Design Document

[2IS70] App Development

03-03-2019

Yoke

Tar van Krieken - 1244433
Bram van Leeuwen - 0996101
Dylan Mijling - 1237996
Karolina Strahilova - 1284029
Yeochan Yoon - 1227750
Ivo Zenden - 1222833

Class diagrams

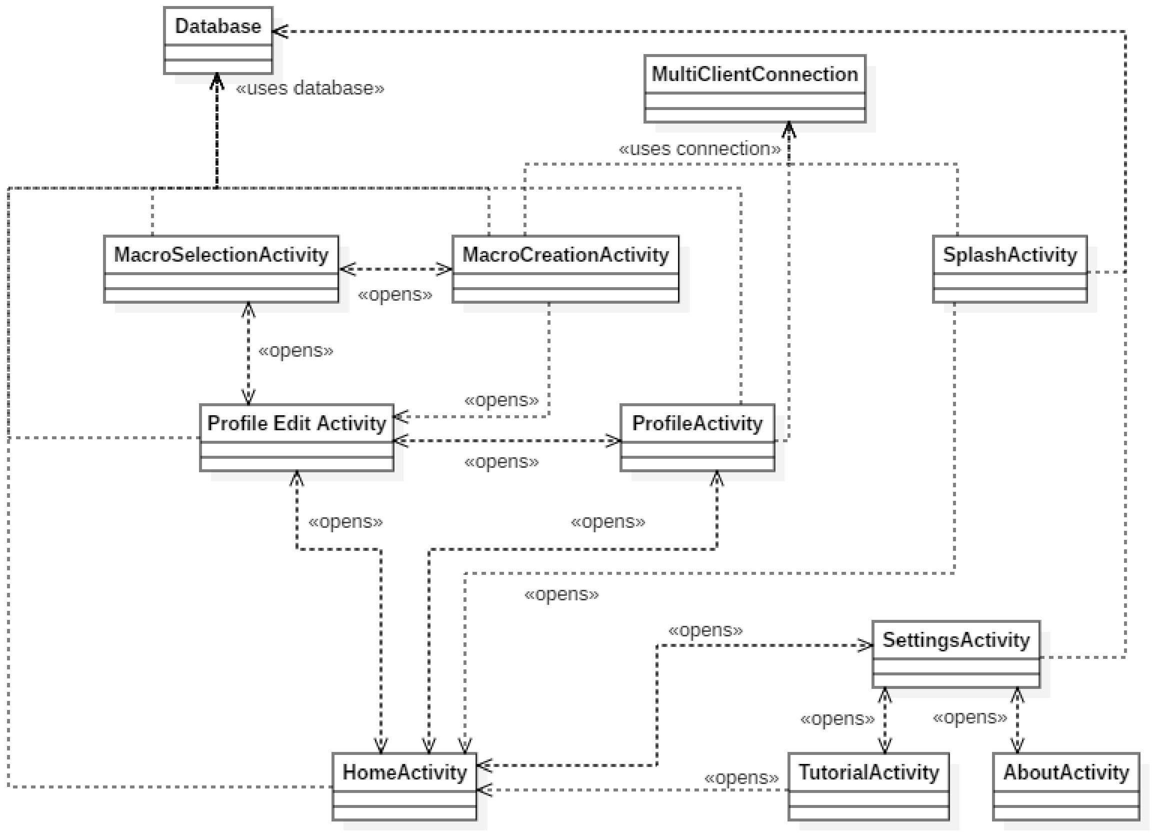
Terminology

- Macro: A command that can be executed by a computer, as well as its visual representation in the app (I.e. name, image, etc)
- Server: The computer software that receives the macros:
- Client/App: The phone software that sends the macros:
  - When the term Client is used, there is more focus on the connection
  - When the term App is used, there is more focus on the user interface and experience

General remarks

- Whenever a constructor isn't explicitly defined, it is public and takes no arguments
- Whenever no return type is defined, it will be of type void
- Whenever a class name includes a package prefix (E.g. java.awt), it refers to a class not made by us, but imported from a library or already present in the environment in another way.
- Whenever an external class is used or extended, only the relevant methods are shown in the diagram.
- Whenever there is a dependency on a whole package, the details of what the exact implementation is have been left out for simplicity. E.g. it doesn't really matter what exact java classes are used in order to create a tray menu for the computer application.
- Whenever multiplicity is left out, it is intended to resemble only allowing multiplicity 1.
- Composition arrows have been used to stress the structural dependency, but it's not necessarily the case that direct association can live on without the reference.
- Interface realizations use the same arrows as class generalizations, as starUML doesn't allow the correct arrow to be used when using this representation for interfaces.
- Relations between native android classes have not been labeled with variable names, as we don't control or are interested in under what name it is stored locally within android's classes. Instead they have been labeled with roles, as indicated by the surrounding arrows (<<roleName>>).

Overview Class Diagram



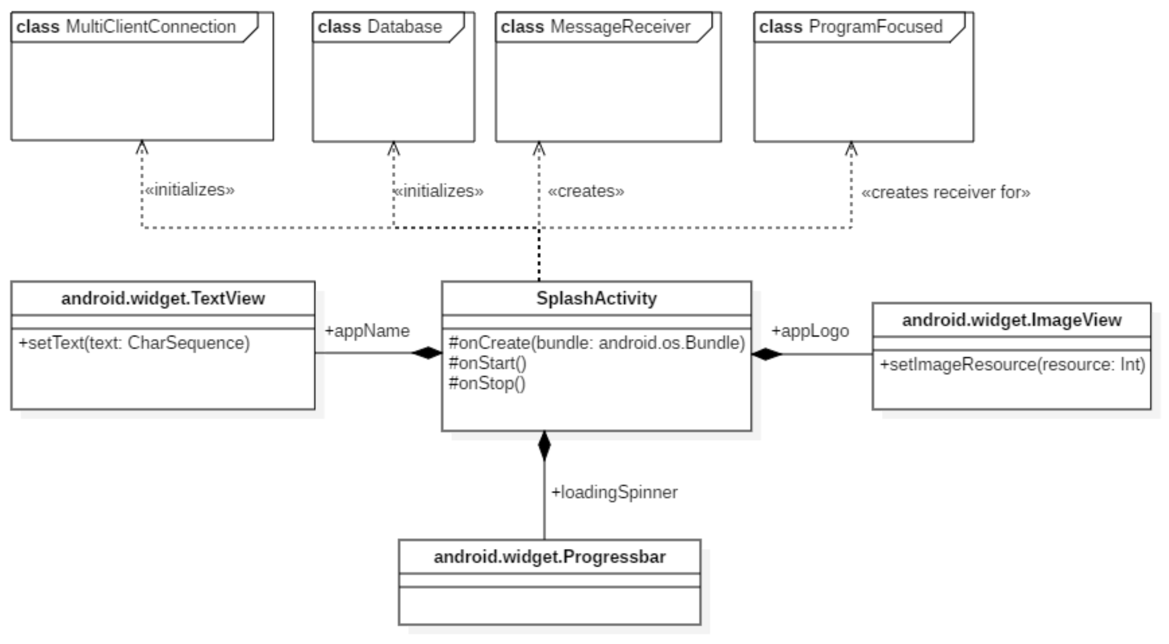
The General Class Diagram gives an overview of all the Activities of the application, stating the relationships between them. It is similar to the navigation diagram from the User Interface Document, however this one is somewhat more precise, also including some of the functionality classes.

Upon the first launch of the application, the Tutorial activity is displayed, giving the user some information as to how to use the application. It then opens the Home activity. For every consecutive launch, the Home activity will be opened after the Splash activity, which depends on the Database to retrieve either the default settings or the user's personalised preferences (for more information see the Database class diagram) and on the MultiClientConnection class (see the Client connection diagram). The Home activity also depends on the Database. From the Home activity the user can navigate to some of the other activities, like the Profile activity, the Profile Edit Activity and the Settings activity.

The Profile activity contains the user interface of the functionality of the application. It can open the Profile Edit activity, which allows the user to create and edit their own profiles by selecting already existing Macros (by starting the Macro Selection activity) or by creating and editing their own (in the Macro Creation activity). Both the Macro Selection and Creation activities depend on the Database. The Macro Creation also depends on the MultiClientConnection Activity, enabling the creation of a custom launcher by requesting the selection of the program on the user's computer.

The Settings activity (explained in detail in the Settings Activity Class Diagram) can open the Tutorial again, should the user need to see it again, as well as an About activity, which was too trivial to explain in a class diagram of its own. It is essentially an activity containing a TextView, giving more information about the app.

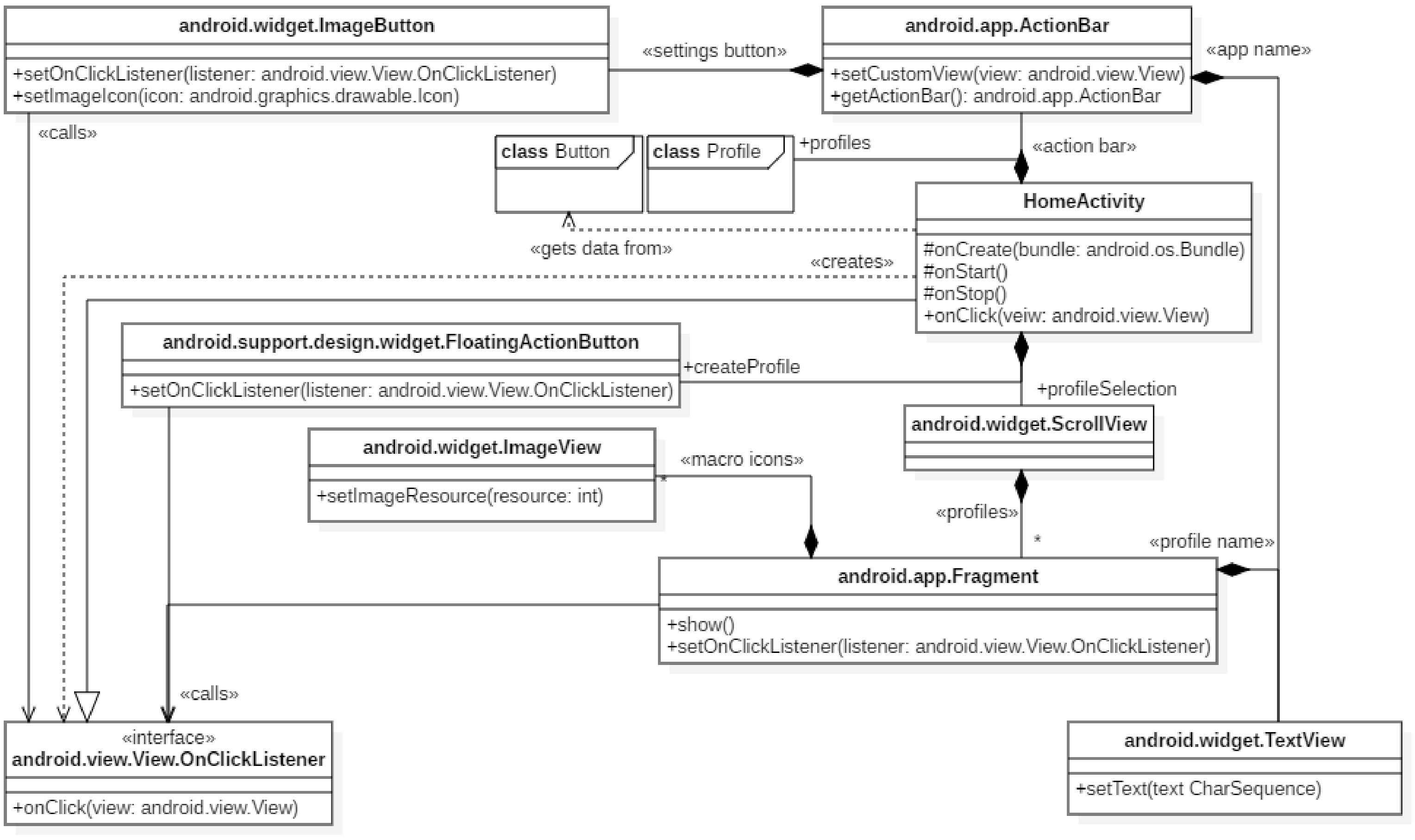
Splash



The splash activity is the first activity (non-interactable) that an user will see when they open our app. It will show our logo using an ImageView and our app name in a TextView. In the splash activity the connection to the MultiClientConnection (server) and the Database is initialized. Since this takes some time, the splash activity will also show a loading icon, which can be done by using a ProgressBar and declaring it as a spinner (So ProgressBar spinner = new ProgressBar()).

Next to this we also wanted for the splash activity to be able to receive ProgramFocused messages, which is a child class of the class message, which would mean that the activity should implement the MessageReceiver interface (This is depicted in the diagram as a class, since we had some trouble with getting it to look like an interface in StarUML).

