System design document for Smurfinator

Isac Jason, Erik Green Blomroos,

Luqas Lundahl, Jonatan Cederberg,

Noel Wiman Ohlson

14 December, 2023
Version 1.0

1 Introduction

This document will describe the applications underlying structure. It will explain how MVC is implemented and how each package is created by the use of design patterns and their relationship with other modules.

1.1 System introduction

Smurfinator is a guessing game written in Java. The JavaFX framework is used to implement the graphical user interface and the MVC pattern is used as the structural framework for the application to make it as modular as possible. Furthermore, the program is easily extensible, by changing the sets of traits, characters and questions, the program could easily be about guessing characters from Star Wars rather than from the Smurfs. The data the program revolves around could also easily be extended by simply creating more traits, questions and characters.

1.2 Definitions, acronyms, and abbreviations

Smurfinator - The main “game”/ also the name for the entire project
Character - The thing the application is trying to guess
Trait - Character has multiple traits (example: happy, magical, leader…)
Questions - A question that corresponds to a trait, can either be answered by yes/ no or by a number between 1-10 (or “don’t know”).
User - A user that the player is logged in to that tracks the users game contributions

Contributions - The amount of characters a user has added to the program
Leaderboard - Rankings entities based on corresponding measurable integer values.
Collection - Pages of characters showing which ones has been guessed through playing.
WindowHandler - The class that handles the fxml windows, takes action when buttons are pressed and displays new information when it comes from the model or controller.

2 System architecture

The application is structured after the MVC pattern and can be divided into 3 distinguishable responsibilities. The view and the controller parts of the structure are closely intertwined thanks to the way JavaFX controller concept works with window handlers.

2.1 Model

The model keeps track of the program's logic and data. Mainly focusing on how the actions called by the controller affect the data and the state of the program, but is also responsible for keeping, saving and creating new data. The model also handles the logic behind different views of the program, such as the main menu, smurfinator, leaderboards, etc. The model's computations are made by the standard Java library, meaning the model uses minimal external dependencies. Effort has also been made in order to assure that the model work is self sufficient, and that it has no dependency on either the view or the controller parts of the program.

The model itself is structured and separated into different packages, minimizing unnecessary dependencies and allowing easier working with the codebase.

2.2 Controller

The controller part of the program is responsible for taking input from the user and delegating tasks to the model. Since the project uses JavaFX for GUI framework, the controller classes get notified of any user input from the corresponding window handlers, and knows which call to make to the model based on inputtype. The controller acts as a simple delegation tool from the different inputs displayed in the view, to the corresponding functionality in the model.

2.3 View

The view- part of the program is implemented through FXML for pure graphics and java for the logic. As mentioned above, much of the logic for the view lies in the windowhandler classes. The way the view is called for update comes from the model, whereas the model has access to the part of the windowhandler that purely regards updating the view. Ideally, no domain logic is handled by the “view”-part of the codebase, especially the methods implemented from the observer-interfaces.

2.4 Application in process

The program launches from the main class by instantiating the model, which is a class that acts as a *facade* to the entire model part of the program, with relevant data. The program starts with a view of the login function, allowing the user to either log in by entering password and username that exists with an existing user. If there are none existing user profiles, the user can either choose to create a new userprofile, or start the program as a guest user. After the login page the user accesses the rest of the program from the main menu view. From this point the user can close the program, navigate to leaderboard or collection, or the guessing game.

The leaderboard is a view showing either characters in a sorted list based on amount of times guessed, or if you change category, users sorted by the amount of contributions to the program in forms of characters created through the guessing game’s character creation sequence. In the collection view, the different characters in the program can be seen by parsing through pages, though only showing names and clear images of those already found through the smurfinator game. From the main menu’s start button, the user begins the main game of the program. Parsing through the game follows receiving questions that can be answered either by yes, no or don’t know alternatives, or by giving an integer answer between 1- 10 (or don’t know). This continues until either the program makes a guess on which character the user is thinking of, or if it can't make a guess, lets the user either choose to create a new character to be added to the program or simply return to the main menu. If the program makes its guess and shows the name and image of the character guessed. After a guess has been made, the user can either create a new character (if the guess is wrong) or continue (return to the main menu).

When in the character creating state, the program keeps asking the rest of the questions left unanswered so far, building a profile of traits for the new character, and when finished, the user is to give a name and an imagepath to finalize the creation process.

