PFS

Parking for Smarties

Rolando Yanez
Quoc-Khanh Vuong
Robert Collins
Khin Htun
## Contents

- Project Abstract .......................................................................................................................... 3  
- Document Revision History ....................................................................................................... 3  
- System Architecture .................................................................................................................. 3  
  - ER Diagram ............................................................................................................................... 6  
- Design Details ............................................................................................................................. 7  
  - Android .................................................................................................................................. 7  
- UML Class Diagram for Server .................................................................................................... 15  
  - Server Side Class Implementation ........................................................................................... 17  
- Testing plan .................................................................................................................................. 23  
  - Server Testing .......................................................................................................................... 23  
  - Android Testing ....................................................................................................................... 24  
  - How testing is carried out ....................................................................................................... 24  
  - Integration tests ...................................................................................................................... 24  
  - System tests ............................................................................................................................. 25  
  - Regression testing ................................................................................................................... 25  
- Weekly Milestones .......................................................................................................................... 26
Project Abstract

Drivers want to get from point A to point B, but what if they’ve finally arrived to point B just to find out that there is no place to park? Now what? Turning back is rarely an option but looking for a place to park can easily be a hassle. Further more; parking congestion can be a liability for the owner of the company that operates the parking lot. The congestion can frustrate drivers, who are hunting for the parking. These frustrated drivers can become tense and aggressive, which then might increase the chances of inadvertent accidents with other vehicles, or even pedestrians, on the property. This can increase insurance rates and bring lawsuits against the company and can be a big disadvantage to those involved. So we ask: why not have a system that does it for us? Our team proudly introduces our product, a system that can avoid these complications for both the patrons and the parking owners. One in which can generically find and reserve parking spaces for our clients and can even do that before they get to their desired destinations.

Document Revision History

Rev. 1.0 2009-02-24 - initial version
Rev. 2.0 2009-03-12 - revised version, updated UML diagrams, include UI

System Architecture

The backbone of our architecture is built upon the three main blocks: the Android (G1 phone), the database, and the server: The fundamental structural organization schema between these three major components is represented in this simple diagram:
There will be an application on the Android that connects to an apache Tomcat server, which will redirect to a corresponding Java class. We chose Tomcat because it is an open source, simple to deploy, and easy to implement. The Tomcat application server will use the DAO (Data Access Objects) to communicate with database. Database will calculate the data and give feedback to the user Android interface through Tomcat which holds the Java DAO that will read data between the server and database. It will send and retrieve data regarding the queries. More about the DAO will be explained in details in a later section.
The design and structural methodology for class to class communication will be performed mostly in the backend between the server and the database. We will use the open source MySQL for the database to store all of the necessary information, tables, and statistics. Three of the tables (customers, parking spots, billing info) is shown in the diagram above. How data is represented within the database will be explained subsequently.

The Android will go to the server for a particular query and the server will retrieve the data from database, do calculations and send the result back to the phone. Based on the user’s request, the server will do all the computing, e.g. generate a Random number EID (E-Identification), which will be an E-ticket for the user to reserve a spot. This EID will be uniquely generated to prevent collision. The server will hold most of the functions and do all the computations, manipulate the data and store them back into the database. All the core methods such as requesting reservation and view available parking spots will be done here.

The Android Graphical User Interface (GUI) will contain all of the classes associated with the GUI and input/output methods. Android applications will also call Java classes in the server that will communicate with the database.
## Design Details
### Android

#### User Interface

<table>
<thead>
<tr>
<th>Name</th>
<th>Figure 1 (Startup screen)</th>
<th>Component</th>
<th>Comment/description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFS Loading ...</td>
<td></td>
<td>PFS Loading image</td>
<td>Appear when application is initiated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Figure 2 (Welcome)</th>
<th>Component</th>
<th>Comment/description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>Button</td>
<td>Proceed to Login (Figure 3)</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Button</td>
<td>Proceed to Register (Figure 4)</td>
<td></td>
</tr>
</tbody>
</table>
Table 1

<table>
<thead>
<tr>
<th>Name</th>
<th>Figure 3(Login)</th>
<th>Content</th>
<th>Component</th>
<th>Comment/description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>Username</td>
<td>Textbox</td>
<td>User Input username</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Password</td>
<td>Textbox</td>
<td>User Input password</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Login</td>
<td>Button</td>
<td>Checking for authentication (bring up GPS loading, Figure 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Back</td>
<td>Button</td>
<td>Return to previous Welcome screen (Figure 2)</td>
</tr>
</tbody>
</table>