Home



android.widget.ImageButton
+setOnClickListener(listener: android.view.View.OnClickListener)
+setImageIcon(icon: android.graphics.drawable.Icon)
«settings button»
android.app.ActionBar
+setCustomView(view: android.view.View)
+getActionBar(): android.app.ActionBar
«app name»
«calls»
class Button
class Profile
+profiles
«action bar»
HomeActivity
#onCreate(bundle: android.os.Bundle)
#onStart()
#onStop()
+onClick(veiw: android.view.View)
«gets data from»
«creates»
android.support.design.widget.FloatingActionButton
+setOnClickListener(listener: android.view.View.OnClickListener)
+createProfile
+profileSelection
android.widget.ScrollView
android.widget.ImageView
+setImageResource(resource: int)
«macro icons»
«profiles»
*
«profile name»
android.app.Fragment
+show()
+setOnClickListener(listener: android.view.View.OnClickListener)
«calls»
«interface»
android.view.View.OnClickListener
+onClick(view: android.view.View)
android.widget.TextView
+setText(text CharSequence)

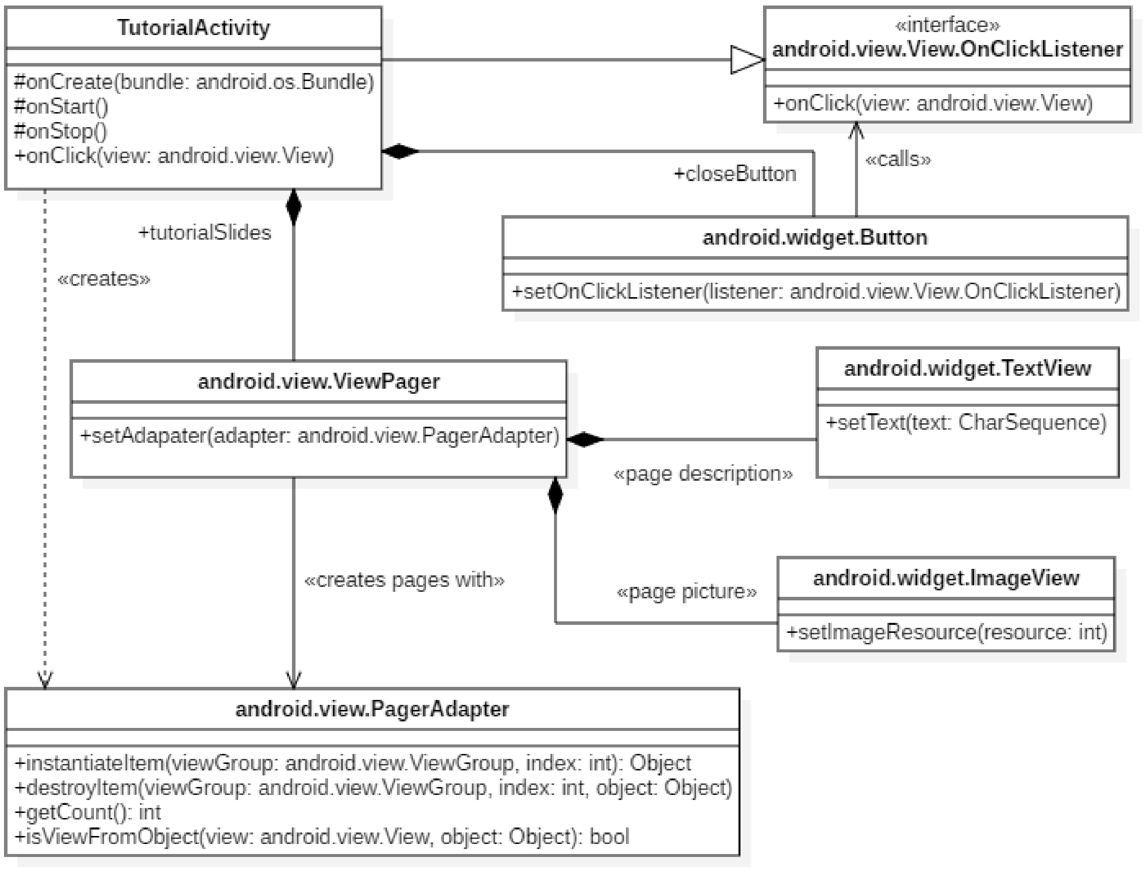
The home page is the first screen where users can actually interact with our app. The home page has a actionbar at the top, so it needs to contain an ActionBar object. The action bar will have a button that will open up the settings activity and since we want to give it the standard settings icon (the gear), we use a ImageButton. We also want our app name in the action bar so it will need a TextView.

Another function of the home page is the ability to press a button that opens the CreateProfile activity. Since we wanted a round button with a plus sign at the bottom right of our screen (since this is the most intuitive for users), we chose for a FloatingActionButton (since we found out that this type of button does all those things automatically).

The main functionality of the home page is of course being able to select one of your profiles. Since we did not want to set a maximum on the amount of profiles that a user can create, and only a fixed amount of profile-previews would fit on a screen, we had to find a way to either scroll on our screen or open up a list in which you can scroll. We decided on the first option, and for this we need a ScrollView.

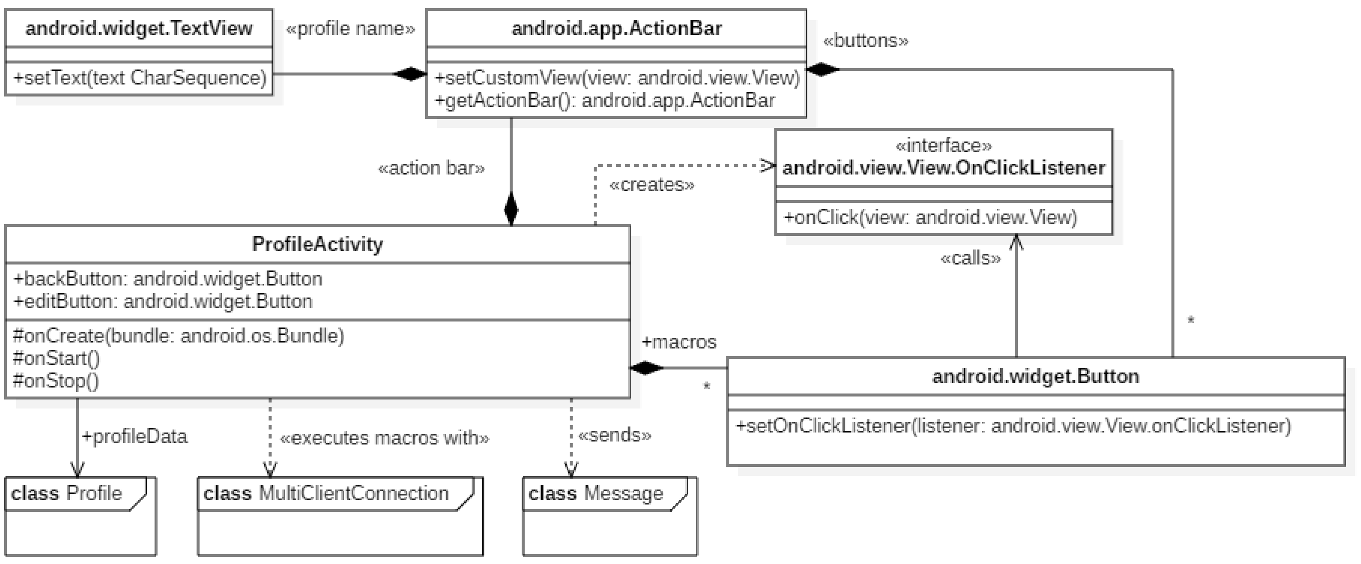
To represent the profile-previews we thought that fragments would be best, since we can create these at runtime and easily put them in a ScrollView. For each preview we wanted to show the buttons that the profile has and the name of the profile. So for each button there would be a ImageView on the preview so this is a one-to-many relation. And since every profile only has one name this is a one-to-one relation from Fragment to TextView. We wanted it so that an user could simply press on the preview (ie the fragment) to open up the corresponding profile. Which meant that the fragment should also implement the View.OnClickListener have a reference to (possibly multiple) View.onClickListener objects.

Tutorial


TutorialActivity
#onCreate(bundle: android.os.Bundle)
#onStart()
#onStop()
+onClick(view: android.view.View)
«interface»
android.view.View.OnClickListener
+onClick(view: android.view.View)
+closeButton
«calls»
android.widget.Button
+setOnClickListener(listener: android.view.View.OnClickListener)
+tutorialSlides
«creates»
android.view.ViewPager
+setAdapater(adapter: android.view.PagerAdapter)
android.widget.TextView
+setText(text: CharSequence)
«page description»
«creates pages with»
«page picture»
android.widget.ImageView
+setImageResource(resource: int)
android.view.PagerAdapter
+instantiateItem(viewGroup: android.view.ViewGroup, index: int): Object
+destroyItem(viewGroup: android.view.ViewGroup, index: int, object: Object)
+getCount(): int
+isViewFromObject(view: android.view.View, object: Object): bool

For the tutorial we decided on a more static tutorial. So it will give a few "pages" that you can swipe between, each with a picture of a screen in the app (corresponding to an activity or an fragment) and some text explaining what everything on that screen does. To do this, the TutorialActivity will need an ViewPager (the holder of the "pages") which has a reference to a PageAdapter (the collection of "pages"). And since we want for each page to show a screen with some explanation, the PageAdapter will contain an ImageView and an TextView per page.

Profile

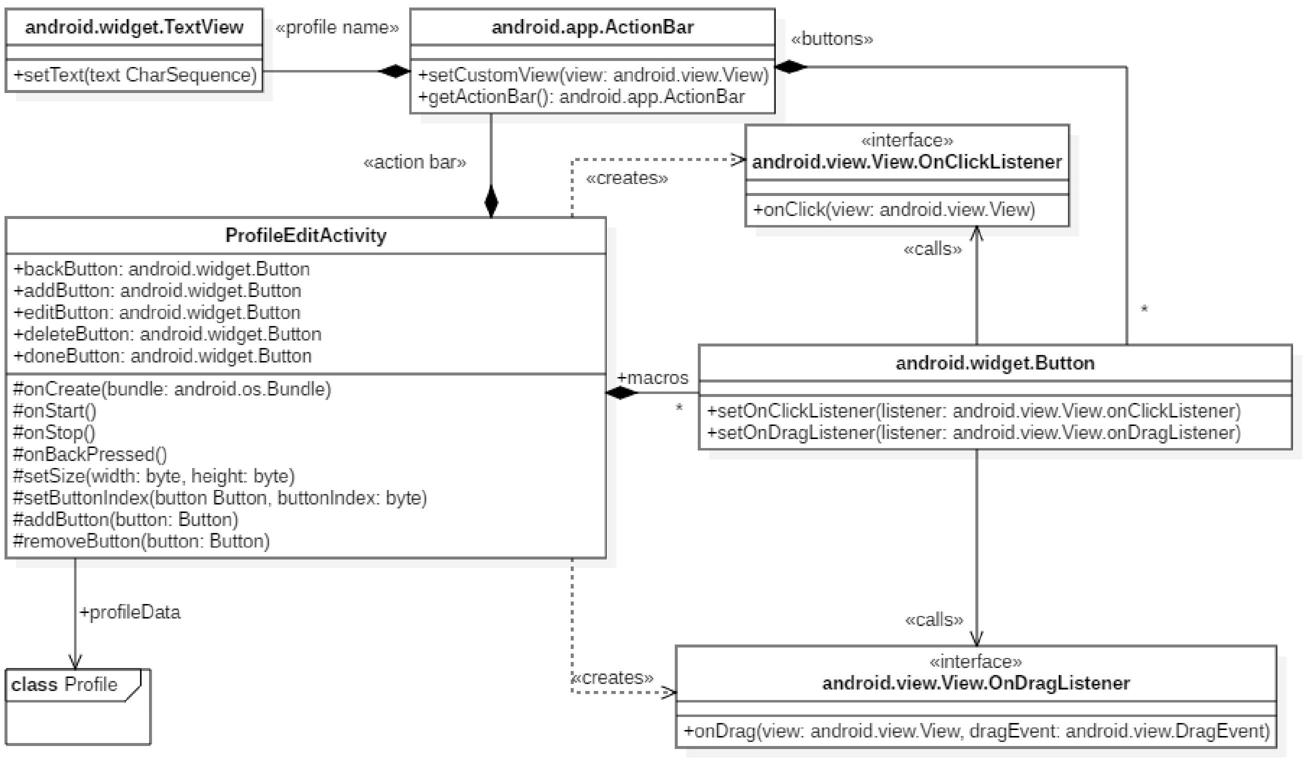


android.widget.TextView
+setText(text CharSequence)
«profile name»
android.app.ActionBar
+setCustomView(view: android.view.View)
+getActionBar(): android.app.ActionBar
«buttons»
«interface»
android.view.View.OnClickListener
+onClick(view: android.view.View)
«action bar»
«creates»
«calls»
ProfileActivity
+backButton: android.widget.Button
+editButton: android.widget.Button
#onCreate(bundle: android.os.Bundle)
#onStart()
#onStop()
+macros
*
*
android.widget.Button
+setOnClickListener(listener: android.view.View.onClickListener)
+profileData
«executes macros with»
«sends»
class Profile
class MultiClientConnection
class Message

Profile activity displays the macros that are registered to a profile selected from the home screen of the app. The onCreate() method initializes the activity and it retrieves the selected profile data from the database. This data consists of button and macro data which the profile serves. It also includes the information of position setups and sizes for each button and macro. The methods in ProfileActivity class are implemented to extract each specification of such data. For example, getButton() method retrieves information of every button of the profile and getWidth() method gets the size specification.

The class implements View.onClickListener interface to enable clicking macros and buttons. This interface associates with the android.widget.Button such that buttons can react to the clicks with its setOnClickListener(View.onClickListener): void method. This page is also composed of an action bar on the top of the screen where it shows a back button and an edit button, which are the attributes declared in the ProfileActivity class. The name of each profile is shown by the TextView.

