Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

BACHELOR'S DEGREE IN COMPUTER ENGINEEERING

Refactoring and Updating the Psychoacoustics Web Toolbox

Supervisor: Prof. Mauro Migliardi

Candidate: Pietro Santolin

Co-supervisor: Prof. Massimo Grassi

ACADEMIC YEAR 2024–2025

Graduation date 17/11/2025

Index

1. Introduction 4
2. The original program 5
2.1. A comprehensive description 5
2.1.1 Users and Guests 5
2.1.2 Referral links 5
2.1.3 Taking a Test 6
2.1.4 Taking a Referral test 6
2.2. Codebase 7
2.3. Original Database 8
2.3.1 Database Concerns 9
3. Refactoring Process 11
3.1. A few words on Legacy Code 11
3.2. Code Dive 11
3.2.1. Documentation 12
3.2.2. Naming rules 12
3.2.3 Files Structure and bad practices 12
3.2.4 Functions and modules migration 14
3.3 Error handling 16
3.4 Improvements and Fixes 16
3.4.1 DeviceInfo field 16
3.4.2 Download issues 17
3.4.3 Anticipated test saving 17
4. New functionalities and improvements 18
4.1 Delete Button 18
4.2.2 Issues with the previous Implementation 18
4.2.1 Implementation and Design 19
4.3 “Save data” checkbox 19
4.4 Download Results with CSV File. 20
4.5 “Save Settings checkbox” 21
5 Future Improvements 22
5.1 Database retouch 22

5.2 OOP Approach 23
5.2.1 User Class 23
5.2.2 Account Class 25
5.2.3 Test Object 26
5.2.4 Code Example 28
5.3 Concluding Remarks 28
Bibliography 29

1. Introduction

This thesis originates from a request by Professor Massimo Grassi from the Department of Psychology to fix and improve a toolbox he designed for running psychoacoustic tests called Psychoacoustics-web (Grassi, 2023).
The toolbox, available for free at the following domain:
https://psychoacoustics.dpg.psy.unipd.it
allows users to perform highly detailed auditory experiments for research purposes using the weighted N-down, one-up threshold method suggested by Levitt (JASA, 1971).

The image shows the user interface displayed by the toolbox during the execution of a test, in this case, a tone intensity test.

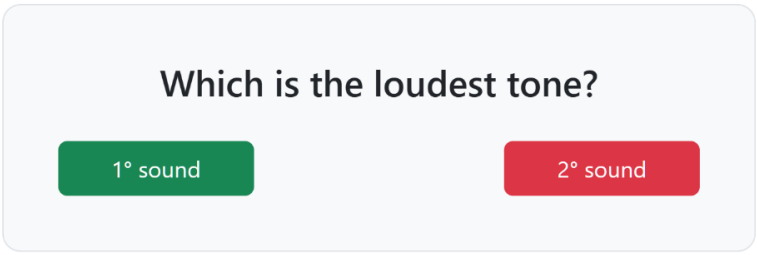

Figure 1. Test dashboard

The interactive interface presents two different sounds, each associated with one of the two buttons. The participant must respond according to the question shown on screen, which varies depending on the type of test being conducted. As the test progresses, the presented sounds gradually decrease in difference until the sequence of required errors is reached, at which point the test ends and the results are displayed.
The toolbox, previously developed by other students, was a functioning product, but not operating at its full potential. While it met the core functional requirements, several defined features remained in an early or incomplete stage of development. Professor Grassi therefore requested software support to refine the program, which is the reason why this thesis was developed: to address the existing issues (also described in Chapter 2) and allow the toolbox to operate according to its original vision.
The development process involved close collaboration between myself and Professor Grassi, which led not only to the resolution of the reported issues but also to the addition of new features that we designed and conceptualized together (Chapter 4). In the last chapter I will also present my personal solutions for the database and software architecture.

2. The original program

In this chapter I will describe the original version of the program: Its functionality, most of which will be maintained through all its further development, the codebase and the database with some of its logical problems, that will be addressed in the later chapter.

2.1. A comprehensive description

Here I will try to outline, as effectively and concisely as possible, the behavior of the original Toolbox and the structure of its database, to ensure a correct understanding of the future analyses that will be carried out in the following chapters. All the behaviours described here will also be present in the final version of the program, unless they are explicitly modified or reworked in the upcoming sections.

2.1.1 Users and Guests

The Toolbox allows the visitors to take the test in two different ways: as a Guest or as a User. When taken as a Guest, no registration is needed, but the site will ask to insert some mandatory demographic data. Guests will only have the chance to download a .csv file with a brief recap of the freshly taken test. If a User is created, all tests taken when logged will be saved in a personal dashboard. Logged Users also have the possibility to download a fully detailed .csv summary of all the tests completed.

If the user is not logged in, the navbar will show -***Sign Up***- and -***Log In***- buttons.
If a user is logged the buttons: -***your tests***-, -***settings***- and -***logout***- will appear.

2.1.2 Referral links

Logged Users have the possibility to create Experiment tests in the *settings* dedicated dashboard. Those tests will have a personalized link called R*eferral link,* different for every User and changeable at will. Users can distribute these links to allow others to take his pre-baked experiment. When another participant accesses the site from this link, s/he will perform the personalized tests as commissioned by the Referrer User. If an already Logged User tries to paste a referral link, the site will force him/her to take the test as a Guest, thus requiring inserting the demographics details (gender and age). Every test taken with a referral link will be linked to the corresponding referrer User. All User's commissioned tests can be visible through a dedicated dashboard and can be downloaded for study intents.

2.1.3 Taking a Test

This paragraph will briefly explain all the steps the program takes when a user decides to perform a test.

