Integration **T**est **P**lan **D**ocument
myTaxiService

Luca Nanni (850113) Giacomo Servadei (854819)

January 20, 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Revision History

Version	1.0

1.2 Purpose and scope

This document represents the *Integration Test Plan* for the application *myTaxiService*.

Purpose The purpose of this document is to test the interaction between the components of the system, in particular the interfaces provided by the single components and how these are used by others.
It will be defined an integration test plan with the sequence of components integrations, the entry criteria, the strategy and tools to be used for the execution of the plan.

Scope Here we summarize the main scope of the application.
The client is the *government of a large city*.
This city already offers a taxi service to its citizens, but the government wants to improve it using a modern and efficient information system.
So we received the request to design and implement this application, called *myTaxiService* which has basically two great objectives:

- Simplify the usage of the taxi service
- Guarantee an efficient management of the taxi queues

1.3 List of definitions and annotations

Refer to the definitions provided in the RASD and DD.

1.4 List of reference documents

- Requirements Analysis and Specification Document (version 2)

- Design Document
- Assignment 4 document
- Assignment 1-2 document
- Slides about the verification tools

Chapter 2

Integration Strategy

2.1 Entry Criteria

- Each component defined in the Design Document has to be already implemented
- Each component must be already tested at module level
- The interactions between the single components and external services (i.e. Google Maps API) must be already tested and implemented
- Interaction between single components and Data Layer must be already implemented and the API (in this case Hibernate) used for accessing the DBMS must be already tested
- The client applications must be already implemented and tested at module and integration level. We will not provide an integration plan for the components inside these applications.

2.2 Elements to be integrated

The elements to be integrated are basically all the components described in the DD (section 2.2). Here we list them:

- **Passenger applications**: Are the (mobile and web) applications that allows the passenger to interact with the system. They are basically a presentation layer and do not provide any internal logic.
- **TaxiDriver application**: Is the mobile application that allows the TaxiDriver to interact with the system.
- **Account Manager**: It manages the accounts, both of Passengers and Taxi Drivers. It also handles the login and logout phases of both users and the registration for the Passenger and the relative account deleting. It exposes the following interfaces:
  - `Passenger`: for the interaction with the Passengers
  - `Taxi Driver`: for the interaction with the Taxi Drivers
- **Ride Manager**: this component is delegated to the management of the life cycle of the incoming requests and reservations from the passengers. From this point of view it can be divided in two main subcomponents:

  - *Request Handler*
  - *Reservation Handler*

  It exposes the following interfaces:

  - `RideHandler`: which can be used by the Passenger mobile application and the web browser (web server)

- **Taxi Manager**: manages the life cycle of the taxi drivers, starting from the availability status and the retrieving of their location. It is in charge of forwarding to the taxi drivers the calls from the passengers.
  It exposes the following interfaces:

  - `StatusHandler`: to communicate with the Taxi driver
  - `TaxiProvider`: to communicate with the RideManager

- **Queue Manager**: it is in charge of the management of the taxi queues in the different zones of the city, applying the decided policy.
  It exposes the following interfaces:

  - `TaxiManagement`: to communicate with the TaxiManager

- **Geographic Engine**: the roles of this component are to implement the algorithms to calculate the zone corresponding to a particular location. It is also in charge of the geographic definition of the taxi zones.
  It exposes the following interfaces:

  - `ZoneLocator`: which provides caulculations methods to the other components
  - `TimeCalculator`: which provides approximative traveling time calculation methods

2.3 Integration testing strategy

During the integration strategy selection we evaluated the following options:

Top Down: We work from the "top" level and simulate the behavior of the lower components through stubs. Our application, anyway, does not have a hierarchical structure in the sense of a central component and subcomponents.

Sandwich: We work at the same time from the "top" level and from the "lower" levels, simulating the middle components through stubs and drivers. We can do the same reasoning as for the top down method.

Bottom Up: We work directly from the "lower" level. This allows us to integrate the single modules in a iterative way and to test the single interactions between them.

Threads: The integration proceeds by taking only *parts* of the components to integrate. It is useful when we are dealing with complex system. Anyway, we believe that our software complexity does not worth the effort to divide the single components in individual testing threads.

Critical Modules: The integration proceed in order of risk for the components in the system. We first integrate the more "risky" (from the implementation/feature/ecc... side) components with the ones used by them and proceed iteratively. As the threads approach, this method is suitable for complex systems and we can do the same reasoning as in the thread method.

