Politecnico di Milano

Software Engineering 2 Project
a.y. 2015-16

MyTaxiService
Integration Test Plan Document
Version 1.1

Casati Fabrizio, 853195
Castelli Valerio, 853992

Referent professor: Di Nitto Elisabetta

February 29, 2016

Contents

Chapter 1

Introduction

1.1 Revision History

Version	Date	Author(s)	Summary
1.1	29/02/16	Valerio Castelli & Fabrizio Casati	Minor fixes.
1.0	21/01/16	Valerio Castelli & Fabrizio Casati	Initial release

1.2 Purpose and Scope

This document represents the Integration Testing Plan Document for myTaxiService.

Integration testing is a key activity to guarantee that all the different subsystems composing myTaxiService interoperate consistently with the requirements they are supposed to fulfill and without exhibiting unexpected behaviors. The purpose of this document is to outline, in a clear and comprehensive way, the main aspects concerning the organization of the integration testing activity for all the components that make up the system. In the following sections we're going to provide:

- A list of the subsystems and their subcomponents involved in the integration activity that will have to be tested
- The criteria that must be met by the project status before integration testing of the outlined elements may begin
- A description of the integration testing approach and the rationale behind it
- The sequence in which components and subsystems will be integrated

- A description of the planned testing activities for each integration step, including their input data and the expected output
- Some performance measures that should be performed on the components to check they are fulfilling the requirements
- A list of all the tools that will have to be employed during the testing activities, together with a description of the operational environment in which the tests will be executed

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Subcomponent: each of the low level components realizing the functionalities of a subsystem.
- Subsystem: a high-level functional unit of the system.

1.3.2 Acronyms

- SDD: Software Design Description.
- DD: Design Document. Used as a synonym of SDD.
- DBMS: Database Management System.
- API: Application Programming Interface.
- RASD: Requirement Analysis and Specification Document.
- SRS: Software Requirements Specifications. Synonym of RASD.
- ETA: Estimated Time of Arrival.
- UI: User Interface.
- GPS: Global Positioning System.
- SDK: Software Development Kit.

1.3.3 Abbreviations

- Req. as for Requirement.
- WebApp as for Web Application.

1.4 Reference Documents

- The project description document: Assignments 1 and 2 (RASD and DD).pdf
- Assignment document: Assignment 4 - integration test plan.pdf
- myTaxiService Requirement Analysis and Specification Document: RASD.pdf
- myTaxiService Design Document: DD.pdf
- The Integration Test Plan Example document: Integration Test Plan Example.pdf

Chapter 2

Integration Strategy

2.1 Entry Criteria

In order for the integration testing to be possible and to produce meaningful results, there are a number of conditions on the progress of the project that have to be met.

First of all, the **Requirements Analysis and Specification Document** and the **Design Document** must have been fully written. This is a required step in order to have a complete picture of the interactions between the different components of the system and of the functionalities they offer.

Secondly, the integration process should start only when the estimated percentage of completion of every component with respect to its functionalities is:

- **100%** for the **Data Access Utilities** component
- At least **90%** for the **Taxi Management System** subsystem
- At least **70%** for the **System Administration** and **Account Management** subsystems
- At least **50%** for the **client applications**

It should be noted that these percentages refer to the status of the project at the beginning of the integration testing phase and they do not represent the minimum completion percentage necessary to consider a component for integration, which must be at least **90%**. The choice of having different completion percentages for the different components has been made to reflect their order of integration and to take into account the required time to fully perform integration testing.

2.2 Elements to be Integrated

In the following paragraph we're going to provide a list of all the components that need to be integrated together.

As specified in myTaxiService's Design Document, the system is built upon the interactions of many high-level components, each one implementing a specific set of functionalities. For the sake of modularity, each subsystem is further obtained by the combination of several lower-level components. Because of this software architecture, the integration phase will involve the integration of components at two different levels of abstraction.

At the lowest level, we'll integrate together those components that depend strongly on one another to offer the higher level functionalities of myTaxiService. In our specific case, this involves the integration of the **Reservation Management**, **Request Management**, **Location Management** and **Taxi Management** subcomponents in order to obtain the **Taxi Management System** subsystem.

For what concerns the building of the **System Administration** and **Account Management** subsystems, the integration activity is actually quite limited; in fact, they simply represent a collection of functionalities belonging to the same area which however are not dependent on one another. As a result of this, their subcomponents don't really interact with each other, and the integration phase will be limited to the task of ensuring that the set of functionalities of each subcomponent is properly exposed by the subsystem. The components involved in this phase are:

- The **API Permissions Management**, **Zone Division Management**, **Taxi Driver Management**, **Service Statistics** and **Plugin Management** subcomponents in order to obtain the **System Administration** subsystem.
- The **Passenger Registration**, **Login**, **Password Retrieval** and **Settings Management** subcomponents in order to obtain the **Account Management** subsystem.

Some of these subcomponents also directly rely on higher level, atomic components: that is the case, for instance, of the dependency on the **Data Access Utilities** component. These dependencies will be taken care of in the integration process.

Finally, we will proceed with the integration of the higher level subsystems. In particular, the integration activity will involve:

- A number of commercial, already existing components used to achieve specific functionalities: these are the **DBMS**, **Mapping Service**, **Notification System** and **Remote Services Interface** components.
- Those components and subsystems specifically developed for myTaxiService, that are:

- On the server side: the **Taxi Management System**, **System Administration**, **Account Management** subsystems, together with the **Data Access Utilities** component.
- On the client side: the **Administration Web Application**, **Passenger Web Application**, **Passenger Mobile Application** and **Taxi Driver Mobile Application** components.

2.3 Integration Testing Strategy

The approach we're going to use to perform integration testing is based on a mixture of the bottom-up and critical-module-first integration strategies.

Using the bottom-up approach, we will start integrating together those components that do not depend on other components to function, or that only depend on already developed components. This strategy brings a number of important advantages. First, it allows us to perform integration tests on "real" components that are almost fully developed and thus obtain more precise indications about how the system may react and fail in real world usage with respect to a top-down approach. Secondly, working bottom-up enables us to more closely follow the development process, which in our case is also proceeding using the bottom-up approach; by doing this we can start performing integration testing earlier in the development process as soon as the required components have been developed in order to maximize parallelism and efficiency.

