PROJECT LINGUAL REFACTOR

SWE 4302
Object Oriented Concepts Lab

Hasin Mahtab
210042174
Department of CSE
B.Sc in Software Engineering

January 19, 2024

Abstract

LinguaL is simple language learning application built on .NET Framework of C#using the WPF technology. It can handle multiple users as well as multiple languages. The idea was inspired from the widely used ”DuoLingo”.

Contents

1 Project Overview

LinguaL is a comprehensive language learning application based on what language a user wants to learn. We have study materials for different languages as Lectures stored in text files and we have exams for each lecture to progress through the language. These materials are stored using text files.

The application can handle multiple users and can track the progress of all the users via utilzing text files as the primary database. We store user information and progress in the text files and also handle loading progress into the application.

2 Previous Project Structure

As we have to handle multiple users, we want to be able to interact as the user who is currently logged into the system. We need to maintain a session for the logged in user. For the previous implementation, I used the Static class and Static attributes. We had static attributes which defined the currently logged in user by taking in the username as string from the login textbox, this way we could have the same user set as the value for the "Global Username" as the static attribute, which were active throughout the whole project and could be changed or updated from any class or method.

We also implemented the multiple language in the same manner. As we have the user and languages stored in their respective directories and text files, we try to access those files using the global strings that have been set for the current session of the application while we log in or set the language we want to learn.

While this implementation was functional, it did not have the good practices of the object oriented concepts for a language like C#. Rather, the Static keyword is considered the contradiction for any Object oriented language out there. As we can mutate the static attributes from anywhere in the project, it destroys the many purposed of the object oriented concepts, such as : Encapsulation and Abstraction. So, to fix these problems we need to refactor the codebase so that it implements the SOLID principles of OOP. We will also try to refactor some of the design principles for our application.

2.1 UML Diagram

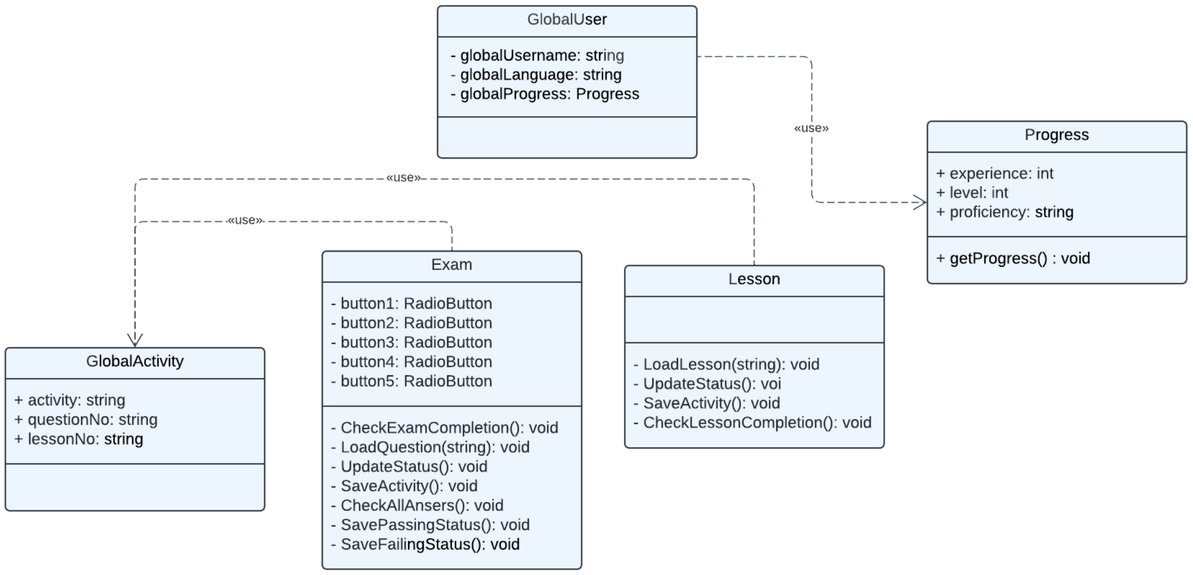

Figure 1: Unrefactored UML Diagram

3 SOLID Principles

SOLID principles are a means to ensuring that codes adhere to the standards for creating code, making it readable, manageable, and scalable. The fundamental notion behind the principles is -

