# Software Engineering 2: myTaxiService

Politecnico di Milano

Chitti Eleonora, De Nicolao Pietro, Delbono Alex

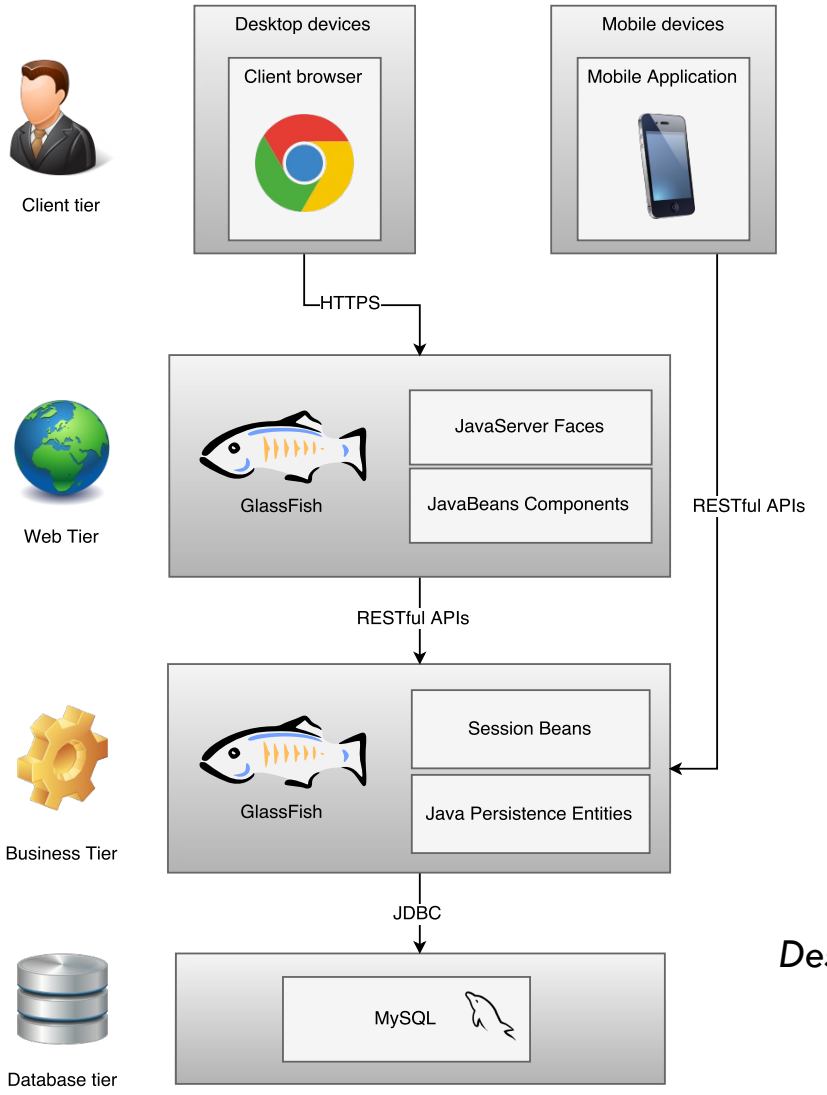
# Design Document

The Design Document for the myTaxiService application is made to provide a functional description of the main architectural components, their interfaces and their interactions, together with the algorithms to implement and the User Interface design.

# Architectural Design: component view

APPLICATION SERVER

- implemented in the business logic tier using Java EE
- runs on GlassFish Server
- access to the DBMS is completely wrapped by the Java Persistence API (JPA).



