



Mémoire
Projet de Programmation
Une surface télécommande mains libres partagée
Année 2014/2015

Guillaume Béchade, Raphaël Jorel, Craig Josse, Antoine Laulan

Table des matières

1	Remerciements	4
2	Introduction générale	5
3	Présentation du projet	6
3.1	Cadre	6
3.2	Sujet	6
4	État de l'art	7
4.1	Android	7
4.2	Les capteurs	8
4.3	Télécommande	9
4.4	Algorithmes d'analyse de données	9
5	Analyse des besoins	10
5.1	Remarques sur l'existant	10
5.2	Besoins fonctionnels	10
5.2.1	Communication	11
5.2.2	Capteurs	11
5.2.3	Interface Homme Machine	12
5.3	Besoins non fonctionnels	12
5.3.1	Comportementaux	13
5.3.2	Organisationnels	15
5.4	Cas d'utilisations	16
5.4.1	Initialisation	16
5.4.2	Connexion	16
5.4.3	Capture de séquence	17
5.5	États du système	18
5.5.1	Diagramme d'états	18
5.5.2	Diagramme de séquences	19
5.6	Spécification des besoins	20
6	Conception	21
6.1	Modèle architectural	21
6.2	Détails de conception	21
6.2.1	Diagramme de paquetages	22
6.2.2	Diagramme de classe	24

7 Environnement de travail	30
7.1 Matériel et lieux de travail	30
7.2 Langages de développement	30
7.2.1 Java	30
7.2.2 XML	30
7.3 Technologies utilisées	30
7.3.1 Eclipse	30
7.3.2 Android SDK	31
7.3.3 SVN	31
8 Réalisation	32
8.1 Aides consultées	32
8.2 Fonctionnalités implémentées	32
8.2.1 Connexion entre smartphone et ordinateur	32
8.2.2 Autour des capteurs	32
8.2.3 Analyse des données	33
8.2.4 Rôle du driver	33
8.2.5 Rôle de l'application	34
8.3 Limites du logiciel	34
8.3.1 Méthode d'analyse	34
8.3.2 Non-portabilité	34
8.3.3 Consommation processeur	34
8.4 Améliorations possibles	34
8.4.1 Une meilleure analyse	34
8.4.2 De l'étalonnage	35
8.4.3 Endormir les threads	35
8.4.4 Capteur de proximité	35
9 Tests	36
9.1 Politique	36
9.2 Capture et analyse	36
9.2.1 Capteurs	37
9.2.2 Analyseurs	37
9.3 Connexion	39
9.4 Interface Homme Machine	41
9.5 RGACDriver	43
9.5.1 Server	43
9.5.2 Controller	43
9.5.3 Device	44
9.6 Performances	45
9.6.1 RGACDriver	45
9.6.2 RGACRemote	45
9.7 Robustesse	47
9.7.1 Perturbations lors de la capture	47
9.7.2 Exécution de RGACDriver sous Linux	47
9.7.3 Taux d'erreurs de l'analyse	47
9.7.4 Réseaux Wi-Fi	48
10 Conclusion générale	49

A	Prototypes	50
A.1	Prototypes implémentés	50
A.2	Prototype papier	54
B	Tests	55
B.1	Connexion	55
B.1.1	Résultats	55
B.1.2	Couverture	56
B.2	Analyseurs	57
B.3	Driver	59
B.4	Performance	61
B.4.1	Driver	61
B.4.2	Application RGACRemote	62

Chapitre 1

Remerciements

Nous tenons à remercier l'ensemble des personnes qui nous ont encadrés et conseillés tout au long de notre projet. Notre professeur de cours magistral, M. Philippe Narbel, qui nous a enseigné l'essentiel des notions de génie logiciel nécessaires à la bonne réalisation d'un projet de programmation. Notre chargé de Travaux Dirigés et professeur M. Adrien Boussicault, qui nous a conseillés et qui a répondu à nos questions lors de nos rendez-vous hebdomadaires. Notre client et professeur M. Serge Chaumette, qui nous a proposé ce projet et qui nous a aidés dans certains de nos choix importants.

Chapitre 2

Introduction générale

Notre projet, une surface télécommande mains libres partagée, a été développé dans le cadre de notre unité d'enseignement Projet de Programmation.

Cette unité d'enseignement fait partie intégrante du second semestre de Master Informatique de l'Université de Bordeaux.

Elle a pour but de nous initier aux techniques de génie logiciel.

Nous avons composé un groupe de quatre étudiants afin d'appliquer toutes les étapes importantes du cycle de vie d'un logiciel à un projet choisi dans une liste proposée par nos enseignants.

Chapitre 3

Présentation du projet

3.1 Cadre

Comme introduit précédemment, notre projet a été développé dans le cadre d'une unité d'enseignement (UE). Le principe de cette UE est que chaque sujet de projet est soumis par un ou plusieurs enseignants. Dans notre cas, c'est M. Serge Chaumette qui a soumis ce sujet, il est de ce fait devenu notre client. C'est-à-dire que nous avons simulé une relation entre un client et un prestataire de service, comme il en existe aujourd'hui dans le monde professionnel. Il a exprimé un besoin et nous avons dû produire un logiciel qui satisfait ce besoin. Pour ce faire, nous avons eu des rencontres régulières avec lui afin de vérifier que le logiciel était bien adéquat. En parallèle, nous avons mis en œuvre les notions apprises lors des cours de M. Narbel et pris en compte les conseils et les remarques de M. Boussicault.

3.2 Sujet

Voici la description du sujet tel qu'il nous a été proposé :

L'objectif de ce projet est de réaliser une télécommande mains libres partagée par plusieurs utilisateurs. Le principe du système est le suivant : on pose un mobile *Android* sur une surface adaptée (ex : table en bois). On agit sur la table (on la tapote par exemple), ces actions sont détectées et récupérées par les capteurs du téléphone pour déclencher des actions. On simulera à l'aide de ce système le contrôle d'une télévision en connectant le mobile avec un navigateur Web fonctionnant sur une station de type PC.

On mettra en œuvre une approche modulaire de sorte à pouvoir gérer différents types de téléphones/-capteurs et de pouvoir augmenter le nombre d'actions réalisables.

Le travail sera précédé d'un état de l'art.

Matériel(s) :

1 Téléphone *Android*

Logiciel(s) :

Eclipse et *SDK Android*

Chapitre 4

État de l'art

Dresser un état de l'art dans un domaine consiste à rechercher toutes les informations existantes concernant ce domaine et à en faire une synthèse. Cela est fait par un travail d'étude bibliographique et une analyse des publications formelles ou informelles concernant le domaine étudié. Cette démarche est préliminaire à tout travail de recherche ou d'application et nous a explicitement été demandé par le client.

4.1 Android



FIGURE 4.1 – Logo Android

Android est un système d'exploitation mobile, c'est-à-dire que, tout comme Windows ou OS X, c'est un programme qui gère le matériel sur lequel il s'exécute (smartphone, tablette, ordinateur ou d'autres) de manière à pouvoir exécuter des logiciels. Et avec l'explosion des ventes de smartphones ces dernières années, *Android* a pris une place importante dans la vie quotidienne de millions de personnes, au point qu'il s'agit du système d'exploitation mobile avec le plus d'applications en circulation.

Le système d'*Android* est composé de deux grandes parties :

- le langage *Java*
- le *Software Development Kit* (kit de développement logiciel)

Ce dernier permet d'avoir un environnement de développement facilitant la tâche du développeur. Le kit de développement donne accès à des exemples, à de la documentation mais surtout à l'API de programmation du système et à un émulateur pour tester ses applications. Stratégiquement, Google utilise la licence Apache pour *Android*, ce qui permet la redistribution du code sous forme libre ou non et d'en faire un usage commercial. Le plugin *Android Development Tool* permet d'intégrer les fonctionnalités du SDK à *Eclipse*. Il faut l'installer comme un plugin classique en précisant l'URL du plugin. Ensuite, il faut renseigner l'emplacement du SDK (préalablement téléchargé et décompressé) dans les préférences du plugin ADT [7].

4.2 Les capteurs

La majorité des appareils modernes sont bien plus que de simples outils pour communiquer ou naviguer sur Internet. Ils ont des capacités sensorielles, matérialisées par leurs capteurs. Ces capteurs nous fournissent des informations brutes avec une précision relative, qu'il est possible d'interpréter pour comprendre les transitions d'état que vit le terminal. On trouve, par exemple, des accéléromètres, des gyroscopes, des capteurs de champ magnétique, etc. Tous ces capteurs nous permettent d'explorer de nouvelles voies, d'offrir de nouvelles possibilités aux utilisateurs.

On peut répartir les capteurs en trois catégories :

- Les capteurs de mouvements : en mesurant les forces d'accélération et de rotation sur les trois axes, ces capteurs sont capables de déterminer dans quelle direction se dirige l'appareil. On y trouve l'accéléromètre, les capteurs de gravité, les gyroscopes et les capteurs de vecteurs de rotation,
- Les capteurs de position : évidemment, ils déterminent la position de l'appareil. On trouve ainsi les capteurs d'orientation et le magnétomètre,
- Les capteurs environnementaux : ce sont trois capteurs (baromètre, photomètre et thermomètre) qui mesurent la pression atmosphérique, l'illumination et la température ambiante.

D'un point de vue technique, on trouve deux types de capteurs. Certains sont des composants matériels, c'est-à-dire qu'il y a un composant physique présent sur le terminal. Ils fournissent des données en prenant des mesures. Certains autres capteurs sont uniquement présents d'une manière logicielle. Ils se basent sur des données fournies par des capteurs physiques pour calculer des données nouvelles.

Il n'est pas rare qu'un terminal n'ait pas tous les capteurs, mais seulement une sélection. Par exemple, la grande majorité des appareils ont un accéléromètre ou un magnétomètre, mais peu ont un thermomètre. De plus, il arrive qu'un terminal ait plusieurs exemplaires d'un capteur, mais calibrés d'une manière différente de façon à avoir des résultats différents [4].

DIFFÉRENTS CAPTEURS ANDROID				
TYPE_ACCELEROMETER	3	m/s ²	Mesure de l'accélération (gravité incluse)	[0] axe x [1] axe y [2] axe z
TYPE_GYROSCOPE	3	Rad/s	Mesure la rotation en termes de vitesse autour de chaque axe	[0] vitesse x [0] vitesse y [0] vitesse z
TYPE_LIGHT	1	Lux	Mesure la luminosité	[0] valeur
TYPE_MAGNETIC_FIELD	3	μ Tesla	Mesure du champ magnétique	[0] axe x [1] axe y [2] axe z
TYPE_ORIENTATION	3	degrés	Mesure de l'angle entre le nord magnétique	[0] Azimut y/nord [1] Rotation x (-180,180) [2] Rotation y (-90,90)
TYPE_PRESSURE	1	KPas	Mesure la pression	[0] valeur
TYPE_PROXIMITY	1	mètre	Mesure la distance entre l'appareil et un objet cible	[0] valeur
TYPE_TEMPERATURE	1	Celsius	Mesure la température	[0] valeur

FIGURE 4.2 – Types de capteurs

Le livre *Professional Android sensor programming* introduit l'utilisation des capteurs avec le système d'exploitation *Android* [8].

4.3 Télécommande

Il existe de nombreuses applications pour smartphone simulant une télécommande afin de piloter une télévision. Il en existe pour chaque type de système d'exploitation mobile. En voici quelques exemples.

Dans l'article *5 Apps to Turn Your Phone into a Universal Remote* nous pouvons observer plusieurs applications qui réalisent la fonction de télécommande mains libres via divers moyen de connexions. Que ce soit avec Bluetooth ou Wi-Fi, toutes ces applications assurent le service de télécommande à l'aide d'une interface présente sur l'écran qui permet de commander l'envoi d'informations au dispositif récepteur. [11] À la différence d'une télécommande classique, notre télécommande utilisera ses différents capteurs afin de transmettre ses "ordres" à la station PC. Et de plus, celle-ci sera utilisée pour piloter un logiciel type "lecteur multimédia" et non une télévision.

4.4 Algorithmes d'analyse de données

Les capteurs des smartphones peuvent être utilisés pour plusieurs services. Ils enregistrent les variations de l'environnement du smartphone. Ensuite, en fonction de l'analyse que l'on fait de ces données, on peut dégager différents comportements. Voici quelques exemples d'analyses de données des capteurs.

Le document *Step Counting Using Smartphone-Based Accelerometer* décrit comment utiliser l'accéléromètre pour transformer l'accéléromètre en podomètre. C'est-à-dire en analysant les mouvements captés par l'accéléromètre et en déduire ainsi les pas faits par l'utilisateur. Cette analyse est réalisée via une relation entre la fréquence des pas qui varie inversement avec la vitesse du mouvement et l'amplitude du signal renvoyé par l'accéléromètre [9].

La publication *Activity recognition using k-nearest neighbor algorithm on smartphone with tri-axial accelerometer* met en avant l'algorithme des *k* plus proches voisins dans la détection d'activités physiques via les capteurs d'un smartphone [10].

Chapitre 5

Analyse des besoins

Cette partie vous présentera les besoins fonctionnels et non-fonctionnels de notre logiciel. Cela constitue le cahier des charges de notre projet. Il a été réalisé en accord avec notre client.

Besoins fonctionnels : Les besoins fonctionnels précisent les fonctionnalités qui seront offertes par le logiciel, qui seront nécessaires au logiciel, et donc ce qu'il doit faire, les services qu'il doit rendre, son comportement effectif. Plus une décomposition en besoins fonctionnels est précise, plus elle se rapproche de la description du logiciel à développer.

Besoins non fonctionnels : Les besoins non fonctionnels précisent les qualités globales que l'on attend du logiciel. Les besoins non fonctionnels sont souvent les plus critiques pour un logiciel et pour un projet donné, au vu du fait que certains types de besoins non fonctionnels ne sont pas nécessairement à exprimer.

5.1 Remarques sur l'existant

Il existe déjà des applications permettant de simuler une télécommande à l'aide de son smartphone. Comme il existe déjà des analyses de données capturées par les capteurs de smartphones. Cependant, il ne semble pas exister d'applications qui mélange les deux.

Nous ne pouvons donc pas nous appuyer sur des travaux semblables pour notre projet. Nous pouvons néanmoins utiliser des informations sur des projets s'en rapprochant afin de nous aider.

Une, si ce n'est la plus grande, difficulté de notre projet est liée à l'analyse des données fournies par les capteurs. Les exemples cités ci-dessus montrent la complexité de ce domaine. Après concertation avec notre client M. Serge Chaumette, nous sommes convenus de produire une analyse plus simple.

5.2 Besoins fonctionnels

L'utilisateur communiquera avec un dispositif ayant des capteurs (type tablette ou smartphone) que nous nommerons terminal dans cette partie. Ensuite, le terminal communiquera avec un ordinateur (toute machine possédant une unité de calcul). Nous verrons de quelles natures seront les communications par la suite. Dans un premier temps, nous parlerons de communications, puis dans un second temps, nous nous attarderons sur les capteurs et enfin sur l'Interface Homme Machine, i.e. le moyen qu'aura l'utilisateur pour contrôler le terminal.

Nous voyons ici que deux services sont nécessaires, un premier au niveau du terminal qui s'occupera de la partie capture, analyse et interface, et un second au niveau de l'ordinateur qui attendra les

informations du terminal. Le premier sera parfois également désigné par "application" et le dernier par "driver" (pilote).

5.2.1 Communication

Une communication est définie comme un échange de messages entre deux objets (ordinateur, smartphone) ou humains. Un message est généralement un support pour le transport d'une ou plusieurs informations. Ici, nous aurons besoin d'une communication entre l'application et le driver.

Différents types

La communication entre l'utilisateur et le terminal est de type physique. Ce dernier agira sur l'environnement du terminal. La communication entre le terminal et l'ordinateur est de type réseau. Une communication réseau est définie comme une communication entre deux ordinateurs. Ce genre de communication nécessite plusieurs étapes.

Ouverture de la communication réseau

Afin de permettre l'échange de messages entre deux ordinateurs, il faut d'abord l'ouvrir, i.e. l'initier. Celui qui souhaite initier la communication envoie à l'autre un message d'initiation de communication, et l'autre lui répond. La communication est établie.

Échange d'information

Ensuite, les deux ordinateurs échangent des messages et ceux-ci transportent des informations propres au contexte dans lequel la communication est établie.

Gestion des erreurs

Des erreurs d'envois, ou des messages erronés, peuvent arriver. Dans ces cas-là, des techniques (voir le protocole TCP pour plus d'informations) permettent de corriger, de détecter et de demander de renvoyer les messages.

Fermeture

Quand l'un des deux ordinateurs souhaite arrêter la communication, il envoie un message à l'autre pour l'en informer.

5.2.2 Capteurs

Un capteur est un appareil qui permet de mesurer une quantité physique (lumière, son, gravité, etc). Dans le cadre de ce projet, notre terminal présente plusieurs capteurs, dont nous allons nous servir. L'utilisation de ces capteurs sera divisée en trois parties : l'écoute de l'environnement, l'analyse des mesures faites, et enfin l'échange entre le terminal et l'ordinateur. Ces différentes parties sont détaillées ci-dessous.

Écoute de l'environnement

L'environnement est le milieu physique, évoluant au cours du temps, dans lequel se trouve le terminal. Ces évolutions ont trait aux quantités que les capteurs mesurent, ainsi, il nous faut les récupérer pour être informés de ces évolutions. Cette action correspond à l'écoute des capteurs.

