Assignment 1:

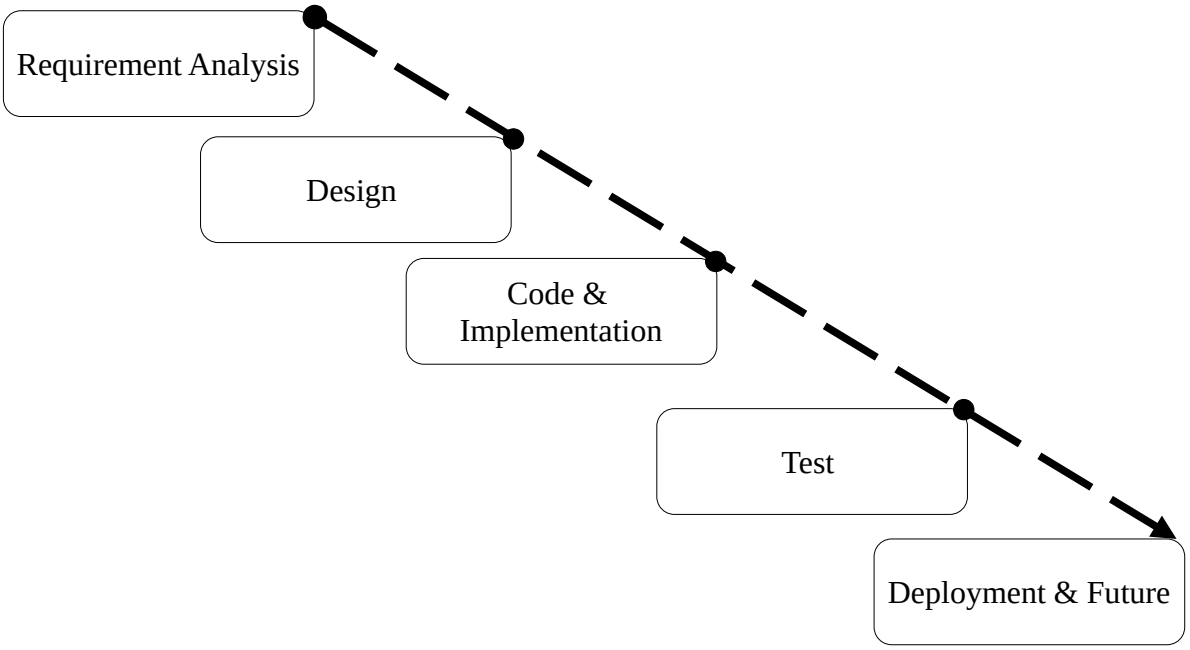
TunePal

Enrique Juan Gamboa
D23125488

INDEX

1. Development planning

The TunePal team is going to use the waterfall methodology to realise their app. This is a linear-style development method in which the team will have to complete everything set in each step of development before continuing to the next. While inflexible and a bit antiquated, done properly this method will give a strong and robust app, with everything for it was initially planned to do tested extensively and in proper working order. Done improperly, and the toll of one mistake done beforehand can ball up into a very big problem for the team and having to go back several steps.
To ensure the former, proper testing will have to be done in each stage of the waterfall methodology.



In the following pages a proposal for proper testing planning is offered for each stage of development in their planning.

i- Requirement Analysis

The requirement analysis phase is crucial as it sets the foundation for the entire development process. It involves gathering, documenting, and analysing the needs and expectations of clients/stakeholders.
Testing here plays a vital role in the phase by making testers take part in requirement review meetings and engaging with stakeholders to get a complete understanding of the project's goals.
This is essential as the methodology being used isn't flexible. This means that any decision taken here is final: any requirement not taken accounted for won't be done and any change in plans will mean a total backtrack.
To help this a Software Requirements Specification document will be redacted during all the meetings. This will ease the process of assuring that everything is accounted for and that a log of what was asked from the application is available for reference, thus creating a guide for the rest of development.
For each requirement from the app, four minimum aspects must be met for it to be accepted into the SRS and developed. Each requirement must be:

1. Proper for the goal, as to not waste time on unnecessary tasks,
2. Doable by the developers, as to not expect the impossible from them,
3. Clear and unambiguous, as to not misinterpret what was asked, and
4. Verifiable, as to determine if the requirement has been met.

