Assignment 2

Team number: 61
Team members

Name	Student Nr.	Email
Michal Vladimir Luptak	2763843	m.v.luptak@student.vu.nl
Tomas Busa	2763572	t.busa@student.vu.nl
Ties Schasfoort	2775596	t.j.b.schasfoort@student.vu.nl
Samuel Sameliak	2763687	s.sameliak@student.vu.nl

Summary of changes from Assignment 1

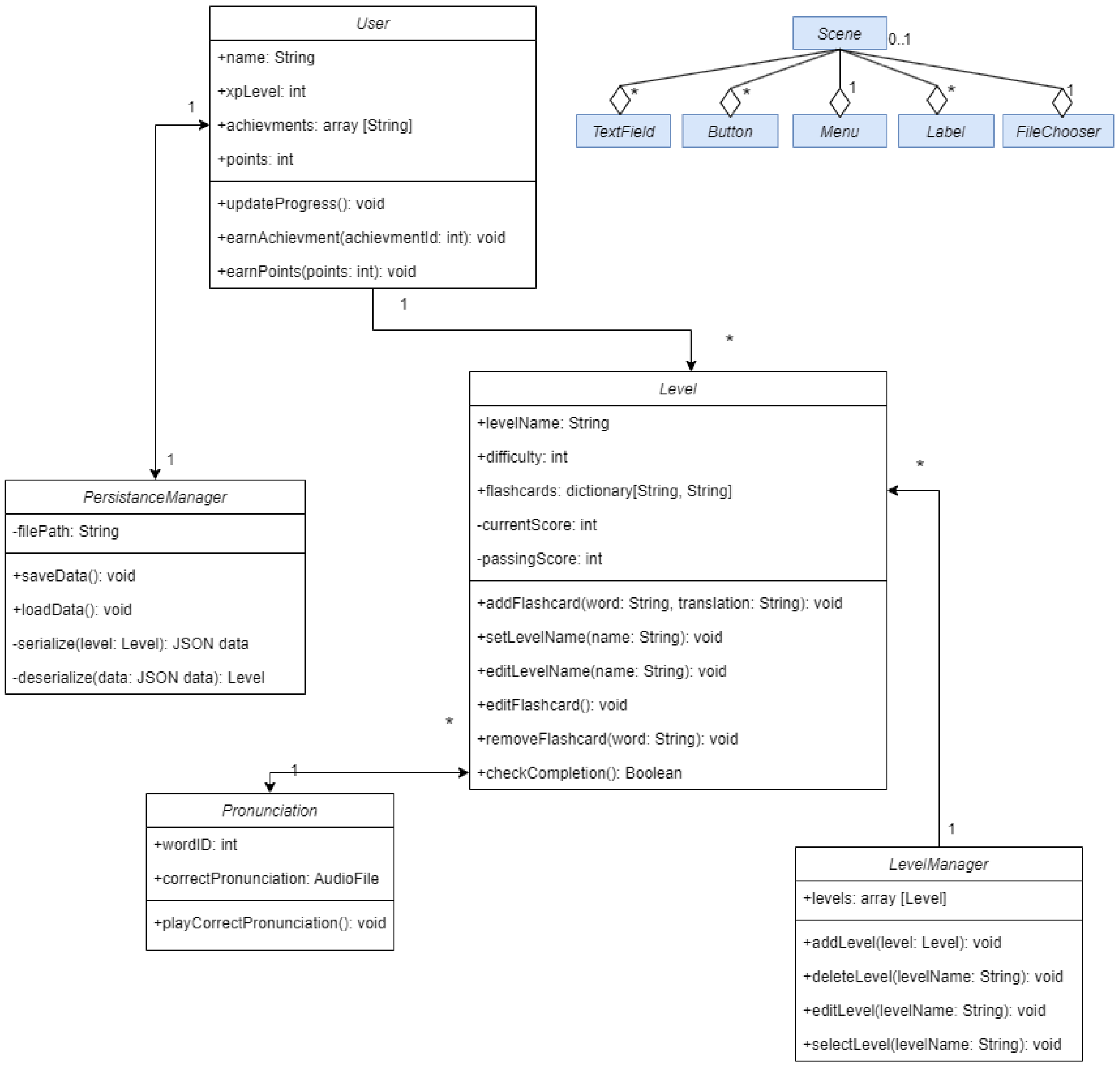
Authors: Ties Schasfoort, Samuel Sameliak, Tomas Busa, and Michal Vladimir Luptak.

Changes we performed in Assignment 1 to address our feedback.

- In the user onboarding feature we added a more extensive description, describing how the initial test looks and how we will determine the starting level of a user based on the result of this initial test.
- We have further elaborated on what gamification features will be added as part of the second feature (Learning Mode). These features include “points” earned for completing levels as well as “achievements” for reaching certain milestones in your learning journey.
- We have changed the description on the levels. This now also includes what the levels are divided by, namely difficulty.
- Furthermore, we have broken down the concrete implementation of our bonus feature “Learn The Pronunciation”, utilising a free API for playing pronunciations of Spanish words.

