

# **RU Parking**

---

**ANDROID-BASED APPLICATION FOR  
SUGGESTING PARKING SPACES**

<http://code.google.com/p/ruparking/>

## **ARCHITECTURE AND DESIGN REPORT**

**Project Members:**

**Cyrus Gerami**

**Ashwin Revo**

**Shweta Sagari**

**Samson Sequeira**

**Vrajesh Vyas**

## **Contents**

### **1. Introduction**

- 1.1 Purpose of the system**
- 1.2 Design goals**
- 1.3 Definitions, acronyms, and abbreviations**
- 1.4 References**
- 1.5 Overview**

### **2. Current software architecture**

### **3. Proposed software architecture**

- 3.1 Overview**
- 3.2 Subsystem decomposition**
- 3.3 Hardware/software mapping**
- 3.4 Persistent data management**
- 3.5 Access control and security**
- 3.6 Global software control**
- 3.7 Boundary conditions**

### **4. Subsystem services**

# 1. Introduction

---

## 1.1 Purpose of the system

In this time where the traffic has increased abundantly and there is a scarcity of available parking spaces, there are no reliable and convenient applications which assist users in finding vacant parking spots. Our application is intended to fill this void.

RU Parking is an application designed to run on Android OS based phones which provides users with information regarding vacant parking spaces. The application uses the data stored on a central server to provide a range of functionalities to all its users. It will assist the user in finding vacant parking spaces by employing real-time assessments of the parking scenario based upon statistical analysis of current and historical data.

The application will be implemented in a way that it is convenient to the user by having a well designed GUI and provide information in a format that is appropriate to the user.

## 1.2 Design goals

- **High-performance:** RUParking should have a low response time and must use the phone's available resources efficiently.
- **Robustness:** RUParking must be robust in the sense that the service should not crash when a large number of users are running the application. Also the application should not hang when there is invalid user input.
- **Reliability:** RUParking should be reliable i.e. the parking information provided to the user must be accurate and it should represent the actual parking scenario.
- **Cost effectiveness:** RUParking must have low development cost. Also the user should not have to spend a lot in obtaining and maintaining the application.
- **Flexibility:** It should be straightforward to change the functionality and also to add new functionality to RUParking.
- **Traceability of requirements:** It should be easy to map the code of RUParking to its specific requirements.
- **Ease of use:** RUParking should be user-friendly. The GUI must be effortless to interact with and it should not take too long for the user to comprehend the information given.

## 1.3 Definitions, acronyms, and abbreviations

**Android OS:** Android is a mobile operating system running on the Linux kernel. It allows developers to write managed code in the Java language, controlling the device via Google-developed Java libraries.

**Mobile User:** End user, which runs this application on a mobile phone, with personal interest.

**Garage Owner:** The owner of any parking garage, which uses this application for evaluation and analysis of block by block parking availability.

**Remote Server:** The server/database, which provides a decision/suggestion based on raw data.

**Destination Address:** The address where the user would like to find out the parking availability situation (most probably the trip destination).

**Parking Space:** Any segment of a street offering more than three road-side parking spots.

**GPS:** Global Positioning System, used for gathering user current coordinates

**SDK:** Software Development Kit, a set of development tools that allows us to create applications for a certain software package.

**GUI:** Graphical User Interface, a type of user interface, which allows people to interact with electronic devices such as mobile phones. A GUI offers graphical icons, and visual indicators, typed command labels or text navigation to fully represent the information and actions available to a user.

## 1.4 Reference

1. Android Developer's Guide  
<http://developer.android.com/guide/index.html>
2. Google Maps API developer documentation  
<http://code.google.com/apis/maps/documentation/index.html>
3. Primospot : A currently available iPhone application which suggests parking based on a database of parking signs and rules.  
<http://primospot.com/>
4. Requirements Analysis Document  
[Requirements Analysis Document.pdf](#)
5. Object-Oriented Analysis Report.pdf  
[Object-Oriented Analysis Report.pdf](#)

## 1.5 Overview

RU Parking application will be implemented as an **open architecture**. The application will be using a **three-tier architectural style**. The complete systems consist of an Android application running on mobile phones and parking information calculation and storage being implemented on a server. The application will communicate with the server through the internet. Our application also uses the Google API to obtain location-based services from Google.

