Projet

ROV'NET

DOSSIER TECHNIQUE

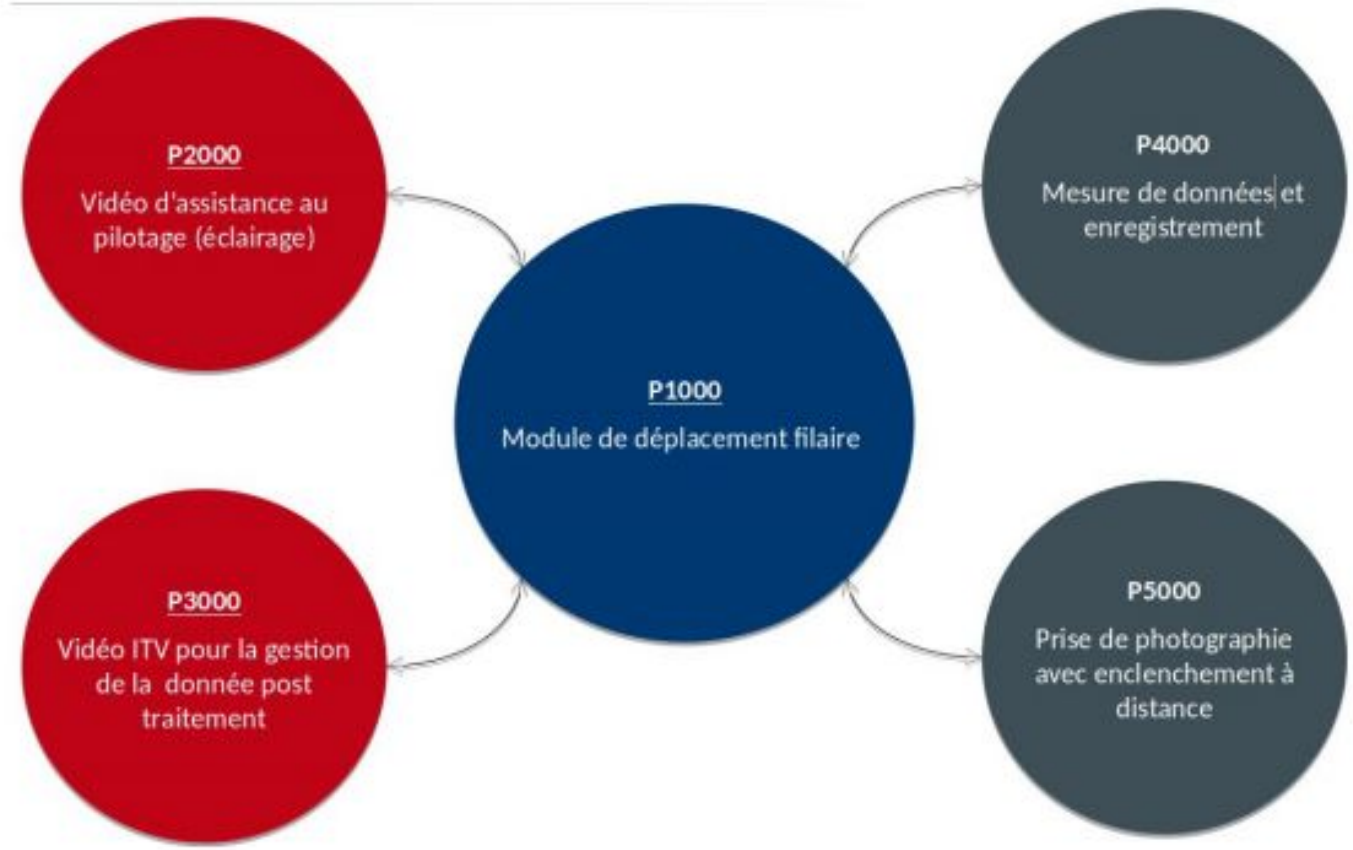
BTS SNIR session 2020

Bonnet Anthony Tenaille Servan

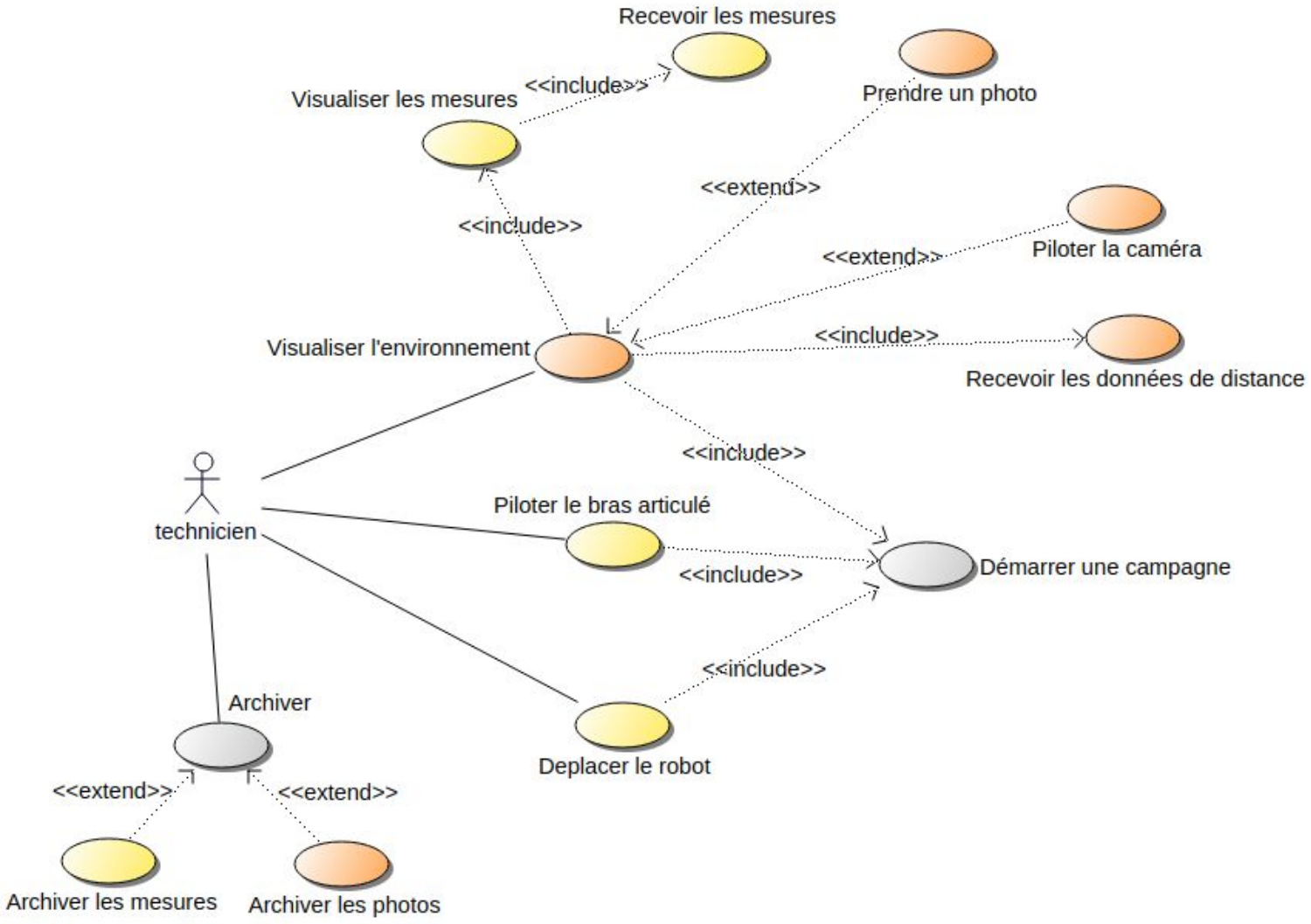
PARTIE COMMUNE

❏ Expression du besoin

Construire un ***R**emotely **O**perated **V**ehicle* (véhicule autoguidé) *low cost* avec des briques modulaires, capable d'être piloté à distance en liaison filaire.
Le ROV a pour but de se déplacer dans un milieu contaminé afin de faire des prises de vues et de collecter des données sur le site.



❑ Cas d’utilisation

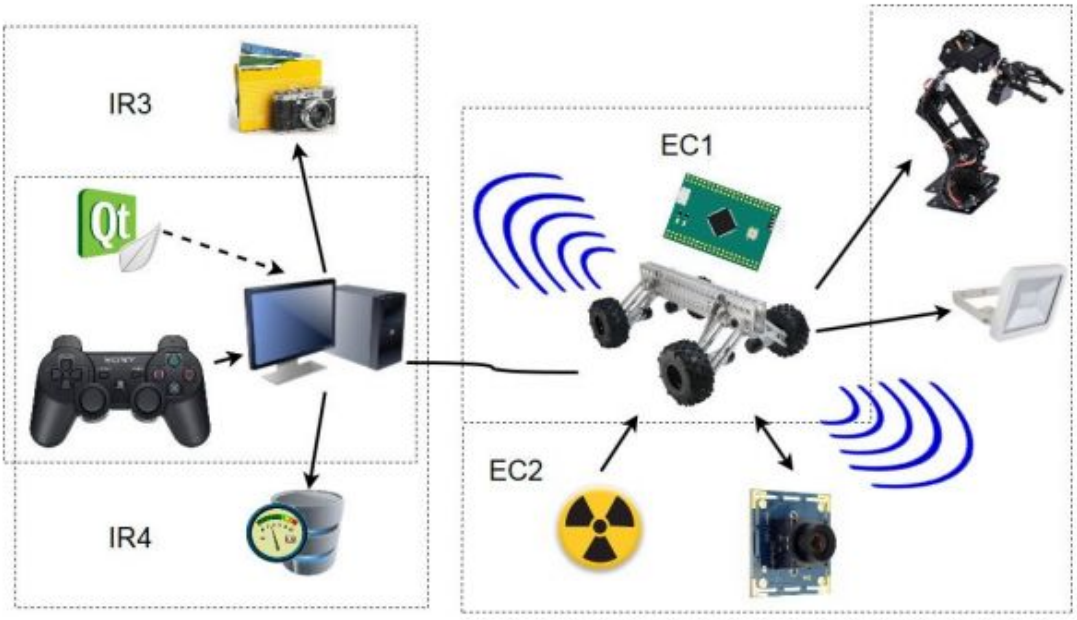


Seul acteur du système, le technicien (ou opérateur) démarrera une campagne (informations, lieux, technicien associé) et aura un aperçu de l’environnement qui intègre obligatoirement les données de distance (ces dernières permettant d’avoir des informations supplémentaires sur l’environnement supposé inconnu) . Ensuite, il pourra s’il le souhaite prendre une photo et piloter la caméra à l’aide d’une manette de type xbox. Le technicien pourra visualiser les mesures en provenance des capteurs et pilotera le robot ainsi que le bras à l’aide de la manette. À la fin de la campagne, le technicien pourra archiver les photos et les mesures.

❏ Identification des tâches

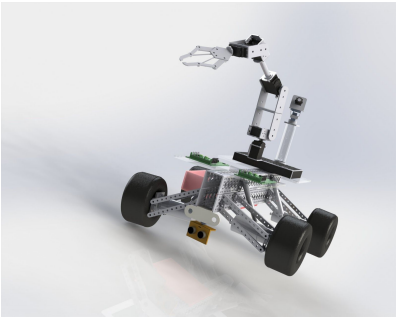
Tâches	Niveau	Itération
Etablir un protocole de communication	Critique	1
Configurer la communication avec le ROV	Critique	1
Prise en charge de la caméra	Critique	1
Prise en charge de la manette	Critique	1
Visualiser l'environnement (flux vidéo)	Important	1
Afficher les données de télémétrie	Important	2
Afficher les données des capteurs	Important	2
Fabrication des trames de déplacement du robot	Important	2
Fabrication des trames de pilotage du bras	Important	2
Fabrication des trames de déplacement de la caméra	Important	2
Prendre une photo	Important	2
Archivage des données	Important	3
Configurer une campagne	Secondaire	3

❑ Ressources



- Ressources matérielles

- Rov



- Manette Xbox 360

- ELP 1 mégapixel USB caméra avec LED infrarouges, lentille 3.6mm, résolution 1280 x 720 pixels

