Project Report: MyVelib App

Oussama Kharouiche, Nizar El Ghazal

May 30, 2023

1 Introduction

The MyVelib App is a bicycle sharing system designed to provide convenient transportation options for users. The project aims to develop a software application that manages the MyVelib network, allowing users to rent and return bikes, view station statistics, and perform various operations through a command-line user interface (CLUI).

2 System Overview

The system has a bunch of Elements interacting with each other : Users, Stations, Bicycles, Subscription cards and VelibNetworks

2.1 User

the User is implemented as a Java class. The class includes all the relevant information about the user : Full name, Type of card, gps coordinates, the network in which he is registered and his current Bicycle if he has any. Moreover, for each user, a unique Id is generated. The id is unique through all the velib networks, which makes it easy to handle Users from different network at the same time without the need to specify the network in which they are registered. In addition to all the getters and setters, the user also has several functions that model his actions in the system (renting/returning bikes, planning rides)

2.1.1 Renting a bicycle

In the real world, the user goes to a station, and takes the bicycle he wants (Or takes a bikes he finds parked in the street). We modeled this a User function, which takes a bicycle as an input, then assigns it to the user's current bicycle and update all the relevant informations.

2.1.2 Returning a bicycle

In the real world, the user will to a place (Docking station or just the street) and perform the return operation. Everything else is done automatically (compution

the duration, charging the user and applying any bonuses) In this situation, we are facing some problems:

- Computing the duration : We thought of using the current time of the machine to record the timestamp of the renting and the timestamp of the returning, this would be more "real' but it's not optimal when testing the program, Hence we produced two versions of return, one takes the duration as an input and one computes it from machine time.
- As we have seen, the user can return the bicycle to either a station or leave it on the street. We need to have two return functions, one takes a ParkingSlot as an additional input, meaning that the user is returning the bicycle to a specific parking slot.
- GPS coordinates : when parking a bicycle on the street, we need to keep track of its gps coordinates. While we do not have a real gps, we just assign the coordinates of the bicycle as the coordinates of the user when returning the bicycle

2.1.3 Ride Planning

The user can select a ride strategy and gives the start and finish gps coordinates. The function returns a string text with the instructions for the ride.

2.2 Docking Stations

We implemented most of the functions relative to the station and its parking slot in a package named dockingstation. There are two types of stations, regular stations and plus stations. The type of station only impacts the bonus given after returning a bicycle to that specific station. Nevertheless, we chose to implement Docking station as an abstract class, which has two concrete classes (RegularStation and PlusStation)

Every attribute and function that is commun to the two station types is defined in the abstract class. Any station has a number of attributes, the main ones are :

- A unique Id : having multiple classes that refer to stations can lead to a problem : the unicity of IDs throughout the classes, the solution we used for this is A separate singleton class for generation the ID.
- The status of the station, which can be either OnService or Offline, we chose to implement this aspect through an enum class, this way, we can add more status (Partially Closed/ Access restricted to a subscription plan) and we can implement functions that depend on the status.
- A list of all the parking slots in the stations. Each slot is either occupied or free and bicycles can be directly accessed through their slots.

Each of the station objects also include a bunch of statistics, which can be relevant for comparing and sorting stations.

In each concrete class, we only define a function that returns the charge bonus gained when returning to that specific type of station. But this design can facilitate expanding each type of stations separatly.

2.3 Bicycle

We implemented most of the functions relative to the Bicycle and its parking slot in a package named Bicycle. There are two types of Bicycles, electrical bicycle and mechanical Bicycle. The type of Bycicle only impacts the pricing policy and riding plans. Nevertheless, we chose to implement Bicycle as an abstract class, which has two concrete classes (ElectricalBicycle and MechanicalBicycle)

Every attribute and function that is commun to the two Bicycles types is defined in the abstract class. Any bicycle has a number of attributes, the main ones are :

- A unique Id : using the same solution as Docking station.
- Indicator whether the bicycle originated from a station (1) or not (0); we need this variable because it impacts the charge computing.
- The parking slot where the bicycle is parked.

Additionally, the Bicycle class provides various methods for manipulating and interacting with bicycles. For example:

- **removeFromSlot()**: This method is responsible for removing the bicycle from its current parking slot. It updates the relevant variables and ensures proper management of the slot's occupancy.
- **setBikeCountFree() / setBikeCountFill()**: These methods update the variables associated with the presence of a bicycle in a parking slot. They are used to keep track of available slots and manage the capacity of docking stations.

