Detailed Design Description

This is the detailed design description for Project 4: the CoPilot app. In this document we will explain in detail everything related to the design of this Android application, including decisions that the team has made about the project, and every other detail necessary to comprehend the important parts of the implementation process only by reading this document.

Contents

Client and product

The client assigned in this project is Volvo Construction Equipment. Using the words of our company contact, the scope of the project is “to implement a G-code program system for autonomous construction equipment and propose how possible and suitable it is to use the system. The evaluation should be carried out by implementing a system in a CoPilot and testing the program on a simulated machine”. G-Code is a programming language often used to control automated machine tools in computer-aided manufacturing (CNC). However, this language is old and outdated, and difficult to manage from a mobile phone, meaning that the task is also to find a better way to display G-code in mobile devices, without using the old format.

In short, what is needed is to create an android application where a user can create paths for autonomous machines to follow. These paths can either be created by physically walking through the workplace (a construction site or a quarry are the main focus for this feature, but any other intended workplace should work just fine), or pinpointing specific locations in the provided map and joining them with lines and curves for the machines to follow.

When referring to paths, it means a list of interconnected points that situate the machine along these predefined points in a designated order, in the real world. The activities the autonomous machine may perform while executing said path are: go forward in a straight line from a point A to a point B, go forward in a curve of a certain angle and radius specified in the G-code from point A to point B, load or unload the cargo, wait for a specified amount of time, wait for a signal, halt indefinitely and continue if a said event takes place.

This information can then be exported to the CoPilot tablet, which is a device installed in a control cabin outside the autonomous machines, ran by an operator. This device has an integrated in-house version of Android, and it allows the operator to control and receive information about the autonomous vehicles. The operator can create new paths, or program a certain path for the autonomous machine to follow when commanded.

Since the android application is intended for both planners (worker that creates the path) and operators, it should be able to run in both mobile phones and tablet devices.

In the current state of the application development it is not necessary to take altitude difference between points into account, but it would be necessary if the application were to be used in a realistic setting.

The client initially had a high level idea of the project, but it was further narrowed down to the requirements that is the base of this design document and all of the current decisions. Before making any decision, it was written down in a shared document and discussed with the client contact in the weekly meetings, where notes were taken on everything the client shared about the current progress, as well as answers to any questions posed. These notes were a necessity for continuous access to the information and to keep a written record of the meetings.

To read more about the requirements elicitation, please refer to the third appendix in our Project Plan document, which describes our requirements in detail.

To fully understand the approach to this project, the reader must know that the client, Volvo Construction Equipment, considers this product as a prototype for a product they are going to develop further once this implementation is done. The client is going to exchange the login system with an in-house developed one, that is restricted to company use only. They are also going to add new functionality, to make the application connect with the autonomous machines and execute the paths designed on them.

In essence, the client had designed a three-step process, which involved the following:

1. Creating an Android application in which planners and operators can plan, edit and execute paths in the autonomous machines available for them.
2. Considering all of the aspects the autonomous machines have to face during the execution of a path: soil composition, inclination of the ground, load and unload maneuvers, activity sensors so the machine stops before hitting a person or an object passing by, etc.
3. Linking the application to the autonomous machines and running the paths on them, so the operators can control multiple machines at once.

This is the full journey they plan on getting, the scope of this project only involves the first step, the core of the product. In the client contact's words, this product is "their way of knowing if what they are thinking about is viable and interesting to develop further". So, the initial goal is only to develop step one, but if there is enough time after finishing the first phase, implementation for the second phase will be initiated. However, for the moment this is not included in the development plan.

The design

To properly develop both a frontend and backend design for the project, close elicitation with the client was conducted. After receiving a description of desirable features, which can be found in Appendix 3, a rough mockup of the application was created and shown to the client. The initial understanding of the application and its functionality was communicated back to the client, who provided feedback on the mockup, and also added some extended requirements which were not defined in the previous description.

All of this feedback can be checked through the notes taken during the meetings in Appendix 1: The questions and answers, and Appendix 2: the notes, of this document.

With this information each team member created a mockup of the GUI to garner individual perceptions on how the application should work, which was then used to create a more finalized version of the mockup. A significant amount of time was spent discussing the advantages and drawbacks of each version when putting all the pieces together. These mockups where then joined in one to show the client, which can be seen in The graphical user interface, a later section of this document.

Main parts of the system

The CoPilot app should have the following functionalities:

- The user should be able to log in to the application.
- The user should be able to log out of the application.
- The user should have the ability to create new paths.
- The user should have the ability to edit previously created paths.
- The operator user should be able to reserve autonomous machines.
- The operator user should be able to release autonomous machines.
- The operator user should be able to execute paths.
- The application should display created paths.
- The application should be able to display paths in improved G-Code format.
- The application should display allocatable autonomous machines.
- The application should be able to send created paths to a cloud database.

To achieve these activities, three things are required: the application itself, which serves as an interface for the user; a database to store the data that should be accessible to multiple users; and lastly a GPS localization service, which will aid in the creation of paths. The GUI representation of the activities, the navigation between them, and how they will be implemented into our system will be described in more detail in the following sections.

External system

The application communicates with certain external systems for some of the activities related to the database and map functionality. The client did not have any demands regarding these external systems. The following systems were picked by the development team after being deemed suitable to the project. The chosen systems are Firebase and Google Maps.

Firebase

Firebase is a mobile development platform that provides a collection of different services. This framework is used for the following reasons.

- Firebase provides end-to-end user authentication. This way, there is no need to implement an authentication system from scratch, leaving more time to focus on desirable features in the product.
- Firebase provides a cloud-hosted real-time database. By using a cloud-hosted database there is no need to set up and maintain a server, or worry about scalability. It is also real-time so whenever data in the database changes, all connected devices will be updated automatically.
- This service has a lot of easy-to-understand documentation. This has proven to be helpful given any problem related to the implementation of the database.

Google Maps API

Google Maps is a web mapping service that provides GPS-localization and offers an open API. We use this external system for the following reasons.

- Google Maps provides a stylized map view, which we can be used for displaying the paths created by the planner.
- Google Maps can get the location of the application user, which planners can use as an input when adding points to the path.