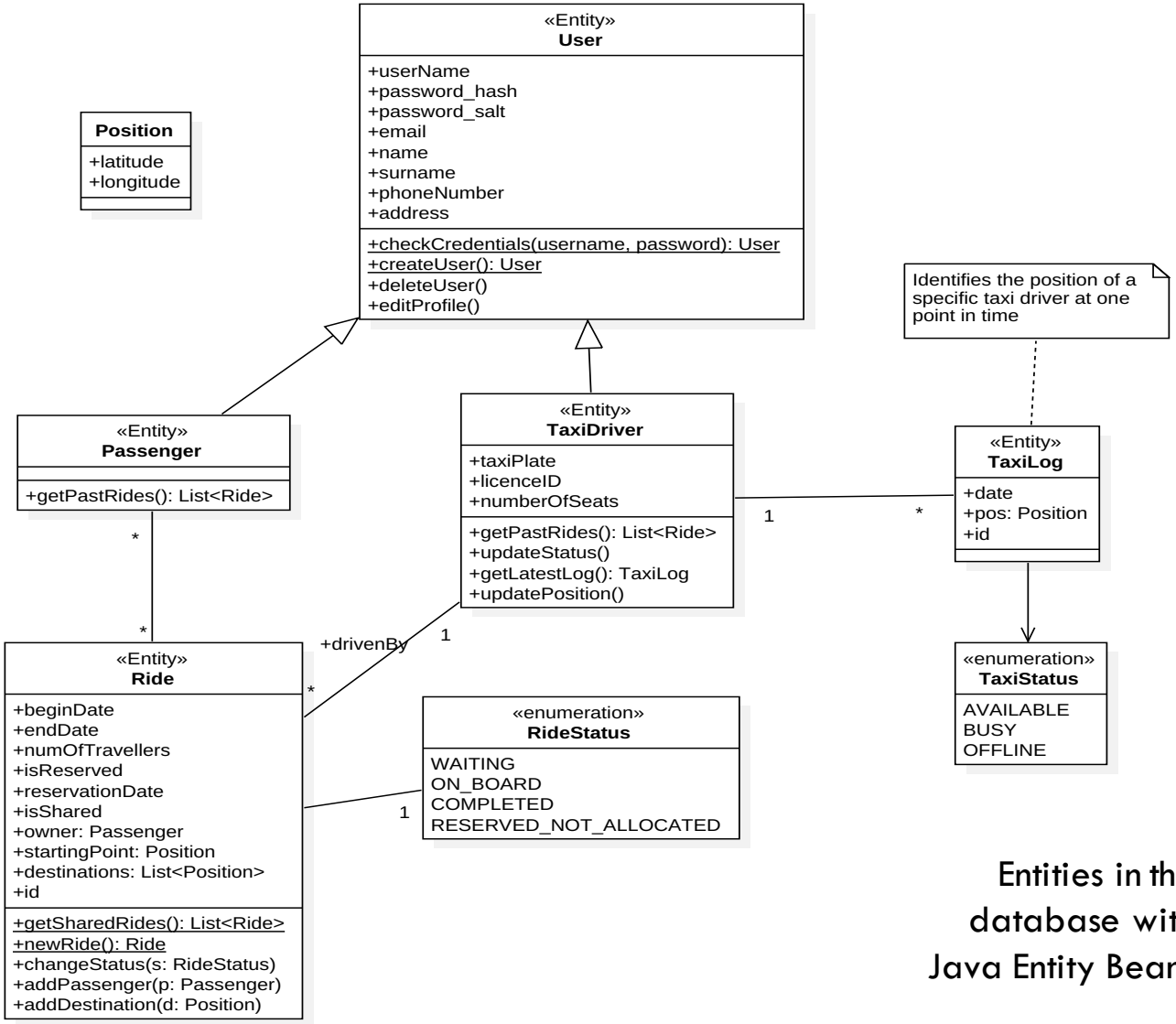

*Description of tiers, with JEE components*

# Architectural Design: high level components

- **Database:**
  - responsible for the data storage and retrieval
  - does not implement any application logic
  - must guarantee ACID properties
- **Application Server:**
  - contains all the application logic of the system
  - the policies, the algorithms and the computation are performed here
  - service-oriented interface
- **Server side plug-ins:**
  - Ride sharing
  - Ride reservation
- **Web server:**
  - does not contain any application logic
- **Mobile application:**
  - communicates directly with the application server
- **User's browser:**
  - not under the control of the system
  - it accesses the web server

# Component view: application server

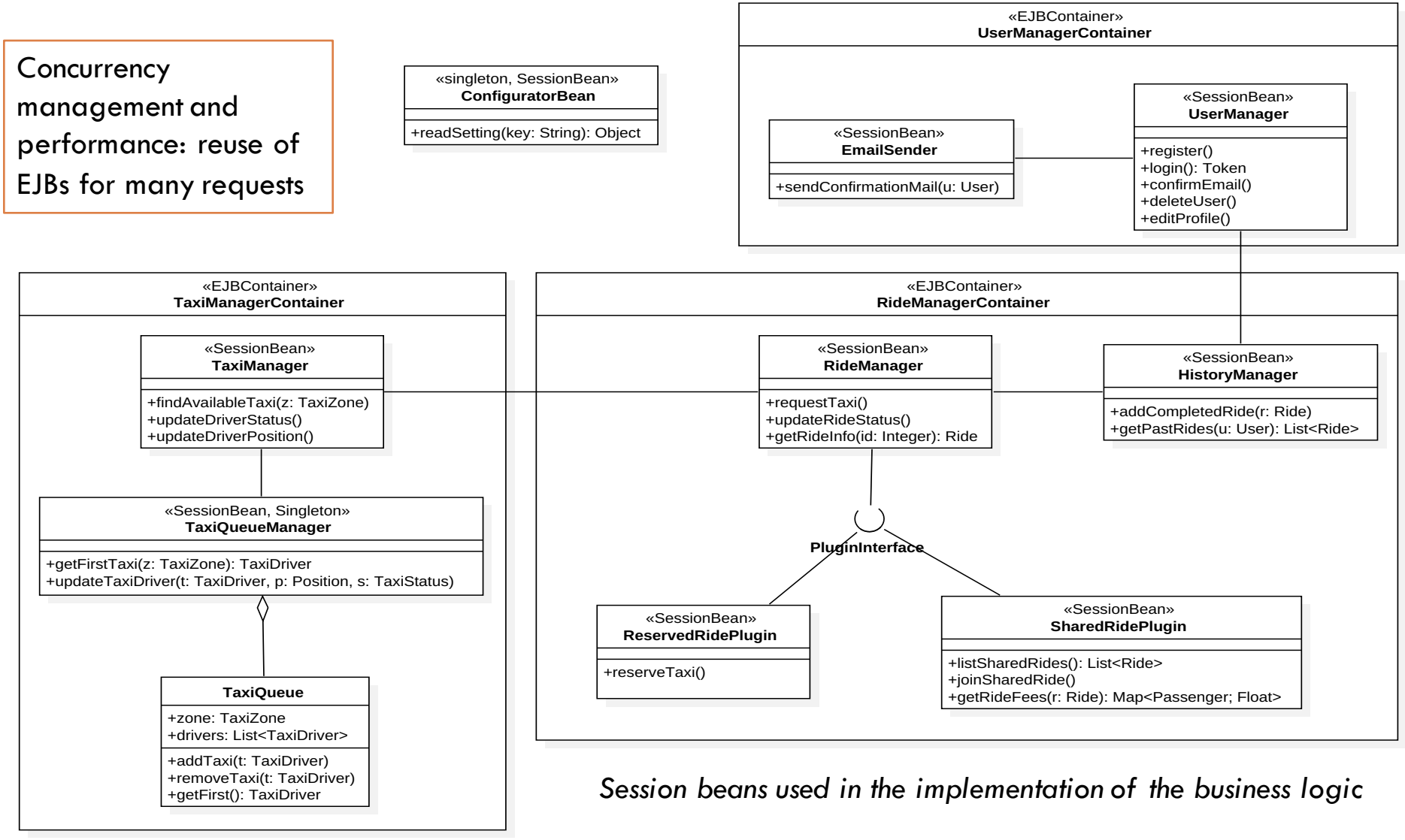
- Custom-built stateless Enterprise JavaBeans (EJB)
- The state of the users is stored in the DB




