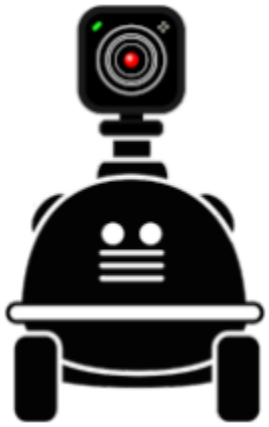
Projet Rov'Net

Dossier Technique



Boffredo Nicolas & Reynier Jacques

BTS SN IR

v 1.0 – 5 juin 2019

Table des matières

Projet Rov'Net

Expression du besoin

Il s'agit de réaliser un programme permettant le contrôle d'un véhicule téléguidé (appelé « rov »). Ce dernier doit être capable de mesurer la température et le taux d'irradiation l'entourant, ainsi que de visualiser son environnement grâce à une caméra. Enfin il devra être en mesure de réaliser une prise d'échantillon grâce à un bras articulé.

Le logiciel devra permettre l'affichage de la caméra en temps réel, ainsi que des données de température et d'irradiation. Il devra également être capable de stocker ces mesures dans une base de données. Enfin, il devra être capable de prendre des photos et de les archiver.

Les contrôles du rov se feront grâce à une manette de type PS3, comportant un mode de déplacement, et un mode de contrôle de bras.

Enfin, la liaison sera filaire grâce à une transmission port-série de type RS232.

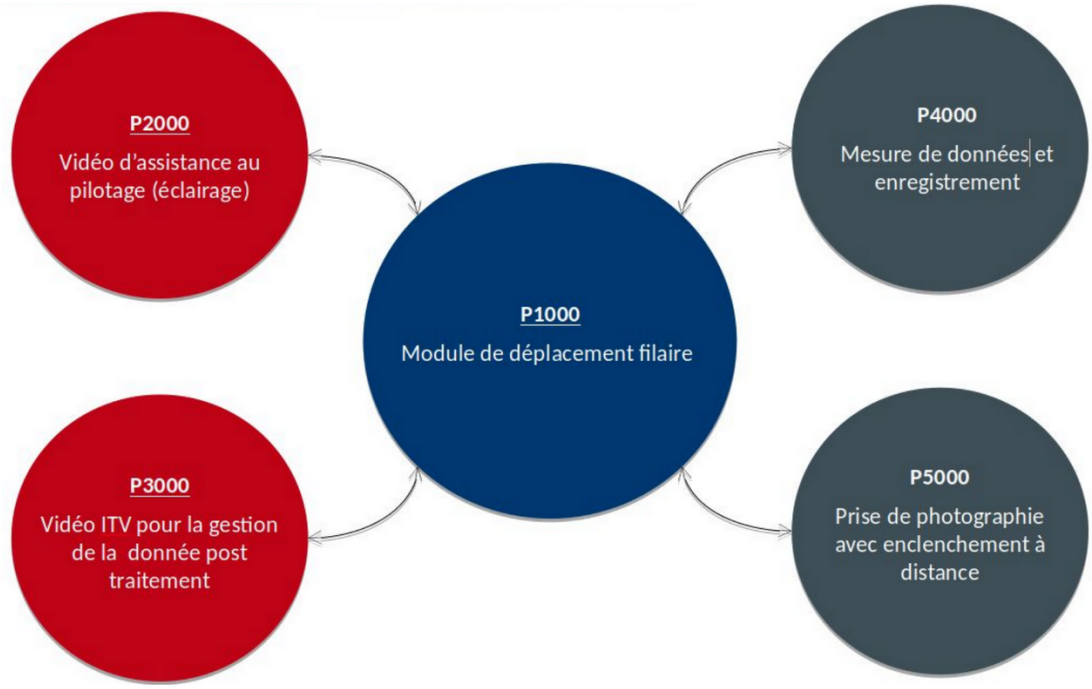


Illustration 1: Schéma du besoin

Présentation

Le rov devra être capable de :

- *se déplacer à des vitesses variables*
- *prélever des échantillons*
- *mesurer la température et le taux d'irradiation*
- *visualiser son environnement via une caméra*
- *signaler les obstacles via un capteur de proximité*

Le logiciel devra être capable de :

- *configurer le rov*
- *démarrer une mission*
- *récupérer, stocker et afficher les mesures prises par le rov*
- *afficher en temps réel le flux vidéo de la caméra*
- *prendre et archiver des photos*

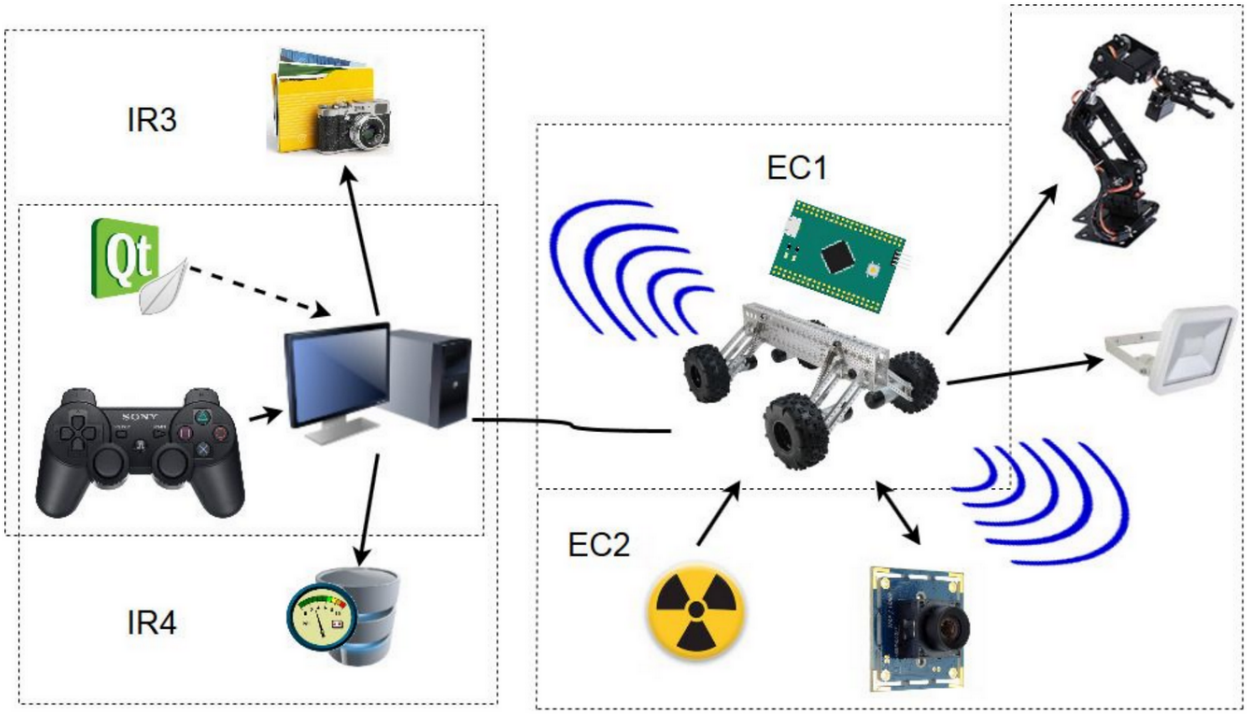


Illustration 2: Vue d'ensemble du système

Identification du travail à réaliser

Étudiants chargés du projet

Option EC :

- DUBUIS Maxime : étudiant 1
- TONNER Grégory : étudiant 2

Option IR :

- REYNIER Jacques : étudiant 3
- BOFFREDO Nicolas : étudiant 4

Répartition des tâches

Étudiant 3 : IR REYNIER Jacques

- ➔ Réception des mesures des capteurs de température et d'irradiation
- ➔ Archivage des mesures dans une base de données
- ➔ Prise en charge d'une manette par le logiciel
- ➔ Envoi de trames liées aux déplacements du rov
- ➔ Envoi de trames liées au bras articulé

Étudiant 4 : IR BOFFREDO Nicolas

- ➔ Permettre le démarrage d'une nouvelle campagne
- ➔ Permettre la visualisation du flux vidéo sur l'IHM
- ➔ Permettre de prendre des captures du retour vidéo
- ➔ Recevoir les données télémétrique
- ➔ Archiver les photos
- ➔ Configurer le contrôle de la caméra

Cas d’utilisation

Présentation

L’acteur de ce système est le technicien chargé de réaliser la mission qui lui est attribuée.

Ce dernier pourra piloter les déplacements du rov et contrôler son bras articulé à l’aide d’une manette. Il aura également une visualisation de l’environnement grâce au retour de la caméra, et pourra consulter les mesures des différents capteurs en temps réel.

Il aura la possibilité de configurer le rov ainsi que sa mission, et pourra au cours de cette dernière réaliser des photos de l’environnement, ainsi que gérer les archives directement dans le logiciel.

Diagramme de cas d’utilisation

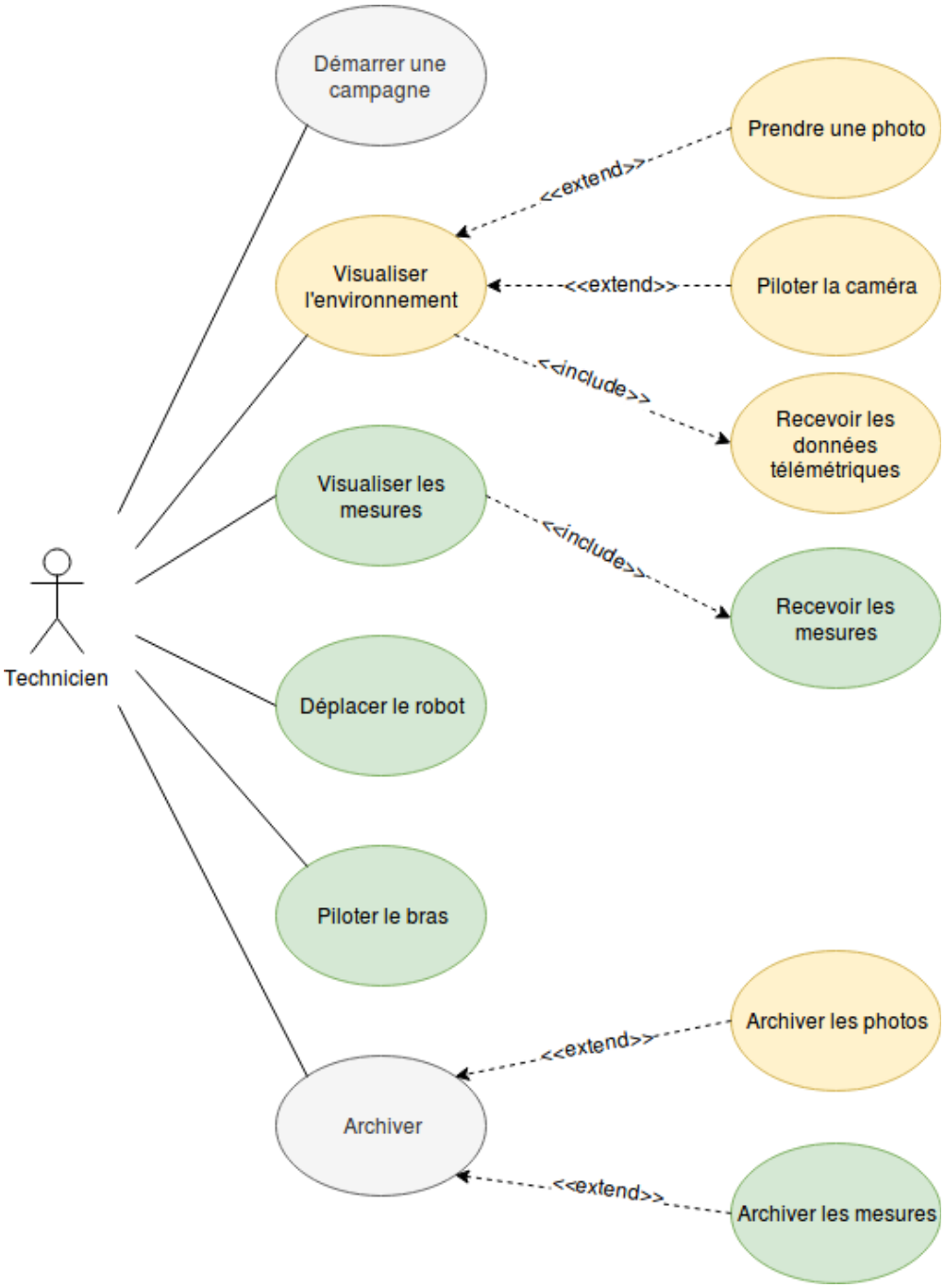


Illustration 3: Diagramme de cas d'utilisation

Descriptif des cas d'utilisation

Cas d'utilisation	Détail
Démarrer une campagne	Réalise la connexion entre le programme et le rov, crée un dossier au nom de la mission contenant les archives de celle-ci.
Visualiser l'environnement	Affiche le flux vidéo sur l'IHM.
Prendre une photo	Prend une photo depuis la caméra.
Piloter la caméra	Contrôle l'orientation de la caméra sur un axe horizontal.
Recevoir les données télémétriques	Reçoit le flux vidéo de la caméra.
Visualiser les mesures	Affiche les mesures des capteurs sur l'IHM. Les capteurs sont : - Température (en °C) - Radioactivité (en …)
Recevoir les mesures	Reçoit les données des capteurs (ci-dessus).
Déplacer le robot	Contrôle les déplacements du rov.
Piloter le bras	Contrôle le bras robotique, 6 axes : - Rotation épaule - Vertical épaule - Vertical coude - Vertical poignet - Rotation poignet - Pression pince
Archiver	Archive les données dans une base de données (Mesures & Photos).
Archiver les photos	Archive les photos dans un sous-répertoire avec la date et l'heure de capture.
Archiver les mesures	Archive les différentes mesures dans une BDD.

Plan de tests de validation

Désignation	Procédure	Résultat attendu	Fonctionnel	Remarques
Une nouvelle campagne est réalisable	Cliquer sur le bouton “Nouvelle Campagne”	Une nouvelle campagne est lancée	Oui	
Contrôler les déplacements du rov	- Avancer (joystick avant) - Reculer (joystick arrière) - Tourner à droite (joystick droite) - Tourner à gauche (joystick gauche)	Tous les déplacements du rov sont possibles	Oui	Des ajustements sont nécessaires pour optimiser les contrôles.
Le bras articulé est contrôlable	Utiliser les touches de la manette : - Joystick gauche - Joystick droit - Croix directionnelle - R1 - L1 - Triangle - Croix - Start	Tous les moteurs du bras sont contrôlables et le bouton start dépose bien le contenu de la pince dans le bac	Oui	Des ajustements sont nécessaires pour optimiser les contrôles.
Les mesures des capteurs apparaissent sur l’IHM	La température, l’irradiation et la distance sont affichées	Les données sont affichées en °C, en µSv/h, et en cm.	Oui	Les tests ont été réalisés par simulateur uniquement.
La caméra est affichée sur l’IHM	Vérifier l’affichage du flux vidéo en temps réel	Le flux vidéo est affiché	Oui	
Prendre une photo	Cliquer sur le bouton “Prendre une photo”	Une photo de la caméra est archivée	Oui	
Archiver les données	Vérifier le contenu de la base de données	Les données sont stockées dans la BDD	Oui	

Diagramme de déploiement

Voici le diagramme de déploiement, représentant l'architecture physique du système et la manière dont les composants sont répartis, ainsi que leurs relations entre eux.

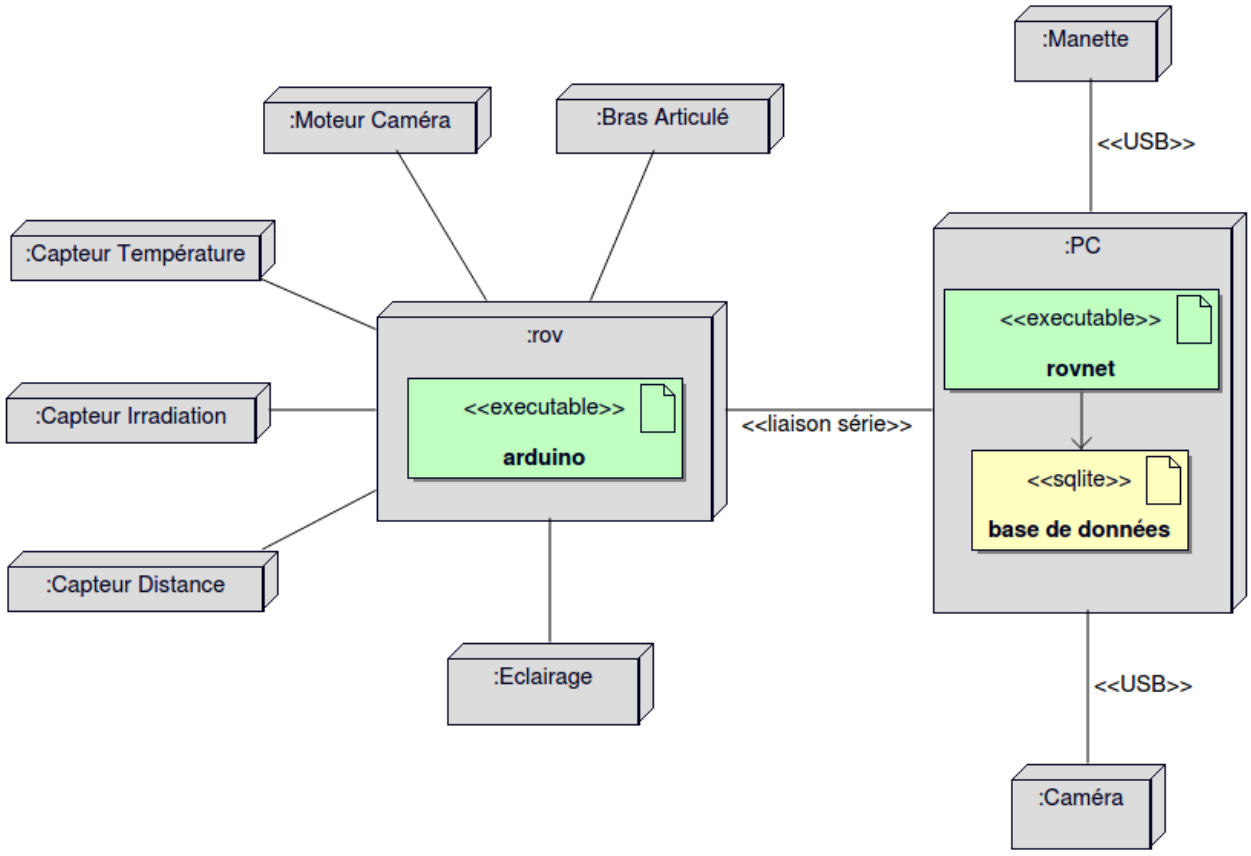


Illustration 4: Diagramme de déploiement

Logiciels utilisés

Qt 5.11.2

Qt est une bibliothèque logicielle orientée objet (API – Application Programming Interface).

Il est principalement dédié à la création d'interfaces graphiques (GUI – Graphical User Interface), fournissant des éléments graphiques prédéfinis nommés ''widgets''.

Ces éléments interagissent entre eux à l'aide d'un mécanisme de signaux et slots, formant la base de la programmation événementielle de Qt.