Profile Edit


android.widget.TextView
+setText(text CharSequence)
«profile name»
android.app.ActionBar
+setCustomView(view: android.view.View)
+getActionBar(): android.app.ActionBar
«buttons»
«action bar»
«creates»
«interface»
android.view.View.OnClickListener
+onClick(view: android.view.View)
«calls»
*
ProfileEditActivity
+backButton: android.widget.Button
+addButton: android.widget.Button
+editButton: android.widget.Button
+deleteButton: android.widget.Button
+doneButton: android.widget.Button
#onCreate(bundle: android.os.Bundle)
#onStart()
#onStop()
#onBackPressed()
#setSize(width: byte, height: byte)
#setButtonIndex(button Button, buttonIndex: byte)
#addButton(button: Button)
#removeButton(button: Button)
+macros
*
android.widget.Button
+setOnClickListener(listener: android.view.View.onClickListener)
+setOnDragListener(listener: android.view.View.onDragListener)
+profileData
«calls»
class Profile
«creates»
«interface»
android.view.View.OnDragListener
+onDrag(view: android.view.View, dragEvent: android.view.DragEvent)

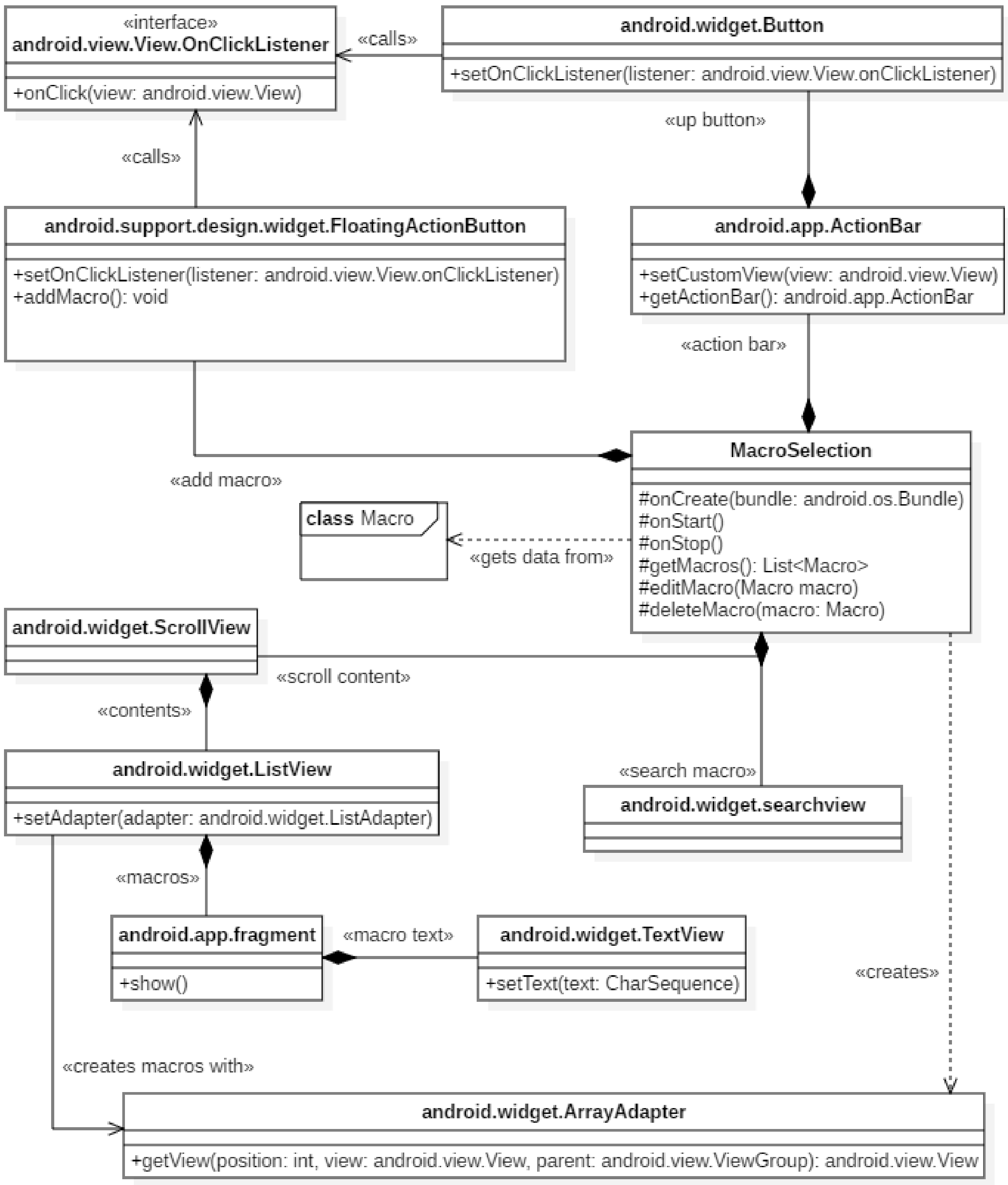
When a user clicks the edit button on the top bar of profile activity, it will switch to the profile edit activity, which is designed for editing the specification of macros from the profile activity. This activity has extra button attributes that are going to be shown on the actionbar.

First is the plusButton which adds a new macro to profile, which is handled by addButton method. Clicking this button directs to the macro selection activity where user can choose which macro to add on to the profile.
The second is the deleteButton which deletes the macro from the profile, which is handled by removeButton method.
The third button is the doneButton which is activated for confirmation of any changes made during the edit activity. User can press this to save the edit. Pressing the backButton during the edit activity would not change anything and exit the edit activity.

The user can also interact with the macro during the edit activity as well. When you hold a macro, it becomes eligible for repositioning by dragging it around the screen. The ProfileEditActivity class allows this event by implementing View.onDragListener interface and providing a button a corresponding setOnDragListener(View.onDragListener) method to appoint an action for a drag event.

When it finishes editing, the change is saved and sent back to the profile database in order to update it. This will ensure the latest version of edit is visible when you open up the profile later.

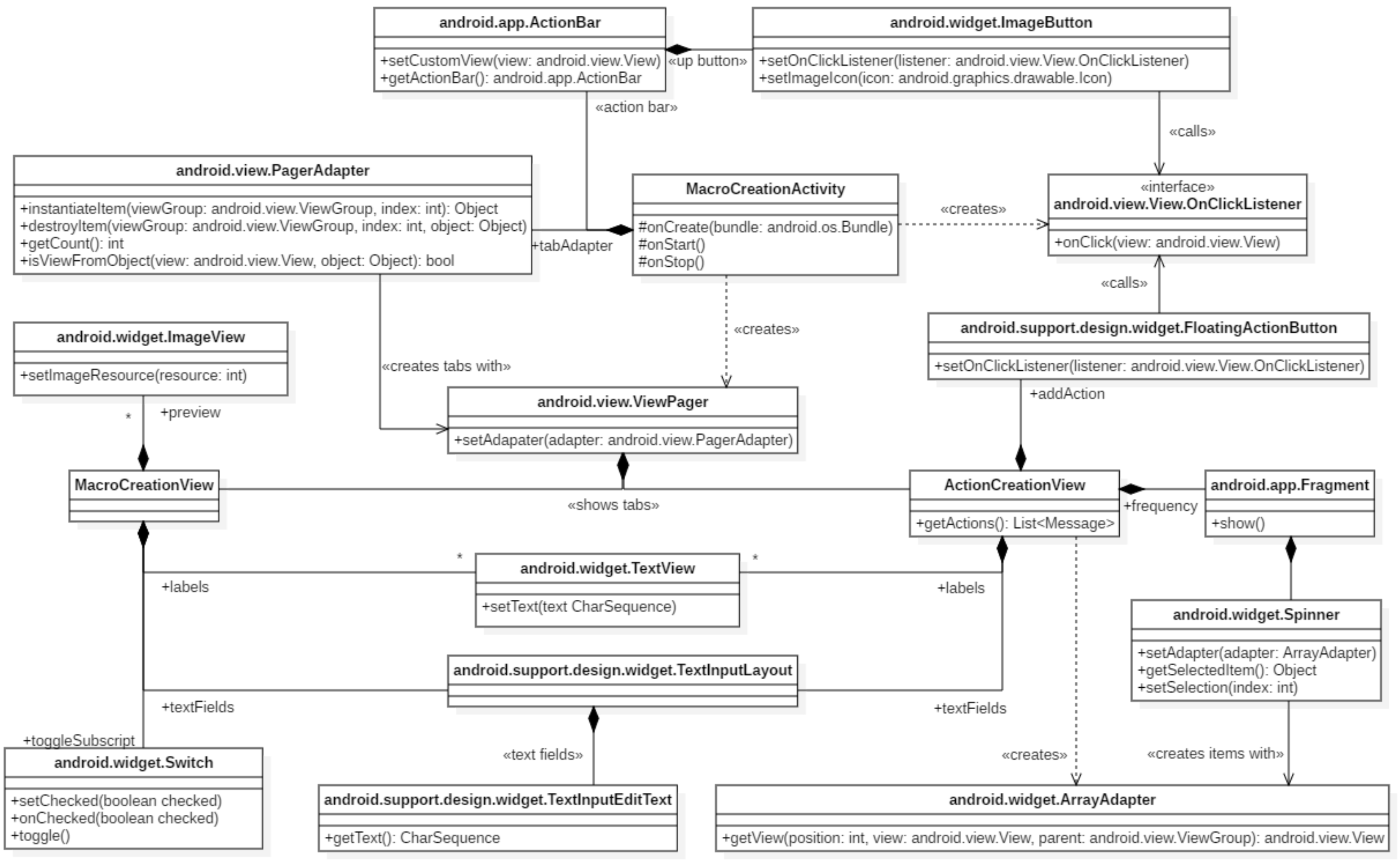
Macro selection


«interface»
android.view.View.OnClickListener
+onClick(view: android.view.View)
«calls»
android.widget.Button
+setOnClickListener(listener: android.view.View.onClickListener)
«up button»
«calls»
android.support.design.widget.FloatingActionButton
+setOnClickListener(listener: android.view.View.onClickListener)
+addMacro(): void
android.app.ActionBar
+setCustomView(view: android.view.View)
+getActionBar(): android.app.ActionBar
«action bar»
MacroSelection
#onCreate(bundle: android.os.Bundle)
#onStart()
#onStop()
#getMacros(): List<Macro>
#editMacro(Macro macro)
#deleteMacro(macro: Macro)
«add macro»
class Macro
«gets data from»
android.widget.ScrollView
«scroll content»
«contents»
android.widget.ListView
+setAdapter(adapter: android.widget.ListAdapter)
«search macro»
android.widget.searchview
«macros»
android.app.fragment
+show()
«macro text»
android.widget.TextView
+setText(text: CharSequence)
«creates»
«creates macros with»
android.widget.ArrayAdapter
+getView(position: int, view: android.view.View, parent: android.view.ViewGroup): android.view.View

The macro selection diagram gives an overview of the macro selection activity. The user will see this activity when he/she wants to select some macros for his profile. In this activity there will also be an add button which will be represented as a floating action button. This button will have a setOnClickListener method and an actual addMacro method such that the user can go to another activity where they can add the macro. Also the user is able to delete and edit in this activity, using the editMacro and deleteMacro functions respectively.

Also the user must be aware about the macros that exist already. These macros will be available in a list. This list will be created with the listOfMacros method. This method will return a list with all the macros already available to the user. Also this activity features an action bar in which the user can go back to the previous screen and of course a search bar. With this search bar the user can easily find back a macro he previously created.

Macro creation


android.app.ActionBar
+setCustomView(view: android.view.View)
+getActionBar(): android.app.ActionBar
«up button»
android.widget.ImageButton
+setOnClickListener(listener: android.view.View.OnClickListener)
+setImageIcon(icon: android.graphics.drawable.Icon)
«action bar»
«calls»
android.view.PagerAdapter
+instantiateItem(viewGroup: android.view.ViewGroup, index: int): Object
+destroyItem(viewGroup: android.view.ViewGroup, index: int, object: Object)
+getCount(): int
+isViewFromObject(view: android.view.View, object: Object): bool
+tabAdapter
MacroCreationActivity
#onCreate(bundle: android.os.Bundle)
#onStart()
#onStop()
«creates»
«interface»
android.view.View.OnClickListener
+onClick(view: android.view.View)
«calls»
android.widget.ImageView
+setImageResource(resource: int)
«creates tabs with»
«creates»
android.support.design.widget.FloatingActionButton
+setOnClickListener(listener: android.view.View.OnClickListener)
+addAction
*
+preview
android.view.ViewPager
+setAdapater(adapter: android.view.PagerAdapter)
MacroCreationView
«shows tabs»
ActionCreationView
+getActions(): List<Message>
+frequency
android.app.Fragment
+show()
*
android.widget.TextView
+setText(text CharSequence)
*
+labels
+labels
android.widget.Spinner
+setAdapter(adapter: ArrayAdapter)
+getSelectedItem(): Object
+setSelection(index: int)
android.support.design.widget.TextInputLayout
+textFields
+textFields
+toggleSubscript
android.widget.Switch
+setChecked(boolean checked)
+onChecked(boolean checked)
+toggle()
«text fields»
android.support.design.widget.TextInputEditText
+getText(): CharSequence
«creates»
«creates items with»
android.widget.ArrayAdapter
+getView(position: int, view: android.view.View, parent: android.view.ViewGroup): android.view.View

The macro creation diagram gives an overview of the macro creation activity. This activity is split up into two views accessible by tabs. The macro view which allows for editing of the visual representation and the the action view which allows for altering of the bound action.

