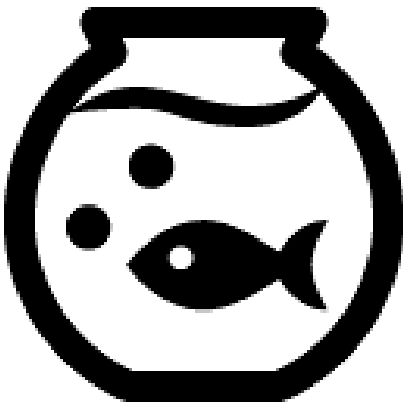
Etudiants EC : GERONA Alexis
LEGEARD Anthony
Etudiants IR : BRESSET Aymeric
CLOART Audrey

Terminale BTS SN
Année : 2016-2017

REVUE FINALE :

PROJET GESAQUA



La Salle Avignon
Frères des Écoles Chrétiennes

Table des matières

Présentation générale du projet

Expression du besoin

Il s'agit de réaliser un programme complet pour assurer le fonctionnement autonome d'un aquarium permettant de recréer artificiellement les conditions de vie initiales du poisson dans son environnement naturel.

Le projet de gestion informatisée d'aquarium consistera en un travail de développement d'une solution logicielle permettant la gestion automatisée d'un aquarium de type « eau douce » installé dans une résidence ou un lieu d'exposition par exemple.

Ce projet sera mené dans le cadre d'une demande croissante de la part des organismes aquariophiles (associations, ...) et des particuliers, qui ont besoin de pouvoir gérer leurs bacs[1] autrement que manuellement.

Le système technique devra permettre de reconstituer le plus fidèlement possible le milieu aquatique d'origine des poissons à l'aide d'un aquarium.

En effet, ces poissons peuvent provenir de milieux différents (rivières tropicales, milieu marin tropical ou tempéré….), et l'aquariophile doit les élever et les faire vivre dans les meilleures conditions possibles.

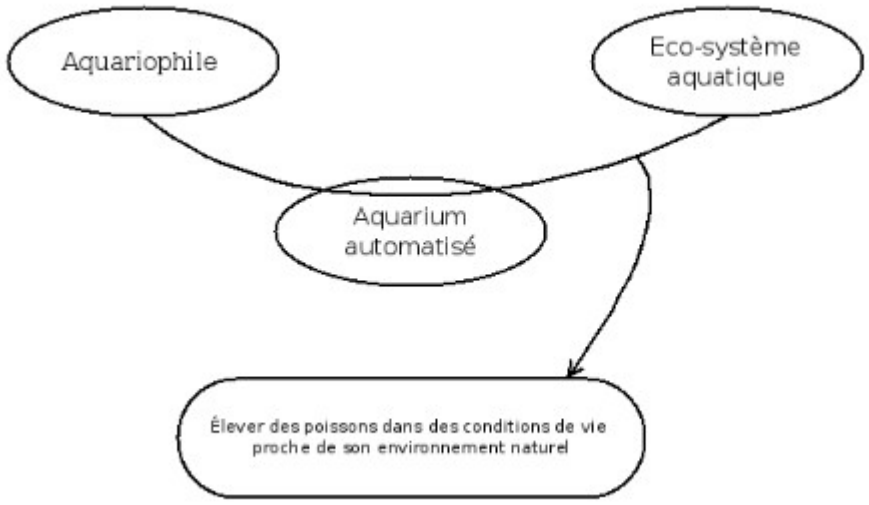


Figure 1 : Schéma du besoin

1 Bac : désigne en aquariophilie l'aquarium (cuve + eau)

Présentation du projet

Le système devra :

- réguler la température et le pH
- maintenir le niveau de l'eau
- commander automatiquement ou manuellement l'ensemble des appareils (l'éclairage, le chauffage, la ventilation, l'oxygénation, la distribution de nourriture, d'engrais liquide, de CO2, de soude, la filtration de l'eau et la pompe de remplissage)
- gérer une programmation par calendrier
- signaler et journaliser les alarmes de température, de pH, de niveau d'eau et de dépassement d'interventions
- communiquer avec l'utilisateur via un mini-écran tactile.

L'aquarium sera équipé de :

- 4 capteurs pour la mesure du pH, le niveau de l'eau, la température de l'eau et de l'air
- 9 à 11 prises 230V/50Hz pour la commande des appareils (chauffage, ventilation, éclairage, filtration, oxygénation, …)

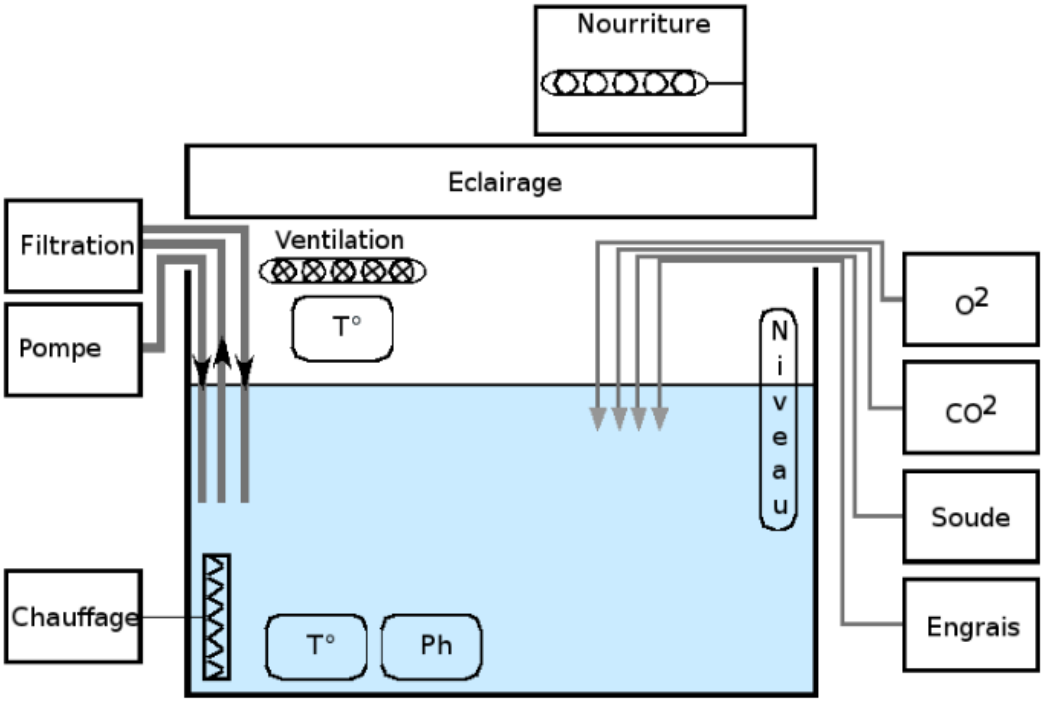


Figure 2 : Exemple d'installation

Exigences

Le système GesAqua permettra :

- le paramétrage des consignes de température, de pH et de niveau de l’eau
- la sélection du mode de gestion (automatique ou manuel)
- le choix des alarmes à surveiller (température, pH, niveau d’eau et échéance des interventions)
- le pilotage des différents modules en fonction du mode de fonctionnement, des consignes et du calendrier
- la visualisation des états, des données et des alarmes sur une tablette tactile
- l’archivage des états, des données et des alarmes dans une base de données.

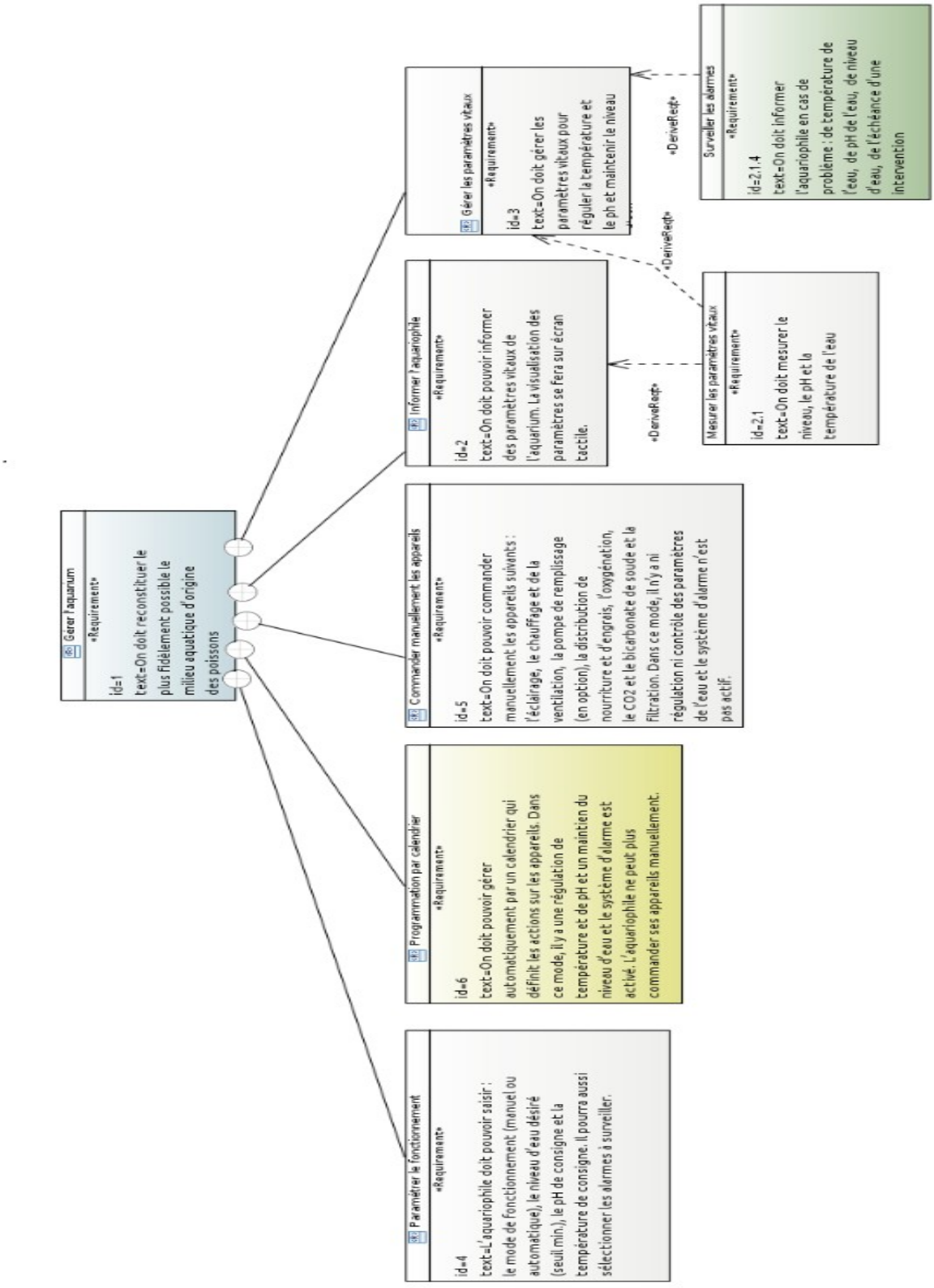


Figure 3 : diagramme des exigences 1/3

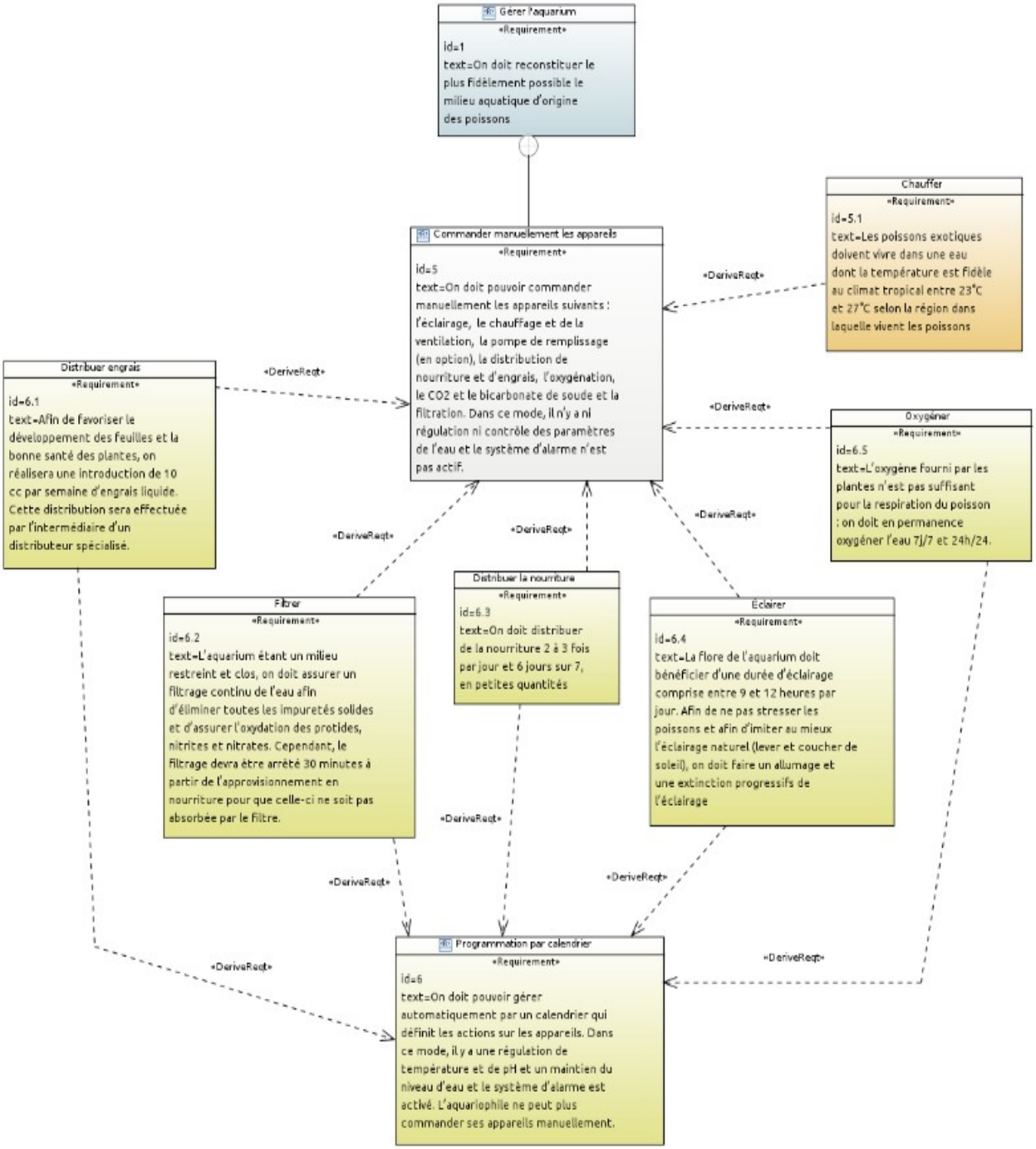


Figure 4 : diagramme des exigences 2/3

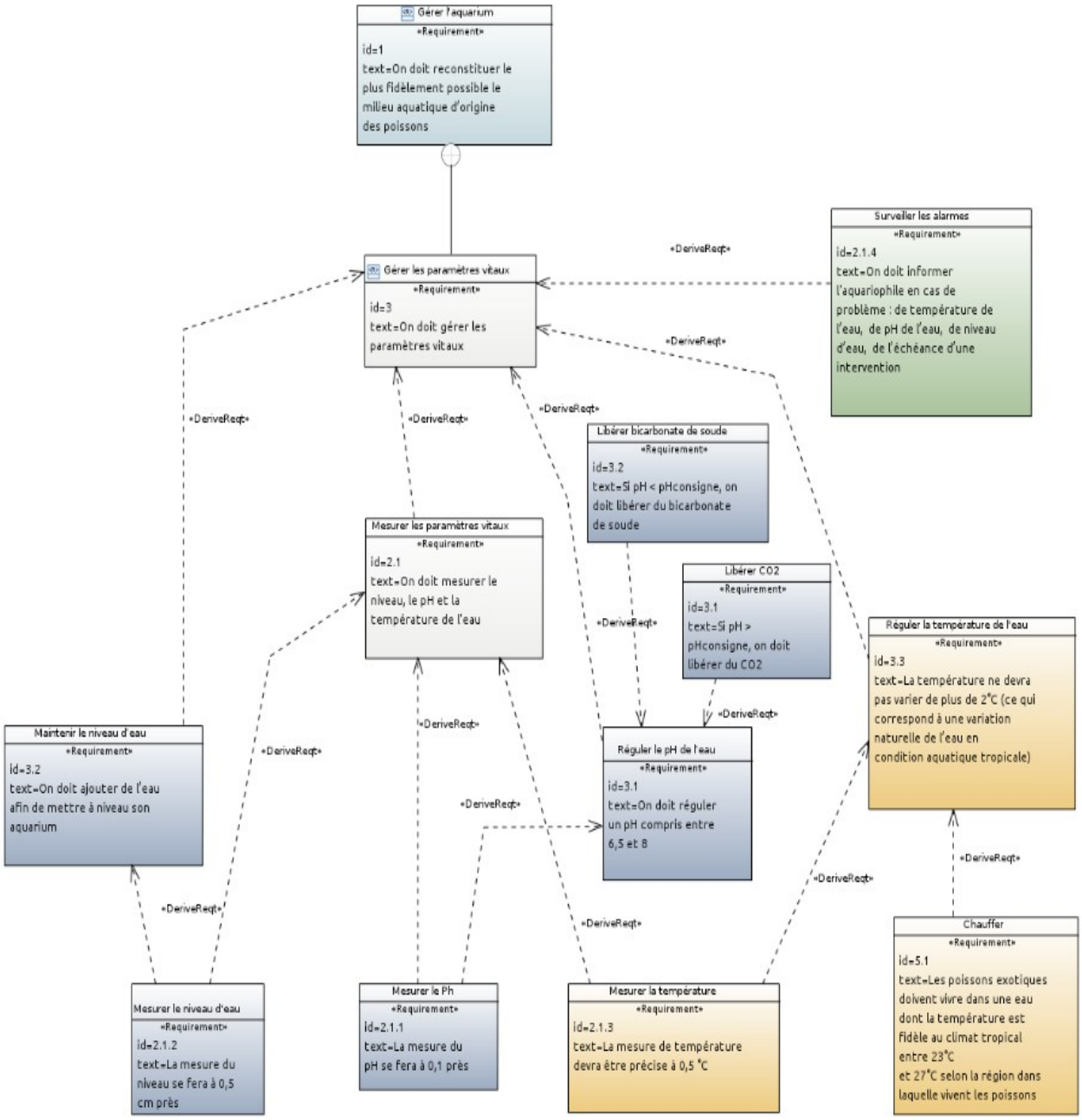


Figure 5 : diagramme des exigences 3/3

Modification du cahier des charges

- Le choix de la communication sans fil s'est porté sur le Bluetooth.
- Les alarmes du système seront dans un module indépendant et seront activées ou désactivées à la disposition du client, donc quel que soit le mode (automatique ou manuel) il y aura la possibilité d'activer ou de désactiver les alarmes.

Synoptique du système

La communication entre la carte Atmel et la tablette tactile Samsung s'effectuera via une liaison Bluetooth 4.0.

L'aquarium est équipé de :

- 2 capteurs analogiques (pH et niveau d'eau) reliés directement à la carte Atmel qui possède des convertisseurs analogiques-numériques (CAN)
- 1 capteur numérique DS18B20 (pour la température de l'eau) relié sur un bus 1-Wire
- 1 capteur numérique DS1621 (pour la température de l'air) relié sur un bus I2C
- 9 à 11 prises 230V / 50Hz reliées à des cartes relais, elles mêmes reliées à 2 circuits I2C

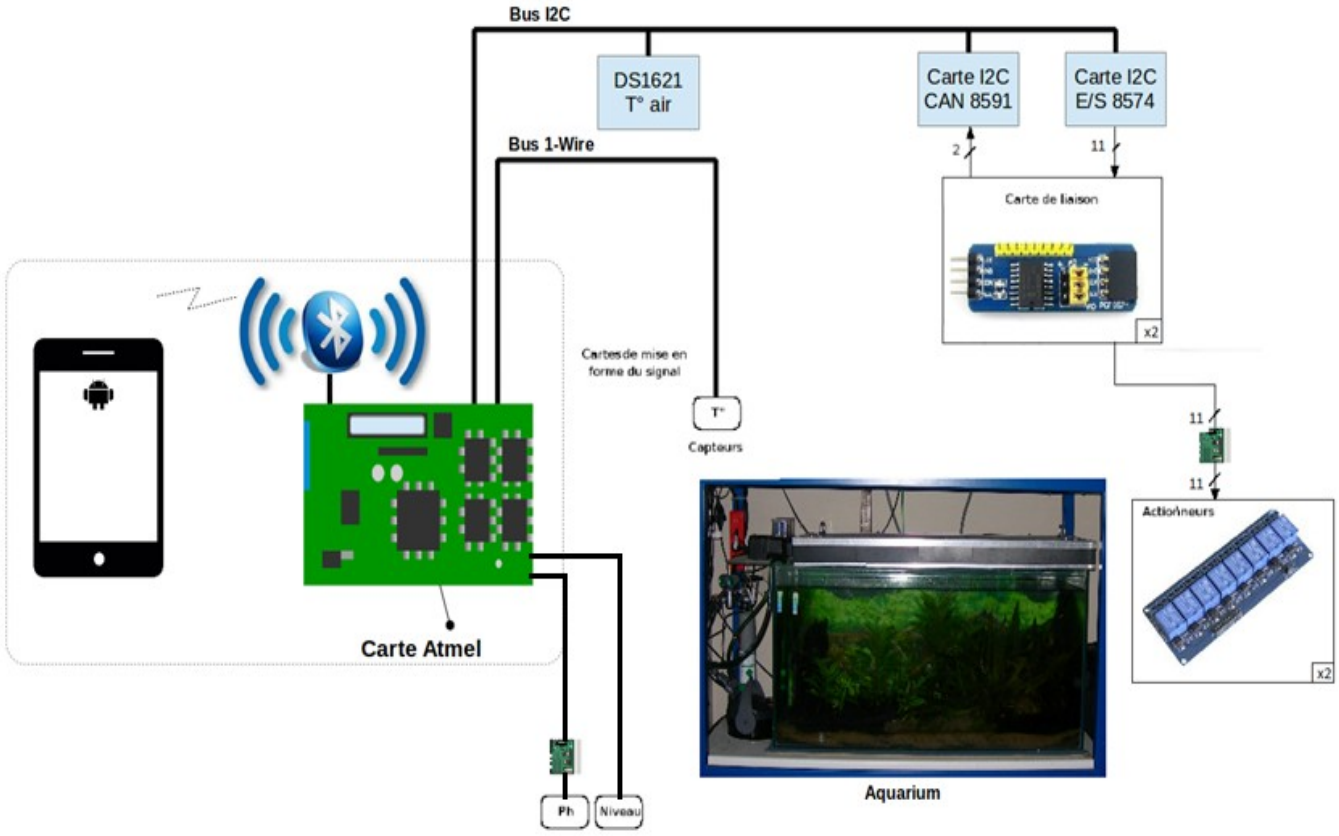


Figure 6 : Vue d'ensemble du système

Identification du travail à réaliser

Etudiants chargés du projet

Option EC :

- GERONA Alexis : étudiant 1
- LEGEARD Anthony : étudiant 2

Option IR :

- BRESSET Aymeric : étudiant 3
- CLOART Audrey : étudiant 4

Répartition des tâches entre étudiants

Etudiant 1 : EC GERONA Alexis

Module : contrôle des paramètres de l'eau

→ Régulation de la température de l'eau
→ Maintien du niveau d'eau
→ Régulation du pH de l'eau
→ Surveillance des alarmes

- Installation : appareils de chauffage et de ventilation
- Mise en œuvre :
  → capteurs de température, de niveau d'eau et de pH
  → environnement de développement
- Configuration : les esclaves I2C
- Réalisation :
  → diagrammes SysML / UML
  → code source
  → et schémas du module
- Documentation :
  → le dossier technique et les documents relatifs au module
  → un guide de mise en route et d'utilisation du module

Etudiant 2 : EC LEGEARD Anthony

Module : gestion automatisée

→ Distribution de la nourriture
→ Commande des appareils
→ Gestion d'une programmation par calendrier
→ Paramétrage du fonctionnement
→ Information de l'utilisateur
→ Communication des ordres et des données

- Installation : appareils de distribution de nourriture et d'engrais, d'oxygénation, de filtration et d'éclairage
- Mise en œuvre :

→ liaison sans fil (Bluetooth)
→ environnement de développement

- Configuration : les esclaves I2C
- Réalisation :
  → diagrammes SysML / UML
  → code source
  → et schémas du module
- Documentation :
  → le dossier technique et les documents relatifs au module
  → un guide de mise en route et d'utilisation du module

Etudiant 3 : IR BRESSET Aymeric

Module : gestion des paramètres de l'eau *en lien avec l'étudiant 1*

→ Commande des appareils par la tablette tactile
→ Signal des alarmes
→ Paramétrage du fonctionnement
→ Information de l'utilisateur
→ Archivage des données
→ Communication des ordres et des données

- Mise en œuvre : environnement de développement
- Configuration : la tablette tactile
- Réalisation :
  → diagrammes UML,
  → code source de l'application
  → IHM du module
- Documentation :
  → le dossier technique et les documents relatifs au module
  → un guide de mise en route et d'utilisation du module

Etudiant 4 : IR CLOART Audrey

Module : commande des appareils *en lien avec l'étudiant 2*

→ Commande des appareils par la tablette tactile
→ Signal des alarmes

→ Paramétrage du fonctionnement
→ Information de l'utilisateur
→ Archivage des données
→ Communication des ordres et des données

- Installation : la liaison sans fil (Bluetooth)
- Mise en œuvre :
  → l'environnement de développement
  → la base de données
- Configuration : la base de données
- Réalisation :
  → diagrammes SysML / UML
  → code source de l'application
  → IHM du module
- Documentation :
  → le dossier technique et les documents relatifs au module
  → un guide de mise en route et d'utilisation du module

Etude préliminaire

Etude d'Android Studio (IDE)

Android Studio est un environnement de développement intégré conçu pour développer des applications Android. Il permet principalement d'éditer les fichiers Java ainsi que les fichiers de configuration d'une application Android.

L'étude d'Android Studio a consisté à écrire une première application Android afin de prendre en main les fonctionnalités de base du logiciel et ainsi de poursuivre le projet plus rapidement et plus efficacement par la suite.

Figure 7 : Logo d'Android Studio

Architecture du système

Voici le bloc principal ainsi que la hiérarchie des blocs qui le composent, qu'ils soient logiciels ou matériels :

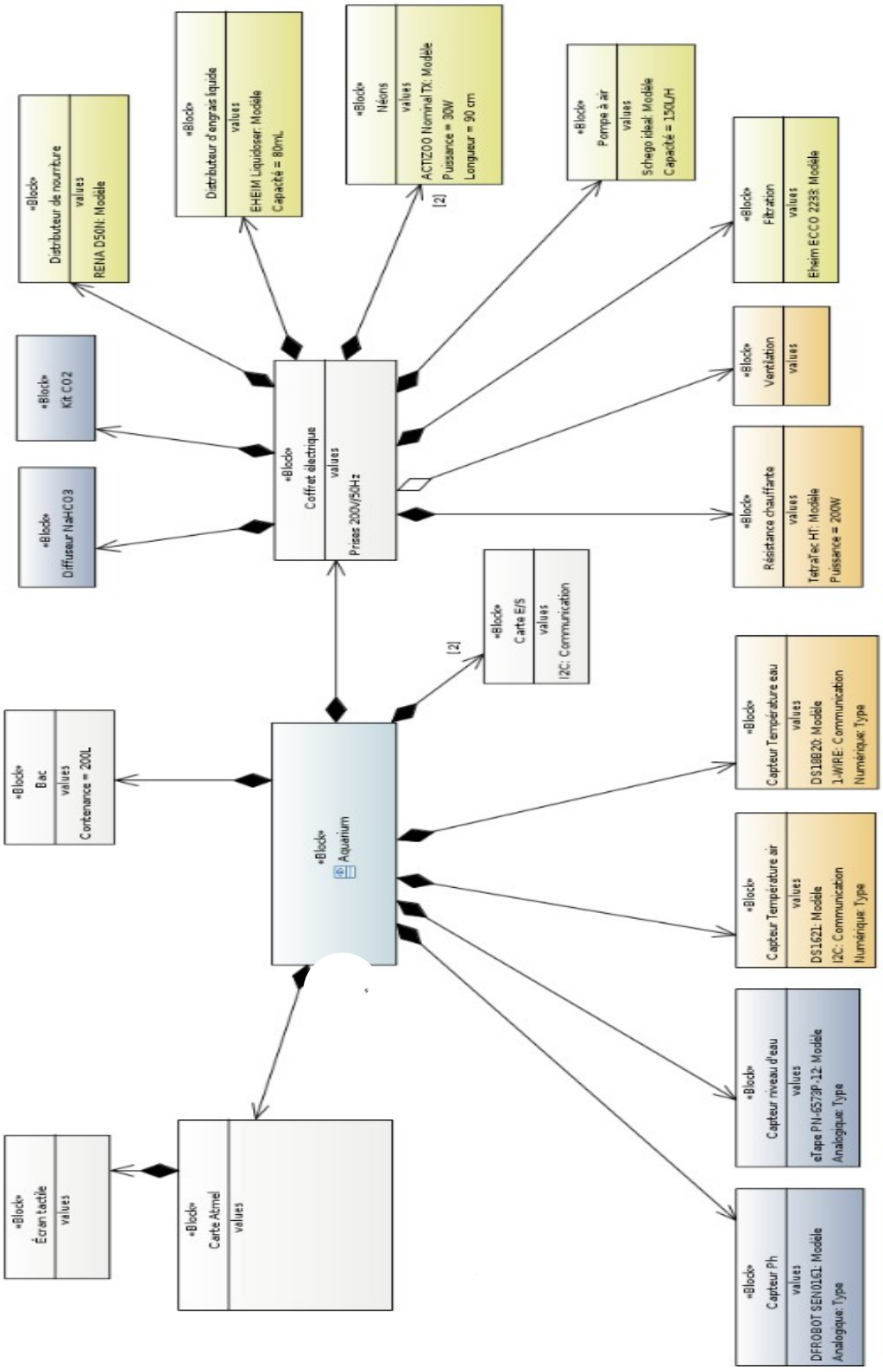
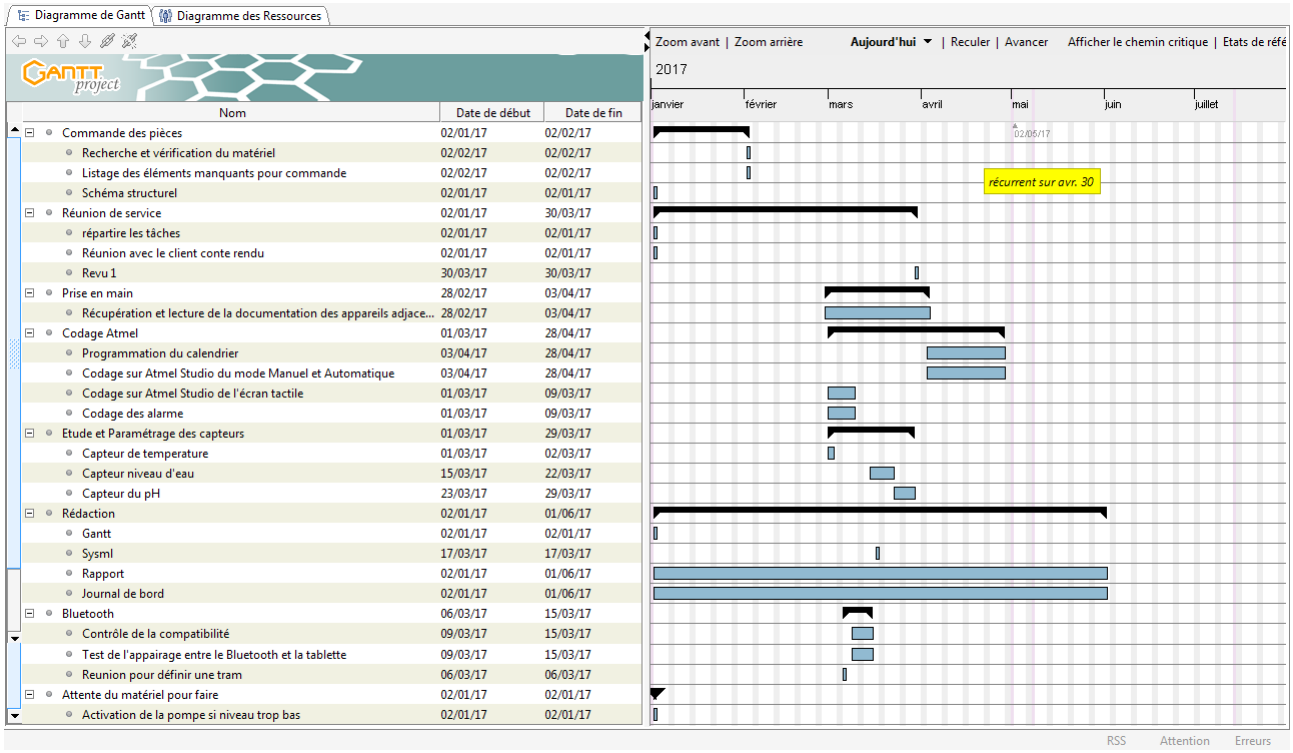
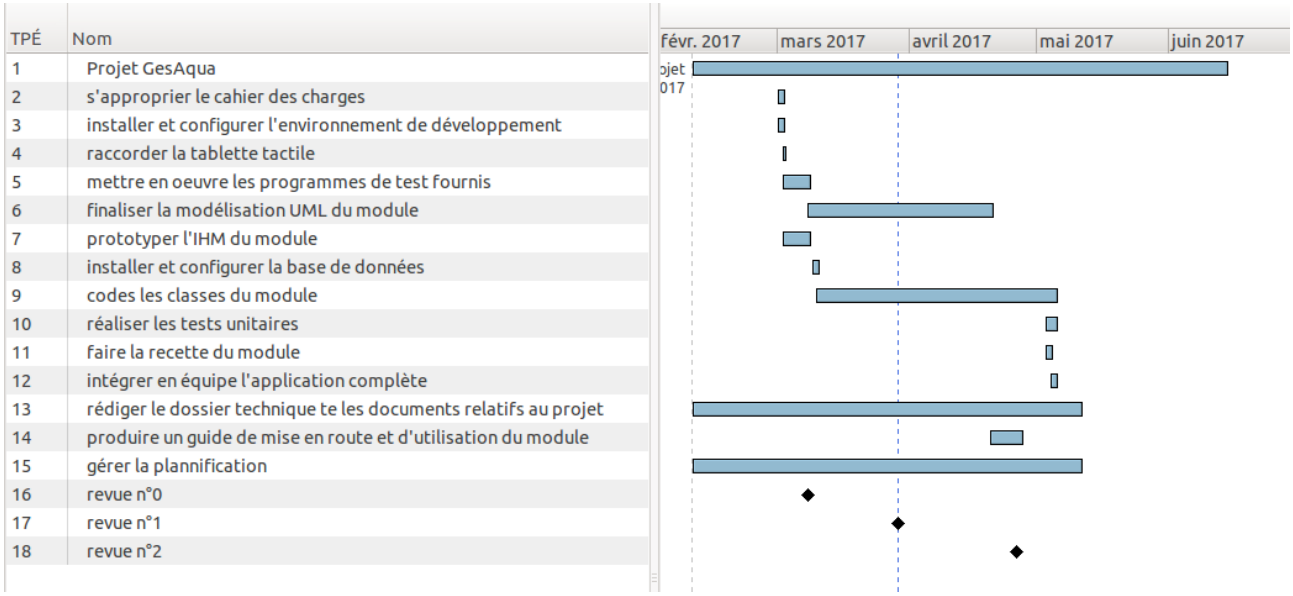


Figure 8 : Diagramme de blocs SysML

Planification des tâches



Nom	Date de début	Date de fin
Commande des pièces	02/01/17	02/02/17
Recherche et vérification du matériel	02/02/17	02/02/17
Listage des éléments manquants pour commande	02/02/17	02/02/17
Schéma structurel	02/01/17	02/01/17
Réunion de service	02/01/17	30/03/17
répartire les tâches	02/01/17	02/01/17
Réunion avec le client conte rendu	02/01/17	02/01/17
Revu 1	30/03/17	30/03/17
Prise en main	28/02/17	03/04/17
Récupération et lecture de la documentation des appareils adjace...	28/02/17	03/04/17
Codage Atmel	01/03/17	28/04/17
Programmation du calendrier	03/04/17	28/04/17
Codage sur Atmel Studio du mode Manuel et Automatique	03/04/17	28/04/17
Codage sur Atmel Studio de l'écran tactile	01/03/17	09/03/17
Codage des alarme	01/03/17	09/03/17
Etude et Paramétrage des capteurs	01/03/17	29/03/17
Capteur de temperature	01/03/17	02/03/17
Capteur niveau d'eau	15/03/17	22/03/17
Capteur du pH	23/03/17	29/03/17
Rédaction	02/01/17	01/06/17
Gantt	02/01/17	02/01/17
Sysml	17/03/17	17/03/17
Rapport	02/01/17	01/06/17
Journal de bord	02/01/17	01/06/17
Bluetooth	06/03/17	15/03/17
Contrôle de la compatibilité	09/03/17	15/03/17
Test de l'appairage entre le Bluetooth et la tablette	09/03/17	15/03/17
Reunion pour définir une tram	06/03/17	06/03/17
Attente du matériel pour faire	02/01/17	02/01/17
Activation de la pompe si niveau trop bas	02/01/17	02/01/17



TPÉ	Nom
1	Projet GesAqua
2	s'approprier le cahier des charges
3	installer et configurer l'environnement de développement
4	raccorder la tablette tactile
5	mettre en oeuvre les programmes de test fournis
6	finaliser la modélisation UML du module
7	prototyper l'IHM du module
8	installer et configurer la base de données
9	codes les classes du module
10	réaliser les tests unitaires
11	faire la recette du module
12	intégrer en équipe l'application complète
13	rédiger le dossier technique te les documents relatifs au projet
14	produire un guide de mise en route et d'utilisation du module
15	gérer la plannification
16	revue n°0
17	revue n°1
18	revue n°2

Déploiement du système

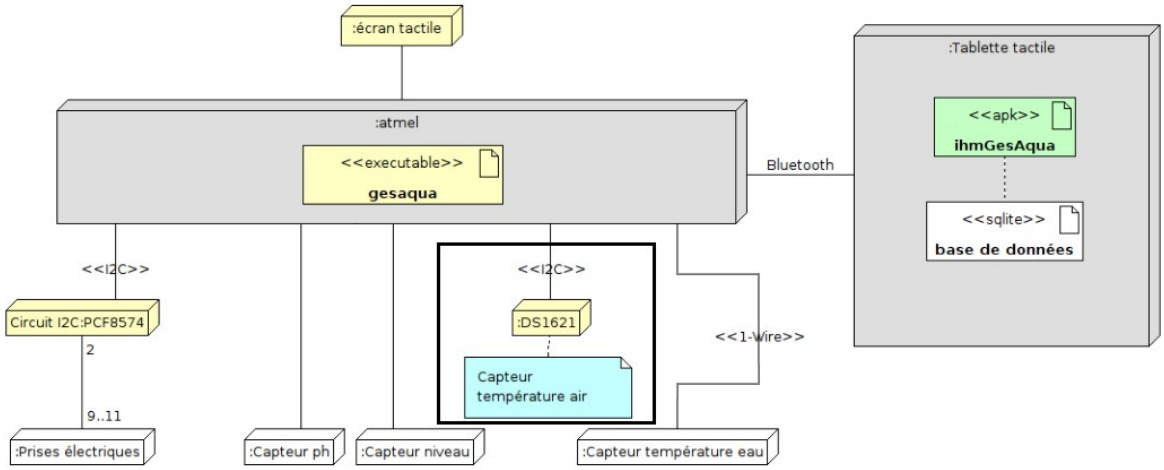


Figure 9 : diagramme de déploiement du système

Prototypage et maquette de l'IHM

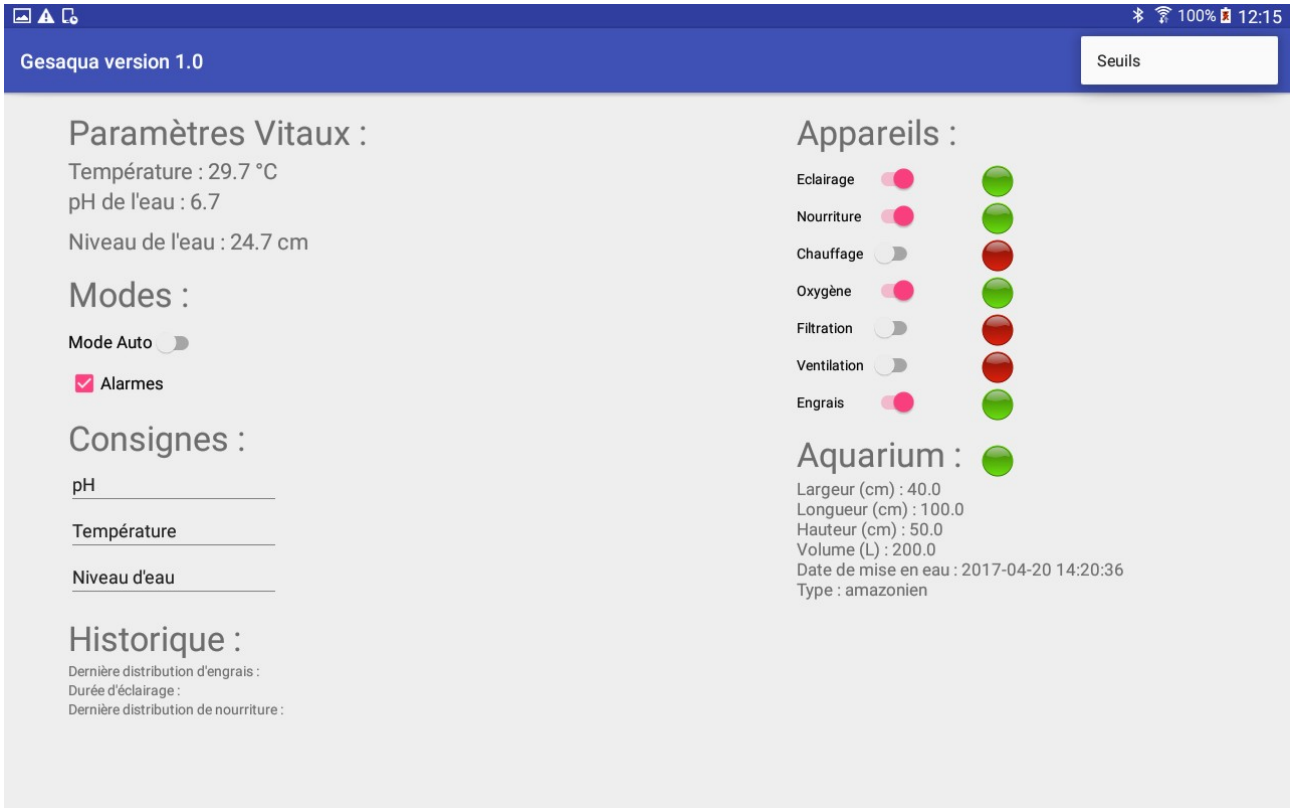


Figure : Capture d'écran de la page principale de l'IHM

Sur cette IHM, on voit que l'aquariophile peut :