La version choisie (5.11.2) fournit les modules suivants :

- QtGamepad
  - Permet la prise en charge d'une manette
  - Disponible depuis Qt 5.7.0
- QtMultimedia
  - Permet la prise en charge de la caméra
  - Disponible depuis Qt 4.6
- QtSerialport
  - Permet la communication via une liaison série RS232
  - Disponible depuis Qt 5.1.0
- QtSQL
  - Permet la prise en charge de bases de données relationnelles SQL

Présentation de l'IHM

Maquette

Notre vue conceptuelle d'une IHM adaptée à nos besoins était la suivante :

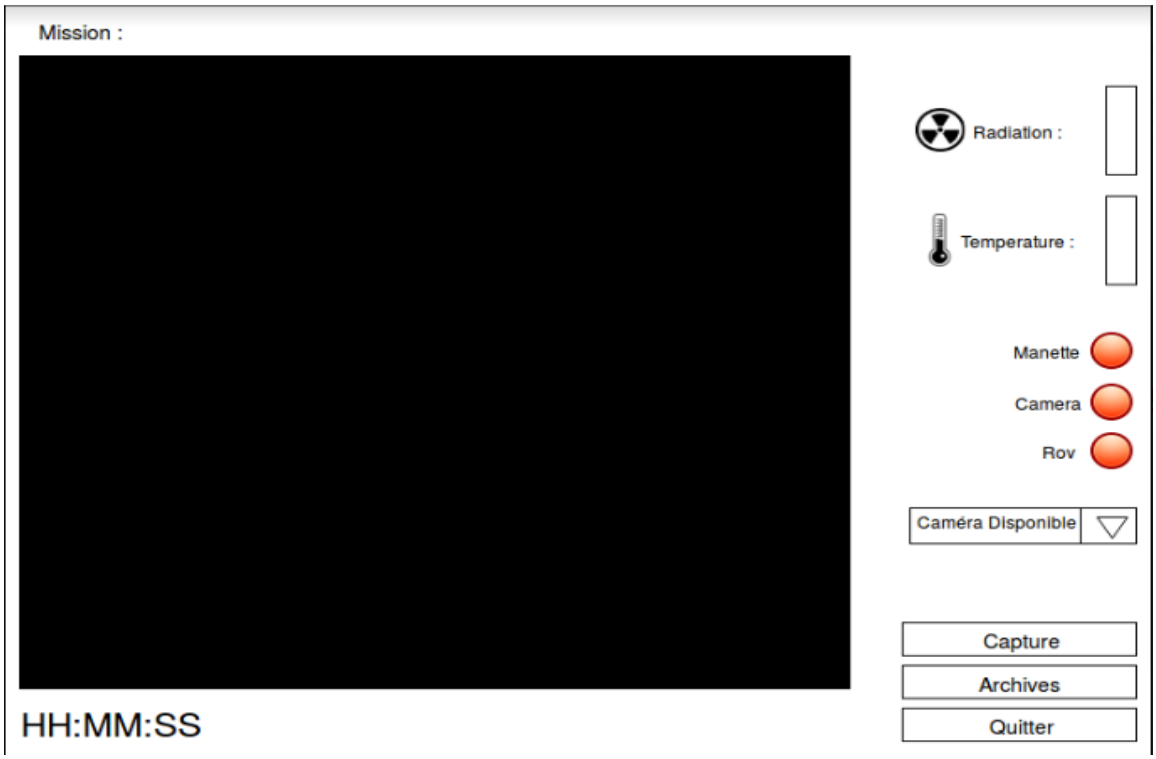


Illustration 5: Maquette prototype de l'IHM

État actuel (v1.0)

L'IHM actuelle est donc la suivante : (voir page suivante)

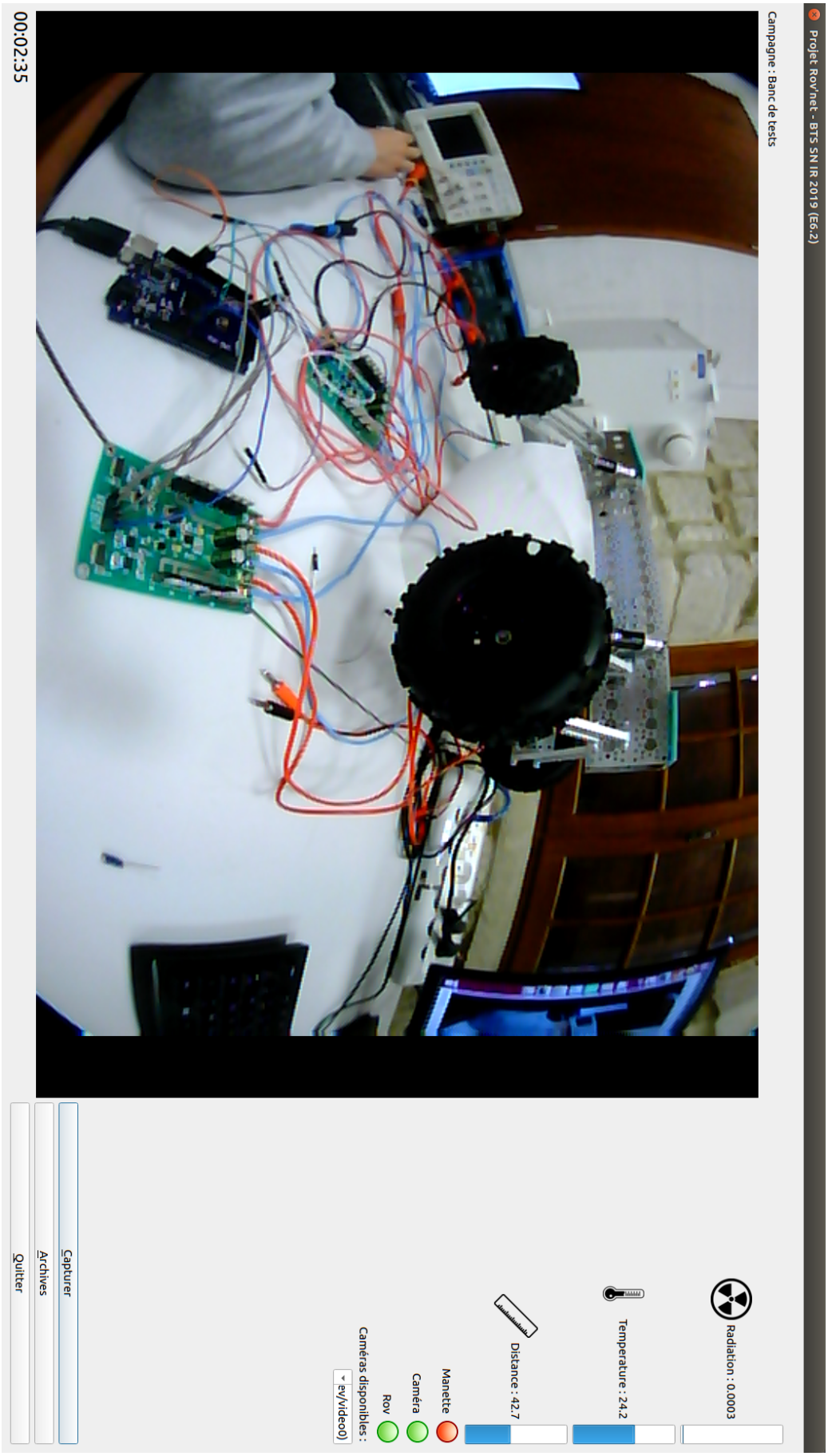


Illustration 6: Capture d'écran de l'IHM

Sur cette IHM, nous pouvons voir les éléments suivants :

- La campagne actuellement en cours (si une campagne est active)
- L’affichage du flux vidéo
- Le temps écoulé depuis le début de la campagne
- Le taux de radiation mesuré
- La température mesurée
- La distance relevée (capteur à ultrason)
- L’état de connexion des différents éléments (caméra, manette, et rov)
- Les caméras disponibles
- Un bouton capturer
- Un bouton archives
- Un bouton quitter

Diagramme de classes

Diagramme original

Dans un premier temps, nous avons élaboré un diagramme de classes du domaine :

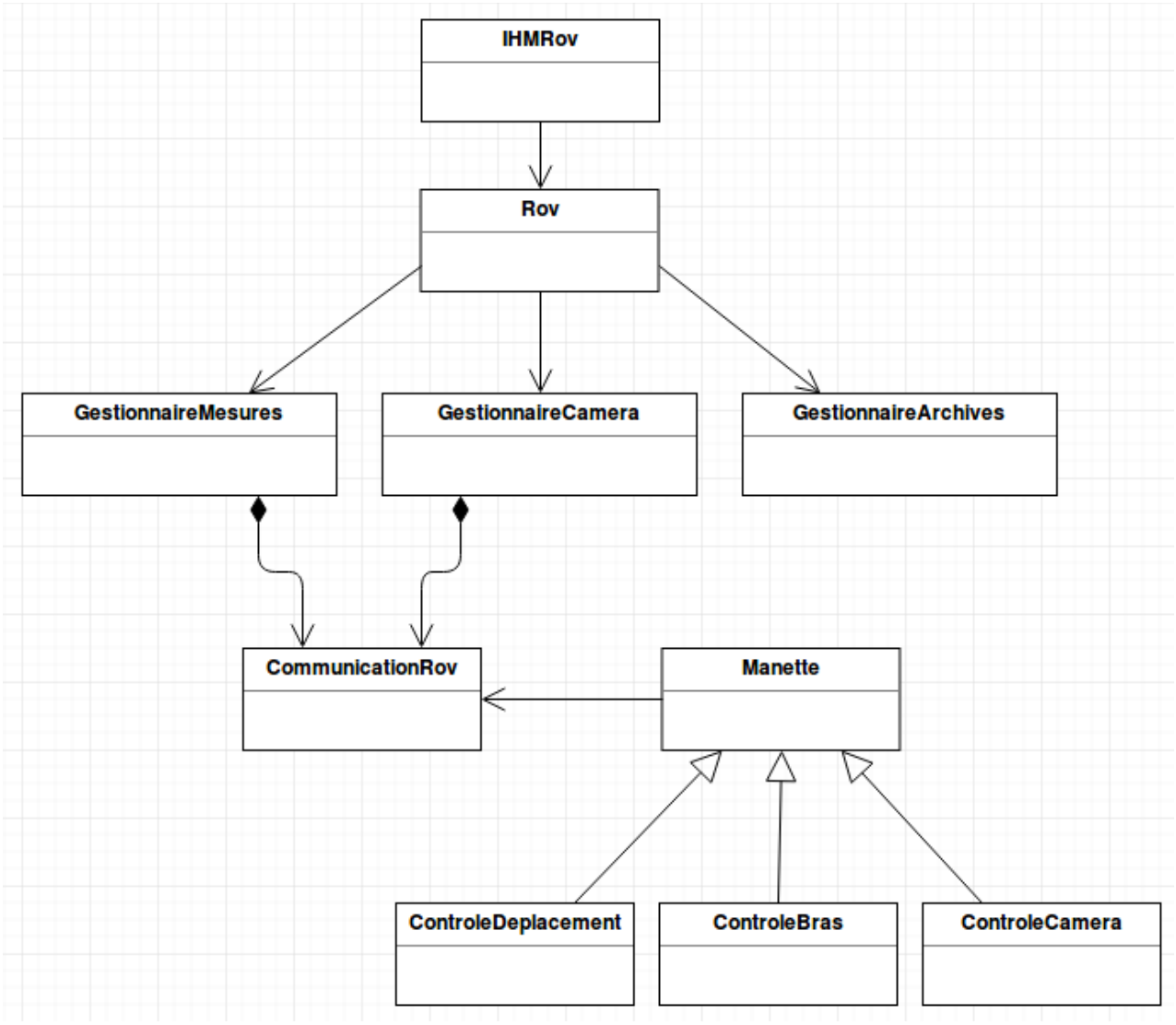


Illustration 7: Diagramme de classes du domaine – v 0.1

Diagramme de classes du domaine

Voici à présent l'état du diagramme de classes du domaine pour la version 1.0 :

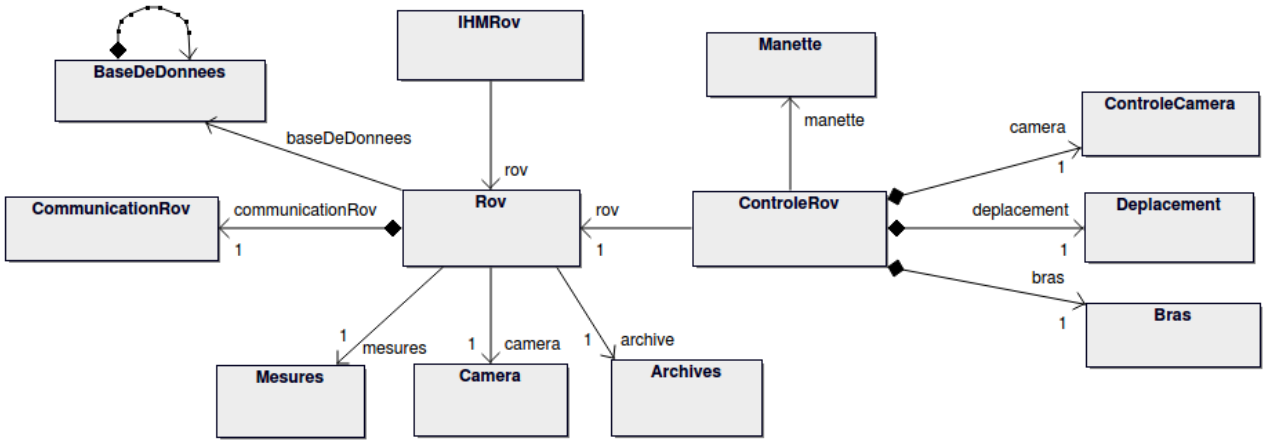


Illustration 8: Diagramme de classes du domaine – v1.0

Description des classes

Nom de la classe	Description	Relations
IHMRov	Gestion de l'IHM du logiciel	Association avec : Rov
Rov	Gestion de la liaison entre les différentes classes du système	Association avec : - BaseDeDonnees - Mesures - Camera - Archives Composition avec : - CommunicationRov
CommunicationRov	Prise en charge du port-série afin d'établir une communication entre le rov et le logiciel	
ControleRov	Gestion de la liaison entre les différentes classes concernant les contrôles du rov	Association avec : - Manette - Rov Composition avec : - ControleCamera - Deplacement - Bras
Manette	Prise en charge d'une manette	
ControleCamera	Gestion des mouvements de la caméra	
Deplacement	Gestion des déplacements du rov	
Bras	Gestion des mouvements du bras articulé	
Mesures	Traite et stocke les relevés des différents capteurs	
Camera	Gestion de la caméra	
Archives	Gestion de l'archivage des photos	
BaseDeDonnees	Gestion d'une base de données comprenant les mesures relevées	BaseDeDonnees est un singleton

Partie personnelle
REYNIER Jacques

Sommaire

Objectifs

Mon travail consiste à mettre en œuvre les contrôles du rov : ses déplacements et la manipulation de son bras articulé via une manette de type PS3.

Je devais également m'occuper de la réception, traitement, et stockage dans une base de données des mesures des différents capteurs (température et irradiation).

Les éléments principaux à mettre en œuvre : une communication par liaison série, une manette PS3, et une base de données SQLite.

Étudiant 3 : IR REYNIER Jacques

- ➔ Réception des mesures des capteurs de température et d'irradiation
- ➔ Archivage des mesures dans une base de données
- ➔ Prise en charge d'une manette par le logiciel
- ➔ Envoi de trames liées aux déplacements du rov
- ➔ Envoi de trames liées au bras articulé

Communication PC / rov

La communication entre le rov et le logiciel se réalise par liaison série.

L'utilisation d'une liaison série nécessite l'utilisation d'un protocole de communication, respectant une nomenclature adaptée à ce projet.

Notre point de vue sera celui du logiciel, nous permettant de distinguer deux types de trames : les trames d'envoi, et les trames de réception.

- Trames d'envoi : PC → rov,
- Trames de réception : rov → PC.

Ce protocole est orienté caractère, et sera codé en ASCII.

Protocole de communication

Format de trame

```
début code valeur fin de trame
```

- Début de trame : **$**
- Champ **code** : taille fixe :
  - 1 caractère pour les trames de réception,
  - 3 caractères pour les trames d'envoi.
- Champ **valeur** : taille variable dépendant de sa valeur.
- Fin de trame : **\n**

Trames de réception

Le code d'une trame de réception est composé d'un unique caractère, spécifiant le type de donnée reçu (température, irradiation, ou distance). Ces trames sont reçues périodiquement. La période varie selon le type de donnée.

Type	Code	Unité	Période
Température	T	°C	12 secondes
Irradiation	R	µSv/h	12 secondes
Distance	D	cm	0.5 secondes

Exemple :

$T20.5\n : Température reçue, égale à 20.5 °C.

Trames d'envoi

Le code d'une trame de réception est composé de 3 caractères :

- Le premier correspond à une action
- Les deux derniers la partie ciblée

Nous avons donc un total de **10 actions** disponibles, pour **7 parties** contrôlables.

Action	Caractère
Avancer	A
Reculer	R
Tourner à droite	D
Tourner à gauche	G
Tourner	T
Lever	L
Ouvrir	O
Fermer	F
Poser	P
Attraper	E

Partie	Caractères
Roues	RO
Epaule	EP
Coude	CO
Poignet	PO
Pince	PI
Bras	BR
Camera	CA

Nous obtenons pour les trames d'envoi, le tableau de définition de codes suivant :

Cible	Type	Code	Valeur	Unité
Roues	Avancer	ARO	0 <-> 3	% Vitesse
	Reculer	RRO	0 <-> 3	% Vitesse
	Tourner à droite	DRO	0 <-> 3	% Vitesse
	Tourner à gauche	GRO	0 <-> 3	% Vitesse
Bras	Epaule	TEP	-1 / 0 / 1	Pas (10°)
		LEP	-1 / 0 / 1	Pas (10°)
	Coude	LCO	-1 / 0 / 1	Pas (10°)
	Poignet	LPO	-1 / 0 / 1	Pas (10°)
		TPO	-1 / 0 / 1	Pas (10°)
	Pince	OPI	0 / 1	Booléen
		FPI	0 / 1	Booléen
	Poser	PBR	0 / 1	Booléen
Caméra	Tourner	TCA	-1 / 0 / 1	Pas (10°)

Exemple :

$TEP1\n : Tourner l'épaule de 10° supplémentaire.

Utilisation de la manette

La manette de contrôle du rov est une manette pour Sony Playstation 3 (*PS3*). Contrairement aux spécifications du cahier des charges nous indiquant l'utilisation de deux manettes pour le contrôle du rov, nous avons fait le choix de n'en utiliser qu'une.

Cela apporte l'avantage d'éviter une confusion entre deux manettes ainsi que des erreurs de manipulation (exemple : déplacement du rov tandis que l'utilisateur souhaite manipuler le bras). De plus, cela facilite la configuration du logiciel, tout en réduisant les coûts à l'achat.

La manette possède 2 modes. Un mode « Bras » et un autre « Déplacements ». Le passage d'un mode à l'autre se fait par appui sur le bouton « select ». Par défaut, la manette se trouve en mode Déplacements.