Table 2

<table>
<thead>
<tr>
<th>Name</th>
<th>Figure 4(Register)</th>
<th>Content</th>
<th>Component</th>
<th>Comment/description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>Username</td>
<td>Textbox</td>
<td>User Input a desired username</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Password</td>
<td>Textbox</td>
<td>User Input password</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Username</td>
<td>Textbox</td>
<td>User confirm password</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phone</td>
<td>Textbox</td>
<td>User Input phone number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Submit</td>
<td>Button</td>
<td>Save new user into database (bring up GPS loading, Figure 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Back</td>
<td>Button</td>
<td>Return to previous Welcome screen (Figure 2)</td>
</tr>
</tbody>
</table>
GPS Loading ...

Thank you!

GPS is now locating a parking structure near you ...

According to your GPS location, these are the nearby parking structure:

- UC Merced, Lake Lot 1
- UC Merced, Lake Lot 2
- UC Merced, Resident Lot

Or you can search by:
City/State

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
<th>Comment/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPS Loading ...</td>
<td>System print a message</td>
<td>Letting the user know that the system is searching for a near parking structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Component</th>
<th>Comment description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure information</td>
<td>List</td>
<td>A list of available parking structures based on GPS Activate any link on list will bring up Reservation, Figure 7 (or similar) Proceed to Search by City/State (Figure 8)</td>
</tr>
<tr>
<td>Lake Lot 1 (or any)</td>
<td>Link</td>
<td></td>
</tr>
<tr>
<td>City/State</td>
<td>Link</td>
<td></td>
</tr>
</tbody>
</table>
**Figure 7 (Reservation)**

Name: UC Merced Lake Lot 1

<table>
<thead>
<tr>
<th>Component</th>
<th>Comment description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure information</td>
<td>Show key data for a parking structure as table</td>
</tr>
<tr>
<td>Reserve Spot</td>
<td>Submit request for reservation (bring up Figure 9)</td>
</tr>
<tr>
<td>Back</td>
<td>Return to previous Nearby Structure screen (Figure 6)</td>
</tr>
</tbody>
</table>

**Comment description**

- Phone: 209-228-8277
- Distance: 1.4 miles
- Capacity: 300
- Empty Spots: 31

**Figure 8 (Search by City/State)**

Name: Figure 8

<table>
<thead>
<tr>
<th>Component</th>
<th>Comment description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>User input a location</td>
</tr>
<tr>
<td>Enter a query</td>
<td>Search by City/State</td>
</tr>
<tr>
<td>Search</td>
<td>(bring up a screen similar to Figure 6)</td>
</tr>
</tbody>
</table>

**Comment description**

- Enter City, State (e.g., Merced, CA)
<table>
<thead>
<tr>
<th>Name</th>
<th>Figure 9(Confirming Reservation)</th>
<th>Content</th>
<th>Component</th>
<th>Comment description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back</td>
<td></td>
<td>Component Button</td>
<td>Button</td>
<td>Return to Reservation (Figure 7)</td>
</tr>
<tr>
<td>OK</td>
<td></td>
<td></td>
<td>Button</td>
<td>User Confirms (Proceed to Exit screen, Figure 10)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Figure 10(Exit)</th>
<th>Content</th>
<th>Component</th>
<th>Comment description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td></td>
<td>Component Button</td>
<td>Button</td>
<td>Exit &amp; Return to Android</td>
</tr>
<tr>
<td>Name</td>
<td>Figure 11(No Result)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td>There are no parking structures near Antioch, CA.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comment description</td>
<td>Return to Search by City/State (Figure 8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td><img src="image.png" alt="Image" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comment description</td>
<td>Return to Nearby Structure (Figure 6)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td><img src="image.png" alt="Image" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comment description</td>
<td><img src="image.png" alt="Image" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td><img src="image.png" alt="Image" /></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comment description</td>
<td><img src="image.png" alt="Image" /></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**GPS Discovery**

We will use the android.location.Location class to get latitude and longitude of the user. We will use the getLatitude() and getLongitude() functions to compute the distance towards the nearest parking structures. Unless we figure out a way to parse the Google SOAP interface we will simply compute the bird of a feather distance and not the road distance.