Since subsystems are fairly independent from one another, the order in which they're integrated together to obtain the full system follows the critical-module-first approach. This strategy allows us to concentrate our testing efforts on the riskiest components first, that is those that represent the core functionalities of the whole system and whose malfunctioning could pose a very serious threat to the correct implementation of the entire myTaxiService infrastructure. By proceeding this way, we are able to discover bugs earlier in the integration progress and take the necessary measures to correct them on time.

It should be noted that **Notification System**, **Remote Services Interface**, **Mapping Service** and **DBMS** are commercial components that have already been developed and can thus be immediately used in a bottom-up approach without any explicit dependency.

2.4 Sequence of Component/Function Integration

In this section we're going to describe the order of integration (and integration testing) of the various components and subsystems of myTaxiService. As a notation, an arrow going from component C1 to component C2 means

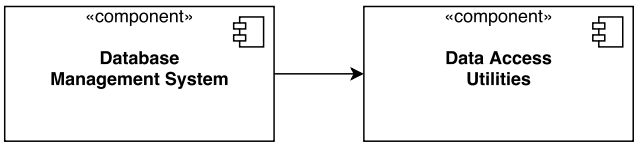
that C1 is necessary for C2 to function and so it must have already been implemented.

2.4.1 Software Integration Sequence

Following the already mentioned bottom-up approach, we now describe how the various subcomponents are integrated together to create higher level subsystems.

Data Access Utilities

The first two elements to be integrated are the **Data Access Utilities** and the **Database Management System** components. We start from here because every other component relies on **Data Access Utilities** to perform queries on the underlying data structure.



Taxi Management System

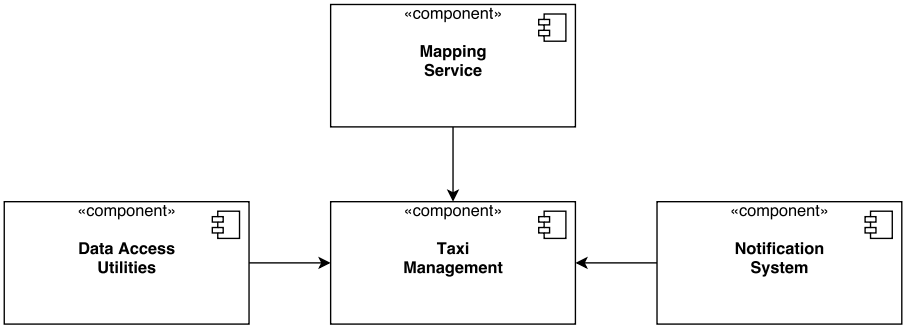
The second step in the integration process is to appropriately connect the subcomponents implementing the **Taxi Management System**. This choice comes from the critical-module-first approach, because taxi management is the single most important functionality of myTaxiService.

In the following diagrams, we are going to show exactly which components must be integrated together in order to implement this functionality using a bottom-up approach.

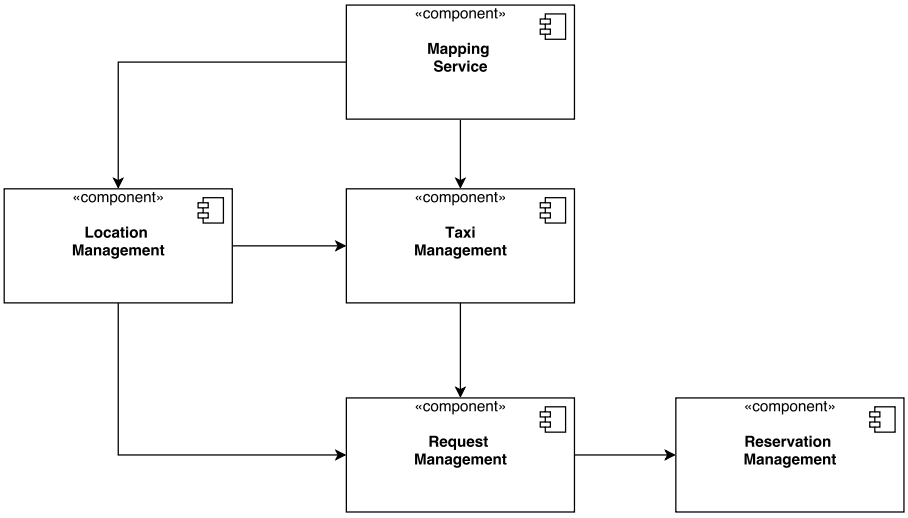
First, we proceed by integrating together the **Request Management** subcomponent with the **Data Access Utilities** and the **Notification System** components.

The same activity is performed between the **Reservation Management** subcomponent and the **Data Access Utilities** and the **Notification System** components.

Finally, we integrate together the **Taxi Management** component with the **Data Access Utilities**, the **Notification System** and the **Mapping Service** components.



At this point, the four sub-components of **Taxi Management System** are ready to be integrated together.



