AAL Programme

SUCCESS - **SU**ccessful **C**aregiver **C**ommunication and **E**veryday **S**ituation **S**upport in dementia care

PROJECT IDENTIFICATION	
PROJECT NUMBER	AAL-2016-089
DURATION	1st March 2017 – 29th February 2020
COORDINATOR	Markus Garschall
COORDINATOR ORGANIZATION	AIT Austrian Institute of Technology GmbH
WEBSITE	www.success-aal.eu

SUCCESS

DOCUMENT IDENTIFICATION	
DELIVERABLE ID	D4.1b Functional specification and integrated architecture support
RELEASE NUMBER / DATE	v1.0 / 16.08.2018
CHECKED AND RELEASED BY	Markus Garschall (AIT Austrian Institute of Technology GmbH)

KEY INFORMATION FROM 'DESCRIPTION OF WORK'	
DELIVERABLE DESCRIPTION	Report on the functional specification and architecture of SUCCESS. This deliverable summarizes the first version of architecture and specification definition based on the output produced so far by the Use Case Definitions
DISSEMINATION LEVEL	Public
DELIVERABLE TYPE	Report
ORIGINAL DUE DATE	M17 / 31/07/2018

AUTHORSHIP & REVIEWER INFORMATION	
EDITOR	Ntalaperas Dimitrios, Vafeiadis Georgios (Singular Logic Romania Computer Application SRL)
PARTNERS CONTRIBUTING	exthex GmbH, AIT Austrian Institute of Technology GmbH, University of Cyprus, Citard Services Ltd
REVIEWED BY	Sabrina Stani (exthex GmbH)

ABBREVIATIONS

ABBREVIATIONS	DESCRIPTION
APK	Android Package Kit
DM	Dialogue Manager
CP	Content Provider
CR	Content Repository

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

AAL PROJECT SUCCESS

In the European research project SUCCESS (SUccessful Caregiver Communication and Everyday Situation Support in dementia care), an innovative mobile training application is being developed. It aims at supporting caregivers of people with dementia (PwD). The users of the application are introduced to evidence-based communication and intervention strategies by reading articles, engaging in conversations with an avatar, and listening to lectures presented by an avatar. This format of learning and the multimodal user interface of the app supports different usage situations and contexts. All implemented features are believed to increase the quality of communication and interaction of care persons with PwD and minimize burden of care. This is done by fostering a deeper understanding for PwD (e.g. understanding why PwD can become aggressive) and supporting the caregiver with useful situation-related suggestions. A remarkable feature of the app is that it is not only focusing on the relationship between the caregiver and the PwD and the behaviour of the PwD, but on the caregiver, too. This is done by highlighting the importance of self-care among caregivers and implementing a meditation and diary feature. SUCCESS supports the PwD to maintain a purposeful life by suggesting meaningful activities that can be adapted to various stages of dementia. Additionally, the app provides a quick help feature and the possibility to personalize the content by using tags. Therefore, SUCCESS is an application that caters to every stage of dementia and supports caregivers in various situations by providing information, a possibility to apply and train the gained knowledge, and tools for self-care.

The research presented is conducted within the SUCCESS project (AAL-2016-089), partially funded by the European Active and Assisted Living Programme and the National Funding Agencies from Austria, Cyprus, Norway and Romania.

EXECUTIVE SUMMARY

The present report documents the second iteration of the integrated architecture of the SUCCESS platform as this is defined until M17 based on the output produced from *T2.2: Definition of use cases and scenarios.* In terms of an agile approach this document constitutes the continuation of the first iteration which was defined at M05. Based on the fact that the first integrated prototype was implemented at M12 including all the components responsible for the execution of the defined functionalities in the first iteration, the present architectural specification focuses on the 4+1 view model. This model is used for describing the architecture of software-intensive systems, based on the use of multiple, concurrent views [1]. Each of these views is used to describe the system and the role of each component to the integrated architecture from the perspective of different stakeholders, such as end-users, developers and project managers.

1. ABOUT THIS DOCUMENT

1.1 ROLE OF THE DELIVERABLE

Specification and implementation of the SUCCESS Platform follows an agile and iterative process whereas, in conjunction with the definition of use cases, the architecture is defined in incremental steps with the aim of converging to a version that fully supports the desired functionality, as this is defined by the use cases.

The present deliverable documents the second iteration of the functional specification and architecture specification of SUCCESS. Since the work of Use Cases is already defined, this iteration provides a more accurate state of the architecture specification. This is the final version of the document as it was scheduled for M17 and documents the full architecture which incorporates all the details of the Use Cases.

1.2 RELATIONSHIP TO OTHER SUCCESS DELIVERABLES

The deliverable is related to the following SUCCESS deliverables:

DELIVERABLE	RELATION
D2.2 Use Cases, Scenarios, Service and Interaction Design Concept	D4.1 relies heavily on D2.2 since the details of the architecture and the functional requirements should be defined as to fully cover the whole range of the Use Cases.
D4.2 Security and privacy infrastructure specification	D4.2 specifies the security and privacy infrastructure for the SUCCESS system based on the architecture defined in the deliverable D4.1 at hand.

2. INTRODUCTION

The aim of SUCCESS is to provide an innovative training application on the user's mobile device to support and accompany formal and informal caregivers, and the public, to appropriately and effectively interact with persons with dementia, based on evidence-based communication and intervention strategies for dementia. The present report presents the second version of the functional specification and the integrated architecture of the SUCCESS platform. The architecture of SUCCESS is designed based on the communication and interfacing of various components of the system. More specifically, the interaction among components is based on interfacing through library modules, where each library consists of one or more components that compose the overall architecture as defined in the first iteration of *D4.3 Specification and Implementation of the Interaction Platform*. Each component is deployed as a micro service, and, for development purposes, as an artifact to a nexus repository, which is part of the development infrastructure that relies on an agile methodology where multiple software releases take place during the development phase.

While the design of the architecture and the definition of the specifications follow an agile approach, the final version of the requirements definition and system architecture follow the work produced in the context of *D2.2 Use Cases, Scenarios, Service and Interaction Design Concept.* To this end, the architecture is designed towards fulfilling the range of the defined functionalities that the SUCCESS platform must provide to the users. For better understanding of this architectural approach around different perspectives, the 4+1 model view is used and analysed in the present document.

3. SYSTEM ARCHITECTURE

Even though the development phase of the platform is still in progress, the functionalities that the application should perform are defined and the development team is continuously working on these aspects. Therefore, taking under consideration the results of *T2.2: Definition of uses cases and scenarios*, the SUCCESS Platform must satisfy a specific set of functionalities which can be summarized as follows:

- Allow users to register and personalize their application settings and personal profile information
- Provide guidance to users regarding a specific situation, both in terms of offering advice per the criteria set by the user and in terms of offering training to the user
- Provide training to the users so they can learn about communication strategies when dealing with persons with dementia
- Offering suggestions and information to users on how to provide for a meaningful life to persons with dementia
- Advising users on how to offer emotional support to persons with dementia.
- Provide the means to users for evaluating the system.

Due to the fact that the SUCCESS platform aims to satisfy the aforementioned functionalities that derived from user requirements, Figure 1 demonstrates the high-level architecture which is composed by a set of application components. These components are considered as the backbone of the application as can be realized in the following sections of the document where the exact functionalities that they satisfy are defined. Briefly, the main components of SUCCESS are:

- **The Output Platform** which is responsible for rendering the user interface that will allow the user to interact with the system.
- **The Dialogue Manager** which is the central part of the SUCCESS application and consists of three parts, the profiler, the rewarder and the adviser, which interact with each other and also with other components of the application.
  - **The Profiler** which administers data about the user in a user model and makes them available to the Output Platform.
  - **The Rewarder** that generates feedback to the user by choosing appropriate feedback for performance measures from the system which is derived from the profiler.
  - **The Adviser** which produces an advising entity which is for certain use cases visually and audibly represented by the avatar component and provides dialogue interaction with the user from content nodes that contain dialogue models.
- **The Affective Avatar** that provides the means to produce visual and audible feedback to the user in a livelier and more natural form.

- **The Content Provider** which is the backend component that is responsible for managing and providing the contents that are presented to the Users.

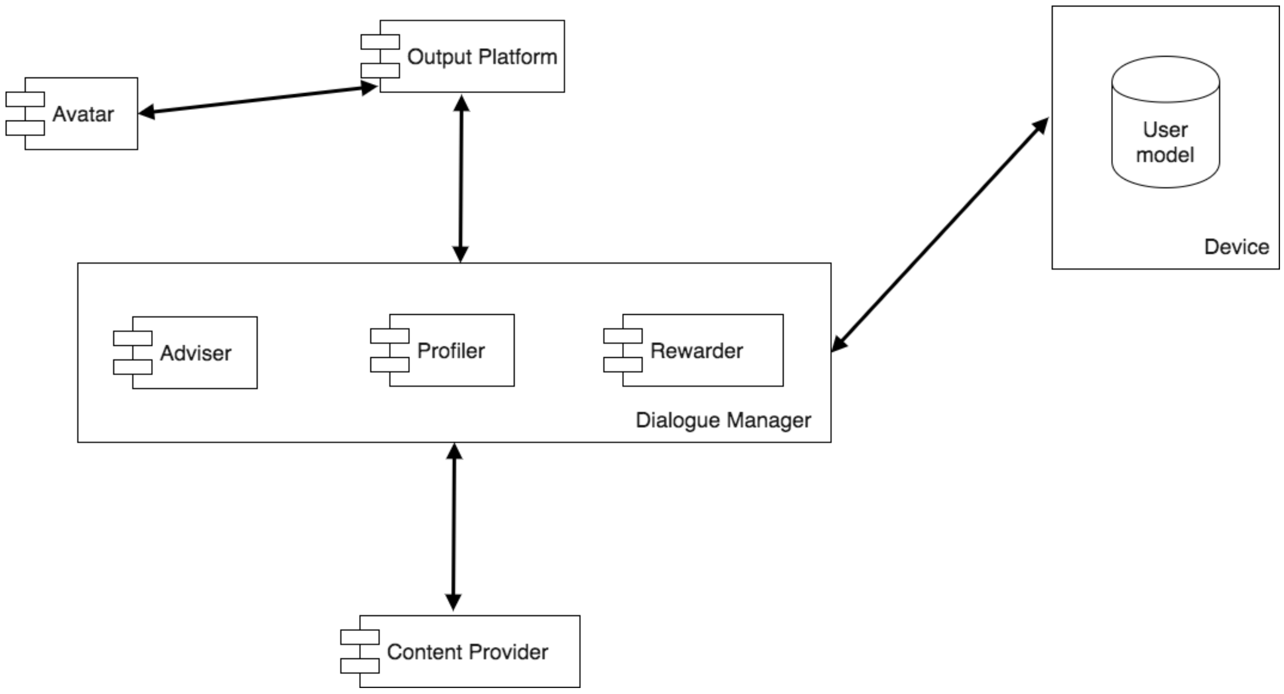


Figure 1 : Architectural Design

4. ARCHITECTURAL VIEW MODEL

The architectural view model that initially introduced in chapter 2 consists of four main views and is used for the detailed description from different perspectives of the implemented architecture in the context of SUCCESS. Figure 2 gives an overview of the model that aims to describe the architecture using five concurrent views. These views can be summarized as: the development view, the logical view, the physical view and the process view. The fifth view denotes use cases, or scenarios that derived from *D2.2 Use Cases, Scenarios, Service and Interaction Design Concept*. The following subsections provide a more detailed definition of each type of view where the interaction of the components of the system is presented from a different point of view per case.

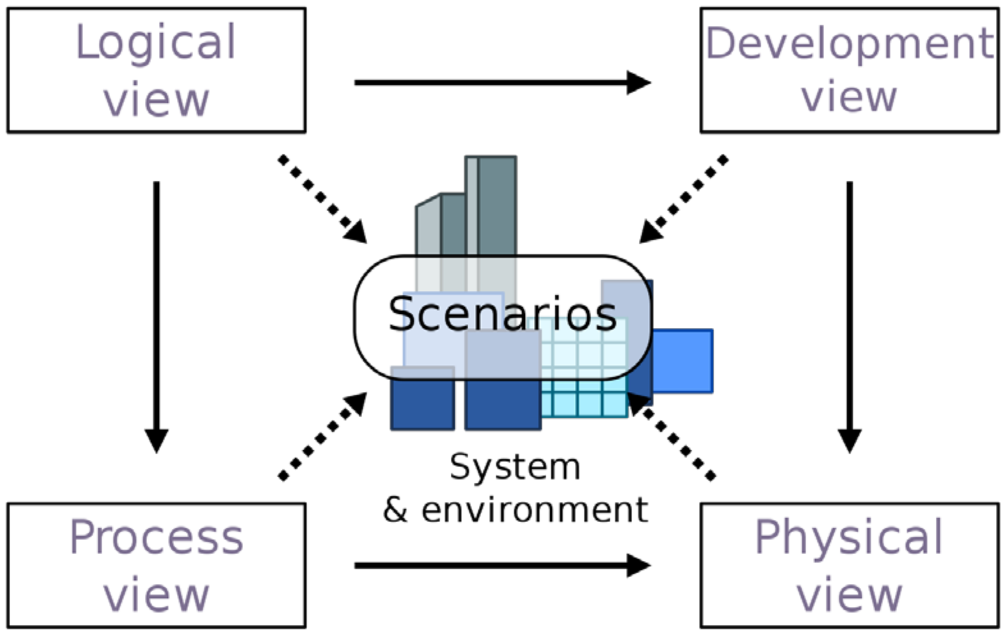


Figure 2: 4+1 Architectural View Model

4.1 PHYSICAL VIEW

The physical view denotes the physical locations of the software and the physical connections between the software components of the system. Also, it can be considered as a deployment view.

SUCCESS, since it is an Android application, can be distributed and installed via an APK file. The APK file includes the code along with any required data and resource files into an Android application archive file with the APK suffix. The APK file represents an Android application to be deployed to the mobile devices that support Android operating system. Therefore, SUCCESS is presented through a typical deployment view of an android application, which is composed by several application components, where each component performs a different role in the overall application behaviour. Figure 3 represents the deployment view of the SUCCESS mobile application.

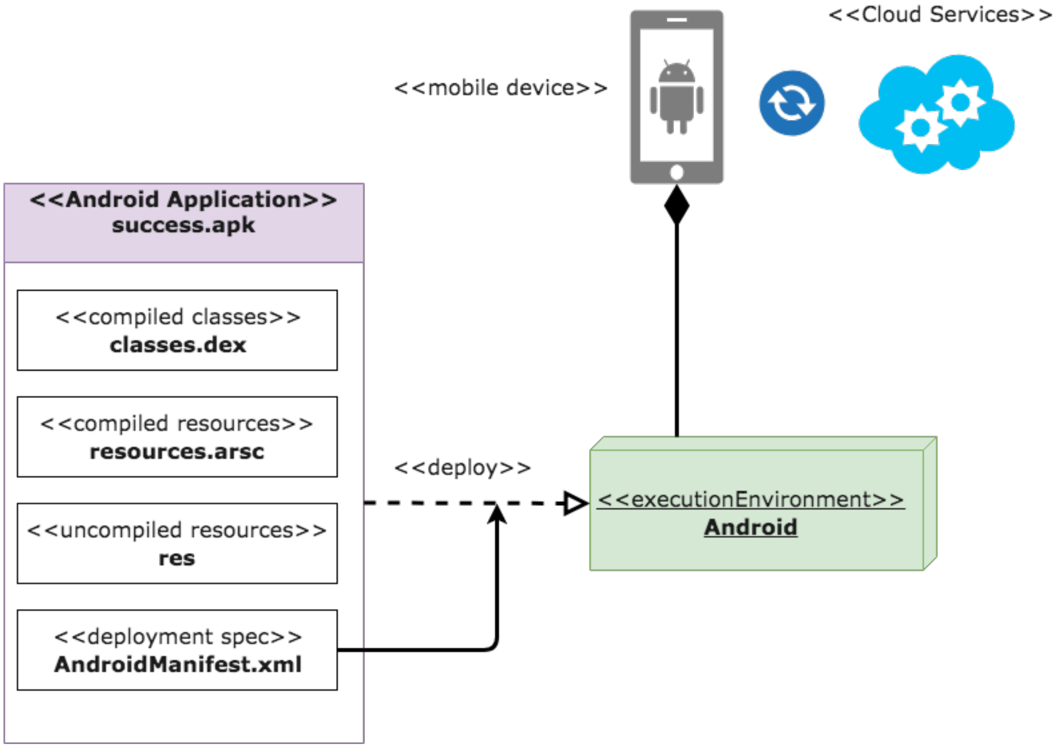


Figure 3: Deployment View of the Mobile Application

4.2 DEVELOPMENT VIEW

The development view presents the system from the programmer's perspective and is focused on software management and the demonstration of the components based on their relationship in the development environment. In this context, Figure 4 illustrates the components of the SUCCESS system as pieces of software, and their interrelationships.

More specifically, all the application components of the system are demonstrated as well as the assembly connectors which connect the provided and requested interfaces. Figure 4 provides a depiction of the level 1 Component Diagram, where a more detailed interconnection of the application components is presented. Each represented component is a modular part of the application, whose behavior is defined by its provided and required interfaces. An Interface is a specification of behavior that implementer components agree to meet. An assembly connector is used in order to bridge a component's required interface with the provided interface of another component. An example case is the requested interface for user’s personalized information by the Output Platform and the provided interface by the Profiler called “Output for Questionnaires”. More detailed definition of the interfaces per component will follow in section 4.5.

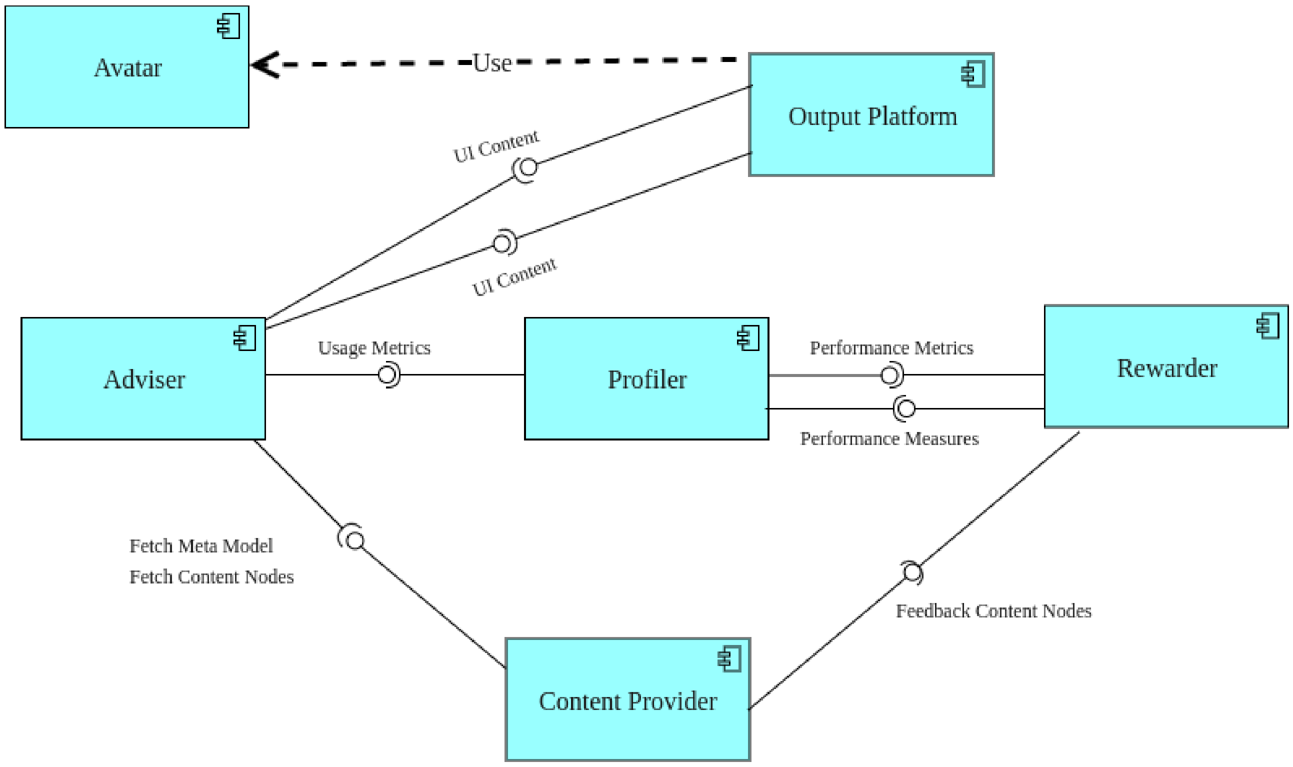


Figure 4: Development View of the Components

4.3 LOGICAL VIEW

The logical view defines the functionalities that the SUCCESS platform should perform and provide to the users in order to ensure that all of the desired functionalities, which derived from the Use Cases, are captured by the system. Based upon this approach, a description of the functionalities that each component provides, is specified and is followed by a level 1 class diagram which depicts each specification through class entities, important methods and the relationships among objects.