By encapsulating bicycle-related functionality within the Bicycle package and utilizing inheritance, we achieve code organization, reusability, and maintainability. The abstract Bicycle class serves as a foundation for handling various bicycle types, while the concrete subclasses introduce specific behaviors and attributes as needed.

2.4 Subscriptions

The Cards package in our system is responsible for managing the different types of cards used by users in the Velib system. It contains classes that represent different card types and provide functionality for computing charges and applying bonuses.

2.4.1 Card Interface

The Card interface is a key component of the Cards package. It defines the common methods and behaviors that all card types should implement. The interface provides a contract for calculating charges for bike usage and applying bonuses to docking stations.

2.4.2 Card Implementations

The Cards package includes concrete classes that implement the Card interface for each specific card type. These classes represent different cards available to users in the Velib system, such as NoCard, Vlibre, and Vmax.

NoCard The NoCard class represents a card that has no specific privileges or benefits. It implements the Card interface and provides the necessary methods for charge computation and bonus application. However, since it is a basic card, the charge computation and bonus application for a NoCard are typically straightforward and do not involve any discounts or additional features.

Vlibre The Vlibre class represents a card that offers some benefits to users in terms of charge computation and bonus application. It implements the Card interface and provides the necessary methods for calculating charges based on bike type, usage duration, and return location. It also includes logic for applying bonuses to docking stations.

Vmax The Vmax class represents a premium card that provides enhanced privileges and benefits to users. It implements the Card interface and offers advanced features. The charge computation and bonus application for a Vmax card may involve different pricing policies and additional perks compared to other card types.

2.4.3 Card Functionality

The Card implementations within the Cards package provide the necessary functionality to compute charges for bike usage and apply bonuses to docking stations. These functionalities play a crucial role in determining the cost of using bikes and providing incentives for users to return bikes to appropriate stations.

The computeCharge() method in each Card implementation calculates the charge for using a bicycle based on factors such as the bike type, usage duration, and whether the bike is returned to a station. This allows for dynamic pricing based on the specific card type and usage conditions. Seing that the price is dependent on the type of bicycle, hence, we implemented it pricing using a visitor that returns the price based on the type of bicycle.

The applyBonus() method applies the corresponding bonus to a specified docking station. This bonus is applied after returning a bike and it rewards

the user time credit. As this is dependent on the type of stations, we chose to implement it using a visitor pattern.

By organizing card-related functionality within the Cards package and utilizing the Card interface, our system maintains flexibility and extensibility. It allows for the introduction of new card types in the future while ensuring consistent behavior across different card implementations.

2.5 Velib Networks

The core of our myVelib system is built upon two main classes: VelibGlobal and VelibSystem. These classes form the backbone of the system's infrastructure and provide essential functionalities for managing multiple Velib networks and individual Velib systems, respectively.

2.5.1 VelibGlobal Class

The VelibGlobal class represents the global instance that encapsulates all the velib systems. It follows the singleton design pattern, ensuring that there is only one instance of VelibGlobal throughout the system. This class is responsible for maintaining a list of VelibSystem objects, representing the individual Velib networks within the system.

The VelibGlobal class provides several key functionalities, including:

- **Adding Velib Systems**: It allows the addition of VelibSystem objects to the list of Velib systems, enabling the creation and management of multiple Velib networks.
- **Searching for Velib Systems**: It provides methods to search for Velib systems based on their ID or name. These methods facilitate the retrieval of specific Velib systems for further operations.
- **Searching for Users and Docking Stations**: It offers methods to search for users and docking stations within the entire collection of velib networks, while also finding the matching network for the user/station. These methods help in locating specific users or stations for various tasks and operations.
- **Setting Up Velib Networks**: It supports the setup of Velib networks by providing a method to create VelibSystem objects with specified parameters such as the number of stations, number of slots per station, size of the network area, and total number of bikes. It ensures proper initialization of the network and checks for the consistency of bike allocation and slot availability.
- **Adding Users to Velib Networks**: It allows the addition of users with different types of cards (NoCard, Vlibre, Vmax) to specific Velib networks. This functionality ensures the inclusion of users in the appropriate networks for subsequent operations.