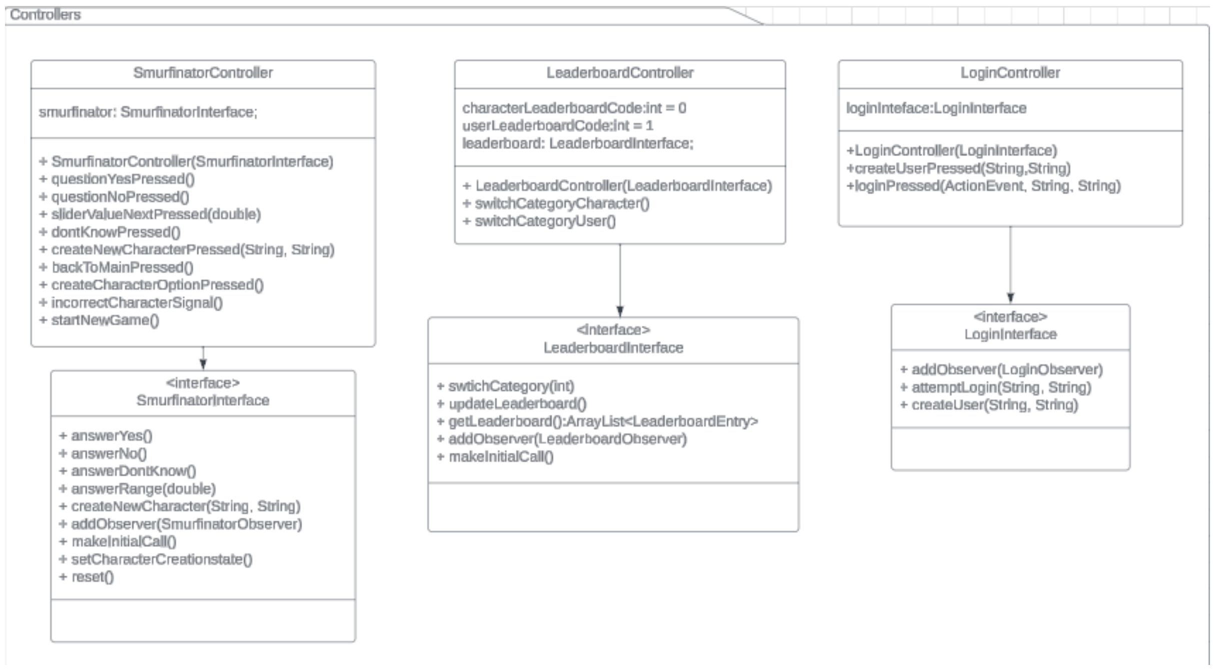
3. System design

This section reflects the structure of the application's design and its motivations using UML-diagrams.

The structure of the project follows the MVC-pattern in order to separate the back end logic of the program from the front end by separating the responsibilities of the program into Model, View and Controller. View displays output to the user and notifies the controller which inputs have been received, the controller takes note of which input has been sent through the view and calls the model to act accordingly and the model processes the input and the logic.