Entities in the database with Java Entity Beans

# Component view: application server

Concurrency management and performance: reuse of EJBs for many requests




*Session beans used in the implementation of the business logic*

# RESTful interface

Communication between front-ends and back-end:
**REST API over HTTPS.**

- Style: **Service-Oriented Architecture**
- Implemented with **JAX-RS**
- **XML** for data representation
- Each session bean exposes some methods
- Plugins can extend the API interface
- Documented in detail in the **DD**

# Architectural design: selected architectural styles and patterns

- **Model-View-Controller** for clients (web + mobile)
- **Client&Server:**

  at multiple levels
  - the application server (client) queries the DB (server)
  - the web + mobile app (clients) communicate with the application server
  - the user's browser (client) communicates with the web server
- **Service-Oriented Architecture:**

  used for the communication between the application server and the front-ends.
- **Plug-ins:**

  used to add functionalities to the application server
- **Thin Client:**

  used for the interaction between user's machine and the system, all the data and the application logic are on the side of the system.
- **Distributed presentation;** the presentation layer is split among two tiers:
  - web server
  - user's browser

# Algorithm Design

- ***Taxi queue management***
  - Input events
    - new request incomes
    - taxi driver accepts a request
    - taxi driver refuses a request
    - taxi driver changes zone
    - taxi driver changes status
- ***Taxi waiting calculation***
  - the delay is computed using the Google Maps Directions API, called as a service from the business logic layer
    - In the table are shown API call paramethers

| | |
|---|---|
| URL | http://maps.googleapis.com/maps/api/directions/xml |
| origin | GPS coordinates of the taxi driver |
| destination | GPS coordinates of the passenger |
| key | API key |
| mode | driving |
| departure_time | now |
| traffic_model | pessimistc |

*Parameters of the Google Maps API call*

# Algorithm Design

- ***Taxi sharing matching***
- ***Taxi fee splitting:***

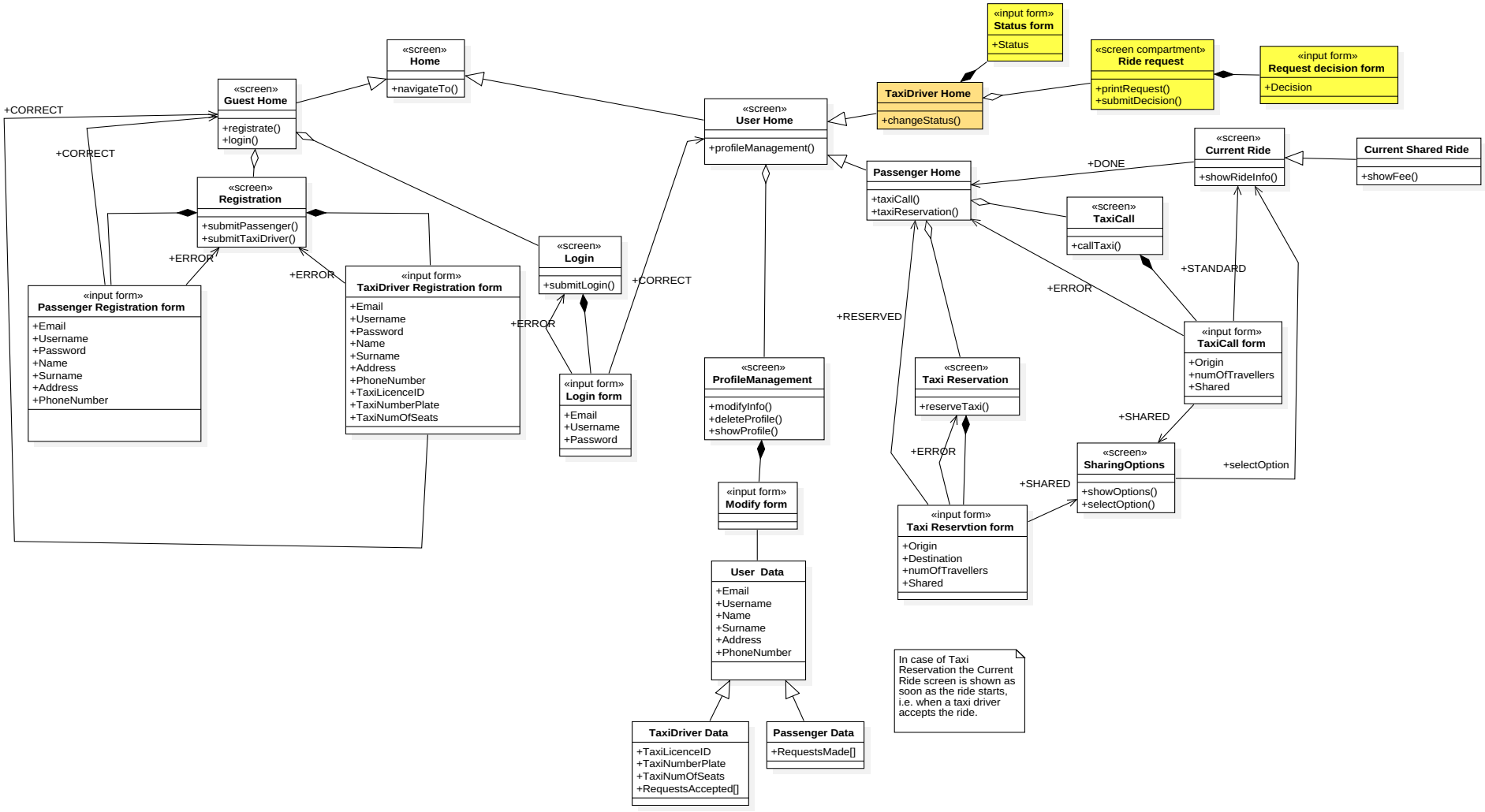
The input data are:

- P : list of passengers;
- $T_i$ : list of travellers associated with passenger i;
- d(t): distance traveled by traveller t.