- **Setting Docking Stations Online/Offline**: It provides methods to set docking stations within a Velib network to online or offline status. This feature allows the management of station availability and service status for efficient bike distribution and maintenance.

The VelibGlobal class acts as a central control point for the entire myVelib system, enabling the coordination and management of multiple Velib networks and their associated operations.

2.5.2 VelibSystem Class

The VelibSystem class represents an individual Velib network within the myVelib system. It encapsulates the data and functionalities related to a specific network, including docking stations, users, and various operations within that network.

Key features of the VelibSystem class include:

- **Managing Docking Stations**: It allows the addition of docking stations to the Velib network and provides methods to search for stations based on their ID. This functionality ensures the availability and proper management of docking stations within the network.
- **Managing Users**: It supports the addition of users to the Velib network and provides methods to search for users based on their ID. This feature facilitates user-related operations such as bike rental, return, and information retrieval.
- **Setting Station Status**: It enables the setting of docking stations to online or offline status within the Velib network. This functionality helps in managing station availability and maintenance.
- **Displaying Network and Station Information**: It offers methods to retrieve and display information about the Velib network, including the list of stations, their status, and the number of available bikes. This functionality allows users and system administrators to access network information and make informed decisions.

The VelibSystem class encapsulates the specific operations and functionalities of an individual Velib network, ensuring effective management and utilization of resources within that network.

The interaction between the VelibGlobal and VelibSystem classes forms the foundation of our myVelib system, enabling the creation, management, and coordination of multiple Velib networks while providing a wide range of functionalities for users and administrators.

3 Advanced functionalities

The MyVelib App consists of several components:

3.1 Ride Planning Functionality

In our myVelib system, we have implemented a ride planning functionality that assists users in finding optimal bike rides based on their preferences and network conditions. This functionality is implemented in the "planning" package, which includes several classes related to ride planning.

3.1.1 RideType Class

The RideType class serves as a factory class that allows users to choose the strategy for ride planning. It provides a method that create instances of different ride planning strategies based on user preferences. The RideType class abstracts away the specific implementation details of each ride planning strategy and provides a unified interface for selecting and using the desired strategy.

3.1.2 RidesPlanning Abstract Class

The RidesPlanning abstract class is the base class for all ride planning strategies. It defines the common structure and behavior that every ride planning strategy must implement. Each specific ride planning strategy extends this abstract class and provides its own implementation for generating ride plans.

3.1.3 Ride Planning Strategies

Within the "planning" package, we have implemented several ride planning strategies, each extending the RidesPlanning abstract class. These strategies offer different approaches to generating ride plans based on various criteria and user preferences. Some of the implemented ride planning strategies include:

- **MinimalWalkingDistance**: This strategy aims to minimize the walking distance for users by finding bike stations closest to their starting and ending locations. It calculates the distance between stations using GPS coordinates and selects the stations with the minimum total walking distance.
- **PreferStreetBike**: This strategy prioritizes street bikes .
- **PreferPlusStation**: This strategy favors plus station as an ending station for the ride.
- **AvoidPlusStation**: This strategy, in contrast to PreferPlusStation, aims to avoid plus station as an ending station.
- **PreserveUniformityRide**: This strategy aims to preserve the uniformity of bike distribution within the network.
- **ElectricalOnlyRide** and **MechanicalOnlyRide**: These strategies allow users to specify their preference for electrical or mechanical bikes only. They consider the user's preference and availability of the specified bike type when generating ride plans.

By encapsulating each ride planning strategy in its own class and extending the RidesPlanning abstract class, we achieve a modular and extensible design. Users can select the desired strategy using the RideType factory class and obtain a corresponding instance of the ride planning strategy. This allows for easy customization and addition of new ride planning strategies in the future.

The ride planning functionality enhances the user experience by providing efficient and personalized ride plans. It considers various factors to ensure optimal bike availability, minimal walking distance, and adherence to user preferences. By incorporating different ride planning strategies, our myVelib system caters to a wide range of user preferences and enhances the overall usability and convenience of the system.

3.2 Command Line User Interface (CLUI)

To provide users with a convenient and interactive way to interact with the myVelib system, we have developed a Command Line User Interface (CLUI). The "velibclui" package includes several classes responsible for handling user commands and displaying relevant information.

