|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Author** | **Reviewer** |
| 1.0 | 10/09/2017 | Leonid Shepetovsky | Shmuel Tyshberovich |
| 2.0 | 30/09/2017 | Leonid Shepetovsky | Shmuel Tyshberovich |



M.Sc. Final Project summary in Computer Science

*by LEONID SHEPETOVSKY (310793583)*

***Demonstrating Software Design Improvement***

***by a***

***Formal Model for Design Decomposition***

2017

**Table of Contents**

[1 Introduction 5](#_Toc492767931)

[2 Background 5](#_Toc492767932)

[3 Project Goals 6](#_Toc492767933)

[4 System Overview 7](#_Toc492767934)

[4.1 System Requirements Analysis 7](#_Toc492767935)

[4.1.1 Functionality overview 7](#_Toc492767936)

[4.1.2 Reverse-engineering system specifications 7](#_Toc492767937)

[4.1.3 Limitations and Assumptions 7](#_Toc492767938)

[4.2 Audacity Design 8](#_Toc492767939)

[4.2.1 Design Overview 8](#_Toc492767940)

[4.2.2 Software Detailed Design 8](#_Toc492767941)

[4.2.3 Audacity Class Diagram – main elements 9](#_Toc492767942)

[5 Formal Analysis 11](#_Toc492767945)

[5.1 The formal Model 11](#_Toc492767946)

[5.1.1 Alloy Overview 11](#_Toc492767947)

[5.1.2 Audacity Alloy Model 12](#_Toc492767948)

[5.1.3 Analyzing the Formal Model 14](#_Toc492767949)

[6 Conclusions 16](#_Toc492767951)

[7 Future Work 16](#_Toc492767952)

[8 Appendices 17](#_Toc492767953)

[8.1 Appendix A: Audacity features derived from UI 17](#_Toc492767954)

[8.1.1 File Menu 17](#_Toc492767955)

[8.1.2 Recording Toolbar 17](#_Toc492767956)

[8.1.3 Playback Meter toolbar 17](#_Toc492767957)

[8.1.4 Mixer Toolbar 18](#_Toc492767958)

[8.1.5 Edit Toolbar 18](#_Toc492767959)

[8.2 Appendix B: Audacity ARS reverse-engineered 19](#_Toc492767960)

[8.2.1 Terms 19](#_Toc492767961)

[8.2.2 Functional Requirements 19](#_Toc492767962)

[8.3 Appendix C: Audacity design impacting features 25](#_Toc492767963)

[9 References 26](#_Toc492767964)

# Introduction

This document summarizes research work made as a final project in M.Sc. studies in Computer Sciences.

The work is based on a research made by Prof. Shmuel Tyszberowicz and Dr. David Faitelson and published in paper a "Improving Design Decomposition" [1]. It addresses an issue of decomposing a software system into subsystems as part of the system's design process. As a solution, it describes a visual notation for diagramming the composition of subsystems, and an automatic technique for suggesting good decompositions.

The goal of current project is to support the above technique by providing a proof by example. Though the above paper already analyzes a simple theoretical software system as an example, we wanted to apply this technique on larger, real and possibly known software system.

The work begun with some initial assumptions and directions that will be described below. However, during the research, these directions have been changed according to the acquired results and the scope of the research began to expand. This document provides an overview on the scope of the work within the limits of the Final Project course requirements. Further research is currently in progress and will be part of my thesis.

# Background

During my studies of Computer Science and Software Engineering, I tried to attend as much courses as I can that deal with Software Engineering. Each course tried to emphasize slightly different topics, but in general all of them turned around of the two phases of the software development process that precede the code writing itself: System Analysis and Design.

However, the most interesting thing I found is that while all these studies begun with the System Analysis part, all actually also finished there as well. The discussion about software design was usually limited by reviewing Design Patterns.

In a contrast, the same studies proposed quite a strong model System Analysis, which can allow legible decomposition of software system requirements of any scale.

Such disproportion in a study depth of these two, equally highly important, blocks of the Software Engineering science raised my attention more and more, until after years of experience as a Software Engineer, I understood that there is actually no existing (or, perhaps, commonly accepted) formal model for building a high-level (and further a detailed) design of a software system form its requirements specification (SRS).

As a consequence, the quality of a software design made by a software architect strongly relies on the architect’s experience, intuition, and even sometimes just his preferences. On the other hand, assuming we have two designs of the same software system made by two different architects (with different experience and preferences), they may be totally different, and hardly comparable. Moreover, it is not always possible to tell which one is better and how close any of them to the "best'' possible variant.