- saisir le mode de fonctionnement : automatique ou manuel
- visualiser les états des appareils de l'aquarium
- visualiser les paramètres de l'eau
- paramétrer les consignes de son choix
- visualiser les mesures effectuées (température, pH et niveau de l'eau)
- les caractéristiques de son bac (cuve + eau)
- possibilité de commander les appareils

Il existe un menu pour aller vers la seconde page de l'application : `Seuils`.
Cette page permettra à l'utilisateur d'entrer les seuils qui ne devront pas être dépassés par le système.[2]

2 Cf Annexes pour le Guide de Mise en Route et d'Utilisation

Figure : Capture d'écran de la page Seuils de l'IHM

Plans des tests de validation

Etudiant 1 : GERONA Alexis

DESCRIPTION	OUI	NON
La température de l'eau est mesurée périodiquement et affichée avec son unité		
La consigne de régulation de la température est réglable à partir de l'écran tactile		
Le pH de l'eau est mesuré périodiquement et affiché		
La consigne de régulation du pH est réglable à partir de l'écran tactile		
Le niveau de l'eau est mesuré périodiquement et affiché		
Les seuils min. et max. du niveau d'eau sont réglables à partir de l'écran tactile		
L'état des commandes des appareils est visible sur l'écran tactile		
Les appareils sont pilotables manuellement sauf si le mode automatique est activé		
En mode automatique, la régulation de la température de l'eau est effective et prend en compte la valeur de la consigne		
En mode automatique, la régulation du pH de l'eau est effective et prend en compte la valeur de la consigne		
En mode automatique, le maintien du niveau de l'eau est effectif et prend en compte la valeur des seuils		

Etudiant 2 : LEGEARD Anthony

DESCRIPTION	OUI	NON
Le choix du mode manuel ou automatiquee est possible et visible sur l'écran tactile		
En mode automatique, la gestion des appareils programmés est effective		
L'état des commandes des appareils est visible sur l'écran tactile		
Les appareils sont pilotables manuellement sauf si le mode automatique est activé		
Les données et les alarmes sont transmises par liaison sans fil		
Les ordres sont reçus par liaison sans fil et traités		

Etudiant 3 : BRESSET Aymeric

DESCRIPTION	OUI	NON
Les données et les alarmes associées au module sont affichées sur la tablette tactile		
La commande des appareils associés au module est possible à partir de la tablette tactile		
Les mesures moyennées sur une heure sont enregistrées périodiquement dans la base de données		
Les nouvelles consignes et seuils min. et max. sont enregistrés dans la base de données		
Les données de l'aquarium (type, mise en eau, dimensions, volume, nombe et taille totale des poissons) sont récupérées à partir de la base de données et affichées avec leurs unités dans l'IHM		
Les prochaines échéances des interventions (analyse de l'eau, intervention, entretien) sont récupérées de la base de données et notifiées si nécessaire		
La sélection des alarmes à surveiller est possible à partir de l'IHM		
Les nouvelles sélections des alarmes à surveiller sont enregistrées dans la base de données		

Etudiant 4 : CLOART Audrey

DESCRIPTION	OUI	NON
Les données et les alarmes associées au module sont affichées sur la tablette tactile		
La commande des appareils associés au module est possible à partir de la tablette tactile		
Le paramétrage du mode automatisé est réalisable		
L'affichage de la durée actuelle d'éclairage en minutes est réelle dans l'IHM		
L'affichage de la dernière distribution de nourriture est visible sur l'écran tactile		
L'affichage de la dernière distribution d'engrais est visible sur l'écran tactile		

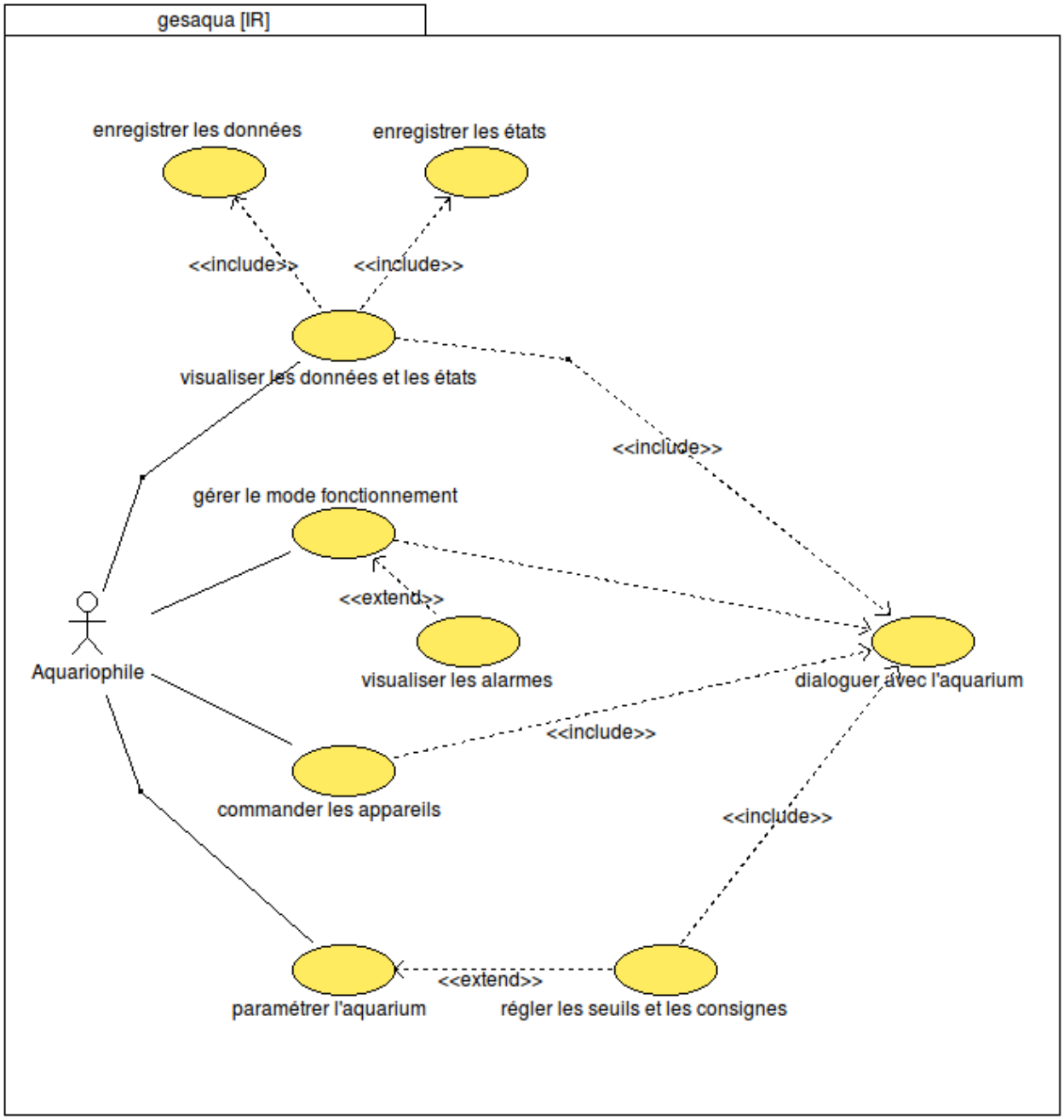
Les cas d'utilisation

L'acteur humain de ce système est l'aquariophile.

Il visualise sur l'écran tactile l'ensemble des mesures effectuées (température de l'eau, pH et niveau d'eau), les caractéristiques de son bac, l'état des appareils de son aquarium et les alarmes de surveillance de dépassement.

Il a la possibilité de paramétrer les consignes (température de l'eau, pH et niveau d'eau) à respecter en mode automatique et les alarmes à surveiller.

Il peut commander manuellement à partir de l'écran tactile l'ensemble de ses appareils ou démarrer une gestion automatique de son aquarium, notamment en cas d'absence.


gesaqua [IR]
enregistrer les données
enregistrer les états
<<include>>
<<include>>
visualiser les données et les états
<<include>>
gérer le mode fonctionnement
<<extend>>
visualiser les alarmes
Aquariophile
dialoguer avec l'aquarium
<<include>>
commander les appareils
<<include>>
paramétrer l'aquarium
<<extend>>
régler les seuils et les consignes

Contraintes fonctionnelles et techniques

Matérielles

- carte Atmel SAM4S
- tablette tactile Samsung Galaxy Tab 4
- bac 200 L (100 cm x 40 cm x 50 cm)
- tableau électrique de 9 prises 230V / 50Hz
- résistance chauffante TetraTec HT 200 W
- bicarbonate de soude
- CO2
- pompe de remplissage
- neons 2 x ACTIZOO Nominal TX 30W (90 cm)
- distributeur d'engrais lisuide EHEIM Liquidoser
- distributeur de nourriture Rena D50N
- capteur de niveau d'eau
- capteur de pH DFROBOT Réf SEN0161
- capteur de température 1-Wire DS18B20
- 2 cartes relais
- 2 cartes i2C

Logicielles

- Android Studio 2.3
- Système d'exploitation de la tablette : Android 4.4.2 (KitKat)
- java 1.8.0
- SDK Android API 25 : Android 7.1.1 (Nougat)
- svn, version 1.6.17 (r1128011)
- doxygen 1.7.6.1
- bouml Bouml 4.23
- Gestionnaire de projet : Planner 0.14.5
- SGBDR[3] : SQLite3

3 Système de Gestion de Base de Données Relationnelles

Seuils pour la survie des poissons

Afin de recréer les conditions initiales de vie des poissons de la façon la plus fidèle possible à leur milieu naturel, il est nécessaire de déterminer certains seuils à ne pas dépasser :

→ la température de l'eau doit se situer entre 23°C et 27°C
→ le pH de l'eau doit se situer entre 6.5 et 8
→ le niveau d'eau doit être suffisant pour assurer leur survie

Les scénarios des cas d'utilisation

Cas d'utilisation 1 : Visualiser les états et données

On doit pouvoir informer l'utilisateur des paramètres vitaux de l'aquarium, l'état des appareils, ainsi que les caractéristiques du bac Cette visualisation se fait sur écran tactile.

Cas d'utilisation 2 : Gérer le mode de fonctionnement

On doit pouvoir gérer l'aquarium automatiquement par un calendrier qui définit les actions sur les appareils. Il y a une régulation de température et de pH, ainsi qu'un maintien du niveau d'eau. L'aquariophile ne peut plus commander ses appareils manuellement.

Cas d'utilisation 3 : Commander les appareils

On doit pouvoir commander manuellement les appareils suivants : l'éclairage, le chauffage et la ventilation, la pompe de remplissage, la distribution de nourriture et d'engrais, l'oxygénation, le CO2 et le bicarbonate de soude, et la filtration. Dans ce mode, il n'y a ni régulation ni contrôle des paramètres de l'eau.

Cas d'utilisation 4 : Paramétrer l'aquarium

L'aquariophile doit pouvoir saisir : le niveau d'eau désiré (seuil minimal), le pH de consigne ainsi que la température de consigne. Il pourra aussi sélectionner les alarmes à surveiller.

Cas d'utilisation 5 : Régler les seuils et les consignes

On doit pouvoir prédéfinir des seuils à ne pas dépasser pour réguler la température et le pH de l'eau, ainsi que de maintenir le niveau d'eau.

Partie Personnelle Bresset Aymeric

Table des matières

Objectifs

Mon travail consiste à développer une application pour Tablette tactile (sous Android) permettant la supervision et le contrôle à distance d'un aquarium de type « eau douce » installé dans une résidence ou lieu d'exposition.

La tablette tactile communiquera avec le système embarqué assurant la gestion automatique de l'aquarium via une liaison sans fil *Bluetooth*.

Pré-requis : Android Studio

Installation

L'application à réaliser étant destinée à des tablettes Android, il me faudra un environnement de développement pour ce système : Android Studio. Cet IDE () intègre le kit de développement Android SDK.

Il m'a fallu l'installer sur mon poste de développement :

- installation du Java SDK :

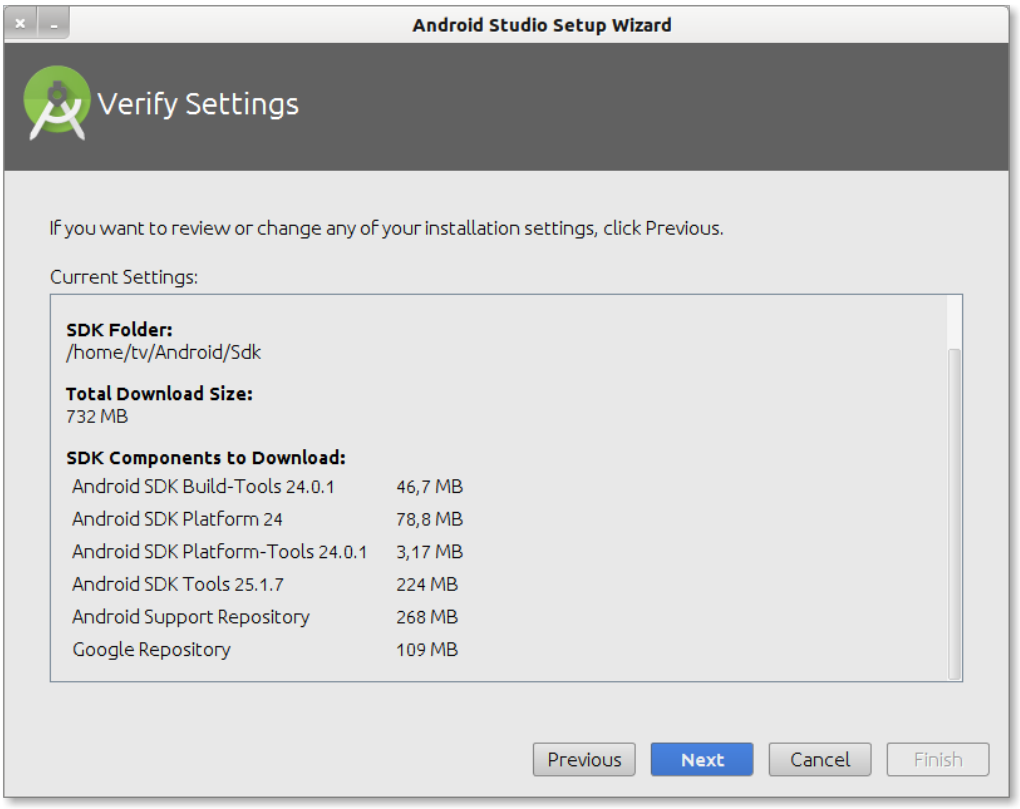
```
$ sudo cp jdk-8u102-linux-x64.tar.gz /usr/local
$ cd /usr/local/
$ sudo tar zxvf jdk-8u102-linux-x64.tar.gz
$ sudo rm jdk-8u102-linux-x64.tar.gz
$ vim $HOME/.bashrc
export PATH=/usr/local/jdk1.8.0_102/bin:$PATH
$ source $HOME/.bashrc
```

- installation d'Android Studio :

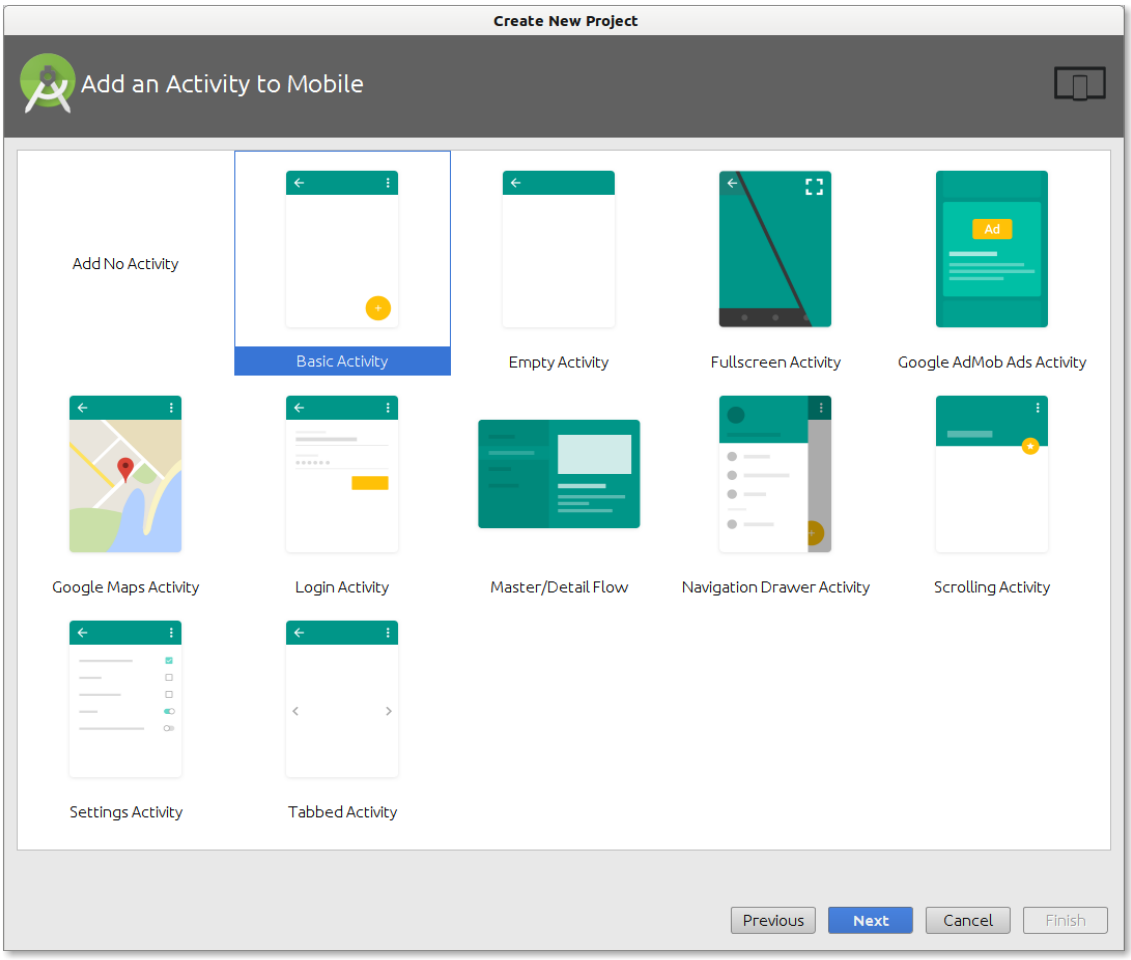
```
$ sudo cp android-studio-ide-143.3101438-linux.zip /usr/local
$ cd /usr/local/
$ sudo unzip android-studio-ide-143.3101438-linux.zip
$ sudo rm android-studio-ide-143.3101438-linux.zip
$ vim $HOME/.bashrc
```

```
export PATH=/usr/local/jdk1.8.0_102/bin:
$PATH:/usr/local/android-studio/bin:
$HOME/Android/Sdk/platform-tools:$HOME/Android/Sdk/tools

$ source $HOME/.bashrc

$ studio.sh
```

Android Studio permet de programmer sur la plupart des versions d'Android avec une génération automatique du squelette d'une application. Il contient aussi des émulateurs qui permettront de tester le code directement sur l'ordinateur et des prévisualisations graphiques pour voir le rendu réel sur divers appareils (smartphones, tablettes).



Lors de la création d'une application sur Android Studio, on nous propose divers types d'IHM (ie. une activité, voir plus loin) pré-construites :

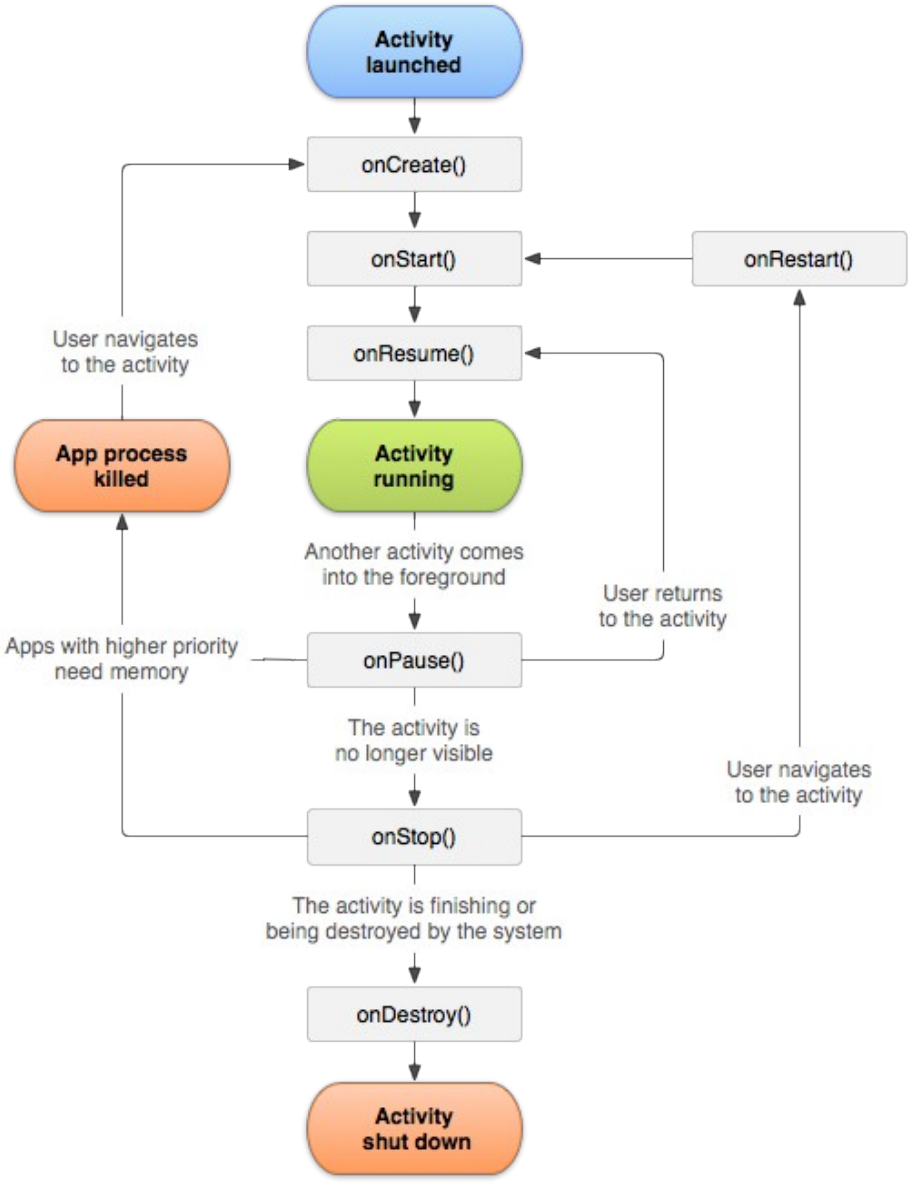


Selon l'activité choisie, il y aura des parties de code déjà fournies ainsi qu'une interface graphique de base. Cela permet une grande personnalisation de l'application et une simplification du travail lors de la création de l'application. Dans notre cas, les fonctionnalités proposées par les diverses activités possible ne nous concernant pas, nous avons du choisir une activité vide.

Les activités

Une **activité** sur Android est une interface graphique définie par l'écran et ce qu'il y a d'affiché dessus. C'est donc une « page de l'application ». Par exemple dans une application on peut avoir une activité pour composer un numéro et une autre activité pour accéder aux contact. Chaque activité peut fonctionner de manière autonome ou avec en interagissant avec les autres activités.

Ce diagramme montre le cycle d'exécution d'une activité.



Une activité est composée de plusieurs méthodes :

- `onCreate()` appelée lors de la création de l'activité : elle permet d'initialiser tous les éléments graphiques et différentes données ;

- `onStart()` appelée lors du lancement de l'activité : c'est le moment ou l'activité est lancée ;
- `onResume()` appelée lorsque l'activité sort du `onPause()` pour reprendre son exécution ;
- `onPause()` appelée lors de l'ouverture d'une autre activité qui met l'activité principale en arriéré plan : elle sera généralement appelée pour sauvegarder l'état actuel de l'activité ;
- `onStop()` appelée lorsque l'activité rend les ressources qu'elle n'utilise plus et se prépare à se faire détruire : cette fonction est appelée lors de la fin de l'exécution du code ou lors de la fermeture de l'application ;
- `onDestroy()` appelée lorsque l'activité est en cours de destruction et libère les dernières ressources qu'elle possède : l'application à fermé les activités restantes qu'elle possédait et rend la mémoire lui restant : l'application se ferme ensuite.

Les layouts

L'interface Homme Machine est générée à partir de plusieurs fichiers dont tous les fichiers contenus dans le dossier « res » du projet.

Il contient tous les éléments graphiques de l'application notamment les menus et les *layouts*.

Les *layouts* sont une mise en page permettant l'organisation de l'interface graphique. On peut soit les déclarer dans un fichier XML (langage de balisage) qui permet de voir avec la simulation graphique d'Android Studio comment se disposent les éléments déclarés.

On peut aussi les déclarer directement dans une activité et les modifier directement en dehors des fichiers XML.

La ligne : `tools:showIn="@layout/activity_main">`

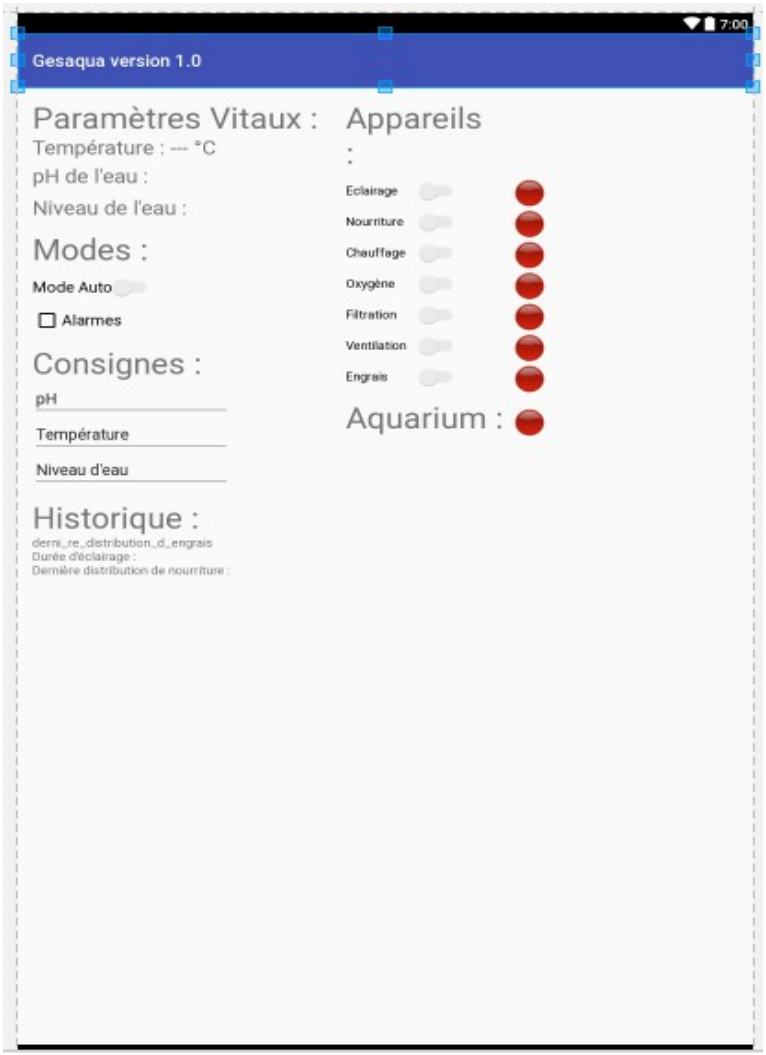
permet d'afficher une mise en page dans une autre mise en page. Dans ce cas, on affiche l'interface de content_main dans *activity_main*. Ce qui permet d'afficher une barre de menu bleue avec marqué « Gesaqua version 1.0 ».

Voici un exemple de fichier de la *layout* activity_main.xml qui contient les éléments d'affichage notamment une t*oolbar* ou barre d'outils qui permet dans notre cas d'afficher le message cité précédemment.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    android:windowSoftInputMode="stateHidden"
    tools:context="com.example.cloart.gesaqua.MainActivity">
```

```
    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">
        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay"
            app:title="Gesaqua version 1.0"/>
    </android.support.design.widget.AppBarLayout>
    <include layout="@layout/content_main" />
</android.support.design.widget.CoordinatorLayout>
```

On obtient la barre en bleu seléctionnée :



Les *widgets*

Les *widgets* ou objet graphique en français sont des objets de personnalisation d'une application.

Il y en a de nombreux ce qui permet une bonne diversité. On peut les placer dans des *layouts* pour gérer leurs emplacement graphique.

Les *widgets* les plus utilisés sont les *TextView* qui permettent d'afficher un texte défini que l'on peut modifier via le code du fichier XML ou dans l'activité directement en récupérant l'Id du *widget* pour ensuite le modifier. Dans ce cas là, on retrouve le *textViewTempérature* qui affiche la température pour modifier le texte qu'il contient avec setText(« ... »).

Les TextView

```
protected void afficherTemperature()
{
    ((TextView) findViewById(R.id.textViewTemperature)).setText("Température : "
+ temperature + " °C");
}
```

Niveau graphique, un *TextView* ressemble à :

Chaque *TextView* est caractérisé par son id, son placement et ce qu'il affiche. On peut modifier son affichage via le fichier xml ou bien dans l'activité associée avec la commande faite au dessus.

Les Switch

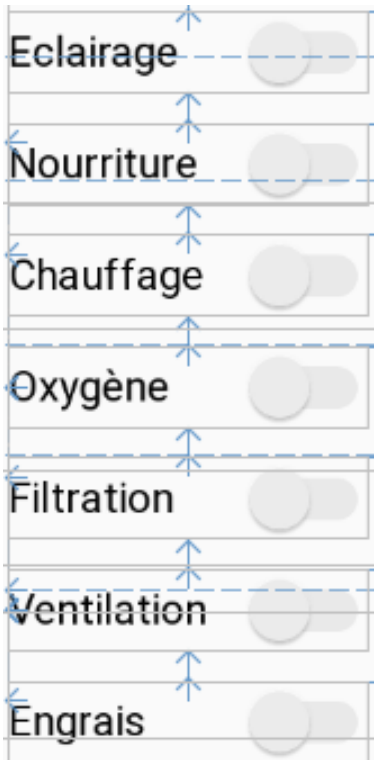
Un *Switch* est un bouton poussoir ou un bouton à 2 états : activé ou désactivé. Les *Switch* permettent donc d'activer ou désactiver un état et permettre ainsi de mettre en marche ou pas un appareil (chauffage , ventilation).

Voici la déclaration d'un *Switch* dans le fichier content_main.xml :

```
<Switch
    android:id="@+id/SwitchChauffage"
    android:layout_width="120dp"
    android:layout_height="wrap_content"
    android:layout_alignStart="@+id/SwitchVentilation"
    android:layout_below="@+id/SwitchNourriture"
    android:layout_marginRight="150dp"
    android:layout_marginTop="10dp"
    android:height="0px"
    android:text="@string/chauffage"/>
```

Dans l'ordre de la déclaration, on donne l'id du *Switch* puis on déclare ses coordonnées et enfin son texte à afficher.

Voici le rendu graphique des *Switch* de l'application via Android Studio *:*



On peut aussi vérifier l'état d'un *Switch* pour faire une action selon l'état à partir de l'activité principale :

```
private void gererInterrupteurChauffage() {
    final Switch chauffage = (Switch) findViewById(R.id.SwitchChauffage);
```

```
    chauffage.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
        /**  permet de faire un bouton switch en créant un CompoundButton et de
faire une écoute des signaux de changement.
         *   Tout est inclu dans la fonction setOnCheckedChangeListener.
         */
        public void onCheckedChanged(CompoundButton buttonView, boolean Check) {
            /* Verifie si le bouton est checké ou pas*/
            if (Check) {
                commanderAppareil(NOM_APPAREIL_CHAUFFAGE, 1);
            } else {
                commanderAppareil(NOM_APPAREIL_CHAUFFAGE, 0);
            }
        }
    });
}
```

Dans ce cas là, on retrouve le *Switch* du chauffage puis on lui attribut un bouton qui vérifiera l'état du *Switch* : Si il est touché donc activé, on appellera une fonction sinon on appelle la même mais on lui donne des arguments différents:

```
if (Check) {
    commanderAppareil(NOM_APPAREIL_CHAUFFAGE, 1);
} else {
    commanderAppareil(NOM_APPAREIL_CHAUFFAGE, 0);
}
```

La fonction commanderAppareil récupère le nom de l'appareil et l'état puis fait une trame via la fonction fabriquerTrameCommandeAppareil pour ensuite envoyer la modification de l'état à l'aquarium via la fonction envoyer(trame) de la classe peripheriqueBluetooth.

```
private void commanderAppareil(String nom, int etat) {
    if(!aquariumConnecte)
        return;
    Appareil appareil = appareils.getAppareil(nom);
    // fabrique la trame de commande et l'envoie à l'aquarium
    String trame = fabriquerTrameCommandeAppareils(appareil.getIdAppareil(),
etat);
    peripheriqueBluetooth.envoyer(trame);
    if(etat == 1) {
        Toast check = Toast.makeText(getApplicationContext(), nom + " activé",
Toast.LENGTH_SHORT);
        check.show();
    }
    else {
        Toast check = Toast.makeText(getApplicationContext(), nom + "
```

```
désactivé", Toast.LENGTH_SHORT);
        check.show();
    }
}
```

La trame envoyée sera étudiée dans un chapitre suivant.

Les boutons (*buttons*)

Les boutons sont aussi des outils essentiels pour une interface graphique. Définit aussi dans le fichier xml, ils permettent de faire des actions lorsqu'on appuie dessus :

```
<Button
    android:id="@+id/buttonValider"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Valider"
    android:onClick="Valider"
    tools:ignore="HardcodedText,RelativeOverlap"
    android:layout_alignParentTop="true"
    android:layout_toStartOf="@+id/buttonRetour"
    android:layout_marginEnd="27dp" />
```

La ligne suivante permet d'appeler une fonction lorsqu'on appuie sur le bouton :

```
android:onClick="Valider"
```

Les *EditTexts*

Les *EditTexts* sont des zone d'écriture pour interagir avec l'utilisateur. Ils peuvent avoir une zone de texte avec une valeur dedans qui sera remplacée lors de la saisie par l'utilisateur. On les définis dans le fichier xml de l'activité concernée :

```
<EditText
    android:id="@+id/editTextTemp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="text"
    android:text="@string/temp_rature"
    tools:ignore="LabelFor" />
```

La ligne contenant *imputType* permet de définir le type de données pouvant être écrite dedans. Dans ce cas, tous les textes sont acceptés mais on peut aussi remplacer pour autoriser une saisie de données numériques seulement en remplaçant « text » par « numberDecimal ».

Le problème avec les *EditText* est que lors du démarrage de l'application ou de l'activité, cela prend immédiatement le « focus » de l'écran tactile (c.a.d qu'on se retrouve dés le début avec le clavier

numérique en marche prêt à taper sur les *EditText.*

Il faut donc rajouter une *View* (vue) qu'on va définir pour lui attribuer le « *focus* » et ainsi éviter d'avoir le clavier d'activé dès le début.

Les Views (vues)

Les *View* sont les objets de base d'une interface graphique Android. Tous les autres objets héritent des *Views* ce qui permet de les afficher.

Voici la déclaration d'une *View* presque invisible qui prend automatiquement le focus de l'écran tactile et qui ne fait rien lors du focus. Cela permet d'éviter le problème rencontré plus tôt.

```
<View
    android:id="@+id/focus_thief"
    android:layout_width="1dp"
    android:layout_height="1dp"
    android:focusable="true"
    android:focusableInTouchMode="true" />
```

Les MenuItems (menus d'objets)

Les *MenuItem* sont des menus permettant de faire des actions selon le choix de l'utilisateur.

Voici la déclaration d'un menu :

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.example.cloart.gesaqua.MainActivity">
    <item
        android:id="@+id/parametres"
        android:orderInCategory="1"
        android:title="@string/parametres" />
</menu>
```

Selon le choix de l'utilisateur, cela va faire différentes actions.

On redéfinit une méthode pour faire « enfler » le menu c'est à dire pour qu'il apparaisse et affiche la liste des possibilités (dans ce cas les seuils).

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    /* fait enfler le menu et rajoute des objets dedans si il y en a */
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

Les Intents

Les ***Intents*** permettent de relier une activité avec une autre et même différentes applications entre elles. Avec les *intents*, on peut transférer des valeurs d'une activité à l'autre et les filtrer pour mettre les bonnes valeurs dans les variables concernées.

L'IHM de l'application est composée de plusieurs pages donc de plusieurs activités. Pour créer une nouvelle activité, il faut la rajouter dans le dossier java puis la faire appeler dans l'activité principale avec un « Intent » qui permet de faire le lien avec une autre activité pour la lancer :

```
Intent intent1 = new Intent(MainActivity.this, Parametres.class);
```

Cette ligne fait le lien entre l'activité principale (MainActivity) et la classe parametres.class .

On lance ensuite la nouvelle activité avec la commande :

```
startActivityForResult(intent1, ID_Intent_Parametres);
return true;
```

Cela lance l'activité avec une attente de retour précis qui est RETOUR.

On peut aussi rajouter des « Extra » qui sont des données qui seront envoyées à la nouvelle activité

Pour cela on fait les lignes suivantes avant de lancer la nouvelle activité :

```
                /** met une valeur qui sera envoyée à l'intent1 lors de sa
            création nommée Tmin */
                intent1.putExtra(Tmin, tempMin);
                intent1.putExtra(Tmax, tempMax);
                intent1.putExtra(Phmin, phMin);
                intent1.putExtra(Phmax, phMax);
                intent1.putExtra(NiveauMin, niveauMin);
                  intent1.putExtra(NiveauMax, niveauMax);
```

La première valeur mise en argument est un String qui sera utilisé pour différencier les valeurs entre elles.