Both of these external systems require a subscription for developers who want to have full use of their services. Firebase and Google Maps do however offer a free version of their service, with the added drawback that they have limited amount of requests per month. But this application is currently only a proof of concept, and will not generate a high amount of requests, so it is highly unlikely that these limits will actually be reached.

The final product should communicate with Volvo CE's own system for user authentication, but in this stage of the application development it is not necessary according to the client. Volvo CE will adapt the CoPilot app to their own system if they decide to continue the development of the application after this implementation is done.

This also applies to the API. The application is currently being developed by using API 18: Android 4.3 (Jelly Bean), but Volvo CE is using their own inhouse API for Android, and there was no information available about that API. The client also suggested to use whatever version is prefered, since they will be changing it afterwards.

Implementation structure

The implementation structure consists of the following diagrams:

- The Deployment Diagram.
- The Package Diagram.
- The Class Diagram.

Deployment Diagram

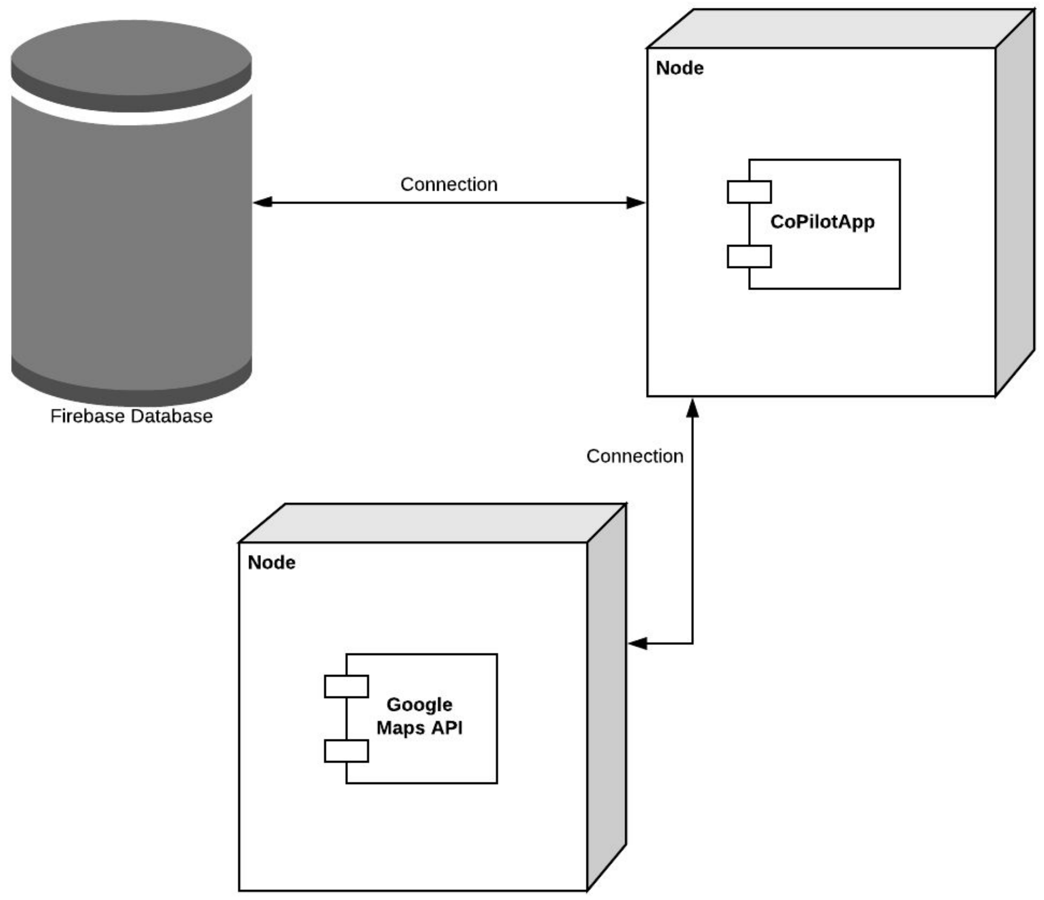


Figure 1: Project's deployment diagram.

The project's deployment diagram, consists only of three main nodes.

- The CoPilot app, designed and implemented by the development team.
- The Firebase database, an external service used to host the database of the application.
- The Google Maps API, an external service used to implement the visual path creation functionality of the application.

There are two connections. The first one is a connection between the Google Maps API and the CoPilot app; this connection is used to show Google Maps in the application, making a simpler interface and easier way to work with the application. This way, when creating new paths, the user can see a map obtained from the Google Maps API and choose the path based on that.

Also the client wanted to be able to switch between the satellite view and the map view, which makes Google Maps one of the best and easiest options to use for this technology.

There is also a connection between the Firebase database and the application, in order to save data and share it between different devices. For example, the same worker could create a path in his mobile, and use that path afterwards in the Volvo console, or in the tablet.

Firebase also provides user authentication, so the authentication was implemented following the Firebase pre-built authentication, to allow further focus on other aspects of the application, more specific to the product being developed.