When first entering the site, the visitor can choose between six different kinds of auditory tests. By clicking on one of them, the site shows the user a series of forms where s/he is asked to insert the demographic data (*demographicData.php*). A checkbox named “save data” can be unticked to prevent the program from saving any of the user’s data. That means that, when performing the test as a Guest, even if some demographics are provided, they won’t be saved in the database. The name insertion is still mandatory to display the Guest’s name later in the test, and it’s the least amount of demographic that the program will accept. When clicking the “NEXT” button, the program will save the freshly created user in the database, if needed. Performing the test as a logged User will avoid this, preventing the creation of any new guests.

Next in the line is the *soundsSettings.php* page. Here are displayed all the parameters needed for the specific test to start. All the parameters can be modified respecting their preconfigured bounds, otherwise the test is not allowed to start. Some of these parameters are unique for the selected test, most of them are shared across all six of them. The nature and use of these parameters will not be discussed in this thesis, since they are strictly connected to the research side of the toolbox, which this analysis deviates. More information can be found in the linked paper. Worth of note is the “save settings” checkbox present in this page. When checked, the program will remember the set variables and will repropose them, if possible, for each one of the six tests. This function will have some repercussion that will be discussed later.

When clicking “START”, before showing the quiz dashboard, the program immediately saves an empty test in the database. From now on an empty test is a test entry without any result data. The test execution core is entitled to some dedicated JavaScript files, one per test type. When the test ends, the results are sent via URL to the saveData.php page, before being shown in the results.php, ending the test. The taken test is saved by retrieving the empty test saved before and updating his result field.

2.1.4 Taking a Referral test

A User or Guest can take a referral test by pasting the given referral link in the browser. A generic given referral will look like this

https://psychoacoustics.dpg.psy.unipd.it/demographicData.php?ref=UGlldHJv

The link immediately redirects to the *demographicData* forms, bypassing the home page. Every valid referral link carries with it a unique base64 string, encoded by the username of the referrer, hence unique. It becomes obvious by the latter that every User can only have one referral test active at a given time. The referral link is verified by clicking “NEXT”, if the user identifying string is invalid it will display an error message. The guest is presented with an information page that will illustrate to him how the test will be performed, providing a quick demo of it. The parameters and settings needed to run the test are directly fetched from the test linked by the referrer, more on this later. From now on the test procedure is identical to that illustrated in the previous paragraph.

For the complete description of the original toolbox, I refer to the thesis describing the original project. (Felline, 2022)

2.2. Codebase

Most of the program is written in plain PHP language, used both as a frontend and backend. Some JavaScript is present in the frontend section, but most of it is used to deliver an interactive user interface for the test execution. The Bootstrap framework is used through all the frontend sections to build a webpage easy to navigate and maintain.

the program’s files are simply grouped in 3 main clusters:

- Php Frontend files: those in the outer folder layer, along with the index.php page. There resides all the “view” sections of the site.
- Php Backend files : contained in the “php” folder. The latter contains all pure php files, used to perform all the business logic of the site. Note that the Frontend files are not exempt from Php code and database interactions, but they’re only used to extract data to show on screen.
- Javascript files: contained in the “js” folder, those files are responsible for sound generation and test algorithms.

Both in the JavaScript and PHP logic, every test has a dedicated page.

Although the structure will later be revamped to allow for more efficient code reuse, the three main groupings just described will remain largely unchanged. This decision is due to the relatively small magnitude of the project (although it certainly presents some grade of complexity) and its procedural nature, which can be easily accommodated in this kind of folder structure.

2.3. Original Database

I will now the original Database scheme, useful to underline some critical design decisions that highlights the travailed development of this toolbox. The Database is composed of 3 main schemes: *account*, *guest* and *test (Fig.2).* I will take some time to discuss these 3 tables, as they are part of the reason for some confused and unavoidable design choices, other than being a fundamental part of the program.

Users are described by the *Guest* table. The table's only key is ID, which is an incremental integer. Every visitor who wants to take a test, will create a new entry in this table, unless they specifically flag for anonymous tests.

The *account* table extends the *Guest* table by its "ID", allowing for an account creation. Here are specified essential fields like "Username" and "Password". The "Username" field, which must be unique for every User, is chosen to be the primary key. Accounts have another unique feature, such as the possibility to have a referral test linked to them, as described in the previous paragraphs. The "fk_GuestTest" and "fk_TestCount" allow the user to always have a referral test linked to him; again, a referral test is used to allow external Guests to take experiments for the referrer account.

The *test* is used to store all the tests taken by Users and Guest, complete with all the settings used and the relative results. Every test is identified by "Guest_ID" and "Test_count". The first is a foreign key to the "ID" of a guest, the second is an incremental integer that contributes to creating a composite primary key.

Figure 2. The original database tables

To ensure that every user created to perform a referral test can be linked to the commissioner, an “fk_Guest” field has been added to the *Guest* table.

2.3.1 Database Concerns

I will now highlight some problems related to this specific Database structure.

It’s important to note that all the following issues and concerns have not been fixed nor retouched, since that meant spending too much time on rewriting critical code. The logic issues of this database emerged halfway through the refactoring process, given the lack of documentation, and I decided to keep it as it is, since the program commissioner Prof. Massimo Grassi, did not request any functionality that could not be done with this Database structure. However, I strongly recommend taking into consideration a correction of this, with a consequential adaptation of the program, not to mention a complete rewrite of the codebase

with an OOP approach if the case the toolbox were to be expanded in magnitude and functionalities.

