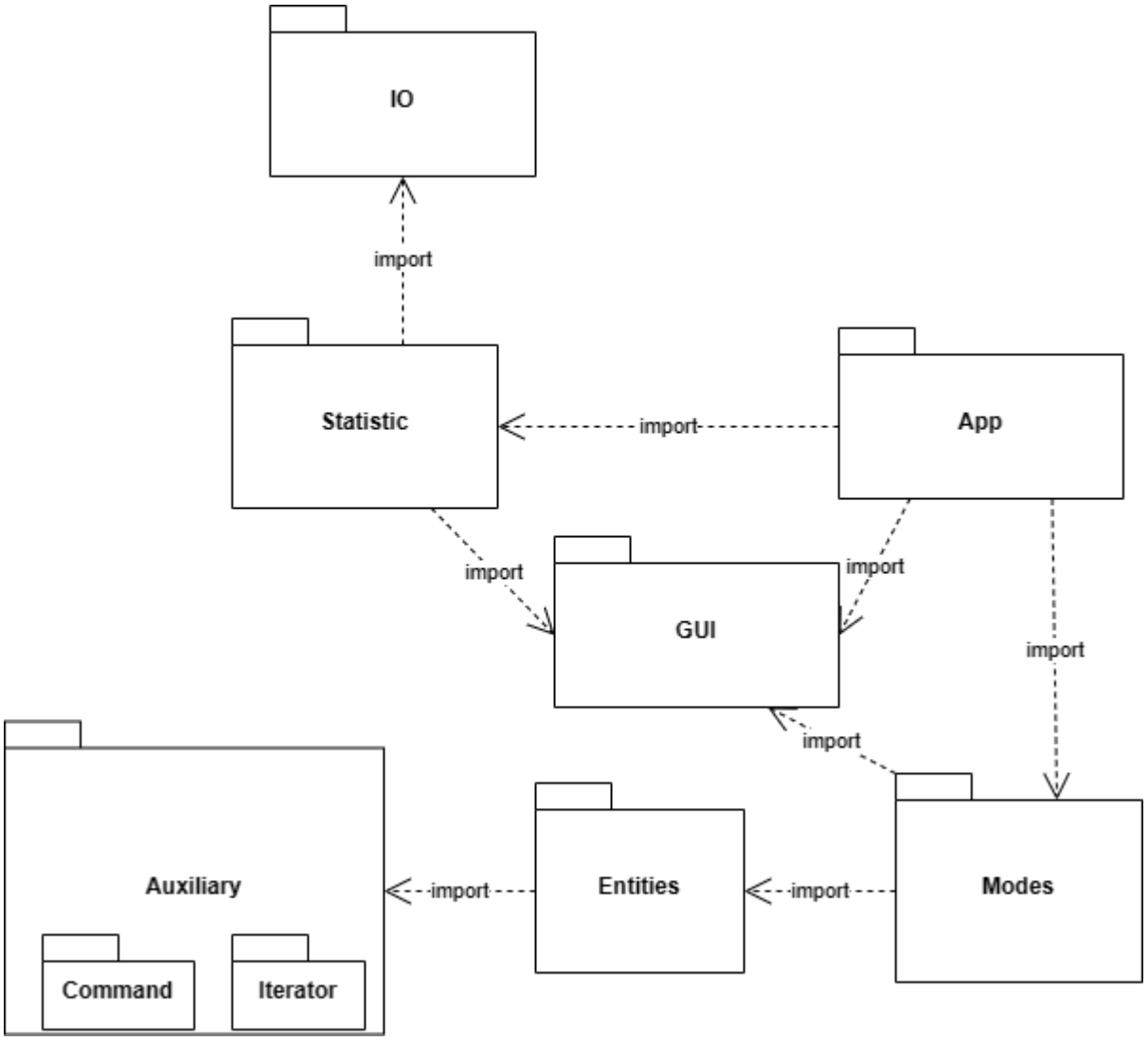
Package diagram



The strategy we used to organize our package is by layers. Each package represents different layers in our system.