Package Diagram

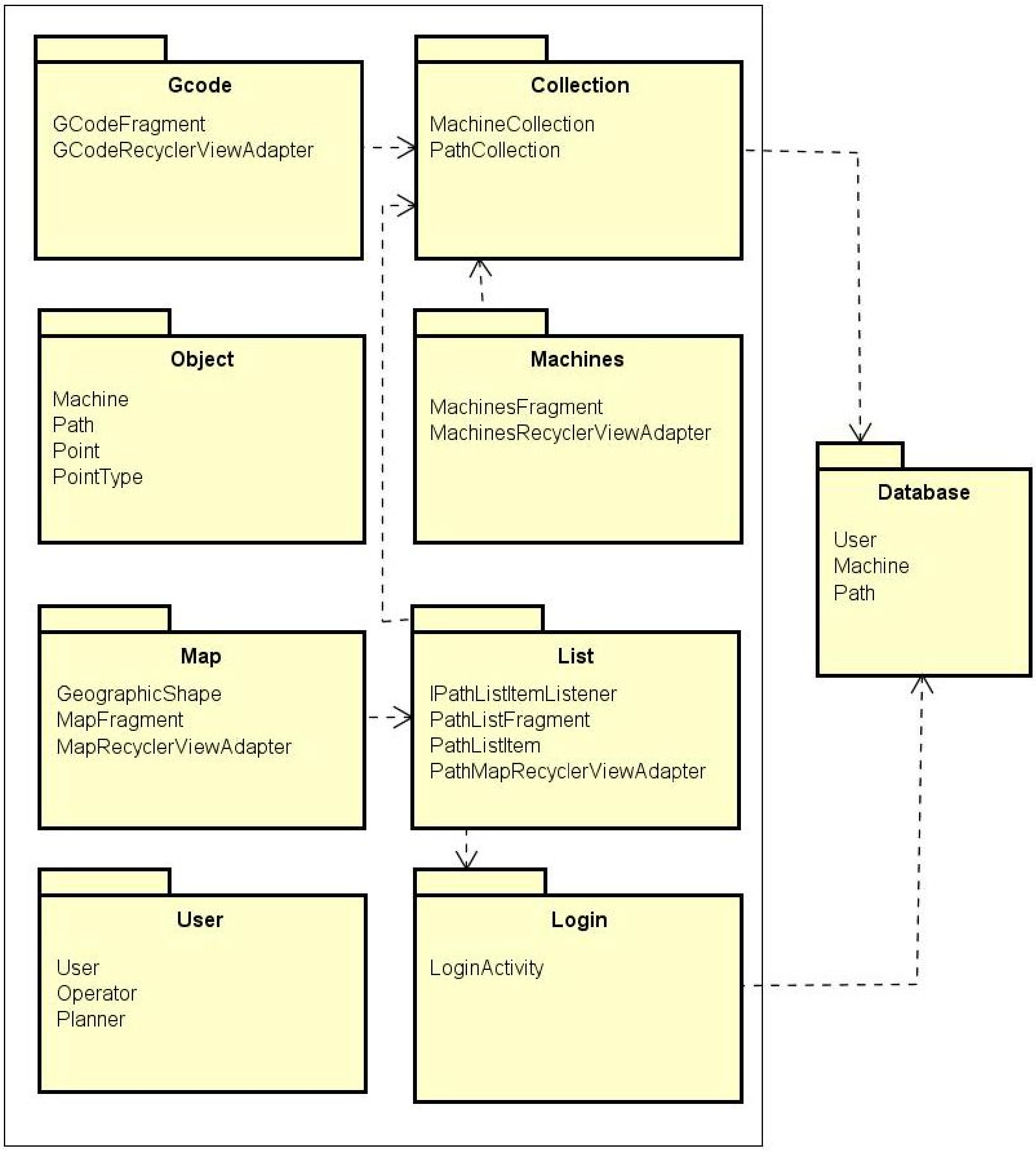


Figure 2: Package diagram. It displays the dependencies between our packages.

The application is distributed in seven packages for a higher code quality. These packages are the following: Database, Object, Login, User, List, Map, Machines, Collection Gcode.

The **Database** package is related with elements that are external to the application. It involves the use of the Firebase database.

It uses objects of other classes, like *Machine*, *Path*, *Point* or *PointType*. When the application requires to load objects of these classes, the database will load the data and return the objects, and after performing the pertinent functionality, if anything needs to be saved to the database, it will.

The **Object** package defines all the objects that can be created and used in the application.

The **Collection** package is the one that is mostly related to the **Database** since it is where the interaction between the application and the **Database** mostly takes place. Everytime the user make changes to the data in the local device this package push the changes to the database. If a change occur at the database the corresponding data in the local device is invalidated and the updated version replaces it.

The **Login** package is in charge of validating a user when they try to log in to the system. It is also in charge of registering a new user into the database and the application. As so, it is also related with the database, since it provides the new User's information, and needs to check if the information of a returning User is valid.

The **User** package works with the users. It contains the definition of both User types (Planner and Operator), and it is used to work with the user information. An example could be the machines an Operator has reserved, or a path a Planner has created.

The **List** package is used along the application to load path lists, adapt them to be visually appealing and interactive.

The **Map** package is also related with the Google Maps API, and contains all the functionality to display the map and create and edit paths inside the application. Following previous analogies, it could be referenced as the middleware between the Google Maps API and the application. This package also interacts with the **List** package, since the path is interpreted as a list of points. Thus, it is useful to manage these points inside the path as a **List** object.

The **GCode** package manages all the G-Code data, and shows it in a table for easy interaction. This also helps with the visual aspect of the G-Code view, making it resemble the old G-Code system further. This also has an interaction with the List package, as the G-Code of a path is a list of points as well.

The **Machine** package manages data related to the allocatable machines, and shows it in a table for easy of use.