Illustration 9: Utilisation de la manette - changer de mode

Schémas d'utilisation

Mode Bras

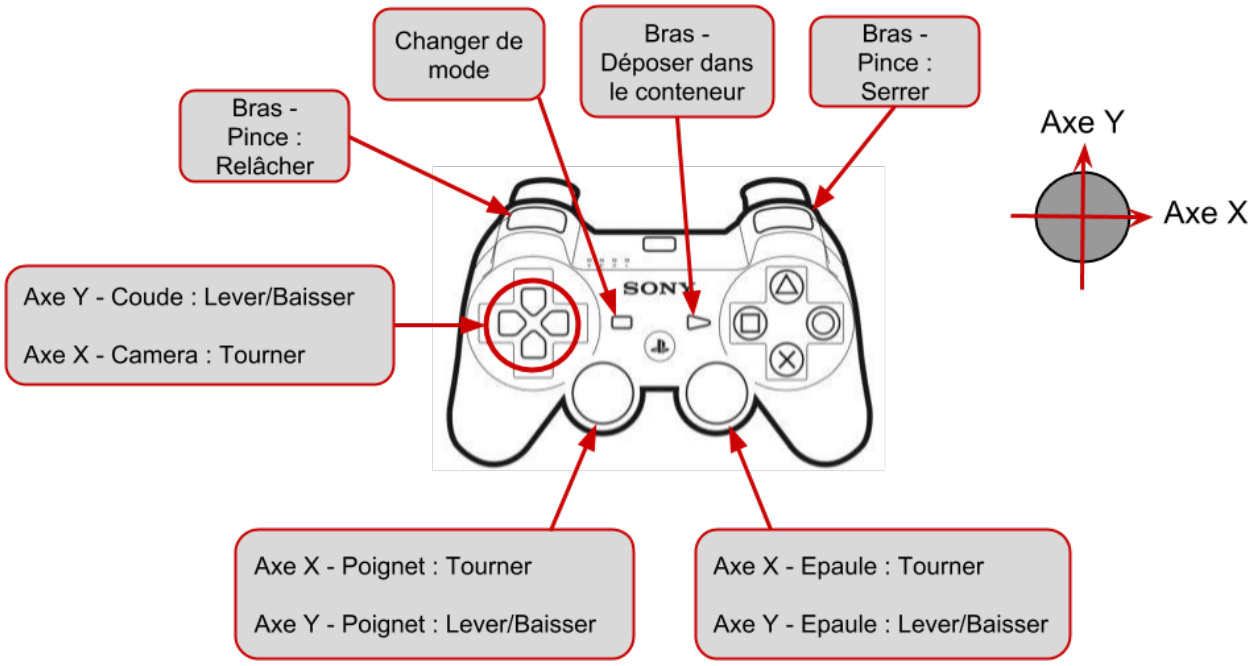


Illustration 10: Utilisation de la manette - mode Bras

Mode Déplacements

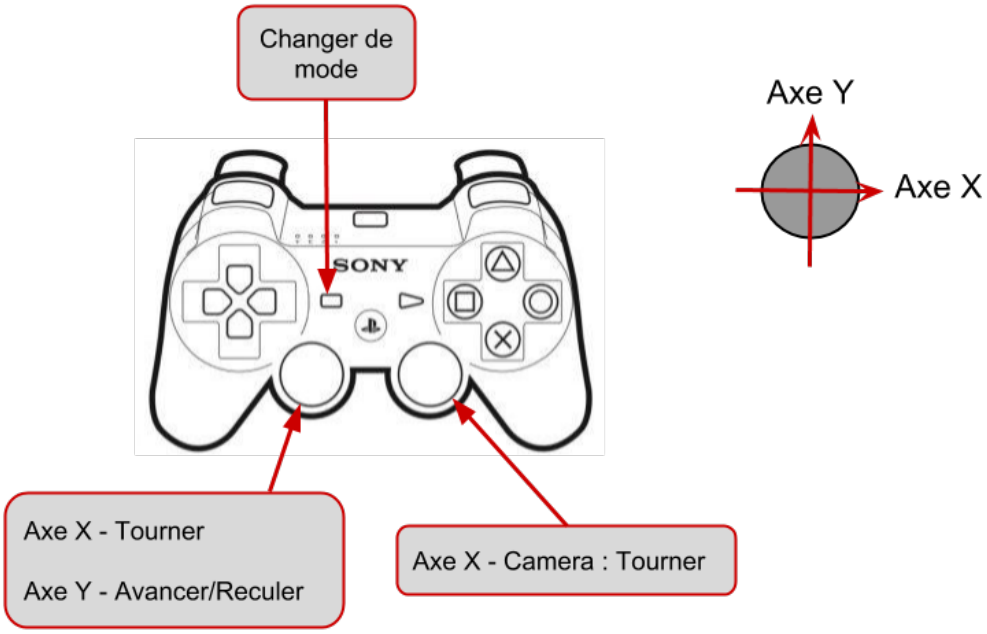


Illustration 11: Utilisation de la manette - mode Déplacements

Installation de la manette

Suite à la contrainte qu'impose la librairie *QtGamepad* à être compatible uniquement avec les manettes de Xbox360, nous devons procéder à une manipulation à chaque branchement de la manette.

Nous allons pour cela utiliser un paquet nommé *xboxdrv.* Ce dernier permet d'émuler n'importe quelle manette en une manette de Xbox360.

Il peut être installé via la commande suivante :

```
$ sudo apt-get install xboxdrv
```

Ce dernier s'exécute comme suit :

```
$ sudo xboxdrv --detach-kernel-driver -s
xboxdrv 0.8.5 - http://pingus.seul.org/~grumbel/xboxdrv/
Copyright © 2008-2011 Ingo Ruhnke <grumbel@gmx.de>
Licensed under GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it under certain
conditions; see the file COPYING for details.

Controller:        PLAYSTATION(R)3 Controller
Vendor/Product:    054c:0268
USB Path:          002:003
Controller Type:   Playstation 3 USB

Your Xbox/Xbox360 controller should now be available as:
  /dev/input/js0
  /dev/input/event12

Press Ctrl-c to quit
```

Gestion des contrôles du rov

Classes concernées

La gestion des contrôles du rov est réalisée à partir des classes suivantes :

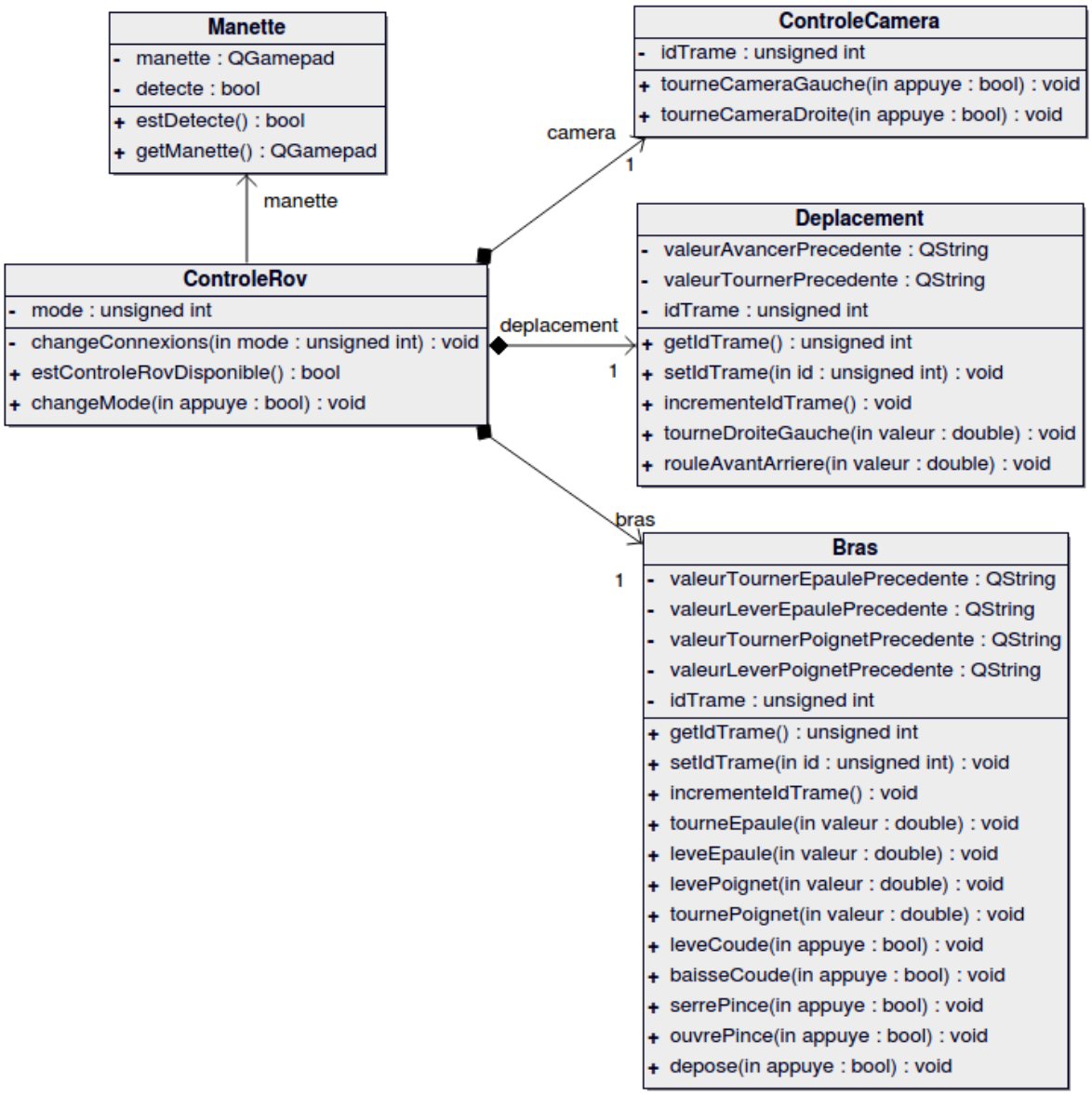
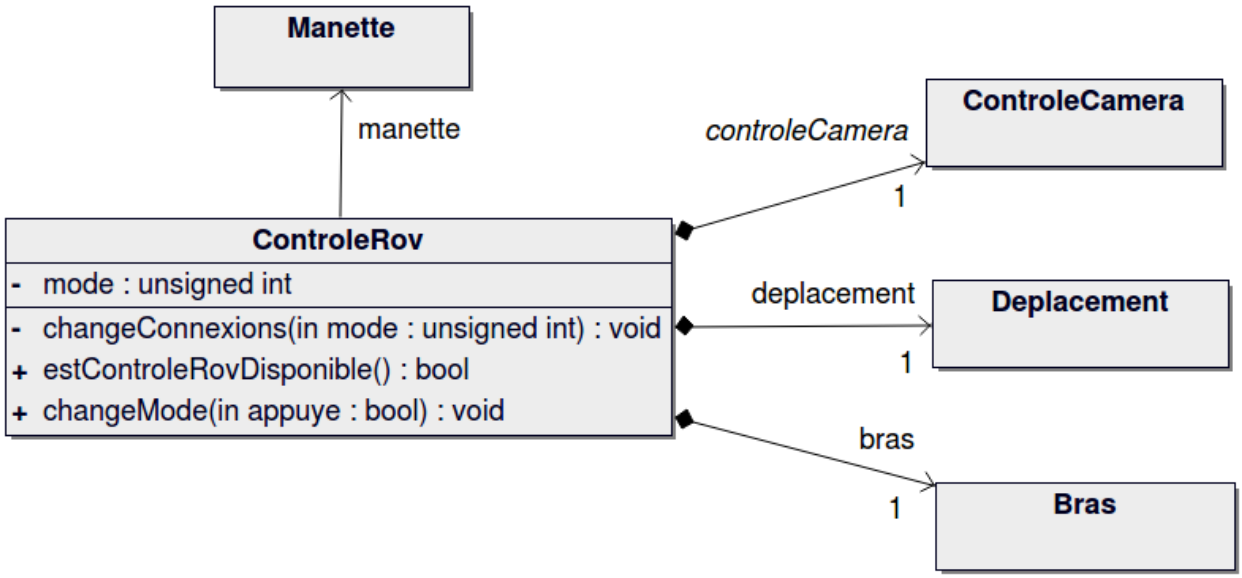


Illustration 12: Diagramme de conception - Zoom sur la partie contrôle du rov

La classe ControleRov



Les contrôles du rov sont assurés par une classe principale : *ControleRov*. Cette dernière est liée aux autres classes par composition (*Exemple : ControleRov est composé d'une manette*).

Attributs

- mode : unsigned int.
  - Mode actuel de la manette. Le mode de la manette va désigner les fonctionnalitées de celle-ci. Cet attribut variera entre 2 valeurs, 0 et 1 :
    - 0 : contrôle du bras,
    - 1 : contrôle des déplacements.
- manette : Manette.
  - Instance d'un objet manette, par association.
- controleCamera : ControleCamera.
  - Instance d'un objet ControleCamera, par composition.
- deplacement : Deplacement.
  - Instance d'un objet Deplacement, par composition.
- bras : Bras.
  - Instance d'un objet Bras, par composition.

Méthodes

➢ ControleRov : constructeur.

```
ControleRov::ControleRov(QObject *parent, Rov *rov) : QObject(parent), rov(rov),
mode(MODE_DEPLACEMENT)
{
    this->bras = new Bras(this);
    this->deplacement = new Deplacement(this);
    this->controleCamera = new ControleCamera(this);
    this->manette = new Manette(this);

    if(manette->estDetecte())
    {
        connect(manette->getManette(), &QGamepad::axisLeftXChanged, deplacement,
&Deplacement::tourneDroiteGauche);
        connect(manette->getManette(), &QGamepad::axisLeftYChanged, deplacement,
&Deplacement::rouleAvantArriere);
        connect(manette->getManette(), &QGamepad::buttonLeftChanged,
controleCamera, &ControleCamera::tourneCameraGauche);
        connect(manette->getManette(), &QGamepad::buttonRightChanged,
controleCamera, &ControleCamera::tourneCameraDroite);
        connect(manette->getManette(), &QGamepad::buttonSelectChanged, this,
&ControleRov::changeMode);
    }

    // Connexions entre le signal émis d'une trame à envoyer au slot d'envoi de
trame de la classe CommunicationRov.
    connect(bras, &Bras::trameCree, this->rov->getCommunicationRov(),
&CommunicationRov::envoieTrame);

    connect(deplacement, &Deplacement::trameCree, this->rov-
>getCommunicationRov(), &CommunicationRov::envoieTrame);
}
```

- Instancie un objet *ControleRov* possédant un objet *bras*, *deplacement*, *controleCamera*, et *manette*.
  - Réalise les connexions des signaux de la manette en mode deplacement.

➢ EstControleRovDisponible : bool.

```
bool ControleRov::estControleRovDisponible() const
{
    return manette->estDetecte();
}
```

- Indique si les contrôles sont disponibles, ce qui correspond à la disponibilité de la manette.

➢ changeMode : void.

```
void ControleRov::changeMode(bool appuye)
{
    if (appuye == 1 && mode == MODE_DEPLACEMENT)
    {
        qDebug() << Q_FUNC_INFO << "Passage en mode Bras";
        mode = MODE_BRAS;
        changeConnexions(MODE_BRAS);
    }
    else if (appuye == 1 && mode == MODE_BRAS)
    {
        qDebug() << Q_FUNC_INFO << "Passage en mode Deplacement";
        mode = MODE_DEPLACEMENT;
        changeConnexions(MODE_DEPLACEMENT);
    }
    else if (mode != MODE_DEPLACEMENT && mode != MODE_BRAS)
    {
        qDebug() << Q_FUNC_INFO << "Erreur de mode ! Retour au mode de
déplacement";
        mode = MODE_DEPLACEMENT;
        changeConnexions(MODE_DEPLACEMENT);
    }
}
```

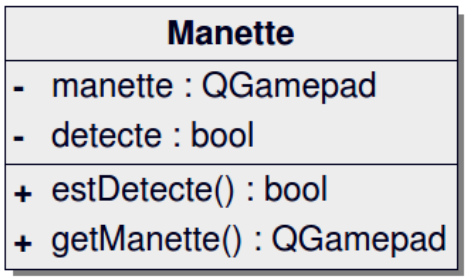
- Change le mode de la manette.
  - Affecte à l'attribut mode le nouveau mode choisi.
  - Réalise les changements de connexion pour correspondre au nouveau mode :
    - Fait appel à la méthode *changeConnexions*.

➢ changeConnexions : void

```
void ControleRov::changeConnexions(int mode)
{
    if (mode == MODE_BRAS)
    {
        disconnect(manette->getManette(), &QGamepad::axisLeftXChanged,
deplacement, &Deplacement::tourneDroiteGauche);
        disconnect(manette->getManette(), &QGamepad::axisLeftYChanged,
deplacement, &Deplacement::rouleAvantArriere);
        connect(manette->getManette(), &QGamepad::axisLeftXChanged, bras,
&Bras::tournePoignet);
        connect(manette->getManette(), &QGamepad::axisLeftYChanged, bras,
&Bras::levePoignet);
       [...]
        connect(manette->getManette(), &QGamepad::buttonL1Changed, bras,
&Bras::lachePince);
        connect(manette->getManette(), &QGamepad::buttonStartChanged, bras,
&Bras::depose);
    }
    else if (mode == MODE_DEPLACEMENT)
    {
        disconnect(manette->getManette(), &QGamepad::axisLeftXChanged, bras,
&Bras::tournePoignet);
        disconnect(manette->getManette(), &QGamepad::axisLeftYChanged, bras,
&Bras::levePoignet);
        [...]
        disconnect(manette->getManette(), &QGamepad::buttonStartChanged,
bras, &Bras::depose);
        connect(manette->getManette(), &QGamepad::axisLeftXChanged,
deplacement, &Deplacement::tourneDroiteGauche);
        connect(manette->getManette(), &QGamepad::axisLeftYChanged,
deplacement, &Deplacement::rouleAvantArriere);
    }
    else
    {
        qDebug() << Q_FUNC_INFO << "erreur mode inconnu !";
    }
}
```

- Change les slots correspondants aux signaux envoyés par la manette.
  - Ajoute ou supprime des connexions.
  - Lie les signaux de la manette à la méthode traitant la partie du rov concerné, suivant le schéma d'utilisation de la manette (disponible page 25).

La classe Manette



Gestion de la manette. Cette classe permet la connexion d'une manette au logiciel. Elle va s'appuyer sur le module *QtGamepad*.

Attributs

- manette : QGamepad.
  - Instance de la manette connectée.
    - *Objet QGamepad, appartenant à la librairie QtGamepad*.

- detecte : bool.
  - Retourne l'état de connexion de la manette (*true* ou *false*).

Les méthodes

- Manette : constructeur

```
Manette::Manette(QObject *parent) : QObject(parent), manette(nullptr),
detecte(false)
{
    #ifndef QT_NO_DEBUG_OUTPUT
    QLoggingCategory::setFilterRules(QStringLiteral("qt.gamepad.debug=true"));
    #endif

    auto manettes = QGamepadManager::instance()->connectedGamepads();

    if (manettes.isEmpty())
    {
        qDebug() << Q_FUNC_INFO << "Aucune manette détectée !";
        detecte = false;
    }
    else
    {
        manette = new QGamepad(*manettes.begin(), this);
        detecte = true;
    }
}
```

  - Instancie un objet Manette :
    - Recherche les manettes connectées au poste.
    - S'il n'y a pas de manette connectée, un message d'erreur est envoyé,
    - Sinon, on instancie un objet QGamepad correspondant à la/les manette(s) trouvée(s).

- estDetecte : bool

```
bool Manette::estDetecte() const
{
    return detecte;
}
```

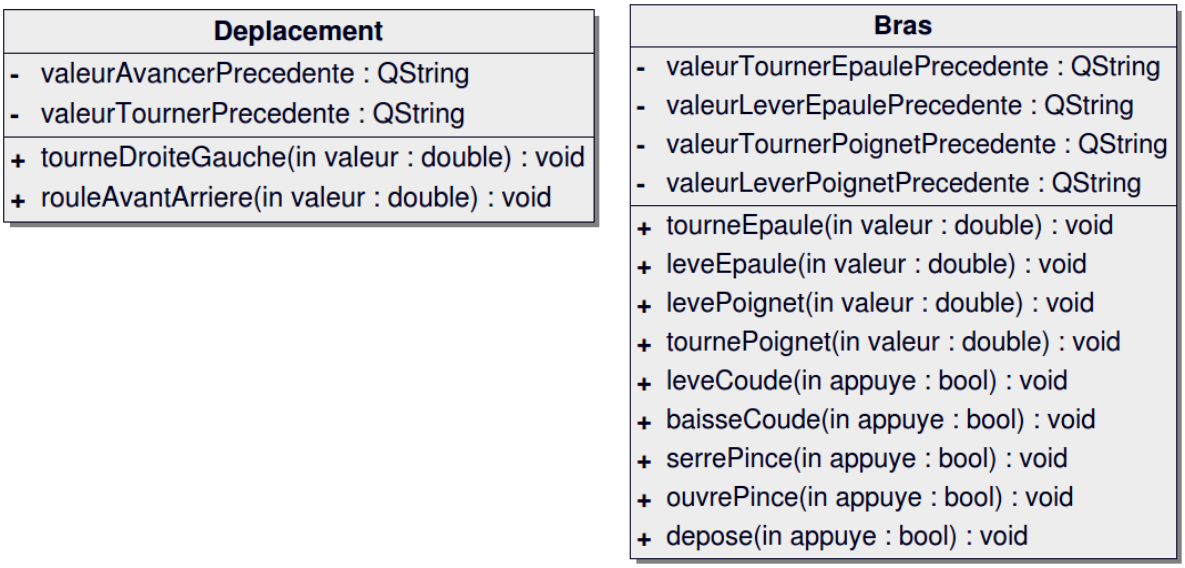
  - Retourne l'état de connexion de la manette (*true* ou *false*).