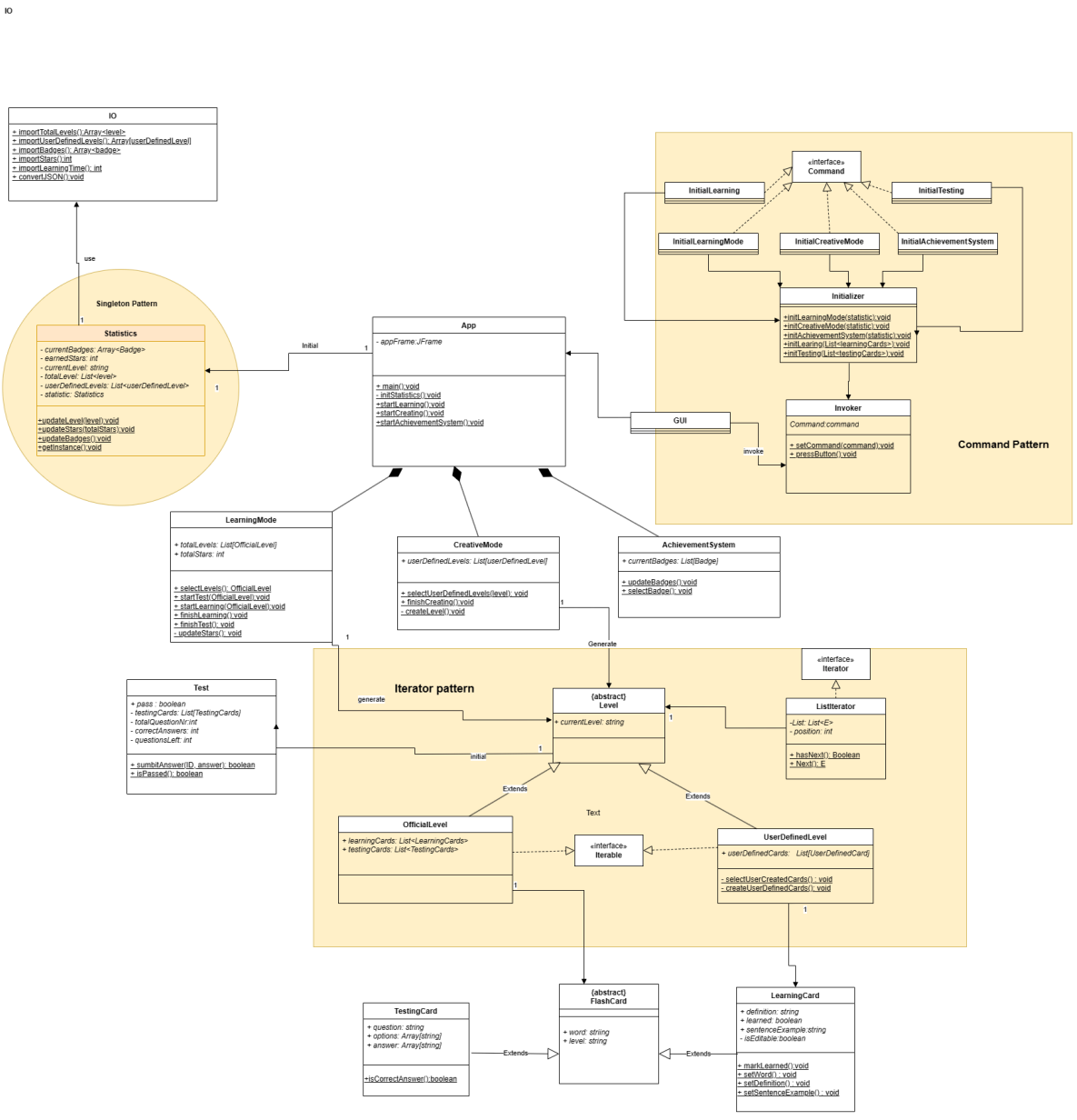
The App package contains the main class in our system as an entry point, it needs a Statistic package to provide data for it, and it consists of different Modes, finally, it needs GUI to show the main menu, so the App package imports these three packages.

The statistics package needs IO's function to import data from a JSON file, so it imports an IO package. Also, it needs a GUI package to show the data inside it, it imports GUI too.

The Modes package all the main modes in our system, and these modes consist of entities(like flashcards, and different difficulty levels), so it imports the entities package, and it also needs GUI to show different modes on the screen, so it imports GUI.

The entities package includes all the basic content in our system, like flashcards, difficulty levels and tests in each level, and we design some patterns to implement it including the Command pattern, the Iterator pattern. To implement these patterns, we need some interface and class. We put them in the auxiliary package (which contains two sub-packages to implement the iterator and command pattern). So the entities package imports the auxiliary package.

