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*. Apart from these, there are a variety of mobile assistan apps for the visually impaired people but most of them 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 issue with apps that need to be purchased and taking into account that the visually impaired people are in general in need of money (unless you are Andreea Bocelli) an free mobile assistant app will be developed, namely *HelpMeSee*. This app will offer a different perspective compared to what is on the market at this point. To achieve the aforementioned objectives*HelpMeSee* will make use of cyber-foraging ("a technique to enable mobile devices to extend their computing power" [1])) using the cloud resources provided by Google or Microsoft. We prefer to use these solutions because they are the work of more experienced developers and the purpose is not to "reinvent the wheel" (e.g. implement proper variants of speech recognition, extracting a description from an image etc.). Also, cyber-foraging has the proven benefits: extends the computing power, saves storage space and extends battery life of mobile devices.

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 voice instructions (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 easily 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 feature 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. If the user can not be found in the contact list then the user is asked to provide his/her phone number.

3.3 Scene description

Microsoft Cognitive Services provides *Computer Vision API* that can be used to distil 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 services, namely to get a meaningful description from an image.

3.4 Text recognition

The same Microsoft cloud services 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 *processCommandButton* button (processes voice commands). Thus, the user has the possibility to access a feature without having to navigate between the back and forth screens. For example if the user is currently walking to a destination but at some point decides to get some information about surroundings he has to use the *scene description* feature, thus will press the *processCommandButton* button speak "Speech Description" and the according screen will be launched. In this manner the user is not forced to abandon one activity in order to perform another.

The widget's size will be slightly bigger than for a normal app in order to facilitate the access to it. 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. the ones that implement the business logic) and those which communicate with the cloud servers to provide services (product classes).

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 button (present in each screen), thus no code duplication. Also the Facade pattern will allow us to introduce the much desired decoupling between clients and products.

3.7 Usability

The architecture for *HelpMeSee* is going to be **M**odel **V**iew **Pr**esenter (MVP). All the data that needs to be persisted 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 that will interact with the model in order to bring data to the view, modify the state of the model, apply UI logic and most important - implement the business logic .