4.3.1 OUTPUT PLATFORM

The Output Platform represents the main user frontend with which the users interact directly with the system. The main purpose of the Output Platform is the rendering of the user interface that will allow the user to interact with the system and use the services provided by SUCCESS. Figure 5 presents a simplified logical view class diagram of the output platform implementation.

Specifically, as depicted in the figure shown below, the majority of the Output Platform classes extent the *AdvisorActivity* Java Class. This class allows the Output platform classes to perform callbacks to the other components according to user requests and render the results. The *App* java class implements methods that are utilized from the Output Platform java classes.

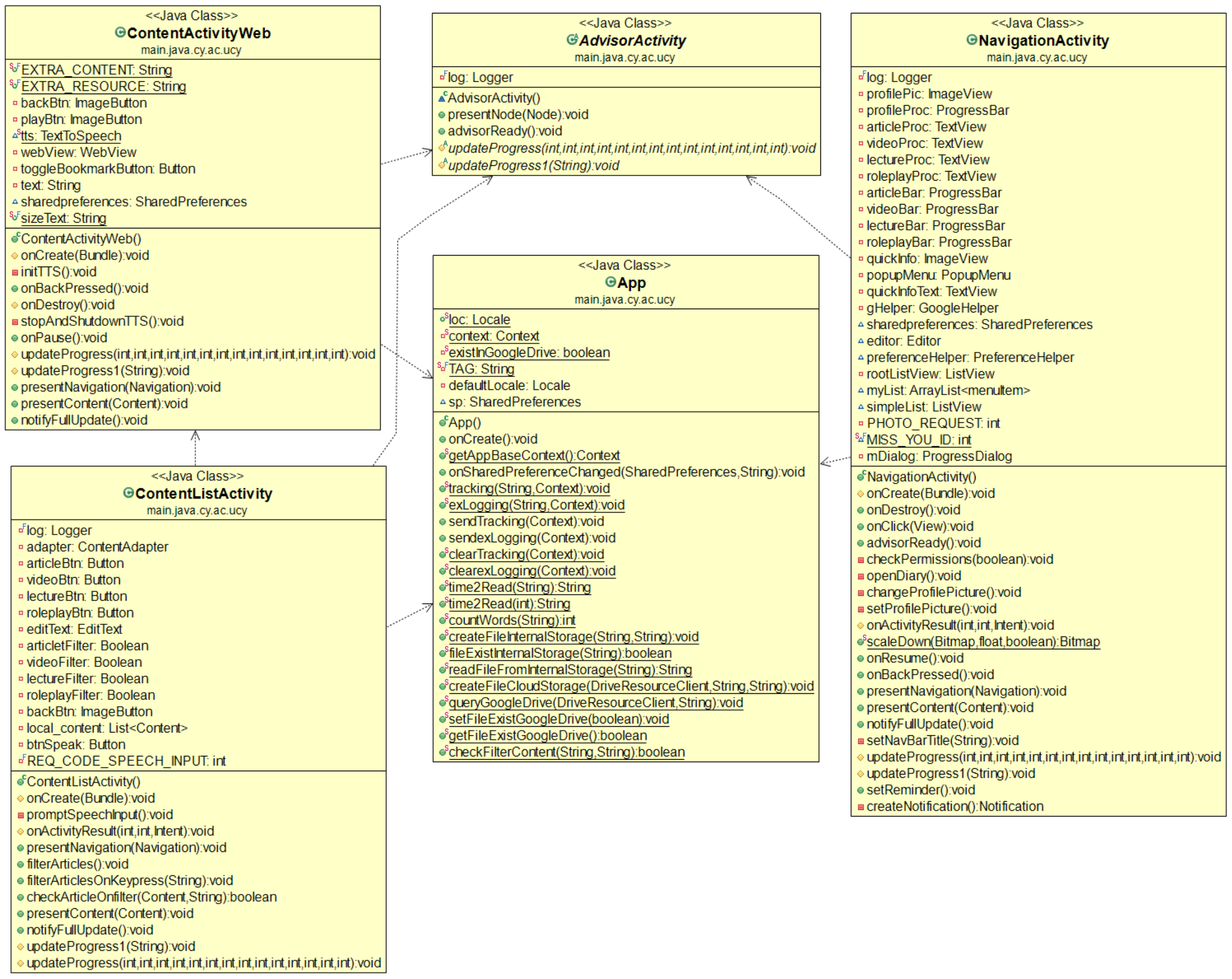

<<Java Class>>
ContentActivityWeb
main.java.cy.ac.ucy
EXTRA_CONTENT: String
EXTRA_RESOURCE: String
backBtn: ImageButton
playBtn: ImageButton
tts: TextToSpeech
webView: WebView
toggleBookmarkButton: Button
text: String
sharedpreferences: SharedPreferences
sizeText: String
ContentActivityWeb()
onCreate(Bundle):void
initTTS():void
onBackPressed():void
onDestroy():void
stopAndShutdownTTS():void
onPause():void
updateProgress(int,int,int,int,int,int,int,int,int,int,int,int,int,int):void
updateProgress1(String):void
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
<<Java Class>>
AdvisorActivity
main.java.cy.ac.ucy
log: Logger
AdvisorActivity()
presentNode(Node):void
advisorReady():void
updateProgress(int,int,int,int,int,int,int,int,int,int,int,int,int,int):void
updateProgress1(String):void
<<Java Class>>
NavigationActivity
main.java.cy.ac.ucy
log: Logger
profilePic: ImageView
profileProc: ProgressBar
articleProc: TextView
videoProc: TextView
lectureProc: TextView
roleplayProc: TextView
articleBar: ProgressBar
videoBar: ProgressBar
lectureBar: ProgressBar
roleplayBar: ProgressBar
quickInfo: ImageView
popupMenu: PopupMenu
quickInfoText: TextView
gHelper: GoogleHelper
sharedpreferences: SharedPreferences
editor: Editor
preferenceHelper: PreferenceHelper
rootListView: ListView
myList: ArrayList<menuItem>
simpleList: ListView
PHOTO_REQUEST: int
MISS_YOU_ID: int
mDialog: ProgressDialog
NavigationActivity()
onCreate(Bundle):void
onDestroy():void
onClick(View):void
advisorReady():void
checkPermissions(boolean):void
openDiary():void
changeProfilePicture():void
setProfilePicture():void
onActivityResult(int,int,Intent):void
scaleDown(Bitmap,float,boolean):Bitmap
onResume():void
onBackPressed():void
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
setNavBarTitle(String):void
updateProgress(int,int,int,int,int,int,int,int,int,int,int,int,int,int):void
updateProgress1(String):void
setReminder():void
createNotification():Notification
<<Java Class>>
App
main.java.cy.ac.ucy
loc: Locale
context: Context
existInGoogleDrive: boolean
TAG: String
defaultLocale: Locale
sp: SharedPreferences
App()
onCreate():void
getAppBaseContext():Context
onSharedPreferenceChanged(SharedPreferences,String):void
tracking(String,Context):void
exLogging(String,Context):void
sendTracking(Context):void
sendexLogging(Context):void
clearTracking(Context):void
clearexLogging(Context):void
time2Read(String):String
time2Read(int):String
countWords(String):int
createFileInternalStorage(String,String):void
fileExistInternalStorage(String):boolean
readFileFromInternalStorage(String):String
createFileCloudStorage(DriveResourceClient,String,String):void
queryGoogleDrive(DriveResourceClient,String):void
setFileExistGoogleDrive(boolean):void
getFileExistGoogleDrive():boolean
checkFilterContent(String,String):boolean
<<Java Class>>
ContentListActivity
main.java.cy.ac.ucy
log: Logger
adapter: ContentAdapter
articleBtn: Button
videoBtn: Button
lectureBtn: Button
roleplayBtn: Button
editText: EditText
articletFilter: Boolean
videoFilter: Boolean
lectureFilter: Boolean
roleplayFilter: Boolean
backBtn: ImageButton
local_content: List<Content>
btnSpeak: Button
REQ_CODE_SPEECH_INPUT: int
ContentListActivity()
onCreate(Bundle):void
promptSpeechInput():void
onActivityResult(int,int,Intent):void
presentNavigation(Navigation):void
filterArticles():void
filterArticlesOnKeypress(String):void
checkArticleOnfilter(Content,String):boolean
presentContent(Content):void
notifyFullUpdate():void
updateProgress1(String):void
updateProgress(int,int,int,int,int,int,int,int,int,int,int,int,int,int):void

Figure 5: Logical View of Output Platform

4.3.2 PROFILER

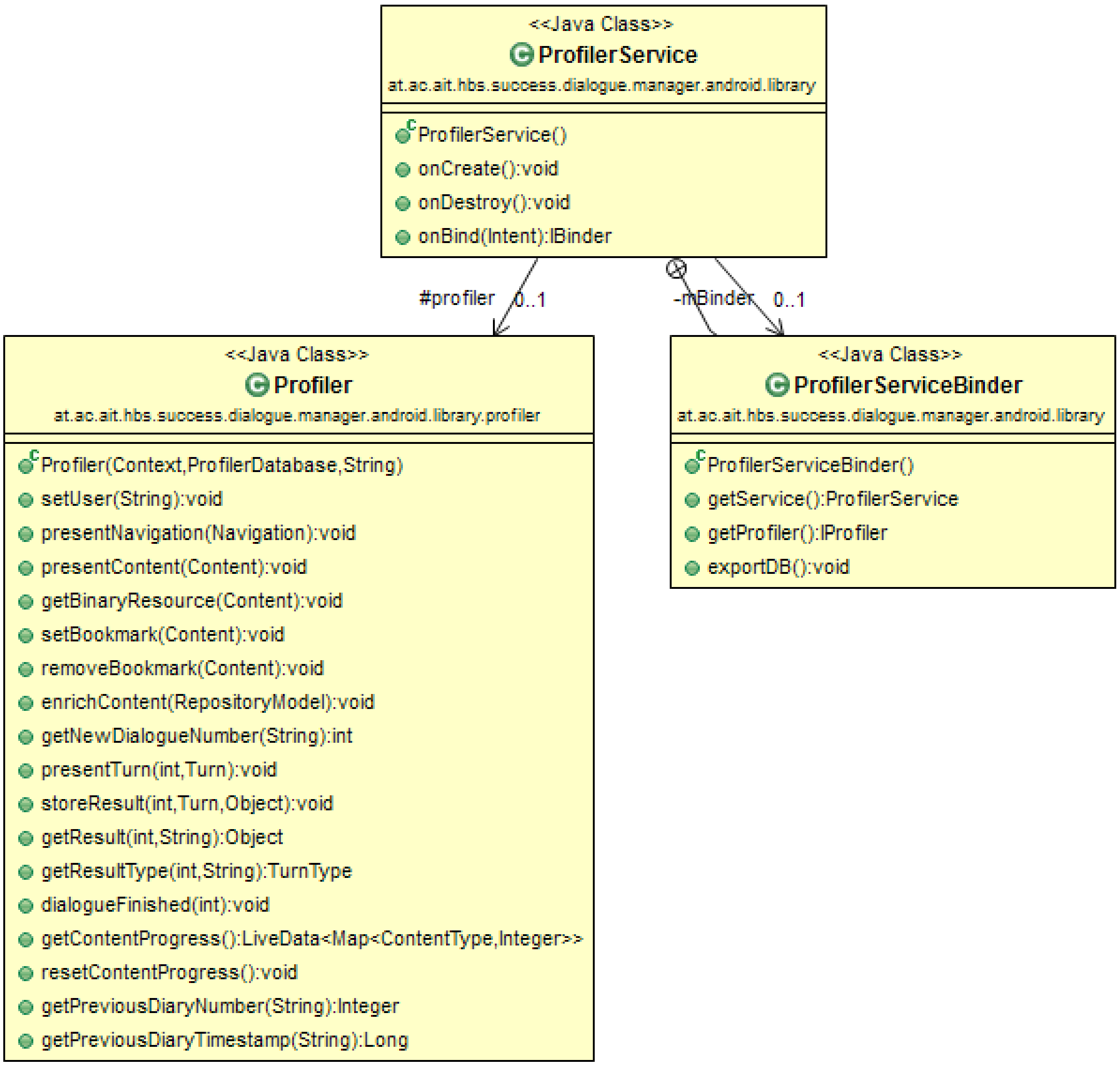
The Profiler component is one of the three components realising the dialogue management in the SUCCESS solution.

The main purpose of the Profiler is to administer data about the user in a user model. The Profiler does this through continuously recording of data during system usage, as well as through initiating active inquiry.

The main tasks are:

- Acquisition of user-relevant data, through: active inquiry, usage patterns, Adviser input etc.
- Administering the internal user model
- Realizing a Questionnaire like dialogue model to produce the initial user model/profile, and managing subsequent updates to it.

The Profiler is implemented as Android service (see Figure 6), so that it starts and runs in the background when the SUCCESS App is started and running.



<<Java Class>>
ProfilerService
at.ac.ait.hbs.success.dialogue.manager.android.library
ProfilerService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
#profiler
0..1
-mBinder
0..1
<<Java Class>>
Profiler
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler
Profiler(Context,ProfilerDatabase,String)
setUser(String):void
presentNavigation(Navigation):void
presentContent(Content):void
getBinaryResource(Content):void
setBookmark(Content):void
removeBookmark(Content):void
enrichContent(RepositoryModel):void
getNewDialogueNumber(String):int
presentTurn(int,Turn):void
storeResult(int,Turn,Object):void
getResult(int,String):Object
getResultType(int,String):TurnType
dialogueFinished(int):void
getContentProgress():LiveData<Map<ContentType,Integer>>
resetContentProgress():void
getPreviousDiaryNumber(String):Integer
getPreviousDiaryTimestamp(String):Long
<<Java Class>>
ProfilerServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
ProfilerServiceBinder()
getService():ProfilerService
getProfiler():IProfiler
exportDB():void

Figure 6: Profiler Service implementation

The *ProfilerService* is extended by the *DialogueService*, which in turn is extended by the *AdvisorService*, as depicted in Figure 7.

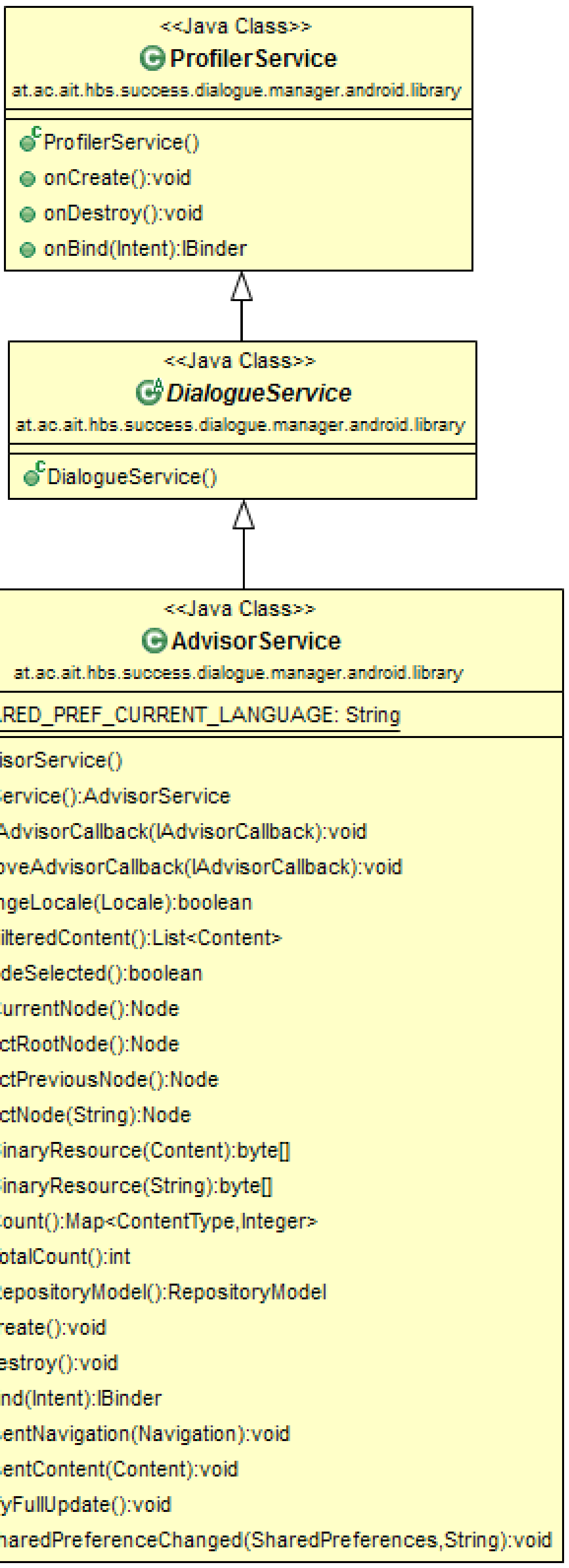

<<Java Class>>
ProfilerService
at.ac.ait.hbs.success.dialogue.manager.android.library
ProfilerService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
<<Java Class>>
DialogueService
at.ac.ait.hbs.success.dialogue.manager.android.library
DialogueService()
<<Java Class>>
AdvisorService
at.ac.ait.hbs.success.dialogue.manager.android.library
SHARED_PREF_CURRENT_LANGUAGE: String
AdvisorService()
getService():AdvisorService
addAdvisorCallback(IAdvisorCallback):void
removeAdvisorCallback(IAdvisorCallback):void
changeLocale(Locale):boolean
getFilteredContent():List<Content>
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
getRepositoryModel():RepositoryModel
onCreate():void
onDestroy():void
onBind(Intent):IBinder
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
onSharedPreferenceChanged(SharedPreferences,String):void

Figure 7: Android dialogue manager service provision

Access to the profiler for other components is provided through the *getProfiler()* method which is exposed by the *ProfilerServiceBinder* class (shown in Figure 6) and ultimately accessible through the *AdvisorService*.

The *Profiler* class implements an interface that defines various methods for other components to interact with the profiler, as shown in Figure 8.