Class diagram:

Author: Tomas Busa


User
+name: String
+xpLevel: int
+achievments: array [String]
+points: int
+updateProgress(): void
+earnAchievment(achievmentId: int): void
+earnPoints(points: int): void
Scene
0..1
TextField
Button
Menu
Label
FileChooser
*
*
1
*
1
1
1
1
*
Level
+levelName: String
+difficulty: int
+flashcards: dictionary[String, String]
-currentScore: int
-passingScore: int
+addFlashcard(word: String, translation: String): void
+setLevelName(name: String): void
+editLevelName(name: String): void
+editFlashcard(): void
+removeFlashcard(word: String): void
+checkCompletion(): Boolean
*
PersistanceManager
-filePath: String
+saveData(): void
+loadData(): void
-serialize(level: Level): JSON data
-deserialize(data: JSON data): Level
*
1
Pronunciation
+wordID: int
+correctPronunciation: AudioFile
+playCorrectPronunciation(): void
1
LevelManager
+levels: array [Level]
+addLevel(level: Level): void
+deleteLevel(levelName: String): void
+editLevel(levelName: String): void
+selectLevel(levelName: String): void

Note: we omitted any getter and setter methods in this diagram to keep the focus on the inter-class structure. Furthermore, some functions that are only used for class-internal functionality are omitted for the same reasons.

User Class:

Representation:

The User class represents an individual who interacts with our application. This class encapsulates the user's personal data, current level at which they are, number of points they have gained and achievements within our app. Before starting modelling the user class we have discussed the entire application and unanimously agreed that there are two main entities that will play main roles. One of these entities was identified as the user. Therefore having the user class was agreed right from the start of the whole modelling process and we didn't have any alternative ideas for this class.

Relevant attributes:

Achievements: Array of achievements, the user will unlock achievements and receive them. The user will be able to display this list of achievements in a GUI.
Points: Users will gain points by correctly answering flashcards inside levels. Hitting certain point milestones will earn them achievements. This integer attribute will track their current amount of points.
xpLevel: This integer represents at which level the user currently is, the user will be able to choose a lower level, however he won't be able to try a level with difficulty attribute higher than this integer.

Relevant operations:

earnAchievement: Once a user has done something to earn an achievement, this method will be responsible for adding the achievement into an achievement array.
earnPoints(points: int): After each flashcard, this method will be responsible to update the amount of points that the user has.

Relevant Associations:

Level (one-to-many): This association indicates that users can only be at one level at a time, however, throughout using the application, there are multiple levels which the user can progress through.
PersistenceManager (one-to-one): This association indicates that there is a dedicated manager responsible for saving and loading levels.

Level Class:

Representation:

This class represents a specific learning level that contains multiple flashcards which are similar in difficulty. This class encapsulates the contents of the stage and provides functionality for its modification. As previously mentioned in the representation paragraph of the user class, our team discussed that the application revolves around two entities interacting together, first entity was the user, the second entity we identified was the levels. These levels are designed for users to engage with and learn from. We discussed possible alternatives, mainly splitting this class into multiple classes and subclasses, however we agreed that this would make it more complex and with the limited time for implementing it, we decided that the simplicity of encapsulating the whole level functionality inside one class outweighs the possible benefits of splitting it into multiple classes, which would provide us with clearer design and code.

Relevant attributes:

Flashcards: A dictionary/map of type [String, String] which will hold the word and the translation for that word.
PassingScore: This integer represents the minimum score that the user has to gather through the level in order to pass this level and advance to the next one.

Relevant operations:

addFlashcard: This method will be responsible for adding a new flashcard into the dictionary/map of this class.

removeFlashcard: This method will be responsible for deleting the flashcard from the dictionary/map of this class.
checkCompletion(): This method will be used to check if the user has met the minimum criteria for passing the level.

Relevant Associations:

LevelManager(one-to-many): This association indicates that levels are managed by the levelManager. LevelManager will be responsible for actions such as adding, deleting or modifying levels.

LevelManager Class:

Representation:

The LevelManager class is responsible for managing and organising levels within our application. LevelManager will be used for adding, deleting and modifying levels. Lastly in our GUI this class will be used to select the level which the user will interact with. For the functionality of manipulation with levels, we discussed it would be best to create a manager class. This will provide us with the ability to manipulate levels without having to interact with any other classes and thus being very easy for the user. Possible alternative might be to encapsulate this functionality inside the user class, however, we think that would just make the user class multi-purpose, which we do not want.

Relevant attributes:

Levels: An array of Level objects that will keep track of the levels that our application offers to the user.

Relevant operations:

add/delete/modifyLevel: These methods are responsible for modification of the available levels that our application offers.
selectLevel: This method will be responsible for choosing the level that the user wants to try.

Pronunciation class:

Representation:

This class encapsulates the functionality of the bonus feature which helps the user to understand the correct pronunciation of a specific word. It is responsible for interacting with the API and playing the correct sound in case the user wants to learn the right pronunciation. When discussing our bonus feature, we knew we would use one or more API calls. We decided that to keep our design clear, it would be best to separate this from any other class. Therefore to encapsulate the functionality we decided to create one class, however splitting this functionality into more classes could also be a good solution. As mentioned previously we opted for one class mainly because of the simplicity. Splitting this functionality into more classes could be better to keep the code more clear, however we could experience problems with complexity.

Relevant attributes:

wordId: Identifies the specific word of which the correct pronunciation should be played to the user.
correctPronunciation: An audio file which will be received from an API call, this file should be played to the user.

Relevant operations:

playCorrectPronunciation: This method will be responsible for the API calls made in order to play the correct pronunciation of a word.

Relevant Associations:

Level(one-to-one): This association indicates that each level will have to call this class in order to support the functionality of playing the correct pronunciation of a word.

PersistanceManager class:

Representation:

This class is responsible for handling the saving and loading of the levels and flashcards from a specific file. Encapsulating the functionality of the persistence feature into one class was a straightforward decision, without any alternative solutions.