First of all, I can immediately note that, in the *account* table, the “fk_GuestTest” and the “Guest_ID” foreign key are inevitably the same. Since every account is related to one and only ID, and the referral test is uniquely identified by the corresponding *guest* id, it becomes obvious that the first field is redundant. To identify the User referral test the “Guest_ID” and “fk_testCount” are enough.
Second, I said that every referral is identified by a referral link, containing a Base64 string; I will refer to this string as *Referral-code* (paragraph 2.1.2). That referral-code is encoded starting from the account Username. This implies that the code has a biunivocal bound with the username. Perplexity appears when I notice that the personal *Referral-code* can be changed in the user setting section, substituted with another randomized Base64 string. This behavior uplifts some concerns, since changing only the *referral-code*, keeping the linked test unchanged, appears a confused and doubtful utility, other than an over-engineered stretch.
Third, I can now understand why a logged user must create a new Guest every time they receive a referral link: every account is linked to a *Guest* account, a guest account can only be linked to a single referrer. That means, that if I try to use the guest our account is linked to, to perform a referral test, that account will be forever bound to that first referral test taken. This problem arises since the “fk_guest” attribute is curiously placed inside a table describing users instead of the one describing tests. A test table provided by both the taker field and the referrer could easily solve this design problem, granting a 1:N bound for the guests toward the test and allowing logged users to take multiple referral tests without being forced to create another *guest* entry. I will try to propose a more functional and solid designed solution for this database in the last chapter.

3. Refactoring Process

I can now start talking about the main arguments of this thesis, one of which is how I structured the refactoring of the described program. This chapter will focus on the analysis of the original code and the correction of the toolbox existing issues, which are the main reasons of the entire refactoring process.

3.1. A few words on Legacy Code

The original code, which will now be referred to as Legacy Code, can still be seen on the original GitHub repository https://github.com/saintpie01/psychoacoustics-web . I was asked to refactor the code as it presented clues of instability and inexperience due to lack of interaction between team members. The main highlighted problems were the extremely high redundancy of the code and the almost complete absence of documentation. The code passed many different hands through its development, testimony of this is the striking difference between code styles and approach that can be noticed among the entire codebase. The prevailing sensation that emerges is that the toolbox was assembled in a “ship-it-now” philosophy, probably due to the lack of time caused by academic circumstances. Time is also the cause of poor *requirements analysis* and absence of *design phase,* which could have helped create a more cohesive and useful toolbox from the start (see chapter 2.2), other than granting a futureproof structure that every project of this magnitude should take into consideration.

It is known that the program has been developed with an Agile approach, *Docker* was used as a development tool, and no debugger was utilized (deducted from the multiple Print statement found in the code). When the legacy code was first downloaded from the original repository and set up in my machine, it had a leftover error, due to a last-ditch effort in development that caused the program not to function properly. When inserting the test setting, and saving the empty test, two identical queries, with the same primary key were performed one after the other.

3.2. Code Dive

I will now discuss the refactoring journey, trying to cover, without deepening into excessive details, all the crucial problematic points found in the legacy code. All the solutions proposed are absolutely not to be considered perfect, knowing there isn’t any “silver bullet” solution in coding. I will, however, justify my decisions and eventually warn if some implementations

still need reworks. All the refactoring was carried on trying to follow at best the PHP coding standard (https://www.php-fig.org/psr/psr-2/)

3.2.1. Documentation

Looking at the code, is immediately noticeable the presence of little to no documentation. Only a few files were correctly and meticulously commented, even though part of the comments were in mixed language (English – Italian). This made the struggle of understanding the program way harder, having to spend a lot of time trying to understand code decisions. For this reason, all the files were commented using the *PHPDoc* standard, to deliver a smooth understanding of the code for future developers.

3.2.2. Naming rules

One of the main topics when delivering a solid and easy codebase is correctly nailing the naming of the codebase components, in order to grant a quick understanding of its role and responsibility. Naming in legacy code was ambiguous on multiple degrees. Most of the variable names were too generic and carried too little information. Some *Session* variables (global variables carrying information between multiple pages) in the backend section, were declared only to be forgotten and not used anymore. For this reason, most of the variables have been renamed or moved to a new structure to better emphasize their nature. Here follows an example of variable renaming and restructure

3.2.3 Files Structure and bad practices

A large part of the effort made consisted in unraveling all the PHP code trying to understand the best way to convey a clean and understandable flux of code. Most of which was poorly written and assembled like patchwork. To give some examples, in some files the *include* statement needed to import the database configuration file was not written at the very beginning of the file like you would expect, but inside the very first try catch often coupled with the *session_start()* statement, also included in the try catch.

All these kinds of statements were moved to the very top of the file outside the *try-catch* structure, trying to convey the best common readability practices. This operation was also performed in the frontend files. In fact these kinds of declarations were made after the HTML opening tag, often in the <*head*> section, contributing to creating a dirty codebase. All the PHP logic code used in the Frontend files has been extracted and positioned right at the top of it, to allow an easy and immediate understanding of the page core code, when present.

Another key point to notice is the rough positioning of the *try-catch* just talked about. When present, those structures were used to nest almost all the code in the file. This type of practice is not inherently wrong, but it reduces readability in the code, contributing to its carelessness appearance. In addition, the error that triggered the block was not displayed nor saved.

3.2.3.1 Conditions and Lazy Evaluations

One of the most hindering factors for code readability and, consequently, maintainability is the completely superficial and inexperienced use of *if-else* conditions. A common practice used when writing clean code is avoiding nesting too many of these structures; this is not a mandatory practice but barely a guideline. Problems arise when looking at the core logic structure of most of the toolbox's backend files.

Figure 3. Nested conditions in the original code

First thing I notice is the absence of *exit statements* after every *header()* action. In PHP the *header(url)* function, used to redirect to another page given a URL, does not act like a return statement, thus all the lines of code after this function will be executed unless an *exit* instruction is placed right after. This oversight forces the code to include an *else* condition right after to avoid the execution of the following code, adding to the indentation degree.

