Decision Tree Engine App

Design Document

2014 - 12 - 04

Group 3:

Valerio Lucantonio, Nathan Chape, Tamara Dancheva,
Anna Enbom, Eric Johansson, Niklas Sjöqvist

1 Background

Decision Tree App Engine is the project assigned to our group, group 3, for the course *Software Engineering 2: Project Teamwork* (DVA313) at Mälardalen University. The purpose of this project is to create an application which allows a user to answer different surveys and to calculate the economic viability of a potential investment. After each survey the user will receive a summary with statistics on questions that "approve" an investment, how many that "disapprove" and how many that still are "uncertain". There will also be an administrator application which will be able to add remove and modifying surveys that will be available for the end user application.

This document will present descriptions of the administrator application, the end user application, the communication between the applications, the architecture of the applications, the software designs and the graphical user interfaces.

2 High Level Description

The overall goal of this project is to implement a system comprised of two distinct parts:

1. an *administrator application*
2. an *end user application*

Each of which has its own requirements and functionalities.

2.1 Administrator Application

The administrator application is comprised of two main sub-sections. The first of which is the *survey panel*. Here, the administrator creates the rudimentary parts of the survey. These include:

a. *Name*: The administrator assigns a name to the survey. Ideally, this should also represent the relative category for which the survey will be designed. eg. *Automation, Industry and Production, Commercial...*etc.
b. *Language(s)*: Here the administrator selects the default language for the survey as well as any additional languages the administrator wishes to include.
c. *Description:* a short description of the survey

The second sub-section is the *question panel*. It is here the administrator designs the *decision tree* and formulates the corresponding questions. This is done by creating and connecting the *nodes* of the tree. Each node represents a question and each subsequent branching node represents a possible answer. These branches, or answers, as you will, can also be assigned a value which is then used to calculate the overall result. This result is then used to represent the

favourable and negative attributes of an end users' decisions. This will be made clearer further in the following section.

Finally, once the administrator has finished designing the tree he or she can then save the surveys in a cloud service that will be explained in furthers sections.

2.2 End User App

The end user app is allows a user to use the surveys created in the administrator application. Upon purchase, the app should contain all currently available surveys. It should also have the capacity to receive new or updated surveys, if and when they are made available by the administrator.

The app itself should contain several functionalities, users should be able to:

a. select a survey among the available in the cloud db eg. *Automation, Industry and Production, Commercial*
b. answer the chosen survey
c. generate a summary of the user preferences

Secondary functionality includes:

c. the option to select, when available, additional information about a question in a survey
d. a typical *about* function which displays information about the app

Once a user completes a survey, a summary should be displayed. It should contain the percentages of both the positive and negative results and a brief text summarizing the result. This result should give the user a clearer picture over the requirements that they may or may not meet over a certain domain (ie. the type of survey). The user should also have the option to save the result and/or send it to any desired email address.

3 System Overview

The hardware in mind is to use a PC for the administrator web based application to generate the surveys, which in turn will be saved on an online database (Figure 4.1). The surveys will be interpreted on Android and iOS smartphones and will not upload any information to the database. Everything that's done on the smartphone will stay local. On the other hand, the smartphone will be able to download the surveys from the database. The internal structure of both applications will be explained in detail in the following two sections.
The current admin application (Appendix 1) which is called *Survey Builder* is a Windows desktop application developed in C#.

4 Software Architecture

4.1 Overview and rationale

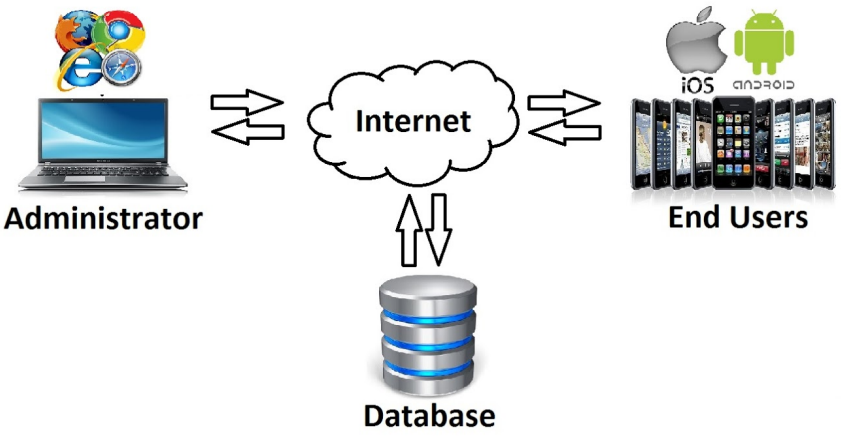

Figure 4.1 Overall architecture

The goal of this project is to develop a solution that enables an administrator to create surveys that will be available for end-users to take by downloading an Android or iOS application. The system previously developed was a desktop application that allowed the administrator to create surveys and after that to deploy an end user application and share it to the (IOS and Android) markets. Obviously a solution like that is really demanding to maintain and it forces users to download a new application *each time* data in it changes which of course is convenient for the user. For this reason we implemented a solution that allows the two applications to share an online database. This allows the end user application to be deployed once while also allowing the administrator to easily upload new surveys to the database without the need to redeploy the end user app.

Since we had some strict requirements on the technologies specified by the customer we have been forced to use HTML5 for the GUIs and to deploy the final applications for IOS and Android. For this reason we chose to use the same technologies for both the applications: javascript for the business logic, HTML5 and CSS3 for the GUIs, and to share the same development framework backbone.js.

4.2 Decomposition Of The System

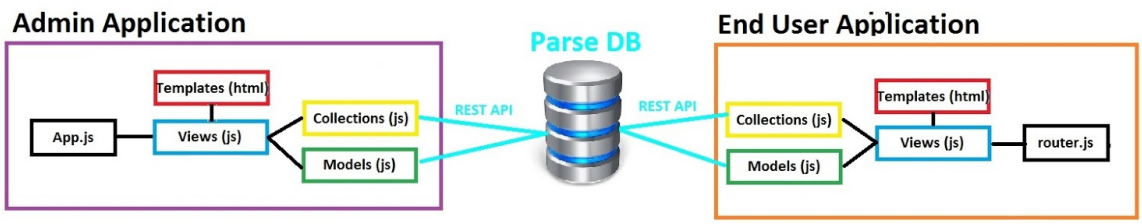

Figure 4.2 General view of the systems components and internal organizzation

A bottom-up elaboration of the system components can be seen in Figure 4.2. We can decompose the application into 3 different units: end user application, administrator application and an online database that is common for the 2 applications. The administrator application is a web based application accessible by the admin from a web browser. The end user application is a mobile application implemented using javascript and deployed for IOS and Android using Cordova.js. Internally the two application are organized following the pattern offered by backbone.js. Backbone.js is a client side MVC Javascript library that allows to structure the application in different packages. The 5 main packages that backbone offers are models, collections, templates, views, and routers:

1. *Models* - This package is responsible for specifying the structure for all the data used in the applications. It allows communication with the database via their methods.
2. *Collections* - A set of models are linked as well with the database.
3. *Templates* - html template files that are dynamically managed by the views package.
4. *Views* - The views are responsible for loading templates and filling them with the data from the models (data binding). Views can respond to events triggered by the user selecting/tapping the elements in the dom.
5. *Router* - manages the views, provides URL routing/history capabilities functionality.