Class Diagram

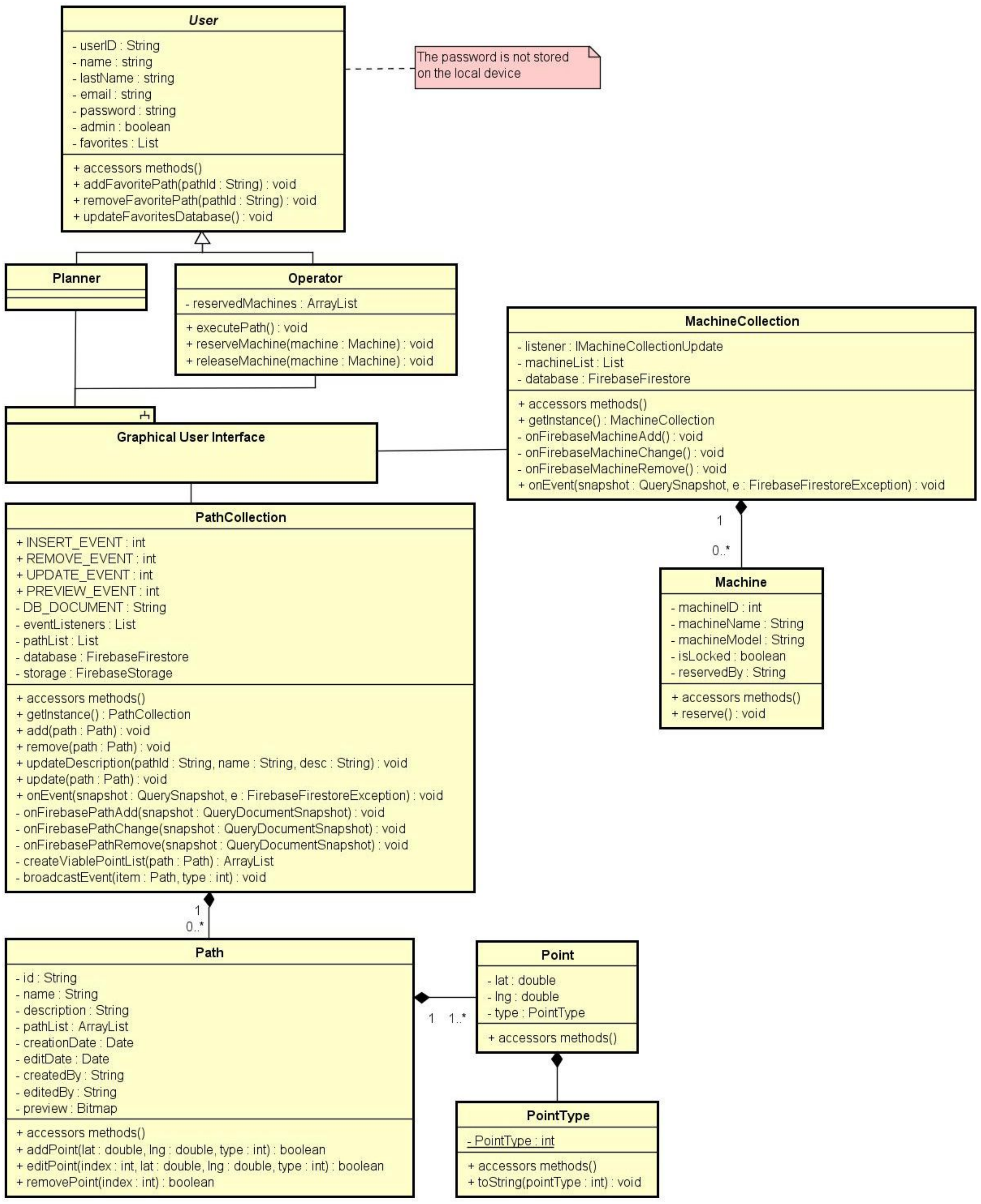

User
- userID : String
- name : string
- lastName : string
- email : string
- password : string
- admin : boolean
- favorites : List
+ accessors methods()
+ addFavoritePath(pathId : String) : void
+ removeFavoritePath(pathId : String) : void
+ updateFavoritesDatabase() : void
The password is not stored on the local device
Planner
Operator
- reservedMachines : ArrayList
+ executePath() : void
+ reserveMachine(machine : Machine) : void
+ releaseMachine(machine : Machine) : void
Graphical User Interface
MachineCollection
- listener : IMachineCollectionUpdate
- machineList : List
- database : FirebaseFirestore
+ accessors methods()
+ getInstance() : MachineCollection
- onFirebaseMachineAdd() : void
- onFirebaseMachineChange() : void
- onFirebaseMachineRemove() : void
+ onEvent(snapshot : QuerySnapshot, e : FirebaseFirestoreException) : void
1
0..*
Machine
- machineID : int
- machineName : String
- machineModel : String
- isLocked : boolean
- reservedBy : String
+ accessors methods()
+ reserve() : void
PathCollection
+ INSERT_EVENT : int
+ REMOVE_EVENT : int
+ UPDATE_EVENT : int
+ PREVIEW_EVENT : int
- DB_DOCUMENT : String
- eventListeners : List
- pathList : List
- database : FirebaseFirestore
- storage : FirebaseStorage
+ accessors methods()
+ getInstance() : PathCollection
+ add(path : Path) : void
+ remove(path : Path) : void
+ updateDescription(pathId : String, name : String, desc : String) : void
+ update(path : Path) : void
+ onEvent(snapshot : QuerySnapshot, e : FirebaseFirestoreException) : void
- onFirebasePathAdd(snapshot : QueryDocumentSnapshot) : void
- onFirebasePathChange(snapshot : QueryDocumentSnapshot) : void
- onFirebasePathRemove(snapshot : QueryDocumentSnapshot) : void
- createViablePointList(path : Path) : ArrayList
- broadcastEvent(item : Path, type : int) : void
1
0..*
Path
- id : String
- name : String
- description : String
- pathList : ArrayList
- creationDate : Date
- editDate : Date
- createdBy : String
- editedBy : String
- preview : Bitmap
+ accessors methods()
+ addPoint(lat : double, lng : double, type : int) : boolean
+ editPoint(index : int, lat : double, lng : double, type : int) : boolean
+ removePoint(index : int) : boolean
1
1..*
Point
- lat : double
- lng : double
- type : PointType
+ accessors methods()
PointType
- PointType : int
+ accessors methods()
+ toString(pointType : int) : void

Figure 3: UML class diagram. It consists of seven different classes: User, Planner, Operator, Machine, PathCollection, Path, Point, PointType.

The PointType class

The PointType class represents the action the autonomous vehicles should take at a specific point. This class only has one attribute:

1. *PointType* is a integer that can only be set to a subset of predefined values which represents different activities. Currently there are seven activities, which are LINE, CURVE, LOAD, UNLOAD, WAIT, START_POINT and lastly FINISH_POINT.

This class only contains one method, beside the constructor and accessor methods:

1. *toString(int pointType)* this method convert a point type value into a string.

The Point class

This class represents the points in the paths. The class contains three attributes:

1. *lat* is the latitude of the coordinate.
2. *lng* is the longitude of the coordinate.
3. *type* is the activity that is to be done at that point. This attribute can be used to signal to the autonomous vehicles that they need to unload their cargo, or begin a turn, for example.

This class only contains a constructor and accessor methods.