Objects and/or inputs needed for testing:

- Requirements given by clients and/or stakeholders in the phase meetings

Products obtained:

- Software Requirements Specification document, which will guide the rest of development

Involvement:

- Clients/Stakeholders
- Testers
- Developers

ii- Design

After the Software Requirements Specification document is finished and endorsed by both the client and the testing team, the design phase begins.
In this phase, the system architect and design team will outline all main parts of the application (front and back-end, databases, UI, etc.) which need designing. Afterwards, each of these components will go under a sequential design process. Once the design for each component is done, the testing team will be able to go ahead and make sure that it complies with what was stated in the SRS document. If they find out it doesn't, they must relay to the architect and design team the faults found. Once made sure that the issues truly are existent, the architect and design team will go ahead and redesign to comply.
Once approved by the testing team, the design costs for each part of the application will be forwarded to the clients. This is imperative as without their approval on every cost, the design cannot go forward.
Once the design and cost for all components has been approved a final meeting to ensure that each part's SRS document compliance and costs is correct, both individually and as a whole. As done before, if one of these aspects aren't correct, the architect and design team will have to redesign to comply.

Objects and/or inputs needed for testing:

- UML diagrams for each part
- Individual part's documentation detailing design, functionality and compliance

Products obtained:

- Design Specification document for use during development and implementation

Involvement:

- Clients
- Testers
- Designers/ System architect

iii- Code & Implementation

In this phase, the system architect and design team's designs are translated into code. Developers begin coding the main components of the application, including the front-end, back-end, databases, and user interface (UI), adhering closely to the approved designs.
The testing team steps in to verify that the implemented code complies with the requirements outlined in the SRS document. Any deviations are communicated back to the development team for rectification.
Iterative refinement is employed, allowing the development team to address identified issues and ensure that the final product aligns with specified requirements and quality standards. For this, a Code Functionality document will be created, where developers will have stated during coding what the intended functionality is and how their code works.
Different code will be developed in different branches of the same source code making use of a version control software such as GitHub. This will allow the iterative design to flourish in multiple places at once, as well as help determine where an error can be found without jeopardizing the rest of the team's progress.
Once the coding for each component is approved by the testing team, the associated development costs are communicated to the clients. Client approval of these costs is essential before proceeding further with the development process.
After approval of the coding and associated costs for all components, a final check ensures that each component complies with the SRS document and approved costs. Any discrepancies are addressed by the development team through necessary adjustments or refinements.

Objects and/or inputs needed for testing:

- UML diagrams for each component.
- Documentation detailing the design, functionality, and compliance of each component.

Products obtained:

- Design Specification document for use during development and implementation.

Involvement:

- Clients
- Testers
- Developers/System architects

iv- Verification

One of the most important steps during development in the waterfall methodology is this one, as it is in this phase that the client's endorsement is fundamental for release. Without their approval, the application won't go on into release, so taking into consideration all their views is essential.
Due to the Software Requirements Specification document, the clients cannot make changes to the requirements. Those were decided in the first phase of development and agreed upon by all parties. Regardless, suggestions must always be taken into account for the betterment of the application during future development cycles.
Small changes can be made before release, such as UX or UI, bus as stated before no big modifications to the additional requirements nor addition to them must be done at this stage.
Once the client is satisfied with the end product, we can go forward with release.