Class diagram



Statistics class

We made our **Statistics class** using a singleton pattern, so we changed all of its attributes into private, and added a *statistic* object as the singleton instance. And accordingly, we add a getInstance() operation in order to get the singleton object. And because it has an associated with **App**, so the mode as composition of **App** can access its data by getInstance().

Iterator interface, **ListerIterator class** and **iterable interface**.

Our **officialLevel** and **UserDefinedLevel** all work like a deck, because they all contain lists of flashcards of their own level. So we get inspiration from the lecture, we decide to use an iterator pattern to iterate all the flashcards inside each level.

We created an iterator interface which includes hasNext() and Next() operations and we made a **ListIterator** as our concrete iterator, which contains a *list* and int *attribute* position. And we decide to adapt E as the type of elements inside the list. With this general type, we can apply our iterator into different lists (we have lists containing different types, like learning Cards and testing cards).

Then we made an **Iterable** interface which contains an *iterator*. And we let our OfficialLevel and **UserDefinedLevel** implement this **Iterable** interface, so that it can iterator their own lists of flashcards.

Command Interface, five different Commands, **Initializer** and **Invoker class**.

Our system is based on GUI, so when a user clicks a button, some corresponding actions will execute. We decided to use a command pattern to implement this.

First, we create a **Command** interface which contains an operation called execute(), and we create five different classes to represent five different actions after a user presses these buttons, each of them containing an execute(), to call different operations in the Initializer class. An **Initializer** class works as a receiver which will actually execute different commands. It contains five different operations inside it.

And we have an **Invoker** class associated with our GUI, which will associate actions on GUI with real logic operations in our system, it sets commands with setCommand(command) operation and does it with pressButton() operation.

So after adapting this pattern, we remove the initialization operation in the other class in our previous diagram, now they are in the **Initializer** class.

Some operations in App class

Actually some operations in App class(like startLearning():void) are placed into the controller of the main page, but logically they belong to the App class.

Statistics class works as a database in our system which will provide all the data other modes need from JSON files. So, each of the three main modes will use it, and all the data should be consistent so that one mode changes some of it, and the other mode can also get the latest data. If every mode makes a different object when it is running, it doesn't have memory efficiency and has a big chance of messing up the database. So we need all the data that is easy to access, reduce the times of its initialization in order to reduce mess, and keep data consistent.

So, we decided to make our **statistics** class as a singleton pattern. It only needs to be initialised once whenever it is first created by the getInstance() operation. It works as a global access object, each mode can access it without causing side effects and messing up the data. After one mode runs over, all the data will be kept inside the single object, and the other mode can access the latest data. And it also promotes the Single Responsibility Principle.

Whenever a user clicks a button to enter a mode, all the data can be accessed by getInstance() and if it is the first call, in the constructor, the instance object will import all the data from the JSON file. After any specific session (like taking a test, or finishing learning), each mode will execute the singleton object's update functions (it works like a setter to update new data).

The design pattern does not impose any constraints on the system. It is really a useful and good design pattern fitting our system.

In some sessions of our system, we need to iterate through a collection of cards, like in the learning or testing. But first, we need to design a nextCard() operation under our **level** class. But to do it, it makes the abstract class overweight, and adds some unrelated variable (like we need a variable to track the position of these cards). And it violates the Single Responsibility Principle.

Instead, we decided to apply an Iterator pattern to solve our problem. It encapsulates the independent logic of iterating elements of a collection into a separate set of interfaces and classes, it complies with the Single Responsibility Principle and is good for maintenance.

We let our **OfficialLevel** and **UserDefinedLevel** to implement the iterator pattern, because they work like decks in our system. Whenever a user starts a learning or takes a test, the

iterator inside this class object will use the iterator to show the whole collection of this level with the user clicking the “next card” button on GUI.

The only constraint of it is it may introduce performance overhead compared to direct iteration over a collection. And it increases memory consumption.

We design using GUI in our system. We use JavaFx to implement our GUI, and we use FXML files to set the layout of a stage and use controllers to add listeners. When a user clicks a button to enter a mode, we hope to add a listener to set up the page of that mode and pass some data. At first, we do such operations inside the controller, but it makes our controller cluttered. So we want to decouple them, using a new set of classes to implement these commands.