Taking all of this options into account, we decided to adopt the **bottom up** integration approach.

2.4 Sequence of Component/Function integration

We have only one sub-system, so we proceed directly to its integration sequence

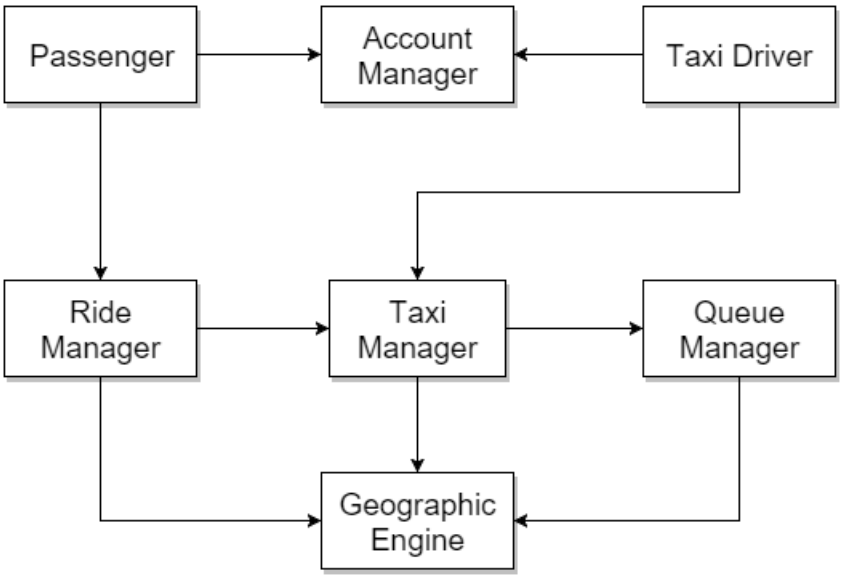

Figure 2.1: Components of the system and their dependencies

In figure 2.1 we show the components of our system as we modeled it in the Design Document. In particular, we show the *dependencies* between them through arrows with this notation: if a component A points to a component B, it means that A *needs* B for its execution. For example: the TaxiManager component needs for its execution components GeographicEngine and QueueManager.

This is the designed integration plan for the components:

ID	**Components interaction**	References
I1	RideManager, TaxiManager, QueueManager $\Longrightarrow$ GeographicEngine	3.1
I2	TaxiManager $\Longrightarrow$ QueueManager	3.2
I3	RideManager $\Longrightarrow$ TaxiManager	3.3
I4	Passenger, TaxiDriver $\Longrightarrow$ AccountManager	3.4
I5	Passenger $\Longrightarrow$ RideManager	3.5
I6	TaxiDriver $\Longrightarrow$ TaxiManager	3.6

Table 2.1: Integration sequence

Chapter 3

Individual steps and test description

3.1 Integration I1

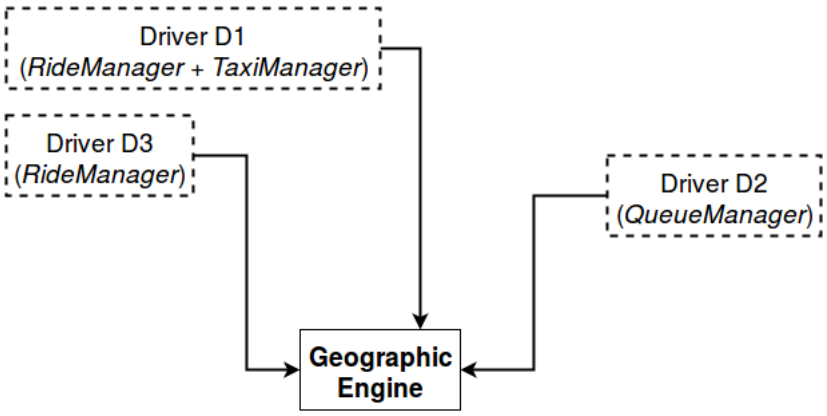

Figure 3.1: Drivers for the integration I1

Test case ID	I1.1
Modules interactions	RideManager, TaxiManager $\Longrightarrow$ GeographicEngine
Drivers	D1 = RideManager + TaxiManager
Input	Ask the zone corresponding to different Locations (both valid and invalid)
Required output	The GeographicEngine must output a taxi zone if the location is valid, otherwise an error

Table 3.1: Test case I1.1

Test case ID	I1.2
Modules interactions	QueueManager $\Longrightarrow$ GeographicEngine
Drivers	D2 = QueueManager
Input	The QueueManager driver asks the GeographicEngine for the list of all the taxi zone in the system
Required output	The GeographicEngine must provide the list of taxi zones

Table 3.2: Test case I1.2

Test case ID	I1.3
Modules interactions	RideManager $\Longrightarrow$ GeographicEngine
Drivers	D3 = RideManager
Input	The RideManager asks the GeographicEngine to calculate the approximate waiting time for the Passenger. It provides couples made of source location (which may be not valid) and destination location (which may be not valid)
Required output	If both the locations are valid, the GeographicEngine must answer with the estimated waiting time. Otherwise with an error.