Ensuite dans le « onCreate » de la classe Paramètres correspondant à la deuxième activité, on récupère les données en vérifiant que c'est celle qui correspond pour les mettre dans les variables locales : On prend la variable qui contient « Tmin » et on la met dans tempMin pour la ligne 1. Si « Tmin » n'a pas de valeurs, on met la valeur par défaut dans la variable donc dans ce cas, 23 .

```
tempMin = intent.getFloatExtra("Tmin", 23);
tempMax = intent.getFloatExtra("Tmax", 27);
PhMin = intent.getFloatExtra("Phmin", 6);
PhMax = intent.getFloatExtra("Phmax", 8);
niveauMin = intent.getFloatExtra("Niveaumin", 25);
niveauMax = intent.getFloatExtra("Niveaumax", 30);
```

Lors de la fin de l'activité, si on clique sur le bouton valider, on renvoi à l'activité principale les données locales :

```
Intent intent = new Intent();
intent.putExtra(Tmin, tempMin);
intent.putExtra(Tmax, tempMax);
intent.putExtra(Phmin, PhMin);
intent.putExtra(Phmax, PhMax);
intent.putExtra(niveauEaumax, niveauMax);
intent.putExtra(niveauEaumin, niveauMin);

setResult(RESULT_OK, intent);
/** met fin a l'activité pour retourner à l'activité principale*/
finish();
```

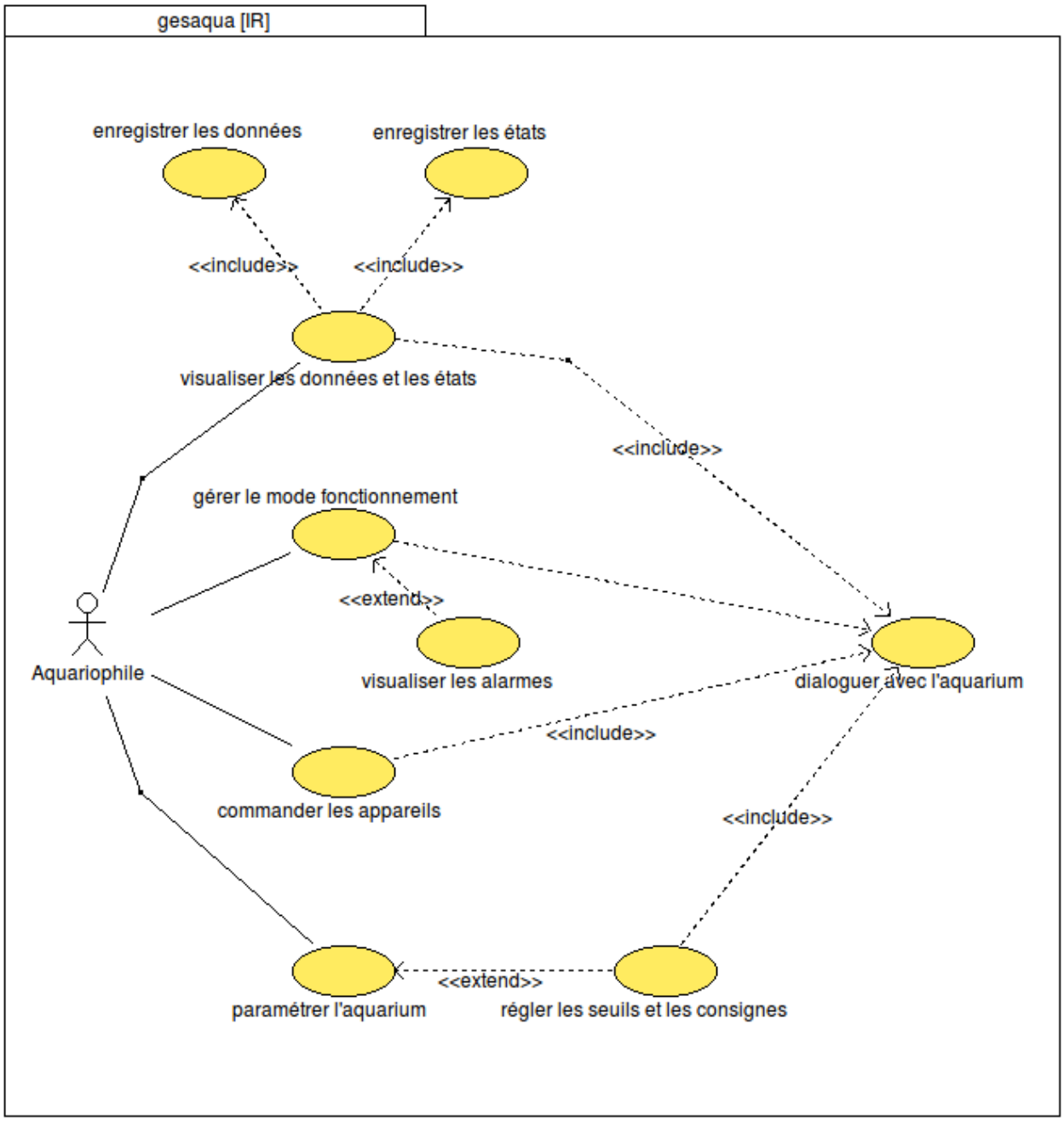
Et on les récupère dans l'activité principale :

```
protected void onActivityResult(int requestCode, int resultCode, Intent intent)
{
    /** Vérifie si on a bien la réponse qui correspond à l'intent1 ( dans ce cas
là RETOUR)*/
    if (requestCode == RETOUR)
    {
        /** Vérifie si c'est bien la réponse attendue*/
        if (resultCode == RESULT_OK)
        {
            /** Remplace les valeurs de tempMin et tempMax de cette activité par
les valeurs saisies dans l'autre activité*/
            tempMin = intent.getDoubleExtra("Tmin", tempMin);
            tempMax = intent.getDoubleExtra("Tmax", tempMax);
            phMin = intent.getDoubleExtra("PhMin", phMin);
            phMax = intent.getDoubleExtra("PhMax", phMax);
            niveauMax = intent.getDoubleExtra("niveauEaumax", niveauMax);
            niveauMin = intent.getDoubleExtra("niveauEaumin", niveauMin);
        }
    }
}
```

Cela permet donc de pouvoir communiquer des informations entre les différentes parties de l’application. Si il n’y a pas de données récupérées de l’activité secondaire, on laisse les valeurs initiales de l’activité principale.

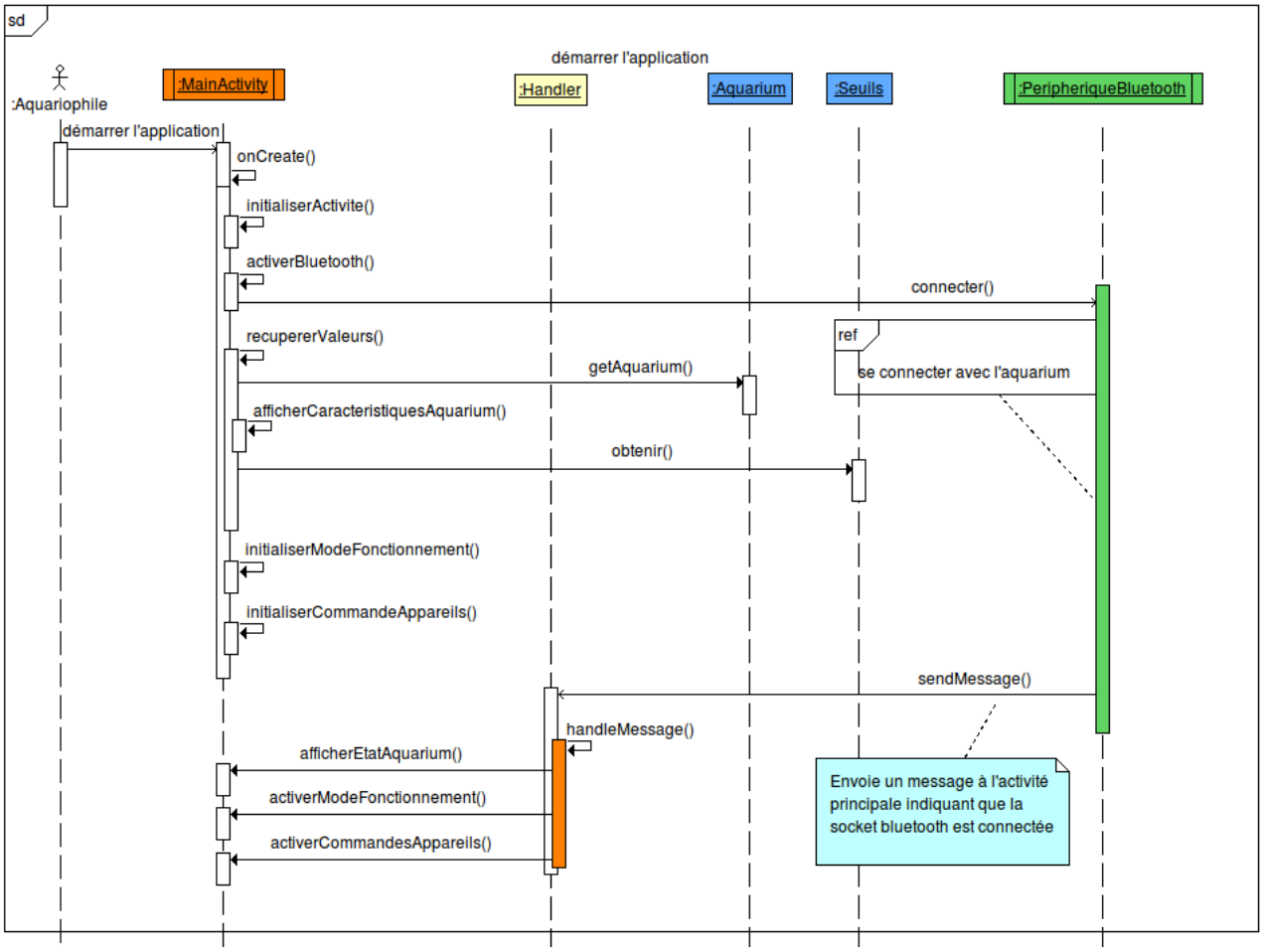
Conception logicielle

Diagramme des cas d'utilisation



Les cas d'utilisations concernent de manière générale la mise en place d'une IHM pour informer l'utilisateur de l'état actuel de l'aquarium. Il faut aussi via les autres cas surveiller et réguler la température et autres paramètres en fonction des alarmes.

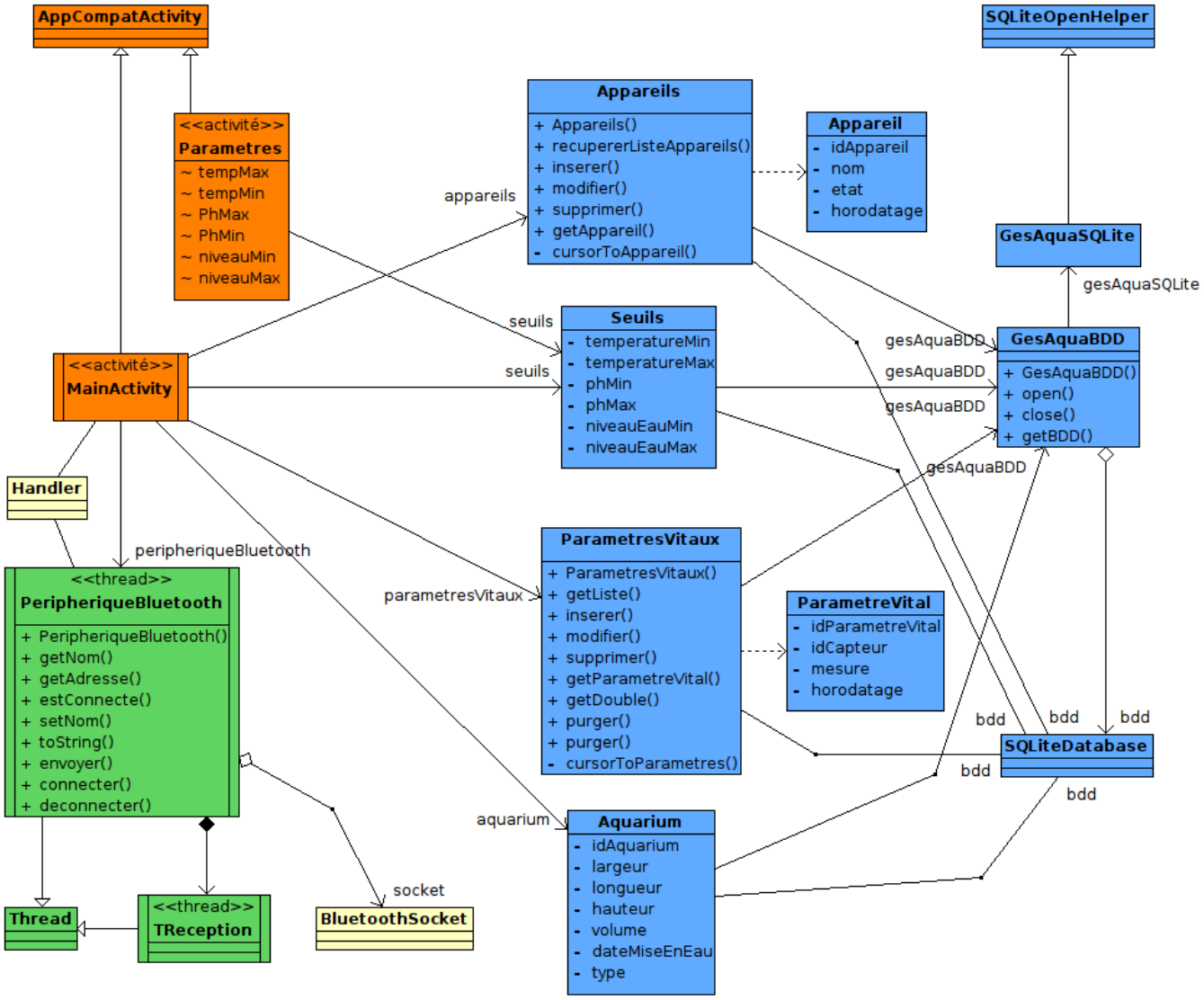
Diagramme de séquence « démarrer l'application »



La code source de la méthode `onCreate()` :

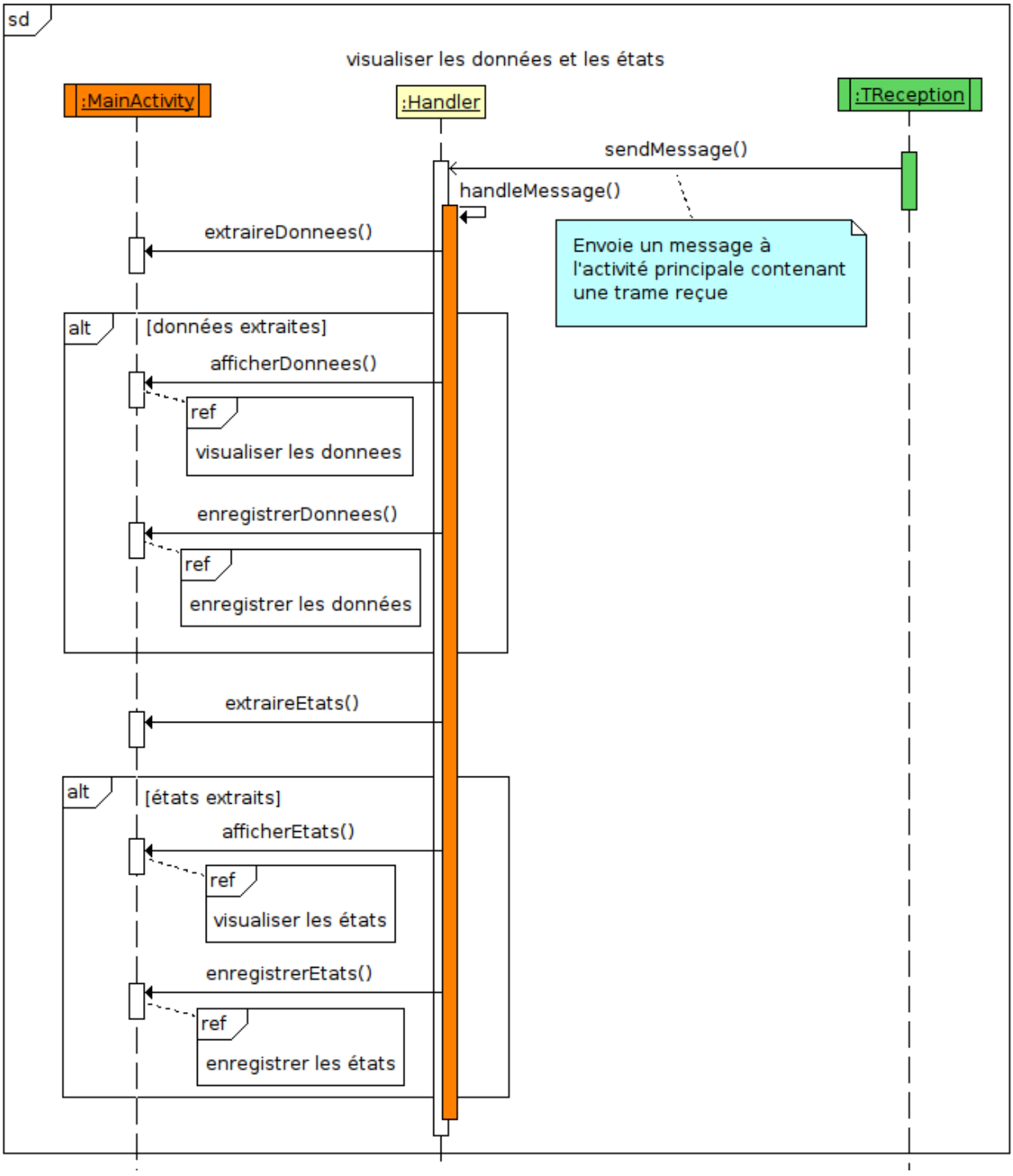
```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // appel la méthode parente

    initialiserActivite();

    activerBluetooth();

    recupererValeurs();

    initialiserModeFonctionnement();

    initialiserCommandeAppareils();

    afficherDonnees();
}
```

Diagramme de classes



Les classes concernant ma partie sont précisées en annexe.

Diagramme de séquence « visualiser données état »



Le code de ce cas d'utilisation est expliqué ci dessous :

```
private void afficherDonnees() {
    if(aquariumConnecte)
```

```
    {
        afficherTemperature();
        afficherPh();
        afficherNiveauEau();
    }
}
```

Permet d'afficher les données vitales.

Pour enregistrerDonnées voir annexe 1.

Pour extraireEtats voir annexe 2.

Pour afficherEtats voir annexe 3.

Pour enregistrerEtats, on fait :

```
private void enregistrerEtats()
{
    if(etatChauffage != etatChauffagePrec)
        modifierEtatAppareil(APPAREIL_CHAUFFAGE, NOM_APPAREIL_CHAUFFAGE,
etatChauffage, R.id.imageViewChauffage, R.id.SwitchChauffage);
    if(etatEclairage != etatEclairagePrec)
        modifierEtatAppareil(APPAREIL_ECLAIRAGE, NOM_APPAREIL_ECLAIRAGE,
etatEclairage, R.id.imageViewEclairage, R.id.SwitchEclairage);
    if(etatNourriture != etatNourrituePrec)
        modifierEtatAppareil(APPAREIL_NOURRITURE, NOM_APPAREIL_NOURRITURE,
etatNourriture, R.id.imageViewNourriture, R.id.SwitchNourriture);
    if(etatEngrais != etatEngraisPrec)
       modifierEtatAppareil(APPAREIL_ENGRAIS, NOM_APPAREIL_ENGRAIS, etatEngrais,
R.id.imageViewEngrais, R.id.SwitchEngrais);
    /* modifie l'état de l'appareil */
    if(etatOxygenation != etatOxygenationPrec)
        modifierEtatAppareil(APPAREIL_OXYGENATION, NOM_APPAREIL_OXYGENATION,
etatOxygenation, R.id.imageViewOxygenation, R.id.SwitchOxygenation);
    /* modifie l'état de l'appareil */
    if(etatFiltration != etatFiltrationPrec)
        modifierEtatAppareil(APPAREIL_FILTRATION, NOM_APPAREIL_FILTRATION,
etatFiltration, R.id.imageViewFiltration, R.id.SwitchFiltration);
    if(etatVentilation != etatVentilationPrec)
        modifierEtatAppareil(APPAREIL_VENTILATION, NOM_APPAREIL_VENTILATION,
etatVentilation, R.id.imageViewVentilation, R.id.SwitchVentilation);
}
```

Ce qui a pour effet de modifier les états des appareils dans la base de données.

Traduction de Trame

Pour traduire la trame on appelle la fonction extraireDonnees(String message). Dans celle ci,il faut tout abord vérifier qu'elle est valide puis la formater pour ensuite la traduire. On vérifie tout d'abord le *checksum* avec la fonction calculer*Checksum*(String trame) qui nous renvoie le *checksum* sous la forme d'un *String.* Ensuite on vérifie que la trame soit valable avec un '$' pour le début de trame, un '*' pour le *checksum* et un '\r' pour la fin de trame avec la fonction verifierTrame(String messsage) qui renvoi un booléen :

```
private boolean verifierTrame(String message)
{
    /* détection d'erreurs via le checksum */
    if (message.contains("$") && message.contains("\r") &&
message.contains("*"))
    {
        /* pour le calcule du checksum */
        String trame = message;
        StringBuffer sbMessageC = new StringBuffer(message);
        StringBuffer sbMessageT = new StringBuffer(message);
        /* Supprime les caractéres jusqu'au checksum */
        sbMessageC.delete(0, sbMessageC.indexOf("*") + 1);
        /* Supprime le checksum */
        sbMessageT.delete(sbMessageT.indexOf("*"), sbMessageT.indexOf("\r") +
1);
        String checksumRecu = sbMessageC.substring(sbMessageC.indexOf("*")+1,
sbMessageC.indexOf("\r"));
        String checksumCalcule = calculerChecksum(sbMessageT.toString());
        //Log.d("checksum", "Checksum recu : " + checksumRecu + " - Checksum
calcule : " + checksumCalcule + " - Vérification : " +
checksumRecu.equals(checksumCalcule));
        if (!checksumRecu.equals(checksumCalcule))
        {
            Log.e("checksum", "Erreur checksum ! (" + checksumRecu + " != " +
checksumCalcule + ")");
            return false;
        }
    }
    return true;
}
```

Si la trame est valide, on peut alors la traduire.

Pour la traduire, on fait un test pour revérifier si elle contient bien un début et fin de trame une comparaison pour savoir de quelle trame il s'agit : trame de données ($d), trame de réglage ($r), trame des seuils ($s), trame des alarmes ($a) et trame des appareils ($e).

Voici un exemple d'une partie de la traduction de la trame de données :

```
/* Vérifie si c'est une trame de données*/
if (message.charAt(1) == 'd') {
    /* Supprime les caractéres jusqu'à la premiére valeur pour la mettre dans
la donnée temperature*/
    sbMessage.delete(0, sbMessage.indexOf(";") + 1);
    champ = sbMessage.substring(0, sbMessage.indexOf(";"));
    if(!champ.isEmpty())
        temperature = Float.parseFloat(champ);
/*Supprime les caractéres jusqu'aux données concernant le niveau de l'eau pour
les affecter a la valeur du niveau d'eau*/
sbMessage.delete(0, sbMessage.indexOf(";") + 1);
```

Envoi de trames :

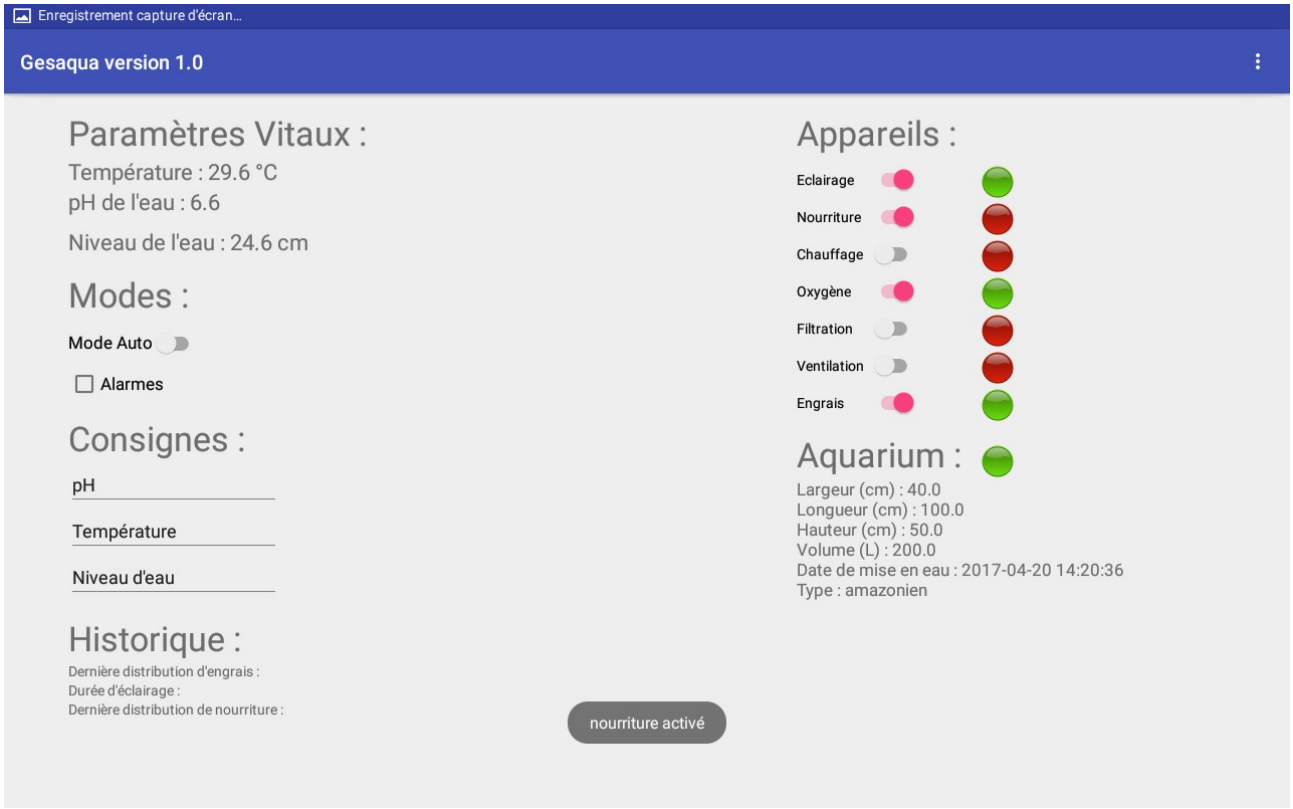
Pour envoyer une trame permettant de changer l'état des appareils de l'aquarium, on utilise la fonction fabriquerTrameCommandeAppareil(**int** idAppareil, **int** etat)

```
/**
 * @brief fabrique une trame avec les données de l'appareil correspondant et
son état
 * @param etat correspond à l'état de l'appareil
 * @param idAppareil correspond à l'indentifiant de l'appareil
 * (Voir protocole de communication avec l'aquarium)
 */
private String fabriquerTrameCommandeAppareils(int idAppareil, int etat) {
    String trame = TRAME_DEBUT + "e" + TRAME_DELIMITEUR;
    // Type : e (commande des appareils)
    List<Appareil> listeAppareils = appareils.recupererListeAppareils();
    for (int i = 0; i < listeAppareils.size(); i++)
    {
        Appareil appareil = listeAppareils.get(i);
        if((i + 1) == idAppareil)
        {
            if ((i + 1) == listeAppareils.size())
                trame += etat;
            else
                trame += (etat + TRAME_DELIMITEUR);
            continue;
        }
        else
        {
            if ((i + 1) == listeAppareils.size())
                trame += appareil.getEtat();
            else
                trame += (appareil.getEtat() + TRAME_DELIMITEUR);
        }
    }
    trame = ajouterChecksum(trame);
    trame += TRAME_FIN;
    return trame;
}
```

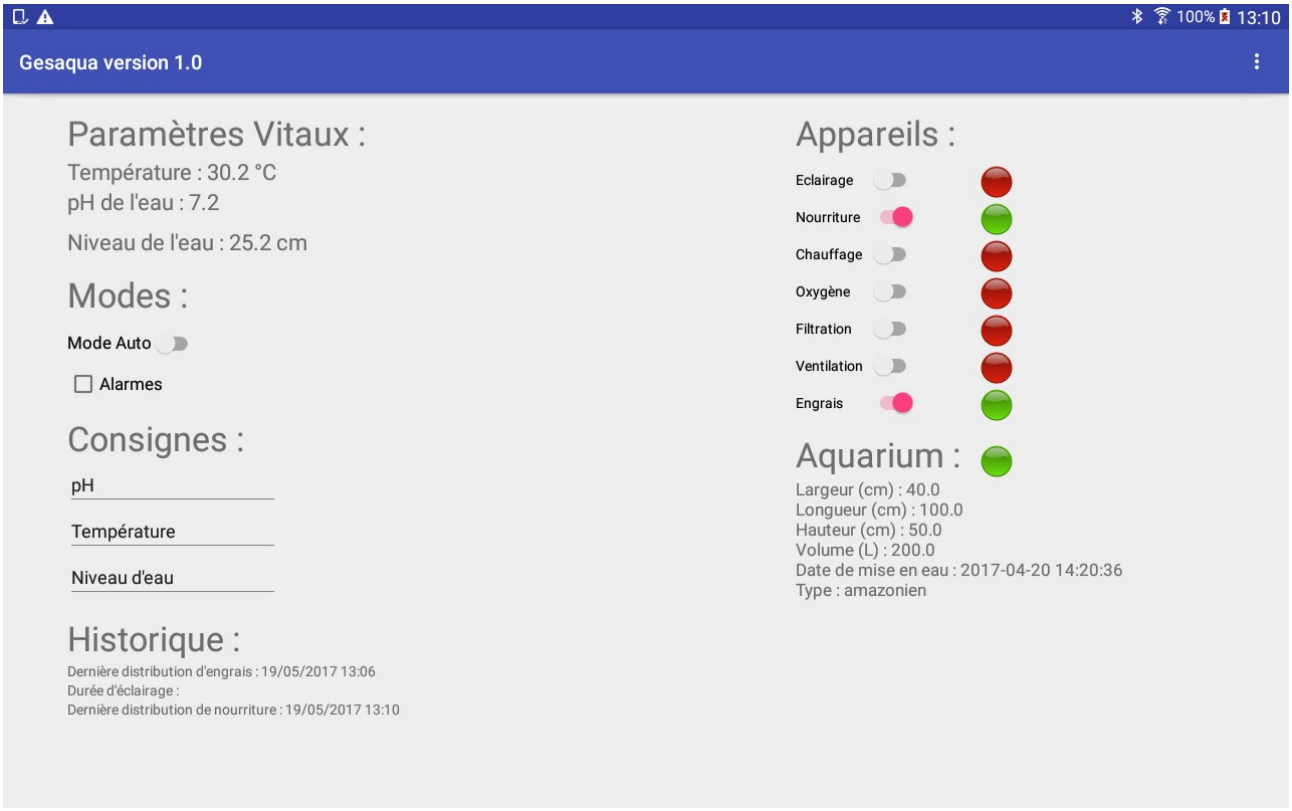
Exemples d'IHM

Main_Activity : Main_Activity est la première activité à être lancée lors du démarrage de l'application :

Activation de la nourriture et désactivation des autres paramètres:



Le voyant s'allume en vert et les autres désactivés se mettent en rouge :



Parametres.class :

Conclusion

Le projet touche à son terme et l'application est opérationnelle. Malgré de nombreux soucis d'affichage ainsi que des problèmes de traduction de trame et de lien avec les bases de données, ce projet s'est bien passé. Le plus difficile fut de s'habituer avec l'environnement de développement qui avait ses propres particularités et le fait que l'application s'exécute sur Android aussi. Le projet fait en langage JAVA fut aussi une contrainte au début de celui-ci mais il nous a permis de renforcer notre connaissance de ce langage. Pour terminer les tâches nous étant confiées par le cahier des charges, il reste a faire le lien avec les bases de données permettant de récupérer les données et les afficher via les BDD, l'enregistrement des seuils dans les bases de données, la gestion des alarmes via l'affichage et les BDD. La gestion des trames est faite et l'IHM se modifie selon l'état des communications. Il reste aussi à gérer le mode automatique en affichant les seuils et en modifiant l'état des *widgets* pour désactiver les commandes manuelles.

Recette étudiant E3

DESCRIPTION	OUI	NON
Les données et les alarmes associées au module sont affichées sur la tablette tactile	X	
La commande des appareils associés au module est possible à partir de la tablette tactile	X	
Les mesures moyennées sur une heure sont enregistrées périodiquement dans la base de données	X	
Les nouvelles consignes et seuils min. et max. sont enregistrés dans la base de données		X
Les données de l'aquarium (type, mise en eau, dimensions, volume, nombe et taille totale des poissons) sont récupérées à partir de la base de données et affichées avec leurs unités dans l'IHM	X	
Les prochaines échéances des interventions (analyse de l'eau, intervention, entretien) sont récupérées de la base de données et notifiées si nécessaire		X
La sélection des alarmes à surveiller est possible à partir de l'IHM		X
Les nouvelles sélections des alarmes à surveiller sont enregistrées dans la base de données		X

Annexes

Annexe 1 : Fonction enregistrerDonnées

```
private void enregistrerDonnees()
{
    ParametreVital parametreVital = null;
    /* enregistre l'échantillon dans la base de données */
    parametreVital = new ParametreVital(CAPTEUR_TEMPERATURE, temperature,
getHorodatageBD());
    long resultat = parametresVitaux.inserer(parametreVital);
    if (resultat != -1) {
        parametreVital.setIdParametreVital((int) resultat);
        Log.d("enregistrerDonnees()", "Id insertion température : " + resultat);
// d = debug
    }
    parametreVital = new ParametreVital(CAPTEUR_NIVEAU_EAU, niveauEau,
getHorodatageBD());
    resultat = parametresVitaux.inserer(parametreVital);
    if (resultat != -1) {
        parametreVital.setIdParametreVital((int) resultat);
        Log.d("enregistrerDonnees()", "Id insertion niveau d'eau : " +
resultat); // d = debug
    }
    parametreVital = new ParametreVital(CAPTEUR_PH, pHEau, getHorodatageBD());
    resultat = parametresVitaux.inserer(parametreVital);
    if (resultat != -1) {
        parametreVital.setIdParametreVital((int) resultat);
        Log.d("enregistrerDonnees()", "Id insertion pH : " + resultat); // d =
debug
    }
    /* Enregistre la valeur */
    seuils.setTemperatureMin(tempMin);
    /* Enregistre la valeur */
    seuils.setTemperatureMax(tempMax);
    /* Enregistre la valeur */
    seuils.setNiveauEauMin(niveauMin);
    /* Enregistre la valeur */
    seuils.setNiveauEauMax(niveauMax);
    /* Enregistre la valeur */
    seuils.setPhMax(phMax);
    /* Enregistre la valeur */
    seuils.setPhMin(phMin);
    /* purge les valeurs enregistrées si nécessaire (par heure) */
    purgerValeurs();
}
```

Annexe 2 : Fonction extraireEtat :

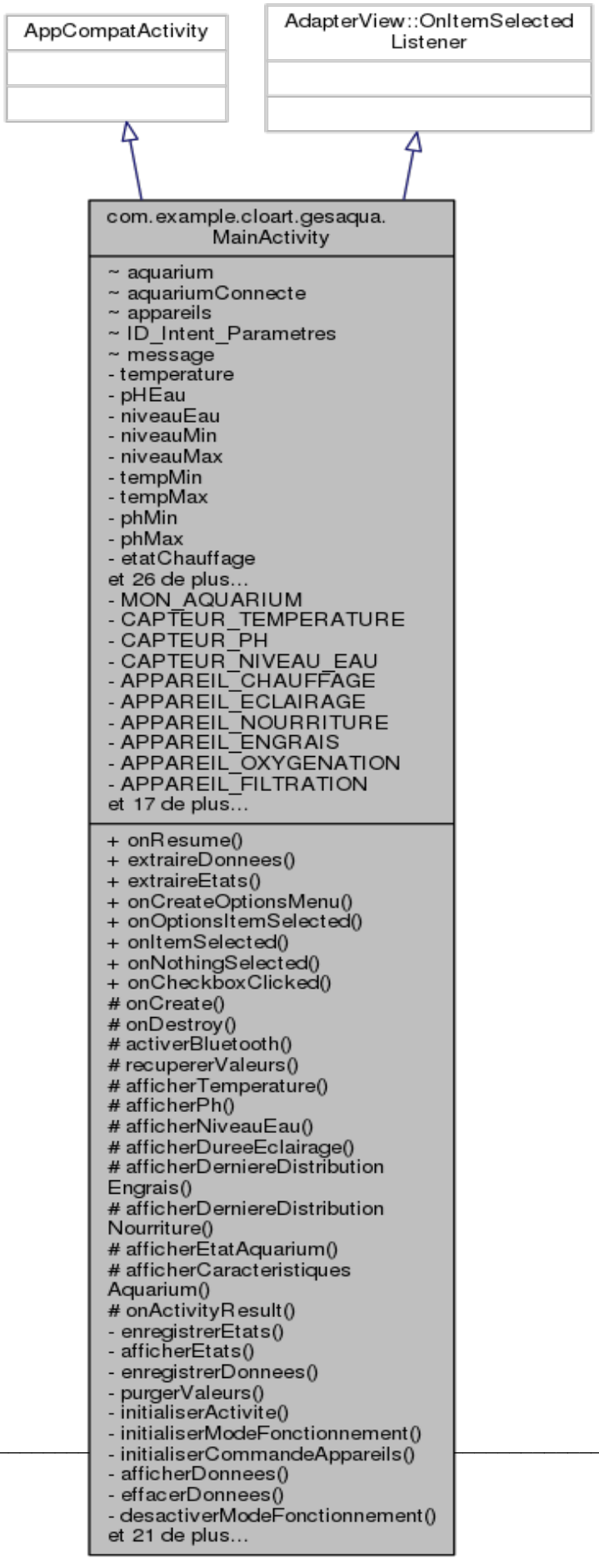
```
public boolean extraireEtats(String message) {
    boolean retour = false;
    StringBuffer sbMessage = new StringBuffer(message);
    int etat;
    if (!verifierTrame(message)) return retour;
    /* présence des délimiteurs ? */
    if (message.contains("$") && message.contains("\r")) {
        /* permet de supprimer tous les caractéres avant le premier '$' pour
avoir un debut de trame valide */
        sbMessage.delete(0, message.indexOf("$"));
        /* Vérifie si le format du message correspond bien a celui des trames
*/
        while (message.contains("\r") && message.contains("$")) {
            /* Vérifie le type de trame */
            if (message.charAt(1) == 'e') {
                /* Supprime les caractéres jusqu'au données suivantes */
                sbMessage.delete(0, sbMessage.indexOf(";") + 1);
                /* extrait l'état */
                etatChauffagePrec = etatChauffage;
                etatChauffage = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf(";")));
                /* idem pour tous les états suivants */
                sbMessage.delete(0, sbMessage.indexOf(";") + 1);
                etatEclairagePrec = etatEclairage;
                etatEclairage = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf(";")));
                sbMessage.delete(0, sbMessage.indexOf(";") + 1);
                etatNourriturePrec = etatNourriture;
                etatNourriture = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf(";")));
                sbMessage.delete(0, sbMessage.indexOf(";") + 1);
                etatEngraisPrec = etatEngrais;
                etatEngrais = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf(";")));
                sbMessage.delete(0, sbMessage.indexOf(";") + 1);
                etatOxygenationPrec = etatOxygenation;
                etatOxygenation = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf(";")));
                sbMessage.delete(0, sbMessage.indexOf(";") + 1);
                etatFiltrationPrec = etatFiltration;
                etatFiltration = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf(";")));
                /* Supprime les caractéres jusqu'au données suivantes */
                sbMessage.delete(0, sbMessage.indexOf(";") + 1);
                etatVentilationPrec = etatVentilation;
                // délimiteur checksum présent ?
                if(message.contains("*"))
                {
                    etatVentilation = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf("*")));
                }
                else
```

```
                {
                    etatVentilation = Integer.parseInt(sbMessage.substring(0,
sbMessage.indexOf("\r")));
                }
                retour = true;
            }
            sbMessage.delete(0, sbMessage.indexOf("\r") + 1);
            message = sbMessage.toString();
        }
    }
    return retour;
}
```