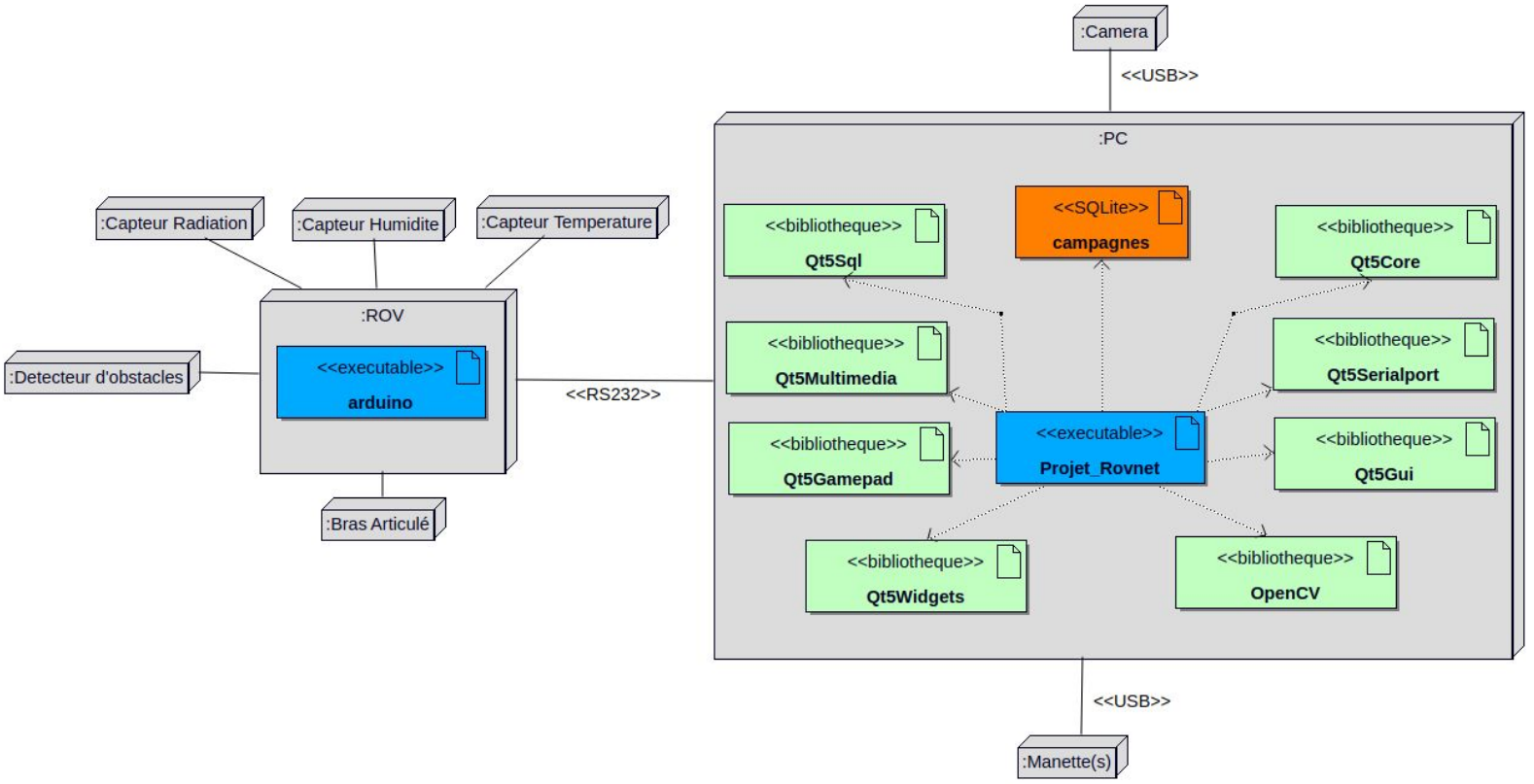
- **Ordinateur**

- Ressources Logicielles

- Qt Creator + framework Qt

- Moteur de base de données relationnelle -> bibliothèque SQLITE

❑ Diagramme de déploiement



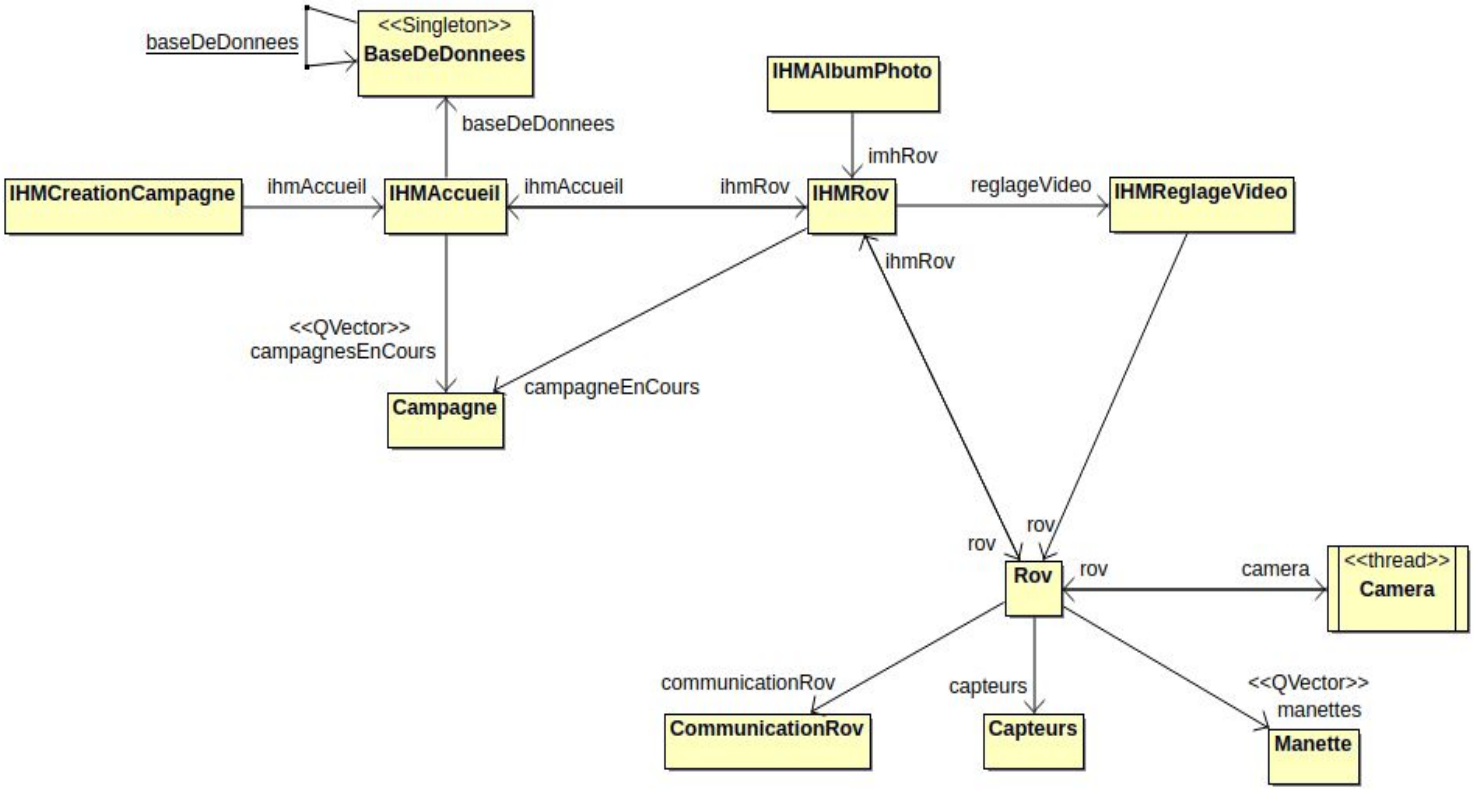
❑ Tests de validation

Classe	Description	Test effectué	Résultat Attendu	Résultat observé
Valide n°1	Une campagne doit pouvoir être configurée et démarrée	Saisie des informations d'une campagne et démarrage de celle-ci	La campagne démarre normalement selon les informations configurées	
Valide n°2	Visualiser les mesures des capteurs	Visualisation des données sur l'IHM	Les données affichées sur l'IHM sont cohérentes à l'environnement actuel du robot	
Valide n°3	Visualiser l'environnement	Visualisation de l'environnement sur l'IHM à l'aide de la caméra embarquée	Le flux vidéo observé sur l'IHM est cohérent avec l'environnement actuel du robot	
Valide n°4	Recevoir les données de télémétrie	Visualisation des données de télémétrie sur l'IHM	Les données de télémétrie sont compréhensibles et cohérentes avec l'environnement actuel du robot	
Valide n°5	Prendre une photo	Prise d'une photo à l'aide d'une fonctionnalité de l'IHM	La photo récupérée est cohérente avec l'environnement actuel	
Valide n°6	Archiver les photos	Enregistrement de la photo prise dans la base de données à l'aide d'une fonctionnalité de l'IHM	Les données archivées sont correctement enregistrées dans la base de données en cohérence avec la structure de la base de données établie	
Valide n°7	Piloter la caméra	Pilotage de la caméra à l'aide de la manette	La caméra doit réagir en cohérence aux instructions envoyées	

		connectée	par la manette	
Valide n°8	Déplacer le robot	Déplacement du robot à l'aide de la manette connectée	Le robot doit réagir en cohérence aux instructions envoyées par la manette	
Valide n°9	Piloter le bras articulé	Pilotage du bras articulé à l'aide de la manette connectée	Le bras articulé doit réagir en cohérence aux instructions envoyées par la manette	
Valide n°10	Archiver les mesures	Récupérer les mesures et les enregistrer dans la base de données	Les données archivées sont correctement enregistrées dans la base de données en cohérence avec la structure de la base de données établie	

❏ Analyse

- Diagramme de classes



- Description des classes:

❖ **IHMAccueil :** Cette classe est une interface permettant la gestion des campagnes (création, suppression, archive et lancement).

- **IHMCreationCampagne :** Cette classe est une interface permettant la création du nouvelle campagne

- **IHMRov :** Cette classe est l'interface de communication avec le rov

- **IHMAlbumPhoto :** Cette classe est une interface permettant de gérer les photos prises durant la campagne

- **IHMReglageVideo :** Cette classe est une interface permettant de gérer les réglages du flux vidéo (contraste, luminosité, saturation, résolution) et de sélectionner une caméra

- **BaseDeDonnees :** Cette classe permet de s'interfacer avec la base de données SQLite contenant les informations d'une campagne ainsi que les mesures et les photos prises

- **Campagne :** Cette classe permet de gérer les informations d'une campagne (nom, informations technicien, lieu, chemin de sauvegarde des photos, date, durée de la campagne, les conteneurs de mesures et de photos)

- **Rov :** Cette classe gère les informations en provenance du flux vidéo et des capteurs afin de les diriger vers l'IHM ainsi que les différents ordres à envoyer au robot

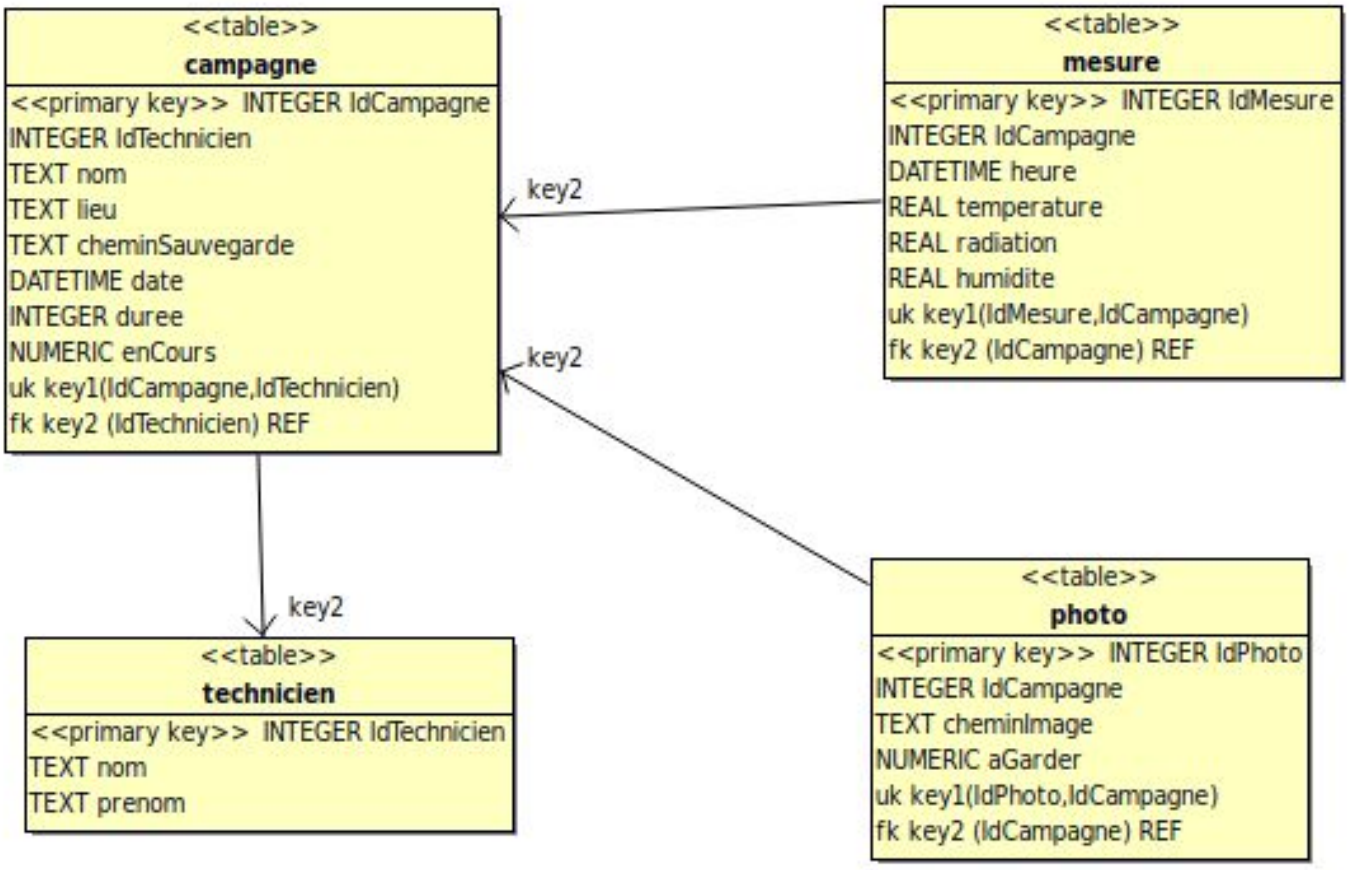
- **Camera :** Cette classe active gère la prise en charge de la caméra dans un thread différent.

- **CommunicationRov :** Cette classe gère la communication avec le rov en prenant en charge la configuration du port série

- **Capteurs :** Cette classe contient les dernières données des différents capteurs présents sur le robot

- **Manette :** Cette classe prend en charge la manette en convertissant les différents événements en signaux Qt

- Schéma relationnel de la base de données



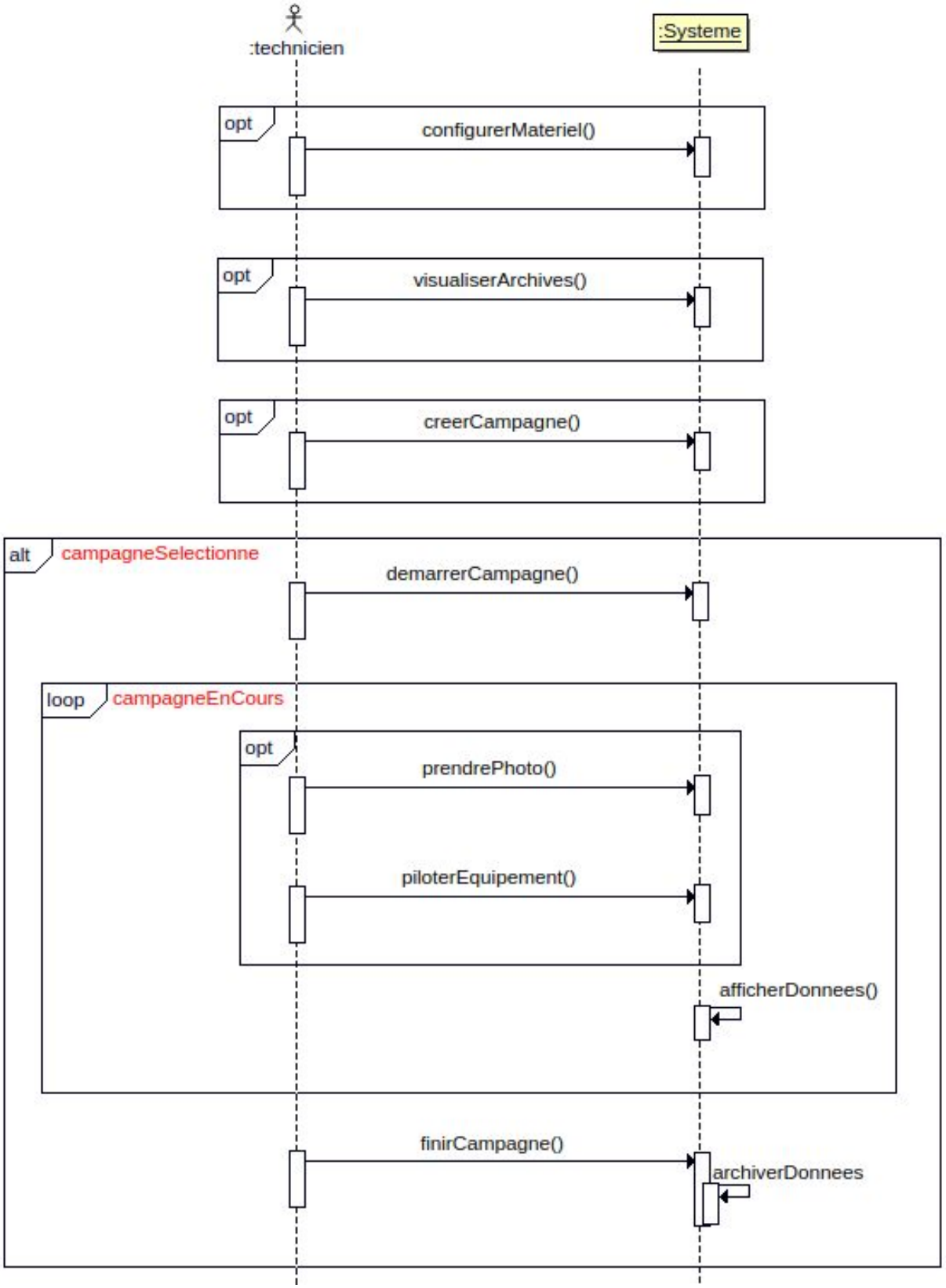
❖**Table campagne :** Contient les informations relatives à chaque campagne, celles-ci étant associées à un technicien. Elle est identifiée par une clé primaire “IdCampagne”. Une campagne est réalisée par un technicien (clé étrangère “idTechnicien”).

❖**Table technicien :** Contient les informations relatives à un technicien enregistré. Elle est identifiée par une clé primaire “IdTechnicien”.

- **Table mesure :** Contient les mesures enregistrées par les capteurs du robot, chaque mesure étant associé à une campagne (clé étrangère “idCampagne”). Une mesure est identifiée par une clé primaire “IdMesure”.

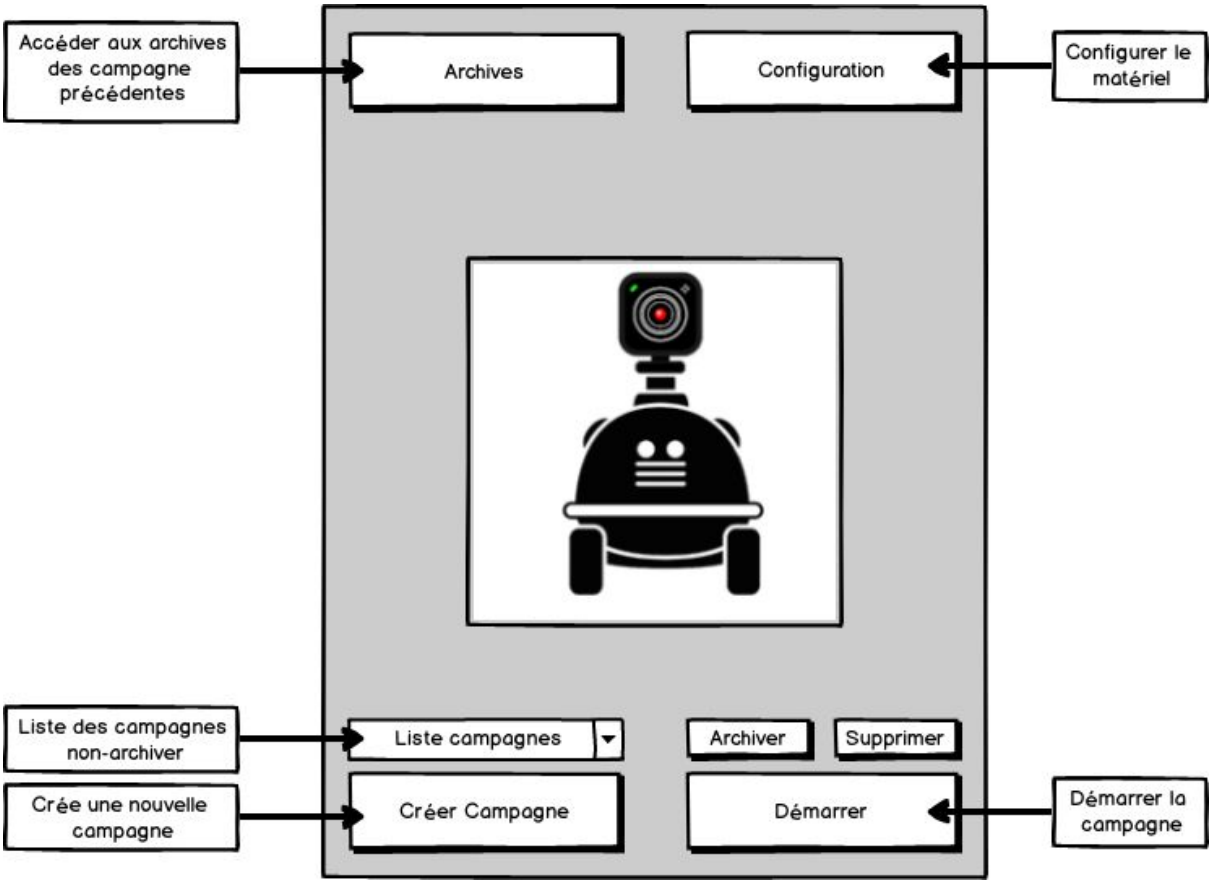
- **Table photo :** Contient les photos prises par le technicien durant une campagne, chaque photo étant associée à une campagne (clé étrangère “idCampagne”). Une photo est identifiée par une clé primaire “IdPhoto”.