Objects and/or inputs needed for testing:

- The application in it's entirety
- Software Requirements Specification document (for reference)

Products obtained:

- Final suggestions
- Suggestions to be applied in future development cycles
- Final application

Involvement:

- Testers
- Developers
- Clients

v- Deployment & Future

During the deployment and future phase the developers must be very aware to address any identified issues in the application. Once a problem is identified and rectified, the testers must ensure the issue is fully resolved. This means crafting comprehensive unit and component tests to thoroughly scan the fixed code. Integration of the new code with the existing codebase is executed seamlessly to maintain the integrity of the application. All this will be done the same way as in the Code and Implementation phase, which proves the versatility of the Code Functionality document and the GitHub structure there.
If the client requests new features, the approach must depend on the scale of the feature. Large-scale features will probably mean starting a new waterfall lifecycle to ensure thorough planning and execution. Otherwise, smaller features that do not require significant changes can be accommodated through a streamlined process. The developers testers collaborate to initiate a new implementation phase, followed by a verification phase mirroring the established procedures.

Objects and/or inputs needed for testing:

- Latest version of the release application
- Latest version of the application's code
- Identified issues and their specifications
- Planned features and their specifications

Products obtained:

- New version of the application with issues fixed/features implemented

Involvement:

- Testers
- Developers
- Clients

2. System Requirements

Here I have specified what I believe the definition for each requirement was, as well as if I thought if a modification should be done. Afterwards I have created simple testing schemas for each requirement as to demonstrate a simple example of how they should be approached.

i- Definitions

1. *Users will be able to create an account and log in:* This functionality ensures that users can personalize their experience within the TunePal application. By creating an account, users can access their saved preferences, playlists, and other personalized features. The login process allows users to securely access their account whenever they use the app.
   No change is needed in this regard.

2. *Users will have access to a shared library of songs but can also add their own private collection:* This feature provides users with a comprehensive music library experience. They can explore a wide range of songs available in the shared library while also having the freedom to add their favourite tracks from their private collection. This personalization adds value to the user experience, allowing them to curate their own music library within TunePal.
   No change is needed in this regard.

3. *The app should be finished by August:* This timeline sets a clear goal for the development team to work towards. Having said that, the only way to assure this is to have a realistic specification set and to follow a very effective project plan.

4. *The library should be searchable by artist, song title, and year of release:* Search functionality is essential for users to quickly find the music they are looking for. By allowing searches based on artist, song title, and year of release, TunePal enhances user convenience and accessibility, enabling them to discover and enjoy their favourite songs with ease.

I believe that searching by year of release is very niche, and searching by genre would be a lot more appealing to the masses. There is no reason these two search features cannot coexist though.

5. *Users should be allowed to log in on up to 2 devices at a time:* This limitation on concurrent logins ensures fair usage of TunePal and prevents abuse of user accounts. It also encourages users to maintain control over their account security by limiting access to a reasonable number of devices simultaneously.
   No change is needed in this regard, and I believe it will favour a consumer-brand relationship if they are able to share their account with one friend.

6. *Songs should take no longer than 5 seconds to download fully onto the user's device:* Fast download times are crucial for providing a smooth and seamless user experience. By ensuring that songs download within 5 seconds, TunePal minimizes wait times for users, allowing them to quickly access and enjoy their favourite music without delays.
   No change is needed in this regard, as long as varying connection qualities are taken into account for leniency.

7. *Users should be able to access the shop to buy premium titles:* This feature expands the revenue streams for TunePal by offering premium content for purchase. Users can browse and buy premium titles from the shop, enhancing their music collection with exclusive content. The ability to fill a basket before proceeding to checkout streamlines the purchasing process, while the option to empty the basket provides flexibility and control to users during their shopping experience.
   No changes are needed in this regard.

ii- Tests

Test Object: Account Creation and Login

Test Basis: Users should be able to create an account and log in to personalize their experience within TunePal.

1. *Test Case:* Account Creation
    - *Input:* User provides valid email address, password, and other required information.
    - *Expected Outcome:* Account is successfully created, and user receives a confirmation email.
2. *Test Case:* Login
    - *Input:* User enters valid credentials (email and password).
    - *Expected Outcome:* User successfully logs into their account and gains access to personalized features.

Test Object: Song Library Management

Test Basis: Users have access to a shared library of songs and can add their own private collection.