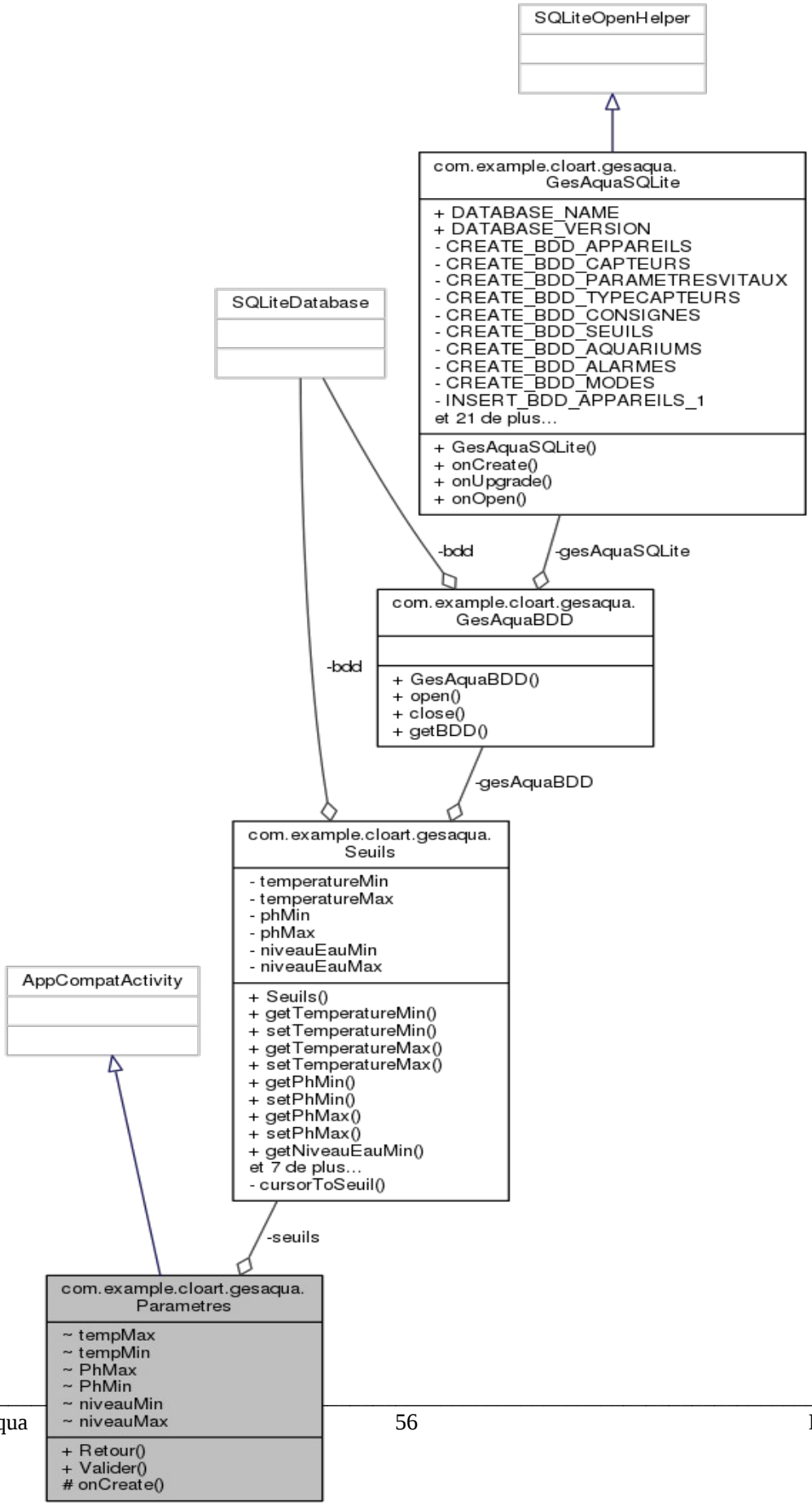
Annexe 3 : Fonction AfficherEtats :

```
private void afficherEtats()
{
    if(etatChauffage != etatChauffagePrec)
        afficherEtatAppareil(APPAREIL_CHAUFFAGE, NOM_APPAREIL_CHAUFFAGE,
etatChauffage, R.id.imageViewChauffage, R.id.SwitchChauffage);
    if(etatEclairage != etatEclairagePrec)
        afficherEtatAppareil(APPAREIL_ECLAIRAGE, NOM_APPAREIL_ECLAIRAGE,
etatEclairage, R.id.imageViewEclairage, R.id.SwitchEclairage);
    if(etatNourriture != etatNourriturePrec)
        afficherEtatAppareil(APPAREIL_NOURRITURE, NOM_APPAREIL_NOURRITURE,
etatNourriture, R.id.imageViewNourriture, R.id.SwitchNourriture);
    if(etatEngrais != etatEngraisPrec)
        afficherEtatAppareil(APPAREIL_ENGRAIS, NOM_APPAREIL_ENGRAIS,
etatEngrais, R.id.imageViewEngrais, R.id.SwitchEngrais);
    if(etatOxygenation != etatOxygenationPrec)
        afficherEtatAppareil(APPAREIL_OXYGENATION, NOM_APPAREIL_OXYGENATION,
etatOxygenation, R.id.imageViewOxygenation, R.id.SwitchOxygenation);
    if(etatFiltration != etatFiltrationPrec)
        afficherEtatAppareil(APPAREIL_FILTRATION, NOM_APPAREIL_FILTRATION,
etatFiltration, R.id.imageViewFiltration, R.id.SwitchFiltration);
    if(etatVentilation != etatVentilationPrec)
        afficherEtatAppareil(APPAREIL_VENTILATION, NOM_APPAREIL_VENTILATION,
etatVentilation, R.id.imageViewVentilation, R.id.SwitchVentilation);
    /* arrêt éclairage ? */
    if(etatEclairagePrec != etatEclairage && etatEclairage == 0)
    {
        calculerDureeEclairage(); /** @todo calculerDureeEclairage() */
        Log.d("enregistrerEtats()", "Durée éclairage : " + dureeEclairage); //
d = debug
        afficherDureeEclairage();
    }
    /** @todo gestion de la dernière distribution de nourriture */
    /** @todo gestion de la dernière distribution d'engrais */
}
```

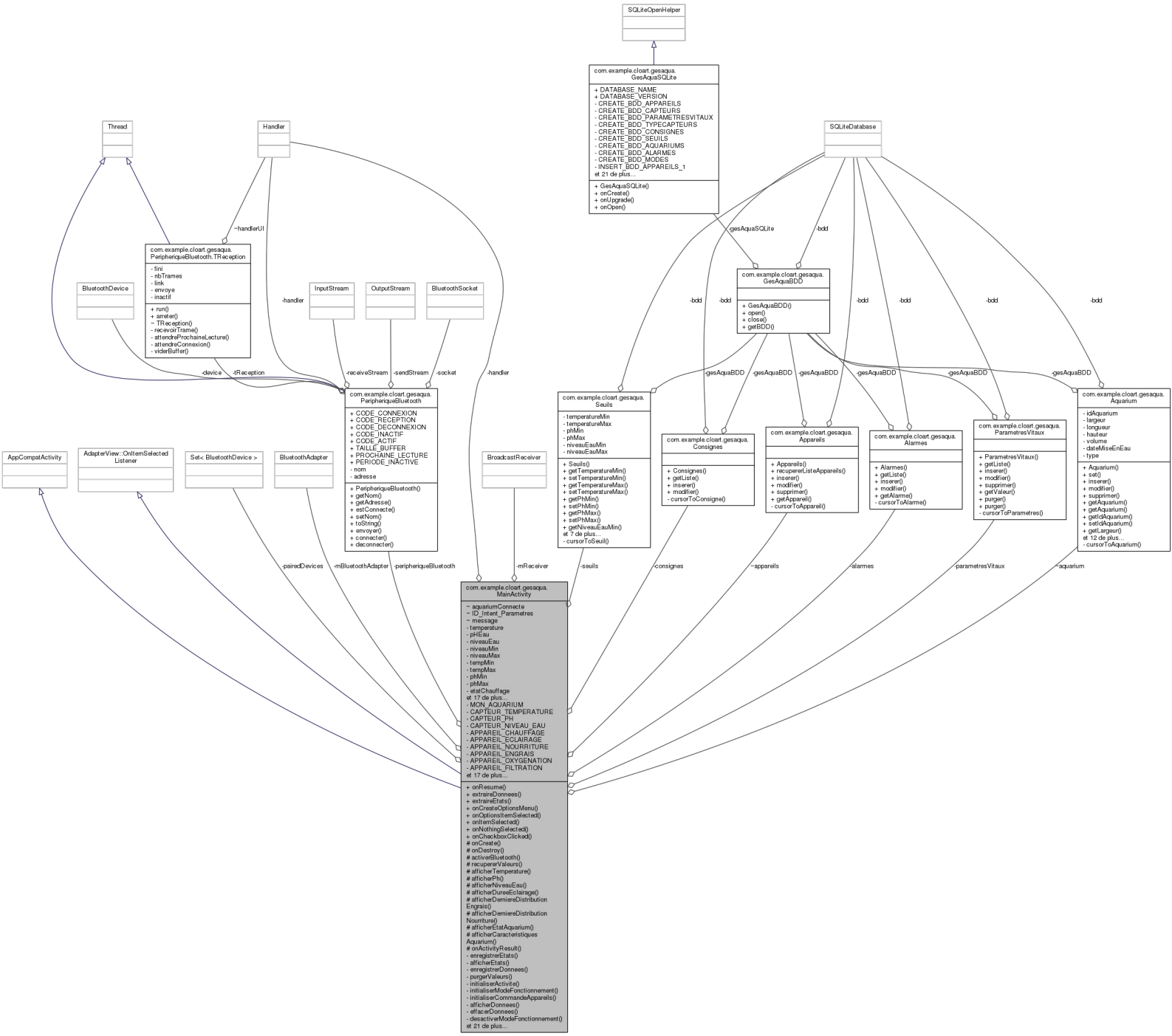
Annexe 4: Diagramme de classe MainActivity



Annexe 5 : diagramme de classe Parametres



Annexe 6 : Diagramme de classe de l'application



Module de commande des appareils - CLOART Audrey (étudiant 4)

Table des matières

Objectifs

Afin de pouvoir gérer l'aquarium, il est nécessaire de pouvoir actionner différents appareils. Grâce à cela, les conditions initiales de vie des poissons pourront être maintenues de façon satisfaisante.

La partie du programme dont je me suis occupée concerne le module de commande des appareils, illustrée par le diagramme des cas d'utilisation ci-dessous.

Ainsi, dans cette partie, l''utilisateur doit pouvoir :

1. Visualiser les données. Ces données sont :

- la dernière distribution d'engrais,
- la dernière distribution de nourriture
- et la durée actuelle d'éclairage (en minutes).

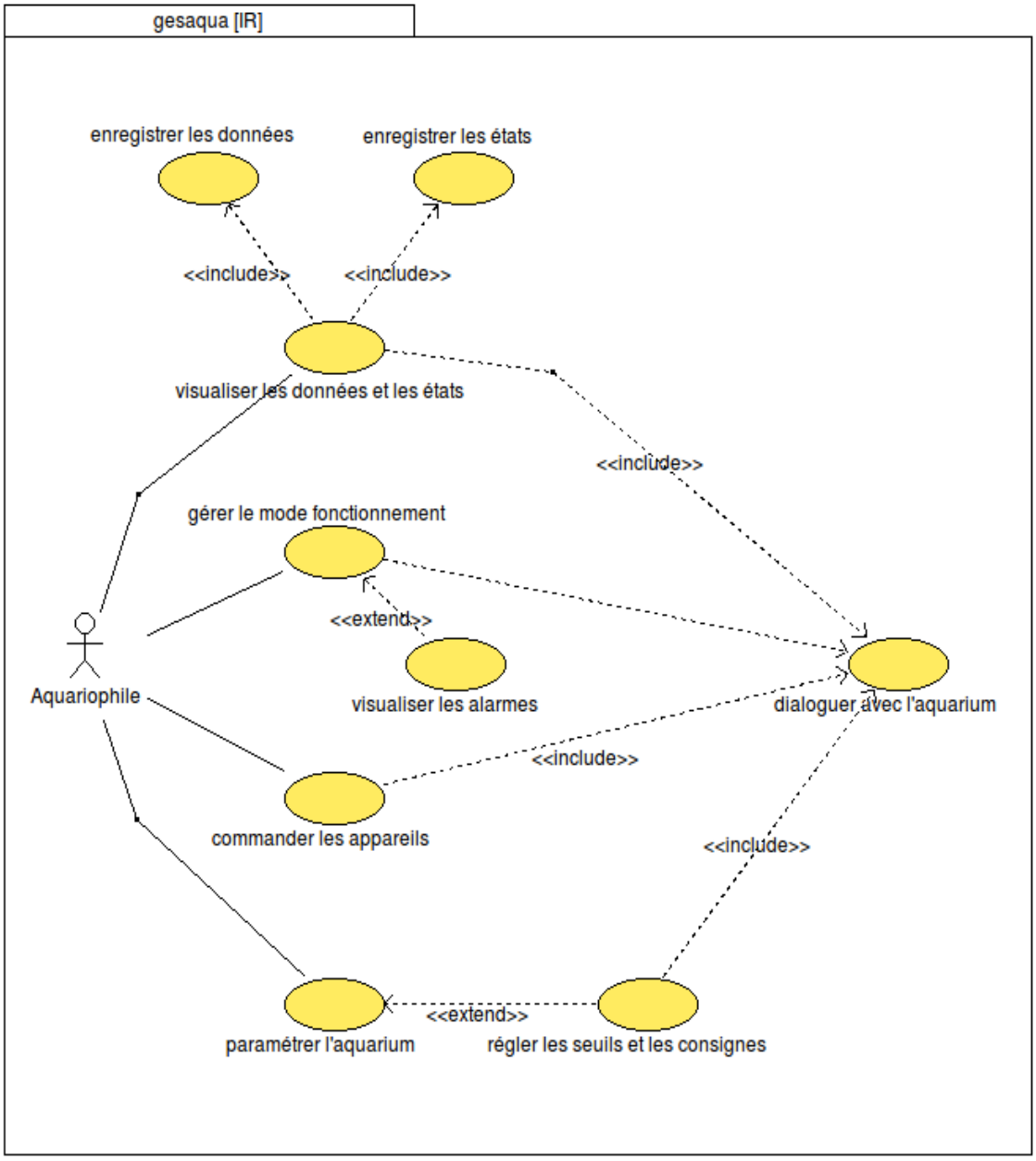
2. Visualiser les états des appareils, s'ils sont activés ou non. Il s'agit des appareils :

- de distribution de nourriture
- de distribution d'engrais
- d'oxygénation
- de filtration
- et d'éclairage.

3. Il doit pouvoir paramétrer le fonctionnement de l'aquarium, c'est-à-dire pouvoir choisir entre un mode automatique ou manuel. Il doit donc pouvoir paramétrer le mode automatique.

4. Il doit pouvoir commander les appareils via la tablette tactile. Concernant la partie dont j'ai la charge, il s'agit de :

- la distribution de nourriture
- la distribution d'engrais
- l'oxygénation
- la filtration
- et l'éclairage.

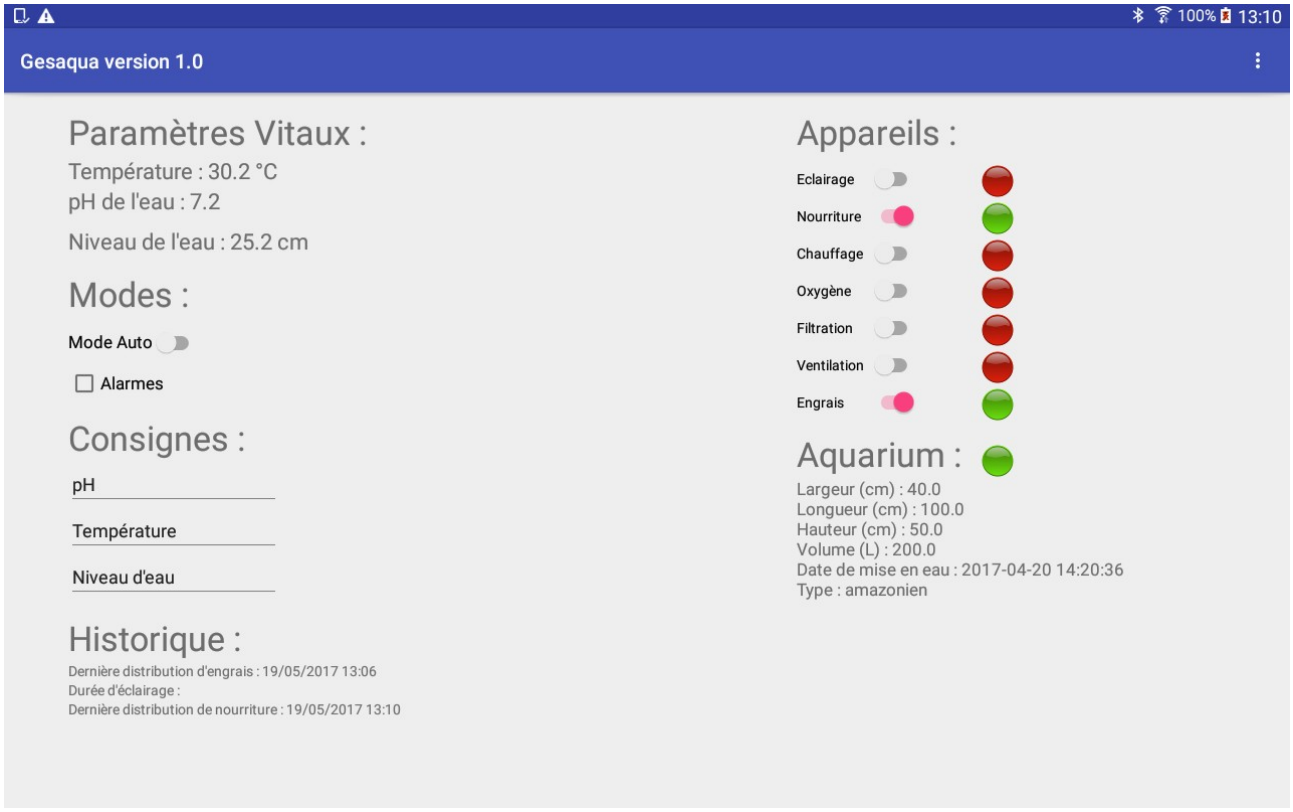


Voici ci-dessous une capture d'écran de l'**IHM**[4] de l'application :

On voit que la partie `Historique` permet de visualiser les données dont je suis responsable.
La partie `Appareils` permet à l'utilisateur de visualiser l'état de ses appareils ainsi que de les commander.
La partie `Modes` permet à l'utilisateur de démarrer le mode automatique ou de repasser en mode

4 Interface Homme Machine

manuel, selon son choix.



Tâches à effectuer

Concernant ma partie, du point de vue du système, le cahier des charges contient les problématiques suivantes :

- Commande des appareils par la tablette tactile
- Signal des alarmes
- Paramétrage du fonctionnement
- Information de l'utilisateur
- Archivage des données
- Communication des ordres et des données

Contraintes matérielles et logicielles

- Android Studio 2.3
- Système d'exploitation de la tablette : Android 4.4.2 (KitKat)
- java 1.8.0
- SDK Android API 25 : Android 7.1.1 (Nougat)
- svn, version 1.6.17 (r1128011)
- doxygen 1.7.6.1
- bouml Bouml 4.23
- Gestionnaire de projet : Planner 0.14.5
- SGBDR[5] : SQLite3
- Tablette tactile Samsung Galaxy Tab 4

Répartition des tâches

J'ai développé ma partie selon les tâches suivantes :

- Module : commande des appareils *en lien avec l'étudiant 2*

- Installation : la liaison sans fil (Bluetooth)

- Mise en œuvre :
  → l'environnement de développement
  → la base de données

- Configuration : la base de données

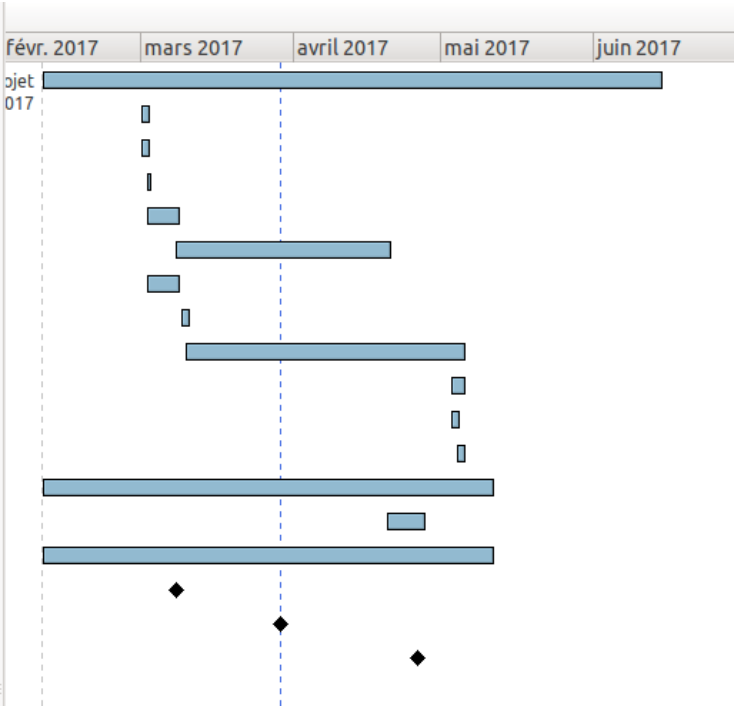
- Réalisation :
  → diagrammes SysML / UML
  → code source de l'application
  → IHM du module

- Documentation :
  → le dossier technique et les documents relatifs au module
  → un guide de mise en route et d'utilisation du module

5 Système de Gestion de Base de Données Relationnelles

Planification des tâches

TPÉ	Nom
1	Projet GesAqua
2	s'approprier le cahier des charges
3	installer et configurer l'environnement de développement
4	raccorder la tablette tactile
5	mettre en oeuvre les programmes de test fournis
6	finaliser la modélisation UML du module
7	prototyper l'IHM du module
8	installer et configurer la base de données
9	codes les classes du module
10	réaliser les tests unitaires
11	faire la recette du module
12	intégrer en équipe l'application complète
13	rédiger le dossier technique te les documents relatifs au projet
14	produire un guide de mise en route et d'utilisation du module
15	gérer la plannification
16	revue n°0
17	revue n°1
18	revue n°2



Communiquer les ordres et les données

Commander les appareils par la tablette

Pour commander les appareils, il est nécessaire de pouvoir dialoguer avec l'aquarium, et donc de s'y connecter.

Pour se connecter à l'aquarium, on utilise une liaison sans fil de type *Bluetooth* 4.0.
Le ***Bluetooth***[6] est un standard de communication permettant l'échange bidirectionnel de données à très courte distance en utilisant des ondes radio UHF[7] sur une bande de fréquence de 2,4 GHz.

Android Studio, qui est l'Environnement de Développement Intégré choisi pour écrire l'application, permet l'accès aux fonctionnalités *Bluetooth* via l'API *Bluetooth* d'Android.

Après avoir accepté la demande d'appairage[8], il y a une connexion[9] entre les périphériques. Après que l'appairage et la connexion aient été effectués, les 2 périphériques peuvent échanger des données.

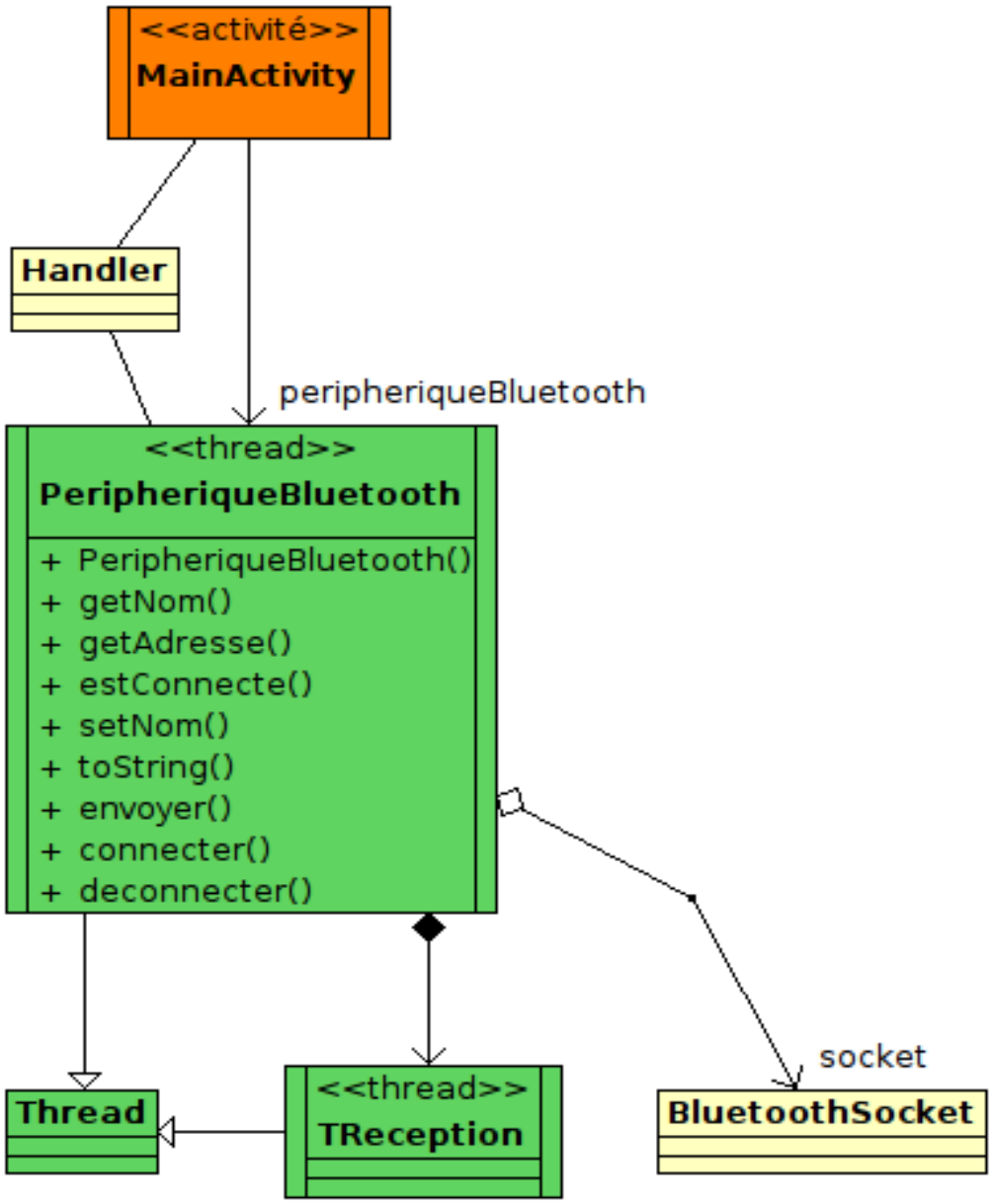
6 CF ANNEXE 1 pour la MISE EN OEUVRE DE LA LIAISON BLUETOOTH
7 Ultra Hautes Fréquences
8 Cf Glossaire
9 Cf Glossaire

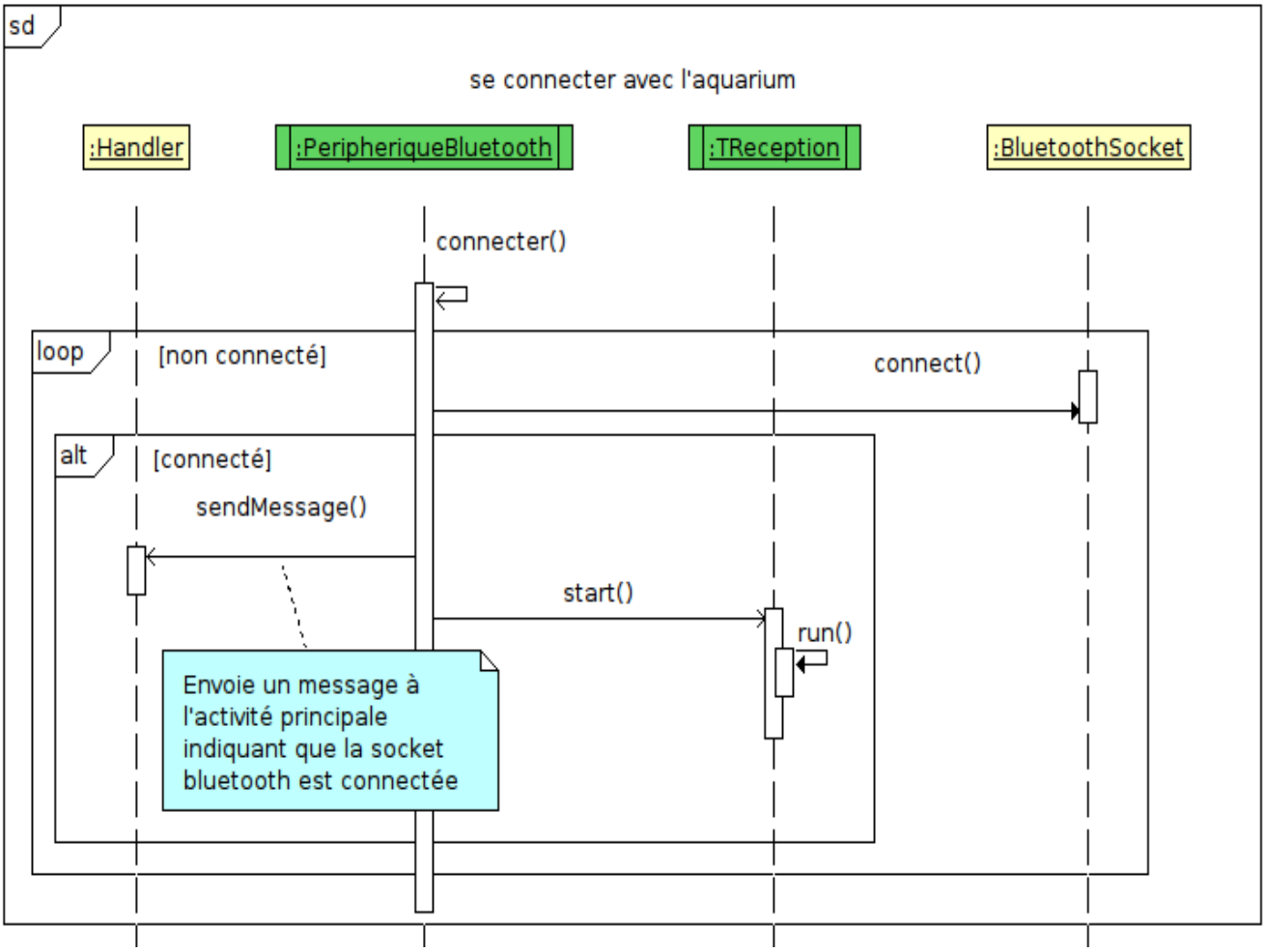
Se connecter avec l'aquarium

Afin de se connecter avec l'aquarium, on voit sur le diagramme de classes partiel ci-dessous que plusieurs classes sont nécessaires :

- La classe `PeripheriqueBluetooth` est celle qui permet la connexion avec l'aquarium. Elle est située dans un *thread*.
- C'est un `Handler`[10] qui fait le lien entre `PeripheriqueBluetooth` et la page principale de l'application.
- `Treception` est un *thread* qui permet de recevoir les trames en provenance de l'aquarium.
- Enfin *BluetoothSocket* est une socket qui permet de connecter les périphériques *Bluetooth* entre eux.

10 Handler : objet qui permet l'envoi de messages


<<activité>>
MainActivity
Handler
peripheriqueBluetooth
<<thread>>
PeripheriqueBluetooth
+ PeripheriqueBluetooth()
+ getNom()
+ getAdresse()
+ estConnecte()
+ setNom()
+ toString()
+ envoyer()
+ connecter()
+ deconnecter()
socket
Thread
<<thread>>
TReception
BluetoothSocket



Sur ce diagramme de séquence, on peut donc voir que ces différentes classes seront utilisées pour se connecter à l'aquarium depuis la tablette tactile :

- La classe `PeripheriqueBluetooth` est celle qui assure le dialogue *Bluetooth* avec l'aquarium
- La classe *`TReception`* est un *thread* de réception des trames en provenance de l'aquarium via le *Bluetooth.*
- La classe *`Handler`*[11] a pour rôle de gérer les communications avec le *thread* de réception des trames *(`TReception`).*
- La classe *BluetoothSocket* est une classe présente dans *l'API Bluetooth* d'Android. Elle sert à créer une connexion entre les périphériques afin qu'ils puissent échanger des données de façon bidirectionnelle.

11 Handler : objet qui permet l'envoi de messages

En effet, afin de créer une connexion entre 2 périphériques, il est nécessaire d'implémenter à la fois le côté client et le côté serveur puisque l'un des périphériques doit ouvrir une *socket* serveur, et l'autre doit initier la connexion en utilisant l'adresse MAC du périphérique serveur : la `BluetoothSocket`.

Le client et le serveur sont considérés comme connectés lorsqu'ils ont tous deux une `BluetoothSocket` connectée sur le même canal RFCOMM[12].

A ce moment là, chaque périphérique peut échanger ses données, de façon bidirectionnelle.

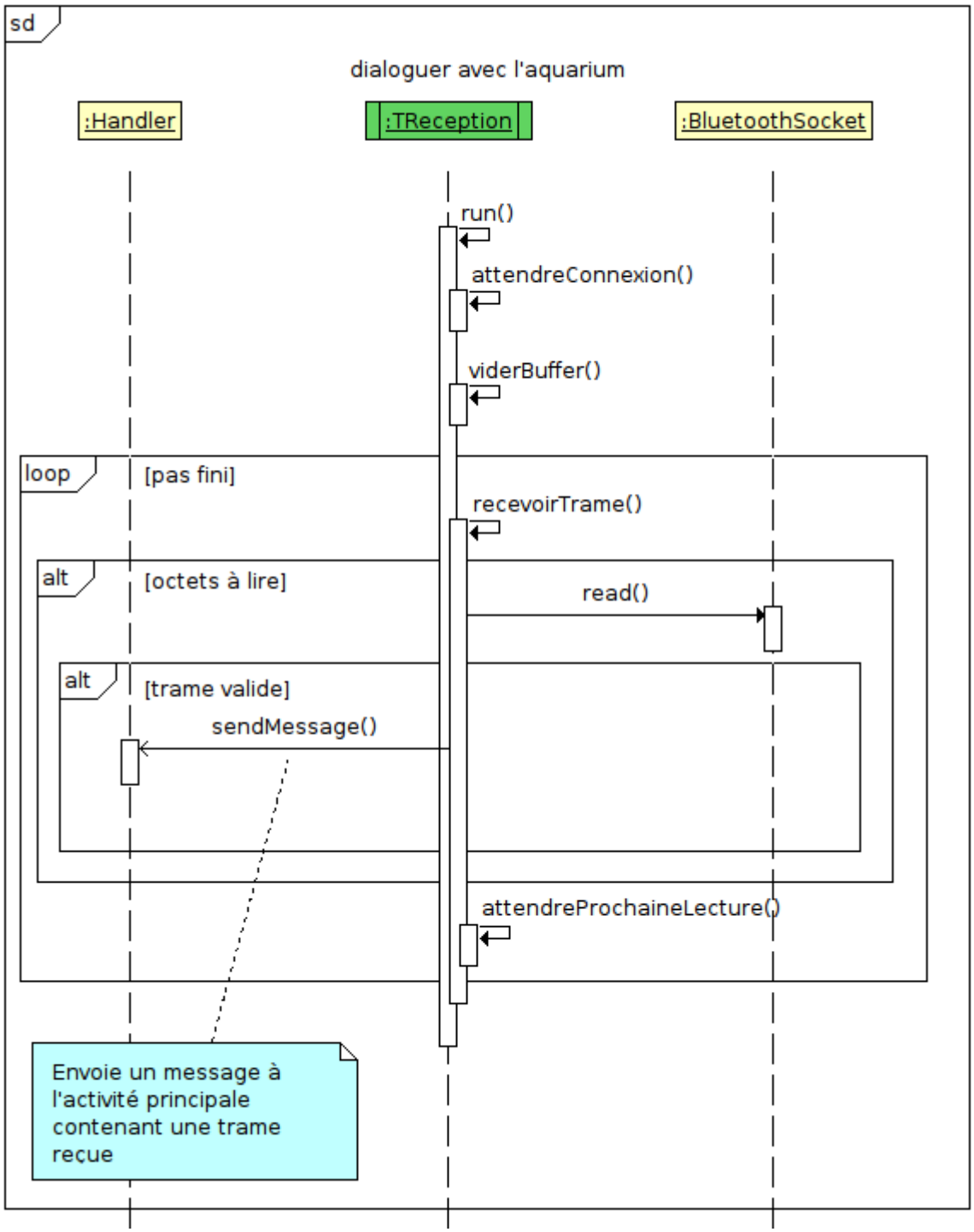
Classe `PeripheriqueBluetooth` :

La classe `PeripheriqueBluetooth` connecte la *socket Bluetooth*. (1)
Lorsque celle-ci est connectée, on envoie un message à l'activité principale. (2)
On démarre alors le *thread* `TReception` et commence la réception de trames (3) (`cf diagramme de séquence ci-dessous 'dialoguer avec l'aquarium'`).

```
//Connecte la socket Bluetooth et démarre le thread Réception

public void connecter()
{
    /* démarre le thread connexion */
    new Thread()
    {
        @Override public void run()
        {
            while(!socket.isConnected())
            {
                try
                {
                    /* Demande de connexion */
                    socket.connect();  //(1)
                    /* connecté ? */
                    if (socket.isConnected())
                    {
                        /* informe l'activité principale */
                        Message msg = Message.obtain();
                        msg.what = PeripheriqueBluetooth.CODE_CONNEXION; //(2)

                     //...

                        /* démarre le thread réception */
                        tReception.start(); //(3)
                    }

                  //...

    }.start();
}
```

12 RFCOMM : cf Glossaire

Dialoguer avec l'aquarium



La classe *`TReception`* est un *thread* de réception des trames en provenance de l'aquarium grâce au *Bluetooth.* Sur ce diagramme de séquence, on peut voir que c'est cette classe qui reçoit les

trames.
Si le *buffer*[13] contient des octets, alors ils sont lus par la classe `BluetoothSocket`.

Dans la classe `TReception` :

On reçoit la trame. (1)

```
@Override public void run()
{
    attendreConnexion();
    viderBuffer();
    /* boucle de réception des trames */
    while(!fini)
    {
        recevoirTrame(); //(1)
    }
}
```

Si la trame lue est valide, alors un message est envoyé à l'activité principale. Ce message contient la trame reçue. (2)

```
private void recevoirTrame()
{
  try
  {
    if(receiveStream.available() > 0)
    {
        byte buffer[] = new byte[PeripheriqueBluetooth.TAILLE_BUFFER];
        if(!fini && socket.isConnected())
        {
          int k = receiveStream.read(buffer, 0,
PeripheriqueBluetooth.TAILLE_BUFFER);

//...
/* extrait la trame du buffer et la transmet à l'activité principale */

          Message msg = Message.obtain();
          msg.what = PeripheriqueBluetooth.CODE_RECEPTION;
          msg.obj = trame;
          handlerUI.sendMessage(msg); //(2)
        }
    }
}}
```

13 Zone mémoire utilisée pour stocker temporairement des données.

Dans `MainActivity.xml`[14] :

On reçoit la trame (3) et on l'affiche (4).

```
final private Handler handler = new Handler() {
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        //...
        else if (msg.what == PeripheriqueBluetooth.CODE_RECEPTION) {
            /* Vérifie si le message n'est pas vide */
            if (msg.obj != null)
            {
                message = (String) msg.obj; //(3)
                /* extrait les données (eau, seuils, alarmes, ...) de la trame
reçue */
                boolean donneesExtraites = extraireDonnees(message);
                if(donneesExtraites)
                {
                    /* provoque l'affichage des données extraites */
                    afficherDonnees(); //(4)
                    /* provoque l'enregistrement des données extraites */
                    enregistrerDonnees();
                }
                /* extrait les états des appareils de la trame reçue */
                boolean etatsExtraits = extraireEtats(message);
                if(etatsExtraits)
                {
                    /* provoque l'affichage des états extraits */
                    afficherEtats(); //(4)
                    /* provoque l'enregistrement des états extraits */
                    enregistrerEtats();
                }
            }
        }}}
```

Les appareils

Les appareils qui doivent pouvoir être commandés par la tablette sont les appareils suivants :

- Distribution de nourriture
- Distribution d'engrais
- Oxygénation
- Filtration
- Eclairage

14 C'est le *thread UI*, il est matérialisé par la page principale de l'application.

Protocole – La trame de type `e`

Le système comporte plusieurs types de trames. Chaque trame va être associée à une utilisation spécifique.