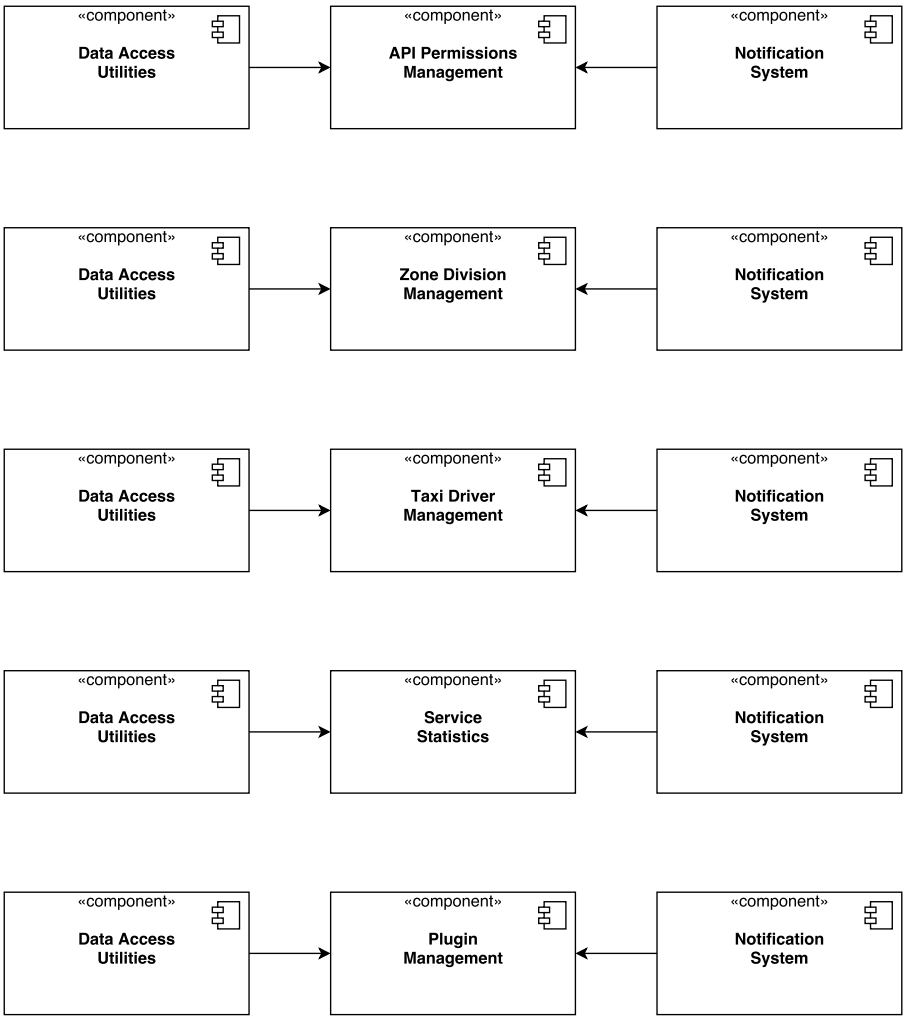
System Administration

The third step in the integration process is to appropriately connect the sub-components implementing the **System Administration** subsystem. This

choice comes from the critical-module-first approach, because system administration is the second most important functionality of myTaxiService. Once it has been integrated and tested, we can use this functionality to more easily populate the database for the following integration tests.

It should be noted that the subcomponents of **System Administration** are loosely coupled together as they cover different aspects of the system administration activity. Because of this, they can be integrated with the other components of the system independently from one another.

In the following diagrams, we are going to show exactly how these subcomponents interact with the other components using a bottom-up approach. The **System Administration** subsystem, which here is not explicitly represented, is simply a wrapper for the methods of these subcomponents that have to be exposed to the other parts of the system and performs additional preprocessing to ensure these methods are properly called.

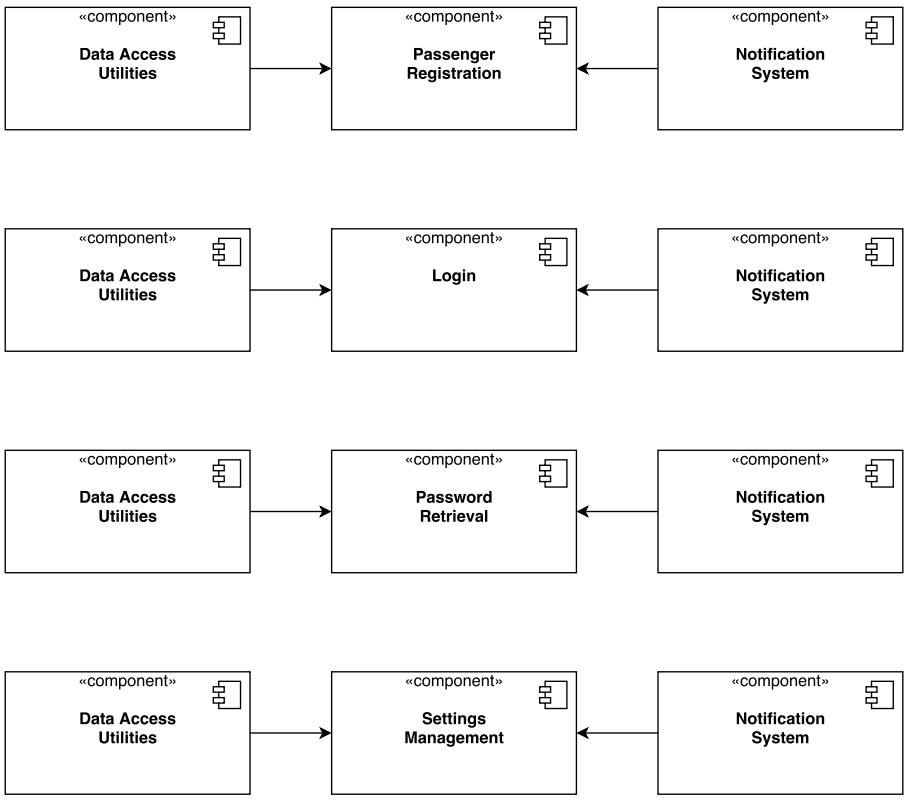


Account Management

The fourth step in the integration process is to appropriately connect the subcomponents implementing the **Account Management** subsystem. This choice is dictated by the bottom-up approach that we are following, because account management is the last functionality that can be implemented without depending on anything but already implemented components.

