Politecnico di Milano

A.A. 2015-2016

Software Engineering 2 Project – “SmartCityAdvisor”

Prof. Di Nitto Elisabetta

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

Mirko Basilico, Federico Dazzan

Table of Contents

1. INTRODUCTION

1.1 Purpose

This document describes the plans for testing the integration of the components developed for SmartCityAdvisor's system. The purpose of this document is to test the interfaces between the components as described in the relative Design Document. Every team member who co-operates in the integration tests should read this document.

1.2 Scope

SmartCityAdvisor provides the following services to the city of Milan and to its citizens:

1. If the level of CO2 is too high or in case of any special situation managed by the government of the city (e.g., the arrival of some VIP, an accident, …), the system limits 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.
2. 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 condition 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.
3. When a citizen wants to go to the city center by car, he/she can signal through the app the address of the place he/she wants to go, and the system will return the address of the closest available parking with corresponding availability.
4. When a citizen wants to reach a place in Milan using the public transport system, he/she can signal through the app the starting and the arrival address and the kind of public transport he/she wants to use (underground, tram or bus). After the citizen has put this data into the app, the system will show: i) name and position of the closest station where to get the desired public transport, ii) name and position of the arrival station, iii) the time when the public transport will arrive at the starting station, iv) the estimated time it will take to arrive to destination.

1.3 List of Definitions and Abbreviations

Driver: a routine that simulates a test call from parent component to child component.

Stub: It is a dummy module that provides the response that would be provided by the real. subelement

JUnit: A framework to write repeatable unit tests.

Sub-System: A system which is part of a larger system.

Entry Criteria: Set of conditions that must be met in order to commence a particular project phase or stage.

Client: Computer hardware or software that accesses the services made available by the server.

Server: Computer program or machine that waits for requests from clients and responds to them.

RASD: Requirements Analysis and Specification Document

DD: Design Document.

ITPD: Integration Test Plan Document (this document).

GUI: Graphical User Interface.

REST: Representational State Transfer (related information on chapter 2.7 of the DD).

DCC: Data Control Center (related information on chapter 1.3 of the RASD).

TCC: Traffic Control Center (related information on chapter 1.3 of the RASD).

1.4 List of Reference Documents

- Assignment 4 - integration test plan.pdf
- SmartCityAdvisor RASD.pdf
- SmartCityAdvisor DD.pdf
- Integration Test Plan Example.pdf

2. INTEGRATION STRATEGY

2.1 Entry Criteria

All the classes and functions must undergo and pass severe JUnit-based unit tests (done by a reasonable number of testers), which should discover issues and fix bugs. Moreover, all the software has to be accurately inspected in order to avoid mistakes hardly detectable during the integration phase that are not dependent on it. Eventually, the documentation of all classes and functions has to be complete and updated in order to be used as reference for integration testing development and reflect the current state of the project.

2.2 Elements to be Integrated

The detailed description of each component's function and the interaction between them is contained in the DD. The general architecture of SmartCityAdvisor can be divided in four main subsystem, each one formed by some components, as follows:

- Application subsystem
  - AccountManager
  - TrafficManager
  - HospitalManager
  - ParkingManager
  - TransportManager
  - DCCManager
  - TCCManager
  - Notifier

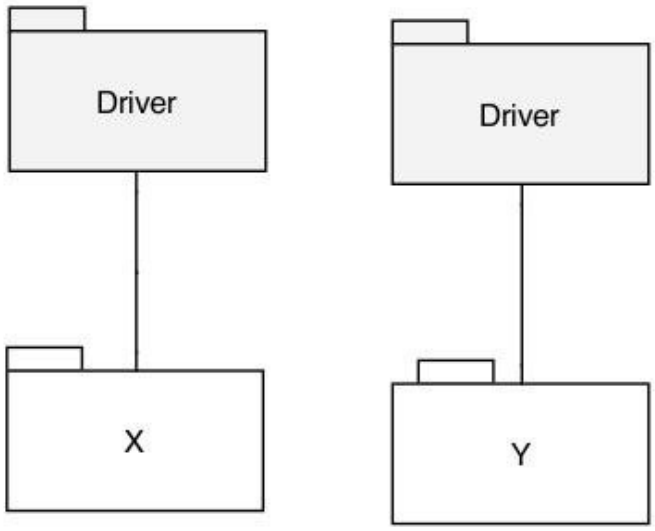
- Routing subsystem
  - Dispatcher
  - SecurityManager

- Client subsystem
  - CitizenWebGUI
  - CitizenMobileGUI
  - GovernmentGUI

- Storage subsystem
  - DataManager

2.3 Integration Testing Strategy

The elements to be tested consist of the integration of the modules developed in the DD for the SmartCityAdvisor project. In order to test them, we chose to adopt the bottom-up integration strategy, which consists in integrating a low hierarchy component with its parent. For each module, a routine that simulates a test call from a parent component to a child component is needed (Driver). The integration tests of lower level code modules are described in the corresponding components' unit tests. Unit tests are described in the Unit Test Plan Document, which we did not have to do (we assume it has been already done).



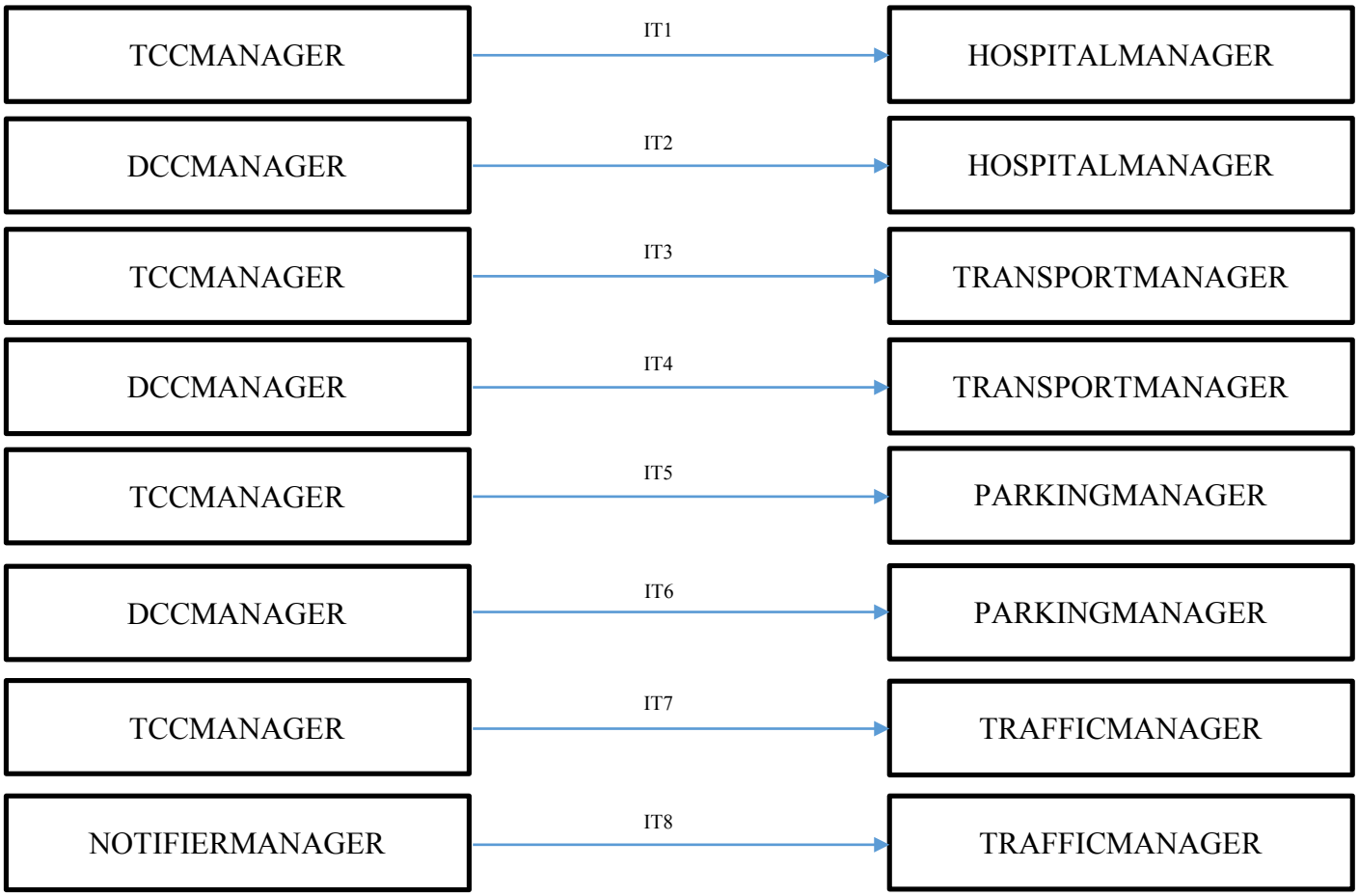
2.4 Sequence of Components/Functions Integrations

In this chapter we are going to expose in details how the integration of the various components and sub-systems is performed.

2.4.1 Software Integrations Sequence

Here we present the bottom-up integration for each identified subsystem, those subsystems have several components that must be checked and tested.

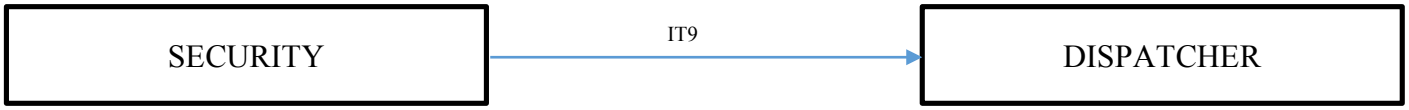
APPLICATION SUBSYSTEM



STORAGE SUBSYSTEM AND CLIENT SUBSYSTEM

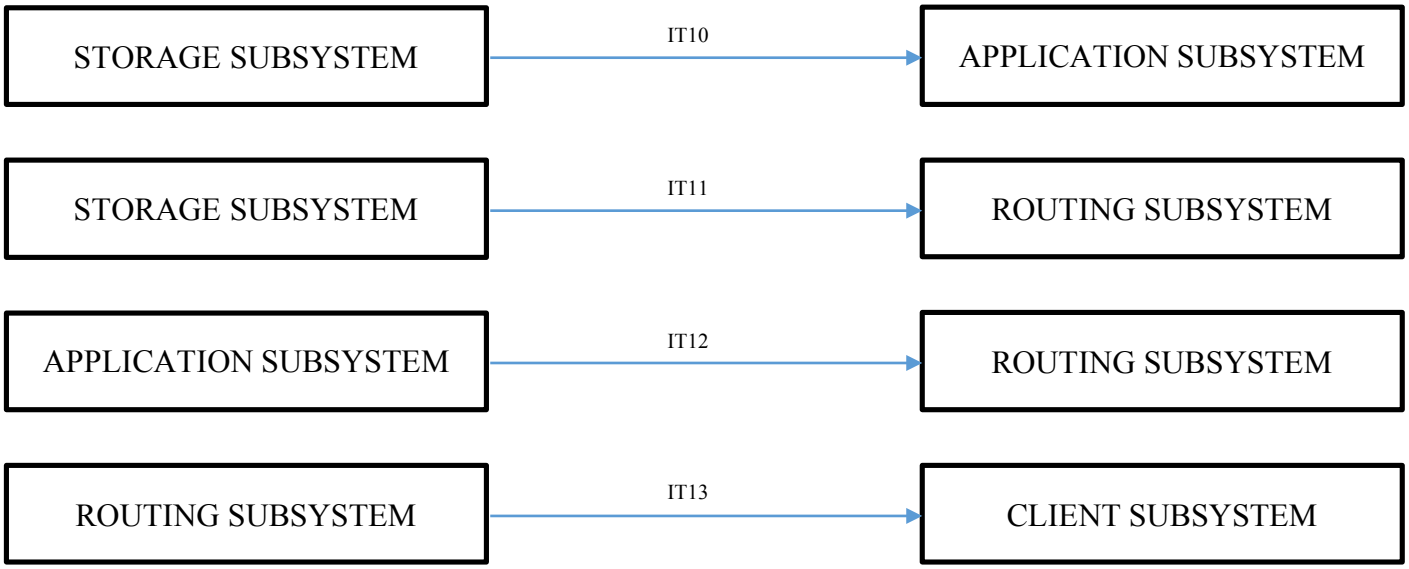
These subsystems do not need any integration at software level.

ROUTING SUBSYSTEM

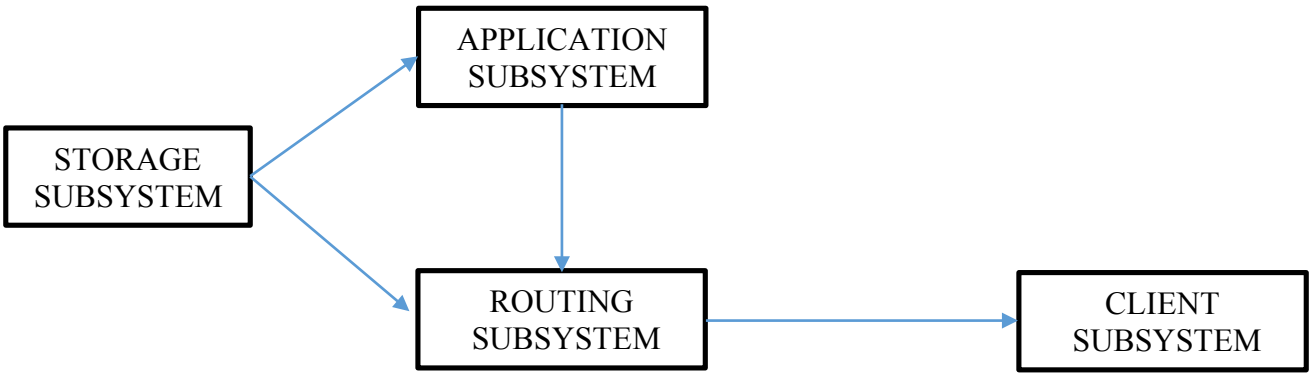


2.4.2 Subsystem Integrations Sequence

After all the software integrations, we integrate all the subsystems that they form in a unique one:



The entire system is shown below:



3. INDIVIDUAL STEPS AND TEST DESCRIPTION

Test Case Identifier	IT1
Test Item(s)	TccManager - HospitalManager
Input Specification	Create the typical input for TccManager
Output Specification	Check if the correct methods are called in HospitalManager
Environmental Needs	/
Purpose	This test case checks whether the calls made by the driver (HospitalManager) work as expected. For instance: • updateTravelTime()

Test Case Identifier	IT2
Test Item(s)	DccManager - HospitalManager
Input Specification	Create the typical input for DccManager
Output Specification	Check if the correct methods are called in HospitalManager
Environmental Needs	IT1 succeded
Purpose	This test case checks whether the calls made by the driver (HospitalManager) work as expected. For instance: • getQueueTimes()

Test Case Identifier	IT3
Test Item(s)	TccManager - TransportManager
Input Specification	Create the typical input for TccManager
Output Specification	Check if the correct methods are called in TransportManager
Environmental Needs	/
Purpose	This test case checks whether the calls made by the driver (TransportManager) work as expected. For instance: • getCoordinates()

Test Case Identifier	IT4
Test Item(s)	DccManager - TransportManager
Input Specification	Create the typical input for DccManager
Output Specification	Check if the correct methods are called in TransportManager
Environmental Needs	/
Purpose	This test case checks whether the calls made by the driver (TransportManager) work as expected. For instance: • getSchedule()

Test Case Identifier	IT5
Test Item(s)	TccManager - ParkingManager
Input Specification	Create the typical input for TccManager
Output Specification	Check if the correct methods are called in ParkingManager
Environmental Needs	/
Purpose	This test case checks whether the calls made by the driver (ParkingManager) work as expected. For instance: • getCoordinates()

Test Case Identifier	IT6
Test Item(s)	DccManager - ParkingManager
Input Specification	Create the typical input for DccManager
Output Specification	Check if the correct methods are called in ParkingManager
Environmental Needs	/
Purpose	This test case checks whether the calls made by the driver (ParkingManager) work as expected. For instance: • getParkingAvailability()

Test Case Identifier	IT7
Test Item(s)	TccManager - TrafficManager
Input Specification	Create the typical input for TccManager
Output Specification	Check if the correct methods are called in TrafficManager
Environmental Needs	/
Purpose	This test case checks whether the calls made by the driver (TrafficManager) work as expected. For instance: • limitTraffic()

Test Case Identifier	IT8
Test Item(s)	NotifierManager - TrafficManager
Input Specification	Create the typical input for NotifierManager
Output Specification	Check if the correct methods are called in TrafficManager
Environmental Needs	IT7 Succeded
Purpose	This test case checks whether the calls made by the driver (TrafficManager) work as expected. For instance: • notifyCitizens()

Test Case Identifier	IT9
Test Item(s)	Security – Dispatcher
Input Specification	Create the typical input for Security
Output Specification	Check if the correct methods are called in Dispatcher
Environmental Needs	/
Purpose	This test case checks whether the calls made by the driver (Dispatcher) work as expected. For instance: • authentication() • authorization()

Test Case Identifier	IT10
Test Item(s)	Storage Subsystem – Application Subsystem
Input Specification	Create the typical input for Storage Subsystem
Output Specification	Check if the correct methods are called in the Application Subsystem and the correct data are retrieved
Environmental Needs	Sample data must be present in the Database
Purpose	Verifies if the Application Subsystem can retrieve data from the Storage Subsystem

Test Case Identifier	IT11
Test Item(s)	Storage Subsystem – Routing Subsystem
Input Specification	Create the typical input for Storage Subsystem
Output Specification	Check if the correct methods are called in the Routing Subsystem
Environmental Needs	Sample data must be present in the Database
Purpose	Verifies if the Routing Subsystem can retrieve data from the Storage Subsystem

Test Case Identifier	IT12
Test Item(s)	Application Subsystem – Routing Subsystem
Input Specification	Create the typical input for Application Subsystem
Output Specification	Check if the correct methods are called in the Routing Subsystem
Environmental Needs	IT1, IT2, IT3, IT4, IT5, IT6, IT7, IT8, IT10 succeded
Purpose	Verifies if Application Subsystem can handle correctly Routing Subsystem methods calls and can properly manage different types of requests

Test Case Identifier	IT13
Test Item(s)	Routing Subsystem – Client Subsystem
Input Specification	Create the typical input for Routing Subsystem
Output Specification	Check if the correct methods are called in the Client Subsystem
Environmental Needs	IT9, IT11, IT12 succeded
Purpose	Verifies if the Client Subsystem can correctly interact with the Routing Subsystem

4. TOOLS AND TEST EQUIPMENT REQUIRED

The software tools used to automate the integration testing are the following:

- Jmeter (http://jmeter.apache.org): it is a tool which may be used to test the performance of the following subsystems:
  - Routing: simulate a heavy load of requests in order to check if the non-functional requirements on the maximum number of simultaneously connected users and on the response times are respected
  - Application: simulate a heavy load of requests on the REST APIs. This subsystem can be also overloaded by a stress test on the Routing subsystem. Tests on both subsystems are useful to identify bottlenecks
  - Storage: check the performance of critical database queries on the test database, in order to know which indexes to add and compare the performance of different equivalent query formulations
- JUnit (http://junit.org): it is the most used framework for unit testing in Java. The use of it in testing the single components is not covered by this document. However, it is important to state that JUnit is also usually used to perform integration testing together with other tools, like Mockito and Arquillian
- Mockito (http://site.mockito.org): it is an open-source test framework useful to generate mock objects, stubs and drivers
- Arquillian (http://arquillian.org/): it is a framework useful to perform integration testing. It provides a series of environment configuration and utilities that can be used to test the integration of the different components of SmartCityAdvisor.
- Manual testing: technique that is needed to simulate the input of the user (passenger or taxi driver) in the relative GUI in order to generate typical input data, useful to integrate the various components of SmartCityAdvisor

5. PROGRAM STUBS AND TEST DATA REQUIRED

Specifications of particular input data or component's stub/driver needed to perform the integration steps described in chapter 3 are included in the list below:

- Test database: In order to perform some test cases, sample user data should be inserted into the database (DataLayer component) and made available for testing. These test data includes a reduced set of instances of all the entities
- Lightweight API client: in order to test the REST APIs without the actual client applications, a simple API client which interacts with the Routing component by simple HTTP requests is needed. This driver needs to be scriptable in order for the tests to be automated
- Drivers: generally, if some components may not be available yet for the integration test phase, they will be replaced with appropriate drivers (that take the part of those software component) in order to test the others

6. HOURS OF WORK

We have worked on this document **for a total of 10 hours**.

7. REFERENCES

Here is a short list of other references for this document:

- Slides of the Software Engineering 2 course (from the Beep Platform)