Table 3.3: Test case I1.3

3.2 Integration I2

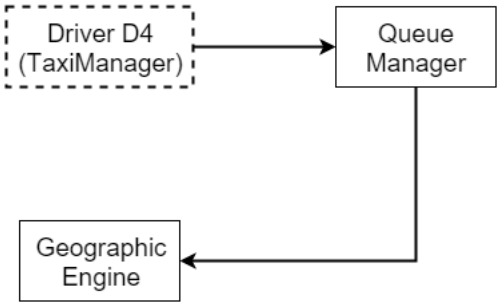

Figure 3.2: Drivers for the integration I2

Test case ID	I2.1
Modules interactions	TaxiManager $\Longrightarrow$ QueueManager
Drivers	D4 = TaxiManager
Input	The TaxiManager driver adds taxi drivers to the data structure and changes them locations. He then remove then or it gets them out of the structures
Required output	The taxi drivers returned, have to respect the priority (the order in which they have been added to the queues) and they have to respect the zone they where added in.

Table 3.4: Test case I2.1

3.3 Integration I3

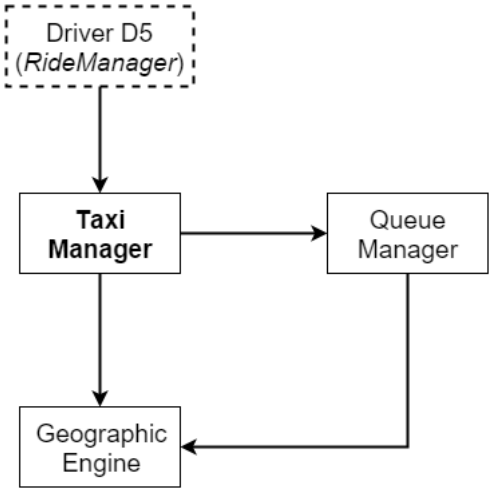

Figure 3.3: Drivers for the integration I3

Test case ID	I3.1
Modules interactions	RideManager $\Longrightarrow$ TaxiManager
Drivers	D5 = RideManager
Input	The RideManager asks the TaxiManager for a taxi given the location of the passenger it is serving. Provide different locations in order to check when the TaxiManager gives error.
Required output	The TaxiManager must retrieve a taxi driver for the call or give an error if there are no taxi driver available.

Table 3.5: Test case I3.1

3.4 Integration I4

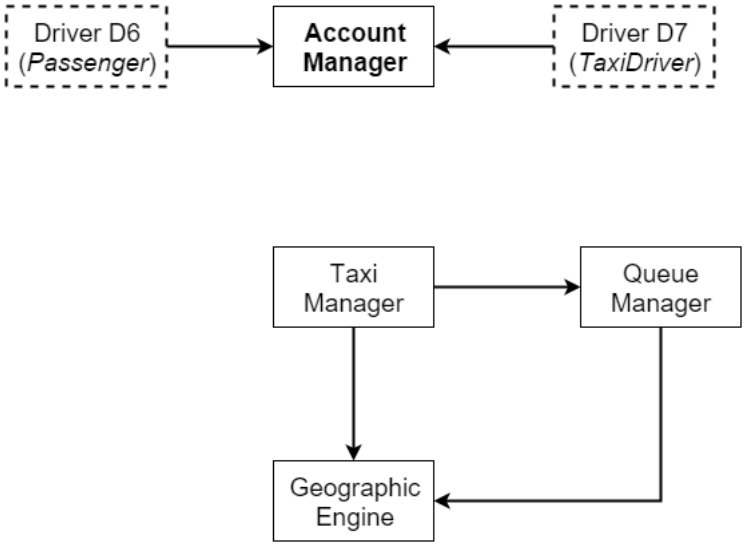

Figure 3.4: Drivers for the integration I4

Test case ID	I4.1
Modules interactions	Passenger $\Longrightarrow$ AccountManager
Drivers	D6 = Passenger
Input	The Passenger driver makes multiple fake accounts using special characters in the username and password. He tries then to login and logout with those accounts and also delete them.
Required output	The AccountManager should provide valid sessions in the login phase and create and delete accounts without errors. It also has to destroy session during the logout phase.