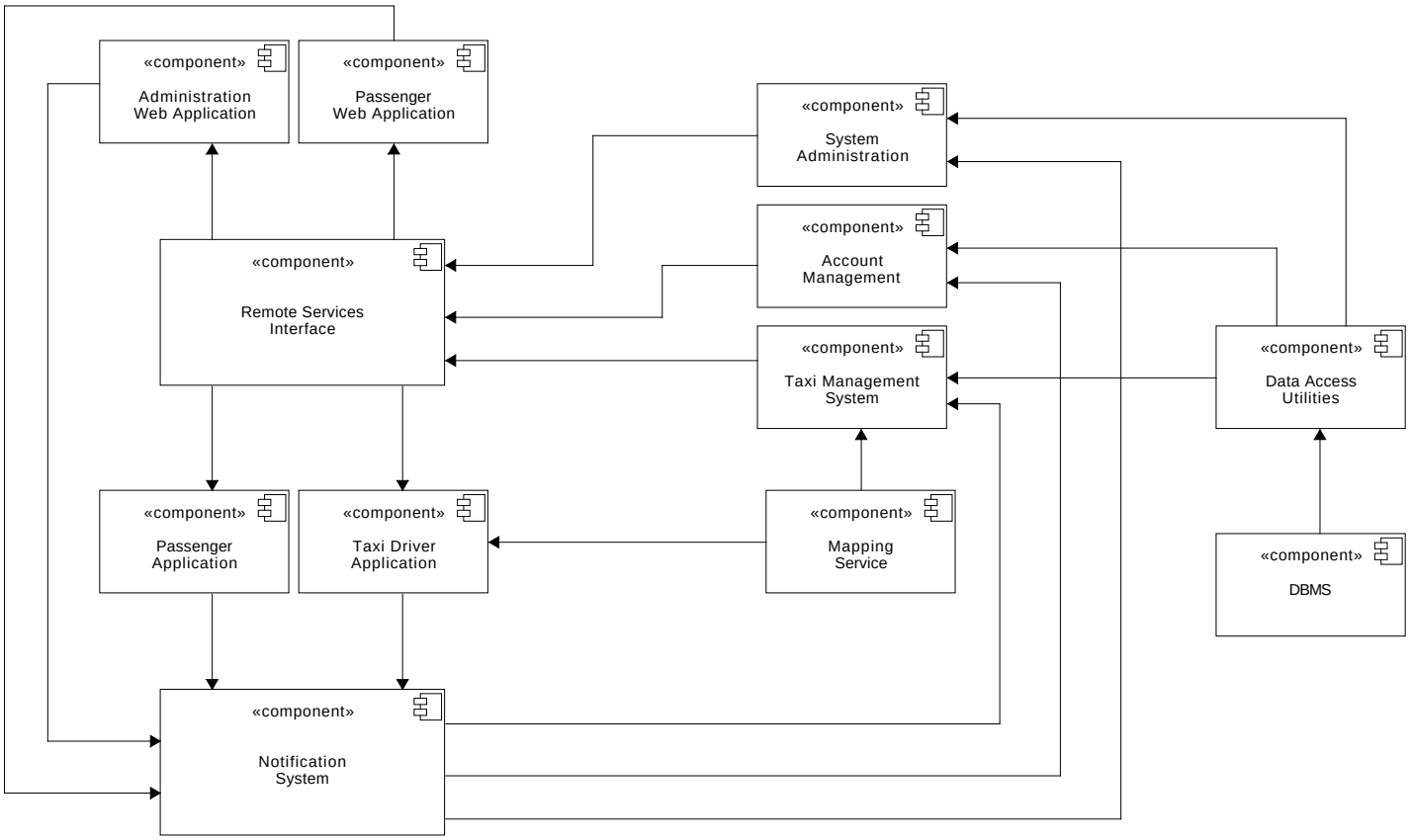
It should be noted that the subcomponents of **Account Management** are loosely coupled together as they cover different operations that can be performed on accounts. Because of this, they can be integrated with the other components of the system independently from one another.

In the following diagrams, we are going to show exactly how these subcomponents interact with the other components using a bottom-up approach. The **Account Management** subsystem, which here is not explicitly represented, is simply a wrapper for the methods of these subcomponents that have to be exposed to the other parts of the system and performs additional preprocessing to ensure these methods are properly called.



2.4.2 Subsystem Integration Sequence

In the following diagram we provide a general overview of how the various high-level subsystems are integrated together to create the full myTaxiService infrastructure.



Chapter 3

Individual Steps and Test Description

In this chapter we'll provide a detailed description of the tests to be performed on each pair of components that have to be integrated.

Each pair of components is described in a specific subsection, identified by the $< caller, called >$ notation, containing the list of methods that the $< caller >$ component invokes on the $< called >$ component.

For each method we're going to provide a brief description of the input values and the corresponding expected effects on the system.

3.1 Taxi Management System

3.1.1 Request Management, Data Access Utilities

insertRequest(request)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A request with an id already existent in the database	An InvalidArgumentValueException is raised.
Formally valid arguments	An entry containing the request data is inserted into the database.
deleteRequest(request)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A request with an inexistent id	An InvalidArgumentValueException is raised.

Formally valid arguments	The entry containing the request data is deleted from the database.
getRequestList()	
Input	*Effect*
Nothing	The list of all pending requests.

3.1.2 Reservation Management, Data Access Utilities

insertReservation(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A reservation with an id already existent in the database	An InvalidArgumentValueException is raised.
Formally valid arguments	An entry containing the reservation data is inserted into the database.
deleteReservation(reservation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A reservation with an inexistent id	An InvalidArgumentValueException is raised.
Formally valid arguments	The entry containing the reservation data is deleted from the database.
updateReservationList(reservationList)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing some null values	A NullArgumentException is raised.
An array of non-null, but inexistent reservations	An InvalidArgumentValueException is raised.
An array of valid and existing reservations	The corresponding entries in the database are updated to set the reservation as completed.
getReservationList()	
Input	*Effect*
Nothing	The list of all pending reservations.

3.1.3 Taxi Management, Data Access Utilities Management

updateQueues(taxiQueue)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing some null values	A NullArgumentException is raised.
An array of non-null queues, but containing null values	An InvalidArgumentValueException is raised.
A non-empty array of valid queues	The content of the queues is updated in the database.
updateTaxiLocation(taxiId, location)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A non-existing taxiId	An InvalidArgumentValueException is raised.
A set of valid parameters	The new location of the taxi is written in the database.
updateTaxiStatus(taxiId, TaxiStatusAvailable)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing taxiId	An InvalidArgumentValueException is raised.
A set of valid parameters	The taxi status is set to available in the database.
getStoredTaxiLocation(taxiId)	
Input	*Effect*
A non-existing taxiId	An InvalidArgumentValueException is raised.
A valid taxiId	Returns the stored taxi location.
getTaxiStatus(taxiId)	
Input	*Effect*
A non-existing taxiId	An InvalidArgumentValueException is raised.
A valid taxiId	Returns the stored taxi status.
endRide(taxiId, currentLocation)	
Input	*Effect*
A null location	A NullArgumentException is raised.

A non-existing taxiId	An InvalidArgumentValueException is raised.
A valid taxiId and currentLocation, the taxi is on a ride and currentLocation is inside city	The ride is considered closed and is finalized in the database.

3.1.4 Taxi Management, Location Management

isLocationInsideCity(location)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A location whose coordinates are invalid	An InvalidLocationException is raised.
A location outside the city	Returns false.
A location inside the city	Returns true.
getZone(location)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A location whose coordinates are invalid	An InvalidLocationException is raised.
A location outside the city	An InvalidLocationException is raised.
A location inside the city	The id of the zone to which the location belongs.

3.1.5 Reservation Management, Location Management

getZone(location)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A location whose coordinates are invalid	An InvalidLocationException is raised.
A location outside the city	An InvalidLocationException is raised.
A location inside the city	The id of the zone to which the location belongs.
checkTaxiDriverLocation(currentLocation, meetingPoint)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A currentLocation whose coordinates are invalid	An InvalidLocationException is raised.