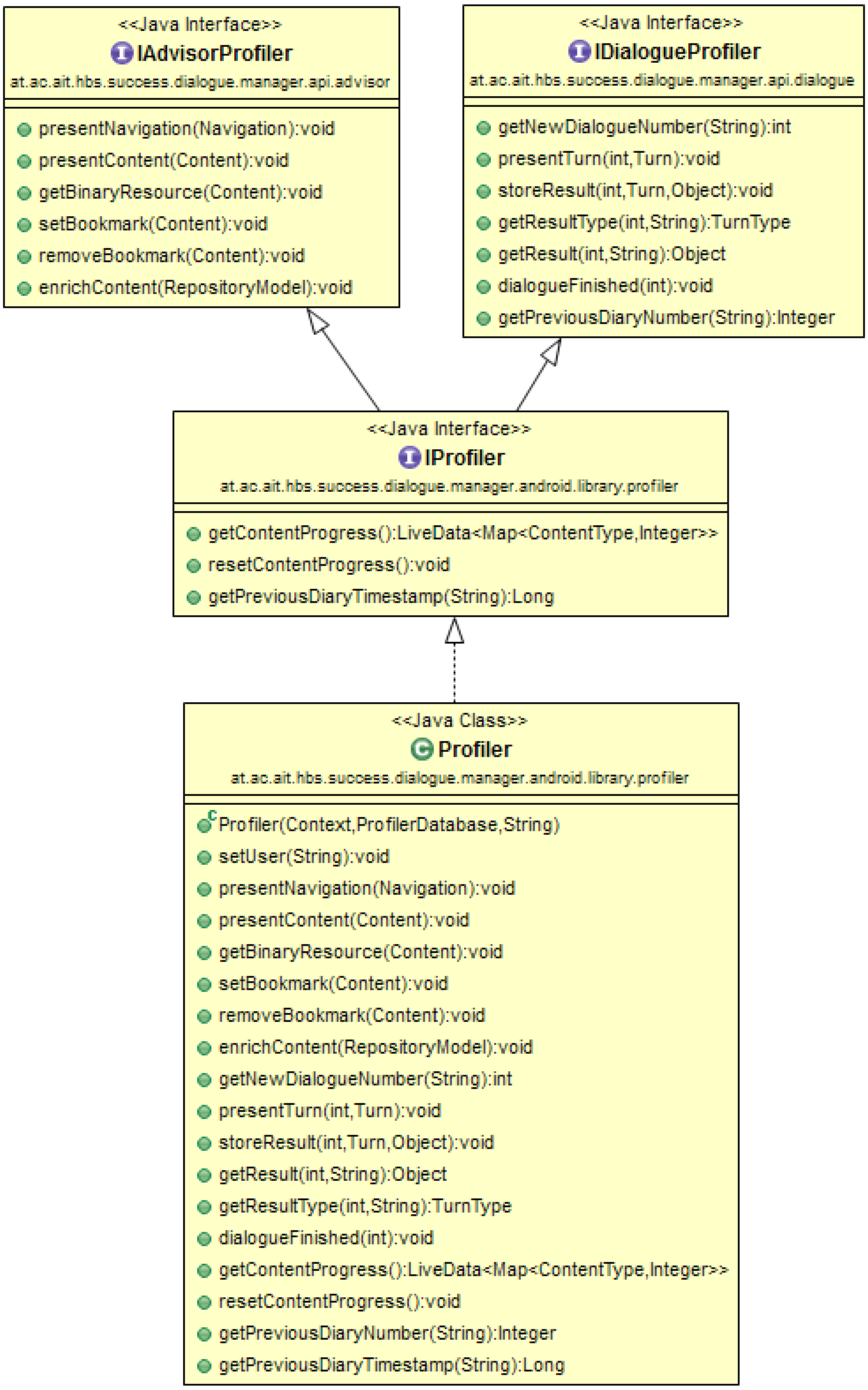

<<Java Interface>>
IAdvisorProfiler
at.ac.ait.hbs.success.dialogue.manager.api.advisor
presentNavigation(Navigation):void
presentContent(Content):void
getBinaryResource(Content):void
setBookmark(Content):void
removeBookmark(Content):void
enrichContent(RepositoryModel):void
<<Java Interface>>
IDialogueProfiler
at.ac.ait.hbs.success.dialogue.manager.api.dialogue
getNewDialogueNumber(String):int
presentTurn(int,Turn):void
storeResult(int,Turn,Object):void
getResultType(int,String):TurnType
getResult(int,String):Object
dialogueFinished(int):void
getPreviousDiaryNumber(String):Integer
<<Java Interface>>
IProfiler
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler
getContentProgress():LiveData<Map<ContentType,Integer>>
resetContentProgress():void
getPreviousDiaryTimestamp(String):Long
<<Java Class>>
Profiler
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler
Profiler(Context,ProfilerDatabase,String)
setUser(String):void
presentNavigation(Navigation):void
presentContent(Content):void
getBinaryResource(Content):void
setBookmark(Content):void
removeBookmark(Content):void
enrichContent(RepositoryModel):void
getNewDialogueNumber(String):int
presentTurn(int,Turn):void
storeResult(int,Turn,Object):void
getResult(int,String):Object
getResultType(int,String):TurnType
dialogueFinished(int):void
getContentProgress():LiveData<Map<ContentType,Integer>>
resetContentProgress():void
getPreviousDiaryNumber(String):Integer
getPreviousDiaryTimestamp(String):Long

Figure 8: Profiler implementation

The Profiler keeps track of the user interaction by using an SQLite database (using Androids *Room Persistence Library* [2]), shown in Figure 9.

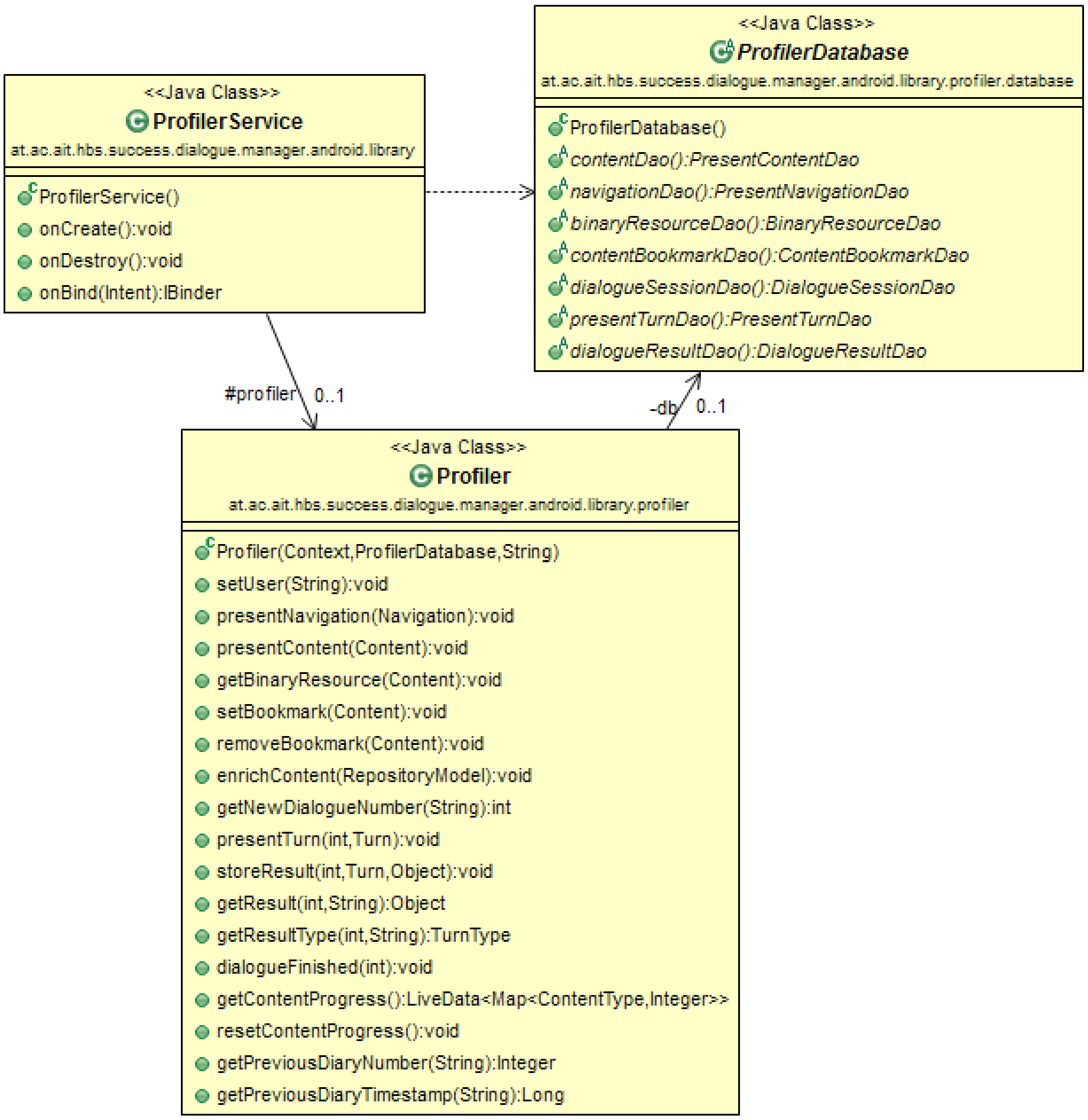
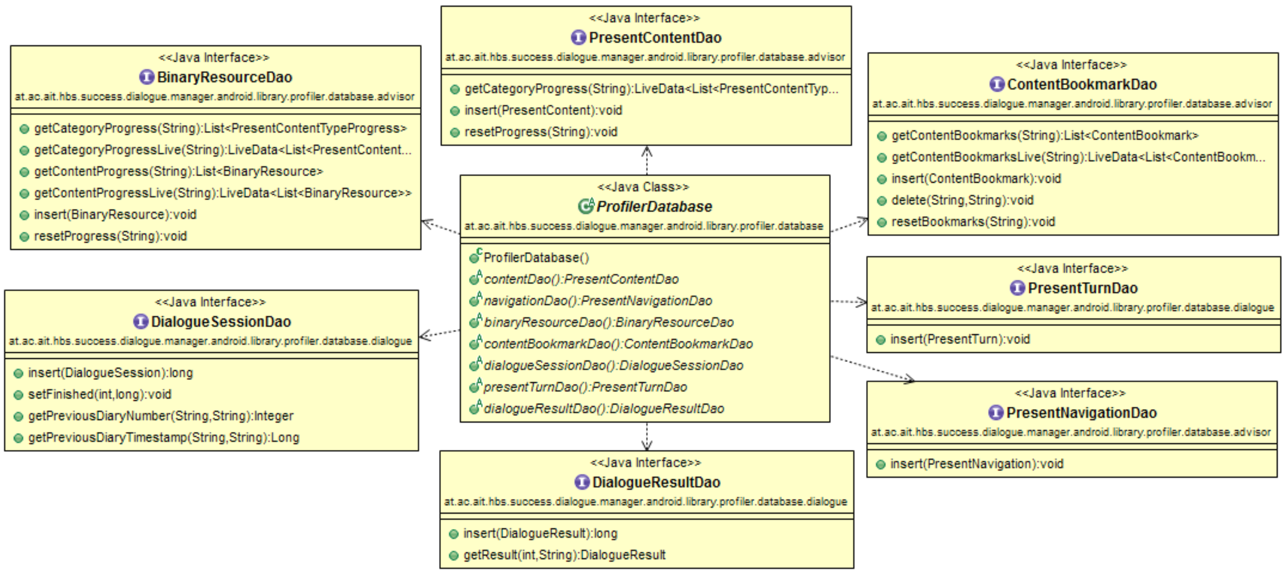

<<Java Class>>
ProfilerService
at.ac.ait.hbs.success.dialogue.manager.android.library
ProfilerService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
<<Java Class>>
ProfilerDatabase
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database
ProfilerDatabase()
contentDao():PresentContentDao
navigationDao():PresentNavigationDao
binaryResourceDao():BinaryResourceDao
contentBookmarkDao():ContentBookmarkDao
dialogueSessionDao():DialogueSessionDao
presentTurnDao():PresentTurnDao
dialogueResultDao():DialogueResultDao
#profiler
0..1
-db
0..1
<<Java Class>>
Profiler
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler
Profiler(Context,ProfilerDatabase,String)
setUser(String):void
presentNavigation(Navigation):void
presentContent(Content):void
getBinaryResource(Content):void
setBookmark(Content):void
removeBookmark(Content):void
enrichContent(RepositoryModel):void
getNewDialogueNumber(String):int
presentTurn(int,Turn):void
storeResult(int,Turn,Object):void
getResult(int,String):Object
getResultType(int,String):TurnType
dialogueFinished(int):void
getContentProgress():LiveData<Map<ContentType,Integer>>
resetContentProgress():void
getPreviousDiaryNumber(String):Integer
getPreviousDiaryTimestamp(String):Long

Figure 9: Profiler database implementation

The *ProfilerDatabase* uses various interfaces to interact with the underlying database, as shown in Figure 10.



<<Java Interface>>
BinaryResourceDao
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database.advisor
getCategoryProgress(String):List<PresentContentTypeProgress>
getCategoryProgressLive(String):LiveData<List<PresentContent...
getContentProgress(String):List<BinaryResource>
getContentProgressLive(String):LiveData<List<BinaryResource>>
insert(BinaryResource):void
resetProgress(String):void
<<Java Interface>>
PresentContentDao
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database.advisor
getCategoryProgress(String):LiveData<List<PresentContentTyp...
insert(PresentContent):void
resetProgress(String):void
<<Java Interface>>
ContentBookmarkDao
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database.advisor
getContentBookmarks(String):List<ContentBookmark>
getContentBookmarksLive(String):LiveData<List<ContentBookm...
insert(ContentBookmark):void
delete(String,String):void
resetBookmarks(String):void
<<Java Class>>
ProfilerDatabase
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database
ProfilerDatabase()
contentDao():PresentContentDao
navigationDao():PresentNavigationDao
binaryResourceDao():BinaryResourceDao
contentBookmarkDao():ContentBookmarkDao
dialogueSessionDao():DialogueSessionDao
presentTurnDao():PresentTurnDao
dialogueResultDao():DialogueResultDao
<<Java Interface>>
DialogueSessionDao
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database.dialogue
insert(DialogueSession):long
setFinished(int,long):void
getPreviousDiaryNumber(String,String):Integer
getPreviousDiaryTimestamp(String,String):Long
<<Java Interface>>
PresentTurnDao
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database.dialogue
insert(PresentTurn):void
<<Java Interface>>
PresentNavigationDao
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database.advisor
insert(PresentNavigation):void
<<Java Interface>>
DialogueResultDao
at.ac.ait.hbs.success.dialogue.manager.android.library.profiler.database.dialogue
insert(DialogueResult):long
getResult(int,String):DialogueResult

Figure 10: Profiler database interfaces

4.3.3 REWARDER

The rewarder component is one of the three component of the three components of the dialogue management in the SUCCESS solution.

The main purpose of the rewarder component is to choose or generate feedback to the user. It does so by choosing appropriate feedback for performance measures from the system (i.e. derived from Profiler data). The rewarder utilises adequate and meaningful content to increase motivation and engagement with the trainings. In general, the Rewarder needs to measure performance indicators and based on these generate feedback to the user.

The rewarded has not been implemented yet and will be available after the next iteration of the development process.

4.3.4 ADVISER

The adviser component is one of the three components realising the dialogue management in the SUCCESS solution.

The main purpose of the advisor is to produce an advising entity that generates a dialogue like interaction from content nodes that contain dialogue models and the structure of the Meta model. The dialogue like interaction is expressed textually and visually and audibly supported by the avatar component.

The main tasks are:

- Converting Meta Model based data structures for content navigation, into a model for dialogue like interaction.
- Interpretation of Avatar content (dialogue models) at run time taking some context information into account; generating real time interaction instructions for the Output platform's avatar component.
- Acquisition of interaction data, for profiling including usage and performance metrics

The Adviser is implemented in Java and wrapper in an Android service (see Figure 11), so that it starts and runs in the background when the SUCCESS App is started and running.

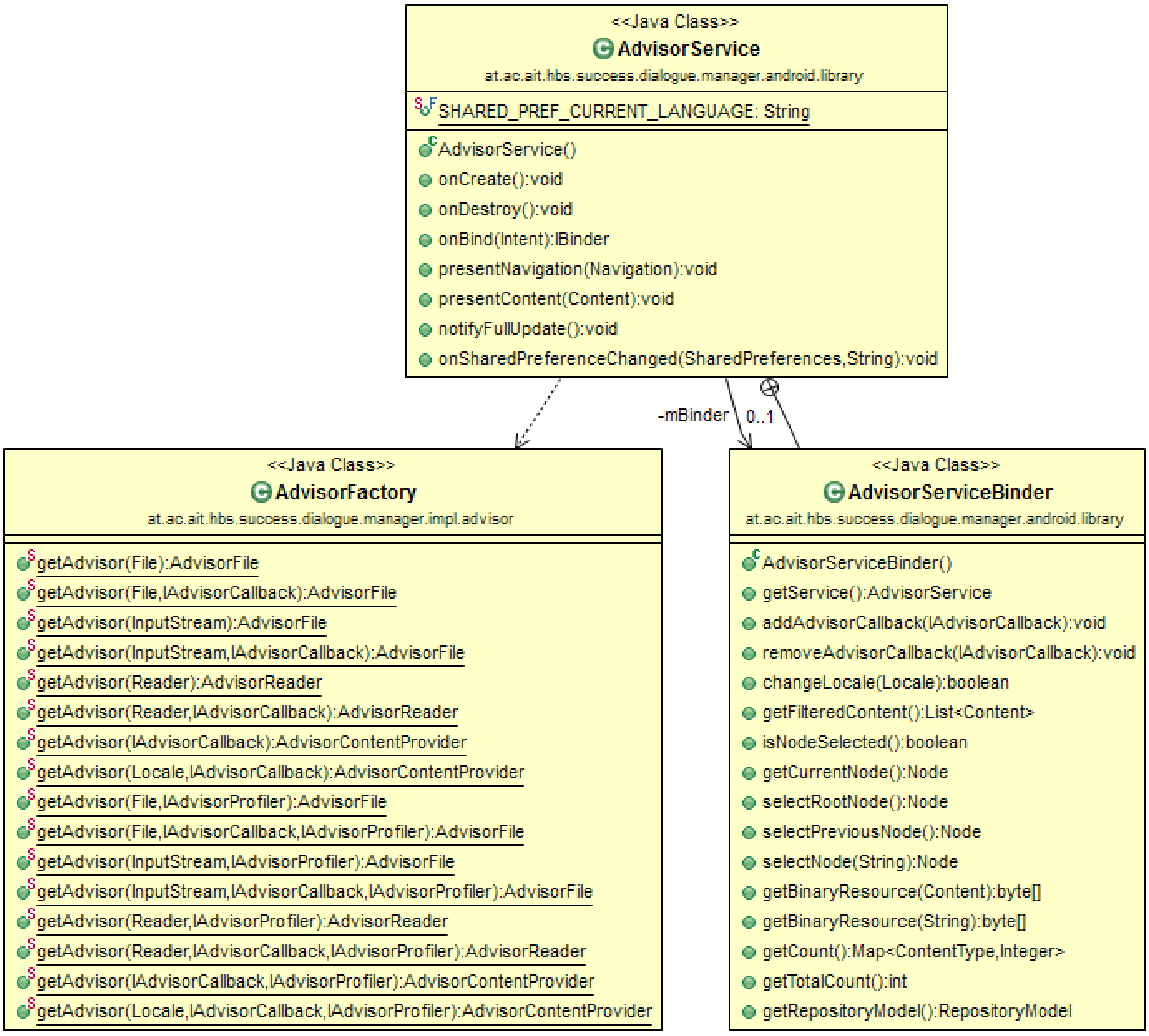

<<Java Class>>
AdvisorService
at.ac.ait.hbs.success.dialogue.manager.android.library
SHARED_PREF_CURRENT_LANGUAGE: String
AdvisorService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
onSharedPreferenceChanged(SharedPreferences,String):void
-mBinder
0..1
<<Java Class>>
AdvisorFactory
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
getAdvisor(File):AdvisorFile
getAdvisor(File,IAdvisorCallback):AdvisorFile
getAdvisor(InputStream):AdvisorFile
getAdvisor(InputStream,IAdvisorCallback):AdvisorFile
getAdvisor(Reader):AdvisorReader
getAdvisor(Reader,IAdvisorCallback):AdvisorReader
getAdvisor(IAdvisorCallback):AdvisorContentProvider
getAdvisor(Locale,IAdvisorCallback):AdvisorContentProvider
getAdvisor(File,IAdvisorProfiler):AdvisorFile
getAdvisor(File,IAdvisorCallback,IAdvisorProfiler):AdvisorFile
getAdvisor(InputStream,IAdvisorProfiler):AdvisorFile
getAdvisor(InputStream,IAdvisorCallback,IAdvisorProfiler):AdvisorFile
getAdvisor(Reader,IAdvisorProfiler):AdvisorReader
getAdvisor(Reader,IAdvisorCallback,IAdvisorProfiler):AdvisorReader
getAdvisor(IAdvisorCallback,IAdvisorProfiler):AdvisorContentProvider
getAdvisor(Locale,IAdvisorCallback,IAdvisorProfiler):AdvisorContentProvider
<<Java Class>>
AdvisorServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
AdvisorServiceBinder()
getService():AdvisorService
addAdvisorCallback(IAdvisorCallback):void
removeAdvisorCallback(IAdvisorCallback):void
changeLocale(Locale):boolean
getFilteredContent():List<Content>
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
getRepositoryModel():RepositoryModel

Figure 11: Adviser Service implementation

The Android library of the dialogue manager comes with a *ServiceUtil* utility class which provides a number of utility methods. One of them is a convenience method *initAdvisorService(Context)* for initializing the *AdvisorService*, shown in Figure 12.