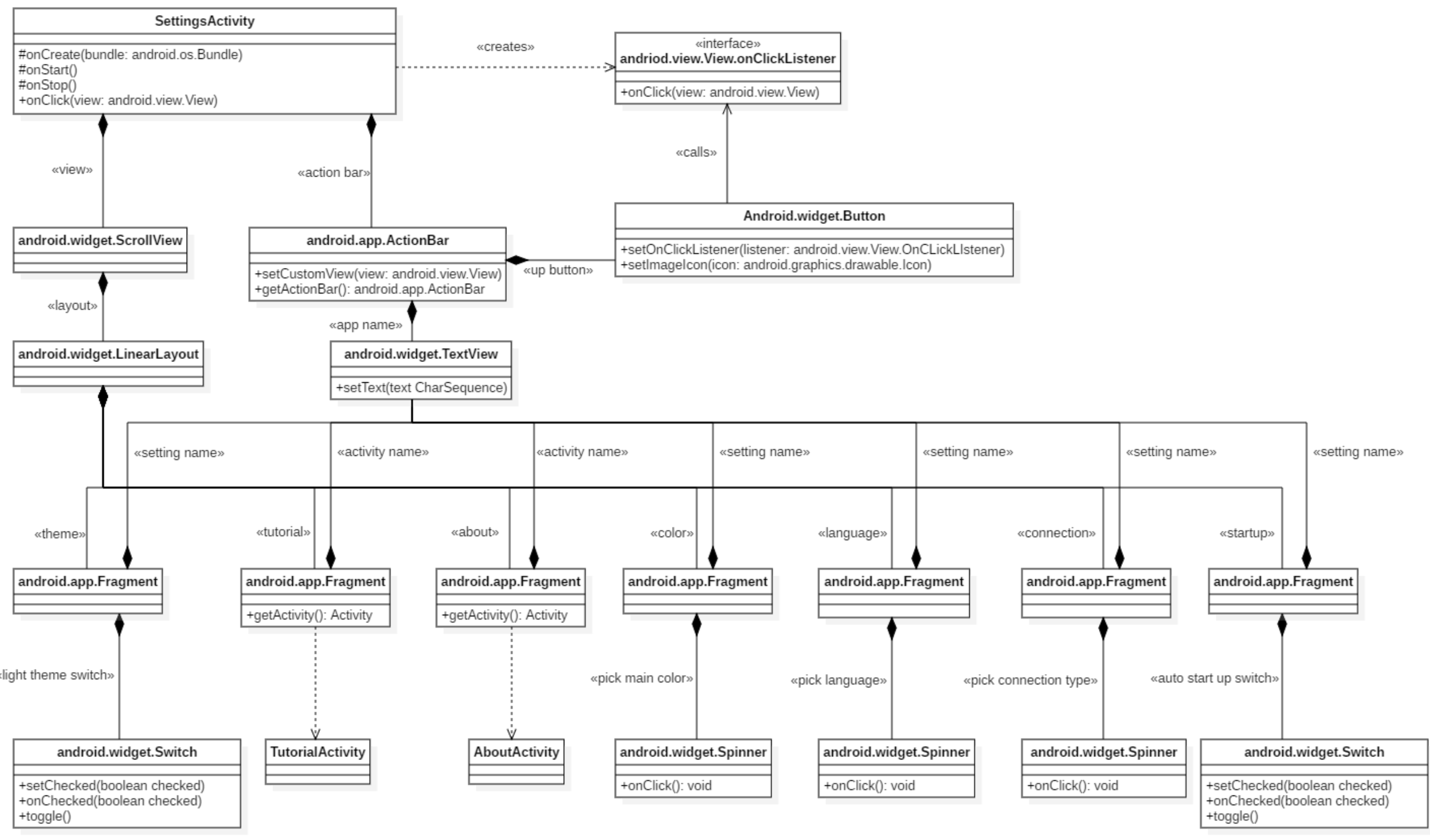
The macro view will feature various TextView which just functions as a text label for the UI. This will make the UI much easier to understand and use. The user has various options to choose from when making a new macro. Such as picking a fore- and background image. Which are done by the methods setForeground and setBackground.

Also the user can add a subscript to the macro button, which is handled by TextInputLayout. As the user is able to add a subscript it is not mandatory by the app, therefore it can be toggled on or off with a switch button. To personalize the macro button to the preferences of the user, the user is able to upload an image or change the color of the added subscript.

The action creation section of the diagram gives an overview of the action creation view. This view can be seen as an equivalent version of the macro creation view. Only this activity is for actions and will feature another screen than the macro creation view. This view can be accessed by clicking on the tab shortcut.

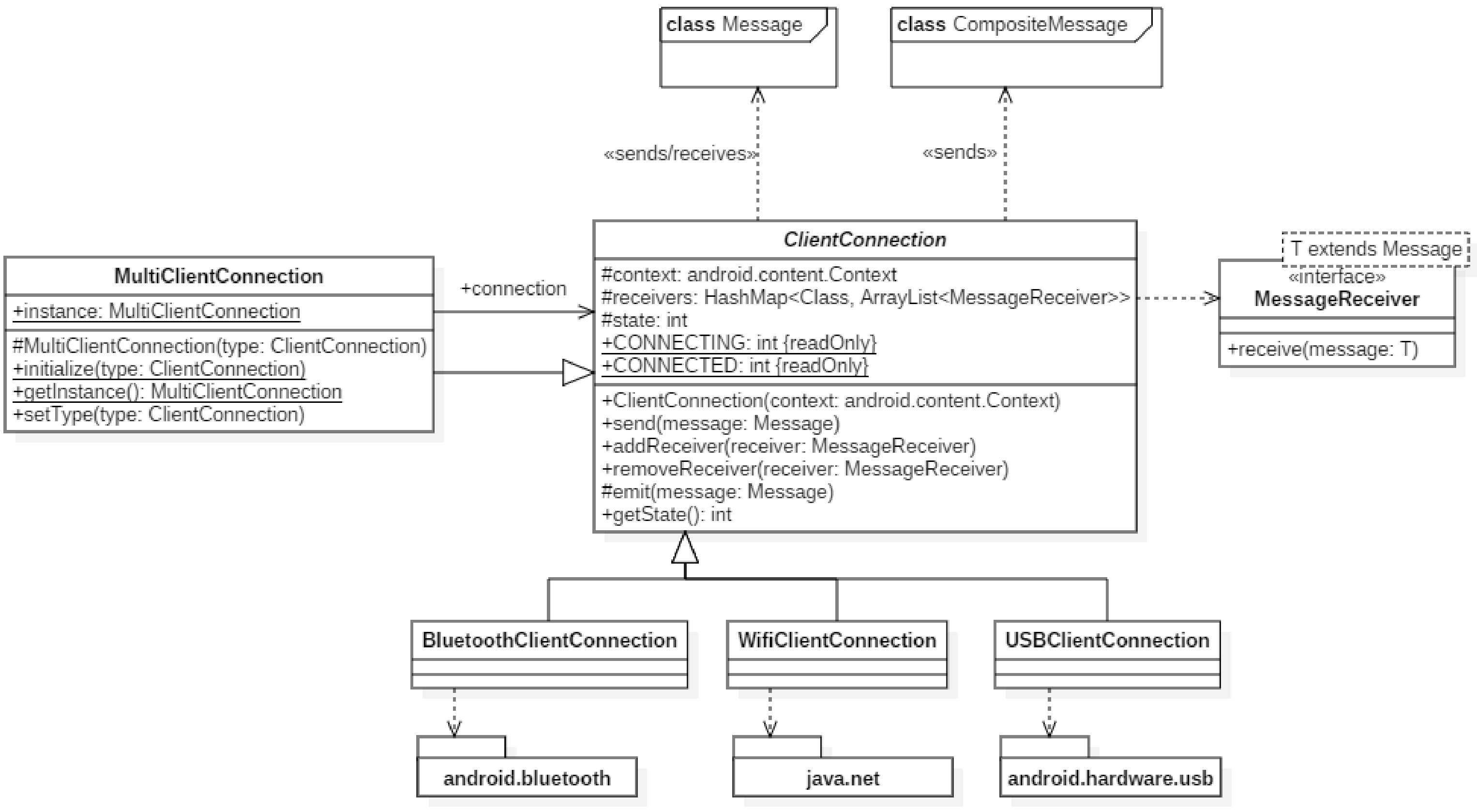
As by the macro creation equivalent, the user has again some options to create their own action. The user can also make their own shortcut sequence by pressing the FloatingActionButton. This will let the user create powerful shortcuts. Also the user is able to name the action, hence the TextInputEditText class will handle this for the user. Also the UI of this activity will feature some TextViews again just to make sure that the user understands the UI completely.
As the user can create powerful shortcut sequences they also should be able to set some repeat amount on their sequence. To handle this, the spinner will feature some numbers from, say for example, zero to five.

Settings



The Settings Class Diagram gives an overview of the Settings Activity. Upon opening it, the user will see a ScrollView containing a LinearLayout of all the Fragments, containing all possible settings options, as well as options to view the About Activity or the Tutorial using Fragment's getActivity method. The fragments also include TextViews containing the names of the settings options.
The settings options include three Spinners. One of them will allow changing the language of the application, where English is the default and the user can also pick Dutch or Bulgarian. The second one will allow the user to change the main colour of the application and the third one can be used to change the connection type. They also include two Switches, one for switching between a Light and Night Themes and the other enabling automatic launch of the PC application upon an active connection between the phone and PC.
The Settings Activity also has an ActionBar that contains a TextView with the name of the application, as well as an Up Button. The Up button is similar to the BackButton that all Android devices have either in their bottom left or bottom right screen corner, but it is not a system-wide button and only allows navigation throughout the application. In this case, when pressed, the Up Button will go back to the Home activity.

Client connection

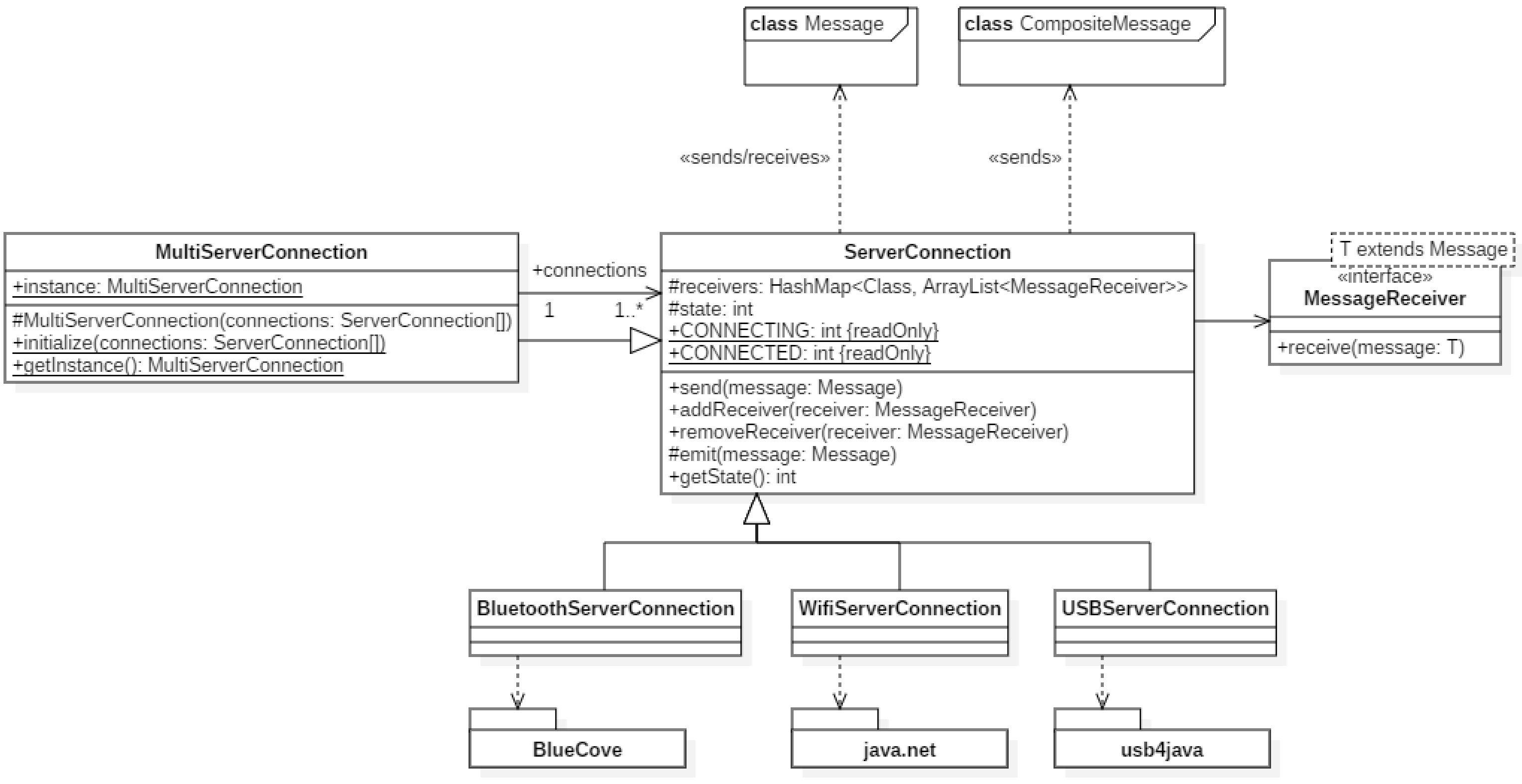

class Message
class CompositeMessage
«sends/receives»
«sends»
ClientConnection
#context: android.content.Context
#receivers: HashMap<Class, ArrayList<MessageReceiver>>
#state: int
+CONNECTING: int {readOnly}
+CONNECTED: int {readOnly}
+ClientConnection(context: android.content.Context)
+send(message: Message)
+addReceiver(receiver: MessageReceiver)
+removeReceiver(receiver: MessageReceiver)
#emit(message: Message)
+getState(): int
MultiClientConnection
+instance: MultiClientConnection
#MultiClientConnection(type: ClientConnection)
+initialize(type: ClientConnection)
+getInstance(): MultiClientConnection
+setType(type: ClientConnection)
+connection
T extends Message
«interface»
MessageReceiver
+receive(message: T)
BluetoothClientConnection
WifiClientConnection
USBClientConnection
android.bluetooth
java.net
android.hardware.usb

The MultiClientConnection class is a singleton class that can be used throughout the android app in order to send and receive data to and from the server. It has to be initialized with a ClientConnection instance before the getInstance method can be called. This initialization happens in the splash activity. Dependent on the connection type stored in the database, a different ClientConnection will be set. This connection type can also be changed at any point when the user alters the corresponding setting. For the usb and bluetooth connections, android APIs will be used. The local wifi connection can be established using purely native java APIs.

The class has two responsibilities: sending and receiving messages.
In order to receive messages using the class, receiver instances can be added and removed. These receivers take a generic type T indicating on what message type its receive method will be invoked. The class will be able to have multiple receivers for the same message types, and invoke all of them. The class has a protected method that can be called to emit a passed message to all of its receivers.