The image (Fig. 3) immediately shows the direct consequences of this code behavior in a phenomenon called *Pyramid of Doom:* the deepest areas of the code are completely inaccessible to maintainability due to the high number of conditions accumulated to get there. Taking into consideration the worst example: *personalInfoValidation.php,* I can count up to 7 degrees of conditional indentation. Debugging in this kind of environment can be really frustrating and a refactoring is mandatory. Indentation is not a problem per se, but in this particular case can be easily avoided by applying some measurements.

- **Exit Statements**

  As mentioned earlier, redirecting to another page does not stop the code execution. This forces the user to insert the following code in an *else block*, adding another degree of indentation propagated to the end. A simple and effective measure consists in adding an *exit* instruction, as hinted earlier. This will entirely prevent any code to be executed after that, making the else block unnecessary and removable.

- **Condition inversion**

Another contributing factor is the poor logic of these conditionals. To better understand the problem, presented in multiple instances throughout the code I can make a simple example.

```php
if ($saveData) {
    // Process data
    // Save into Database
} else {
    // No elaboration needed
    header("Location: test.php");
    exit;
}

```

Figure 4.

```php
if (!$saveData) {
    // No elaboration needed
    header("Location: test.php");
    exit;
}
// Process data
// Save into Database


```

Figure 5.

Those two snippets of code represent a dummy case for our case, for example, I want to know if I have to create and save a new user or save a freshly taken test based on the "saveData" variable: true if the user agreed to save his data, false otherwise. In the first example, I can notice that if the variable indicates that the data needs to be saved, the entire portion of code dedicated to processing and saving these is enclosed in a block of parentheses. At the end of these, the code for the false condition is present. This choice forces a large and usually complex portion of code to be enclosed into parenthesis no matter what, compromising the readability. In the second example (fig. 5) I show a better version of this evaluation: I immediately check that the condition is false to exit the program right away. This way, the subsequent code does not need indentation, as I have already filtered the conditions beforehand. This is called '*early exit pattern'* and I inserted it into the code replacing and adjusting the previous code.

Finally, let's turn our attention to the use of *lazy evaluation*. Looking at the old code, I can sometimes notice portions of code that get executed **before** knowing if the elaboration will be used or not. This carelessness sometimes happens with complex portions of codes and even Database queries, burdening the program by performing operations that could be easily avoided. I refactored the code positioning these complex portions of code right before they're needed, being sure no overheads are created.

3.2.4 Functions and modules migration

One of the first maintainability topics is an efficient creation of modular code, in an attempt to reuse most of what was already written. This concept helps us reuse code under the form of functions, which turn out to be extremely useful when maintaining the codebase, allowing us

to modify or replace the outdated code by only substituting one module. Scrolling through the program I can immediately observe how many portions of code had been copy-pasted through the entire program, making the codebase heavier, inelegant and inefficient. In fact, not a single function was created in the entire codebase (exception for the download-handler file which included two). To give some basic yet explanatory example I can observe a specific block of code, used to connect to the SQL database.

```
$conn = new mysqli($host, $user, $password, $dbname);
if ($conn->connect_errno)
    throw new Exception('DB connection failed');
mysqli_set_charset($conn, "utf8");
```

Figure 6. Block of code used to open a connection with the Database

Before the refactoring this was the most repeated section of code in the codebase, sometimes inserted multiple times in the same file. I exported this function in a different one, allowing the creation of the desired *mysqli* object with a connectdb() call (no parameters needed, assuming they are already stated in the config.php file).

Another example, this time more problematic, can be found in the test insertion queries. These queries were used in two distinct situations: creating a referral test or saving a test we just took. Given the nature of the tests, the original program used three different queries to front the different parameters needed for each one. A small change to the insertion query meant having to modify up to six different portions of code. To front this problem, I added a universal method of insertion. the creation of a test can now be performed with the following function:

```
insertTest($id, $count, $referralName, $testTypeCmp,
$param, $results, $score, $geometricScore, $conn)
```

In case of referral test the results parameters will be omitted and in case of tests taken by users we omit the $referralName parameter. This function is the result of multiple steps of data sanitation, to allow the insertion of the parameters ($param array) without creating any conflict. Note that in a Clean Code optic, this function has way too many parameters, but they're needed to deliver enough granularity and control in these SQL insertions since no OOP is used.

Note: When working with queries in PHP, it is essential to always use **prepared statements** to enhance security and prevent SQL injection vulnerabilities. During the code refactoring I

maintained the original SQL methodology due to time constraints. It is important that this point is taken into account in future versions of the program.

Other functions were created to improve code readability. For example, a recurrent operation is checking if a user is logged or not. The operation was performed with this line of code

```
if (isset($_SESSION['usr']))
```

In addition to the ambiguous variable name 'usr', which was later changed, the code is overly verbose and difficult to read. For this reason, it was replaced with this function.

```
if (isUserLogged())
```

All the exported functions were organized into two separate files: one for functions requiring Database interaction (*database_functions.php*) and another for those that did not (*helpers.php*). These files collect more than 10 different functions used in the entire code. In the view code, migrations have been made to unify the common and reusable HTML modules across pages, such as the navbar. The modules were exported in a dedicated */modules* folder

3.3 Error handling

Debugging the code requires a clear understanding of the returned errors, the original code did not have any fast method (outside looking at the *php_error_log* file in the *Apache* server, who catches *all* the server errors) to log the errors of the backend PHP pages, since they can only be written inside a HTML tag. To solve this, I created an error log file which catches all the errors and warnings returned by the executed code. The file is located in a */log/errors* folder inside the root, and it is particularly useful to monitor all the possible errors that can occur in production. I reserved particular care in correcting all the errors and most of the recurring warnings, but only an intensive use of the toolbox can let rare errors emerge. This solution also lets the server admins access the logs remotely from the browser.

3.4 Improvements and Fixes

I will now list some functionalities that had been changed or fixed, compared to the original toolbox. This will not include the added functionalities, which will be discussed in the dedicated chapter.