The Path class

This class represents the two dimensional paths that the autonomous vehicles can follow. This class has nine attributes, which are:

1. *id* is a unique identification for the path.
2. *name* is the name of the path, which does not necessarily need to be unique.
3. *description* is a short description of the path.
4. *pathList* is an ArrayList that contains all the points in the path.
5. *creationDate* is the date when the path was created.
6. *editDate* is the date when the path was last edited.
7. *createdBy* is the userID of the original creator of the path instance.
8. *editedBy* is the userID of the user of that made the most recent edit of the path.
9. *preview* is a Bitmap which contains a preview image of the path.

This class has three methods beside the accessor and constructor methods:

1. *addPoint(double lat, double lng, int type)* adds a new point to the path, if the path is still valid after the insertion. The function returns true if the node was added; otherwise, it will return false.
2. *editPoint(int Index, double lat, double lng, int type)* edits the content of a existing node if the path is still valid after the change. The function return true if the node was changed, otherwise it will return false.
3. *removePoint(int Index)* removes a node from the path, as long as the path is still valid after the change. The function return true if the node was removed, otherwise it will return false.

The Machine class
The Machine class represents the autonomous vehicles that will follow the path. This class only have four attributes:

1. *machineID* is the unique identification of each machine. This value is assigned by Volvo CE.
2. *machineName* is the name assigned to the machine by the owner of the machine and it does not necessarily need to be unique.
3. *machineModel* is the model of the Machine.
4. *isLocked* is a Boolean used to reserve the machine. If the attribute is set to true, the machine is reserved by someone and cannot be allocated by another operator until it is released. If the attribute is set to false, the machine is available for allocation by any operator.
5. *reservedBy* contains the userID of the user who reserved the machine. If the machine is not allocated this attribute will contain an empty string.

The Machine class only has one method and a constructor:

1. *reserve()* toggles between true and false for the reserved attribute.

The User class
The User class is an abstract class, and it represents the application user and the activities they can perform using the CoPilot app. This class contains seven attributes:

1. *userID* is a unique identification for the user that the system assigns.
2. *name* is the first name of the user.
3. *lastName* is the last name of the user
4. *email* is the work email of the user, which is used in the authentication process.
5. *password* is the password the user uses during authentication.
6. *admin* is a boolean that determine if the user has administrator privileges.
7. *favorites* is a list that contains the identification of path the user assigned as favorites.

This class has three methods besides accessor methods and constructor:

1. *addFavoritetPath(String pathid)* is used to add a new favorite path to user.
2. *removeFavoritePath(String pathid)* is used to remove a favorite path from user.
3. *updateFavoritesDatabase()* is a method that is used to update the database when changes to the users favorite paths occur.

The Operator class
The Operator class represents the users that will operate the autonomous vehicles while they drive along the predefined path. This type of user has the ability to reserve and release machines from a pool of available machines. The operator class inherits methods and attributes from the user class, but it has one attribute of its own:

1. *reservedMachines* is an ArrayList that contains all the machines the operator has reserved.

This class also has three additional methods:

1. *executePath(Path path)* allows Operator to send machines on a path.
2. *reserveMachine(Machine machine)* reserves a available machine, which means no other operators will be able to allocate that machine until it is released.
3. *releaseMachine(Machine machine)* releases a reserved machine, which means that other operators will be able to allocate that machine.

The Planner class
The Planner class represents users that are only able to create or edit paths, and will not be able to reserve machines or execute paths. The planner class inherits all its methods and attributes from the User class.

The PathCollection class
The PathCollection class is a singleton class that handles a list of all the available path objects, and provides database interaction for handling of path creation, editing and removal. Thus, when a path is created or modified, the information of the user who changes the path is saved. This has also been useful when retrieving information from this class, since the relation between them helps with other features. This class contains nine attributes:

1. *INSERT_EVENT* is used to identify insert events.
2. *REMOVE_EVENT* is used to identify remove events.
3. *UPDATE_EVENT* is used to identify insert event.
4. *PREVIEW_EVENT* is used to identify preview event.
5. *DB_DOCUMENT* is a reference to the document that stores paths in the database.
6. *eventListeners* is a list that contains listener that listens for events related to the paths.
7. *pathList* is a list that contains all the of path that are stored in the database.
8. *database* is a handle for communication between the application and the database.
9. *storage* is a handle for the cloud file storage.

This class contains eleven methods besides the accessors and constructo methods:

1. *getInstance()* is a method that gets the singleton of the pathCollection class.
2. *add(Path path)* adds a path to the database.
3. *remove(Path path)* removes a path from the database.
4. *updateDescription()* updates the name or description of an existing path in the database, it also logs the current date and the identification of the user who performed the edit.
5. *update(Path path)* updates a paths content and preview image. The method also logs the current date and the identification of the user who performed the edit.

6. *onEvent(QuerySnapshot snapshot, FirebaseFirestoreException e)* is a method that listen for firebase events (add, update and remove).
7. *onFirebasePathAdd(QueryDocumentSnapshot snapshot)* this method adds newly created paths, that are created by other users, to the paths that are stored on the local device.
8. *onFirebasePathChange(QueryDocumentSnapshot snapshot)* this method updates paths stored on the local device when another user make a change to an existing path in the database.
9. *onFirebasePathRemove(QueryDocumentSnapshot snapshot)* this method removes an existing path from the local device, when another user with admin status removes a path from the database.
10. *createViablePointList(Path path)*
11. *broadcastEvent()*

The MachineCollection class
The MachineCollection class is a singleton class that handles a list of all the available machine objects, and provides database interaction for handling of path creation, editing and removal. This MachineCollection class contains three attributes:

1. *listener* listener that listens for events related to the paths.
2. *machineList* is a list that contains all the of machines that are stored in the database.
3. *database* is a handle for communication between the application and the database.

This class contains four methods besides the accessors and constructor methods:

1. *getInstance()* is a method that gets the singleton of the MachineCollection class.
2. *onFirebaseMachineAdd()* This method adds newly created machines, that are created by other users, to the machines that are stored on the local device.
3. *onFirebaseMachineChange()* This method updates machines stored on the local device when another user make a change to an existing path in the database.
4. *onFirebaseMachineRemove()* this method removes an existing machine from the local device, when another user with admin status removes a path from the database.