3.2.1 Command Handling Classes

The "velibclui" package includes various command handling classes, each dedicated to handling a specific user command. These classes encapsulate the logic and operations associated with each command, making the code modular and easy to maintain. Some of the implemented command handling classes include:

- **HandleAddUserCommand**: Handles the "addUser" command, allowing users to add new users to the myVelib system with specified names, card types, and network names.
- **HandleDisplayStationCommand**: Handles the "displayStation" command, displaying statistics of a specific station in a given Velib network.
- **HandleDisplayStationInfoCommand**: Handles the "displayStationInfo" command, providing detailled infos about a station from it's id.
- **HandleDisplayStationsCommand**: Handles the "displayStations" command, displaying a list of all stations in a given Velib network.
- **HandleDisplaySystemsCommand**: Handles the "displayNetworks" command, displaying a list of all Velib networks available in the myVelib system.
- **HandleDisplayUserCommand**: Handles the "displayUser" command, displaying statistics information about a specific user in a given Velib network.
- **HandleDisplayUserInfoCommand**: Handles the "displayUserInfo" command, providing detailed information about a particular user.

- **HandleDisplayUsersCommand**: Handles the "displayUsers" command, displaying a list of all users in a given Velib network.
- **HandleOfflineCommand**: Handles the "offline" command, setting a specified station in a Velib network to offline status.
- **HandleOnlineCommand**: Handles the "online" command, setting a specified station in a Velib network to online status.
- **HandlePlanRideCommand**: Handles the "planRide" command, allowing users to plan bike rides based on their preferences and network conditions.
- **HandleRentBikeCommand**: Handles the "rentBike" command, allowing users to rent bikes from specified stations in a Velib network or from the street.
- **HandleReturnBikeCommand**: Handles the "returnBike" command, allowing users to return bikes to specified stations in a Velib network or to the street.
- **HandleSetupCommand**: Handles the "setup" command, setting up a new Velib network with specified parameters.
- **HandleSortStationCommand**: Handles the "sortStation" command, sorting stations in a Velib network based on a specified sorting policy.

These command handling classes ensure that user commands are processed correctly and perform the necessary operations in the myVelib system. They interact with the core system components and utilize their functionalities to provide users with the requested information and perform desired actions.

3.2.2 Exception Handling

To handle exceptional scenarios and ensure the proper execution of user commands, the "velibclui" package also includes custom exception classes. These exceptions are thrown when the input arguments provided for a command are invalid or when a sorting policy is not recognized. The implemented exception classes include:

- **InvalidArgumentSizeException**: This exception is thrown when the number of arguments provided for a command is incorrect or insufficient.
- **InvalidSortingPolicyException**: This exception is thrown when an unrecognized sorting policy is specified for the "sortStation" command.

By incorporating exception handling, the CLUI ensures that users are notified of any incorrect or invalid inputs and helps prevent unexpected behavior or errors in the system.

The CLUI provides an intuitive and user-friendly interface for interacting with the myVelib system. Users can easily execute various commands, view network information, manage users, rent and return bikes, plan rides, and perform other essential operations through simple and straightforward commands. The modular design of the CLUI allows for easy extensibility, making it possible to add new commands and functionalities in the future to further enhance the user experience.

4 Design Patterns

In this project, we used several design patterns to achieve modularity, maintainability, and extensibility. Here is a recap of the design patterns used:

1. Singleton Pattern:
    - The `VelibGlobal` class implements the Singleton pattern. It ensures that only one instance of the `VelibGlobal` class exists throughout the application, allowing global access to the Velib systems.
    - Id generators for other classes also use this pattern to ensure the unicity of the id.
2. Factory Method Pattern:
    - The `HandleSetupCommand` class utilizes the Factory Method pattern to create instances of `VelibSystem` objects based on the provided setup parameters. It encapsulates the creation logic and allows for flexibility in creating different types of Velib systems.
    - The `RidesPlaning` class also utilizes the factory pattern to create customized instances of rides planning
3. Strategy Pattern:
    - The `PlanningStrategy` interface and its implementations represent the Strategy pattern. It allows users to choose different strategies for planning rides, providing flexibility and enabling the system to switch between strategies dynamically.
    - Pricing is also implemented through strategy pattern, allowing each type of card to have its own charge computing method.
4. Template Method Pattern:
    - The `Bicycle` class implements the Template Method pattern. It defines a skeleton algorithm for managing bicycles and delegates certain steps to subclasses (`ElectricalBicycle`, `MechanicalBicycle`) to provide specific implementations.
5. Iterator Pattern:

- The `VelibSystem` class provides an iterator to iterate over its collection of docking stations. It allows clients to traverse the docking stations without exposing the underlying data structure.

