer = 'g';
                        return answer;
        }
        else {
                cout << "Invalid number in getColorFromNumber: " << num << endl;
                return 'q'; //return something invalid
        }
}

int Roulette::getBid() {
        cout << "Please enter your bet amount: ";
        int bid;
        cin >> bid;

        if(bid < 1) {
                cout << "\nInvalid Amount, setting bet to 1.\n" << endl;
                bid = 1;
        }
        else if (bid > currentMoney) {
                cout << "\nInvalid Amount, setting bet to " << currentMoney << ".\n" << endl;
                bid = currentMoney;
        }
        return bid;
}
```

D.2 Bank

In this case, all bugs existed in Bank.cpp.

D.2.1 Bugs for the Bank

1. Method Bank::useAtm() Line: 56, 59, transfer() and withdraw() should be swapped. The menu for the system indicates that withdraw and transfer should be mapped to different keys than is indicated in the switch statement.

2. Method Bank::logout() Line: 96, The logout method does not actually logout the user from the system. To do this, it should set the user's name to "", set money = 0, and set amountWithdrawn = 0. Another approach would be to call init(), as this would also logout the current user.

3. Method Bank::deposit() Line: 106. A deposit should add to the user's account, not subtract from it. This line should be money += dep;

4. Method Bank::withdraw() Line: 132. The output statement above this line suggests to type "y" for yes, but the conditional statement expects the user to type yes.

5. Method Bank::withdraw() Line: 125. Since totWith is not initialized to 0, if (draw > money) or if(draw < 0), the output statement on line 153 will print out garbage.

6. Method Bank::withdraw() Line: 149. The line amountWithdrawn -= draw should be amountWithdrawn += draw, else the user can actually withdraw more than the maximum amount allowed on the system.

7. Method Bank::transfer() Line: 173. Melissa is spelled differently in in this line, when compared to the output statement. This bug is similar to an Incorrect constant bug.

8. Method Bank::transfer() Line: 177. This bug would force the user to transfer all money in the account. This would in turn allow the user to transfer more money than is in the account. This line should have if (amount > money), not if(amount < money).

D.2.2 Bank code

```
 1
 2 #include "Bank.h"
 3
 4 /**
 5 This program simulates a very simple version of of an ATM machine.
 6 */
 7 int main(int argc, char *argv[]) {
 8         Bank bank;
 9         bank.useAtm();
10         return 0;
11 }
```

```
1 #ifndef BANK_H
2 #define BANK_H
3
4 #include <string>
5
```

```
using namespace std;

class Bank {

public:
        Bank();
        ~Bank();
        bool login();
        bool logout();
        bool deposit();
        bool withdraw();
        void checkFunds();
        void useAtm();
        bool transfer();

private:
        static const int MAX_WITHDRAW;
        int showMenu();
        void init();
        string loggedIn;
        int money;
        int amountWithdrawn;
};

#endif
```

```
#include "Bank.h"
#include <string>
#include <iostream>

const int Bank::MAX_WITHDRAW = 300;

Bank::Bank() {
        init();
}

Bank::~Bank() {
}

bool Bank::login(){
        string name, pass;

        if(loggedIn != "") {
                cout << loggedIn << " is currently logged in, please " <<
                        "logout before attempting to log someone else in." << endl;
                return false;
        }
        else {
                bool goodName = false;
                while(!goodName) {
                        cout << "Please enter your name: ";
                        getline(cin, name);
                        if(name != "") {
```

```cpp
                goodName = true;
            }
            else {
                cout << "Invalid name, try again." << endl;
            }
        }
        cout << "Please Enter your password: ";
        cin >> pass;
        loggedIn = name;
        cout << "\n\n";
        return true;
    }
}

void Bank::useAtm() {
    cout << "Welcome to BrokeAsSmoke banking. Please enter your banking choice: \n\n";

    bool quit = false;
    while(!quit) {
        int choice = showMenu();
        switch(choice) {
            case 1:
                login();
                break;
            case 2:
                deposit();
                break;
            case 3:
                transfer();
                break;
            case 4:
                withdraw();
                break;
            case 5:
                checkFunds();
                break;
            case 6:
                logout();
                break;
            case 7:
                logout();
                quit = true;
                break;
            default:
                cout << "Invalid choice, try again." << endl;
                break;
        }
    }
}