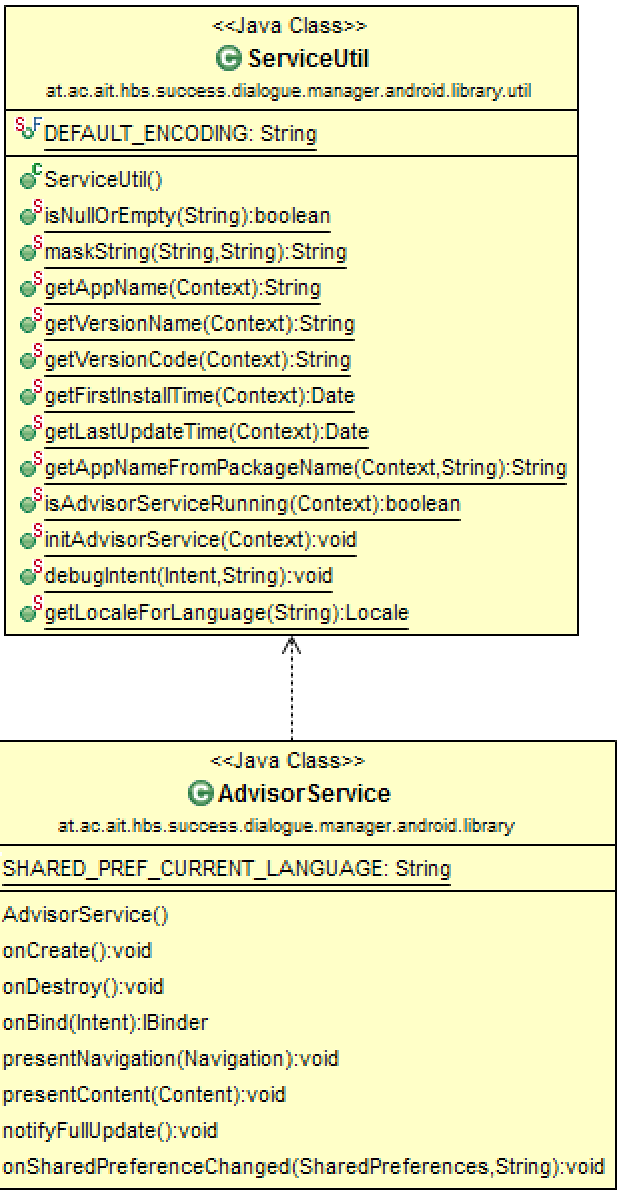

<<Java Class>>
ServiceUtil
at.ac.ait.hbs.success.dialogue.manager.android.library.util
DEFAULT_ENCODING: String
ServiceUtil()
isNullOrEmpty(String):boolean
maskString(String,String):String
getAppName(Context):String
getVersionName(Context):String
getVersionCode(Context):String
getFirstInstallTime(Context):Date
getLastUpdateTime(Context):Date
getAppNameFromPackageName(Context,String):String
isAdvisorServiceRunning(Context):boolean
initAdvisorService(Context):void
debugIntent(Intent,String):void
getLocaleForLanguage(String):Locale
<<Java Class>>
AdvisorService
at.ac.ait.hbs.success.dialogue.manager.android.library
SHARED_PREF_CURRENT_LANGUAGE: String
AdvisorService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
onSharedPreferenceChanged(SharedPreferences,String):void

Figure 12: Adviser Service Utility implementation

Depending on how input is provided for the Adviser by the system, the *AdvisorFactory* class gets and returns the appropriate *Advisor* instance, shown in Figure 13. In SUCCESS, the *ContentProvider* provides the data for the *Advisor*.

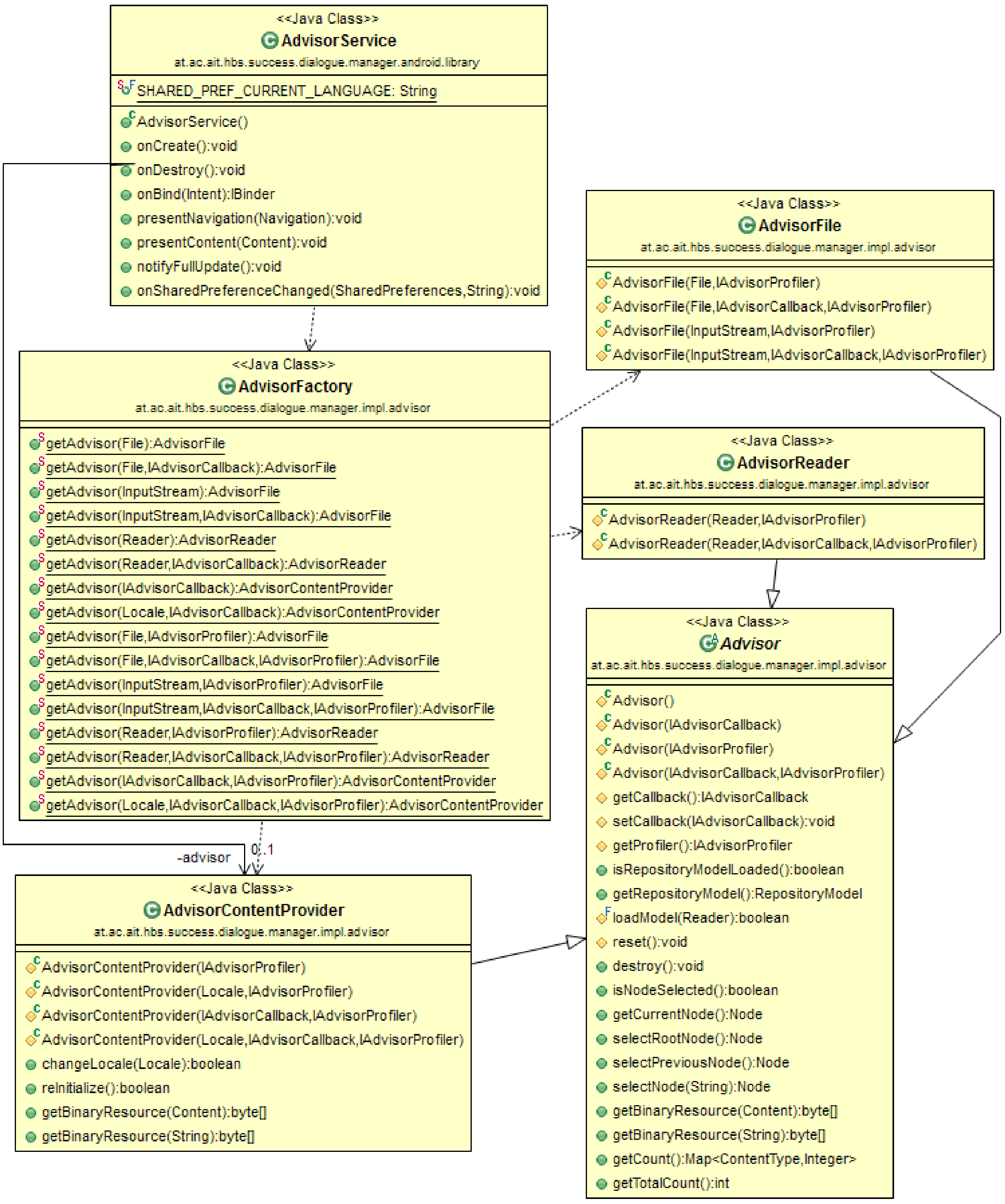

<<Java Class>>
AdvisorService
at.ac.ait.hbs.success.dialogue.manager.android.library
SHARED_PREF_CURRENT_LANGUAGE: String
AdvisorService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
onSharedPreferenceChanged(SharedPreferences,String):void
<<Java Class>>
AdvisorFile
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
AdvisorFile(File,IAdvisorProfiler)
AdvisorFile(File,IAdvisorCallback,IAdvisorProfiler)
AdvisorFile(InputStream,IAdvisorProfiler)
AdvisorFile(InputStream,IAdvisorCallback,IAdvisorProfiler)
<<Java Class>>
AdvisorFactory
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
getAdvisor(File):AdvisorFile
getAdvisor(File,IAdvisorCallback):AdvisorFile
getAdvisor(InputStream):AdvisorFile
getAdvisor(InputStream,IAdvisorCallback):AdvisorFile
getAdvisor(Reader):AdvisorReader
getAdvisor(Reader,IAdvisorCallback):AdvisorReader
getAdvisor(IAdvisorCallback):AdvisorContentProvider
getAdvisor(Locale,IAdvisorCallback):AdvisorContentProvider
getAdvisor(File,IAdvisorProfiler):AdvisorFile
getAdvisor(File,IAdvisorCallback,IAdvisorProfiler):AdvisorFile
getAdvisor(InputStream,IAdvisorProfiler):AdvisorFile
getAdvisor(InputStream,IAdvisorCallback,IAdvisorProfiler):AdvisorFile
getAdvisor(Reader,IAdvisorProfiler):AdvisorReader
getAdvisor(Reader,IAdvisorCallback,IAdvisorProfiler):AdvisorReader
getAdvisor(IAdvisorCallback,IAdvisorProfiler):AdvisorContentProvider
getAdvisor(Locale,IAdvisorCallback,IAdvisorProfiler):AdvisorContentProvider
<<Java Class>>
AdvisorReader
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
AdvisorReader(Reader,IAdvisorProfiler)
AdvisorReader(Reader,IAdvisorCallback,IAdvisorProfiler)
<<Java Class>>
Advisor
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
Advisor()
Advisor(IAdvisorCallback)
Advisor(IAdvisorProfiler)
Advisor(IAdvisorCallback,IAdvisorProfiler)
getCallback():IAdvisorCallback
setCallback(IAdvisorCallback):void
getProfiler():IAdvisorProfiler
isRepositoryModelLoaded():boolean
getRepositoryModel():RepositoryModel
loadModel(Reader):boolean
reset():void
destroy():void
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
-advisor
0..1
<<Java Class>>
AdvisorContentProvider
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
AdvisorContentProvider(IAdvisorProfiler)
AdvisorContentProvider(Locale,IAdvisorProfiler)
AdvisorContentProvider(IAdvisorCallback,IAdvisorProfiler)
AdvisorContentProvider(Locale,IAdvisorCallback,IAdvisorProfiler)
changeLocale(Locale):boolean
reInitialize():boolean
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]

Figure 13: Adviser Factory and Adviser implementation

As mentioned before, when using the dialogue manager in Android, the *AdvisorService* must be initialized once when the app starts through the *ServiceUtil* convencience method *initAdvisorServ-ice(Context)*. When the *AdvisorService* is running, any class that needs the *Advisor* can simply ex-tend the helper class *AdvisorActivity* which in turn extends the abstract class *AbstractServiceActivi-*

ty. *AbstractServiceActivity* automatically binds to the *AdvisorService* and provides access to the *AdvisorService* through the member variable *advisorService*. Figure 14 shows this relationship.

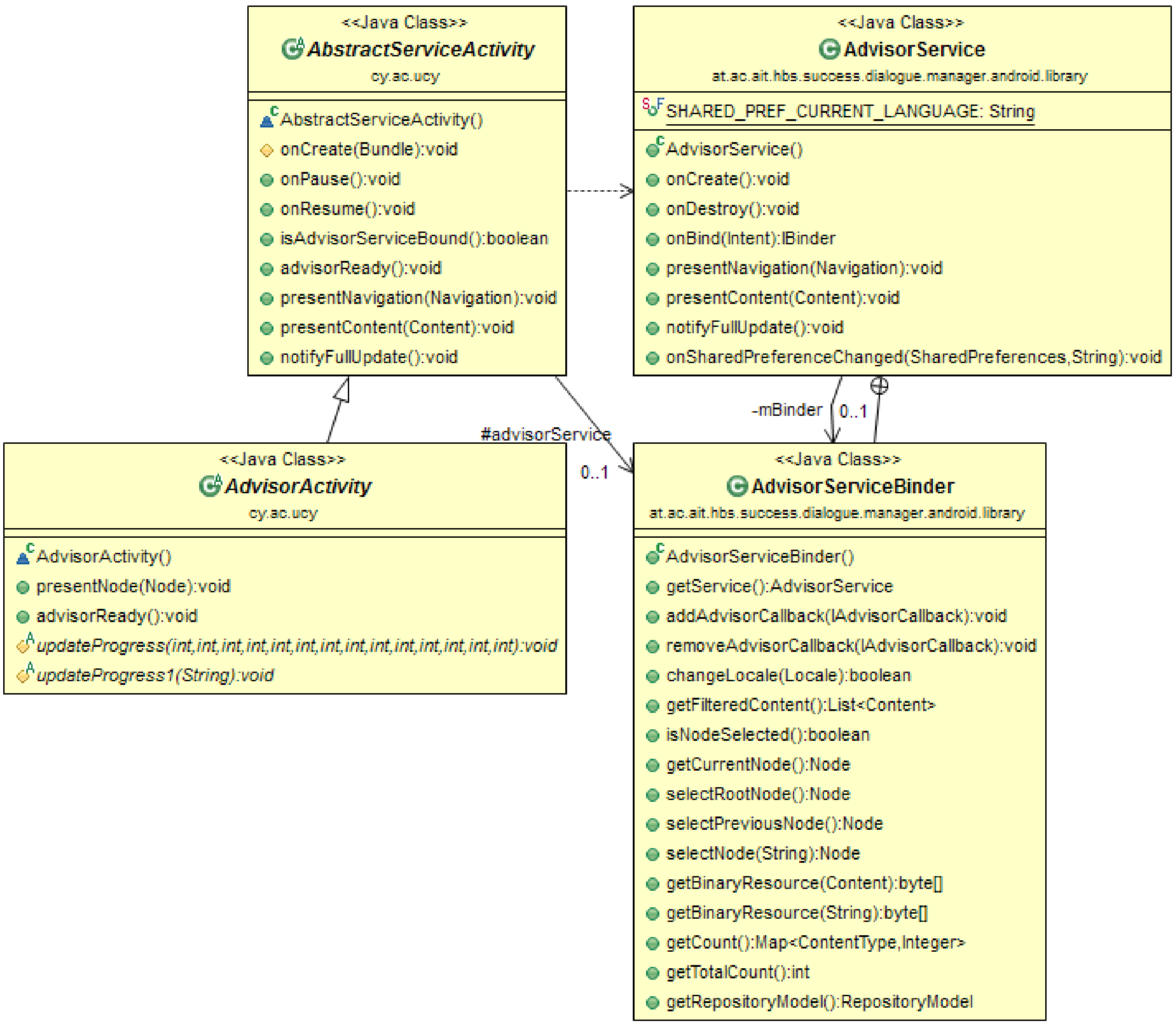
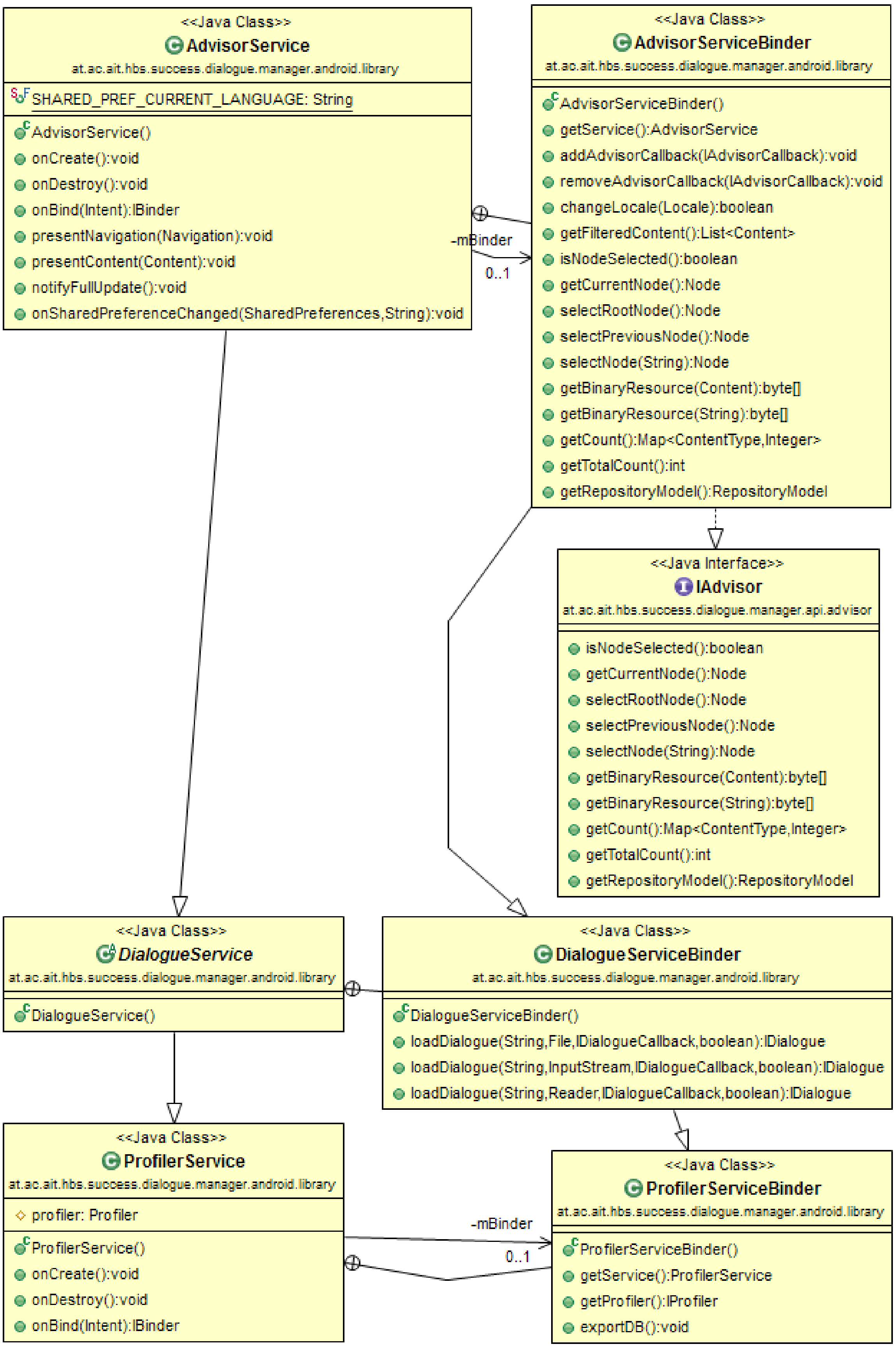