Graphical user interface

Graphical user interface for mobile

The graphical user interface (GUI) in this application consists of the login screen, path list, path view, G-Code view and machine reservation view. The android design guidelines was attempted to be upheld when developing the GUI, to make it as easy as possible for new users to understand.

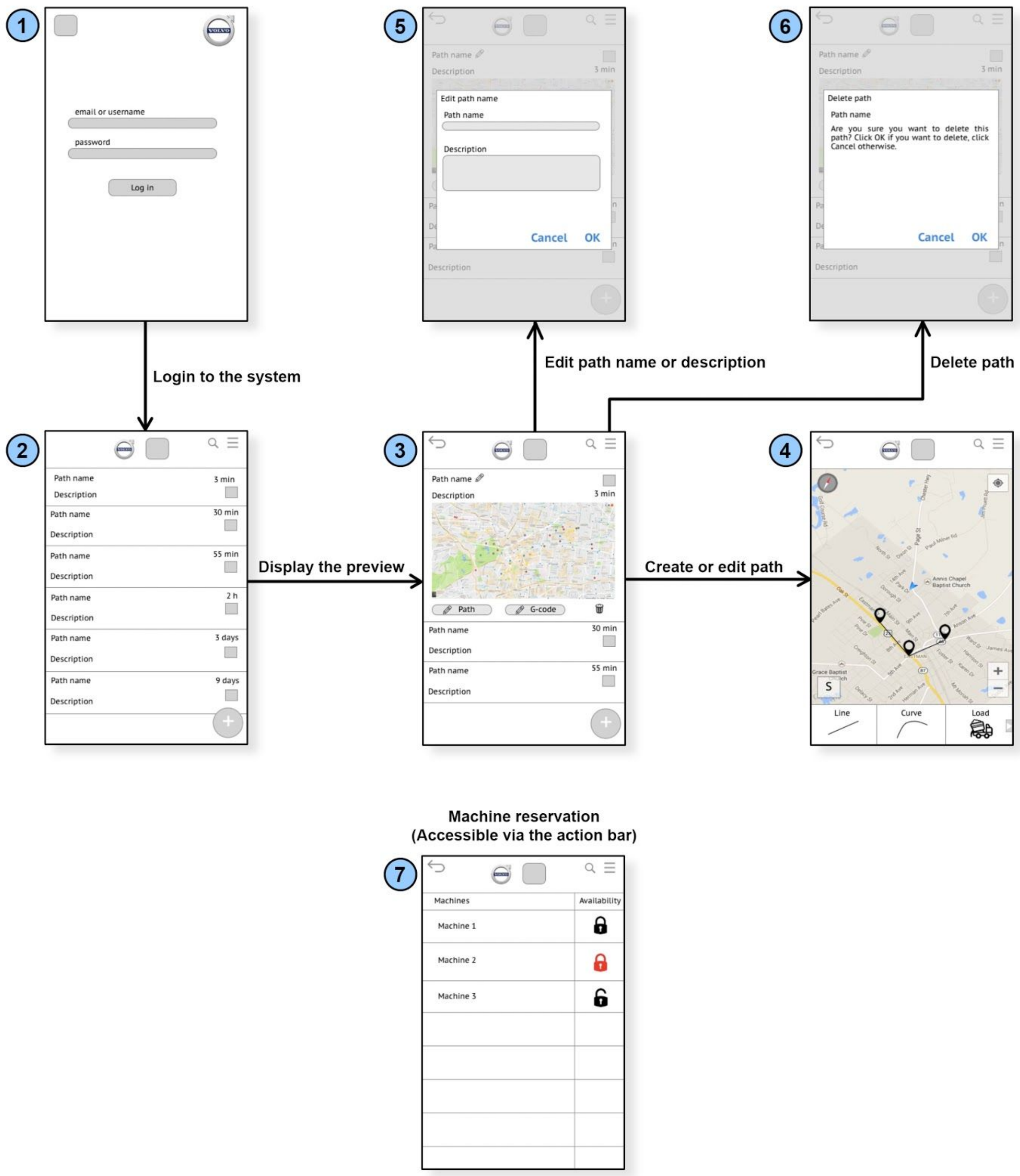


Figure 4: GUI design and mockup. It depicts the navigation flow between the GUI pages in mobile view: (1)Login, (2)Path list, (3)Path list preview, (4)Path view, (5)Edit dialog, (6)Delete dialog, (7)Reservation view.

On the first page of the GUI, the user will see the login screen where they are prompted to enter their email and password. The user will be authenticated if there exists a user with those credentials in the cloud database.

If the login attempt was successful the PathList view will appear. In this view, the user can browse through the already existing paths, where the most recently created path is located at the top of the list. Users can select a path in the list and a preview map of the selected path will appear.

This preview gives the user the option to edit the path name and description by pressing the pen icon beside the path's name. That action will open a dialog were the user can enter the new name or description. The user can also permanently delete the selected path by pressing on the trashcan icon in the preview, if said user is an admin. This action will open another dialog that asks for confirmation, where the user can either accept or cancel the operation.