6. Command Pattern:

   - The `HandleXCommand` classes (e.g., `HandleSetupCommand`) implement the Command pattern. They encapsulate requests as objects, allowing parameterized invocations of methods and decoupling the invoker (`VelibTerminal`) from the receiver (`VelibSystem`, `Handle` classes).

We employed these design patterns to enhance the overall structure, flexibility, and maintainability of the Velib system. They enable modular development, separation of concerns, and the ability to introduce new features and variations in a structured manner.

5 Advantages and Limitations

5.1 Advantages

Modularity and Extensibility: The system is designed using a modular approach, with well-defined components such as Velib systems, docking stations, users, and cards. This modular design promotes extensibility, allowing for easy addition of new features or modifications to existing ones without affecting the entire system.

Code Reusability: The implementation leverages object-oriented principles, encapsulation, and inheritance to ensure code reusability. Common functionalities are implemented in abstract classes or interfaces, which can be inherited or implemented by specific classes. This reduces code duplication and enhances maintainability.

Scalability: The system supports multiple Velib systems, docking stations, and users. This scalability enables the management of large-scale Velib networks with ease.

Use of Design Patterns: The implementation incorporates several design patterns such as Singleton, Factory, Strategy, Template Method, and Observer. These patterns promote code organization, flexibility, and maintainability.

5.2 Limitations

Simplicity vs. Real-World Complexity: The implementation of the Velib system may simplify certain real-world complexities to maintain code clarity and simplicity. However, this can result in some deviations from the intricacies and dynamics of a real Velib system.

Assumptions and Simplifications: The implementation assumes certain conditions and simplifies certain aspects of the Velib system, such as predefined bike and slot numbers, simplified pricing policies, and ideal user behaviors.

These assumptions and simplifications may not accurately reflect the real-world complexities and variations in a Velib system.

User Interface Considerations: The implementation does not include an extensive user interface or graphical representation of the Velib system. While it provides a terminal-based interface for executing commands, a more user-friendly graphical interface could enhance the overall user experience.

Performance and Optimization: The implementation may not incorporate advanced performance optimization techniques, especially when dealing with large-scale Velib networks or handling concurrent operations. Further optimization considerations may be required for real-time or high-performance Velib systems.

6 Using the Command Line User Interface (CLUI)

The Velib system implementation includes a Command Line User Interface (CLUI) that allows users to interact with the system through text-based commands. This section provides an overview of how to use the CLUI and the available commands.

6.1 Launching the CLUI

To launch the CLUI, navigate to the directory where the Velib system is installed through a project manager (eclipse/intellij..), go to the file src.fr.cf.group13.myVelib and open the file VelibTerminal.java .Then you can run the file by clicking on the run button (or pressing Shift+F10 on intellij)

Once launched, the CLUI will display a prompt symbol ($) indicating that it is ready to receive commands.

6.2 Available Commands

- **setup 'name' or 'name' 'nstations' 'nslots' 's' 'nbikes':**

  The "setup" command is used to create a new Velib network with the given name and optional parameters. If no optional parameters are provided, default values will be used. This command allows customization of the number of stations, number of slots per station, size of the network area, and total number of bikes in the network.

- **displayNetworks:**

  The "displayNetworks" command displays the list of available Velib networks. It provides an overview of the existing networks that users can interact with.

- **displayStations 'velibnetworkName':**

  The "displayStations" command displays the list of stations in the specified Velib network. It provides information about each station, such as its ID, location, and availability of bikes and slots.

- **displayUsers 'velibnetworkName':**

  The "displayUsers" command displays the list of users in the specified Velib network. It provides information about each user, including their ID, name, and current credit balance.

- **addUser 'userName' 'cardType' 'velibnetworkName':**

  The "addUser" command adds a new user with the given name and card type to the specified Velib network. The card type can be "Vlibre", "Vmax", or "None", representing different membership card options for users.

- **offline 'velibnetworkName' 'stationID':**

  The "offline" command sets a specific docking station in the specified Velib network to offline status. This command is useful when a station needs to be temporarily taken offline for maintenance or repairs.