Following this pattern the application is modularized in a way that fits the requirements of the project. This also makes easier for us to produce a more readable code and document it accordingly without too much difficulty.

4.3 Data Persistence

In order to make a decision about which approach we should use to solve this issue, we considered mainly three scenarios. The previous application developed with Xamarin forms in C# used a special type of output file with the extension *.str* to persist the data about the survey created by the administrator. Considering that one of the requirements is that the user should be able to fill the survey offline, this approach of distributing the file is satisfied. However the main disadvantage is that it doesn't support future modifications and additions in a flexible manner. The administrator would have to redeploy the application whenever there is a change forcing the user to either download the application again or update.

The second approach was to use an online database to persist the data, in which case we avoid redeployment of the application and we facilitate modifications and additions. However since offline access to the survey is a hard requirement, we combined and used the strengths of each approach to design a dynamic solution that fits all the requirements and is convenient for both parties (administrator and end-users).

The final solution is to use an online database, extract the content of each table using REST API provided by Parse dbms and create for each table in the DB a Collection that is saved locally on the end user's device each time application is accessed. In this way according to the requirements, final users can instantiate the application when they have internet connection and then answer surveys offline.

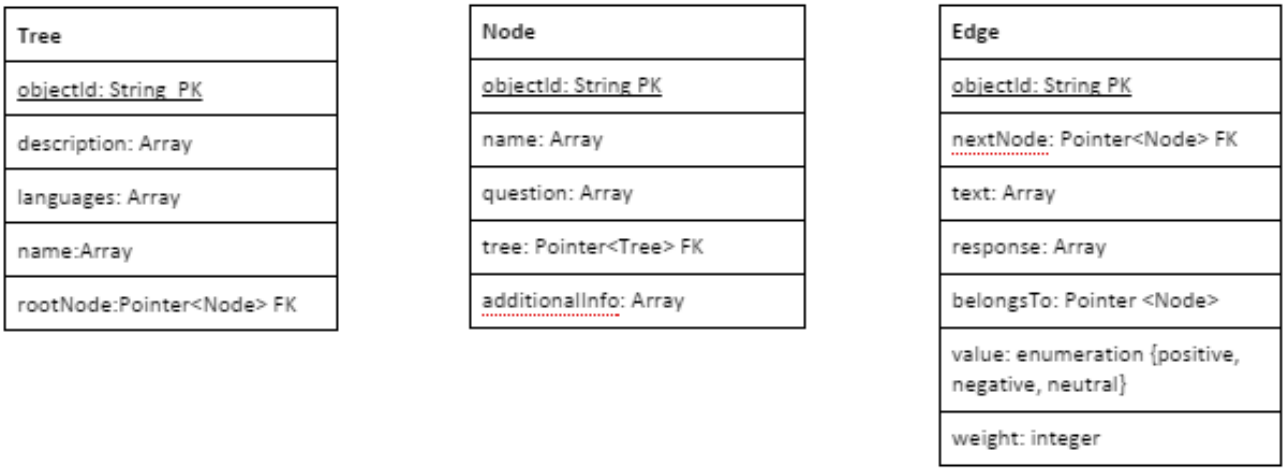

Fig 4.3 Database schema

Database schema

- Array: type that allows to store languages strings (in a json format i.e {english: survey}) directly in the object without using a dedicated table
- Pointer<Table>: Pointer to a an entry in the specified table

5 Software Design

For the two applications we have designed both static and dynamic models. For the end user application we provided a class diagram with all the entities defined and a navigation map to explain the flow that the application will have. For the admin application we provided a static representation, always through a class diagram, but not a dynamic representation because we think it's quite easy to understand the pattern used among the system. As stated previously both applications will be developed following the backbone.js pattern explained in the previous section. The applications will have the same pattern with four main packages: *views*, *templates*, *collections* and *models*. The **views** package contains different javascript classes, each responsible of loading the correct HTML page contained in the **template** package and to respond to all the events triggered by the user in the application. Each view, except for the statics, is associated with one or more **collections** that performs retrieving of data from the database through REST API. Each collection of the system is associated with a table in the database with a pattern 1 to 1, and represent a list of objects of the form specified in its model. A **model** specifies the structure of a single object in a db table or in a collection and each model is associated with a table in the database with a pattern 1 to 1 as well as for the collections.