1. *Test Case:* Accessing Shared Library
    - Input: User navigates to the shared library section.
    - *Expected Outcome:* User can view a list of available songs in the shared library.
2. *Test Case:* Adding Songs to Private Collection
    - Input: User selects songs to add to their private collection.
    - *Expected Outcome:* Selected songs are successfully added to the user's private collection.

Test Object: Search Functionality

Test Basis: The library should be searchable by artist, song title, and year of release.

1. *Test Case:* Search by Artist
    - *Input:* User searches for songs by a specific artist.
    - *Expected Outcome:* Songs by the specified artist are displayed in search results.
2. *Test Case:* Search by Song Title
    - *Input:* User searches for songs by a specific title.
    - *Expected Outcome:* The song with the specified title is displayed in search results.
3. *Test Case:* Search by Year of Release
    - *Input:* User searches for songs released in a specific year.
    - *Expected Outcome:* Songs released in the specified year are displayed in search results.
4. *Test Case:* Search by Genre
    - *Input:* User searches for songs released in a specific genre
    - *Expected Outcome:* Songs released in the specified genre are displayed in search results.

Test Object: Device Limitation

Test Basis: Users should be allowed to log in on up to 2 devices at a time.

1. *Test Case:* Logging in on Two Devices
    - *Input:* User logs in on two devices simultaneously.
    - *Expected Outcome:* User is successfully logged in on both devices.
2. *Test Case:* Logging in on Third Device
    - *Input:* User attempts to log in on a third device while already logged in on two devices.
    - *Expected Outcome:* User receives an error message indicating device limit reached.

Test Object: Song Download Speed

Test Basis: Songs should take no longer than 5 seconds to download fully onto the user's device.

1. *Test Case:* Song Download Speed
    - *Input:* User initiates download of a song with a proper connection
    - *Expected Outcome:* The song is fully downloaded onto the user's device within 5 seconds.

Test Object: Premium Titles Purchase

Test Basis: Users should be able to access the shop to buy premium titles.

1. *Test Case:* Adding Premium Titles to Basket
    - *Input:* User selects premium titles for purchase.
    - *Expected Outcome:* Selected premium titles are added to the user's basket.
2. *Test Case:* Emptying Basket
    - *Input:* User clicks on the button to empty their basket.
    - *Expected Outcome:* User's basket is cleared, removing all items.
3. *Test Case:* Proceeding to Checkout
    - *Input:* User proceeds to checkout after adding premium titles to the basket.
    - *Expected Outcome:* User can successfully complete the purchase transaction.

3. Unit Testing

In the following pages, unit tests will be realised upon the *tunepalapi.py* file to assure correct functionality and working condition.
The coverage part will state what parts of the code is to be analysed, as well as the expected behaviour of each of them. This will be useful during the bug report afterwards.
In the bug report, failing elements will be brought to light and fixed accordingly as to satisfy the expected behaviour stated in the coverage.
Each test done has it's objective stated and what is being tested as to clarify the purpose of it, and each fix is mentioned in the original file with the problematic code commented out.

i- Coverage

Unit tests for the TunePal API are contained within the file test_tunepalapi.py. Each test includes its respective test objective and test object.
The testing coverage encompasses both code and logic. All functions have been thoroughly tested, providing confidence in the quality of the API. Here's an overview of the coverage for each function:

Constructor:
Tested with various inputs including 0, negative numbers, float numbers, no input, and strings to cover most possible invalid inputs.
Assumption: The API will always have a valid song list to construct itself.

_build_song_window:
Tested with normal page index, negative index, and index exceeding the length of the song list.

add_song:
Tested with valid new song, existing song, new song with existing title, and song with no title.

get_songs:
Tests directly depend on _build_song_window's behaviour.

next_page and previous_page:
Tested to ensure they increase and decrease the page index by one. The API can handle negative and very high indexes, potentially returning an empty list for extreme indexes.

set_page_size:
Tested with the same inputs as the constructor.

search:
Tested with string and integer queries matching existing songs, empty query, and queries returning no songs.

get_songs_since:
Tested with valid year, invalid year like "asdf", and a year that returns an empty string.