- **online 'velibnetworkName' 'stationID':**

  The "online" command sets a specific docking station in the specified Velib network back to online status. This command is used to reactivate a station after it has been offline.

- **rentBike 'userID' 'biketype' 'stationID' or 'userID' 'biketype' 'GPS_x' 'GPS_y':**

  The "rentBike" command allows a user to rent a bike from a specified station in the Velib network. The user is identified by their ID, and the bike type can be specified as 0 (any type), 1 (mechanical), or 2 (electrical). Users can choose to rent a bike by providing either the station ID or their GPS coordinates.

- **returnBike 'userID' 'duration' 'stationID' or 'userID' 'duration' 'GPS_x' 'GPS_y':**

  The "returnBike" command is used to return a bike rented by the user to a specified station in the Velib network. The user is identified by their ID, and the duration of the bike rental is provided. Users can return a bike by specifying either the station ID or their GPS coordinates.

- **displayStationInfo 'stationID':**

  The "displayStationInfo" command displays detailed information about a specific docking station in the Velib network.

- **displayStation 'velibnetworkName' 'stationID':**

  The "displayStation" command displays statistics about a specific docking station in the given Velib network.

- **displayUser 'velibnetworkName' 'userID':**

  The "displayUser" command displays statistics and information about a specific user in the given Velib network.

- **displayuserinfo 'userID':**

  The "displayuserinfo" command displays detailed information about a specific user in the Velib network.

- **sortStation 'velibnetworkName' 'sortpolicy':**

  The "sortStation" command sorts the stations in the specified Velib network based on the specified policy. The sorting policy can be chosen from available options to arrange the stations according to specific criteria.

- **display 'velibnetworkName':**

  The "display" command provides a comprehensive display of both the stations and users in the specified Velib network.

- **planRide 'userID' 'planningstrategy' 'startingGPS_x' 'startingGPS_y' 'endingGPS_x' 'endingGPS_y':**

  The "planRide" command allows users to plan a ride using the given planning strategy and GPS coordinates. Users can specify their ID, the planning strategy to be used, and the starting and ending GPS coordinates to generate an optimal ride plan.

- **runtest 'testFile = eval/testNinput.txt':**

  The "runtest" command is used to execute a series of commands from the specified test file. This command allows for automated testing and evaluation of the Velib Terminal's functionality using predefined test cases.

6.3 Executing Commands

To execute a command, simply type the command followed by the required arguments (if any), and press Enter. The CLUI will process the command and provide the corresponding output or perform the desired action.

For example, to set up a Velib network, you can use the **setup** command followed by the required parameters:

```
$ setup MyVelib 10 20 1000 100
```

Similarly, you can use other commands to perform various actions within the Velib system, such as displaying network information, adding users, renting bikes, returning bikes, and more.

6.4 Error Handling

The CLUI incorporates error handling mechanisms to provide feedback in case of invalid commands or unexpected issues. If a command is not recognized or contains incorrect arguments, the CLUI will display an error message indicating the issue. It is important to ensure that commands are entered correctly and that the required arguments are provided as expected.

If you encounter any errors or need assistance, you can use the **help** command to get information about the available commands and their usage.

6.5 Exiting the CLUI

To exit the CLUI, you can use the **quit** or **exit** command. Simply type the command and press Enter:

```
$ quit
```

Upon exiting, the CLUI will display a farewell message, and you will return to the regular terminal or command prompt.

Using the Command Line User Interface (CLUI) provides a convenient way to interact with the Velib system, allowing you to set up networks, manage stations and users, and perform various operations related to bike rentals and returns.

6.6 Test scenario

This scenario focuses on the user registration process and bike rental operations. In this scenario, we first register three users with different card types in the MyVelib network. Then, bike rental operations are performed using various parameters such as user ID, bike type, and station ID. The commands for returning bikes and displaying user and station information are also executed. Additionally, the network and station displays are sorted based on different criteria. Lastly, ride planning is demonstrated using different strategies.

7 Task Distribution

Task	**Assigned To**
bycicle	Oussama
cards	Nizar
dockingstation	Nizar
planning	Oussama
system	Nizar
Usercasescenario and user	Oussama
velibclui	Nizar
Tests	Oussama and Oussama
Terminal	Nizar and Oussama

Table 1: Task Distribution