To send a message, one only has to call the send message with a message as an argument. Additionally, the class has a method to return its state, which indicates whether or not it has established a connection.

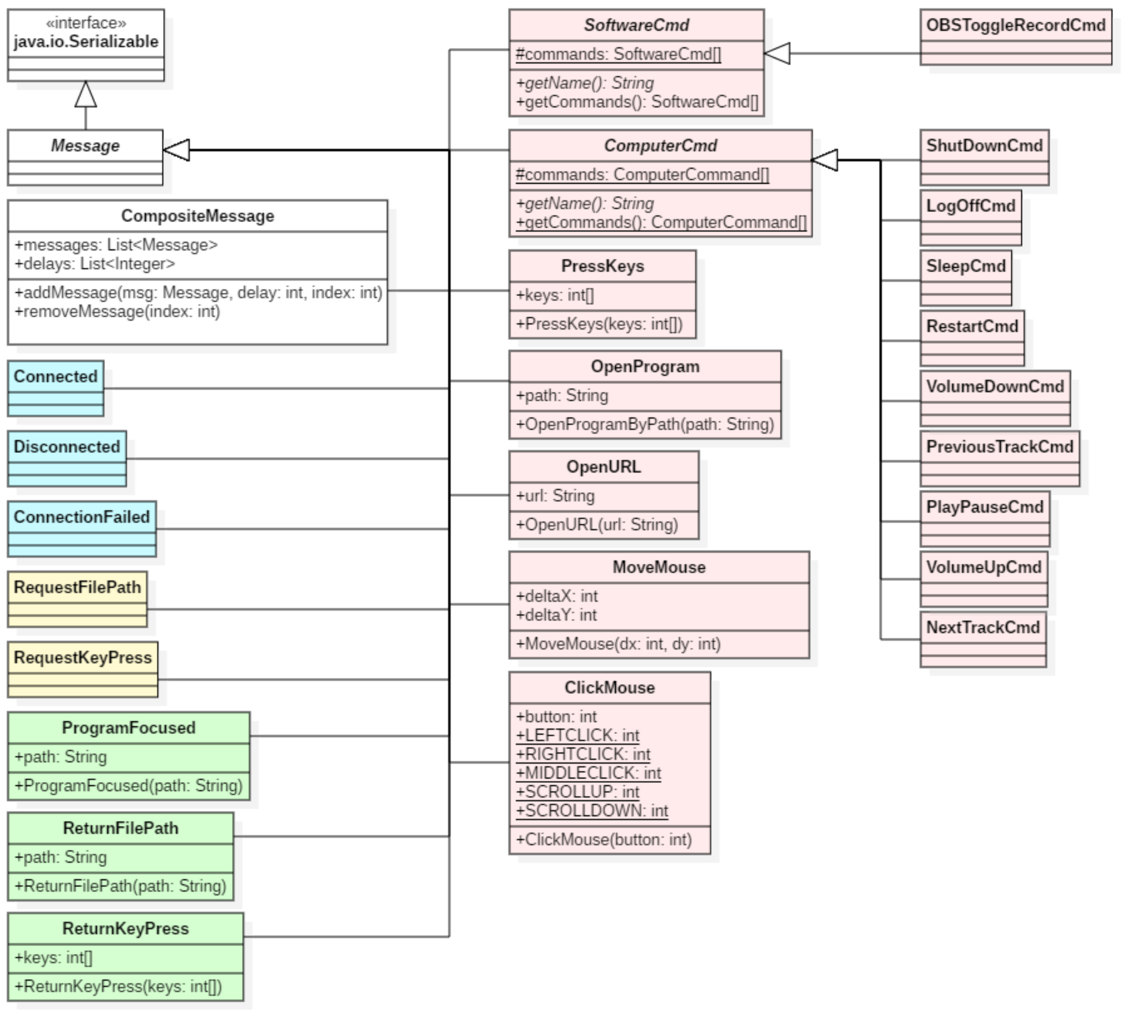
Server connection


class Message
class CompositeMessage
«sends/receives»
«sends»
MultiServerConnection
+instance: MultiServerConnection
#MultiServerConnection(connections: ServerConnection[])
+initialize(connections: ServerConnection[])
+getInstance(): MultiServerConnection
+connections
1
1..*
ServerConnection
#receivers: HashMap<Class, ArrayList<MessageReceiver>>
#state: int
+CONNECTING: int {readOnly}
+CONNECTED: int {readOnly}
+send(message: Message)
+addReceiver(receiver: MessageReceiver)
+removeReceiver(receiver: MessageReceiver)
#emit(message: Message)
+getState(): int
T extends Message
«interface»
MessageReceiver
+receive(message: T)
BluetoothServerConnection
WifiServerConnection
USBServerConnection
BlueCove
java.net
usb4java

The MultiServerConnection class is rather analogues to the MultiClientConnection class, except that it will store instances of all possible connection types at once. This allows it to always establish a connection to the phone, even if the phone decides to change the connection type. If the server connection only had a single connection type enabled at once, the user would have to change the settings on both the phone and the computer which is not desirable.

The responsibilities and the way the class is interfaced with is equivalent to those of the MultiClientConnection class.

Messages



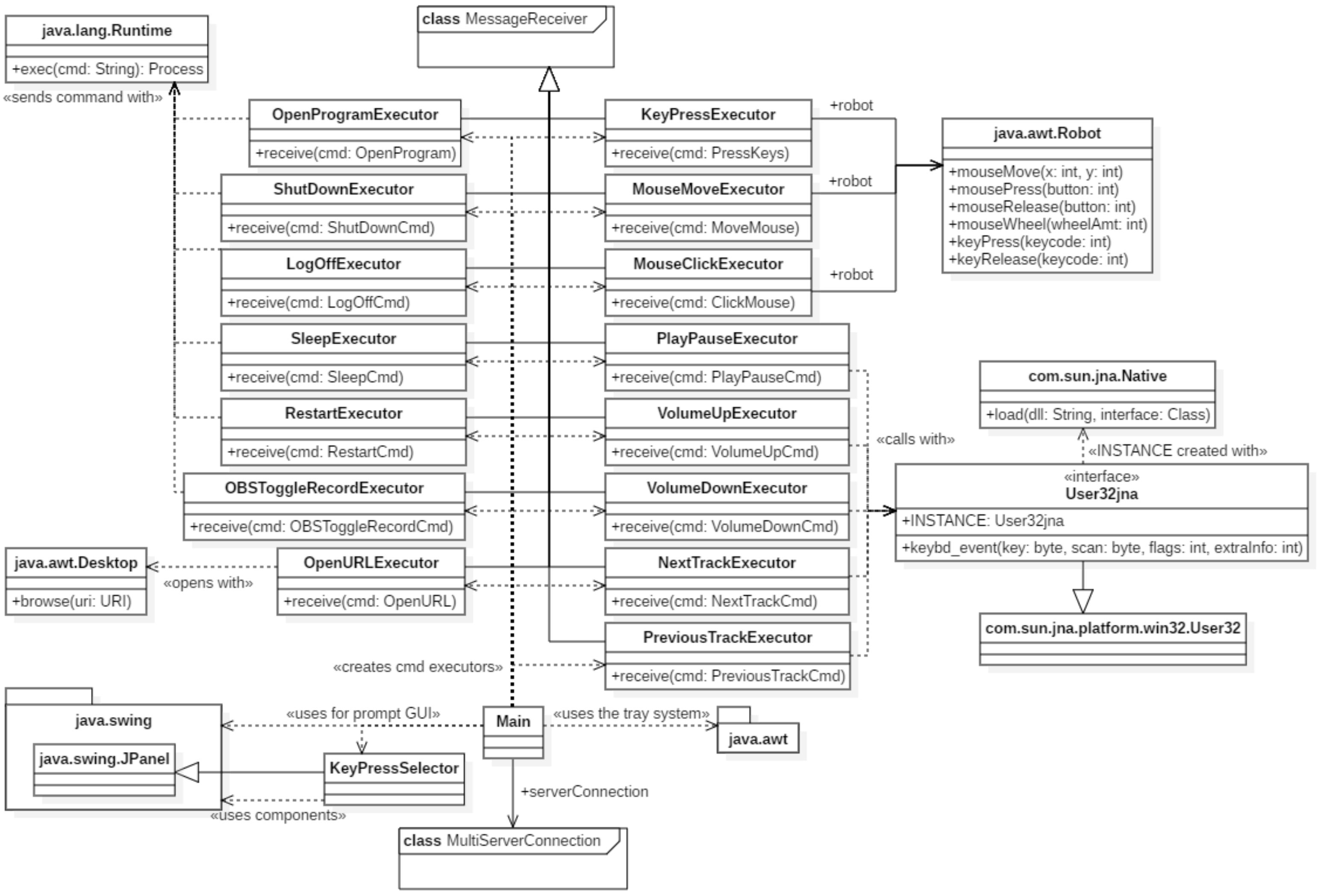
The app defines one abstract core Message class, which is extended by many concrete Message classes. The core class implements the java serializable interface, allowing messages to easily be serialized and either transferred over the network, or saved in our database.

A CompositeMessage class is provided that allows to store a sequence of messages with a specific delay that should be between them. The connection classes will recognize this message type specifically, and are able to extract the data in order to send the individual message with the correct delays.

The remaining class are separated in 4 groups:

- Red: The specific macros sent by the client that will be received on the server
- Blue: The connection event messages to inform the client and server about connection changes
- Yellow: The data prompts that the client can send to the server, to request data from the server
- Green: The data that the server can provide to the client, either return values of prompts or based on computer events

Server



The server will have one Main root class that sets everything up. This only consists of 2 components:

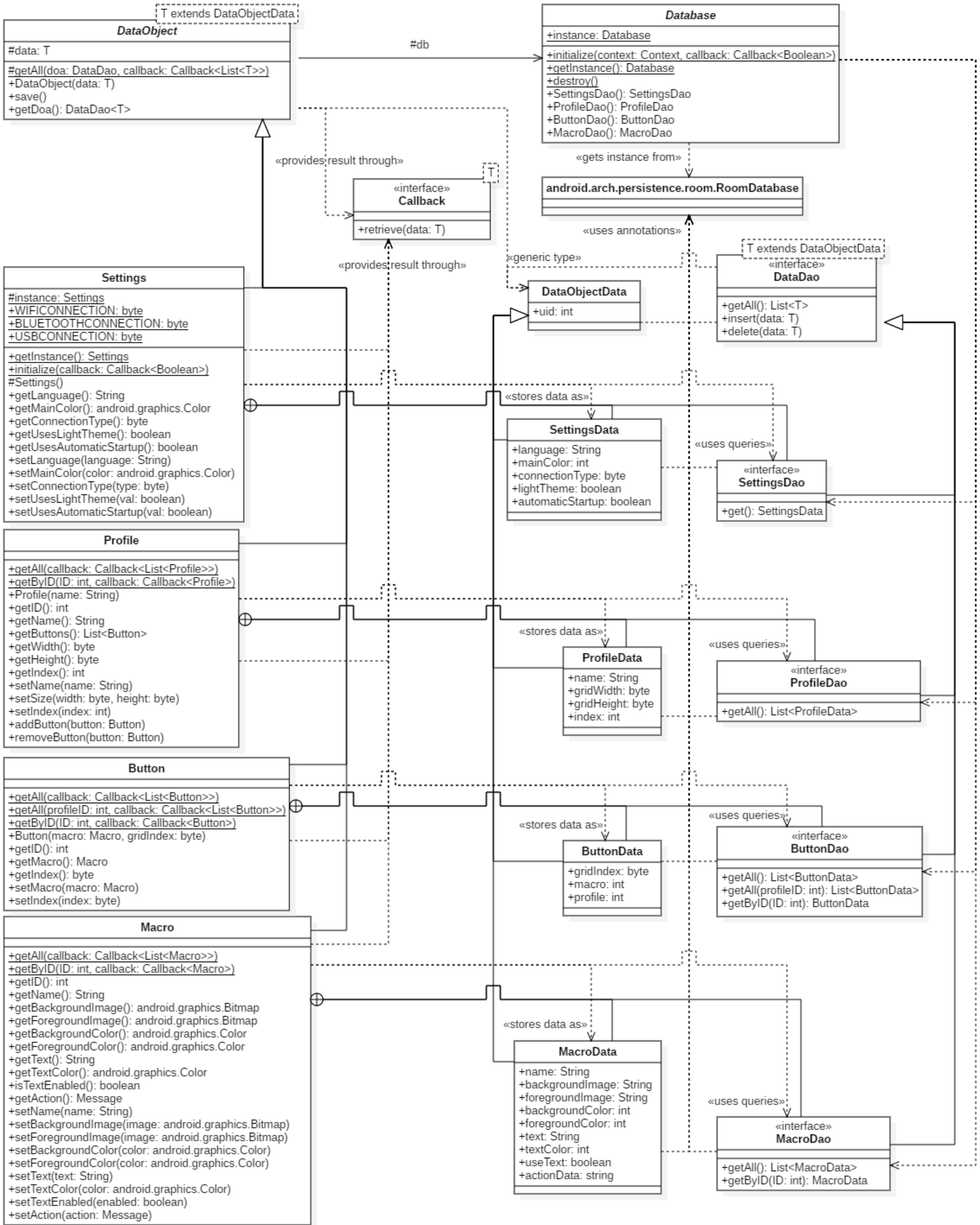
- Creating a simple tray menu for interaction with the software
- Initializing the connection singleton by providing all receivers:
    - Instances of all the command executors
    - Anonymous classes for connection events (because the behaviour will be trivial)
- Setting up a loop that checks what program has focus, and sending an event to the client once focus changes

The server uses several different systems in order to execute its macros:

- Java's Robot class is used to execute simple key presses, mouse movement and clicks
- Java's Desktop class is used to open a URL using the user's default browser

- Java’s Runtime class is used to execute low level operations like turning off the computer, or opening programs.
    - The more advanced software integration macros will have to be written in another language (E.g. AHK) and opened from java, as java isn’t well suited for native interactions that are required for this.
- JNA is used for media controls, as well as potentially some future low level operations (If the program would be extended after the course). This is an alternative to using a seperate exe file, but is more limiting. It however allows us to keep more of our code to be written in java, yet provide some native connection to the computer.

