UNIVERSITATEA "POLITEHNICA" TIMIȘOARA

COURSE: SOFTWARE PROJECT MANAGEMENT

HelpMeSee

Mobile personal assistant application for visually impaired people
Project: Software Design Methods

Author:
Marius OLARIU

Lecturer:
Dr. Răzvan CIOARGĂ

Contents

1 Description of the project

Nowadays software runs the world but most of the software solutions are designed for healthy people. Thus, the visually impaired people are somehow left aside. However, recently things have begun to change and since everyone now owns a smartphone, there have been developed a series of apps for the visually impaired people. Most successful apps are *Be My Eyes* (establishes a live video connection with an volunteer that describes the surroundings to the blind person), BeSpecular(the visually impaired person takes a photo, attaches a voice message to it and sends it to a volunteer that provides help) or the most helpful (in my opinion) *Seeing Ai* from Microsoft. The later app is an assistant which is able to recognize "saved" persons and describe their emotions, read text aloud from labels/books etc., provide description of images from other apps (e.g. Facebook) by importing them into *Seeing Ai* or scene description. However, it is developed only for *iOS*. The other apps in this sector have a free and limited variant and in order to make use of all the features the user has to purchase it.

To address the above mentioned drawbacks an app should be developed for this sector, namely *HelpMeSee*. This app should be an mobile personal assistant that will offer a different perspective compared to what's on the market at this point. If this app will be a success then

2 Requirements

2.1 Functional requirements

2.1.1 Directions

The app should provide assistance to the blind user when he/she tries to get to a location. The user will be guided using spoken descriptions (e.g. ”Go straight 100 meters”, ”You are about to cross the street”).

2.1.2 Location

The user can ask the app to retrieve his current location (e.g. ”Bld. Industriei, nr. 6) and has the possibility to share his current location to a list of predefined friends.

2.1.3 Scene description

By taking a photo the user is able to obtain information about his/her surroundings (e.g. ”A red car is parked in front of you.”) and this phrase will be read out loud.

2.1.4 Text recognition

The app should be able to recognize text and read it out loud. The sources of text could be: text written on paper, labels on products or handwritten notes.

2.2 Nonfunctional requirements

2.2.1 Usability

There must be designed mechanisms that allow the users to navigate through app and access its features easily given their condition.

2.2.2 Maintainability

Due to the fact that bugs are inevitable the implementation of the app must adhere to **O**bject **O**riented **D**esign **P**rinciples (e.g. **S**ingle Class Responsibility, **O**pen-close, **L**iskov substitution, **I**nterface segregation, **D**ependency inversion - SOLID) and goals (loose coupling, high cohesion) so that the time spent fixing bugs will be reduced.

2.2.3 Extensibility

Based on the feedback received from users the app can be extended with new features without great effort. Thus, the architecture of the app must be chosen carefully.

3 Specifications

3.1 Directions

In order to implement this part the app will use *Google Directions API* that will provide the fastest walking route from A to B. The fastest route is provided in json file having a certain format. The user is notified from 20 to 20m with spoken instructions about the itinerary.

3.2 Location

Again, *Google Directions API* and the **G**lobal **P**ositioning **S**ystem (GPS) of the phone will be used to retrieve the location of the user. The location is displayed on the phone and read aloud. Also, using the voice command "friends" the app will send user's location to a list of predefined friends using the **S**hort **M**essaging **S**ervice(SMS). To add a friend in this list the user should use the voice command "add friend" followed by the name of the friend and his/her phone number.

3.3 Scene description

Microsoft Cognitive Services provides *Computer Vision API* that can be used to distill actionable information from images. Therefore, the user takes a photo and then the app will communicate using **Re**presentational **S**tate **T**ransfer (REST) with Microsoft cloud servers to get the description of the image.

3.4 Text recognition

The same *Computer Vision API* can be used to detect and extract text from an image.