Ainsi les trames de commande des appareils sont des trames de type `e`.
Le protocole a été défini comme suit :
Exemple de trame de type `e :`

```
$e;0;1;0;0;0;0;0*YY\r
```

Les différents champs de la trame

```
$ Caractère de début de trame
e Type de trame : ici « e » car c'est le type de trame réservé aux appareils
; Délimiteur des champs
```

7 champs de valeur (1 pour chaque appareil) qui valent 0 si l'appareil est désactivé ou 1 s'il est activé. Dans l'ordre, on a la liste d'appareils suivants :

- chauffage
- éclairage
- distributeur de nourriture
- distributeur d'engrais
- oxygénation
- filtration
- ventilation

```
* Caractère qui marche le début du checksum
YY Valeur du checksum
\r Caractère de fin de trame
```

Donc dans notre exemple :

```
$e;0;1;0;0;0;0;0*YY\r
```

Les valeurs des différents champs de données sont :

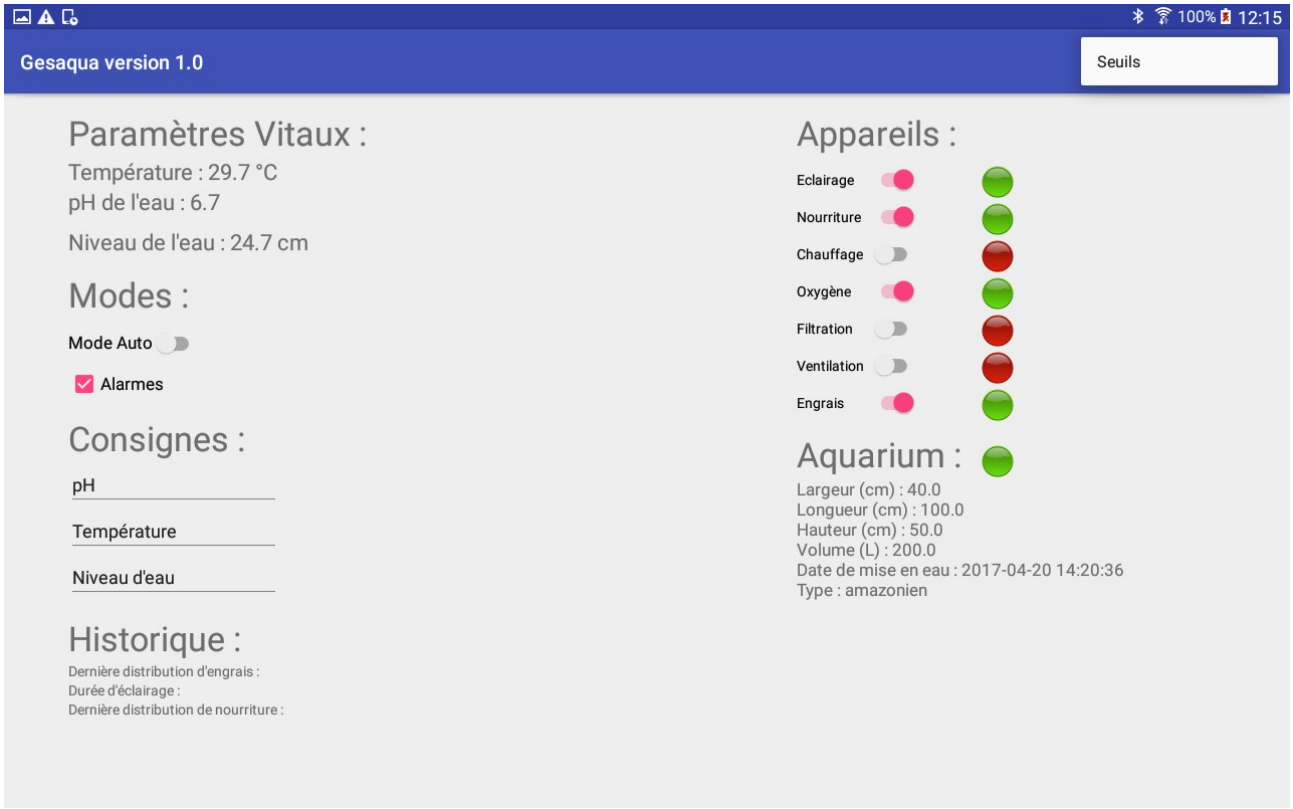
- chauffage : 0
- éclairage : 1
- distributeur de nourriture : 0
- distributeur d'engrais : 0
- oxygénation : 0
- filtration : 0
- ventilation : 0

Commander les appareils

Dans l'activité principale `MainActivity.xml` :

Pour commander un appareil, on lui envoie une trame *Bluetooth* de type `e`.
C'est l'utilisateur qui va pouvoir envoyer cette trame en cliquant sur l'interrupteur correspondant à l'appareil qu'il souhaite démarrer.

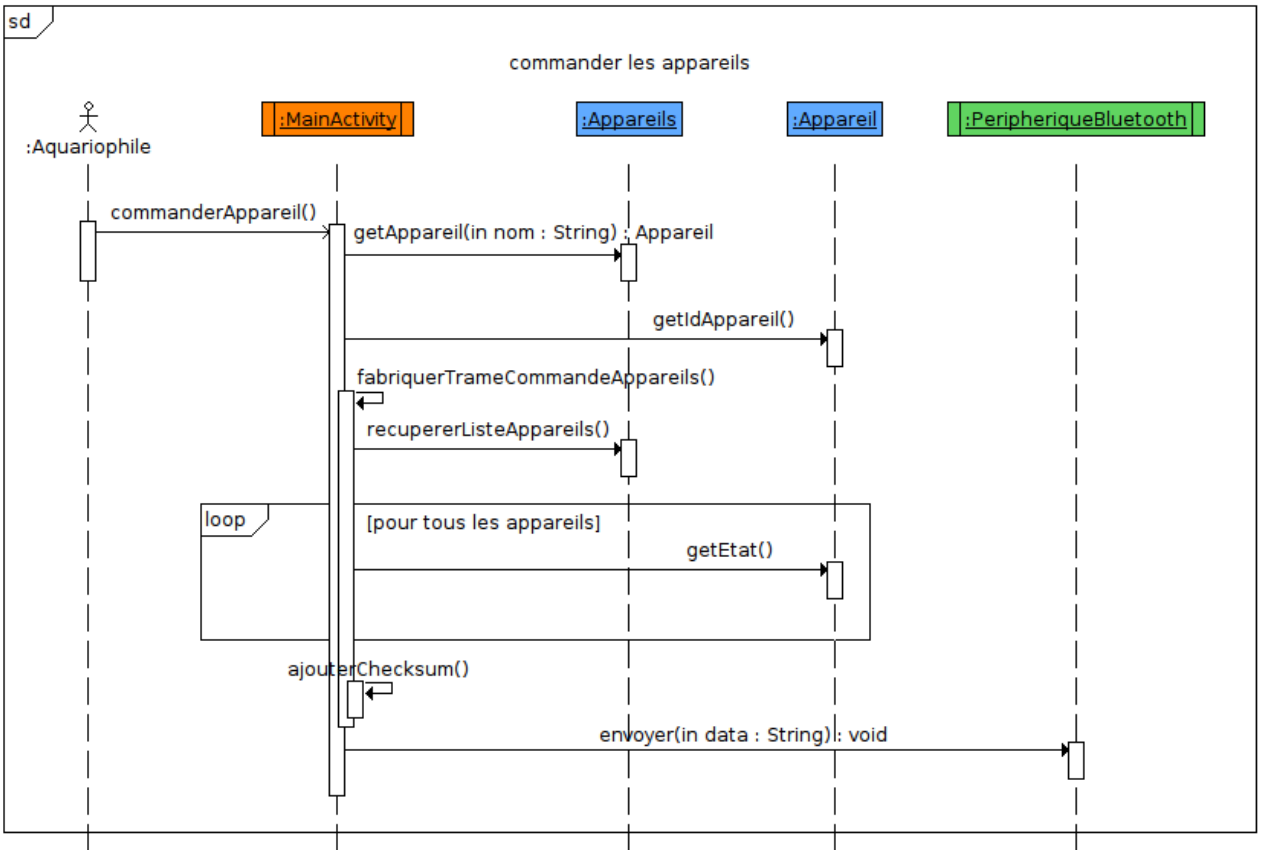
Ici, dans la partie `Appareils` de l'IHM, on voit que certains interrupteurs ont été activés. Il s'agit de l'éclairage, du distributeur de nourriture, de l'oxygène ainsi que du distributeur d'engrais.
Les interrupteurs se colorent lorsqu'ils sont activés par l'utilisateur.



Le diagramme de séquence suivant nous montre comment les appareils seront commandés. La trame sera envoyée à un objet `PeripheriqueBluetooth` puisque c'est cette classe qui assure le dialogue *Bluetooth* avec l'aquarium.

Une fois que l'utilisateur a activé l'interrupteur correspondant à l'appareil de son choix,

- une trame correspondant à l'appareil en question est fabriquée
- elle se base sur le nouvel état de l'appareil
- on envoie cette trame au périphérique *Bluetooth* pour communiquer l'ordre de la tablette vers l'aquarium.



Dans `MainActivity.xml` :

On relie l'interrupteur au `Handler` afin de pouvoir envoyer la trame à l'aquarium grâce à la connexion *Bluetooth*.

```
final private Handler handler = new Handler() {
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        if (msg.what == PeripheriqueBluetooth.CODE_CONNEXION) {
            activerCommandesAppareils();
}
```

```
private void activerCommandesAppareils()
{
    sw = (Switch) findViewById(R.id.SwitchEclairage);
    sw.setEnabled(true);
}
```

On fabrique la trame de commande et on l'envoie à l'aquarium :

C'est un signal, le fait d'activer ou de désactiver l'interrupteur, qui va envoyer la trame.

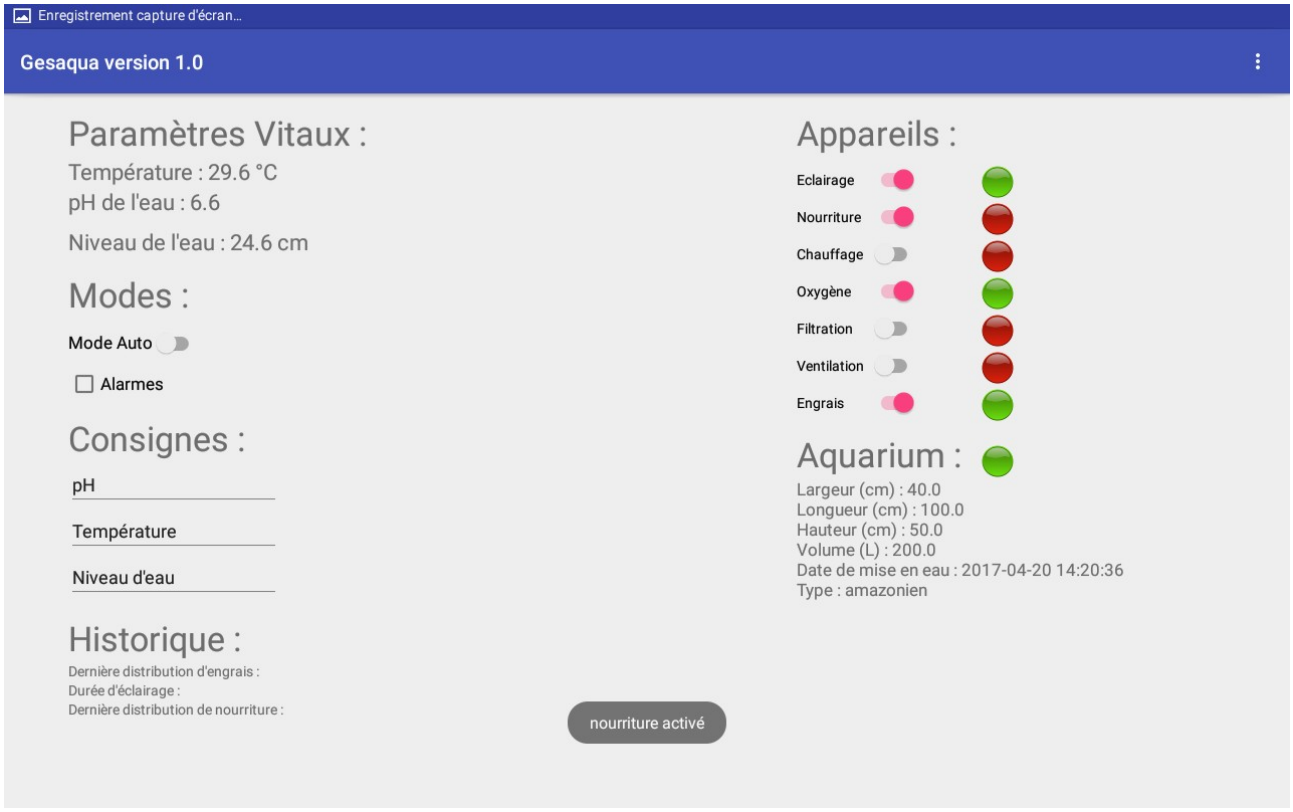
Lorsqu'un changement se produit au niveau de l'interrupteur gérant un appareil, son état change :

- 1 pour activé
- 0 pour désactivé

On fait une écoute des signaux de changement (1). Lorsqu'un changement se produit, l'état de l'appareil change. (2)

```
private void gererInterrupteurEclairage() {
    final Switch eclairage = (Switch) findViewById(R.id.SwitchEclairage);
    eclairage.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
        /**  permet de faire un bouton switch en créant un CompoundButton et
de.
         *  Tout est inclu dans la fonction setOnCheckedChangeListener.
         */
        public void onCheckedChanged(CompoundButton buttonView, boolean Check) {
            /** Verifie si le bouton est checké ou pas*/ //(1)
            if (Check) {
                commanderAppareil(NOM_APPAREIL_ECLAIRAGE, 1); //(2)
            } else {
                commanderAppareil(NOM_APPAREIL_ECLAIRAGE, 0); //(2)
            }
        }
    });
}
```

Ici l'interrupteur `nourriture` a été activé, mais la trame n'a pas encore été envoyée puisque l'image à sa droite a gardé la même couleur (ici rouge, qui signifie que l'appareil n'est pas activé).



Le changement est signalé par un nouvel état. C'est cet état qui est envoyé à la fonction qui va fabriquer la trame (1).

```
private void commanderAppareil(String nom, int etat) {
    if(!aquariumConnecte)
        return;
    Appareil appareil = appareils.getAppareil(nom);
    String trame = fabriquerTrameCommandeAppareils(appareil.getIdAppareil(),
etat); //(1)
    peripheriqueBluetooth.envoyer(trame);

    }
}
```

On fabrique la trame destinée aux appareils.

```
private String fabriquerTrameCommandeAppareils(int idAppareil, int etat) {
    String trame = TRAME_DEBUT + "e" + TRAME_DELIMITEUR;

    List<Appareil> listeAppareils = appareils.recupererListeAppareils();
//la trame contient tous les appareils
    for (int i = 0; i < listeAppareils.size(); i++)
    {
//permet d'obtenir l'état de fonctionnement de l'appareil
 // 1 si l'appareil est en état de marche (alimenté, connecté) et 0 sinon
```

```
        Appareil appareil = listeAppareils.get(i);
        if((i + 1) == idAppareil)
        {
            if ((i + 1) == listeAppareils.size())
                trame += etat;
            else
                trame += (etat + TRAME_DELIMITEUR);
            continue;
        }
        else
        {
            if ((i + 1) == listeAppareils.size())
                trame += appareil.getEtat();
            else
                trame += (appareil.getEtat() + TRAME_DELIMITEUR);
        }
    }
    trame = ajouterChecksum(trame);
    trame += TRAME_FIN;
    return trame;
}
```

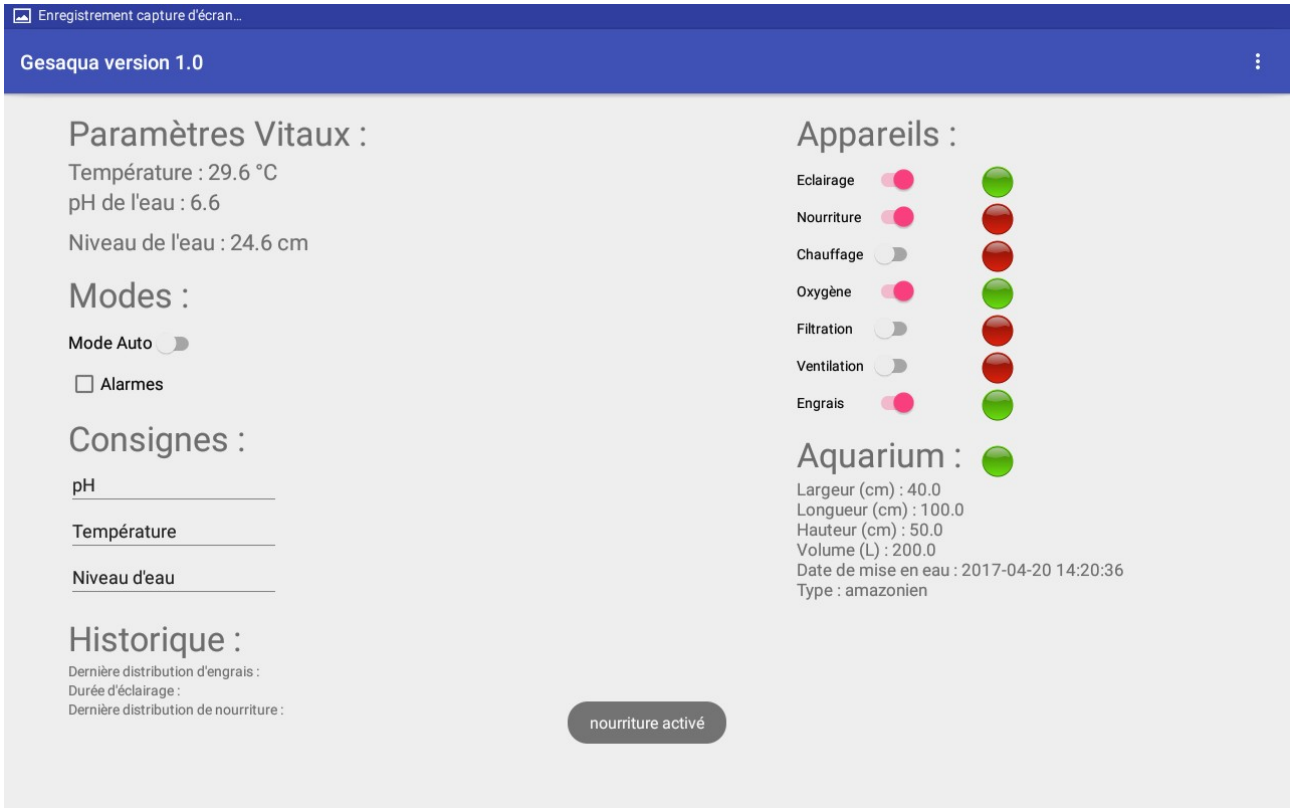
Puis cette trame est envoyée par un objet `PeripheriqueBluetooth`. (2)

```
private void commanderAppareil(String nom, int etat) {
    if(!aquariumConnecte)
        return;
    Appareil appareil = appareils.getAppareil(nom);
    String trame = fabriquerTrameCommandeAppareils(appareil.getIdAppareil(),
etat);
    peripheriqueBluetooth.envoyer(trame); //(2)

    }
}
```

Dans la classe `PeripheriqueBluetooth` :

On envoie la trame de la tablette vers l'aquarium : (3)

```
public void envoyer(String data)
{
    if(socket == null)
        return;
    try
    {
        sendStream.write(data.getBytes()); //(3)
        sendStream.flush();
    }
    //...
}
```



La trame est envoyée : l'image à côté de l'interrupteur se colore en vert car l'appareil est activé.

Visualiser les états et données

Pour visualiser les états et les données, il est nécessaire de les avoir enregistrés au préalable et donc de les avoir archivés dans une base de données.

Archiver les données

Configuration de la base de données[15]

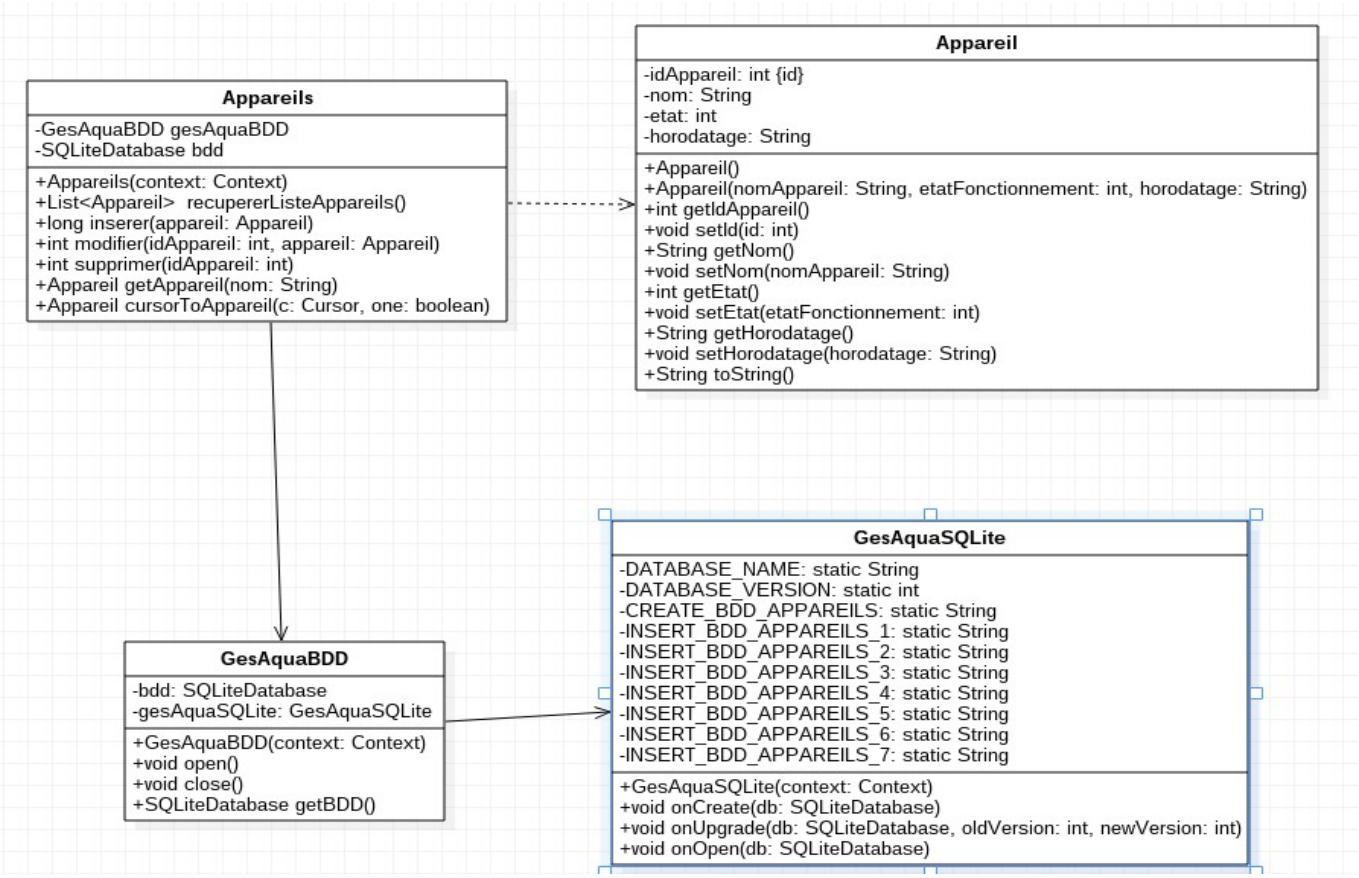
SQLite[16] est intégrée dans chaque appareil Android. C'est la technique dite *embedded SQL* : les instructions en langage SQL seront incorporées dans le code source.

Le *package* `android.database` contient toutes les classes nécessaires pour travailler avec des bases de données. Le *package* `android.database.sqlite` contient les classes spécifiques à *SQLite*.

Sur le diagramme de classe ci-dessous, on voit que pour créer une base de données il faut :

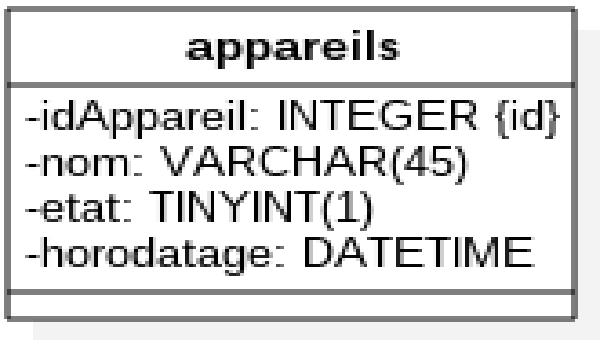
- Une classe dont chaque objet représentera un tuple dans la table
- Une classe pour manipuler les tuples : faire des insertions, suppressions ou mises à jour dans la base de données. Cette classe doit pouvoir manipuler des objets du même type que celui des tuples : ici c'est la classe `Appareils` qui permet de manipuler des tuples de type `Appareil`
- Une classe pour instancier la base de données : créer les tables et y entrer les enregistrements initiaux : `GesAquaSQLite`
- Une classe pour avoir accès à la base de données, qui permettra de l'ouvrir ou de la fermer : `GesAquaBDD`

15 Cf ANNEXE 2 pour plus de détails
16 SQLite : bibliothèque qui propose un moteur de base de données relationnelles


Appareil
-idAppareil: int {id}
-nom: String
-etat: int
-horodatage: String
+Appareil()
+Appareil(nomAppareil: String, etatFonctionnement: int, horodatage: String)
+int getIdAppareil()
+void setId(id: int)
+String getNom()
+void setNom(nomAppareil: String)
+int getEtat()
+void setEtat(etatFonctionnement: int)
+String getHorodatage()
+void setHorodatage(horodatage: String)
+String toString()
Appareils
-GesAquaBDD gesAquaBDD
-SQLiteDatabase bdd
+Appareils(context: Context)
+List<Appareil> recupererListeAppareils()
+long inserer(appareil: Appareil)
+int modifier(idAppareil: int, appareil: Appareil)
+int supprimer(idAppareil: int)
+Appareil getAppareil(nom: String)
+Appareil cursorToAppareil(c: Cursor, one: boolean)
GesAquaSQLite
-DATABASE_NAME: static String
-DATABASE_VERSION: static int
-CREATE_BDD_APPAREILS: static String
-INSERT_BDD_APPAREILS_1: static String
-INSERT_BDD_APPAREILS_2: static String
-INSERT_BDD_APPAREILS_3: static String
-INSERT_BDD_APPAREILS_4: static String
-INSERT_BDD_APPAREILS_5: static String
-INSERT_BDD_APPAREILS_6: static String
-INSERT_BDD_APPAREILS_7: static String
+GesAquaSQLite(context: Context)
+void onCreate(db: SQLiteDatabase)
+void onUpgrade(db: SQLiteDatabase, oldVersion: int, newVersion: int)
+void onOpen(db: SQLiteDatabase)
GesAquaBDD
-bdd: SQLiteDatabase
-gesAquaSQLite: GesAquaSQLite
+GesAquaBDD(context: Context)
+void open()
+void close()
+SQLiteDatabase getBDD()

Définition de la classe `Appareil` :

On va créer la table '**appareils**' dans la base de données `''gesaqua.db''`



On créé une classe `Appareil` avec ses attributs ainsi que des accesseurs pour pouvoir les lire ou écrire. Cette classe représente un tuple de la table '**appareils**'.

```
public class Appareil
{
//Les attributs représentent les champs qui seront dans la table.
    private int idAppareil;
    private String nom;
    private int etat;
    private String horodatage;

//accesseurs
public String getNom()
{
    return nom;
}

public void setNom(String nomAppareil)
{
    this.nom = nomAppareil;
}
//...
}
```

Définition d'une classe `Appareils` :

Elle manipule des objets de type `Appareil`.

On va donc créer une classe `Appareils` afin de pouvoir exécuter des requêtes SQL sur la table '**appareils**', telles que `insert()` (1) ou `update()`.(2)

```
public class Appareils {
    private GesAquaBDD gesAquaBDD;
    private SQLiteDatabase bdd;
public Appareils(Context context)
{
    gesAquaBDD = new GesAquaBDD(context);
    bdd = gesAquaBDD.getBDD();
}
//...

//Insertion d'un tuple
public long inserer(Appareil appareil)
{
    ContentValues values = new ContentValues();
    values.put("nom", appareil.getNom());
    values.put("etat", appareil.getEtat());
    values.put("horodatage", appareil.getHorodatage());
    return bdd.insert("appareils", null, values); //(1)
}

//Mise à jour d'un tuple
public int modifier(int idAppareil, Appareil appareil)
{
    ContentValues values = new ContentValues();
    values.put("nom", appareil.getNom());
    values.put("etat", appareil.getEtat());
    values.put("horodatage", appareil.getHorodatage());
    return bdd.update("appareils", values, "idAppareil = " + idAppareil, null);
//(2)
}
}
```

Définition de la classe `GesAquaSQLite` :

Pour créer et mettre à jour une base de données *SQLite* dans une application Android, on doit créer une classe qui hérite de `SQLiteOpenHelper`[17].

17 SQLiteOpenHelper : Classe qui permet de gérer la création ainsi que la version de base de données.

Cette classe se charge d'ouvrir la base de données si elle existe ou de la créer si elle n'existe pas, et de pouvoir la mettre à jour si nécessaire. On s'en sert pour définir l'ensemble des tables de la base de données qui seront produites lors de l'instanciation.

On implémente les méthodes `onCreate()` et `onUpgrade()` :

- `onCreate()` : pour créer une base de données qui ne l'est pas encore
- `onUpgrade()` : si la version de la base de données évolue, cette méthode permettra de mettre à jour le schéma de base de données existant ou de supprimer la base de données existante et la recréer par la méthode `onCreate()`.

```
public class GesAquaSQLite extends SQLiteOpenHelper()
{
// Pour création des tables
private static final String CREATE_BDD_APPAREILS =
        "CREATE TABLE appareils (" +
                "idAppareil INTEGER PRIMARY KEY NOT NULL ," +
                "nom VARCHAR(45) NULL ," +
                "etat TINYINT(1) NULL ," +
                "horodatage DATETIME );"; //DATETIME : format "YYYY-MM-DD
HH:MM:SS"
// Pour insertion des données initiales
private static final String INSERT_BDD_APPAREILS_1 = "INSERT INTO appareils
VALUES(1,'chauffage',0,'2017-03-24 16:31:10');";
private static final String INSERT_BDD_APPAREILS_2 = "INSERT INTO appareils
VALUES(2,'eclairage',0,'2017-03-24 16:31:20');";
private static final String INSERT_BDD_APPAREILS_3 = "INSERT INTO appareils
VALUES(3,'nourriture',0,'2017-03-24 16:31:20');";
private static final String INSERT_BDD_APPAREILS_4 = "INSERT INTO appareils
VALUES(4,'engrais',0,'2017-03-24 16:31:20');";
private static final String INSERT_BDD_APPAREILS_5 = "INSERT INTO appareils
VALUES(5,'oxygenation',0,'2017-03-24 16:31:20');";
private static final String INSERT_BDD_APPAREILS_6 = "INSERT INTO appareils
VALUES(6,'filtration',0,'2017-03-24 16:31:20');";
private static final String INSERT_BDD_APPAREILS_7 = "INSERT INTO appareils
VALUES(7,'ventilation',0,'2017-03-24 16:31:20');"
//Création d'une base de données et insertion des données initiales
public void onCreate(SQLiteDatabase db)
{
    // 1. création des tables
    db.execSQL(CREATE_BDD_APPAREILS);

    // 2. insertion des données initiales
    db.execSQL(INSERT_BDD_APPAREILS_1);
    db.execSQL(INSERT_BDD_APPAREILS_2);
    db.execSQL(INSERT_BDD_APPAREILS_3);
    db.execSQL(INSERT_BDD_APPAREILS_4);
    db.execSQL(INSERT_BDD_APPAREILS_5);
    db.execSQL(INSERT_BDD_APPAREILS_6);
    db.execSQL(INSERT_BDD_APPAREILS_7);
}
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
```

```
{
    // on supprime la table puis on la recrée
    db.execSQL("DROP TABLE appareils;");
}
```

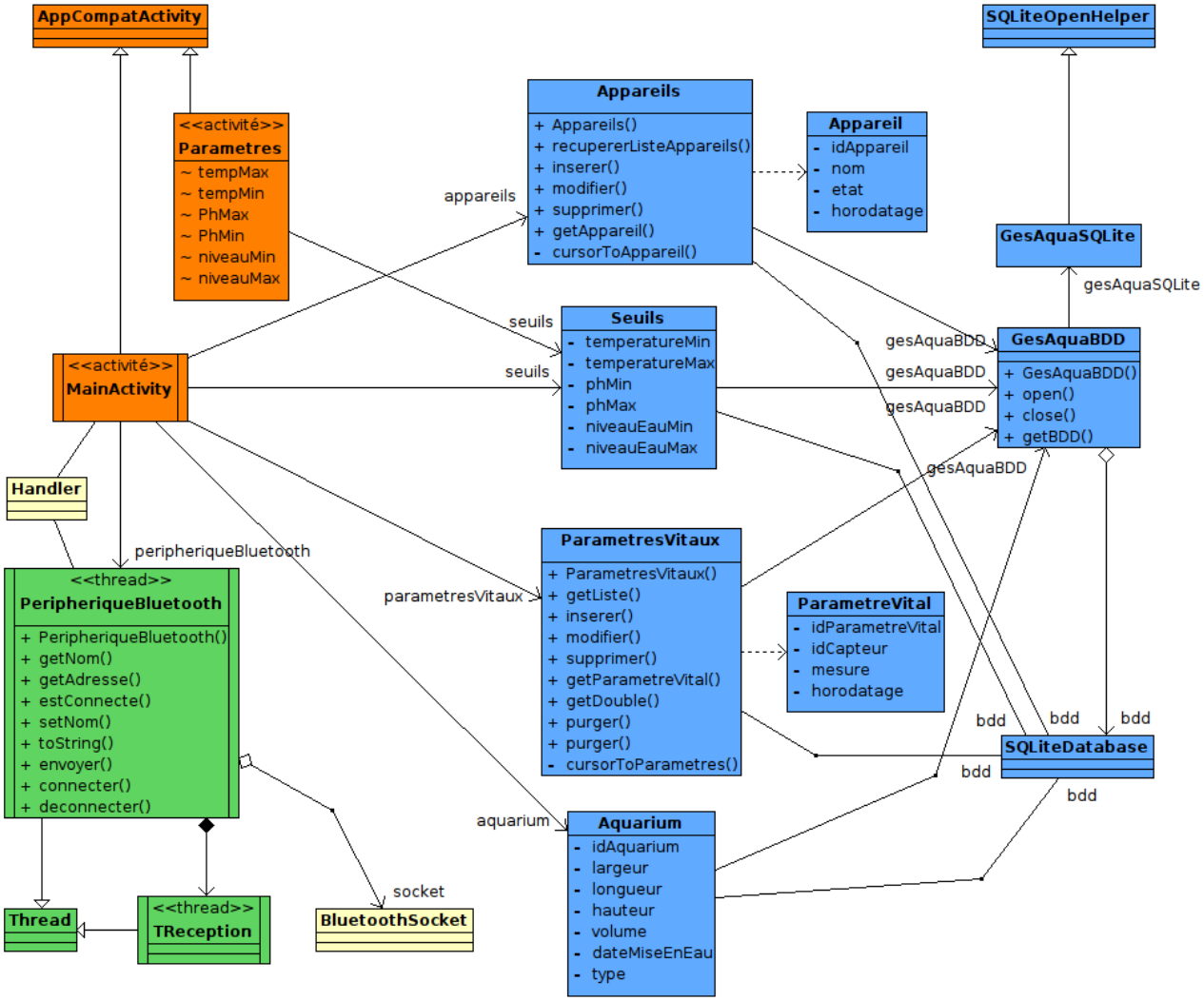
Définition de la classe `GesAquaBDD` :

Pour accéder à la base de données de l'application, on utilise `SQLiteDatabase`, qui est la classe de base pour travailler avec une base de données *SQLite* sous Android et fournit des méthodes pour ouvrir, effectuer des requêtes, mettre à jour et fermer la base de données. On crée donc une classe `GesAquaBDD`, qui hérite donc de `SQLiteDatabase`, pour accéder à la base de données de l'application.

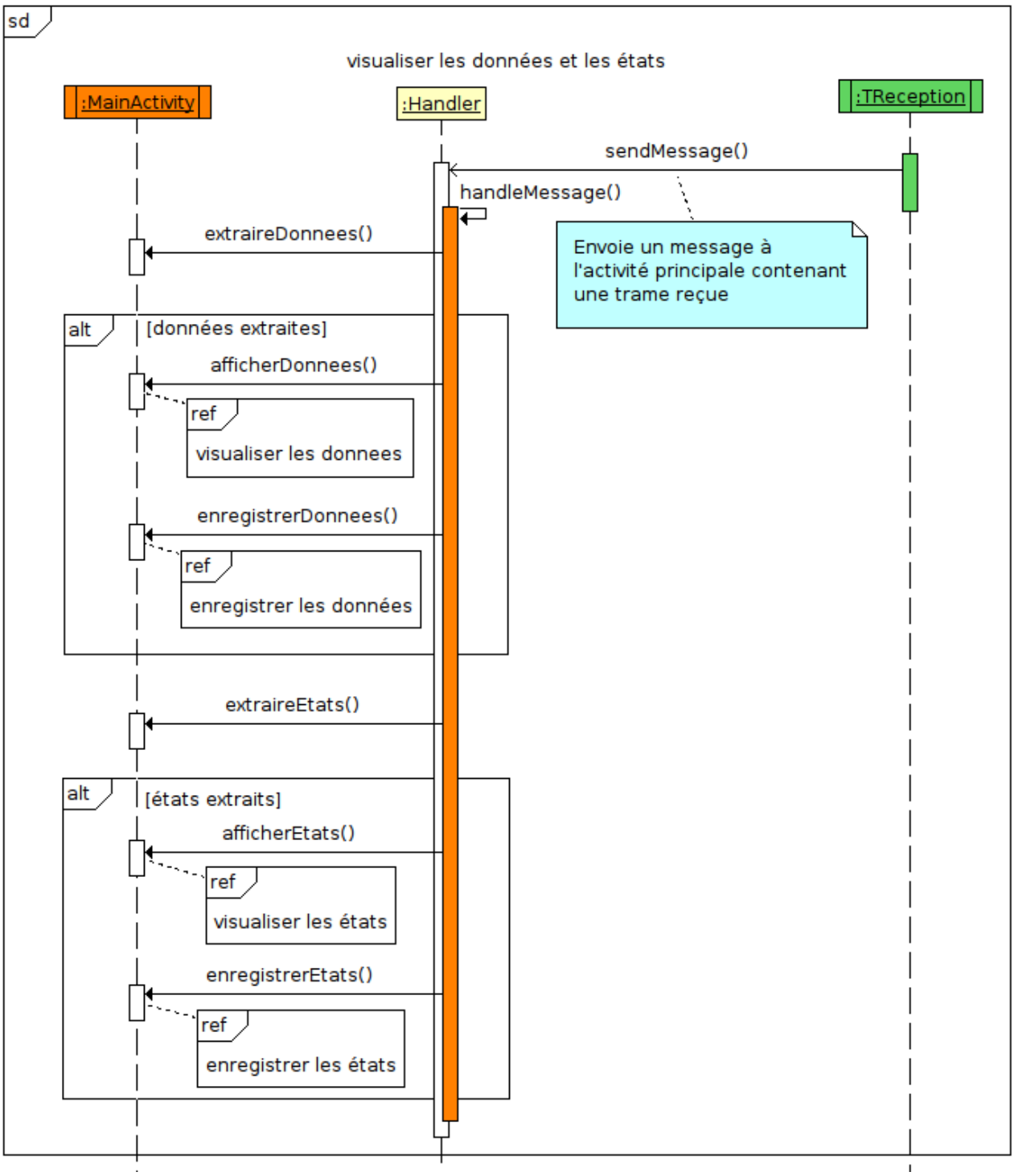
```
public class GesAquaBDD {

//...

//Ouverture de la base de données en écriture
public void open()
{
    if (bdd == null)
        bdd = gesAquaSQLite.getWritableDatabase();
}
//Fermeture
public void close()
{
    if (bdd != null)
        if (bdd.isOpen())
            bdd.close();
}
//Accès à la base de données
public SQLiteDatabase getBDD()
{
    if (bdd == null)
        open();
    return bdd;
}
}
```

La base de données est maintenant crée.