Now, we decide to take Command Pattern, associate all the buttons with a specific Command, and use an invoker to pass these commands into an initializer to do such a command. Through this way, we decouple the sender(pressing button on GUI) and receiver(our initialization operations), it promotes a more flexible design where the sender doesn’t need to know anything about the receiver's interface or implementation details, reducing code redundancy and increasing readability.

We associate specific commands with different buttons. Like when a user clicks the “learning mode” button, we add an event listener to it. It first creates an invoker object, and the invoker sets the specific command of this button (initLearningMode(statistic):void), and lets our Initializer object execute this specific command. So now the learning mode is set up, and needed data will show in the learning mode window.

It doesn’t not impose too many constraints in our current system. It adds a little bit of complexity and memory overhead. And with the system scale up, it may increase some testing complexity.

State machine diagrams

Learning Mode class

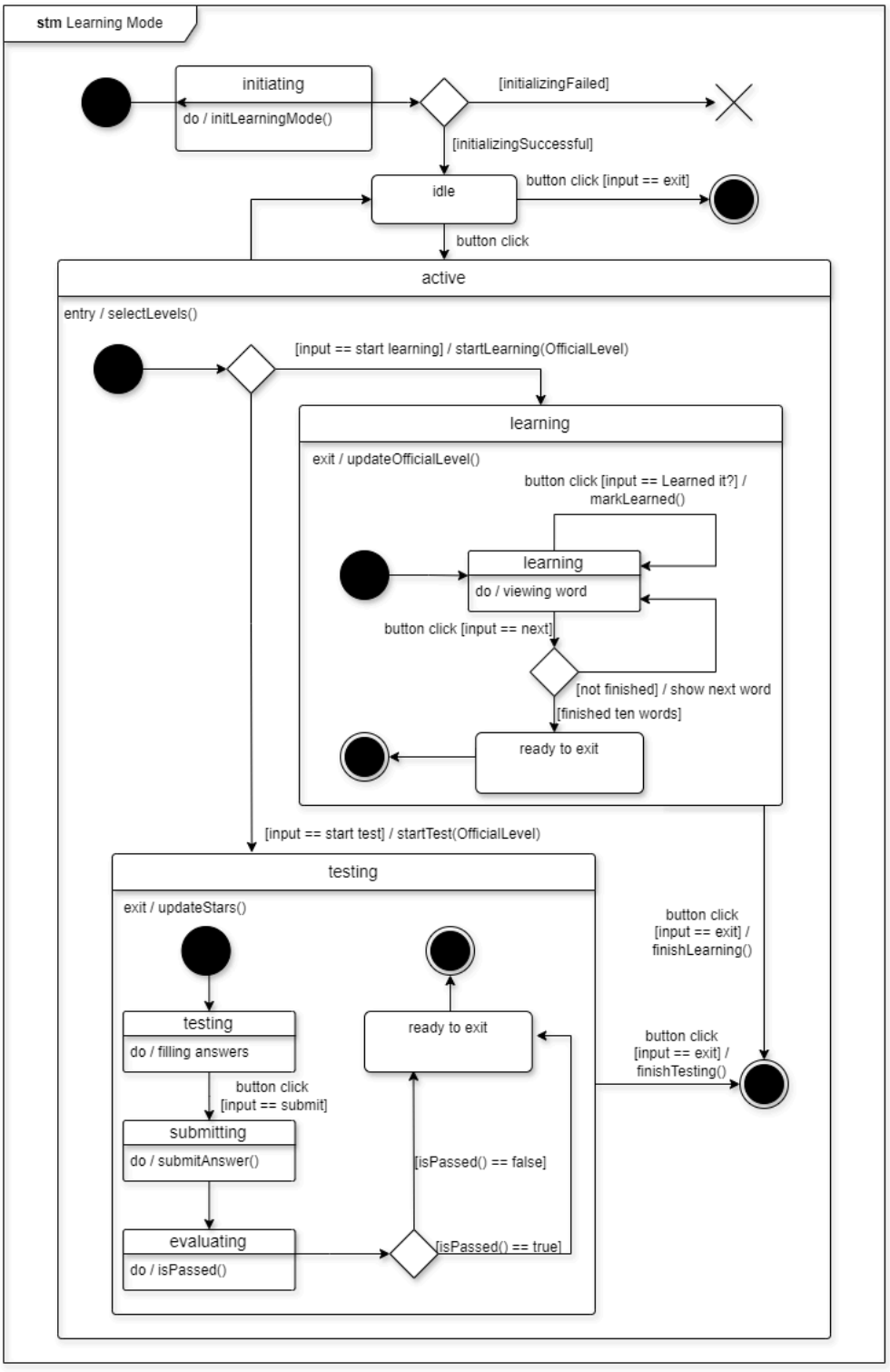


Figure 3: State Machine Diagram for Learning Mode class

Before starting, the learning mode is initialized in the initializing state by initLearningMode(). If initialized successfully, the user shall be able to start learning or take a test from the home menu of the learning mode, starting with the ~~initial~~ state idle. If not successful, indicated by [initializingFailed], the program terminates. If the user decides not to proceed with learning or testing, they may click on the *"exit"* button, which is the trigger event that initiates the transition without any additional activities, leading to the final state. When the user transits from idle state to active state, as indicated by the internal activity, the user first has to choose between the three levels (easy, medium and advanced).

If the user wants to learn, they may click on the *"start learning"* button, which triggers the event startLearning(OfficialLevel), leading their entry into the composite state learning. Within this state, the user starts at the initial state, proceeding the the inner state with the same name learning. At this moment, the user is currently looking at the menu of this state, with two buttons available, *"learned it?"* and *"next"*. The user may click on *"learned it?"* as a trigger event to mark the word as learned, which ensures it does not appear in the tests. The user may also click on *"next"* to proceed viewing the next word, which is the trigger event that leads to a decision node. If all ten words are finished, it proceeds to the next state ready to exit; if not, it transits back to learning state and the next word is shown. When exiting the composite state learning, updateOfficialLevel() is executed.

Once the user feels confident, they shall be able to click on the *"take the test"* button, which triggers the event startTest(OfficialLevel), transiting the user to the composite state testing. The user first has to fill in answers for all the questions in testing state, then with the trigger event button click [input == submit], the user transits to the submitting state and submits the answers with submitAnswer(). After these, in the evaluating state, isPassed() is called and the answers are evaluated. The following is a decision node. If the user answers all the ten questions correctly, they obtain a star and is permitted to unlock the next level, which is indicated by the transition [isPassed() == true]; if not, they fail the test, indicated by [isPassed() == false]. Both transitions lead to the state ready to exit, followed by the final state. When exiting the composite state testing, updateStars() is executed.

It is important to note that during both learning and testing session, the user can choose to exit at any point by clicking the *"exit"* button, which is represented in the diagram by the transition connecting the composite state learning and testing and the final state of the active state. So, the two states "ready to exit" serve no actual function but only showing that the object may remain inside forever waiting for the trigger event button click [input == exit].