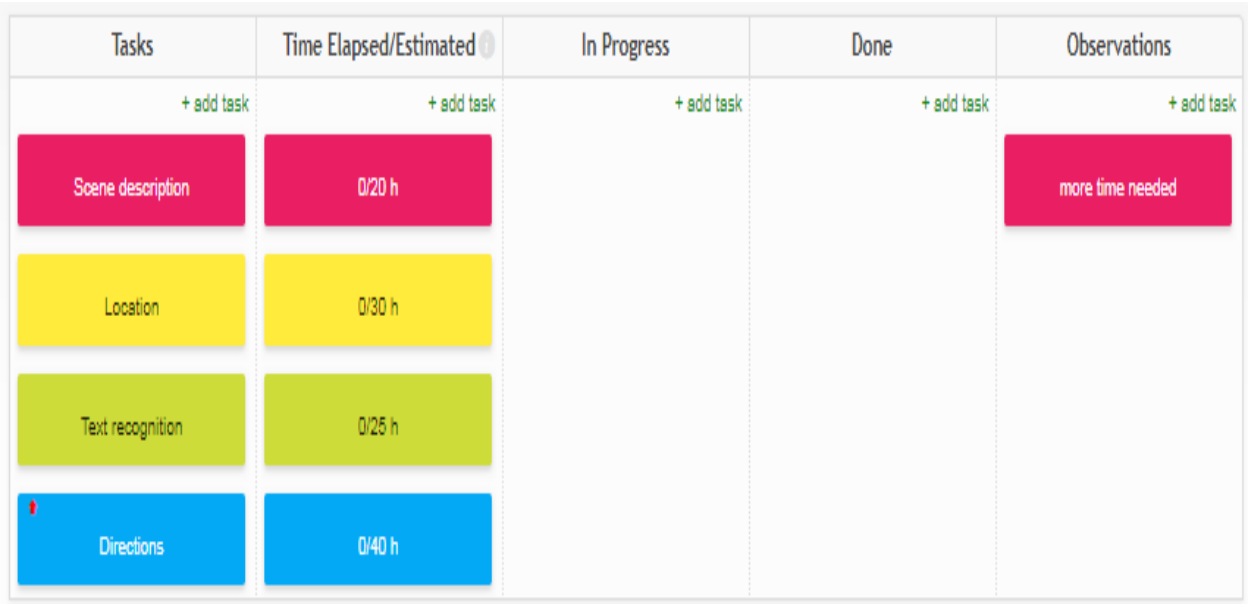


Figure 1: Tasks board

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" [2] or one-size-fits-all for a complex software system. According to Sommerville[3] 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

4.1.1 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*) and end up with a codebase that is readable and maintainable. This architecture is based on three types of classes: Model, View and Presenter. The *Model* is responsible for the state of data, a *View* for rendering UI elements and handling input/output from/to the user and the *Presenter* implements the business logic (moreover, works like a glue between the *View* and *Model*).

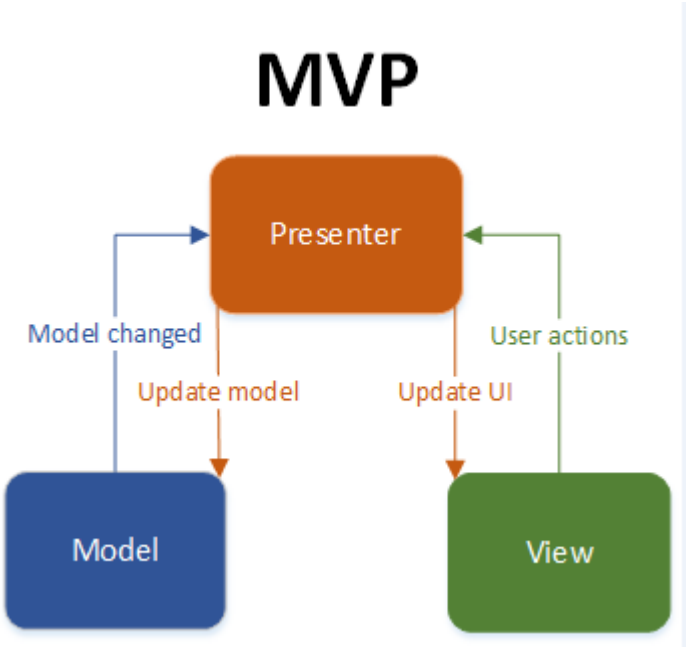


Figure 2: MVP architecture [4]

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.

4.1.2 Unified Modeling Language

According to Martin Fowler [5] 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" [5]. *Use Cases* represent "a technique for capturing the functional requirements of the system"[5] by using a graphical representation and short instruction-like naratives (describe interactions of the users and system).

4.1.3 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.

4.1.4 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.