There are two types of users this application is targeted at: Mobile Users and Garage Owners. The data that will be presented will be dependent on the type of user and will be in a format that is suitable to the type of user.

## 2. Current software architecture

---

The park space applications in the current market, base their decisions on static information about the parking spaces which means they can only direct the user to a parking space. These applications maintain a database of the legal spaces and their restrictions like parking hours, duration of parking etc. Such a system is inefficient since it does not provide any information about parking availability.

An example of such a system is Primospot, which is available for iPhone and Android market. By using client/server type architecture, this application will provide a map with all possible park spots marked in it. One can see “possible” park spots but not “available” ones. Here comes the difference from our application which will show possible park spots along with an availability factor.

There are applications that claim to identify vacant parking spaces in real-time by connecting sensors to the parking meters. Such a system requires large-scale changes in the existing infrastructure, which is not financially feasible. Also it is not applicable for street-side parking that provides for majority of the parking spaces in metropolitan cities. Also these parking spaces are usually metered in which case they are occupied by one vehicle for a short time like two hours. So there is a need to identify the vacancies in these parking spaces in real-time. Since these spaces do not have sensors this has not been achieved so far. RU Parking accomplishes this using the real-time data provided by the Roadside Parking Monitoring project group.

## 3. Proposed software architecture

---

### 3.1 Overview

RU Parking application will be implemented as an **open architecture**. This means that a layer can not only access the layer immediately below it but also layers at deeper level. We have applied partitioning and layering to obtain a subsystem decomposition of our system. RU parking is decomposed into **7 subsystems**; mobile phone and a centralized server are the hardware nodes on which these subsystems will be implemented. A portable Android application will be running on the phone and the parking information will be calculated and stored on the server. The application will communicate with the server through the internet. This will ensure minimum computational load on the phone and also decrease the response time. Our application also uses the Google API to obtain location-based services from Google. We are designing our application using a **three-tier architectural style**.

There are two types of users this application is targeted at: Mobile Users and Garage Owners. The data that will be presented will be dependent on the type of user and will be in a format that is suitable to the type of user. This is ensured by using password-based authentication at the time of installation.