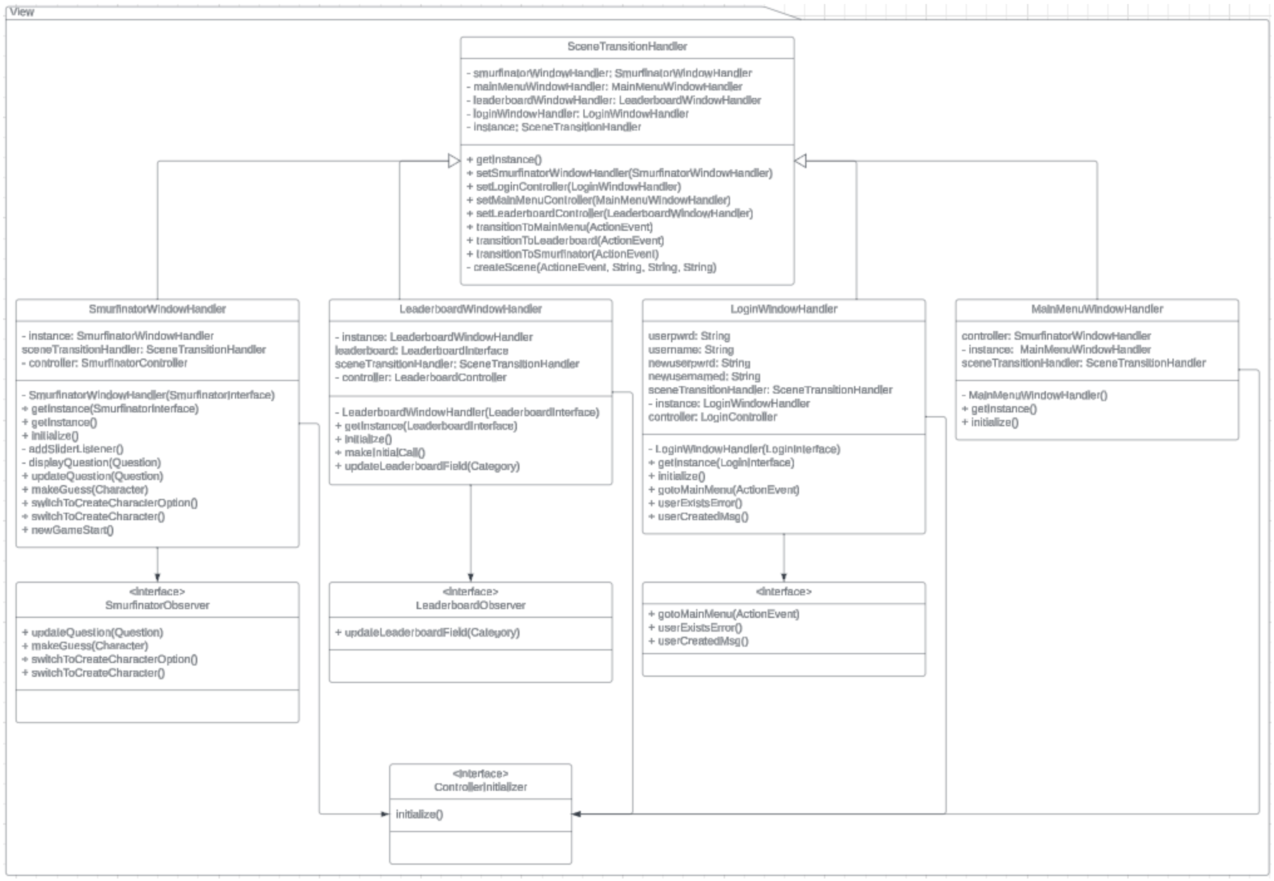
3.1 Project UML diagram

3.1.1 Controller diagram


Controllers
SmurfinatorController
smurfinator: SmurfinatorInterface;
+ SmurfinatorController(SmurfinatorInterface)
+ questionYesPressed()
+ questionNoPressed()
+ sliderValueNextPressed(double)
+ dontKnowPressed()
+ createNewCharacterPressed(String, String)
+ backToMainPressed()
+ createCharacterOptionPressed()
+ incorrectCharacterSignal()
+ startNewGame()
<interface>
SmurfinatorInterface
+ answerYes()
+ answerNo()
+ answerDontKnow()
+ answerRange(double)
+ createNewCharacter(String, String)
+ addObserver(SmurfinatorObserver)
+ makeInitialCall()
+ setCharacterCreationstate()
+ reset()
LeaderboardController
characterLeaderboardCode:int = 0
userLeaderboardCode:int = 1
leaderboard: LeaderboardInterface;
+ LeaderboardController(LeaderboardInterface)
+ switchCategoryCharacter()
+ switchCategoryUser()
<Interface>
LeaderboardInterface
+ swtichCategory(int)
+ updateLeaderboard()
+ getLeaderboard():ArrayList<LeaderboardEntry>
+ addObserver(LeaderboardObserver)
+ makeInitialCall()
LoginController
loginInteface:LoginInterface
+LoginController(LoginInterface)
+createUserPressed(String,String)
+loginPressed(ActionEvent, String, String)
<interface>
LoginInterface
+ addObserver(LoginObserver)
+ attemptLogin(String, String)
+ createUser(String, String)