On a le diagramme de classes suivant :



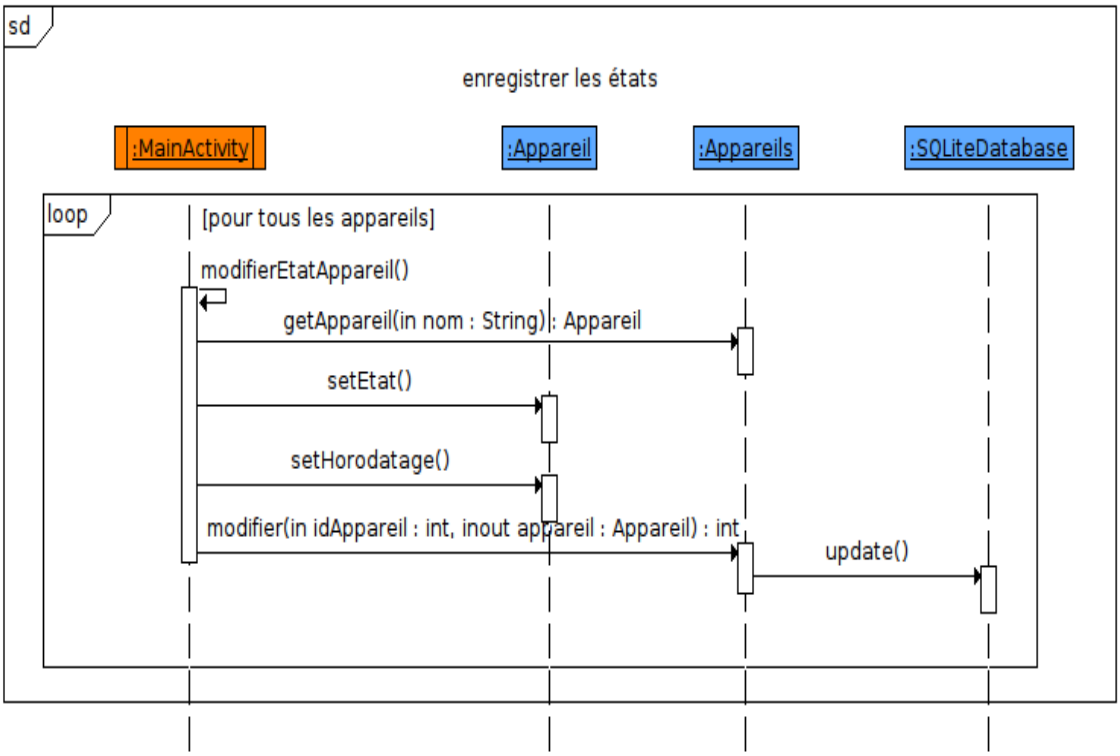
C'est un `Handler` qui fait le lien entre la page principale de l'application et les messages reçus par `Treception` comme on peut le voir sur le diagramme de séquence suivant.
Les états et les données contenus dans les messages du `Handler` sont extraits de la trame au niveau de l'activité principale `MainActivity`. De là, on peut les visualiser et les enregistrer dans la base de données.


sd
visualiser les données et les états
:MainActivity
:Handler
:TReception
sendMessage()
handleMessage()
extraireDonnees()
Envoie un message à l'activité principale contenant une trame reçue
alt
[données extraites]
afficherDonnees()
ref
visualiser les donnees
enregistrerDonnees()
ref
enregistrer les données
extraireEtats()
alt
[états extraits]
afficherEtats()
ref
visualiser les états
enregistrerEtats()
ref
enregistrer les états

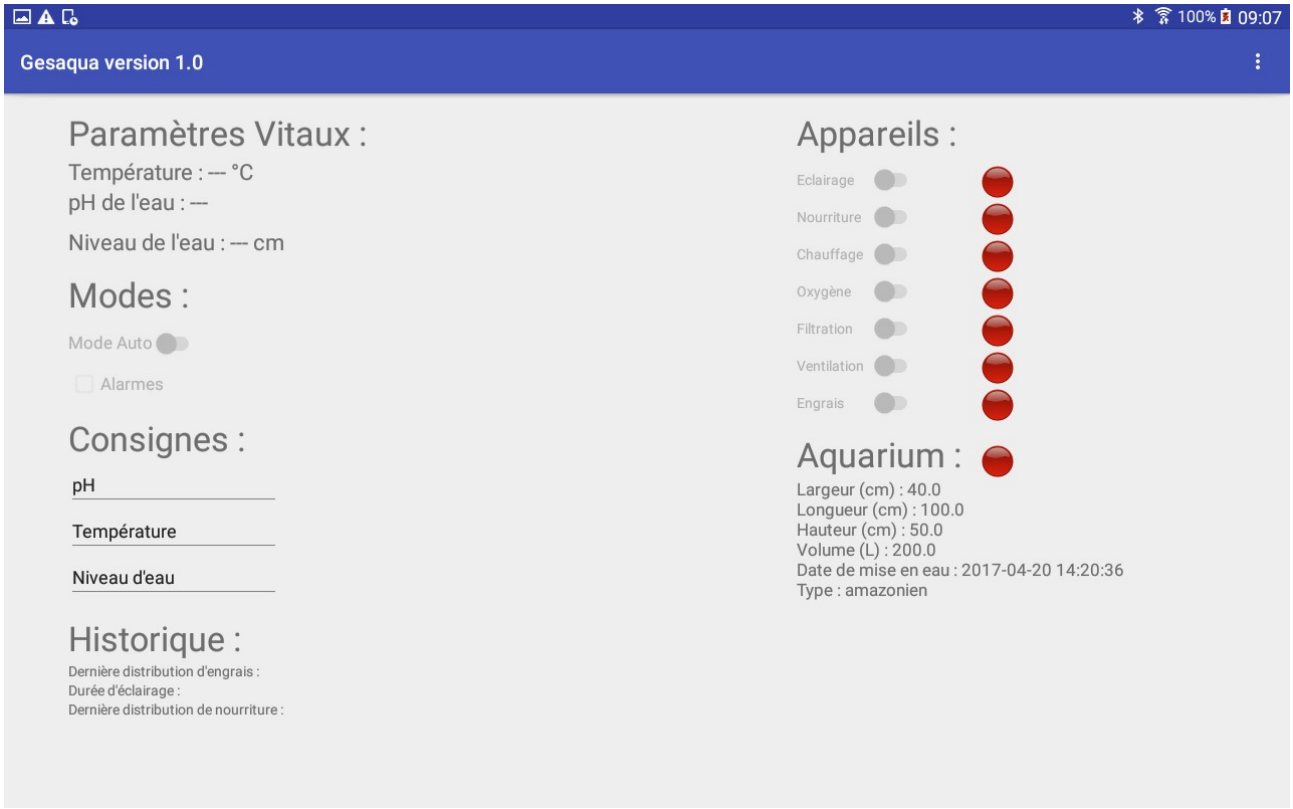
Enregistrer les états :

Les états des appareils sont enregistrés dans la base de données.

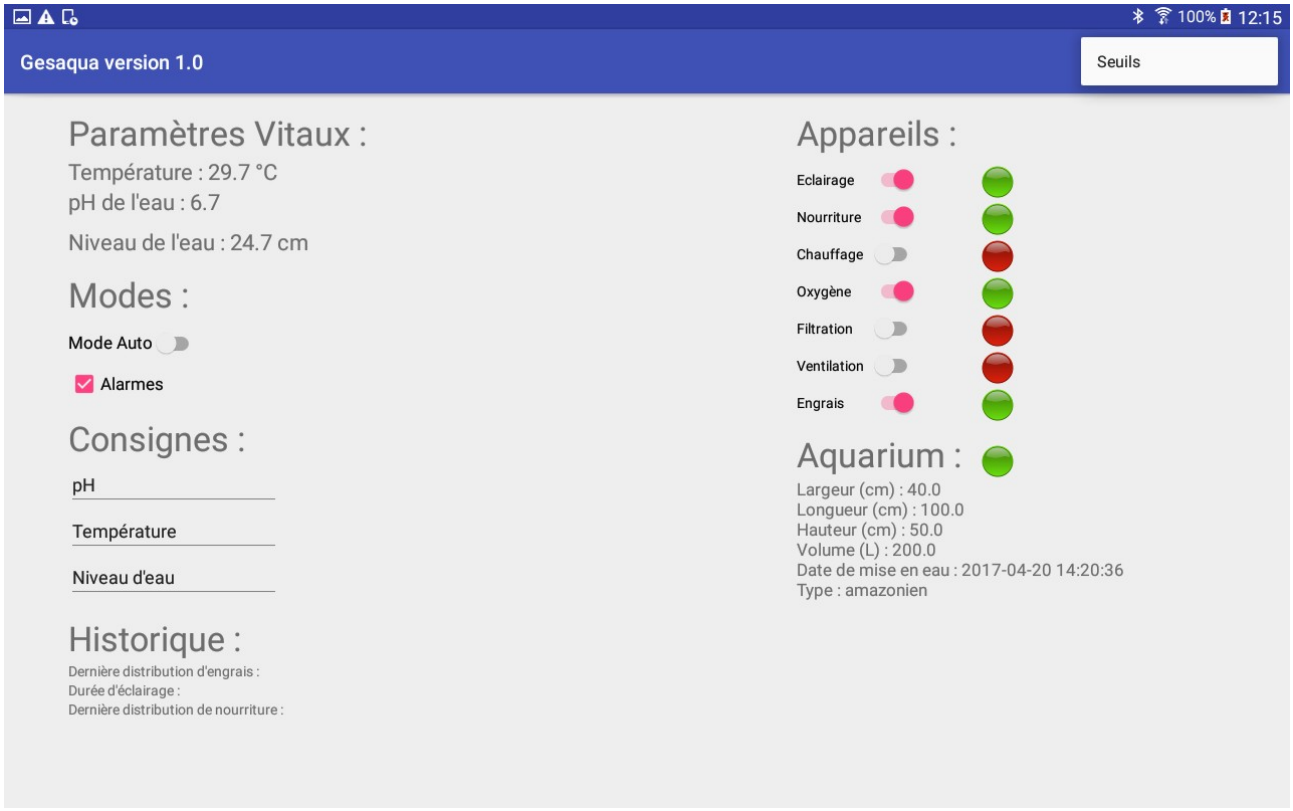
Sur le diagramme de séquence ci-dessous, on peut voir que les états des appareils sont modifiés un par un : on récupère les différents appareils avant de modifier leur état. Puis on met à jour la base de données avec ces nouveaux états.



Sur cette capture d'IHM on peut voir que les appareils sont tous désactivés par défaut.
Puis on va pouvoir les activer un par un en cliquant sur les interrupteurs correspondants.



Ici, on peut voir qu'en cliquant sur les interrupteurs, les appareils correspondants s'activent : ici il s'agit de l'éclairage, du distributeur de nourriture de l'oxygénation ainsi que du distributeur d'engrais.



Pour enregistrer l'état de l'appareil :

Dans le fichier `MainActivity.xml` :

Le nouvel état obtenu en cliquant modifie l'état de l'interrupteur. (1)

```
private void enregistrerEtats()
{
  if(etatEclairage != etatEclairagePrec)
        modifierEtatAppareil(APPAREIL_ECLAIRAGE, NOM_APPAREIL_ECLAIRAGE,
etatEclairage, R.id.imageViewEclairage, R.id.SwitchEclairage); //(1)
}
```

Lorsque l'état d'un appareil est modifié par l'utlisateur, on modifie également son état dans la base de données grâce à la méthode `modifier()` de la classe `Appareils` : (2)

```
private void modifierEtatAppareil(int idAppareil, String nomAppareil, int etat,
int idImageView, int idSwitch)
{
    Appareil appareil = appareils.getAppareil(nomAppareil);
    appareil.setEtat(etat);
    appareil.setHorodatage(getHorodatageBD());
    appareils.modifier(idAppareil, appareil); //(2)
}
```

Dans la classe `Appareils` :

On met à jour le tuple dans la base de données (3) : l'état de l'appareil est donc enregistré.

```
public int modifier(int idAppareil, Appareil appareil)
{
    ContentValues values = new ContentValues();
    values.put("nom", appareil.getNom());
    values.put("etat", appareil.getEtat());
    values.put("horodatage", appareil.getHorodatage());
    return bdd.update("appareils", values, "idAppareil = " + idAppareil, null);
//(3)
}
```

Enregistrer les données :

Les données à enregistrer sont liées aux états puisqu'il s'agit des dernières distribution d'engrais, de nourriture ainsi que de la durée d'éclairage.

Pour enregistrer les données, on extrait les données de la trame reçue par *Bluetooth*, donc on extrait les états des appareils de la trame de type `e` .

C'est grâce à la fonction `enregistrerEtat()` que l'on modifie l'état de l'appareil dans la base de données. On ne modifie cet état que s'il est différent de l'état précédent.

C'est grâce à la modification de cet état que l'on va pouvoir modifier les données, puisqu'en même temps qu'un tuple est enregistré, son champs `horodatage` est modifié. (1)

```
private void enregistrerEtats()
{
    if(etatEngrais != etatEngraisPrec)
       modifierEtatAppareil(APPAREIL_ENGRAIS, NOM_APPAREIL_ENGRAIS, etatEngrais,
R.id.imageViewEngrais, R.id.SwitchEngrais);
}
```

```
private void modifierEtatAppareil(int idAppareil, String nomAppareil, int etat,
int idImageView, int idSwitch)
{
    Appareil appareil = appareils.getAppareil(nomAppareil);
    appareil.setEtat(etat);
    appareil.setHorodatage(getHorodatageBD());
    appareils.modifier(idAppareil, appareil);
}
```

On effectue un `update()` dans la base de données.

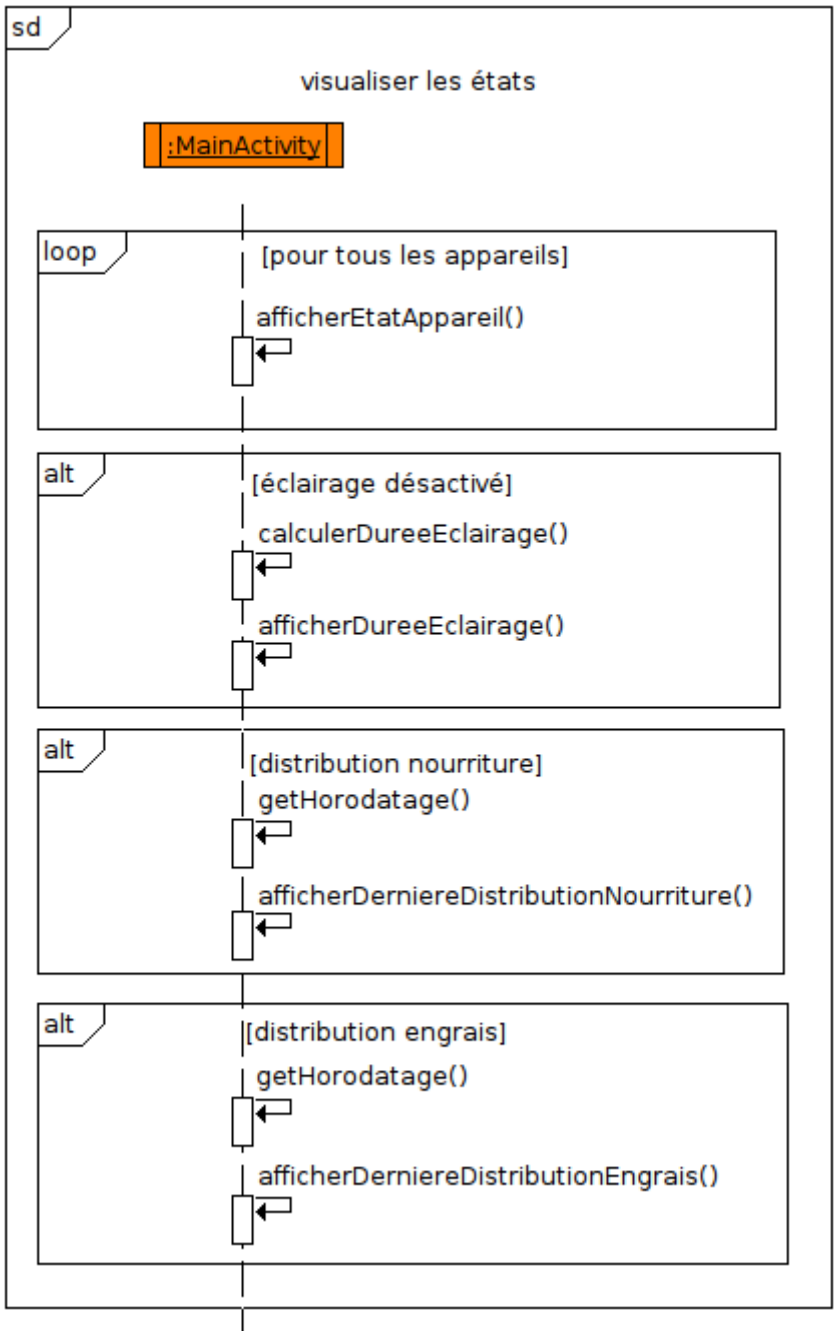
```
public int modifier(int idAppareil, Appareil appareil)
{
    ContentValues values = new ContentValues();
    values.put("nom", appareil.getNom());
    values.put("etat", appareil.getEtat());
    values.put("horodatage", appareil.getHorodatage()); //(1)
    return bdd.update("appareils", values, "idAppareil = " + idAppareil, null);
}
```

Informer l'utilisateur

Visualiser les états :

On visualise les états des appareils grâces à des d'images (on/off). Lorsque l'image est rouge, l'appareil est désactivé. Si elle est verte, alors l'appareil est activé.
Ce sont ces images qui attestent que la trame a bien été transmise puisque leur couleur va changer. Ainsi, l'utilisateur est informé.

Afin de visualiser les états des appareils, on peut voir sur le diagramme de séquence ci-dessous qu'un affichage suffit. Et pour ce qui est des données, on affiche la dernière distribution, donc la dernière fois que l'interrupteur a été activé.



Au démarrage de l'application on récupère les états des appareils. A ce stade, ils sont tous désactivés par défaut.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    recupererValeurs();
}
```

On récupère l'état de l'appareil puis on l'affiche dans l'IHM (1).

```
protected void recupererValeurs()
{
    appareils = new Appareils(getApplicationContext());

    /* les états des appareils */
    Appareil appareil;
    Switch sw;

    appareil = appareils.getAppareil(NOM_APPAREIL_ECLAIRAGE);
    etatEclairage = appareil.getEtat();
    etatEclairagePrec = etatEclairage;
    afficherEtatAppareil(APPAREIL_ECLAIRAGE, NOM_APPAREIL_ECLAIRAGE,
etatEclairage, R.id.imageViewEclairage, R.id.SwitchChauffage); //(1)
    sw = (Switch) findViewById(R.id.SwitchEclairage);
    if(etatEclairage == 1)
        sw.setChecked(true);
    else
        sw.setChecked(false);

}
```

On affiche le nouvel état de l'appareil au niveau de l'interrupteur.
Cet affichage se fait par l'intermédiaire de l'interrupteur qui sera activé si l'appareil est activé (donc son état = 1) ou désactivé si l'appareil est désactivé (donc son état = 0). (2)

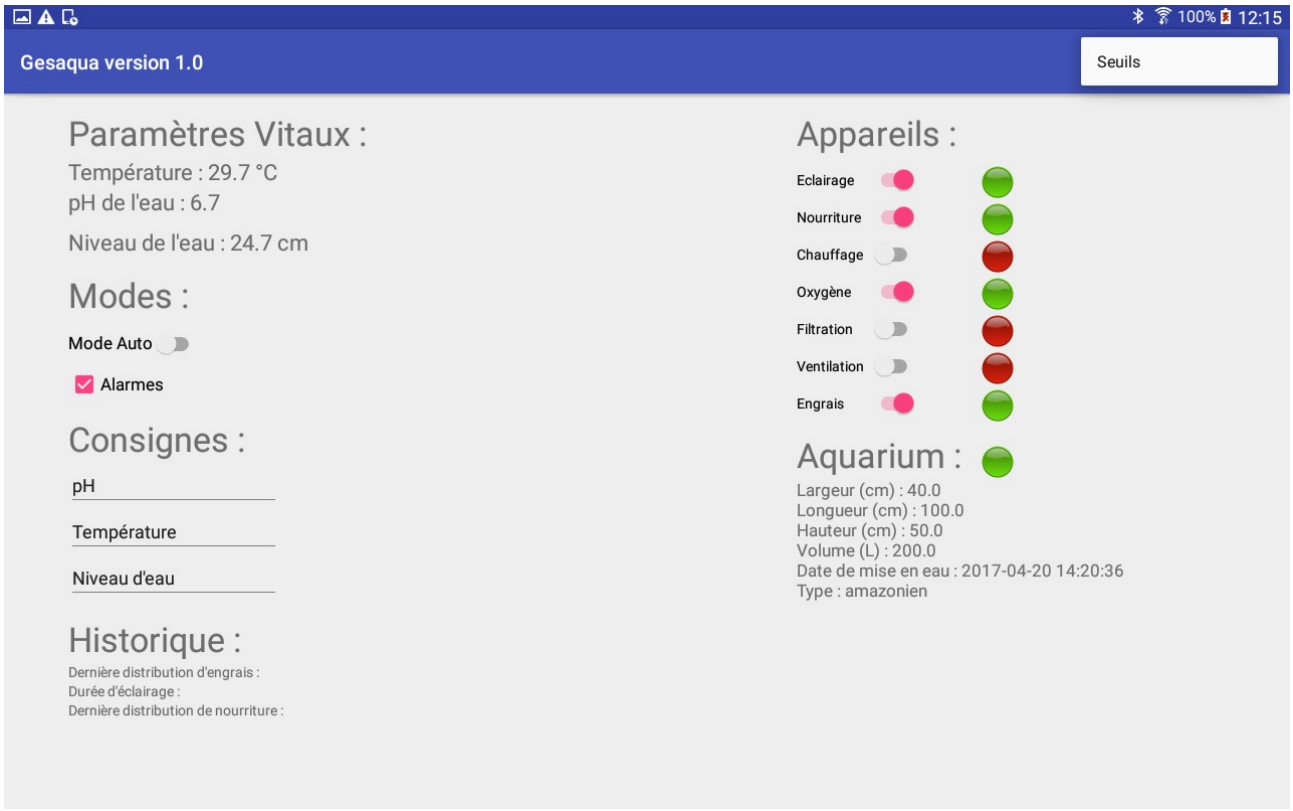
```
private void afficherEtats()
{

    if(etatEclairage != etatEclairagePrec)
        afficherEtatAppareil(APPAREIL_ECLAIRAGE, NOM_APPAREIL_ECLAIRAGE,
etatEclairage, R.id.imageViewEclairage, R.id.SwitchEclairage); //(2)

    }
```

Les images vertes ou rouges se colorent en fonction de l'état de l'appareil (3) : rouge si désactivé et verte si activé.

```
private void afficherEtatAppareil(int idAppareil, String nomAppareil, int etat,
int idImageView, int idSwitch)
{

    ImageView img = (ImageView) findViewById(idImageView);
    Switch sw = (Switch) findViewById(idSwitch);
    if (etat == 1) //(3)
    {
        img.setImageResource(R.drawable.on);
        sw.setChecked(true);
    }
    else
```

```
    {
        img.setImageResource(R.drawable.off);
        sw.setChecked(false);
    }
}
```

Sur cette capture d'écran de l'IHM, on peut voir que 4 appareils ont été activés : l'éclairage, la distribution de nourriture, l'oxygène et la distribution d'engrais.



Visualiser les données

Les données dont je me suis occupée sont :

- la dernière distribution d'engrais
- la dernière distribution de nourriture
- la durée d'éclairage en minutes

Ces données sont consignées dans la partie `Historique` de l'IHM.

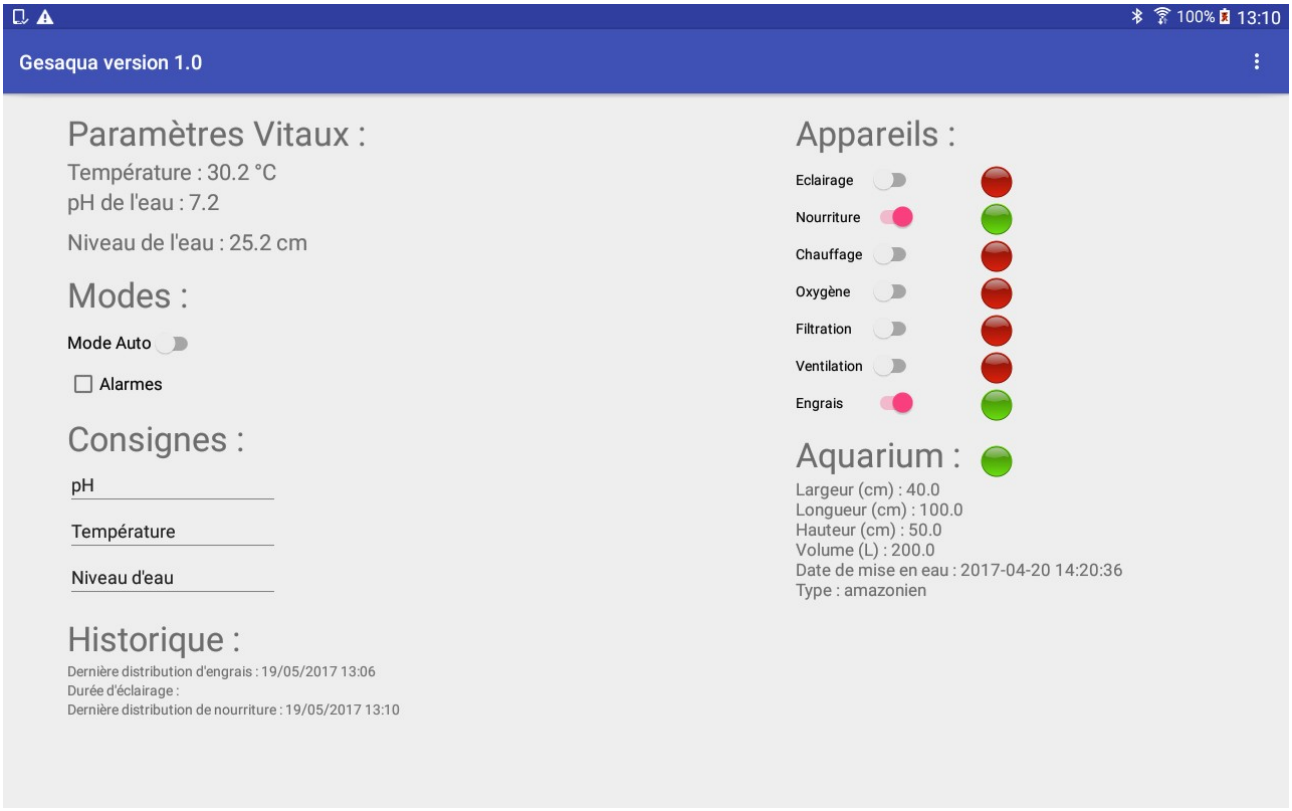
On récupére l'horodatage de la dernière distribution d'engrais : (1)

```
private void afficherEtats()
{
    if(etatEngraisPrec != etatEngrais && etatEngrais == 1)
    {
        dernieredistributionEngrais = getHorodatage(); //(1)
        afficherDernieredistributionEngrais();
    }
}
```

Puis on affiche le résultat : (2)

```
protected void afficherDernieredistributionEngrais()
{
    ((TextView)
findViewById(R.id.dernieredistributionEngrais)).setText("Dernière distribution
d'engrais : " + dernieredistributionEngrais); //(2)
}
```

Sur cette capture d'écran d'IHM, on peut voir le résultat en bas à gauche au niveau de la partie `Historique`.



Plan des tests de validation

DESCRIPTION	OUI	NON
Les données et les alarmes associées au module sont affichées sur la tablette tactile	X	
La commande des appareils associés au module est possible à partir de la tablette tactile	X	
Le paramétrage du mode automatisé est réalisable		
L'affichage de la durée actuelle d'éclairage en minutes est réalle dans l'IHM		
L'affichage de la dernière distribution de nourriture est visible sur l'écran tactile	X	
L'affichage de la dernière distribution d'engrais est visible sur l'écran tactile	X	

Annexes

Table des matières - ANNEXES

Annexe 1 : Mise en oeuvre de la liaison *Bluetooth* :

Introduction

Le *Bluetooth* est un standard de communication permettant l'échange bidirectionnel de données à très courte distance en utilisant des ondes radio UHF[18] sur une bande de fréquence de 2,4 GHz.

La Communication Bluetooth

Android Studio permet l'accès aux fonctionnalités *Bluetooth* via l'*API Bluetooth* d'Android afin de permettre l'échange de données entre la tablette et d'autres appareils possédant une liaison *Bluetooth*.

Fonctionnement

Afin que des périphériques puissent se transmettre des données, ils doivent d'abord s'appairer.

- Etre **appairé** signifie que 2 périphériques « connaissent » leur existence respective, partagent une clé pour leur authentification et ont la capacité d'établir une connexion entre eux.
- Etre **connecté** signifie que les 2 périphériques partagent un canal RFCOMM[19] et sont capables de se transmettre des données.

Il est nécessaire d'être appairé à un périphérique avant de pouvoir établir une communication via un canal RFCOMM. L'appairage est fait de façon automatique lorsque la connexion est initialisée via une API *Bluetooth* Android.

La connexion se fait via une socket. Le client et le serveur sont considérés comme connectés lorsqu'ils ont tous deux une socket de type `BluetoothSocket` connectée sur le même canal RFCOMM.

Après que l'appairage et la connexion aient été effectués, les 2 périphériques peuvent échanger des données de façon bidirectionnelle.

Activer le *Bluetooth* sur la tablette

Permissions *Bluetooth*

Afin de pouvoir utiliser le *Bluetooth* dans l'application, il est nécessaire d'avoir la permission

18 Ultra Hautes Fréquences
19 Cf Glossaire

de le faire, et donc de déclarer cette permission pour pouvoir établir une communication.

La permission standard est `BLUETOOTH`, mais la plupart des applications ont besoin d'une permission supplémentaire pouvoir initier la recherche d'autres périphériques *Bluetooth* : `BLUETOOTH_ADMIN`.

On déclare ces permissions dans le fichier `AndroidManifest.xml` :

```
<manifest>
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
</manifest>
```

Mise en oeuvre

Au démarrage de l'application, on active le *Bluetooth* :

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

//...
    activerBluetooth();

}
```

On suit les étapes suivantes :

- Rendre le périphérique visible (1)
- Rechercher de nouveaux périphériques (2)
- Rechercher parmi les périphériques déjà appairés (3)
- Connecter les périphériques (4)

Dans MainActivity.xml :

```
private BluetoothAdapter mBluetoothAdapter =
BluetoothAdapter.getDefaultAdapter();  //(1)
private Set<BluetoothDevice> pairedDevices =
mBluetoothAdapter.getBondedDevices(); //(3)
private PeripheriqueBluetooth peripheriqueBluetooth; //(4)
```

```
protected void activerBluetooth()
{
```

```
    // A-t-on un adaptateur bluetooth ?
    if (mBluetoothAdapter == null) {
        Toast.makeText(getApplicationContext(), "Bluetooth non présent !",
Toast.LENGTH_SHORT).show();
    } else if (!mBluetoothAdapter.isEnabled()) { // le bluetooth est-il
désactivé ?
        // Demande de l'activation du bluetooth
        Intent discoverableIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);  //(1)

discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
        startActivity(discoverableIntent);
    }
else {
        // le bluetooth est déjà activé
    }
    // Recherche des périphériques  //(2)
    IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
    registerReceiver(mReceiver, filter);
    mBluetoothAdapter.startDiscovery();
    BluetoothDevice myDevice = null;
    peripheriqueBluetooth = null;
    // Pour tous les périphériques bluetooth appairés
    for (BluetoothDevice device : pairedDevices) {       //(3)
        // le périphérique bluetooth de l'aquarium ?
        if (device.getName().equals("HC-05")) { //Simulateur

            myDevice = device;
            peripheriqueBluetooth = new PeripheriqueBluetooth(myDevice,
handler);
            break; // on peut sortir
        }
    }
    if (peripheriqueBluetooth != null)
    {
        peripheriqueBluetooth.connecter();   //(4)
    }
}
```

Rendre le périphérique visible

L'adaptateur Bluetooth

Avant que l'application puisse communiquer via une liaison *Bluetooth*, il faut vérifier que le *Bluetooth* est supporté par le périphérique, et s'assurer qu'il est activé.
Si le *Bluetooth* est désactivé, il est possible de l'activer sans devoir quitter l'application, à l'aide d'un objet `BluetoothAdapter`.

L'utilisation de `BluetoothAdapter` est requise pour toute utilisation du *Bluetooth* au sein de l'application. (1)
Pour l'avoir, on utilise la méthode statique `getDefaultAdapter()` . Elle retourne un objet de tye `BluetoothAdapter` qui représente l'adaptateur *Bluetooth* de la tablette elle-même. Il n'y a qu'un seul adaptateur *Bluetooth* pour tout le système, et l'application intéragit avec lui via l'objet de type `BluetoothAdapter`.
Dans le cas où `getDefaultAdapter()` retourne **null**, cela veut dire que le périphérique ne supporte pas le Bluetooth, il ne pourra donc pas utiliser ce mode de communication.

Dans `MainActivity.xml` :

```
private BluetoothAdapter mBluetoothAdapter =
BluetoothAdapter.getDefaultAdapter();  //(1)
```

La visibilité

Les périphériques Android ne sont pas visibles par défaut et un périphérique *Bluetooth* repond à une recherche de périphérique uniquement s'il est visible. Il faut donc rendre notre tablette visible. (2)

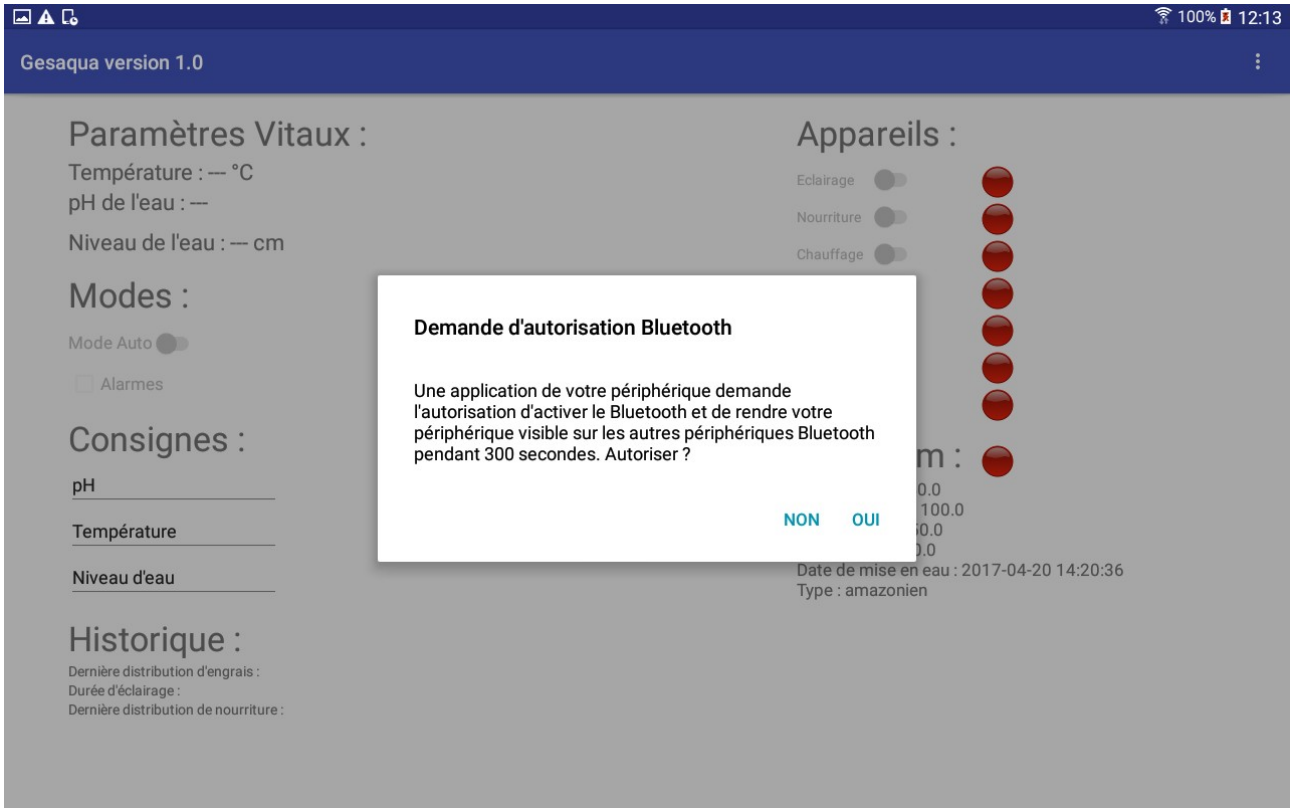
Si le *Bluetooth* n'a pas été activé sur la tablette avant de lancer l'application, le fait de la rendre visible pour les autres périphériques *Bluetooth* activera automatiquement le *Bluetooth* sur la tablette.
Grâce à `EXTRA_DISCOVERABLE_DURATION`, on peut régler une durée de visibilité car elle est de 120 secondes par défaut.
Ici on rend le périphérique visible pendant 5 minutes (soit 300 secondes). (3)

Dans `MainActivity.xml` :

```
protected void activerBluetooth()
{
    // A-t-on un adaptateur bluetooth ?
    if (mBluetoothAdapter == null) {  //(1)
        Toast.makeText(getApplicationContext(), "Bluetooth non présent !",
Toast.LENGTH_SHORT).show();
    } else if (!mBluetoothAdapter.isEnabled()) { // le bluetooth est-il
désactivé ?
        // Demande de l'activation du bluetooth
        Intent discoverableIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);  //(2)

discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
        startActivity(discoverableIntent); //(3)
    }
else {
```

```
        // le bluetooth est déjà activé
      }
//...

}
```

Une fenêtre de dialogue s'ouvre afin de demander la permission à l'utilisateur de rendre la tablette visible. Si l'utilisateur répond oui, alors le périphérique devient visible pendant la durée spécifiée. On peut le voir sur la capture d'écran de l'IHM ci-dessous.



Recherche de nouveaux périphériques

Pour détecter de nouveaux périphériques, Android fournit une classe `BroadcastReceiver`. Les objets `BroadcastReceiver` servent à recevoir les informations concernant chaque périphérique *Bluetooth* trouvé.

C'est `ACTION_FOUND` qui représente un nouveau périphérique découvert. (1)

```
private final BroadcastReceiver mReceiver = new BroadcastReceiver()
{
```

```
    public void onReceive(Context context, Intent intent)
    {
        ArrayList mArrayList = new ArrayList();
        String action = intent.getAction();
        // Périphérique bluetooth trouvé ?
        if (BluetoothDevice.ACTION_FOUND.equals(action)) //(1)
        {
            BluetoothDevice device =
intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            mArrayList.add(device.getName() + "\n" + device.getAddress());
        }
    }
};
```

```
private Set<BluetoothDevice> pairedDevices =
mBluetoothAdapter.getBondedDevices();


protected void activerBluetooth()
{
    // ...
    // Recherche des périphériques
    IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
    registerReceiver(mReceiver, filter);
    mBluetoothAdapter.startDiscovery();
    BluetoothDevice myDevice = null;
    peripheriqueBluetooth = null;
// …
}
```

On deconnecte les periphériques bluetooth lors de la fin de l'application car la recherche de périphérique est une opération qui demade beaucoup de ressources.

```
protected void onDestroy() {
    /* on déconnecte le bluetooth ce qui arrêtere la réception des trames */
    Log.d("onDestroy()", "deconnecter()"); // d = debug
    peripheriqueBluetooth.deconnecter();
    if (mBluetoothAdapter != null) {
        mBluetoothAdapter.cancelDiscovery();
        unregisterReceiver(mReceiver);
    }
    super.onDestroy();
}
```

Trouver les périphériques déjà appairés

Si on connaît déjà l'appareil auquel on veut se connecter, on cherche parmi ceux qu'on

connaît pour savoir si celui qu'on cherche s'y trouve déjà.

Pour cela on appelle la méthode `getBondedDevices(),` qui retourne une liste d'objets `BluetoothDevice` qui représente les périphériques appairés. (1)

Dans `MainActivity.xml` :

```
private Set<BluetoothDevice> pairedDevices =
mBluetoothAdapter.getBondedDevices();  //(1)
protected void activerBluetooth()
{

    // Pour tous les périphériques bluetooth appairés
    for (BluetoothDevice device : pairedDevices) {        //(3)
        // le périphérique bluetooth de l'aquarium ?
        if (device.getName().equals("HC-05")) { //Simulateur

            myDevice = device;
            peripheriqueBluetooth = new PeripheriqueBluetooth(myDevice,
handler);
            break; // on peut sortir
        }
    }
```

Connecter les périphériques entre eux

Pour assurer une communication et échanger des données, il faut un côté serveur (le périphérique qui attend et accepte la connexion) et un côté client (le périphérique qui fait la demande de la connexion).

La tablette a le rôle du client car c'est elle qui initie la connexion. (1)

Dans `MainActivity.xml` :

```
private PeripheriqueBluetooth peripheriqueBluetooth;


protected void activerBluetooth()
{
// ...
    if (peripheriqueBluetooth != null)
    {
        peripheriqueBluetooth.connecter(); //(1)
    }
}
```

Pour lancer une connexion avec un périphérique distant qui accepte les connexions sur une socket de serveur ouvert, on doit d'abord :

- Obtenir un objet `BluetoothDevice` qui représente le périphérique distant, puis l'utiliser pour acquérir une socket de type BluetoothSocket et lancer la connexion.
  Pour utiliser un UUID[20] correspondant, on code en dur la chaîne UUID dans l'application. (2)

- Lancer la connexion en appelant `connect()` (3).

- Parce que `connect()` est un appel de blocage, il est nécessaire d'effectuer cette procédure de connexion dans un thread distinct du thread d'activité principale (UI).(4)
  Une fois qu'un client appelle cette méthode, le système effectue une recherche SDP[21] pour trouver le périphérique distant avec l'UUID correspondant.

Si la recherche est réussie et que l'appareil distant accepte la connexion, il partage le canal RFCOMM à utiliser pendant la connexion.