Relevant attributes:

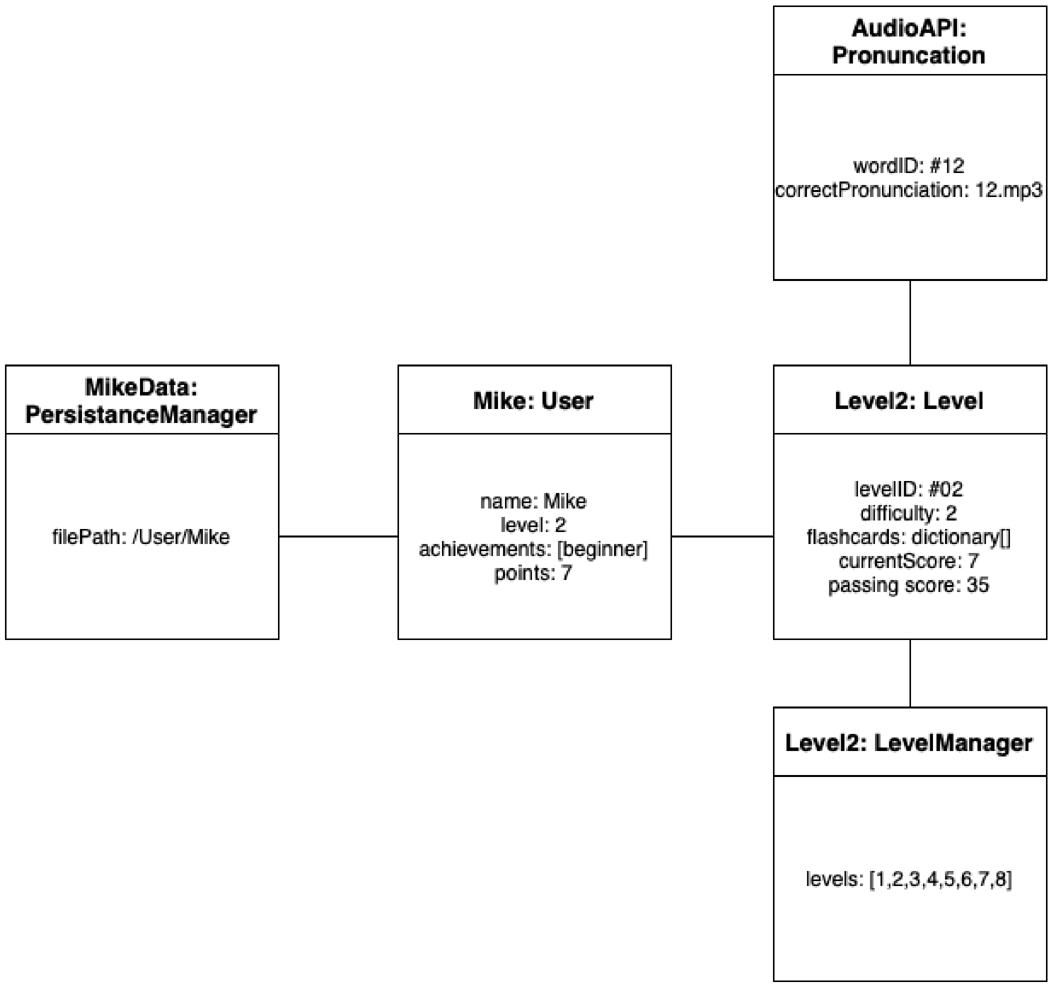
filePath: The path to the file where the data should be saved, or loaded from.

Relevant operations:

save/loadData: These methods will be responsible for saving/loading levels and flashcards to/from files.
Serialize: This method will be internally used by the save method mentioned above to parse data into JSON data which then will be saved in a file.
Deserialize: This method will be internally used by the load method, it will take JSON data

Object diagram:

Author: Samuel Sameliak

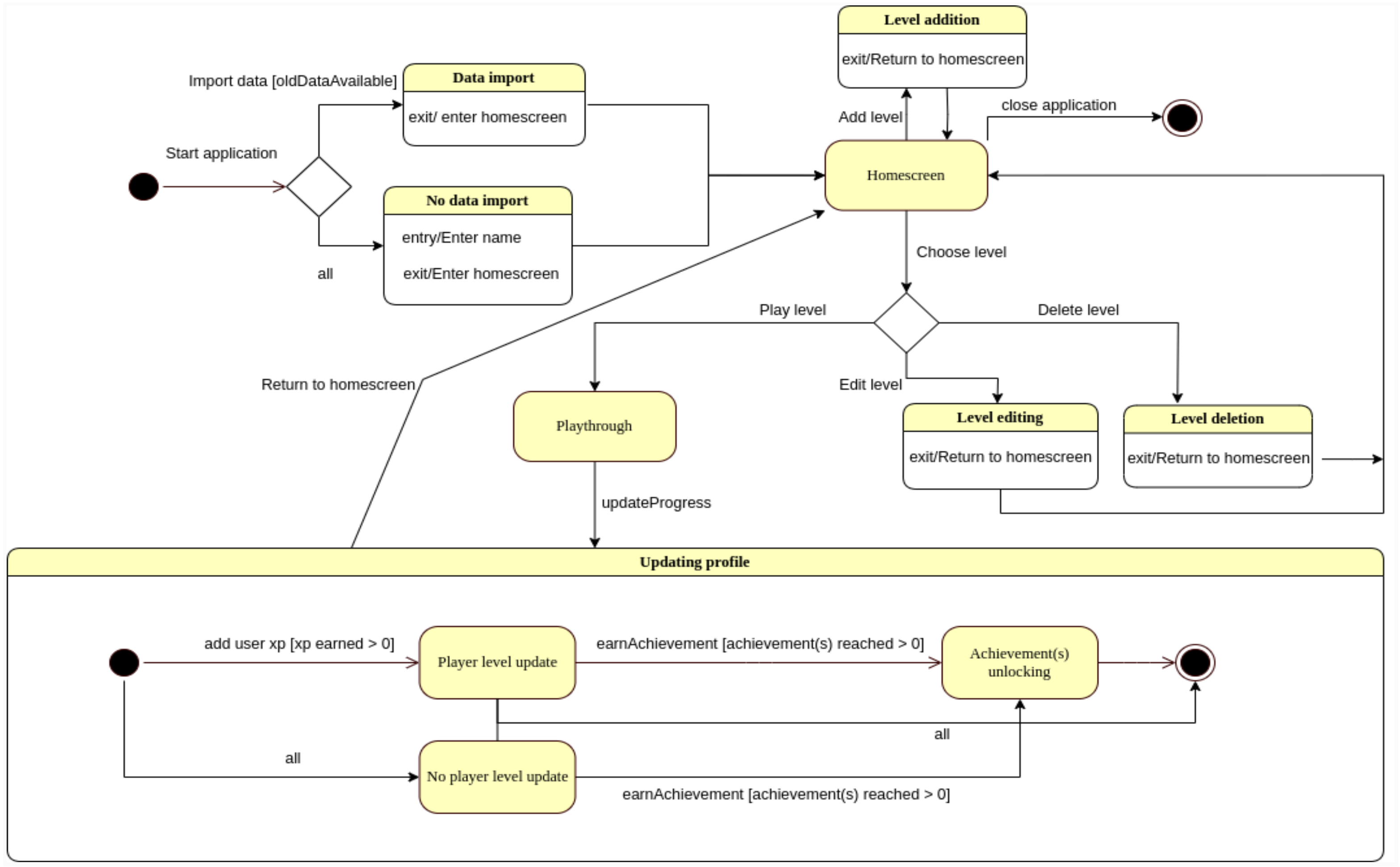


This object diagram presents the state of the application, when the user is in level. "MikeData: PersistenceManager" manages the persistent data, for example storage at the file path "/User/Mike". "Mike: User" represents a user profile with the name Mike, who is currently on level 2, has beginner achievement and has earned 7 points out of 35 needed to pass this level. "Level2: LevelManager" manages game levels, from level 1 to 8. "AudioAPI: Pronunciation" is responsible for pronunciation side of the application, with a specific word identified by ID #12 and its correct pronunciation linked to the audio file "12.mp3".

State machine diagrams:

Author: Ties Schasfoort

User Class state machine:



The state machine above is based on the user class. Even though it may only be restricted to that class, we also decided to make it as a high-level overview of our entire system and the way the user can interact with it. This will make it clearer to get an idea of how the system works and what methods/attributes are needed for the user class.

The initial state is when the user starts the application. The user then gets to decide to either import their already existing progress (if available), or enter a name which results in making a new profile. This is represented by the decision node and the two outgoing edges, the top one representing the former decision and the bottom one the latter. After this decision, the system can either have the state in which an old state is restored from the imported data, or a new account is made. After this the user is send to the homescreen.

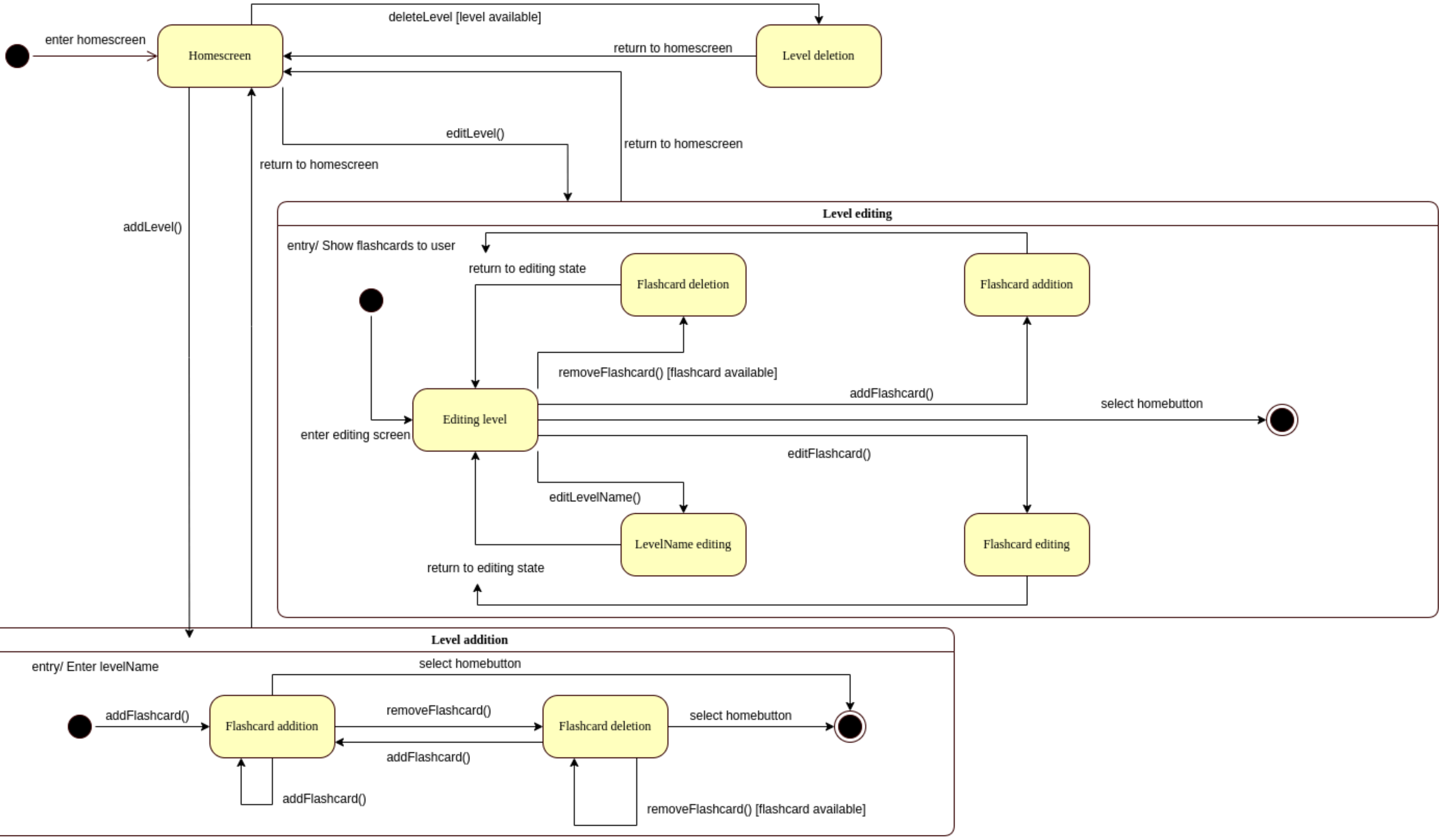
From the homescreen the user can either choose a level, or add one. In the latter case, the user makes a new level and returns to the homescreen after finishing adding it. In the former case, the user can choose to do one of three things:

1. The user deletes the selected level and after that returns to the homescreen.
2. The user edits a level and after that returns to the homescreen.

The third option requires a bit more elaboration than the other two:
The user first plays a level. After they did this, the progress made by the user will be registered and added to the user's profile. It does this in the "updated profile" composite state.

In this composite state the first two options are whether the user earned xp or not. These two options lead to their respective states: player level (not) updated, where this attribute of the player class is updated. After that, the user has either earned achievement(s) or not. If they did, the state changes to Achievement(s) unlocked and these achievement(s) are added to the user class. After these are added, the end state of this composite state is reached. If they did not get any achievements, the end of this composite state is reached instantly. Finally, the composite state is exited and the user returns to the home screen.

Level Manager Class state machine:



The state machine above is based on the LevelManager class. It provides a low-level overview of its functionality.

The initial state is when the user selected either to enter a name to create a new profile or import existing levels. From this initial state the user enters the homescreen. When at the homescreen, the user has 3 options regarding managing the levels:

1. They can edit a level
2. They can add a level
3. They can delete a level.

Deleting a level is simple: the user presses the trash icon to delete the level and all its elements like flashcards and associated difficulty.

Choosing to edit a level requires further elaboration on its composite state:
After the user selected to edit a level they are first shown the existing flashcards (both the initial word and its translation).
Then the user enters the editing state. We added this state to get a clear view of the transitions from this state to having performed an operation (like level addition) and back. This is instead of having transitions from each state to each other, like from level addition to level deletion and back.
The editing state transitions to the following 5 states, and vice versa:

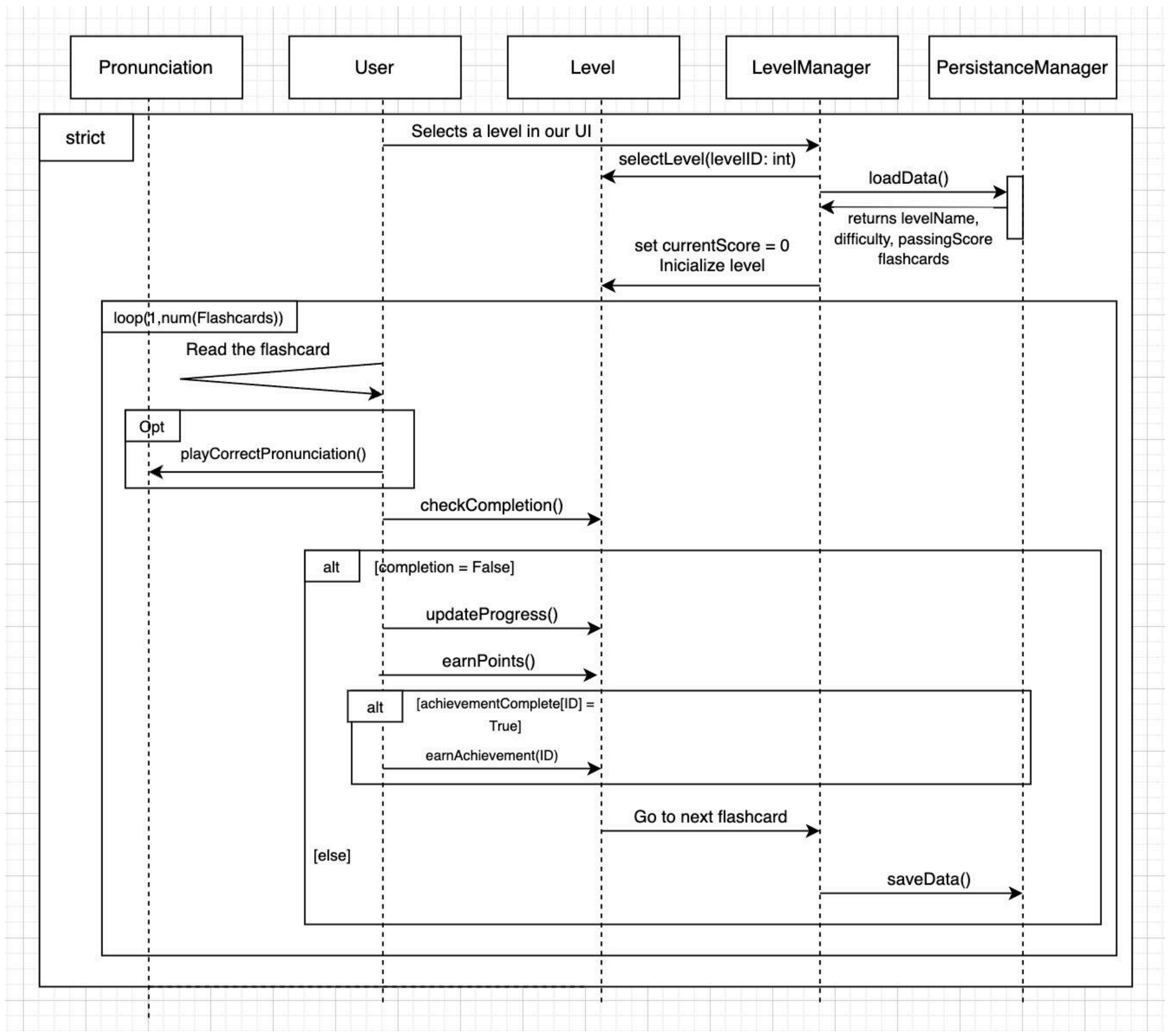
1. Flashcard deletion = of course this can only be done when there are flashcards available.
2. Flashcard addition
3. LevelName editing
4. Flashcard editing
5. Terminal state = after user selects to go back to the homescreen, the composite state exits.