<<Java Class>>
AbstractServiceActivity
cy.ac.ucy
AbstractServiceActivity()
onCreate(Bundle):void
onPause():void
onResume():void
isAdvisorServiceBound():boolean
advisorReady():void
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
<<Java Class>>
AdvisorService
at.ac.ait.hbs.success.dialogue.manager.android.library
SHARED_PREF_CURRENT_LANGUAGE: String
AdvisorService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
onSharedPreferenceChanged(SharedPreferences,String):void
-mBinder
0..1
#advisorService
0..1
<<Java Class>>
AdvisorActivity
cy.ac.ucy
AdvisorActivity()
presentNode(Node):void
advisorReady():void
updateProgress(int,int,int,int,int,int,int,int,int,int,int,int,int,int):void
updateProgress1(String):void
<<Java Class>>
AdvisorServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
AdvisorServiceBinder()
getService():AdvisorService
addAdvisorCallback(IAdvisorCallback):void
removeAdvisorCallback(IAdvisorCallback):void
changeLocale(Locale):boolean
getFilteredContent():List<Content>
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
getRepositoryModel():RepositoryModel

Figure 14: Adviser Service instantiation and provision

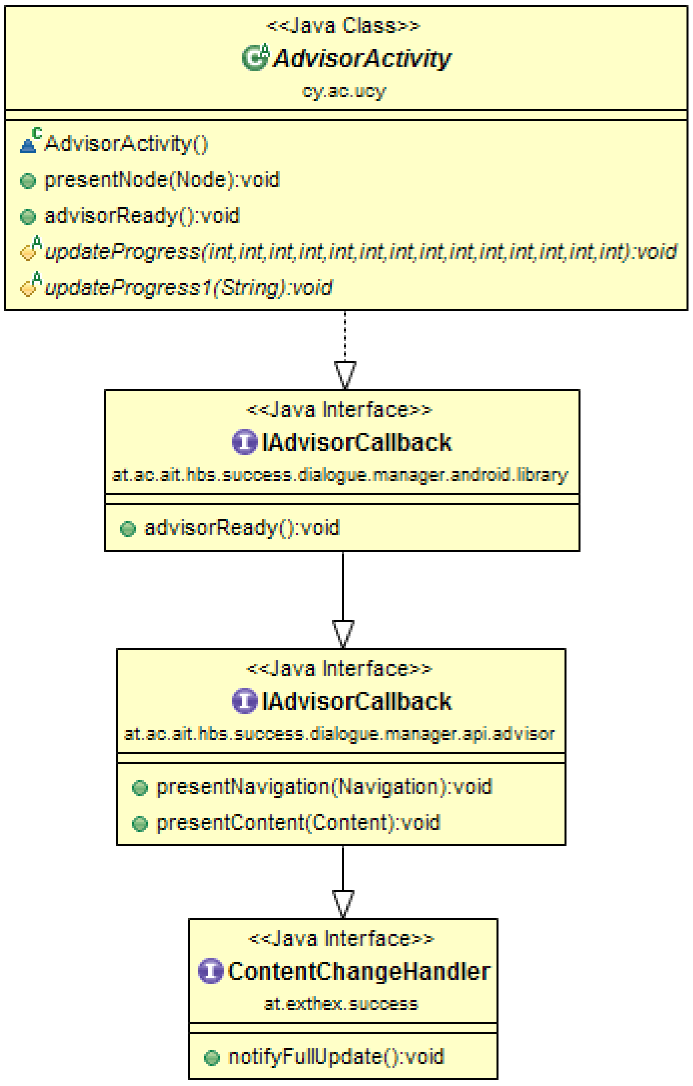
Figure 15 shows the methods available through the *AdvisorService*, which are implemented in inner classes of the parent classes *AdvisorService*, *DialogueService* and *ProfilerService*, suffixed by the word *Binder*, from which the *AdvisorServiceBinder* implements additional methods from the *IAdvisor* interface.



<<Java Class>>
AdvisorService
at.ac.ait.hbs.success.dialogue.manager.android.library
SHARED_PREF_CURRENT_LANGUAGE: String
AdvisorService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
onSharedPreferenceChanged(SharedPreferences,String):void
-mBinder
0..1
<<Java Class>>
AdvisorServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
AdvisorServiceBinder()
getService():AdvisorService
addAdvisorCallback(IAdvisorCallback):void
removeAdvisorCallback(IAdvisorCallback):void
changeLocale(Locale):boolean
getFilteredContent():List<Content>
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
getRepositoryModel():RepositoryModel
<<Java Interface>>
IAdvisor
at.ac.ait.hbs.success.dialogue.manager.api.advisor
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
getRepositoryModel():RepositoryModel
<<Java Class>>
DialogueService
at.ac.ait.hbs.success.dialogue.manager.android.library
DialogueService()
<<Java Class>>
DialogueServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
DialogueServiceBinder()
loadDialogue(String,File,IDialogueCallback,boolean):IDialogue
loadDialogue(String,InputStream,IDialogueCallback,boolean):IDialogue
loadDialogue(String,Reader,IDialogueCallback,boolean):IDialogue
<<Java Class>>
ProfilerService
at.ac.ait.hbs.success.dialogue.manager.android.library
profiler: Profiler
ProfilerService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
-mBinder
0..1
<<Java Class>>
ProfilerServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
ProfilerServiceBinder()
getService():ProfilerService
getProfiler():IProfiler
exportDB():void

Figure 15: Adviser Service interface methods

The class extending the *AdvisorActivity* class must implement various callbacks and two abstract methods so that the Adviser is able to push information to the extending class. Those callback methods are defined in the *IAdvisorCallback* and *ContentChangeHandler* Interfaces as well as in the *AdvisorActivity* class, shown in Figure 16.



<<Java Class>>
AdvisorActivity
cy.ac.ucy
AdvisorActivity()
presentNode(Node):void
advisorReady():void
updateProgress(int,int,int,int,int,int,int,int,int,int,int,int,int,int):void
updateProgress1(String):void
<<Java Interface>>
IAdvisorCallback
at.ac.ait.hbs.success.dialogue.manager.android.library
advisorReady():void
<<Java Interface>>
IAdvisorCallback
at.ac.ait.hbs.success.dialogue.manager.api.advisor
presentNavigation(Navigation):void
presentContent(Content):void
<<Java Interface>>
ContentChangeHandler
at.exthex.success
notifyFullUpdate():void

Figure 16: Adviser callback methods

The following list provides an overview about each callback method:

- advisorReady():
  This method is called when the Adviser was successfully instantiated and the *AdvisorService* is ready.
- presentNavigation(Navigation):
  This method is called when a navigation node is selected.
- presentContent(Content):
  This method is called when a content node is selected.

- notifyFullUpdate():
  This method is called when the content provider published a content update or when the content provider's content tree is reloaded.
- updateProgress(…):
  This method is called when the users content progress is updated.

Whenever the Adviser requests the navigation tree and content nodes from the content provider, the model structure is validated (i.e. if all node ids and references exist) using the *validateModel()* of the *RepositoryModel* class, as shown in Figure 17.

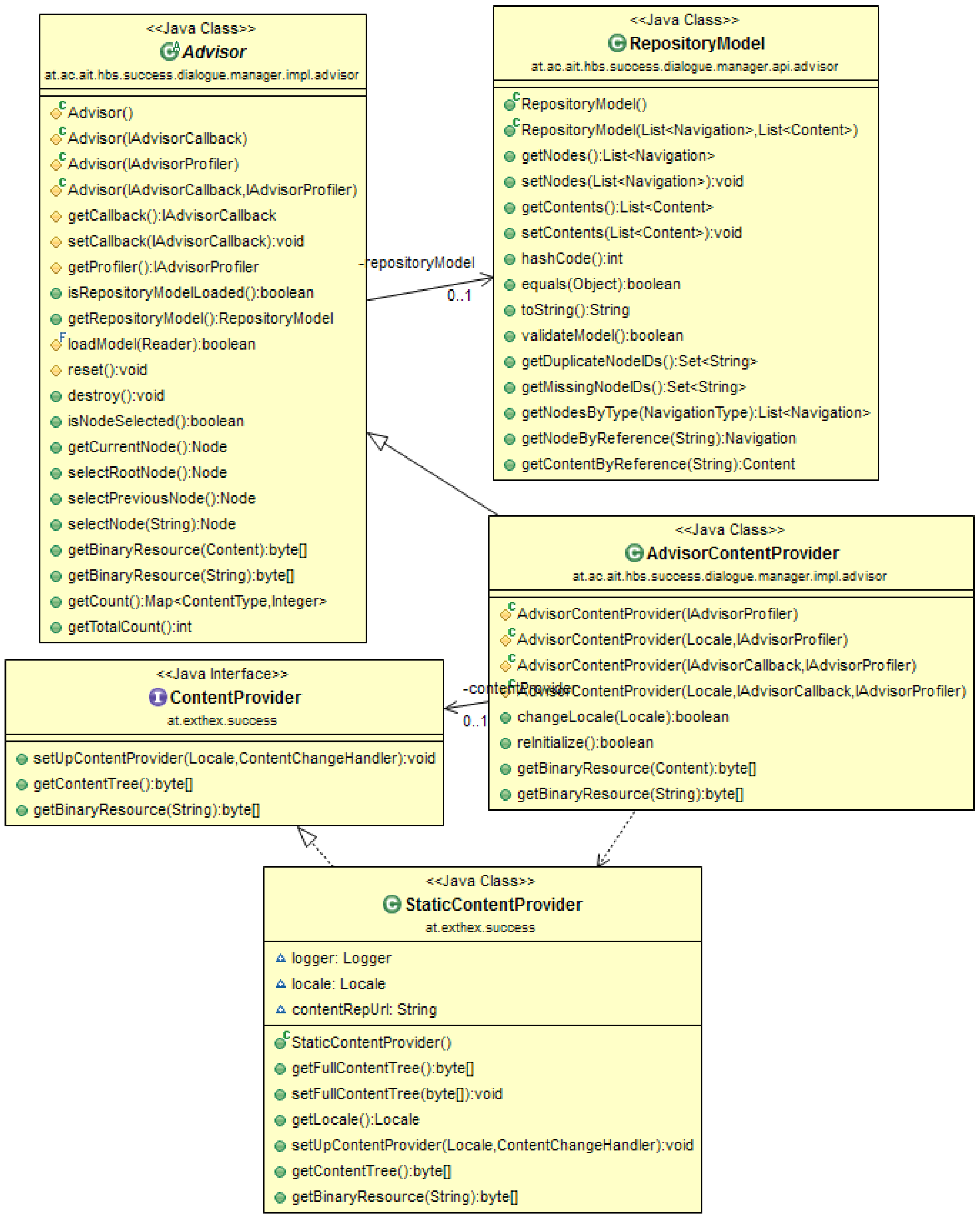
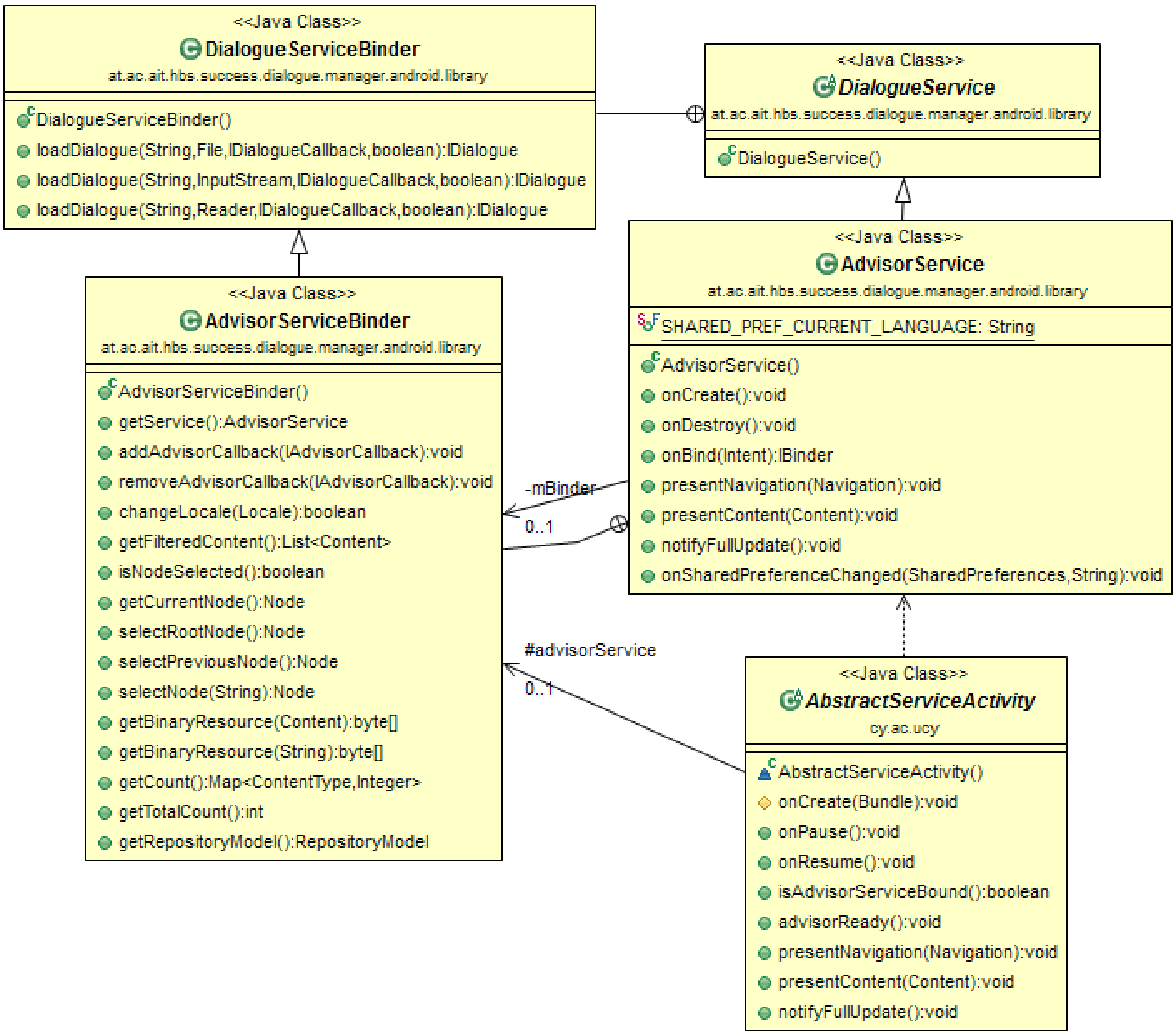

<<Java Class>>
Advisor
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
Advisor()
Advisor(IAdvisorCallback)
Advisor(IAdvisorProfiler)
Advisor(IAdvisorCallback,IAdvisorProfiler)
getCallback():IAdvisorCallback
setCallback(IAdvisorCallback):void
getProfiler():IAdvisorProfiler
isRepositoryModelLoaded():boolean
getRepositoryModel():RepositoryModel
loadModel(Reader):boolean
reset():void
destroy():void
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
-repositoryModel
0..1
<<Java Class>>
RepositoryModel
at.ac.ait.hbs.success.dialogue.manager.api.advisor
RepositoryModel()
RepositoryModel(List<Navigation>,List<Content>)
getNodes():List<Navigation>
setNodes(List<Navigation>):void
getContents():List<Content>
setContents(List<Content>):void
hashCode():int
equals(Object):boolean
toString():String
validateModel():boolean
getDuplicateNodeIDs():Set<String>
getMissingNodeIDs():Set<String>
getNodesByType(NavigationType):List<Navigation>
getNodeByReference(String):Navigation
getContentByReference(String):Content
<<Java Class>>
AdvisorContentProvider
at.ac.ait.hbs.success.dialogue.manager.impl.advisor
AdvisorContentProvider(IAdvisorProfiler)
AdvisorContentProvider(Locale,IAdvisorProfiler)
AdvisorContentProvider(IAdvisorCallback,IAdvisorProfiler)
AdvisorContentProvider(Locale,IAdvisorCallback,IAdvisorProfiler)
changeLocale(Locale):boolean
reInitialize():boolean
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
-contentProvider
0..1
<<Java Interface>>
ContentProvider
at.exthex.success
setUpContentProvider(Locale,ContentChangeHandler):void
getContentTree():byte[]
getBinaryResource(String):byte[]
<<Java Class>>
StaticContentProvider
at.exthex.success
logger: Logger
locale: Locale
contentRepUrl: String
StaticContentProvider()
getFullContentTree():byte[]
setFullContentTree(byte[]):void
getLocale():Locale
setUpContentProvider(Locale,ContentChangeHandler):void
getContentTree():byte[]
getBinaryResource(String):byte[]

Figure 17: Navigation and content model validation

Contents of type *Lecture*, *Diary* and *Roleplay* are presented as interactive turn-based sessions, where the user must provide an answer at every turn in order to get to the next turn. This user dialogue interaction is implemented in a dedicated dialogue Java package. A *Dialogue* instance is loaded through the *loadDialogue* method of the *DialogueServiceBinder* inner class of the *Dia-logueService* class. The *loadDialogue* method is available through the *AdvisorService*, shown in Figure 18.



<<Java Class>>
DialogueServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
DialogueServiceBinder()
loadDialogue(String,File,IDialogueCallback,boolean):IDialogue
loadDialogue(String,InputStream,IDialogueCallback,boolean):IDialogue
loadDialogue(String,Reader,IDialogueCallback,boolean):IDialogue
<<Java Class>>
DialogueService
at.ac.ait.hbs.success.dialogue.manager.android.library
DialogueService()
<<Java Class>>
AdvisorServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
AdvisorServiceBinder()
getService():AdvisorService
addAdvisorCallback(IAdvisorCallback):void
removeAdvisorCallback(IAdvisorCallback):void
changeLocale(Locale):boolean
getFilteredContent():List<Content>
isNodeSelected():boolean
getCurrentNode():Node
selectRootNode():Node
selectPreviousNode():Node
selectNode(String):Node
getBinaryResource(Content):byte[]
getBinaryResource(String):byte[]
getCount():Map<ContentType,Integer>
getTotalCount():int
getRepositoryModel():RepositoryModel
<<Java Class>>
AdvisorService
at.ac.ait.hbs.success.dialogue.manager.android.library
SHARED_PREF_CURRENT_LANGUAGE: String
AdvisorService()
onCreate():void
onDestroy():void
onBind(Intent):IBinder
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void
onSharedPreferenceChanged(SharedPreferences,String):void
-mBinder
0..1
#advisorService
0..1
<<Java Class>>
AbstractServiceActivity
cy.ac.ucy
AbstractServiceActivity()
onCreate(Bundle):void
onPause():void
onResume():void
isAdvisorServiceBound():boolean
advisorReady():void
presentNavigation(Navigation):void
presentContent(Content):void
notifyFullUpdate():void

Figure 18: Load Dialogue provision through the Adviser Service

The *Dialogue* is instantiated in a similar fashion as the *Advisor* instance by using a *DialogueFactory* class, which returns, based on the input parameters of the calling class, the appropriate *Dialogue* instance, as depicted in Figure 19.