3.4.1 DeviceInfo field

The original Database included a not null field in the Test table called *DeviceInfo*. This field stored the device's information gathered by the program. Since this field could not be NULL,

the program crashed every time a query was performed without filling it. This happened often during the initial testing, leading to the deduction that this was one of the main reasons for the program instability. The field was converted allowing it to store the NULL value, fixing further unexpected behaviours.

3.4.2 Download issues

The original download function had an issue related to server permission. When running the program in the server, the files responsible for the download and file creation, need to have the correct write privileges. The latter weren't conceded in the original toolbox, so no file could be created nor downloaded. When running the program in a local machine it all worked as intended.

Another subtle issue was related to the creation of the file. When clicking the download button, the program creates a new csv file with a generic fixed name (*download.csv*), it then fills it with the proper data, sends it to the user and deletes it. This procedure is extremely dangerous. It can happen that two users click the download button at the same exact time, the file gets filled with both user's data and returned to just one of them. This happens because the program creates only a single file to work with, with the hopeful assumption that no user will use it at the same time. To correct this, I added the user ID to the file at creation time (eg. User #392 will work on a file called *download392.csv*), this way there can not be any conflict, every user will work on a unique dedicated file. A flush instruction was also added to prevent the presence of unwanted characters at the start of the document.

3.4.3 Anticipated test saving

As I already anticipated in the program description in Chapter 2, every time the parameters of a test were set and saved, before executing it, the empty test was stored in the database. Later, once the test was completed, the empty test was retrieved and filled with the result. I know that even if the test parameters were saved in the record, the program still stored the test parameters in SESSION variables to pass them later to the JavaScript code (used to reproduce sounds and the main experiment logic). Furthermore, if a user interrupted the experiment, the empty test would still remain in the database, thus crowding the user dashboard. To solve this illogical behavior, the preemptive saving of the test was removed. Now, the record insertion is carried out exclusively at the end of the experiment, prior approval for data saving. The test settings used to perform the experiment are now saved in a dedicated SESSION array, to provide a more organized variable environment.

4. New functionalities and improvements

4.1 Delete Button

A strongly requested feature was the ‘Delete’ button in the ‘Your Tests’ area. Before the addition of this button, there was no way to remove tests accumulated in the personal dashboard, including referral tests (which previously appeared in the dashboard; see Chapter 3) and incomplete tests. The only available method was to access the phpMyAdmin database dashboard and manually delete the test, an operation that was overly technical and time-consuming. The introduction of the ‘Delete’ button now allows users to permanently remove a selected test, thus overcoming this limitation.

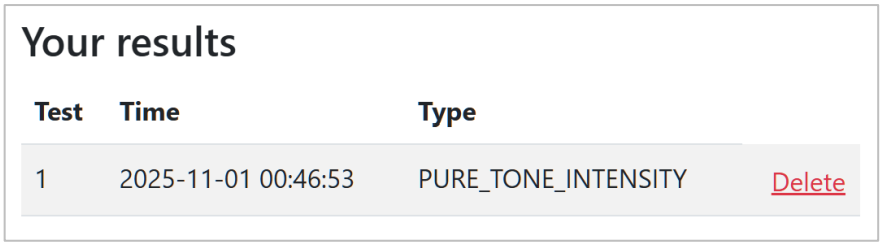

Figure 7

4.2 ‘Your Experiment’ Section

As mentioned in the previous chapters, each user could have only one active referral test at a time. Creating a new test would delete the previous one. The final solution was to allow users to save multiple referral tests in the database, which could then be marked as ‘Active’ through a dedicated button. On this occasion, the label Referral test within the toolbox was replaced with Experiment to make the functionality more immediately understandable.

4.2.2 Issues with the previous Implementation

Previously, each referral test created was stored as an empty test, and it was linked to the User table through the fields “Username” and “fk_TestCount” (its primary key). In this way, every time the user referral changed, the previous one associated with that specific User remained in the database but were no longer accessible This caused unnecessary clutter in the tables, in addition, the user was essentially forced to create a new experiment every time the active one was no longer needed, since there was no way to save them for future uses. Moreover, there was no way to view the settings of the currently active referral test except by entering the Change Settings page and looking at the data automatically loaded.

4.2.1 Implementation and Design

In the Settings section, directly below the panel for creating a new experiment, a box titled Your Experiment was introduced. Each time a new experiment is created, it will be listed here in a dedicated tab.

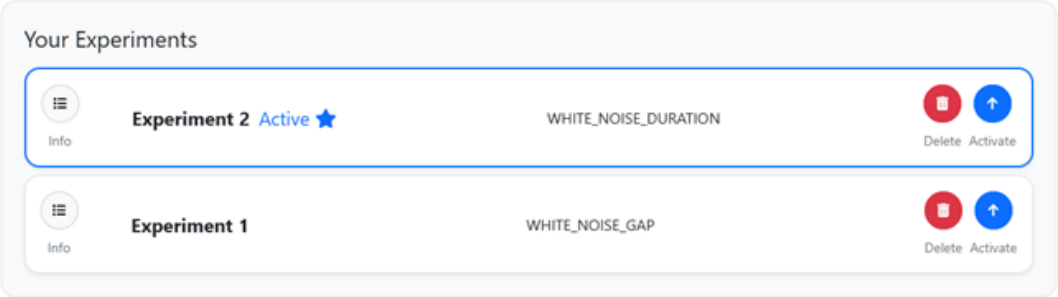

Figure 8

As shown in the figure, it is now possible to assign a name to each experiment, which is directly visible from the dashboard. The tab also includes a Delete button, allowing the experiment to be removed from the database, and an Activate button, which sets the selected experiment as the active one by updating the "fk_TestCount" field in the User table.