- getManette : QGamepad

```
QGamepad * Manette::getManette()
{
    return manette;
}
```

  - Retourne l'attribut manette : la manette actuellement connectée.

Les classes Deplacement et Bras



Réalise les contrôles du bras articulé et des déplacements du rov. Ces classes récupèrent les signaux émis par la manette, et émettent en retour les trames correspondantes.

Ces deux classes possèdent la même architecture avec des méthodes types :

- Des attributs de précédentes valeurs émises,
- Plusieurs méthodes correspondant aux mouvements demandés. Pour une meilleure compréhension et pour le reste du document, nous appellerons ces méthodes des **actions**.

Attributs

- valeur*[Action]*Precedente : void
  - Dernière donnée émise par l'action :
    - Cet attribut est utilisé dans les méthodes d'action, afin de ne pas émettre deux fois de suite la même trame.
    - *Exemple : valeurAvancePrecedente, valeurTourneEpaulePrecedente*, … .

Méthodes d'actions

Les méthodes d'actions se scindent en deux types, selon leur mode d'activation :

- Par mouvement d'un joystick :
  - La manette émet une valeur comprise entre -1 et 1 pour chaque axe (x et y), correspondant à la position dans laquelle le joystick se trouve.
  - La manette envoie une trame à chaque changement de position du joystick.

- Par pression d'un bouton :
  - Émet un booléen correspondant à l'appui sur le bouton :
    - 1 : le bouton est appuyé,
    - 0 : le bouton est relâché.

Nous pouvons donc prendre deux méthodes d'exemple, chacune correspondant à un type d'activation :

- leveEpaule pour une activation par joystick,
- serrePince pour une activation par appui.

Détail des méthodes :

- leveEpaule : void

```
void Bras::leveEpaule(double valeur)       // Joystick droit axe y
{
    QString trame = "$LEP";
    int direction = 0;      // Par défaut, la direction est indiquée à 0

    if (valeur <= -0.5)
        direction = 1;

    else if (valeur >= 0.5)
        direction = -1;

    if (valeurLeveEpaulePrecedente != direction)
    {
        valeurLeveEpaulePrecedente = direction;
        trame += QString::number(direction) + "\n";
       emit trameCree(trame);
    }
}
```

  - Émet une trame permettant de lever ou baisser l'épaule du bras articulé.
    - Reçoit la position du joystick compris entre -1 et 1 :
      - Si la valeur reçue est inférieure à -0.5, alors le pas sera égal à 1,
      - Si la valeur est supérieure à 0.5, alors le pas sera égal à -1.
    - Si le pas à envoyer n'est pas identique au dernier pas émis, alors crée la nouvelle trame.
    - Stocke le pas émis dans l'attribut *valeurLeverEpaulePrecedente,* puis envoie la trame au rov.
      - La nouvelle trame sera émise uniquement si elle est différente de la précédente (*réponse à un bug visualisé lors des premiers tests, dû à la haute sensibilité du joystick*).

- serrerPince : void

```
void Bras::serrePince(bool appuye)          // R1
{
    QString trame = "$FPI" + QString::number(appuye) + "\n";

    emit trameCree(trame);
}
```

  - Émet une trame permettant de serrer la pince du bras articulé.
    - Crée une trame comprenant la valeur booléenne de l'appui :
      - 0 : la pince ne serre pas,
      - 1 : la pince serre.

Gestion des mesures

Les mesures récupérées proviennent d'un capteur de température, d'un compteur Geiger, ainsi que d'un capteur de proximité. Ces dernières sont transmises par liaison série, puis réceptionnées via la classe *CommunicationRov*.

Le logiciel va afficher les dernières données reçues sur l'IHM, et archiver les mesures de température et d'irradiation dans une base de données, si l'utilisateur le souhaite, avec une fréquence paramétrable.

Diagramme de séquence : recevoir les mesures

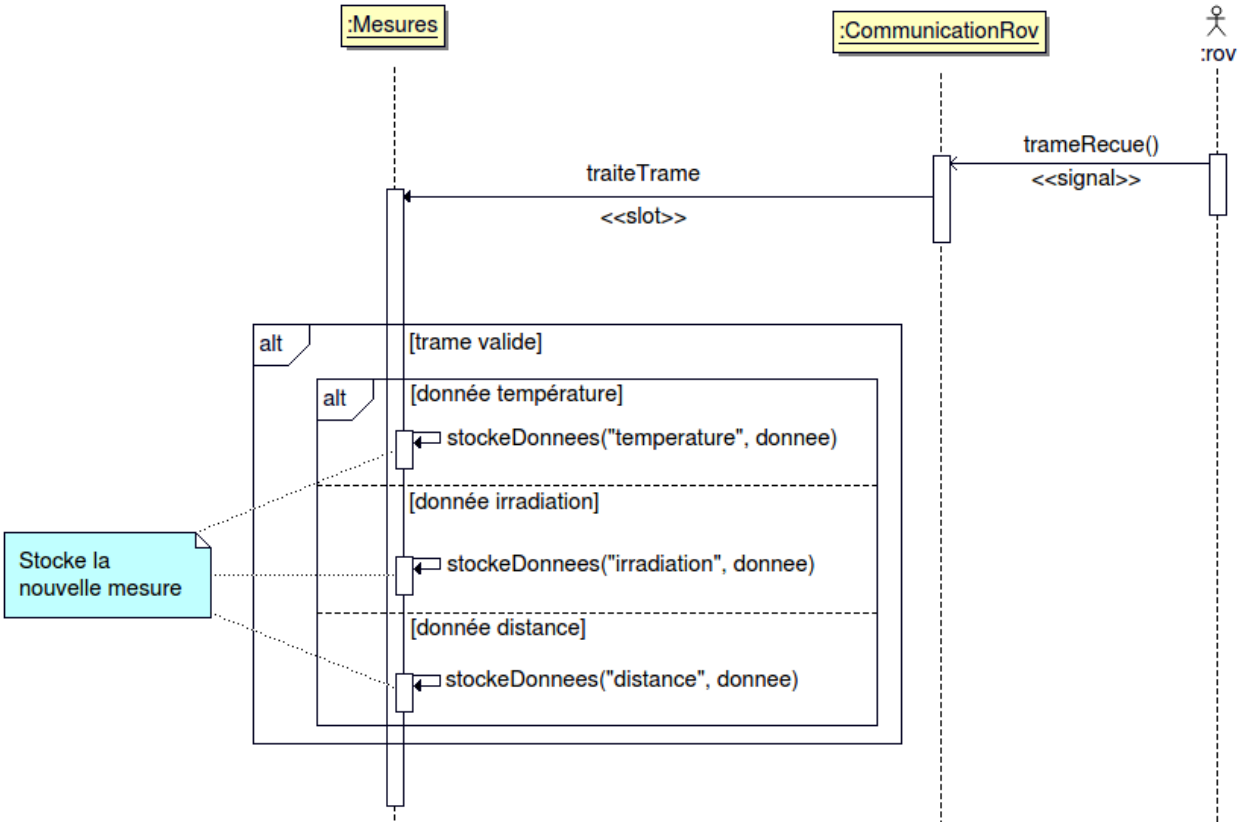
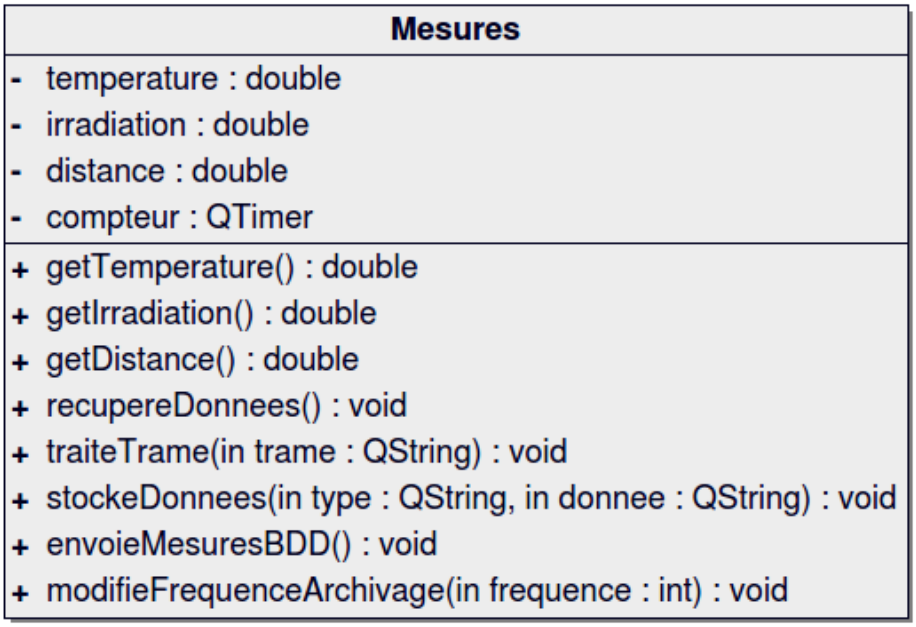


Illustration 13: Diagramme de séquence – Recevoir les mesures

Voici le déroulement de la réception des données :

1. Un objet *CommunicationRov* reçoit une nouvelle trame, et émet un signal nommé *trameRecue*, comportant la trame reçue.
2. Le slot *traiteTrame* de la classe *Mesures* est déclenché grâce à la connexion établie dans la classe *Rov.*
3. La méthode *traiteTrame* vérifie la validité de la trame, puis la traite et stocke son contenu dans les attributs correspondants via la méthode *stockeDonnees*.

La classe Mesures



La classe Mesures permet le traitement et le stockage des dernières données reçues des capteurs de température et d'irradiation.

Attributs

- temperature : double.
  - Dernière température reçue, en °C.
- irradiation : double.
  - Dernier taux d'irradiation reçu, en µSv/h.
- distance : double.
  - Dernière mesure du capteur de proximité reçu, en cm.
- compteur : QTimer.
  - Compteur émettant un signal toutes les x secondes.
  - Ce signal sera connecté au slot envoieMesuresBDD.

Méthodes

- Mesures : Mesures (constructeur).

```
Mesures::Mesures(QObject *parent) : QObject(parent), temperature(-999.0),
irradiation(-999.0), distance(-999.0)
{
    compteur = new QTimer(this);
    connect(compteur, SIGNAL(timeout()), this, SLOT(envoieMesuresBDD()));
    compteur->start(30000);

}
```

  - Instancie l'objet Mesures, avec -999.0 en tant que valeur initiale pour les attributs temperature, irradiation, et distance
  - Démarre un compteur émettant un signal timeout toutes les 30 secondes (modifiable par la suite).
    - Ce signal est connecté à la méthode envoieMesuresBDD.

- getTemperature : float.

```
double Mesures::getTemperature() const
{
    return this->temperature;
}
```

  - Retourne la valeur de l'attribut temperature.

- getIrradiation : double.

```
double Mesures::getIrradiation() const
{
    return this->irradiation;
}
```

  - Retourne le contenu de l'attribut irradiation.

- getDistance : double.

```
double Mesures::getDistance() const
{
    return this->distance;
}
```

  - Retourne le contenu de l'attribut distance.

- stockeDonnees : void.

```
void Mesures::stockeDonnees(QString type, QString donnee)
{
    if (type == "irradiation")
    {
        this->irradiation = donnee.toDouble();
        emit irradiationActualisee(this->irradiation);
    }

    if (type == "temperature")
    {
        this->temperature = donnee.toDouble();
        emit temperatureActualisee(this->temperature);
    }

    if (type == "distance")
    {
        this->distance = donnee.toDouble();
        emit distanceActualisee(this->distance);
    }
}
```

  - Stocke la donnée passée en argument dans l'attribut correspondant au type
  - Émet un signal indiquant que la donnée reçue est actualisée.

- envoieMesuresBDD : void

```
void Mesures::envoieMesuresBDD()
{
    emit mesuresBDDPrete(this->temperature, this->irradiation);
}
```

  - Émet un signal à destination de la classe baseDeDonnees comprenant la température et l'irradiation actuelle
    - Cette méthode est activée toutes les x secondes. La connexion entre le signal émis et le slot cible est réalisé dans le constructeur de la classe rov.

- traiteTrame : void.

```
void Mesures::traiteTrame(QString trame)    // Exemple de trame : "$T32.5\n"
{
    bool trameValide = true;

    if (trame.startsWith("$") && trame.endsWith("\n"))
    {
        trame.remove(QChar('$'));
        trame.remove(QChar('\n'));

        for(int i = 1; i < trame.length(); i++)
        {
            if(!trame[i].isDigit() && trame[i] != ".")
                trameValide = false;
        }
    }
    else
        trameValide = false;

    if(trameValide)
    {
        if(trame.startsWith('T'))
            stockeDonnees("temperature", trame.remove('T'));
        else if(trame.startsWith('R'))
            stockeDonnees("irradiation", trame.remove('R'));
        else if(trame.startsWith('D'))
            stockeDonnees("distance", trame.remove('D'));
        else
            trameValide = false;

    }

    if(!trameValide)
        qDebug() << Q_FUNC_INFO << "ERREUR ! Trame invalide";
}
```

  - Traite la trame reçue en argument :
    - Vérifie la validité de la trame (débute par le symbole ‘$’ et termine par ‘\n’)
    - Retire les symboles de début et de fin de trame
    - Vérifie si le contenu de la trame à la suite du caractère de type est bien un nombre (entier ou réel)
    - Fait appel à la fonction *stockeDonnees* afin de stocker les données trouvées
    - Si la trame a été détectée comme non valide, affiche un message d’erreur.

- modifieFrequenceArchivage : void.

```
void Mesures::modifieFrequenceArchivage(int frequence)
{
    compteur->setInterval(frequence * 1000);
}
```

  - Modifie la fréquence d'archivage.
    - La fréquence d'archivage est régie par la période du compteur. Nous modifions donc l'intervalle du compteur avec comme argument la nouvelle fréquence souhaitée en secondes, multipliée par 1000 pour l'avoir en millisecondes.

La mesure d'irradiation

La mesure de la radioactivité est réalisée à l'aide d'un compteur Geiger. Ce dernier a pour but de relever le nombre de particules radioactives dans l'air. Dans notre cas, le compteur choisi relève les particules « Bêta » et « Gamma ».

Les particules radioactives

Les noyaux radioactifs émettent trois types de rayonnements : les rayonnements alpha, bêta, et gamma. Ces rayonnements sont émis lorsqu'un noyau, instable, va chercher à se stabiliser.

Rayonnement	Description
Alpha (α)	Émission d'une « particule alpha », un noyau d'hélium. Cette émission va réduire le nombre de masse de 4 et le numéro atomique de 2 chez le noyau atomique père : $^{A}_{Z}\mathrm{X} \rightarrow {}^{A-4}_{Z-2}\mathrm{Y} + {}^{4}_{2}\mathrm{He}$
Bêta (β)	Émission d'un électron ou d'un positron, permettant de transformer un neutron en proton, ou l'inverse. Nous parlons de désintégrations bêta moins ou bêta plus : Désintégration β^- : neutron, proton, electron, antineutrino $n \rightarrow p^+ + e^- + \bar{\nu}_e$ Désintégration β^+ : proton, neutron, positron, neutrino $p^+ \rightarrow n + e^+ + \nu_e$
Gamma (γ)	Émission d'un ou plusieurs photons, visant à désexciter le noyau. Ce dernier intervient généralement après l'émission d'une particule alpha ou bêta, ces derniers excitant le noyau père. *Exemple avec un noyau de nickel 60 * (noyau excité) :* Nickel 60 excité, Nickel 60 à l'état fondamental, Photons gamma $^{60}\mathrm{Ni}^* \rightarrow {}^{60}\mathrm{Ni} + 2\,\gamma$

Les unités de mesure

Le taux de radiation peut être mesuré avec 3 unités de grandeurs :

- Le Becquerel
- Le Gray
- Le Sievert

Chacune de ces unités permettent de mesurer un phénomène précis lié à la radioactivité. Ces derniers sont réunis dans le tableau explicatif suivant :

PRINCIPALES GRANDEURS ET UNITÉS INTERNATIONALES UTILISÉES DANS LE DOMAINE DES RAYONNEMENTS IONISANTS		
Notion / grandeur mesurée	**Unité**	**Définition / caractéristique**
Energie de rayonnement (E)	électronvolt (eV)	1 électronvolt = 1,6. 10^{-19} Joule
Activité d'un corps radioactif (A)	becquerel (Bq)	Nombre de désintégrations par seconde. Réduite de moitié au bout d'une période, au quart au bout de 2 périodes, etc.
Dose absorbée par un organisme (D)	gray (Gy)	Energie absorbée par unité de masse. Dose (Gy) = Energie (Joule) / Masse (kg)
Dose équivalente (H_T)	sievert (Sv)	Dose absorbée x facteur de pondération radiologique. Ce facteur de pondération radiologique (W_R) dépend du type de rayonnement, il vaut 1 pour les rayons X, gamma et bêta, vaut 20 pour les particules alpha, et est variable pour les neutrons (en fonction de leur énergie). En effet, à dose absorbée égale, les effets biologiques dépendent de la nature des rayonnements (α, β, γ, X ou neutrons). La dose équivalente est dite « engagée » quand elle résulte de l'incorporation dans l'organisme de radioéléments jusqu'à l'élimination complète de ceux-ci, soit par élimination biologique, soit par décroissance physique.
Dose efficace (E)	sievert (Sv)	**Exposition externe** Somme des doses équivalentes pondérées délivrées aux différents tissus et organes du corps. La pondération correspond à l'application d'un facteur de pondération tissulaire (W_T) à la dose équivalente pour chaque organe ou tissu. La dose efficace correspond à l'évaluation d'une dose corps entier. **Exposition interne** Lorsqu'elle est due à une exposition par ingestion ou inhalation, la dose efficace engagée est exprimée par unité d'incorporation (DPUI) en Sv/Bq. Pour un radionucléide inhalé ou ingéré, cette DPUI est à multiplier par l'activité (en Bq) absorbée. Lorsque l'exposition est à la fois externe et interne, la dose efficace est la somme des doses efficaces dues à une exposition externe et interne.

Illustration 14: Principales grandeurs de rayonnements - D'après l'Institut de Radioprotection et de Sûreté Nucléaire (IRSN)

Notre compteur va mesurer le taux d'irradiation en µSv/h, unité fréquemment utilisée pour ces utilisations.

Archivage des mesures

L'archivage des mesures est réalisée à l'aide d'une base de données. Cette dernière archive les mesures de température et d'irradiation, selon une période choisie.

La base de données

Le choix de la base de données s'est porté sur une BDD SQLite.

SQLite est un moteur de base de données relationnelle, écrite en C.

Cela en fait un très bon candidat pour les systèmes embarqués. Ainsi, il est le moteur de base de données la plus utilisée dans le monde, intégrant des logiciels comme Firefox, Skype, etc.

Contrairement aux serveurs de bases de données traditionnels, comme MySQL ou PostgreSQL, sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégrée aux programmes. L'intégralité de la base de données (déclarations, tables, index et données) est stockée dans un fichier indépendant de la plateforme.