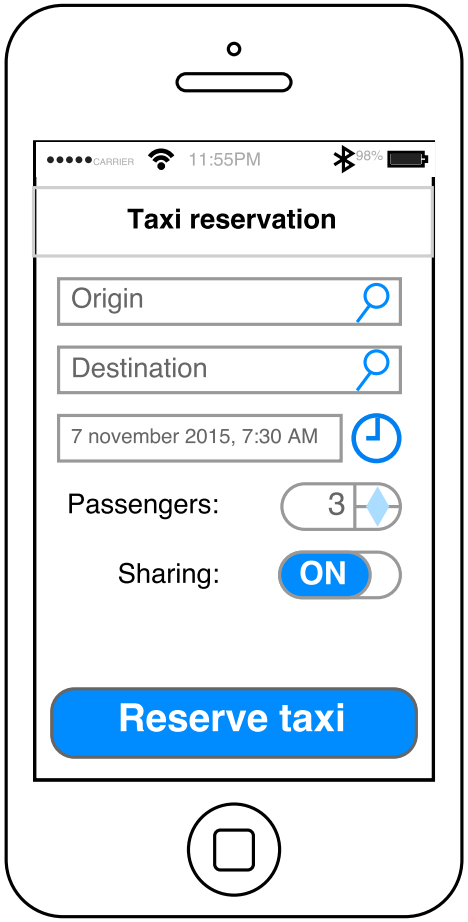
The algorithm computes $f_i$, that is the percentage of the fee that the passenger i has to pay. $f_i$ is expressed as a decimal number, i.e. $0 \leq f_i \leq 1$.

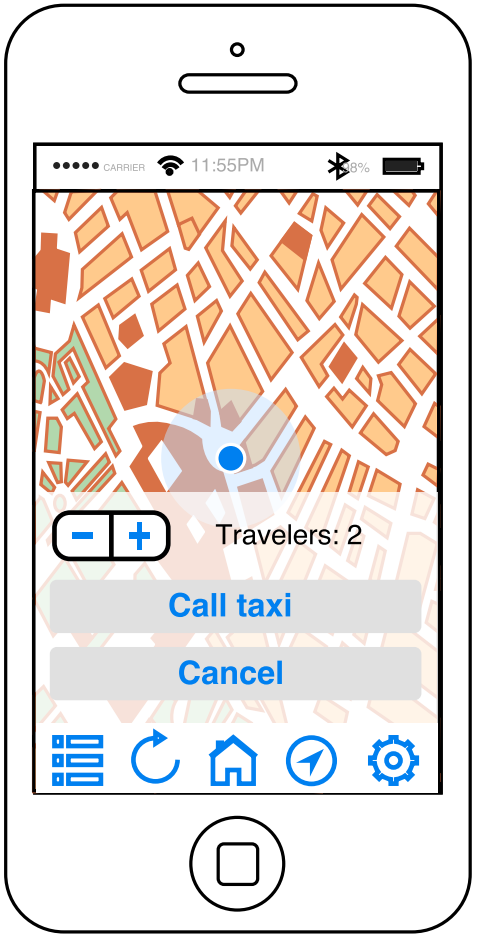
$$f_i = \frac{\sum_{t \in T_i} d(t)}{\sum_{p \in P} \sum_{t \in T_p} d(t)}$$