5.1 Administrator Application Static View

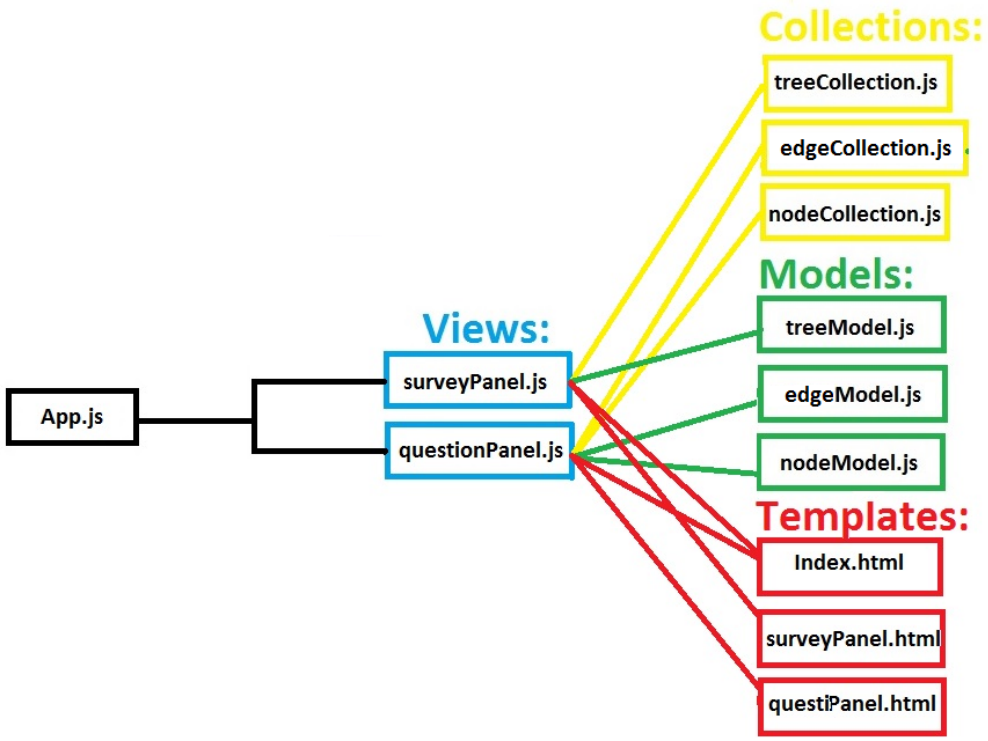

Figure 5.1: Administrator application static view

The admin application starts in **App.js** which is responsible for initializing the database connection and routing the application to the first view. Since the application will have one html page with two tabs, as you will see in the GUI section, we divided the business logic in two classes (views). Following is a description of each of the package.

View Package: All the classes in this package contains two basic methods: **initialize:** it's the method where the view is initialized and whether data will be fetched through a collection or a model., **render:** is where the html page corresponding to the view is loaded and also where the models previously fetched will be displayed.The classes contained in the Views package are:

- **surveyPanel.js:** the class responsible for loading the panel where the admin can add, modify and delete surveys and to react to the user interaction. As you will understand in the GUI description of the survey panel, in this view there is, for each form, one method for interaction (save/modify/delete) with the data. Also, in addition to the basic methods there are the following methods:
    - **newTree**: create a new tree
    - **modifyTree:** for modifying an existing tree

- **deleteTree:** for deleting an existing tree
- **addLanguage:** for adding a new language to the tree
- **deleteLanguage:** for deleting an existing language from the tree
- **addTreeInformation:** for adding the name and the description
- **updateTreeInformation:** for updating the name and the description.

SurveyPanel.js includes TreeModels.js that provides methods to create and update a single tree whilst TreeCollection.js provides methods to retrieve all the available trees in the system. They are contained both in tab.js.

- **questionPanel.js:** controls the question panel and the events defined on it. In the question panel there are forms to create, modify and delete questions and related answers. Hence, in addition to the basic methods, there are at least the following methods:
  - **getBelongingTree:** returns the tree to which a node belongs
  - **addNode:** to create a node
  - **modifyNode:** to modify a node
  - **deleteNode:** to delete a node
  - **addNodeInformations:** to add "name"," question" and "additional information" to a node
  - **editNodeInformations:** to modify "name"," question" and "additional information" belonging to a node
  - **addAnswerToNode:** to add one answer to the node
  - **editAnswer:** to edit the information concerning an answer

  QuestionPanel.js includes TreeModels.js which provides methods to retrieve a single tree. NodeModel.js and EdgeModel.js to add, modify or remove a node or edge to/from the db, and NodeCollection.js and EdgeCollection.js that provide methods to retrieve all the available nodes and edges in the db.

Actually the 2 panels are not two different js, because we weren't be able to divide the two implementations, but we divided them logically and for the purpose of this document they can be divided logically in this elements.

Collection Package: there are three different collections

- **TreeCollection.js:** it's the collection linked to tree table in the database. It has the methods to get all the surveys (or trees) in the database. The model of the tree in TreeCollection is TreeModel.
- **NodeCollection.js:** it's the collection linked to node table in the database. It has the methods to get all the questions (or nodes) in the database. The model of the tree in NodeCollection is NodeModel.
- **EdgeCollection.js:** it's the collection linked to edge table in the database. It has the methods to get all the answers (or edges) in the database. The model of the edge in EdgeCollection is EdgeModel.

Model Package: contains all the models for the application. Each model specifies the type of data we use in the application. In particular, each model will have an attribute for each column in a corresponding table in the database. Basically, for each entry in the collection fetched from the database, a model of a corresponding type will be created: i.e. when treeCollection is fetched, for each entry of the collection there will be a TreeModel created where its attributes are the attributes of the single entry. Each of the models provide different methods to create, modify and delete an object in the corresponding table of the database. We have three models:

- **TreeModel.js**: represents a tree (or survey). It has the following attributes:
  - **name:** a string containing the name of the tree.
  - **description:** the text containing the description of the tree.
  - **languages**: the languages in which the system is available.
  - **rootNode:** the reference to the first node of the tree.
  - **objectId:** the id of the tree.
- **NodeModel.js:** represents a node (or question). It has the following attributes:
  - **name:** a string representing the name of the question.
  - **question:** the actual text of the question.
  - **additional information:** text containing explanations useful for the user to answer the question.
- **EdgeModel.js:** represents an edge (or answer). It has the following attributes:
  - **text**: the text of the answer
  - **response:** the text that will be shown in the summary if the answer is chosen from the user
  - **value:** if the answer is positive, negative or neutral in relation to the question
  - **nextNode:** the reference to the next question, if the user chooses this answer for the current question. If it is null, it means that the user is at a leafnode
  - **belongsTo:** the reference to the node to which the answer belongs.

5.2 Administrator Application Dynamic View

We don't provide a dynamic view of the system since we adhere to the backbone pattern. Moreover, the application is a single page so there isn't too much interaction to show. Basically, the navigation is event driven: when the admin wants to add, modify or remove trees, nodes or edges, he fills data in the appropriate form, and implicitly triggers a method in the view in which the form belongs (so for survey's forms surveyPanel and for nodes and edges forms, the questionPanel) and the view calls the appropriate model (or collection) to add, modify or delete an element. When the model is added, modified or removed the view is rendered again to showing the updated state of the database.