Nous nous servirons en particulier de l'accéléromètre et du microphone, le premier mesurant l'accélération linéaire (le mouvement du dispositif, quantifié sur les trois coordonnées de l'espace : x, y et z) et le second, les caractéristiques sonores de l'environnement (le bruit effectué autour de lui, par exemple une frappe dans les mains).

Analyse

L'analyse consiste à détecter des séquences de variations de l'environnement. Certaines pourront correspondre à des séquences que nous aurons prédefinies. Dans le cas où une séquence appartient à l'ensemble de celles reconnues, le terminal signale quelle séquence il a reconnu à l'aide d'identifiants uniques associés aux séquences. Dans le cas contraire, la séquence sera ignorée et le dispositif sera en attente de nouvelles modifications de son environnement.

Échange

Après avoir reconnu une séquence, le terminal transmet au driver l'identifiant unique de la séquence reconnue, via la liaison établie comme décrite précédemment. Le driver utilise cet identifiant pour exécuter une action qui lui est associé.

5.2.3 Interface Homme Machine

L'utilisateur est au centre du service (interaction avec le terminal), c'est-à-dire que c'est lui qui interagit avec l'environnement dont les quantités physiques sont mesurées par les capteurs. C'est donc lui qui est à l'origine des échanges effectués entre le terminal et l'ordinateur. L'interface Homme Machine (IHM) est la partie visible de l'application, c'est ce que voit l'utilisateur sur l'écran du terminal. Il peut ainsi être guidé par l'IHM concernant la marche à suivre, les interactions qu'attend l'application par exemple.

Démarrage

L'IHM doit permettre à l'utilisateur de démarrer l'application. Après l'avoir fait, c'est lui qui initie la liaison entre le terminal et l'ordinateur. Pour ce faire, l'utilisateur doit saisir une information à l'application que le driver lui fournit pour établir une communication avec ce dernier. L'IHM doit informer l'utilisateur si la liaison est un succès ou, au contraire un échec. Dans le cas d'un échec, l'utilisateur peut faire une nouvelle tentative.

Interactions

Après être entré en communication avec le driver, l'application rentre dans un état dans lequel l'utilisateur peut agir sur l'environnement du terminal. Il doit réaliser l'une des séquences prédefinies par le driver pour que l'application réagisse.

Arrêt

À l'instar du démarrage, l'utilisateur doit avoir l'opportunité de terminer une connexion ou de quitter l'application via l'utilisation de l'IHM.

5.3 Besoins non fonctionnels

Parlons maintenant des besoins ayant attrait aux qualités globales du service. Chaque besoin et sous besoin devra être éprouvé par des tests pour vérifier qu'ils répondent à nos attentes.

5.3.1 Comportementaux

Système d'exploitation Android

L'application doit fonctionner sur le système d'exploitation (SE) *Android* (Système d'exploitation pour appareils mobiles).

Dispositifs Android Différents terminaux disposant du SE *Android* doivent être capable de la faire fonctionner.

Tests : Tester le bon fonctionnement sur, au minimum, deux dispositifs *Android*. L'application doit avoir le même comportement sur les terminaux, c'est-à-dire détecter les séquences dans le même ordre. Pour s'assurer de ce bon fonctionnement, il faudra donc se munir de plusieurs terminaux et exécuter les même séquences pour vérifier la cohérence des réponses données.

Versions d'Android L'utilisation de l'application doit être possible sur plusieurs versions du SE *Android*.

Tests : Tester le bon fonctionnement sur, au minimum, deux versions récentes du SE *Android*. De la même manière que précédemment, il faudra avoir des terminaux mais cette fois avec des versions du SE *Android* différentes. Les terminaux devront être les mêmes au niveau matériel.

Utilisation simultanée de plusieurs capteurs

L'usage de plusieurs capteurs en simultané doit permettre d'augmenter le nombre et la précision des interactions possibles avec l'application.

Tests : Tester que l'action sur un capteur ne produise pas d'effets non désirés sur un autre type de capteur. Pour ce faire, il sera nécessaire de repérer les effets indésirés et ainsi amener l'application dans l'état où elle pourrait se tromper, en faisant une séquence particulière.

Vélocité

Une télécommande quelle qu'elle soit, se doit de réagir vite, si bien que l'utilisateur a l'impression qu'elle fonctionne instantanément. C'est pourquoi l'application doit analyser et traiter les données en un temps acceptable. Entre le moment où l'utilisateur interagit et le moment où l'action est effectuée sur l'ordinateur, il doit s'écouler un temps inférieur à une ou deux secondes. Le total du temps de chaque partie listée ci-dessous, excepté le temps nécessaire à la réalisation d'une séquence, ne doit pas excéder cette durée.

Création d'une séquence : Comme décrit précédemment, une séquence est une suite de mouvements de l'utilisateur. Il est nécessaire pour une utilisation plus agréable de l'application, que le temps séparant deux mouvements d'une séquence soit court voire très court. Cela permet en cas d'échec de pouvoir recommencer plus vite mais aussi de diminuer le risque d'interférences entre deux mouvements.

Tests : Calculer le temps moyen d'une séquence en fonction de son nombre de mouvements. Ceci peut être fait simplement en mesurant le temps entre le début de la séquence et sa fin, puis en divisant ce temps par le nombre de mouvements de la séquence. Il faudra évidemment faire cela sur plusieurs séquences différentes, une bonne centaine de fois, pour avoir des résultats assez fiables.

Envoi de l'information : Si une séquence est reconnue, son identifiant unique est transmis au driver. Il serait souhaitable que cet envoi se fasse encore une fois dans un temps relativement court (moins d'une demi-seconde).

Tests : Calcul du temps entre l'envoi de l'identifiant par le terminal et la réception par l'ordinateur. Ce calcul sera compliqué d'un point de vue logiciel, au vu du fait que ce sont deux services différents qui communiqueront, sur deux appareils à distance. Une estimation du temps pourra être faite "à la main" pour juger de l'acceptabilité du temps de transfert.

5.3.2 Organisationnels

Ci-dessous voici le diagramme de Gantt de notre projet. L'organisation de chacune des phases du logiciel dans le temps.

Besoins/ semaines	1 05/01	2 12/01	3 19/01	4 26/01	5 2/02	6 9/02	7 16/02	8 23/02	9 2/03	10 9/03	11 16/03	12 23/03	13 29/03	14 6/04	15 13/04
Prise de contact															
Définition du sujet avec le client															
Bibliographie															
Prototypes															
Rédaction cahier des charges															
Planification des tâches															
Scénario test contraintes															
Architecture															
Implémentation des tests															
Implémentation fonctionnalités de l'application															
Rédaction de la doc															
Rédaction mémoire															
Rendu mémoire + code															
Préparation Soutenance															
Soutenance															

FIGURE 5.1 – Diagramme de Gantt

Gestionnaire de version

Afin de bien coordonner le travail, et veiller au bon développement de chaque phase du logiciel nous avons utilisé un gestionnaire de version.

5.4 Cas d'utilisations

Nous montrerons dans cette partie quelques cas d'utilisations de notre logiciel à l'aide de diagramme de cas d'utilisations. Seront détaillés : l'initialisation du système, la connexion entre l'ordinateur et le terminal ainsi que la capture d'une séquence par le terminal.

5.4.1 Initialisation

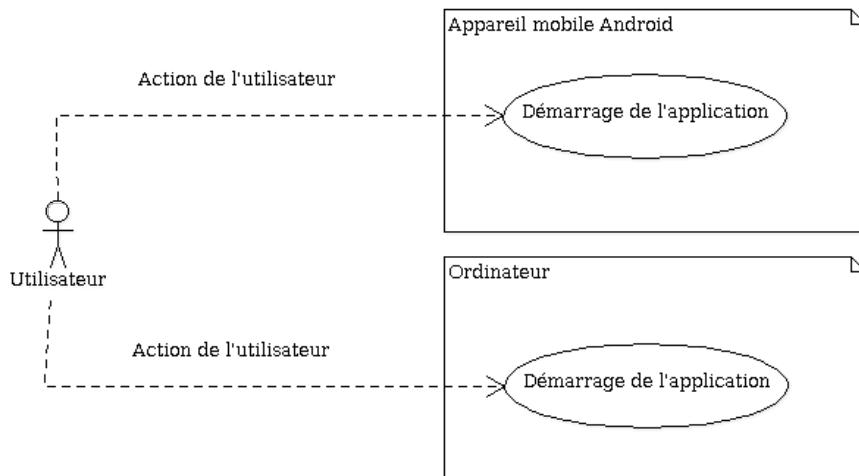


FIGURE 5.2 – Initialisation du système

Description : Ce diagramme représente l'initialisation de notre logiciel par l'utilisateur. Celui-ci doit agir sur l'appareil *Android* ainsi que sur l'ordinateur pour commencer.

5.4.2 Connexion

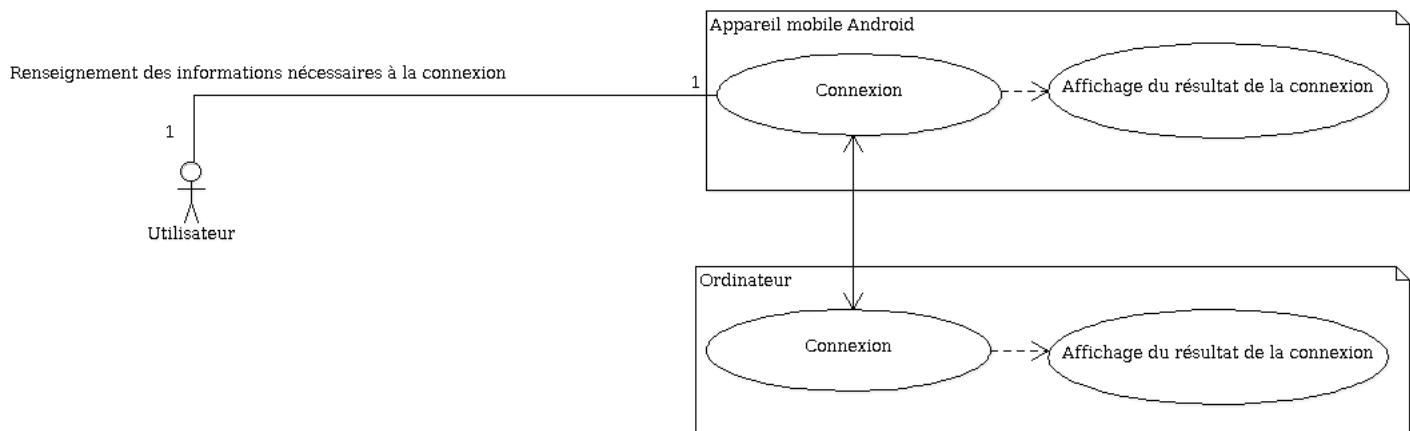


FIGURE 5.3 – Connexion

Description : Ce diagramme représente la connexion entre l'ordinateur et l'appareil *Android*. L'utilisateur doit renseigner des informations sur l'appareil afin que celui-ci puisse se connecter à l'ordinateur.

5.4.3 Capture de séquence

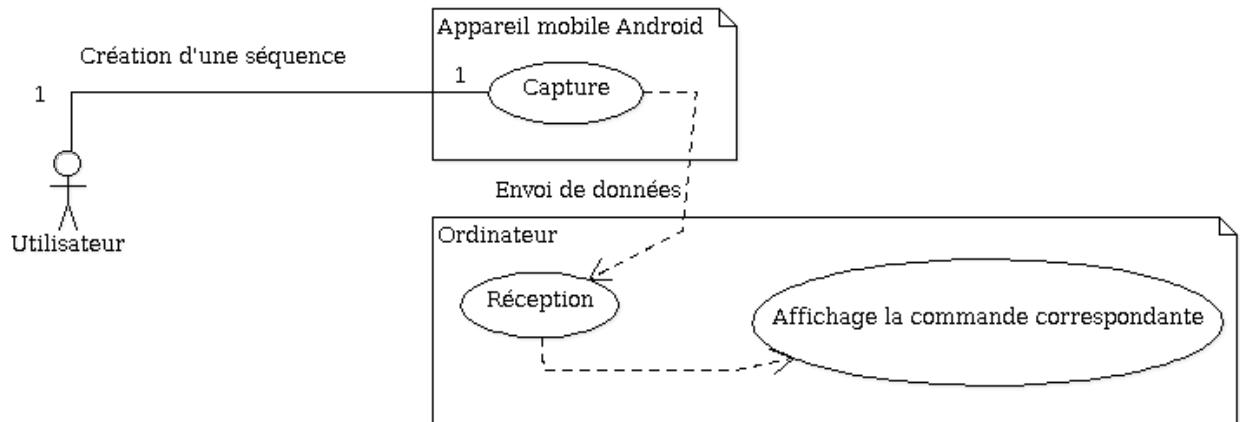


FIGURE 5.4 – Capture de séquence reconnue

Description : Ce diagramme représente le fonctionnement du système lorsque l'appareil capture une séquence reconnue par l'appareil mobile *Android*. Celui-ci transmet alors des informations à l'ordinateur via la connexion réseau.

5.5 États du système

5.5.1 Diagramme d'états

Définition : Le diagramme d'états décrit le comportement dynamique des objets dans le temps. On y retrouve les besoins fonctionnels clairement identifiés.

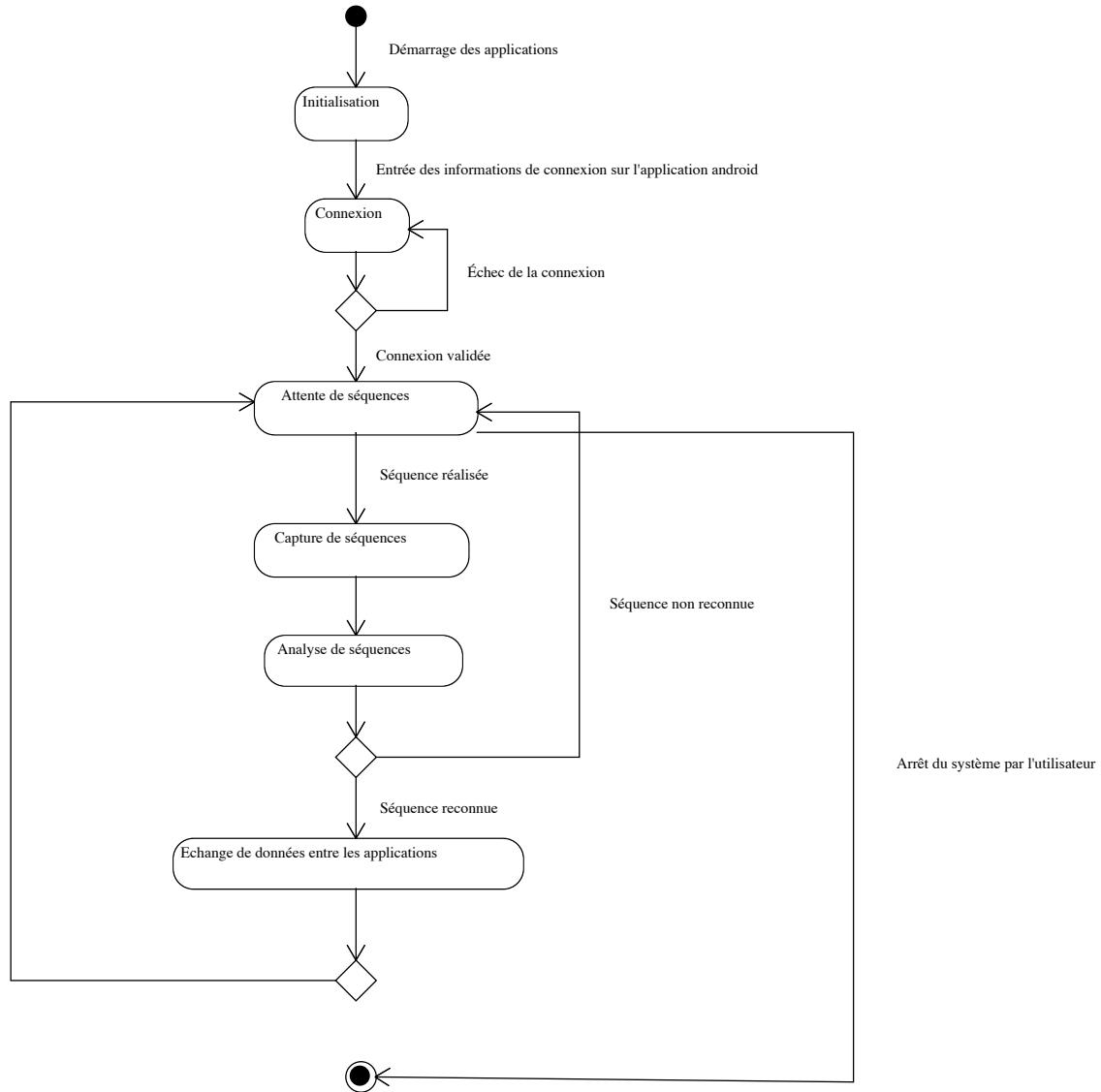


FIGURE 5.5 – Diagramme d'états

Description : Ce diagramme représente les différents états par lequel passe le logiciel au cours d'un cycle de vie. Il représente également les relations qui existent entre ces états.

5.5.2 Diagramme de séquences

Définition : Les diagrammes de séquences sont la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique. Le temps y est représenté explicitement par une dimension (la dimension verticale) et s'écoule de haut en bas.