- Diagramme de séquence système

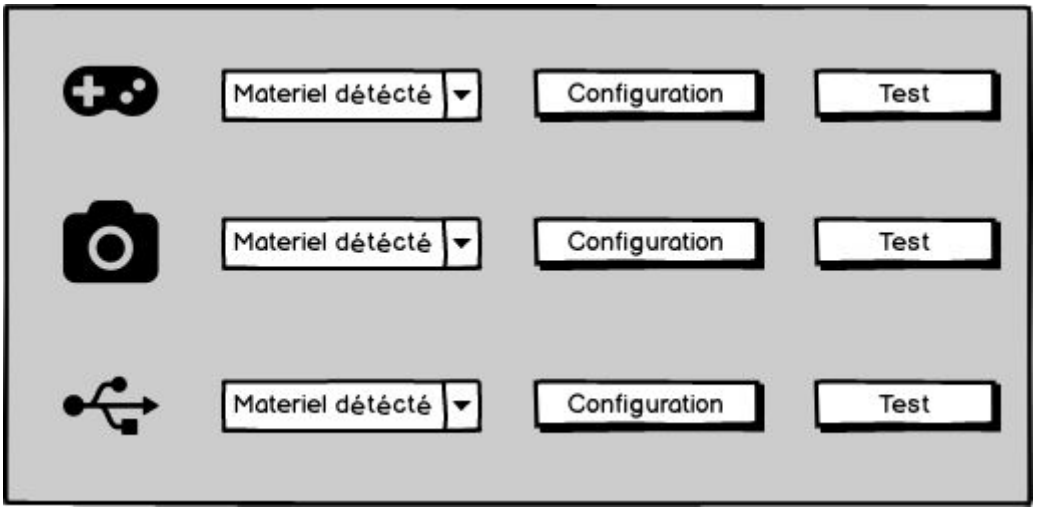


- Maquettes IHM

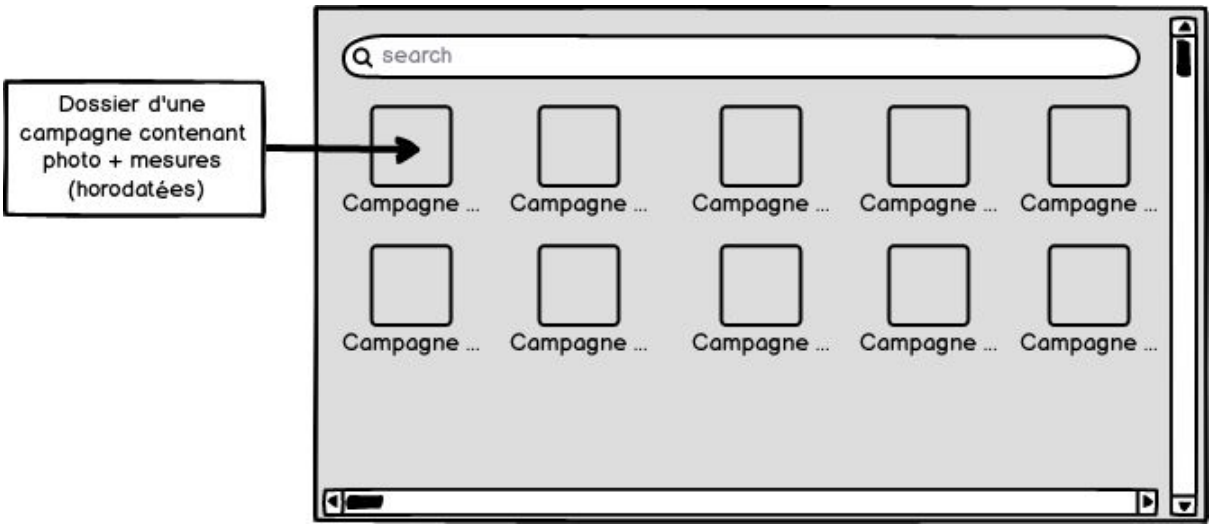
IHM page d'accueil



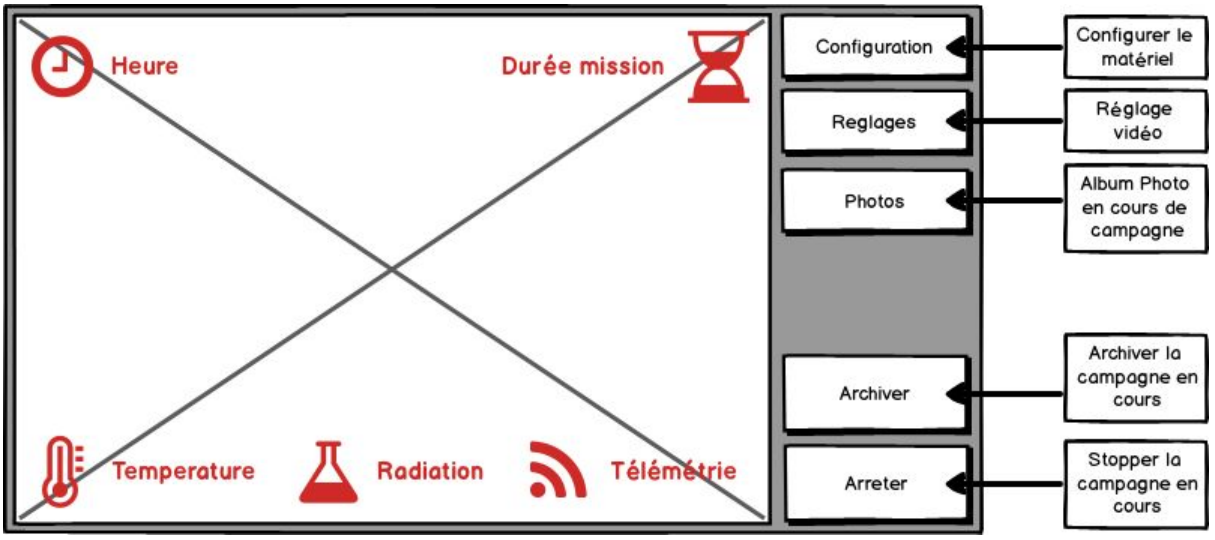
IHM configuration



IHM archives



IHM campagne



❏ Protocole de communication

Caractéristiques générales

Protocole orienté ASCII
Entête de trame : '**$**'
Délimiteur de champs : '**;**'
Délimiteur de fin de trame : "**\r\n**"
Format général :

$TYPE;{PARAMETRES}\r\n

TYPE : Type de trame (déplacement, caméra, …)
PARAMETRES : Données associées au TYPE de trame

Remarques :
{} : champ présent obligatoirement
[] : champ optionnel
| : ou

Trames

❏ Application => Rov

$DEP;{A|R|0};PUISSANCE;{G|D|0}\r\n

DEP : Trame déplacement du robot
{A|R|0} : Axe X de déplacement (**A** = Avancer, **R** = Reculer, **0** = Pas de mouvement)
PUISSANCE : Puissance moteur exprimée en pourcentage (**0** -> **100**)
{G|D|0} : Axe Y de déplacement (**G** = Gauche, **D** = Droite, **0** = Pas de déplacement)

$CAM;{G|D|0};{H|B|0}\r\n

CAM : Trame de pilotage de la caméra
{G|D|0} : Axe X de déplacement (**G** = Gauche, **D** = Droite, **0** = Pas de déplacement)
{H|B|0} : Axe Y de déplacement (**H** = Haut, **B** = Bas, **0** = Pas de déplacement)

$BRAS;{GAUCHE|DROITE|MONTE|DESCEND|AVANCE|RECULE}\r\n

BRAS : Trame de commande du bras
{GAUCHE|DROITE|MONTE|DESCEND|AVANCE|RECULE|0} : Type de demande de pilotage du bras

$ORDRE;{INIT|PINCE|BAC}\r\n

ORDRE : Trame de commande du bras
{INIT|PINCE|BAC} : Type d'ordre (**INIT** = Retour à l'état initial de la pince, **PINCE** = Ouvrir ou fermer la pince à 100%, **BAC** = Mettre l'objet dans le bac

- Rov => Application (trame périodique)

$CAP;TEMPERATURE;RADIATION\r\n

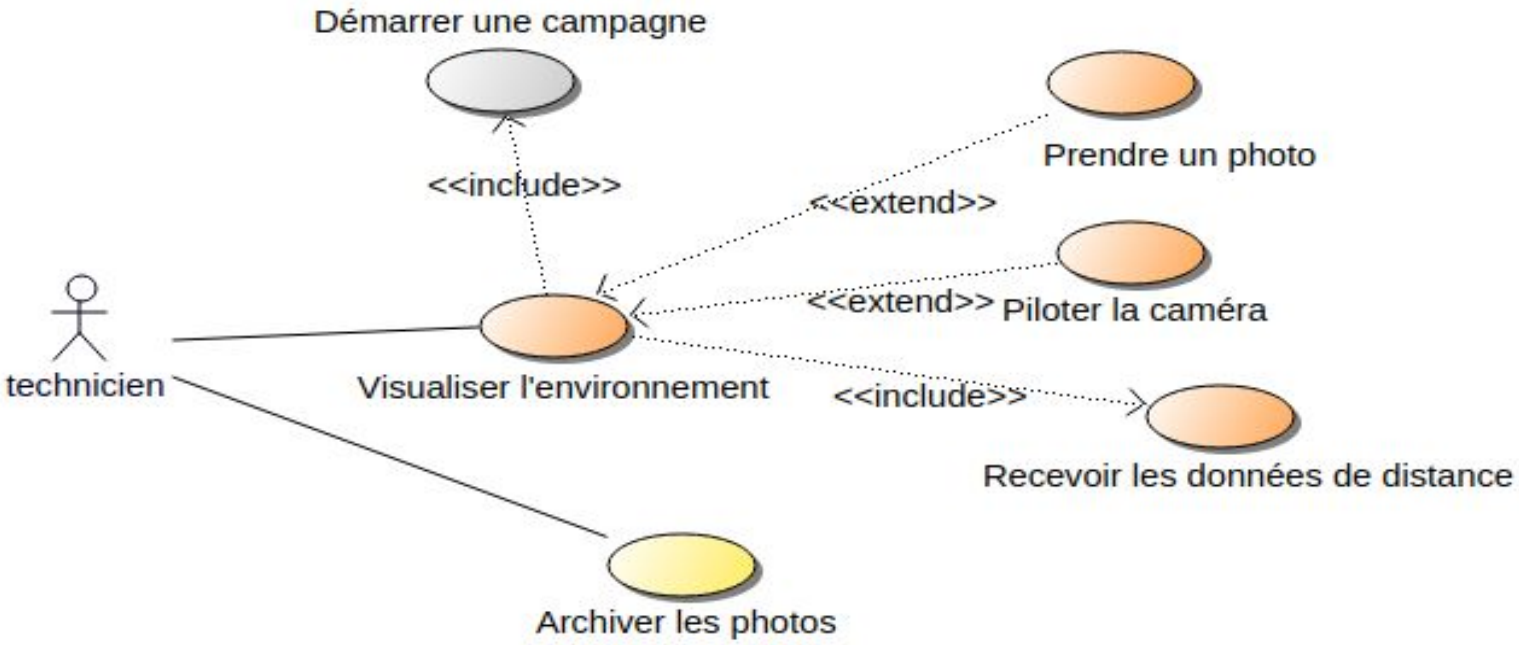
CAP : Trame d'état des capteurs
TEMPERATURE : Température(dixième de °C)
RADIATION : radiation(uS/h)
Période : 5 secondes

$TEL;DISTANCE;ANGLE\r\n

TEL: Trame de télémétrie
DISTANCE : distance (cm)
ANGLE : angle (0 - 180°)
Période : 250 millisecondes

PARTIE PERSONNELLE : BONNET Anthony (IR)

❏ Cas d'utilisations personnels



Le technicien, seul acteur du système, aura pour rôle de diriger la campagne. Dans ma partie ce technicien pourra :

- Démarrer une campagne, c'est à dire reprendre une campagne non archivée ou en créer une nouvelle avec toutes les informations relatives à celle-ci (nom de la campagne, nom/prénom du technicien, lieu de la campagne).

- Une fois la campagne démarrée l'acteur pourra visualiser l'environnement qui comprend les données du détecteur d'obstacle. À partir de cet environnement il pourra s'il le souhaite prendre des photos directement à l'aide de la manette et également contrôler l'orientation de la vue de la caméra à l'aide de cette même manette.

- Il pourra lorsqu'il le souhaite mettre fin à la campagne et archiver les photos prises durant la campagne.

❏ Planification des tâches personnelles

Le développement est découpé en plusieurs itérations, elles-mêmes contenant plusieurs objectifs simples afin de réaliser les fonctionnalités demandées :

- **Itération numéro 1 :**
    - Prise en charge de la caméra
    - Visualiser l'environnement
    - Configurer la communication avec le rov

- **Itération numéro 2 :**
    - Prise en charge de la manette
    - Fabrication des trames de déplacement de la caméra
    - Prendre une photo

- **Itération numéro 3 :**
    - Archivage des données
    - Configurer une campagne

❏ Introduction à Qt

Le langage utilisé pour le développement de cette partie personnelle est le C++ avec l'aide de la bibliothèque logicielle Qt. Cette dernière est notamment une plateforme de développement d'interfaces graphiques GUI.

Qt fournit également un ensemble de classes décrivant des éléments non graphiques :accès aux données (fichier, base de données), connexions réseaux (socket), gestion du multitâche (thread), XML, etc…

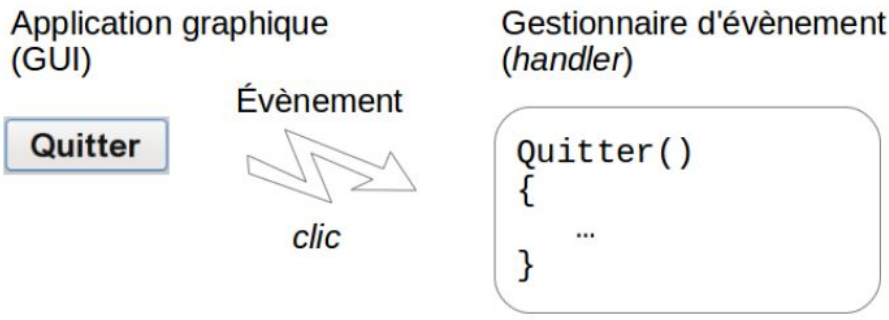
❑ Programmation événementielle

La programmation événementielle est une programmation basée sur les événements.

Le programme sera principalement défini par ses réactions aux différents événements qui peuvent se produire, c'est-à-dire des changements d'état, par exemple l'incrémentation d'une liste, un mouvement de souris ou de clavier etc.

La programmation événementielle est architecturée autour d'une boucle principale fournie et divisée en deux sections : la première section détecte les événements, la seconde les gère.

Pour chaque évènement à gérer, il faut lui associer une action à réaliser (le code d'une fonction ou méthode) : c'est le gestionnaire d'événement (handler).



La programmation événementielle des applications Qt est basée sur un mécanisme appelé *signal/slot* :

– Un signal est émis lorsqu'un événement particulier se produit. Les classes de Qt possèdent de nombreux signaux prédéfinis mais vous pouvez aussi hériter de ces classes et leur ajouter vos propres signaux.

- Un slot est une fonction qui va être appelée en réponse à un signal particulier. De même, les classes de Qt possèdent de nombreux slots prédéfinis, mais il est très courant d'hériter de ces classes et de créer ses propres slots afin de gérer les signaux qui vous intéressent.L'association d'un signal à un slot est réalisée par une connexion (connect()).

❏ Module Qt

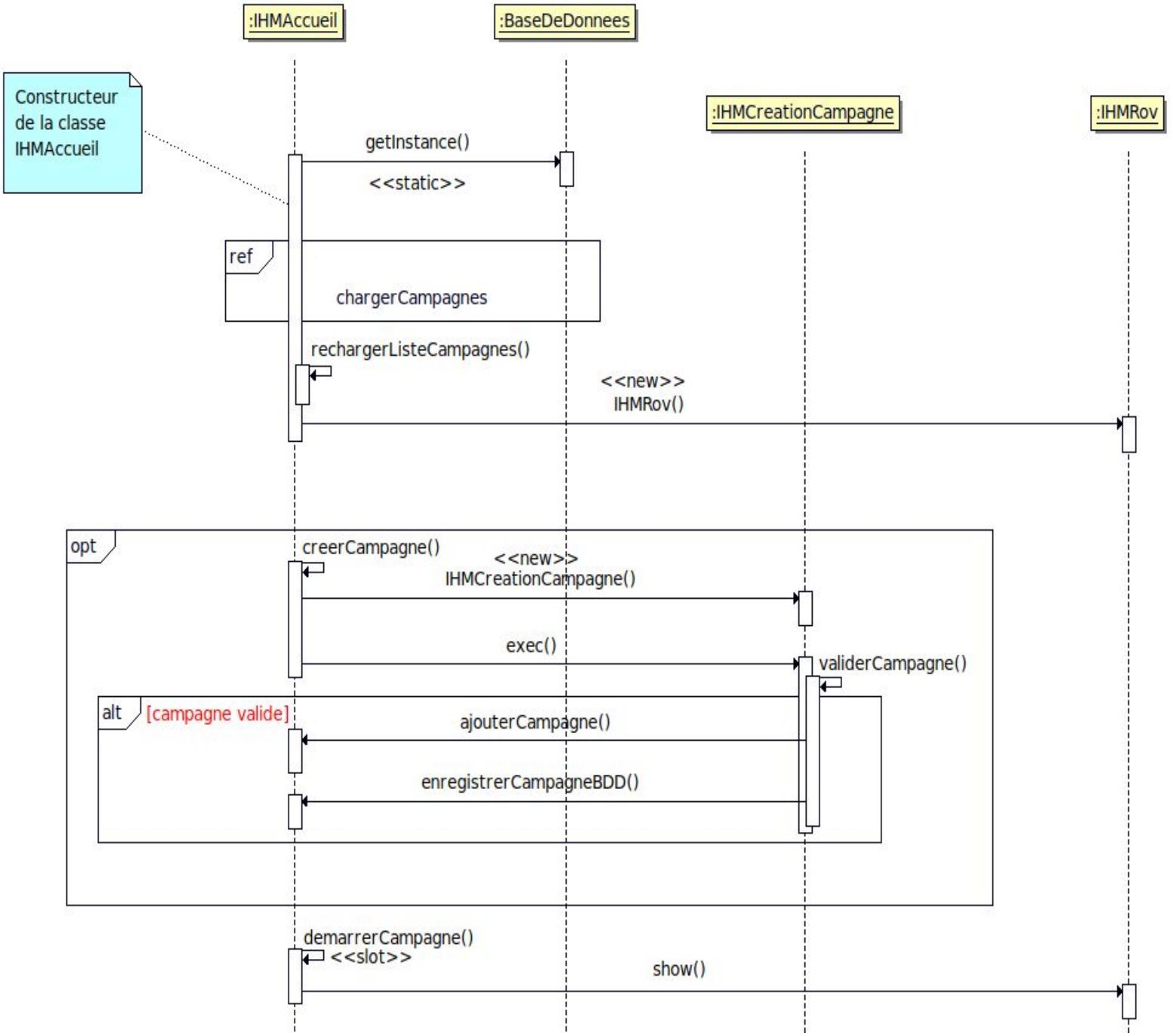
Qt sépare sa bibliothèque en modules, il faut activer un module dans un projet Qt pour pouvoir accéder aux classes qu'il regroupe.