## 3.2 Subsystem decomposition

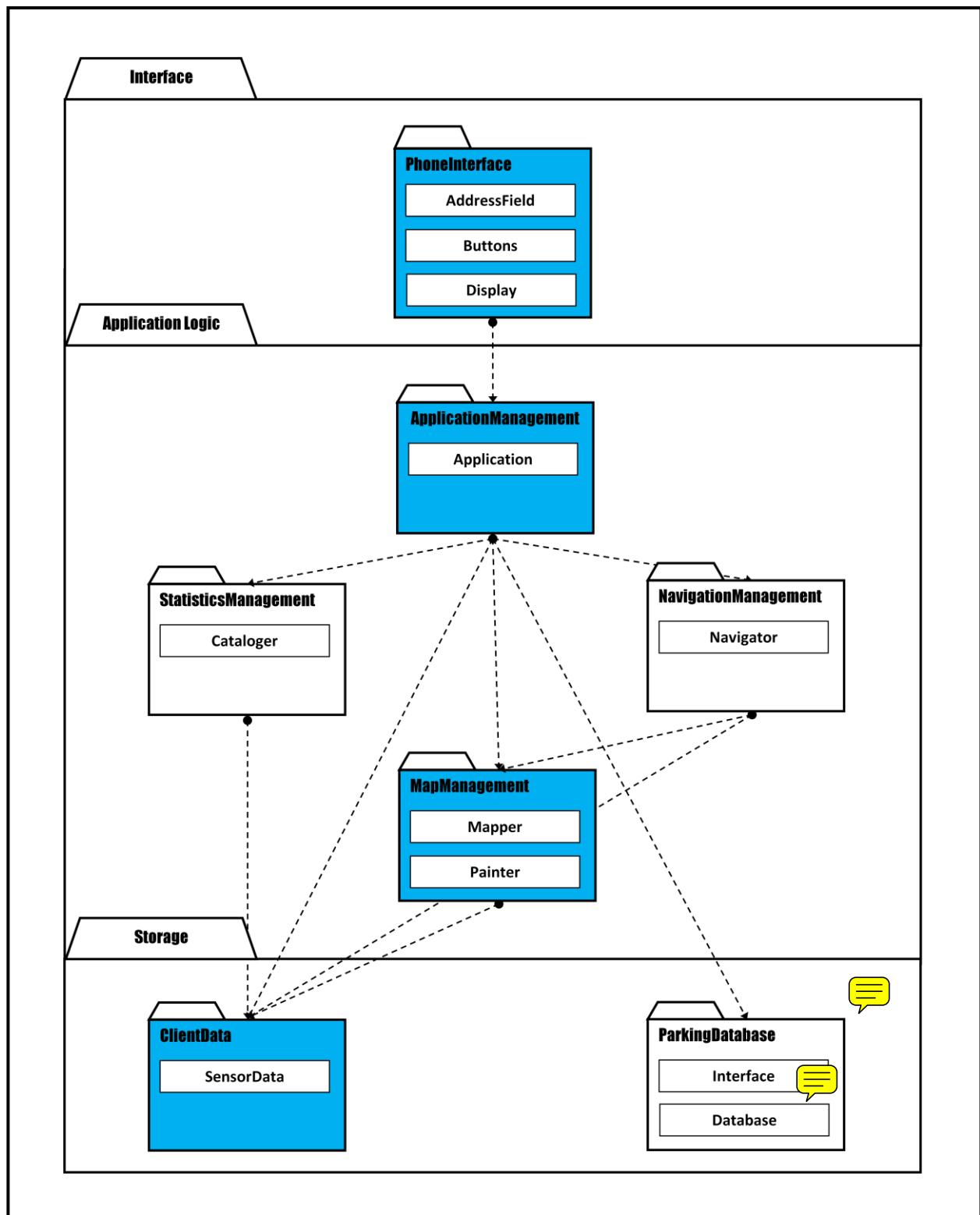


Figure 3.1 RU Parking subsystem decomposition



We have decomposed RU Parking into 7 subsystems as shown in Figure 3.1. Each subsystem has a specific functionality and it provides services to other subsystems. The decomposition is done with the objective of minimizing coupling and maximizing cohesion. The three-tier architectural style is used to provide layering and partitioning which makes the system organized. The classes that are to be implemented are included in the corresponding subsystems with their specific functions defined.

## Subsystem Description

### PhoneInterface

This subsystem provides an interface between the user and other subsystems in the application logic layer. It contains the classes AddressFields, Buttons and Display. AddressFields class provides a text field where the user can type in the destination address. Buttons class is the base class which is inherited by classes which each provide separate functionality when clicked. The Display class provides a display for the maps and the statistics which are seen by the user.

### ApplicationManagement

This subsystem provides an interface between the PhoneInterface subsystem and other subsystems in the application logic layer. It is responsible for access control and delegation of control to other subsystems. This contains a class application which takes the destination address as input parameter, converts this address into GPS coordinates and then activates other subsystems depending on its functionality.

### MapManagement

This subsystem generates colored map with parking spot information. It gets activated by the ApplicationManagement subsystem which passes the address coordinates. This class contains classes Mapper and Painter which use the parking information file to create relevant maps and activates the PhoneInterface subsystem which then displays the map.

### StatisticsManagement

This subsystem generates detailed list with parking spot information. It gets activated by the ApplicationManagement subsystem which passes the address coordinates. It contains class Cataloger which generates a list of parking spots closest to the destination address and activates the PhoneInterface subsystem which displays the list.

### NavigationManagement

This subsystem provides route to the selected destination address. It contains class Navigator which uses Google API's to calculate the route information. It activates the PhoneInterface subsystem which then displays this information.

**ParkingDatabase**

This subsystem provides parking information to the ClientData subsystem. It contains class Database which maintains the updated list of all parking information and accepts queries from the ApplicationManagement subsystem and returns relevant information to it.

**ClientData**

This subsystem contains relevant parking information for each application. It is accessed by the ApplicationManagement subsystem.

### 3.3 Hardware/software mapping

RUParking is the multi-user application as many users can access the application from their mobile phones. This requirement needs some central database and an internet/web connection between the database and the mobile.

The system is of Distributed nature. It has two hardware Nodes: 1) Mobile 2) Server.