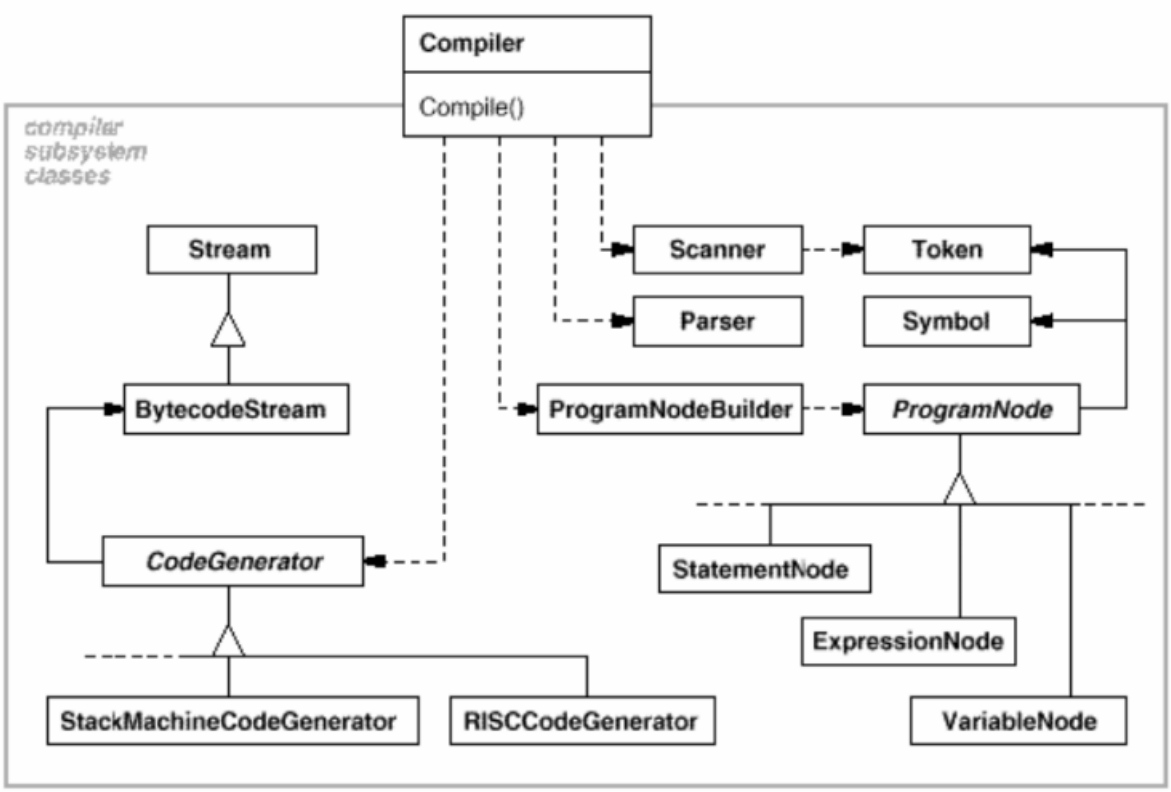


Figure 3: Example of applicability for Facade pattern [6]

4.1.5 Abstract Factory

Provides an interface for creating and managing families of objects without specifying their concrete classes [6]. For example when we design an app that has to change the look and feel of its GUI widgets according to user's preferences (the user can choose the look and feel at runtime). Although the GUI appearance will change, the functionality of widgets will remain the same. In the part of the code where the functionality of the widgets is implemented depends we would end up with a lot of *ifs* that instantiate handle the concrete widget classes and more important with dependencies to that handle which widget classes. To solve the problem we could create an factory that would handle the widget instantiation and changing the widgets would mean just to change the factory that creates and retrieves them.
By using this pattern we obtain independence on how products are created and represented, low coupling between clients and products.