Les différents modules utilisés dans le projet sont informés dans le fichier **`Projet_Rovnet.pro`** :

```
QT   += core gui serialport gamepad multimedia sql
```

- QtCore : Classes de base non graphiques utilisées par les autres modules
- QtGui : Composants d'interfaces graphiques (IG ou, en anglais, GUI, Graphical User Interface)
- QtSerialPort : Donne accès au matériel et aux ports séries virtuels.
- QtGamepad : Permet aux applications Qt de prendre en charge l’utilisation de manette.
- QtMultimedia : Classes pour les fonctionnalités multimédias de bas niveau
- QtSql : Classes pour l'intégration des bases de données avec SQL

❏ Scénario : Démarrer une campagne



- Déroulement du scénario

Constructeur IHMAccueil :

- ➔ Récupère un objet baseDeDonnees
- ➔ Une méthode privée chargerCampagnes() est appelée afin de récupérer les campagnes non terminées
- ➔ Une méthode privée rechargerListeCampagnes() remplit la liste déroulante présente dans l'ihm afin de pouvoir choisir la campagne que l'on voudra démarrer
- ➔ Initialise un nouvel objet ihmRov

Objet ihmAccueil:

- ➔ Possibilité de creerCampagne() :
  - ◆ Celle-ci initialise un nouvel objet IHMCreationCampagne
  - ◆ Puis l'appelle de la méthode exec() sur l'objet ihmCreationCampagne

Objet ihmCreationCampagne:

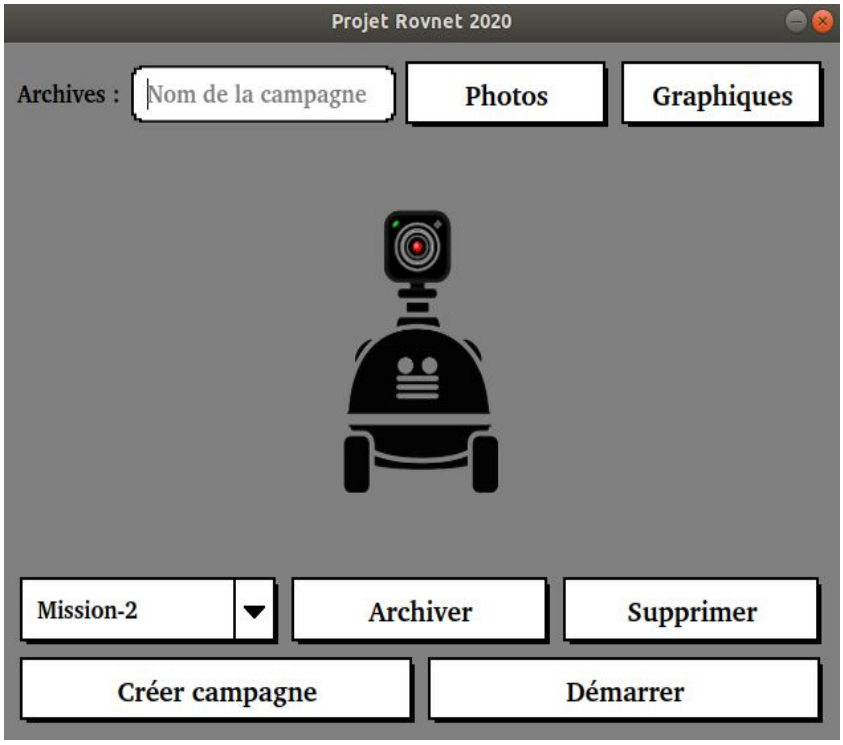
- ➔ Une méthode privée validerCampagne() est appelée qui permettra si la campagne est valide (Formulaire de création de campagne rempli sans erreur) :
  - ◆ D'ajouter la campagne créée dans le conteneur de Campagne grâce à l'objet ihmAccueil et sa méthode ajouterCampagne()
  - ◆ D'enregistrer les informations de la campagne dans la base de données grâce à la méthode enregistrerCampagneBDD() de l'objet ihmAccueil

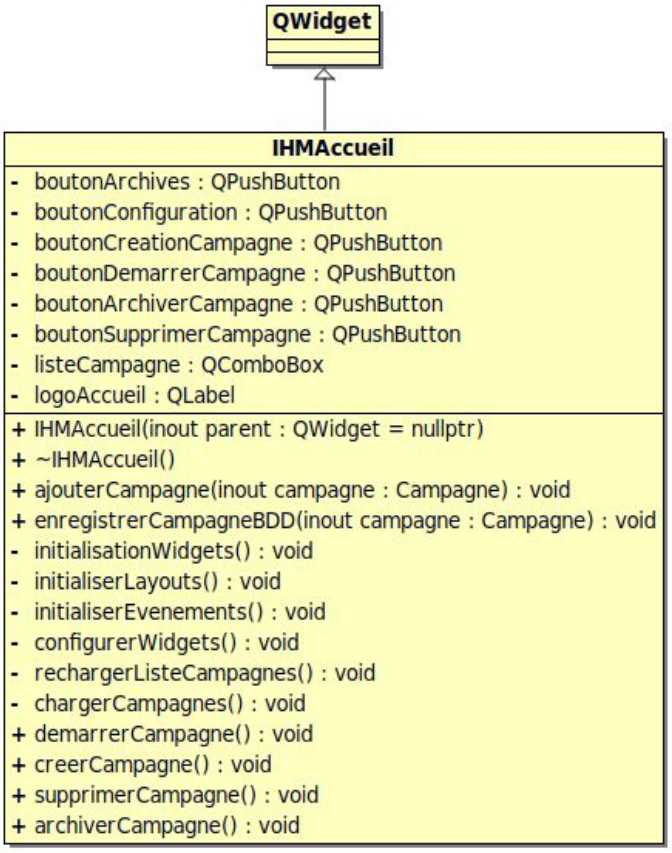
Objet ihmAccueil:

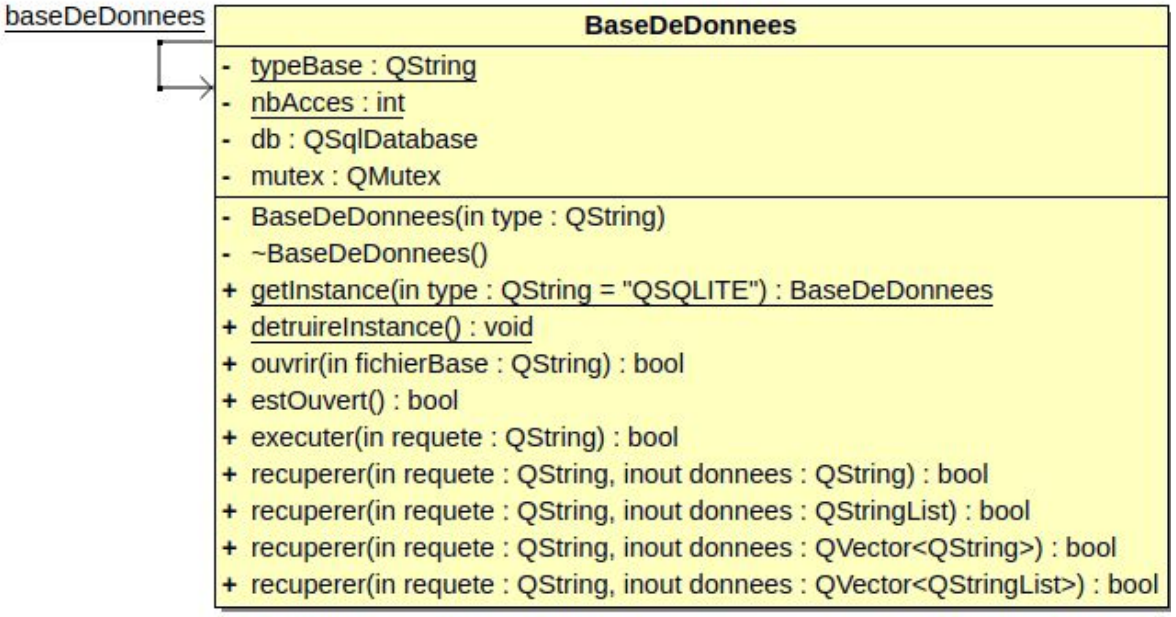
- ➔ Après un signal venant de l'ihmAccueil, le slot demarrerCampagne() est déclenché celui-ci appelle la méthode show() de l'objet ihmRov afin de démarrer la campagne

- Explications techniques

La classe IHMAcueil héritant de QWidget est une Interface Homme Machine permettant de créer une nouvelle campagne, d'archiver ou supprimer une campagne et enfin de démarrer la campagne sélectionnée.



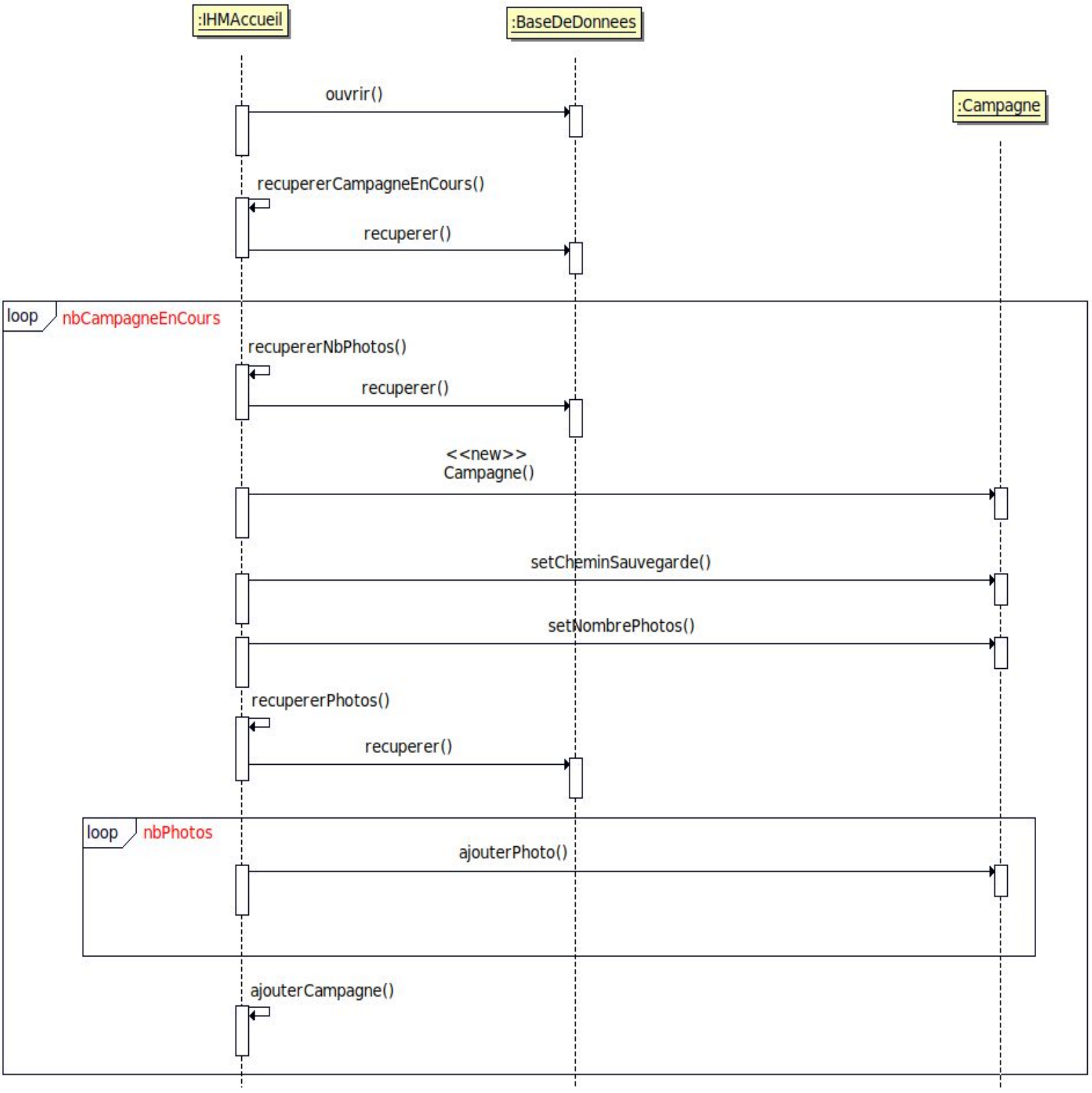




La classe BaseDeDonnees fait partie de l'un des patrons de conceptions nommé Singleton. En effet cette classe n'a besoin d'être instanciée qu'une seule fois afin d'éviter les conflits d'accès à la base de données. La méthode static `getInstance()` permet de ne récupérer qu'un seul et même objet durant la durée de vie du logiciel, le constructeur étant privé l'objet ne peut être instancié qu'au sein de sa propre classe.

```
 static BaseDeDonnees* getInstance(QString type="QSQLITE");

BaseDeDonnees* BaseDeDonnees::getInstance(QString type)
{
      if(baseDeDonnees == nullptr)
      baseDeDonnees = new BaseDeDonnees(type);

      nbAcces++;

      return baseDeDonnees;
}
```

❏ Diagramme ref : chargerCampagnes



La méthode `recuperer()` est une méthode surchargée permettant de récupérer selon les besoins de la requête envoyée :

- Un champ d'un seul enregistrement
- Plusieurs champs d'un seul enregistrement
- Un champ de plusieurs enregistrements
- Plusieurs champs de plusieurs enregistrements

Dans ce scénario cette méthode est appelée pour diverses requêtes :

- `recupererCampagneEnCours()` permet de récupérer plusieurs champs de plusieurs enregistrements afin de remplir le conteneur de campagne avec les informations des campagnes non-terminés et de les mettre disponible dans l'IHM

- `recuperNbPhotos()` permet de récupérer la somme des photos pour une campagne donnée

- `recupererPhotos()` permet de récupérer plusieurs champs de plusieurs enregistrements afin de remplir le conteneur de Photo d'une campagne

La requête pour récupérer les campagnes en cours (les plus récentes en premier) est de la forme :

```
SELECT campagne.nom, campagne.lieu, technicien.nom, technicien.prenom,
campagne.date, campagne.duree, campagne.enCours FROM campagne INNER JOIN
technicien ON campagne.idTechnicien = technicien.idTechnicien WHERE
campagne.enCours = '1' ORDER BY campagne.date DESC;
```

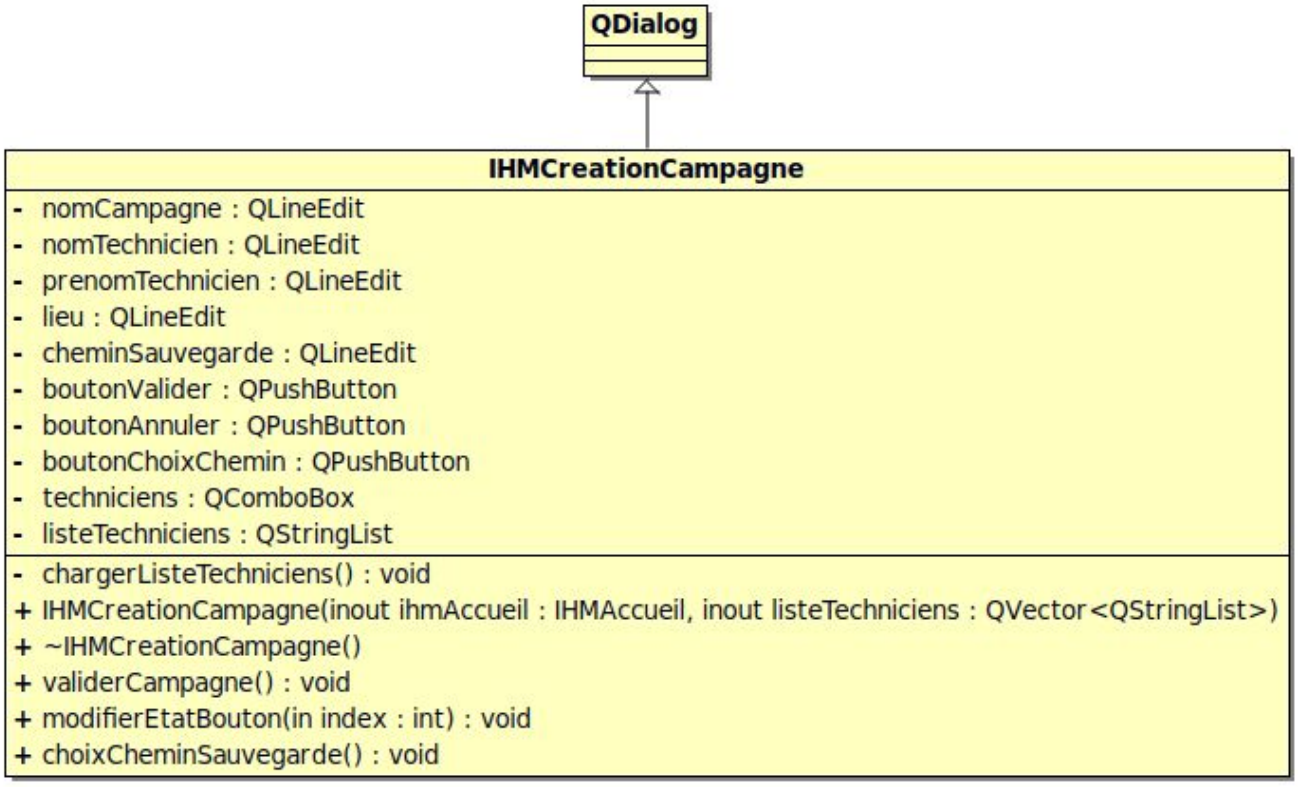
La requête pour récupérer le nombre de photos d'une campagne est de la forme :

```
SELECT COUNT(IdPhoto) FROM photo INNER JOIN campagne ON photo.IdCampagne
= campagne.IdCampagne WHERE campagne.IdCampagne = 'idCampagne'
```

La requête pour récupérer les photos est de la forme :

```
SELECT cheminImage, aGarder FROM photo INNER JOIN campagne ON
photo.IdCampagne = campagne.IdCampagne WHERE campagne.enCours = '1' AND
photo.IdCampagne = 'idCampagne'
```

- `creerCampagne()` instancie un nouvel objet ihmCreationCampagne héritant de QDialog.




```
void IHMAccueil::creerCampagne()
{
	IHMCreationCampagne *ihmCreationCampagne = new IHMCreationCampagne(this);
	ihmCreationCampagne->exec();
}
```

- La méthode `exec()` est appelée afin de bloquer la fenêtre dans sa propre boucle d'événements.