- **Single Responsibility Principle (SRP):** A class should only have one cause to change, i.e. one obligation or job. This principle encourages developers to design classes that have a clear and specific purpose.
- **Open/Closed Principle (OCP):** Software aspects (such as classes, modules, and functions) should be extensible but not modifiable. This encourages using inheritance and interfaces to provide new functionality instead of altering old code.
- **Liskov Substitution Principle (LSP):** Superclass objects should be replaceable with subclass objects without impacting programme correctness. In other words, a derived class should be able to substitute for its base class without affecting the program's behaviour.
- **Interface Segregation Principle (ISP):** A class should not have to implement interfaces that it does not need. Instead of a large, monolithic interface,

it is desirable to have smaller, more specific interfaces that are only implemented by the classes that need them.

- **Dependency Inversion Principle (DIP):** High-level modules should not rely on low-level modules; instead, they should rely on abstraction. Abstractions should not be dependent on specifics, but rather the reverse. This approach encourages the use of dependency injection and inversion of control to decouple components and increase system flexibility.

Following these standards allows developers to create more modular, scalable, and maintainable software systems. SOLID principles help to make software easier to design, maintain, and expand, resulting in more robust and adaptable solutions.

For the LinguaL Project, we will be focusing on the following principles: single responsibility principle, open/closed principle, interface segregation principle, and dependency inversion principle. Not all of them could be properly implemented because some parts of the code were restricted by the UI elements, but I attempted to implement the business logic using these guidelines.

4 Applying SOLID to LinguaL

4.1 Single Responsibility Principle

The previous project structure used the business logic and text file related logic in the same class files. This violates the rule of SRP, which states that we should not have more than one reason or stakeholders to change a class and its methods. So we implemented SRP by separating the concerns from the single classes to separate classes. We have created separate directories for the different types of class:

- In the **Handler** directory, we have the file handler classes that directly work with the text files to read and write the data for the user data, lessons and exam files and progress.

- In the **Model** directory, we have the classes and interfaces related to the core business logic of the application that takes care of the User, Activity, Progress and Exam results.

- In the **Utiliy** directory, we have the extra helper classes for the exam management, password hashing and user authentication.

- In the **View** directory, we have the XAML files along with the UI related logic for each XAML file, which handle the logic for the UI elements itself.

This makes sure that all classes are organized according to their responsibilities. As for the classes, each class only has methods related to the one core responsibility for that class, e.g., the ExamManagement class only checks the correct answers and calculates the exam result for the logged-in user. It does not read any data from the text files or save anything to the text files. So we are ensuring this class has only one reason to change in the future, which is to change how we calculate the result.

4.2 Open/Closed Principle

For the previous project, we had separate UI pages for each exam and lesson when we loaded them from the text files. And that ended up in writing lots of duplicate code for the same type of UI with some small changes in the UI page class file. Another violation of OCP was that if we wanted to add a different type of exam, we had to implement the whole NewExam class with almost identical method calls. To ensure that our project is open to extension without modification to the existing code, we implemented the following:

- **IExam**, an interface that implements methods for loading the questions from a text file to the UI and also checks the UI inputs, comparing them using the ExamManagement class to calculate the results.

- **ILesson**, an interface that implements methods for loading the lessons from a text file to the UI and saves the completed lessons.
- **IActivity**, an interface that implements methods to save the users current activity, which might be an exam or a lesson that the user has taken, and using the file writer handler, writes the activities to a text file, which can later be loaded into a calendar using the file reader handler.
- **IStatus**, an interface that implements methods to update the status for an exam or a lesson taken by the current user.
- **ISaveable**, an interface that saves and updates the passing or failing status of a user.

Now, if we want to extend the types of exams, we just need to implement these interfaces, which are needed for the new type of exam and then define the methods as necessary for that type of exam, this means, we don't need to modify anything from the existing exam type classes.

4.3 Liskov Substitution Principle

For this project, as we are not using any Inheritence from base type to child type, we did not implement anything that might follow the Liskob Substitution Principle.