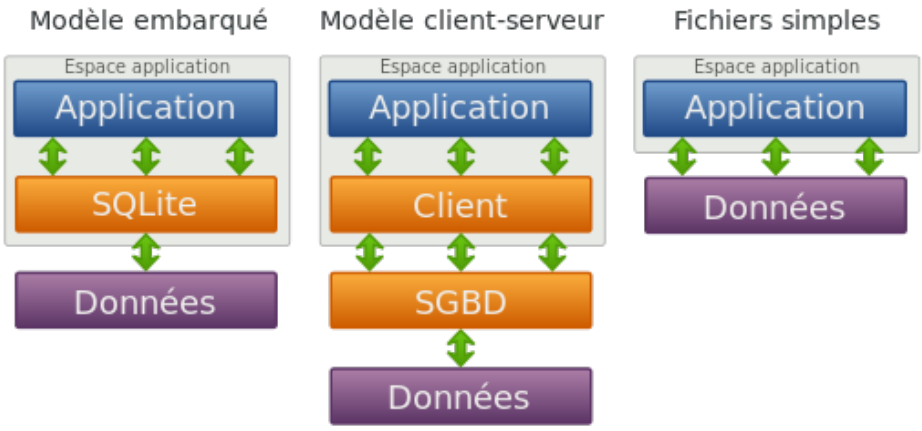


Illustration 15: Les différents modèles de gestion de base de données

Structure de la BDD

Notre base de données est composée de 3 tables, dont voici la structure :

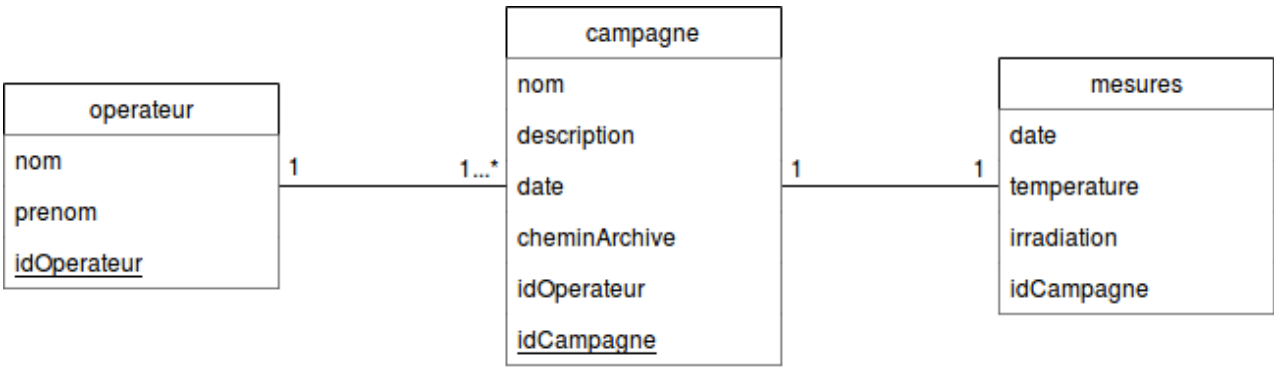


Illustration 16: Structure de la base de données

Table	Description
operateur	Opérateurs ayant accès au déploiement du rov. Ces derniers sont caractérisés par leur nom et prénom. L'opérateur (ou technicien) créera une nouvelle campagne.
campagne	Campagnes réalisées. Elles sont caractérisées par l'id de l'opérateur ayant réalisé la campagne, le nom et une description de la mission, la date de cette dernière et enfin le chemin d'accès à ses archives.
mesures	Archives des mesures. Ces dernières sont identifiables par l'id de la campagne à laquelle elles sont liées, la date de réception de la mesure, ainsi que les deux mesures enregistrées : la température et l'irradiation.

La table mesures

Les mesures sont stockées dans une table nommée mesures. Cette dernière est structurée ainsi :

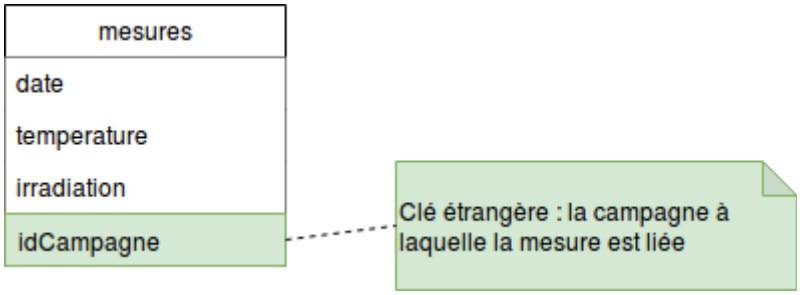


Illustration 17: Structure de la table mesures

date : **DATETIME**
temperature : **DOUBLE**
irradiation : **DOUBLE**
idCampagne : **INT**

L'entrée de nouvelles données de mesures dans la BDD est réalisée par une commande **INSERT**.

Exemple pour l'entrée suivante :

Température	Irradiation	idCampagne
22,4°C	0,02	1

Requête SQL :

```
INSERT INTO mesures
VALUES(datetime('now', 'localtime'), 22.4, 0.02, 1);
```

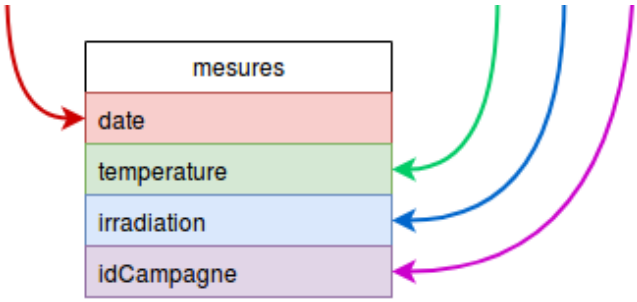


Diagramme de séquence : archiver les mesures

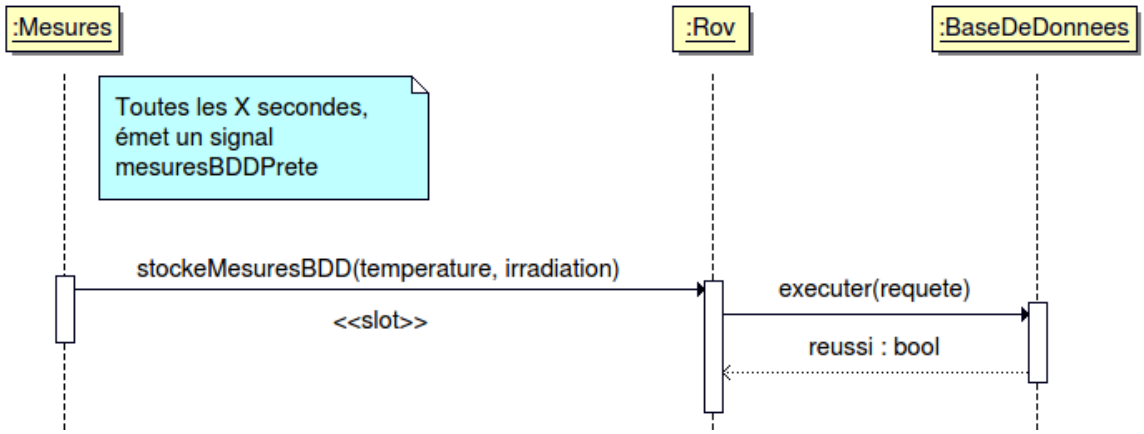


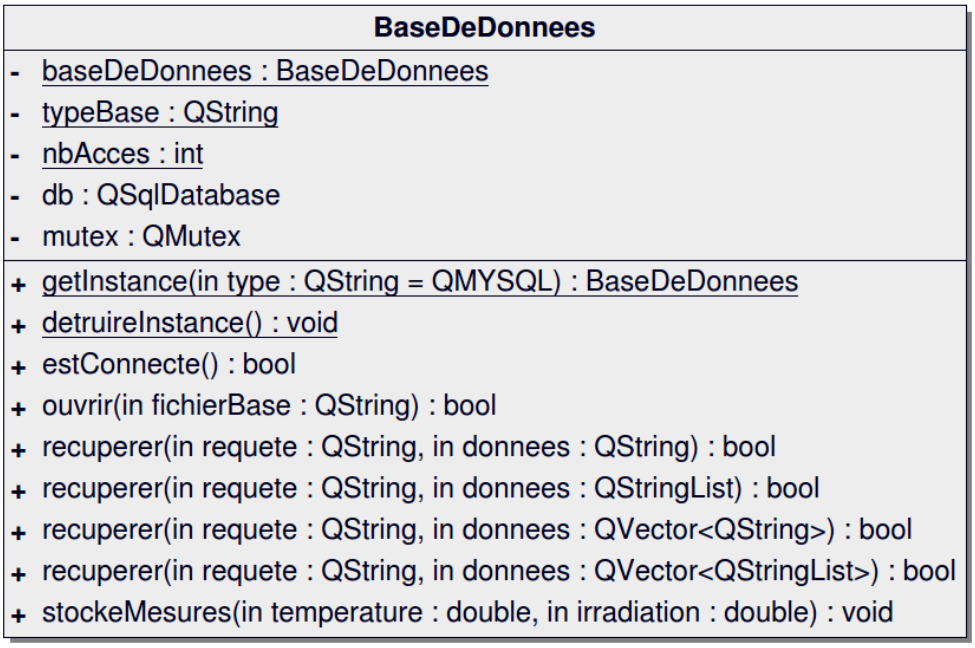
Illustration 18: Diagramme de séquence - Archiver les mesures

L'archivage des mesures se réalise en plusieurs étapes :

1. Un signal nommé *mesuresBDDPrete* est émis toutes les x secondes, contenant la dernière donnée de température et d'irradiation relevé.
2. Le signal active un slot nommé *stockeMesuresBDD*, qui va créer la requête SQL à émettre contenant les données demandées.
3. La requête est émise à l'aide de la méthode *executer* dans la classe *base de données*. Ce dernier va retourner un booléen représentant la réussite ou l'échec de la requête.

Classes et méthodes liées à l'archivage des mesures

La classe BaseDeDonnees

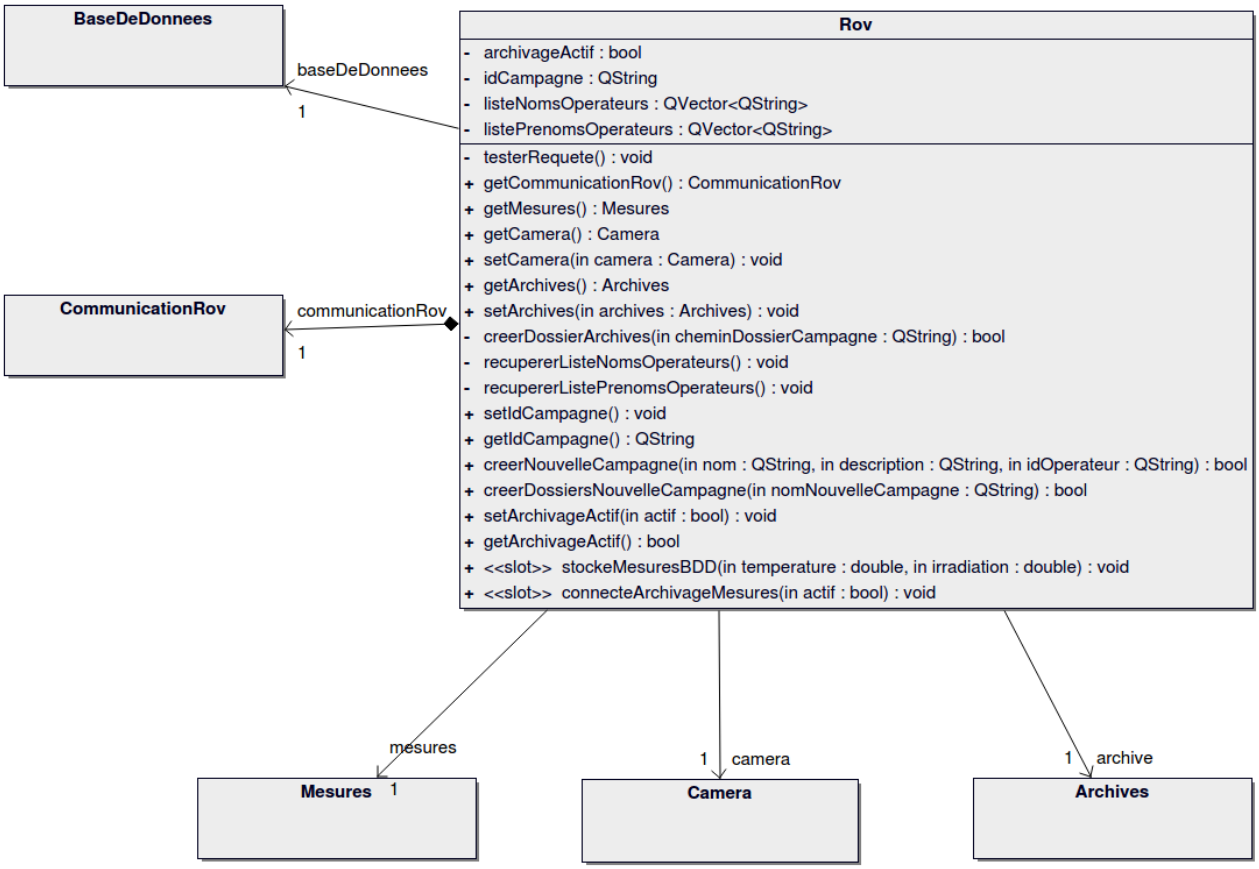


Cette classe permet la gestion d'une base de données SQLite, et met à disposition des méthodes permettant d'interagir avec cette dernière.

Cette classe est un singleton, ce qui permet à l'intégralité des classes du programme, d'avoir une seule et même instance de cette dernière.

L'instance est récupérée en appelant la méthode statique *getInstance()*. Ensuite, il faut ouvrir la base de données SQLite en utilisant la méthode *ouvrir()* en lui passant en paramètre le nom de la base de données : **rovnet.sqlite**. Les requêtes SQL SELECT seront réalisées par les méthodes *recuperer()*. Pour les requêtes SQL INSERT, UPDATE et DELETE, on utilisera la méthode *excuter()*.

La classe Rov



La classe Rov est la plus proche de la base de données. Les requêtes à transmettre seront donc émises depuis cette classe.

Cette dernière contient aussi des attributs en lien avec la mission actuelle, tel que le numéro d'identifiant de la campagne.

Attributs (utilisés pour l'archivage des mesures)

- archivageActif : bool.
  - Indique si l'archivage des mesures est actif ou non.

- idCampagne : QString.
  - Numéro d'identification de la campagne en cours.

Méthodes (utilisées pour l'archivage des mesures)

- getArchivageActif : QString.

```
bool Rov::getArchivageActif() const
{
    return this->archivageActif;
}
```

  - Retourne l'attribut archivageActif.

- setArchivageActif : void.

```
void Rov::setArchivageActif(bool archivageActif)
{
    this->archivageActif = archivageActif;
    connecteArchivageMesures(archivageActif);
}
```

  - Actualise l'attribut archivageActif avec le nouvel état.
  - Appelle la fonction connecteArchivageMesures en lui indiquant le nouvel état.

- connecteArchivageMesures : void.

```
void Rov::connecteArchivageMesures(bool archivageActif)
{
    if(archivageActif)
        connect(mesures, SIGNAL(mesuresBDDPrete(double, double)), this,
SLOT(stockeMesuresBDD(double, double)));
    else
        disconnect(mesures, SIGNAL(mesuresBDDPrete(double, double)), this,
SLOT(stockeMesuresBDD(double, double)));
}
```

  - Connecte ou déconnecte le signal permettant d'archiver les mesures :
    - Si l'archivage est demandé : connecte le signal *mesuresBDDPrete* de la classe *Mesures* au slot *stockeMesuresBDD*,
    - Sinon, le déconnecte.

- stockeMesuresBDD : void.

```
void Rov::stockeMesuresBDD(double temperature, double irradiation)
{
    QString requete = "INSERT INTO mesures VALUES(datetime('now', 'localtime'),
" + QString::number(temperature) + ", " + QString::number(irradiation) + ", " +
idCampagne + ");";
    bool reussi = baseDeDonnees->executer(requete);
    qDebug() << requete;

    if (!reussi)
        qDebug() << Q_FUNC_INFO << "ERREUR ! Echec de l'envoi des mesures dans
la BDD !";
}
```

  - Archive les mesures données en argument dans la base de données.
    - Envoie une requête SQL à destination de la table mesures. Cette requête est décrite page 49.
      - Si la requête échoue, envoie un message d'erreur.

Paramétrage d’une campagne

Présentation

Chaque nouvelle campagne possède des besoins différents. Il est alors utile de pouvoir paramétrer la campagne.

Nous avons alors 3 paramètres disponibles :

- L’activation, ou non, de l’archivage des mesures dans une BDD,
- Le choix de la fréquence d’archivage des mesures dans la BDD (si la première option est sélectionnée),
- Le choix du port de communication utilisé.

Fenêtre Paramètres

Le paramétrage de la mission se réalise via la fenêtre suivante :

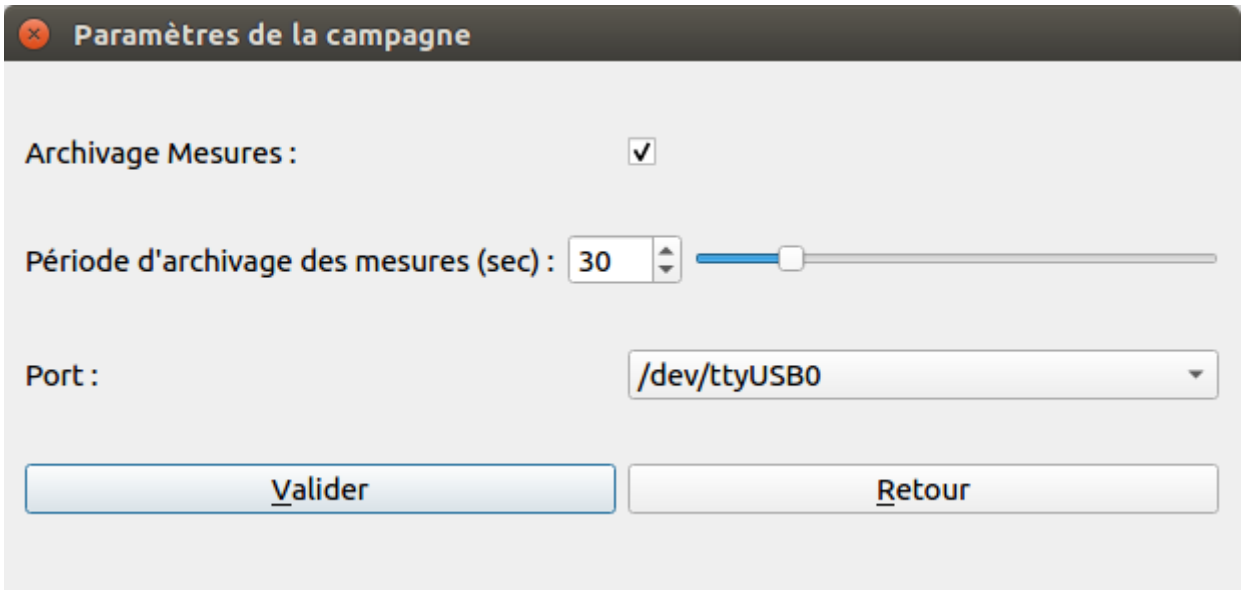


Illustration 19: IHM - Fenêtre Paramètres

Lorsque la case « Archivage Mesures » est décochée, le paramétrage « Période d'archivage des mesures » se grise et n'est plus disponible.

La période d'archivage des mesures minimum est de 12. Cela correspond à la période de réception des mesures des capteurs de température et d'irradiation, qui s'actualisent toutes les 12 secondes.

