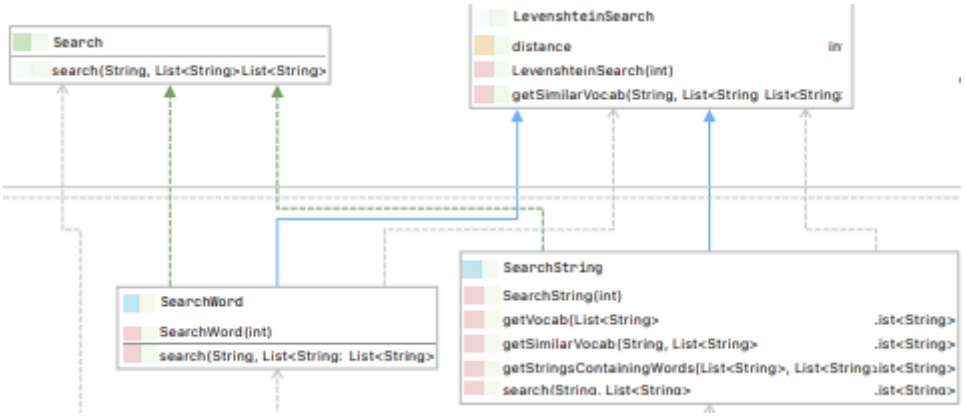
Design Patterns Implemented in the code

- **Strategy design Pattern**
- **Dependency Injection**
- **Builder**
- **Factory design Pattern**

Strategy design Pattern

Package: Algorithms Package

File: LevenshteinSearch, Search, SearchString, SearchWord



How

We have two search method which we want to implement: SearchWord and SearchString. Both are the child class of LevenshteinSearch class. Search method has two parameters the input which is taken as a list and the options which is also taken as a list.

This is how the searching happens

We have to handlers for searching in the Conference Controller class: handleSearch and handleSearchUser.

We create SearchString object in handleSearch because it checks for Events and Messages which is general. Then we call the search method which goes to the Search Interface and perform the search algorithm in SearchString. This method searches for **all** the Events or Messages which have a minimum loss of similarity which is found through the method distance in the levensheinSearch class

We create SearchWord object in handleSearchUser because it checks for one specific User to be found. Then we call the search method which goes to the Search Interface and performs the search algorithm in SearchWord. This method searches for **one** User to be found. It gets all the Similar vocab from the getSimilarVocab method which uses the levenshteinSearch to get the most Similar value. And this is returned as the output

Notice that these two search algorithms are implemented would return the most similar strings or string.

Why

This is the strategy design pattern. The idea behind using this is to have 2 search algorithms which are related but we want only one of them to be executed. For example, SearchWord is implemented in handleSearchUser and SearchString is implemented in handleSearch. One is a general approach,

and the other is a specific one. The principles which this design pattern protects is the Open Closed Principle. This is because we can easily change the implementation of the Client file which is Conference class and this change would not affect the changes in the algorithm package, as this is encapsulated. This design pattern shows interface Segretation of 2 different search algorithm and makes use of Liskovs Substitution principle as search method can be SearchStrings or SearchWord.

Dependency Injection

Classes: All classes except classes in the UI, MainMenu, Agorithms and Gateway package

How

Used in all almost all files For example the use of UUIDs and the use of Unique string for Users. We are going to take one example from User.java file in User class we use a method addEventId which takes in the EventId and stores it.

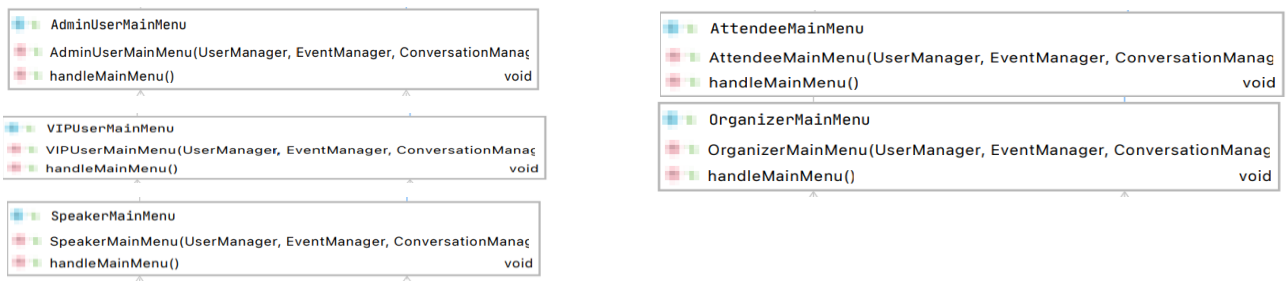
```
/**
 * Adds an event to the user's list of events
 * @param eventId: UUID of the event
 * @return boolean indicating if the action was successful
 */
public boolean addEventId(UUID eventId) {
    /*
    If eventId is not already in eventIds, then add eventId a
     */
    if (!this.eventIds.contains(eventId)) {
        this.eventIds.add(eventId);
        return true;
    }
    else {
        return false;
    }
}
```