- Lors de la validation de la création de la campagne toutes les informations relatives à celle-ci sont enregistrées dans la base de données grâce à la méthode :

```
void enregistrerCampagneBDD(Campagne *campagne);
```

Avant d'enregistrer la campagne il faut récupérer l'id du technicien concerné :

```
SELECT technicien.IdTechnicien FROM technicien WHERE
technicien.nom = 'nomRecherche' AND technicien.prenom =
'prenomRecherche';
```

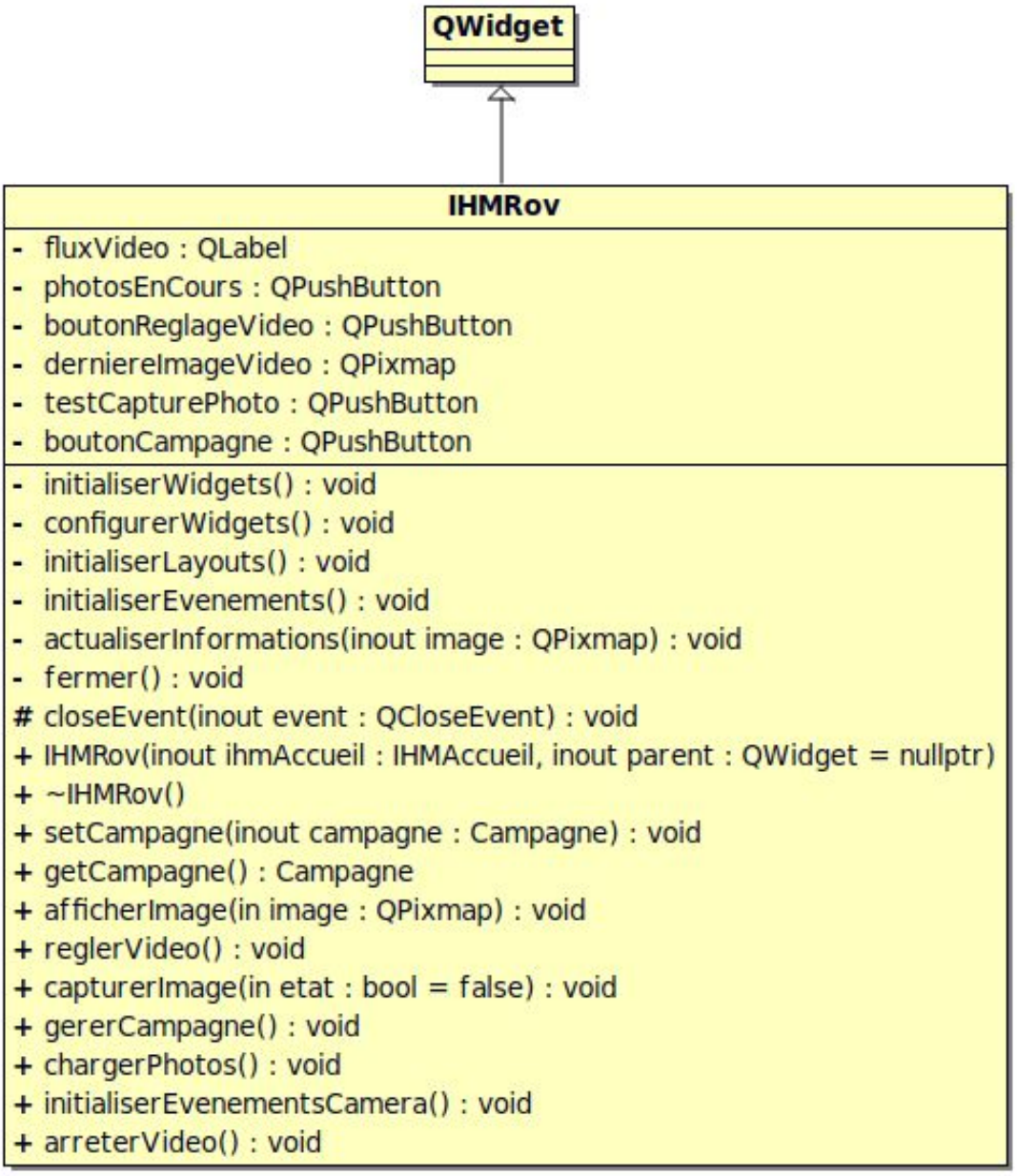
Récupère l'id du technicien sélectionné dans la liste déroulante, si le retour engendre une erreur (technicien non enregistré), le technicien est enregistré dans la base de données :

```
INSERT INTO technicien (nom, prenom) VALUES
('NomTechnicien','PrenomTechnicien');
```

Enfin la requête pour enregistrer la campagne créée est de la forme :

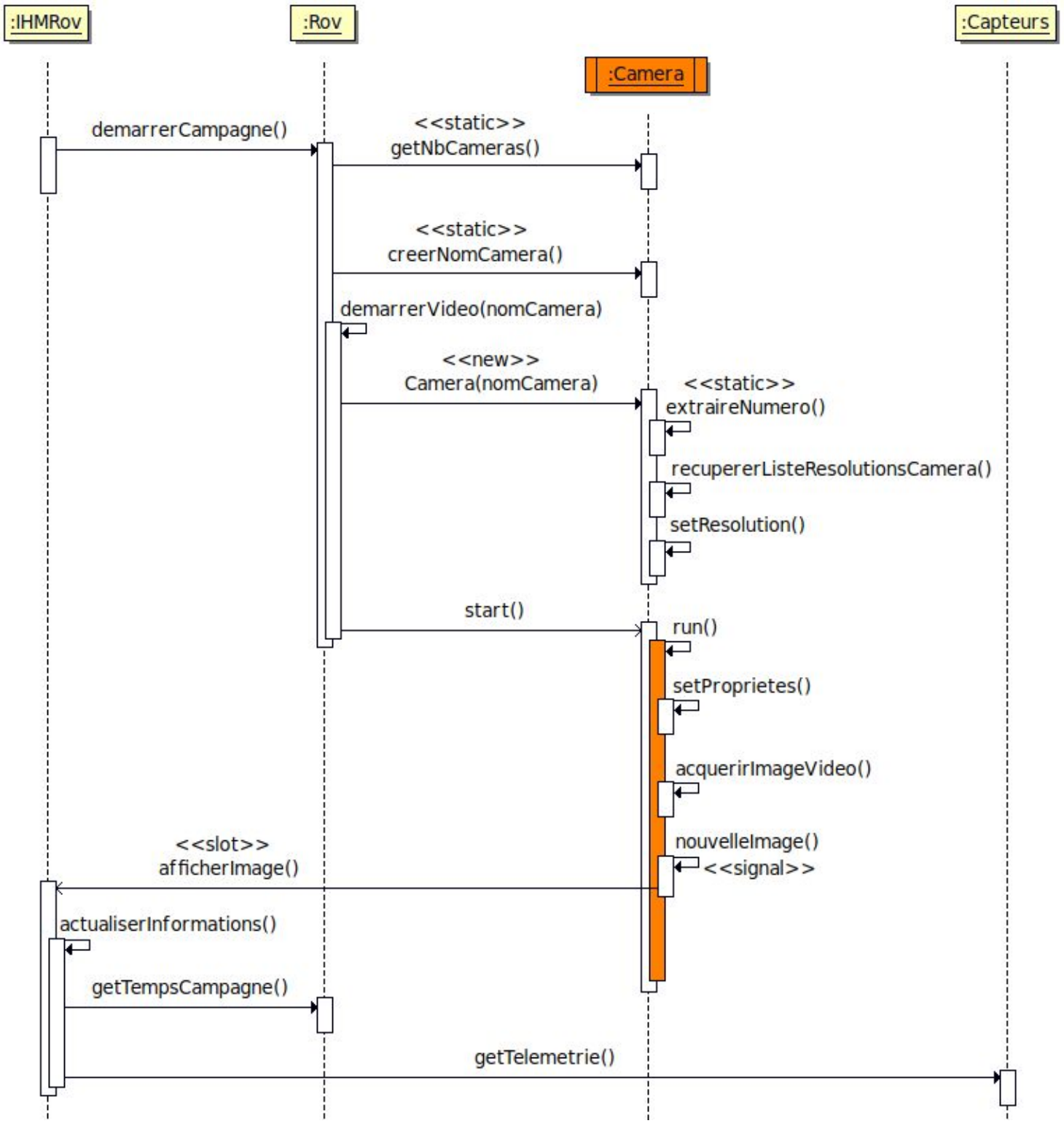
```
INSERT INTO campagne (idTechnicien, nom, lieu, date, duree,
enCours, cheminSauvegarde) VALUES ('idTechnicien', 'NomCampagne',
'Lieu', 'Date', 'Duree', '1', 'cheminSauvegarde');
```

Pour finir le scénario, l'objet ihmRov appel sa méthode show :



Cette objet est l'interface avec la campagne en cours.

❑ Scénario : visualiser l'environnement



- Déroulement du scénario

Ce scénario démarre par l'appel de la méthode `demarrerCampagne()` présente dans la classe Rov.

Objet rov :

- ➔ Récupère le nombre de caméras détectées à l'aide de la méthode static getNbCameras()
- ➔ Créé le nom de la caméra (ici la caméra par défaut)
- ➔ La méthode demarrerVideo() s'occupe de :
  - ◆ Instancier un nouvel objet camera en prenant en paramètre le nom de la caméra
  - ◆ Appelle la méthode `start()` afin de lancer le thread de l'acquisition vidéo

Objet camera:

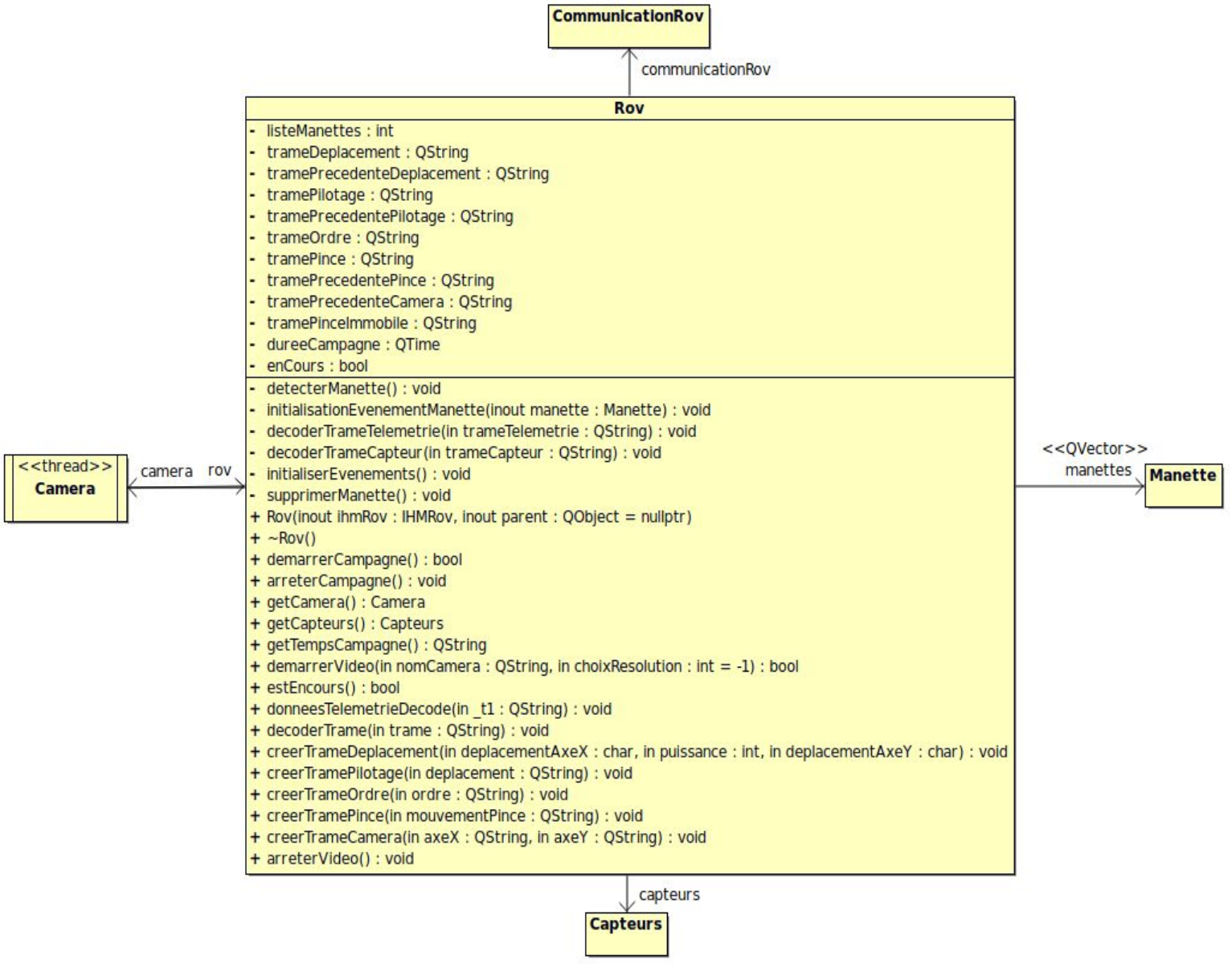
- ➔ La méthode `start()` appelle la méthode `run()` celle-ci:
  - ◆ Initialise les propriétés de la caméra (luminosité, contraste, saturation) à l'aide de la méthode `setProprietes()`
  - ◆ Se place dans une boucle qui tourne tant que la caméra est ouverte et que l'interruption n'est pas demandé. Cette boucle:
    - Acquiert la nouvelle image du flux vidéo grâce à la méthode `acquerirImageVideo()`
    - Envoie un signal contenant cette dernière image `nouvelleImage()`
    - Modifie les paramètres de la caméra s'il y a eu des changements grâce à la méthode `setProprietes()`

Objet ihmRov:

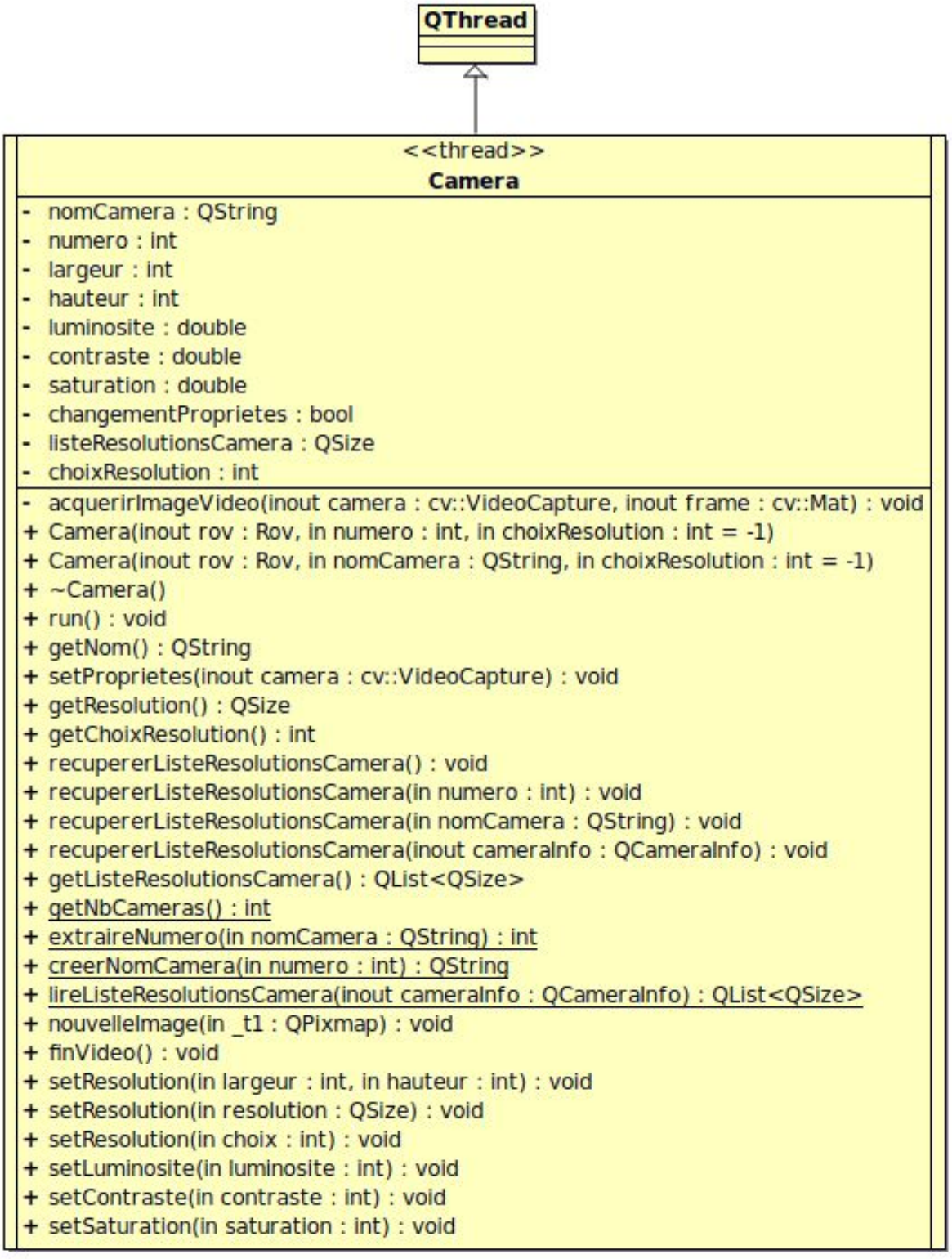
- ➔ À la réception du signal `nouvelleImage()`, le slot `afficherImage()` récupère la dernière image et l'affiche sur l'emplacement réservé au flux vidéo de l'ihm, ce slot s'occupe de:
  - ◆ Actualiser les informations de l'environnement:
    - Heure de l'acquisition de la nouvelle image
    - Temps de campagne effectif à la prise de la nouvelle image grâce à la méthode `getTempsCampagne()`
    - Dernière information du capteurs de distance grâce à la méthode `getTelemetrie()`

- Explications techniques

La classe Rov s'occupera de créer les relations avec les autres classes (manette, capteur, communicationRov, camera)



La classe active Camera hérite de QThread car il faut assurer l'acquisition périodique d'images dans l'ihm. L'acquisition nécessite de faire une boucle pour capturer les images ce qui bloquerait le thread principal de l'ihm.