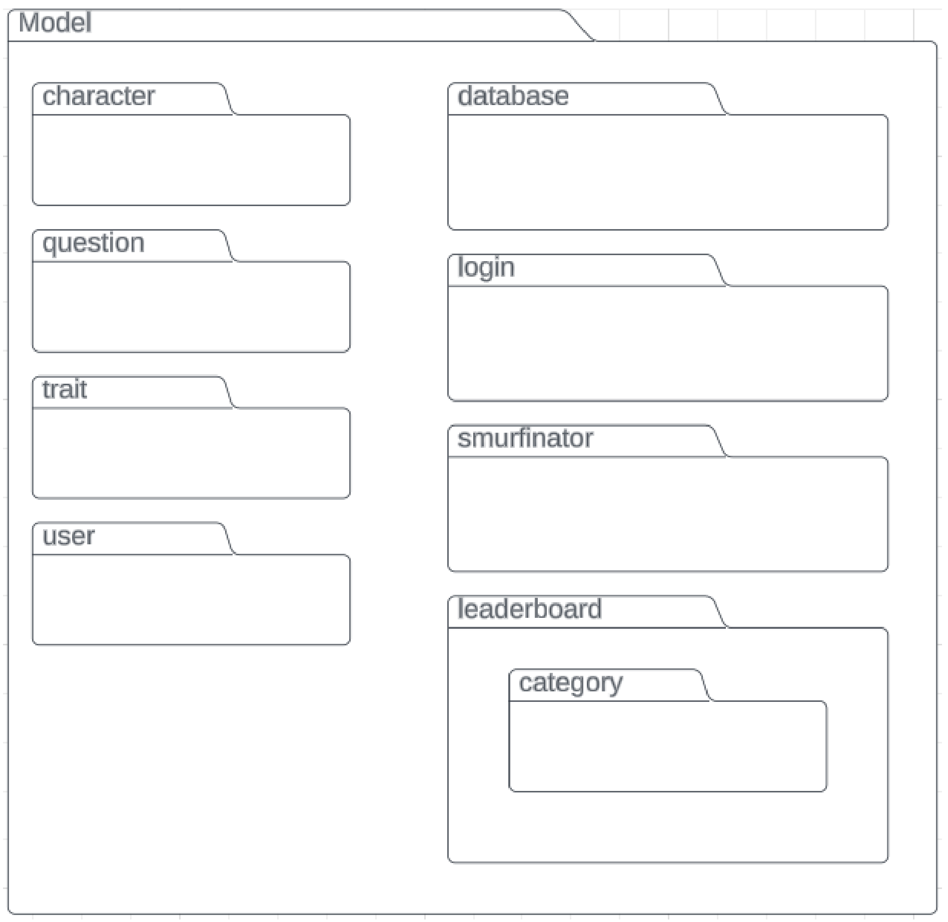
UML diagram over the Controllers

3.1.2 View diagram


View
SceneTransitionHandler
- smurfinatorWindowHandler: SmurfinatorWindowHandler
- mainMenuWindowHandler: MainMenuWindowHandler
- leaderboardWindowHandler: LeaderboardWindowHandler
- loginWindowHandler: LoginWindowHandler
- instance: SceneTransitionHandler
+ getInstance()
+ setSmurfinatorWindowHandler(SmurfinatorWindowHandler)
+ setLoginController(LoginWindowHandler)
+ setMainMenuController(MainMenuWindowHandler)
+ setLeaderboardController(LeaderboardWindowHandler)
+ transitionToMainMenu(ActionEvent)
+ transitionToLeaderboard(ActionEvent)
+ transitionToSmurfinator(ActionEvent)
- createScene(ActioneEvent, String, String, String)
SmurfinatorWindowHandler
- instance: SmurfinatorWindowHandler
sceneTransitionHandler: SceneTransitionHandler
- controller: SmurfinatorController
- SmurfinatorWindowHandler(SmurfinatorInterface)
+ getInstance(SmurfinatorInterface)
+ getInstance()
+ initialize()
- addSliderListener()
- displayQuestion(Question)
+ updateQuestion(Question)
+ makeGuess(Character)
+ switchToCreateCharacterOption()
+ switchToCreateCharacter()
+ newGameStart()
LeaderboardWindowHandler
- instance: LeaderboardWindowHandler
leaderboard: LeaderboardInterface
sceneTransitionHandler: SceneTransitionHandler
- controller: LeaderboardController
- LeaderboardWindowHandler(LeaderboardInterface)
+ getInstance(LeaderboardInterface)
+ initialize()
+ makeInitialCall()
+ updateLeaderboardField(Category)
LoginWindowHandler
userpwrd: String
username: String
newuserpwrd: String
newusernamed: String
sceneTransitionHandler: SceneTransitionHandler
- instance: LoginWindowHandler
controller: LoginController
- LoginWindowHandler(LoginInterface)
+ getInstance(LoginInterface)
+ initialize()
+ gotoMainMenu(ActionEvent)
+ userExistsError()
+ userCreatedMsg()
MainMenuWindowHandler
controller: SmurfinatorWindowHandler
- instance: MainMenuWindowHandler
sceneTransitionHandler: SceneTransitionHandler
- MainMenuWindowHandler()
+ getInstance()
+ initialize()
<Interface>
SmurfinatorObserver
+ updateQuestion(Question)
+ makeGuess(Character)
+ switchToCreateCharacterOption()
+ switchToCreateCharacter()
<Interface>
LeaderboardObserver
+ updateLeaderboardField(Category)
<Interface>
+ gotoMainMenu(ActionEvent)
+ userExistsError()
+ userCreatedMsg()
<Interface>
ControllerInitializer
initialize()