Why

This is done because we want to decrease the coupling between two classes. Here by storing eventIds rather than event objects we are using abstraction. As we are removing the unnecessary methods which will not be used in the User class and its only purpose is to store eventIds. This is design pattern helps us to follow the single responsibility principle because the purpose of this method is to store eventIds and not its methods.

Builder

Classes: We use the builder design pattern in 4 classes different MainMenu Classes: AdminUserMainMenu, VIPUserMainMenu, AttendeeMainMenu, OrganizerMainMenu, SpeakerMainMenu



```java
/**
 * Displays the different menus for the different users
 * Organizers have 4 options: messages, Event Manager, Attendee and Speakers
 * Speaker have 2 options: messages and talks
 * Attendees have 2 options: Messages and Events
 * for each of these an input by the user is given to navigate to the new tab
 */
public void handleMainMenu() {
    try {
        List<String> adminOptions = Arrays.asList("Messages", "COVID Tracker", "Delete Empty Events",
                "View Archived Messages");
        presenter.printMenuRequest();
        presenter.printOptions(adminOptions);
        presenter.printIndexRequest();

        String input = parser();
        int index = checkInt(input, limit: 4);
        if (index == 0) { messenger.handleMessenger(); }
        if (index == 1) { userTracker.handleAdminTracker(); }
        if(index == 2){
            eventScheduler.handleDeleteEmptyEvent();
        }
        if (index == 3) {messenger.handleMessengerViewArchivedConversations();}
    } catch(Exception e) {
        handleMainMenu();
    }
}
```

We will show how and why we used it in AdminUserMainMenu Class

How

We use different instances in the AdminUserMainMenu Class, these include UserMessenger (messenger), UserTracker, eventScheduler.

First we set the different options which the user can see. For the Admin User these are:

- Messages (option 0)
- COVID Tracker (option 1)
- Delete Empty Events (option 2)
- View Archive Messages (option 3)

For each one we call different methods.

- For Messages we call the handleMessenger method from the messenger class instance variable. This method lets us create and delete messages sent to different users.
- For COVID Tracker we call the handleAdminTracker method from the userTracker class instance variable. This method lets us track, alert, mark and unmark different Users in the event
- For Delete Empty Events we call handleDeleteEvent method from the eventScheduler class instance variable . This method lets us event the events which have no attendee

Why

Notice this is done in a stepwise fashion and encapsulates the different implementations. This helps the programmer follow what is happening in the code very easily. And have Single Responsibility and Open Closed principles hold

We also observe that messenger (UserMessenger) has different actors such as Attendees and VIPUsers and AdminUsers. This design patterns helps us separate these actors my creating different Menus and the different methods which are required from different Users. Thus, we know that single responisibnty principle holds.

We can also add more options in this menu, so this class is also closed for modification but open for extension. So open closed principle holds.

Factory design Pattern

Classes: The classes where this pattern is used is in LoginReceiver and UserGenerator.