# Project Goals

In their research, published in "Improving Design Decomposition" [1] paper, prof. Shmuel Tyszberowicz and Dr. David Faitelson deal with improving design decomposition. They suggested a model that provides a way to determine components of a software system based on formalized functional requirements to this system. Those components may be realized as atomic design units such as classes in an object-oriented model. The proposed model is based on the basic principles of software design: loose coupling and strong cohesion.

The goal of this project is to support the suggested model by using a large example, proving its availability and scaling. To provide such a proof we planned to choose some open source code and perform the below steps:

* acquire system's design and requirements;
* create a formal expression of the requirements;
* create a design from the formal requirements using the suggested model; and
* compare the received design with the existing one and determine subjective measurable value of improvement.

The selection criteria of the system that will be used in this work was: a widely used open-source system, with a potentially known design. (Many open source code systems do not provide their initial design.) The candidate software systems have been those described in "The Architecture of Open Source Applications" [2]. The selected software system was [Audacity](http://aosabook.org/en/audacity.html) - a popular sound recorder and audio editor. This system was selected as it is a non-trivial system, yet with a reasonable size, and a list of features that can be analyzed within the time frame we had.

# System Overview

## System Requirements Analysis

### Functionality overview

[The Audacity](http://www.audacityteam.org/) is a widely used open-source audio editor with a strong supporting community. There is a Wiki site maintained by the community. It contains [user manual](http://manual.audacityteam.org/) and [release notes](http://wiki.audacityteam.org/wiki/Release_Notes) sections. However, there is no official Software Requirements Specification document.

The software development started in the fall of 1999 by Dominic Mazzoni and Roger Dannenberg at Carnegie Mellon University, and the system was released on May 28, 2000. Since then and until today it remains maintained, and new features and improvements are periodically released.

To support many different platforms, audio formats, codecs, filters, etc., the system is heavily plugin-based.

### Reverse-engineering system specifications

Because there is no software the lack of initial specification, we have extracted the requirements list for our study from the software itself. the system features set is a result of the evolutionary process of system's development since 2000 and until today, We use the latest version of the software, as available at the time of the study beginning.

This reverse-engineering process was based on the software UI. First, we tried and documented all the options and features that appear on, or are accessible from, the main window. [Appendix A](#_Appendix_A:_Audacity) contains the detailed list of those functions.

Next, the Application Requirement Specification (ARS) was derived from the above list of functionalities. [Appendix B](#_Appendix_B:_Audacity) contains the derived ARS.

### Limitations and Assumptions

Later on, when we have studied Audacity's design, we have derived a shorter list of features to be formalized–those features which have a direct impact on the system's high-level structure. It is important to understand that it is almost impossible (or at least not time-efficient), and therefore impractical, to formalize all small features of a software. In the case of Audacity, those features which are/can be implemented as plug-ins have very small (if any) effect on the system's design. Also, many features are just a single line of code implementation in the software, or just an additional loop or interface implementation. For example: support of different audio formats, different playback devices, mixing modes, etc. However, the few features listed in [Appendix C](#_Appendix_C:_Audacity) are the ones which determined almost the entire design of the software system. Of course, this statement is questionable and someone can decide that other features (may) have an impact on the design, which is sufficient to include them in the study For reasons of time and complexity of the study, we decided to stick to the list of features mentioned in [Appendix C](#_Appendix_C:_Audacity).

##  Audacity Design

### Design Overview

A software design overview is provided by [James Crook](http://aosabook.org/en/intro1.html#crook-james) in chapter of "[The Architecture of Open Source Applications](http://aosabook.org/en/audacity.html)" [2]. The chapter does not provide the full system's high-level design in the formal way, however it addresses and explains most of the important considerations, i.e. both functional and non-functional requirements which are used for the system's design. Some of those considerations are taken into consideration in our study despite the fact that they are not referring to any specific functionality visible from Audacity's User Interface.

### Software Detailed Design

This part is completely based on the reverse-engineering work. Audacity is mainly written in C++ (compatible for both Linux and Windows OSs).

Audacity's code is difficult to read and understand; but this is the nature of most of the long-term evolving open-source software. Therefore, the reverse-engineering work was done in the following steps:

* Auto generation of software API documentation from the source-code using the [Doxygen](http://www.stack.nl/~dimitri/doxygen/) tool. The generated interactive documentation can be found in GitHub: <https://github.com/leonidsh119/audacity-doxygen.git>
* Determining the base classes that are defining the skeleton of system's structure. Actually, most classes do not have much impact (if any) on the system's high-level design, and it is neither possible nor practical to map all of them inside our diagrams., There are, however, a few classes of very high importance in the system's architecture, and it was highly important to understand their function. The auto-generated documentation helped to outline these classes. [Section 4.2.3](#_Audacity_Class_Diagram) presents the class diagram describing main Audacity classes.