Song constructor:
Tested with arguments for title, artist, and release year; only title; and an empty title.

All functions will be thoroughly tested, covering the most significant possibilities. The testing coverage for the TunePal API is extensive, providing assurance that the API behaves properly in various situations.

ii- bug report

During unit testing various errors have come up. The following pages will describe each and what approach has been taken to overcome said errors.

add_song:
lists have no "add" function, but they do have "append". Also the existence of the song in the list was never checked, so an if statement has been added.

```
"""Adds a song, but only if it isn't already in the list"""
def add_song(self, title: str, artist: str, release_year: str):
    #self.songs.add(Song(artist, title, release_year))
    # No such function as add, use append. Also, the order of the parameters is wrong.
    # Also never checked if the song is already in the list>
    new_song = Song(title, artist, release_year)
    if new_song not in self.songs:
        self.songs.append(new_song)
```

current_page_index:
the variable was mistaken with *current_page* in places, specially in the *__init__*.

```
def __init__(self, page_size=None):

    self.page_size = page_size
    #Current page index was never set, so an error would come up when trying to use the API
    self.current_page_index = 0
    with open('songlist.csv') as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            title = row['Song Clean']
            artist = row['ARTIST CLEAN']
```

```
"""Tells the API to move to the previous page"""
def next_page(self):
    #self.current_page_index = self.current_page_index + 1
        #Was using current_page instead ofcurrent_page_index

    self.current_page_index = self.current_page_index + 1

"""Tells the API to move to the next page"""
def previous_page(self):
    self.current_page_index = self.current_page_index - 1
```

Song class:
added default values to artist and year, as they might not be available. Forces the inclusion of the title.

```
#Added defaults for artist and release_year so that the user doesn't have to provide them if they don't have them
#Force the user to provide a title, as it is the most important part of the song
def __init__(self, title: str, artist: str = "-", release_year: str = "-"):
    if not title:
        raise ValueError("Title cannot be empty")
    self.title = title
    self.artist = artist
    self.release_year = release_year
```

Search:
this method wouldn't search for artist names, only songs. Also the *hits* array wouldn't be able to store all the results, so an auxiliary list called *songs_results* was made. This also applies to *get_songs_since*

```
def search(self, starts_with_query: str):
    #Hits might get overwhemled with too many songs, so a global list will store the hits
    hits = []
    for song in self.songs:
        # Never checked for artist, only title
        if song.title.startswith(starts_with_query) or song.artist.startswith(starts_with_query):
            hits.append(song)
        # Get the hits and store them in a global list
        self.songs_results = hits

    return self._build_song_window(self.songs_results)

"""Allows users to filter out old-person music. Filter songs to only return
    songs released since a certain date. e.g. a query of 2022 would only return
    songs released this year
"""
def get_songs_since(self, release_year: str):
    hits = []
    for song in self.songs:
        if song.release_year > release_year:
            hits.append(song)
    #Get the hits and store them in a global list
    self.songs_results = hits
    return self._build_song_window(self.songs_results)
```

get_songs_since:
the method wouldn't include the year searched, and never tested if it was a number.

```python
def get_songs_since(self, release_year: str):
    hits = []
    for song in self.songs:
        #Never checked if the release year was valid. Also the year wasn't being included in the results
        if not song.release_year != "-" or not release_year.isdigit():
            raise ValueError("Invalid release year")
        if song.release_year >= release_year:
            hits.append(song)
    #Get the hits and store them in a global list
    self.songs_results = hits
    return self._build_song_window(self.songs_results)
```

page_size:
could be inputted with 0, negatives or text. Now it's tested.

```python
"""Set the page_size parameter, controllinig how many songs are returned"""
def set_page_size(self, page_size: int):
    # Never checked if the page size was valid
    if isinstance(page_size, int) and page_size > 0:
        self.page_size = page_size
```