1) Mobile: This is basically the mobile hardware, which provides interface between the user and the RUParking application. It has subsystems PhoneInterface, ApplicationManagement, MapManagement, ListManagement, NavigationManagement and ClientData. These subsystems provide the functionality for all use cases. The ClientData subsystem is the local database stored in the mobile memory. The mapping of ClientData on Mobile allows the system to access the Server only once and minimizes the overall response time of system.

The Virtual Machine on Mobile is the Android operating system. SQLite is part of the Android OS. We intend to use SQLite platform for ClientData.



2) Server: This node has the subsystem ParkingDatabase, which basically handles the central database for parking space availability.

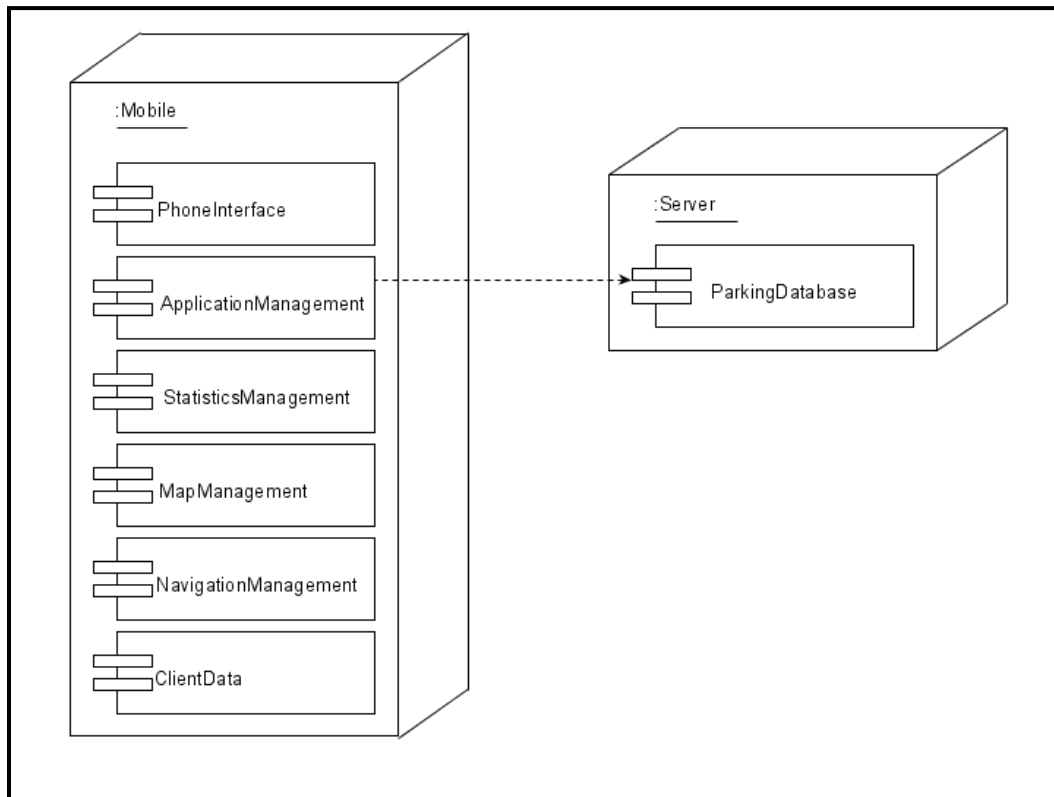


Figure 3.2 Allocation of RUParking subsystems to hardware

### 3.4 Persistent data management

Persistent data is the data which lives longer than a single execution of data and can reopen for future references. Persistent data must survive either a controlled shutdown or an unexpected crash. Entity objects identified in previous analysis documents would be candidates for persistent data.

#### Identify Persistent Data



In our system, entity object **Server** is the persistent data as it contains parking space availability database. This data is referred to whenever the user requests parking space availability information. This database remains in the Server after every execution and can be referred to later.

#### Selecting a storage management strategy

Storage management strategy affects the overall performance of the system and depends on various factors such as the size and type of data, information density, multiple access, etc.

In this system, the database maintains an extensive data list with small structured objects (Address, ParkingAvailability) for real-time parking space availability. This database can be accessed by two different platforms (MobileUser and GarageOwner). Also, multiple users can concurrently access the database. Based on the specifications of our system, it is most efficient to use **relational database** as our storage management strategy. With this type of database, data can be queried for InputAddress more effectively.