### Audacity Class Diagram – main elements

###

###

# Formal Analysis

## The formal Model

### Alloy Overview

For the formal model description, it was chosen the [Alloy modelling language](http://alloy.mit.edu/alloy/) [3].

Alloy is a lightweight modelling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer, and provides a visualizer for checking solutions and counterexamples it finds. Alloy is a declarative specification language for expressing complex structural constraints and behavior in a software system. Alloy provides a simple structural modeling tool based on first-order logic. The mathematical underpinnings of the language were heavily influenced by the Z notation [3], although the syntax of Alloy owes more to languages such as Object Constraint Language. Alloy is targeted at the creation of micro-models that can then be automatically checked for correctness. Alloy specifications can be checked using the Alloy Analyzer. Although Alloy is designed with automatic analysis in mind, Alloy differs from many specification languages designed for model-checking by permitting the definition of infinite models. The Alloy Analyzer is designed to perform finite scope checks even on infinite models.

Below we provide a brief description of Alloy's notation. The detailed specification is provided in paper "Alloy: A Lightweight Object Modelling Notation" [5].

Alloy models are relational in nature, and are composed of several different kinds of statements. Because it is a declarative language the meaning of a model is unaffected by the order of statements.

* **Signatures** define the vocabulary of a model by creating new sets

**sig** Object{} defines a signature *Object*

**sig** List{ head : **lone** Node } defines a signature *List* that contains a field *head* of type *Node* and multiplicity *lone* - this establishes the existence of a relation between *List*s and *Node*s such that every *List* is associated with no more than one head *Node*

* **Facts** are constraints that are assumed to always hold
* **Predicates** are parameterized constraints, and can be used to represent operations
* **Functions** are expressions that return results
* **Assertions** are assumptions about the model that can be checked using the [Alloy Analyzer](https://en.wikipedia.org/wiki/Alloy_Analyzer)

### Audacity Alloy Model

The reverse-engineering process of Audacity's design (see [section 4.2](#_Software_Detailed_Design)) outlined the key elements which became a basis for our formal model. The actions described in this model correspond to the key design features, namely:

* Import
* Cut
* Paste
* Zoom-In
* Zoom-Out
* Undo
* Redo

See [Appendix C](#_Appendix_C:_Audacity_1) for detailed explanation of the features and their effect on the system's design.

In order to provide formal description of the above actions, our model consists from the following atoms and relations:



Where:

* ***Sample*** represents a single sample in the audio file
* ***SamplesContainer*** represents a generic sequence of samples
* ***Clipboard*** – the sequence of samples used to hold samples between cut and paste operations
* ***Track*** – the sample container represents an audio channel that is initially loaded from a file, but further may be edited by cut/paste (or other) operations.
* ***Window*** represents the visible part of a track
* ***History*** – this is the formal way to represent an over-the-time memory of the model states to allow formal definition of the Undo and Redo actions.

Since the entire Alloy code with facts, predicates and checks for all the actions will overload this document, it is not presented here, however the full code is open-source and can be easily found on GitHub: <https://github.com/leonidsh119/audacity-alloy-model>

### Analyzing the Formal Model

When the formal model is created, we want to analyze it in a way that will help us to produce a design which will be optimal in the terms of strong cohesion and loose coupling.

* **loosely coupled** system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. Subareas include the coupling of classes, interfaces, data, and services
* The **cohesion** of a module is a measure for how well the internal parts of a module (e.g. the methods and attributes of a class) belong together. Having a high cohesion means, that a module should only comprise responsibilities which belong together.

The idea behind analyzing of the formal model is described in the prof. Shmuel Tyszberowicz and dr. David Faitelson work which is summarized in the "Improving Design Decomposition" article.