**Android Communication with Server**

For server communication our application will make use of two packages the Java.net package and the android.net package. Java.net will allow us to use a sockets interface that will talk to a custom servlet on our server. We will require android.net to query networking information. Http request are another possibility using the HttpURLConnection class in Java.net. The ConnectivityManager and NetworkInfo classes in android.net will be of interest in query the networking status information.

**User Program**

*Profile registration*

Our application will show a user registration page with a series of text boxes and a submission button.
Discovery

Polling GPS data will be the default location determination method. Will will then show a list of structures with their distances. Below will be a zip code text field to search for a different parking structure.

Reservations

Upon selecting a parking structure, there will be a reservation button where it will fill in your profile information.

Administrator Program

The administrator application will show initially a listing of functions:

- view global stats
- view profiles
- view parking assignments
- view charges
Server Side Class Implementation

DispatchAction Extends HttpServlet

DispatchAction class is a servlet class which acts as a controller. It will analyze the request object, process it and return the respond object back to the G1 phone.

public User createUser()

This method will use Connection object, UserDAO, and User classes to register a new user in the database. After successful insertion, the user's information will be stored in session object to recognize the user until he signs out.

User user = new User()

UserDAO userDao = new UserDAO(Connection conn)

PreparedStatment ps = new PreparedStatment("insertion statement into db")

ps.executeUpdate();

public User logIn()

This method will also use the Connection object, UserDAO, as well as User classes to retrieve the user's information from the database by using the user name and password provided in the requested object. After the successful retrieval, the information will be stored in session object. If this method cannot get the information, it will return null instead of the User object.

User user = new User()

UserDAO userDao = new UserDAO(Connection conn)
PreparedStatement ps = new PreparedStatement("retrieval statement into db")
ps.execute();

Connection extends java.sql.Connection
This class is subclass of the SQL Connection which will be used by all DAO objects to read and write to the tables from database.

public Connection static getConnection()
The function gets the Connection to the database. It is a static method so that any class can call this routine without creating an object first.

DriverManager.getConnection("URL", Driver);

User
This class the base class for Admin and Owner classes. All the functionalities of the user will be able to access from this class except for logIn and createUser which are handled by the DispatchAction Class.

public boolean isAdmin()
This routine will return ture if the user is an admin, false otherwise.

    If (status == 2)
    return true;
    else
    return false;
public boolean isOwner() 

This method will check whether the user is owner or not. If so, it will create a owner object instead of the regular user object.

    if(status == 3)
        return true;
    else
        return false;

public boolean makeReservation() 

Upon the request, the function will create a Reservation object to make a reservation for the user.

    Reservation reservation = new Reservation(User user);
    reservation.getSpots(Connection conn);
    return reservation.makeReservation();

public int resetPassword() 

If the user needs to reset the password for his account, this method will handle the request. However, the user needs to login to his account first to reset the password. The new password will be returned.

public boolean logout() 

The updated information will be written into the database and the session object for the user will be deleted.
public int viewReservation()

If the user needs to see the reservation that he has already made, the electronic identification (eID) will be returned. Using this eID, we can get the parking identification from the SPOTS table to redirect the page where he will be able to view his reservation. If there is no reservation, this method will return a negative number to indicate the error. Moreover, the eID is the key that will be needed for the user to type in at the entrance of the parking structure.

public double[] viewStat()

The return values, array of double will provide the users with the statistics of the particular parking structure so that he can avoid the rush hour.

public double[] charges()

When the user wants to check the charges by our parking system, the function will provide him with the data. The array will contain the time in millisecond counting from the January 1, 1970 and the associated charges.

Admin

This class is the direct subclass of the User class. In addition to the User’s functionalities, it has extra methods to operate and maintain the system.

public boolean resetSystem()
This method gives the administrator access to reset the statistics. It will return true if successful, false otherwise.

public boolean[] viewGlobal()

If a user complains that somebody parked at his assigned parking spot, the administrator will use this viewGlobal method to check whether the complaint is a valid one or not. If so, he can reassign that user to another available parking spot.

public void manageProfiles()

The administrator will be able to delete fraudulent accounts and reset the password for the user.

public boolean setRate(double)

If the parking rate needs to be adjusted, this function will set the new rate.

Owner

This class is the descendent of the Admin class. It inherits the functions and protected members from both User and Admin class.

public double[] viewStat()

This method overwrites the User’s viewStat. This function is intended to have more details of the statistics of the parking structure.

CreateID

The sole purpose of this class to generate a random for the electronic identification(eID) which will be used to link the users and their assigned parking spots.
**public int generateEID( )**

The integer length of 15 will be created using the time as a seed to generate a random number.