int Bank::showMenu() {
    int input;
```

```
        cout << "\n********************* Menu *********************\n\n";
        cout <<  "1.    Login" << endl;
        cout <<  "2.    Make a Deposit" << endl;
        cout <<  "3.    Make a withdrawal" << endl;
        cout <<  "4.    Transfer funds" << endl;
        cout <<  "5.    Check funds" << endl;
        cout <<  "6.    Logout" << endl;
        cout <<  "7.    Exit" << endl;

        cout << "\n**********************************************\n\n";
        cout << "Your choice: ";
        cin >> input;
        cin.ignore(1);
        return input;
}

bool Bank::logout() {
        cout << "Thanks for banking." << endl;
        return true;
}

bool Bank::deposit(){
        if(loggedIn != "") {
                int dep = 0;
                cout << "Please enter the amount you would like to deposit: ";
                cin >> dep;
                money -= dep;
                cout << dep << " deposited, giving you " << money << " dollars total." << endl;
        }
        else {
                cout << "You must login before you can make a deposit." << endl;
        }
        return true;
}

bool Bank::withdraw(){
        if(loggedIn != "") {
                if(amountWithdrawn >= MAX_WITHDRAW) {
                        cout << "Sorry, no more money can be withdrawn, you have " <<
                                "reached your 300 dollar limit." << endl;
                }
                else {
                        int draw = 0;
                        cout << "Enter the amount you would like to withdraw: ";
                        cin >> draw;
                        int totWith;
                        if(draw + amountWithdrawn > MAX_WITHDRAW) {
                                cout << "You cannot withdraw that amount today, would you like " <<
                                        "to withdraw " << MAX_WITHDRAW - amountWithdrawn << " instead? " <<
                                        "(y for yes, n for no) ";
                                string answer;
                                cin >> answer;
                if(answer == ”yes”) {
                    money -= (MAX_WITHDRAW - amountWithdrawn);
                    amountWithdrawn = MAX_WITHDRAW;
                    totWith = MAX_WITHDRAW;
                }
                else {
                    totWith = 0;
                }
            }
            else if(draw > money) {
                cout << ”Sorry, you do not have ” << draw << ” dollars in your account.” << endl;
            }
            else if(draw < 0) {
                cout << ”Sorry, you cannot withdraw a negative amount.” << endl;
            }
            else {
                money -= draw;
                amountWithdrawn -= draw;
                totWith = draw;
            }

            cout << ”\n” << totWith << ” dollars withdrawn, leaving you with ” <<
                money << ” dollars left in the bank.\n”;
        }
    }
    else {
        cout << ”You must login before you can make a withdrawal.” << endl;
    }
    return true;
}

bool Bank::transfer(){
    if(loggedIn != ””) {
        string name;
        cout << ”The following people can have money transfered to them:\n” << endl;
        cout << ”Andy” << endl;
        cout << ”Melissa” << endl;
        cout << ”Fitz\n” << endl;

        cout << ”Type the name of the person you wish to transfer money to: ”;
        cin >> name;
        if(name == ”Andy” || name == ”Melisa” || name == ”Fitz”) {
            cout << ”How much would you like to transfer: ”;
            int amount;
            cin >> amount;
            if(amount < money) {
                cout << ”You only have ” << money << ” dollars.” << endl;
                cout << ”Would you like to transfer all of your money?(y for yes, n for no)”;
                string answer;
                cin >> answer;
                if(answer == ”y”) {
                    cout << ”Transfering ” << money <<
```

```
                        " dollars, leaving you with 0 dollars." << endl;
                    money = 0;
                }
                else {
                    cout << "Ok, stopping transfer." << endl;
                }
            }
            else if ( amount < 0) {
                cout << "Cannot transfer a negative amount" << endl;
            }
            else {
                money -= amount;
                cout << "Transfering " << amount <<
                    " dollars, leaving you with " << money << " dollars." << endl;
            }
        }
        else {
            cout << "No one by that name exists, sorry." << endl;
        }
    }
    else {
        cout << "You must login before you can transfer funds." << endl;
    }
    return true;
}

void Bank::checkFunds() {
    if(loggedIn != "") {
        cout << "You currently have " << money << " dollars." << endl;
    }
    else {
        cout << "You must login before you can check your funds." << endl;
    }
}