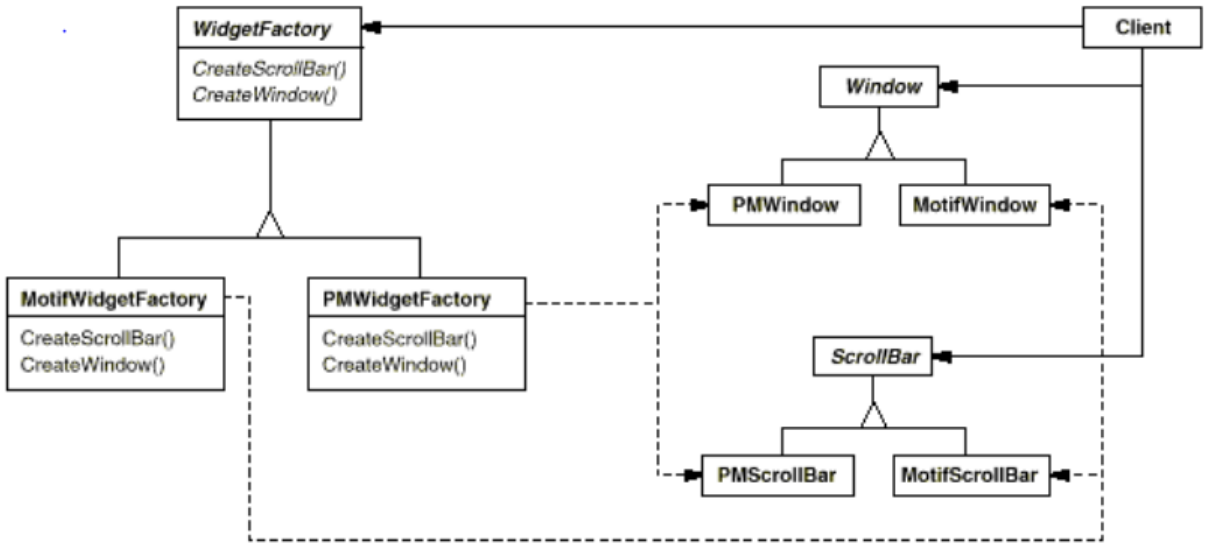


Figure 4: Example of applicability for Abstract Factory pattern [6]

4.1.6 Singleton

By applying the Singleton pattern to a class we ensure that the respective class has only one instance and there is provided a global point of access to this instance. Often, in an app we need to have only one instance of a class, e.g. : the class that handles the connection to database, the class that handles the printing jobs, a window manager etc. . This pattern is often used in conjunction with other patterns like *Abstract Facory* or *Facade*.

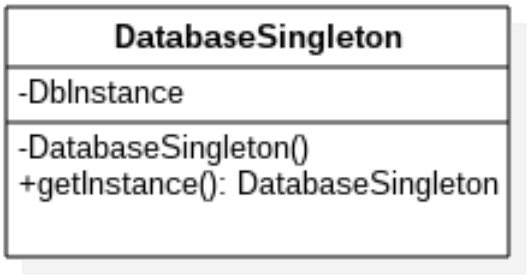


Figure 5: Only one Database connection is needed per system

4.2 Concrete application of the SDM

Basically, the HelpMeSee app is centered around the navigation feature, i.e. helping the visually impaired person to get to a certain destination, as can be seen in the *Use Case* diagram too. Thus, the user can choose to use the *Directions* feature and at any point in time to switch shortly to use another feature (while on his way to destination) using the *processCommandButton* (present in every screen). By doing so we

improve the usability of the app because the user does not have to quit his current activity (navigation) to get extra information about environment.

HelpMeSee use cases:

Get to location B

1. The user launches the *Directions* screen from the *Main-Menu* screen.
2. The user provides destination location using text or voice input.
3. The app renders the path to destination.
4. The app notifies the user that everything is set-up and good to go.
5. As user moves the app notifies him/her with voice instructions.

Where am I?