5.3 End-User application Static view

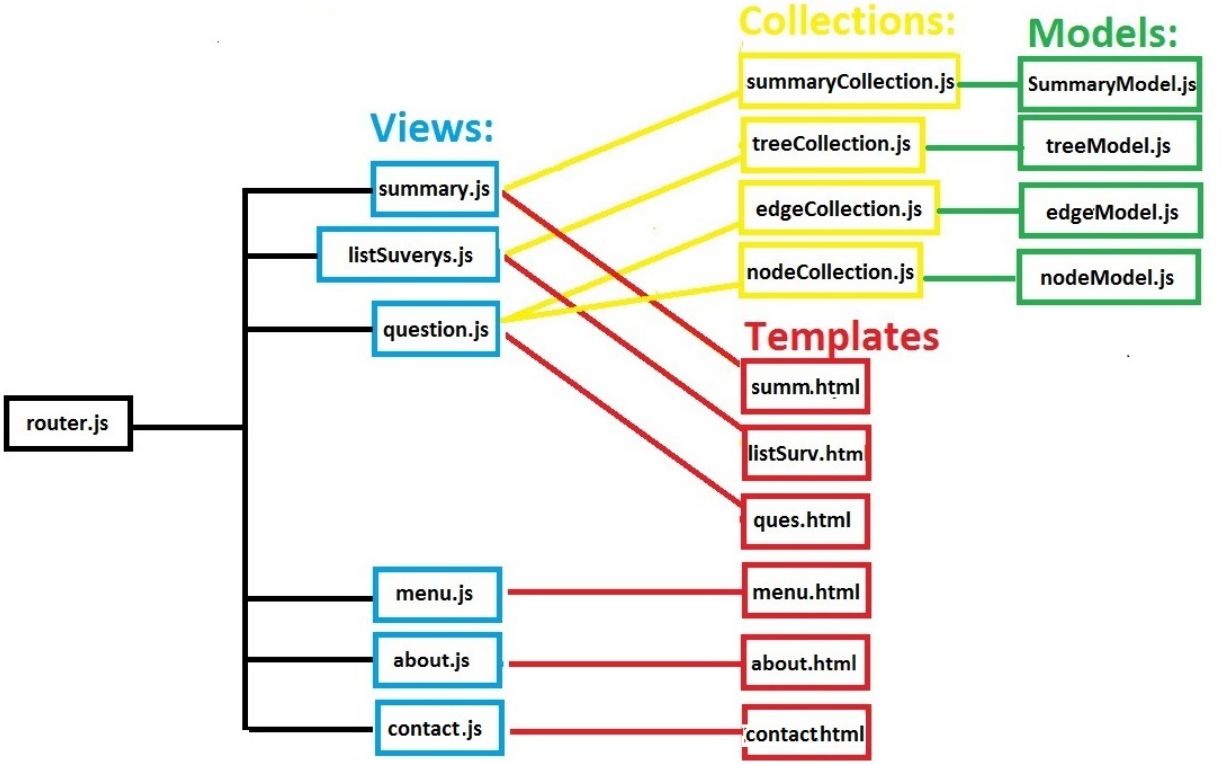

Figure 5.2

The end user application starts from a **router** that will be responsabile for loading the first view in the system and then for redirecting it to the selected view based on user interaction. The pattern is easy: there is a default view (in this case menu) that will be loaded when the application starts to run, and then based on the input of the user the router will be in charge of changing pages. The particularity of this application is that collections and methods have read-only methods, since this application doesn't need to update the content of the database Following is a precise explanation of the individual components in the systems:

Views Package: in this package there are 7 different classes. All the classes in this package contains at least 3 methods: **initialize:** where the view is initialized and where data will be fetched through collection or models, **render:** where the html page corresponding to the view is loaded and where the models previously fetched will be displayed, **changeView:** is the method called when the user "taps" an object on the screen that fetches another page. In

addition to these two methods, each view will have a method for each of the different actions the user can do in the page currently loaded: i.e. as you can see in the following GUI section for menu view there will be 4 buttons to go in different pages: each button will corresponds to a function in the menu.js as method that in this case, redirects the application to another page. The classes contained in the Views package are:

- **menu.js:** is a static view responsible for showing the menu page, this is the first screen that will be loaded when the application starts. menu.js is responsible for loading menu.html and for responding to all the events that a user can trigger from it. As you will see in the graphical interface section and in the navigation map, from menu it is possible to navigate to three other static views (about, tips and subscription) and also to the list of the survey.
- **about.js:** will load the page containing the static contact information. From this view it's only possible to go back to the menu view.
- **contact.js:** allows a user to send questions to the administrator, with email and personal information.
- **listSurvey.js:** is the view in charge of retrieving (through treeCollection.js) and lists all the surveys (or trees) contained in the database. This view loads the html page containing an entry for each survey in the database. In this page the user will choose the survey that he/she wants to answer.
- **question.js:** is the view where the content of each question and relative answers in the survey are iteratively loaded. Basically, through the linked structure we defined in the database it is possible to sequentially accede questions in each survey: starting from a root node entry contained as the entry for the "tree" table, it is possible to identify the first question in the system. Answering that question, the user will be redirected to a new question based on the answer he/she gave. This will be done iteratively for each question in the database until answering a question brings the user to get the summary of the survey just answered. In this view it will use a NodeCollections.js and EdgeCollection.js instance to fetch all the questions and answer from the db (that the user has chosen).

  NB. the first question will be always to choose the language of the survey, but to be more slim in the text we omitted the class language.js that works exactly like a question but once at the beginning of the survey to answer.
- **summary.js:** is the view responsible for showing the summary at the end of the survey. It loads the summary.html and fills it with the information obtained while the user was answering the survey. This class uses an instance of surveyModel.js as a model for summarizing the data.

Template Package: contains all the html pages of the system. As said before each view.js requires a html template and in all this html pages are in the template package.

Collection Package: contains all the collections (or list of models). There are 4 collections:

- **TreeCollection.js:** it's the collection linked to tree table in the database. It has the methods to get all the surveys (or trees) in the database. The model of the tree in TreeCollection is TreeModel.
- **NodeCollection.js:** it's the collection linked to node table in the database. It has the methods to get all the questions (or nodes) in the database. The model of the tree in NodeCollection is NodeModel.
- **EdgeCollection.js:** it's the collection linked to edge table in the database. It has the methods to get all the answers (or edges) in the database. The model of the edge in EdgeCollection is EdgeModel.
- **SummaryCollection.js:** it's the collection or list of all the summaryModels created while the user is answering one tree. It contains as much as models as the questions answered by the user.

Model Package: contains all the models of the application. Each model specifies the kind of data we use in the application. In particular, each model will have an attribute for each column in the corresponding table in the database. In this case the model package doesn't communicate with the database since the connection is established once and the data is retrieved with Collections.js. Basically, for each entry in the collection that is fetched from the database a model of corresponding type will be created: i.e. when treeCollection is fetched, for each entry of the collection will be created a TreeModel where the attributes are the attributes of the single entry. We have four models:

- **TreeModel.js**: represents a tree (or survey). It has as attributes:
    - **name:** a string containing the name of the tree.
    - **description:** the text containing the description of the tree.
    - **languages**: the languages in which the system is available.
    - **rootNode:** the reference to the first node of the tree.
    - **objectId:** the id of the tree.
- **NodeModel.js:** represents a node (or question). It has the following attributes
    - **name:** a string for the name of the question.
    - **question:** the actual text of the question.
    - **additional information:** text containing explanations useful for the user to answer the question.
- **EdgeModel.js:** represents an edge (or answer). It has the following attributes:
    - **text**: the text of the answer
    - **response:** the text that will be shown in the summary if the answer is chosen from the user
    - **value:** if the answer is positive, negative or neutral relative to the question

- **weight:** a weight associated to the question.
- **nextNode:** the reference to the next question, if the user chooses this answer for the current question. If it is null, it means that the user is at a leafnode
- **belongsTo:** the reference to the node to which the answer belongs.

