POLITECNICO DI MILANO

Master Degree course in Computer Science Engineering

Fourth Assignment: Test Plan Document

Instructor: Prof. Elisabetta Di Nitto

Authors:

Brunato Andrea	851413
Costantini Lorenzo	852599
Dell’Orto Alessandro	853050

TABLE OF CONTENT

1. Introduction

1.1 Revision History

Currently there are no other versions of this document.

1.2 Purpose

Integration test plan is useful for giving the right directives on when and how the developer team should begin to test the interfaces functionalities offered and used between the various components. The main goal of this activity is to detect and delete all possible errors and bugs among the interfacing between the different components.

1.3 Scope
This document regards the EasyTaxi project whose requirements and design phase is described in the correspondent reference documents written below.
In particular, this document should be read alongside the Design Document of EasyTaxi.

1.4 List of Definitions and Abbreviations
Driver: Piece of software used in the bottom-up testing approach simulating only the external behaviour of the corresponding component.
Stub: Piece of software that works similar to the component which is referenced by the component being tested, but it is much simpler that the actual component.
Taxi Driver: In order to avoid ambiguity whenever in this document it’s written Taxi Driver with the “Driver” word with the capital letter, it is intended as the driver (piece of software) relating to the Taxi Driver Manager Component.
Request Driver: this is the driver of the Request Manager and not the one of the Request Collector.

In the documents we used the following abbreviations:

I1,I2,I3...: Identifier for the interface to be tested.
I1T1,I1T2,I1T3...: Identifier for a test that is planned to be run on the specific exposed interface.

1.5 List Of Reference Documents
- Fourth assignment description
- Requirement Analysis and Specification Document of our EasyTaxi project
- Design Document of EasyTaxi

2. Integration Strategy

2.1 Entry Criteria

During the application development, the integration test for a component will begin only when the following conditions are met:

- The implementation of the component whose integration is going to be tested, is already completed by developers. No other additional functions have to be coded inside that component.
- The considered components have already passed every unit test.

For what regards integration of two subsystems, it's necessary that all the components inside both subsystems have been unit tested and their integration limited to the scope of the correspondent subsystems has been successfully proved.

2.2 Elements to be Integrated

The elements whose integration steps will be described later in this document are the components visible in the section 3.C or 3.D of the Design Document. The section 3.F of the Design Document is really important because it describes the meaning of the interfaces that are going to be tested in this document. For example, integration test I1 described later in the document refers to the login of taxi drivers, I2 both to the login and registration interfaces that link the User Manager to the Account Manager, and so on.
The subsystems we have identified are the tiers of our application (client, server and database subsystems).

2.3 Integration Testing Strategy

The chosen approach is the bottom-up integration strategy: we have decided to begin with the most independent components and then proceed to test the more dependent ones.
This approach is the best for our system because the most independent components are the most difficult and the longest to implement.

It is more likely to find some bug with this components than with the others: applying tests beginning from the most critical ones will grant the developer team the plenty of time to correct every kind of error and inconsistency they may find. The bottom up approach has been chosen also for other aspects: indeed the main goal is to test every component of our system without forcing the development of complex test classes.

With the bottom-up approach the developers will only have to build some very simple drivers that mock calling methods instead of more difficult stubs that simulates the behavior of some called class when a top down approach is chosen. We designed this strategy also aiming to keep the complexity of the interaction between different components low such that if a test fails it will be trivial to understand where the problem is.

For example, the Request Manager component is the one with the highest number of dependencies and at the same time it’s also one of the easiest to implement. It uses mostly methods exposed by other components like the Queue Manager and the Shared Ride Manager, which are computationally complex as we have seen in the design document. The Request Manager complexity consists in calling those methods at the right moment, managing the control flow of the sequential operations in order to let the request processing advance until the user finds a taxi driver for his ride. If we had decided to start to implement, unit test and test the integration of the Request Manager at the first step of our integration testing plan, we would have had to create stubs simulating the logic behind the management of taxi queues and the search of compatible shared rides.

Using this approach, we can also reuse the code of the Request Manager Driver for the future implementation, given the fact that it acts as a controller calling many functions of other components and that the function of a driver is the same.

The disadvantages of using a bottom up approach is that the application can be visible only at the end of the process, because the components in the client tier are the last ones to be developed and integrated.