3.5 Usability

Each screen of the app will have a button in the bottom-right corner, the "speech input" button. Thus, the user has the possibility to access a feature without having to navigate between the screens. For example if the user is currently walking to a destination but at some point decides to get info about surroundings, he takes a photo - gets the description, and then continues the journey - all this without leaving the directions screen and not being force to abandon one activity in order to perform another.

The widgets size will be slightly bigger than for a normal app. Also, for a simple *tap* the widget name will be spoken out loud and for *double-tap* the widget functionality will be accessed.

3.6 Maintainability

There will be implemented a series of design patterns known for their positive effect on this maintainability alongside good coding practices (e.g comments, descriptive attributes/methods/class names, reduced size methods etc.). The Factory Method will be used in order to reduce coupling between client classes(e.g. Controller/Presenter classes from Model-View-Presenter architecture) and those which communicate with the cloud services.

For accessing each subsystem (i.e. directions, locations, scene description, text recognition) the Facade combined with Singleton pattern will be used. By doing so we do not need to reimplement the functionality of the "speech input" button (present in each screen), thus no code duplication. Also the Facade pattern will allow us to modify the subsystems without having to worry where they are used (since the access to them is through the Facade).

3.7 Usability

The architecture for *HelpMeSee* is going to be **M**odel **V**iew **P**resenter (MVP). All the subsystems that communicate with the network and implement business logic will be a part of the *model*. A *view* will contain just the mappings between the UI elements and the objects that control them (no logic is added, for example handling user input actions). The purpose of the a view(screen) is just to display data and notify the *presenter* about user's actions. Next, each *view* will have a presenter, i.e. a interface, that will interact with the model in order to bring data to the view, modify the state of the model and to apply the UI logic.