Diagramme de séquence : enregistrer les paramètres

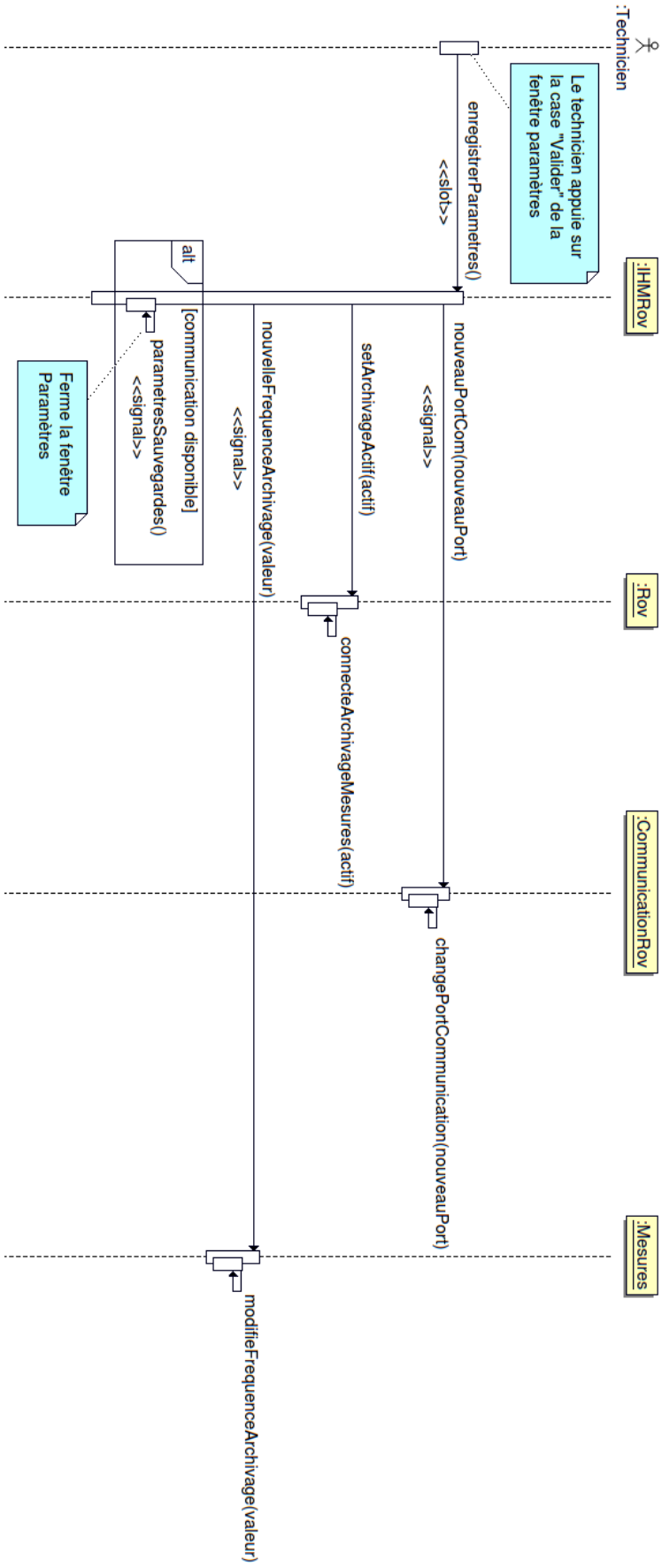


Illustration 20: Diagramme de séquence - Enregistrer les paramètres

Lorsque le technicien appuie sur le bouton « Valider » après avoir réglé les paramètres, la séquence débute :

Le port de communication sélectionné est envoyé par un signal vers la classe *CommunicationRov*, qui va changer la communication en conséquence.

1. L'état d'activation des archives est mis à jour dans la classe *Rov*, qui va changer les connexions en conséquence.

2. La nouvelle fréquence d'archivage est envoyé par un signal vers la classe *Mesures*, qui va changer l'intervalle de son compteur en conséquence.

3. Enfin, si la connexion avec le rov est fonctionnelle, un signal est émis indiquant le succès et va fermer la fenêtre de paramètres. Sinon, un message d'erreur est envoyé et la fenêtre ne se ferme pas.

Tests de validation

Désignation	Procédure	Résultat attendu	Fonctionnel	Remarques
La manette est connectée	Vérifier réception des signaux	Les signaux sont reçus	Oui	-
Le changement de mode de la manette est fonctionnel	Appuyer sur la touche Select de la manette	La manette passe bien d'un mode à l'autre	Oui	-
Les trames de déplacement du rov sont envoyées au rov	- Avancer (joystick avant) - Reculer (joystick arrière) - Tourner à droite (joystick droite) - Tourner à gauche (joystick gauche)	Les trames sont bien créées et envoyées par liaison série	Oui	Nécessite des ajustements, pour optimiser les déplacements
Les trames de pilotage du bras sont envoyées au rov	Utiliser les touches de la manette : - Joystick gauche - Joystick droit - Croix directionnelle - R1 - L1 - Start	Les trames sont bien créées et envoyées par liaison série	Oui	Nécessite des ajustements, pour optimiser la manipulation du bras
Les mesures des capteurs apparaissent sur l'IHM	La température, l'irradiation et la distance sont affichées	Les données sont affichées en °C, en µSv/h, et en cm	Oui	Les tests ont été réalisés sur simulateur uniquement
La com en liaison série est fonctionnelle	Vérifier que le programme est bien connecté au rov	La com est établie	Oui	-
Archiver les données	Vérifier le contenu de la base de données	Les données sont stockées dans la BDD	Oui	-

Partie personnelle – Nicolas Boffredo

Sommaire

Objectifs

Afin de savoir dans quel milieu va naviguer le Rov, il est nécessaire d'avoir et de gérer un retour vidéo de cet environnement. D'autre part, il sera possible de prendre des photos de l'environnement pendant une campagne.

Ceci permettra de définir l'environnement TQC (« Tel Que Construit »).

Mon travail consiste donc à assurer :

- Le gestion de la caméra :
  - Gérer le fonctionnement de la caméra,
  - Réceptionner le flux vidéo,
  - Afficher le flux vidéo.
- La gestions des captures :
  - Rendre possible la prise de capture,
  - Formater les captures,
  - Sauvegarder les captures,
  - Visualiser ces captures.
- Le pilotage de la caméra
  - Gérer le fonctionnement de la manette
  - Associer la caméra et la manette (lié les mouvements de la caméra à des boutons sur la manette)

Ces fonctionnalités permettront à l'utilisateur de *Visualiser l'environnement*, de *Prendre une photo* et de *Piloter la caméra*, si les données télémétriques sont bien réceptionnées.

Diagramme des cas d'utilisation

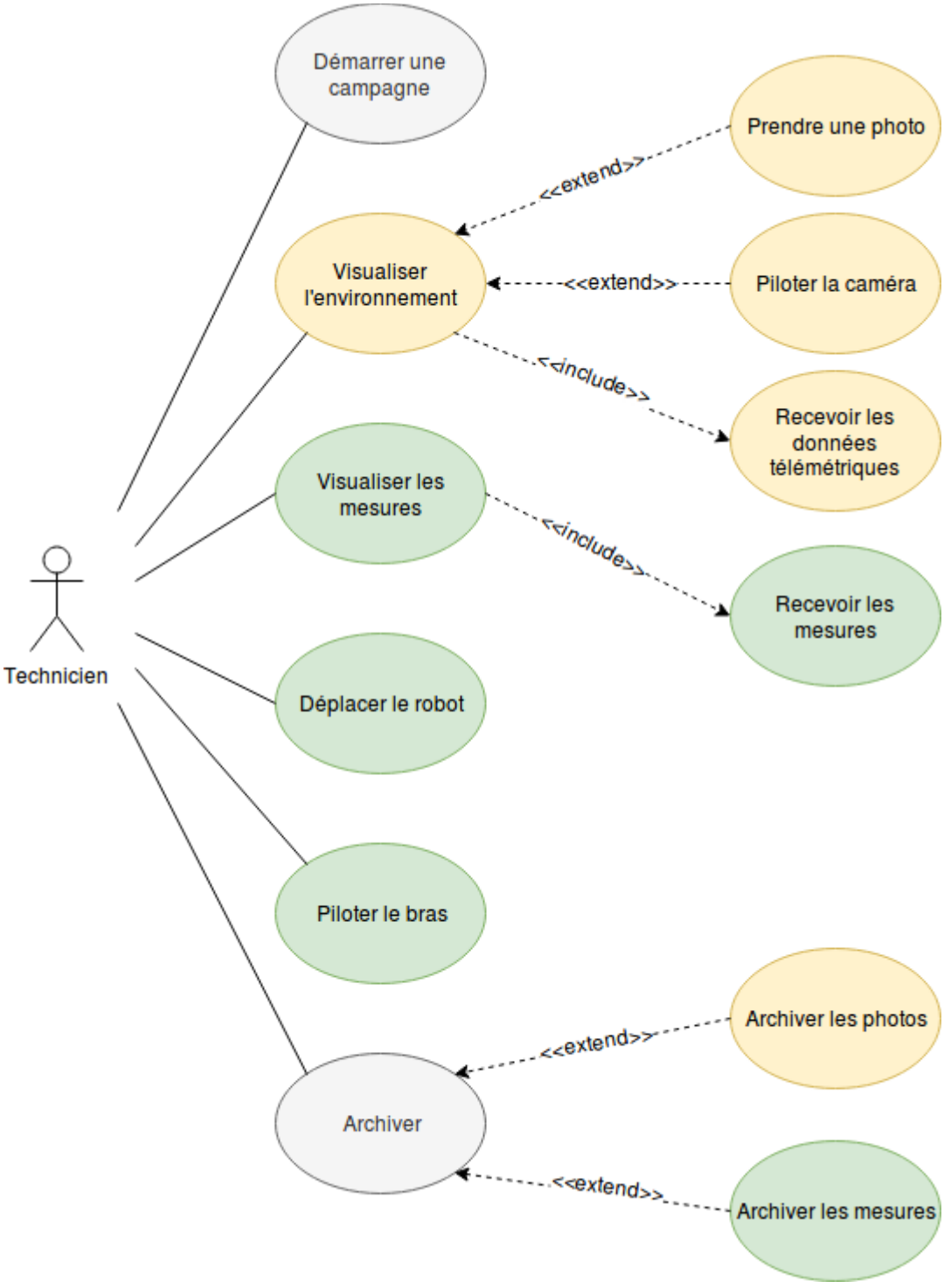
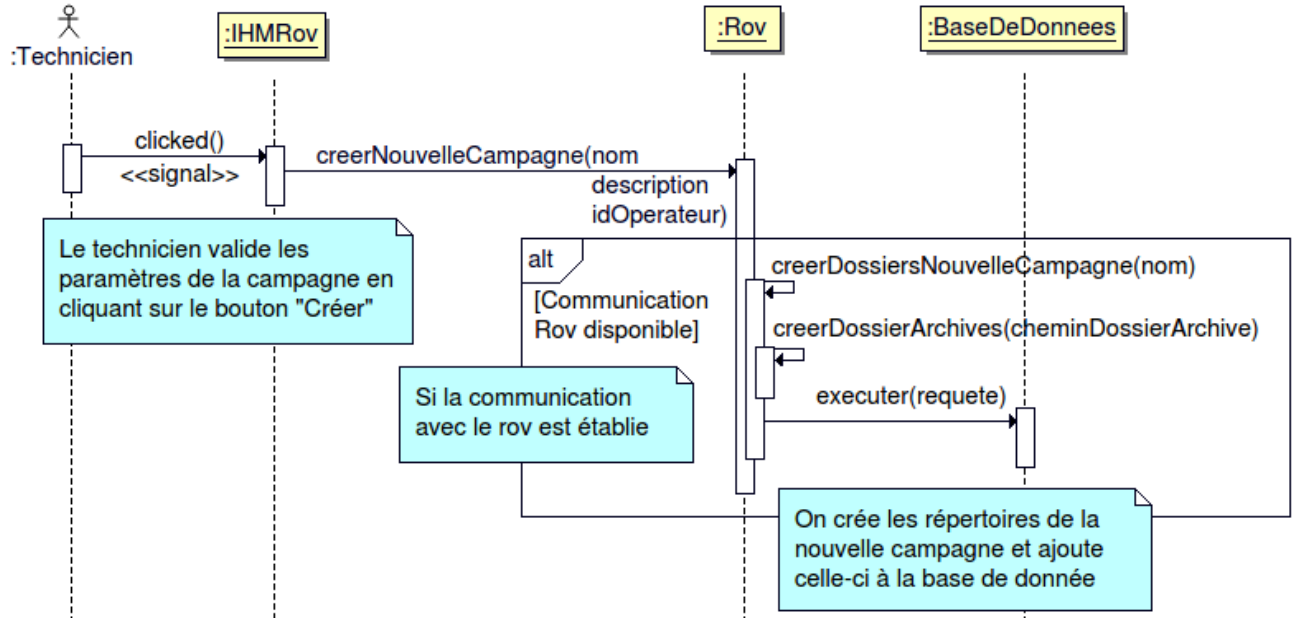


Diagramme des cas d'utilisation : représentation des fonctions offertes par le système, produisant un résultat observable et intéressant pour un acteur (ici le Technicien).

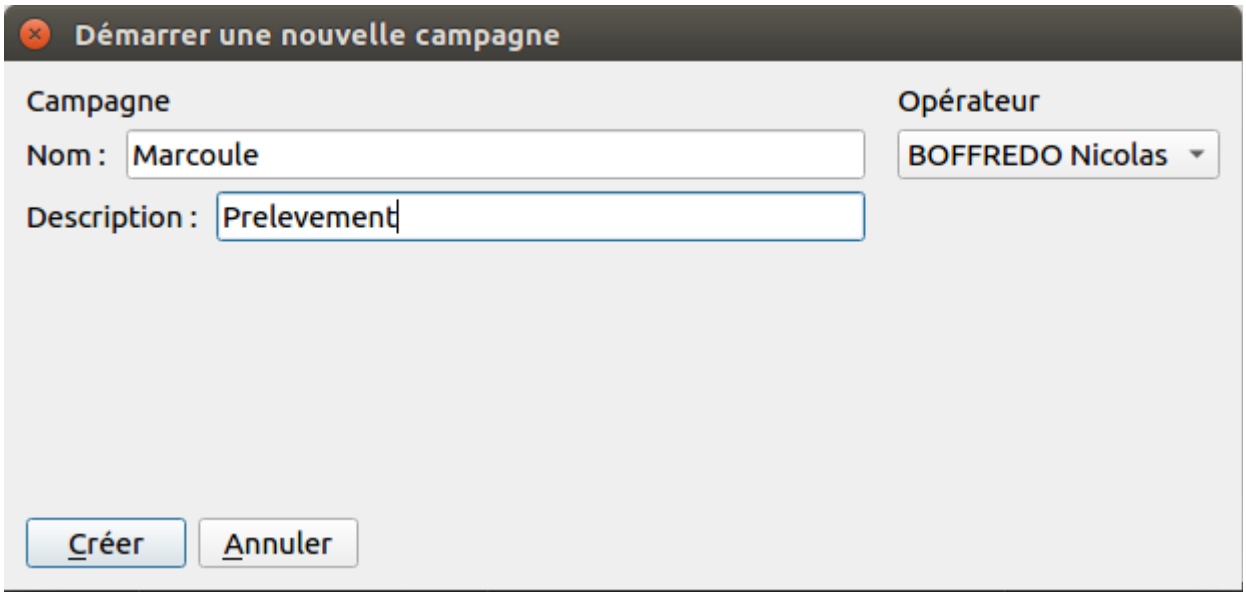
Démarrer une nouvelle campagne

L'objectif principal du Rov est de pouvoir effectuer une campagne.

Lors de l'ouverture de l'application, la création d'une nouvelle campagne est la première chose que pourra réaliser le technicien.



Sur l'IHM (Interface Homme-Machine) :



```
bool Rov::creerNouvelleCampagne(QString nom, QString description, QString
idOperateur)
{
    if(communicationRov->estCommunicationRovDisponible())
    {
        if(creerDossiersNouvelleCampagne(nom))
        {
            QString cheminArchives = archives->getCheminArchives();
            QString requete = "INSERT INTO 'campagnes'(nom, description, date,
cheminArchives, idOperateur) VALUES ('" + nom + "', '" + description + "',
datetime('now', 'localtime'), '" + cheminArchives + "', '" + idOperateur + "')";
            bool retour = baseDeDonnees->executer(requete);
            if(!retour)
            {
                return false;
            }

            QString idCampagne;
            bool requeteRecupIdCampagne = baseDeDonnees->recuperer("SELECT
idCampagne FROM campagnes WHERE cheminArchives = '" + cheminArchives + "';",
idCampagne);
            if(requeteRecupIdCampagne)
            {
                setIdCampagne(idCampagne);
            }
            else
            {
                return false;
            }
            return true;
        }
        else
            return false;
    }
    else
        return false;
}
```

Lors de son appel, si la communication avec le rov est établie :

- On crée le répertoire qui va stocker toutes les données de la campagne avec le nom entré dans la fenêtre de création d'une nouvelle campagne avec la méthode creerDossiersNouvelleCampagne(nomNouvelleCampagne).

```
bool Rov::creerDossiersNouvelleCampagne(QString nomNouvelleCampagne)
{
    QDir dossierApplication(QApplication::applicationDirPath());
    if(dossierApplication.mkdir(nomNouvelleCampagne))
    {
        QString cheminDossierCampagne = QApplication::applicationDirPath() + "/"
+ nomNouvelleCampagne;
        qDebug() << Q_FUNC_INFO << cheminDossierCampagne;
        return creerDossierArchives(cheminDossierCampagne);
    }
    return false;
}
```

- on crée le répertoire des archives dans ce même dossier avec la méthode creerDossierArchives(cheminDossierCampagne). Celui ci se nommera toujours Archives.

```
bool Rov::creerDossierArchives(QString cheminDossierCampagne)
{
    qDebug() << Q_FUNC_INFO;
    QDir dossierArchives(cheminDossierCampagne);
    if(dossierArchives.mkdir("Archives"))
    {
        QString cheminDossierArchives = cheminDossierCampagne + "/Archives";
        qDebug() << Q_FUNC_INFO << archives << cheminDossierArchives;
        archives->setCheminArchives(cheminDossierArchives);
        return true;
    }
    return false;
}
```

- Si les répertoires ont bien été créés, alors on peut enregistrer la nouvelle campagne dans la base de donnée avec la requête SQL[1] :

  INSERT INTO 'campagnes'(nom, description, date, cheminArchives, idOperateur) VALUES ('" + nom + "', '" + description + "', datetime('now', 'localtime'), '"

On ajoute un tuple[2] à la table campagne, en indiquant : son nom, sa description, la date de création et le technicien intervenant sur cette campagne.

1 SQL : *Structured Query Language*, langage permettant d'ajouter, modifier ou supprimer des données dans les bases de données.

2 Tuple : enregistrement ou ligne d'une table dans une base de données.

La caméra

Caractéristiques techniques :

- Liaison : USB
- Format vidéo : MJPEG
- Résolution :
  - 2592x1944 à 15fps, 5M pixels
  - 1280x720 à 30fps, 1M pixels
- Mouvement du support : Panoramique[3]

Mise en œuvre

Pour vérifier si la caméra est reconnue :

```
$ ls /dev/ | grep video
video0
```

Cette commande retourne la liste des fichiers de périphériques comprenant le nom « video ».

Remarque : Le nom de chaque caméra dépend de l'ordre dans lequel elles ont été branchées (ex : Caméra 1 = video0 ; Caméra 2 = video1 etc..)

Linux a besoin d'un paquet[4] de prise en charge vidéo : **v4l-utils** *(Video4Linux)*

- Pour vérifier sa présence :

```
$ dpkg -l | grep v4l


ii  v4l-utils                                  1.10.0-1
amd64        Collection of command line video4linux utilities
```

3 Mouvement Panoramique : mouvement horizontal.

4 paquet : archive de fichiers informatiques contenant les informations et les procédures nécessaires à une installation.

- S'il est absent, il faut l'installer :

```
$ sudo apt-get install v4l-utils
```

Le module Qt utilisé nécessite également l'installation d'un autre paquet (**libpulse-dev**)

- Pour vérifier sa présence :

```
$ dpkg -l | grep libpulse


ii  libpulse-dev:amd64                          1:8.0-0ubuntu3.10
amd64        PulseAudio client development headers and libraries
```

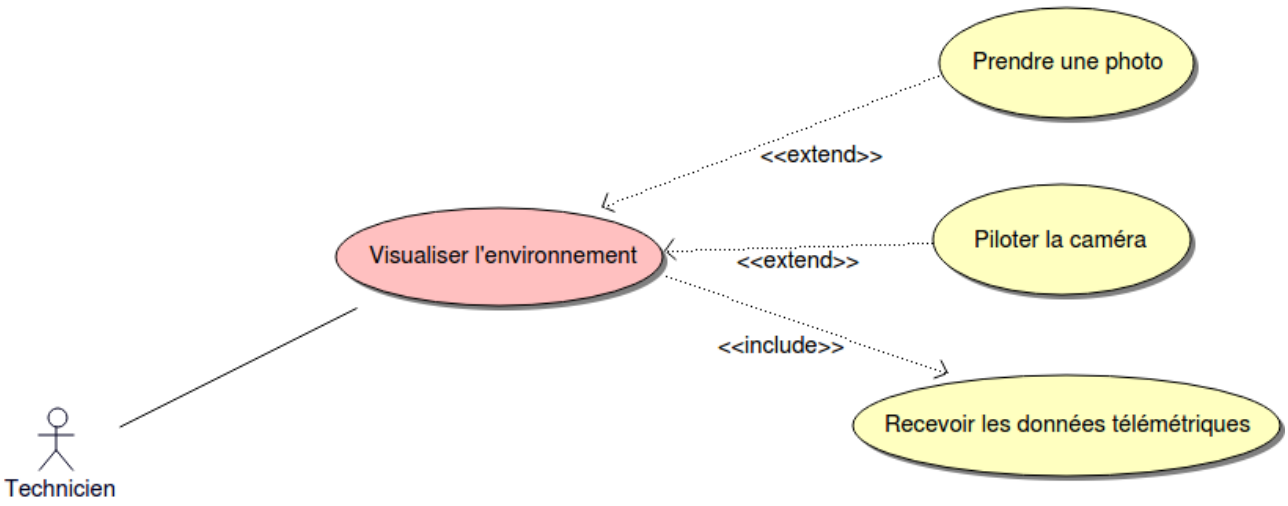
- S'il est absent, il faut l'installer :

```
$ sudo apt-get install libpulse-dev
```

La classe Camera

Camera
- camera : QCamera - cadreFluxVideo : QCameraViewFinder - captureImage : QCameraImageCapture - formatSauvegardeTemps : QString - dateImage : QString - etatCamera : QString - <<QList>> cameras : QCameraInfo
- nommerCapture() : QString - demarrerCamera(in cameraSelectionnee : QCameraInfo) : void + estCameraDisponible() : bool + getCadreFluxVideo() : QCameraViewFinder + getEtatCamera() : QString + envoieListeCamera() : void

La classe *Camera* est utilisée dans le cas d'utilisation "Visualiser l'environnement"



Afin de visualiser l'environnement, il faut s'assurer du fonctionnement de la caméra. La méthode **estCameraDisponible()** a été créée dans ce but.