UML diagram for the View

3.1.3 Model diagram

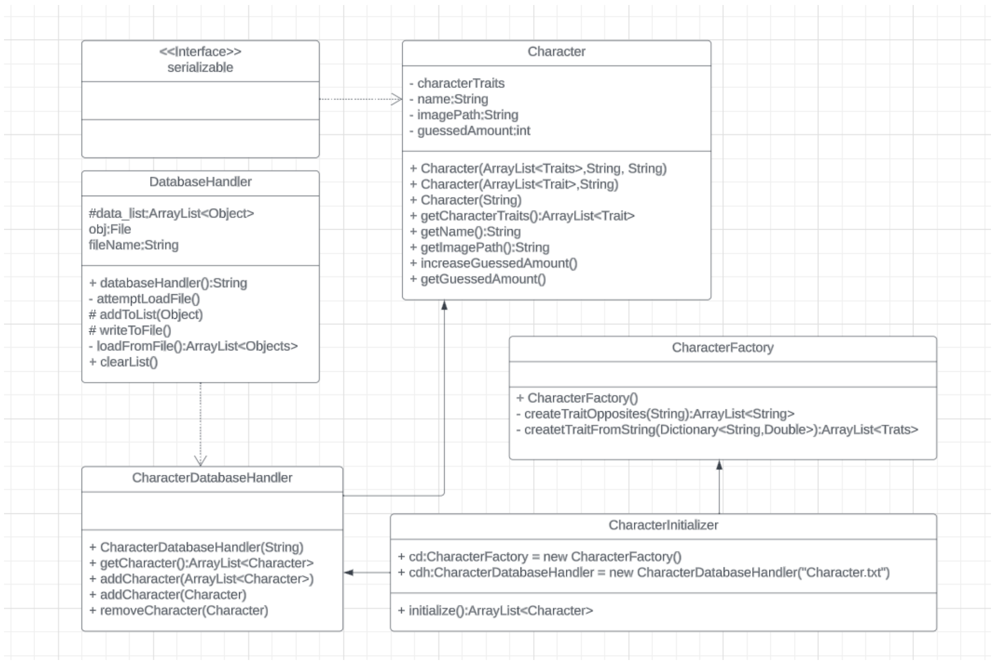


The packages of the model

3.1.4 Implementation of the MVC-pattern and the relations between the modules

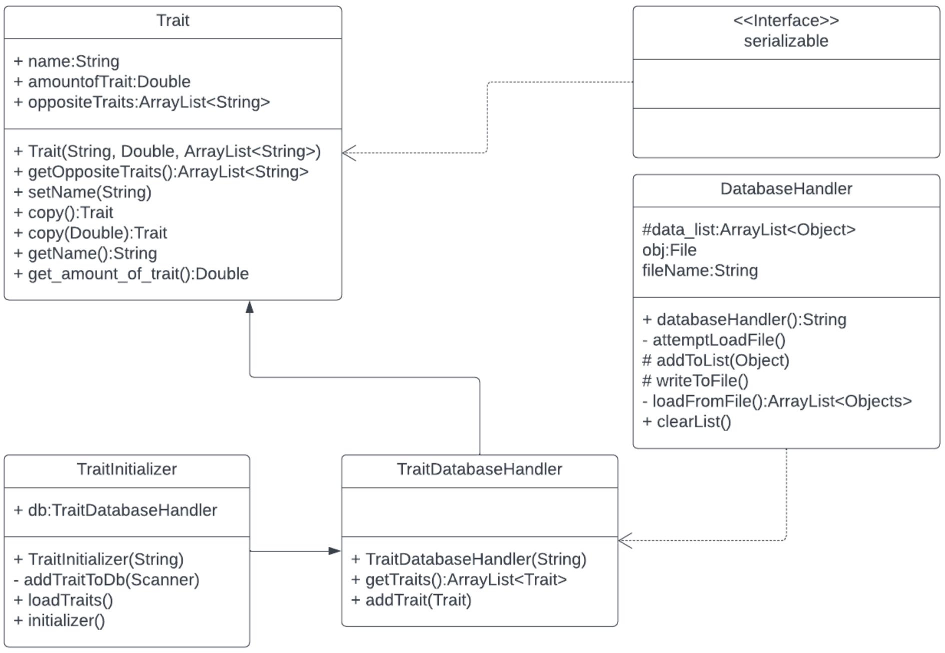
3.2 Class diagrams

3.2.1 Character diagram



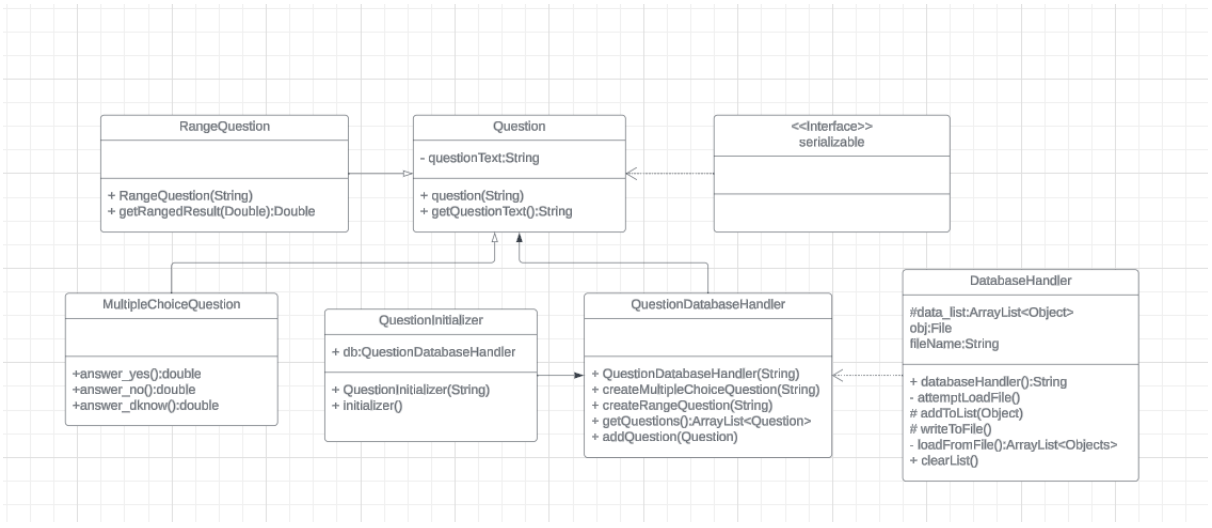
<<Interface>>
serializable
Character
- characterTraits
- name:String
- imagePath:String
- guessedAmount:int
+ Character(ArrayList<Traits>,String, String)
+ Character(ArrayList<Trait>,String)
+ Character(String)
+ getCharacterTraits():ArrayList<Trait>
+ getName():String
+ getImagePath():String
+ increaseGuessedAmount()
+ getGuessedAmount()
DatabaseHandler
#data_list:ArrayList<Object>
obj:File
fileName:String
+ databaseHandler():String
- attemptLoadFile()
# addToList(Object)
# writeToFile()
- loadFromFile():ArrayList<Objects>
+ clearList()
CharacterFactory
+ CharacterFactory()
- createTraitOpposites(String):ArrayList<String>
- createtTraitFromString(Dictionary<String,Double>):ArrayList<Trats>
CharacterDatabaseHandler
+ CharacterDatabaseHandler(String)
+ getCharacter():ArrayList<Character>
+ addCharacter(ArrayList<Character>)
+ addCharacter(Character)
+ removeCharacter(Character)
CharacterInitializer
+ cd:CharacterFactory = new CharacterFactory()
+ cdh:CharacterDatabaseHandler = new CharacterDatabaseHandler("Character.txt")
+ initialize():ArrayList<Character>