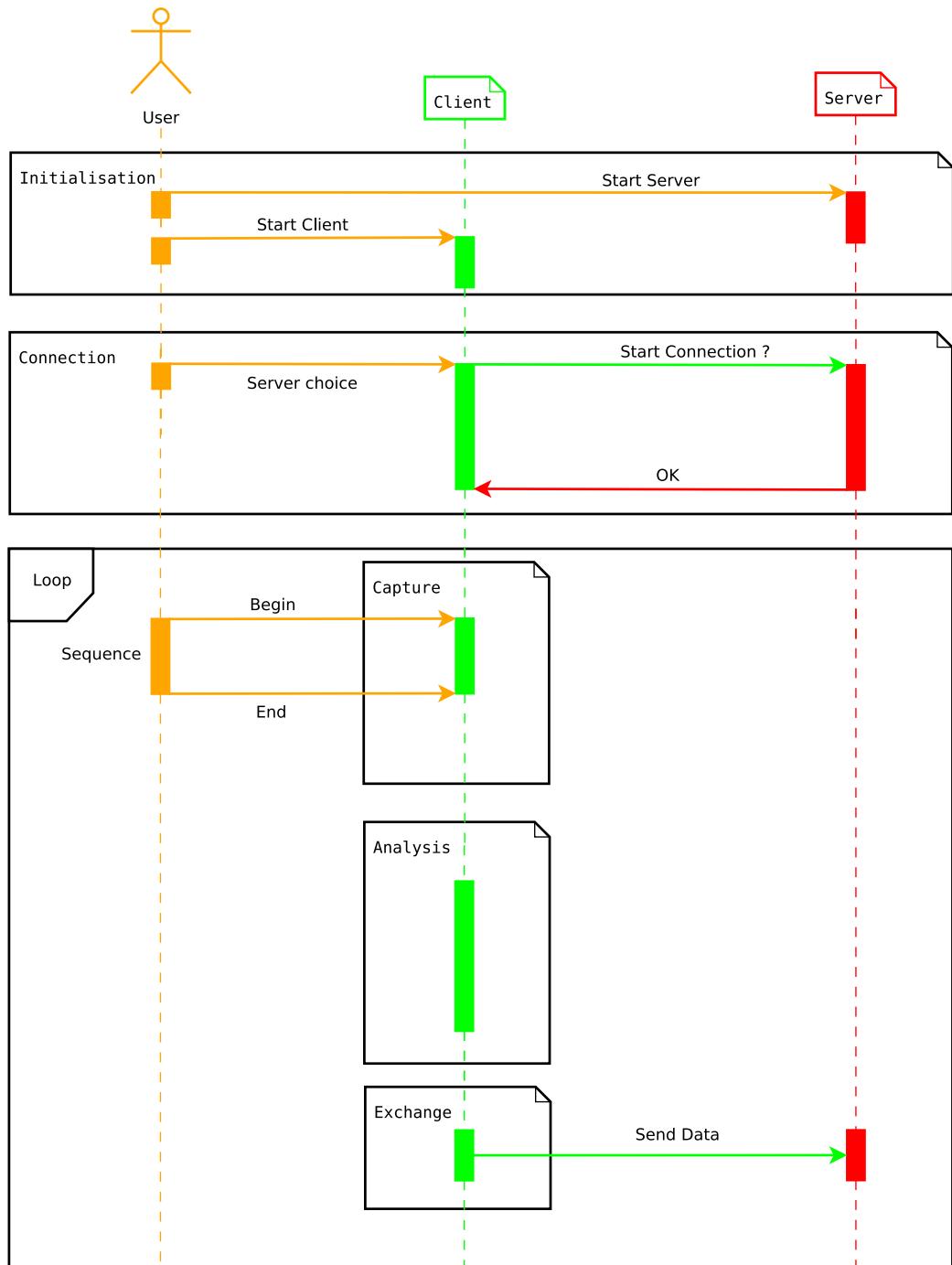


FIGURE 5.6 – Capture de séquence reconnue

Description : Le diagramme de séquence ci-dessus représente le déroulement classique de l'utilisation de notre logiciel. Nous entendons par classique le fait qu'il n'y ait pas de problèmes lors de l'usage de notre logiciel par l'utilisateur. "Pas de problèmes" signifie quant à lui que l'utilisateur ne rencontre aucune erreur de connexion et que celui-ci n'exécute que des mouvements reconnus par notre logiciel. Dans un premier temps, l'utilisateur démarre l'application et le driver, c'est la partie **Initialisation**. Dans un second temps, l'utilisateur agit sur l'application pour que celui-ci se connecte au driver, c'est la partie **Connection**. Dans ce diagramme, la connexion est un succès.

Et dans un dernier temps, l'utilisateur exécute des séquences de mouvements, c'est la partie **Loop**. Ces séquences sont ensuite capturées et analysées par l'application. À la suite de l'analyse, l'application transmet au driver les données correspondant aux séquences qu'il a reconnues. Le loop signifie que l'enchaînement Capture / Analysis / Exchange peut être exécuté plusieurs fois.

5.6 Spécification des besoins

Afin de nous aider dans la spécification des besoins de notre logiciel, nous avons développé plusieurs prototypes. Cela nous a permis d'identifier plusieurs fonctionnalités dont pourrait avoir besoin l'utilisateur mais aussi les problèmes que l'on pourrait rencontrer au cours de notre développement. Nous avons imaginé une utilisation "classique" de notre application afin d'essayer de ne rien oublier. Pour ce faire, nous avons créé plusieurs prototypes implémentés sous formes de petits logiciels. Ainsi qu'un prototype papier afin d'imaginer à quoi pourrait ressembler notre Interface Homme Machine. Pour plus de détails sur les prototypes, vous pouvez consulter les annexes de ce document.

Chapitre 6

Conception

Notre logiciel sera composé de deux parties. La première, nommée application, sera située sur le smartphone et sera en charge de la capture et de l'analyse des données fournies par les capteurs. Tandis que l'autre, nommée driver, sera située sur un ordinateur et sera en charge d'effectuer des actions sur une application en fonction des résultats de l'analyse faite par le smartphone.

6.1 Modèle architectural

Afin de faire communiquer nos deux parties, nous avons choisi d'utiliser le patron architectural (Pattern Architecture) client-serveur. L'environnement client-serveur désigne un mode de communication à travers un réseau entre plusieurs applications : l'un, qualifié de client, envoie des requêtes ; l'autre, qualifié de serveur, attend les requêtes du client et y répond. Dans notre cas, le client correspond à l'application sur le smartphone et le serveur correspond au driver sur l'ordinateur.

Interface homme machine : Il nous a été spécifié dans le sujet de notre projet que nous devions créer une application *Android* pour le smartphone. Cela implique la création d'une Interface Homme Machine *Android* avec toutes les conventions d'implémentations qui lui sont associées.

6.2 Détails de conception

Dans cette partie, nous vous présenterons les détails de conception de notre logiciel. Nous commencerons par présenter une vue globale de notre architecture. Ceci sera fait à l'aide d'un diagramme de paquetages. Ensuite, nous continuerons par plusieurs diagrammes de classes. Le premier présentera une vue un peu plus précise de notre architecture tandis que les suivants présenteront une vue encore plus précise sur des parties spécifiques de l'architecture. Dans cette partie, le driver pourra être appelé ordinateur tandis que l'application pourra être appelée smartphone.

6.2.1 Diagramme de paquetages

Définition :

Les diagrammes de paquetages représentent les graphes d'utilisation d'un ensemble de composants de niveau supérieur, et sont constitués de composants (paquetages) et de liens entre les composants. Le diagramme offre une vue d'ensemble de l'architecture.

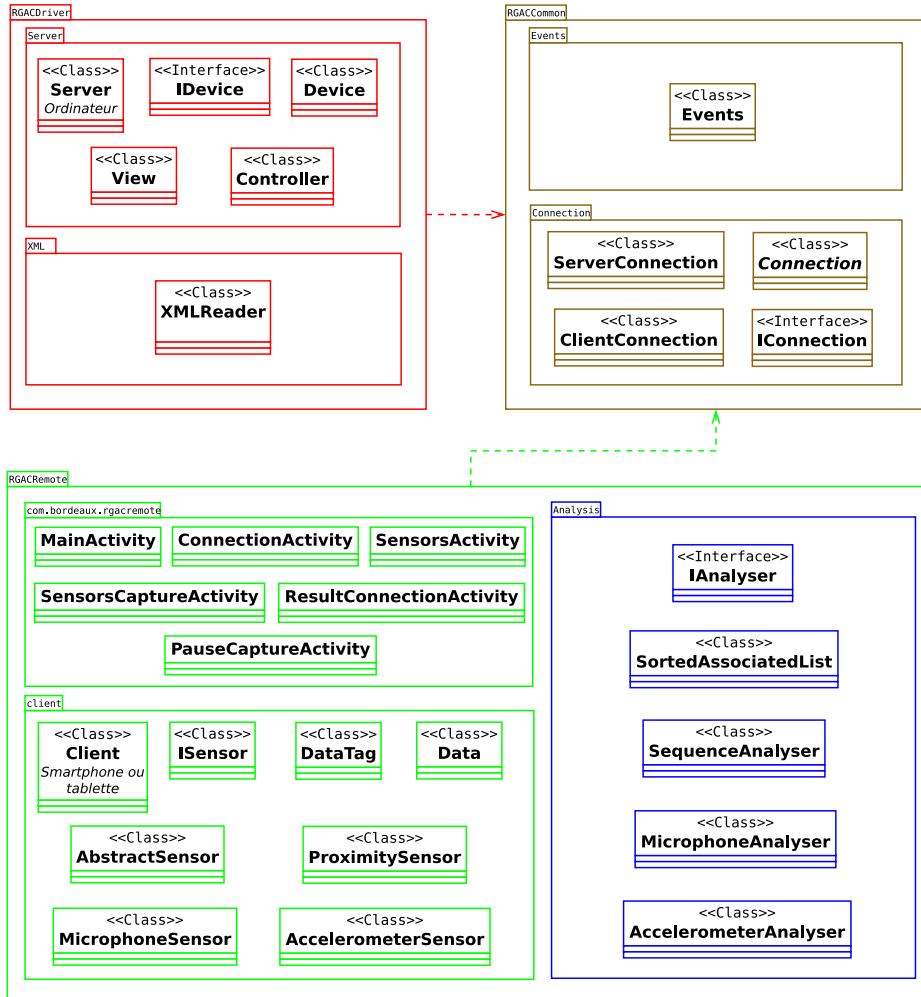


FIGURE 6.1 – Diagramme de paquetages

Description :

L'ensemble de nos classes sera regroupé dans trois gros paquetages. Ils seront eux mêmes composés de plusieurs paquetages. Voici nos trois gros paquetages :

- le paquetage **RGACCommon** : il réunit l'ensemble des classes et paquetages qui ont un rapport avec la connexion entre l'ordinateur et le smartphone, ainsi que le langage qu'ils utilisent pour communiquer,
- le paquetage **RGACDriver** : il réunit l'ensemble des classes qui sont associées à l'ordinateur,
- le paquetage **RGACRemote** : il réunit l'ensemble des classes et paquetages qui sont associés au smartphone.

Le paquetage RGACCommon est composé de deux paquetages :

- le paquetage **Connection** : il rassemble les classes nécessaires à la connexion entre le driver de l'ordinateur et l'application du smartphone. C'est lui qui fera le lien entre le paquetage *Server* et le paquetage *Client*. Et c'est lui qui va permettre l'échange de données entre les deux,
- le paquetage **Events** : il rassemble un ensemble de variables statiques afin d'établir un langage commun entre le client et le serveur.

Le paquetage RGACDriver est composé de deux paquetages :

- le paquetage **Server** : il est composé de l'ensemble des classes permettant de réaliser la connexion serveur. Il contient également toute la partie de gestion du logiciel piloté (*Device*),
- le paquetage **XML** : il est composé de la classe réalisant la lecture des fichiers de configuration en langage XML.

Le paquetage RGACRemote est composé de trois paquetages :

- le paquetage **Client** : il regroupe les classes propres à l'application se situant sur le smartphone. C'est cette même application qui sera à l'origine de la capture des séquences de l'utilisateur,
- le paquetage **Analysis** : il regroupe les classes nécessaires à l'analyse des séquences capturées par le *Client*. Ce paquetage sera en lien direct avec celui du *Client*,
- le paquetage **com.bordeaux.rgacremote** : il regroupe les classes et les fichiers générés automatiquement par *Android* pour constituer l'IHM de l'application du smartphone.

6.2.2 Diagramme de classe

Introduction :

Le diagramme de classe représente les classes intervenant dans le système. Le diagramme de classe est une représentation statique des éléments qui composent un système et de leurs relations (utilisation / appartenance / implémentation / héritage). Nous présenterons, dans un premier temps, un diagramme de classe général présentant une vue un peu plus détaillée de notre architecture par rapport à celle du diagramme de paquetages. Ensuite, nous présenterons des diagrammes de classes plus précis avec des détails sur les données membres et sur les méthodes de nos classes. Ces diagrammes plus précis seront accompagnés de descriptions.

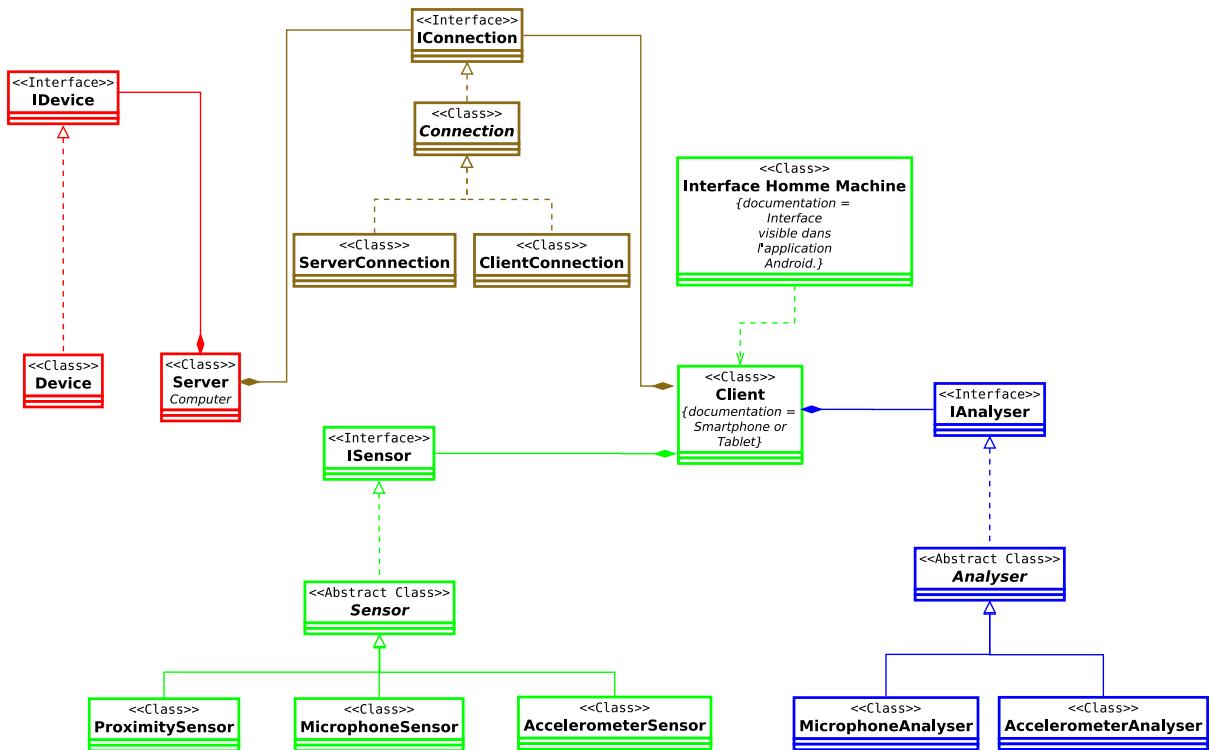


FIGURE 6.2 – Diagramme de classe général

Description : Ce diagramme de classe représente l'architecture logicielle de notre projet. Les classes, interfaces, et classes abstraites sont différencierées par un code couleur. Ce code couleur est associé à des paquetages différents. Chaque diagramme sera accompagné d'une description générale ainsi que la liste des classes qui le compose. Une courte description sera donnée pour chaque classe de la liste.