Database


T extends DataObjectData
DataObject
#data: T
#getAll(doa: DataDao, callback: Callback<List<T>>)
+DataObject(data: T)
+save()
+getDoa(): DataDao<T>
#db
Database
+instance: Database
+initialize(context: Context, callback: Callback<Boolean>)
+getInstance(): Database
+destroy()
+SettingsDao(): SettingsDao
+ProfileDao(): ProfileDao
+ButtonDao(): ButtonDao
+MacroDao(): MacroDao
«provides result through»
T
«interface»
Callback
+retrieve(data: T)
«gets instance from»
android.arch.persistence.room.RoomDatabase
«uses annotations»
«provides result through»
«generic type»
T extends DataObjectData
«interface»
DataDao
+getAll(): List<T>
+insert(data: T)
+delete(data: T)
DataObjectData
+uid: int
Settings
#instance: Settings
+WIFICONNECTION: byte
+BLUETOOTHCONNECTION: byte
+USBCONNECTION: byte
+getInstance(): Settings
+initialize(callback: Callback<Boolean>)
#Settings()
+getLanguage(): String
+getMainColor(): android.graphics.Color
+getConnectionType(): byte
+getUsesLightTheme(): boolean
+getUsesAutomaticStartup(): boolean
+setLanguage(language: String)
+setMainColor(color: android.graphics.Color)
+setConnectionType(type: byte)
+setUsesLightTheme(val: boolean)
+setUsesAutomaticStartup(val: boolean)
«stores data as»
SettingsData
+language: String
+mainColor: int
+connectionType: byte
+lightTheme: boolean
+automaticStartup: boolean
«uses queries»
«interface»
SettingsDao
+get(): SettingsData
Profile
+getAll(callback: Callback<List<Profile>>)
+getByID(ID: int, callback: Callback<Profile>)
+Profile(name: String)
+getID(): int
+getName(): String
+getButtons(): List<Button>
+getWidth(): byte
+getHeight(): byte
+getIndex(): int
+setName(name: String)
+setSize(width: byte, height: byte)
+setIndex(index: int)
+addButton(button: Button)
+removeButton(button: Button)
«stores data as»
ProfileData
+name: String
+gridWidth: byte
+gridHeight: byte
+index: int
«uses queries»
«interface»
ProfileDao
+getAll(): List<ProfileData>
Button
+getAll(callback: Callback<List<Button>>)
+getAll(profileID: int, callback: Callback<List<Button>>)
+getByID(ID: int, callback: Callback<Button>)
+Button(macro: Macro, gridIndex: byte)
+getID(): int
+getMacro(): Macro
+getIndex(): byte
+setMacro(macro: Macro)
+setIndex(index: byte)
«stores data as»
ButtonData
+gridIndex: byte
+macro: int
+profile: int
«uses queries»
«interface»
ButtonDao
+getAll(): List<ButtonData>
+getAll(profileID: int): List<ButtonData>
+getByID(ID: int): ButtonData
Macro
+getAll(callback: Callback<List<Macro>>)
+getByID(ID: int, callback: Callback<Macro>)
+getID(): int
+getName(): String
+getBackgroundImage(): android.graphics.Bitmap
+getForegroundImage(): android.graphics.Bitmap
+getBackgroundColor(): android.graphics.Color
+getForegroundColor(): android.graphics.Color
+getText(): String
+getTextColor(): android.graphics.Color
+isTextEnabled(): boolean
+getAction(): Message
+setName(name: String)
+setBackgroundImage(image: android.graphics.Bitmap)
+setForegroundImage(image: android.graphics.Bitmap)
+setBackgroundColor(color: android.graphics.Color)
+setForegroundColor(color: android.graphics.Color)
+setText(text: String)
+setTextColor(color: android.graphics.Color)
+setTextEnabled(enabled: boolean)
+setAction(action: Message)
«stores data as»
MacroData
+name: String
+backgroundImage: String
+foregroundImage: String
+backgroundColor: int
+foregroundColor: int
+text: String
+textColor: int
+useText: boolean
+actionData: string
«uses queries»
«interface»
MacroDao
+getAll(): List<MacroData>
+getByID(ID: int): MacroData

The database makes use of the Android's Room library. It allows us to set up an SQL database using rather high level definitions. This way new object types can be added to the database with relative ease, yet have high performance (due to using a SQL based database).

Every object type consists of 3 parts:

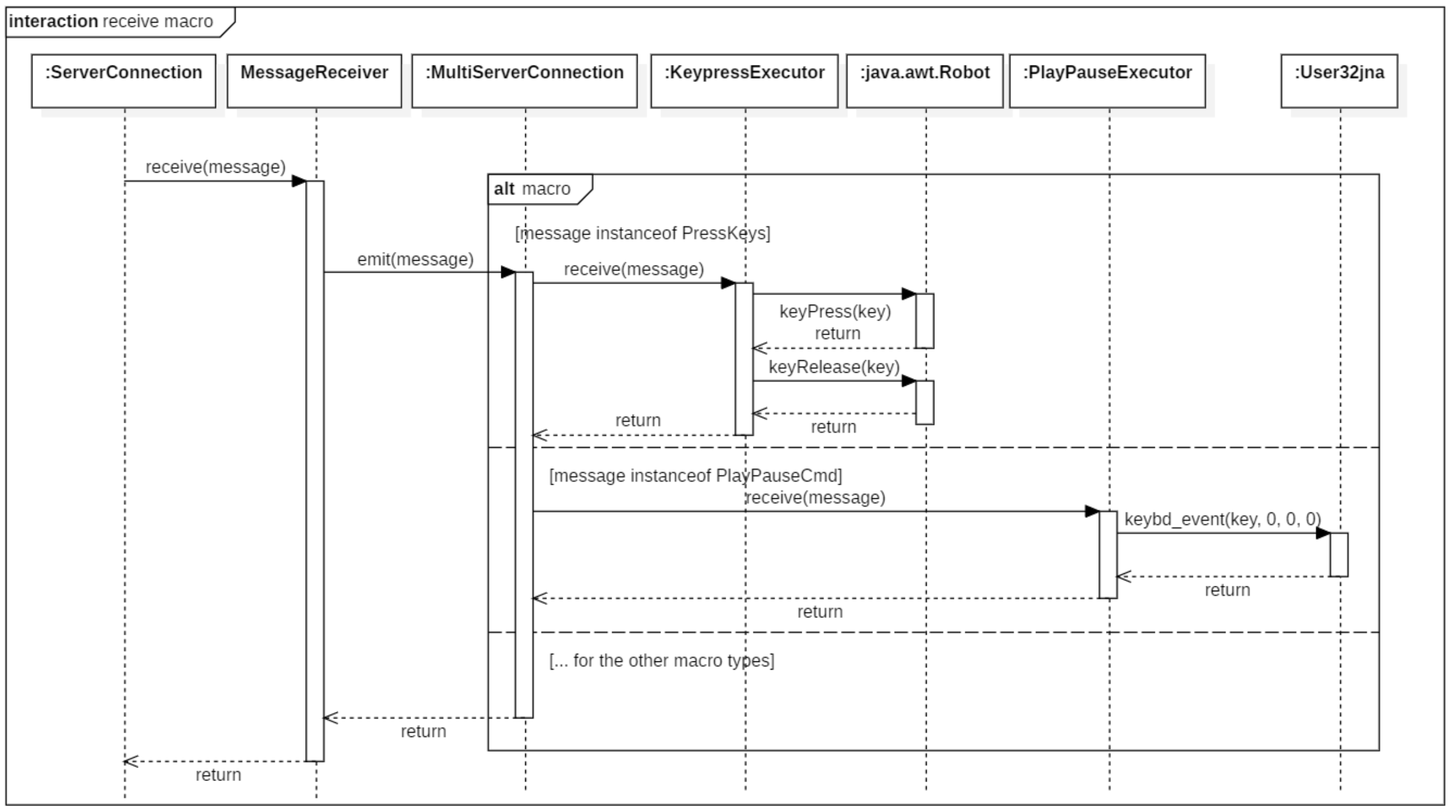
- The high level class that defines a usual java way of retrieving and changing the data of the object
- A data class that describes exactly what fields the database should store, and what type this data has
- A dao interface that defines how the data class instances are converted to SQL data rows and vice versa

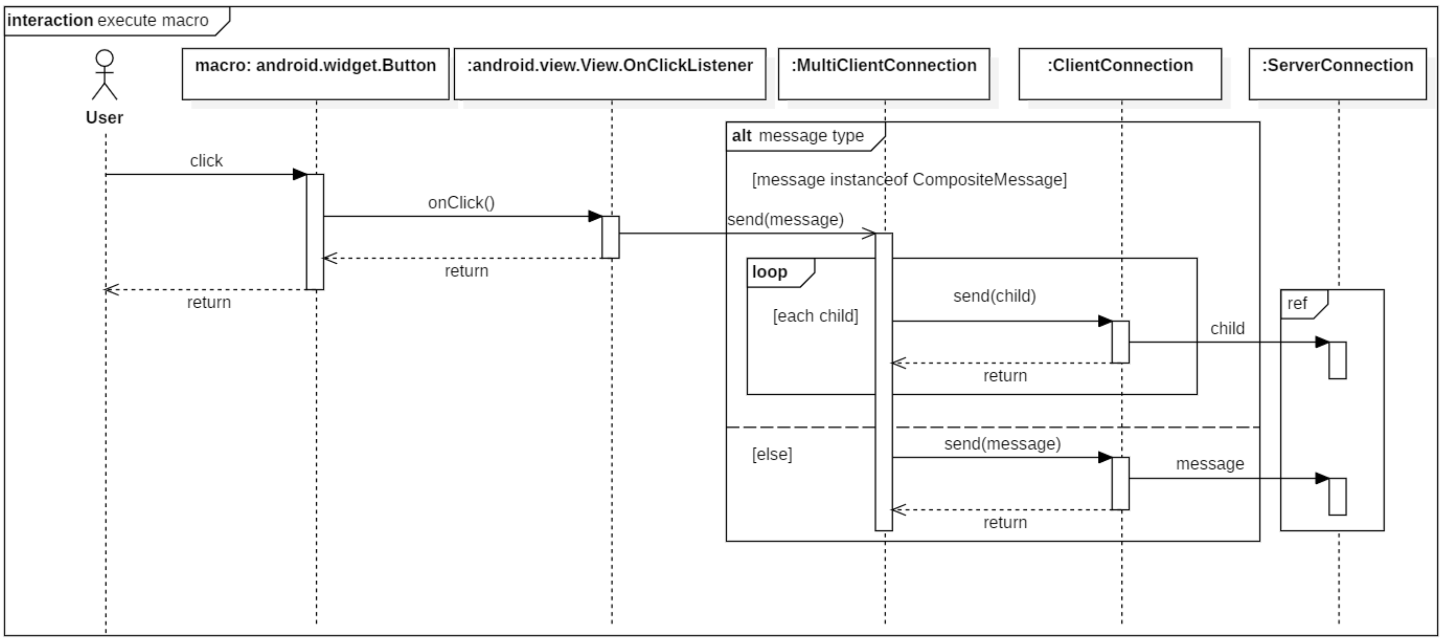
The database stores 4 object types:

- Settings: The settings data of which only a single instance will exist, that stores all of the user's global data. This could be extended to allowing for multiple settings 'profiles' in the future.
- Profile: The macro profiles of the user, which store the index in the profile list, the profile name, and the profile's grid size. This class automatically establishes a connection with the Button data type, such that the profile's buttons can be altered as well.
- Button: The buttons that are contained on a profile. A button is simply the relationship between a macro and a profile. A button is a macro placed in a profile, at a specific index. Because macros can be included in multiple profiles, and profiles include multiple macros. This class automatically establishes a connection with the Macro data type such that the macro can be extracted.
- Macro: The macro that can be executed on the server. Stores the message that will be send to the server, as well as the visual representation of the macro within the app.

Sequence diagrams

Macro execution



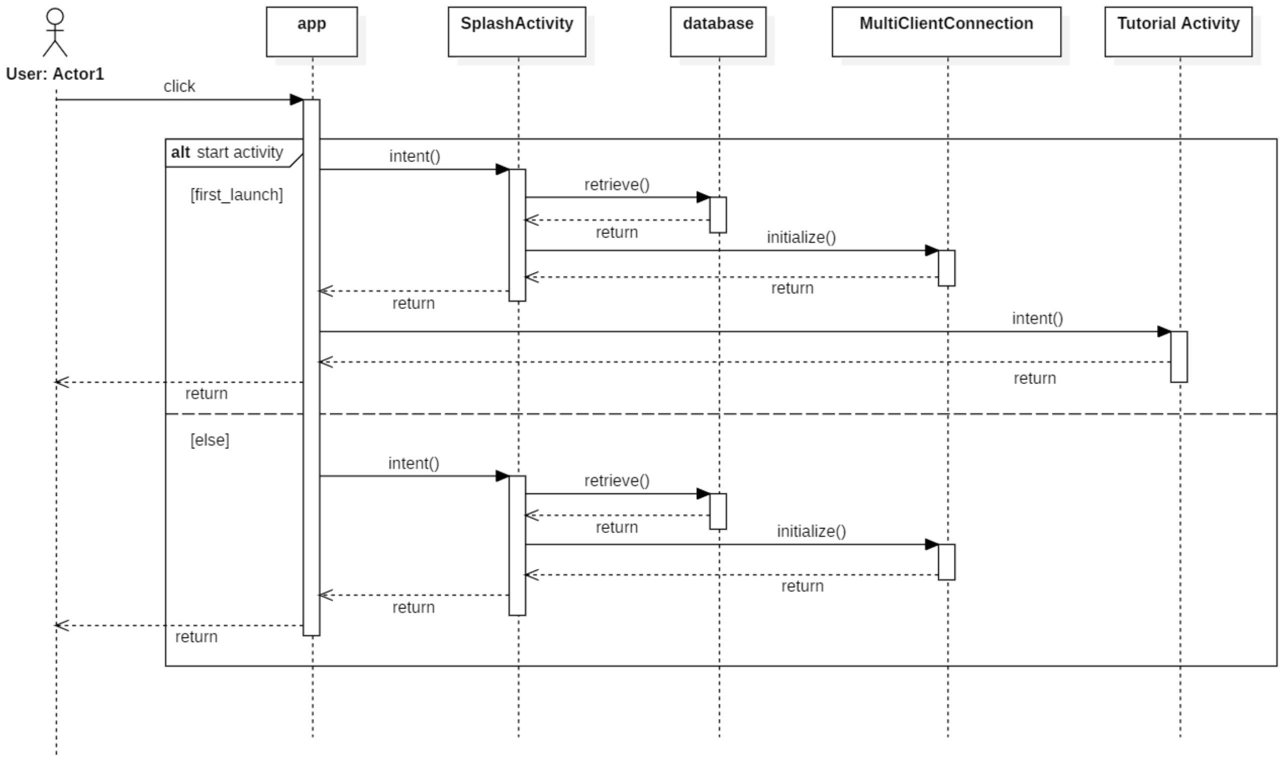


When the use clicks a macro button in the profile activity, said macro should execute on the computer. To achieve this, the message is sent using the MultiClientConnection class. This class checks whether the message is a composite message or a normal message. If it’s a normal message, it will be forwarded to a specific ClientConnection class, which depends on the selected connection type. If it's a composite message, the messages will be extracted and forwarded to the ClientConnection class as well, with the correct interval.