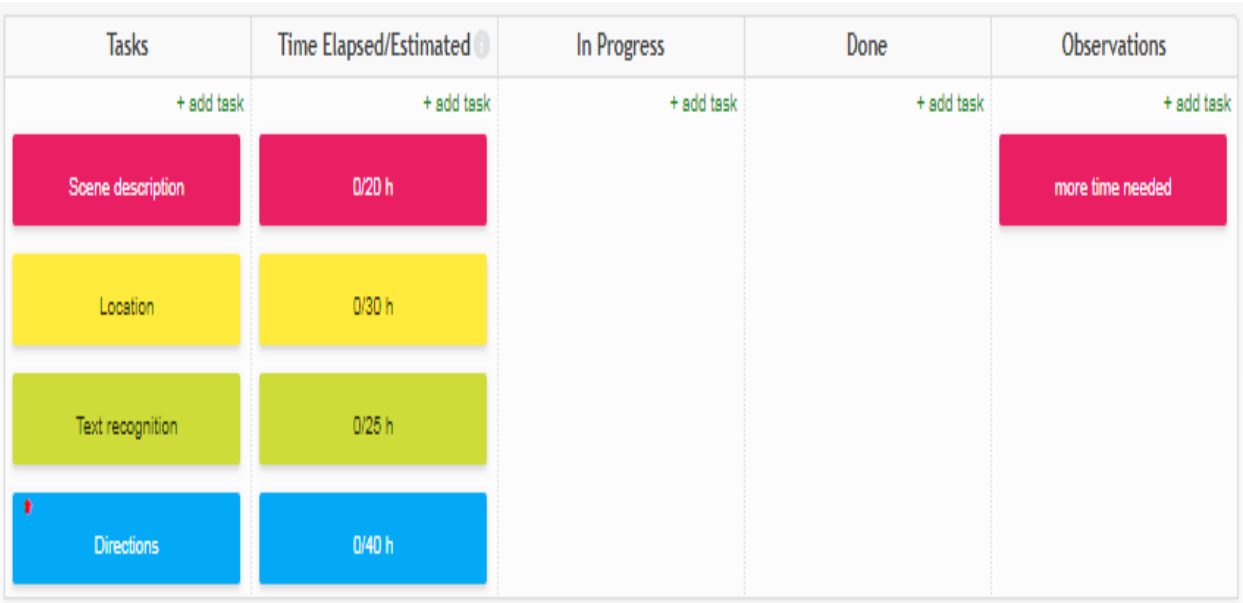

Figure 1: Tasks board

4 Software Design Methods

Software Design Methods (SPM) guide the software engineer in transforming the requirements into an executable software system. They offer theoretical foundations and at the same time some degree of freedom to innovate. However, there is no "silver bullet" [**?**] or one-size-fits-all for a complex software system. According to Sommerville[**?**] the SDM can categorized in:

- Top-down structured design
- Data-driven design
- **O**bject-**o**riented **d**esign (OOD)

Due to the fact that for this project we use an **O**bject-**O**riented paradigm for developing software we will discuss only the OOD methods applied for *HelpMeSee*, that is:

- **U**nified **M**odeling **L**anguage (UML)
- **D**esign **P**atterns (DP)
- **M**odel-**V**iew-**P**resenter (MVP) architecture

4.1 Theoretical Background

4.2 Model-View-Presenter

By using the MVP architecture we maximize the code that can be tested with unit tests, separate business logic from UI (thus we adhere to OOP goals of low coupling, high cohesion or *Single Responsability Principle*). This architecture is based on three types of classes: Model, View and Presenter. The *Model* is responsible for business logic and state of persisted data, *View* for rendering UI elements and the *Presenter* works like a glue between the two (i.e. changes the model data state, fetches data to *View*).

In particular for Android development, we will not end up with huge *Activity* classes (they represent screens). Very often Android programming beginners end up with big Activity classes since there is not a clear distinction if such a class is an UI element or a business logic class, however, we will not detail further on this issue.

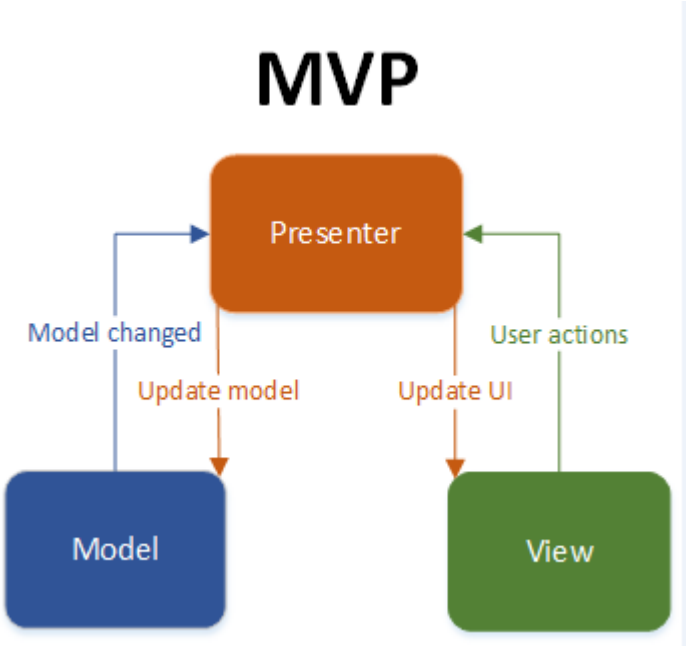

Figure 2: MVP architecture [**?**]

4.3 Unified Modeling Language

According to Martin Fowler [**?**] UML "is a family of graphical notations [...] that help in describing and designing software systems, particularly software systems built using the object-oriented (OO) style". From this family of diagrams we will be using only *Class Diagrams* and *Use Cases*. Class Diagrams represent "the types of objects in system and the various [...] relationships that exist among them" . In the requirements part we have used *Use Cases* to capture the functional requirements of the app.

4.4 Design Patterns

Design patterns represent solutions to commonly occurring problems when developing OO software systems. In other words, a DP is a template for how to solve a software development problem.