4.4 Interface Segregation Principle

For the exam and lesson UI class,we have implemented interfaces, but we had to break down the interfaces into more specific interfaces so that we maintained the SRP in the interfaces. We could use the SavePassingProgress and SaveFailginProgress to the IStatus interface, but then when we implement that to the ILesson, we would also need to impement those methods, but there are no SavingProgress in the lesson page. So we separated the interface IStatus into IStatus and ISaveable so that we can implement them in the lesson and exam when needed.

4.5 Dependency Inversion Principle

For the project, we are depending on which user is currently logged in and instead of creating the user object inside the constructor for the classes, we are passing the user object from the initial login and authentication class, which ensures we have an intermediate abstraction so that the higher-level modules do not directly depend on the lower-level modules. We are using dependency injection to make sure that DIP is implemented.

We are also using dependency injection in other class constructors to pass the instances of different text file handlers and other obejct instances. So that we are not dependent on the concrete class.

4.6 UML Diagram

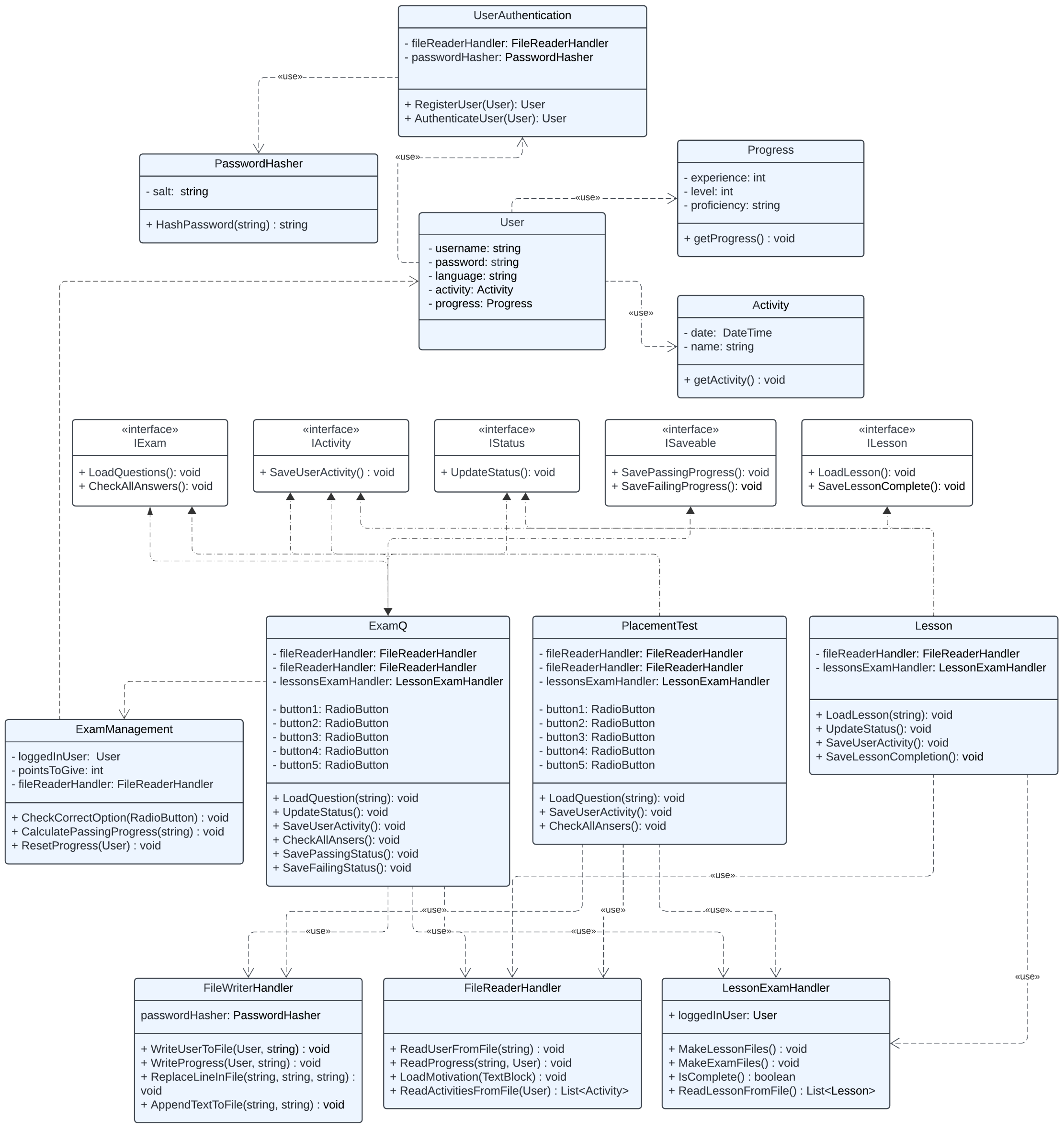
UserAuthentication
- fileReaderHandler: FileReaderHandler
- passwordHasher: PasswordHasher
+ RegisterUser(User): User
+ AuthenticateUser(User): User
«use»
PasswordHasher
- salt: string
+ HashPassword(string) : string
Progress
- experience: int
- level: int
- proficiency: string
+ getProgress() : void
User
- username: string
- password: string
- language: string
- activity: Activity
- progress: Progress
Activity
- date: DateTime
- name: string
+ getActivity() : void
«interface» IExam
+ LoadQuestions(): void
+ CheckAllAnswers(): void
«interface» IActivity
+ SaveUserActivity() : void
«interface» IStatus
+ UpdateStatus(): void
«interface» ISaveable
+ SavePassingProgress(): void
+ SaveFailingProgress(): void
«interface» ILesson
+ LoadLesson(): void
+ SaveLessonComplete(): void
ExamQ
- fileReaderHandler: FileReaderHandler
- fileReaderHandler: FileReaderHandler
- lessonsExamHandler: LessonExamHandler
- button1: RadioButton
- button2: RadioButton
- button3: RadioButton
- button4: RadioButton
- button5: RadioButton
+ LoadQuestion(string): void