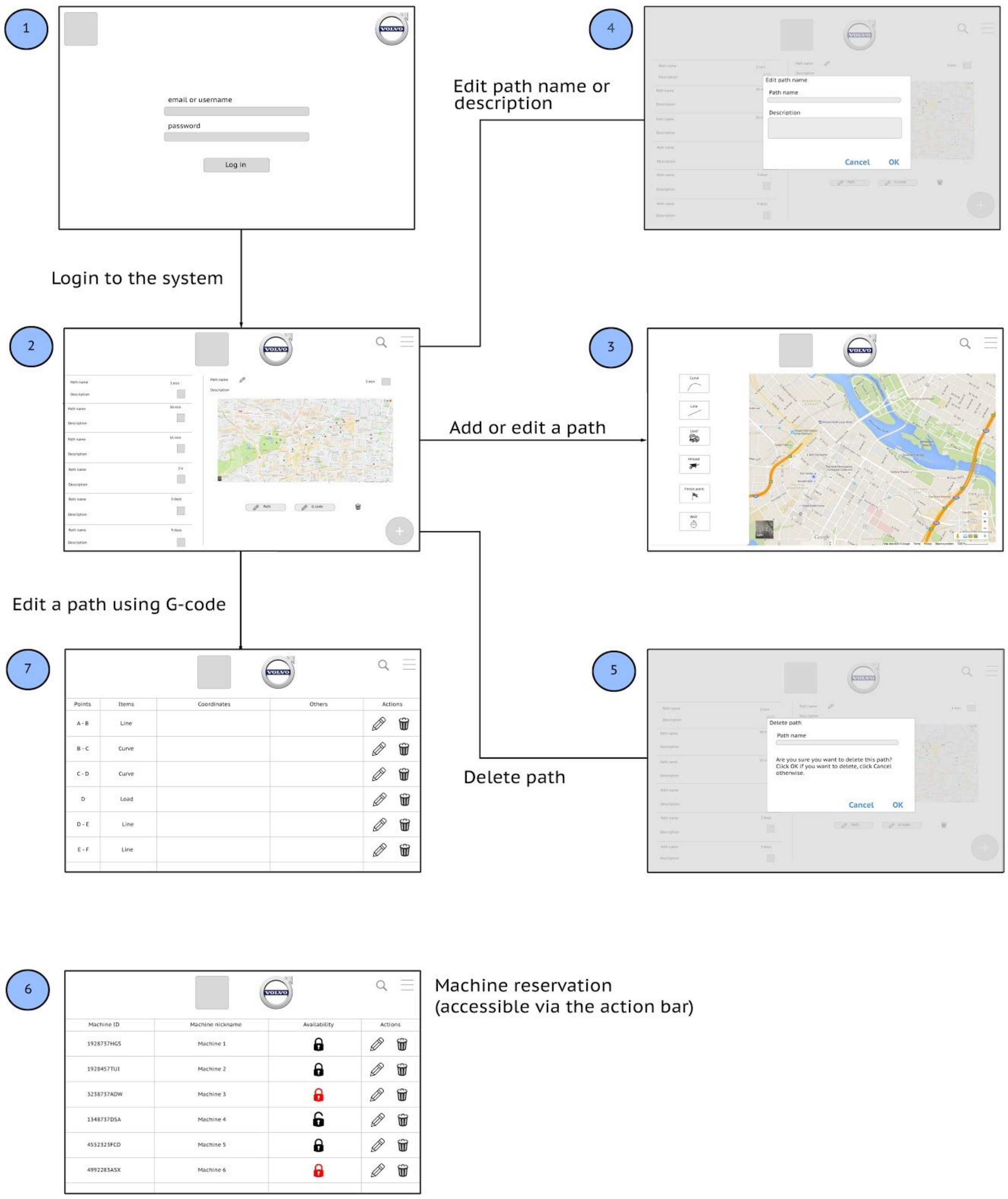
The user can edit points in the selected path by pressing the button labeled "path", this will change to the path view. From this page they will be able to view the entire path from a "bird's eye perspective", where the points in the path are presented on the map as icons, which are connected by a line. The map we use to display the path is provided via Google Maps. The user can select the point they wish to edit, which will open a dialog box where the new values can be entered.

The user can also create a new path by pressing on the plus sign in the pathlist view. After providing the path's name and description, the path view will appear. On the bottom of this page there is a row of buttons, which represent different types of points the user can add to the current path. The user can click on a position on the map, or choose to use the current position, to place a marker for where the next point will be. After the user presses on any on the buttons on the bottom, the location of this marker is used as an input to place the new point on the map. This way the user can walk along the path the autonomous vehicles will take and continuously add new points to the path as needed. When the user enters a finish point, the path is completed and can be uploaded to the database.

If the user is an operator (which is a subclass of the basic user) they have the ability to reserve and release machines. This activity is done in the reservation view, which is accessible from the action bar. In this GUI page the pool of allocatable machines is displayed in a list, where each machine has an identification number, nickname and status. The machine status can either be reserved or available, which is indicated by the lock symbol on the right. An open lock indicates that the machine is available, a clasped lock indicates that you have reserved that machine and a red clasped lock indicates that the machine is reserved by another operator.

The user can log out from the system at any time by accessing the action bar menu (three bars in the top right corner), and pressing logout. This action will take the user back to the login screen, and the user will be disconnected from the system. This means that the user will be unable to use any of the application's functionality until another login attempt is successful.

Graphical user interface for tablet



Points	Items	Coordinates	Others	Actions
A - B	Line			
B - C	Curve			
C - D	Curve			
D	Load			
D - E	Line			
E - F	Line			

Machine ID	Machine nickname	Availability	Actions
1928737HGS	Machine 1		
1928457TUI	Machine 2		
3238737ADW	Machine 3		
1348737DSA	Machine 4		
4552323FCD	Machine 5		
4992283ASX	Machine 6		

Figure 5: GUI design and mockup. It depicts the navigation flow between the GUI pages in tablet view: (1)Login, (2)Path list, (3)Path view, (4)Edit path dialog, (5)Delete dialog, (6)Reservation view, (7)Edit G-Code dialog.

This design (mockup of the actual application) was created to visualize what changes could be made if the application were to be executed in a tablet instead of a mobile phone. The only major change is the preview of the paths, which is happening in the same view as the main list, but instead of expanding below, it appears in the right. The rest of them are slightly modified versions of the GUI pages explained above, and as such there is no need for any additional description.

Appendix 1: Client feedback

Questions asked to the client and the answers we received during the meetings.

- What is your idea for the application?
  He gave us the document attached as appendix 3.

- What functionalities do you expect the application to have (use cases)?
  - Register?
  - Login?
  - etc.

  Answer: He expects to have a basic register and login, since they will be changed with the Volvo ones later on. He wants a list of paths and the possibility to edit those paths both with a map and with their G-code. He wants to reserve machines and different user types with different rights of use.

- If needed, do you have a programming language preference for the backend/server communication?
  Answer: No.

- How often would you like to have updates and/or meetings with the group?
  Answer: Weekly meeting / more emails if necessary.