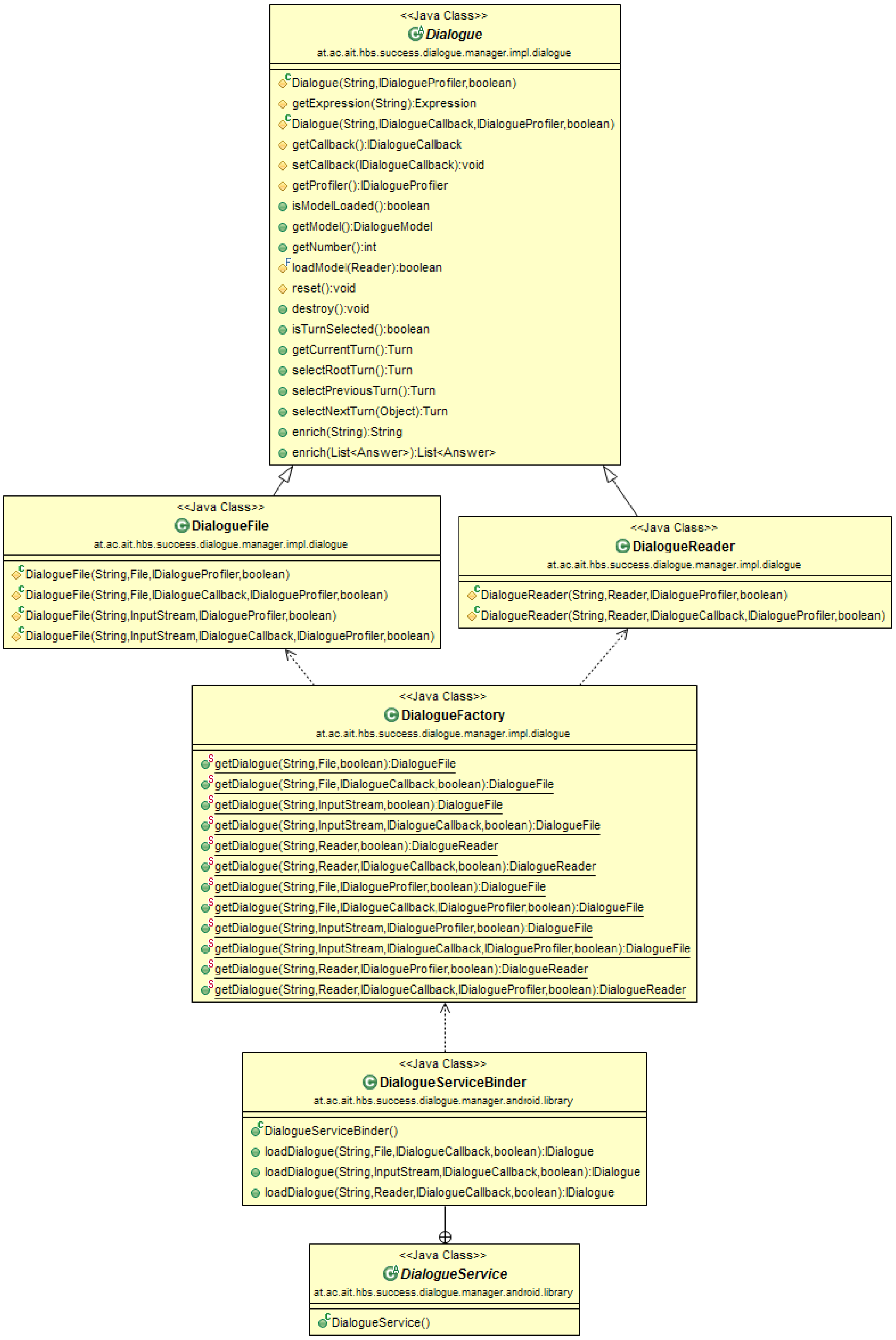

<<Java Class>>
Dialogue
at.ac.ait.hbs.success.dialogue.manager.impl.dialogue
Dialogue(String,IDialogueProfiler,boolean)
getExpression(String):Expression
Dialogue(String,IDialogueCallback,IDialogueProfiler,boolean)
getCallback():IDialogueCallback
setCallback(IDialogueCallback):void
getProfiler():IDialogueProfiler
isModelLoaded():boolean
getModel():DialogueModel
getNumber():int
loadModel(Reader):boolean
reset():void
destroy():void
isTurnSelected():boolean
getCurrentTurn():Turn
selectRootTurn():Turn
selectPreviousTurn():Turn
selectNextTurn(Object):Turn
enrich(String):String
enrich(List<Answer>):List<Answer>
<<Java Class>>
DialogueFile
at.ac.ait.hbs.success.dialogue.manager.impl.dialogue
DialogueFile(String,File,IDialogueProfiler,boolean)
DialogueFile(String,File,IDialogueCallback,IDialogueProfiler,boolean)
DialogueFile(String,InputStream,IDialogueProfiler,boolean)
DialogueFile(String,InputStream,IDialogueCallback,IDialogueProfiler,boolean)
<<Java Class>>
DialogueReader
at.ac.ait.hbs.success.dialogue.manager.impl.dialogue
DialogueReader(String,Reader,IDialogueProfiler,boolean)
DialogueReader(String,Reader,IDialogueCallback,IDialogueProfiler,boolean)
<<Java Class>>
DialogueFactory
at.ac.ait.hbs.success.dialogue.manager.impl.dialogue
getDialogue(String,File,boolean):DialogueFile
getDialogue(String,File,IDialogueCallback,boolean):DialogueFile
getDialogue(String,InputStream,boolean):DialogueFile
getDialogue(String,InputStream,IDialogueCallback,boolean):DialogueFile
getDialogue(String,Reader,boolean):DialogueReader
getDialogue(String,Reader,IDialogueCallback,boolean):DialogueReader
getDialogue(String,File,IDialogueProfiler,boolean):DialogueFile
getDialogue(String,File,IDialogueCallback,IDialogueProfiler,boolean):DialogueFile
getDialogue(String,InputStream,IDialogueProfiler,boolean):DialogueFile
getDialogue(String,InputStream,IDialogueCallback,IDialogueProfiler,boolean):DialogueFile
getDialogue(String,Reader,IDialogueProfiler,boolean):DialogueReader
getDialogue(String,Reader,IDialogueCallback,IDialogueProfiler,boolean):DialogueReader
<<Java Class>>
DialogueServiceBinder
at.ac.ait.hbs.success.dialogue.manager.android.library
DialogueServiceBinder()
loadDialogue(String,File,IDialogueCallback,boolean):IDialogue
loadDialogue(String,InputStream,IDialogueCallback,boolean):IDialogue
loadDialogue(String,Reader,IDialogueCallback,boolean):IDialogue
<<Java Class>>
DialogueService
at.ac.ait.hbs.success.dialogue.manager.android.library
DialogueService()

Figure 19: Dialogue instantiation through the Dialogue Factory

Before the dialogue is processed, the dialogue model is validated (i.e. if all dialogue turn ids and references exist), similarly to the navigation and content model of the Content Provider through the Adviser, using a *validateModel()* method of the *DialogueModel* class, shown in Figure 20.

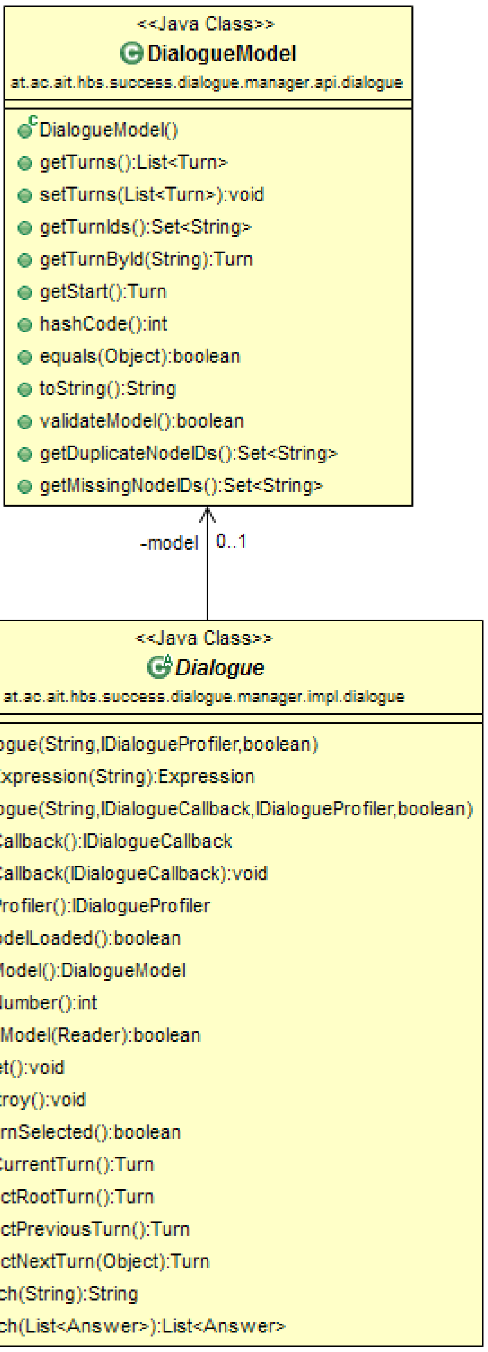
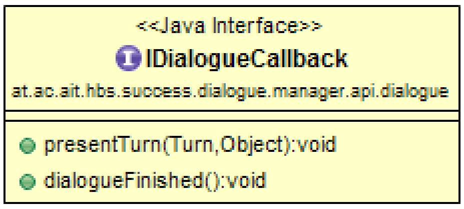

<<Java Class>>
DialogueModel
at.ac.ait.hbs.success.dialogue.manager.api.dialogue
DialogueModel()
getTurns():List<Turn>
setTurns(List<Turn>):void
getTurnIds():Set<String>
getTurnById(String):Turn
getStart():Turn
hashCode():int
equals(Object):boolean
toString():String
validateModel():boolean
getDuplicateNodeIDs():Set<String>
getMissingNodeIDs():Set<String>
-model
0..1
<<Java Class>>
Dialogue
at.ac.ait.hbs.success.dialogue.manager.impl.dialogue
Dialogue(String,IDialogueProfiler,boolean)
getExpression(String):Expression
Dialogue(String,IDialogueCallback,IDialogueProfiler,boolean)
getCallback():IDialogueCallback
setCallback(IDialogueCallback):void
getProfiler():IDialogueProfiler
isModelLoaded():boolean
getModel():DialogueModel
getNumber():int
loadModel(Reader):boolean
reset():void
destroy():void
isTurnSelected():boolean
getCurrentTurn():Turn
selectRootTurn():Turn
selectPreviousTurn():Turn
selectNextTurn(Object):Turn
enrich(String):String
enrich(List<Answer>):List<Answer>

Figure 20: Dialogue model validation

A class or Android activity that needs to present content in a dialogue like form must get an instance of the *Dialogue* class and must implement the *IDialogueCallback* interface to receive updates from the *Dialogue* class. Figure 21 shows the *IDialogueCallback* interface, while Figure 22 shows the *IDialogue* interface which provides information on how to interact with the *Dialogue* instance.

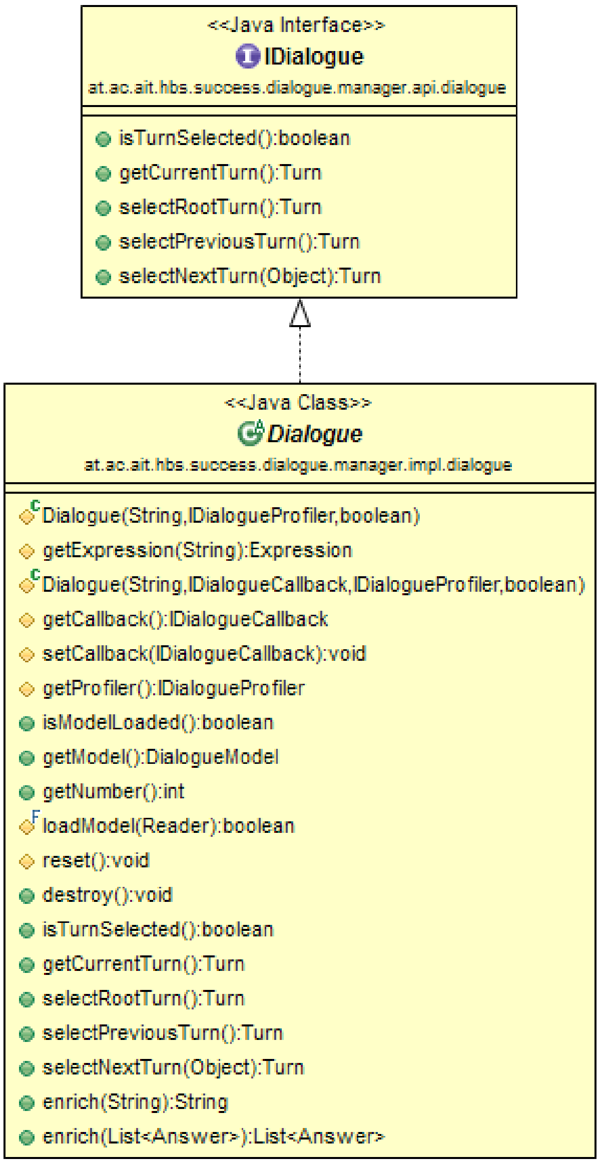


<<Java Interface>>
IDialogueCallback
at.ac.ait.hbs.success.dialogue.manager.api.dialogue
presentTurn(Turn,Object):void
dialogueFinished():void

Figure 21: Dialogue callback interface

The following list provides an overview of each *DialogueCallback* method:

- presentTurn(Turn, Object):
  This method is called by the *Dialogue* instance with the new Turn information and the answer to the previous Turn.
- dialogueFinished():
  This method is called by the *Dialogue* instance when the dialogue is finished.



<<Java Interface>>
IDialogue
at.ac.ait.hbs.success.dialogue.manager.api.dialogue
isTurnSelected():boolean
getCurrentTurn():Turn
selectRootTurn():Turn
selectPreviousTurn():Turn
selectNextTurn(Object):Turn
<<Java Class>>
Dialogue
at.ac.ait.hbs.success.dialogue.manager.impl.dialogue
Dialogue(String,IDialogueProfiler,boolean)
getExpression(String):Expression
Dialogue(String,IDialogueCallback,IDialogueProfiler,boolean)
getCallback():IDialogueCallback
setCallback(IDialogueCallback):void
getProfiler():IDialogueProfiler
isModelLoaded():boolean
getModel():DialogueModel
getNumber():int
loadModel(Reader):boolean
reset():void
destroy():void
isTurnSelected():boolean
getCurrentTurn():Turn
selectRootTurn():Turn
selectPreviousTurn():Turn
selectNextTurn(Object):Turn
enrich(String):String
enrich(List<Answer>):List<Answer>

Figure 22: Dialogue interface and implementation

The following list provides an overview about each *Dialogue* interface method:

- isTurnSelected():
  Method that checks if a turn was selected in the dialogue. Returns true if a turn was selected previously, false otherwise.
- getCurrentTurn():
  Method that returns the turn that is currently selected in the dialogue (if available, null if not).
- selectRootTurn()
  Returns the root turn of the dialogue (if available, null if not).
- selectPreviousTurn()
  Method that returns the turn that is "before" the last selected turn in the dialogue (if available, null if not).
- selectNextTurn(Object input)
  Method that returns the next turn if the input is valid for the current turn. Returns the desired turn or null, if this turn does not exist.

4.3.5 CONTENT PROVIDER

The content provider is the backend component that is responsible for managing and providing the contents that are presented to the Users. The CP communicates with the Content Repository (CR), which stores the content of SUCCESS in a hierarchical way. In addition, the CP communicates with the Dialogue Manager and converts and provides the required resources from the CR. The most important tasks of the CP are the:

- Creation of the Content Tree
- Providing of the necessary binary resources

For the creation of the content tree the CP converts data from the CR into YAML format which is then forwarded to the DM. The content tree contains all necessary meta-data which is needed to present the content in the Output Platform.

The second important task is the transmitting of the actual binary resources. If the user requests content, the DM makes a request to the CP with the reference and the CP gives the corresponding binary resource back to the DM. Figure 23 presents the level 1 class diagram of the content provider, where the aforementioned main tasks and functionalities are demonstrated.