The other composite state is included for representing the “Level addition” state:
First the user enter a name for the level upon entering the state. After this the user can only add a flashcard, editing the name is only possible when selecting “editLevel()”. After adding the first flashcard, the user can choose to either delete it, return to the homescreen or add another flashcard. Deleting a flashcard can only be done when there is one present, which there is after adding one. This is the reason as to why there is a guard on the self-transition by the “flashcard deletion” state, but not on the transition from “flashcard addition” -> “flashcard deletion”.
When the user is done adding or deleting flashcards when creating the level, they can select the homescreen button which takes them back to the homescreen where they can again manage the levels.

Sequence diagrams:

Author: Michal Lupták

User Completing a Level Sequence Diagram:



Sequence Diagram Description:
The sequence diagram has strict sequencing, and begins with the `User` selecting a level through the user interface (UI). The `LevelManager` then sends a `selectLevel(levelID: int)` message to the `Level`, indicating their choice of level.

The `LevelManager` then interacts with the `PersistenceManager` by sending a `loadData()` message to retrieve the level data, which includes `levelName`, `difficulty`, `passingScore`, and `flashcards`. The `PersistenceManager` processes this request and returns the required data to the `LevelManager`. Upon receiving this data, the `LevelManager` initialises the chosen `Level` with the retrieved data and sets the `currentScore` to zero, indicating the start of the level for the user.

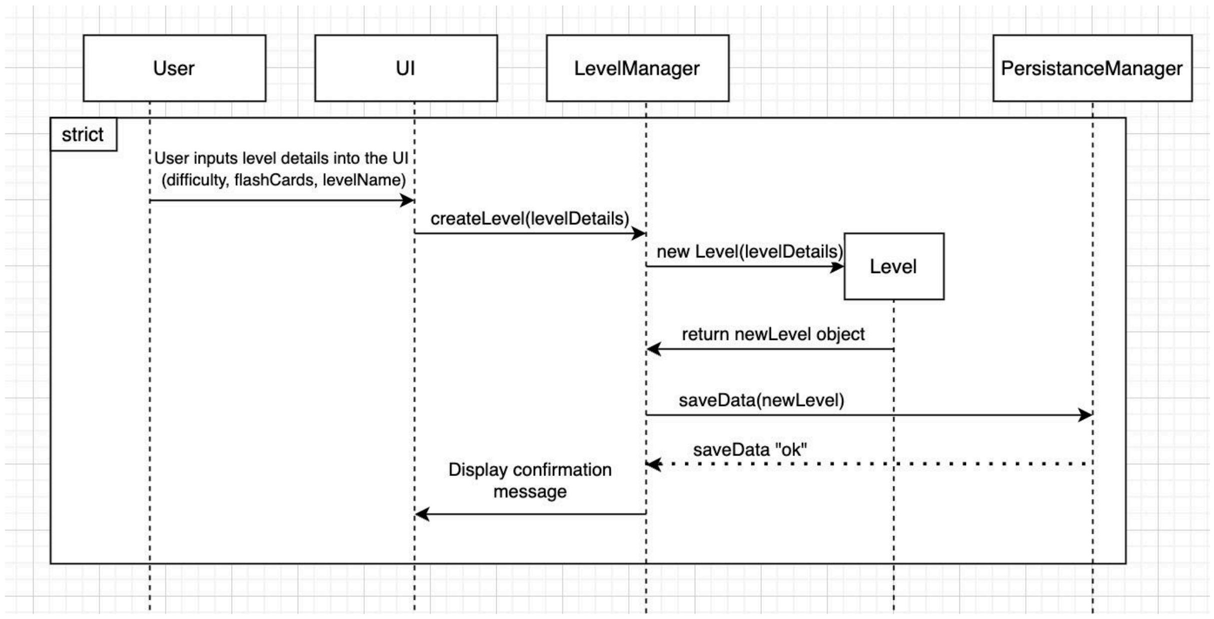
Within the level, the user engages in a loop to read through each of the flashcards, represented by the `loop(1, num(Flashcards))` fragment in the diagram. For each flashcard, there is an optional (`Opt`) interaction where the `User` can request the correct pronunciation of the word from the `Pronunciation` object using the `playCorrectPronunciation()` message. Whether or not the user opts to hear the pronunciation, they proceed to answer or interact with the flashcard.