3.2.2 Trait diagram

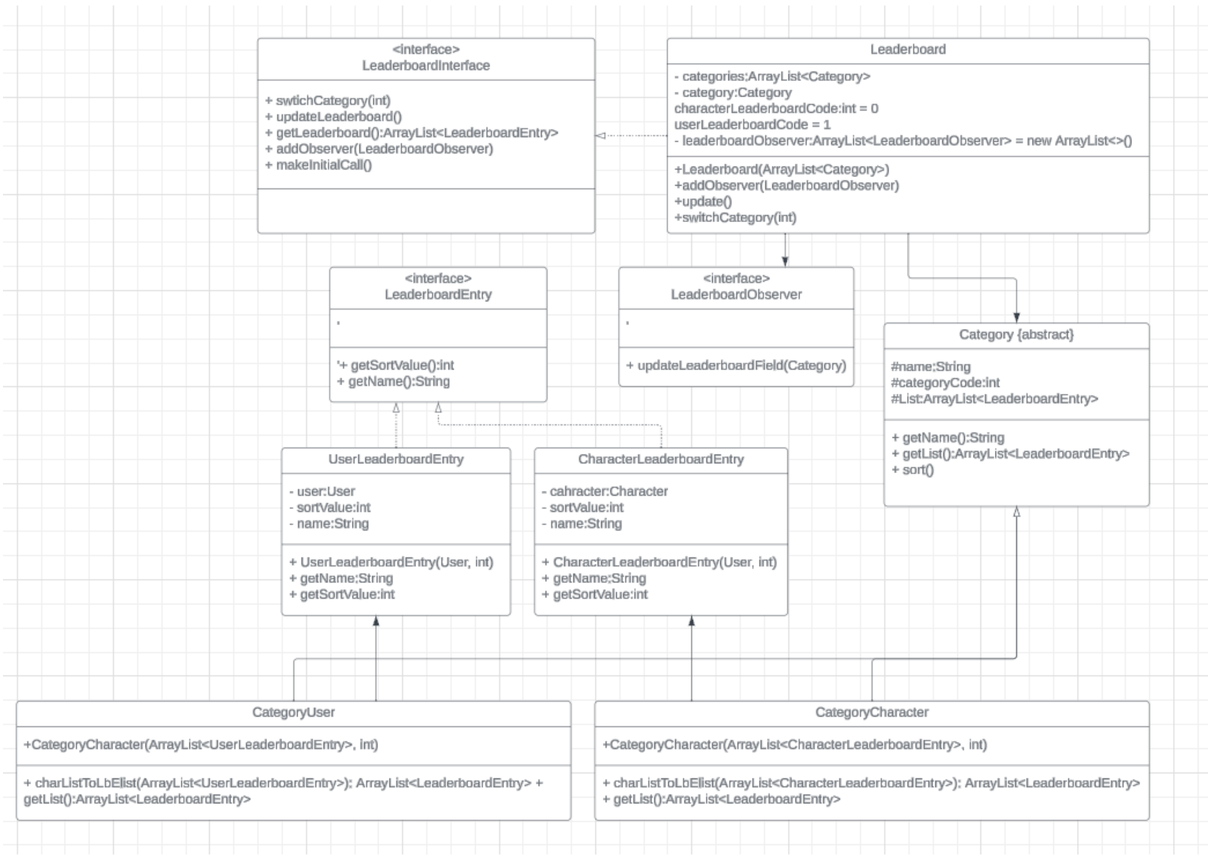


Trait
+ name:String
+ amountofTrait:Double
+ oppositeTraits:ArrayList<String>
+ Trait(String, Double, ArrayList<String>)
+ getOppositeTraits():ArrayList<String>
+ setName(String)
+ copy():Trait
+ copy(Double):Trait
+ getName():String
+ get_amount_of_trait():Double
<<Interface>>
serializable
DatabaseHandler
#data_list:ArrayList<Object>
obj:File
fileName:String
+ databaseHandler():String
- attemptLoadFile()
# addToList(Object)
# writeToFile()
- loadFromFile():ArrayList<Objects>
+ clearList()
TraitInitializer
+ db:TraitDatabaseHandler
+ TraitInitializer(String)
- addTraitToDb(Scanner)
+ loadTraits()
+ initializer()
TraitDatabaseHandler
+ TraitDatabaseHandler(String)
+ getTraits():ArrayList<Trait>
+ addTrait(Trait)