# User Interface Design : UX diagram

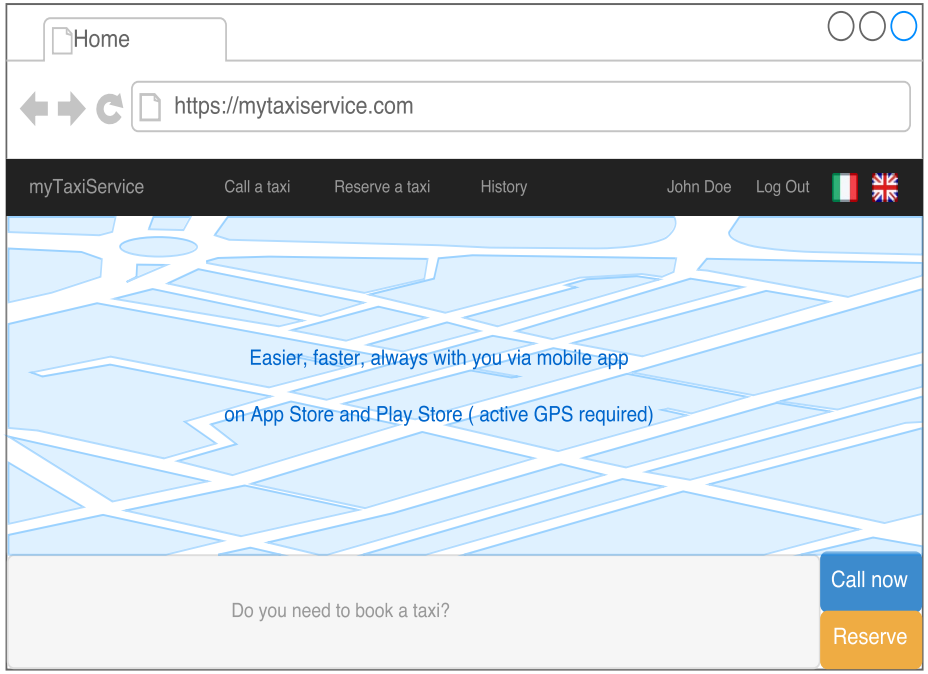


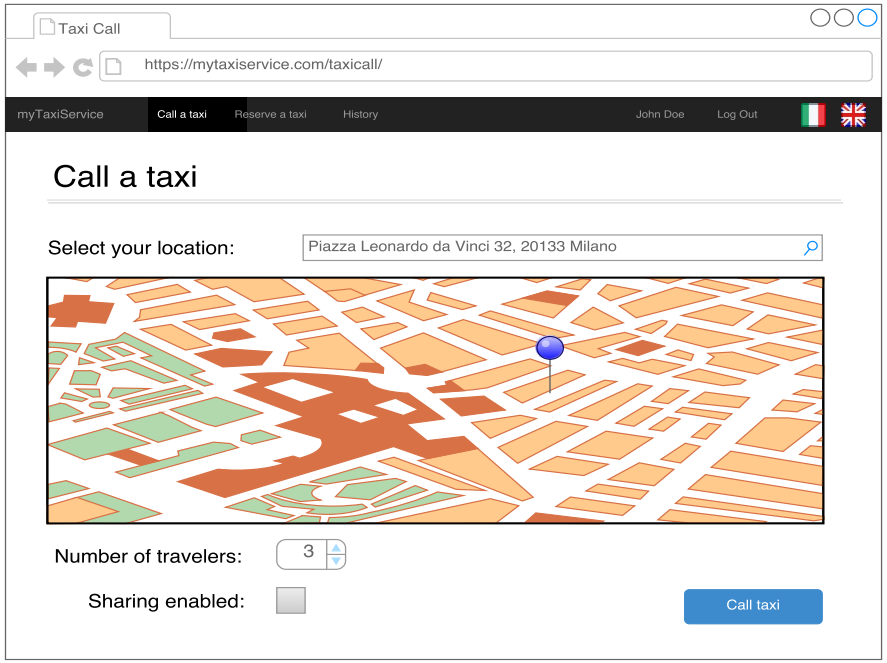
# User Interface Design : mockups

# User Interface Design : mockups



# Code Inspection

**Version 1.0**

The classes assigned are:

- CoordinatorLog
- CoordinatorLogSection
- RecoveryManager

They are located in the com.sun.jts.CosTransactions package of Java GlassFish Server.

# JTS: Java Transaction Service

- The inspection is about the JTS implementation by Sun.
- *"JTS specifies the implementation of a **transaction manager** which supports the Java Transaction API (JTA) specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 Specification at the low-level."*
- The classes assigned to us implement the **Reliability Control System** of the transaction manager.
  It ensures two important properties of transactions:
  - **Consistency**
  - **Durability**

# CosTransactions package

The assigned classes are the following:

- **CoordinatorLog**
  - keeps the transaction log, writes it to persistent storage
  - the log is similar to the DBMS logs
- **CoordinatorLogSection**
  - it's a section of the log
  - CoordinatorLog has a hashtable of sections
- **RecoveryManager**
  - handles recovery of data from the log in case of failure
  - like the warm restart protocol in DBMSs

# Results of inspection

## CoordinatorLog class

- Dump method does nothing
- Commented out code (L.112)
- Duplicated code:
  couples of methods that differ only for ***logPath*** parameter

## RecoveryManager class

- Methods getUniqueRMSet() (L.792) and getInDoubtXids() (L.864) start with “get”, but they are not getters of any attribute of the class
- Commented out code:
  - L.1305-1349
  - L.1759-1816

In both classes JavaDoc is missing or incomplete (paramethers not explained) for some methods

# CoordinatorLog class: startKeypoint

```
synchronized static boolean startKeypoint( LogLSN keypointStartLSN ) {
        CoordinatorLogStateHolder logStateHolder = defaultLogStateHolder;
```

```
synchronized static boolean startKeypoint( LogLSN keypointStartLSN, String logPath ) {
        CoordinatorLogStateHolder logStateHolder = getStateHolder(logPath);
```

*Brutish programming!*

```
byte[] keypointStartRecord = {(byte) 'K',
                              (byte) 'E',
                              (byte) 'Y',
                              (byte) 'S',
                              (byte) 'T',
                              (byte) 'A',
                              (byte) 'R',
                              (byte) 'T'};
logStateHolder.logFile.write(LogFile.UNFORCED,
             keypointStartRecord,
             LogFile.KEYPOINT_START,
             keypointStartLSN);
```

# CoordinatorLogSection class: doFinalize and reUse

```
/**Destroys the contents of a CoordinatorLogSection.
     *
     * @param
     *
     * @return
     *
     * @see
     */
public void doFinalize() {
        if( unwrittenObjects != null )
            unwrittenObjects.removeAllElements();

        if( unwrittenData != null )
            unwrittenData.removeAllElements();

        if( writtenObjects != null )
            writtenObjects.removeAllElements();

        if( writtenData != null )
            writtenData.removeAllElements();

        sectionName = null;

        unwrittenObjects = null;
        unwrittenData    = null;
        writtenObjects   = null;
        writtenData      = null;
    }
```

```
/**Cleans up the CoordinatorLogSection and
     * returns it to the pool for re-use
     *
     * Note: the implementation of the cache does not ensure
     *       that when an object is re-used there are no
     *       outstanding references to that object. However, the
     *       risk involved is minimal since reUse() replaces the
     *       existing call to finalize(). The existing call to
     *       finalize also does not ensure that there are no
     *       outstanding references to the object being finalized.
     *
     * @param
     *
     * @return
     *
     * @see
     */
    synchronized void reUse() {         // Arun 9/27/99

        if( unwrittenObjects != null )
            unwrittenObjects.removeAllElements();

        if( unwrittenData != null )
            unwrittenData.removeAllElements();

        if( writtenObjects != null )
            writtenObjects.removeAllElements();

        if( writtenData != null )
            writtenData.removeAllElements();

        sectionName = null;
        unwrittenEmpty   = true;
        writtenEmpty     = true;

    SectionPool.putCoordinatorLogSection(this);
    }
}
```

# Integration Test Plan Document

The Integration Test Plan Document (ITPD) purpose is to determine how to accomplish the integration test of the software, which tools will be used and which approach will be followed.