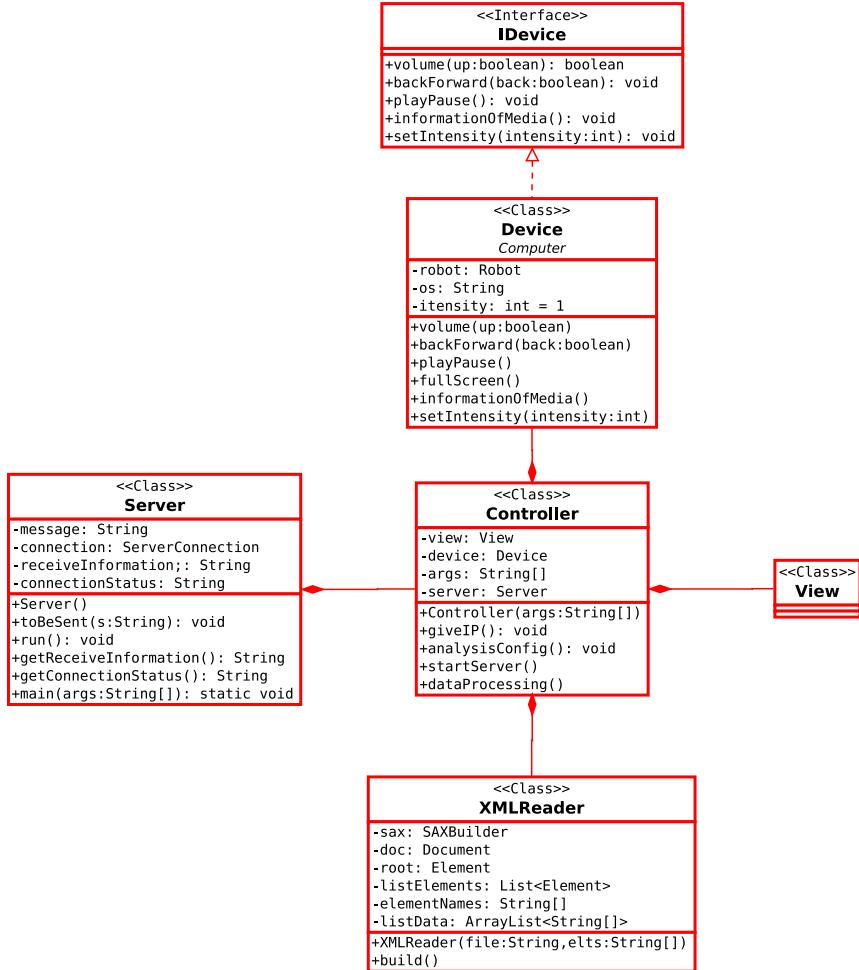


FIGURE 6.3 – Diagramme de classe RGACDriver

Description : Les classes présentes dans le paquetage *RGACDriver* permettent la gestion du driver, cela consiste à piloter le logiciel cible (*Device*) via les ordres envoyés par le smartphone. Ce paquetage a deux objectifs, gérer la communication du serveur et gérer la lecture et l'interprétation des fichiers de configuration pour envoyer des ordres appropriés au logiciel cible.

Nous décrirons en détails le contenu de chaque classe dans la liste suivante :

- l'interface **IDevice** : cette interface spécifie les différentes commandes qui pourront être demandées au logiciel cible,
- la classe **Device** : cette classe réalise les actions spécifiées par l'interface *IDevice*,
- la classe **Controller** : elle gère la coopération entre les différents composants du paquetage. Le contrôleur dirige le serveur, le traitement des informations via les fichiers de configurations et l'affichage des résultats,
- la classe **Server** : elle entretient la communication avec le smartphone via l'objet *ServerConnection*,
- la classe **View** : affiche les diverses informations sur la communication et l'adresse IP du serveur,
- la classe **XMLReader** : elle permet de lire les fichiers de configurations XML.

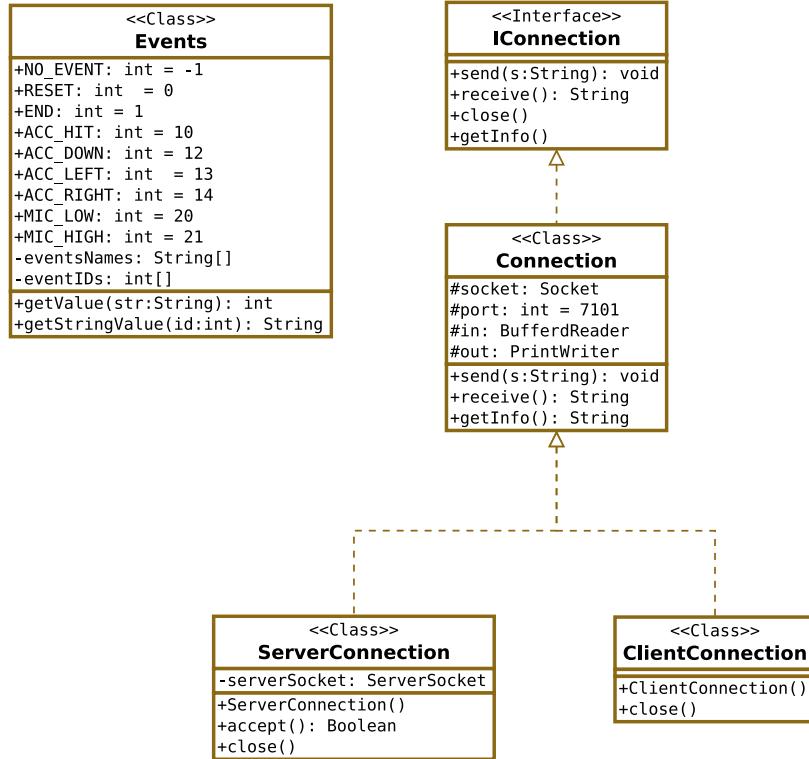


FIGURE 6.4 – Diagramme de classe RGACCommon

Description : Le paquetage *RGACCommon* gère la connexion entre la partie driver présente sur l'ordinateur et la partie application sur le smartphone.

Nous décrirons en détails le contenu de chaque classe dans la liste suivante :

- l'interface **IConnection** : l'interface *IConnection* permet de spécifier les méthodes communes à tous les éléments de la connexion. Les éléments client et serveur offrent des services d'envoi de données, méthode *send()*, ainsi que leurs réceptions, méthode *receive()*. Ils ont aussi la possibilité de clore la connexion avec la méthode *close()*. Nous avons également rajouté une méthode *getInfo()* permettant de connaître l'état de la connexion et l'adresse IP du serveur hôte,
- la classe abstraite **Connection** : cette classe abstraite implémente les parties communes au deux classes *ServerConnection* et *ClientConnection*. c'est à dire les méthodes *send()*, *receive()* et *getInfo()*,
- la classe **ClientConnection** : elle gère la connexion côté client,
- la classe **ServerConnection** : elle gère la connexion côté serveur.

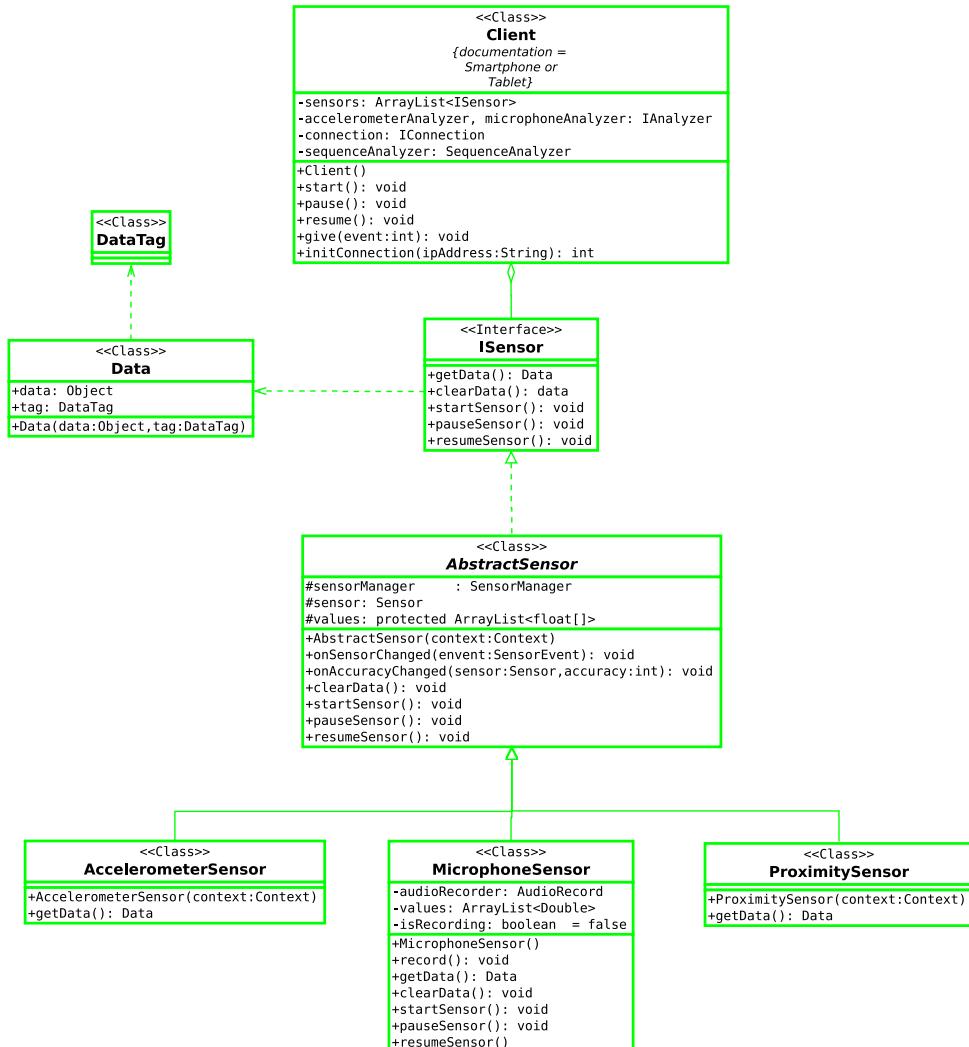


FIGURE 6.5 – Diagramme de classe Client

Description : Le paquetage *Client* contient les classes nécessaires à la capture des données par les capteurs du smartphones.

Nous décrirons en détails le contenu de chaque classe dans la liste suivante :

- la classe **Client** : cette classe gère la connexion client ainsi que la capture des données en provenance des capteurs,
- l’interface **ISensor** : détermine les fonctions à implémenter dans les différentes classes de gestion des capteurs,
- la classe abstraite **AbstractSensor** : factorise le code partagé entre les différents capteurs,
- la classe **MicrophoneSensor** : gère l’acquisition des données du microphone,
- la classe **AccelerometerSensor** : gère l’acquisition des données de l’accéléromètre,
- la classe **ProximitySensor** : gère l’acquisition des données du capteur de proximité,
- la classe **DataTag** : spécifie les différents types de données retournés par les capteurs,
- la classe **Data** : lie les types de données retournés par les capteurs avec l’objet réel.

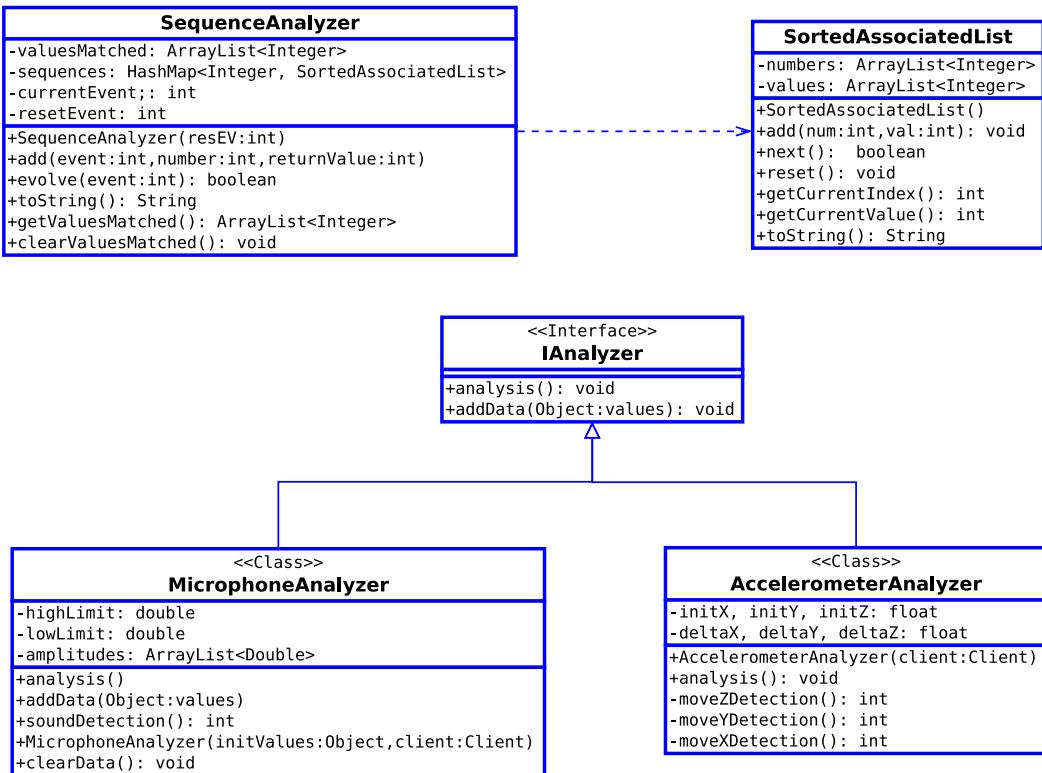


FIGURE 6.6 – Diagramme de classe Analysis

Description : Le paquetage *Analysis* contient les classes responsables de l'analyse des données renvoyées par les capteurs du smartphone. Ce sont elles qui sont en charge de parcourir les tableaux où sont stockées ces mêmes données.

Nous décrirons en détails le contenu de chaque classe dans la liste suivante :

- l'interface **IAnalyzer** : spécifie les fonctions que chaque classe d'analyser devra implémenter,
- la classe **MicrophoneAnalyzer** : réalise l'analyse des données capturées par le microphone,
- la classe **AccelerometerAnalyzer** : réalise l'analyse des données capturées par l'accéléromètre,
- la classe **SequenceAnalyzer** : gère la progression des séquences de mouvements,
- la classe **SortedAssociatedList** : range par ordre croissant des couples de valeurs suivant la première du couple.

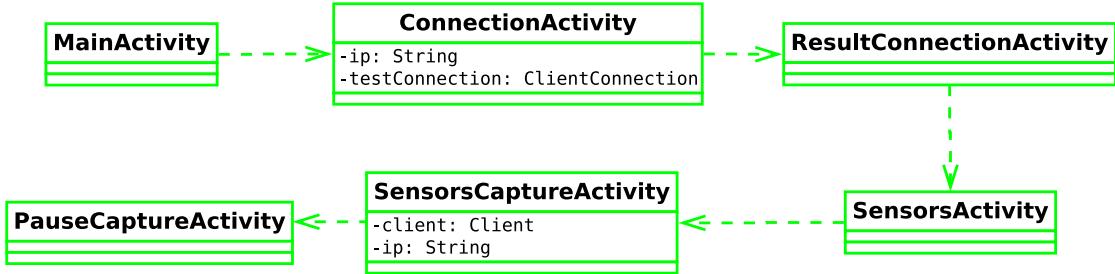


FIGURE 6.7 – Diagramme de classe IHM

Description : L'interface homme machine (IHM) de l'application située sur le smartphone est représentée par ce diagramme de classe. Nous avons fait le choix de ne montrer que les classes qui sont importantes. C'est-à-dire, que les classes responsables de l'affichage de diverses aides pour l'utilisateur n'apparaissent pas dans ce diagramme. Notons que, en *Android*, une classe est appelée "Activité". Il n'y pas de détails quant aux méthodes des activités car les standards *Android* nous imposent des noms de méthodes prédéfinis. Pour les IHM *Android*, il est conseillé de créer une nouvelle activité lorsque l'affichage graphique change de façon significative. C'est pourquoi, il y a plusieurs activités et non pas une seule. De plus, chaque activité *Java* est associée à un fichier *XML* responsable de l'affichage des éléments graphiques de l'activité. Typiquement la taille, la couleur et la position dans l'écran du smartphone. La relation définie par les flèches entre les activités est la relation d'appel / création. C'est-à-dire que la première activité *MainActivity* crée la seconde, *ConnectionActivity*, et ainsi de suite. Nous décrirons en détails le contenu de chaque activité dans la liste suivante :

- la classe **MainActivity** : c'est la première classe qui est créée lorsque l'application sur le smartphone est lancée. Elle contient quelques détails sur la marche à suivre ainsi qu'un bouton permettant de passer à l'activité suivante,
- la classe **ConnectionActivity** : elle est responsable de la connexion entre le smartphone et l'ordinateur. L'utilisateur doit entrer l'adresse IP de l'ordinateur afin de pouvoir initier une connexion. Cependant, cette activité se charge simplement de tester la connexion avec l'IP fournie. Elle coupe la connexion une fois que le test est positif. Si la connexion est réussie, cette activité se chargera de transmettre à la suivante l'adresse IP fournie par l'utilisateur,
- la classe **ResultConnectionActivity** : elle est responsable de l'affichage du résultat de la connexion. Il y a deux cas possibles. Le premier, la connexion est réussie et donc l'activité affiche le succès et passe à l'activité suivante, à savoir *SensorsActivity*. Le second, la connexion a échoué et donc l'activité affiche l'échec et devient alors "bloquante". C'est-à-dire qu'elle ne permet pas le passage à l'activité *SensorsActivity* mais oblige un retour à l'activité *ConnectionActivity* afin que l'utilisateur essaye une nouvelle adresse IP.
- la classe **SensorsActivity** : elle propose une liste de boutons. Le premier lance la capture des données par les capteurs. Le second est une liste des commandes à appliquer au smartphone, et le troisième affiche des informations sur les capteurs,
- la classe **SensorsCaptureActivity** : elle apparaît lorsque l'utilisateur souhaite initier la capture. C'est elle qui est en charge d'instancier l'objet *client*. Qui lui même initialise la capture et l'analyse des données. L'utilisateur peut ensuite mettre en pause cette capture à l'aide d'un bouton,
- la classe **PauseCaptureActivity** : elle est appelée lorsque l'utilisateur souhaite mettre en pause la capture des données. Celui-ci dispose ensuite de deux choix. Le premier lui permet de relancer la capture tandis que le second lui permet de retourner à l'activité *SensorsActivity*.

Chapitre 7

Environnement de travail

Nous décrirons dans cette partie l'environnement de travail qui a été mis en place pour la réalisation de notre projet. Les langages de développement et les technologies utilisés nous ont été imposés par notre client.