When an experiment is active, it means that each user's personal referral link (which, as mentioned earlier, is unique) will lead to the execution of that selected experiment. By activating another experiment, the referred user can reuse the same link to participate in the new one. By implementing this new functionality, the previous Change Settings button was removed, since its primary purpose was to display the experiment settings. In its place, a more convenient Info button was added, which opens a popup window listing all the settings of the selected experiment. On this occasion, the PHP file responsible for handling settings changes was also removed, as it was no longer in use. On this occasion, the "ref_name*"* field was added to the *test* table. With this implementation, each user can now access all the tests associated with their username in the database, with the ability to delete them and view their results.

4.3 "Save data" checkbox

As stated in chapter 2.1.3, the "save data" checkbox allowed the test participants to block the site from saving *any* data created by them. In the original program, even when the checkbox was deactivated, the textboxes for entering personal data did not disappear from the interface, forcing the user to enter their name in order to continue.

In the current version of the program, once the checkbox is off, the textboxes for entering personal data disappear, allowing the participant to proceed without actually entering any

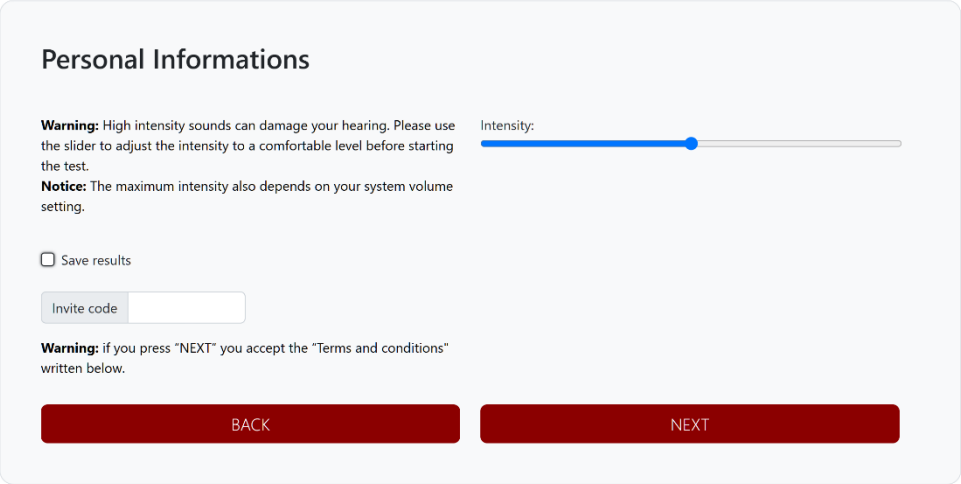

Figure 9. No entry fields appears when "Save results" is unselected

information.

4.4 Download Results with CSV File.

The original program displayed the test score only in the browser once completed, but it was not explicitly stored in the database. The final score was calculated by a JavaScript file through the analysis of the Results string. Although this string was saved in a dedicated field within the Test table, it was difficult to interpret and did not allow for an immediate understanding of the outcome. At the request of Professor Grassi, the fields "Score" and "GeometricScore" were added to the Test table, populated with the readable data returned by the JavaScript calculation. This data can therefore be easily accessed once the CSV file is downloaded. This was the only major database-level modification implemented, as it did not interfere with the core functionality of the program nor with the related SQL insert and update queries. However, it did require a reset of the existing client databases to ensure the correct functioning of the new program.

Let's go over all the changes I made to the original schema. First of all, it is worth noting that the *Tests* table now has its own unique key, instead of the previous composite key made up of *ID* and a progressive number, which in my opinion was quite fragile.

In the current implementation, it is necessary to count the number of tests already associated with a user in order to determine the next progressive value, an operation that could instead be replaced by a randomly generated or hash-based identifier. Secondly, the name of the table

has been slightly changed to better reflect the nature of its contents. The database logic still revolves around the *Guest* table, now renamed *User*, which contains the ID of every test-taker. The Test table now contains one more foreign key: *referrer_id.* This field allows every Account to take an experiment without having to switch to a Guest account, because the referrer information will be stored directly in the Test record. This method still preserves all the meaningful information about the test just taken, like who was performed and commissioned by, allowing all the users with a personal account to access, or just have a glimpse of all the tests taken, including the experiments.

4.5 “Save Settings checkbox”

As mentioned in paragraph 2.1.3, the “Save Settings” checkbox, available for logged-in users, allowed the parameters of the most recently executed test to be saved and automatically reloaded in the settings screen when attempting another test.

The issue with this feature was that the parameters were “saved” and “retrieved” using the user’s referral test as storage. In practice, when the checkbox was selected on the *saveSettings.php* page, a test with the chosen parameters was created, and its key was inserted into the *account* table using the fields intended for the referral test.

This ambiguous behavior raised serious doubts about the usefulness and consistency of the feature, which is why it was removed.

5 Future Improvements

At the current state, the program is fully functional and does not require further adjustments. In this chapter, however, I will attempt to propose my own solution with an object-oriented approach. I will also suggest a database modification that could improve the overall functioning of the suite.

5.1 Database retouch

I return to an issue discussed in Chapter 2 concerning the creation of a new temporary user each time a referral test is performed. As mentioned earlier, this problem arises from the fact that the referral ID (the code of the user who commissioned the experiment) is stored directly in the *Guest* table. As a result, an experiment can be linked to one, and only one, guest. It would be more useful and flexible if, instead of having to create a new user, a logged-in user could perform a test directly using their own account. The following simplified diagram (Fig. 10) illustrates a possible solution to this problem. To keep the page uncluttered, only the relevant fields are shown.