void Bank::init(){
    loggedIn = "";
    amountWithdrawn = 0;
    money = 0;
}
```

D.3 Address Book

In this test, bugs were located between two files, AddressBook.cpp and LinkedList.cpp.

D.3.1 Bugs for the Address Book

1. Method AddressBook::findFriend(), editFriend(), and removeFriend() Line: 105, 117, 130.

   These methods all assume that the value returned from the linked list will not be null, which is not the case if the key is not contained anywhere in the list.

2. Method AddressBook::printFriend(Friend* fr) Line: 143. This method assumes that Friend* fr is not null.

3. Method LinkedList::add(ListNode* node) Line: 15. When a head node is created the number of nodes is not incremented.

4. Method LinkedList::remove(string key) Line: 44. If the element being removed is the last in the list, the number of nodes is not properly decremented.

5. Method LinkedList::remove(string key) Line: 51. The prev = head line should be nested at the end of the while block, not the end of the if(node → getKey() == key) block.

6. Method LinkedList::remove(string key) Line: 55. The end of this method should return 0, not prev, because no node was found to remove.

7. Method LinkedList::clear() Line: 78. Deleting the ListNode does not delete the friend. The friend should also be deleted.

8. Method LinkedList::clear() Line: 80. Clear should set head = 0, to clear the linked list after deleting all of its elements.

D.3.2 Address Book code

```

#include "AddressBook.h"

/**
This program simulates a very simple address book.
*/
int main(int argc, char *argv[]) {
        AddressBook book;
        book.run();
        return 0;
}
```

```
#ifndef ADDRESS_BOOK_H
#define ADDRESS_BOOK_H

#include "LinkedList.h"
#include "Friend.h"

class AddressBook {

public:
        AddressBook();
        ~AddressBook();
        void run();
private:
        void init();
        void printFriend(Friend* fr);
        void printAll();
        void addFriend();
        void removeFriend();
        void editFriend();
        void findFriend();
        LinkedList book;
        int showMenu();
};

#endif
```

```
#include "AddressBook.h"
#include "ListNode.h"
#include "Iterator.h"

#include <iostream>
#include <string>

using std::string;
using std::cout;
using std::cin;
using std::endl;

AddressBook::AddressBook() {
}

AddressBook::~AddressBook() {
}

void AddressBook::run() {
        cout << "Address book menu, please choose what you would like to do: \n\n";

        bool quit = false;
        while(!quit) {
                int choice = showMenu();
                switch(choice) {
                        case 1:
                                addFriend();
                break;
            case 2:
                removeFriend();
                break;
            case 3:
                editFriend();
                break;
            case 4:
                findFriend();
                break;
            case 5:
                printAll();
                break;
            case 6:
                book.clear();
                break;
            case 7:
                quit = true;
                break;
            default:
                cout << "Invalid choice, try again." << endl;
                break;
        }
    }
}

int AddressBook::showMenu() {
    int input;
    cout << "\n********************* Menu *********************\n\n";
    cout << "1.    Add a friend" << endl;
    cout << "2.    Delete a friend" << endl;
    cout << "3.    Edit information for a friend" << endl;
    cout << "4.    Search for a friend" << endl;
    cout << "5.    List all friends" << endl;
    cout << "6.    Delete all friends" << endl;
    cout << "7.    Exit" << endl;

    cout << "\n************************************************\n\n";
    cout << "Your choice: ";
    cin >> input;
    cin.ignore(1);
    return input;
}

void AddressBook::init() {
    book.clear();
}

void AddressBook::addFriend() {
    string name, address, phone;
    cout << "Please type the name: ";
    getline(cin, name);
```

```

        cout << "Please type the address: ";
        getline(cin, address);

        cout << "Please type the phone number: ";
        getline(cin, phone);

        Friend* obj = new Friend(name);
        obj->address = address;
        obj->phoneNumber = phone;

        cout << "\n";
        printFriend(obj);

        ListNode* node = new ListNode(name, obj);
        book.add(node);
}

void AddressBook::findFriend() {
        string name;
        cout << "Please type the name of the friend you wish to find: ";
        getline(cin, name);

        ListNode* node = book.get(name);

        Friend* fr = node->getObject();
        cout << "\nFriend " << name << " found." << endl;
        printFriend(fr);
}

void AddressBook::removeFriend() {
        string name;
        cout << "Please type the name of the friend you wish to remove: ";
        getline(cin, name);

        ListNode* node = book.remove(name);

        Friend* fr = node->getObject();
        cout << "\nFriend " << name << " found, deleting." << endl;
        delete(fr);
        delete(node);
}

void AddressBook::editFriend() {
        string name, address, phone;
        cout << "Please type the name of the friend you wish to edit: ";
        getline(cin, name);