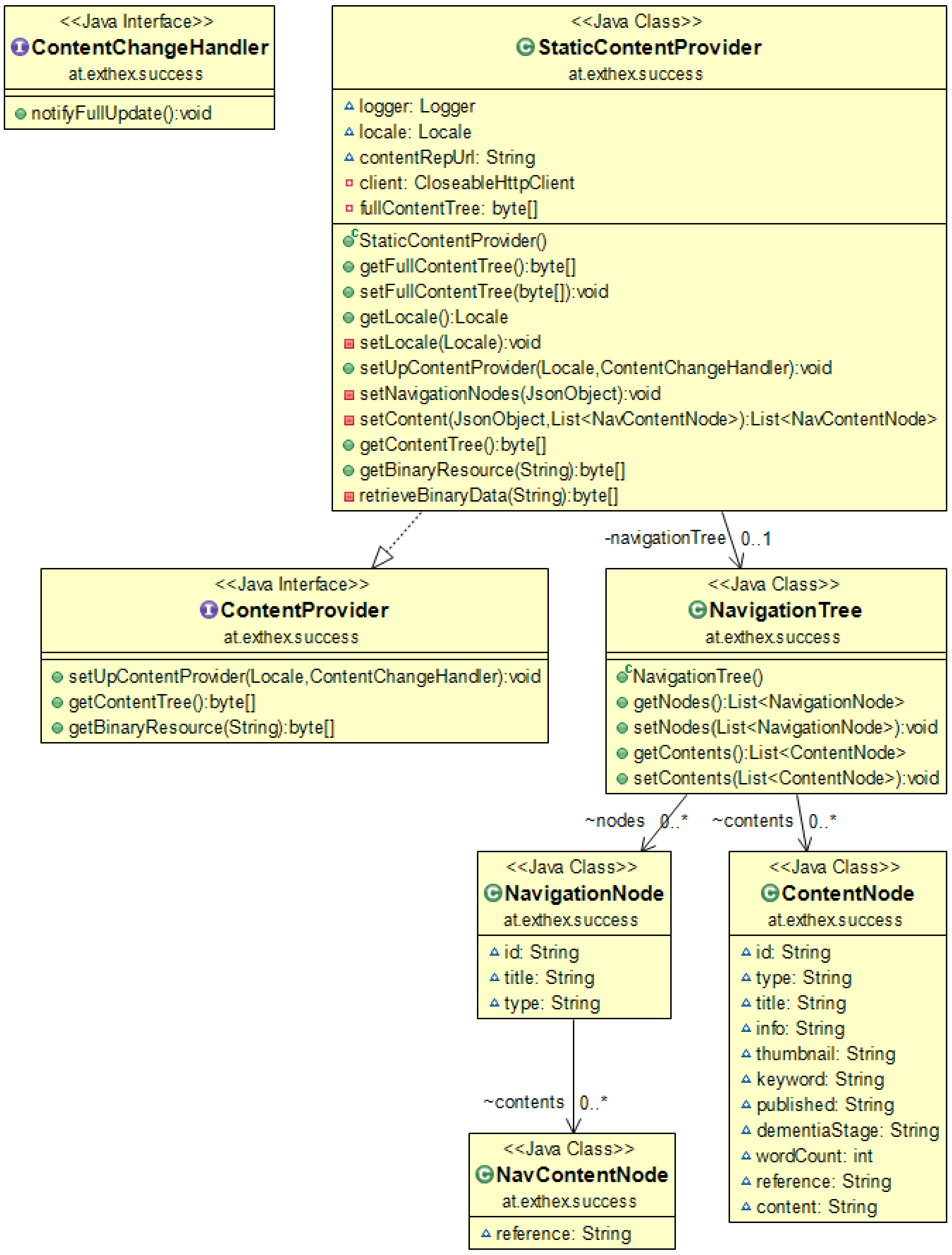
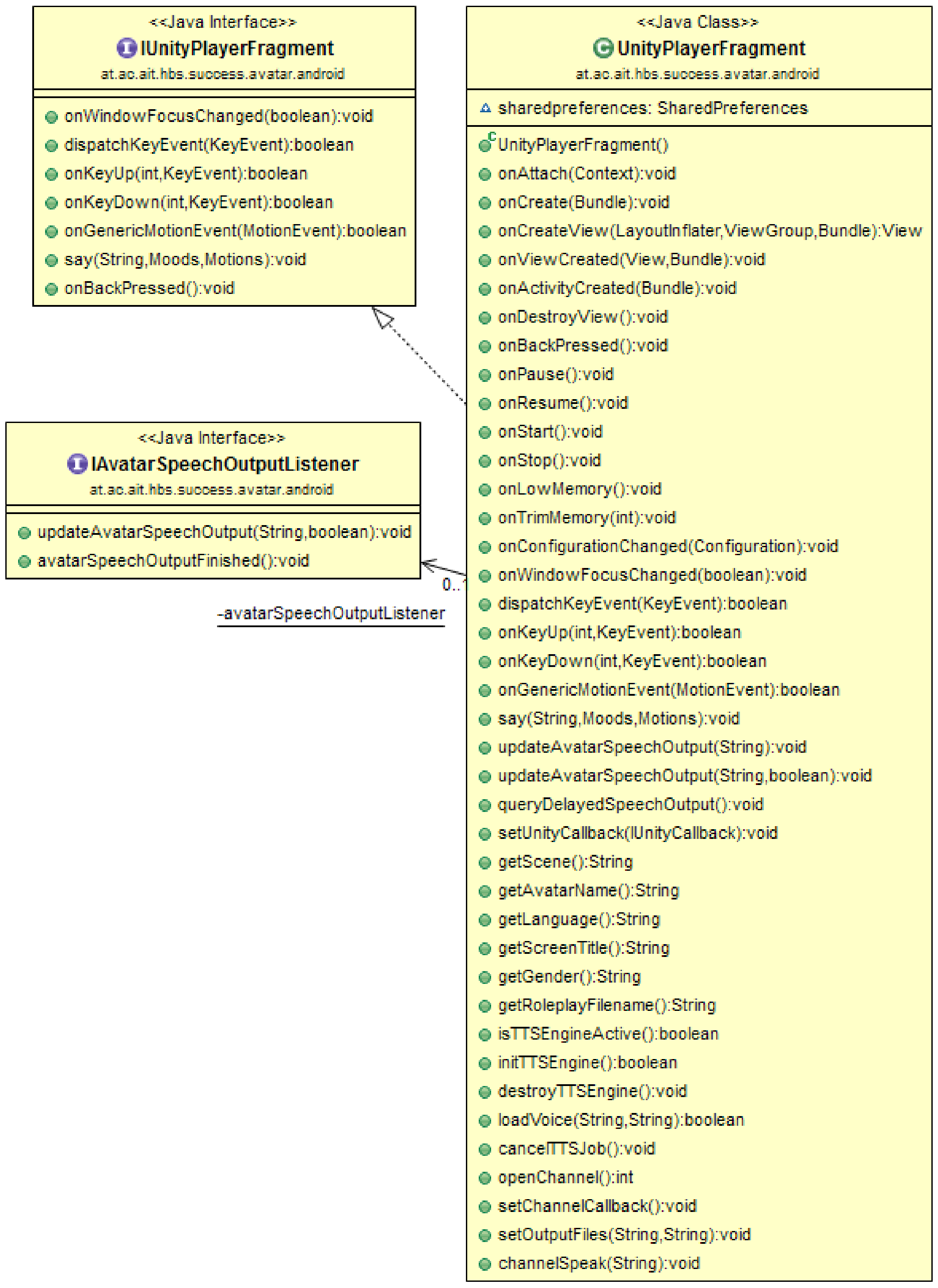


Figure 23: Logical View of Content Provider

4.3.6 AVATAR

The purpose of the avatar component is to provide a more natural way of interaction with the SUCCESS system. The avatar is used in lecture sessions to guide the user through dialogue interactions or acts as an interaction partner in roleplay sessions.

The avatar is implemented in Unity and wrapped in an Android fragment activity [3], implementing an Interface that defines how other components can interact with the Avatar component, shown in Figure 24.



<<Java Interface>>
IUnityPlayerFragment
at.ac.ait.hbs.success.avatar.android
onWindowFocusChanged(boolean):void
dispatchKeyEvent(KeyEvent):boolean
onKeyUp(int,KeyEvent):boolean
onKeyDown(int,KeyEvent):boolean
onGenericMotionEvent(MotionEvent):boolean
say(String,Moods,Motions):void
onBackPressed():void
<<Java Interface>>
IAvatarSpeechOutputListener
at.ac.ait.hbs.success.avatar.android
updateAvatarSpeechOutput(String,boolean):void
avatarSpeechOutputFinished():void
0..1
-avatarSpeechOutputListener
<<Java Class>>
UnityPlayerFragment
at.ac.ait.hbs.success.avatar.android
sharedpreferences: SharedPreferences
UnityPlayerFragment()
onAttach(Context):void
onCreate(Bundle):void
onCreateView(LayoutInflater,ViewGroup,Bundle):View
onViewCreated(View,Bundle):void
onActivityCreated(Bundle):void
onDestroyView():void
onBackPressed():void
onPause():void
onResume():void
onStart():void
onStop():void
onLowMemory():void
onTrimMemory(int):void
onConfigurationChanged(Configuration):void
onWindowFocusChanged(boolean):void
dispatchKeyEvent(KeyEvent):boolean
onKeyUp(int,KeyEvent):boolean
onKeyDown(int,KeyEvent):boolean
onGenericMotionEvent(MotionEvent):boolean
say(String,Moods,Motions):void
updateAvatarSpeechOutput(String):void
updateAvatarSpeechOutput(String,boolean):void
queryDelayedSpeechOutput():void
setUnityCallback(IUnityCallback):void
getScene():String
getAvatarName():String
getLanguage():String
getScreenTitle():String
getGender():String
getRoleplayFilename():String
isTTSEngineActive():boolean
initTTSEngine():boolean
destroyTTSEngine():void
loadVoice(String,String):boolean
cancelTTSJob():void
openChannel():int
setChannelCallback():void
setOutputFiles(String,String):void
channelSpeak(String):void

Figure 24: Unity Player Fragment with Unity Player Fragment interface implementation and Avatar Speech Output Listener callback interface

The following list provides an overview about each *IUnityPlayerFragment* interface method:

- onWindowFocusChanged(boolean), dispatchKeyEvent(KeyEvent), onKeyUp(int, KeyEvent), onKeyDown(int, KeyEvent), onGenericMotionEvent(MotionEvent), onBackPressed()
  Those are native Android callback methods that should be implemented by the parent Activity and forwarded to the Avatar Fragment, so that certain actions can take place in Unity, like for example pausing the Avatar animation when the user tabs on the Smartphone.

- say(String, Moods, Motions)
  This method is used tell the Avatar what it should say (through the *String* argument), along with which Mood and Motion should be used for the given text (through the *Moods* and *Motions* arguments).

Currently available *Moods* and *Motions* are defined as Java Enums, shown in Figure 25.

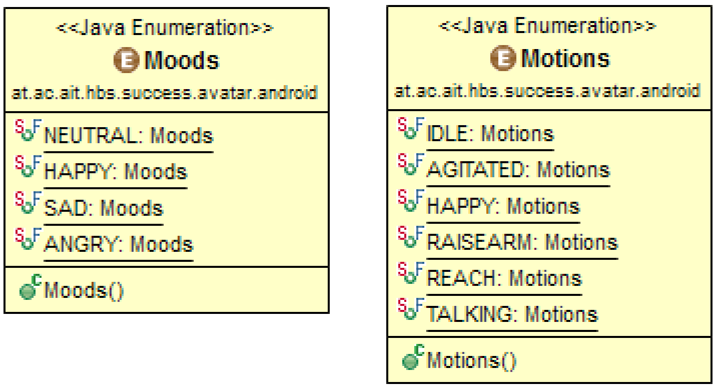


Figure 25: Moods and Motions Java enumerations

The *IAvatarSpeechOutputListener* defines callback methods that may be implemented by the parent Activity of the Fragment:

- updateAvatarSpeechOutput(String, Boolean)
  The Avatar Fragment calls this method continuously while the Avatar is speaking by providing the currently spoken word as argument. This enables the parent Activity to show the currently spoken text to the user, for example in a “speech bubble”.

- avatarSpeechOutputFinished()
  This method is called when the Avatar speech output is finished.

The *UnityPlayerFragment* fragment itself instantiates the *UnityPlayer* class provided by Unity and communicates with the underlying Unity engine through the *UnityPlayer* class and a dedicated Avatar *IUnityCallback* interface, depicted in Figure 26.

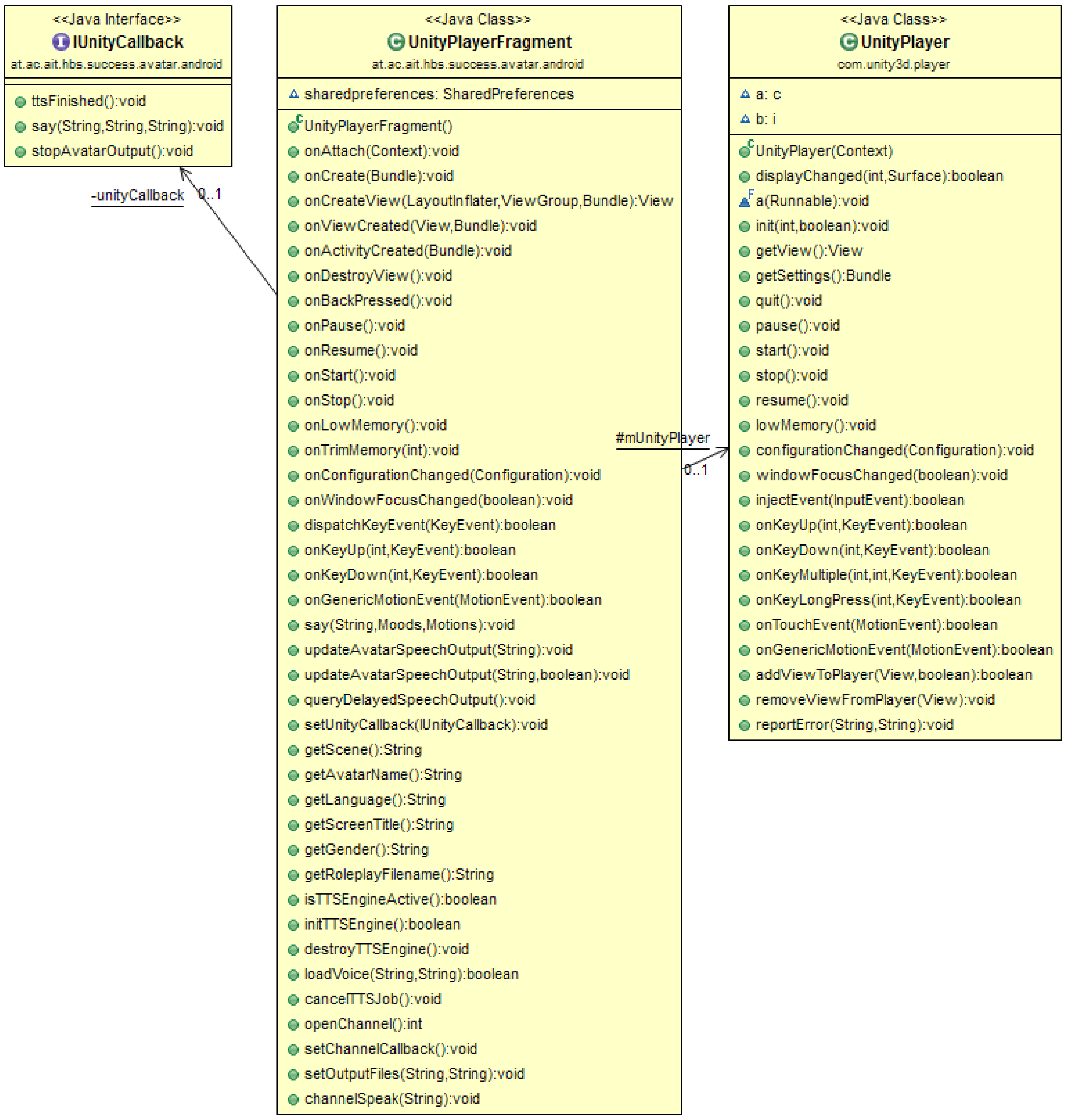

<<Java Interface>>
IUnityCallback
at.ac.ait.hbs.success.avatar.android
ttsFinished():void
say(String,String,String):void
stopAvatarOutput():void
-unityCallback
0..1
<<Java Class>>
UnityPlayerFragment
at.ac.ait.hbs.success.avatar.android
sharedpreferences: SharedPreferences
UnityPlayerFragment()
onAttach(Context):void
onCreate(Bundle):void
onCreateView(LayoutInflater,ViewGroup,Bundle):View
onViewCreated(View,Bundle):void
onActivityCreated(Bundle):void
onDestroyView():void
onBackPressed():void
onPause():void
onResume():void
onStart():void
onStop():void
onLowMemory():void
onTrimMemory(int):void
onConfigurationChanged(Configuration):void
onWindowFocusChanged(boolean):void
dispatchKeyEvent(KeyEvent):boolean
onKeyUp(int,KeyEvent):boolean
onKeyDown(int,KeyEvent):boolean
onGenericMotionEvent(MotionEvent):boolean
say(String,Moods,Motions):void
updateAvatarSpeechOutput(String):void
updateAvatarSpeechOutput(String,boolean):void
queryDelayedSpeechOutput():void
setUnityCallback(IUnityCallback):void
getScene():String
getAvatarName():String
getLanguage():String
getScreenTitle():String
getGender():String
getRoleplayFilename():String
isTTSEngineActive():boolean
initTTSEngine():boolean
destroyTTSEngine():void
loadVoice(String,String):boolean
cancelTTSJob():void
openChannel():int
setChannelCallback():void
setOutputFiles(String,String):void
channelSpeak(String):void
#mUnityPlayer
0..1
<<Java Class>>
UnityPlayer
com.unity3d.player
a: c
b: i
UnityPlayer(Context)
displayChanged(int,Surface):boolean
a(Runnable):void
init(int,boolean):void
getView():View
getSettings():Bundle
quit():void
pause():void
start():void
stop():void
resume():void
lowMemory():void
configurationChanged(Configuration):void
windowFocusChanged(boolean):void
injectEvent(InputEvent):boolean
onKeyUp(int,KeyEvent):boolean
onKeyDown(int,KeyEvent):boolean
onKeyMultiple(int,int,KeyEvent):boolean
onKeyLongPress(int,KeyEvent):boolean
onTouchEvent(MotionEvent):boolean
onGenericMotionEvent(MotionEvent):boolean
addViewToPlayer(View,boolean):boolean
removeViewFromPlayer(View):void
reportError(String,String):void

Figure 26: Unity Player integration and callback interface

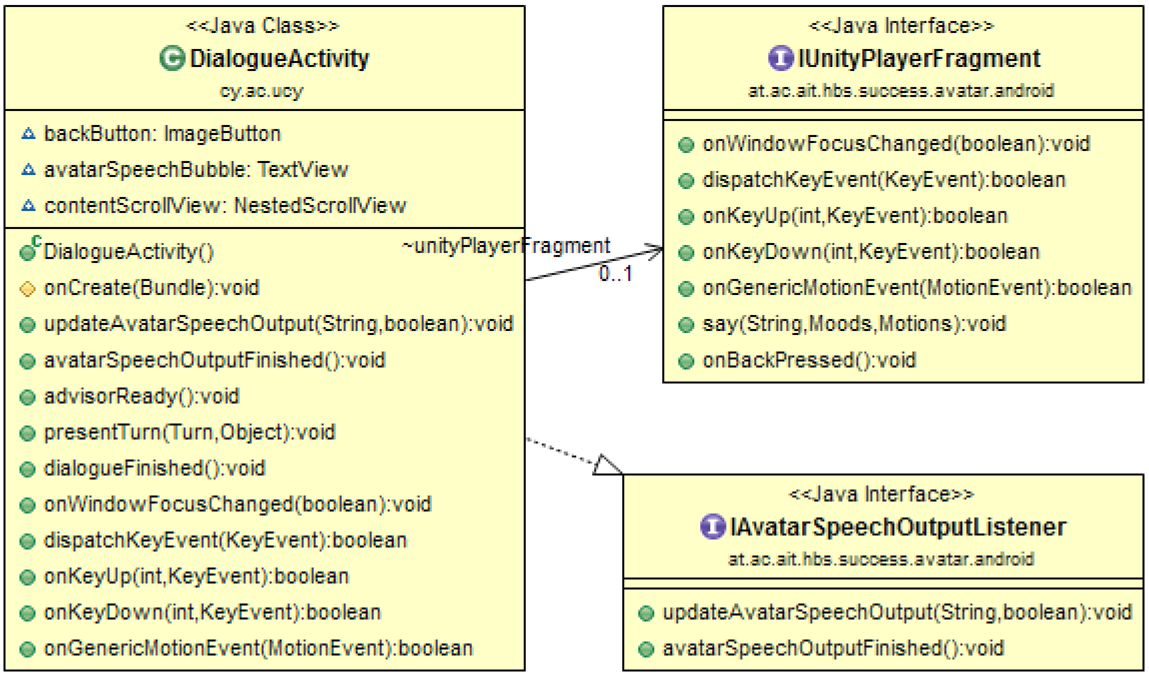
The following list provides an overview about the most important *UnityPlayer* and *IUnityCallback* interface methods:

- onWindowFocusChanged(boolean), onKeyUp(int, KeyEvent), onKeyDown(int, KeyEvent), onGenericMotionEvent(MotionEvent)
  Implementations for the native Android callback methods and should be forwarded by the Avatar fragment using the UnityPlayer class.
- injectEvent(InputEvent)
  Unity convenience method to handle different input events registered on the Android device. It can be seen as a "shortcut" for above mentioned methods.

- quit(), pause(), start(), stop(), resume()
  Methods used to control the Unity engine runtime behaviour.
- say(String, String, String)
  Method used to tell the Avatar what it should say (first argument), along with which Mood and Motion should be used for the given text (second and third arguments, respectively).
- ttsFinished()
  Method used to inform Unity when an Android TTS (text-to-speech) action has finished.
- stopAvatarOutput()
  Method used to stop the currently playing Avatar voice and animation output (for example to skip a dialogue turn because the user heard it already).

Other public methods from the *UnityPlayerFragment* are only used from within the Unity application to communicate with the *UnityPlayerFragment* Java class.

Using Androids fragment functionality makes this component most flexible regarding its usages within a running Android application, since any Android activity within an application can make use of the Avatar fragment, providing an additional way of user interaction and interaction experience for the SUCCESS application. The usage of the Avatar fragment in one of the SUCCESS Android activities is shown in Figure 27.