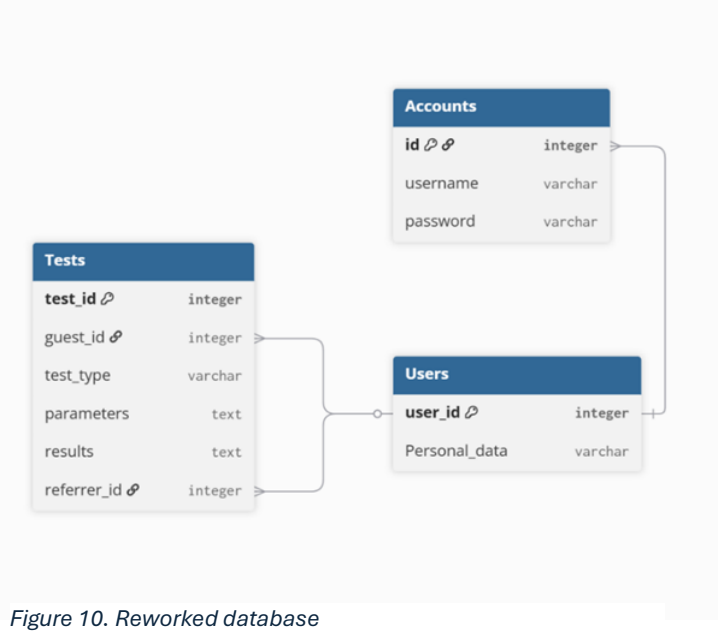

Figure 10. Reworked database

The referral ID, inside the Account table has been removed. The main reason is that the existing method only allows for one test to be active for each account (Chap. 2.1.4), with this change, I completely removed this forced constraint. The main idea is that I can easily refer to a single test using its new ID, this operation was previously inconvenient since it had a composite key. With a simple query I can easily display all the experiments created by a

single user in his dashboard, making it possible share multiple tests simultaneously, the generic new experiment link would look like this:

https://psychoacoustics.dpg.psy.unipd.it/demographicData.php?ref=583247

hence, removing the Base64 username and directly addressing the test. Exposing the test ID does not bring any security concerns since it does not carry any meaningful information with it. A minor detail, also present in the current version of the database, is that once a user activates an experiment, they can only deactivate it either by deleting it or by activating another experiment, which then becomes active in its place. This problem can be mitigated by designing a new Tests table (Fig. 11)

With this table, I introduced a new boolean attribute called "*active_flag*". This handy addition makes it possible to determine whether an experiment ID contained in a link can still be used to perform an experiment or not, thus preventing users from taking unwanted experiments without the need to delete or overwrite them. One last possible improvement would be to replace "*referrer_id*" with "*experiment_id*". This not only allows us to know who commissioned the test, but also which experiment it belongs to. This turns out to be particularly useful when a researcher wants to quickly identify all test instances associated with a specific experiment, without having to perform lengthy and resource-consuming data analysis.

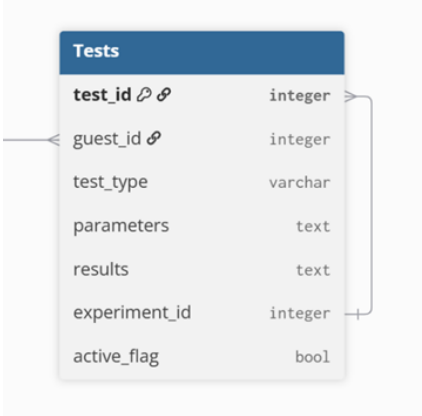

Figure 11. Reworked Test table

5.2 OOP Approach

This last section aims to outline an object-oriented approach to the program, suggesting interfaces that could be useful for a similar project or for a future update of this one.

At the moment, the toolbox does not require these changes, but they could be considered if the project were to be expanded in the future. All examples provided will be written in PHP 8.0, as it is the main language used in the project. Class names and method arguments will follow the proposed database schema, and all getters and setters will be omitted for simplicity.

5.2.1 User Class

The **User** class (old Guest) will be the pivot for our approach, since all the other elements (Tests and Accounts) depend on its presence. Here follows its interface declaration

```php
interface UserInterface {
    public function __construct(?int $id = null, ?string $personalData = null);
    public function register(): int;
    public function updateUser(): void;
    public function saveTest(?array $results = []): int;
    public function performTest(?int $experimentID = null): void;
    public function newTest(Test $test): void;

}
```

The constructor allows either passing an existing ID to retrieve an already registered user or passing parameters to create a new one. This hybrid constructor design is due to the fact that PHP does not support operator overloading, so both cases have been combined into a single constructor with optional parameters.

If the constructor detects a valid *$id* parameter, it knows that an existing **User** is being requested, so it ignores the other parameters and builds the **User** object by loading data from the database. Once the object is populated, if it represents a new user, it can be inserted into the database with the `register()` method. The method returns an int value, which is the ID of the freshly created record. To prevent the same user being inserted more than once, the method must provide a way to check if the *$id* is already initialized, that means that the object already represents an existing User and therefore cannot insert itself again into the DB. If that were to happen, the method would manually raise an `Exception`. In addition to this, I suggest declaring the *$id* as a `readonly` attribute, this prevents it from being modified once initialized, ensuring that every **User** object created is related to one and only one User record.

This approach relies on the fact that the *$id* returned is provided by the database, but it can still be used if it's decided to be calculated in an alternative way. In the latter case, however, the `register()` method would automatically throw an exception, since it would attempt to execute an insertion query using an already existing primary key.

The interface also provides two other methods used to perform and save a test that will be analyzed after illustrating the **Test** object.