Table 3.6: Test case I4.1

Test case ID	I4.2
Modules interactions	TaxiDriver $\Longrightarrow$ AccountManager
Drivers	D7 = TaxiDriver
Input	The TaxiDriver driver tries to login and logout with real and fake accounts.
Required output	The AccountManager should provide valid sessions in the login phase and destroy sessions during the logout phase.

Table 3.7: Test case I4.2

3.5 Integration I5

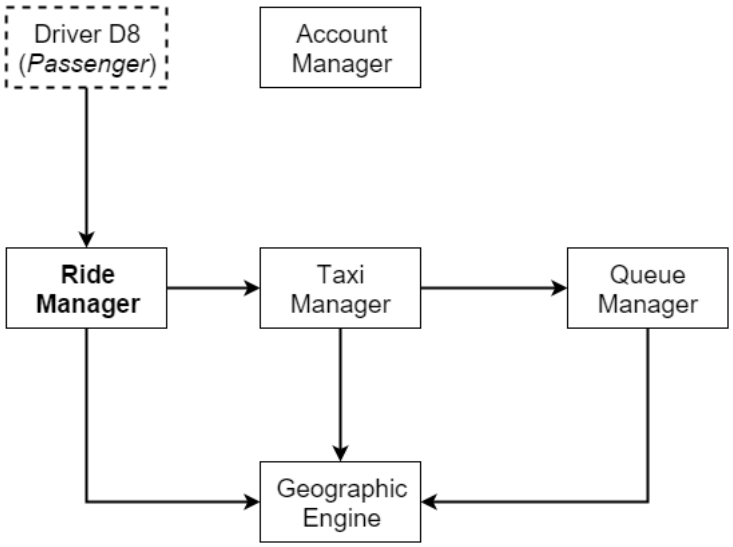

Figure 3.5: Drivers for the integration I5

Test case ID	I5.1
Modules interactions	Passenger $\Longrightarrow$ RideManager
Drivers	D8 = Passenger
Input	The Passenger makes different requests with different parameters: locations and number of passengers must be both valid and not
Required output	The RideManager must respond with an acknowledgement in case a taxi driver has been found, or with an error, in case the input data are not valid or there are not taxi drivers available

Table 3.8: Test case I5.1

Test case ID	I5.2
Modules interactions	Passenger $\Longrightarrow$ RideManager
Drivers	D8 = Passenger
Input	The Passenger makes different reservations with different parameters: locations, number of passengers and time must be tried with both valid and not valid values
Required output	The RideManager must respond with an acknowledgement in case the reservation has been stored, or with an error, in case the input data are not valid

Table 3.9: Test case I5.2

3.6 Integration I6

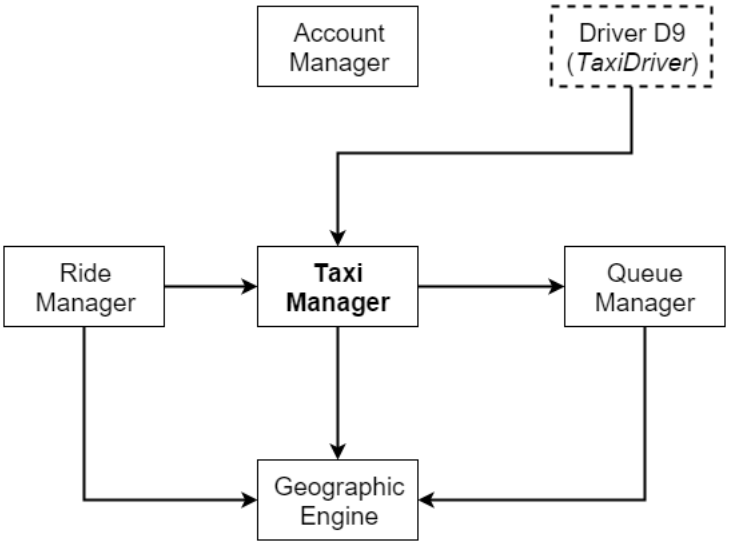

Figure 3.6: Drivers for the integration I6

Test case ID	I6.1
Modules interactions	TaxiDriver $\Longrightarrow$ TaxiManager
Drivers	D9 = TaxiDriver
Input	The TaxiDriver driver changes both availability status and location of taxi drivers.
Required output	The TaxiManager has to put taxi drivers in the right zone based on the location and the availability status.