Random rd = new Random(System.currentTimeMillis);

double db = rd.nextDouble();

return db * 1000000000000000

**UserDAO**

The purpose of this class is to read and write the User table of the database. All the members in this DAO class have private access. However, these members can be set or get by using setter and getter respectively.

**SpotsDAO**

This DAO class is the only way for the java files to communicate with the table called Spots. All the members have private access which can be access by getter. Although eID can be set, the parking ID is not allowed to be set. Since the parking identifications are hard coded by the administrator during the deployment, it should not be able to write through application. If the correction is needed, the administrator should directly access to the database server to do so.

**BillingDAO**

This class is to retrieve and update all the charges. Most retrieval cases will be requested by the user and all update cases will be done by the server.
Testing plan

Since we are using the Eclipse platform to conduct our JAVA development, we will take advantage of various third-party plugins for unit testing. All classes will be unit tested. Systems tests will occur with our use case acceptance tests and possibly custom shell scripts to simulate input. Unit testing will be automated by collecting logical groups of unit tests into test suites, which should ease regression testing. In JAVA assertions can be enabled or disabled at will which is another capability we would need for regression testing when we make revisions after the midterm.

Server Testing

We will primarily be utilizing the JUnit testing framework to produce unit tests. These unit tests will allow us to practice the XP principle of test driven development. Another testing framework that could prove useful in testing the application server is TestNG. TestNG has features that JUnit does not possess such as Dependent methods and support for parameters. For classes that are used in more complex interactions where unit testing would be difficult, we will employ a Mock Objects testing framework called JMock. JMock will allow us to test out class functionality that may rely on the existence of a database for example or the states in our reservation system.

Testing of the database will be aided with the use of DBUnit which is an extension of JUnit. It will allow us to test out our database manipulation functions to ensure that reservations are made and profiles are stored.
Android Testing

Android testing will be done with Positron, which is Android and Selenium, a web testing suite, bundled together in the Android SDK. From the website, Positron runs Android stories, integration tests, or functional tests which would cover most of our testing needs.

How testing is carried out

All testing frameworks mentioned have an Eclipse plugin, therefore testing will be highly integrated into the build process. In some cases, running a test is as simple as clicking a button and receiving a visual cue as to whether the test passed or not. Grouping of the tests into test suites will allow us to run a series of unit tests with the click of a button, which would almost fully automate our testing needs. From its website, TestNG covers unit, functional, end-to-end, and integration testing to name a few. Our goal is that all code committed to the repository pass all current tests.

Integration tests

Integration tests will be eased with tools such as DBUnit and TestNG. We will test out the system in interaction pairs between a class and a database or webserver and phone before doing system tests. This implies that the components themselves will at least be at the committed stage. Testing at this level should help isolate critical bugs before System testing can occur. As soon as two components that interact together are checked in, we should run a test between them.
**System tests**

System tests will be carried out by the frameworks that support the use of functional tests such as TestNG. Doing so will automate testing between 3 or more modules since testing between two modules should have been completed at the integration stage. However, we realize that simply using the frameworks cannot account for every case we can conceive, therefore we will augment testing with custom scripts such as shell scripts or Perl scripts to further tackle the cases that do not lend themselves to those frameworks. Lastly, real human testing should occur in which we run through sample workflows and compare to the specification documents use case, acceptance tests.

**Regression testing**

We employ a simple strategy in our regression handling, make a unit test to show that a bug has been fixed. This is an easy and automated way to perform the regression test and should account for most of our testing needs. As to the frequency, we should do a regression test if that section of code is modified to ensure that code that is checked in is passing all of our unit tests.
**Weekly Milestones**

Week 7 (3/2): Have stack completely set up on Server
   Investigate HTTP request vs. Socket interaction

Week 8 (3/9): Implement ability to make reservation (function on server, call from phone)

Week 9 (3/16): View Reservation Function on Server and on phone.

Midterm demo on 3/19

Week 10 (3/23): [Spring break]

Week 11 (3/30): Implement Log In/Log out functions (functionality on server, sign in on phone.)

Week 12 (4/6): Create User and Cancel Reservation working

Week 13 (4/13): View Stats

Week 14 (4/20): View Global Assignments

Week 15 (4/27): Implement charging system and View Charges function

Week 16 (5/4): Integration and functionality testing
   Final demo on 5/7