App class

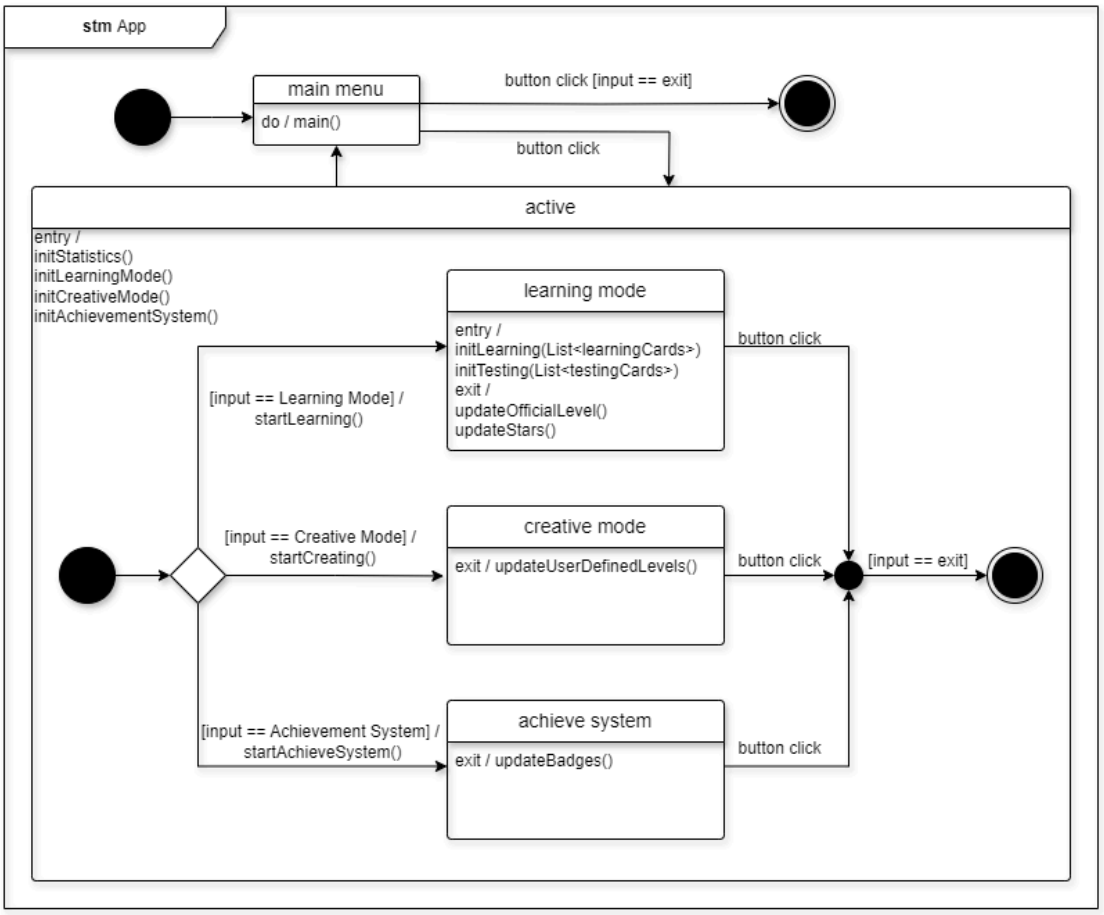