- **SummaryModel.js:** represents an entry of the summary that should be produced after answering the tree. It has as attributes:
  - **question:** the text of the question answered
  - **answer:** what the user answered
  - **response:** the correspondent "response" field of the answer
  - **value:** the correspondent "value" field of the answer

5.4 End-User application dynamic view

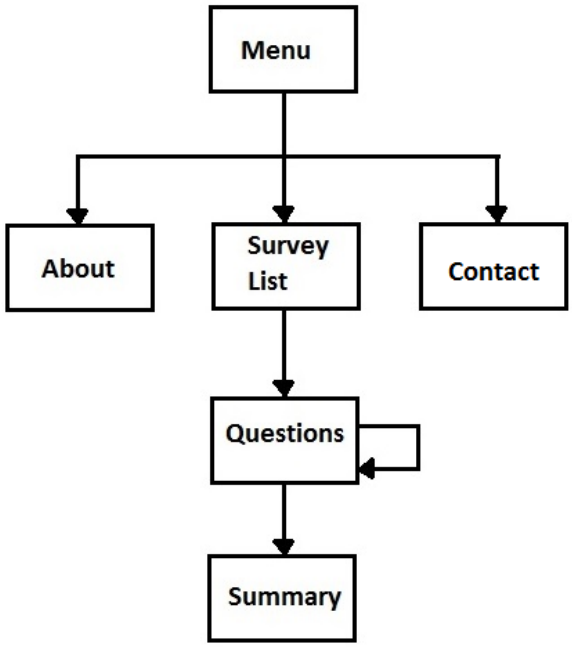

Figure 5.4 End-User application dynamic view

Initially, the first thing the end user application does upon starting, is to fetch from the database (via the router) all the surveys through the TreeCollection.js, NodeCollection.js and EdgeCollection.js. Following which the application loads the first page **menu**. Then as shown in the sketches below, the user can choose to select either the **list survey screen** or **contact screen** or **about screen**. **About** contains information about the app, **contact** allows the user to send a request with text to the administrator. If the user taps on **List Survey**, the application

loads the page with the list of all the surveys contained in the collections fetched before. Tapping one survey, questions start: the first page loaded will be **question** and the first question will be to choose a language from those the available for the survey. Upon selection, the application will redirect to the **question** page where the first question is the one contained in the rootNode field of the selected tree. From now on, a user answering a question will be redirected to the question contained in nextNode of the answer selected (so the redirect is to the same page changing the content). For each question the user answers, it will create a SurveyModel.js containing the information related to user answers, and each of the models will be attached to a global SurveyCollection.js. These iterations end when the user gives an answer where the nextNode field is null. So the application goes to the **summary** screen where it shows all the SurveyModels contained in the global SurveyCollection. This view asks the user if he wants to have this summary sent by email, and from there it will be possible to come back to the menu.

6 Graphical User Interface

6.1 Graphical User Interface - End-user application

Below follows the applications different views together with a description of each views functionality, the critique it received during user testing and possible improvements that can be made. The test description and individual feedback can be found in the Test specification document, appendix 1.

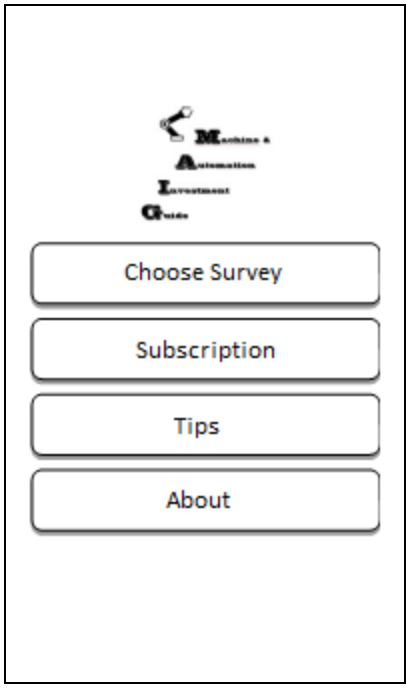

Figure 6.1 Menu

Figure 6.2 Survey List

Functionality

- Access the list of decision trees
- Access the payment site (not to be implemented)
- Button for Tips is supposed to be Contact to email the app administrator
- About Button, has no specified functionality other than to show text(undefined by client)

After selecting “Choose Survey” a new view is presented where all the decision trees are listed. By selecting one you begin traversing it in the form of a survey.

Improvement areas

Several users have complained about the cluttered nature of the logo and that the text within is too small to be readable on a phone, this applies to all views. They have also asked if this is an established logo or a new individual logo for the application. The client should therefore reflect about how he wants to brand the application.

So far, all test subjects have questioned the use of a subscription system which indicates that this sort of payment system might be hard to cater to the market.

One tester also asked if it was possible to access and view reports from trees you previously have traversed or if it was only limited to email. This might be a feature to take into consideration.

Figure 6.3 Select Language

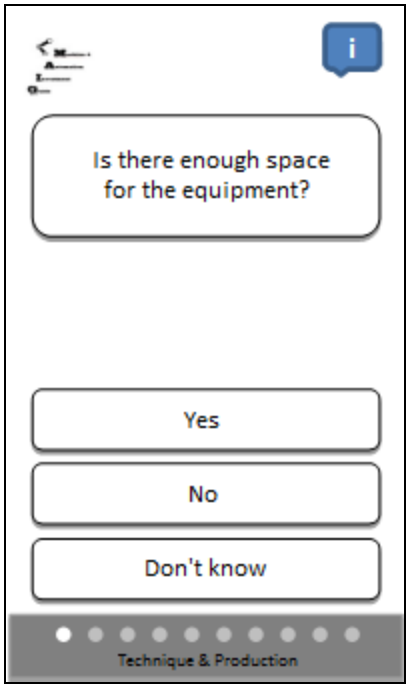

Figure 6.4 Questions

Functionality

When you select a tree to traverse you will first be asked to select a language to view it in. After that the tree will be presented as a series of questions which you'll be able to navigate by swiping left and right or by answering the current question to automaticly go to the next one.

