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

Navid Heidari (798726) Hamidreza Hanafi (841408)

Tuesday 5th July, 2016
version 1.0

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 *SmartCityAdivsor*.

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 Milano city*.
The government wants to offer citizens and their control center an application which improve the quality of life in the city.
So we received the request to design and implement this application, called *SmartCityAdivsor* which has basically these great objectives:

- If the level of CO_2 is too high or in case of any special situation managed by the control center (e.g., the arrival of some VIP, an accident,), limit the traffic that enters in the city center by diverting it through paths that avoid the center. This is done both by controlling the traffic lights and by alerting the citizens through the app and through some large displays that are installed at the main entrances of the city.

- When a citizen signals through the app that he/she has to go to an emergency room, provide suggestions on which hospital to choose based on: i) the problem of the citizen and the specializations available in the hospitals, ii) the status of queues in the various emergency rooms, iii) the location of the citizen and iv) the situation of traffic.

- If the level of CO_2 is too high, it should send alert to control center to manange city energy consumption to reduce CO_2 level in the air.

- Help citizens to find empty parking places with respect to their location in the city center.

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
- Project Document

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, ATM 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:

- **Citizen applications**: Are the (mobile and web) applications that allows the citizen to interact with the system. They are basically a presentation layer and do not provide any internal logic.
- **ControlCenter application**: Is the web application that allows the ControlCenter to interact with the system.
- **Account Manager**: It manages the accounts, both of Citizens and Control Centers. It also handles the login and logout phases of both users and the registration for the Citizen and the relative account deleting. It exposes the following interfaces:
  - `Citizen`: for the interaction with the Citizens
  - `Control Center`: for the interaction with the Control Centers
- **Emergency Room Manager**: The Emergency Room Manager is responsible for finding best emergency room for Citizens. Internally it is composed of the following components each with a precise job:

- *SearchHandler*

  It exposes the following interfaces:

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

- **Parking Manager**: The Parking Manager is responsible for the for the management of the life cycle of parking searches and reservations. Internally it is composed of the following components each with a precise job:

  - `SearchHandler`
  - `ReservationHandler`

  It exposes the following interfaces:

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

- **TrafficLight Manager**: The Traffic Manager is responsible for the limiting the traffic in the city center through managing traffic lights. It also inform citizens about the changes.
  It exposes the following interfaces:

  - `TrafficLightHandler`: which can be used by the control center web browser (web server)

- **Energy Manager**: The Energy Manager is responsible for the reducing the CO_2 in the city center through managing city lights and etc. It also inform citizens about the changes.
  It exposes the following interfaces:

  - `EnergyHandler`: which can be used by the control center web browser (web server)

- **Notification Manager**: The notification manager in in charge of taking messages from other components and send them to target user(s).
  It exposes the following interfaces:

  - `NotificationManagement`: which can be used by the control center web browser (web server) or Citizen's mobile and web applications.

- **Geographic Engine**: the roles of this component are to implement the algorithms to calculate the time needed to reach a position and to get best public transportation to reach the destination.
  It exposes the following interfaces:

  - `ATMCalculator`: which provides caulculations to get best public transportation to reach the destination
  - `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

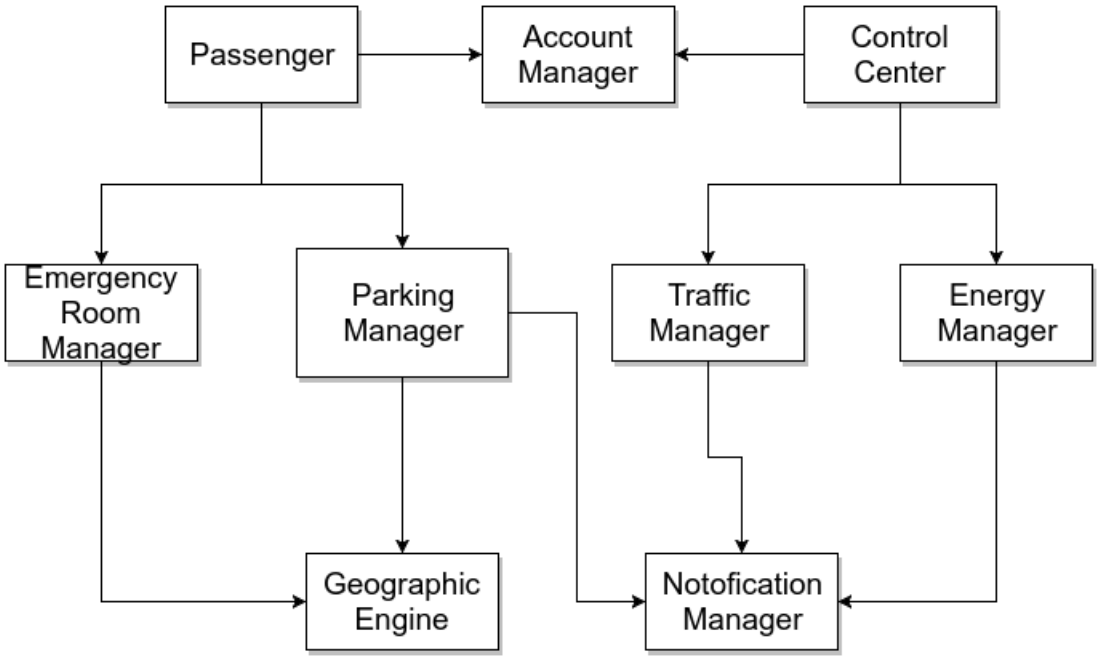

Figure 2.1: Components of the system and their dependencies

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 ParkingManager component needs for its execution components GeographicEngine and NotificationManager.

This is the designed integration plan for the components:

ID	**Components interaction**	References
I1	EmergencyRoomManager, ParkingManager $\Longrightarrow$ GeographicEngine	3.1
I2	ParkingManager, TrafficManager, EnergyManager $\Longrightarrow$ NotificationManager	3.2
I3	Citizen, ControlCenter $\Longrightarrow$ AccountManager	3.3
I4	Citizen $\Longrightarrow$ ParkingManager, EmergencyRoomManager	3.4
I5	ControlCenter $\Longrightarrow$ EnergyManager, TrafficManager	3.5

Table 2.1: Integration sequence

Chapter 3

Individual steps and test description

3.1 Integration I1

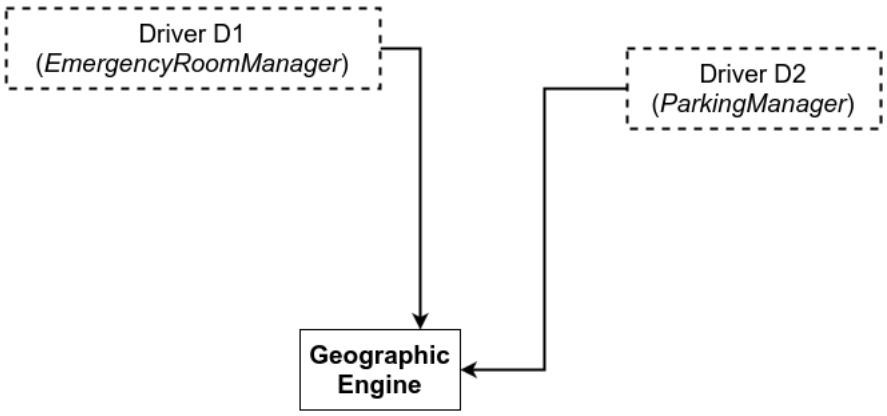

Figure 3.1: Drivers for the integration I1

Test case ID	I1.1
Modules interactions	EmergencyRoomManager $\Longrightarrow$ GeographicEngine
Drivers	D1 = EmergencyRoomManager
Input	Ask the best public transportation to reach the destination
Required output	The GeographicEngine must output a series of public transportations if the location is valid, otherwise an error

Table 3.1: Test case I1.1

Test case ID	I1.2
Modules interactions	ParkingManager $\Longrightarrow$ GeographicEngine
Drivers	D2 = ParkingManager
Input	The ParkingManager driver asks the GeographicEngine for the time to reach the destination according to user's location
Required output	The GeographicEngine must provide the approximate time

Table 3.2: Test case I1.2

3.2 Integration I2

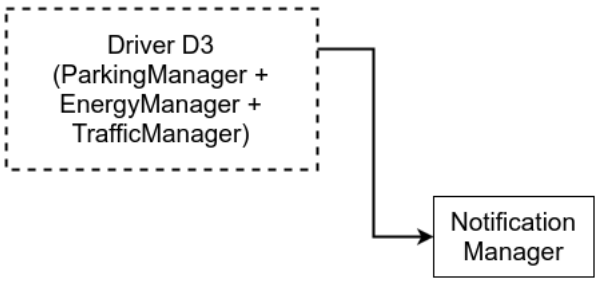

Figure 3.2: Drivers for the integration I2

Test case ID	I2.1
Modules interactions	ParkingManager, EnergyManager, TrafficManager $\Longrightarrow$ NotificationManager
Drivers	D3 = ParkingManager, EnergyManager, TrafficManager
Input	Sends notification to target users
Required output	If the notification successfully sent to the user.

Table 3.3: Test case I2.1

3.3 Integration I3

Figure 3.3: Drivers for the integration I3

Test case ID	I3.1
Modules interactions	Citizen $\Longrightarrow$ AccountManager
Drivers	D4 = Citizen
Input	The Citizen 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 3.4: Test case I3.1

Test case ID	I3.2
Modules interactions	ControlCenter $\Longrightarrow$ AccountManager
Drivers	D5 = ControlCenter
Input	The ControlCenter 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 3.5: Test case I3.2

3.4 Integration I4

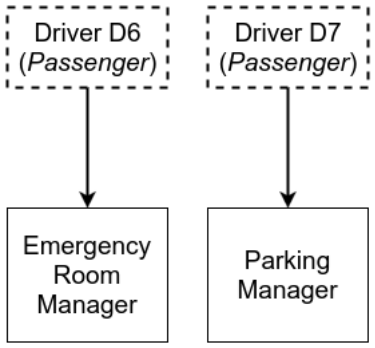

Figure 3.4: Drivers for the integration I4

Test case ID	I4.1
Modules interactions	Citizen $\Longrightarrow$ EmergencyRoomManager
Drivers	D6 = Citizen
Input	The Citizen makes different search requests with different parameters: locations and problems must be both valid and not
Required output	The EmergencyRoomManager must respond with an acknowledgement in case a emergency room has been found, or with an error, in case the input data are not valid or there are no emergency rooms available

Table 3.6: Test case I4.1

Test case ID	I4.2
Modules interactions	Citizen $\Longrightarrow$ ParkingManager
Drivers	D7 = Citizen
Input	The Citizen makes different reservations with different parameters: locations and time must be tried with both valid and not valid values
Required output	The ParkingManager 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 3.7: Test case I4.2

3.5 Integration I5

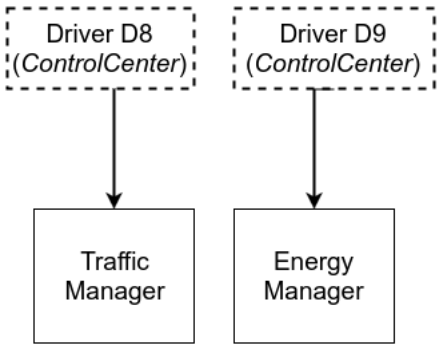

Figure 3.5: Drivers for the integration I5

Test case ID	I5.1
Modules interactions	ControlCenter $\Longrightarrow$ TrafficManager
Drivers	D8 = TrafficManager
Input	The ControlCenter makes different requests with different parameters: location, times and durations
Required output	The TrafficManager must respond with an acknowledgement in case of in handles successfully, or with an error, in case the input data are not valid

Table 3.8: Test case I5.1

Test case ID	I5.2
Modules interactions	ControlCenter $\Longrightarrow$ EnergyManager
Drivers	D9 = ControlCenter
Input	The ControlCenter makes different requests with different parameters: percentage, times and durations
Required output	The EnergyManager must respond with an acknowledgement in case of in handles successfully, or with an error, in case the input data are not valid

Table 3.9: Test case I5.2

3.6 Final integration - Complete test of the system

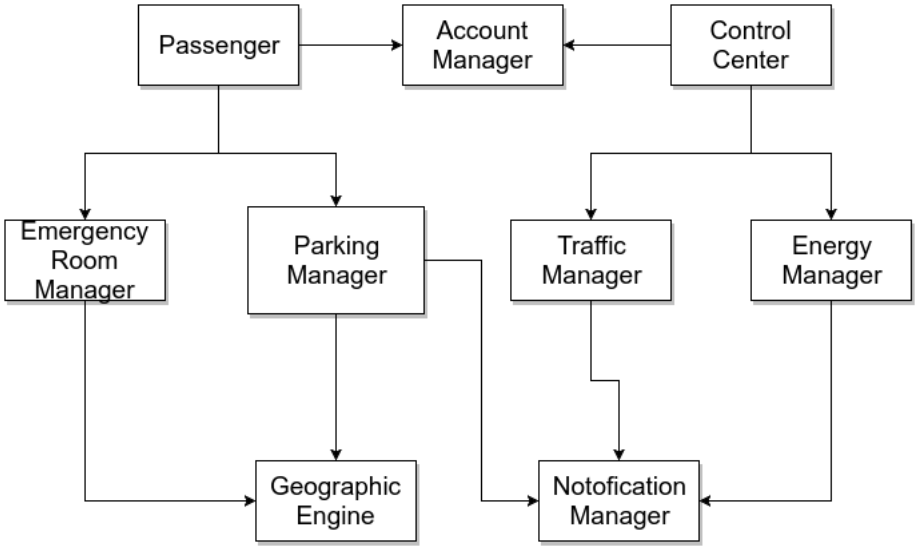

Figure 3.6: Final integration test - The complete system

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.

3.7 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.

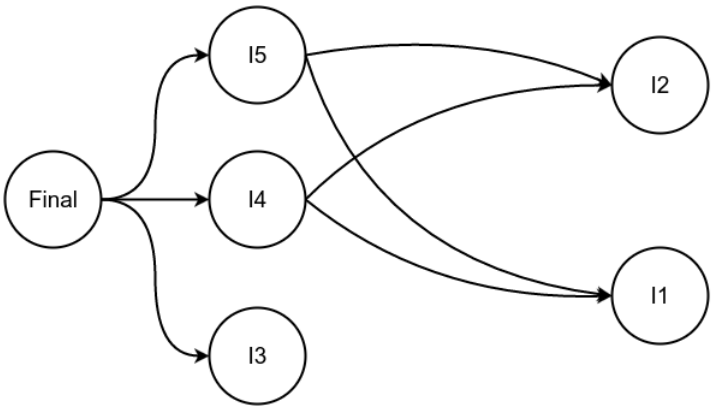

Figure 3.7: Dependency graph of the integrations

A possible integration plan is showed in figure 3.8

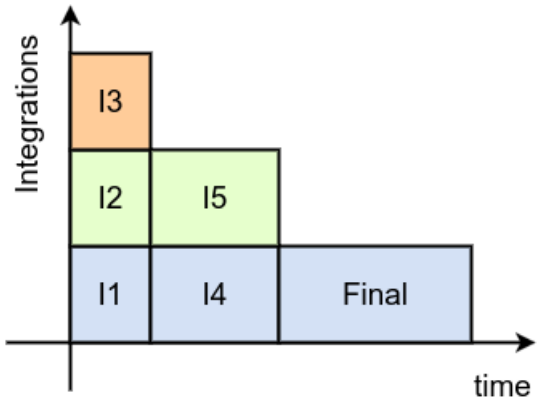

Figure 3.8: Schedule of a possible integration plan

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 I2 and I3 proceed in parallel
2. Follow integrations I4 and I5 in parallel
3. 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)	EmergencyRoomManager
Tested component	GeographicEngine
Tested interface	`ATMCalculator`: • `ATMSeries getATM(Location source, Location destination)`
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)	ParkingManager
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 5.2: Driver D2

Driver name	D3
Mocked component(s)	ParkingManager + EnergyManager + TrafficManager,
Tested component	NotificationManager
Tested interface	`NotificationManagement`: • `Boolean sendNotification(String Message, User targetUser, Time time, Boolean sendToAll)`
Required Data	message of the notification, Target user, the time it should be send to user and whether it should be send to all or specific user

Table 5.3: Driver D3

Driver name	D4
Mocked component(s)	Citizen
Tested component	AccountManager
Tested interface	`Citizen`: • `Citizen login(String username, String password)` • `void logout(Citizen citizen)` • `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 5.4: Driver D4

Driver name	D5
Mocked component(s)	ControlCenter
Tested component	AccountManager
Tested interface	`ControlCenter`: • `ControlCenter login(String username, String password)` • `void logout(ControlCenter controlcenter)`
Required Data	Set of strings for username and passwords. Also username and password empty and incorrect

Table 5.5: Driver D5

Driver name	D6
Mocked component(s)	Citizen
Tested component	EmergencyRoomManager
Tested interface	`EmergencyRoomHandler`: • `EmergencyRooms searchRoom(Location source, Problem problem)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...) and the problem of citizen

Table 5.6: Driver D6

Driver name	D7
Mocked component(s)	Citizen
Tested component	Citizen
Tested interface	`ParkingManager`: • `ParkingSpots Search(Location location` • `Response makeReservation(Location location, Date date, Time time, Citizen citizen)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...). Set of dates and the user who requested

Table 5.7: Driver D7

Driver name	D8
Mocked component(s)	ControlCenter
Tested component	**TrafficManager**
Tested interface	`TrafficHandler`: • `Boolean limitTraffic(Location location, Date date, Time time, int duration)`
Required Data	Set of locations: both inside the city (valid) and out (not valid), or incorrect (misspelled,ecc...), set of dates and duration

Table 5.8: Driver D8

Driver name	D9
Mocked component(s)	ControlCenter
Tested component	**EnergyManager**
Tested interface	`EnergyHandler`: • `Boolean reduceEnergy(int percentage, Date date, Time time, int duration)`
Required Data	Set of numbers, set of dates and duration

Table 5.9: Driver D9

Chapter 6

Appendix

6.1 Working hours

- Navid Heidari: 8 hours
- Hamidreza Hanafi: 8 hours