5.2.2 Account Class

To describe the **Account** class, I need to show a simplified class implementation

```php
class Account {
    private static ?Account $instance = null;

    private int $id;
    private string $username
    private string $password;

    private function __construct(int $id, string $username, string $password) {}

    public static function create(User $usr, string $username, string $password): Account {
        if (self::$instance != null) return self::$instance;
        if ($usr->getID() == NULL) throw new Exception("User has no valid ID");
        if (self::isValidID($usr->getID()) throw new Exception("Account already existing");
        self::$instance = new Account($id, $username, $password);
        return self::$instance;
    }

    public static function logIn(int $username, string $password): bool {return true;}
    public static function isValidID(int $id): bool {return true;}
    public function changePassword(string $oldPassword, string $newPassword): bool {return true;}
    public function isLogged() : bool {return true;};

}
```

The **Account** class is based on the *Singleton* design pattern (Gamma, Helm, Johnson, & Vlissides, 1995). The constructor is intentionally declared as `private`, and the access points for creating the object are the `create()` and `logIn()` methods. Both methods follow the singleton's logic (only `create()` is implemented in this snippet), ensuring that no more than one **Account** instance can be created within the same session. Both methods are declared as `static` to let the developer access them without having to create the object first. The `create()` method is used to create a new account: It takes *$username, $password*, and a **User** object as arguments. The `User` object plays a fundamental role in the creation of a new account: As explained in the previous paragraph, a **User** object represents a valid record in the database only when its *$id* attribute is not NULL, meaning that a user has been loaded from an existing ID or a new one has been created and linked to that object. This way, the `Account::create()` method can ensure that the **Account** record being inserted is actually associated with an existing user.

Hence,

```
if ($usr->getID() == NULL){...}
```

this condition can only be valid if the **User** object does not represent an existing user yet.

On the same track, the following statement

`if (self::isValidID($usr->getID()){...}`

uses the static `Account::isValidID()` helper to check if the record **Account** associated with that ID already exists.

The self-explaining `logIn()` method is used to log into an existing account, the returned value is used to verify the success of the operation, this represents an alternative way to instantiate the *Singleton*. This method does not necessarily need to use the `isValidID()` helper since the SELECT query can already tell if the user exists or not by the username and password provided. Finally, the `isLogged()` method now provides an elegant way to check the presence of a Logged Account in the existing session.

5.2.3 Test Object

Lastly, I need an object that represents the test that the user will perform. Given the wide variety of tests, it comes natural to define a common **Test** interface that serves as foundation for creating the classes corresponding to the different tests. This approach allows for standardized and simplified integration of any future tests.

```
interface Test {
    public function __construct(array $params = []);
    public function perform(): void;
    public function save(): bool;
}

class AmplitudeTest implements Test {...}
class FrequencyTest implements Test {...}
```

The raw instantiation of the **Test** class would only need the main parameter of the test, leaving aside all the IDs needed to correctly identify the test-taker. Now I can return to discuss the remaining functions of the **User** class, illustrating how they work. The **User** method

`newTest(Test $test): void;`

allows an instantiated **Test** to be associated with a user. This is done by initializing an internal private variable of type **Test**, called *$currentTest,* which enables the **User** class to manage one test at a time. This operation also attaches the *$userID* to the test, linking it to the user who performed it. The **User** method

`performTest(?int $experimentID = null): void;`

is used to actually start the test, redirecting the test partecipant to the appropriate test page based on the derived class stored in its private **Test** variable. This behaviour serves as an implementation of the *Strategy* pattern, effectively allowing different test behaviours to be selected at runtime through the inherited `Test::perform()` method.
I added an optional parameter *$experimentID* to allow the execution of an experiment test. Since, according to the implementation proposed in Chapter 5.1, each test can be associated with a unique ID, passing it directly to the perform method would enable a direct retrieval of the desired test. Alternatively, a static method could be added within the **Test** class to allow cloning of an existing test in the Database, defined as follows:

```
public static function cloneTest(int $testID) : Test
```

The returned test could then be passed to `User::newTest()`. This method would also automatically strip the "UserID" of the cloned test to provide a clean **Test** object. The internal behavior of this method can be compared to that of a *Factory*, providing a common access point for creating multiple objects that inherit from the same **Test** interface.
The last method

```
saveTest(?array $results = []): int
```

handles the actual saving of the test. Here too, the common interface is used to implement a *strategy* pattern, allowing the correct query to be executed at runtime. The *$results* parameter contains the final test results, which are gathered by a Javascript routine and must be passed externally. Executing it with no parameter would save the test without result, which turns handy to store Experiment test, as stated in chapter 2.
The last detail worth noting is that it is possible, for example, to create a Test object like `$test1` and call `$test1->perform()`. This starts the execution of an anonymous test, since the user ID is only assigned by the **User** object. This behavior is intentional and allows a test to be performed anonymously, so without any insertion into the database, when de-flagging the "Save Results" checkbox.

5.2.4 Code Example

The following code shows an example of implementation, where a participant create a new account and then takes an experiment, using the classes I just declared

```
// User creation
$user = new User($name, $age, $sex);
$user->register();

// Account creation
$account = Account::create($user, $username, $password);

// Experiments gets associated to current user
$test = Test::cloneTest($experimentID);
$user->newTest($test);

$user->performTest();
// user gets redirected to another page
$user->saveTest($results);
```

5.3 Concluding Remarks

The proposed solutions, both for the database restructuring and for the object-oriented implementation, should not be seen as definitive, but rather as a reflection of my own perspective on how such a problem could be addressed. I have tried to keep the class descriptions as concise as possible, in order to focus on their usage methods and to cover, as tightly as possible, all potential use cases of the toolbox. I invite anyone who comes into contact with this project to critically analyze the proposed solution in order to determine whether it can be used as a reference responsibly, taking into account any new requirements that may have emerged.

Bibliography

Felline, A. (2022). *Implementazione web based per strumenti di analisi psicoacustica*.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Prentice Hall.

Grassi, M. C. (2023). PSYCHOACOUSTICS-WEB: a free online tool for the estimation of basic auditory sensitivity thresholds.