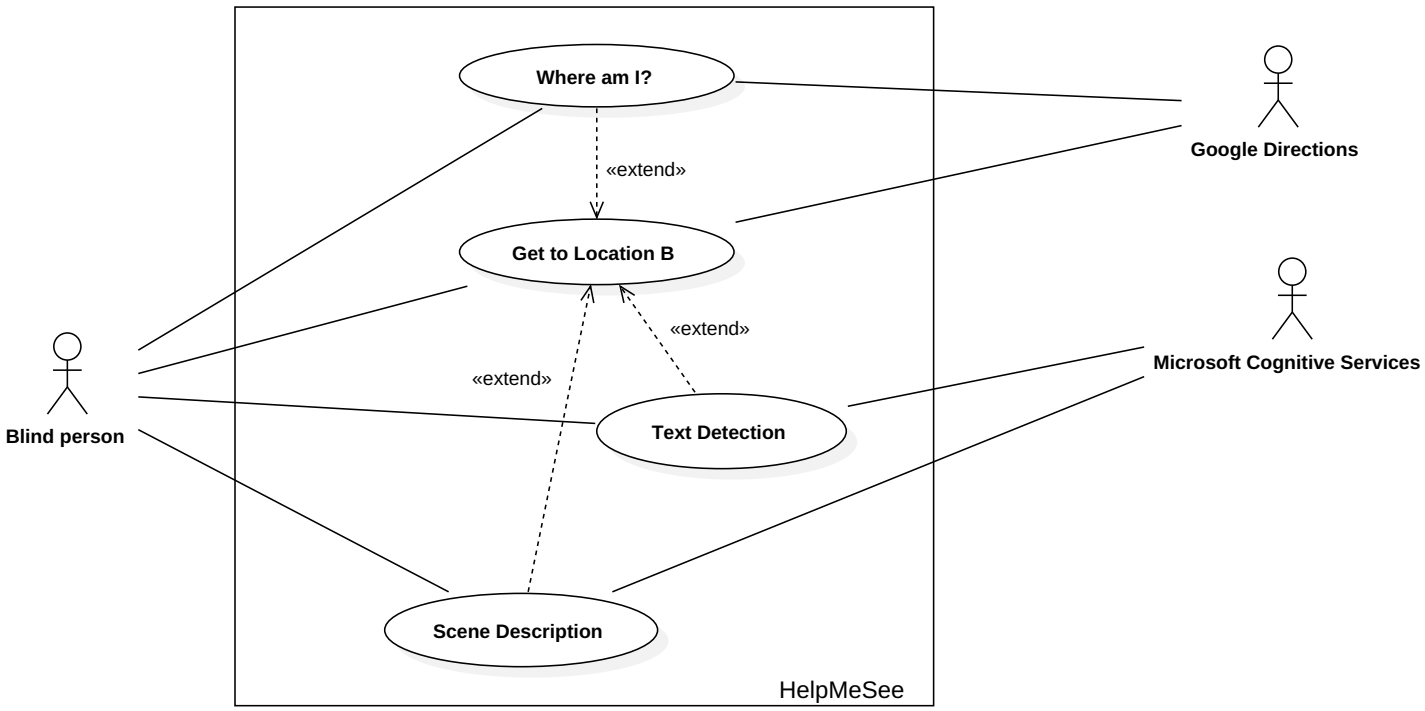
1. The user launches the *Location* screen from *Main-Menu* screen or through *processCommandButton*.
2. The app. displays user location on the screen and reads it out loud.
3. The user can choose to send his location to a list of friends.
   (a) The user can add a new friend to the "friends list"
   (b) The user can play again the reading of location by tapping the half upper part of the screen twice.