Table 3.10: Test case I6.1

3.7 Final integration - Complete test of the system

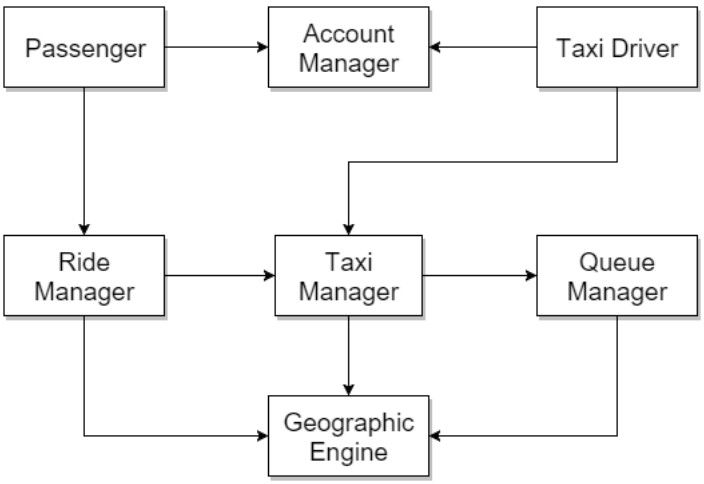

Figure 3.7: Final integration test - The complete system

The final integration step is simply the test of the whole functioning software. The real mobile/web applications will be used and will be monitored their interaction with the system server side, through the testing of all the requirements reported in the design document.

3.8 Dependencies and final test plan

Here we show the dependency graph between the integrations. An arrow from integration I_i to integration I_j means that integration I_i needs first the execution of integration I_j in order to be carried on.

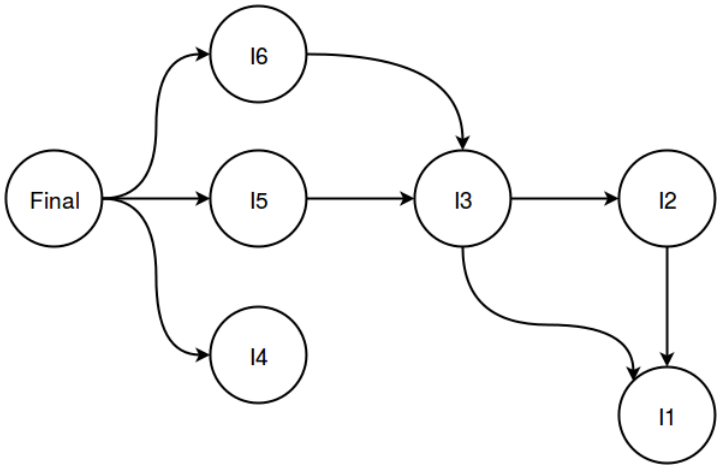

Figure 3.8: Dependency graph of the integrations

A possible integration plan is showed in figure 3.9

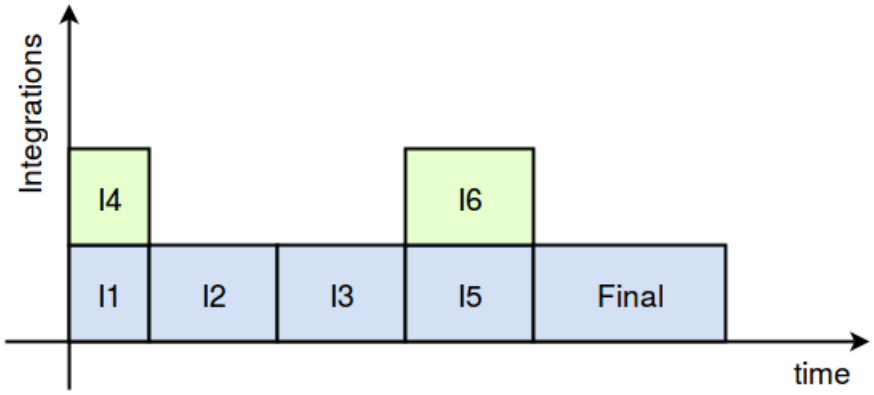

Figure 3.9: Schedule of a possible integration plan

As we notice, we can introduce some parallelism between the different integrations.
The graph is to be read as the sequence of integration steps during the testing phase. If an integration step is over an other or more, it means that can be executed in parallel with it.