- Which technology are we going to use for communication with the machinery: Bluetooth, WiFi…?
  Answer: We will not be connecting with the actual machines, so no need to worry about that.

- Does the application needs to know where the autonomous machines are located? Or is it a CoPilot job?
  Answer: Since we will not have any actual contact with the machines, this feature is not necessary.

- Do we need to calculate the time it would take the machine to complete the path designed or is that a “planner” job? (the person that plans the path).
  Answer: Not for the first prototype. Maybe later.

- What happens if two people using the CoPilot try to access the same autonomous machines? The operator claims them, so nobody else can command them, or is there a boss that assigns them to each operator?
  Answer: The operators have to be able to reserve machines, so they can be the only ones using them, until they decide to release them.

- Do we need a star or something to add favourite paths?
  Answer: Yes, he wants this feature.

- Do we need a description for the paths?
  Answer: Yes, he wants this feature.

- The path that is to be executed in the autonomous vehicles, is that in G-code or what?
  Answer: We do not really care. He has stated that every employer probably has an in-house one we do not know about, so there is no need for us to focus on this task.

- Do we need to be able to delete paths from firebase?
  Answer: No, he does not want to be able to delete them from the database, in case something happens. There has to be some mechanism to prevent weird things from happening. Only some kind of master user should have the authority to delete paths, in case they are needed.

- If a worker is in a quarry, do we have to worry about if he can use machines from others quarries?
  Answer: We don't have to bother with that, as it will come in another state of the product, and not in the prototype.

- How are the machines going to be identified?
  Answer: Unique serial number marked in the machine, and probably a nickname given by the quarry they are working in.

- Which Android version is the CoPilot tablet API based on?
  Answer: He does not know by heart, but it is not a problem we should be taking care of at the moment.

Appendix 2: Meeting notes

Notes from the client meetings, that are not answers to specific questions, but sorted by category.

General

- The acceptance testing will be in person after the holidays.
- For now, it is considered good enough if the Earth is flat when working within a 3 km radius.
- The machines have their own interpreted language, so G-code is only needed on the application side. It is meant to help the worker create the path.
- The machines should be recognized by their unique id (static, given by Volvo), and the nickname the new owners of the machine potentially give them (can be changed).
- Originally, G-code is just text. The current idea of reinterpreting a modern version of G-code with a better interface is great. This needs to be developed further so that it can be edited properly in mobile phones and tablets. The client suggested that we make multiple designs for this.

UI

- The login is great. They want it to be done by email and password, and they will change it later, so it is just for the prototype.
- The trash icon needs to be removed (the possibility to remove paths from the database), from all users except the admin. This means a new actor should be added to our diagrams, the admin.
    - After later discussion, it was concluded that there is no need for a new actor, since both the Planner and the Operator can be admins as well. We have added that possibility to the User classes.
- The first path shown in the list should be the last path edited or created, and with a preview of it.
    - The rest of the paths should follow below, without a preview.

Functionality

- They want the user to be able to see where they are in the map at the moment, and also change between regular map view and satellite view. They want the worker to be able to use their location to create points in the map, and also just clicking on the map should create a new point.

- The operator controls the machines they have reserved, and they can hand-over their machines to another operator, or release them for another operator to use.
- Both users can see the G-code and change it, but only the operator can run it on the machines.
- A halt-point should be added as a possible path action (including it to the scrollable menu on the maps).
    - A wait-point should also be added.
- Being able to view multiple already existing paths at once while constructing a new path, so that it is possible to see if they work good together. Another type of menu is needed for this.
- The possibility of adding new points between already exisiting points in the G-code version of the path should be considered.
- They mentioned it would be nice if the paths had decision points where the machines would have the choice of going in multiple different directions.

Appendix 3: Document supplied by Volvo.

1. Background.

The Volvo Group is one of the world’s leading manufacturers of trucks, buses, construction equipment and marine and industrial engines under the leading brands Volvo, Renault Trucks, Mack, UD Trucks, Eicher, SDLG, Terex Trucks, Prevost, Nova Bus, UD Bus, Sunwin Bus and Volvo Penta.

With Volvo Construction Equipment, one of the world’s leading providers of products and services to the construction industry under the brands Volvo, SDLG and Terex Trucks, you will be part of a global and diverse team of highly skilled professionals who work with passion, trust each other and embrace change to stay ahead. We make our customers win.

Right now there is a strong automation trend in the vehicle and machine business and all vehicle manufacturers are working in this field to build knowledge and develop products. Volvo Construction Equipment has also been working in this field for a long time and have developed several demonstrators , for instance, a wheel loader, an articulated hauler, a small dump truck and an excavator.

Figure 1: Left) A prototype of a small rigid hauler designed to be fully automated. Right) The SARPA hauler

In order to controlled and program a automated machine by a operator a programming language is needed. This enable a possibility to prepare the work prior to entering the working area.

In this work a programming tool will be developed for the Volvo CoPilot so a operator can command the machine with G-code STEP-NC Canned_cycle commonly used in computer numerical control (CNC) machines. Obviously there is not a perfect match with the language and a wheeled machines, so a proposal for how this will be managed is welcomed.

For the operator the workflow will have high similarity to “Vägstakning”.The track the machines will follow are built up of straight (raksträcka), curve (Kurvradie) and clothoid(Klotoid) roadsegments. How will clothoid be managed in this language? All the movements the machine will have must be described in a map reference system.

Are RT_90 good or is there a much better system for international use? How will the operator remotely interact with the machine and down load the code between different CoPilots? When the path is created the tool must be able to simulate how the machine probably will behave, similar to a Computer-aided_manufacturing (CAM) tool. In this way the operator can detect if the program will behave the way it is supposed.

2. Objectives

Project scope:

- The scope of this project work is to implement a G-code program system for autonomous constructions equipment and propose possibility and its suitability to use that system. The evaluation should be carried by implement a system in a CoPilot and test the program on a simulated machine.

Deliverables:

- A report summarizing the evaluated systems both from simulations and from test drives with the CoPilot.
- Code to CoPilot.

3. Time plan

The development team has filled this information in their Project Plan, section named Time Plan as well. For more data regarding this topic, please check that document.