2.4 Sequence of Component/Function Integration

We present the order of component integration, but the integration is not forced to be completely sequential and some tasks can be performed in parallel despite having an incremental numeration. While a person of the team is working on the unit and integration testing of the Shared Ride Manager, another person could work on the Queue Manager and the other one on the Account Manager.

It's very probable that the Shared Ride Manager and the Queue Manager development could last more time with respect to the Account Manager which should be a simple task. The person who finishes the latter task can start the development and testing of the Request and Reservation Collector, and at the end of this new work he can begin to develop and integrate the Request Manager with all the other ready components.

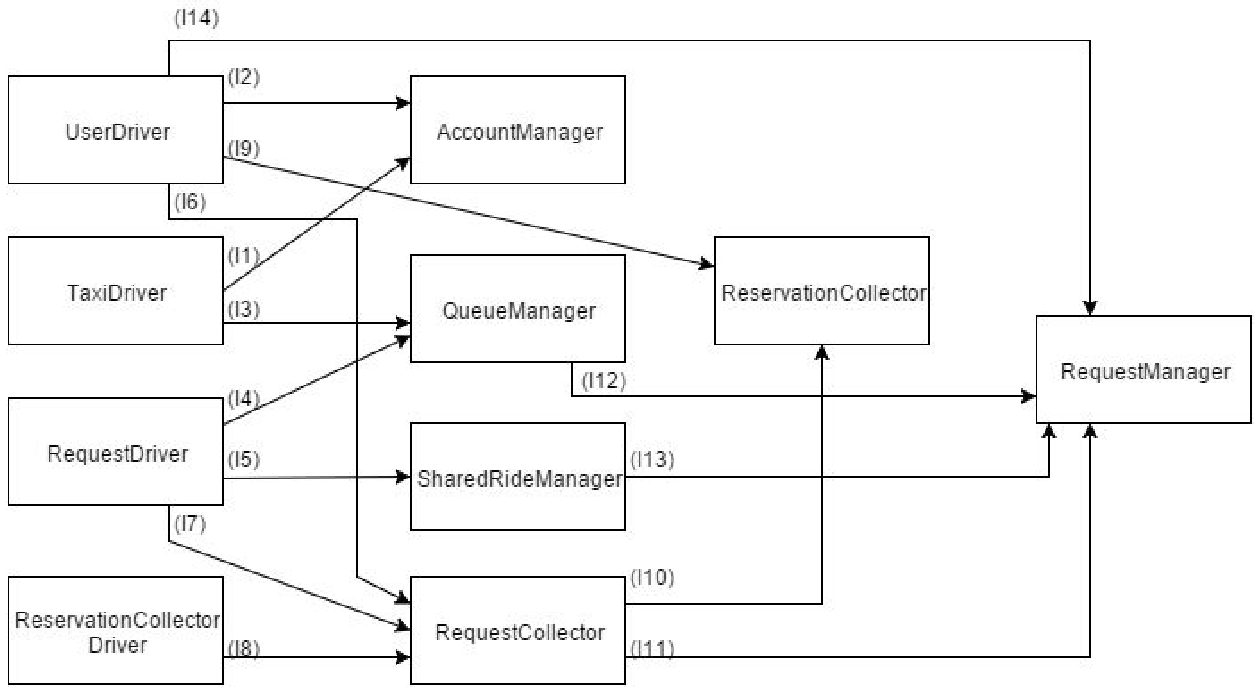
If some components of the server tier are still not ready (we think at the Queue Manager and Shared Ride Manager) one of the team member can start in parallel the development of the User Manager component who is not directly related to these two mentioned component.

The Taxi Driver Manager should be the last element to be tested and integrated; according to the bottom-up approach the Queue Manager needs to be finished and tested before the Taxi Driver front-end application.

However, if there is an urgency to release the project and the Shared Ride Manager is not completed yet, the Taxi Driver Manager implementation can be anticipated and the project can be released initially without the shared ride additional feature, after having tested the integration of the client and the server subsystem with a part of the Request Manager and the Shared Ride Manager still incomplete.