Improvement areas

When it comes to the Questions view the testers tried to click on the dots at the low base line instead of swiping in an attempt faster navigate through the tree. This is a feature that possibly could increase user friendliness and should be taken into consideration when moving on with the project.

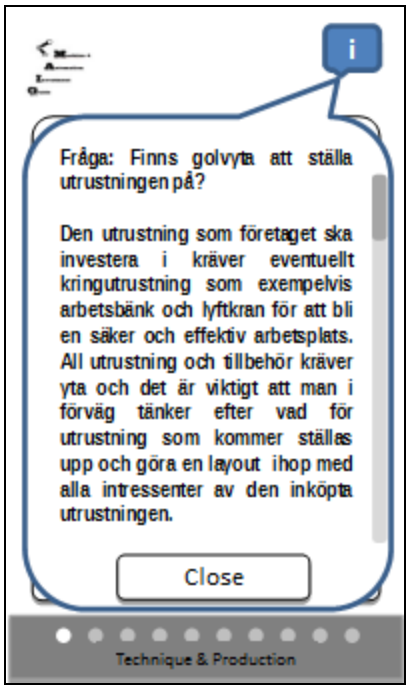

Figure 6.5 Questions (info pop-up)

Functionality

By pressing the (i) button at the top of the screen in the right corner, a popup will appear containing more extensive information about the current question.

Improvement areas

Several users found that the information given was too extensive and deterred them from using the feature, redefining this functionality might be necessary.

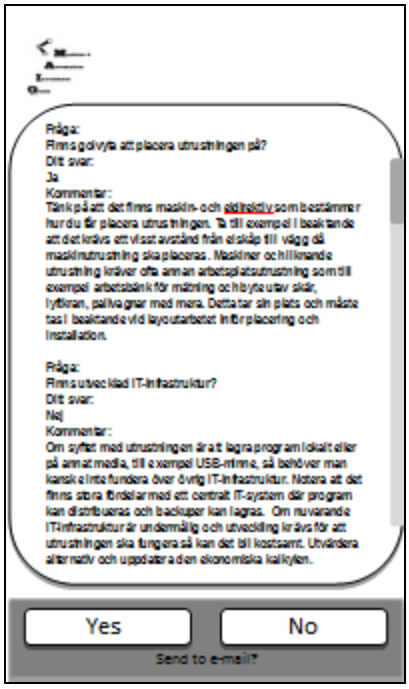

Figure 6.6 Summary

Functionality

At the end of the decision tree a preview of the document will be generated and the option to email the actual report will be presented. If the user selects to not email the report he or she will return back to the main menu if “Yes” the application will send an intent to the standard email client in the phone and attach the report automatically.

Improvement areas

Some users were surprised by the sudden mention of a report, they expected a final result view and not a generated document since no information of such a thing was given before this view in the application.

Some also reflected at this point that each tree should begin with a description of the purpose and result of the tree before they begin to traverse it.

In the Summary view the testers tried to zoom since the text was so small it barely could be read on a phone screen. If the intuitive reaction from the user is to zoom then a good improvement should be to either increase to text size by default or to include zoom functionality in this view.

Some also completely missed the small “Send to e-mail” text at the bottom of the view and once it was pointed out to them they asked what and where the email would be sent. This is part of the application which functionality is essential and is therefore strongly recommended to be redesigned to clarify the above mentioned topics.

6.2 Graphical User Interface - Admin application

Figure A1 and A2 in the Appendix 1 show the current admin application. The customer that will use the admin application, Andreas, likes the GUI design of the current application. Therefore, we will in general stick to its graphical design but make some adaptations.

The tab layout will be kept, but instead of three tabs as in the current applications there will be two tabs: “Survey” and “Question”.

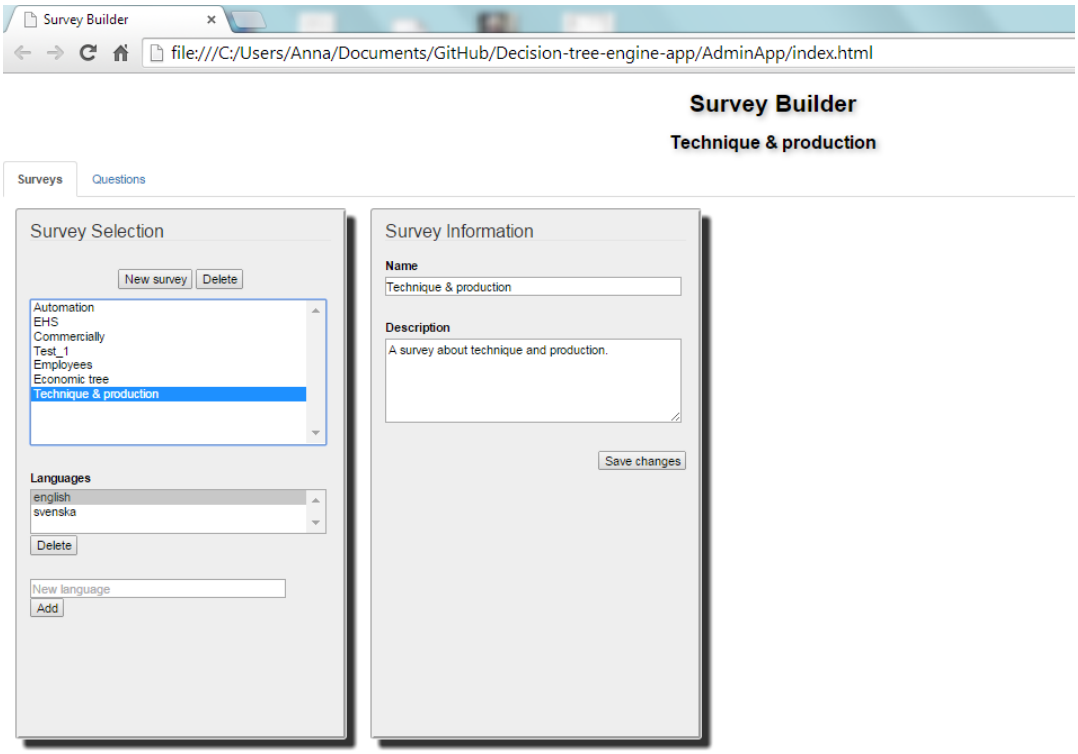

Figure 5.1. Admin application GUI – Survey tab.

Figure 5.1 shows the final graphical interface of the Survey tab.