7.1 Matériel et lieux de travail

L'essentiel de notre logiciel a été développé dans les locaux du *CREMI*. Ces locaux sont mis à disposition des étudiants de l'Université de Bordeaux et ceci en libre service. Cet endroit a été pour nous un lieu idéal de réunion ainsi qu'un lieu propice au travail. Cependant nous avons aussi, à l'aide de nos ordinateurs personnels, développé chez nous lors de nos temps libres.

7.2 Langages de développement

7.2.1 Java

La plupart des applications *Android* sont développées en *Java*, bien qu'il soit possible aussi d'en écrire en *C++*. Comme le sujet du projet nous imposait *Java*, nous avons utilisé ce langage. Le driver a également été développé en *Java* afin de rester en cohérence avec l'application *Android*, et éviter divers problèmes qui auraient pu survenir en cas d'utilisation de langages différents dans un même projet.

Pour notre développement nous avons utilisé la version 1.8 de *Java*.

7.2.2 XML

Les interfaces graphiques sous *Android* sont décrites en *XML*, dans un fichier particulier. A son lancement, l'application analyse le fichier et affiche les différents éléments graphiques suivant la description donnée.

7.3 Technologies utilisées

7.3.1 Eclipse

Le projet a été réalisé avec l'Environnement de Développement Intégré (Integrated Development Environment) *Eclipse*, bien connu par les développeurs *Java*. Outre le fait qu'il ait des multiples fonctionnalités, il intègre aussi une extension pour créer des applications *Android*. Il existe également *Android Studio* mais le sujet nous imposait le logiciel *Eclipse*.

7.3.2 Android SDK

Le *Software Development Kit Android* est un ensemble d'outils facilitant la création et la gestion d'application *Android*. Il automatise beaucoup de tâches pour le développeur, notamment en générant des ressources, et propose aussi un émulateur de terminal *Android*.

7.3.3 SVN

Pour coordonner notre équipe de développement, nous avons utilisé le gestionnaire de version *SVN*. En plus d'être performant, *Eclipse* propose une extension pour en faire l'usage. C'est-à-dire que nous pouvions utiliser directement ce gestionnaire de version via *Eclipse*, sans avoir besoin de passer par un terminal UNIX.

Chapitre 8

Réalisation

Intéressons-nous à présent à la réalisation du logiciel, aux différentes techniques d'implémentations, aux choix qui ont été faits, mais aussi aux limites et aux problèmes rencontrés. Nous parlerons également des améliorations possibles qui auraient pu être faites avec plus de temps.

8.1 Aides consultées

Au cours du développement, nous nous sommes servis de plusieurs ressources différentes, notamment des tutoriels sur le système *Android*. Notre principale source d'inspiration a été le site de l'API *Android* [1], ainsi que des sites tels que OpenClassrooms [3] notamment pour l'utilisation des sockets en java [5], ou Developpez.com [2]. Concernant l'utilisation du microphone nous nous sommes appuyés sur le site web developer.samsung.com [6].

8.2 Fonctionnalités implémentées

8.2.1 Connexion entre smartphone et ordinateur

La façon la plus simple de communiquer via le réseau est l'utilisation de sockets. Bien qu'il existe différentes bibliothèques externes permettant une gestion plus automatisée du réseau, nous maîtrisons mieux les techniques basées sur les sockets, étant également conscient que cette façon de procéder est assez "bas-niveau".

La connexion fonctionne via l'utilisation de sockets. L'ordinateur initialise la côté serveur via l'objet *ServerConnection* qui initialise un objet *ServerSocket* et la socket de communication à laquelle le smartphone pourra se connecter via l'objet *ClientConnection* qui initialisera la socket côté client. L'utilisateur spécifiera l'adresse IP du serveur qui sera donnée en paramètre à la socket cliente du smartphone ayant pour objectif d'initier la connexion avec le serveur. Concernant le port utilisé par les sockets, nous avons choisi le port par défaut 7101 qui est un port non utilisé par les applications de base du monde UNIX.

8.2.2 Autour des capteurs

L'API Java fournit tout ce qui est nécessaire pour récupérer les informations venant des capteurs. Il n'y a qu'une seule manière de gérer l'accéléromètre, en revanche deux méthodes sont possibles pour le microphone. L'une des deux compresse les données et l'autre non. Nous avons choisi cette dernière pour avoir les valeurs brutes.

Chaque capteur est représenté par une classe. L'idée était de faire une classe abstraite, *AbstractSensor*, présentant la plupart des méthodes utilisées pour gérer les capteurs, afin de minimiser l'effort si nous voulions en rajouter un nouveau. Seuls, le microphone et l'accéléromètre ont des manières différentes d'être gérés. Il n'y a donc pas eu possibilité de faire du code commun entre les deux. En revanche, le capteur de proximité fonctionne selon le même schéma que l'accéléromètre, la classe *AbstractSensor* nous a donc été utile. La classe représentant le microphone réalise un traitement de plus sur les données, une *Root Mean Square*, afin de ne récupérer que la valeur efficace de l'amplitude du signal mesurée.

Les implémentations se trouvent dans les classes *AbstractSensor*, *AccelerometerSensor*, *MicrophoneSensor* et *ProximitySensor*.

8.2.3 Analyse des données

Pour analyser les données, il y a plusieurs techniques différentes. Les plus compliquées sont celles basées sur l'analyse du signal, mais nous n'avions pas à chercher si compliqué. Nous avions pensé à deux méthodes différentes : la première se basant sur la comparaison des données par rapport à des seuils, la seconde sur les variations entre deux valeurs adjacentes. Nous avons essayé les deux, et avons constaté que la première donnait de meilleurs résultats.

Comme chaque capteur mesure des valeurs de natures différentes sur l'environnement (la plupart du temps), il est nécessaire qu'il existe des analyses de données différentes pour chaque type.

Les analyseurs de données sont représentés par des classes. Chaque analyseur parcourt les données fournies par la capture, et prévient le smartphone lorsqu'il détecte qu'un mouvement a été fait par l'utilisateur et reconnu.

Les classes implémentées pour cette partie sont *AccelerometerAnalyzer* et *MicrophoneAnalyzer*, les codes d'analyses se situent dans les méthodes *analysis()*.

8.2.4 Rôle du driver

Le driver contrôle un logiciel à côté. La principale question a été de savoir lequel. Nous avons choisi VLC, vu qu'il se rapproche d'un logiciel de télévision. Il se sert aussi d'un fichier de configuration écrit en *XML* pour permettre une plus grande souplesse d'utilisation. La bibliothèque utilisée pour lire des fichiers *XML* est *JDOM*, choisie pour sa popularité.

L'ordinateur joue le rôle du serveur dans le cadre de ce service. Le driver est donc lancé en premier et indique une adresse IP (visuellement) à laquelle le terminal devra se connecter. Il est muni de deux fichiers de configuration : le premier spécifie des séquences d'événements et leur associe des identifiants, le second lie ces identifiants à des chaînes de caractères représentant l'action à effectuer. Pour lire ces fichiers, il utilise un objet de type *XMLReader* se basant sur la bibliothèque *JDOM*, la lecture des données se situe dans la méthode *build()*.

Lorsque le smartphone se connecte, il lui envoie les informations lues dans le premier fichier. Il attend ensuite que le smartphone lui renvoie des identifiants à partir desquels il choisira l'action à effectuer, en fonction du second fichier de configuration. Ce second fichier n'est pas vraiment utile mais pratique, il est nécessaire de modifier la fonction de choix d'actions si l'on veut adapter le comportement du driver.

Le driver contrôle le logiciel VLC en tapant des raccourcis claviers sur l'ordinateur à l'aide d'un objet de type *Robot*. Les raccourcis claviers ne sont pas les mêmes d'un système d'exploitation à un autre, il a fallu récupérer le nom du système d'exploitation sur lequel le driver est exécuté pour adapter les raccourcis claviers. De plus, certains logiciels de vidéos ont des raccourcis en communs. Par exemple un appui sur la barre espace du clavier met en "pause". Nous pouvons donc contrôler divers

logiciels qui utilisent ces mêmes raccourcis comme *iTunes*, *mPlayer* ou même *Youtube*. La seule et unique condition est d'avoir le logiciel en premier plan sur l'écran de l'ordinateur.

8.2.5 Rôle de l'application

Le smartphone est au centre du service, autant au niveau utilisateur qu'au niveau analyse des données. Il a le rôle du client, par rapport à l'ordinateur. Il doit donc tout d'abord se connecter à ce dernier, duquel il recevra un premier message venant du driver.

Après cela, il initialise ses capteurs et les analyseurs associés, puis se lance. Après son lancement, il passe son temps à récupérer les informations des capteurs pour les envoyer vers les analyseurs. Il gère également la fenêtre de temps pendant laquelle un utilisateur peut exécuter une séquence. À chaque fois qu'un analyseur signale un mouvement, il le comptabilise et déplace la fenêtre de temps. Si au bout d'un certain temps (une seconde environ), rien n'a été de nouveau détecté, il regarde si une séquence spécifiée par le driver a été reconnue et il envoie l'identifiant associé le cas échéant. Il réinitialise ensuite la fenêtre de temps et la comptabilisation des mouvements détectés.

Outre la fonction d'analyse de l'environnement, le smartphone offre également une interface. Celle-ci permet à l'utilisateur de se connecter à l'ordinateur via l'adresse IP que ce dernier fournit lors de son lancement, mais aussi des aides au cas où l'utilisateur serait démunie quant à la marche à suivre.

8.3 Limites du logiciel

8.3.1 Méthode d'analyse

L'analyse de données est basée sur des comparaisons de valeurs par rapport à des seuils. Cette méthode est peu sûre, car une modification de l'environnement durant plusieurs secondes sera considérée comme plusieurs mouvement de la part de l'utilisateur. De plus, elle est totalement dépendante du type de mouvement que va effectuer l'utilisateur. C'est-à-dire que si celui-ci ne fait pas de mouvement nets et précis, cela peut fausser l'analyse.

8.3.2 Non-portabilité

De plus, les seuils de dépassement sont fixés en dur dans le code, et ils sont basés sur un seul terminal que nous avons utilisé pour nos tests. De ce fait, l'application n'est pas portable, il sera nécessaire de changer ces valeurs pour chaque nouveau terminal.

8.3.3 Consommation processeur

Après avoir fait quelques tests, nous nous sommes rendus compte que l'application utilisait d'avantage le processeur lorsqu'elle était en fond de tâche (au repos) que lorsqu'elle était active. Ceci vient sûrement du fait que les différents threads qui sont utilisés pour les analyses et récupérations de données des capteurs ne sont pas endormis lors du passage en fond de tâche. Ces derniers ne font que passer la main aux autres threads. Nous n'avons pas réussi à endormir les threads et à les réveiller.

8.4 Améliorations possibles

8.4.1 Une meilleure analyse

L'amélioration principale serait au niveau de l'analyse des données renvoyées par les capteurs. Nous savons que c'est le gros point faible de l'application, car nous avons implémenté une méthode très simple. Pour la rendre plus intéressante, il faudrait se baser sur des algorithmes de traitement du

signal, notamment si l'on souhaite utiliser le microphone.

Dans tous les cas, il serait nécessaire d'appliquer des filtres pour éliminer le bruit, et ensuite rendre plus fine la détection de variations sur les valeurs mesurées par les capteurs.

8.4.2 De l'étalonnage

L'application ne s'adapte pas à n'importe quel terminal, à cause des valeurs de seuils fixées en dur dans le code. Une amélioration envisageable serait d'effectuer un étalonnage de l'appareil (option proposée à l'utilisateur par exemple) afin de récupérer les valeurs mesurées par les capteurs au repos (terminal immobile et couché), ainsi que celles mesurées lorsque l'utilisateur agit sur l'environnement du terminal, en lui demandant d'exécuter une suite de mouvements adaptés à l'opération.

8.4.3 Endormir les threads

Pour la consommation processeur, l'idée a plus ou moins déjà été donnée. Il s'agirait d'arriver à endormir les threads lorsque l'application passe en fond de tâche, et les réveiller une fois qu'elle est remise au premier plan.

8.4.4 Capteur de proximité

Nous aurions pu également intégrer le capteur de proximité pour capturer des données. Nous y avons pensé un peu tardivement ce qui explique qu'il n'est pas fonctionnel dans notre logiciel mais qu'une classe lui soit réservée. De plus, le capteur de proximité renvoie des données "binaires". C'est-à-dire que soit il détecte un objet soit il ne le détecte pas. Il n'y a pas de juste milieu. C'est pourquoi nous pensons que son utilisation ainsi que l'analyse de ses données auraient pu être faite facilement.

Chapitre 9

Tests

9.1 Politique

Initialement, nous avions prévu de commencer le développement de tests avant le logiciel en lui-même (comme renseigné dans notre diagramme de Gantt). Cependant, une partie de nos tests fonctionnels ont été faits lors de l'implémentation de nos prototypes. De plus, nous avions au préalable produit une architecture "brute". C'est-à-dire que nous étions conscients du fait qu'il nous faudrait rajouter des classes et des paquetages au cours du développement. Nous avons donc revu notre politique de tests logiciel.

Il nous a donc fallu décaler la date de départ du développement de nos tests. C'est-à-dire que nous avons implémenté nos tests pendant et après le développement de notre logiciel.

Les tests réalisés pendant, nous ont permis de nous aider lorsque nous étions bloqués. Ils ont essentiellement été des tests fonctionnels.

Les tests réalisés après sont les plus nombreux, majoritairement des tests structurels et unitaires. Ils nous ont permis de corriger plusieurs bugs et sont pour la plupart, associés à des tests de couverture afin de justement tester la couverture de notre code.

Nos tests ont été écrits à l'aide de l'outil *JUnit4*, et la couverture à l'aide de l'outil *Emma*, tous deux disponibles avec *Eclipse*.

Nous décrirons dans les parties suivantes les tests qui ont été implémentés.

9.2 Capture et analyse

Les classes gérant la capture et l'analyse sont les suivantes :

- la classe **MicrophoneSensor**,
- la classe **AccelerometerSensor**,
- la classe **ProximitySensor**,
- la classe **MicrophoneAnalyzer**,
- la classe **AccelerometerAnalyzer**,
- la classe **SortedAssociationList**,
- la classe **SequenceAnalyzer**.

Elles sont situées dans les paquetages *client* et *analysis*.

Les tests associés à ces classes sont situés dans le paquetage *analysis.tests*.

9.2.1 Capteurs

Les capteurs sont gérés dans les classes de l'API *Android*. Dans notre application, les classes les concernant ne font que faciliter la gestion de ceux-ci. Les tests à effectuer sont principalement de la couverture de code, puisque les données que les capteurs mesurent dépendent de l'environnement, il est donc impossible de prédire les résultats. La plupart des fonctions contiennent très peu de branchements conditionnels, car elles ne font que des choses basiques, comme initialiser l'objet ou récupérer les données depuis les capteurs physiques et les renvoyer.

De ce fait, il nous a paru plus important de nous concentrer sur la partie analyse, plus algorithmique, plutôt que de s'attarder sur un point qui nous échappait en partie.

9.2.2 Analyseurs

Les analyseurs sont chargés de détecter des variations dans les données mesurées. Il est donc important de s'assurer du bon fonctionnement des algorithmes, en leur injectant des données et en vérifiant que les résultats sont conformes à nos attentes. Ce sont des tests fonctionnels. Comme les objets faisant les analyses peuvent détecter plusieurs variations différentes, il faut aussi des tests qui parcourent tous les branchements conditionnels, des tests structurels. Ces deux types de tests sont réalisés avec des tests unitaires.

Pour juger du bon fonctionnement de ces derniers tests, des tests de couverture ont été mis en oeuvre. Les résultats des tests de couverture sont donnés en annexe.

SortedAssociationList :

Cet objet permet de ranger des couples de valeurs entières dans des tableaux, triés par ordre croissant, selon la première valeur de chaque couple, que nous nommerons "clé". Ces clés doivent être strictement positives et sont uniques. C'est-à-dire que l'on ne trouvera pas plusieurs occurrences d'une même clé. La seconde valeur peut être tout simplement nommée "valeur" ou "valeur associée".

Nous pouvons progresser dans la liste de clés, une par une grâce à un compteur, initialisé à zéro. Dès lors qu'une clé est atteinte par le compteur, la position de la clé dans le tableau est retenue. Comme les valeurs de clés sont strictement positives et triées par ordre croissant, chaque clé est atteignable. L'avancement est fait grâce à une fonction, et il est possible de récupérer la clé et la valeur courante.

Les tests consistent donc à bien vérifier que l'on ne peut pas rentrer de valeurs interdites en tant que clés, puis à tester les avancements, vérifier que nous récupérons les valeurs attendues en fonction des couples que l'on a enregistrés et l'état de l'avancement. Les tests passent tous, car les problèmes qu'ils ont mis en lumière ont été corrigés. Les tests couvrent le code de cet objet à raison de 100%, chaque ligne est donc visitée au moins une fois.

SequenceAnalyzer :

Nous appelons "événement" ce que les analyseurs de données détectent comme une variation au niveau des données, par exemple un déplacement vers la gauche, une frappe sur la table, etc... L'analyseur de séquences gère l'avancement des séquences d'événements, permettant d'offrir plus de possibilités à l'utilisateur. Par exemple, faire deux actions différentes s'il tape une ou deux fois sur la table. Il se base sur l'objet précédent, en associant les événements à des *SortedAssociationList*. Sachant que pour un événement donné, les clés correspondent au nombre de fois où celui-ci doit être exécuté, et les valeurs à des identifiants associés. Au plus un événement peut être en cours d'avancement, dès lors qu'un autre événement est détecté, l'objet remet tous les autres à zéro.