After the user interacts with a flashcard, the `Level` object checks for completion of that particular card by invoking `checkCompletion()`. If the level is not completed (`completion = False`), the user updates their progress with the `updateProgress()` message. If the user completes the flashcard correctly, they earn points, represented by the `earnPoints()` message sent to the `User` object. Earning points can trigger achievements; this is depicted with an `alt` fragment where if an `achievementComplete[ID]` condition is true, the `earnAchievement(ID)` message is sent to the `User`.

After this, the loop continues to the next flashcard. Once all the flashcards have been completed, and the user has finished the level, the `Level` object sends a `saveData()` message to the `PersistenceManager`, which then serialises and saves the user's new state, including the updated level progress, points, and any new achievements.

Key Elements and Interactions:
- **Actors**: User, LevelManager, PersistenceManager, Level, and Pronunciation.
- **Messages**: `selectLevel`, `loadData`, `playCorrectPronunciation`, `checkCompletion`, `updateProgress`, `earnPoints`, `earnAchievement`, and `saveData`.
- **Fragments of Interaction**: A loop for iterating over flashcards, an option for playing pronunciation, alternatives for earning points and achievements, and the final save action upon completion of the level.

Level Creation Sequence Diagram:


User
UI
LevelManager
PersistanceManager
strict
User inputs level details into the UI (difficulty, flashCards, levelName)
createLevel(levelDetails)
new Level(levelDetails)
Level
return newLevel object
saveData(newLevel)
saveData "ok"
Display confirmation message

Sequence Diagram Description:
The sequence diagram depicts the strict sequencing of events during the level creation process in the application. It starts with the `User` inputting level details into the user interface (UI), which includes elements such as difficulty, flashCards, and levelName.

Once the user inputs the level details, a message `createLevel(levelDetails)` is sent to the `LevelManager`. The `LevelManager` then creates a new `Level` instance using the `new Level(levelDetails)` constructor, which initialises the level with the provided details.

After the `Level` object is created, the `LevelManager` returns the new `Level` object. This newly created level is then sent to the `PersistenceManager` with the `saveData(newLevel)` message, indicating that the level's data should be persisted to storage.

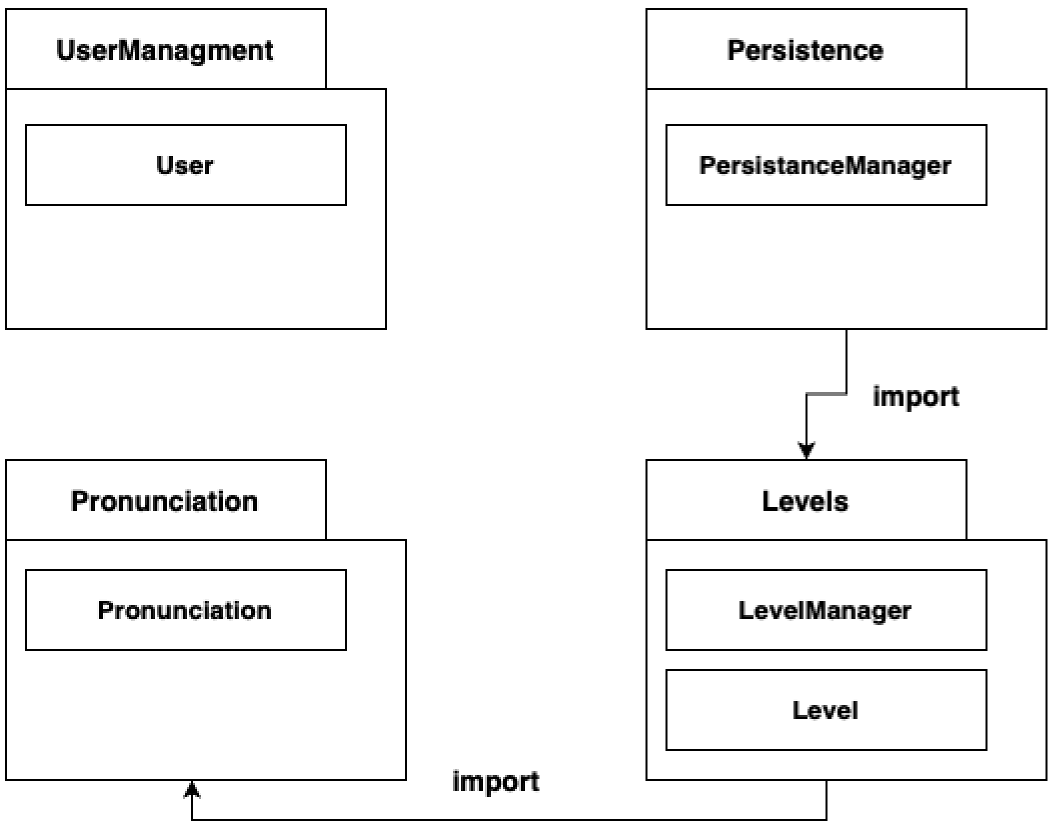
The `PersistenceManager` processes the request and saves the new level data. Upon successful save operation, the `PersistenceManager` sends a confirmation message `saveData "ok"` back to the `LevelManager`.