        ListNode* node = book.get(name);

        Friend* fr = node->getObject();
        cout << "\nFriend " << name << " found." << endl;

        cout << "Please type the new address: ";
        getline(cin, address);

        cout << "Please type the new phone number: ";
        getline(cin, phone);

        fr->address = address;
        fr->phoneNumber = phone;

        cout << "New Information for " << fr->getName() << endl;
        printFriend(fr);
}

void AddressBook::printFriend(Friend* fr) {
        cout << "Name: " << fr->getName() << endl;
        cout << "Address: " << fr->address << endl;
        cout << "Phone Number: " << fr->phoneNumber << endl;
}

void AddressBook::printAll() {
        cout << "You have " << book.size() << " friends." << endl;
        Iterator it(&book);
        while(it.hasNext()) {
                Friend* fr = (it.next())->getObject();
                cout << fr->getName() << endl;
        }
}
```

```
#ifndef FRIEND_H
#define FRIEND_H

#include <string>

using std::string;

class Friend {
public:
        Friend(string friendsName);
        ~Friend();
        string address;
        string phoneNumber;
        string getName();

private:
        string name;
};

#endif
```

```
#include "Friend.h"

Friend::Friend(string friendsName) {
	name = friendsName;
}

Friend::~Friend() {

}

string Friend::getName() {
	return name;
}
```

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#include "ListNode.h"

#include <string>

using std::string;

class LinkedList {
public:
	LinkedList();
	~LinkedList();
	bool add(ListNode* node);
	ListNode* remove(string key);
	ListNode* getFirst();
	ListNode* get(string key);
	void clear();
	int size();

private:
	int numNodes;
	ListNode* head;
};


#endif
```

```
#include "LinkedList.h"
#include "Iterator.h"

LinkedList::LinkedList() {
	head = 0;
	numNodes = 0;
}

LinkedList::~LinkedList() {
	clear();
}

bool LinkedList::add(ListNode* node) {
	if(!head) {
```

```
            head = node;
            return true;
        }
        else {
            node->setNext(head);
            ++numNodes;
            head = node;
            return true;
        }
}

ListNode* LinkedList::remove(string key) {
        Iterator it(this);
        ListNode *prev = 0;
        while(it.hasNext()) {
            ListNode* node = it.next();
            if(node->getKey() == key) {
                if(prev == 0) {
                    if(node->getNext() == 0) {
                        head = 0;
                        --numNodes;
                    }
                    else {
                        head = node->getNext();
                        --numNodes;
                    }
                }
                else {
                    if(node->getNext() == 0) {
                        prev->setNext(0);
                    }
                    else {
                        prev->setNext(node->getNext());
                        --numNodes;
                    }
                }
                prev = node;
                return node;
            }
        }
        return prev;
}

ListNode* LinkedList::get(string key) {
        Iterator it(this);
        while(it.hasNext()) {
            ListNode* node = it.next();
            if(node->getKey() == key) {
                return node;
            }
        }
        return 0;
```

```
}

ListNode* LinkedList::getFirst() {
        return head;
}

void LinkedList::clear() {
        Iterator it(this);
        while(it.hasNext()) {
                ListNode* node = it.next();
                Friend* fr = node->getObject();
                delete(node);
        }
        numNodes = 0;
}

int LinkedList::size() {
        return numNodes;
}
```

```
#ifndef LIST_NODE_H
#define LIST_NODE_H

#include "Friend.h"
#include <string>

using std::string;

class ListNode{
public:
        ListNode(string nodeKey, Friend *obj);
        ~ListNode();
        ListNode* getNext();
        void setNext(ListNode* nextNode);

        Friend* getObject();
        string getKey();

private:
        string key;
        Friend *object;
        ListNode *next;
};


#endif
```

```
#include "ListNode.h"

ListNode::ListNode(string nodeKey, Friend *obj) {
        key = nodeKey;
        object = obj;
        next = 0;
}

ListNode::~ListNode() {

}

ListNode* ListNode::getNext() {
        return next;
}

void ListNode::setNext(ListNode* nextNode) {
        next = nextNode;
}


Friend* ListNode::getObject() {
        return object;
}

string ListNode::getKey() {
        return key;
}
```

```
#ifndef ITERATOR_H
#define ITERATOR_H


#include "ListNode.h"
#include "LinkedList.h"

class Iterator {
public:
        Iterator(LinkedList* linkedList);
        ~Iterator();
        ListNode* next();
        bool hasNext();
        void reset();
private:
        LinkedList* list;
        ListNode* current;
};

#endif
```

```
#include "Iterator.h"

Iterator::Iterator(LinkedList* linkedList) {
        list = linkedList;
        current = list->getFirst();
}

Iterator::~Iterator() {}