Les tests principaux concernent donc l'avancement dans les différentes séquences, notamment en croissant différents événements, et en vérifiant que l'objet réagit bien, et donne les bons résultats. L'ajout de nouveaux événements est également testé. Avec ces tests, le comportement de l'objet a pu être amélioré car quelques bugs ont été découverts, et nous avons également l'assurance que toutes les lignes du code ont été visitées au moins une fois, car les tests de couvertures affichent un résultat de 100% de code couvert.

AccelerometerAnalyzer :

L'analyse des données venant de l'accéléromètre est faite dans cet objet. Les tests servent à s'assurer qu'il renvoie les événements attendus lorsque l'on lui donne des valeurs d'accélération qui dépassent les seuils fixés. Pour faire une bonne couverture, il faut lui donner assez de données à analyser pour que l'objet passe par tous les chemins possibles, c'est-à-dire tous les événements qu'il est censé détecter. La couverture révèle que 99.3% du code est visité, alors que chaque fonction est à 100% couverte. Le problème viendrait d'une instruction *while* qui est pourtant exécutée, car c'est celle qui conditionne la boucle d'analyse.

MicrophoneAnalyzer :

À l'instar de l'objet précédent, les tests sont quasiment de même natures, sauf que ce sont sur des événements concernant les données du microphone. De même, les tests de couverture montrent que le code est visité à 98.2%, avec le même problème... difficilement compréhensible.

9.3 Connexion

Pour rappel le paquetage *connection* est composé des classes suivantes :

- la classe abstraite **Connection** : classe abstraite,
- la classe **ClientConnection** : connexion côté client,
- la classe **ServerConnection** : connexion côté serveur.

Les tests associés à ces classes sont situés dans le paquetage *connection.tests*.

Description des tests :

Le test *ServerSendReceiveTest* est associé à la classe *ServerConnection*. Il permet de tester le bon fonctionnement de l'envoi et la réception de message par le serveur.

Le test *ClientSendReceiveTest* est associé à la classe *ClientConnection*. Il permet de tester le bon fonctionnement de l'envoi et la réception de message par le client.

Ils permettent de tester le module de connexion en boîte grise. Voici la liste des buts de ces tests :

- vérifier le bon fonctionnement de l'initialisation,
- vérifier le bon fonctionnement de l'envoi de message,
- vérifier le bon fonctionnement de la réception de message,
- vérifier la fermeture correcte de la connexion côté client et côté serveur.

Pour réaliser ces tests nous devons tout d'abord lancer le test *ServerSendReceiveTest* puis le test *ClientSendReceiveTest*. Le serveur va initialiser la connexion et le client pourra alors se connecter. Les deux tests sont réalisés de la manière suivante :

- une fonction précédée de la balise *@Before*, initialise les objets de la connexion,
- une fonction précédée de la balise *@After*, ferme la connexion,
- une fonction précédée de la balise *@Test*, effectue les envois et les réceptions de message.

Tout au long du test nous vérifierons que la méthode *getInfo()* nous renvoie bien un message cohérent en fonction de l'état de la connexion. Pour pouvoir manipuler plus aisément les différentes exceptions générées par des erreurs liées au réseau nous avons créé une exception *ConnectionException* située dans la classe éponyme. Pour tester si l'exception que nous avons créé est bien utilisée, nous avons initialisé la connexion côté client avec une mauvaise adresse IP. Dans le cas où une erreur *ConnectionException* est détectée nous initialisons de nouveau la connexion avec l'adresse « 127.0.0.1 ».

```
@Before
public void setUp() throws Exception {
    try {
        this.client = new ClientConnection("bad IP");
    } catch (ConnectionException e) {
        this.client = new ClientConnection("127.0.0.1");
    }
    System.out.println("Client information :\n" + this.client.getInfo());
}
```

Observations et couverture :

À l'aide des résultats visibles en annexe B.1.1 et des tests de couverture en annexe B.1.2 nous pouvons constater que l'exception *ConnectionException* est bien "catchée" pour initialiser correctement la connexion pour la suite du test.

Concernant les tests de couverture nous pouvons observer plusieurs choses. La première est que les tests du client et du serveur passent par toutes les méthodes de la classe abstraite *Connection* (*send()*, *receive()* et *getInfo()*). Nous notons 84,4% de couverture sur la classe *Connection*. Pour tester le reste du code de cette classe il faudrait tester les levées d'exceptions des méthodes *send()* et *receive()*. Ces méthodes sont liées aux fonctions présentes dans l'API *socket* et pour cette raison, nous ne les testerons pas davantage.

De plus, nous pouvons observer que le serveur est couvert par le test à 47,3% seulement car avec ce test nous ne regardons pas les cas d'erreurs liés aux différentes levées d'exceptions. Pour ce faire, nous pouvons lancer deux serveurs sur la même adresse IP (lancer deux fois le test *ServerSendReceiveTest*) pour tester les erreurs liées au constructeur de la classe *ServerConnection*. En revanche les levées d'exceptions des méthodes *accept()* et *close()* ne sont pas testables avec ces tests.

Le code de *ClientSendReceiveTest* couvre 62,9% du code car comme pour le test serveur nous ne testons pas les possibles levées d'exception de la méthode *close()*.

9.4 Interface Homme Machine

Pour rappel voici les activités gérant l'interface homme machine :

- l'activité **MainActivity**,
- l'activité **ConnectionActivity**,
- l'activité **ResultConnectionActivity**,
- l'activité **SensorsActivity**,
- l'activité **SensorsCaptureActivity**,
- l'activité **PauseCaptureActivity**.

Nous avons précédemment défini ces activités comme les activités les plus importantes du paquetage *com.bordeaux.rgacremote*. C'est pourquoi nous ne testons que ces six activités.

Les tests associés à ces activités sont situés dans le paquetage *com.bordeaux.rgacremote.tests*.

Description des tests :

Chaque activité possède un test qui lui est associé. Son nom est composé du nom de l'activité qu'il teste + la chaîne de caractères "Test". Par exemple l'activité de test de *MainActivity* est l'activité *MainActivityTest*. Les activités tests ont été écrites à l'aide de l'outil *Android JUnit Test*, disponible dans la bibliothèque *Android* de base.

Les six activités de tests ont sensiblement les mêmes fonctionnalités, à quelques détails près. C'est-à-dire qu'elles ont pour fonction de créer l'activité à laquelle elles sont associées puis de vérifier à l'aide d'assertions que les objets visuels de l'activité sont correctement créés. De plus, certains tests vérifient que leurs dimensions sont correctes par rapport à celles qui sont données dans les fichiers *XML* des activités.

Nous avions aussi dans l'idée de vérifier, si chaque activité en charge d'en créer une autre le faisait. C'est-à-dire si l'activité est bien appelée et si elle n'est pas nulle. Cependant, pour une raison que nous n'avons pas su expliquer les tests ne fonctionnaient pas. Et ceci malgré la consultation de la partie dédiée aux tests sur le site de l'API *Android*. Nous avons donc choisi de contourner le problème en ajoutant une méthode privée, *isCallable()* à chaque activité devant en créer une autre. Cette méthode a pour but de tester si l'activité peut effectivement être appelée ou pas. Cela permet d'éviter d'avoir à le tester directement dans une activité test. Nous sommes cependant conscients que cela ne permet pas de garantir le fait que l'appel est correct, car il aurait évidemment fallu tester le bon fonctionnement de cette méthode, *isCallable()*.

Observations et couverture :

Tout d'abord nous n'avons pas su configurer notre plugin de couverture de code, *Emma*, afin qu'il puisse prendre en compte la couverture des *Android JUnit Test*. Nous n'avons donc pas d'idées quant au pourcentage de couverture du code de notre IHM.

Concernant les activités tests certaines donnent un résultat positif et d'autres un résultat négatif.

Les activités avec résultat positif sont :

- *MainActivityTest*,
- *ConnectionActivityTest*,
- *PauseCaptureActivityTest*.

Les activités avec résultat négatif sont :

- *SensorsActivityTest*,
- *SensorsCaptureTest*,
- *ResultConnectionActivity*.

Ces trois activités semblent générer le même problème, à savoir, un pointeur nul. Nous n'avons pas pu corriger le problème en grande partie par manque de temps.

Améliorations possibles :

Afin d'améliorer nos tests, il faudrait tester que chaque éléments de chaque classe est créé dans des dimensions correctes. Et que lorsque celui-ci affiche une chaîne de caractères celle-ci soit correcte également. De plus, il faudrait également tester les classes que nous avons qualifié de "moins importantes".

9.5 RGACDriver

RGACDriver est composé des classes *Server*, *Controller* et *Device*.

Les tests associés à ces classes sont situés dans le paquetage *server.tests*.

En voici les résultats :

9.5.1 Server

La classe *testServer* permet de tester la connexion du serveur en boîte noire.

Voici les étapes de ce test :

- il démarre le serveur,
- il vérifie que le message initial envoyé au client est correct,
- il vérifie que le serveur affiche bien les informations reçues,
- il vérifie l'état de la connexion,
- il vérifie que le serveur reçoit bien les informations que le client lui envoie toutes les 200 millisecondes.

Le code du test peut être modifié pour changer le nombre de message que le client envoie au serveur. Le test sera donc fait sur un temps plus long.

Observations : Nous pouvons remarquer que selon les ordinateurs, si nous abaissons le délai entre deux messages du client en dessous de 200 millisecondes, le serveur ignore les derniers messages. Et si nous l'abaissons en dessous de 120 millisecondes, le serveur ne reçoit pas le troisième message et donc, fait échouer le test. Ceci peut être considéré comme un comportement normal étant donné que le serveur récupère les informations toutes les 120 millisecondes.

Améliorations possibles et recouvrement : Nous pouvons voir que le code du serveur est couvert par le test à hauteur de 67.5%. Ceci est dû au fait que nous ne testons pas le code de la fonction *main()*, ni les levées d'exceptions lorsque l'envoi ou la réception d'un message a échoué.

Pour tester l'échec de la réception, il aurait fallu tuer le client pendant qu'il envoie un message au serveur, ce qui aurait eu pour effet la levée de l'exception correspondante. Et pour tester l'échec de l'envoi, il aurait fallu tuer le serveur pendant qu'il reçoit des données. Cela aurait aussi eu pour effet une levée d'exception.

9.5.2 Controller

La classe *testController* permet de tester la classe *Controller* qui est testée en boîte grise.

Voici les étapes de ce test :

- il démarre le *Controller*,
- il instancie un *KeyListener* pour écouter les actions effectuées sur le clavier
- il exécute les différentes méthodes du main,
- il vérifie que la méthode *analysisConfig* analyse bien le fichier de configuration XML que l'on envoie au smartphone.

Notons qu'il est important de préciser que lors de l'exécution de ce test, il ne faut en aucun cas appuyer sur une touche du clavier. Il ne faut pas non plus que le système d'exploitation puisse récupérer la touche clavier avant le programme de test. C'est pourquoi, il faut bien vérifier que les touches *F11*, *haut*, *bas*, *gauche*, *droite*, *Ctrl* (pour Linux et Windows) et les touches *cmd*, *alt* (pour MacOS) ne soient

pas récupérées par le système d'exploitation.

Après avoir testé les différentes valeurs possibles pour lesquelles le *Controller* effectue une action sur le *Device*, nous passons au test en boîte noire.

Ce test a pour but d'envoyer des valeurs entières ou décimales aléatoires ainsi que des chaînes de caractères au *Server*. Le *Controller* les récupérera et les analysera par la suite. Le but de ce test est de vérifier la présence de comportements anormaux du *Controller* ou du *Server* dans des situations extrêmes.

Observations : Nous avons observé dans le test *testController* que lorsque le Controller fait une action sur le *Device*, il faut attendre (avec une intensité minimale et avec des actions claviers), au moins 500 millisecondes pour que le controller puisse à nouveau analyser une action. Dans le test boîte noire, nous remarquons que lorsque nous envoyons des chaînes de caractères, le client n'envoie pas la partie après le \n. Ceci est un comportement normal de la connexion.

Améliorations possibles et recouvrement : Nous pouvons voir que le code du *Controller* est couvert par le test à raison de 86.1%. Le test ne vérifie pas les levées d'exception dans les méthodes *analysisConfig()*, *giveIp()* et *startServer()*. Nous ne pouvons tester les levées d'exception ni l'affichage correct de l'adresse IP dans la méthode *giveIp()* car on ne peut pas forcer le système d'exploitation à renvoyer des valeurs différentes à *InetAddress.getLocalHost()*.

9.5.3 Device

La classe *testDevice* permet de tester la classe *Device* qu'il teste en boîte blanche.

Voici les étapes de ce test :

- il démarre le *Device*,
- il instancie un *KeyListener* pour écouter les actions effectuées au clavier,
- il lance les différentes méthodes de *Device*,
- il vérifie que pour chaque système d'exploitation, il appuie sur les bonnes touches.

Comme précisé ci-dessus dans le *testController*, il ne faut pas appuyer sur une touche du clavier pendant le test et il ne faut pas que le système d'exploitation capture la touche appuyée avant notre programme de test.

Observations : Le *Device* réagit normalement au test, il appuie sur les bonnes touches pour tous les systèmes d'exploitations.

Améliorations possibles et recouvrement : Nous pouvons voir que le *Device* est couvert par le test à raison de 99.2%. La fonction "mal" testée est *setIntensity()* avec un cas non testé : si la valeur passée en paramètre est supérieure à 10.

9.6 Performances

Nous avons également fait quelques tests de performance pour notre logiciel.

9.6.1 RGACDriver

À l'aide de l'outil *JVM Monitor*, disponible sous *Eclipse*, nous avons pu analyser le comportement de notre driver au niveau de sa consommation mémoire ainsi que du nombre de threads qu'il génère sur presque deux heures. Pendant ce temps, il a reçu une centaine d'informations qu'il a analysé et éventuellement traité. (voir annexe B.4.1 pour les graphiques).

Concernant la consommation mémoire, notre driver utilise entre 5Mo et 44Mo. Cette variation peut être expliquée par le fait que le *Garbage Collector* passe que lorsque certaines conditions sont vérifiées (voir définition du *Garbage Collector* pour plus d'informations).

Quant au nombre de threads nous pouvons constater qu'il utilise entre 17 et 19 threads (annexe B.4.1).

9.6.2 RGACRemote

Les tests suivants ont été réalisés sur un *Nexus 4* fait par *Google* et *LG* avec la version 5.0.1 d'*Android Lollipop*.

Consommation CPU

Nous avons pu observer les consommations CPU en fonction de l'état de l'application. C'est-à-dire que les consommations sont différentes lorsque l'application est en train d'effectuer des actions et lorsqu'elle est en veille.

RGACRemote effectue des actions lorsqu'elle analyse les données en provenance du microphone et de l'accéléromètre. Nous pouvons voir que notre application consomme en moyenne 10.5% de CPU. L'accès au capteur consomme quant à lui, en moyenne 6% de CPU et le "system server" consomme lui en moyenne 12.3%. Notons que le "system server" se charge de l'affichage des activités d'*Android*, de la connexion Wi-Fi, de la gestion d'énergie, de la récupération des informations du microphone pour notre application. La commande qui nous a permis de voir ces différents résultats n'a elle, consommé que 3.5% de CPU.

Nous pouvons voir que la consommation totale de CPU n'est pas très élevée ce qui indique une faible consommation électrique de notre application (voir annexe B.4.2 figure B.13 pour la consommation CPU de *RGACRemote* en action).

RGACRemote se retrouve en veille lorsque l'utilisateur a appuyé sur le bouton "Pause" ou lorsque celui-ci quitte ou change d'application. Sur cette capture d'écran suivante, nous pouvons facilement voir que *RGACRemote* consomme trop de CPU en veille alors qu'elle est censée ne rien faire. Nous pouvons facilement constaté la présence d'un problème (voir annexe B.4.2 figure B.14 pour voir la consommation CPU de *RGACRemote* en veille).

Consommation batterie

Nous avons également fait des tests pour connaître la consommation de la batterie par notre application. Pour le test de batterie, le téléphone était en mode avion avec le Wi-Fi et le Bluetooth actifs. L'écran était également allumé pendant tout le processus de test avec la luminosité réglée au minimum et en désactivant l'adaptation de luminosité automatique. Sur un total de 210 minutes de test, avec un envoi de 187 événements au driver, nous avons observé que nous utilisons environ 30%

de batterie. Sachant toutefois que sur ces 30% notre application n'en concerne que 2.
La batterie est consommée de la façon suivante :

- 22% sont consommés par l'écran du smartphone,
- 2% sont consommés par l'OS *Android*,
- 2% sont consommés par notre application, *RGACRemote*,
- 1% est consommé par la connexion wifi,
- 3% sont consommés par d'autres services.

Le test révèle également que notre application peut être active pour une durée d'environ 9h.
Pour plus de détails vous pouvez consulter les captures d'écrans se situant en annexe B.4.2 paragraphe "Consommation batterie".

9.7 Robustesse

9.7.1 Perturbations lors de la capture

Nous avons testé le comportement de notre application pendant la réception d'un appel ou d'un message texte.

La réception de ces notifications perturbent l'analyse des informations. Chaque notification est captée par le microphone, même en ayant le smartphone sur vibrer, un événement de détection sonore élevé est envoyé au driver.

Améliorations possibles : En ce qui concerne *RGACRemote*, il est important que des notifications *Android* ne perturbent pas son bon fonctionnement. Il faudrait donc les détecter et mettre en pause la capture et l'analyse pour prévenir d'éventuels faux résultats.