<<Java Class>>
DialogueActivity
cy.ac.ucy
backButton: ImageButton
avatarSpeechBubble: TextView
contentScrollView: NestedScrollView
DialogueActivity()
onCreate(Bundle):void
updateAvatarSpeechOutput(String,boolean):void
avatarSpeechOutputFinished():void
advisorReady():void
presentTurn(Turn,Object):void
dialogueFinished():void
onWindowFocusChanged(boolean):void
dispatchKeyEvent(KeyEvent):boolean
onKeyUp(int,KeyEvent):boolean
onKeyDown(int,KeyEvent):boolean
onGenericMotionEvent(MotionEvent):boolean
~unityPlayerFragment
0..1
<<Java Interface>>
IUnityPlayerFragment
at.ac.ait.hbs.success.avatar.android
onWindowFocusChanged(boolean):void
dispatchKeyEvent(KeyEvent):boolean
onKeyUp(int,KeyEvent):boolean
onKeyDown(int,KeyEvent):boolean
onGenericMotionEvent(MotionEvent):boolean
say(String,Moods,Motions):void
onBackPressed():void
<<Java Interface>>
IAvatarSpeechOutputListener
at.ac.ait.hbs.success.avatar.android
updateAvatarSpeechOutput(String,boolean):void
avatarSpeechOutputFinished():void

Figure 27: Avatar Android Fragment (Unity Player Fragment) usage in Dialogue Activity

4.4 PROCESS VIEW

The process view deals with the system processes and how they communicate, and focuses on the non-functional requirements of the system such as availability, performance, scalability, etc. This view presents tasks that the system has, interfaces among components within the system, possible messages sent and received, and how aspects like performance, availability, fault-tolerance, and integrity are being addressed. In the context of SUCCESS, workflow diagrams will be used which will describe the processes and the interfacing among the main components of the system. To this end, in the subsection 4.4.1 the process flow of the communication between the Output Platform and the Dialogue Manager will be defined, mainly focused on the way that the content is provided to the user. Subsection 4.4.2 will provide the flow of the communication between the Dialogue Manager and the Content Provider, emphasizing on how both components handle the content that is provided to the end users.

4.4.1 OUTPUT PLATFORM – DIALOGUE MANAGER

The Output Platform interacts with the Dialogue Manager by rendering the proper content that is requested by the user. The Output Platform is mainly responsible for coordinating a user driven process, where it relays the user input to the dialogue manager. According to this, the dialogue manager pushes the proper information to the Output Platform which provides the means to present the information to the end user in a unified manner in order to sustain uniformity.

The workflow diagram in Figure 28 shows the communication between Output Platform and Dialogue Manager. At first, the user has to log into the application(Output Platform), through Google API, and gets authenticated and authorized in order to access the Success application services. As soon as the user has successfully logged in, he requests the Output Platform for a resource to be provided(e.g., file, video, lecture, etc.). The request is forwarded to the Dialogue Manager which then pushes the proper information to the Output Platform to be presented to the user.

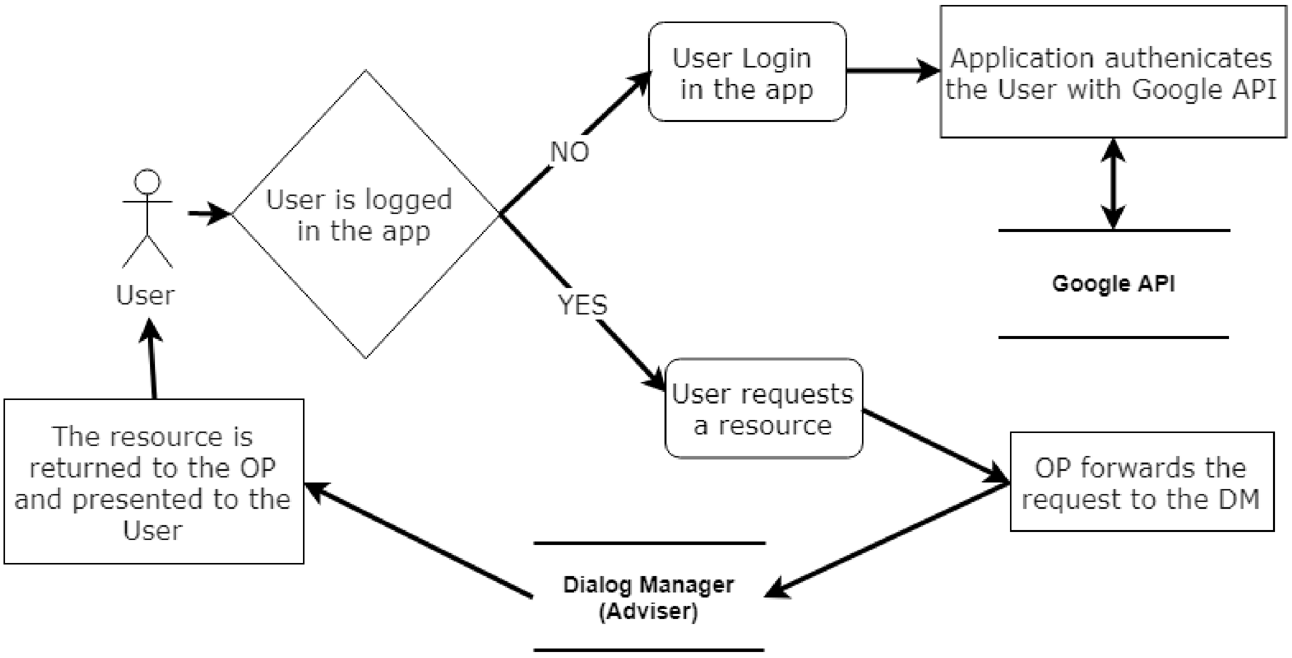

User Login in the app
Application authenicates the User with Google API
NO
User
User is logged in the app
YES
Google API
User requests a resource
The resource is returned to the OP and presented to the User
OP forwards the request to the DM
Dialog Manager (Adviser)

Figure 28: Output Platform – Dialog Manager Process Flow Diagram

4.4.2 DIALOGUE MANAGER – CONTENT PROVIDER

The workflow diagram in Figure 29 shows the communication between Dialogue Manager, Content Provider and Content Repository. When the user installs the app, the DM sets up the CP with language as parameter. After that, the CP retrieves the whole content tree in the right language from the CR and parses the tree into turn cards in a YAML file. Simultaneously, the DM registers the Content Change Handler, to receive possible updates of the content. Afterwards the DM retrieves the content tree, which returns the YAML file in byte[]. Next, the DM can retrieve the binary resources with the parameter reference. If the resource is not already stored locally, the CP retrieves the binary resource of the content from the CR. Irrespective of whether the binary resource was already stored locally at the CP or whether it has to be fetched first, the binary resource is returned to the DM.

In case the admin makes a change to the content tree, the CP is informed and calls a method of the previously registered Content Change Handler, which will update the content. Subsequently, the DM calls the method responsible for the content tree retrieval again, and the flow starts from this point again.

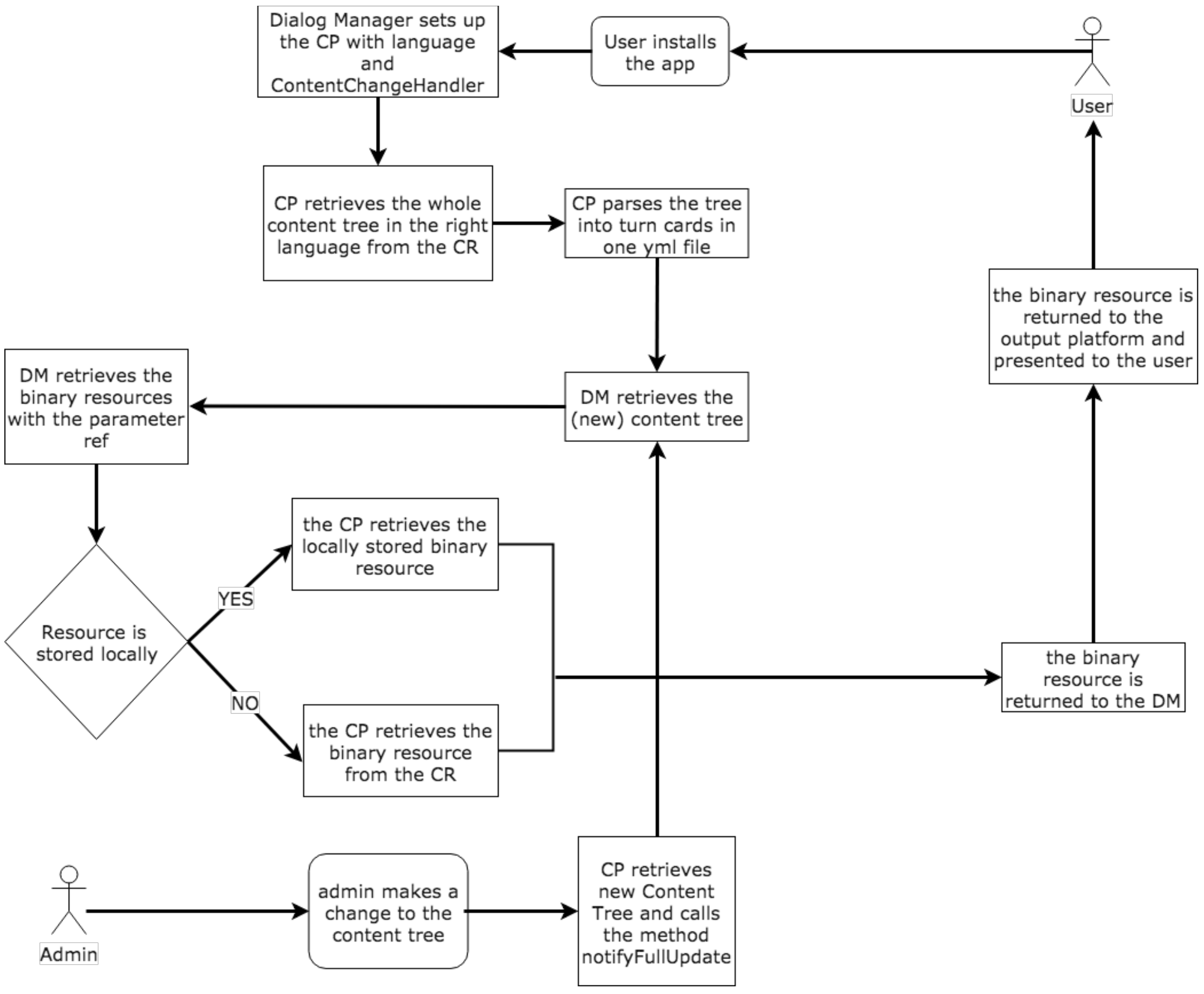

Dialog Manager sets up the CP with language and ContentChangeHandler
User installs the app
User
CP retrieves the whole content tree in the right language from the CR
CP parses the tree into turn cards in one yml file
the binary resource is returned to the output platform and presented to the user
DM retrieves the binary resources with the parameter ref
DM retrieves the (new) content tree
the CP retrieves the locally stored binary resource
YES
Resource is stored locally
NO
the binary resource is returned to the DM
the CP retrieves the binary resource from the CR
Admin
admin makes a change to the content tree
CP retrieves new Content Tree and calls the method notifyFullUpdate

Figure 29: Dialogue Manager - Content Provider Process Flow Diagram

4.5 INTERFACES

Since the different stakeholder perspectives were defined in the previous chapters, the fifth view of the 4+1 view model, the scenario view, will help to capture the requirements so that all the stakeholders understand how the system is intended to be used. In this context, two approaches are presented, the first one focuses on the user interfaces, how the user will use the system and which component provides the functionalities of the application. The second one is the communication interface that provides a technical representation of the communication among the components.

4.5.1 USER INTERFACES

4.5.1.1 OUTPUT PLATFORM

Table 1 depicts the User Interfaces of the Output Platform that have been already implemented at the first development phase by covering the range of activities defined in the Use Cases. Additionally, it defines the activities that will be implemented in the next iteration of the development process.

Interface	Description
Login Screen	The Login Screen provides a graphical interface to the user, which can be used to login to the platform.
Communication Strategies Screen	This screen provides innovative training with the avatar for evidence-based communication and intervention strategies
Situation Guidance Screen	This screen provides general methods and guides users to effectively respond to specific situations
Emotional Support Screen	This screen provides strategies for emotional support as to maintain on the carers' own identity
Care Activities Screen	This screen helps carers to create meaningful activities, thus, maintaining a sense of purpose at the individual's level of ability.
Stigmatisation Avoidance Screen	This screen assists the users to learn how to cope with behaviours and address their own feelings and avoid stigmatisation
Gamification Screens	Various screens will present games to provide an interesting and innovative way of teaching and guidance

Table 1: User Interfaces of the Output Platform

4.5.1.2 PROFILER

The Profiler provides information to various sections of the user interface shown by the output platform, which is listed in Table 2.

Interface	Description
Main Screen	Information about how much and which content the user has seen, or "content progress information".
Main Screen	Information on whether the "diary dialogue" interaction should be initiated again based on how much time passed since the last "diary dialogue" interaction.
Settings	Possibility to reset the content progress to the initial values of 0, i.e. nothing has been seen yet.
Content list	Information on whether a content element is bookmarked

Content list	Information on whether a content element is new
Content list	Information on whether a content element was read
Content screen	Possibility to add or remove a bookmark from the content
Dialogue content	Information for each dialogue turn

Table 2: User Interfaces of the Profiler

4.5.1.3 REWARDER

Table 3 shows the sole user interface that the Rewarder has. It is used to provide the feedback generated to the user.

Interface	Description
Feedback Screen	Provides feedback information

Table 3: User Interfaces of the Rewarder

4.5.1.4 ADVISER

Table 4 shows the User Interfaces for the Adviser. The adviser needs to offer a UI for various content types and be able to display avatars according to the dialogue execution state. Moreover, based on user performance data Adviser filters the content accordingly.

Interface	Description
Navigation and Content	Queries and prepares navigation and content information from the Content Provider for the Output Platform
Navigation and Content	Updates navigation and content elements when changes are published by the Content Provider
Content list	Filters content based on user preference
Dialogue	Provides dialogue style interaction with the user

Table 4: User Interfaces of the Adviser

4.5.1.5 CONTENT PROVIDER

As can be seen in Table 5, the Content Provider offers two user interfaces for Administrator users to be able to manage the Meta Model and the content of the content nodes. It defines the activities that will be implemented in the next iteration of the development process.

Interface	Description
Meta Model Management Interface	Administrative User Interface for managing the content tree structure i.e. the Meta Model structure (may also be scripted in the first version)
Content Node Management Interface	Administrative User Interface for managing the binary content of specific Content Nodes

Table 5: User Interfaces of the Content Provider

4.5.1.6 AVATAR

As defined in Table 6, the Avatar provides an additional user interface through talking and moving as well as a simulation of certain interactive situations.

Interface	Description
Dialogue	Provides an additional user interface and interaction experience through a talking and moving Avatar
Roleplay	Provides a simulation of certain interactive situations for the user with an Avatar

Table 6: User Interfaces of the Avatar

4.5.2 COMMUNICATION INTERFACES

4.5.2.1 OUTPUT PLATFORM

The Interfaces of the Output Platform with other modules are depicted in Table 7. The Profiler needs to communicate the output of the questionnaires to the Output Platform and provides the usage metrics needed by the Adviser. In the data level, the Profiler needs to interface with the User Model in order to be able to retrieve information about the user.

Interface	Description	Interfacing Component	Type	Format
Login	User can login to the application	Google Sign-In	Google API	HTTPS
Sync with cloud	User preferences can be stored and synchronized throughout different devices.	Google Sign-In	Google Drive REST API	HTTPS
User Actions	User can navigate through the application to retrieve information.	Adviser	Java internal	Java Objects/binary

Table 7: Communication Interfaces of the Output Platform

4.5.2.2 PROFILER

The Interfaces of the Profiler with other modules are depicted in Table 8. The Profiler provides interfaces for general application activity as well as interfaces for both adviser and dialogue specific activities. In the data level, the Profiler needs to interface with the Profiler Database in order to be able to retrieve information about the user. The interfaces are internal Java where interfaces are declared inside another interface or class and exchanges Java objects.

Interface	Description	Interfacing Component	Type	Format
IProfiler	Profiler interface for general application activity	Output Platform	Java internal	Java Objects
IAdvisorProfiler	Profiler interface for adviser specific activities	Adviser Output Platform	Java internal	void
IDialogueProfiler	Profiler interface for dialogue specific activities	Adviser/Dialogue	Java internal	Java Objects
ProfilerDatabase	Database interface to read and write user profile data	ProfilerDatabase	Java internal	Java Objects

Table 8: Communication Interfaces of the Profiler

4.5.2.3 REWARDER

Table 9 lists the interfaces the Rewarder shares with some of the components of the SUCCESS solution. Due to the fact that this component is under development the specific interfaces are not defined yet, but the description of the interfaces that are intended to be used is provided.

Interface	Description	Interfacing Component	Type	Format
To be defined	Send Output Requests for Feedback	Output Platform		
	Receive User Interaction data			
	Receive feedback content nodes	Content Provider		
	Send Performance Measures	Profiler		
	Receive Performance Metrics			

Table 9: Communication Interfaces of the Rewarder

4.5.2.4 ADVISER

Table 10 lists the interfaces of the Adviser with the other components of the SUCCESS solution. According to the current state of the architecture, most interfaces are internal Java where interfaces are declared inside another interface or class and exchange Java objects. The Adviser communicates with the Output Platform by generating real time interaction instructions for the Output platform's avatar component.

Interface	Description	Interfacing Component	Type	Format
IAdvisor	Provide navigation/content tree and select navigation/content nodes	Output Platform	Java internal	Java Objects
	Query binary resources	Output Platform	Java internal	binary
IAdvisorCallback	Callback interface enabling the Adviser to push information to components using the Adviser	Output Platform	Java internal	Java Objects
IDialogueCallback	Callback interface enabling the Dialogue to push information to components using the Dialogue	Output Platform	Java internal	Java Objects

Table 10: Communication Interfaces of the Adviser

4.5.2.5 CONTENT PROVIDER

Table 11 lists the interfaces the Content Provider shares with the other components of the SUCCESS solution. The Meta Model API which is used to transverse the Content Tree that is shared with three components (Rewarder, Adviser and Output Platform) and the data will be in YAML. The actual contents of content nodes will be transmitted to the Adviser in binary format and the data from the Content Repository will be in JSON format.

Interface	Description	Interfacing Component	Type (REST, queue, socket)	Format (e.g. json)
Meta Model API	This interface will allow interaction with the Content Tree (traversing the tree and node structure)	Rewarder Adviser Output Platform	Java internal	YAML
Content API	This interface will provide binary content from specific content nodes	Adviser	Java internal	binary
Repository API	This interface will provide data (content, content tree) from the Content Repository	Content Provider	REST	JSON binary

Table 11: Communication Interfaces of the Content Provider

4.5.2.6 AVATAR

Table 12 lists the interfaces the Avatar shares with the Output Platform. The interfaces are intended to provide the actual interaction among the Avatar and the Output Platform and based on the current state of development, most interfaces are internal Java and exchange Java objects.

Interface	Description	Interfacing Component	Type	Format
IUnityPlayerFragment	Provides the means to interact with the Avatar	Output Platform	Java internal	Java Objects
IAvatarSpeechOutputListener	Callback interface enabling the parent Activity to react on currently spoken text by the Avatar	Output Platform	Java internal	Java Objects

Table 12: Communication Interfaces of the Avatar

REFERENCES

[1] Kruchten, Philippe B. "The 4+ 1 view model of architecture." IEEE software 12.6 (1995): 42-50, available online: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf, last accessed: 17/07/2018

[2] https://developer.android.com/topic/libraries/architecture/room

[3] https://developer.android.com/guide/components/fragments