The details on what every connection class does have been left out, as all three of them will simply call different methods on the library/APIs that we rely on. The same goes for the speciific ServerConnection implementations.

When a ServerConnection receives the macro, it will pass it to the MultiServerConnection which will forward it to the corresponding macro executor. In the sequence diagram two examples are given, but this would obviously contain all executors. The individual executor then makes use of its external dependency to execute the macro.

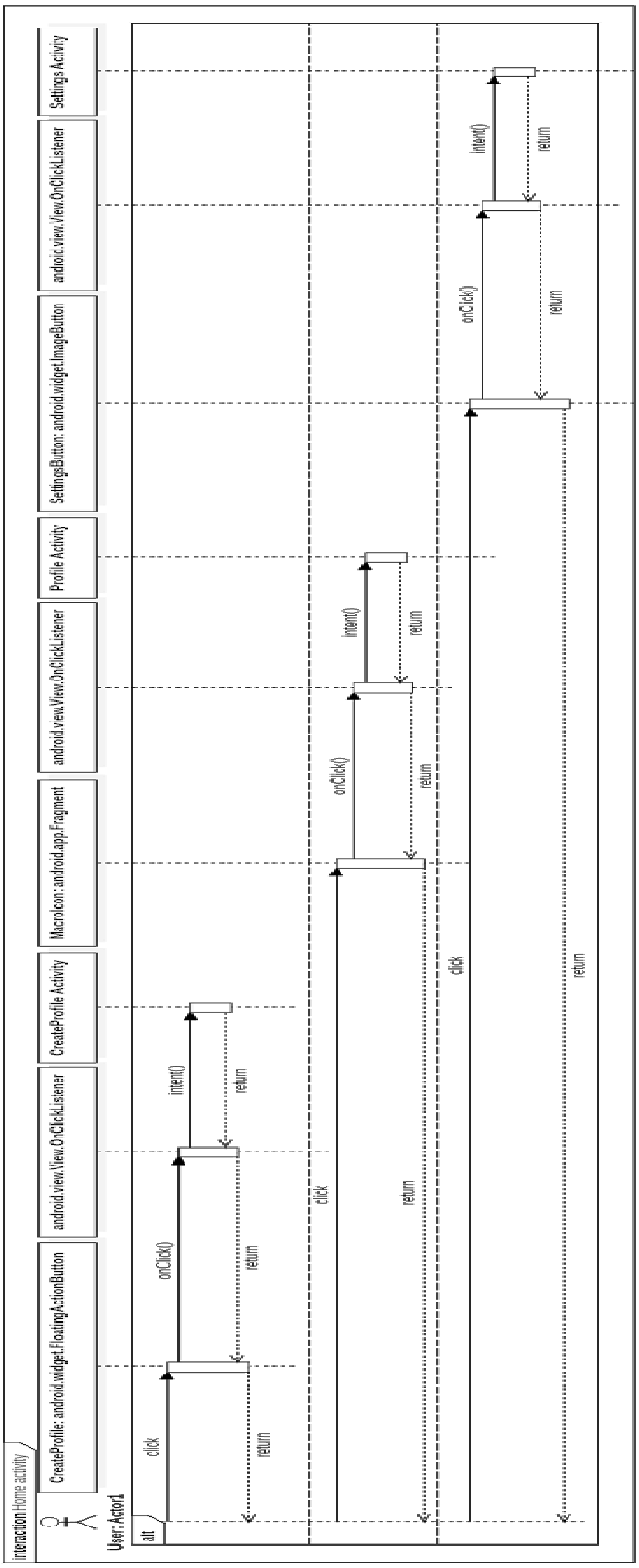
Splash activity



When a user clicks (or launch) the application, it will start the splash activity with the intent call. While splash activity is active, it requires to make requests to both database and MultiClientConnection at the almost concurrent rate. It retrieves the data from the database and initializes MultiClientConnection for any data transfer required in the app. Once this process is done, the splash activity ends and directs to the first page (home) of the app.

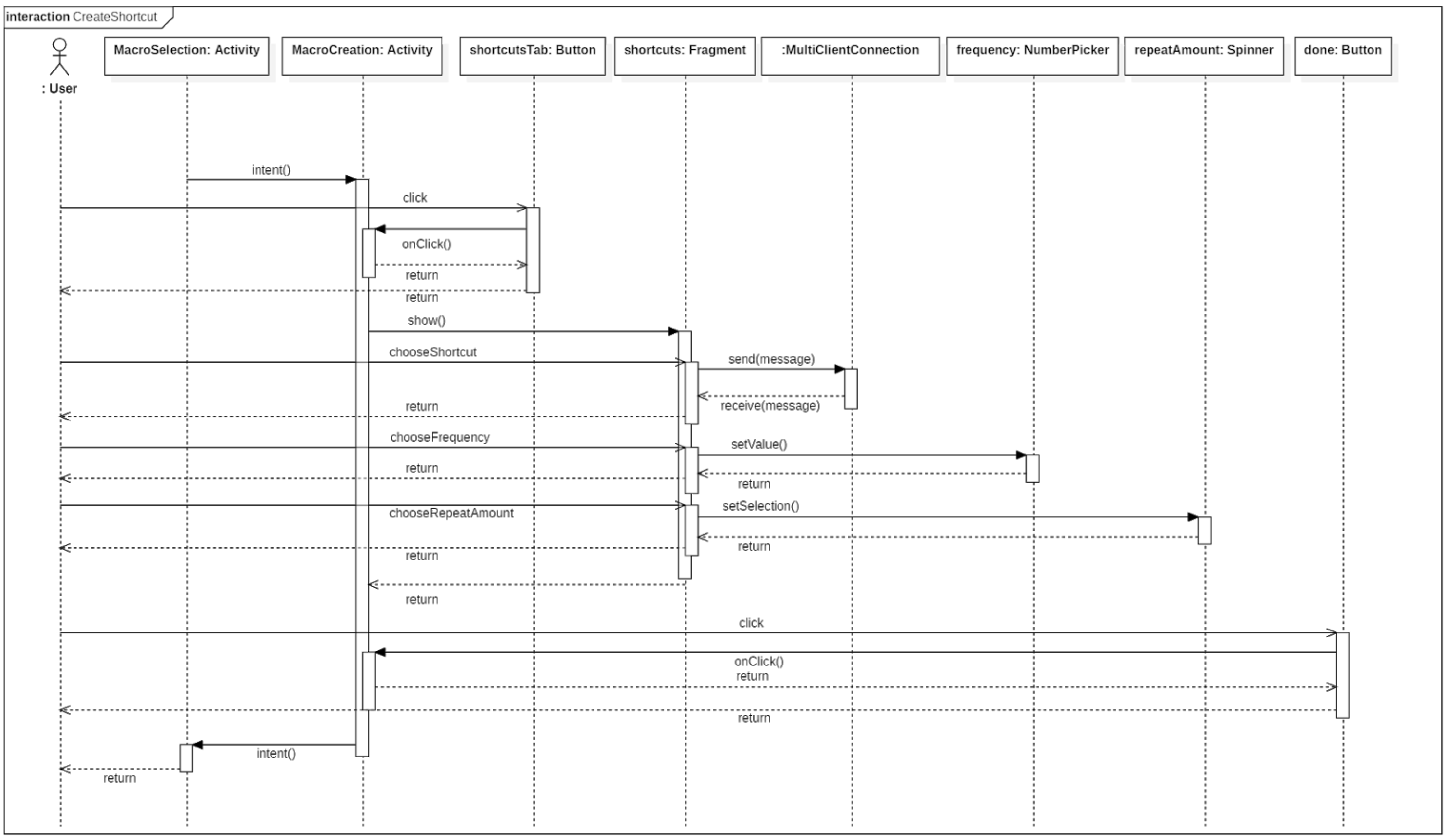
The diagram also depicts an extra sequence that tutorial activity is executed after the tutorial activity upon the very first launch of application. Otherwise it will not be visible and only found in the setting.

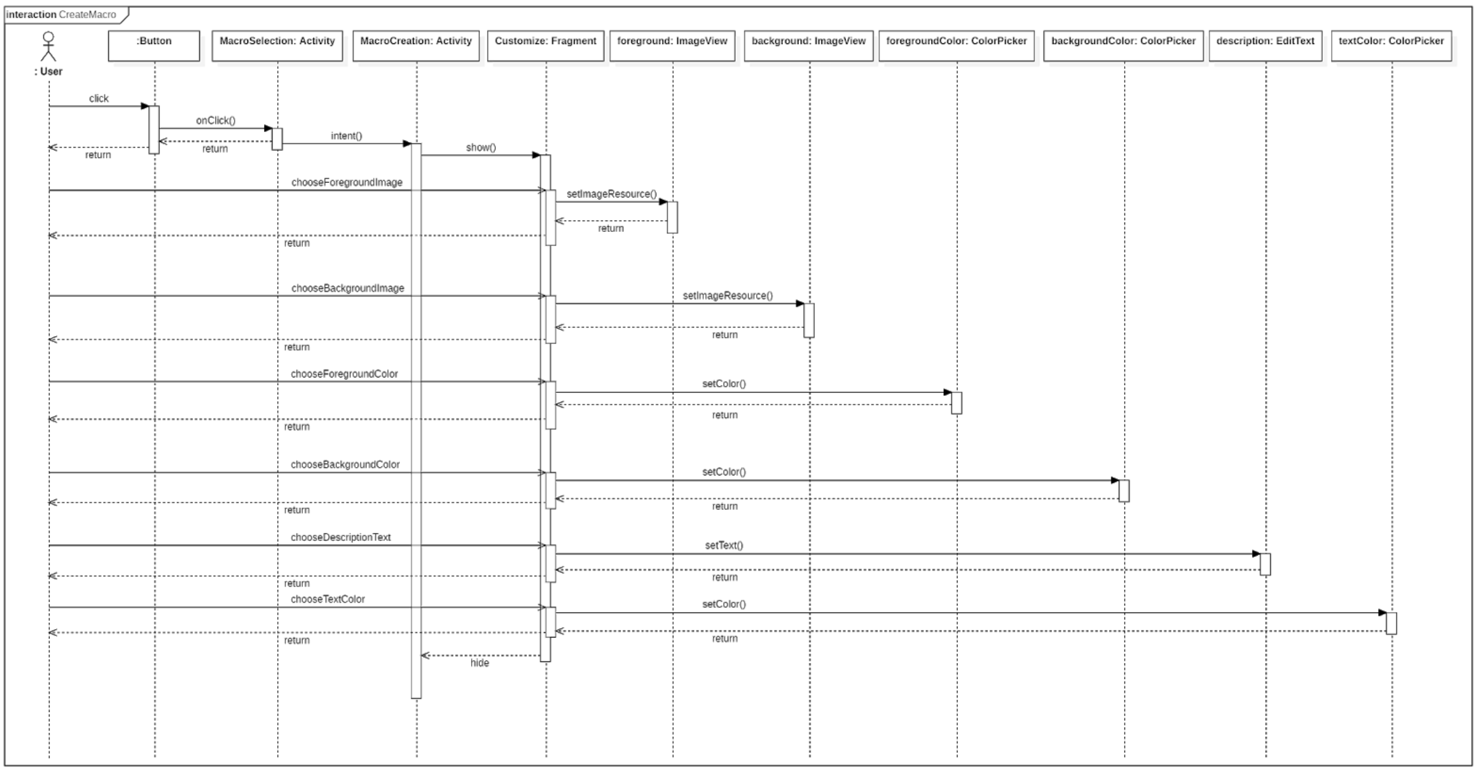
Home activity


interaction Home activity
User: Actor1
CreateProfile: android.widget.FloatingActionButton
android.view.View.OnClickListener
CreateProfile Activity
MacroIcon: android.app.Fragment
android.view.View.OnClickListener
Profile Activity
SettingsButton: android.widget.ImageButton
android.view.View.OnClickListener
Settings Activity
alt
click
onClick()
intent()
return
return
return
click
onClick()
intent()
return
return
return
click
onClick()
intent()
return
return
return

The Home activity is also the Open profile activity of the application. When opened the user has three alternative actions they can take. The first one is clicking on the FloatingActionButton. This button opens the CreateProfile activity. The second option is picking one of the already existing profiles. Clicking any of them will open the Profile activity corresponding to the profile the user chose. The third and final action this activity allows is clicking the Settings IconButton. This will open the Settings Activity, where the user can change settings like language or connection type.