+ UpdateStatus(): void
+ SaveUserActivity(): void
+ CheckAllAnsers(): void
+ SavePassingStatus(): void
+ SaveFailingStatus(): void
PlacementTest
- fileReaderHandler: FileReaderHandler
- fileReaderHandler: FileReaderHandler
- lessonsExamHandler: LessonExamHandler
- button1: RadioButton
- button2: RadioButton
- button3: RadioButton
- button4: RadioButton
- button5: RadioButton
+ LoadQuestion(string): void
+ SaveUserActivity(): void
+ CheckAllAnsers(): void
Lesson
- fileReaderHandler: FileReaderHandler
- lessonsExamHandler: LessonExamHandler
+ LoadLesson(string): void
+ UpdateStatus(): void
+ SaveUserActivity(): void
+ SaveLessonCompletion(): void
ExamManagement
- loggedInUser: User
- pointsToGive: int
- fileReaderHandler: FileReaderHandler
+ CheckCorrectOption(RadioButton) : void
+ CalculatePassingProgress(string) : void
+ ResetProgress(User) : void
FileWriterHandler
passwordHasher: PasswordHasher
+ WriteUserToFile(User, string) : void
+ WriteProgress(User, string) : void
+ ReplaceLineInFile(string, string, string) : void
+ AppendTextToFile(string, string) : void
FileReaderHandler
+ ReadUserFromFile(string) : void
+ ReadProgress(string, User) : void
+ LoadMotivation(TextBlock) : void
+ ReadActivitiesFromFile(User) : List<Activity>
LessonExamHandler
+ loggedInUser: User
+ MakeLessonFiles() : void
+ MakeExamFiles() : void
+ IsComplete() : boolean
+ ReadLessonFromFile() : List<Lesson>

Figure 2: Refactored UML Diagram

5 Fixing Design Smells

The design smells that were fixed in this project are

- **Rigidity:** The classes were tightly coupled making it hard to change any part of the code. Now after implementing SRP and OCP the code is now more loosely coupled.
- **Immobility:** Any part of the code that is a useful method could not be used in any other class. But now after implementing SRP and ISP, we can just call those methods from the object instances.
- **Fragility:** Any part of the code would break due to dependency of classes. Implementing DIP with dependency injection made the classes less dependent on each other and we can easily change or modify a class without breaking the whole application.
- **Needless duplication:** Now that methods can be called from any object instance, we don't need to write duplicate code for different classes.

References

- LinguaL - Main Branch
- LinguaL - Refactor Branch