Video4Linux permet la prise en charge des périphériques caméras :


```
$ sudo apt-get install libv4l-dev v4l-utils
```

L'utilitaire v4l2-ctl permet de contrôler les périphériques **Video4Linux** :

```
$ v4l2-ctl --list-device
USB 2.0 Camera: HD USB Camera (usb-0000:00:14.0-1):
    /dev/video0
    /dev/video1
```

La méthode creerNomCamera() permet de récupérer ce nom de périphérique

```
QString Camera::creerNomCamera(int numero)
{
	QString nom;

	if(numero >= 0)
	{
	nom = QString("/dev/video") + QString::number(numero);
	}
	return nom;
}
```

À l'inverse la méthode extraireNumero() permet de récupérer le numéro associé

```
int Camera::extraireNumero(QString nomCamera)
{
	int numero = -1;

	if(nomCamera.contains("/dev/video"))
	{
	QString n = nomCamera.right(nomCamera.indexOf("/dev/video"));
	bool ok;
	numero = n.toInt(&ok);
	if(ok)
		return numero;
	}
	return numero;
}
```

Le constructeur de la classe Camera est surchargé permettant d'être instancié soit par un numéro de caméra soit par un nom de caméra:

```
Camera(Rov *rov, QString nomCamera, int choixResolution=-1);
Camera(Rov *rov, int numero, int choixResolution=-1);
```

La classe QMediaRecorder fournit par Qt possède une méthode supportedResolutions() , qui retourne une liste de QSize correspondant à toutes les résolutions supportées par la caméra passée en paramètre du constructeur de QMediaRecorder.

```
QMediaRecorder *mediaRecorder = new QMediaRecorder(camera, this);
camera->load();
if(mediaRecorder->supportedResolutions().size() > 0)
{
    foreach (const QSize &resolution, mediaRecorder->supportedResolutions())
    {
        listeResolutionsCamera.push_back(resolution);
    }
}
```

La méthode `start()` de la classe QThread se charge de démarrer le Thread, pour ceci elle appellera la méthode `run()` présente dans la classe Camera.

```
void Camera::run()
{
      …

      cv::VideoCapture camera(numero);
      cv::Mat frame;

      …
}
```

Pour la gestion de la caméra la bibliothèque OpenCv est utilisée.
OpenCV (*Open Computer Vision*) est une bibliothèque libre d'analyse d'images et de vision par ordinateur en langage C/C++, initialement développée par Intel, spécialisée dans le traitement d'images en temps réel. Cette bibliothèque libre est distribuée sous licence BSD.

Pour installer les bibliothèques de développement d'**OpenCV sur linux**, il faudra faire :

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

Les fichiers d'en-têtes sont installées dans les répertoires /usr/include/opencv et /usr/include/opencv2

```
$ ls -l /usr/include/opencv
total 44
-rw-r--r-- 1 root root 2523 mai   12  2017 cvaux.h
-rw-r--r-- 1 root root 2374 mai   12  2017 cvaux.hpp
-rw-r--r-- 1 root root 3153 mai   12  2017 cv.h
-rw-r--r-- 1 root root 2649 mai   12  2017 cv.hpp
-rw-r--r-- 1 root root 2176 mai   12  2017 cvwimage.h
-rw-r--r-- 1 root root 2424 mai   12  2017 cxcore.h
-rw-r--r-- 1 root root 2443 mai   12  2017 cxcore.hpp
-rw-r--r-- 1 root root 2257 mai   12  2017 cxeigen.hpp
-rw-r--r-- 1 root root  129 mai   12  2017 cxmisc.h
-rw-r--r-- 1 root root 2226 mai   12  2017 highgui.h
-rw-r--r-- 1 root root 2145 mai   12  2017 ml.h

$ ls -l /usr/include/opencv2
...
```

API C++

En C++, **OpenCV** fournit une classe VideoCapture pour l'acquisition vidéo en provenance d'une caméra ou d'un fichier.

Pour réaliser une 'capture', l'API C++ met à disposition trois méthodes :

- bool VideoCapture::grab() qui réalise l'acquisition de la prochaine image (*frame*) du fichier vidéo ou de la caméra et renvoie vrai (true) en cas de succès.
- bool VideoCapture::retrieve(Mat& image, int channel=0) qui décode et renvoie l'image (*frame*) précédemment acquise (*grab*). Si il n'y a aucune image (caméra déconnectée, ou plus d'images dans le fichier vidéo), la fonction retourne faux (false).
- bool VideoCapture::read(Mat& image) qui regroupe les deux fonctions précédentes (cvGrabFrame() et cvRetrieveFrame()) en un seul appel ce qui la rend plus pratique. L'opérateur >> peut aussi être utilisé.

La méthode VideoCapture::read(Mat& image) est ici utilisée grâce à la surcharge de l'opérateur >>.

```
void Camera::acquerirImageVideo(cv::VideoCapture &camera, cv::Mat
&frame)
{
	camera >> frame;
}
```

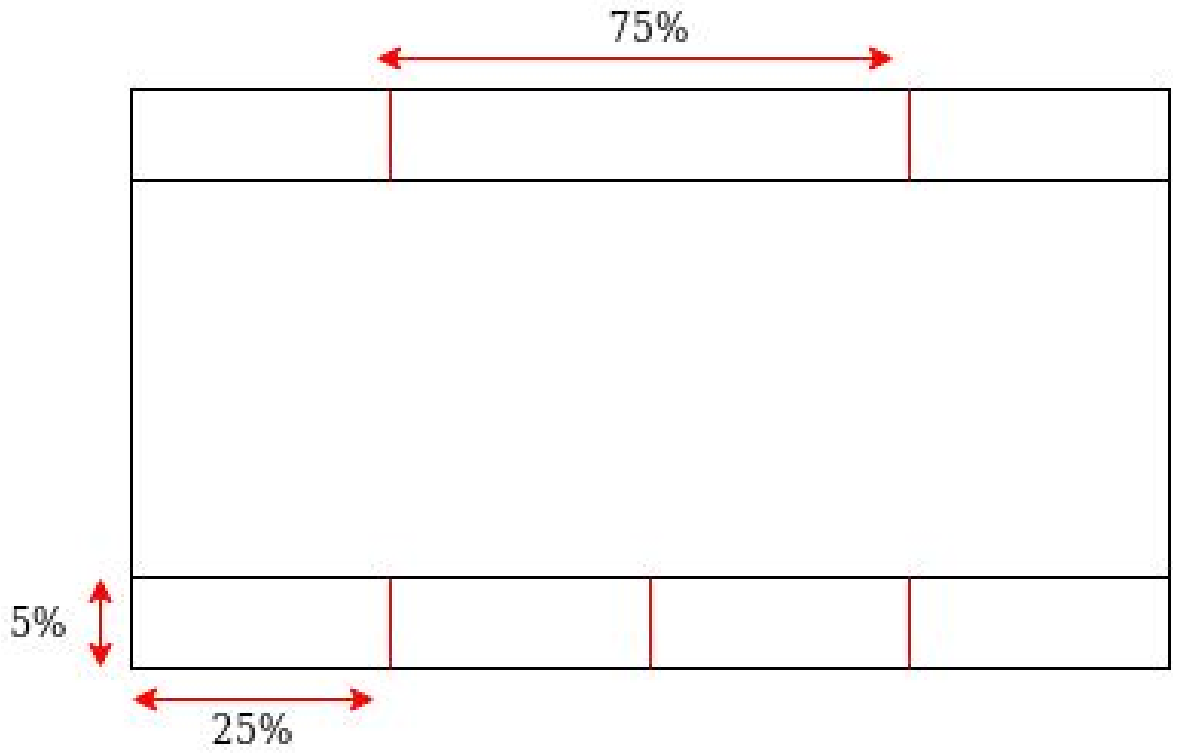
Cette ‘capture’ est transformée en QPixmap (classe de Qt permettant de gérer les images) et interceptée par l’ihmRov afin de l’afficher dans un emplacement réservé. Toutes ces images capturées contribuent à ce que l’on appelle le flux vidéo.

```
connect(camera, SIGNAL(nouvelleImage(QPixmap)), ihmRov, SLOT(afficherImage(QPixmap)));
```

Avant d’afficher cette nouvelle image la méthode `actualiserInformations()` est appelée, cette méthode permet une incrustation dans l’image de l’heure, de la durée et des informations liés au détecteur d’obstacle, grâce une classe QPainter de Qt:

Afin d'obtenir une incrustation optimale, il faut tenir compte de la taille de l'image qui change en fonction de la résolution de la caméra.

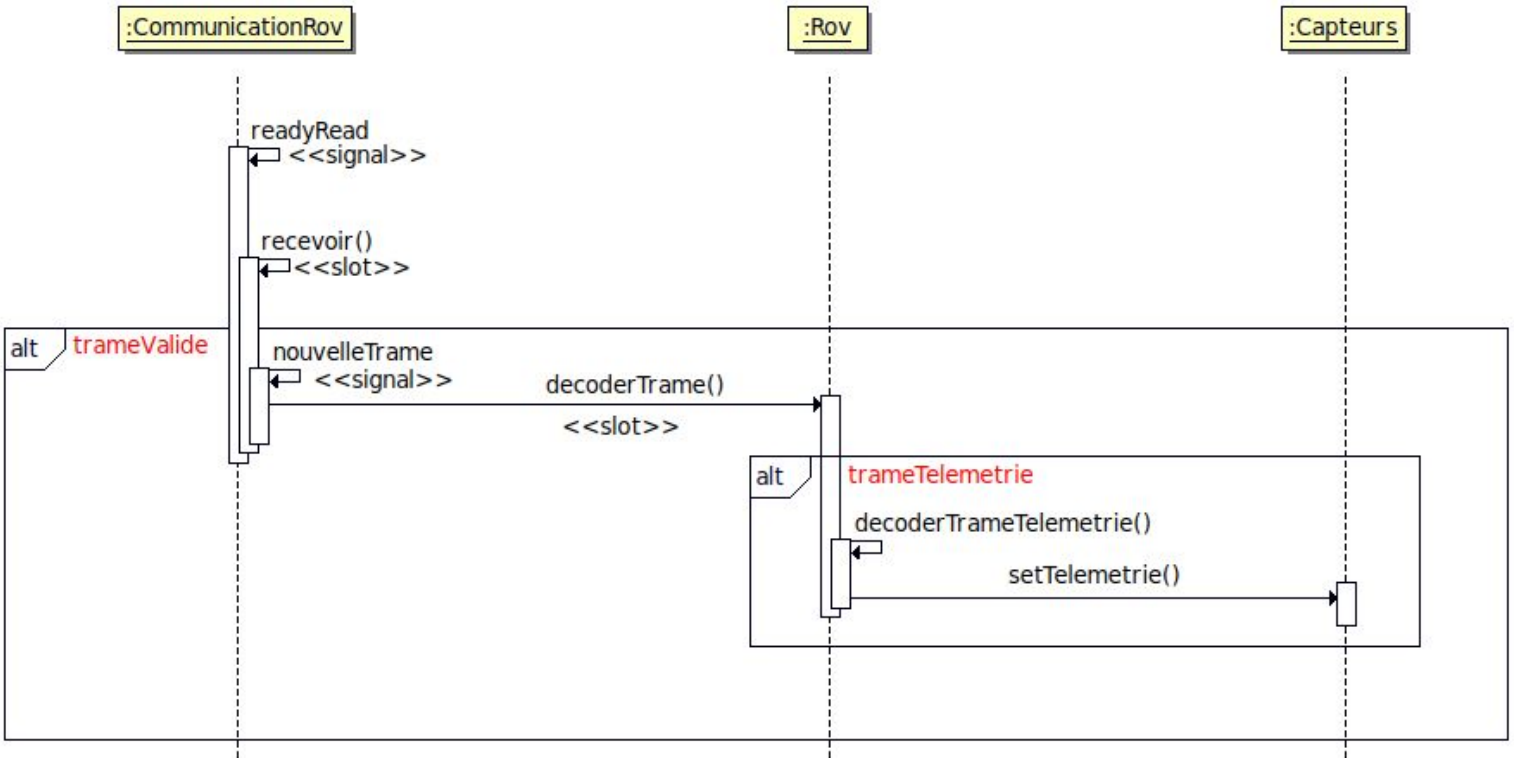
Afin de pallier cette contrainte un découpage en bandeaux est réalisé, suivant le schéma ci-dessous :




```
QRect bandeauBasGauche( 0, image.height()*0.95, image.width()*0.25, image.height()*0.05
);
QRect bandeauBasCentreGauche( image.width()*0.25, image.height()*0.95,
image.width()*0.25, image.height()*0.05 );
QRect bandeauBasCentreDroite( image.width()*0.50, image.height()*0.95,
image.width()*0.25, image.height()*0.05 );
QRect bandeauBasDroite( image.width()*0.75, image.height()*0.95, image.width()*0.25,
image.height()*0.05 );
```

4 bandeaux sont créés en bas de l'image avec une largeur de 5% et une longueur de 25% . Ces 4 bandeaux dépendant d'un pourcentage de la taille de l'image ne bougeront pas quelle que soit la résolution appliquée. Ne reste plus qu'à incruster les informations dans ces bandeaux.

❏ Scénario : recevoir les données de distance



- Déroulement du scénario

Objet communicationRov :

- ➔ À la réception du signal `readyRead()`, un slot `recevoir()` est déclenché
- ➔ Ce slot vérifie si la trame est valide (commence par le début de trame et finit par la fin de trame) avant d'envoyer un signal `nouvelleTrame()` contenant la trame reçue

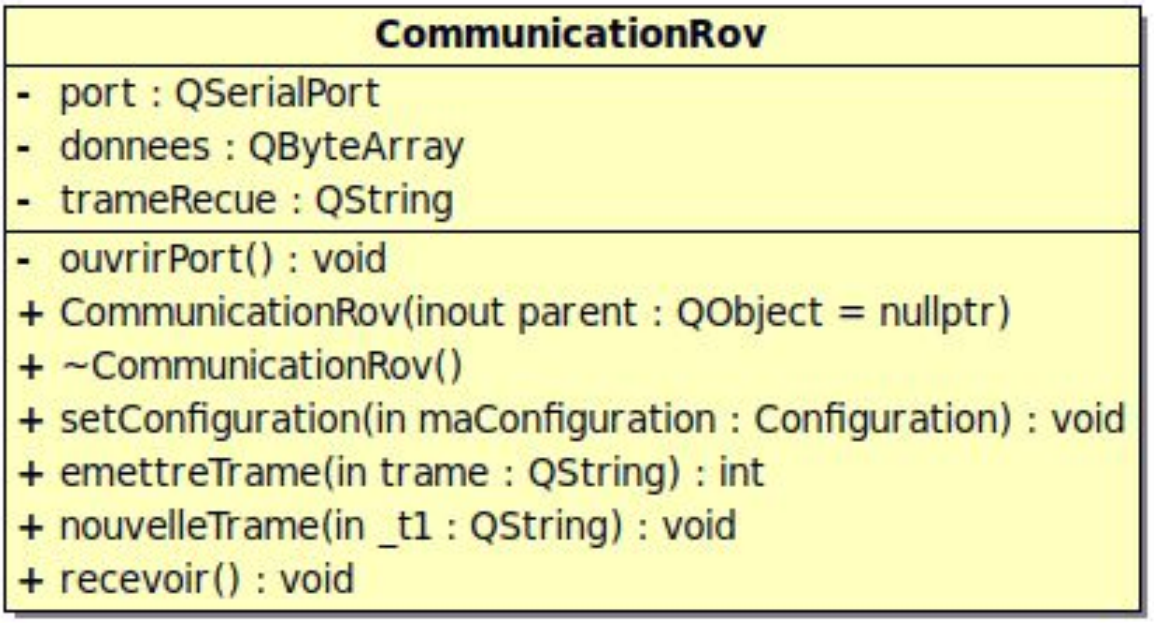
Objet rov:

- ➔ Le signal `nouvelleTrame()` est réceptionné par un slot `decoderTrame()`
- ➔ Ce slot vérifie de quel type de trame il s'agit et envoie le décodage correspondant

Objet capteurs:

- ➔ Les données décodées sont enregistrées grâce à la méthode `setTelemetrie()`

- Explications techniques



La classe CommunicationRov est une classe permettant de dialoguer avec le robot. La communication utilisée pour l'échange de données passe par une liaison USB/RS232. Pour configurer le port série on utilise une classe de Qt : QSerialPort.