2.4.1 Software Integration Sequence

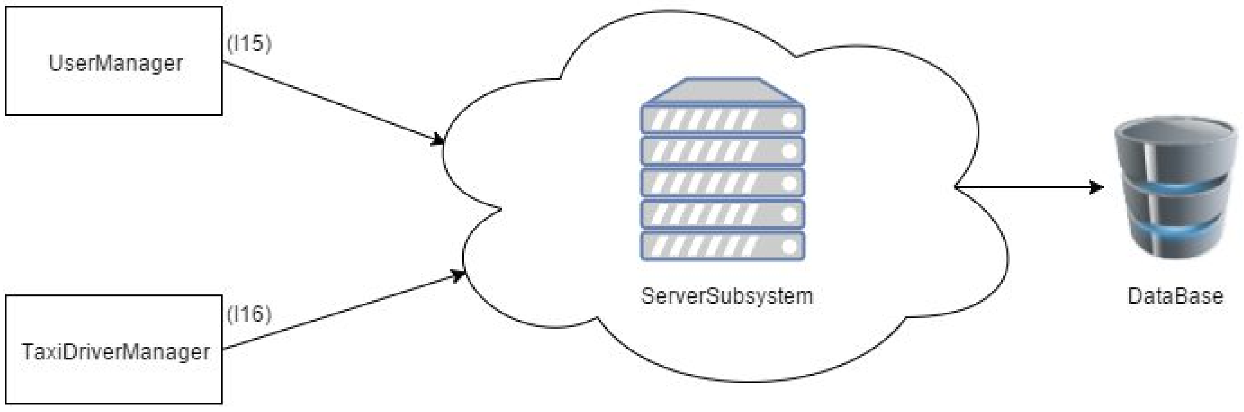
The following diagram shows the order of integration of the components of the server subsystem: as written before, tasks from I1 to I8 can be done in parallel. After I8 the Request Collector interfaces (I9 and I10) are tested and there's no need of the Reservation Collector driver anymore. In fact the Reservation Collector is then tested with the real and finished Request Collector component. After the realization of the Request Manager component, the latter is directly tested with the other server components already completed (I11, I12, I13).



2.4.2 Subsystem Integration Sequence

After the integration testing of the Request Manager component, the whole server subsystem is ready to be tested with respect to the client tier. The User Manager and the Taxi Driver Manager have been already unit tested and with the integration test I15 and I16 we see if by substituting those components to their previous drivers we discover system errors.

The test of the integration with the database system has been done before the integration test I1 and it’s not covered in this document. With this result, it’s possible to add tuples in the database to perform some tests, as written in the section 3, being sure that in case of test failure the problem it’s not related to the database.



3. Individual Steps and Test Description

In the environmental needs we don’t specify the drivers needed to do the test because it’s already visible in the section 2.4.
The environmental needs are considered as preconditions of the tests.

Test Case Identifier	I1T1
Tested Interface	Login (Taxi driver)
Tested Item	Account Manager
Input Specifications	The Taxi Driver sends to the AccountManager a correct input for the login procedure.
Output Specification	The login is successful
Environmental Needs	The database is initialized with a tuple representing the credentials regarding a driver account.

Test Case Identifier	I1T2
Tested Interface	Login (Taxi driver)
Tested Item	Account Manager
Input Specifications	The Taxi Driver sends to the Account Manager an input for the login procedure.
Output Specification	The login into the system is negated.
Environmental Needs	The database is initialized with a tuple representing some credentials regarding a driver account that doesn’t match with the input.

Test Case Identifier	I1T3
Tested Interface	Login (Taxi driver)
Tested Item	Account Manager
Input Specifications	Taxi Driver calls the method to be logged-out the system.
Output Specification	The Account Manager logs out the corresponding taxi driver
Environmental Needs	The database is initialized with a tuple representing the credentials regarding a driver account. The login into the system has been already done.

Test Case Identifier	I2T1
Tested Interface	Login (User)
Tested Item	Account Manager
Input Specifications	The User Driver sends to the AccountManager a correct input for the login procedures.
Output Specification	The login is successful
Environmental Needs	The database is initialized with a tuple representing the credential regarding an user account.

Test Case Identifier	I2T2
Tested Interface	Login (User)
Tested Item	Account Manager
Input Specifications	The User Driver sends to the AccountManager a incorrect input for the login procedures.
Output Specification	The login into the system is negated.
Environmental Needs	The database is initialized with a tuple representing some credentials regarding an user account that doesn’t match with the input.