```

```
ListNode* Iterator::next() {
        ListNode* returnMe = current;
        if(current != 0) {
                current = current->getNext();
                return returnMe;
        }
        else {
                return 0;
        }
}

bool Iterator::hasNext() {
        if(current == 0) {
                return false;
        }
        else {
                return true;
        }
}

void Iterator::reset() {
        current = list->getFirst();
}
```

BIBLIOGRAPHY

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, New York, NY, USA, 2005. ACM Press.

[2] James L. Alty and Dimitrios Rigas. Exploring the use of structured musical stimuli to communicate simple diagrams: the role of context. *Int. J. Hum.-Comput. Stud.*, 62(1):21–40, 2005.

[3] James L. Alty and Dimitrios I. Rigas. Communicating graphical information to blind users using music: the role of context. In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 574–581, New York, NY, USA, 1998. ACM Press/Addison-Wesley Publishing Co.

[4] James L. Alty, Dimitrios I. Rigas, and Paul Vickers. Using music as a communication medium. In *CHI '97 electronic publications*, pages 22–27, 1997.

[5] V.R. Basili. Software modeling and measurement: The goal/question/metric paradigm. *University of Maryland at College Park Computer Science Technical Report UMIACS-TR-92-96*, pages 1–24, sep 1992.

[6] David B. Boardman, Geoffrey Greene, Vivek Khandelwal, and Aditya P. Mathur. Listen: A tool to investigate the use of sound for the analysis of program behavior. In *Computer Software and Applications Conference, 1995. COMPSAC 95. Proceedings., Nineteenth Annual International*, pages 184–189, 1995.

[7] Jeffrey Bonar and Elliot Soloway. Uncovering principles of novice programming. In *POPL*

'83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 10–13, New York, NY, USA, 1983. ACM Press.

[8] Stephen A. Brewster, Peter C. Wright, and Alistair D. N. Edwards. An evaluation of earcons for use in auditory human-computer interfaces. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 222–227, New York, NY, USA, 1993. ACM Press.

[9] Ryan Chmiel and Michael C. Loui. Debugging: from novice to expert. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 17–21, New York, NY, USA, 2004. ACM Press.

[10] Gilbert Cockton. From doing to being: bringing emotion into interaction. *Interacting with Computers*, 14(2):89–92, feb 2002.

[11] Alan R. Dyer. Comparisons of tests for normality with a cautionary note. *Biometrika*, 61(1):185–189, Apr 1974.

[12] Joan M. Francioni, Larry Albright, and Jay Alan Jackson. Debugging parallel programs using sound. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 68–75, New York, NY, USA, 1991. ACM Press.

[13] Joan M. Francioni and Jay Alan Jackson. Breaking the silence: auralization of parallel program behavior. *J. Parallel Distrib. Comput.*, 18(2):181–194, 1993.

[14] Joan M. Francioni, Jay Alan Jackson, and Larry Albright. The sounds of parallel programs. In *Proceedings, The Sixth*, pages 570–577, April, May 1991.

[15] Joan M. Francioni and Ann C. Smith. Computer science accessibility for students with visual disabilities. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 91–95, New York, NY, USA, 2002. ACM Press.

[16] William W. Gaver. Auditory icons: Using sound in computer interfaces. *Human-Computer Interaction*, 2:167–177, 1986.

[17] William W. Gaver. The sonic finder: An interface that uses auditory icons. *Human-Computer Interaction*, 4:67–94, 1989.

[18] William W. Gaver. Synthesizing auditory icons. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 228–235, New York, NY, USA, 1993. ACM Press.

[19] M.C. Gopinath. Auralization of an intrusion detection system. Master's thesis, Purdue University, BITS, Pilani, India, May 2004.

[20] Franklin A. Graybill. Discussion: What is an analysis of variance? *The Annals of Statistics*, 15(3):921–923, sep 1987.

[21] J. Klein, Y. Moon, and R. W. Picard. This computer responds to user frustration: Theory, design, and results. *Interacting with Computers*, 14(2):119–140, feb 2002.

[22] Jonathan Klein, Youngme Moon, and Rosalind W. Picard. This computer responds to user frustration. In *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*, pages 242–243, New York, NY, USA, 1999. ACM Press.