La méthode compte le nombre de caméras disponibles, et retourne un booléen à l'état "vrai" si au moins une caméra a été détectée.

```
bool Camera::estCameraDisponible()
{
    if (cameras.count() > 0)
        return true;

    else
        return false;
}
```

Une fois qu'une caméra est disponible, il devient possible de visualiser le retour vidéo en la démarrant avec la méthode **demarrerCamera(cameraSelectionnee)**.

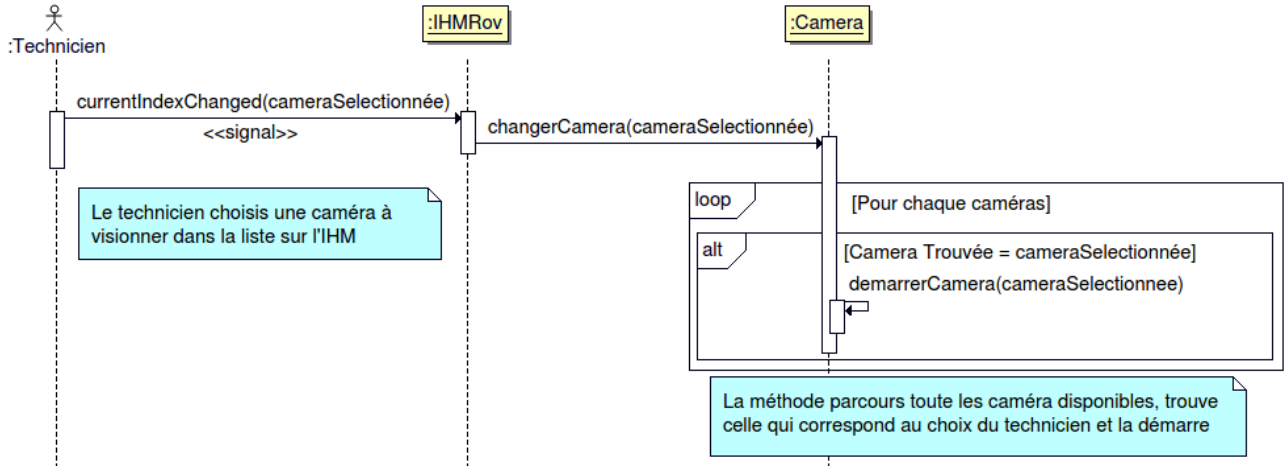
Si au moins une caméra est disponible :

- On instancie un objet correspondant à cette caméra
- On crée le cadre où afficher le retour vidéo sur l'IHM

```
$ ls /dev/ | grep video
video0
video1
```

video0 et *video1* correspondent à deux caméras présentes.

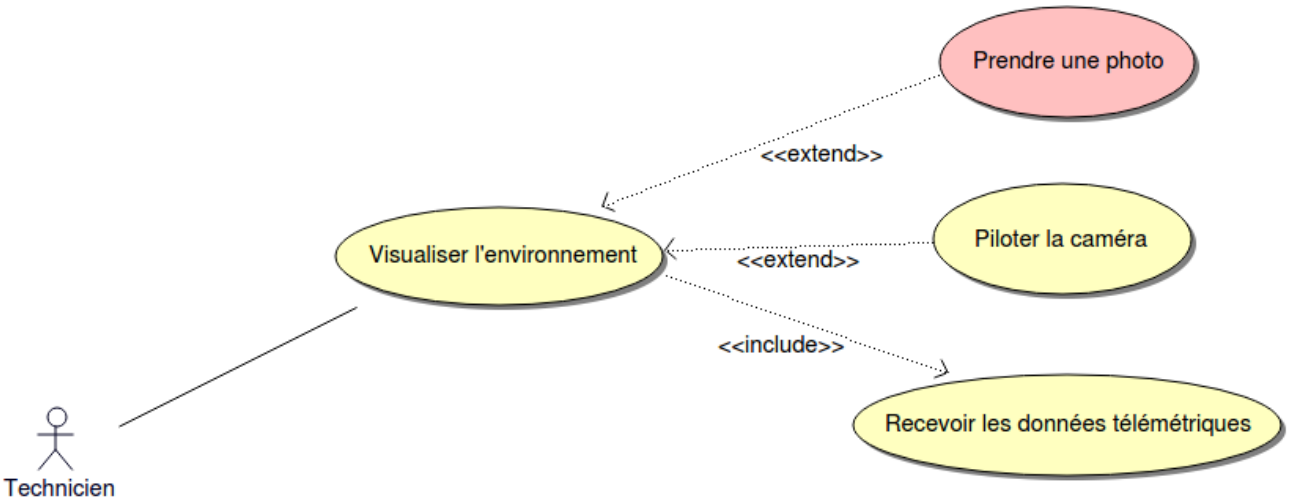
Il est possible de basculer de l'une à l'autre depuis l'IHM :



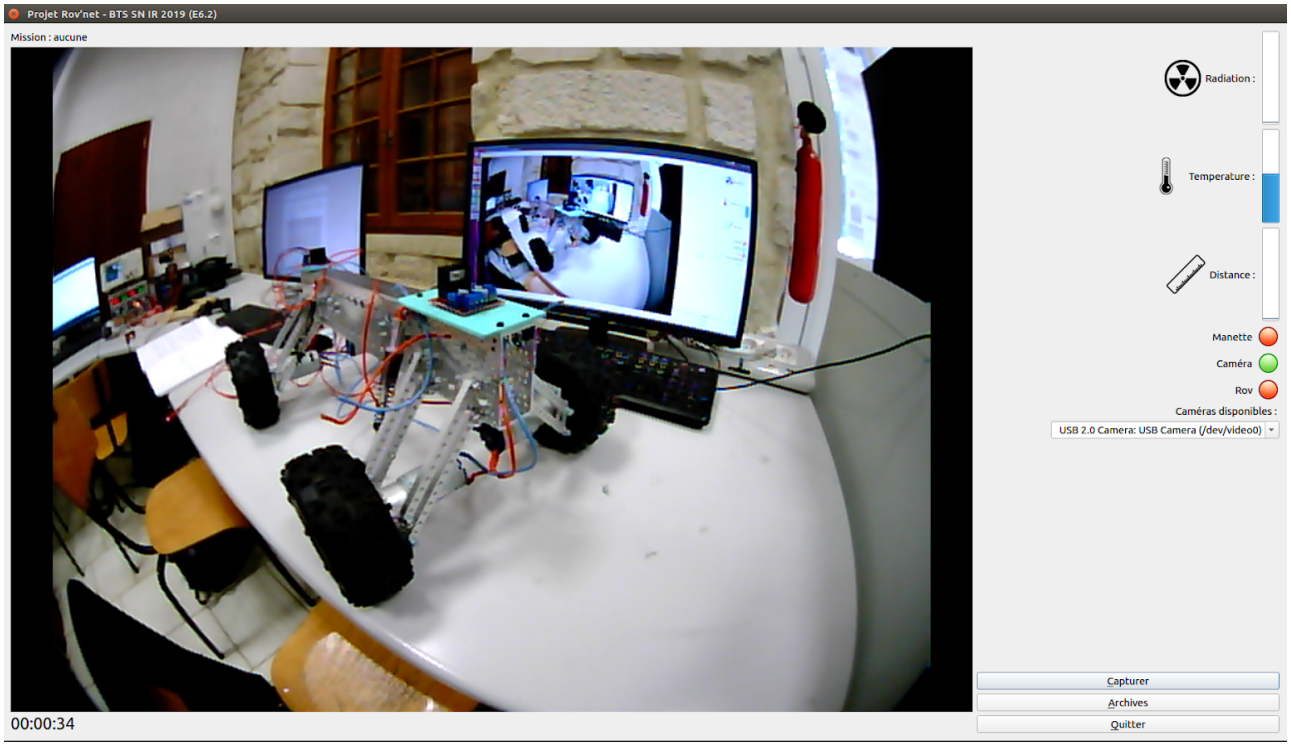
Une fois la caméra disponible, il devient possible de prendre **des captures** du flux vidéo.

Gestion des captures

Dans le diagramme des cas d'utilisation, le technicien peut « Prendre une photo » lorsqu'il « Visualise l'environnement ».

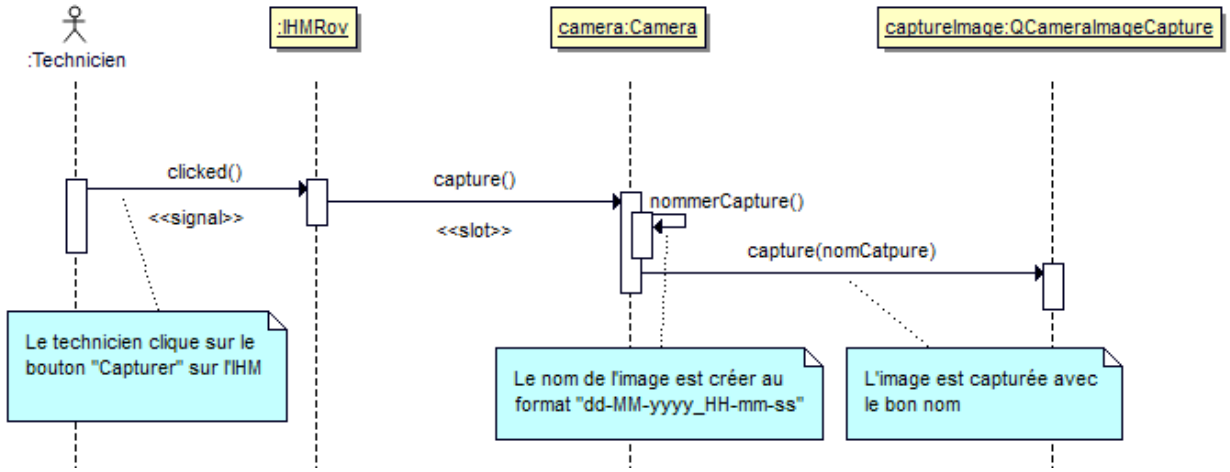


Il est possible de prendre une photo depuis l'IHM grâce au bouton **Capturer**.



Prendre une photo

Ce qui correspond au diagramme de séquence[5] :



Lorsque l'on clique sur le bouton Capturer, un signal[6] clicked() est émis et déclenche l'exécution du slot[7] capture().

Le slot capture() de la classe Camera permet de prendre une capture d'écran :

```
void Camera::capture()
{
    qDebug() << Q_FUNC_INFO;
    QString nomCapture = this->nommerCapture();
    qDebug() << Q_FUNC_INFO << "nomCapture" << nomCapture;
    captureImage->capture(nomCapture);
}
```

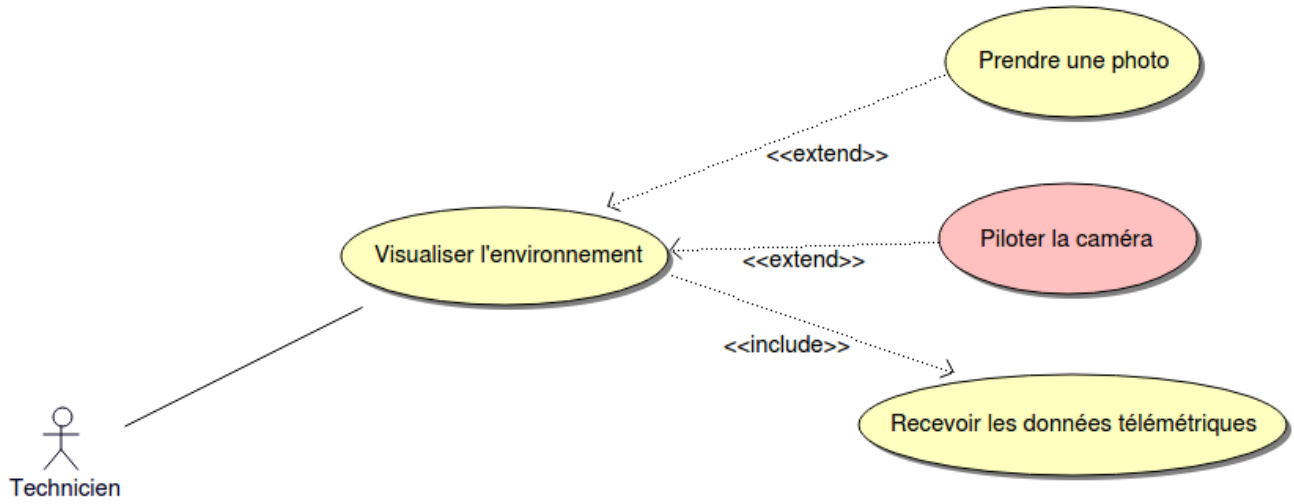
La capture est horodatée[8] et est stocké dans l'archive de la campagne en cours.

5 Diagramme de Séquence : diagramme UML dont le but est de décrire comment les objets interagissent entre eux au cours du temps.
6 signal : émis lorsqu'un événement se produit (ici lorsqu'on clique sur un bouton).
7 slot : fonction ou méthode appelée en réponse à un signal.
8 Horodatée au format : YYYY-MM-DD_hh-mm-ss.jpg

La classe ControleCamera

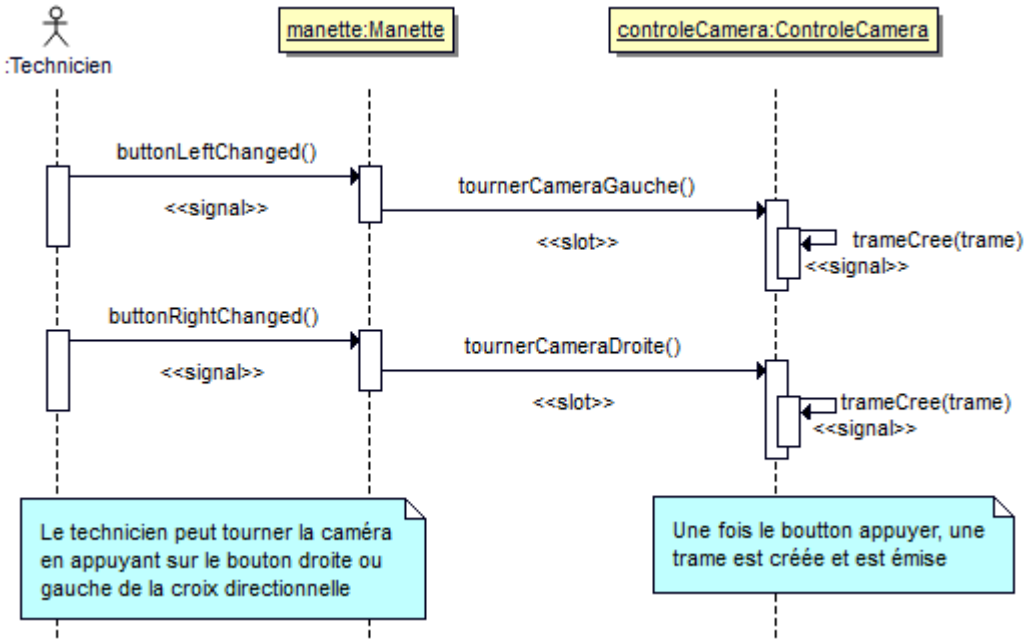
ControleCamera
- idTrame : unsigned int
+ tourneCameraGauche(in appuye : bool) : void + tourneCameraDroite(in appuye : bool) : void

Il a été demandé que la caméra puisse être pilotée sur un angle panoramique.



Piloter la caméra

De ce cas d'utilisation, on a le diagramme de séquence suivant :



Les deux méthodes (*tournerCameraGauche()* et *tournerCameraDroite()*) sont pratiquement identique :

```
void ControleCamera::tourneCameraGauche(bool boutonAppuye)
{
    QString trame = "$TCA" + QString::number(-boutonAppuye) + "\n";
    emit trameCree(trame);
}
```

```
void ControleCamera::tourneCameraDroite(bool boutonAppuye)
{
    QString trame = "$TCA" + QString::number(boutonAppuye) + "\n";
    emit trameCree(trame);
}
```

Les deux méthodes fabriquent et émettent une trame correspondant au protocole défini.

Protocole de communication

Format de trame

```
début code valeur fin de trame
```

- Début de trame : **$**
- Champ **code** : taille fixe :
  - 1 caractère pour les trames de réception,
  - 3 caractères pour les trames d'envoi.
- Champ **valeur** : taille variable dépendant de sa valeur.
- Fin de trame : **\n**

  Le type de trame nous intéressant ici, est la trame d'envoi composée de 3 caractères :
  - Le premier correspond à une action
  - Les deux derniers la partie ciblée

Nous avons donc un total de **10 actions** disponibles, pour **7 parties** contrôlables.

Action	Caractère
Avancer	A
Reculer	R
Tourner à droite	D
Tourner à gauche	G
Tourner	T
Lever	L
Ouvrir	O
Fermer	F
Poser	P
Attraper	E

Partie	Caractères
Roues	RO
Epaule	EP
Coude	CO
Poignet	PO
Pince	PI
Bras	BR
Camera	CA

Le tableau de définition :

Cible	Type	Code	Valeur	Unité
Roues	Avancer	ARO	0 <-> 3	% Vitesse
	Reculer	RRO	0 <-> 3	% Vitesse
	Tourner à droite	DRO	0 <-> 3	% Vitesse
	Tourner à gauche	GRO	0 <-> 3	% Vitesse
Bras	Epaule	TEP	-1 / 0 / 1	Pas (10°)
		LEP	-1 / 0 / 1	Pas (10°)
	Coude	LCO	-1 / 0 / 1	Pas (10°)
	Poignet	LPO	-1 / 0 / 1	Pas (10°)
		TPO	-1 / 0 / 1	Pas (10°)
	Pince	OPI	0 / 1	Booléen
		FPI	0 / 1	Booléen
	Poser	PBR	0 / 1	Booléen
Caméra	Tourner	TCA	-1 / 0 / 1	Pas (10°)

*Pour faire tourner la caméra à gauche, la trame construite sera donc : $**TCA-1\n***

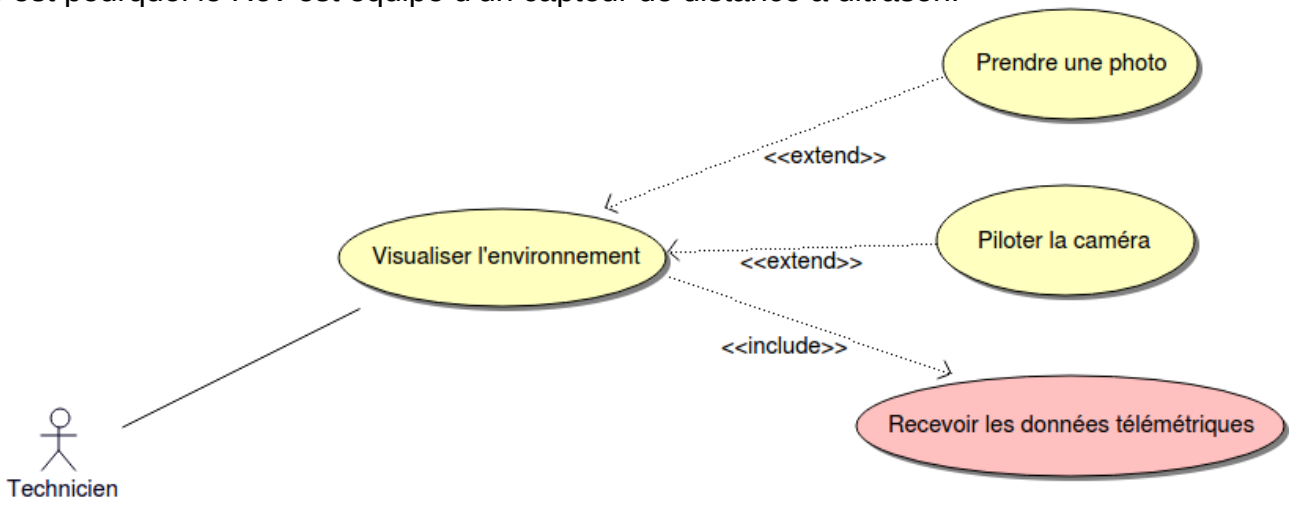
La manette utilisée est une manette PS3. Les touches attribuées au déplacement de la caméra retournent une valeur booléenne.