Test Case Identifier	I2T3
Tested Interface	Login (User)
Tested Item	Account Manager
Input Specifications	User Driver calls the method to be logged-out the system.
Output Specification	The AccountManager logs out the corresponding user.
Environmental Needs	The database is initialized with a tuple representing the credentiality regarding a user account. The login into the system has been already done.

Test Case Identifier	I2T4
Tested Interface	Registration
Tested Item	Account Manager
Input Specifications	The User Driver sends to the AccountManager a correct input for the registration procedures.
Output Specification	The registration process is successful and a new tuple is added into the database.
Environmental Needs	None.

Test Case Identifier	I2T5
Tested Interface	Registration
Tested Item	Account Manager
Input Specifications	The User Driver sends to the AccountManager an incorrect input for the registration procedures.
Output Specification	The registration fails.
Environmental Needs	The database should be initialized with a tuple containing the same username of the one specified in input

Test Case Identifier	I3T1
Tested Interface	Change Taxi State
Tested Item	Queue Manager
Input Specifications	The Taxi Driver sends an HTTP request to change the state of a taxi from Available To Busy
Output Specification	The Taxi Driver state is changed to Busy
Environmental Needs	The queue of the zone in which the instance of taxi created is located is initialized with some mock taxis.

Test Case Identifier	I3T2
Tested Interface	Change Taxi State
Tested Item	Queue Manager
Input Specifications	The Taxi Driver sends an HTTP request to change its TaxiState from Busy To Available
Output Specification	The Taxi Driver state is changed to Available
Environmental Needs	The taxi instance is present in one of the queues of taxis of the Queue Manager

Test Case Identifier	I4T1
Tested Interface	Taxi Search
Tested Item	Queue Manager
Input Specifications	The Request Driver asks for an available taxi for a given instance of request.
Output Specification	The first available taxi in the stack is returned.
Environmental Needs	The queue is filled with some taxi instances. The Request Driver has a current request attribute different from null.

Test Case Identifier	I4T2
Tested Interface	Manage Taxi Driver Answer
Tested Item	QueueManager
Input Specifications	The Request Driver reports the answer of a taxi instance to a request. The answer is no, represented by a boolean attribute.
Output Specification	The corresponding taxi is put at the bottom of the stack
Environmental Needs	The stack is filled with some taxi instances.

Test Case Identifier	I4T3
Tested Interface	Manage Taxi Driver Answer
Tested Item	QueueManager
Input Specifications	The Request Driver reports the answer of a taxi instance to a request. The answer is yes, represented by a boolean attribute.
Output Specification	The corresponding taxi is deleted from the queue of the avalaible taxis of a certain zone.
Environmental Needs	The stack is filled with some taxi instances.

Test Case Identifier	I5T1
Tested Interface	Compute Rides
Tested Item	Shared Ride Manager
Input Specifications	The current request of the Request Driver
Output Specification	A compatible request instance. The database whose ER diagram is described in the DD is updated at the right row of the field “SharedWith”
Environmental Needs	The Request Driver has a current request attribute different from null and the database is filled in the Request table in order to let the algorithm to find a compatible shared route, which is described in the Design Document

Test Case Identifier	I5T2
Tested Interface	Compute Rides
Tested Item	Shared Ride Manager
Input Specifications	The current request of the Request Driver
Output Specification	No compatible request instances. The request isn’t shared.
Environmental Needs	he Request Driver has a current request attribute different from null and the database is filled in the Request table such that the algorithm mustn’t find a compatible shared route.

Test Case Identifier	I6T1
Tested Interface	Add Request
Tested Item	Request Collector
Input Specifications	An HTTP request from the User Driver containing a new simple or shared request
Output Specification	The request is added into the database as a new tuple and it’s also added at the bottom of the queue of pending requests.
Environmental Needs	None

Test Case Identifier	I6T2
Tested Interface	Add Request
Tested Item	Request Collector
Input Specifications	An HTTP request from the User Driver containing a new simple or shared request
Output Specification	The request is refused, it isn’t saved into the database and a notification is returned.
Environmental Needs	None

Test Case Identifier	I7T1
Tested Interface	Retrieve Request
Tested Item	Request Collector
Input Specifications	The Request Manager invokes the getter methods in order to retrieve a pending request.
Output Specification	The first pending request from the queue of the Request Collector
Environmental Needs	The queue of the Request Collector is initialized with some tuples representing some request instances.