The goal of the analysis is to determine clusters of state variables (entities, classes, etc.) with strong connections inside the clusters and loose connections between the clusters. For our abstract Audacity model, the dependencies between the model atoms (state variables of the system) and operations (methods and functions in the system) are summarized in the table below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Actions\Entities** | **\_samples** | **\_tracks** | **\_start** | **\_end** | **\_winsamples** | **history** | **present** |
| **Import** | r | w | w | w | w | w | w |
| **Cut** | w |   | w | w | w | w | w |
| **Paste** | w |   |   |   | w | w | w |
| **ZoomIn** | r |   | w | w | w | w | w |
| **ZoomOut** | r |   | w | w | w | w | w |
| **Undo** | w | w | w | w | w |   | w |
| **Redo** | w | w | w | w | w |   | w |

Where:

* "**r**" represents the "**read-only**" dependency from operation to state variable
* "**w**" represents the "**read-write**" dependency from operation to state variable

Note that in the weighted graph the "**read-write**" dependency will have double weight from "**read-only**" dependency.

##

Abstract Audacity variable-operation dependency graph

# Conclusions

Making the correct conclusions is the most challenging part of this project.

We began the work with a goal of creating a measurable comparison between the existing design of Audacity and the design derived from analysis of formalized set of requirements.

However, even though, the Audacity is not a very large system, taking in account all its parts, modules and requirements will make the work infinite.

From the other side, the too simplified abstract model appeared to be insufficient to point on a good decomposition because it disregarded some important behaviors of the implementation which are not directly related to some concrete feature, but rather affect the entire system's usability. Such a behavior is the BlockFiles structure of the sample sequences in track which allow more efficient operations.

Another behavior is the cached sequences of samples with reduced sampling rate for different zoom levels. Although these behaviors are related to the system's performance (the first one) or to the representation (the second), but they affecting the entire system's design.

# Future Work

The above results giving us directions to continue researching the applicability of the discussed model. We desire to extend the abstract model with additional layers that will represent concrete implementations of the Audacity subsystems such as: History (supporting the undo/redo mechanism), BlockFiles (supporting efficient referencing between track sections and samples), Zoom sampling and others.

 This study is already in progress and being of more research nature it is continued as the topic of a thesis work.