A meetingPoint whose coordinates are invalid	An InvalidLocationException is raised.
A currentLocation outside the city	Returns false.
A meetingPoint outside the city	An InvalidLocationException is raised.
Both currentLocation and meetingPoint inside the city, but not within 50m from one another	Returns false.
Both currentLocation and meetingPoint inside the city, within 50m from one another	Returns true.

3.1.6 Reservation Management, Request Management

requestTaxi(passengerId, passengerLocation, destination)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A passengerId not correctly formatted	An InvalidArgumentFormatException is raised.
A passengerLocation whose coordinates are invalid	An InvalidLocationException is raised.
A destination whose coordinates are invalid	An InvalidLocationException is raised.
A passengerLocation outside the city	An InvalidLocationException is raised.
A valid set of parameters	A new request is created and handled; refer to the RASD for the specific outcomes of this operation.

3.1.7 Request Management, Taxi Management

existsAvailableTaxiDriver(request, zone)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An inexistent zone	An InvalidArgumentValueException is raised.
A zone with invalid fields	An InvalidArgumentValueException is raised.

A valid set of parameters	Returns true if a taxi driver is available to serve the request, false otherwise.
getAvailableTaxiDriver(request)	
Input	*Effect*
A null request	A NullArgumentException is raised.
An unassigned request	An InvalidArgumentValueException is raised.
An assigned request	A reference to the taxi driver that has been selected to serve the request.
getETA(taxiDriver, passengerLocation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A passengerLocation whose coordinates are invalid	An InvalidLocationException is raised.
An inexistent taxi driver	An InvalidArgumentValueException is raised.
A valid taxiDriver and passengerLocation	Returns the estimated time of the arrival of the taxi to the location.
taxiDriverDroppedRequest(taxiId)	
Input	*Effect*
An invalid taxiId	An InvalidArgumentValueException is raised.
A valid taxiId	If the taxi driver is allowed to drop the request (check RASD conditions) it will return true, notify the passenger that his request has been dropped and update the database accordingly; otherwise it will return false.
sendCurrentLocation(taxiId, location)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A location whose coordinates are invalid	An InvalidLocationException is raised.
An invalid taxiId	An InvalidArgumentValueException is raised.
taxiId is valid, location is inside city	Location is set as the new position of the taxi, the taxi is moved into the new zone, its status is set to available and the modification is written in the database.

taxiId is valid, location is outside city	Location is set as the new position of the taxi, the taxi is moved into the out-of-city queue, its status is set to outide-city and the modification is written in the database.

3.2 System Administration

3.2.1 Zone Management, Data Access Utilities

insertZones(zones)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing some null values	A NullArgumentException is raised.
An array of non-null, but invalid zones	An InvalidArgumentValueException is raised.
An array of valid, but already existing zones	An InvalidArgumentValueException is raised.
A non-empty array of valid and not existing zones	The zones are inserted in the database.
insertZone(zone)	
Input	*Effect*
zone is null	A NullArgumentException is raised.
An invalid zone	An InvalidArgumentValueException is raised.
A valid, but already existent zone	An InvalidArgumentValueException is raised.
A valid and not existing zone	The zone is inserted in the database.
updateZones(zones)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing some null values	A NullArgumentException is raised.
An array of non-null, but invalid or inexistent zones	An InvalidArgumentValueException is raised.

A non-empty array of valid, existing zones	The data associated with the specified zones is updated in the database.
deleteZone(zoneId)	
Input	*Effect*
A non-existing zoneId	An InvalidArgumentValueException is raised.
A valid zoneId	The zone data is removed from the database.
getZoneList()	
Input	*Effect*
Nothing	Returns the zones that are stored in the database.

3.2.2 API Permission Management, Data Access Utilities

checkPassword(adminId, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty password	An InvalidArgumentValueException is raised.
An invalid adminId	An InvalidSecurityLevelException is raised.
A valid adminId and password, but password does not correspond to an authorized one	Returns false.
A valid adminId and password, and password does correspond to an authorized one	Returns true.
verifyPermission(appId, operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing appId	An InvalidSecurityLevelException is raised.
A non-existing operation	An InvalidSecurityLevelException is raised.
A valid appId and operation, but the app hasn't enough privileges to execute the desired operation	Returns false.

A valid appId and operation, and the app has enough privileges to execute the desired operation	Returns true.
grantPermission(appId, operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operation	An InvalidSecurityLevelException is raised.
A valid appId and operation	Insert a new $< appId, operation >$ pair in the permission table in the database.
revokePermission(appId, operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operation	An InvalidSecurityLevelException is raised.
A valid appId and operation, but the $< appId, operation >$ is not present in the database	An InvalidArgumentValueException is raised.
A valid appId and operation, and the $< appId, operation >$ is present in the database	Remove the $< appId, operation >$ pair from the permission table of the database.

3.2.3 Taxi Driver Management

Data Access Utilities

insertTaxiDrivers(taxiDriverList)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing some null values	A NullArgumentException is raised.
An array of valid, but already existing taxi drivers	An InvalidArgumentValueException is raised.

A non-empty array of valid and not existing taxi drivers	The taxi drivers are inserted in the database.
insertTaxiDriver(taxiDriver)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An invalid taxi driver	An InvalidArgumentValueException is raised.
A valid, but already existent taxi driver	An InvalidArgumentValueException is raised.
A valid and not existing taxi driver	The taxi driver is inserted in the database.
updateTaxiDriver(taxiDriver)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An inexistent taxi driver	An InvalidArgumentValueException is raised.
A taxi driver with some null or empty fields	An InvalidArgumentValueException is raised.
A valid taxi driver	The data associated with the specified taxi driver is updated in the database.
deleteTaxiDriver(taxiDriverId)	
Input	*Effect*
A non-existing taxiDriverId	An InvalidArgumentValueException is raised.
A valid taxiDriverId	The taxi driver data is removed from the database.

3.3 Account Management

3.3.1 Passenger Registration, Data Access Utilities

insertPassenger(passenger)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-null passenger	The passenger data is inserted in the database.
deletePassenger(passengerId)	
Input	*Effect*
A non-existing passengerId	An InvalidArgumentValueException is raised.