### 3.5 Access control and security

The access matrix for this system is shown below:

Objects Actors	Application	Cataloger	Mapper
MobileUser	selectMapManagement() selectNavigation() getCoordinates() update()		createMap() changeView() getAddress()
GarageOwner	selectStatManagement() getCoordinates() update()	createList() changeRadius() getAddress()	

Authentication for our software will be done at the beginning of the installation. As the user starts to install the application on a mobile phone, the application installer will request an authentication key. This key is provided to the user at the time of purchasing this application. Depending on this authentication key, the installer will differentiate between a mobile user and a garage owner in terms of authentication and access rights.

### 3.6 Global software control

The control flow of our system is procedure-driven. Our application uses a Procedure-Driven approach which can be described as follows. When the application starts, it awaits an input from the user. After getting an input, the ApplicationManagement subsystem issues a procedure call to MapManagement, StatisticsManagement or NavigationManagement depending on the user's input. And each of these subsystems call the ClientData subsystem for data to process accordingly.


Also, note that the overall control flow of the application along with the server is actually Event-Driven, since the server will give information to the system only when the application requests data from it.



### 3.7 Boundary conditions



Boundary conditions consist of initialization, termination and exceptions handling.

Initialization: 

The application is initialized when the user opens the application by clicking on the icon on a mobile phone. And the procedure is started when the user enter an address or chooses the current location. Which is described in our InputAddress usecase.

Termination:

The application is terminated when the user has reached the destination parking spot or prefers to have no further use for the application and presses the exit button on the phone. In this case, the file containing the data in the phone memory will be erased.

Exceptional Cases:

- 1) Internet Failure: if this connection is lost before ApplicationManagement control fetches data from the server, the application will need to wait for the connection to be fixed and the user should retry at a later time. If the connection is lost after the fetching, the application will work until the user requests an Update. At this point the Update usecase will be problematic and again the user will have to retry later.
- 2) Server Failure: In this case the application will not work since there will not be any persistent data on the phone. So, the application will remain idle until reconnected with the server and after that, it will get data and continue with the procedure.

## 4. Subsystem services



# Package phoneinterface

Provides an interface between the user and other subsystems in the application logic layer.

See:

[Description](#)

### Class Summary

<a href="#">AddressField</a>	This class consists of a text field for the user to provide the destination address to the system.
<a href="#">Buttons</a>	This class is the base class which describes the template for all the buttons.
<a href="#">Display</a>	This class handles the display functions required to display contents on the screen.

## Package phoneinterface Description

Provides an interface between the user and other subsystems in the application logic layer. Contains the classes AddressFields, Buttons and Display. AddressFields class provides a text field where the user can type in the destination address. Buttons class is the base class which is inherited by classes which each provide a separate functionality when clicked. Display class provides a display for the maps and the statistics which are seen by the user.

phoneinterface

### Class AddressField

java.lang.Object

**phoneinterface.AddressField**

```
public class AddressField
extends java.lang.Object
```

This class consists of a text field for the user to provide the destination address to the system. The text field takes an input of type String. The input from the user is the parameter sent to the Application Management subsystem. \* It belongs to the input terminal subsystem which is the interface between the user and the application.

**Version:1.0**

## Constructor Summary

[AddressField](#) ()

## Method Summary

void	<a href="#">createForm</a> () Generates the input text field that is displayed when the user starts the application.
------	---

### Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### AddressField

```
public AddressField()
```

## Method Detail

### createForm

```
public void createForm()
    throws IOException
    Generates the input text field that is displayed when the user starts the application.
```

**Parameters:**  
destination\_address - a String storing the destination address

**Throws:**  
IOException - if the input format is incorrect

phoneinterface

## Class Buttons

java.lang.Object

phoneinterface.Buttons

```
public class Buttons
    extends java.lang.Object
```



This class is the base class which describes the template for all the buttons. It is inherited by all the subclass buttons each of which provide a different service to the user. It belongs to the input terminal subsystem which is the interface between the user and the application.

**Version:**

1.0

## Constructor Summary

[Buttons](#) ()

## Method Summary

void [submit](#) ()

This function in the superclass Button calls the Application Management subsystem.