4.4.1 Facade

The *Facade* pattern provides a high-level interface that eases the interaction with a complex subsystem. A real-life example of this pattern is when a client wants to place an order to a company (that does not have an online solution for this). First, the client browses a catalogue to search for products codes/names. Next, client calls a customer service representative (acts as a Facade) that register the order, creates the bill and gives the destination address to the shipping department. Coming back to the software domain, this pattern

promotes decoupling between client and subsystem(s) but in the same time it does not prevent clients to interact directly with the subsystem. Facade pattern is often used in combination with the *Singleton* pattern.

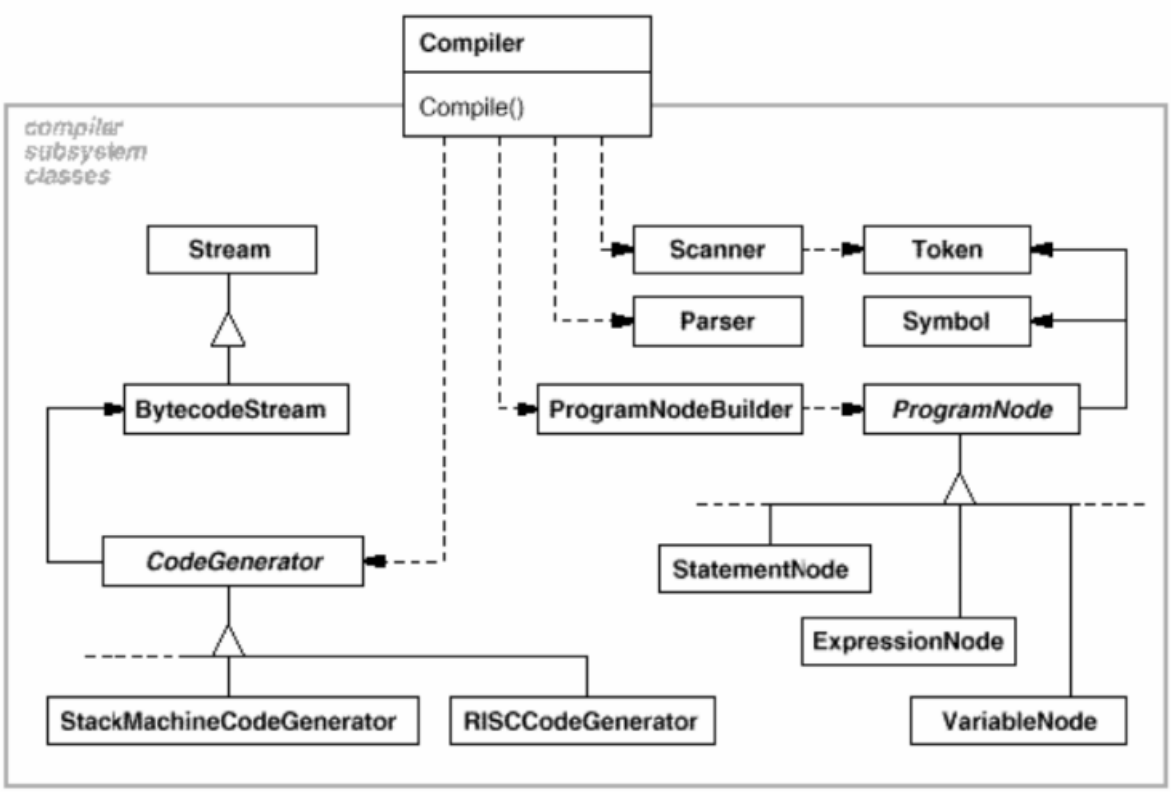

Figure 3: Example of applicability for Facade pattern [**?**]

4.4.2 Factory Method

4.4.3 Singleton

4.5 Concrete application of the SDM

Abbreviations