A valid passengerId	The passenger data is removed from the database.
updatePassengerData(passenger)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-null valid passenger	The passenger data is updated in the database.
confirmPassenger(passengerId)	
Input	*Effect*
A non-existing passengerId	An InvalidArgumentValueException is raised.
A passengerId that has already been confirmed	An InvalidArgumentValueException is raised.
A passengerId that hasn't already been confirmed	The passenger status in the database is set to "confirmed".

3.3.2 Login, Data Access Utilities

checkCredentials(user, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An inexistent user	An InvalidArgumentValueException is raised.
An empty password	An InvalidArgumentValueException is raised.
A valid user and password combination, which however is not the correct one	Returns an InvalidCredentialError.
A correct and valid user and password combination	Returns a session cookie.

3.3.3 Password Retrieval, Data Access Utilities

verifyUserSecretCode(user, secretCode)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A valid user and secretCode combination, which however is not the correct one	Returns false.

A correct and valid user and secretCode combination	Returns true.
updateUserPassword(user, secretCode, newPassword)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A valid user and secretCode combination, which however is not the correct one	An InvalidSecurityLevelException is raised.
A correct and valid user and secretCode combination, but an incorrectly formatted password	An InvalidArgumentFormatException is raised.
A correct and valid user and secretCode combination, and a correctly formatted password	Updates the user password in the database.

3.3.4 Settings Management, Data Access Utilities

getUserSettings(userId)	
Input	*Effect*
A non-existing userId	An InvalidArgumentValueException is raised.
A valid userId	A structure containing all pairs $< setting, value >$ for the given userId preferences.
updateUserSettings(userId, settings)	
Input	*Effect*
A non-existing userId	An InvalidArgumentValueException is raised.
A null settings object	A NullArgumentException is raised.
An empty settings array	An InvalidArgumentValueException is raised.
A settings array containing some null values	A NullArgumentException is raised.
A valid userId and array of settings	The preferences of the given user are updated in the database accordingly to the $< setting, value >$ contained in the settings array.

3.4 Integration Between Subsystems

3.4.1 Remote Services Interface, Taxi Management System

requestTaxi(passengerId, passengerLocation, destination)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A passengerId not correctly formatted	An InvalidArgumentFormatException is raised.
A passengerLocation whose coordinates are invalid	An InvalidLocationException is raised.
A destination whose coordinates are invalid	An InvalidLocationException is raised.
A passengerLocation outside the city	A passengerId not correctly formatted
A valid set of parameters	A new request is created and handled; refer to the RASD for the specific outcomes of this operation.
endRide(taxiId, currentLocation)	
Input	*Effect*
A null location	A NullArgumentException is raised.
A currentLocation whose coordinates are invalid	An InvalidLocationException is raised.
An invalid taxiId	An InvalidArgumentValueException is raised.
A valid taxiId and currentLocation, but taxi is not on a ride	An InvalidOperationException is raised.
A valid taxiId and currentLocation, the taxi is on a ride and currentLocation is inside city	The ride is considered closed and is finalized in the database, the taxi changes its status to available.
A valid taxiId and currentLocation, the taxi is on a ride and currentLocation is outside city	The ride is considered closed and is finalized in the database, the taxi changes its status to outside-city.
acceptRide(taxiId, request)	
Input	*Effect*
A null request	A NullArgumentException is raised.
An invalid taxiId	An InvalidArgumentValueException is raised.

A request that has already been assigned to another taxi	An InvalidOperationException is raised.
A valid taxiId and request, but taxi is already on a ride	An InvalidOperationException is raised.
A valid taxiId and request, but taxi is outside-city	An InvalidOperationException is raised.
A valid taxiId and request, but taxi is unavailable	An InvalidOperationException is raised.
A valid taxiId and request, and the taxi is available	The taxi status is set to currently-riding, the taxi is removed from its zone queue, the request is marked as being served and the modifications are written to the database.
refuseRide(taxiId, request)	
Input	*Effect*
A null request	A NullArgumentException is raised.
An invalid taxiId	An InvalidArgumentValueException is raised.
A valid taxiId and request	The taxi is moved to the last position of its zone queue and the request is marked as refused by the specified taxi driver.
togglePressed(taxiId)	
Input	*Effect*
An invalid taxiId	An InvalidArgumentValueException is raised.
A valid taxiId and the taxi is available	The taxi status is set to unavailable and is written in the database.
A valid taxiId, the taxi is unavailable and the taxi is inside the city	The taxi status is set to available, the taxi is moved to the queue of its current zone and these modifications are written in the database.
A valid taxiId, the taxi is unavailable and the taxi is outside the city	An InvalidOperationException is raised.

3.4.2 Remote Services Interface, Account Management

insertPassenger(passenger)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A passenger with one or more null fields	An InvalidArgumentValueException is raised.
A passenger with one or more empty fields	An InvalidArgumentValueException is raised.
A passenger with the same mail address of an existing passenger	An InvalidArgumentValueException is raised.
A valid passenger	The passenger data is inserted in the database and a registration confirmation mail is sent to him. The passenger status is set to "pending confirmation".
deletePassenger(passengerId)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing passengerId	An InvalidArgumentValueException is raised.
A valid passengerId	The passenger data is removed from the database and a deletion confirmation mail is sent to him.
updatePassengerData(passenger)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A passenger with one or more null fields	An InvalidArgumentValueException is raised.
A passenger with one or more empty fields	An InvalidArgumentValueException is raised.
A passenger that is not present in the database	An InvalidArgumentValueException is raised.
A valid passenger	The passenger data is updated in the database.
confirmPassenger(passengerId)	
Input	*Effect*
A non-existing passengerId	An InvalidArgumentValueException is raised.
A passengerId that has already been confirmed	An InvalidArgumentValueException is raised.

A passengerId that hasn't already been confirmed	The passenger status in the database is set to "confirmed".
checkCredentials(user, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An inexistent user	An InvalidArgumentValueException is raised.
An empty password	An InvalidArgumentValueException is raised.
A valid user and password combination, which however is not the correct one	Returns an InvalidCredentialError.
A correct and valid user and password combination	Returns a session cookie.
verifyUserSecretCode(user, secretCode)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A valid user and secretCode combination, which however is not the correct one	Returns false.
A correct and valid user and secretCode combination	Returns true.
updateUserPassword(user, secretCode, newPassword)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A valid user and secretCode combination, which however is not the correct one	An InvalidSecurityLevelException is raised.
A correct and valid user and secretCode combination, but an correctly formatted password	An InvalidArgumentFormatException is raised.
A correct and valid user and secretCode combination, and a correctly formatted password	Updates the user password in the database.
getUserSettings(userId)	
Input	*Effect*

A non-existing userId	An InvalidArgumentValueException is raised.
A valid userId	A structure containing all pairs $< setting, value >$ for the given userId preferences.
updateUserSettings(userId, settings)	
Input	*Effect*
A non-existing userId	An InvalidArgumentValueException is raised.
A null settings object	A NullArgumentException is raised.
An empty settings array	An InvalidArgumentValueException is raised.
A settings array containing some null values	A NullArgumentException is raised.
A valid userId and array of settings	The preferences of the given user are updated in the database accordingly to the $< setting, value >$ contained in the settings array.