In particular in this graph we state that:

1. Integrations I1 and I4 proceed in parallel
2. Follows integration I2
3. Follows integration I3
4. Follow integrations I5 and I6 in parallel
5. Follows the final integration

Chapter 4

Tools and test equipment required

4.1 JUnit

Junit is the framework of choice for the evaluation and statistical analysis of the test results. The reasons for this choice are:

- It is the most famous testing framework, and there are good chances that it is well known in the development team devoted to the project.
- It uses the Java programming language, and since we are planning to use the JEE environment, its integration will be simpler
- There is a graphical plug-in for the Eclipse IDE, which will be the programming environment of choice during the development.

4.1.1 DbUnit

In addition we can use also use *DbUnit*, which is a JUnit extension that we can use to initialize the database with a "state" that is well known. The real advantage of this approach is that we divide the test data creation from the test code.

4.2 Versioning system

We believe that the usage of a versioning system (like *Git*) is essential for the testing phase success, as for the development phase. Developers will provide corrections and discuss enhancements, with the side effect of creation of a lot of intermediate different versions of the same unit or set of units.

Chapter 5

Program Stubs and test data required

5.1 Drivers and data required

Here we summarize the drivers required by the integration test plan described in 2.4 and in chapter 3.
Since we are using the bottom up approach and the unit testing is supposed to have been already done, we do not need any stub for the components.

Driver name	D1
Mocked component(s)	RideManager + TaxiManager
Tested component	GeographicEngine
Tested interface	`ZoneLocator`: • `getTaxiZone(Location location)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...)

Table 5.1: Driver D1

Driver name	D2
Mocked component(s)	QueueManager
Tested component	GeographicEngine
Tested interface	`ZoneLocator`: • `List<Zone> getTaxiZones()`
Required Data	NULL

Table 5.2: Driver D2

Driver name	D3
Mocked component(s)	RideManager
Tested component	GeographicEngine
Tested interface	`TimeCalculator`: • `Time getTimeEstimation(Location source, Location destination)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...)

Table 5.3: Driver D3

Driver name	D4
Mocked component(s)	TaxiManager
Tested component	QueueManager
Tested interface	`TaxiManagement`: • `void addTaxi(TaxiDriver taxi, Zone zone)` • `TaxiDriver getTaxi(Zone zone)` • `void removeTaxi(TaxiDriver taxi)` • `void changeZone(TaxiDriver taxi, Zone zone)`
Required Data	Set of zones and taxi driver objects

Table 5.4: Driver D4

Driver name	D5
Mocked component(s)	RideManager
Tested component	TaxiManager
Tested interface	`TaxiProvider`: • `Response provideTaxi(Location location)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...)

Table 5.5: Driver D5

Driver name	D6
Mocked component(s)	Passenger
Tested component	AccountManager
Tested interface	`Passenger`: • `Passenger login(String username, String password)` • `void logout(Passenger passenger)` • `boolean register(String username, String password)` • `boolean deleteAccount(String username, String password)`
Required Data	Set of strings for username and passwords. Also username and password empty and incorrect

Table 5.6: Driver D6

Driver name	D7
Mocked component(s)	TaxiDriver
Tested component	AccountManager
Tested interface	`TaxiDriver`: • `TaxiDriver login(String username, String password)` • `void logout(TaxiDriver taxidriver)`
Required Data	Set of strings for username and passwords. Also username and password empty and incorrect

Table 5.7: Driver D7

Driver name	D8
Mocked component(s)	Passenger
Tested component	RideManager
Tested interface	`RideHandler`: • `Response makeRequest(Location location, int numberOfPassengers, Passenger passenger)` • `Response makeReservation(Location source, Location destination, int numberOfPassengers, Date date, Time time, Passenger passenger)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...). Set of numbers, also negative. Set of dates, also in the past

Table 5.8: Driver D8

Driver name	D9
Mocked component(s)	TaxiDriver
Tested component	TaxiManager
Tested interface	`StatusHandler`: • `boolean setAvailabilityStatus(TaxiDriver taxiDriver, AvailabilityStatus status)` • `void setLocation(TaxiDriver taxiDriver, Location location)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...). Set of statuses, both valid and invalid

Table 5.9: Driver D8