Test Case Identifier	I8T1
Tested Interface	Move To Request
Tested Item	Request Collector
Input Specifications	A list of reservation instances
Output Specification	The reservations that are going to start 10 minutes before the booked time are added as new requests into the queue of pending requests
Environmental Needs	The database is initialized with some reservation instances and a timer is set. At regular time intervals the timer should let the method of the described interface to create the output specified above.

Test Case Identifier	I9T1
Tested Interface	Add Reservation
Tested Item	Reservation Collector
Input Specifications	An HTTP Request with a reservation instance to add into the database.
Output Specification	The Reservation collector accepts the reservation and saves it into a corresponding tuple into the database. A confirmation is sent to the user.
Environmental Needs	None

Test Case Identifier	I9T2
Tested Interface	Add Reservation
Tested Item	Reservation Collector
Input Specifications	An HTTP Request with a reservation instance to add into the database.
Output Specification	The Reservation collector doesn’t accept the reservation because the database contains already five tuples referred to the same user. A notification is sent to the user.
Environmental Needs	Some tuples in the database in order to show the above described output in case of test success.

Test Case Identifier	I9T3
Tested Interface	Delete Reservation
Tested Item	Reservation Collector
Input Specifications	An HTTP Request with a reservation instance to delete from the database.
Output Specification	The Reservation collector delete the reservation tuple from the database.
Environmental Needs	A tuple in the reservation table correspondent to the input specification data.

Test Case Identifier	I9T4
Tested Interface	Delete Reservation
Tested Item	Reservation Collector
Input Specifications	An HTTP Request with a reservation instance to delete into the database.
Output Specification	The Reservation collector doesn’t delete the reservation because it lasts too little time to the start of the booked ride.
Environmental Needs	A tuple in the database whose data are inserted in order to show the above described output in case of test success.

The tests from I10 to I13 test the Request Manager component and that the sequence of operations for the request processing is done in the correct order. It also tests that by introducing the real components instead of drivers no errors of integration are introduced.

Test Case Identifier	I14T1
Tested Interface	Notify Fee
Tested Item	RequestManager
Input Specifications	The current request processed by the Request Manager
Output Specification	Notification via SMS or E-mail of the acceptance of a request
Environmental Needs	The database should contain tuples in the Request and User table to verify that a coherent notification is sent

4. Tools and Test Equipment Required

The tools that will be used in order to perform all tests are Mockito and Arquillian with JUnit.
With these tools the developer team is asked to build some drivers implementing a really trivial logic.
It is also planned a session on manual testing for all what concerns the web presentation and GUI of our system. Manual testing in this case is preferable because it’s the most intuitive and simpler way to test graphical interfaces such as buttons, input form and so on.
After the end of integration testing session some performance tests with the JMeter tool is recommended to verify that the most critical algorithms reacts well satisfying the nonfunctional requirements declared in the RASD, in particular the scalability: the algorithm to be tested are the search of compatible shared requests when the number of tuples in the request table become really large and the search in available taxi queues when the number of users who asks for requests grows really fast.

5. Program Stubs and Test Data Required

All the methods of a certain component, which are used as calling methods for the test of other components interfaces, will be allocated in the same class; these classes containing related drivers are indicated in the section 2.4.1 picture. To give an example of the notation used, User Driver contains all the methods (drivers) of the User Manager, Taxi Driver contains the methods of the Taxi Driver Manager, etc.
The following drivers will be implemented:

- **User Driver**: It will mock methods generated by all the user possible actions (i.e. login, registration, forwarding of reservations and requests)
- **Taxi Driver**: It will mock methods generating all the taxi driver possible actions: it mainly has to mock the 'Login' and the 'Change Taxi State' interfaces.
- **Request Driver**: It will mock many interfaces that links the Request Manager components with most of the other server components. It will have a method that will ask the first pending request to the Request Collector in order to test the latter component, other methods useful for giving a start and an input for the Shared Ride Manager and Queue Manager computations.
- **Reservation Collector Driver**: It will only be useful for the operation of converting a reservation into a request when the start of the booked ride is incoming.

Beside these drivers, it will be also added into the database some initial testing tuples as written in section 3.
We have managed to avoid writing stubs as mentioned in section 2.

6. Hours Of Work

Andrea Brunato: 12 hours
Alessandro Dell'Orto: 12 hours
Lorenzo Costantini: 12 hours