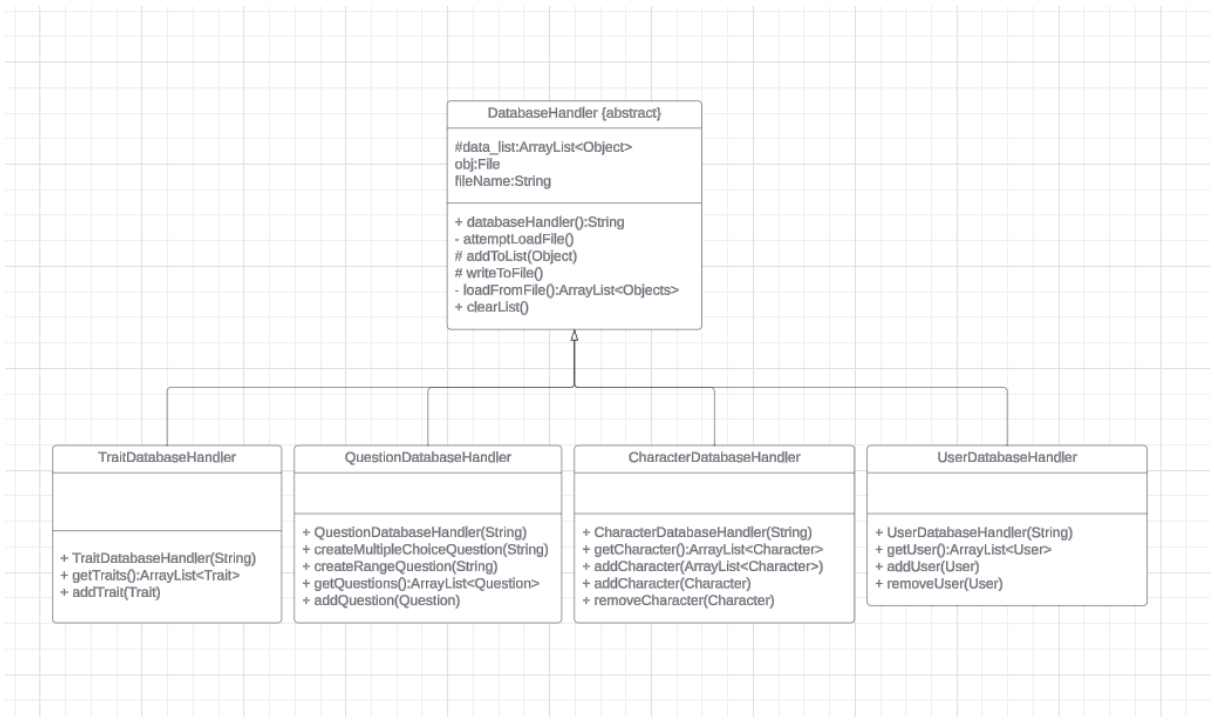
3.2.3 Question diagram



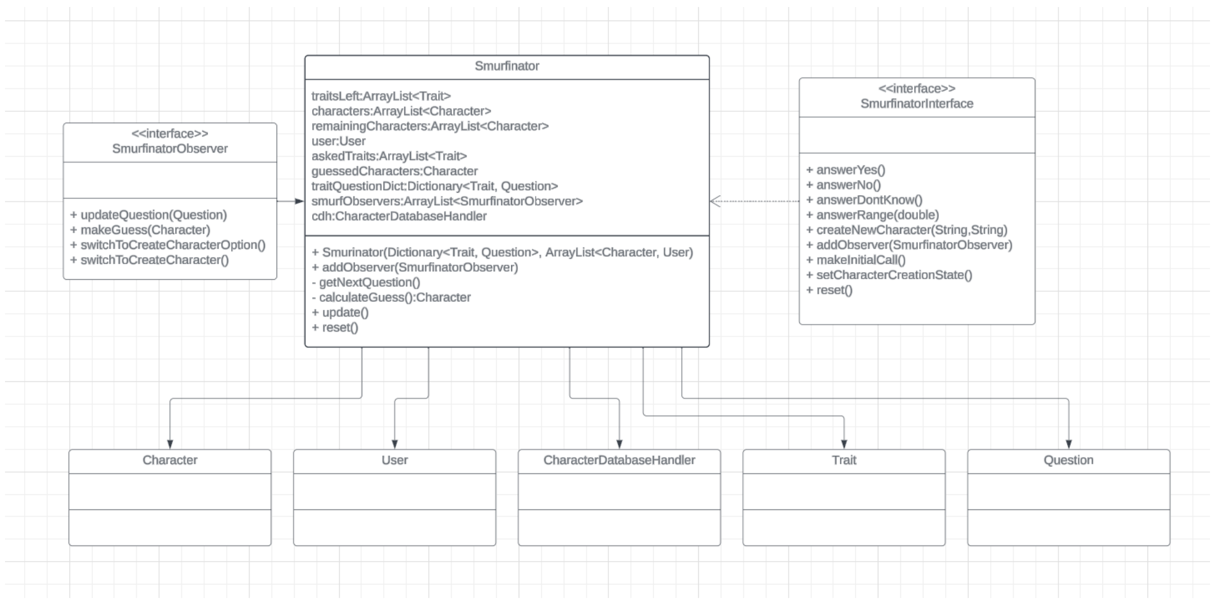
3.2.4 Leaderboard diagram


<Interface>
LeaderboardInterface
+ swtichCategory(int)
+ updateLeaderboard()
+ getLeaderboard():ArrayList<LeaderboardEntry>
+ addObserver(LeaderboardObserver)
+ makeInitialCall()
Leaderboard
- categories:ArrayList<Category>
- category:Category
characterLeaderboardCode:int = 0
userLeaderboardCode = 1
- leaderboardObserver:ArrayList<LeaderboardObserver> = new ArrayList<>()
+Leaderboard(ArrayList<Category>)
+addObserver(LeaderboardObserver)
+update()
+switchCategory(int)
<interface>
LeaderboardEntry
'+ getSortValue():int
+ getName():String
<interface>
LeaderboardObserver
+ updateLeaderboardField(Category)
Category {abstract}
#name:String
#categoryCode:int
#List:ArrayList<LeaderboardEntry>
+ getName():String
+ getList():ArrayList<LeaderboardEntry>
+ sort()
UserLeaderboardEntry
- user:User
- sortValue:int
- name:String
+ UserLeaderboardEntry(User, int)
+ getName:String
+ getSortValue:int
CharacterLeaderboardEntry
- cahracter:Character
- sortValue:int
- name:String
+ CharacterLeaderboardEntry(User, int)
+ getName:String
+ getSortValue:int
CategoryUser
+CategoryCharacter(ArrayList<UserLeaderboardEntry>, int)
+ charListToLbElist(ArrayList<UserLeaderboardEntry>): ArrayList<LeaderboardEntry> +
getList():ArrayList<LeaderboardEntry>
CategoryCharacter
+CategoryCharacter(ArrayList<CharacterLeaderboardEntry>, int)
+ charListToLbElist(ArrayList<CharacterLeaderboardEntry>): ArrayList<LeaderboardEntry>
+ getList():ArrayList<LeaderboardEntry>

3.2.5 Database diagram


DatabaseHandler {abstract}
#data_list:ArrayList<Object>
obj:File
fileName:String
+ databaseHandler():String
- attemptLoadFile()
# addToList(Object)
# writeToFile()
- loadFromFile():ArrayList<Objects>
+ clearList()
TraitDatabaseHandler
+ TraitDatabaseHandler(String)
+ getTraits():ArrayList<Trait>
+ addTrait(Trait)
QuestionDatabaseHandler
+ QuestionDatabaseHandler(String)
+ createMultipleChoiceQuestion(String)
+ createRangeQuestion(String)
+ getQuestions():ArrayList<Question>
+ addQuestion(Question)
CharacterDatabaseHandler
+ CharacterDatabaseHandler(String)
+ getCharacter():ArrayList<Character>
+ addCharacter(ArrayList<Character>)
+ addCharacter(Character)
+ removeCharacter(Character)
UserDatabaseHandler
+ UserDatabaseHandler(String)
+ getUser():ArrayList<User>
+ addUser(User)
+ removeUser(User)

3.2.6 Smurfinator diagram


Smurfinator
traitsLeft:ArrayList<Trait>
characters:ArrayList<Character>
remainingCharacters:ArrayList<Character>
user:User
askedTraits:ArrayList<Trait>
guessedCharacters:Character
traitQuestionDict:Dictionary<Trait, Question>