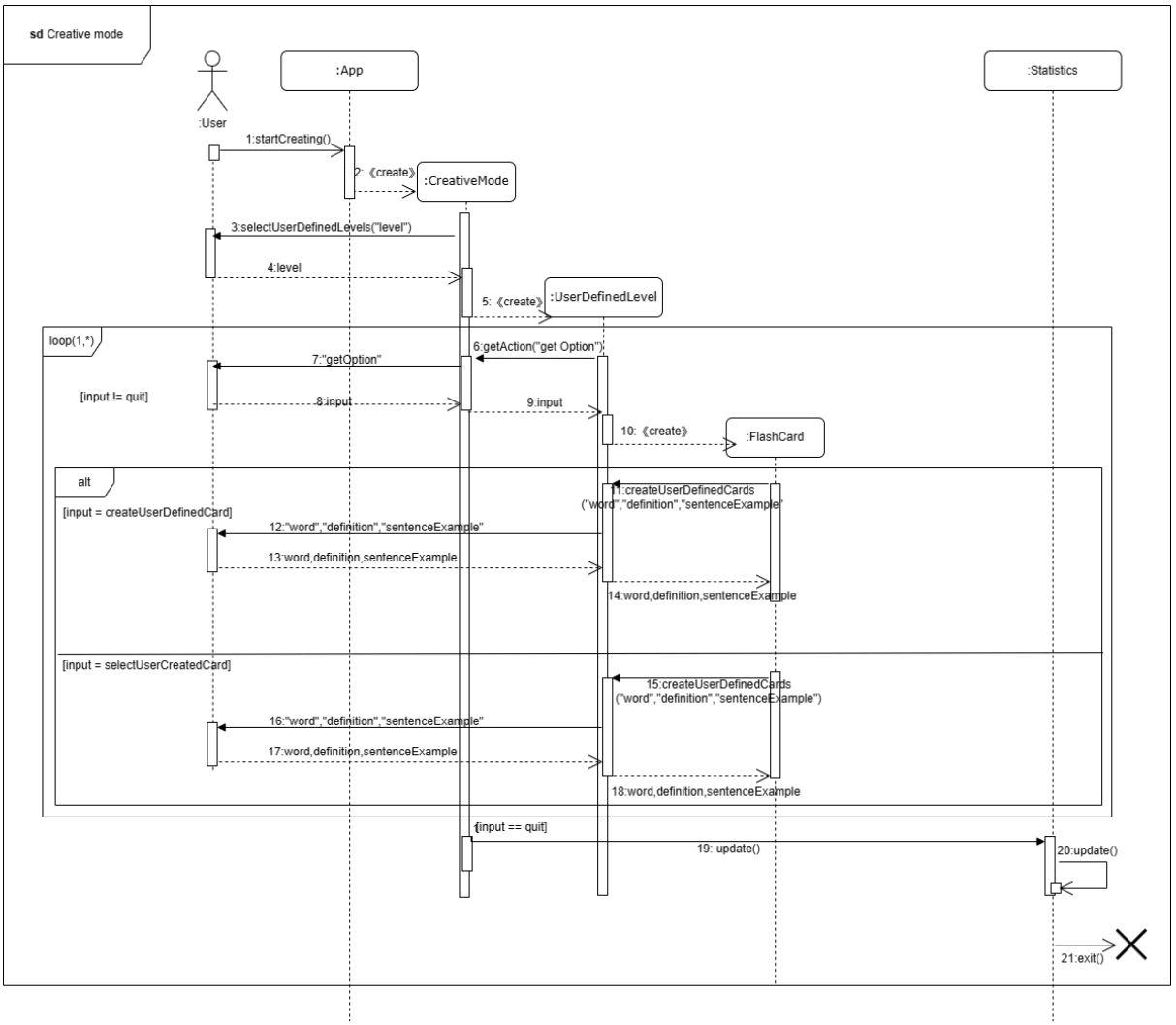
Figure 4: State Machine Diagram for App class