9.7.2 Exécution de RGACDriver sous Linux

Nous avons testé le comportement de *RGACDriver* sur Linux, habituellement il est testé sous Mac OS X. Et sous Mac OS X, *RGACDriver* nous affiche l'adresse IP de l'ordinateur sur le réseau local. Nous avons pu observer que sous Ubuntu 14.10, l'adresse IP était mauvaise, quoi qu'il se passe avec le réseau de l'ordinateur, elle affiche "127.0.1.1", la boucle locale.

Améliorations possibles : Il aurait fallu pour chaque interface réseau (avec l'objet *NetworkInterface*), afficher l'adresse IP correspondante.

9.7.3 Taux d'erreurs de l'analyse

Nous avons effectué plusieurs batteries de tests sur nos séquences prédéfinies afin de calculer le pourcentage d'erreurs dans la reconnaissance de ces séquences. Nous avons fait une centaine de tests pour chaque mouvement prédéfini.

Voici la liste des résultats (les pourcentages sont des pourcentages de réussite) :

- succession de LEFT / RIGHT (ou BACK / FORWARD pour VLC) : **86%** ,
- succession de UP / DOWN (ou VOLUME UP / VOLUME DOWN pour VLC) : **72%** ,
- succession de "un tap" (ou PLAY PAUSE pour VLC) : **73%** ,
- succession de "deux taps" (ou FULLSCREEN pour VLC) : **81%** ,
- succession de "trois taps" (ou INFORMATION OF MEDIA pour VLC) : **79%** ,
- succession de MIC LOW : **88%** ,
- succession de MIC HIGH : **70%** .

Soit un pourcentage moyen de : **78.4%**

Améliorations possibles : L'amélioration évidente serait d'atteindre 100% pour toutes les séquences. Cependant, nous remarquons que notre application fonctionne tout de même plus de 3 fois sur 4. Afin de nous rapprocher des 100% on aurait pu fournir à l'utilisateur la possibilité de régler la sensibilité de la capture en fonction de son smartphone mais aussi en fonction du support sur lequel est posé le smartphone. De plus, pour l'utilisation du microphone il faudrait pouvoir supprimer le "bruit de fond".

Nous aurions aussi pu trouver un meilleur moyen pour définir les seuils utilisés lors de l'analyse afin d'affiner les comparaisons et donc réduire le taux d'erreurs.

9.7.4 Réseaux Wi-Fi

Pour une raison que l'on ignore, la communication entre le smartphone et l'ordinateur ne fonctionne pas quand ils sont tous les deux connectés sur le réseau Eduroam en Wi-Fi. Néanmoins notre application semble fonctionner sur la majorité des routeurs.

Améliorations possibles : Il aurait fallu se renseigner plus précisément sur le fonctionnement du réseau Eduroam, et voir éventuellement les ports filtrés.

Chapitre 10

Conclusion générale

Ce projet nous a permis de nous familiariser avec l'univers de développement d'un logiciel *Android*. Bien que nous n'ayons pas utilisé les plus importantes fonctionnalités de l'API, cette première approche nous a initié à la programmation embarquée et le système de fonctionnement d'une application *Android*, ainsi qu'à la description d'interface graphique.

Cette première expérience du génie logiciel fut intéressante, tant au niveau du travail d'équipe, qu'au niveau de la recherche et l'implémentation de solutions pour arriver aux résultats attendus. Bien évidemment, le succès n'est pas toujours au rendez-vous, mais la satisfaction d'un début de travail fonctionnel est déjà une récompense.

Concernant l'application générale de notre logiciel, nous pourrions imaginer piloter différents appareils juste en agissant sur les capteurs du smartphone.

Annexe A

Prototypes

A.1 Prototypes implémentés

Voici des captures d'écrans des différents prototypes implémentés ainsi qu'une description de leurs fonctionnements. Leurs développements nous ont fait gagner un temps précieux.

Le prototype de test du capteur accéléromètre :



FIGURE A.1 – TestAccelero repos



FIGURE A.2 – TestAccelero détection

Description : La première capture d'écran montre l'interface du logiciel test "au repos". C'est-à-dire, lorsque l'accéléromètre ne détecte aucune modification sur l'axe Z. La seconde montre l'interface lorsqu'il détecte un mouvement sur l'axe Z. Le prototype utilise un algorithme simple pour détecter la présence de "tap" sur la surface où est posé le smartphone. Il calcule la différence entre l'ancienne et la nouvelle valeur de la position du smartphone sur l'axe Z. Si cette différence est supérieure à une certaine limite, il détecte un tap. Pour détecter plusieurs "tap", l'algorithme attend sept changements de valeurs sur l'axe Z par rapport au "tap" précédent. Et pour améliorer les résultats, il abaisse aussi la précision du "tap".

L'intérêt de ce prototype est de comprendre le fonctionnement de l'accéléromètre d'un smartphone *Android*. Avec la mise en place d'un algorithme simple, avec un système de comparaison de valeurs contiguës, nous avons réussi à capter des mouvements du smartphone selon un axe particulier. Nous avons aussi pu voir la différence de sensibilité entre les différents smartphone *Android* que nous avions en notre possession.

Le prototype de test du capteur microphone :



FIGURE A.3 – TestMicro repos

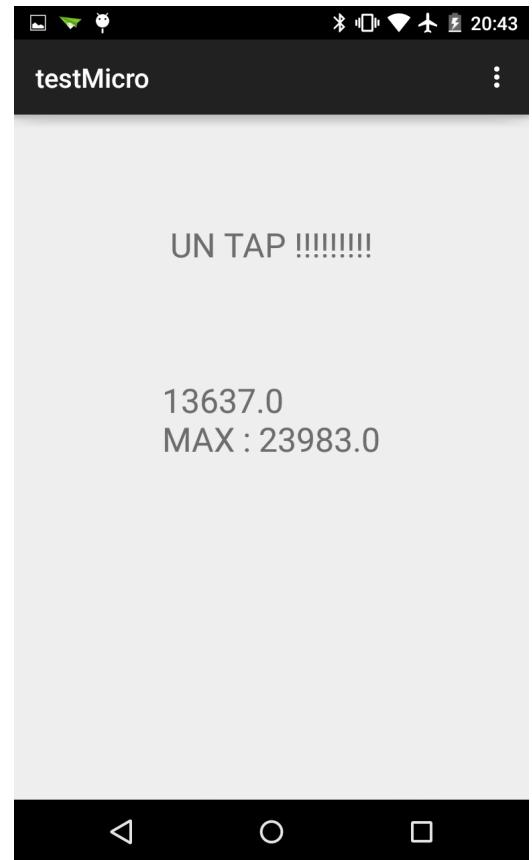


FIGURE A.4 – TestMicro détection

Description : La première capture d'écran montre l'interface du logiciel test "au repos". C'est-à-dire, lorsque le microphone ne détecte aucune modification. La seconde montre l'interface lorsqu'il détecte une intensité sonore notable. La détection des intensités sonores s'effectue avec le même algorithme que le prototype *TestAccelero*. La valeur max indique la valeur maximum détectée par le smartphone. L'intérêt de ce prototype est de comprendre le fonctionnement du microphone d'un smartphone *Android*. En effet, il y a deux manières qui permettent de récupérer le son sur *Android* : avec l'objet *AudioRecord* et avec l'objet *MediaRecord*. Ce dernier, nous permet d'avoir facilement l'amplitude avec en contre-partie un son compressé, alors qu'avec *AudioRecord* nous avons un son de meilleure qualité avec lequel nous pouvons faire des analyses plus précises. Donc, le prototype *TestMicro* nous a permis de voir les différents éléments d'implémentation pour capturer le son ainsi que leurs avantages et inconvénients.

Le prototype de test d'une connexion réseau :

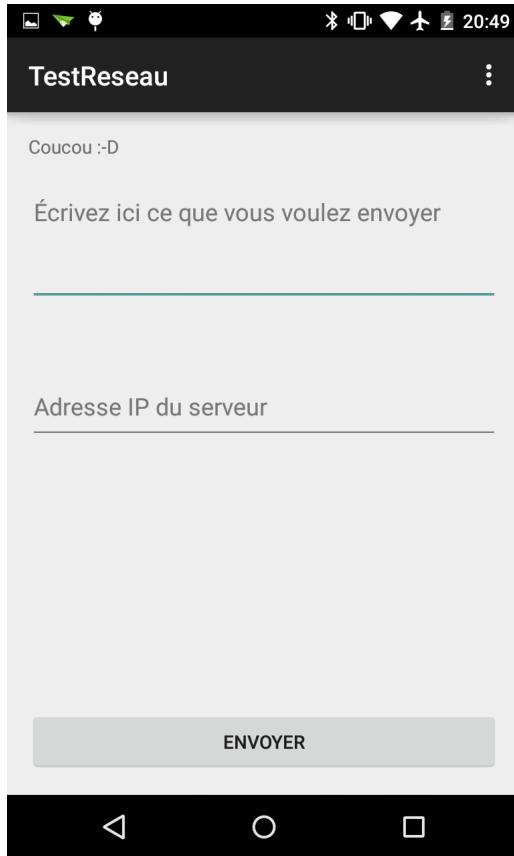


FIGURE A.5 – Test Réseau

Description : La capture d'écran montre l'interface du logiciel test au démarrage de l'application qui joue le rôle de client. Pour faire fonctionner ce prototype, il faut au préalable avoir démarré le serveur qui écoute sur le port 7101. Ensuite, on tape le message que l'on souhaite envoyer, on tape l'adresse IP du serveur et enfin on appuie sur le bouton envoyer. Lorsque l'on appuie sur ce dernier, le client ouvre une socket sur le port 7101 et réceptionne ce que lui envoie le serveur (prototype nommé Serveur), et l'affiche à la place d'un message prédéfini et pour finir il envoie le message qu'on a tapé précédemment.

L'intérêt de ce prototype est de voir qu'on utilise les mêmes méthodes *Java* pour pouvoir envoyer et recevoir des chaînes de caractères entre le smartphone *Android* et un ordinateur.

A.2 Prototype papier

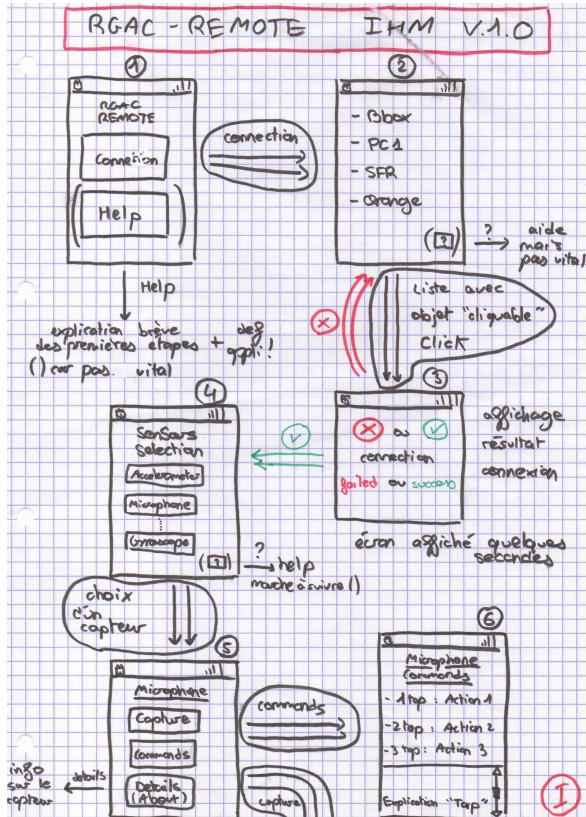


FIGURE A.6 – Test Micro repos

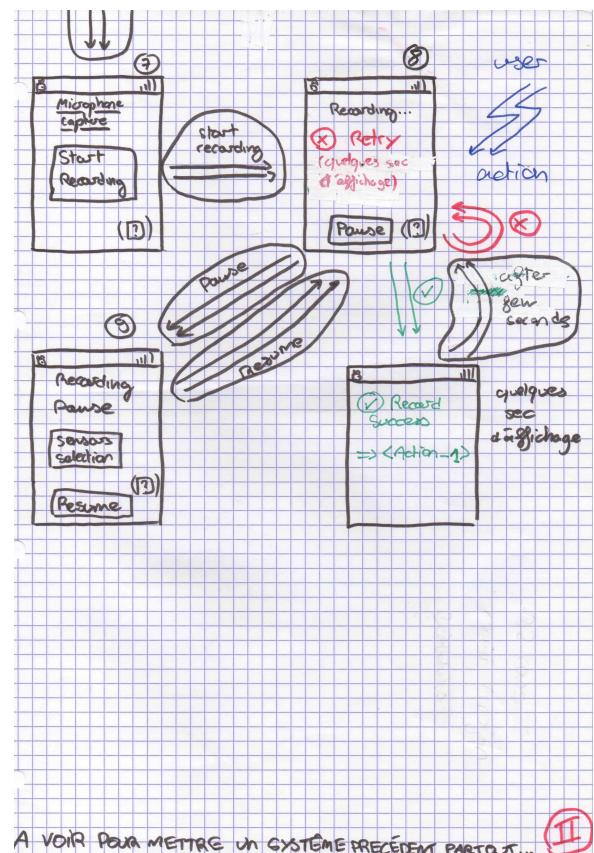


FIGURE A.7 – Test Micro détection

Description : Les deux images ci-dessus montrent un dessin papier que nous avons fait au moment de la spécification des besoins. C'était la première idée de la forme de notre IHM (Interface Homme Machine). Elle nous a permis d'imaginer ce dont pourrait avoir besoin l'utilisateur sur l'interface du logiciel. Mais ces dessins nous ont surtout permis de gagner du temps lors du développement de l'IHM notamment grâce au fait de savoir vers quoi on se dirige avec l'aide d'une visualisation papier.

Annexe B

Tests

B.1 Connexion

B.1.1 Résultats

```
Waiting for connection...
Server information :
The Server is connected to /127.0.0.1 .

Server Reception : Message1 send by client
Server Reception : Message2 send by client
Server information after test :
The Server is connected to /127.0.0.1 .| 

Information after close function : The Connection is closed.
```

FIGURE B.1 – résultat serveur

```
Client information :
The Server is connected to /127.0.0.1 .

Client : Message1 send by Server
Client : Message2 send by Server
Client information after test:
The Server is connected to /127.0.0.1 .

Information after close function : The Connection is closed.
```

FIGURE B.2 – résultat client

B.1.2 Couverture

connectionTest	47,5 %	87	96	183
ClientSendReceiveTest.java	0,0 %	0	90	90
ServerSendReceiveTest.java	93,5 %	87	6	93
connection	53,7 %	102	88	190
ClientConnection.java	0,0 %	0	35	35
ServerConnection.java	47,3 %	26	29	55
ServerConnection	47,3 %	26	29	55
Connection.java	84,4 %	76	14	90
Connection	84,4 %	76	14	90
receive()	77,8 %	21	6	27
send(String)	80,0 %	24	6	30
getInfo()	93,3 %	28	2	30
Connection()	100,0 %	3	0	3
ConnectionException.java	0,0 %	0	10	10

FIGURE B.3 – couverture ServerSendReceiveTest

connectionTest	43,7 %	80	103	183
ServerSendReceiveTest.java	0,0 %	0	93	93
ClientSendReceiveTest.java	88,9 %	80	10	90
connection	56,8 %	108	82	190
ServerConnection.java	0,0 %	0	55	55
Connection.java	84,4 %	76	14	90
Connection	84,4 %	76	14	90
receive()	77,8 %	21	6	27
send(String)	80,0 %	24	6	30
getInfo()	93,3 %	28	2	30
Connection()	100,0 %	3	0	3
ClientConnection.java	62,9 %	22	13	35
ClientConnection	62,9 %	22	13	35
close()	41,7 %	5	7	12
ClientConnection(String)	73,9 %	17	6	23
ConnectionException.java	100,0 %	10	0	10
ConnectionException	100,0 %	10	0	10
ConnectionException(String)	100,0 %	10	0	10

FIGURE B.4 – couverture ClientSendReceiveTest

B.2 Analyseurs

TestSortedAssociatedList (2) (6 avr. 2015 17:27:23)					
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions	
Analyzer	18,7 %	392	1 707	2 099	
src	18,7 %	392	1 707	2 099	
tests	15,7 %	182	979	1 161	
analysis	25,4 %	210	617	827	
AccelerometerAnalyzer.java	0,0 %	0	294	294	
SequenceAnalyzer.java	0,0 %	0	214	214	
MicrophoneAnalyzer.java	0,0 %	0	109	109	
SortedAssociatedList.java	100,0 %	210	0	210	
SortedAssociatedList	100,0 %	210	0	210	
SortedAssociatedList()	100,0 %	19	0	19	
add(int, int)	100,0 %	84	0	84	
getCurrentIndex()	100,0 %	8	0	8	
getCurrentValue()	100,0 %	8	0	8	
next()	100,0 %	48	0	48	
reset()	100,0 %	7	0	7	
toString()	100,0 %	36	0	36	
events	0,0 %	0	111	111	

FIGURE B.5 – SortedAssociationList

TestSequenceAnalyzer (6 avr. 2015 17:24:46)					
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions	
Analyzer	41,8 %	878	1 221	2 099	
src	41,8 %	878	1 221	2 099	
tests	42,7 %	496	665	1 161	
analysis	46,2 %	382	445	827	
AccelerometerAnalyzer.java	0,0 %	0	294	294	
MicrophoneAnalyzer.java	0,0 %	0	109	109	
SortedAssociatedList.java	80,0 %	168	42	210	
SequenceAnalyzer.java	100,0 %	214	0	214	
SequenceAnalyzer	100,0 %	214	0	214	
SequenceAnalyzer(int)	100,0 %	19	0	19	
add(int, int, int)	100,0 %	32	0	32	
clearValuesMatched()	100,0 %	4	0	4	
evolve(int)	100,0 %	99	0	99	
getValuesMatched()	100,0 %	3	0	3	
resetAll()	100,0 %	16	0	16	
toString()	100,0 %	41	0	41	
events	0,0 %	0	111	111	