Finally, the `LevelManager` displays a confirmation message back to the `User`, indicating that the new level has been successfully created and saved. This provides feedback to the user that the level creation process is complete.

Key Elements and Interactions:
- **Actors**: User, UI, LevelManager, PersistenceManager.
- **Messages**: `createLevel`, `new Level`, `saveData`, `saveData "ok"`.
- **Fragments of Interaction**: The process is encapsulated within a strict sequence to ensure that the level creation steps happen in a specific order without deviation.

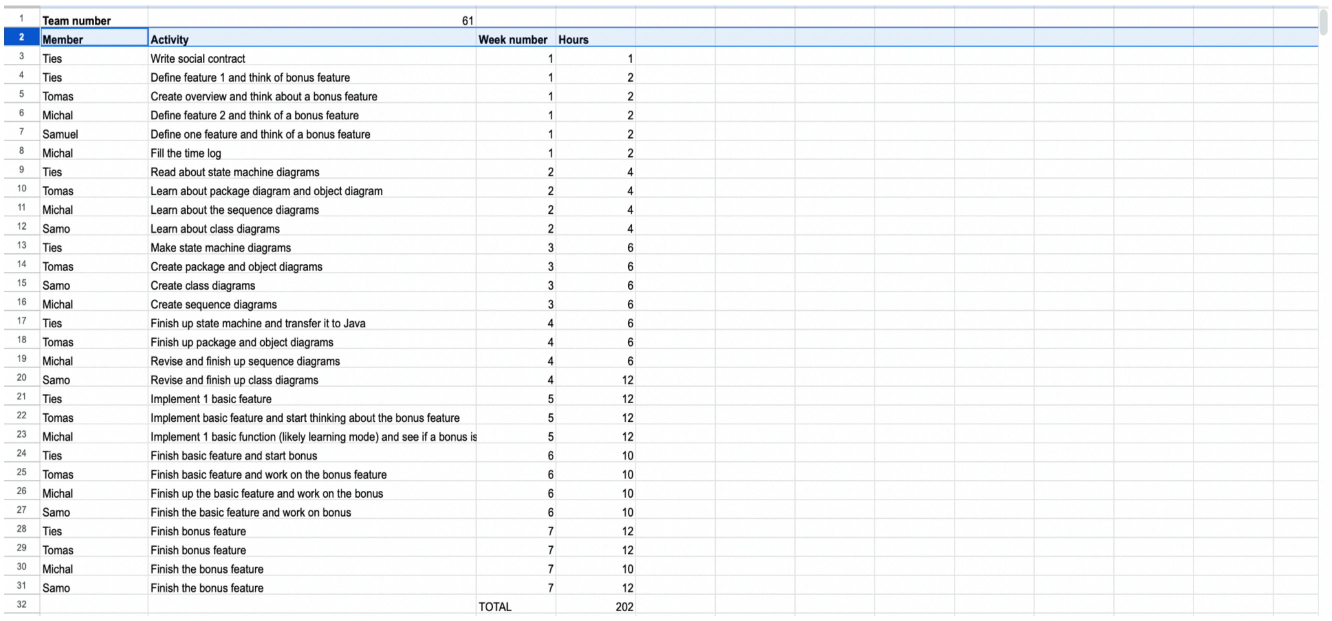
Package diagram:

Author: Samuel Sameliak



The image above is a package diagram, which is divided into four packages. "UserManagement", "Persistence", "Pronunciation" and "Levels". The "UserManagement" package contains a "User" class. "Persistence" package includes a "PersistenceManager" class. “Levels” package includes “LevelManager” and “Level” classes, simultaneously it imports the “Persistence” package.”Pronunciation” package contains “Pronunciation” class and it imports “Levels” package.

Time logs:



Team number	61		
Member	**Activity**	**Week number**	**Hours**
Ties	Write social contract	1	1
Ties	Define feature 1 and think of bonus feature	1	2
Tomas	Create overview and think about a bonus feature	1	2
Michal	Define feature 2 and think of a bonus feature	1	2
Samuel	Define one feature and think of a bonus feature	1	2
Michal	Fill the time log	1	2
Ties	Read about state machine diagrams	2	4
Tomas	Learn about package diagram and object diagram	2	4
Michal	Learn about the sequence diagrams	2	4
Samo	Learn about class diagrams	2	4
Ties	Make state machine diagrams	3	6
Tomas	Create package and object diagrams	3	6
Samo	Create class diagrams	3	6
Michal	Create sequence diagrams	3	6
Ties	Finish up state machine and transfer it to Java	4	6
Tomas	Finish up package and object diagrams	4	6
Michal	Revise and finish up sequence diagrams	4	6
Samo	Revise and finish up class diagrams	4	12
Ties	Implement 1 basic feature	5	12
Tomas	Implement basic feature and start thinking about the bonus feature	5	12
Michal	Implement 1 basic function (likely learning mode) and see if a bonus is	5	12
Ties	Finish basic feature and start bonus	6	10
Tomas	Finish basic feature and work on the bonus feature	6	10
Michal	Finish up the basic feature and work on the bonus	6	10
Samo	Finish the basic feature and work on bonus	6	10
Ties	Finish bonus feature	7	12
Tomas	Finish bonus feature	7	12
Michal	Finish the bonus feature	7	10
Samo	Finish the bonus feature	7	12
		TOTAL	202