In the left panel, the user can add a new survey, delete a survey and make language settings. All surveys are displayed in the list and the current survey is marked, the name of the current survey is also displayed on the top of the page. Below the surveys list is a list with the languages of the survey . The selected language is the one that the name and description will be saved in. Below the list is a button to delete the selected language. To add a new language, the user writes the name of the language in the text field in the bottom and clicks the Add button.

In the right panel, the user sets or changes the name and the description of the surveys. As mentioned, what language the name and description are written in is set in the language list.

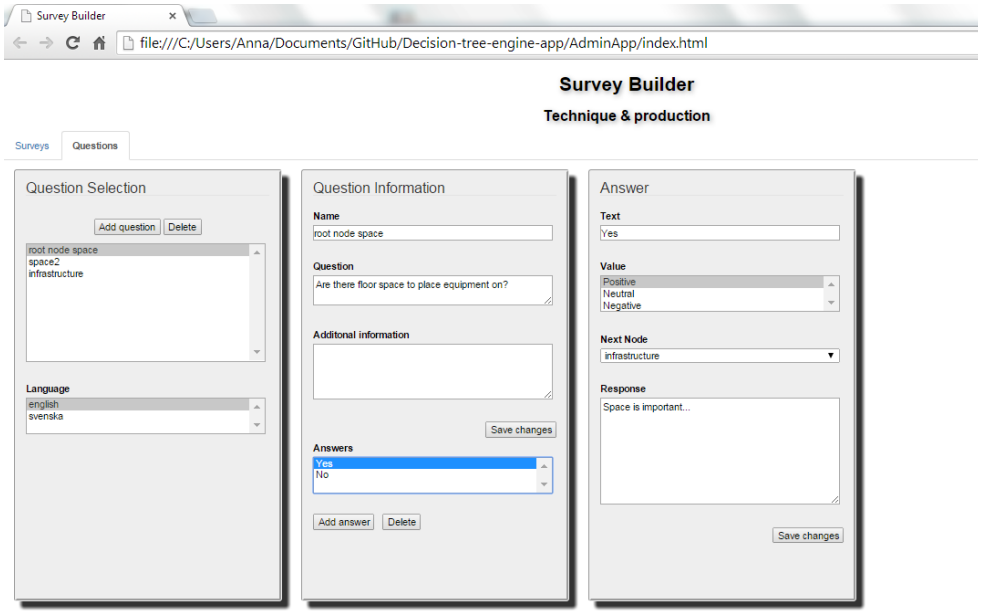

Figure 5.2. Admin application GUI – Questions tab.

Figure 5.2 shows the final graphical interface of the Question tab.

The left panel contains all the nodes of the currently selected survey. The buttons are to add a question and to delete an existing one. In the list of languages the user marks the language that the information will be saved in.

The panel in the center contains text fields for writing the name, question and additional information of the question. Below is a list of the answers connected to the question. The user clicks the button Add answer to add a new answer and Delete to remove the currently selected answer. When a answer is selected, its information is displayed in the right panel.

The panel to the right contains controls to set information about the currently selected answer. The text fields are for writing name and description. The list is for choosing the value of the edge and the drop down list contains all questions of the tree, to set the next question that the answer will lead to.

Figure 5.3 and 5.4 show our initial sketch of the admin application. The content and layout is basically the same, but some alterations has been made for simplicity and more usability.

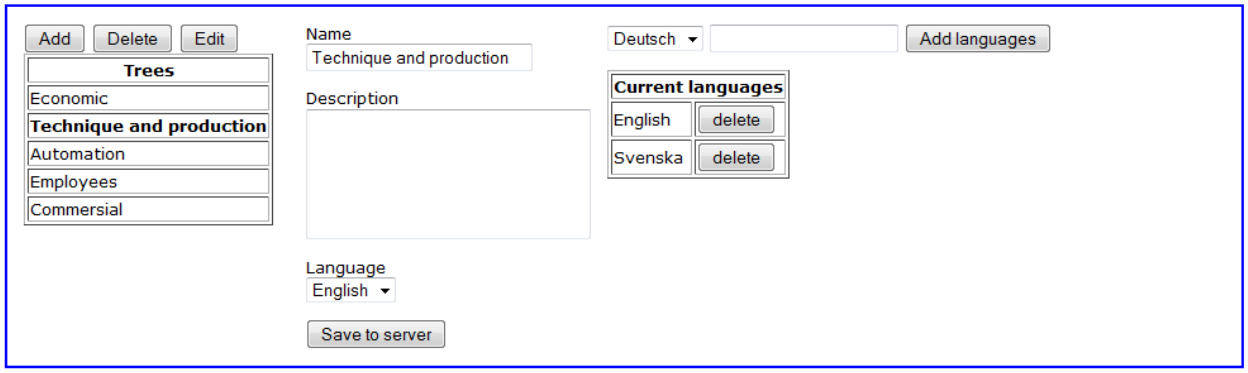

Figure 5.3. Initial sketch of Admin application GUI – Survey tab.

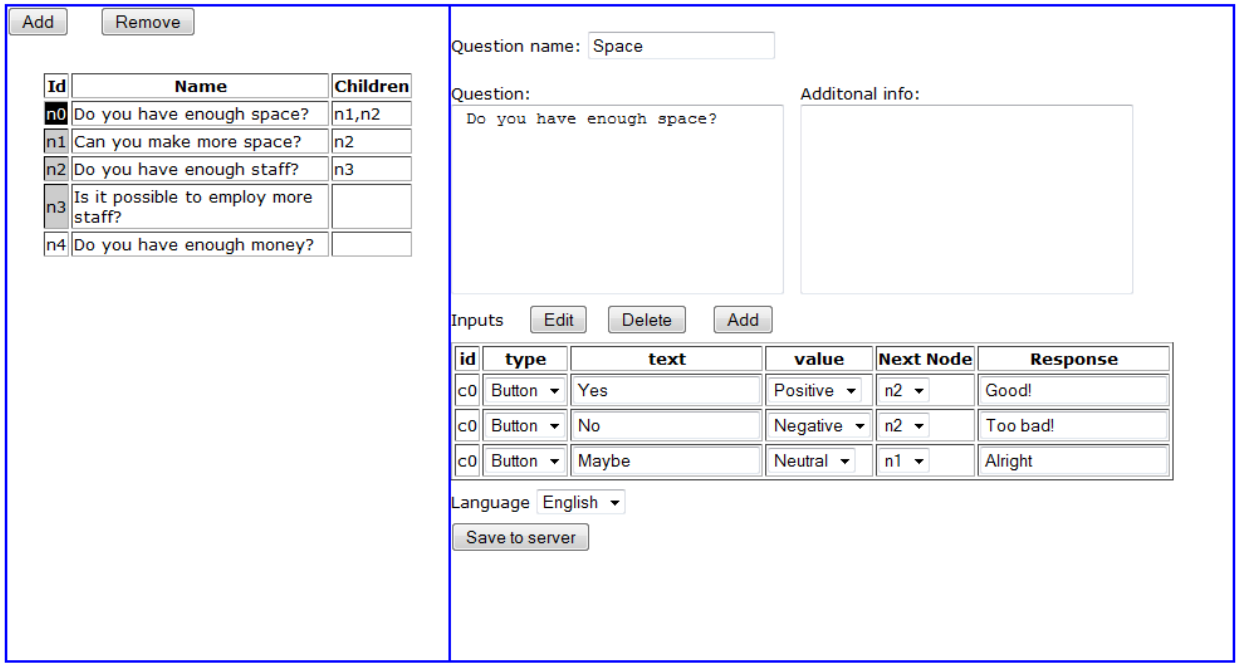

Figure 5.4. Initial sketch of Admin application GUI – Questions tab.