Macro Creation





Since creating a macro involves quite a numerous amount of steps, we decided to show it in two sequence diagrams. The first diagram focuses on the looks of the macro, i.e. what picture and colors should be used. And the second diagram focuses on the action of the macro, i.e. what happens when you activate the macro. The first diagram starts with a user pressing a button in the MacroSelection activity to open the MacroCreation activity.

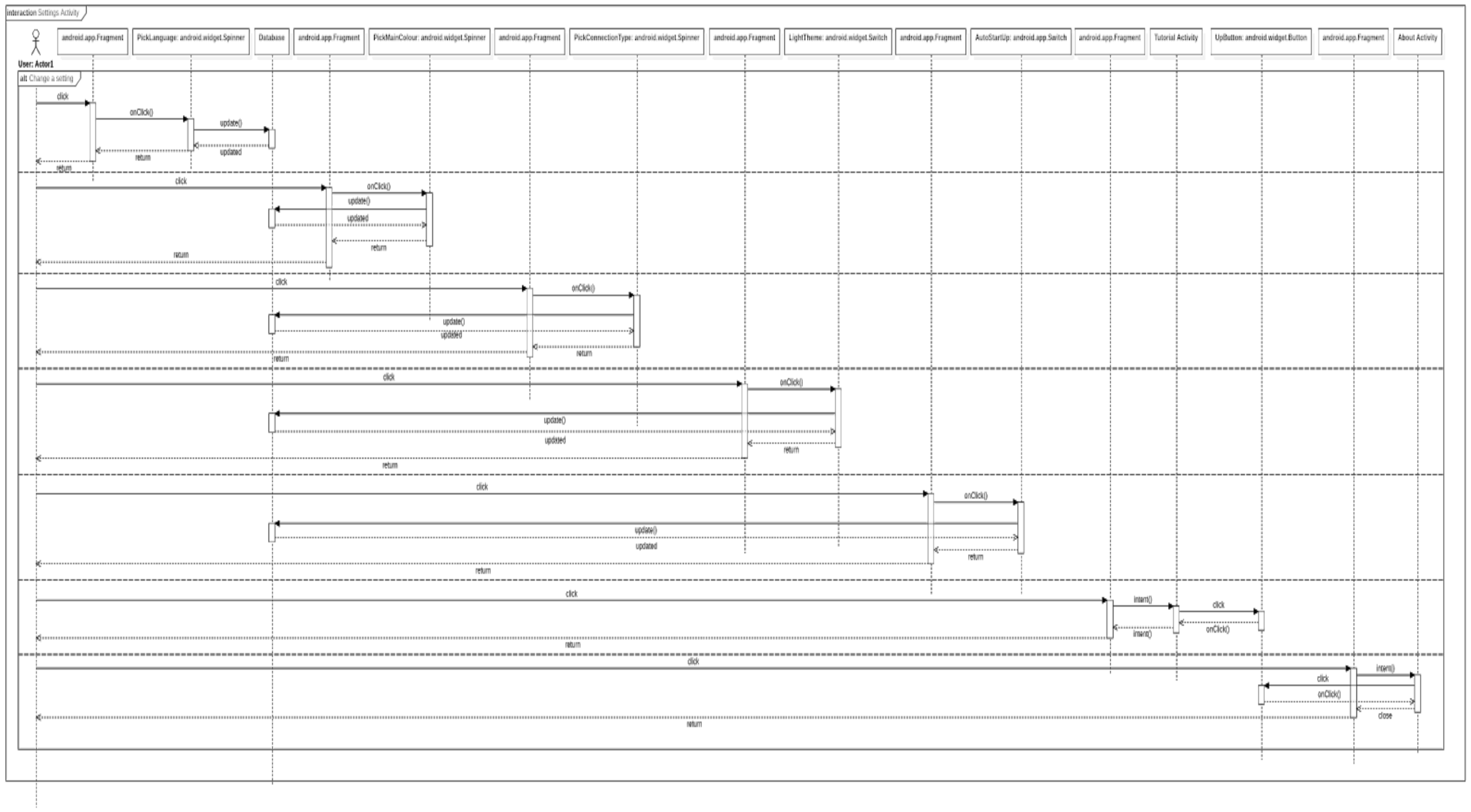
On start, the MacroCreation activity shows the fragment that is used to customize the macro. In this fragment, there are multiple aspects of the macro that the user can alter. The user can set the fore- and background image, and alter the color of these images. The user is also able to set a descriptive text for the macro, and choose the text color for this description.

Since we decided on splitting up the sequence diagram, we had to find a way to get the activation of the MacroCreation activity in the second diagram. For this we used the intent, which is not actually what happens. This is just to get a correct representation for the rest of the diagram. Normally the MacroCreation activity would just still be active as can be seen in the previous diagram. When in the MacroCreation activity the shortcuts tab is pressed, the activity gets a signal, and hides the customize fragment (which can be seen in the first diagram) and shows the shortcuts fragment (which can be seen in the second diagram).

First the user can set the shortcut for the macro. This is the action that will be performed when the macro is pressed. This is done by sending a message via the MultiClientConnection to the connected pc. The application that will run on the pc will then allow you to set the shortcut, and send the shortcut back to your phone. The user can also set the frequency between the actions that a macro performs. For example, one macro can pause a video and turn the sound off of your pc, and with frequency you can manage the time between these two actions.

Afterwards the user can also select how many times he/she would like for the action to be performed, e.g. an user could make a button for “ctrl” + “z” and would like to execute it 4 times with one press. Once the user is done with creating the macro, they can press the done button which will close the MacroCreation activity and open the MacroSelection activity.

Settings


interaction Settings Activity
User: Actor1
android.app.Fragment
PickLanguage: android.widget.Spinner
Database
android.app.Fragment
PickMainColour: android.widget.Spinner
android.app.Fragment
PickConnectionType: android.widget.Spinner
android.app.Fragment
LightTheme: android.widget.Switch
android.app.Fragment
AutoStartUp: android.app.Switch
android.app.Fragment
Tutorial Activity
UpButton: android.widget.Button
android.app.Fragment
About Activity
alt Change a setting
click
onClick()
update()
updated
return
intent()
close

The Settings activity sequence diagram is a rather big one. It is wrapped up in one Combined Fragment representing the alternative actions the user can take. These actions are represented by instances of the different settings option classes, which, when clicked, allow the user to personalise different things like the main colour of the application or switch between the light and night themes. All the updated settings options are then stored in the database and can be retrieved for every launch that follows. Apart from that, the Settings activity also includes fragments that link to the Tutorial and About activities. The user can view them, then go back to the Settings activity.

Design Decisions

The application starts with a tutorial screen after the splash (loading) screen when the application is opened for the first time, to explain the different parts of the application to the user. The tutorial will also ask for the user's preferred use of the application, in order to the best preset profile that corresponds to the user's needs.

Our application has a lot of screens, which means that it also has a lot of Activities. We try to keep the implementation as simple as possible, so we have opted for more Activities and more different UI screens, than wanting to fit everything into a single screen. This would have created an unnecessary increase in work and difficulty, which is not needed for our application.

However, we are using Fragments in our application to reduce the amount of work the application has to do in order to retrieve the macros. The Fragments are in itself equal, but each Fragment has different macros based on the profile the Fragment belongs to. The Fragments are needed to display the macros for each profile on the Profile Activity itself and on the Home Activity, in which the macros are smaller thumbnail-like images.

When there is no sufficient connection between the application and the remote host, a prompt will appear on the user's device instructing them to check their connection settings or change the connection type (Bluetooth (Default) / Wi-Fi / Wired). This prompt is constructed as a fragment, meaning it can easily be displayed on each Activity.

The application supports both a portrait and landscape orientation, but it is not necessary to make a separation in layout for each orientation. Most activities incorporate a *ScrollView* into their base, which makes it easy to adapt the application to a landscape orientation. The only exceptions to this rule are the *Tutorial page* and the *Splash page*, in which we will restrict the user to a portrait only mode. This is to ensure simplicity in the implementation and to unify the experience of the user when launching the application (for the first time). Also, we do not want the screen to update when loading the necessary information to use the application.

Our application sends and receives data to and from a remote host, which is why the application needs a server to handle this data. The server will be located on the remote host and will be initialized by a separate programs executable, which will have been created independently to the Android application. The application does not work without a connection to the server. The application is able to load all profiles and macros, but will not be able to use the macros as there is no receiver to use the macros.

We have also decided to only use the Windows tray to run our remote program, as there is no need to create an entire UI. The mobile application already has all features that are necessary for all operations. The remote program runs a server which will establish a connection with the users device. The server acts a middle-man in order to receive requests from the mobile application regarding the shortcut to be returned and to send the shortcut back to the mobile application to be used for the macro. The server can be seen in the corresponding server diagram.

The macros and profiles will be stored directly on the user's phone and will thus not be available online. All previously created macros (including some preset macros) will be available to choose from in the Macro Selection Activity. This Activity queries the local database for all macros. Because the macros and profiles are stored locally, the user does not have to be only to create, edit, or delete any of the macros or profiles. This increases the accessibility of the app, as there does not have to be a constant connection to the internet or the remote host. Despite the offline capabilities of the application, there still has to be a connection to the remote host in order to create non-default shortcuts or to use the macros themselves.

Should the need arise to add sharing capabilities to the application, then the app can be easily updated by adding a type of codes (e.g. QR-codes) to share the profiles or macros between users. It is not necessary for the application to have an online implementation (already) as most desires after the initial release can be covered without the need for a seperate server element to store all users' shortcuts, macros, and profiles.

Tracing

In the tracing chapter, you can find all the requirements that were formulated in the Requirement Document. This time the requirements are linked with the corresponding class diagrams. In the table below you will find the the ID of the requirement, the priority, the corresponding class and the method/attribute that will be used for the specific requirement. The UML class diagrams can be found in the chapter Class Diagrams.

Some special remarks for some of the requirements. As for the requirements *UR03B1* till *UR03C2* those requirements are marked with N/A, because those requirements are about the OS versions of the server. As the server is written in Java, and there are as many native Java classed used as possible. Known from the capabilities of Java, the server could therefore run on different OS versions. As this cannot be modeled in a UML class diagram it is mentioned here. Then for the requirements *UR07A1* till *UR07A3*, those requirements are very specific profiles. Those presets will be added to the application, but again this cannot be modeled in a UML diagram. Those presets will be present in the database. Then we have the not included ones. The name not included is really trivial, therefore no extra explanation is required.

ID	Priority	Class	Method/ Attribute
UR01	**M**	UserProfile	

UR01A	**M**	UserProfile
		MacroCreation
			uploadImg()
			setForeground()

UR01B	**C**	Not included	
UR02	**M**	OpenProgram	OpenProgramByPath(path: String)
UR03	**M**	Main	
UR03A1	**M**	ClientConnection	ClientConnection(context: android.content.Context)
UR03A2	**S**	ClientConnection	ClientConnection(context: android.content.Context)
UR03A3	**S**	ClientConnection	ClientConnection(context: android.content.Context)
UR03B1	**M**	N/A	
UR03B2	**S**	N/A	
UR03B3	**W**	N/A	
UR03C	**M**	N/A	
UR03C2	**M**	N/A	
UR04	**M**	PickLanguage	pick(String: currentLanguage)
UR04A	**M**	PickLanguage	pick(String: currentLanguage)
UR04B	**S**	PickLanguage	pick(String: currentLanguage)
UR05	**M**	PickMainColour	pick(String: currentColour)
UR05A	**C**	MacroCreation	setColors()
UR05B	**C**	PickMainColour	pick(String: currentColour)
UR06	**M**	Main Message	Multiple functions

UR06A1	**M**	Main Message	Multiple functions
UR06A2	**M**	Main Message	Multiple functions
UR06A3	**M**	Main Message	Multiple functions
UR06A4	**S**	Main Message	Multiple functions
UR06A5	**S**	Main Message	Multiple functions
UR06A6	**C**	Main Message	Multiple functions
UR06B	**M**	MacroCreation	editMacro()
UR06B1	**S**	MacroCreation	uploadImg()
UR06B2	**S**	ActionCreation	setSelection(int)
UR07	**M**	HomeActivity	
UR07A1	**M**	N/A	
UR07A2	**M**	N/A	
UR07A3	**M**	N/A	
UR07B	**M**	MacroSelection	listOfMacros()
UR07B1	**S**	ProfileEditActivity	onDragEvent()
UR07C	**M**	MacroSelection	listOfMacros()
UR07C1	**M**	ProgramFocused	ProgramFocused(path: String)
UR07D	**C**	Profile	Profile(name: String)
UR08A	**C**	Not included	

UR08B	**M**	ProfileEditActivity
		MacroSelection
			addButton(button: Button)
			listOfMacros()

UR08B1	**M**	MacroCreation	getPath()

UR08B2	**M**	ActionCreation	getActiont()
UR09A	**M**	API	API Function
UR09B	**M**	API	API Function
UR09C	**C**	Not included	
UR09C1	**C**	Not included	

UR10A	**C**	VolumeUpExecutor
		VolumeDownExecutor
			receive(cmd: VolumeUpCmd)
			receive(cmd: VolumeDownCmd)

UR10B	**C**	MouseClickExecutor	receive(cmd: ClickMouse)
UR11A	**S**	MouseMoveExecutor	receive(cmd: MoveMouse)
UR11A1	**S**	Not included	
UR11B	**C**	KeyPressExecutor	receive(cmd: PressKeys)
UR11B1	**C**	Not included	

UR12A	**M**	Settings
		PickLanguage
		PickMainColour
			pick(String: currentLanguage)
			pick(String: currentColour)

UR12B	**M**	HomeActivity	
UR12C	**M**	MacroSelection	listOfMacros()