3.4.3 Remote Services Interface, System Administration

checkPassword(adminId, password)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An empty password	An InvalidArgumentValueException is raised.
An invalid adminId	An InvalidSecurityLevelException is raised.
A valid adminId and password, but password does not correspond to an authorized one	Returns false.
A valid adminId and password, and password does correspond to an authorized one	Returns true.
verifyPermission(appId, operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing appId	An InvalidSecurityLevelException is raised.

A non-existing operation	An InvalidSecurityLevelException is raised.
A valid appId and operation, but the app hasn't enough privileges to execute the desired operation	Returns false.
A valid appId and operation, and the app has enough privileges to execute the desired operation	Returns true.
grantPermission(adminId, appId, operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operation	An InvalidSecurityLevelException is raised.
An invalid adminId	An InvalidSecurityLevelException is raised.
A valid appId and operation	Insert a new $< appId, operation >$ pair in the permission table in the database.
revokePermission(adminId, appId, operation)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
A non-existing operation	An InvalidSecurityLevelException is raised.
An invalid adminId	An InvalidSecurityLevelException is raised.
A valid appId and operation, but the $< appId, operation >$ is not present in the database	An InvalidArgumentValueException is raised.
A valid appId and operation, and the $< appId, operation >$ is present in the database	Remove the $< appId, operation >$ pair from the permission table of the database.
askZoneList(adminId)	
Input	*Effect*
An invalid adminId	An InvalidSecurityLevelException is raised.
A valid adminId	Returns the zones that are stored in the database.

askDriverList(adminId)	
Input	*Effect*
An invalid adminId	An InvalidSecurityLevelException is raised.
A valid adminId	Returns a list of the taxi drivers that are stored in the database.
insertZones(adminId, zones)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An invalid adminId	An InvalidSecurityLevelException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing some null values	A NullArgumentException is raised.
An array of non-null, but invalid zones	An InvalidArgumentValueException is raised.
A valid adminId and a non-empty array of valid zones	The data associated with the existing zones is updated in the database, while new zones are inserted from scratch.
insertTaxiDrivers(adminId, taxiDriverList)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An invalid adminId	An InvalidSecurityLevelException is raised.
An empty array	An InvalidArgumentValueException is raised.
An array containing some null values	A NullArgumentException is raised.
An array of non-null, but invalid taxi drivers	An InvalidArgumentValueException is raised.
A valid adminId and a non-empty array of taxi drivers	The data associated with the existing taxi drivers is updated in the database, while new taxi drivers are inserted from scratch.
sendSetupData(adminId, data)	
Input	*Effect*
A null parameter	A NullArgumentException is raised.
An invalid adminId	An InvalidSecurityLevelException is raised.
An empty array	An InvalidArgumentValueException is raised.

An array containing some null values	A NullArgumentException is raised.
An array of non-null, but invalid or existing taxi drivers and zones	An InvalidArgumentValueException is raised.
A valid adminId and a non-empty array of non-existing taxi drivers and zones	The data associated with the new taxi drivers and zones are inserted in the database.

Chapter 4

Performance analysis

While a full fledged performance analysis of the entire myTaxiService infrastructure will be executed only in the system integration phase, it is still useful to perform some preliminary measures on components whose performances can be tested in isolation.

In particular, it is appropriate to verify that the applications for all the target mobile platforms, regardless whether they're destined to taxi drivers or to passengers, have reasonable CPU and main memory usages.

As specified in the RASD, the performance requirements of the mobile applications are the followings:

- They must run correctly on smartphones with single core processors clocked at 800Mhz or more.
- They must use no more than 64 MB of RAM to execute.

Furthermore, even though no strict value is fixed at this point, the storage occupation should be reasonably small. Given the current trends in the size of the image assets needed to support high resolution devices, it is reasonable to expect a 30MB size cap; however, this number should be reconsidered during the development phase taking into account the improvements in the smartphone and tablet technology that may occur meanwhile.

These tests will be performed using the appropriate performance analysis tool provided with the SDK of each mobile platform.

Chapter 5

Tools and Test Equipment Required

5.1 Tools

In order to test the various components of myTaxiService more effectively, we are going to make usage of a number of automated testing tools.

For what concerns the business logic components running in the Java Enterprise Edition runtime environment, we are going to take advantage of two tools.

The first one is the **Arquillian integration testing framework**. This tool enables us to execute tests against a Java container in order to check that the interaction between a component and its surrounding execution environment is happening correctly (as far as the Java application server is involved). Specifically, we are going to use Arquillian to verify that the right components are injected when dependency injection is specified, that the connections with the database are properly managed and similar container-level tests.

The second tool is the **JUnit framework**. Though this tool is primarily devoted to unit testing activities, it's still a valid instrument to verify that the interactions between components are producing the expected results. In particular, we are going to use it in order to verify that the correct objects are returned after a method invocation, that appropriate exceptions are raised when invalid parameters are passed to a method and other issues that may arise when components interact with each other.

Furthermore, as we have already mentioned briefly in the previous chapter of this document, we are going to use specific performance analysis tools to make sure that the applications for all the target mobile platforms, regardless whether they're destined to taxi drivers or to passengers, have reasonable CPU and main memory usages. Depending on the specific platform we are targeting, the tools we are going to use are:

- On Android: the Memory Profiler, Memory Monitor and Allocation Tracker tools to monitor main memory usage; the Traceview Walkthrough to monitor method execution time and the Battery Profiler to monitor energy consumption.
- On iOS: the full suite of performance analysis tools provided by the Xcode IDE. This includes Instruments as a general performance profiling tool, MallocDebug to find memory leaks, Activity Monitor and BigTop to monitor system statistics such as CPU, disk, network and memory usage.
- On Windows Phone: the Windows Phone Application Analysis toolkit, specifically the Windows Performance Analyzer tool.