Text Detection

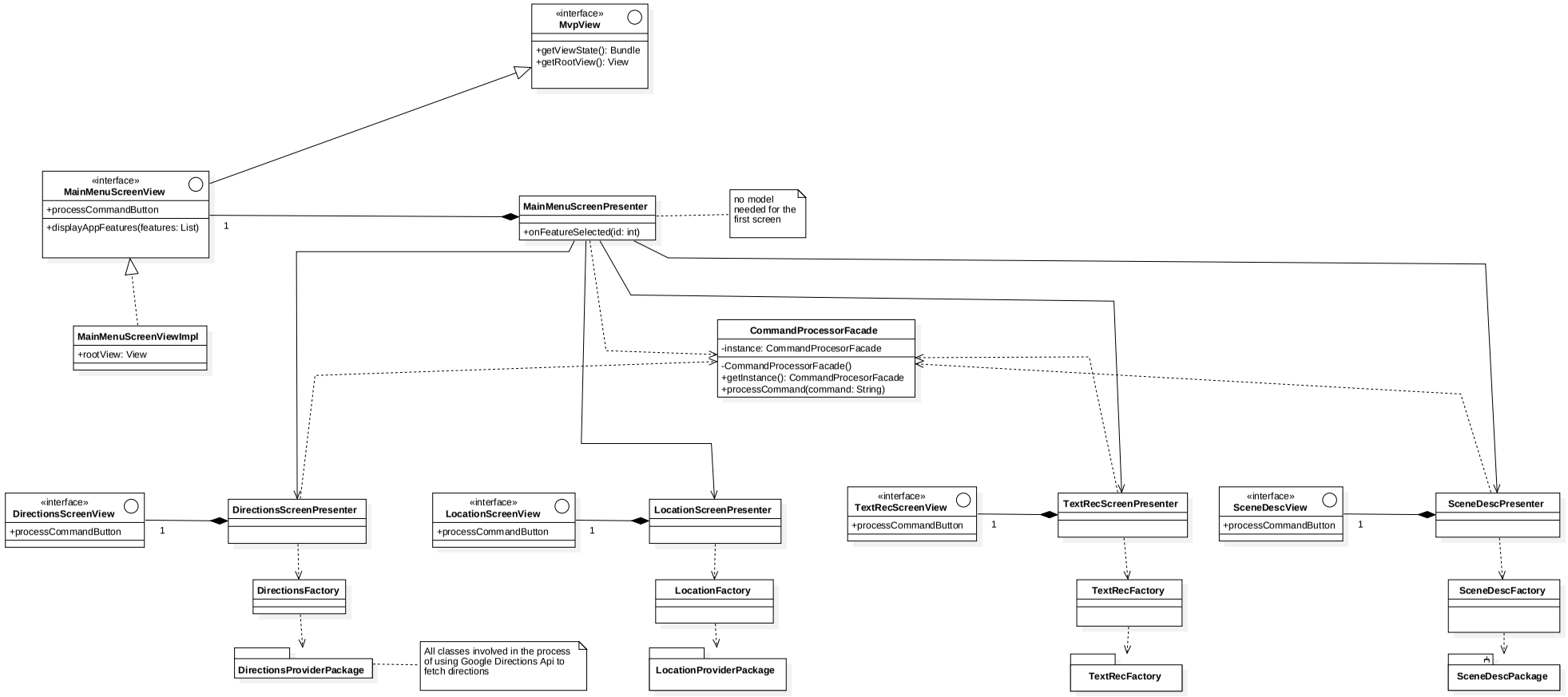
1. The user launches the *TextDetection* screen from *Main-Menu* screen
2. The app will start a timer of 5 seconds, when it expires it will take a photo
   (a) The user can take the photo before the timer expires using *takePictureButton*
3. The app reads loud the detected text
4. User can take another photo or replay the previous detected text

Scene Description
The interactions between the system and the user are the same as with those described in the *Text Detection* scenario. The only difference is represented by the backend algorithm that processes the photo.

HelpMeSee::UseCaseDiagram



HelpMeSee::ClassDiagram



Model-View-Presenter

Views

HelpMeSee has 5 screens: *Main-Menu*, *Directions*, *Locations*, *Scene Description* and *Text Recognition*. For each view there is defined a set of operations that they need to handle, by implementing the following interfaces: *MvpView*, ***[ScreenName]****View* and ***[ScreenName]****Listener* . For example, the *Main-Menu* screen uses the following interfaces: *MvpView*, *MainMenuScreenView* and *MainMenuScreenListener*.

MvpView defines the two services that all the screens need to provide, namely: *getViewState():Bundle* (a mapping of key-value pairs that describe the properties displayed on the screen at a certain point in time, e.g the text displayed on some widget) and *getRootView():View* (usually the parent layout for an Android screen). These two operations are necessary to each *Presenter* to implement the logic and to update the screen if necessary. Next, ***[ScreenName]****View* defines the particular operations that the screen needs to handle, for instance the *MainMenuScreenView* interface contains only *+displayAppFeatures(features:List)* method. For each screen the associate listener is the *Presenter* (implements ***[ScreenName]****Listener)* that contains the logic for handling a certain user input (the responsability is took out from the screen).

As can be seen above, the only responsibilities of *Main-Menu* screen is to display data provided by someone else (i.e the Presenter) and to pass the user interactions with the screen to the presenter. By doing so we respect the MVP architecture requirements for a View. This approach is repeated for the other screens too.