Cette classe dispose des méthodes permettant de configurer et mettre en place la liaison série :

- setPortName() : nom du port
- setBaudRate() : vitesse de transmission
- setDataBits() : nombre de bits de données
- setStopBits() : nombre de bits de stop
- setParity() : active ou non la parité
- open() : ouvre le port série
- close() : ferme le port série

Pour la lecture des données, la classe QSerialPort hérite du signal de son parent QIOdevice

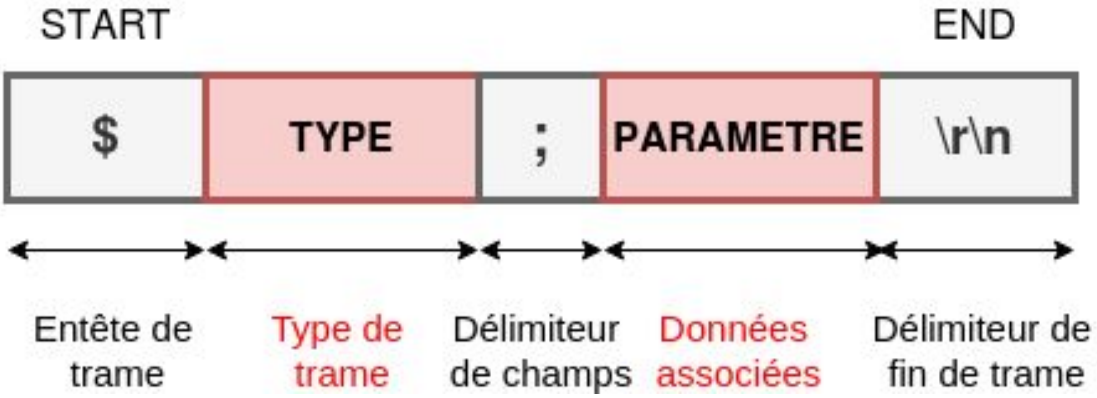
- readyRead()

Ce signal est émis chaque fois qu'une nouvelle donnée est disponible en lecture depuis le port virtuel.

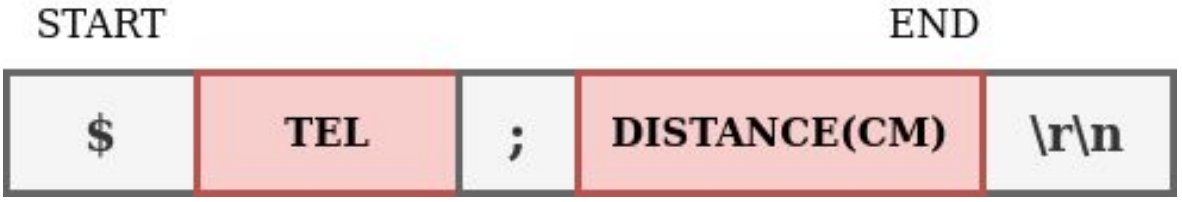
```
connect(communicationRov, SIGNAL(nouvelleTrame(QString)), this, SLOT(decoderTrame(QString)));
```

La connexion de ce signal au slot `decoderTrame()` nous permet de traiter les informations en provenance du robot

- ❏ Rappel protocol



Une trame est valide si elle respecte la structure du protocole. Pour le traitement des données en provenance du capteur de distance un trame type sera :



Le décodage de la trame est pris en charge par la méthode `decoderTrameTelemetrie()`

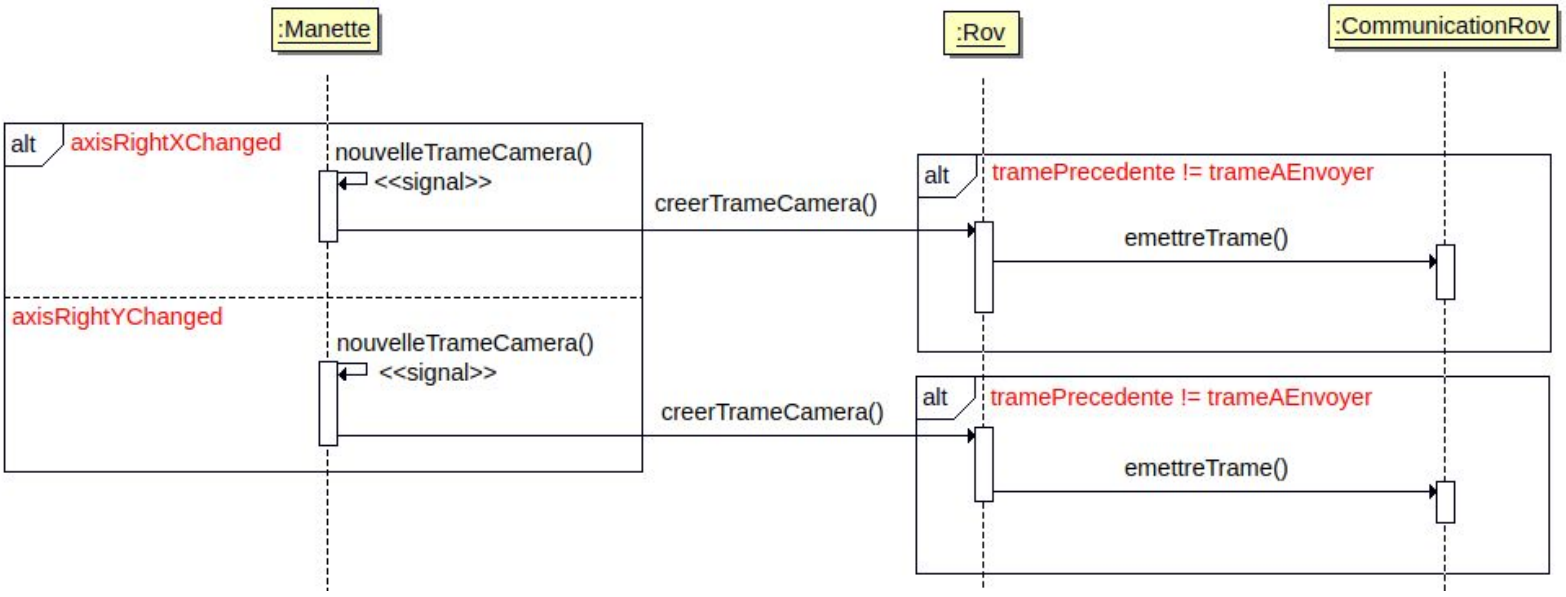
```
void Rov::decoderTrameTelemetrie(QString trameTelemetrie)
{
      capteurs->setTelemetrie(trameTelemetrie.section(";",1,1));
}
```

La classe QString de Qt offre une méthode section() qui prend en paramètre dans l'ordre:

- Le caractère sur lequel faire une découpe
- un entier correspondant à la position de départ
- un entier correspondant à la position de fin

La chaîne retournée se compose des champs du début à la fin inclusive de la position

❑ Scénario : piloter la caméra

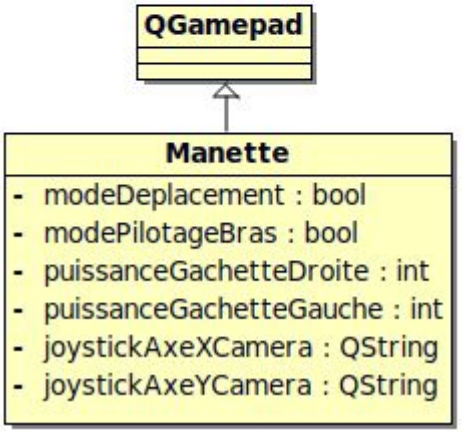


- Explications techniques

La classe manette hérite d'une classe de Qt : QGamepad. cette dernière permet de prendre en charge les différentes manettes connectées au système.

Le constructeur de la classe Manette prend en paramètre un entier correspondant au numéro de la manette connectée. Une liste des manettes détectées est disponible par une classe Qt :

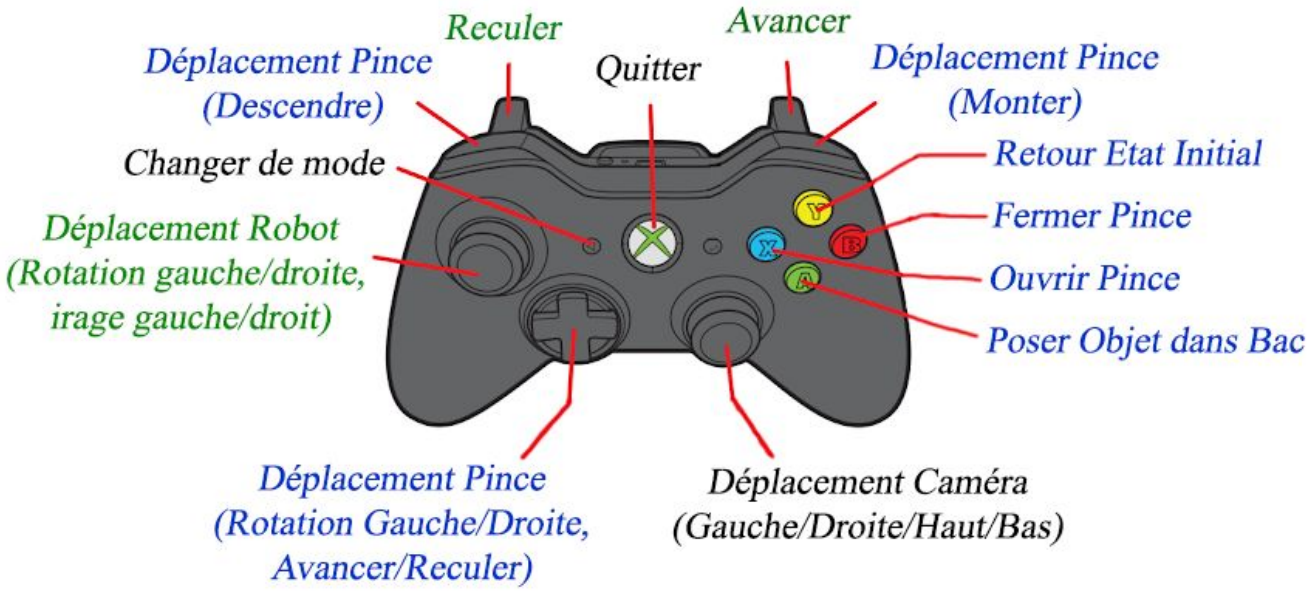
```
QGamepadManager::instance()->connectedGamepads();
```



La classe QGamepad offre des signaux qui s'interface avec les événements de la manette connectée.

Les signaux qui nous intéressent correspondant au pilotage de la caméra selon la configuration (ci-dessous) qui a été choisie sont:

- axisRightXChanged()
- axisRightYChanged()

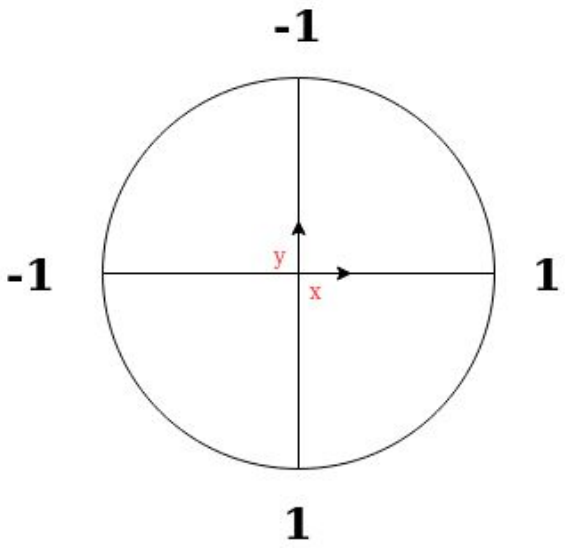


Ces signaux ont été connectés à des slots permettant la prise en charge de l'événement associé et envoyer la trame correspondante :

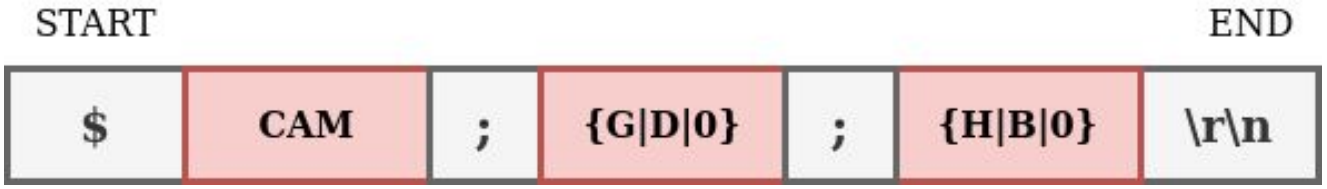
```
connect(manette, SIGNAL(axisRightXChanged(double)), manette,
SLOT(changerAxeXJoystickDroit(double)));

connect(manette, SIGNAL(axisRightYChanged(double)), manette,
SLOT(changerAxeYJoystickDroit(double)));
```

Ce schéma représente l'axe du joystick, les signaux précédents retourne un double suivant cette logique.
Par exemple lorsque le joystick est enfoncé sur la gauche le signal envoyé est :

```
axisRightXChanged(-1)
```



- ❏ Explication trame camera




```
$CAM;G;0\r\n
```

La trame ci-dessus permet d'envoyer l'ordre à la caméra de tourner sur la gauche, si l'on veut donner l'ordre à la caméra de s'arrêter de tourner à gauche la trame suivante doit être envoyée:

```
$CAM;0;0\r\n
```

Le mécanisme est le même pour tourner à droite, en haut ou en bas.

```
#define TAUX_VALIDITE_JOYSTICK 0.50

void Manette::changerAxeXJoystickDroit(double valeur)
{
	if(valeur >= TAUX_VALIDITE_JOYSTICK)
	joystickAxeXCamera = "D";
	else if(valeur <= -TAUX_VALIDITE_JOYSTICK)
	joystickAxeXCamera = "G";
	else
	joystickAxeXCamera = "0";

	emit nouvelleTrameCamera(joystickAxeXCamera, joystickAxeYCamera);
}
```

Le développement de ces ordres doit prendre en compte une particularité physique de la manette : le joystick étant très sensible, même au repos (sans action de l'utilisateur), des signaux sont envoyés constamment.

Un taux de validité est donc mis en place de l'ordre de 0.5 (La valeur maximale /2). Ceci nous évite l'envoi de trame non voulu.

```
void Rov::creerTrameCamera(QString axeX, QString axeY)
{
	QString trameCamera = DEBUT_TRAME_CAMERA ";" + axeX + ";" + axeY + "\r\n";

	if(tramePrecedenteCamera != trameCamera)
	{
	tramePrecedenteCamera = trameCamera;
	communicationRov->emettreTrame(trameCamera);
	}
}
```

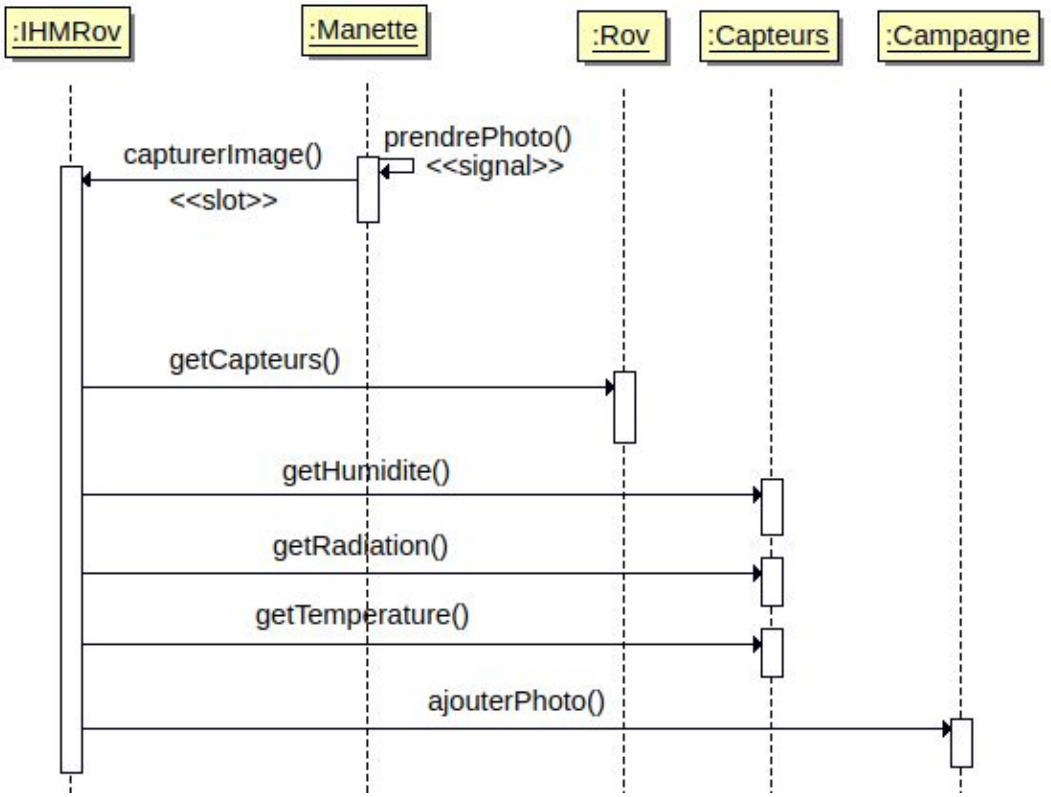
Le slot `creerTrameCamera()` récupère la nouvelle trame caméra destinée à être envoyée au rov et effectue une dernière vérification avant de l'envoyer.