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Buttons

public **Buttons** ()

## Method Detail

### submit

public void **submit** ()

This function in the superclass Button calls the Application Management subsystem.

phoneinterface

## Class Display

java.lang.Object

View

**phoneinterface.Display**

```
public class Display
extends View
```

This class handles the display functions required to display contents on the screen. This class depends on the underlying hardware of the system which could be a mobile device or a personal computer. It extends the base class View which is specified in the Android API.

**Version:**  
1.0

## Constructor Summary

[Display](#) ()

## Method Summary

void	<a href="#"><u>displayMaps</u></a> () Displays the colored map containing parking information on the screen.
void	<a href="#"><u>displayStats</u></a> () Displays the list of parking information on the screen.

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Display

```
public Display()
```

## Method Detail

### displayMaps

```
public void displayMaps()
    Displays the colored map containing parking information on the screen.
```

### displayStats

```
public void displayStats()
    Displays the list of parking information on the screen.
```

# Package applicationmanagement

Provides an interface between the PhoneInterface subsystem and other subsystems in the application logic layer.

See:

[Description](#)

## Class Summary

[Application](#)

This class manages the control processes in the Application Management subsystem.

## Package applicationmanagement Description

Provides an interface between the PhoneInterface subsystem and other subsystems in the application logic layer. Responsible for access control and delegation of control to other subsystems. Contains a class application which takes the destination address as input parameter, converts this address into gps coordinates and then activates other subsystems depending on its functionality.

applicationmanagement

## Class Application

java.lang.Object

**applicationmanagement.Application**

```
public class Application
extends java.lang.Object
```

This class manages the control processes in the Application Management subsystem. It is called by the ActionListener functions implemented by the buttons which pass the user inputs to this class. The class triggers the correct control subsystem depending on this input.

Version:

1.0

## Constructor Summary

[Application](#) ()

## Method Summary

float	<a href="#"><u>getCordinates</u></a> () Returns coordinates that can then be used by ParkingDatabase subsystems.
float	<a href="#"><u>getCordinates</u></a> (java.lang.String destination_address) Returns coordinates that can then be used by ParkingDatabase subsystems.
void	<a href="#"><u>selectMapManagement</u></a> () This function contains the decision logic to be executed for delegation of control in the application.
void	<a href="#"><u>selectNavigatorManagement</u></a> () This function contains the decision logic to be executed for delegation of control in the application.
void	<a href="#"><u>selectStatManagement</u></a> () This function contains the decision logic to be executed for delegation of control in the application.
void	<a href="#"><u>update</u></a> () Updates the information file stored on the client machine.

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Application

```
public Application()
```

## Method Detail

### getCordinates

```
public float getCordinates (java.lang.String destination_address)
```

Returns coordinates that can then be used by ParkingDatabase subsystems. The destination\_argument must have address in a specific format.

This method uses the Google API

#### Parameters:

desination\_address - a String storing the destination address

#### Returns:

coordinates a float value corresponding to the GPS coordinates of destination address

---

## getCoordinates

`public float getCoordinates()`

Returns coordinates that can then be used by ParkingDatabase subsystems. This function is called when currentAddress Button is clicked by the user.

This function uses the Android phone GPS.

### **Parameters:**

`destination_address` - a String storing the destination address

### **Returns:**

coordinates a float value corresponding to the GPS coordinates of destination address

---

## selectMapManagement

`public void selectMapManagement()`

This function contains the decision logic to be executed for delegation of control in the application.

### **Parameters:**

`type` - an object of class Button

---

## selectStatManagement

`public void selectStatManagement()`

This function contains the decision logic to be executed for delegation of control in the application.

### **Parameters:**

`type` - an object of class Button

---

## selectNavigatorManagement

`public void selectNavigatorManagement()`

This function contains the decision logic to be executed for delegation of control in the application.

### **Parameters:**

`type` - an object of class Button

---

## Update

```
public void update()
```

Updates the information file stored on the client machine. It delegates control to respective classes.

# Package mapmanagement

Generates colored map with parking spot information.

See:

[Description](#)

Class Summary	
<a href="#">Mapper</a>	This class creates the map which is to be displayed on the screen.
<a href="#">Painter</a>	This class converts the relevance factor to a matching color.