# Integration Strategy

## Entry Criteria

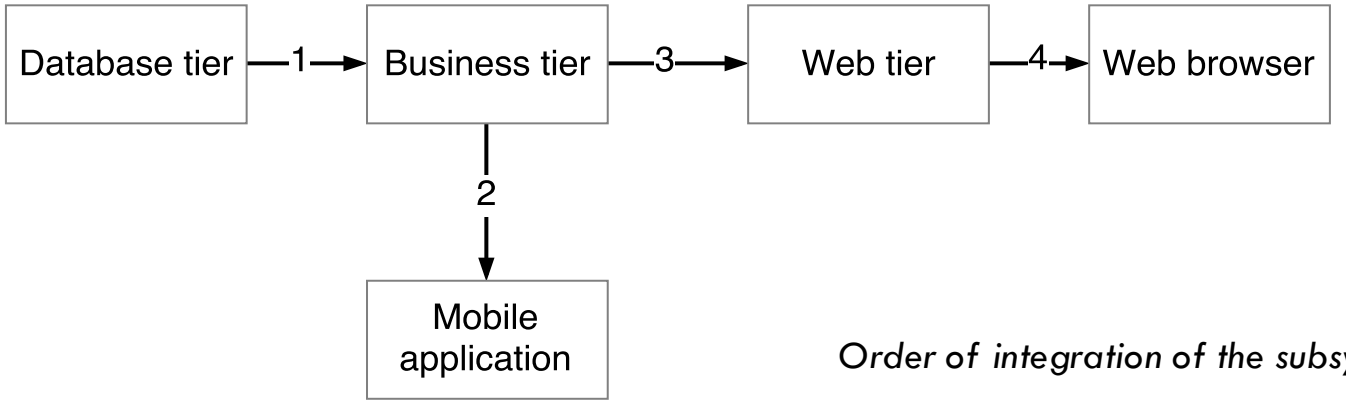
- **Unit test:**
  - minimum coverage of 90% of the lines of code
  - with JUnit
- **Code inspection**
  - maintainability
  - respect of conventions
  - find issues that could increase testers' effort
- **Documentation**
  - of all classes and functions,
  - with JavaDoc

## Integration Testing Strategy

***Bottom-up*** approach.

We assume to have the unit tests for the smallest components, so we can proceed from the bottom.

# Integration Strategy: subsystems



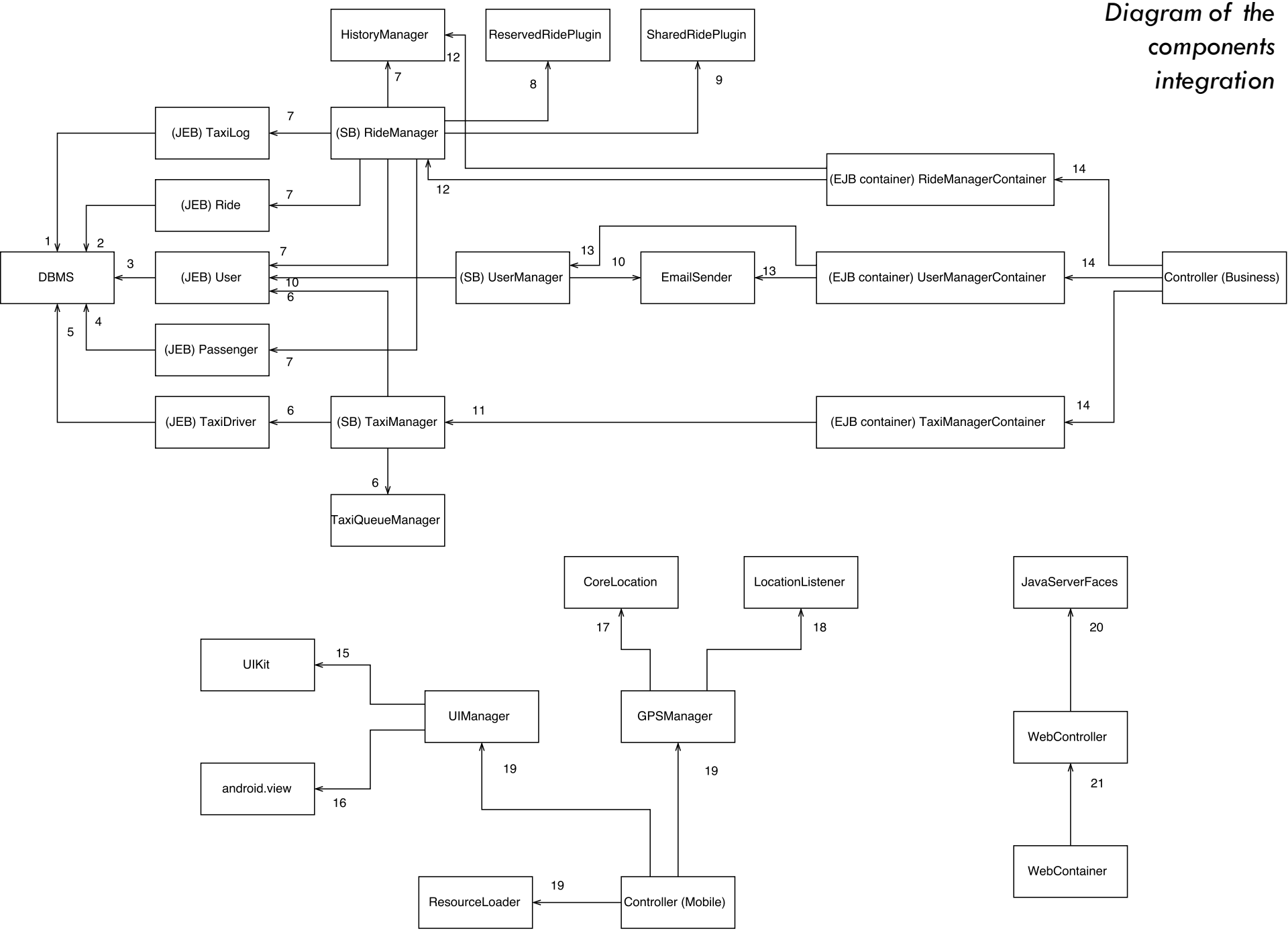

*Order of integration of the subsystems*

- Integration process **from server to client**
  - Client needs a working business tier
  - Server can be tested without client
- Integration of **mobile** before web tier
  - Taxi drivers need the mobile app