Creation of different Menus in the login Receiver file

How:

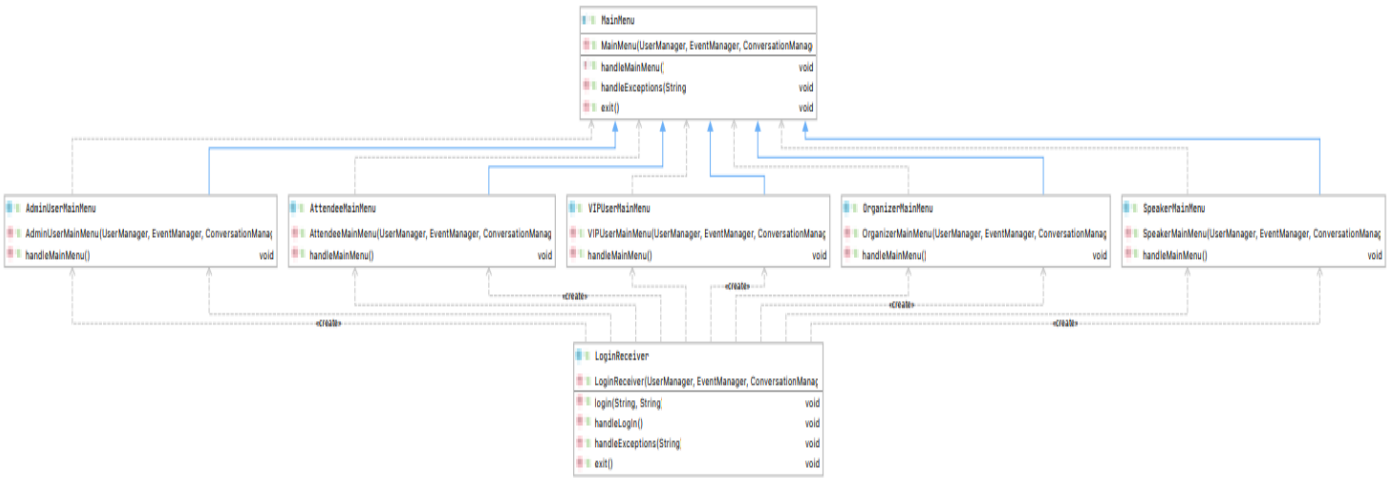
```
public void handleLogIn() {
    try {
        presenter.printUsernameRequest();
        String username = parser();

        presenter.printPasswordRequest();
        String password = parser();

        login(username, password);

    } catch(Exception e) {
        handleLogIn();
    }
}
```

```
public void login(String userid, String password) {
    if (userManager.findUserId(userid, password)) {
        User usr = userManager.getUser(userid);
        userManager.setCurrentUser(usr);
        if (usr.isOrganizer()) {
            OrganizerMainMenu organizerMainMenu = new OrganizerMainMenu(userManager, eventManager,
                    conversationManager);
            organizerMainMenu.handleMainMenu();
        } else if (usr.isSpeaker()) {
            SpeakerMainMenu speakerMainMenu = new SpeakerMainMenu(userManager, eventManager, conversationManager);
            speakerMainMenu.handleMainMenu();
        } else if (usr.isVIPUser()) {
            VIPUserMainMenu vipUserMainMenu = new VIPUserMainMenu(userManager, eventManager, conversationManager);
            vipUserMainMenu.handleMainMenu();
        } else if (usr.isAdminUser()) {
            AdminUserMainMenu adminUserMainMenu = new AdminUserMainMenu(userManager, eventManager, conversationManager);
            adminUserMainMenu.handleMainMenu();
        } else { //isAttendee() Switched vip check to be before attendee since VIPUser is a subclass of Attendee
            AttendeeMainMenu attendeeMainMenu = new AttendeeMainMenu(userManager, eventManager, conversationManager);
            attendeeMainMenu.handleMainMenu();
        }
    } else {
        presenter.printError( errorMsg: "Username and Password are incorrect. Please try again.");
        handleLogIn();
    }
}
```



Both methods are from the class LoginReceiver

The handleLogIn method uses polymorphism and calls the method login without knowing the type of the user. In the login Method we direct different to different Menus depending on there UserType which has been predefined (This is done when creating Users in UserGenerator). For example, if the user is an Organizer then the organizer Main Menu is created. And handleMainMenu method is called. The UML shows the creation of different MainMenus from loginReceiver to the specific mainMenus

Why

This is done because many classes of the same subclass need to instantiated depending on the type of the subclass. This is done to create for users easily and this design pattern uses Open Closed principle as we can extend the login method to have more types of users without changes in the pre-existing code. (Making, 2020)

References

Making, S. (2020). *Design Patterns.* Retrieved from Source Making: https://sourcemaking.com/design_patterns

Some Ideas are taken from lecture notes by Lindsey Shorshers design Pattern pdf