Nous avons opté pour un déplacement pas à pas avec un pas *de 10*°.

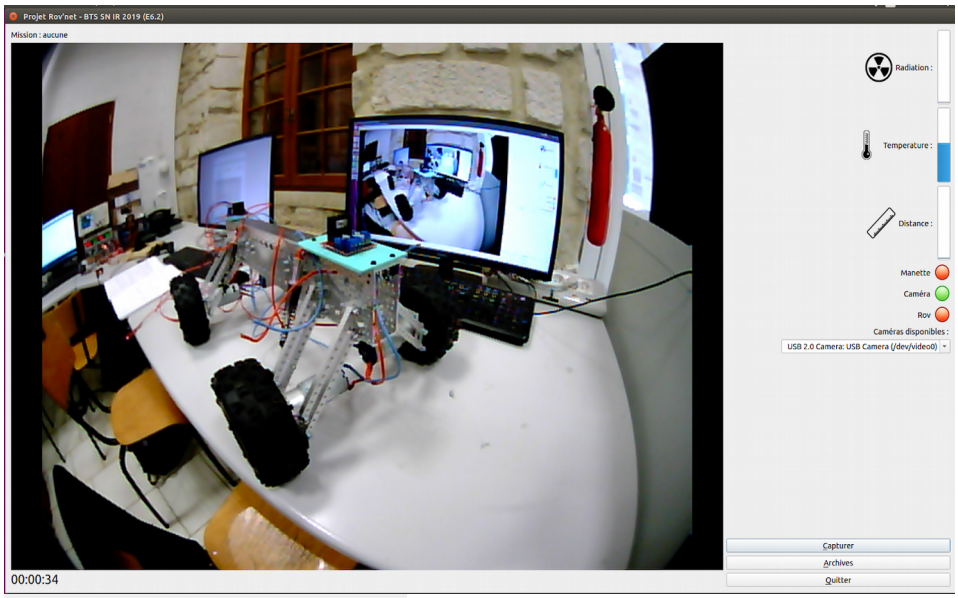
La classe Mesure

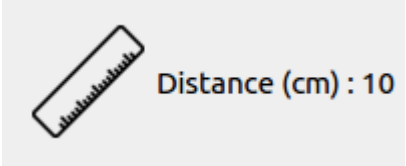
Le technicien ne peut voir qu'à travers la caméra du Rov, ce qui peut être handicapant pour évaluer les distances.

C'est pourquoi le Rov est équipé d'un capteur de distance à ultrason.



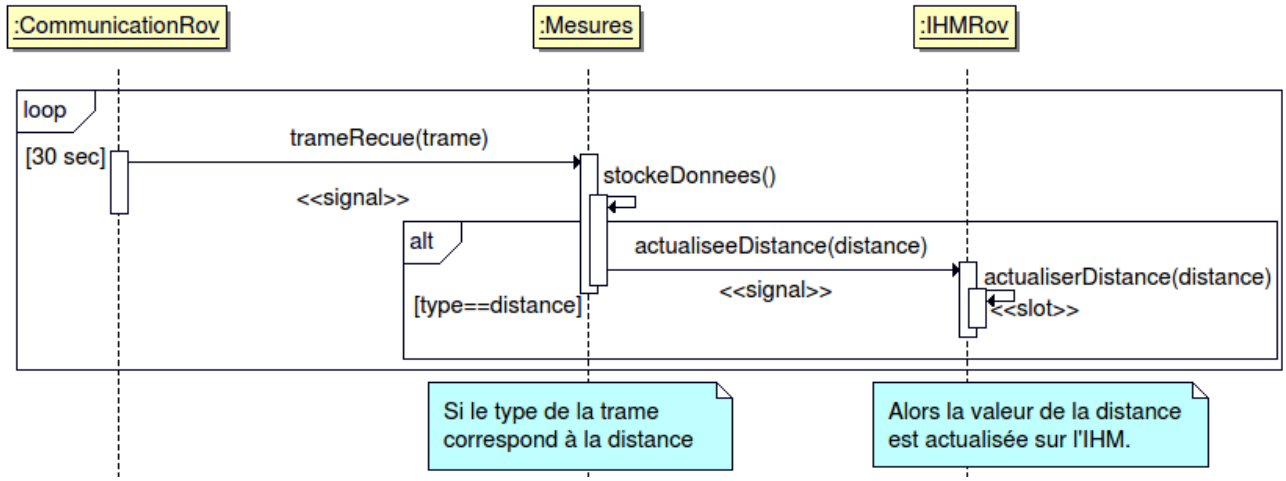
Cette mesure peut être visible sur l'IHM et est reconnaissable de par son logo et son nom.





Recevoir les données télémétriques

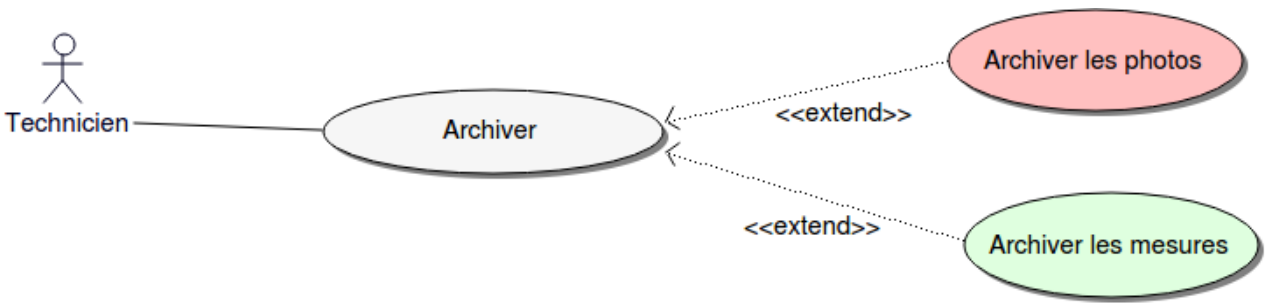
Ce qui correspond au diagramme de séquence suivant :



L’archivage

La classe **Archives** s’occupe de l’archivage des photos prisent par le technicien.

Archives
- cheminDossierArchives : QString - modeleArchives : QFileSystemModel - indexArchives : QModelIndex - fenetreArchives : QDialog - fenetreImages : QDialog - dossierArchives : QDir - vueArchives : QListView - estFenetreArchivesOuverte : bool - labelImage : QLabel - labelImageDate : QLabel - labelImageHeure : QLabel - labelImageRadiation : QLabel - labelImageTemperature : QLabel - boutonFermerArchives : QPushButton
- initialiserFenetreArchives() : void - verifierDossierArchives() : void + getImage(in indexArchives : QModelIndex) : QString + getDateImage(in indexArchives : QModelIndex) : QString + getHeureImage(in indexArchives : QModelIndex) : QString + getCheminArchives() : QString + getModeleArchives() : QFileSystemModel + getIndexArchives() : QModelIndex + fermerArchives() : void + ouvrirFenetreArchives() : void + afficherImage(in indexArchives : QModelIndex) : void

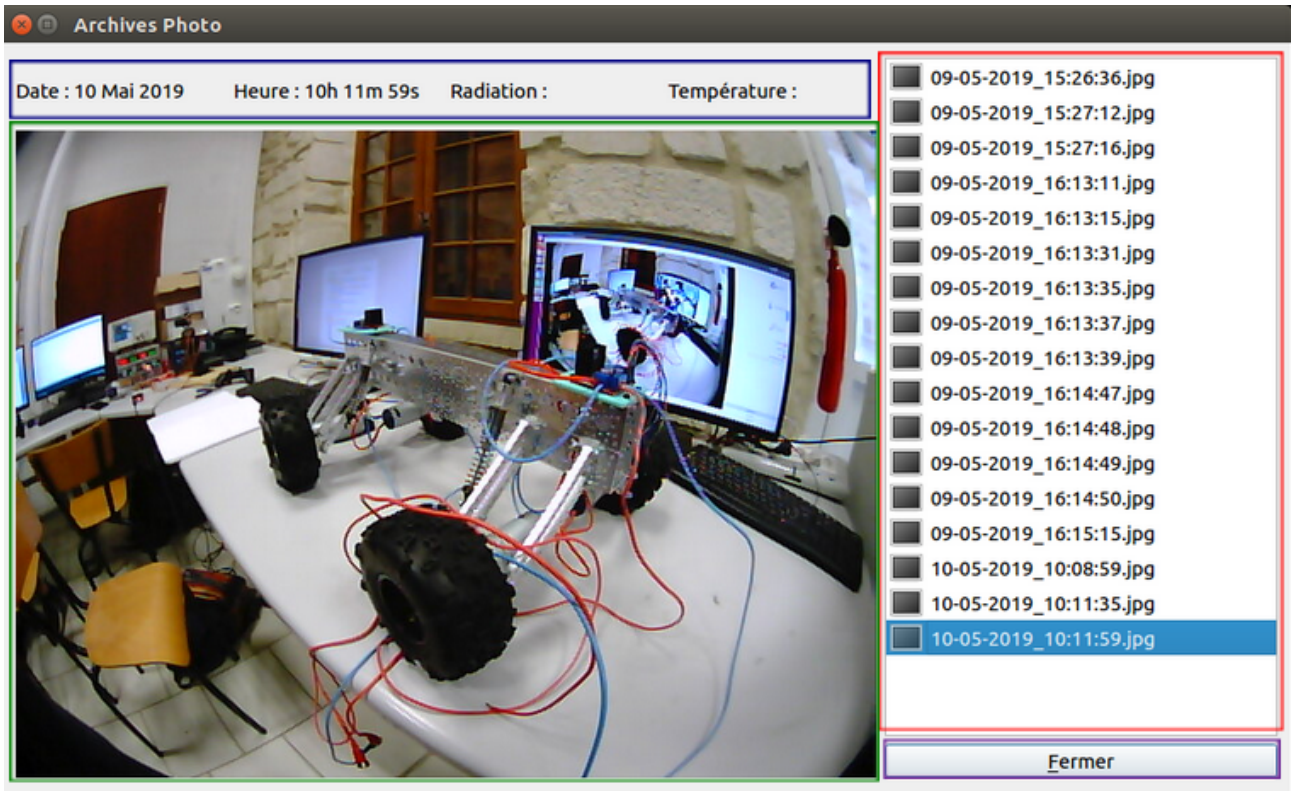


On peut y naviguer directement depuis l'IHM :

Le bouton « Archiver » nous ouvre la fenêtre des archives *(voir ci-dessous)*

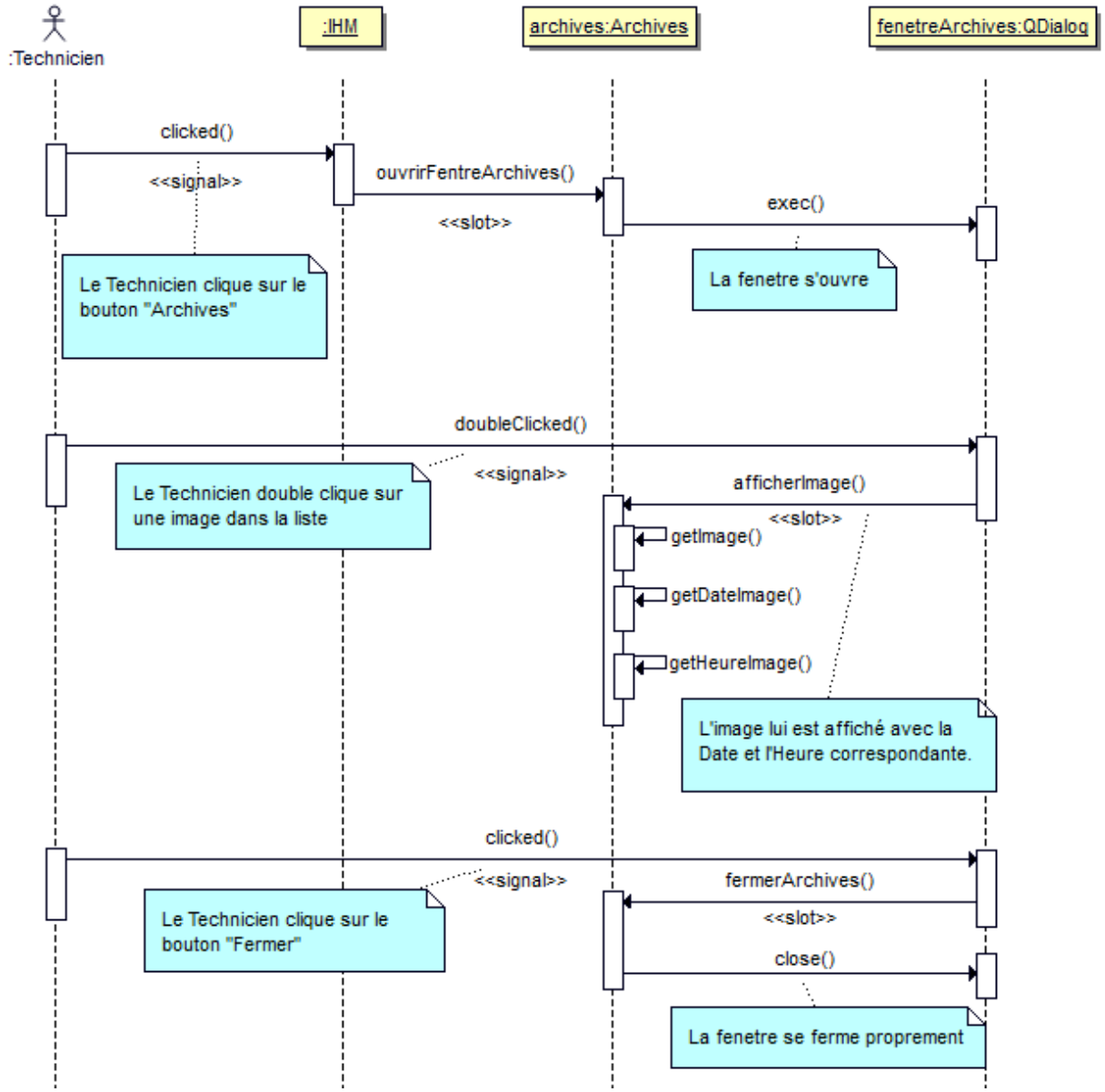
Cette fenêtre sera composée de :

- La liste des images capturées et horodatées
- L'affichage de la photo sélectionnée
- La date, l'heure, le taux de radiation et la température pour la photo sélectionnée
- Un bouton pour quitter la fenêtre



Naviguer dans les archives

Ce qui nous donne le diagramme de séquence :



La base de données

Le chemin de l'archive sera stocké (comme les mesures de radiations et de température) dans la base de données SQLite[9].

La base de données est composée des tables suivantes :

```
CREATE TABLE 'campagnes'
(
'nom' VARCHAR,
'description' VARCHAR,
'date' DATETIME,
'cheminArchives' VARCHAR,
'idOperateur' INTEGER,
'idCampagne' INTEGER PRIMARY KEY AUTOINCREMENT,
FOREIGN KEY(idOperateur) REFERENCES operateurs(idOperateur)
)
```

```
CREATE TABLE 'mesures'
(
'date' DATETIME,
'temperature' DOUBLE,
'irradiation' DOUBLE,
'idCampagne' INTEGER,
FOREIGN KEY(idCampagne) REFERENCES campagnes(idCampagne)
)
```

La table **mesures** se remplit automatiquement toutes les 30 secondes (par défaut) des mesures fournies par les capteurs sur le rov.

L'enregistrement automatique peut également être stoppé dans les paramètres.

9 SQLite est une bibliothèque en C proposant un moteur de base de données relationnelle accessible par le langage SQL.

```
CREATE TABLE 'operateurs'
(
'nom' VARCHAR,
'prenom' VARCHAR,
'idOperateur' INTEGER PRIMARY KEY AUTOINCREMENT
)
```

La table **operateurs** contient les deux opérateurs :

```
INSERT INTO 'operateurs' ('nom', 'prenom')
VALUES ('BOFFREDO', 'Nicolas');

INSERT INTO 'operateurs' ('nom', 'prenom')
VALUES ('REYNIER', 'Jacques');
```

Test de validation

Désignation	Procédure	Résultat attendu	Fonctionnel
Une nouvelle campagne est réalisable	Au démarrage, entrer le nom et la description de la nouvelle campagne, ainsi que l'opérateur chargé de cette campagne.	Une nouvelle campagne est lancée et est enregistrée dans la base de donnée	Oui
La caméra est pilotable	Utiliser les boutons droit et gauche de la croix directionnelle.	Le servomoteur fixé à la caméra se déplace de droite à gauche, par pas de 10°, aux commandes de l'opérateur.	Oui
Les mesures du capteur télémétrique	La distance est affiché avec les mesures de température et d'irradiation	La mesure de la distance s'affiche et s'actualise toute les 30sec	Oui
La caméra est affichée sur l'IHM	Vérifier l'affichage du flux vidéo en temps réel	Le flux vidéo est affiché	Oui
Prendre une photo	Cliquer sur le bouton "Prendre une photo"	Une photo de la caméra est archivée	Oui
Archiver les photos	Les photos archivées sont stockées dans un sous-dossier de la campagne, et peuvent être visionnable depuis l'IHM	Les photos sont stockées dans un sous-dossier "Archives" de la campagnes et peuvent être visualisé dans un navigateur, accessible depuis le bouton Archives sur l'IHM	Oui

Glossaire

Rov : (Remotely Operated Vehicle) Un « véhicule téléguidé », le plus souvent un petit robot contrôlé à distance

Campagne : Mission à réaliser.

TQC : (Tel Que Construit) Une représentation TQC se veut le plus proche du réel.

Liaison RS232 : Norme standardisant une voie de communication de type liaison série.

Transmission série : Forme de transmission de données où les éléments se succèdent les uns après les les autres, en opposition avec la transmission parallèle.

Base de Données : Conteneur permettant de stocker et de retrouver l'intégralité des données brutes ou d'informations.

SQL : (Structure Query Language) Langage de requêtes structurées et normalisé servant à exploiter des bases de données relationnelles.

SQLite : Bibliothèque écrite en C proposant un moteur de base de donnée relationnelle. Accessible par le langage SQL.

API : (Application Programming Interface) Interface de programmation d'application.

IHM : (Interface Homme-Machine) Interface permettant de connecter une personne à une machine.

Diagramme de classe : Schéma normalisé présentant les classes d'un système et ses relations.

Diagramme de séquence : Schéma normalisé décrivant les interactions entre les objets au cours du temps, et les responsabilités qu'ils assument.

Paquet : Archive comprenant les informations et les procédures nécessaires à une installation.

Singleton : Classe ne permettant qu'une seule et unique instance d'elle même.