Presenters

The purpose of presenters is to implement the business logic and since they are distinct for each screen there is no common interface. Moreover since no entity in the app depends on their interface (but rather, they depend on other entities) they are implemented as classes. As an example of presenter, the *MainMenuPresenter* acts as bridge between the features that the app provides (entry point for using one of them). Consequently, the following snippet provides an overview of how the business logic that needs to be implemented.

Listing 1: onFeatureSelected() mock

```
public void onFeatureSelected(long id) {
final AppFeaturesEnum selectedFeature =
    AppFeaturesEnum.valueOf(appFeaturesList.get((int) id));

  //TODO : launch the selected feature
  switch (selectedFeature) {
     case LOCATION:
     Log.i(MAIN_SCREEN_TAG, "Launch Location screen");
     break;

     case DIRECTIONS:
       Log.i(MAIN_SCREEN_TAG, "Launch DIRECTIONS screen");
     break;

     case SCENE_DESCRIPTION:
     Log.i(MAIN_SCREEN_TAG, "Launch SCENE_DESCRIPTION screen");
     break;

     case TEXT_RECOGNITION:
     Log.i(MAIN_SCREEN_TAG, "Launch TEXT_RECOGNITION screen");
```

```
            break;

        default:
            Log.e(MAIN_SCREEN_TAG, "Feature doesn't exist!");
        }
    }
```

Model

For the moment the app does not need a proper model because the user never changes the state of the data (e.g. for instance it cannot change the directions for a certain route or change the text describing a scene in an image using user inputs).

Facade

As mentioned before in this document, in order to make the app more usable the user can give voice commands, these commands are for the moment the names of the features that the user wants to access: Location, Directions, Text Recognition, Scene Description. In each screen there will be the button *processCommandButton* that will start the microphone for recording. After the command is decoded an action will be taken concordantly, e.g when on track to a destination user wants to obtain a "scene description" (it will launch *SceneDescription* screen by pronouncing "scene description"). However, using this approach each presenter would end up "knowing" about the other features of the app (not to mention the duplicate code involved) and this would represent an escalation of the whole MVP architecture. In order to overcome this issue the *Facade* pattern (implemented as a Singleton) is used having the following interface:

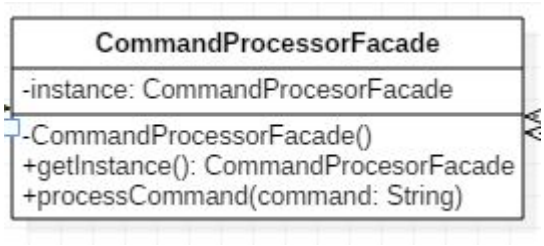


Figure 6: Facade pattern concrete implementation

Now, the button will pass only the voice command (e.g. "Location") to the *Facade* that will handle the ease the access to features and solves the coupling problem aforementioned.

Abstract Factory

For each feature there is a package that provides the functionality. However, over it is not useful to instantiate over and over again the same objects or the clients to know about the specific classes that they need to instantiate. Therefore, an Abstract factory is defined for each package (e.g *LocationFactory*, *DirectionsFactory* etc, they can be seen in the UML diagram), and these factories will control and retrieve object instantiation for each class in a certain package. Also, they are implemented as singletons (since it doesn't make sense to have more than one factory per package) and will use a *Object Pool* to retrieve the already created and reusable objects.

Singleton

Singleton is one of the simplest and most useful design pattern and for our project was used for the factory classes and *CommandProcessorFacade* class.

Abbreviations

app application

apps applications

MVP Model View Presenter

SDM Software Design Methods

UML Unified Modeling Language

References

[1] G. A. Lewis, P. Lago, and P. Avgeriou, "A decision model for cyber-foraging systems," in *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*. IEEE, 2016, pp. 51–60.

[2] F. Brooks and H. Kugler, *No silver bullet*. April, 1987.

[3] I. Sommerville, *Software engineering*. New York: Addison-Wesley, 2010.

[4] V. Zukanov, "Activities in android are not ui elements," 2015, [accessed 25-March-2018]. [Online]. Available: https://www.techyourchance.com/activities-android/

[5] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[6] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.