## Package mapmanagement Description

Generates colored map with parking spot information. Gets activated by the Application Management subsystem which passes the address coordinates. Contains classes Mapper and Painter which use the parking information file to create relevant maps. Activates the PhoneInterface subsystem which then displays the map.

mapmanagement

## Class Mapper

```
java.lang.Object
```

```
mapmanagement.Mapper
```

---

```
public class Mapper  
extends java.lang.Object
```

This class creates the map which is to be displayed on the screen.

**Version:**

1.0

---

## Constructor Summary

[Mapper](#) ()

## Method Summary

int	<a href="#"><u>changeView</u></a> (int typeView) Returns address that can then be displayed by the Lister.
void	<a href="#"><u>createMap</u></a> (SensorData place) Creates the map to be displayed to the user.
java.lang.String	<a href="#"><u>getAddress</u></a> (SensorData place) Returns address that can then be displayed by the Cataloger.

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Mapper

```
public Mapper()
```

## Method Detail

### createMap

```
public void createMap(SensorData place)
```

Creates the map to be displayed to the user. It interfaces with the Google API to generate the relevant map.

**Parameters:**

place - is an object of type SensorData containing the location of the each parking spot.

---

## changeView

```
public int changeView(int typeView)
```

Returns address that can then be displayed by the Lister. This function will call the createList function in the Lister class.

**Parameters:**

typeView - integer which contains an integer corresponding to the type of map.

**Returns:**

typeView integer which contains an integer corresponding to the type of map.

---

## getAddress

```
public java.lang.String getAddress(SensorData place)
```

Returns address that can then be displayed by the Cataloger.

**Parameters:**

place - object containing the information corresponding to location of the parking spot.

**Returns:**

address a String corresponding to the list of address to be displayed.

mapmanagement

## Class Painter

```
java.lang.Object
```

```
mapmanagement.Painter
```

---

```
public class Painter  
extends java.lang.Object
```

This class converts the relevance factor to a matching color.

**Version:**

1.0

---

## Constructor Summary

<a href="#"><u>Painter</u></a> ()	
-----------------------------------	--

## Method Summary

java.lang.String	<a href="#"><u>colorMap</u></a> (SensorData place) Decides the color which is to be displayed to the for every location on the map based on its relevance factor.
------------------	--



## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Painter

```
public Painter()
```

## Method Detail

### colorMap

```
public java.lang.String colorMap(SensorData place)
```

Decides the color which is to be displayed to the for every location on the map based on its relevance factor.

**Parameters:**

place - is an object of type SensorData containing the location of the each parking spot.

**Returns:**

color String indicating which color the location should be displayed in

# Package statisticsmanagement

Generates detailed list with parking spot information.

See:

[Description](#)

## Class Summary

### [Cataloger](#)

This class implements all the functions needed to generate the list of statistics for the user.

## Package statisticsmanagement Description

Generates detailed list with parking spot information. Gets activated by the Application Management subsystem which passes the address coordinates. Contains class Cataloger which generates a list of parking spots closest to the destination address . Activates the PhoneInterface subsystem which displays the list.

statisticsmanagement

# Class Cataloger

java.lang.Object

**statisticsmanagement.Cataloger**

```
public class Cataloger
extends java.lang.Object
```

This class implements all the functions needed to generate the list of statistics for the user. The list contains the address of a street and the number of parking spots available on that street. Also a factor indicating the relevance of the information shown will be displayed.

**Version:**

1.0

## Constructor Summary

[Cataloger](#) ()

## Method Summary

void	<a href="#">changeRadius</a> (float radius) Returns address that can then be displayed by the Lister.
void	<a href="#">createList</a> (float radius) This function will generate a list containing parking spot information in the given radius.
java.lang.String	<a href="#">getAddress</a> (SensorData place) Returns address that can then be displayed by the Lister.

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Cataloger

```
public Cataloger ()
```

## Method Detail

## getAddress

```
public java.lang.String getAddress(SensorData place)
```

Returns address that can then be displayed by the Lister.

**Parameters:**

place - is an object of type SensorData containing the location of the each parking spot.

**Returns:**

address a String corresponding to the list of address to be displayed.

---

## changeRadius

```
public void changeRadius(float radius)
```

Returns address that can then be displayed by the Lister. This function will call the createList function in the Lister class.