# Integration Strategy: components

- Components integrated **from the most independent** to the less one
- Avoiding useless stubs
- Components integrated within their classes
  - Creation of an integrated subsystem

# *Diagram of the components integration*



# Tools and Test Equipment Required

- **Apache JMeter:** to test the performance of subsystems
  - **Web tier:** simulation of heavy load on the web tier to check the response time with maximum number of simultaneously connected users
  - **Business tier:** simulate a heavy load on the REST API

  Tests on both sides are useful to identify bottlenecks.
- **JUnit:** framework for unit testing in Java
  - unit tests of the single components
  - also used for integration testing
- **Arquillian:**
  - test of the containers and their integration with JavaBeans
- **Mockito:** generates mock objects, stubs and drivers

# Program Stubs and Test Data Required

- **Test database:**
  - reduced set of instances of all the entities described in the ER diagram of the Design Document
- **Lightweight API client:**
  - interacts with the business tier by simple HTTP requests
- **Drivers for the Java Entity Beans:**
  - to test the Java Entity Beans when the Business Tier is not fully developed
- **Stub of the Business Tier:**
  - minimum set of data to test the web tier when the business tier is not fully developed
- **Mock e-mail sender and receiver**

# Project Plan

# Function Points estimation: weights

| Function | Type | Complexity | Weight |
|---|---|---|---|
| User | ILF | High | 15 |
| Passenger | ILF | Avg | 10 |
| TaxiDriver | ILF | High | 15 |
| TaxiLog | ILF | High | 15 |
| Ride | ILF | High | 15 |
| Maps | EIF | High | 6 |
| User registration | EI | Avg | 4 |
| User profile management | EI | Low | 3 |
| User login | EI | Low | 3 |
| Standard taxi call | EI | High | 6 |
| Taxi availability | EI | Low | 3 |
| Taxi reservation | EI | High | 6 |
| Notification to users | EO | Low | 4 |
| Confirmation mails | EO | Low | 4 |
| Notifications to taxi drivers | EO | Low | 4 |
| Taxi driver ride request | EQ | Low | 4 |
| Ride sharing | EQ | High | 6 |

# Function Points estimation: count

| **Function Type** | **Complexity-Weight** | | | **Total Points** |
|---|---|---|---|---|
| | Simple | Medium | Complex | |
| External Input | $3 \cdot 3$ | $1 \cdot 4$ | $3 \cdot 6$ | 31 |
| External Output | $3 \cdot 4$ | $0 \cdot 5$ | $0 \cdot 7$ | 12 |
| External Inquiry | $1 \cdot 3$ | $0 \cdot 4$ | $1 \cdot 6$ | 9 |
| Internal Logic File | $0 \cdot 7$ | $1 \cdot 10$ | $4 \cdot 15$ | 70 |
| External Interface File | $0 \cdot 5$ | $0 \cdot 7$ | $1 \cdot 10$ | 10 |
| **Total Points** | 24 | 14 | 94 | **132** |
| **SLOC** | $\times 53$ | | | **6992** |

# COCOMO: Scale Drivers

- **Precedentness:** Low
  - We have some experience of software design but most of the notions are new.
- **Development flexibility:** Nominal
  - We have to follow a prescribed process but be have a certain degree of flexibility
- **Risk resolution:** High
  - We carried out a detailed risk analysis
- **Team cohesion:** Very high
- **Process maturity:** High
  - Corresponding to CMM Level 3

# COCOMO: Cost Drivers

| **Code** | **Name** | **Factor** | **Value** |
|---|---|---|---|
| RELY | Required Software Reliability | Nominal | 1.00 |
| DATA | Data base size | Nominal | 1.00 |
| CPLX | Product Complexity | Nominal | 1.00 |
| RUSE | Required Reusability | High | 1.07 |
| DOCU | Documentation match to life-cycle needs | Nominal | 1.00 |
| TIME | Execution Time Constraint | Nominal | 1.00 |
| STOR | Main Storage Constraint | Nominal | 1.00 |
| PVOL | Platform Volatility | Low | 0.87 |
| ACAP | Analyst Capability | High | 0.85 |
| PCAP | Programmer Capability | Nominal | 1.00 |
| APEX | Application Experience | Very low | 1.22 |
| PLEX | Platform Experience | Very low | 1.19 |
| LTEX | Language and Tool Experience | Low | 1.09 |
| PCON | Personnel Continuity | Very high | 0.81 |
| TOOL | Usage of Software Tools | Nominal | 1.00 |
| SITE | Multisite Development | High | 0.93 |
| SCED | Required Development Schedule | High | 1.00 |
| **Total** | $EAF = \prod_i C_i$ | | 0.943 |

# COCOMO: Effort

$$\textbf{Effort} = 2.94 \times \text{EAF} \times \text{KSLOC}^{E}$$

- **EAF** $= \Pi_i C_i$: product of all cost drivers
- **E** $= 0.91 + 0.01 \times \Pi_i SF_i$: depends on the scale drivers
- **KSLOC**: Kilo-Source Lines of Code, as estimated by FPs

Total effort: **21.8 person-months**

# COCOMO: Duration

$$\textbf{Duration} = 3.67 \times (PM)^{0.28+0.2\times(E-0.91)}$$

- PM: effort (person-months)
- E: exponent depending on the scale drivers (same one of the effort equation)
- Total duration: 9.5 months.
  - This would result in 2.3 developers needed for this project.

Since our group consists of 3 people, a reasonable development time would be:

$$21.8 \text{ pm} / 3 \text{ people} = 7.3 \text{ months}$$

approximated to **8 months**.