Starting from the state main menu, the user can exit the app by clicking on the *"exit"* button, indicated as button click [input == exit] or click on buttons for three features to start using the app. A click on one of the buttons will lead the user to active state, when entering, initStatistics(), initLearningMode(), initCreativeMode() and initAchievementSystem() will be executed for initializing the functions..

The three states, which are learning mode, creative mode and achievement system, can be accessed via three transitions: [input == Learning Mode] / startLearning(), [input == Creative Mode] / startCreating() and [input == Achievement System] / startAchieveSystem(), following a decision node. When entering the state learning mode, specified initializations are executed, which are initLearning(List<learningCards>) and initTesting(List<testingCards>). When exiting the three states, updating functions are executed to update the new data to the **statistics**, which are updateOfficialLevel() and updateStars() for learning mode, updateUserDefinedLevels() for creative mode and updateBadges() for achievement system. It is important to note that, within the three states, since this diagram serves for presenting the process of the entire app, detailed internal activities are not described.

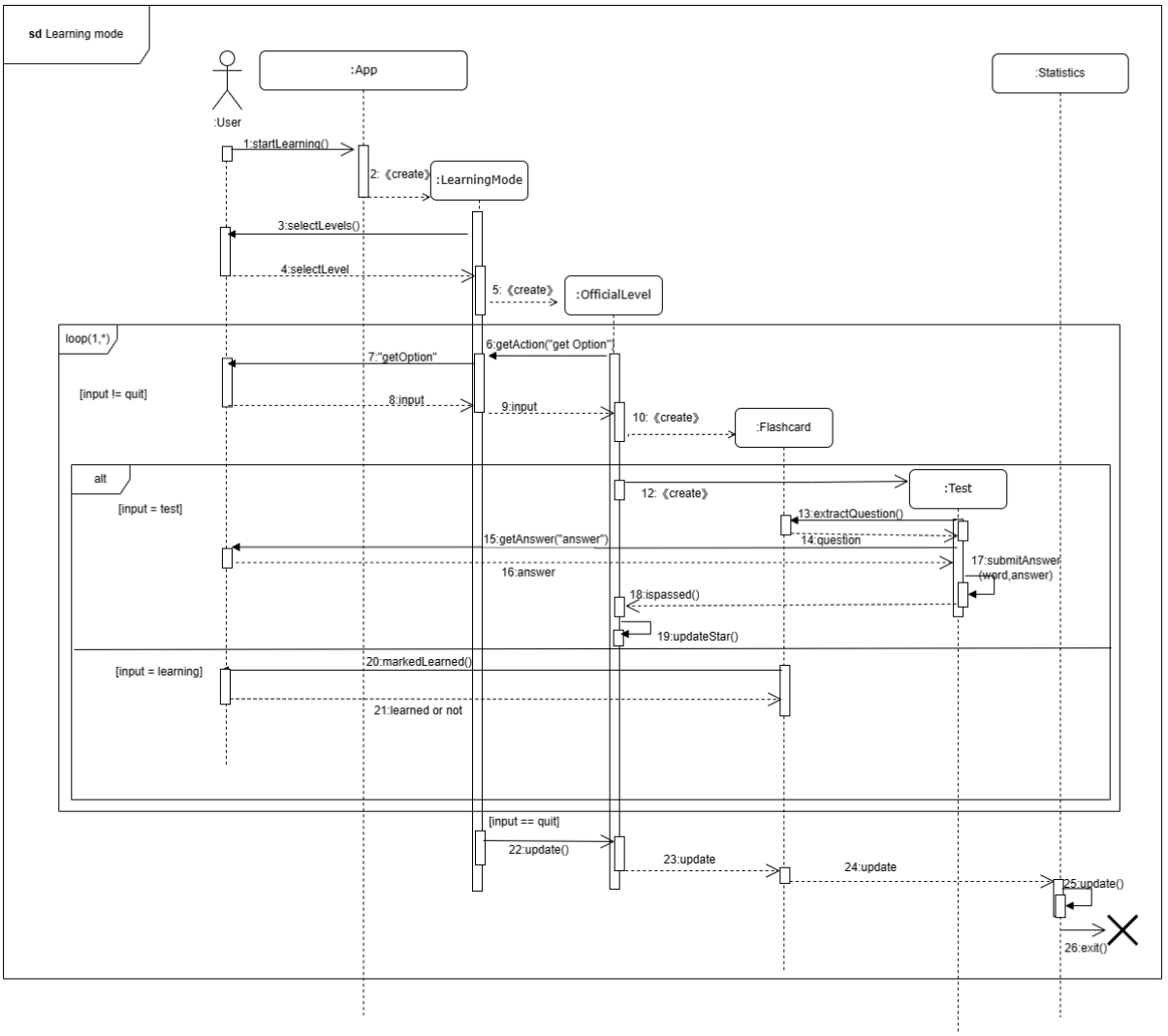
To exit the current modes, the user can always click on the button *“exit”* as the trigger event indicated as button click [input == exit], to get to the final state of the composite state active, followed by the return to the main menu state.

Sequence diagrams



The creative mode sequence diagram illustrates the communication sequence between a User and instances of the classes **App**, **Creativemode**,**OfficialLevel**, **FlashCard** and **Statistics** following a successful startCreative() start. This message triggers the initiation of the CreativeMode class, where the User is prompted to input their next action. This prompts the interaction between **CreativeMode** and User through the selectUserDefinedLevel(“level”) method, with the result being stored in the input variable.