La classe `PeripheriqueBluetooth` :

```
public class PeripheriqueBluetooth extends Thread
{

    private BluetoothDevice device = null;
    private BluetoothSocket socket = null;

// ...

    public PeripheriqueBluetooth(BluetoothDevice device, Handler handler)
    {
        // ...

            socket =
device.createRfcommSocketToServiceRecord(UUID.fromString("00001101-0000-1000-
8000-00805F9B34FB")); //(2)


    }

    public void connecter()
    {
        /* démarre le thread connexion */
        new Thread() //(4)
        {
            @Override public void run()
            {
                while(!socket.isConnected())
                {
                    try
                    {
```

20 Cf Glossaire
21 Cf Glossaire

```
                        /* Demande de connexion */
                        socket.connect(); //(3)
                        /* connecté ? */
                        if (socket.isConnected())
                        {
                            /* démarre le thread réception */
                            tReception.start();
                        }
                    }
                }
            }.start();
    }
```

Transférer des données entre les périphériques

Gestion d'une connexion :

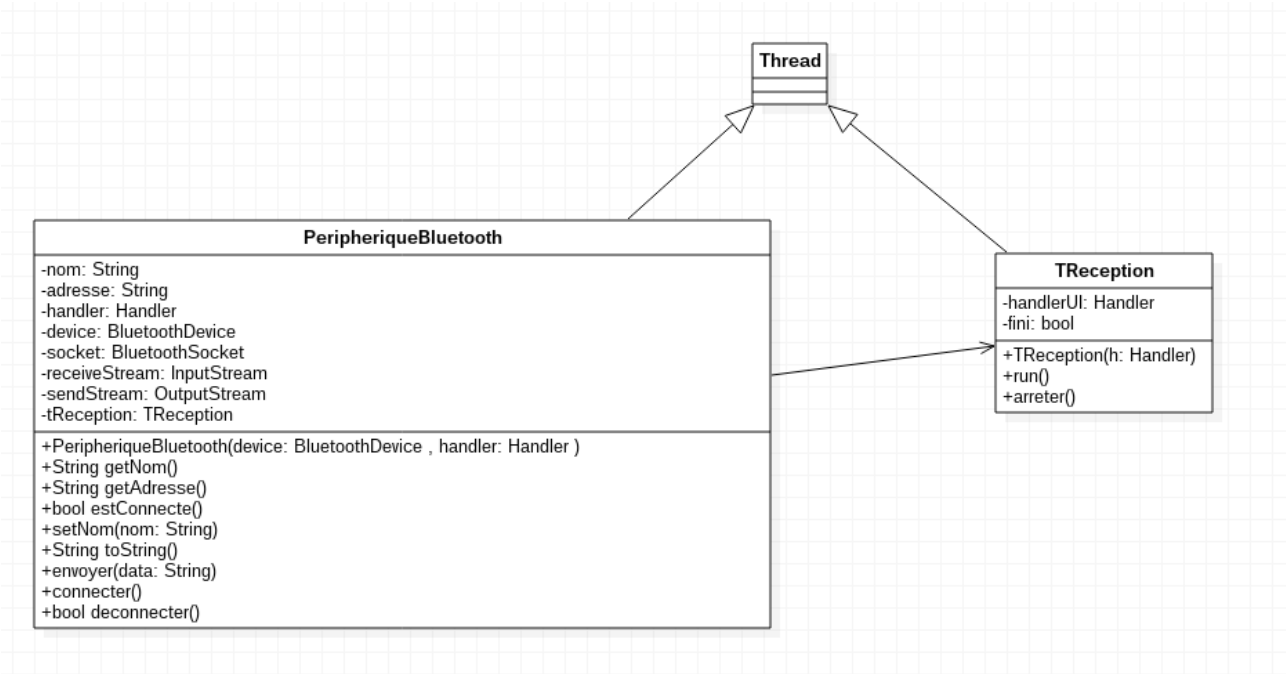
Après avoir connecté plusieurs périphériques avec succès, chacun a une `BluetoothSocket` connectée. On peut alors partager des informations entre les périphériques. À l'aide de `BluetoothSocket`, la procédure générale de transfert de données est la suivante:

- Utiliser `InputStream` et `OutputStream` qui gèrent les transmissions via le socket en utilisant `getInputStream ()` et `getOutputStream (),` respectivement.

- Lire et écrire des données sur les flux en utilisant `read ()` et `write ()`.

Lorsque la lecture retourne avec les données du flux, les données sont envoyées à l'activité principale à l'aide d'un gestionnaire de membres `Handler` de la classe parent. Cette communication se fera dans un *thread car* le *thread UI* (le thread principal) est responsable de l'affichage et des interactions avec l'utilisateur et c'est le seul thread qui doit modifier l'affichage.

L'envoi de données sortantes utilise la méthode `write()` du thread de l'activité principale et lui passe les octets à envoyer sur le périphérique distant.



Envoyer des données à l'aquarium

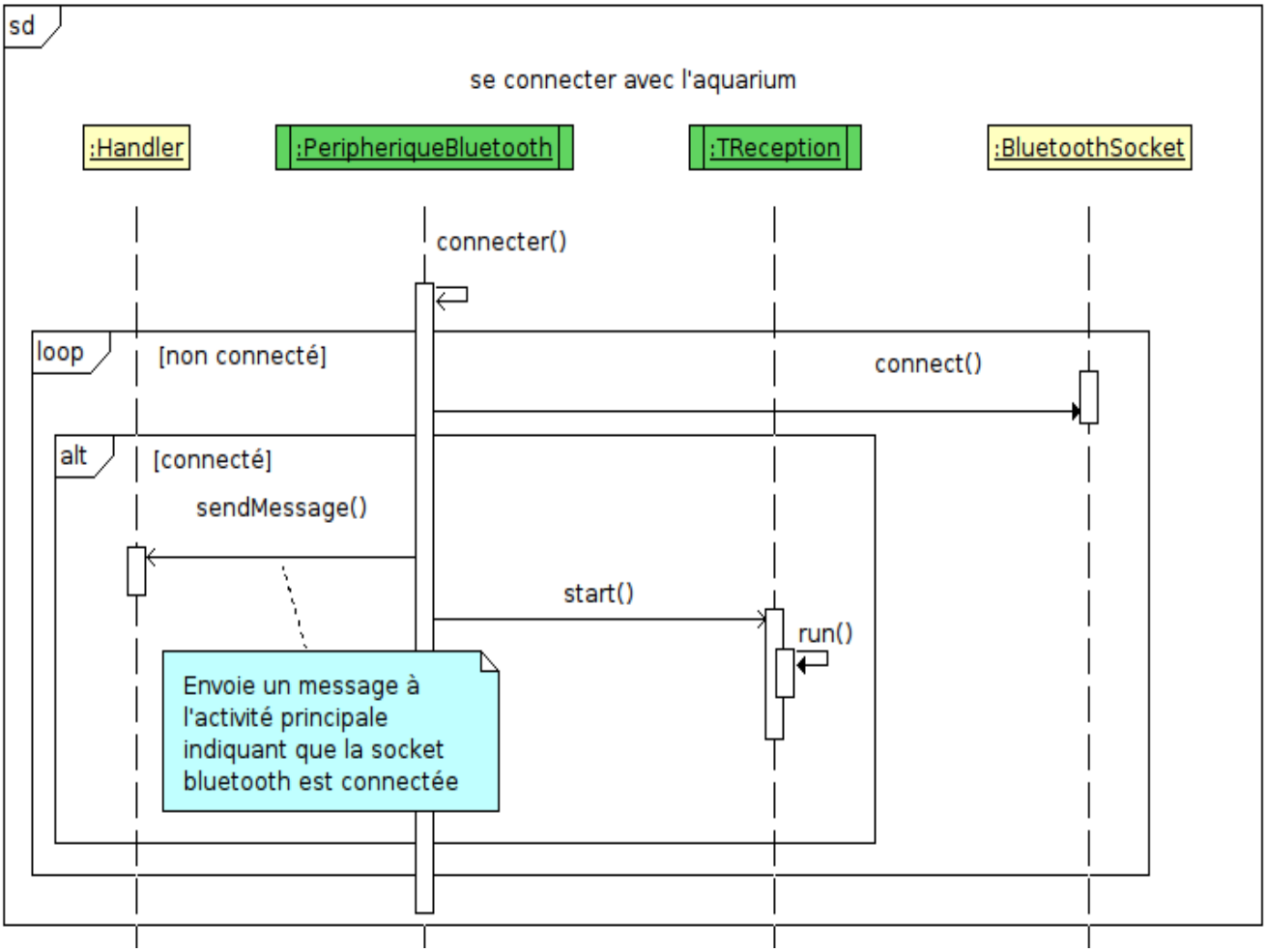
On utilise `OutputStream` pour l'envoi des données, car il s'agit d'un flux sortant. (1)
On écrit des données sur le flux grâce à `write` () (2).

Dans la classe `PeripheriqueBluetooth` :

```
// ...

    private OutputStream sendStream = null; //(1)
    private TReception tReception;

    public void envoyer(String data)
    {
        if(socket == null)
            return;
        try
        {
            sendStream.write(data.getBytes()); //(2)
            sendStream.flush();
        }

// ...
    }
```

Recevoir des données de l'aquarium

Lors de la connexion du périphérique *Bluetooth*, on démarre le *thread* de réception (cf Connecter les périphériques entre eux), comme on peut le voir sur le diagramme de séquence suivant :



Le *thread* de réception fait appel à la classe `TReception` car il s'agit du *thread* de réception des trames en provenance de l'aquarium via le *Bluetooth*. (1)

Dans la classe `PeripheriqueBluetooth` :

```
    public void connecter()
    {
        new Thread()
        {
            @Override public void run()
            {
                try
                {
                    socket.connect();
                    Message msg = Message.obtain();
                    msg.arg1 = CODE_CONNEXION;
                    handler.sendMessage(msg);
                    tReception.start(); //(1)
                }
            }
        }.start();
    }
```

La classe `TReception` :

On réceptionne les trames en provenance de l'aquarium via le *Bluetooth* :(2)

```
        @Override public void run()
        {
            attendreConnexion();
            viderBuffer();
            /* boucle de réception des trames */
            while(!fini)
            {
                recevoirTrame();  //(2)
            }
        }
```

On utilise `InputStream` pour la réception des données, car il s'agit d'un flux entrant. (3)
On lit les données du buffer grâce à `read ()` (4).
Puis on transmet les données à l'activité principale. (5)

```
        private void recevoirTrame()
        {
           // ...
                if(receiveStream.available() > 0) //(3)
                {
                    // ...
                    if(!fini && socket.isConnected())
                    {
                        int k = receiveStream.read(buffer, 0,
PeripheriqueBluetooth.TAILLE_BUFFER); //(4)


            /* transmet la trame à l'activité principale */
                        Message msg = Message.obtain();
```

```
                        msg.what = PeripheriqueBluetooth.CODE_RECEPTION;
                        msg.obj = trame;
                        handlerUI.sendMessage(msg); //(5)

                            }
                        }
                    }
                }
            }
```

Un Handler pour gérer les communications avec le thread réseau :

Un `Handler` est un objet qui permet l'envoi de message. Il gère les communications avec le *thread* de réception des trames. Lors de sa création, il est associé à un *thread*, et délivre les messages selon leur place dans la file d'attente.

Dans le fichier `MainActivity.xml` :

```
final private Handler handler = new Handler() {
    public void handleMessage(Message msg) {
        super.handleMessage(msg);

// ...
        else if (msg.what == PeripheriqueBluetooth.CODE_RECEPTION) {
            // ...
                message = (String) msg.obj;
       /* extrait les données (eau, seuils, alarmes, ...) de la trame reçue */
                boolean donneesExtraites = extraireDonnees(message);
                if(donneesExtraites)
                {
                    /* provoque l'affichage des données extraites */
                    afficherDonnees();

                }
                /* extrait les états des appareils de la trame reçue */
                boolean etatsExtraits = extraireEtats(message);
                if(etatsExtraits)
                {
                    /* provoque l'affichage des états extraits */
                    afficherEtats();

                }
            }
        }
    }
};
```

Exemple en envoyant une trame à un appareil

Une fois la trame fabriquée, on l'envoie depuis **`MainActivity.xml`** (1)

```
private void commanderAppareil(String nom, int etat) {

// ...
    peripheriqueBluetooth.envoyer(trame); // (1)
// ...
}
```

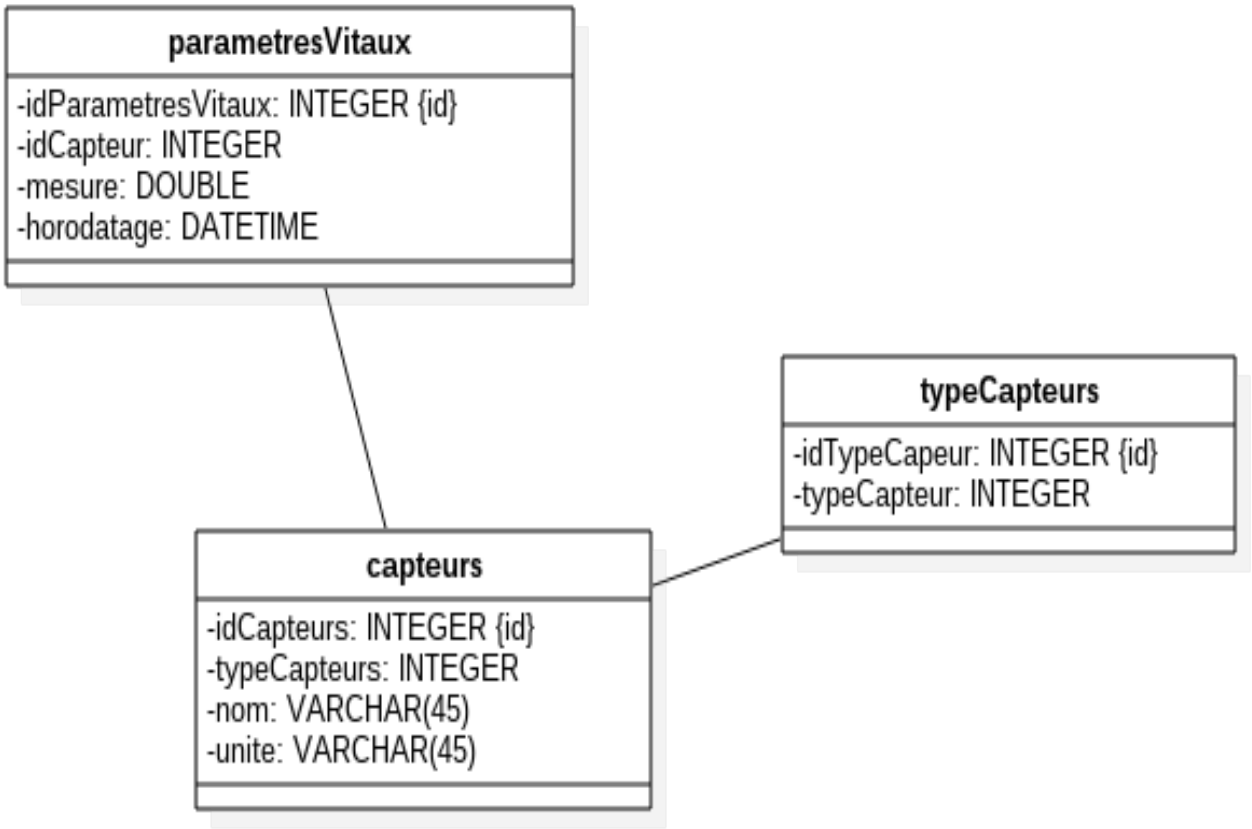
Puis la trame est envoyée à l'aquarium en uilisant `write()` (2)

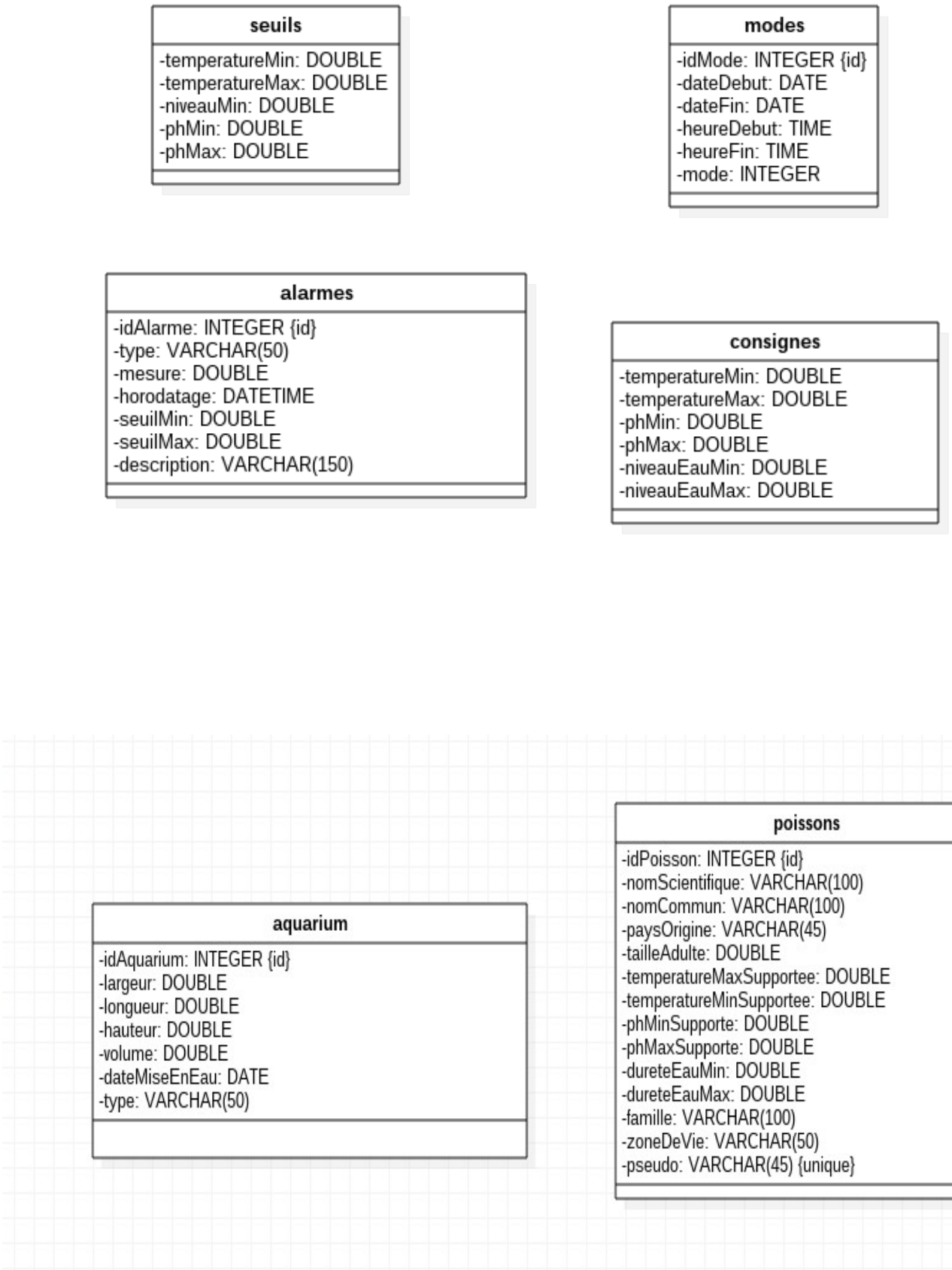
Dans `PeripheriqueBluetooth` :

```
    public void envoyer(String data)
    {
        if(socket == null)
            return;
        try
        {
            sendStream.write(data.getBytes()); //(2)
            sendStream.flush();
        }

// ...
    }
```

Annexe 2 : Configuration de la base de données

Les tables




seuils
-temperatureMin: DOUBLE
-temperatureMax: DOUBLE
-niveauMin: DOUBLE
-phMin: DOUBLE
-phMax: DOUBLE
modes
-idMode: INTEGER {id}
-dateDebut: DATE
-dateFin: DATE
-heureDebut: TIME
-heureFin: TIME
-mode: INTEGER
alarmes
-idAlarme: INTEGER {id}
-type: VARCHAR(50)
-mesure: DOUBLE
-horodatage: DATETIME
-seuilMin: DOUBLE
-seuilMax: DOUBLE
-description: VARCHAR(150)
consignes
-temperatureMin: DOUBLE
-temperatureMax: DOUBLE
-phMin: DOUBLE
-phMax: DOUBLE
-niveauEauMin: DOUBLE
-niveauEauMax: DOUBLE
aquarium
-idAquarium: INTEGER {id}
-largeur: DOUBLE
-longueur: DOUBLE
-hauteur: DOUBLE
-volume: DOUBLE
-dateMiseEnEau: DATE
-type: VARCHAR(50)
poissons
-idPoisson: INTEGER {id}
-nomScientifique: VARCHAR(100)
-nomCommun: VARCHAR(100)
-paysOrigine: VARCHAR(45)
-tailleAdulte: DOUBLE
-temperatureMaxSupportee: DOUBLE
-temperatureMinSupportee: DOUBLE
-phMinSupporte: DOUBLE
-phMaxSupporte: DOUBLE
-dureteEauMin: DOUBLE
-dureteEauMax: DOUBLE
-famille: VARCHAR(100)
-zoneDeVie: VARCHAR(50)
-pseudo: VARCHAR(45) {unique}

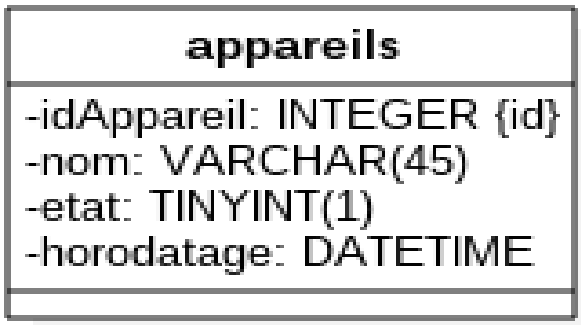
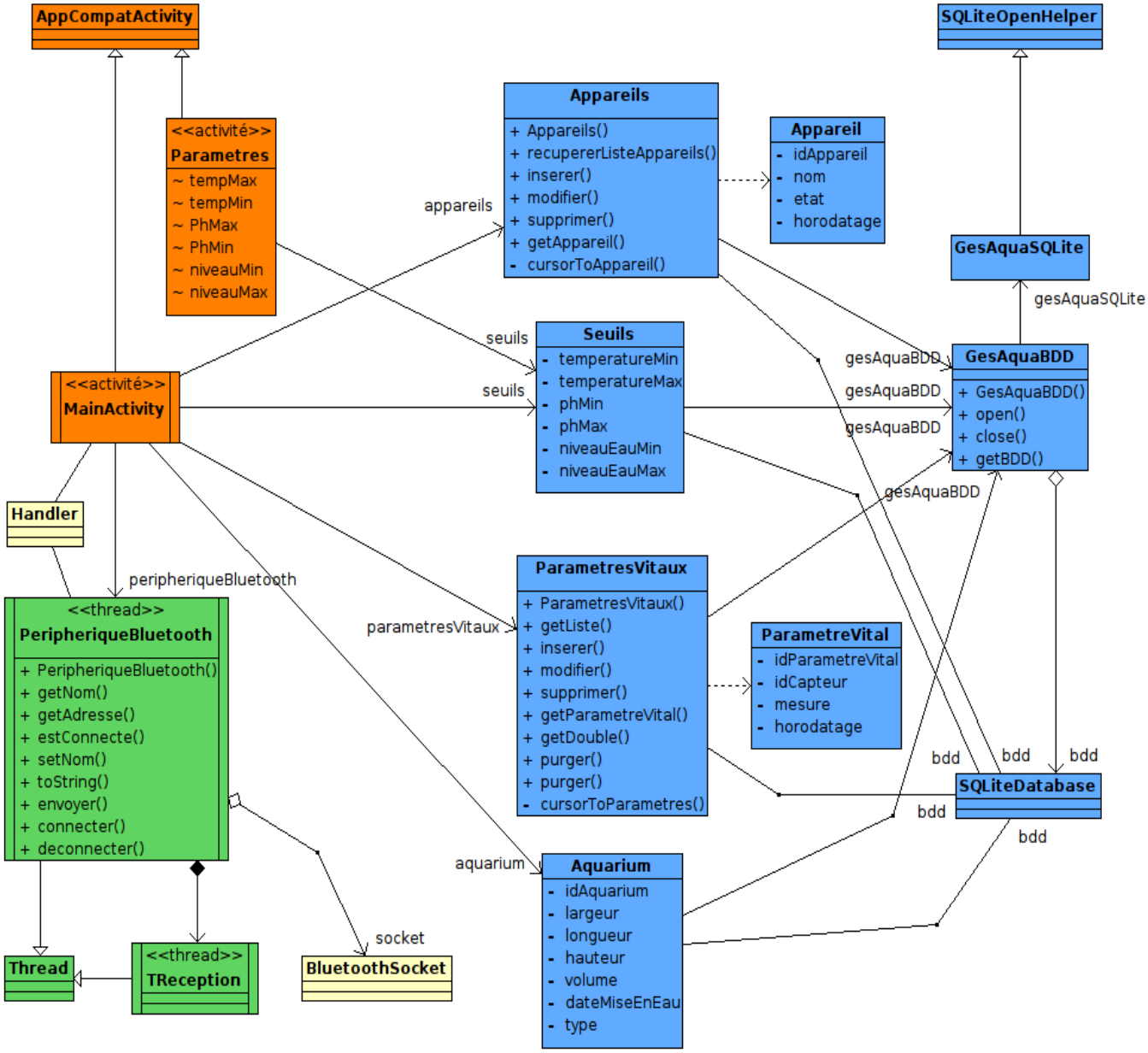

appareils
-idAppareil: INTEGER {id}
-nom: VARCHAR(45)
-etat: TINYINT(1)
-horodatage: DATETIME

Diagramme de classes



Contenu du fichier de configuration de la base de données

```
CREATE DATABASE IF NOT EXISTS gesaqua;
USE gesaqua;
--------------------------------------------------------------------------
-- Structure de la table `appareils`

CREATE TABLE IF NOT EXISTS `appareils` (
  `idAppareil` INTEGER PRIMARY KEY NOT NULL ,
  `nom` VARCHAR(45) NULL ,
  `etat`TINYINT(1) NULL ,
  `horodatage` DATETIME ); -- DATETIME : format `YYYY-MM-DD HH:MM:SS`

-- Contenu de la table `appareils`

INSERT INTO `appareils` VALUES(1,'chauffage',0,'2017-03-24 16:31:10');
INSERT INTO `appareils` VALUES(2,'eclairage',0,'2017-03-24 16:31:20');
INSERT INTO `appareils` VALUES(3,'nourriture',0,'2017-03-24 16:31:20');
INSERT INTO `appareils` VALUES(4,'engrais',0,'2017-03-24 16:31:20');
INSERT INTO `appareils` VALUES(5,'oxygenation',0,'2017-03-24 16:31:20');
INSERT INTO `appareils` VALUES(6,'filtration',0,'2017-03-24 16:31:20');
INSERT INTO `appareils` VALUES(7,'ventilation',0,'2017-03-24 16:31:20');

--------------------------------------------------------------------------
-- Structure de la table `capteurs`

CREATE TABLE IF NOT EXISTS `capteurs` (
  `idCapteurs` INTEGER PRIMARY KEY NOT NULL ,
  `typeCapteurs`INTEGER NULL ,
  `nom`VARCHAR(45) NULL ,
  `unite` VARCHAR(45) NULL ,
   CONSTRAINT fk_capteurs_1 FOREIGN KEY (typeCapteurs ) REFERENCES typeCapteurs
(idtypeCapteurs ) );
```

```
-- Contenu de la table `capteurs`

INSERT INTO `capteurs` VALUES(1,1,'temperature eau','°C');
INSERT INTO `capteurs` VALUES(2,1,'temperature air','°C');
INSERT INTO `capteurs` VALUES(3,2,'ph','');
INSERT INTO `capteurs` VALUES(4,3,'niveau d''eau','cm');
--------------------------------------------------------------------------------
-- Structure de la table `parametresVitaux`

CREATE TABLE IF NOT EXISTS `parametresVitaux` (
  `idParametresVitaux` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL ,
  `idCapteur` INTEGER NULL ,
  `mesure` DOUBLE NULL ,
  `horodatage` DATETIME, --DATETIME : format `YYYY-MM-DD HH:MM:SS`
 CONSTRAINT fk_parametresVitaux_1 FOREIGN KEY (idCapteur ) REFERENCES capteurs
(idCapteurs ));

----------------------------------------------------------------------
-- Structure de la table `typeCapteurs`

CREATE TABLE IF NOT EXISTS `typeCapteurs` (
  `idtypeCapteurs` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL ,
  `typeCapteurs` VARCHAR(45) NULL );

-- Contenu de la table `typeCapteurs`

INSERT INTO `typeCapteurs` VALUES(1,'temperature');
INSERT INTO `typeCapteurs` VALUES(2,'ph');
INSERT INTO `typeCapteurs` VALUES(3,'niveau');
-----------------------------------------------------------
-- Structure de la table `consignes`
```

```
CREATE TABLE IF NOT EXISTS `consignes` (
  `temperatureMin` DOUBLE NULL ,
  `temperatureMax` DOUBLE NULL ,
  `phMin` DOUBLE NULL ,
  `phMax` DOUBLE NULL ,
  `niveauEauMin` DOUBLE NULL );

-- Contenu de la table `consignes`

INSERT INTO `consignes` VALUES(24,26,6.8,7.2,34);
--------------------------------------------------------------
-- Structure de la table `seuils`

CREATE TABLE IF NOT EXISTS `seuils` (
   `temperatureMin` DOUBLE NULL ,
   `temperatureMax` DOUBLE NULL ,
   `phMin` DOUBLE NULL ,
   `phMax` DOUBLE NULL ,
   `niveauEauMin` DOUBLE NULL ,
   `niveauEauMax` DOUBLE NULL );

-- Contenu de la table `seuils`

INSERT INTO `seuils` VALUES(23,27,5.5,9,20,40);
---------------------------------------------------------------
-- Structure de la table `aquariums`

CREATE TABLE IF NOT EXISTS `aquariums` (
   `idAquarium` INTEGER PRIMARY KEY NOT NULL ,
   `largeur` DOUBLE NULL ,
   `longueur` DOUBLE NULL ,
   `hauteur` DOUBLE NULL ,
```

```
   `volume` DOUBLE NULL ,
   `dateMiseEnEau` DATETIME,  ---DATETIME : format `YYYY-MM-DD HH:MM:SS`
   `type` VARCHAR(50) NOT NULL);

-- Contenu de la table `aquariums`

INSERT INTO `aquariums` VALUES(1,40,100,50,200,'2017-04-20 14:20:36','amazonien');
--------------------------------------------------------------------------
-- Structure de la table `alarmes`

CREATE TABLE IF NOT EXISTS `alarmes` (
   `idAlarme` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL ,
   `type` VARCHAR(50) NOT NULL ,
   `mesure` DOUBLE NULL ,
   `horodatage` DATETIME ,   ---DATETIME : format `YYYY-MM-DD HH:MM:SS`
   `seuilMin` DOUBLE NOT NULL ,
   `seuilMax` DOUBLE NOT NULL ,
   `description` VARCHAR(150) );

-- Contenu de la table `alarmes`

INSERT INTO `alarmes` VALUES(1,'pH',6.2,'2017-04-21 14:31:36',6.5,8,'pH trop bas');
INSERT INTO `alarmes` VALUES(2,'niveau',19.8,'2017-04-21 14:31:36',20,40,'niveau : eau trop
basse');
INSERT INTO `alarmes` VALUES(3,'temperature',19.3,'2017-04-21 14:31:36',23,27,'température
trop basse');
-----------------------------------------------------------------------------
-- Structure de la table `modes`

CREATE TABLE IF NOT EXISTS `modes` (
 `idMode` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL ,
 `dateDebut` DATE ,
 `dateFin` DATE ,
```

```
 `heureDebut` TIME ,
 `heureFin` TIME ,
 `mode` INTEGER NOT NULL );  --- 0 pour mode manuel et 1 pour mode automatique

-- Contenu de la table `modes`

INSERT INTO `modes` VALUES(1,'2017-04-21','2017-04-21','14:31:36','18:43:36',0);
INSERT INTO `modes` VALUES(2,'2017-04-21','2017-04-24','14:31:36','18:43:36',1);
```

Annexe 3 : Guide de mise en route et d'utilisation

Table des matières

Installation d’un périphérique Android

Passer le smartphone ou la tablette en mode développeur (`Paramètres` -> `A propos du téléphone` -> `Numéro de build`) puis activer le débugage USB (`Paramètres` -> `Options pour les développeurs`).

Brancher le smartphone ou la tablette via le port USB et vérifier qu’il est détecté par le système :

```
$ dmesg
...
[72893.663574] usb 3-9.4.2: New USB device found, idVendor=0502, idProduct=3472
[72893.663581] usb 3-9.4.2: New USB device strings: Mfr=2, Product=3,
SerialNumber=4
[72893.663584] usb 3-9.4.2: Product: MT65xx Android Phone
[72893.663587] usb 3-9.4.2: Manufacturer: MediaTek
[72893.663589] usb 3-9.4.2: SerialNumber: AAEEQCBQAYRCRWDQ
[72893.664509] scsi14 : usb-storage 3-9.4.2:1.0
[72894.660646] scsi 14:0:0:0: Direct-Access     Linux    File-CD Gadget   0000
PQ: 0 ANSI: 2
[72894.660945] scsi 14:0:0:1: Direct-Access     Linux    File-CD Gadget   0000
PQ: 0 ANSI: 2
[72894.662178] sd 14:0:0:0: Attached scsi generic sg10 type 0
[72894.662552] sd 14:0:0:1: Attached scsi generic sg11 type 0
[72894.663606] sd 14:0:0:0: [sdj] Attached SCSI removable disk
[72894.663818] sd 14:0:0:1: [sdk] Attached SCSI removable disk
```

```
$ lsusb
...
Bus 003 Device 016: ID 0502:3472 Acer, Inc.
```

Sous Linux, il est nécessaire de créer un fichier de règles *udev* qui contient une configuration USB

pour chaque type de périphérique réel. En tant que *root*, créer le fchier /etc/udev/rules.d/51-android.rules

et y inscrire la ligne suivante (en précisant en hexadécimal votre *idVendor*) :

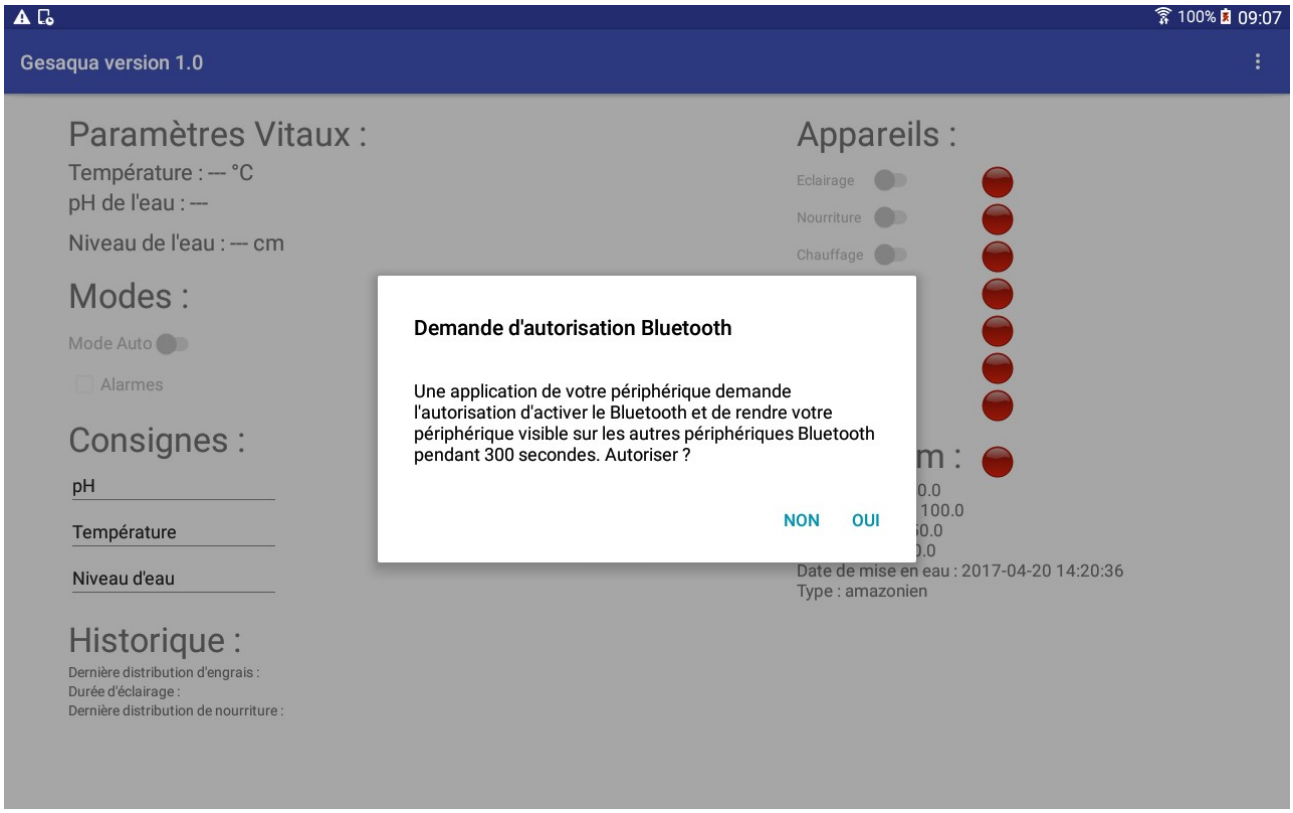
```
$ sudo vim /etc/udev/rules.d/51-android.rules
SUBSYSTEM=="usb", ATTR{idVendor}=="0502", MODE="0666"
```

Brancher le téléphone et vérifer qu'il est reconnu :

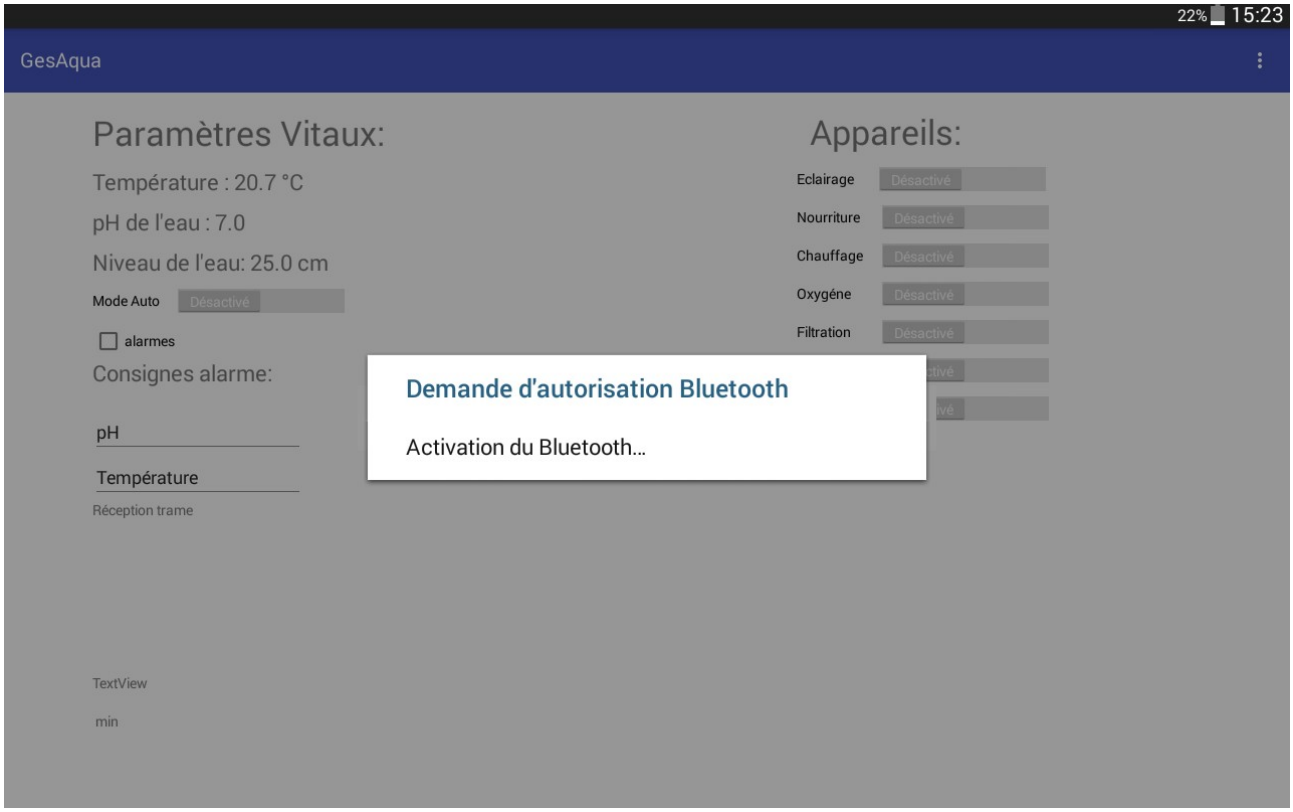
```
$ adb devices
List of devices attached
AAEEQCBQAYRCRWDQ    device
```

Mise en route de l'application

Lors de la mise en route de l'application, et si le Bluettoth de la tablette n'a pas été activé, une fenêtre s'affiche pour demander l'autorisation d'activer le *Bluetooth*. Il est indispensable de répondre `OUI` car sans connexion *Bluetooth*, la communication entre la tablette et l'aquarium est impossible.