Finally, it should be noted that despite the usage of automated testing tools, some of the planned testing activities will also require a significant amount of manual operations, especially to devise the appropriate set of testing data.

5.2 Test Equipment

All the integration testing activities have to be performed within a specific testing environment.

Since myTaxiService incorporates both a set of client components and a backend infrastructure, we must define the characteristics of the devices that have to be used in each of these two areas.

For what concerns the mobile side of the testing environment, the following devices are required:

- At least one Android smartphone for each display size from 3" to 6" at steps of 1/2".
- At least one Android tablet for each display size from 7" to 12" at steps of 1/2".
- At least one iOS smartphone for each member of the iOS product family.
- At least one iOS tablet for each display size of the iOS product family.
- At least one Windows Phone smartphone for each display size from 3" to 6" at steps of 1/2".

These devices will be used to test both the native mobile applications and the mobile versions of the web applications. It should be noted that these are general guidelines to drive the selection of the testing devices in a way

that covers the widest range of possible configurations. Some display sizes or resolutions may not be offered by all product families.

As a general note, we should consider the possibility of performing an analysis of the smartphone market to identify the most common display sizes and resolutions right before starting the integration testing phase, in order to better reflect the typical usage scenarios we will encounter in the real operating environment.

Regarding the desktop web applications, they will be tested using a set of normal desktop and notebook computers. There are no specific requirements on display resolution, operating system and processing power.

As for the backend testing, the business logic components should be deployed on a cloud infrastructure that closely mimics the one that will be used in the operating environment. Specifically, the testing cloud infrastructure needs to run the same operating system, the same Java Enterprise Application Server, the same Notification System and Remote Services Interface middleware (message brokers) and the same DBMS. As such, it is strongly suggested to use a scaled down version of the final operating cloud infrastructure chosen from the same service provider.

Depending on the actual implementation decisions, the specific software components may change. As a preliminary draft we assume to be using the **Red Hat OpenShift cloud infrastructure**, that is built upon the following software components:

- The **Red Had Enterprise Linux** distribution
- The **Java Enterprise Edition** runtime
- The **GlassFish Java Application Server**
- The **GlassFish Message Broker**
- The **Apache Web Server** as an HTTP load balancer
- The **Oracle Database Management System**.

Chapter 6

Required Program Stubs and Test Data

6.1 Program Stubs and Drivers

As we have mentioned in the Integration Testing Strategy section of this document, we are going to adopt a bottom-up approach to component integration and testing.

Because of this choice, we are going to need a number of drivers to actually perform the necessary method invocations on the components to be tested; this will be mainly accomplished in conjunction with the JUnit framework.

Here follows a list of all the drivers that will be developed as part of the integration testing phase, together with their specific role.

- **Data Access Driver**: this testing module will invoke the methods exposed by the **Data Access Utilities** component in order to test its interaction with the **DBMS**.
- **Request Management Driver**: this testing module will invoke the methods exposed by the **Request Management** subcomponent, including those with package-level visibility, in order to test its interaction with the **Data Access Utilities**, the **Notification System**, the **Location Management** and the **Taxi Management** components.
- **Reservation Management Driver**: this testing module will invoke the methods exposed by the **Reservation Management** subcomponent, including those with package-level visibility, in order to test its interaction with the **Data Access Utilities**, the **Notification System** and the **Request Management** components.
- **Location Management Driver**: this testing module will invoke the methods exposed by the **Location Management** subcomponent, in-

cluding those with package-level visibility, in order to test its interaction with the **Mapping Service** external component.

- **Taxi Management Driver**: this testing module will invoke the methods exposed by the **Taxi Management** subcomponent, including those with package-level visibility, in order to test its interaction with the **Data Access Utilities**, the **Notification System**, the **Location Management** and the **Mapping Service** components.

- **API permissions Management Driver**, **Zone Divison Management Driver**, **Taxi Driver Management Driver**, **Service Statistics Driver**, **Plugin Management**, **Passenger Registration Driver**, **Login Driver**, **Password Retrieval Driver** and **Settings Management Driver**: each testing module will invoke the methods exposed by its correspondent component to test its interaction with the **Data Access Utilities** and the **Notification System** components.

- **Taxi Management Driver**: this testing module will invoke the methods exposed by the **Taxi Management** subsystem to test its interactions with the **Data Access Utilities**, the **Notification System** and the **Mapping Service** components.

- **Account Management Driver**: this testing module will invoke the methods exposed by the **Account Management** subsystem to test its interactions with the **Data Access Utilities** and the **Notification System** components.

- **System Administration Driver**: this testing module will invoke the methods exposed by the **Taxi Management** subsystem to test its interactions with the **Data Access Utilities** and the **Notification System** components.

While the bottom-up approach in general doesn't require the usage of any stubs as the system is developed from the ground up, a full test of the core system isn't possible without introducing a few of them. In fact, there is a mutual dependency between the clients (which send requests) and the core system (which replies to them). Since we are developing and integrating the system from the core, we are going to introduce stubs to simulate the presence of clients until they are fully developed. In practice, the only purpose of these stubs is to write on a log that they have correctly received the messages.

6.2 Test Data

In order to be able to perform the battery of tests that we have specified, we are going to need:

- A list of both valid and invalid candidate taxi drivers to test the **Taxi Driver Management** component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Taxi license not compliant with the legal format
  - Driving license not compliant with the legal format
- A list of both valid and invalid candidate passengers to test the **Passenger Registration** component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Invalid mobile phone number
  - Invalid email address
- A list of both valid and invalid candidate city zones to test the **Zone Division Management** component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Zones built as sequences of location vertices not producing a convex area, including the degenerate case in which the set has cardinality less than three
  - Zones built as sequences of vertices representing invalid or null locations
  - Overlapping zones
- A list of both valid and invalid candidate taxi requests to test the **Request Management** component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Location is outside the city

- A list of both valid and invalid candidate taxi reservations to test the **Reservation Management** component. The set should contain instances exhibiting the following problems:
  - Null object
  - Null fields
  - Source location is outside the city
  - The time of the meeting does not respect the validity range

  More specific information about the required test data can be found by analyzing the inputs of all the test cases described in chapter 3.

Appendix A

Changelog

- Version 1.1: fixed small typos.
- Version 1.0: initial release.

Appendix B

Hours of work

To redact this document, we spent 30 hours per person.