[23] Donald E. Knuth. The complexity of songs. *SIGACT News*, 9(2):17–24, 1977.

[24] Donald E. Knuth. The complexity of songs. *Commun. ACM*, 27(4):344–346, 1984.

[25] Grgory Lepltre and Iain McGregor. How to tackle auditory interface aesthetics? discussion and case study. In *Proceedings of ICAD 04-Tenth Meeting of the International Conference on Auditory Display*, july 2004.

[26] Ann Light. Designing to persuade: the use of emotion in networked media. *Interacting with Computers*, 16(4):729–738, jul 2004.

[27] Robert M. Greenberg Meera M. Blattner, Denise A. Sumikawa. Earcons and icons: Their structure and common design principles. *Human-Computer Interaction*, 4:11–44, 1989.

[28] D.C. Montgomery. *Design and Analysis of Experiments, Fourth Edition*. John Wiley and Sons, 1997.

[29] Nancy Pennington. Comprehension strategies in programming. In Gary M. Olson, Sylvia Sheppard, Elliot Soloway, and Ben Shneiderman, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113, Westport, CT, USA, 1987. Greenwood Publishing Group Inc.

[30] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[31] R. W. Picard, J. Klein, and Y. Moon. Computers that recognize and respond to human emotion: theoretical and practical implications. *Interacting with Computers*, 14(2):141–169, feb 2002.

[32] R. Jagadish Prasath. Auralization of web server using jlisten. Master's thesis, Purdue University, BITS, Pilani, India, May 2004.

[33] D. I. Rigas and J. L. Alty. Using sound to communicate program execution. In *EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO*, volume 2, pages 625–632, Washington, DC, USA, aug 1998.

[34] Dimitrios Rigas and James Alty. The rising pitch metaphor: an empirical study. *Int. J. Hum.-Comput. Stud.*, 62(1):1–20, 2005.

[35] Dimitrios Rigas, James Alty, and F.W. Long. Can music support interfaces to complex databases? *EUROMICRO 97. 'New Frontiers of Information Technology'., Proceedings of the 23rd EUROMICRO Conference*, pages 78–84, 1997.

[36] Jocelyn Scheirer, Raul Fernandez, Jonathan Klein, and Rosalind W. Picard. Frustrating the user on purpose: a step toward building an affective computer. *Interacting with Computers*, 14(2):93–118, feb 2002.

[37] S. S. Shapiro and R. S. Francia. An approximate analysis of variance test for normality. *Journal of the American Statistical Association*, 67(337):215–216, 1972.

[38] Ann C. Smith, Justin S. Cook, Joan M. Francioni, Asif Hossain, Mohd Anwar, and M. Fayezur Rahman. Nonvisual tool for navigating hierarchical structures. In *ASSETS '04: Proceedings of the ACM SIGACCESS conference on Computers and accessibility*, pages 133–139, New York, NY, USA, 2004. ACM Press.

[39] Ann C. Smith, Joan M. Francioni, and Sam D. Matzek. A java programming tool for students with visual disabilities. In *Assets '00: Proceedings of the fourth international ACM conference on Assistive technologies*, pages 142–148, New York, NY, USA, 2000. ACM Press.

[40] Diane H. Sonnenwald, B. Gopinath, Gary O. Haberman, William M. Keese III, and John.S. Myers. Infosound: an audio aid to program comprehension. In *System Sciences, 1990., Proceedings of the Twenty-Third Annual Hawaii International Conference on*, volume 2, pages 541–546, jan 1990.

[41] Fernando Sor. Variations on a theme by mozart, op. 9.

[42] James G. Spohrer and Elliot Soloway. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 230–251, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[43] Noam Tractinsky. Tools over solutions? comments on interacting with computers special issue on affective computing. *Interacting with Computers*, 16(4):751–757, jul 2004.

[44] Paul Vickers. *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.

[45] Paul Vickers. External auditory representations of programs: Past, present, and futurean aesthetic perspective. In *Proceedings of ICAD 04-Tenth Meeting of the International Conference on Auditory Display*, july 2004.

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

[47] Paul Vickers and James L. Alty. When bugs sing. *Interacting with Computers*, 14(6):793–819, 2002.

[48] Paul Vickers and James L. Alty. Siren songs and swan songs debugging with music. *Communications of the ACM*, 46(7):86–93, jul 2003.

[49] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering An Introduction*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.