Using this variable, a **OfficialLevel** class is initialized. The main loop in the **CreativeMode** class then prompts the user for the action they wish to perform, offering three options: selectUserCreatedCards(), CreateDefinedCards(), and quit(). The selectUserCreatedCards() method allows the user to modify previously created flashcards, while CreateDefinedCards() enables the addition of a new flashcard into the Array<UserCreatedCards>, both of whose parameters are “word” ,“definition” and “sentenceExample” .These two methods trigger interactions between the User and the **FlashCards** class. In the end, when the loop ends, the data in **Statistics** is updated via the update() from **CreativeMode** class. The sequence ends by exit().



This sequence diagram illustrates the communication flow between a User and instances of the classes **App**, **LearningMode**, **Level**, **Test** and **Statistics** after the user has successfully chosen to startLearning() .This message triggers the initiation of the **LearningMode** class, where the User is prompted to input their next action. Subsequently, there's an interaction between **LearningMode** and User facilitated by the selectLevel() method, storing the result in an input variable. This variable is then used to initialize the **UserDefinedLevel** class. In

the main loop of the **LearningMode** class, the user is prompted for their preferred action, with options including startLearning(), startTest(), or quit(). The startLearning() method allows the user to read flashcards with the desired level,where they could decide the word is learn or not via markedLeared(), while startTest() enables them to take a test. When the user is taking a test, by extractQuestion() ,**Test** class could read questions from the **FlashCard** class and receive the “answer” via getAnswer(). Then isPassed() goes on in **UserDefinedLevel** with the information in submitAnswer(“word”,”answer”) out of the purpose of updateStar() in **Level**,which is used to decide the current level . A final update() takes place in **Statistics** when the user chooses to quit() and the exit() executes to end the sequence.