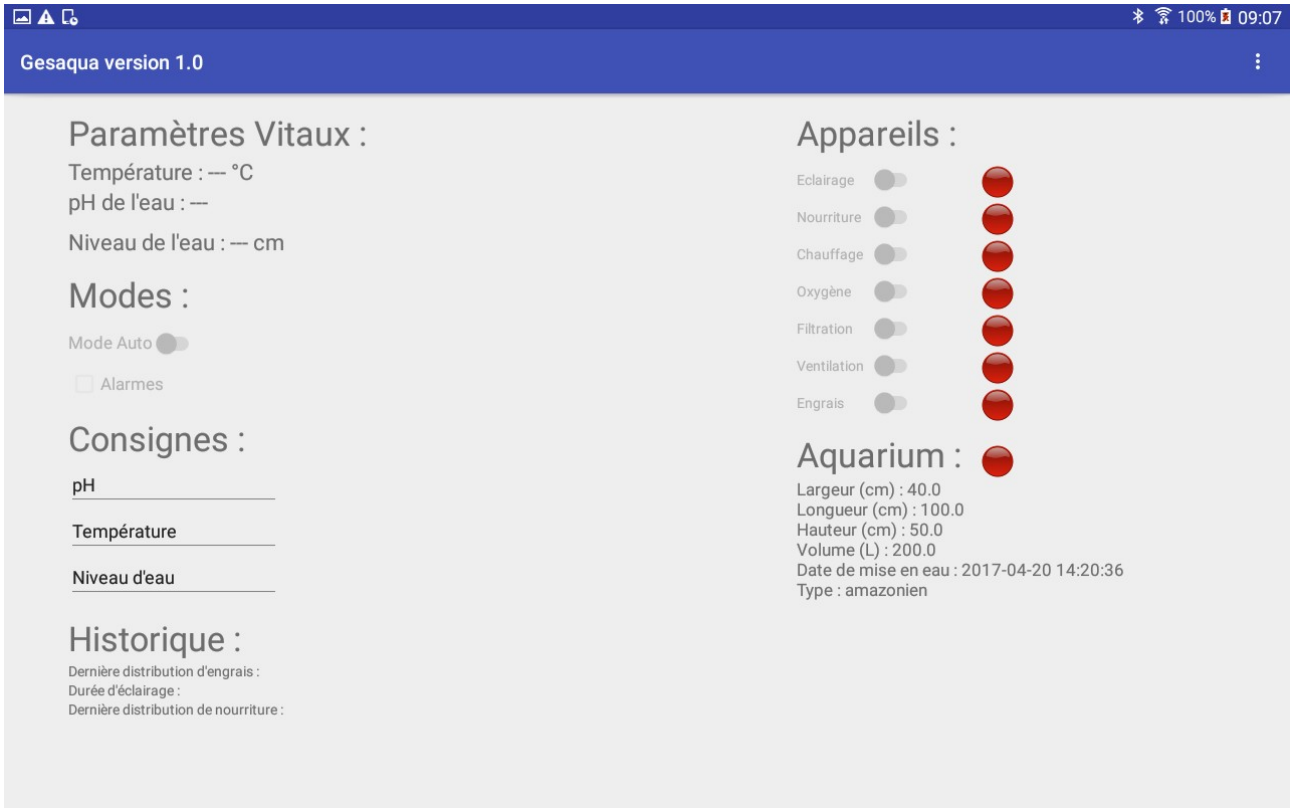


Une fois la demande d'activation du *Bluetooth* effectuée, une fenêtre s'affiche et vous prévient que le *Bluetooth* de la tablette est en cours d'activation.



Le *Bluetooth* est alors activé. On le voit grâce à l'icône en haut à droite de l'écran de la tablette.

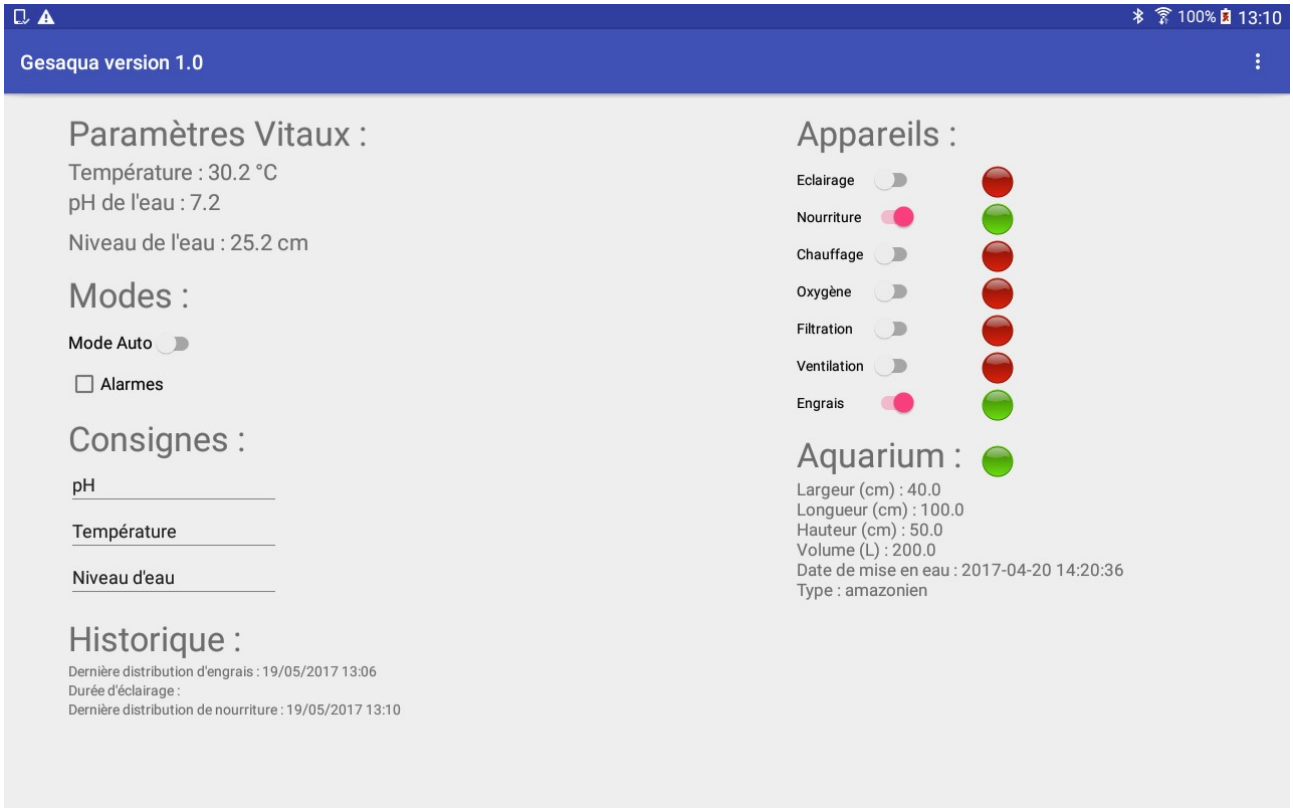
On se trouve alors sur la page d'accueil de l'application.



Page d'accueil de l'application

Depuis cette page d'accueil de l'application, il vous est possible de :

- visualiser les `Paramètres Vitaux` de votre aquarium : la température de l'eau (en °C), le pH de l'eau ainsi que le niveau d'eau présent dans l'aquarium (en cm). Ceux-ci changent périodiquement au fur et à mesure de la réception des données.
- consulter l'`Historique` : visualiser la dernière distribution d'engrais et de nourriture
- consulter les états des `Appareils` et les activer ou les désactiver grâce aux interrupteurs
- visualiser les caractéristiques de votre `Aquarium`
- entrer manuellement des `Consignes` de pH, température et niveau d'eau que le système devra maintenir
- vous pouvez choisir, dans `Modes`, un mode automatique. Lorsque cette case est décochée, le mode est manuel par défaut.



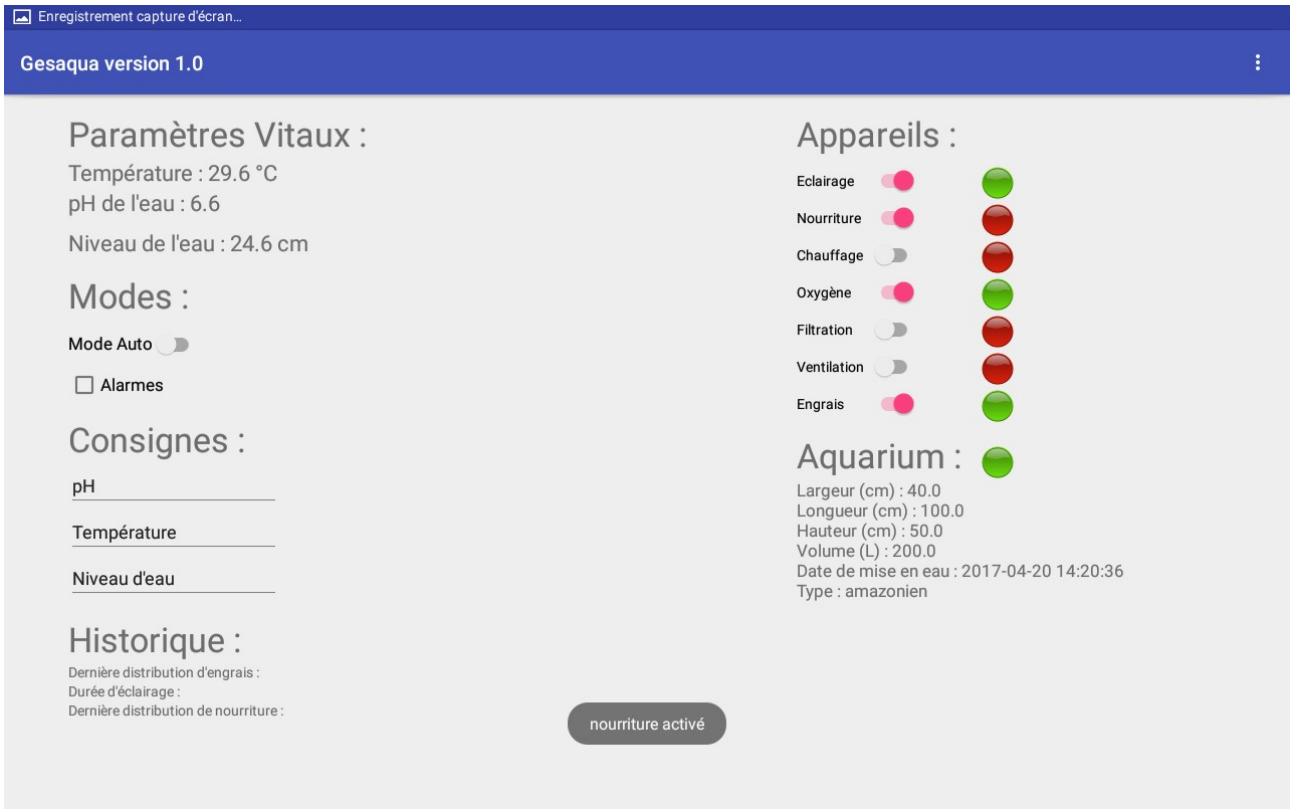
Commander un appareil

Il suffit de cliquer sur l'interrupteur correspondant.

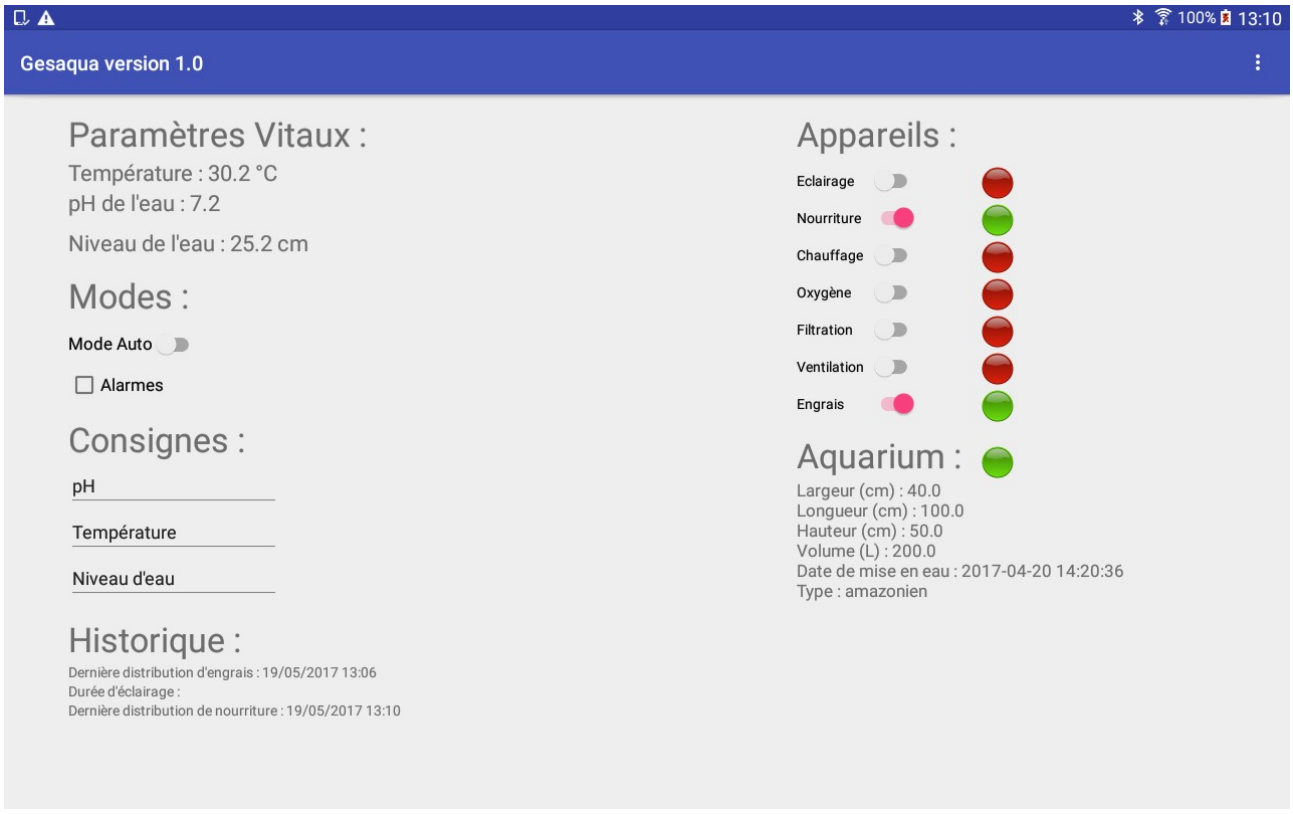
Pour activer un appareil :

Exemple : ici on clique sur l'interrupteur `nourriture.`

Un affichage apparaît en bas de la page pour vous préciser quel interrupteur a été activé.



Puis l'image à côté de l'interrupteur se colore en vert pour confirmer que l'appareil a été activé.

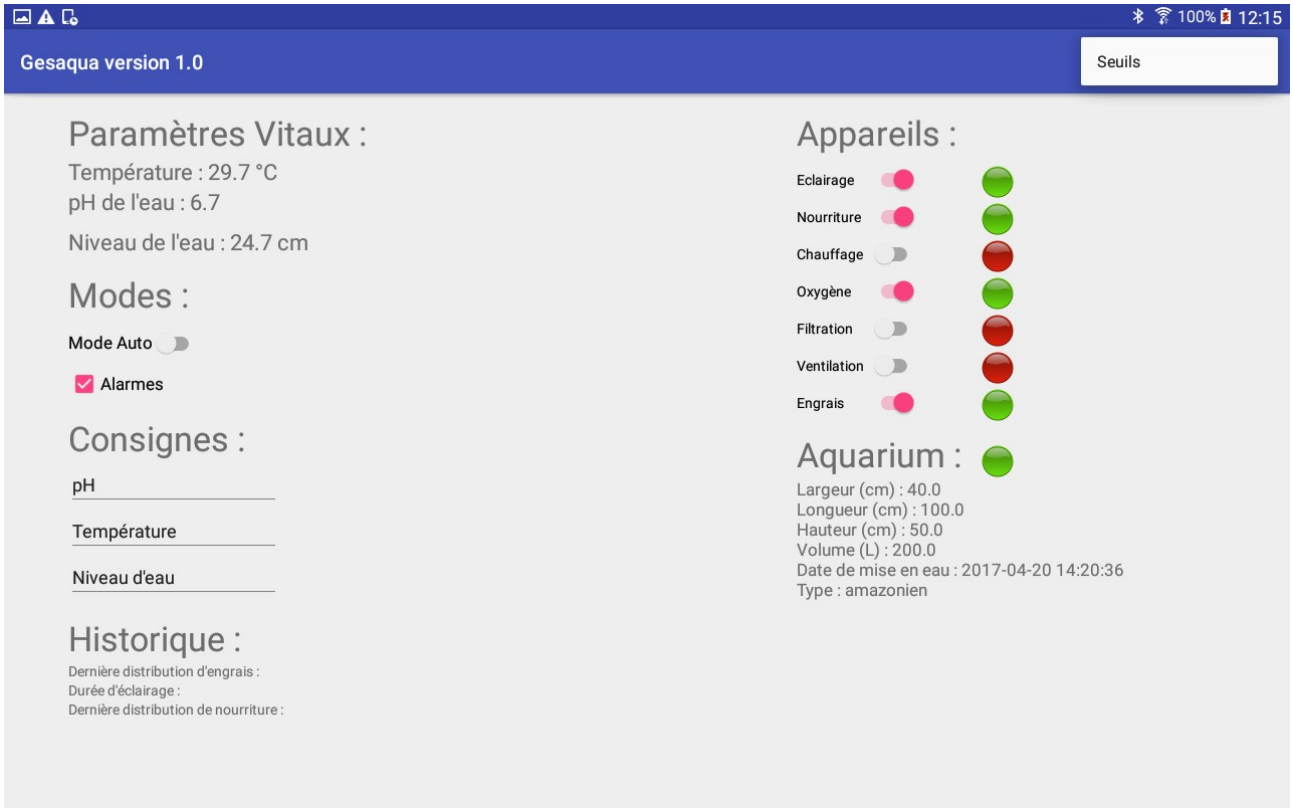


Pour désactiver un appareil

Il suffit de re-cliquer sur l'interrupteur, qui se décolore. L'image à côté de l'interrupteur se recolore en rouge : l'appareil est donc désactivé.

Pour naviguer vers une autre page

Grâce au menu en haut à droite de l'écran, il est possible de naviguer vers une autre page de l'application : la page `Seuils`.



Voici une capture d'écran de la page `Seuils`. Il vous est possible de rentrer manuellement les seuils de température (en °C), de pH et de niveau d'eau (en cm) que vous voulez choisir pour votre aquarium. Ils représentent les seuils que vous jugez acceptables et qui ne devront pas être dépassés.

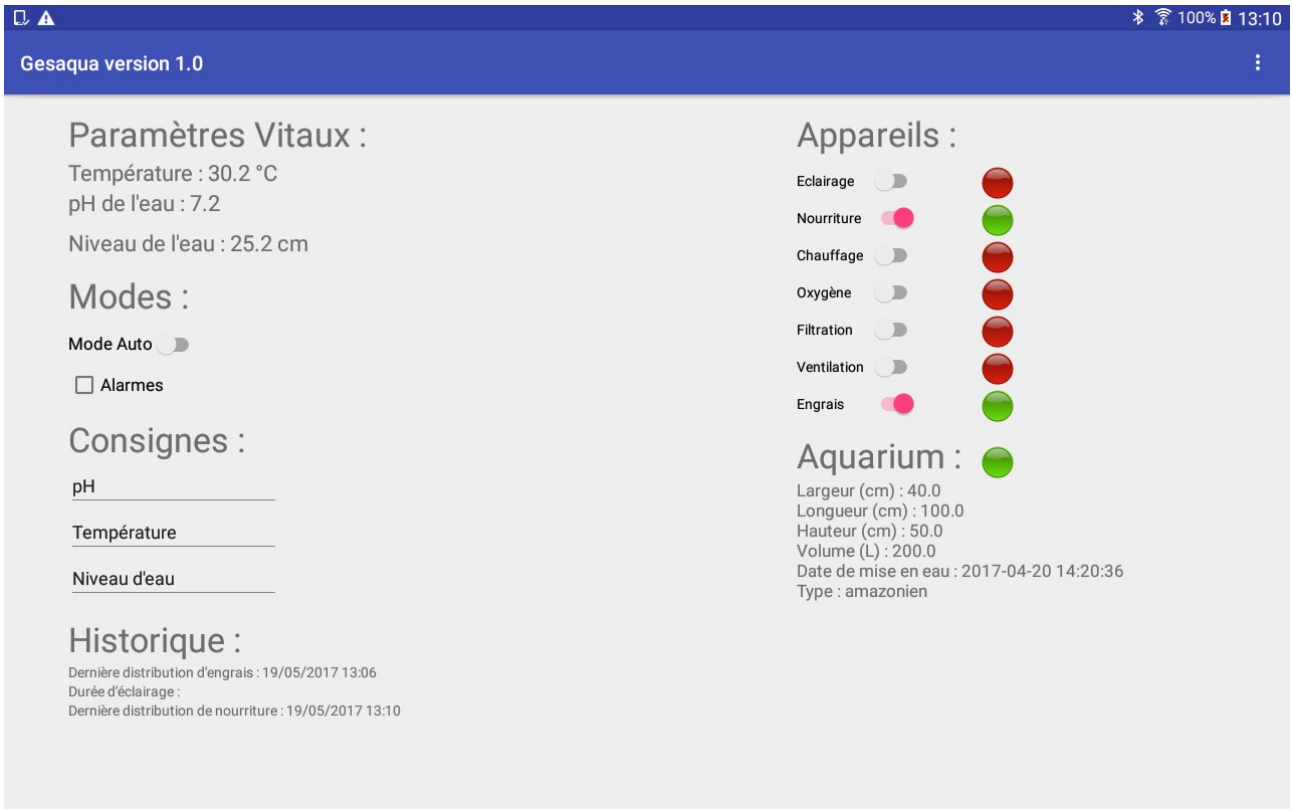
Une fois les seuils entrés manuellement, il suffit de cliquer sur le bouton `VALIDER.`

D'ici il est possible de retourner sur la page d'accueil de l'application grâce au bouton `RETOUR` .


100% 08:18
GesAqua
Seuils de température (°C) :
VALIDER
RETOUR
Min :20.0
Max:40.0
Seuils de PH :
Min :0.0
Max:14.0
Seuils de niveau d'eau (cm) :
Min :0.0
Max:50.0

Pour gérer le mode de fonctionnement

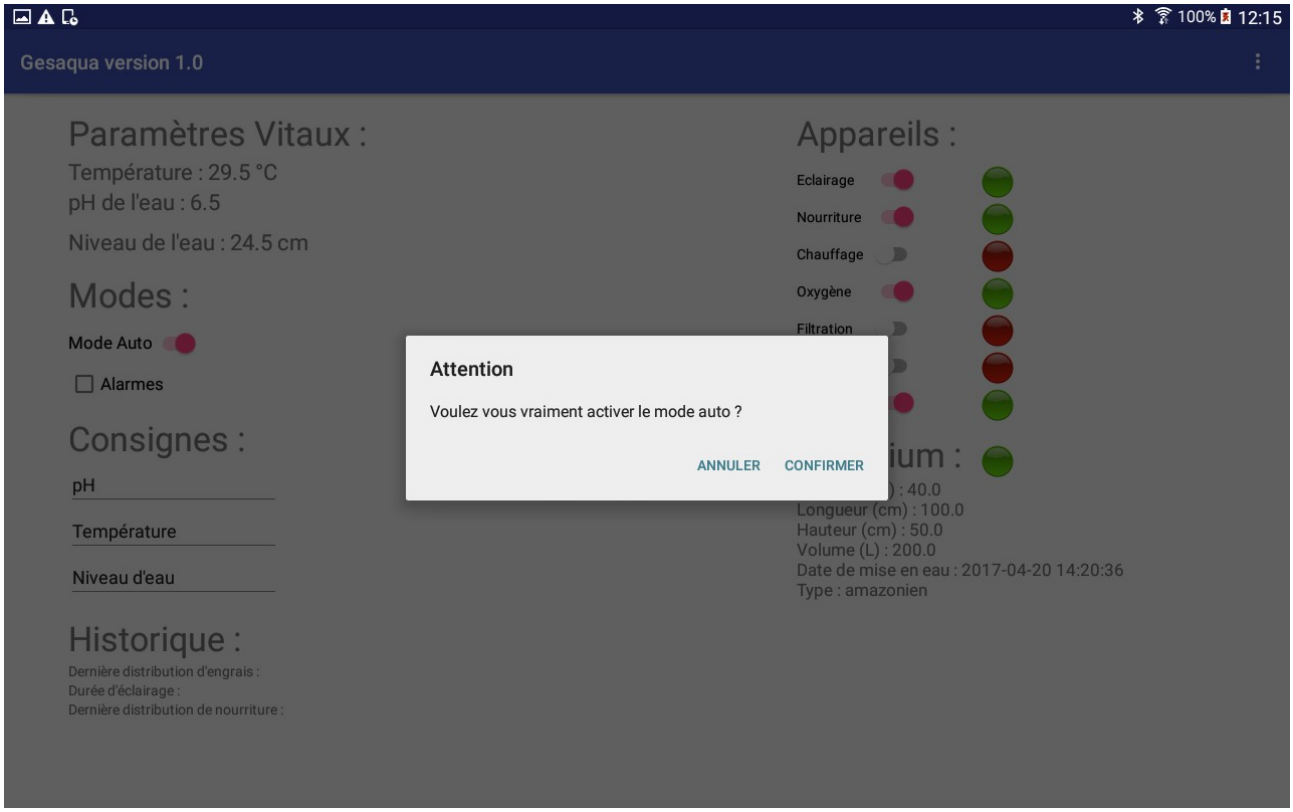
Depuis la page principale, dans la partie `Modes` se trouve un interrupteur. Activé, celui-ci active le mode automatique de la tablette.



Après avoir cliqué sur l'interrupteur du mode automatique, une fenêtre s'ouvre pour vous demander de confirmer que vous voulez bien passer en mode automatique en cliquant sur `CONFIRMER`.

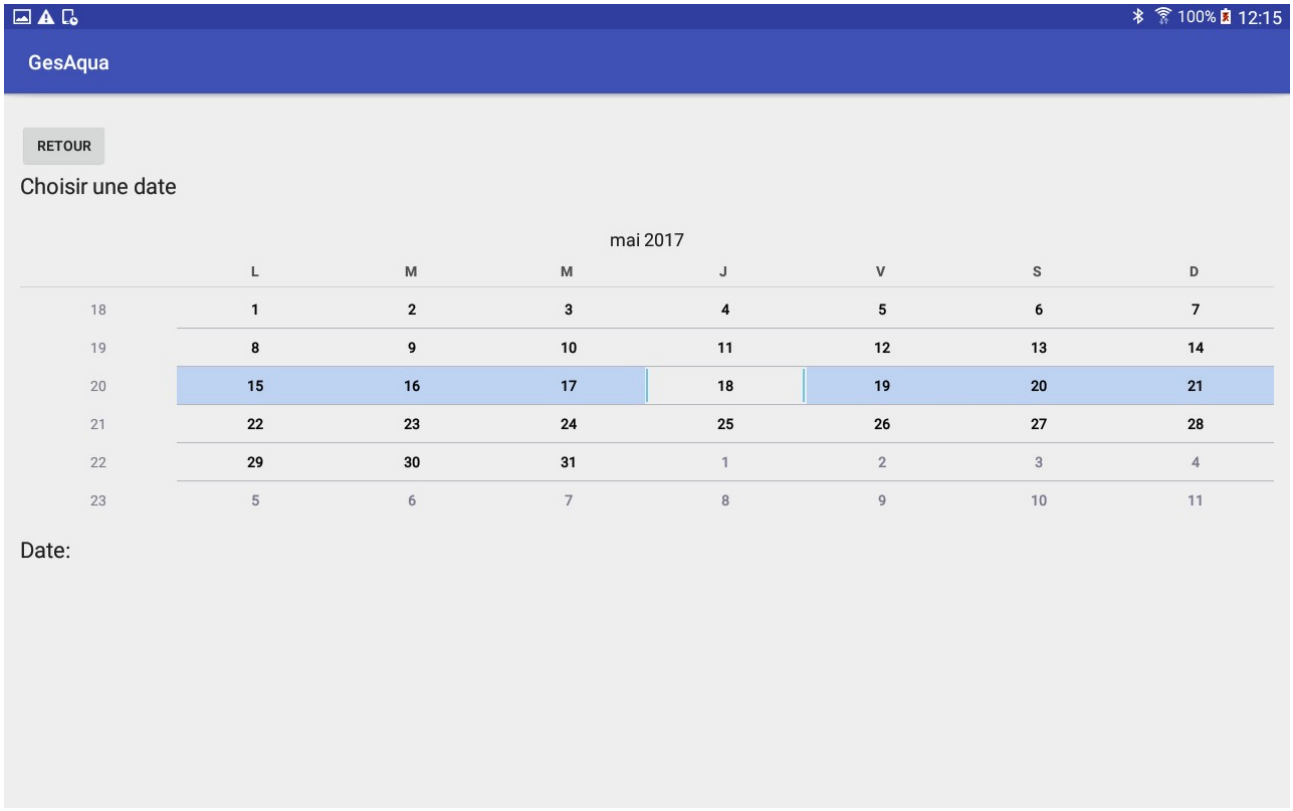
Attention lorsque le mode automatique est activé, le contrôle des appareils ne peut plus se faire en mode manuel.

Pour revenir au mode manuel, il suffit de re-cliquer sur l'interrupteur.



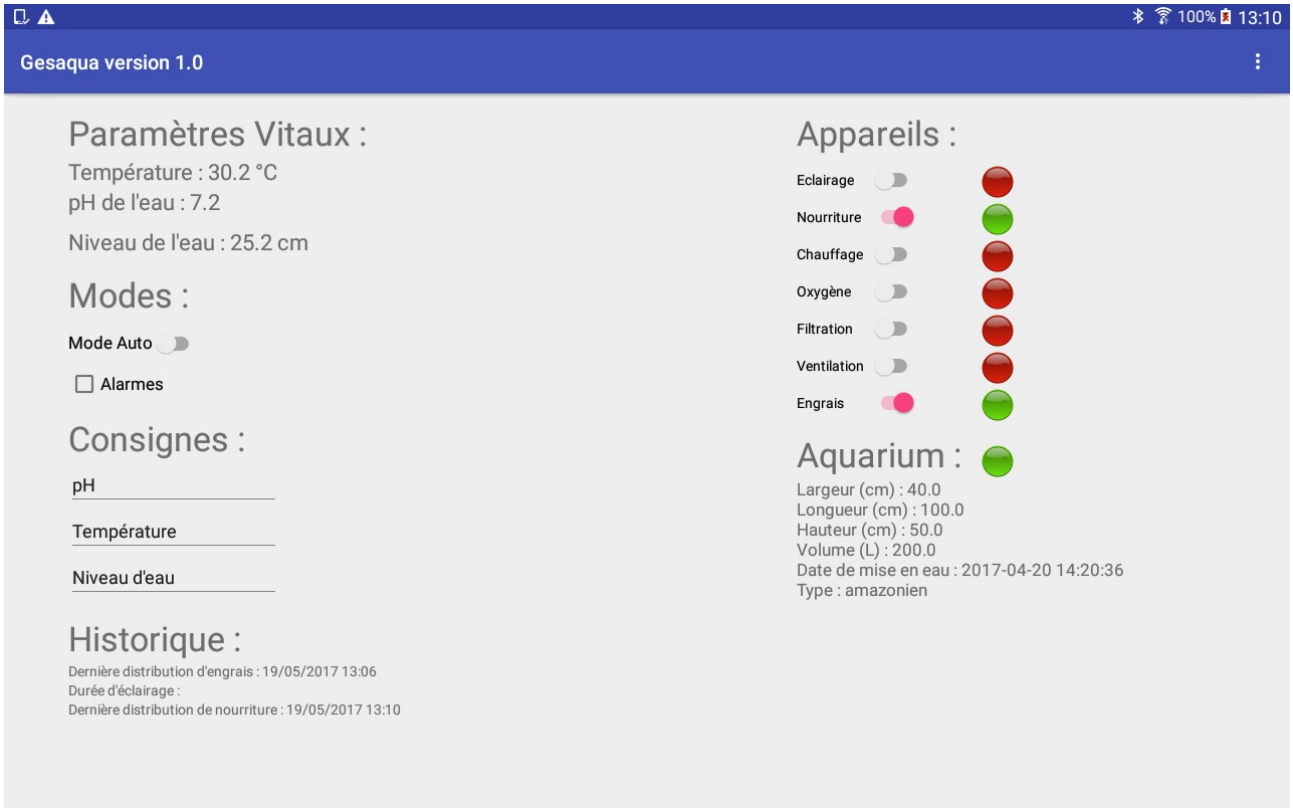
Lorsque vous avez confirmé vouloir passer en mode automatique, une autre page s'ouvre et vous pouvez voir un calendrier. Celui-ci vous permet de choisir le moment où vous souhaitez demarrer le mode automatique.

Vous pouvez retourner sur la page principale en cliquant sur le bouton `RETOUR`.

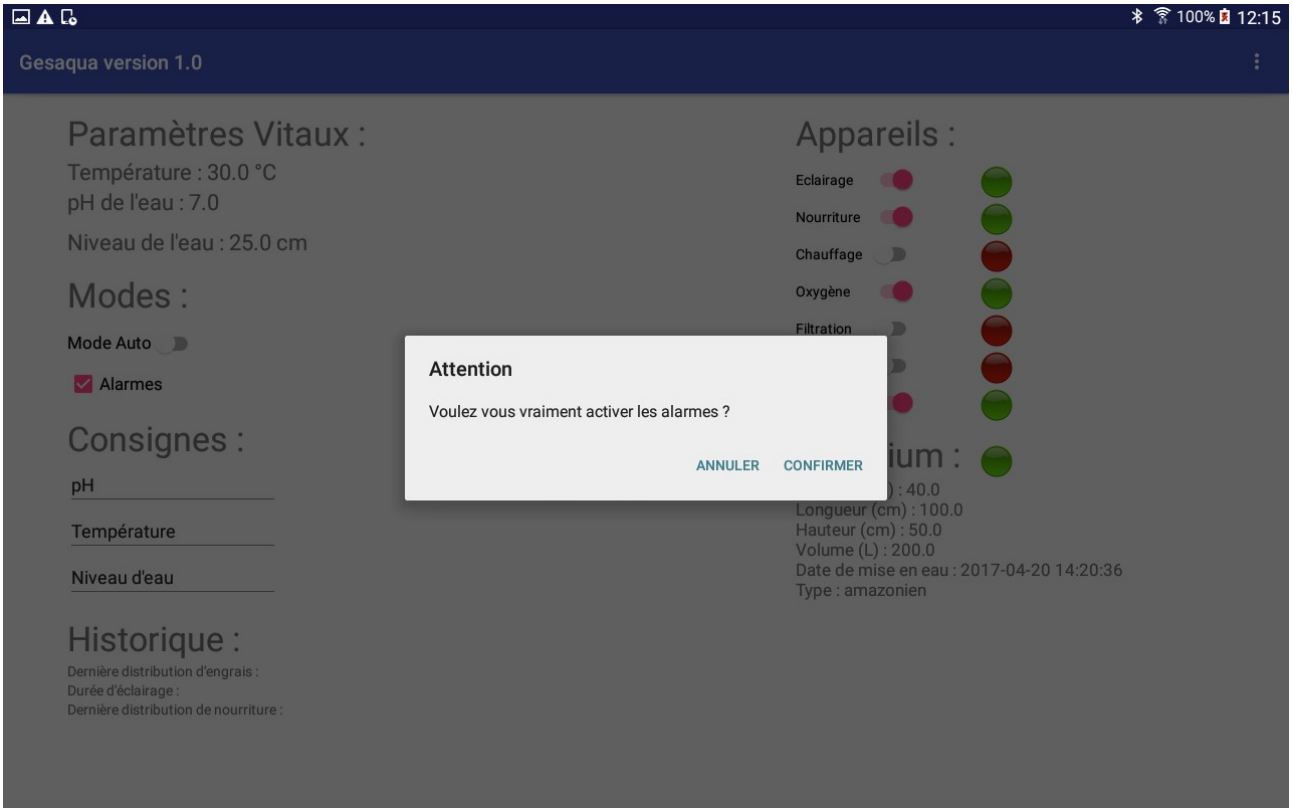

100% 12:15
GesAqua
RETOUR
Choisir une date
mai 2017
L M M J V S D
18 1 2 3 4 5 6 7
19 8 9 10 11 12 13 14
20 15 16 17 18 19 20 21
21 22 23 24 25 26 27 28
22 29 30 31 1 2 3 4
23 5 6 7 8 9 10 11
Date:

Pour activer les alarmes

Sur le page principale de l'application, vous trouverez une case à cocher appelé `Alarmes`. Pour activer les alrmes, il suffit de cocher cette case.



Lorsque cette case est cochée, une fenêtre apparaît pour vous demander de confirmer que vous voulez bien activer les alarmes. Si c'est le cas, cliquez sur `CONFIRMER`.



Pour désactiver les alarmes, il fat décocher la case `Alarmes`.

Glossaire :

Application Android :

- IDE : environnement de développement intégré
- SQLite : moteur de base de données relationnelle accessible par le langage SQL et intégré dans chaque appareil Android.
- Activité : Une activité est la composante principale d'une application sous Android. L'activity est le métier de l'application et possède généralement une View au minimum, c'est-à-dire un écran graphique. Ainsi dans une application standard, on pourrait trouver une activité qui liste des contacts, une activité qui ajoute un nouveau contact, et une activité qui affiche le détail d'un contact. Le tout forme un ensemble cohérent, mais chaque activité pourrait fonctionner de manière autonome.
- Intent : Les Intents permettent de communiquer entre les différentes activités de notre application, mais aussi du téléphone. Ils sont en quelque sorte le « messager » pour lancer une activité. Ainsi une activité peut en lancer une autre soit en passant un intent vide, soit en y passant des paramètres. Les Intent Filters jouent le rôle de filtre. Ils permettent de contrôler d'où provient l'Intent (ou d'autres paramètres) afin de lancer ou non l'activité.
- Le fichier AndroidManifest.xml : Le fichier manifest permet de décrire votre application. On y retrouve :

  → le nom du package de l'application. Il servira d'identifiant unique ;
  → tous les composants de l'application (Activities, Services, BroadCast Receivers, Content providers).
  → on détermine dans quels processus les composants de l'application seront contenus ;
  → les permissions nécessaires pour le bon fonctionnement de l'application ;
  → les informations contenant les versions de l'Android API requis pour exécuter votre application ;
  → les bibliothèques utilisées par votre application.

Bluetooth :

- Appairage : être appairé signifie que 2 périphériques « connaissent » leur existence respective, partagent une clé pour leur authentification et ont la capacité d'établir une connexion entre eux.

- Connexion : être connecté signifie que les 2 périphériques partagent un canal RFCOMM et sont capables de se transmettre des données. Il est nécessaire d'être appairé à un périphérique avant de pouvoir établir une communication via un canal RFCOMM. L'appairage est fait de façon automatique lorsque la connexion est initialisée via une API *Bluetooth* Android.
- RFCOMM : ***Radio Frequency Communication***.. Il s'agit d'un protocole de transport qui permet d'émuler un port série de type RS-232. Il peut supporter jusqu'à 60 connexions simultanées (canaux RFCOMM) entre 2 périphériques *Bluetooth*.

- Broadcast Receiver : Un Broadcast Receiver permet d'écouter ce qui se passe sur le système et éventuellement de déclencher une action si besoin. C'est souvent par ce mécanisme que les services sont lancés.

- UUID : ***Universally Unique Identifier*** ou identifiant universel unique. Système permettant à des systèmes distribués d'identifier de façon unique une information sans coordination. Les UUID sont destinés à l'identification de composants logiciels (*plugins*), des différents membres dans un système distribué ou d'autres applications nécessitant une identification sans ambiguïté. centrale importante.

- Recherche SDP : ***Service Discovery Protocol*** .Ce protocole est la base de la recherche de service sur tous les équipements bluetooth. A l'aide des informations SDP, les services et les caractéristiques de ces services peuvent être demandées et ceci après qu'une connexion entre plusieurs éléments bluetooth aient été établie.

La connectique :

- 1-Wire : La communication sur le bus 1-wire est caractérisée par un ensemble de pulse

  « changement d’état du bus ». Ce bus est généralement utilisé en domotique pour des thermomètres ou autres instruments de mesure météorologiques. «chan
- I2C : bus série synchrone bidirectionnel half-duplex, où plusieurs équipements, maîtres où esclaves, peuvent être connectés au bus.