**Parameters:**

radius - a float value storing the range of distance to be considered.

---

## createList

```
public void createList(float radius)
```

This function will generate a list containing parking spot information in the given radius.

**Parameters:**

radius - a float value storing the range of distance to be considered.

# Package navigationmanagement

Provides route to the selected destination address.

See:

[Description](#)

## Class Summary

[Navigator](#)

This class provides the route to guide the user from his current location to the destination parking spot.

## Package navigationmanagement Description

Provides route to the selected destination address. Contains class Navigator which uses Google API's to calculate the route information. Activates the PhoneInterface subsystem which then displays this information.

navigationmanagement

# Class Navigator

java.lang.Object

navigationmanagement.Navigator

```
public class Navigator
extends java.lang.Object
```

This class provides the route to guide the user from his current location to the destination parking spot.

**Version:**

1.0

## Constructor Summary

<a href="#">Navigator</a> ()	
------------------------------	--

## Method Summary

void	<a href="#">requestNavigation</a> (SensorData start, SensorData destination) Generates navigation information for the user to the destination parking spot.
------	--

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Navigator

```
public Navigator()
```

## Method Detail

### requestNavigation

```
public void requestNavigation(SensorData start,  
                               SensorData destination)
```

Generates navigation information for the user to the destination parking spot. The function uses the Google Maps API's to get this information. This information is then set to the Output Terminal to be displayed on the screen.

**Parameters:**

**start** - object containing the information corresponding to location of the parking spot.  
**destination** - object containing the information corresponding to location of the parking spot.

## Package parkingdatabase

Provides parking information to the File subsystem.

See:

[Description](#)

## Class Summary

<a href="#">database</a>	This class provides the interface between the Server and the application.
--------------------------	---

## Package parkingdatabase Description

Provides parking information to the File subsystem. Contains class Database which maintains the updated list of all parking information. Accepts queries from the Application Management subsystem and returns relevant information to it.

parkingdatabase

### Class database

java.lang.Object

**parkingdatabase.database**

---

```
public class database  
extends java.lang.Object
```

This class provides the interface between the Server and the application.

---

## Constructor Summary

<a href="#">database</a> ()	
-----------------------------	--

## Method Summary

SensorData	<a href="#">submitAddressToServer</a> (float coordinates) Returns the updated file depending on the input coordinates.
------------	---

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### database

```
public database()
```

## Method Detail

### submitAddressToServer

```
public SensorData submitAddressToServer(float coordinates)
```

Returns the updated file depending on the input coordinates.

#### Parameters:

coordinates - of the specified address

# Package clientdata

Contains relevant parking informaton for each application.

See:

[Description](#)

## Class Summary

<a href="#">SensorData</a>	This class contains the parking spot information on the phone.
----------------------------	--

# Package clientdata Description

Contains relevant parking informaton for each application. Accessed by the Application Management subsystem.

clientdata

## Class SensorData

java.lang.Object

**clientdata.SensorData**

```
public class SensorData  
extends java.lang.Object
```

This class contains the parking spot information on the phone. It contains an object Loc which has three variables corresponding to the location and relevance of the parking spot.

**Version:**

1.0

## Constructor Summary

<a href="#">SensorData</a> ()	
-------------------------------	--

## Method Summary

<a href="#">SensorData</a>	<a href="#">getMapData</a> (int typeView) Returns data to the mapmanagement subsystem depending on the view selected.
<a href="#">SensorData</a>	<a href="#">getStatData</a> (int typeRadius) Returns data to the statisticsmanagement subsystem depending on the radius specified.

## Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### SensorData

```
public SensorData()
```

## Method Detail

### getMapData

```
public SensorData getMapData(int typeView)
```

Returns data to the mapmanagement subsystem depending on the view selected.

**Parameters:**

typeView - integer which contains an integer corresponding to the type of map.

**Returns:**

SensorData object containing the information corresponding to location of the parking spot.

---

### getStatData

```
public SensorData getStatData(int typeRadius)
```

Returns data to the statisticsmanagement subsystem depending on the radius specified.

**Parameters:**

typeRadius - a float value storing the range of distance to be considered.

**Returns:**

SensorData object containing the information corresponding to location of the parking spot.