# Risks: project risks

- **Delays over expected deadlines**
- Lack of communication among team members
- Lack of experience in programming with the specific frameworks
- **Requirements change**

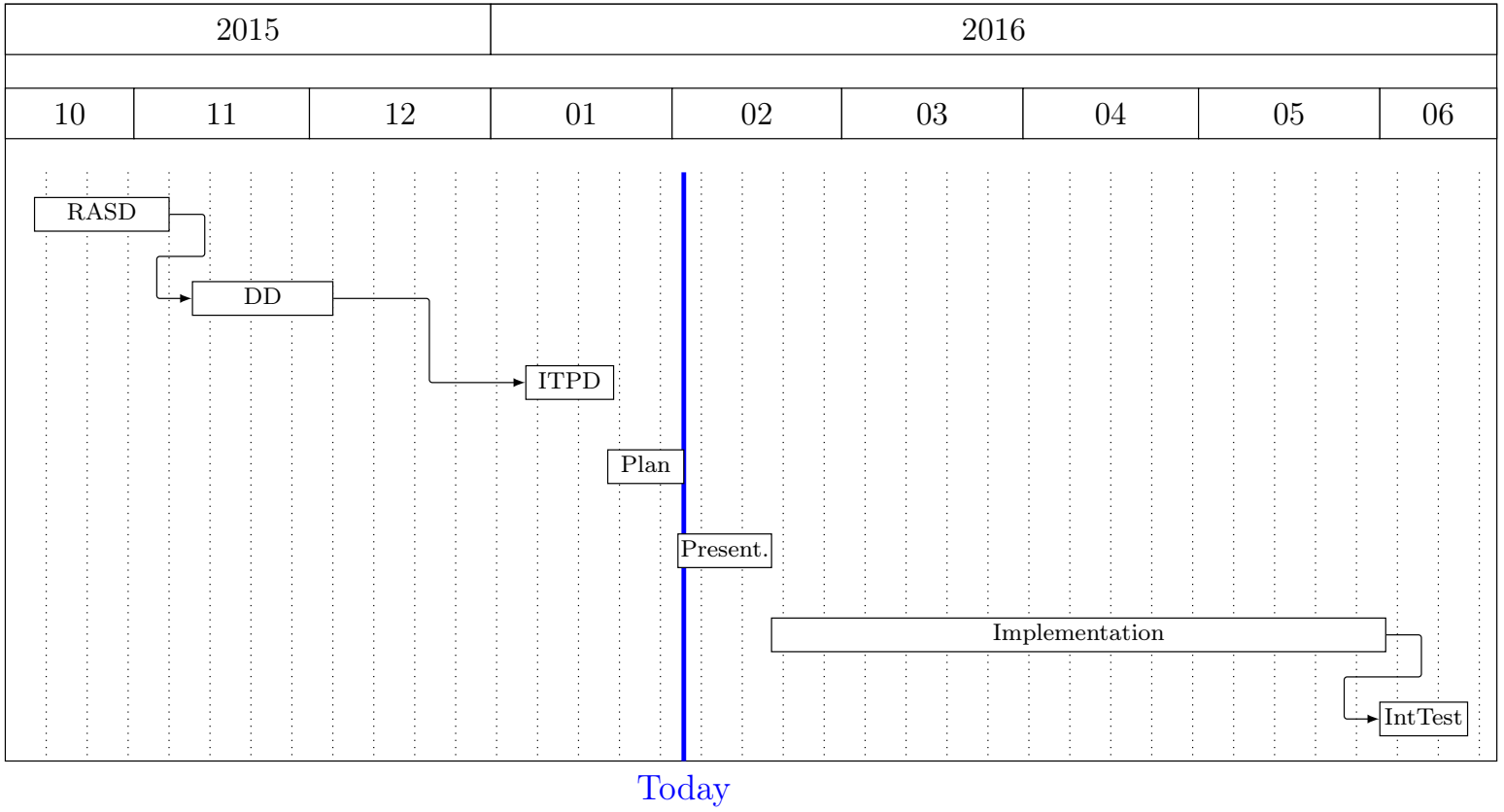
# Risks: technical risks

- Integration testing failure
- **Scalability issues**
- Spaghetti code
- **Deployment issues**
- Data loss
- Data leaks

# Risks: economical risks

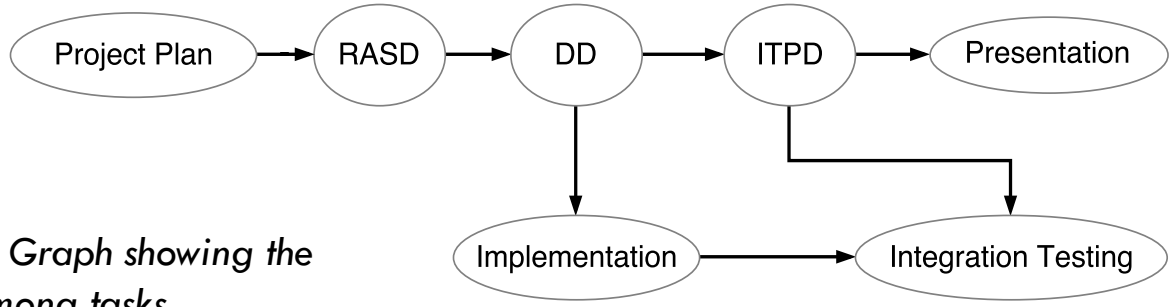
- *Bankruptcy:* not enough income to sustain development and deployment costs
- *Regulations change* by states and local authorities
- ***Competitors* who can put us out of the market**

# Tasks and schedule




*Gantt chart of the project*

# Tasks and schedule




*Directed Acyclic Graph showing the dependencies among tasks.*

| Activity | Start Date | Deadline |
|---|---|---|
| RASD | 2015-10-15 | 2015-11-06 |
| DD | 2015-11-11 | 2015-12-04 |
| ITPD | 2016-01-07 | 2016-01-21 |
| Project Plan | 2016-01-21 | 2016-02-02 |
| Presentation | 2016-02-02 | 2016-02-17 |
| Implementation | 2016-02-18 | 2016-06-01 |
| Integration Testing | 2016-06-01 | 2016-06-15 |

*Schedule for project tasks*