FIGURE B.6 – SequenceAnalyzer

TestAccelerometerAnalyzer (6 avr. 2015 17:28:39)					
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions	
▼ Analyzer	26,3 %	552	1 547	2 099	
▼ src	26,3 %	552	1 547	2 099	
► tests	22,4 %	260	901	1 161	
▼ analysis	35,3 %	292	535	827	
► SequenceAnalyzer.java	0,0 %	0	214	214	
► SortedAssociatedList.java	0,0 %	0	210	210	
► MicrophoneAnalyzer.java	0,0 %	0	109	109	
▼ AccelerometerAnalyzer.java	99,3 %	292	2	294	
▼ AccelerometerAnalyzer	100,0 %	231	0	231	
► AccelerometerAnalyzer(IClient)	100,0 %	47	0	47	
► addData(Object)	100,0 %	6	0	6	
► analysis()	100,0 %	9	0	9	
► moveXDetection()	100,0 %	63	0	63	
► moveYDetection()	100,0 %	63	0	63	
► moveZDetection()	100,0 %	43	0	43	
► events	0,0 %	0	111	111	

FIGURE B.7 – AccelerometerAnalyzer

TestMicrophoneAnalyzer (6 avr. 2015 17:27:58)					
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions	
▼ Analyzer	10,3 %	217	1 882	2 099	
▼ src	10,3 %	217	1 882	2 099	
► tests	9,5 %	110	1 051	1 181	
▼ analysis	12,9 %	107	720	827	
► AccelerometerAnalyzer.java	0,0 %	0	294	294	
► SequenceAnalyzer.java	0,0 %	0	214	214	
► SortedAssociatedList.java	0,0 %	0	210	210	
▼ MicrophoneAnalyzer.java	98,2 %	107	2	109	
▼ MicrophoneAnalyzer	100,0 %	74	0	74	
► MicrophoneAnalyzer(Object, ICl...)	100,0 %	17	0	17	
► addData(Object)	100,0 %	6	0	6	
► analysis()	100,0 %	9	0	9	
► soundDetection()	100,0 %	42	0	42	
► events	0,0 %	0	111	111	

FIGURE B.8 – MicrophoneAnalyzer

B.3 Driver

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
RGACDriver	59,7 %	1 185	800	1 985
src	59,7 %	1 185	800	1 985
server.tests	45,4 %	398	478	876
TestDevice.java	0,0 %	0	332	332
TestServer.java	0,0 %	0	137	137
TestController.java	97,8 %	398	9	407
server	73,0 %	706	261	967
Device.java	60,8 %	220	142	362
Controller.java	86,1 %	323	52	375
Controller	86,1 %	186	30	216
startServer()	60,5 %	26	17	43
analysisConfig()	90,2 %	119	13	132
Controller(String[])	100,0 %	25	0	25
dataProcessing()	100,0 %	8	0	8
giveIp()	100,0 %	8	0	8
Server.java	67,5 %	83	40	123
View.java	74,8 %	80	27	107
xml	57,0 %	81	61	142
MainExemple.java	0,0 %	0	61	61
XMLReader.java	100,0 %	81	0	81

FIGURE B.9 – Controller

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
RGACDriver	34,8 %	691	1 294	1 985
src	34,8 %	691	1 294	1 985
server	37,1 %	359	608	967
Controller.java	0,0 %	0	375	375
Server.java	0,0 %	0	123	123
View.java	0,0 %	0	107	107
Device.java	99,2 %	359	3	362
Device	99,2 %	359	3	362
setIntensity(int)	85,0 %	17	3	20
Device()	100,0 %	23	0	23
backForward(boolean)	100,0 %	128	0	128
fullScreen()	100,0 %	31	0	31
informationOfMedia()	100,0 %	39	0	39
playPause()	100,0 %	9	0	9
volume(boolean)	100,0 %	112	0	112
server.tests	37,9 %	332	544	876
TestController.java	0,0 %	0	407	407
TestServer.java	0,0 %	0	137	137
TestDevice.java	100,0 %	332	0	332
xml	0,0 %	0	142	142

FIGURE B.10 – Device

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▼ RGACDriver				
▼ src				
▼ server				
► Controller.java	10,1 %	201	1 784	1 985
► Device.java	10,1 %	201	1 784	1 985
► View.java	8,6 %	83	884	967
► Server.java	0,0 %	0	375	375
► Device.java	0,0 %	0	362	362
► View.java	0,0 %	0	107	107
▼ Server.java	67,5 %	83	40	123
▼ Server				
► main(String[])	63,6 %	56	32	88
► run()	0,0 %	0	27	27
► Server()	84,4 %	27	5	32
► getConnectionStatus()	100,0 %	17	0	17
► getReceiveInformation()	100,0 %	3	0	3
► toBeSent(String)	100,0 %	3	0	3
► toBeSent(String)	100,0 %	6	0	6
► server.tests	13,5 %	118	758	876
► xml	0,0 %	0	142	142

FIGURE B.11 – Server

B.4 Performance

B.4.1 Driver

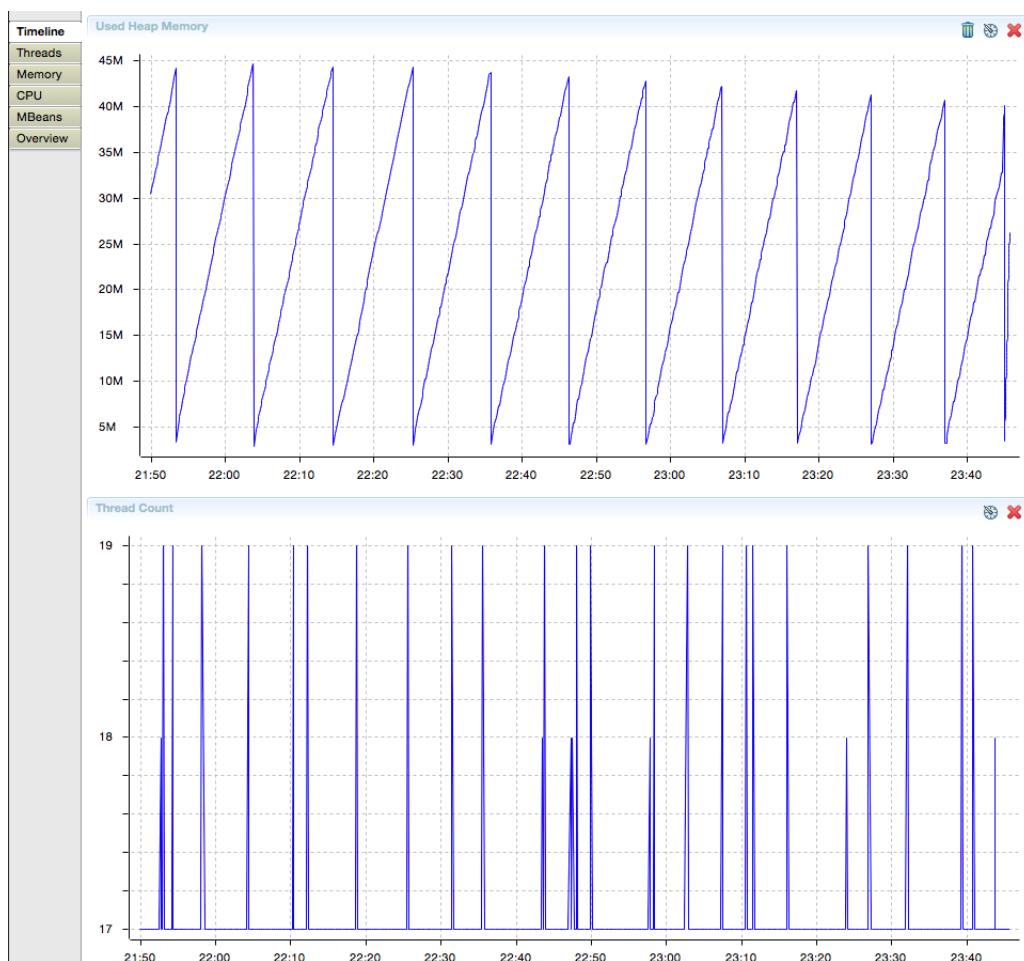


FIGURE B.12 – Informations sur le serveur

B.4.2 Application RGACRemote

Consommation CPU

```
Mem: 1718984K used, 160828K free, 0K shrd, 103288K buff, 976952K cached
CPU: 13.5% usr 25.2% sys 0.0% nic 61.1% idle 0.0% io 0.0% irq 0.0% sirq
Load average: 0.74 1.37 1.73 1/1211 17068
 PID  PPID USER      STAT  VSZ %VSZ CPU %CPU COMMAND
1780  1460 system    S < 1622m 88.1   0 12.3 system_server
16821 1460 app_144   S 1485m 80.6   0 10.5 {eaux.rgacremote} com.bordeaux.rgacremote
1461   1 nobody     S 9560  0.5   0  6.1 /system/bin/sensors.qcom
16760 16720 app_97   R <  940  0.0   0  3.5 top -d 1
```

FIGURE B.13 – Informations sur l’application RGAC en action.

```
Mem: 1722392K used, 157420K free, 0K shrd, 103304K buff, 976960K cached
CPU: 30.0% usr 69.9% sys 0.0% nic 0.0% idle 0.0% io 0.0% irq 0.0% sirq
Load average: 0.85 1.10 1.57 4/1177 17429
 PID  PPID USER      STAT  VSZ %VSZ CPU %CPU COMMAND
17284 1460 app_144   S 1485m 80.6   1 97.3 {eaux.rgacremote} com.bordeaux.rgacremote
17253 16720 app_97   R <  940  0.0   0  0.9 top -d 1
```

FIGURE B.14 – Informations sur l’application RGAC en veille.

Consommation batterie

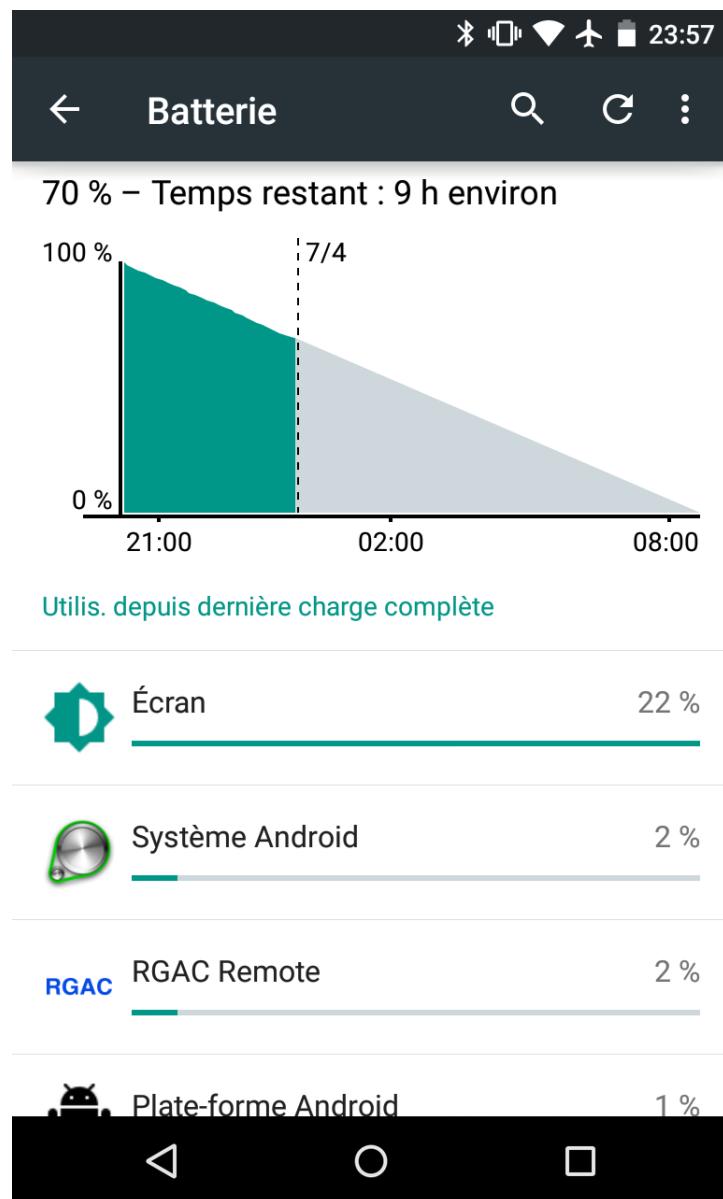


FIGURE B.15 – Informations sur la batterie (1)

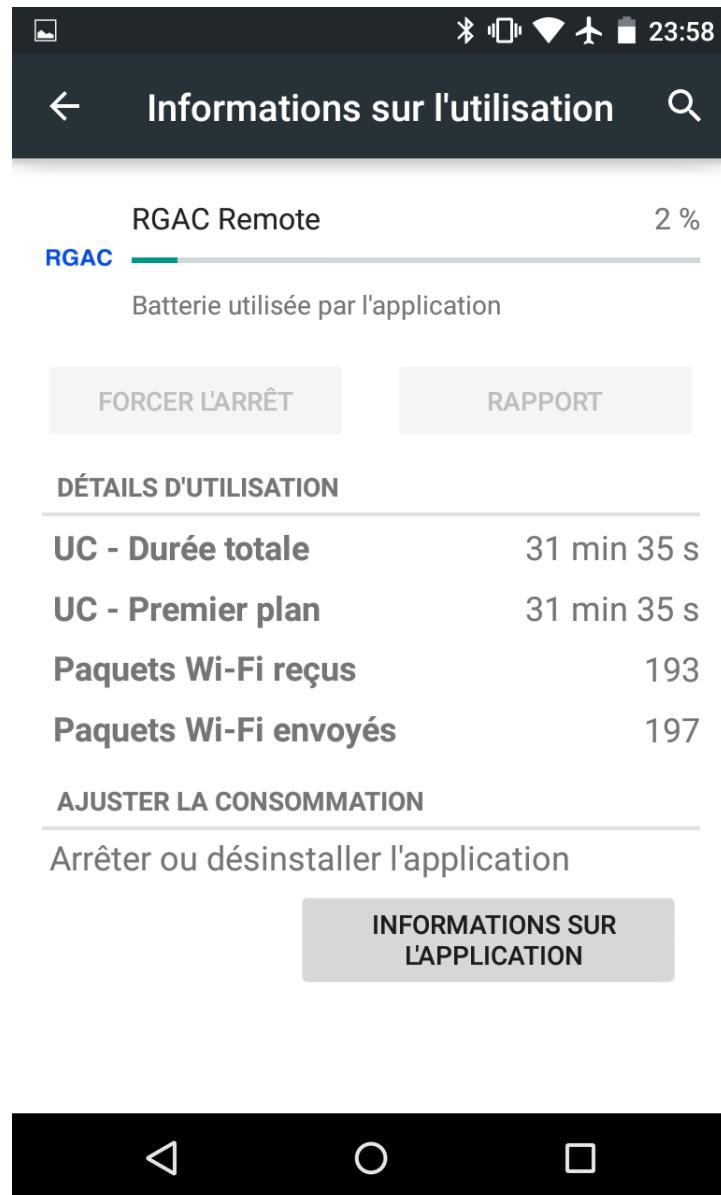


FIGURE B.16 – Informations sur la batterie (2)

Bibliographie

- [1] Api android. <https://developer.android.com/>. [Online ; accessed 04-March-2015].
- [2] Developpez.com. <http://www.developpez.com/>. [Online ; accessed 04-March-2015].
- [3] Openclassrooms. <http://openclassrooms.com/>. [Online ; accessed 04-March-2015].
- [4] Openclassrooms capteurs. <http://openclassrooms.com/courses/creez-des-applications-pour-android/les-capteurs>. [Online ; accessed 04-March-2015].
- [5] Openclassrooms socket. <http://openclassrooms.com/courses/introduction-aux-sockets-1>. [Online ; accessed 03-March-2015].
- [6] Samsung microphone. <http://developer.samsung.com/technical-doc/view.do;jsessionid=6NpqJZJWzfnhTLrSDCq1hN81xnhZKCQTfGQ9177HWYB1nt6cvQ4q!-1904197406?v=T000000086>. [Online ; accessed 03-March-2015].
- [7] Jean-Francois Lalande. Développement sous android, June 2013. <http://www.univ-orleans.fr/lifo/Members/Jean-Francois.Lalande/enseignement/android/cours-android.pdf>.
- [8] Greg Milette and Adam Stroud. *Professional Android sensor programming*. John Wiley & Sons, 2012.
- [9] Ms. Najme Zehra Naqvi, Dr. Ashwani Kumar, Aanchal Chauhan, and Kritka Sahni. Step counting using smartphone-based accelerometer, 2012. <http://www.enggjournals.com/ijcse/doc/IJCSE12-04-05-266.pdf>.
- [10] Hakob Sarukhanyan Sahak Kaghyan. Activity recognition using k-nearest neighbor algorithm on smartphone with tri-axial accelerometer, 2012. <http://www.foibg.com/ijima/vol01/ijima01-2-p06.pdf>.
- [11] Andrew Tarantola. 5 apps to turn your phone into a universal remote. 2013. <http://gizmodo.com/5982909/5-apps-to-turn-your-phone-into-a-universal-remote>.