smurfObservers:ArrayList<SmurfinatorObserver>
cdh:CharacterDatabaseHandler
+ Smurinator(Dictionary<Trait, Question>, ArrayList<Character, User)
+ addObserver(SmurfinatorObserver)
- getNextQuestion()
- calculateGuess():Character
+ update()
+ reset()
<<interface>>
SmurfinatorObserver
+ updateQuestion(Question)
+ makeGuess(Character)
+ switchToCreateCharacterOption()
+ switchToCreateCharacter()
<<interface>>
SmurfinatorInterface
+ answerYes()
+ answerNo()
+ answerDontKnow()
+ answerRange(double)
+ createNewCharacter(String,String)
+ addObserver(SmurfinatorObserver)
+ makeInitialCall()
+ setCharacterCreationState()
+ reset()
Character
User
CharacterDatabaseHandler
Trait
Question

3.3 Design patterns

3.3.1 Observer pattern

We have used the observer pattern for communicating between our view, controller and model. The different model classes each have a list of observers that we use to send signals to the view. And in the same view when a button is pressed the WindowHandler makes the controller send signals to the model through its observers. With this we have managed to create as much separation as we could between the model view and controller.

3.3.2 MVC pattern

We have implemented the MVC pattern by creating a model that is smart, a controller that only sends signals and a view that only takes signals and only handles their own corresponding logic. The model is the only part of the program that does any type of model logic. Which is a sign of a well built program that follows the MVC pattern.

3.3.3 Strategy pattern

We have used the strategy pattern to easily change between different implementations of a class in Leaderboard. Leaderboard contains categories where the category used can either be a category of users or a category of characters.

3.3.4 Factory pattern

We use the factory pattern when creating characters to not have to add code that does not belong in the files where characters are created. An example is the method that creates traits from strings which belong in the factory but not elsewhere in the model.

4 Quality

4.1 Tests

For tests we have used JUnit.
A large number of files in this program are view and controllers which are not subject to the same testing as the model, which is why they are not tested with JUnit.

4.2 Known issues

- If the player answers wrong on one question the character that they might have thought of will be removed instead of still being able to be guessed on.
- Compendium is not connected to a user and does not show the characters that the user has found.
- Users are never updated with the amount of contributions.

5 References

- JavaFX - used to implement the graphical user interface.