References

[1] “Apache Cordova Guide Overview”. Retrieved 2014-12-4.
http://cordova.apache.org/docs/en/4.0.0/guide_overview_index.md.html#Overview
[2] “Ratchet Getting Started”. Retrieved 2014-12-4.
http://goratchet.com/getting-started/

APPENDIX 1:

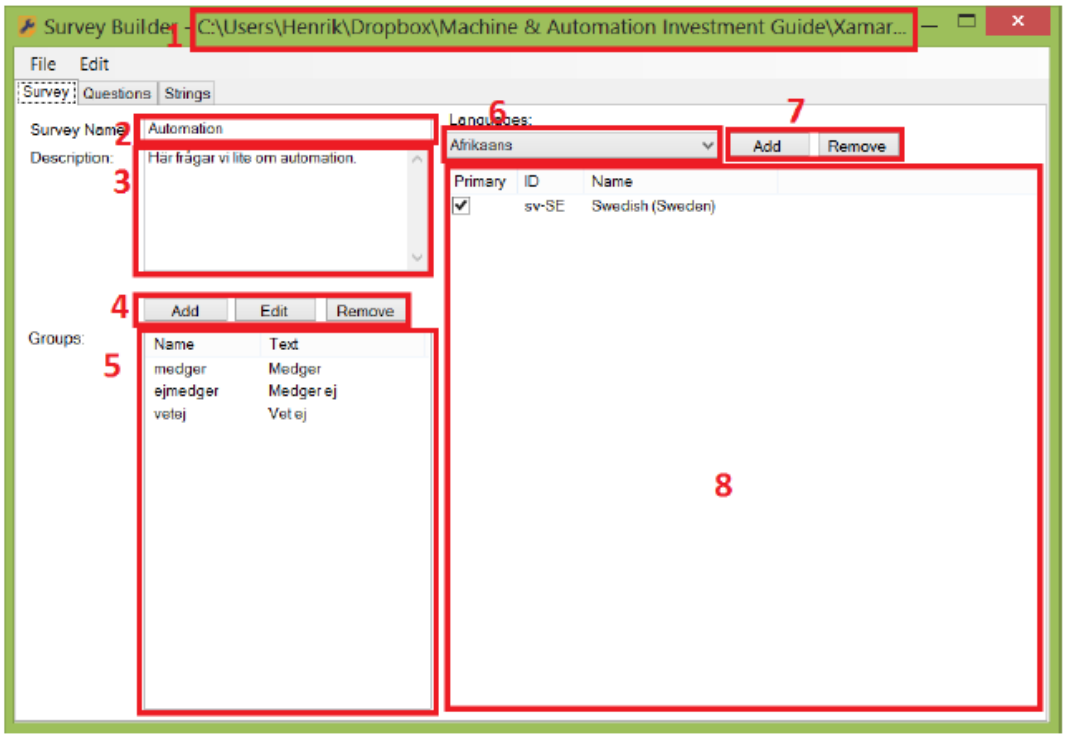

Figure A1: Current Survey Panel

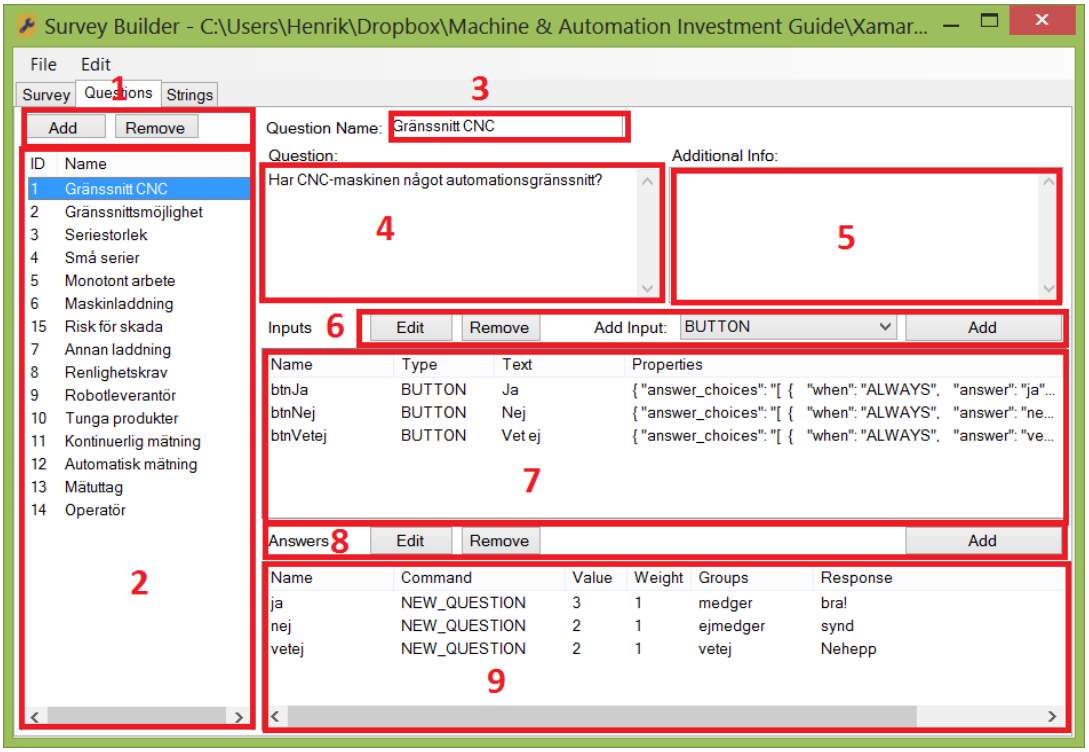

Figure A2: Current Question panel