# Appendices

## Appendix A: Audacity features derived from UI

### File Menu

#### Project

##### New (opens another instance of Audacity)

##### Save

##### Save as

##### Save compressed

##### Close (kills current instance of Audacity)

#### Audio

##### Import

##### Export

### Recording Toolbar

#### Microphone level monitor

#### Start

#### Stop

#### Set preferences

##### Style

##### Type

##### Orientation

##### Refresh Rate

### Playback Meter toolbar

#### Playback level monitor

#### Set preferences

##### Style

##### Type

##### Orientation

##### Refresh Rate

### Mixer Toolbar

#### Set recording volume (between 0.0 and 1.0)

#### Set playback volume (between 0.0 and 1.0)

### Edit Toolbar

##### Time-domain track editing options

###### Cut selection

###### Copy selection

###### Paste cut or copied

###### Trim (clean signal except selection)

###### Silence (clean signal selection)

###### Undo action

###### Redo undone action

## Appendix B: Audacity ARS reverse-engineered

### Terms

#### Project - A list of tracks processed together within a global scope of settings

#### Track - A waveform in project

#### Selection - A part from track in time domain

### Functional Requirements

#### Project

##### Settings

###### Allow setting project rate

###### Display actual rate

###### Allow editing “Media Tags” key-value list

##### Tracks

###### Maintain an ordered list of tracks

###### Allow re-ordering tracks

###### Allow importing track from files of different audio formats

###### Allow exporting any track from project into a file of selected audio format

###### Show pane per track

##### History

###### Manage and display a list of all actions

###### Allow undoing actions

###### Allow re-doing undone actions

#### Time domain

##### Display

###### Display a global time ruler.

###### Allow negative range that will not be played

###### During playback/recording display current position (real-time)

###### Allow zoom in and out from along the time scale

##### Selection

###### Allow specifying selected time segment begin by a start position

###### Allow specifying selected time segment end by an end position

###### Allow specifying selected time segment end by a section length

###### Allow snapping a selection of start/end positions to rounded number of seconds

#### Track

##### Settings (allow changing)

###### Name

###### Gain

###### Left-to-Right balance

###### Rate (from list)

###### Type: Mono, Left, Right, Stereo

##### Display

###### Signal waveform in linear amplitude scale

###### Signal waveform in logarithmic amplitude scale (dB)

###### Signal spectrogram

###### Display waveforms/spectrograms for each channel in multi-channel (stereo) tracks

##### Editing

###### Cut selection

###### Copy selection. Allow copy selection from one track to another.

###### Paste cut or copied part of signal into selected point in time

###### Trim: clean signal inside selection

###### Silence: clean signal outside selection

###### Change single sample amplitude

###### Envelope: Allow creating amplitude gradient.

##### Mixing

###### Move track along time dimension.

###### Move all tracks synchronously along time domain.

###### Allow negative start time: section before 0 will not be played.

###### Split multi-channel (stereo) track into multiple single-channel tracks (two mono or left-side and right-side)

###### Combine two single-channel (mono, left or right) tracks into single stereo track

###### Swap between left and right channels in stereo track

###### Convert between mono, left-side and right-sight tracks

###### Set sample data format (16-bit PCM, 24-bit PCM, 32-bit float)

#### Transport

##### Playback

###### Allow playback of one or multiple tracks in project

###### Mute: allow enable/disable tracks for playback. All unmuted tracks will be played during playback or record process

###### Allow pause, resume and stop playback process

###### Allow selection playback

###### Allow playback starting from specified time

###### Allow resetting playback starting time to the beginning

###### Allow moving playback starting time to the end (latest time in the project)

###### Allow playback with specified ration between 0.01 and 3 of the original speed.

##### Recording

###### Allow recording option together with playback. Create a new track for the recorded audio signal

#### Device

##### Selection

###### Allow selecting recording device from the list of installed HW

###### Allow selecting recording channel number (mono/stereo)

###### Allow selecting playback device from the list of installed HW

###### Allow selecting Audio API from the list of installed SW

##### Monitoring

###### Provide monitoring for recording signal

###### Provide monitoring for playback signal

###### Allow start and stop monitoring

###### Display level for left and right channel

###### Display mark on maximum reached level

###### Indicate signal’s saturation

###### Allow specifying signal level meter in RMS or Gradient

###### Allow specifying signal level meter linear or in dB

###### Allow specifying refresh rate between in range [1-100] Hz

## Appendix C: Audacity design impacting features

|  |  |  |  |
| --- | --- | --- | --- |
|  | Feature | Description | Impact |
| 1 | Import | Adds a new Track to an opened project with the samples referenced from a selected source file. | Adds a new "Track" entity into the system's domain model. |
| 2 | Cut | Removes a selected sub-sequence of samples from a track and stores into "clipboard" buffer  | Affects a structure of "Track" entity as well as its sub-entities. Track is the central structure in the system. Also affects the "Clipboard" entity. |
| 3 | Paste | Inserts a sub-sequence of samples stored in "clipboard" starting from a selected position in track | Affects a structure of "Track" entity as well as its sub-entities. Track is the central structure in the system. Also affects the "Clipboard" entity. |
| 4 | Zoom-in | Adjusts a threshold window of a range of samples currently visible in the screen into a sub-range | Affects a structure of "Track" entity as well as its sub-entities. Track is the central structure in the system. |
| 5 | Zoom-out | Adjusts a threshold window of a range of samples currently visible in the screen into a super-range | Affects a structure of "Track" entity as well as its sub-entities. Track is the central structure in the system. |
| 6 | Undo | Performs a rollback of the entire system's state before the last performed action. | The undo/Redo mechanism maintains a stack of historical system state. The Undo and Redo actions affects the entire state of the system by replacing the current state with the one from the stack. |
| 7 | Redo | If the last action was the "Undo" action, performs a rollback of this action, so the action rolled-back by the undo action is disregarded. | The undo/Redo mechanism maintains a stack of historical system state. The Undo and Redo actions affects the entire state of the system by replacing the current state with the one from the stack. |

# References

[1][Improving Design Decomposition](https://link.springer.com/article/10.1007/s00165-017-0428-0) *by* *prof. Shmuel Tyszberowicz and dr. David Faitelson*

[2][The Architecture of Open Source Applications](http://aosabook.org/en/index.html)
*by Amy Brown (Author), Greg Wilson (Contributor)*

[3][Alloy: a language & tool for relational models](http://alloy.mit.edu/alloy/)
*official site of Alloy language on MIT*

[4]The Z Notation: A reference manual
*by J. M. Spivey, Programming Research Group, University of Oxford*

[5][Alloy: A Lightweight Object Modelling Notation](https://pdfs.semanticscholar.org/8260/082839dad0b445c4d2de1e33e7958ed8e3f1.pdf)

*Daniel Jackson, laboratory for Computer Science, Massachusetts Institute of Technology, November 27, 2001*

[5] **Abstract State Machines, B and Z**
*by Egon Börger Michael Butler Jonathan P. Bowen Paul Boca (Eds.)*

[6] **A Scenario-Based Validation Language for ASMs***by A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra*