Final Presentation

Software Engineering 2 Project

A. Pirovano A. Vetere

Politecnico di Milano

April 22, 2017

Outline

POLITECNICO MILANO 1863

Outline - Introduction

POLITECNICO MILANO 1863

Introduction

The project we have been assigned is called *myTaxiService* and it is a complex software system that should improve in several ways a preexisting taxi service in a town.
In order to rationalize, clarify, and put in **structured and standardized documents** all the relevant concepts and informations, we designed and delivered several documents such as the **RASD**, the **DD**, the **ITPD** and the **PPD**.
These slides will only present an overview of the concepts thoroughly described in the above mentioned documents.

POLITECNICO
MILANO 1863

Tools

We composed the documents we had to using some tools such as:

- **TexStudio:** to compile LaTeX document.
- **StarUML, Astah Professional:** to draw UML diagrams.
- **Alloy Analizer 4.2:** to checking model consistency.
- **Balsamiq Mockups 3.0:** to build mockups.
- **SourceTree:** to allow team collaboration.
- **GitHub:** for storing the project.
- **Skype:** for team collaboration

POLITECNICO
MILANO 1863

Outline - Requirement Analysis and Specification

1. Introduction
2. Requirement Analysis and Specification
   - Overview
   - UML Diagrams
   - Alloy
3. Design
   - Architectural Design
   - Algorithm Design
   - User Interface Design
4. Integration Test Plan
   - Overview
   - Integration Sequence Diagrams
5. Project Plan
   - Plan Contents
   - Cost Models
   - Tasks Scheduling

POLITECNICO MILANO 1863

Outline

POLITECNICO
MILANO 1863

Overview

The aim of *myTaxiService* is to improve taxi service usage and management in a large city by simplifying the access of passengers to the service and optimizing the management of taxi queues.
Overall, *myTaxiService* will lead to several benefits for taxi drivers, passengers and the government of the city.

Goals

This new service pretends to achieve various goals, such as:

- **G5:** A registered passenger can request a taxi ride when logged into the service.
- **G8:** A registered passenger can cancel a taxi request if he/she is viewing it.
- **G15:** A taxi driver can accept to give a ride to a registered passenger that requested one.
- **G17:** A taxi driver can notify the end of a ride.
- **G19:** A registered passenger can only take a ride from a taxi driver who is first in his current zone waiting queue.
- **G22:** Further services can be built on the top of the existing one through a set of given **API**s

POLITECNICO MILANO 1863

Actors - Taxi Drivers and Passengers

Below are listed the three main actors that will interact with the application once deployed:

- **Taxi Driver:** Owner of a vehicle who is given the permission to provide the service.
- **Non Registered Passenger:** A person that needs to move from a position to another one among the city and wants to use *myTaxiService* in order to do so, but has not registered yet to the service.
- **Registered Passenger:** A formerly non registered passenger that has registered to *myTaxiService*.

POLITECNICO MILANO 1863

Actors - Developers and Administrators

Under a certain point of view, we could consider even an additional user class which is the one of the **Developers** that will be using the project **API**s to develop further services based on the provided ones.
Even the **Administrators** could be considered as a further user class of the system, but that is a bit out of the scope of this first part of the presentation, which is more intended to the explain things around the core business of *myTaxiService*.

Product Perspective

Our *myTaxiService* is a **completely new product**, not based on previous ones.
It relies on **location data** received via **Internet** from each **taxi driver smartphone** application: all the involved smartphones already have a **GPS antenna** installed inside, that communicates their position to the service.
Being a partially **distributed application**, *myTaxiService* requires a fully operative **Internet** connection in order to work properly, both on server and client side: **no service is intended to be provided offline**.

POLITECNICO
MILANO 1863

More on Product Perspective

This software provides three separate **End User Interfaces**, one of which is accessible via **Web**, and a dedicated **Administrator** interface that is only accessible through a **LAN**.
All the data generated by this software are stored in a database, accordingly to current normative and laws about privacy and personal data management.
In addition, several **APIs** are provided in order to allow further improvements and expansions of the software: in this way additional services like **Taxi Sharing** could be built on the top of the existing ones.

Outline

POLITECNICO MILANO 1863

UML Diagrams

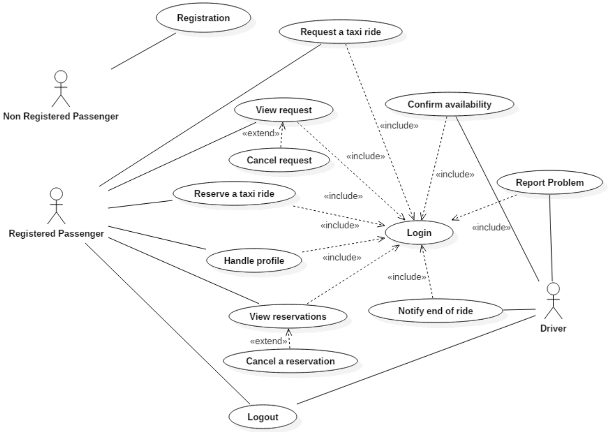
We provided a variety of UML diagrams, each type having a different purpose.

- **UML Use Case:** Shows the identified **use cases** in relation with the **involved actors**.
- **UML Sequence Diagram:** Indicates, for a given **use case**, the **interaction** between the **actors involved** and the **system**.
- **UML State chart:** Explains the **different states** in which:
  - *The Taxi driver (TD)* can be during the use of myTaxiService.
  - *The Passenger Application* can be during the Registered passenger (RP) navigation flow.
- **UML Class diagram:** Points out the different **software entities** involved in the application and the **relationships** between them.

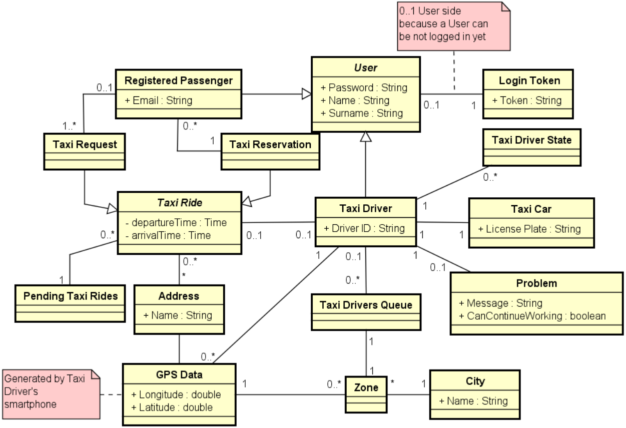
POLITECNICO MILANO 1863

UML Use Case Diagram

This is perhaps the most useful diagram that can be designed in the early phase of the development of a software project.



Furthermore we designed a class diagram for an early evaluation of the basic software components that consists in a sort of **Model** for *myTaxiService*.



Outline

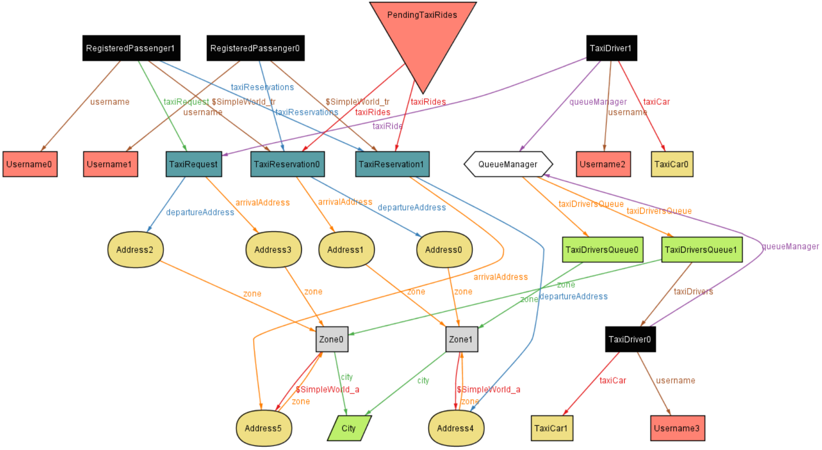
Alloy

Alongside the **UML Class Diagram** we built **Alloy Models** using the **Alloy** modeling language with the help of **Alloy Analyzer 4.2**.
The tool didn't find a proof of the inconsistency of our **Alloy Models**, and that along with the **Automatic Generation** (and **Manual Verification**) of interesting worlds, made us aware of the **Consistency** of those **Models** within a reasonable level of confidence.

POLITECNICO
MILANO 1863

Alloy Simple World

Here is an example of one among the simplest world we generated and double checked using both **Alloy Analyzer 4.2** and **manual checking**.



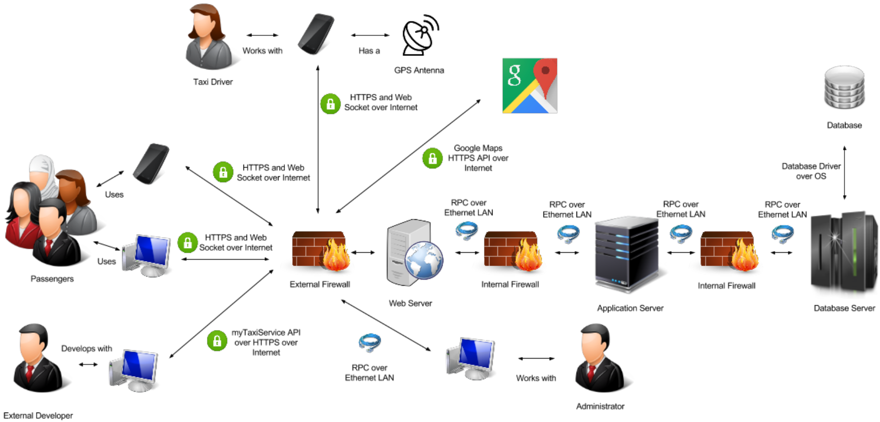
POLITECNICO MILANO 1863

Outline - Design

POLITECNICO MILANO 1863

Outline

Overview Diagram



High level components and their interaction

The system is composed of many **distributed** components: those will communicate with a **Client-Server** style and through **Point to Point** messaging system.

- The **Client-Server** style is used to give the many Clients connected to the Server the opportunity of sending different requests (e.g. a **Taxi Ride Reservation** or **Taxi Ride Request**).
- The **Point to Point** bidirectional communication channel is made necessary to enable the Server the delivery of various messages and requests to the Clients:
  - Generic notifications
  - Service messages
  - The request of serving a Taxi Ride (to a Taxi Driver)
  - The request of an updating GPS Data (to a Taxi Driver)

POLITECNICO MILANO 1863

Layers

The selected software architecture follows the principles of the **Model View Controller** architectural pattern, therefore three main software components have been identified and those are:

- The **Model**
- The **View**
- The **Controller**

Model, **View** and **Controller** are then mapped to three different relevant software layers.

Layers - View

This layer processes **Clients** commands, and converts them into requests addressed to the **Controller** layer. The **View** is connected to the **Controller** through a communication facility (e.g. The Internet). We imagined four different types of **View**, each one designed specifically to access *myTaxiService* system in a different way and by a specific kind of user:

- Passenger Web View
- Passenger Application View
- Taxi Driver Application View
- Administrator View

POLITECNICO
MILANO 1863

Layers - Controller

This second Layer is split in two families of components with specialized functionalities:

- **Networking Components Family:**
  - Groups the **Communication Components** that are involved in sending messages to the various Views, following the logic implements in the Business Components Family.
  - Dispatches a particular request to the relative **View**.
- **Business Components Family:** In this family are included all the software components that implement the system logic. Their role is:
  - Processing requests
  - Generating either **synchronous responses** (e.g registration or login procedure) or **asynchronous events** (e.g adding a **Taxi Ride** and sending a **Taxi Driver**).

POLITECNICO MILANO 1863

Layers - Model

The third and last Layer is the **Model**. It:

- Guarantees a high level interface to store and manage all the *myTaxiService* relevant data.
- Abstracts a **Relational Database** in a software component that is in direct connection with the **Controller**

It has the responsibility of **receiving** and **handling** all the model updating needs of the **Business Components**.

POLITECNICO
MILANO 1863

Tiers

The system is divided in **four** different tiers:

- **Clients:** The distributed clients of the application.
- **Web Server:** An outer server that dynamically generates web pages, receives requests, dispatches messages and contacts other servers.
- **Application Server:** The most important Tier of the system. Here are done all the logics and calculations that constitute the core part of *myTaxiService*.
- **Database Server:** In this Tier it is hosted the Database that allows data persistence.

Component View

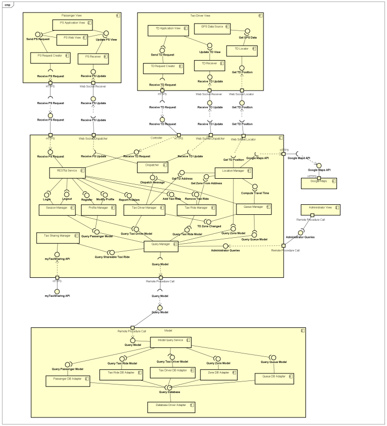
Several components has been designed to provide all the functionalities needed for *myTaxiService* to work. **Five** mayor subsystems have been identified:

- Passenger View, Taxi Driver View, Administrator View
- Controller
- Model

POLITECNICO
MILANO 1863

Component View - UML Component Diagram

This diagram maps system **features into different software components**, and show **how these components interact** in order to **deliver the required functionalities**. It helps showing **Layers organization** and the **MVC implementation**.



Deployment View

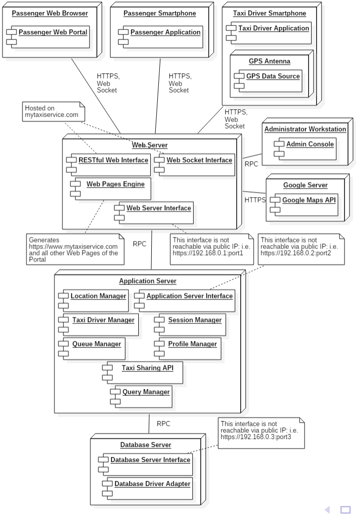
The best way found to **deploy** the software components identified, is to consider **7 different nodes** (8 if considering the Google Server contacted to use Google Maps API):

- Passenger Web Browser, Passenger Smartphone, Taxi Driver Smartphone, Administrator Workstation
- Web Server
- Application Server
- Database Server

POLITECNICO MILANO 1863

Deployment View - UML Deployment Diagram

The following diagram shows how **software components** are mapped into the **physical system**.



Runtime View

In this subsection are proposed some of the most meaningful **UML Sequence Diagrams** with respect to show how software components interacts in order to deliver a specific functionality. The chosen functionalities are:

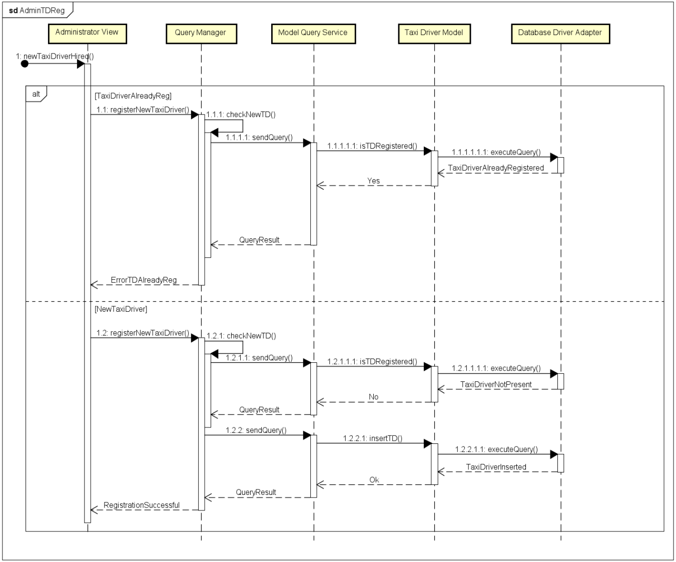
- Taxi Driver Registration (done by an Administrator)
- Handling of a Taxi Reservation (done by the Taxi Ride Manager)

There are other functionalities whose **UML Sequence Diagram** is not reported here for space and time constraints:

- Passenger Login
- Passenger Registration
- Queue Management
- Taxi Ride Request Handling
- Taxi Driver Report Problem
- Taxi Driver Position Update

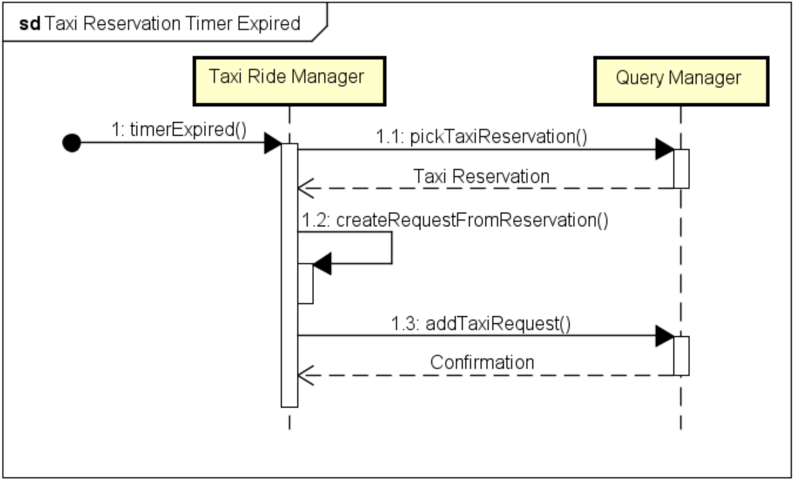
POLITECNICO
MILANO 1863

Taxi Driver Registration



POLITECNICO MILANO 1863

Handling of a Taxi Reservation



Selected architectural styles and patterns - MVC

Several architectural styles and patterns were chosen in order to build *myTaxiService* as a modern software. The main pattern that was recursively adopted is the **Model View Controller** architectural pattern:

- **System Level:** All the clients that use *myTaxiService* (i.e. the Passengers, the Taxi Drivers, and the Administrator) are seen as Views, that following the Cocoa MVC pattern, are connected to a Controller, the Web Server, that through the Application Server is itself connected to the Model hosted on the Database Server.
- **Client Level**
- **Server Level**
  - Web Server
  - Application Server
  - Database Server

POLITECNICO
MILANO 1863

Selected architectural styles and patterns - Client-Server

The **Client-Server** style is used for all the requests done by the various clients connected to the Web Server of *myTaxiService*. The **Taxi Driver Application** and the **Passenger Application** can use a standardized **Client-Server** protocol via **HTTPS** that follows the principle of a **RESTful Service**. The **Administrator** application it's connected via **RPC** to the **Web Server** and can perform more critical requests, like the registration of a new **Taxi Driver** into the system. It is required that the **Administrator** application opens a **RPC** connection to the **Web Server** to start the communication.

Selected architectural styles and patterns - Point to Point

A **Point to Point** bidirectional messaging system is established between the **Clients** and the **Web Server** at the *boot* of the client application. The client should explicitly request a connection to the server that is listening for clients' connections. It is the connection over **Web Socket** protocol that allows the **Web Server** to send asynchronous messages and requests to which the client can respond using the same channel. The main reasons why this protocol is used are sending a **Taxi Ride** proposal to a given **Taxi Driver**, that can either accept or deny the proposal, and allowing the server to ask the **Taxi Driver** an updated geolocation data.

POLITECNICO
MILANO 1863

Selected architectural styles and patterns - Conclusion

The **Client-Server** style and **Point to Point** bidirectional messaging system are used to implement properly the **MVC** pattern in this three **Layers**, four **Tiers** system.

POLITECNICO MILANO 1863

Other design decisions - HTTPS and Web Socket

Several technologies have been chosen in order to best fit the needs of the system to be. Not all the required functionalities of *myTaxiService* are already mapped onto specific products because in those cases the choice done would matter less. But for the cases in which a technology has already been proposed, it is because a clear design decision was mandatory. As for the communication protocols between clients (excluded the Administrator client) and the server have been chosen:

- **HTTPS:** The secure version of **HTTP** was a mandatory choice as security and privacy concerns are of major importance nowadays.
- **Web Socket:** This innovative socket technology has been chosen although is relatively new because it implements a full duplex socket communication channel using web technology and therefore using the port 80, which is in almost every case not blocked by any firewall.

POLITECNICO
MILANO 1863

Other design decisions - Internet and Firewalls

For what concerns the network reachability has been chosen to make discoverable only the Web Server assigning it a public IP. All the other servers in *myTaxiService* system are reachable only within the enterprise network. Between the **Web Server** and the external network is installed a firewall that controls all the incoming connections. In particular it must accept only incoming **HTTPS** connections, **Web Socket** connections and **RPC** connections. A firewall is also used to protected the **Database Server** from the **Application Server** in the unlikely case that the **Application Server** is attacked through the **Web Server** or the **Application Server** for some reasons stops working correctly and start behaving in a way that will damage the application **Model**.

POLITECNICO MILANO 1863

Outline

POLITECNICO MILANO 1863

Queue Management

What will follow are slides containing algorithms (in form of Java methods, without loss of generality) that explain how the association of a Taxi Ride to an Available Taxi Driver is managed, and how that specific Taxi Driver is found.

Queue Management - Manage Taxi Ride

```java
private void manageTaxiRide(TaxiRide taxiRide) {
   // This method call searches for a suitable Taxi Driver
   TaxiDriver taxiDriver = getTaxiDriver(taxiRide.getStartingZone());

   // This means that no suitable Taxi Driver has been found
   if (taxiDriver == null) {
      // Need to reinsert the Taxi Ride in the pending list
      queryManager.insertTaxiRide(taxiRide);
      return;
   }

   // Ask Taxi Driver to accept the Taxi Ride Request
   boolean accepted = dispatcher.notifyTaxiDriver(taxiDriver, taxiRide, RESPONSE_TIMEOUT_SEC);

   if (!accepted) {
      // Need To reinsert the Taxi Ride in the pending list
      queryManager.insertTaxiRide(taxiRide);
      // Reinsert the Taxi Driver in the Queue, so that he/she will be in
      // the last position of that Queue
      queryManager.enqueueTaxiDriver(taxiDriver);
   } else // Request accepted
   {
      // Add the taxiRide - taxiDriver match to the model
      queryManager.addRideDriverMatch(taxiRide, taxiDriver);
      // Set his/her status to WORKING
      queryManager.updateTaxiDriverStatus(taxiDriver, TaxiDriverStatus.WORKING);
      // Compute the Taxi Driver ETA
      int travelTime = locationManager.computeTravelTime(taxiDriver.getCurrentAddress(),
            taxiRide.getStartAddress());
      // Notify the TD ETA to the interested Passenger
      dispatcher.notifyPassenger(taxiRide.getPassenger(), taxiRide, travelTime);
   }
}
```

POLITECNICO
MILANO 1863

Queue Management - Get Taxi Driver

```java
private TaxiDriver getTaxiDriver(Zone taxiRideStartingZone) {
   // Search for an available Taxi Driver in the Taxi Ride Starting Zone,
   // If found removes it from the queue and returns it;
   // BLOCKING METHOD, NULL returned after TD_SEARCH_TIMEOUT
   TaxiDriver taxiDriver = queryManager.dequeueTaxiDriver(taxiRideStartingZone, TD_SEARCH_TIMEOUT_SEC);

   // Taxi Driver found in Taxi Ride Starting Zone
   if (taxiDriver != null) {
      return taxiDriver;
   }

   // Taxi Driver not found in Taxi Ride Starting Zone
   // Search in near zones
   Set<Zone> nearZones = taxiRideStartingZone.getAdjacentZones();
   for (Zone nearZone : nearZones) {
      // Search for an available Taxi Driver in the selected Zone -
      // If found removes it from the queue and returns it;
      // BLOCKING METHOD, NULL returned after TD_SEARCH_TIMEOUT
      taxiDriver = queryManager.dequeueTaxiDriver(nearZone, TD_SEARCH_TIMEOUT_SEC);
      // Taxi driver found
      if (taxiDriver != null) {
         return taxiDriver;
      }
   }

   // No suitable Taxi Driver can be found
   return null;
}
```

POLITECNICO
MILANO 1863

Geolocation - A First Approach

Another interesting design choice that has been made concerns the way in which the **GPS coordinates** obtained from a given **Taxi Driver** are mapped into a specific **Zone**. It could have been possible, to do such a thing:

1. Obtain **GPS Data** via **Web Socket** from the selected **Taxi Driver**.
2. Calculate the nearest **Address** of the given **GPS Data** using **Google Maps HTTPS API**.
3. Query the **Model** to obtain the **Zone** to which belongs the given **Address**.

But this solution requires to have a precomputed data structure that associates every **Address** in the **City** to the corresponding **Zone** (that could have been a relational table with as many rows inside as **Addresses** in the **City**, each address associated with the corresponding **Zone**), that is heavy to manage and maintain, although if correctly installed and filled, it gives for certain good performances.

Geolocation - The Chosen Approach

A less heavy weight solution has been found: this solution expects every **Zone** of the **City** to be divided in several convex **Polygons**, for instance **Triangles**, that have interesting properties for our application. In *myTaxiService*, **Zones** of regular shape are intended to be designed, and therefore the number of **Triangles** in which a **Zone** should be decomposed is very limited. So, such a flow is followed:

1. Obtain **GPS Data** via **Web Socket** from the selected **Taxi Driver**
2. For each **Zone**, check if the the **Point** that the **Longitude** and **Latitude** from **GPS Data** identify is contained inside any **Triangle** in which the **Zone** is divided. If it is so, then the **Zone** is found. If that's not the case, then another **Zone** could contain the given Point. If no **Zone** contains the Point, then we can assume that the **Point** refers to **GPS Data** that identify a geographical point outside of the **City**.

The computation of the **Point in Triangle** test is simple and efficient (e.g. using barycentric coordinates).

POLITECNICO MILANO 1863

Outline

POLITECNICO
MILANO 1863

GUI Design

In this section we provide the **most important and meaningful mockups** for every class of screens we have designed. In particular we identified **three classes** of graphical user interfaces:

- **Passenger Mockups:** both Web based and Mobile Application based.
- **Taxi Driver Mockups:** only Mobile Application based.
- **Administrator Mockups:** only Desktop Application based.

POLITECNICO
MILANO 1863

Passenger Mockups

In the following slides are shown **sequence of graphical states** that the application has to **render** in order to **create** and **handle** a Taxi Ride. Once logged in, the **Registered Passenger** will be redirected in his/her personal **Home page**, where he/she will be able to request or reserve a **Taxi Ride** and manage his/her personal profile.

Passenger Mockups - Overview

Here the **Registered Passenger** can perform different actions:

- Request a Ride
- Reserve a Ride
- Logout
- Modify his profile
- Throw away a selected Ride

By clicking the **"Request a ride"** and **"Reserve a ride"** buttons the user is allowed to perform the relative actions.

Once requested a ride, the "Request a ride" button is disabled, in order to prevent multiple useless requests.

POLITECNICO
MILANO 1863

Passenger Mockups - Personal Homepage

This is an example of the **Registered Passenger**'s **Home page**. The screen is divided in two parts:

- The **left** one contains **Taxi Requests**
- The **right** one contains **Taxi Reservations**

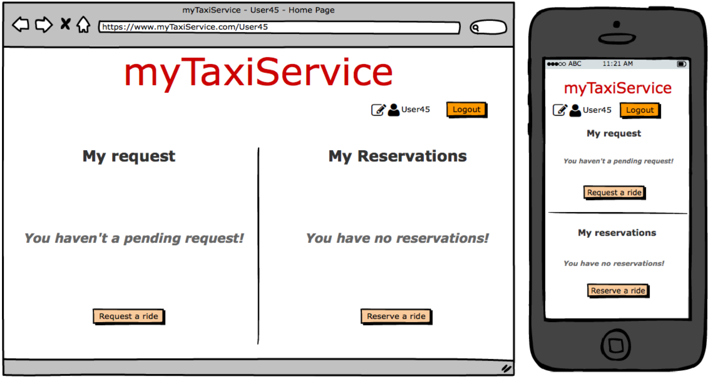


Figure: Empty RP home page.

Passenger Mockups - Request a Ride

Figure: Taxi request.

Passenger Mockups - Reserve a Ride

Figure: Taxi reservation.

Passenger Mockups - Personal Homepage with **Taxi Rides**

Below is shown a common **state** with one **Taxi Request** active and one **Taxi Reservation** booked. Through the trash icon the user is allowed to cancel a selected **Taxi Ride**.

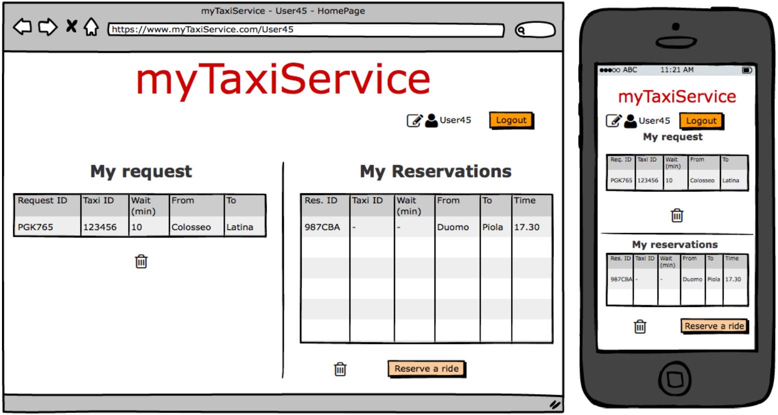


Figure: Populated RP home page.

Taxi Driver Mockups

In the following slides are shown **the sequence of the graphical states** that the application has to **render** in order to make the **Taxi Driver** able to **handle** a **Taxi Ride**.

Taxi Driver Mockups - Overview

The **Taxi Driver** personal screen is divided into **two** sections:

- Pending Rides Space
- Serving Ride Space

When the system sends a **Taxi Ride** to a specific **Taxi Driver**, it is placed in the **Pending Requests** space. Here the **Taxi Rider** can accept or deny it. If it is accepted the **Taxi Request** is moved from the previous to the second space. Once the ride is finished, the **Taxi Driver** has to push the **Notify End Of Ride** button, in order to notify the system the ending of the given ride. Through the **Report Problem** button, the **Taxi Driver** has the possibility, *in every moment of his/her working time*, to signal an accident or a problem. In order to better handle the problem, the **Taxi Driver** is asked to **signal** if the problem is solvable or not.

- If it is solvable, then the system **does not assign** a new **Taxi Driver**.
- If it is not solvable, then the system **assigns** the incomplete ride to the next **Taxi Driver** in the **Zone Queue**.

POLITECNICO
MILANO 1863

Taxi Driver Mockups - No Requests

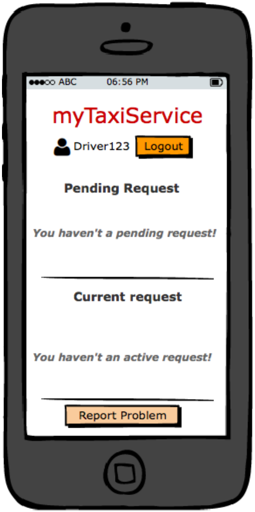


Figure: Taxi Driver Homepage without pending **Taxi Requests**.

POLITECNICO MILANO 1863

Taxi Driver Mockups - A Pending Request

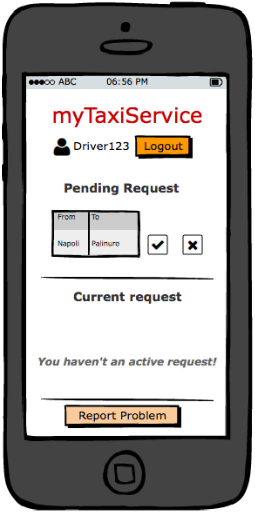


Figure: Taxi Driver Homepage with a **PENDING Taxi Request**.

Taxi Driver Mockups - A Active Request

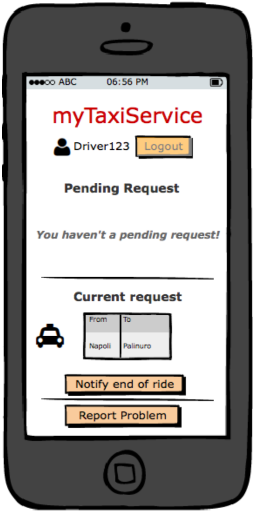


Figure: Taxi Driver Homepage with an **ACTIVE Taxi Request**.

Administrator Mockups

The system architecture does not admit the usage of a textual interface (e.g. a CLI). For this reason we decided to **provide a thin desktop interface to the Administrator**. Thus, the Administrator can perform his actions using an intuitive, fast, and lightweight GUI.

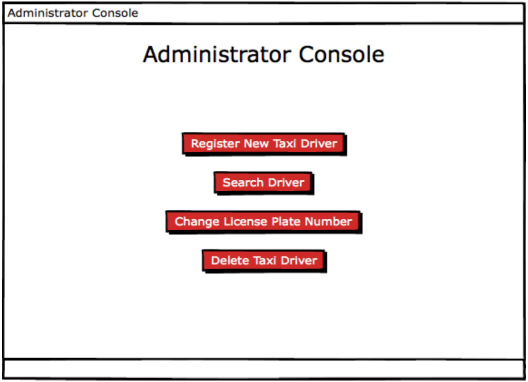


Figure: Example of an Administrator screen.

Outline - Integration Test Plan

POLITECNICO
MILANO 1863

Outline

POLITECNICO MILANO 1863

Scope and Approach

This project phase is highly based on the **Design** one.

We will clearly state the order in which the software components identified in the **Component View** of the **Design** part have to be integrated one with each other in order to guarantee a well tested final software.

The **bottom-up integration testing approach** has been chosen, because for a medium sized project like *myTaxiService*, it is best to proceed step by step in a careful yet coherent integration strategy.

Entry Criteria

Before starting the integration testing of any software component that has been designed for *myTaxiService* system, few points have to be underlined:

- The **internal functions** of the considered component must be **unit tested** using an appropriate framework.
- We suppose that **Google Maps API** are well tested by **Google** and thus we can use them without testing any further.
- We assume that the **GPS Data Source** module in the **Taxi Driver View** uses the **GPS Drivers** of the underlying operating system that are already tested, and the same is assumed for the **Database Driver Adapter** in the **Model** referring to Database Drivers.

POLITECNICO
MILANO 1863

Integration Testing Strategy

We considered **Model**, **Controller** and **Views** as **Subsystems**.

- **Model:** In order to test its relevant functionalities, the tester has to use the related part of the **Controller** to interact with the **Model**. In this way, the **Model** will be completely integration tested by exploring all the possible actions that the **Controller** can do on it.
- **Controller:** The test sequence adopts the same strategy used to test the **Model**. The only difference from a higher point of view, is that in this case the **Controller** is tested on the basis of the already tested **Model** using **Views** actions.
- **Views:** The **Taxi Driver View**, **Passenger View** and **Administrator View** are tested above the already tested **Controller** and **Model** using **UI** testing automators or manual testing.

POLITECNICO MILANO 1863

Outline

POLITECNICO MILANO 1863

Convention adopted - Blocks

- **Yellow:** This block is not dependent on any lower level component in *myTaxiService* and therefore it is integrated as a starting point in the current diagram.
- **Blue:** This block is going to be fully integrated on the top of its parents.
- **Green:** This block is not going to be fully integrated within the current diagram but needs further integration testing in subsequent diagrams.
- **Red:** This block represents a stub component, that replaces the real component mocking its functionalities.

Convention adopted - Arrows

We use **arrows** to link different blocks in order to explain the **precedence** between software components integration.

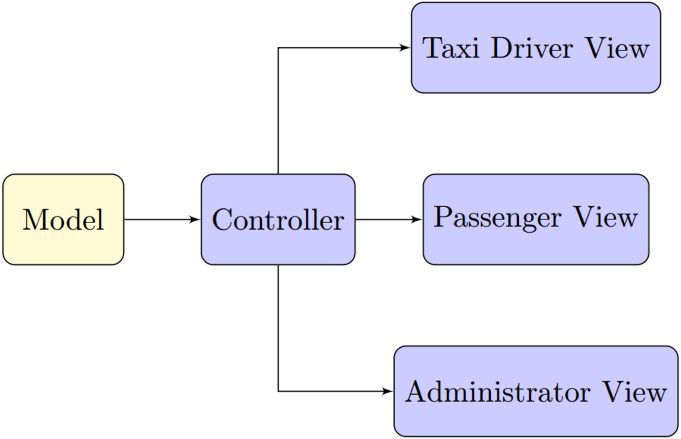
Every **arrow** has the following meaning:

- It helps the tester to follow the right order in the whole integration process.
- It starts from a block and ends into another block. The block from which it starts is called parent and the other one child. In particular **it means that the child block can be integrated only if its parents are already integrated**.

Moreover if a block is pointed by several arrows, its integration process can begin only when all the parent blocks are integrated.

POLITECNICO
MILANO 1863

Subsystems Integration Sequence



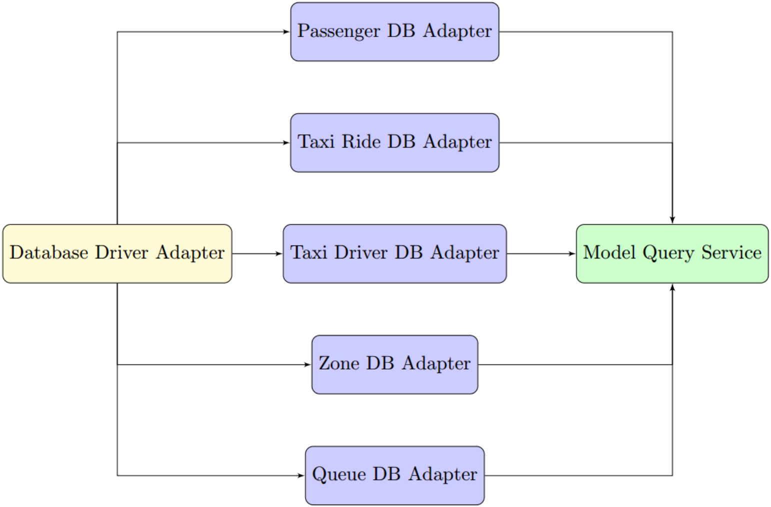
Software Components Integration Sequence

We provided six **Software Components Integration Sequence Diagrams**:

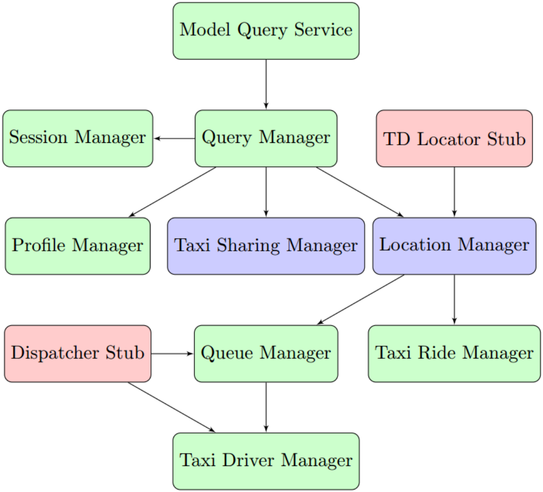
- **Model Integration Sequence**
- **Controller Business Components Integration Sequence**
- **Controller Networking Components Integration Sequence**
- **Passenger View Integration Sequence**
- **Taxi Driver View Integration Sequence**
- **Administrator View Integration Sequence**

We are going to show only the first two diagrams for sake of brevity.

Model Integration Sequence



Controller Business Logic Integration Sequence



POLITECNICO
MILANO 1863

Program Stubs And Data Required

In order to perform the proposed testing strategy, we need few **Stubs** in order to make the not yet integrated components work, because we want to respect the **bottom-up strategy**.

To better catch the need for introducing **Stubs**, an example of a specific Stub usage is proposed below.
In order to integrate the **Location Manager** in I8T4 we need a component that mocks **TD Locator** functionalities in a predefined way. Given the fact that the **TD Locator** is a component of the **TD View**, we have decided to introduce its **Stub**.
The real **TD Locator** will be integrated when the integration procedure arrives to **TD View**.

In conclusion there is the need for some sample data to be in the **Database** and some sample **GPS data** are needed.

Outline - Project Plan

POLITECNICO MILANO 1863

Outline

Plan Contents

The **Project Plan** consists in tables, **Gantt diagrams**, charts and natural language descriptions of the planning, scheduling and management of *myTaxiService* development.

In order to estimate the project effort, we followed the assumption that the **dimension of the software** can be characterized by correlating **the kind of functionalities offered** with **the source lines of code (SLOC) of the software itself**

Outline

POLITECNICO MILANO 1863

Function Points Approach

The **Function Points approach**, defined in 1975 by Allan Albrecht:

- Consists in a technique to assess the effort needed to design and develop custom software applications.
- Correlates the kind of functionalities offered with the source lines of code of the software itself.

This technique consists in combining the following program characteristics to obtain a final result:

- **Internal Logic Files:** Data used and managed by the application
- **External Logic Files:** Data used by the application but generated and maintained by other applications.
- **External Input:** Elementary operations to elaborate data coming from the external environment.
- **External Output:** Elementary operations that generate data for the external environment, and they usually include the elaboration of data
- **External Inquiry:** Elementary operations that involve input and output, without significant elaboration of data.

POLITECNICO
MILANO 1863

Function Points Summary

All the calculated FP_i sums up to FP, which is the total Function Points value:

$$
\begin{aligned}
FP &= FP_{ILF} + FP_{ELF} + FP_{EI} + FP_{EO} + FP_{EIQ} \\
&= 73 + 17 + 30 + 43 + 6 \\
&= 169
\end{aligned}
$$

The total FP value is then multiplied by a constant factor $k_{i,j}$ that depends on the programming language i used to develop the software and the company gearing ratio j.
The gearing ratio is the level of a company's debt related to its equity capital, usually expressed in percentage form.
This final calculation gives us the number of SLOC n_{SLOC} estimated for *myTaxiService*:

$$
\begin{aligned}
n_{SLOC} &= FP \cdot k_{Java,Avg} \\
&= 169 \cdot 53 \\
&= 8957 \text{ SLOC}
\end{aligned}
$$

COCOMO II - Parameters

COCOMO II - Constructive Cost Model

Software Size

Sizing Method	Function Points		
Unadjusted Function Points	169	Language	Java

Software Scale Drivers

Precedentedness	Nominal	Architecture / Risk Resolution	Nominal	Process Maturity	High
Development Flexibility	High	Team Cohesion	Very High		

Software Cost Drivers

Product		**Personnel**		**Platform**	
Required Software Reliability	Nominal	Analyst Capability	Nominal	Time Constraint	Nominal
Data Base Size	Low	Programmer Capability	High	Storage Constraint	Nominal
Product Complexity	High	Personnel Continuity	High	Platform Volatility	Low
Developed for Reusability	High	Application Experience	Nominal	**Project**	
Documentation Match to Lifecycle Needs	Nominal	Platform Experience	Nominal	Use of Software Tools	High
		Language and Toolset Experience	High	Multisite Development	Nominal
				Required Development Schedule	Nominal

Maintenance Off

Software Labor Rates

Cost per Person-Month (Dollars) 1500

Calculate

POLITECNICO MILANO 1863

COCOMO II - Results

Results

Software Development (Elaboration and Construction)

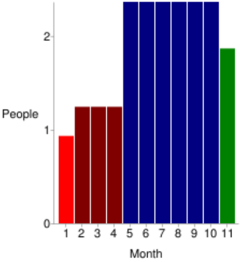
Effort = 18.8 Person-months
Schedule = 9.7 Months
Cost = $28154

Total Equivalent Size = 8957 SLOC

Acquisition Phase Distribution

Phase	Effort (Person-months)	Schedule (Months)	Average Staff	Cost (Dollars)
Inception	1.1	1.2	0.9	$1689
Elaboration	4.5	3.6	1.2	$6757
Construction	14.3	6.0	2.4	$21397
Transition	2.3	1.2	1.9	$3379

Staffing Profile



Software Effort Distribution for RUP/MBASE (Person-Months)

Phase/Activity	Inception	Elaboration	Construction	Transition
Management	0.2	0.5	1.4	0.3
Environment/CM	0.1	0.4	0.7	0.1
Requirements	0.4	0.8	1.1	0.1
Design	0.2	1.6	2.3	0.1
Implementation	0.1	0.6	4.9	0.4
Assessment	0.1	0.5	3.4	0.5
Deployment	0.0	0.1	0.4	0.7

Your output file is http://csse.usc.edu/tools/data/COCOMO_January_23_2016_06_29_45_485566.txt

Created by Ray Madachy at the Naval Postgraduate School. For more information contact him at rjmadach@nps.edu

POLITECNICO
MILANO 1863

Outline

POLITECNICO
MILANO 1863

Tasks

Task	Description	Completed?
T1a	RASD - Writing	Yes
T1b	RASD - Presentation	Yes
T2a	DD - Writing	Yes
T2b	DD - Presentation	Yes
T3a	ITPD - Writing	Yes
T3b	ITPD - Presentation	Yes
T4a	PPD - Writing	Yes
T4b	Final Presentation	Yes
T5	Implementation	No
T6	Unit Testing	No
T7	Integration Testing	No
T8	System Testing	No
T9	User Acceptance - Alpha Testing	No
T10	User Acceptance - Beta Testing	No
T11	Release To Market	No

POLITECNICO
MILANO 1863

Gantt Diagram