SI l'on reprend toujours en compte la particularité physique ci-dessus, on s'aperçoit qu'une multitude de trames 0 sont envoyées

```
$CAM;0;0\r\n
```

La condition `tramePrecedenteCamera != trameCamera` permet donc de n'envoyer uniquement que le trames désirées.

❑ Scénario : Prendre une photo



- Déroulement du scénario

Objet manette :

➔ À la réception de l'événement correspondant à la prise d'une photo, la manette envoi un signal prendrePhoto()

Objet ihmRov :

➔ Ce dernier signal est réceptionné par un slot de ihmRov `capturerImage()`

➔ L'objet capteurs contenant les dernières informations issues des capteurs du robot est récupéré

➔ Les informations issues des capteurs du robot sont extraites
➔ La photo est ajoutée dans l'albumPhoto de l'objet campagne

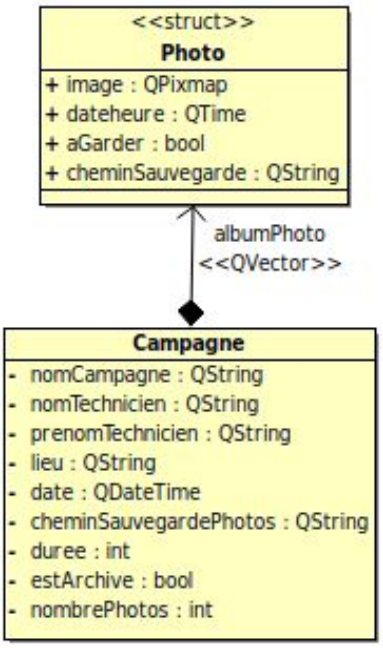
- Explications techniques

La prise de photo est gérée par la connexion :

```
connect(manette, SIGNAL(buttonR3Changed(bool)), ihmRov, SLOT(capturerImage(bool)));
```

Une structure Photo a été mis en place afin d'avoir des informations complémentaires :

```
struct Photo
{
	QPixmap image;
	QTime dateheure;
	bool aGarder;
	QString cheminSauvegarde;
};
```



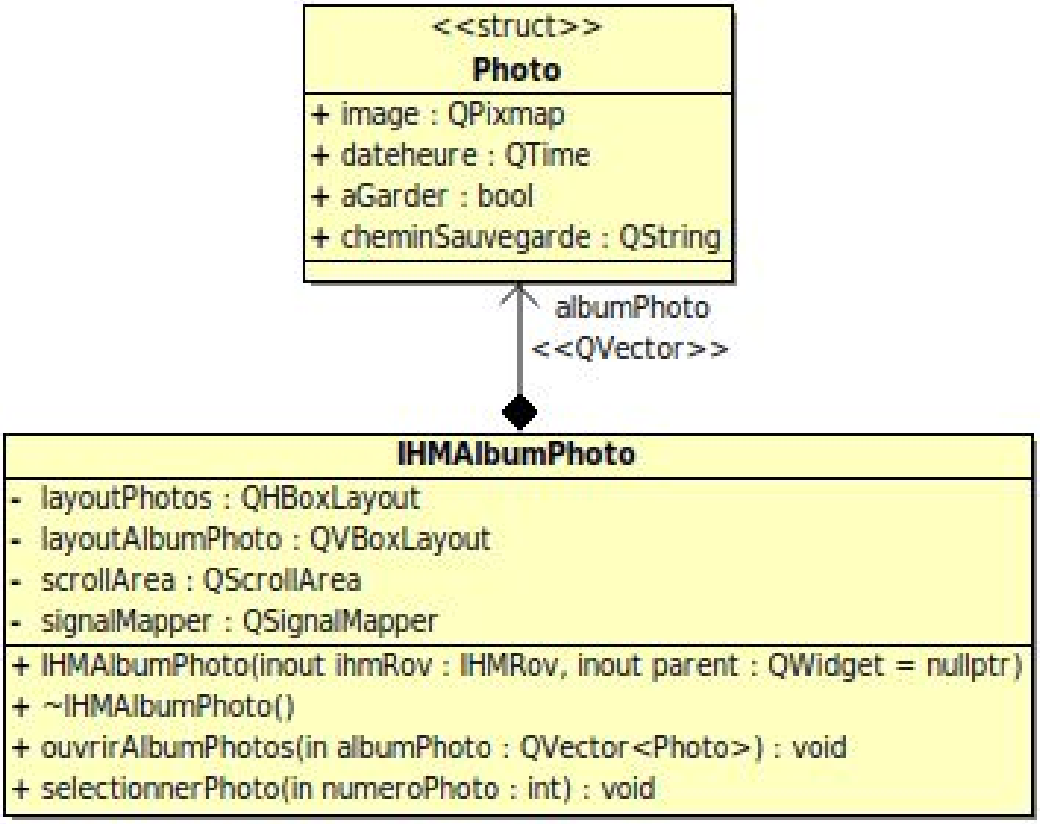
Une photo est caractérisée par :

- Son image (ici transformé en QPixmap)
- La date/heure de la prise
- Un booléen permettant à l'utilisateur de la garder ou non
- Son chemin de sauvegarde

La photo prise est ajoutée au conteneur de photos :

```
void Campagne::ajouterPhoto(Photo &photo)
{
	albumPhoto.push_back(photo);
}
```

Ce conteneur de photos est pris en charge par la classe IHMAlbumPhoto

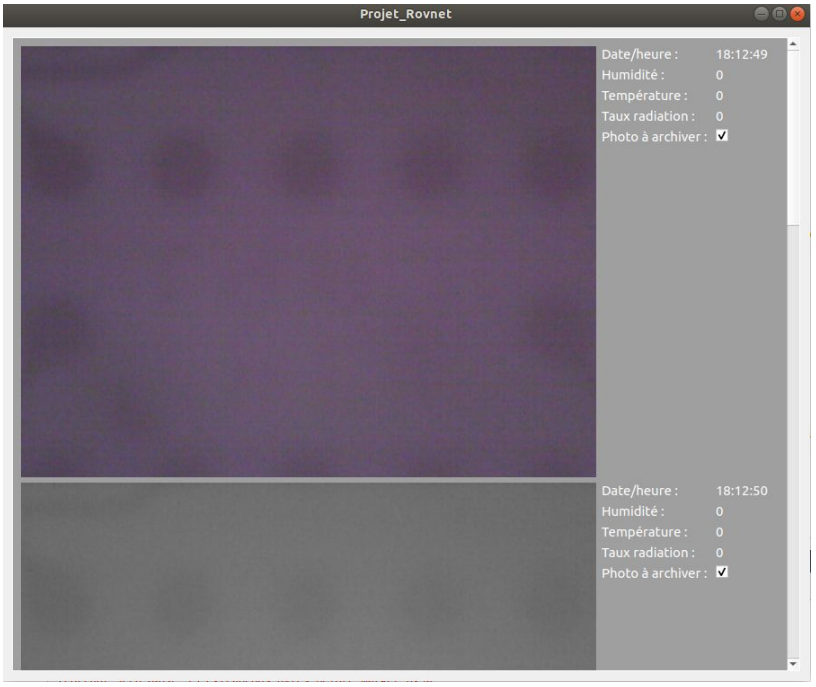


La classe IHMAlbumPhoto est une Interface Homme Machine permettant d'avoir une vision visuelle sur les photos prises par le technicien durant la campagne.

La méthode `ouvrirAlbumPhotos(QVector<Photo> albumPhoto)` prend en paramètre le conteneur de photos et rempli l'IHM des différentes photos et informations :

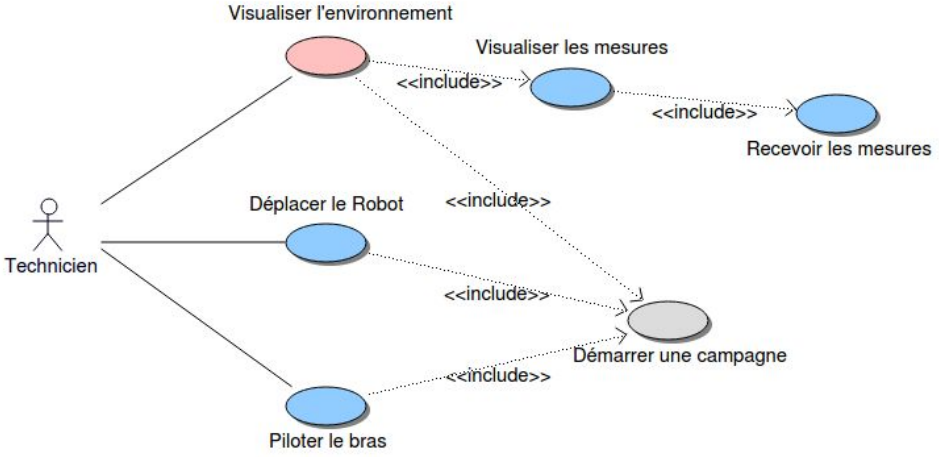
Cette IHM contient une liste déroulante de toute les photos présente dans le conteneur de Photo.

Une case à cocher (*checkbox*) permet de modifier le membre aGarder de la photo.



Partie Personnelle TENAILLE

Diagramme de cas d'utilisation personnel



Le technicien doit pouvoir :

- Visualiser les mesures que capture le robot ce qui inclut de les recevoir
- Piloter le bras articulé
- Déplacer le robot

Ce qui inclut pour tous ces cas de démarrer une campagne

Planification des tâches personnel

itération 1 :

Recevoir et afficher les données des capteurs

Prise en charge de la manette et fabrication des trames de déplacement du robot

itération 2 :

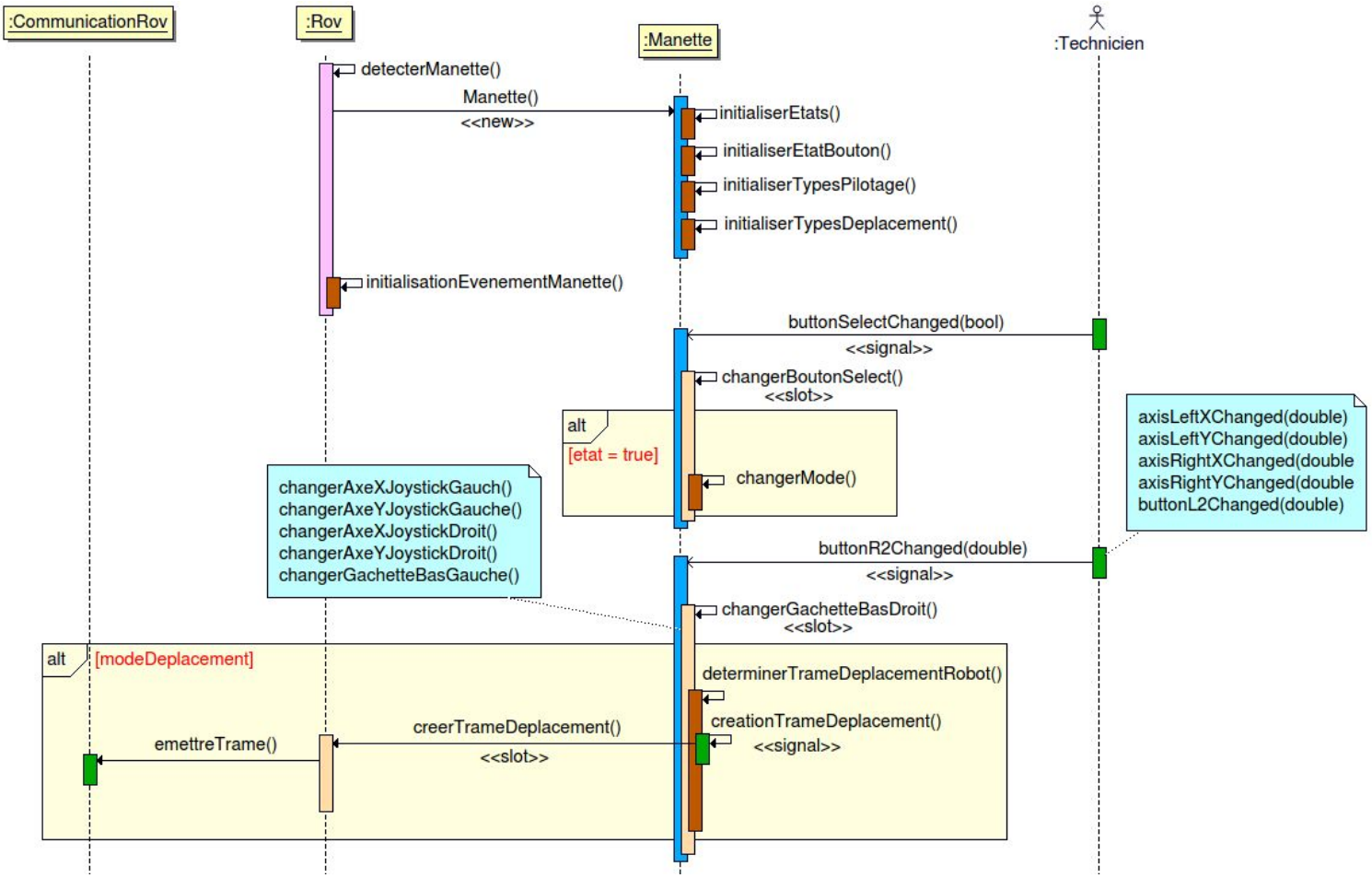
Prise en charge de la manette et fabrication des trames de pilotage du bras

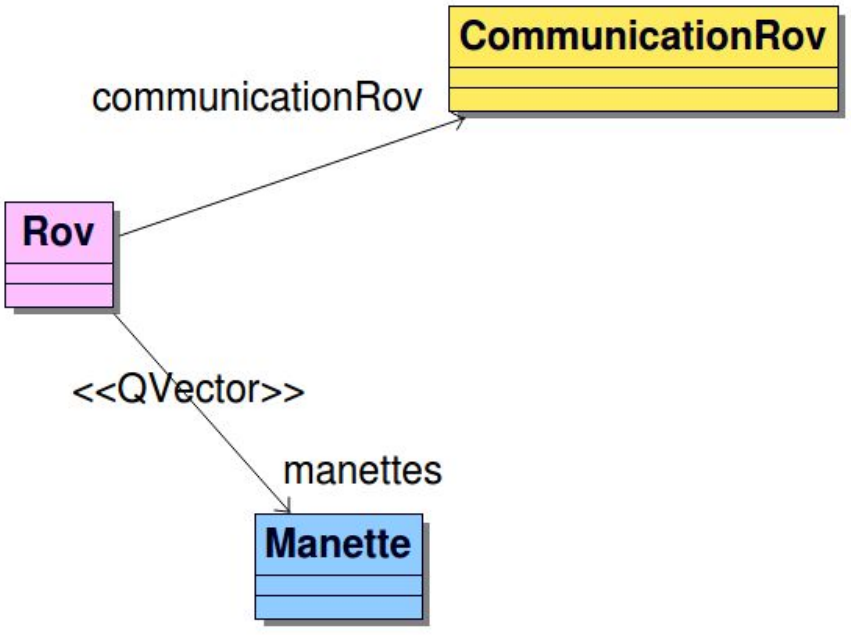
itération 3 :

Archivage des données

Réaliser IHM configuration campagne

Scénario : déplacer le robot





Déroulement :

- Exécution de la méthode detecterManette()
- Création d'un objet de la classe Manette
- Exécution des méthode initialiserEtats(), initialiserEtatBouton(), initialiserTypesPilotage(), initialiserTypesDeplacement() de l'objet de la classe Manette
- Exécution de la méthode initialisationEvenementManette() de l'objet de la classe Rov

- Envoi du signal buttonSelectChanged() de la part du technicien avec la manette
- Exécution de la méthode slot changerBoutonSelect() de l'objet de la classe Manette
- Test l'état du signal
- Exécute la méthode changerMode()

- Envoi d'un signal de déplacement de la part du technicien avec la manette
- Exécution d'une méthode slot de déplacement de l'objet de la classe Manette

Si le modeDeplacement est vrai on exécute:

- Exécute la méthode determinerTrameDeplacementRobot()
- Envoi le signal creationTrameDeplacement() avec les valeurs des déplacements
- Exécute la méthode slot creerTramedeplacement() de l'objet de la classe Rov
- Exécute la méthode emettreTrame() de l'objet de la classe CommunicationRov

Explications techniques

Pour chaque signal envoyé par le technicien, la manette émet un signal unique associé à chaque bouton, au niveau logiciel on créer une connexion entre le signal et la méthode slot concernée :

```
connect(manette, SIGNAL(axisLeftXChanged(double)), manette,
SLOT(changerAxeXJoystickGauche(double)));
  connect(manette, SIGNAL(axisLeftYChanged(double)), manette,
SLOT(changerAxeYJoystickGauche(double)));
  connect(manette, SIGNAL(axisRightXChanged(double)), manette,
SLOT(changerAxeXJoystickDroit(double)));
  connect(manette, SIGNAL(axisRightYChanged(double)), manette,
SLOT(changerAxeYJoystickDroit(double)));
  connect(manette, SIGNAL(buttonL2Changed(double)), manette,
SLOT(changerGachetteBasGauche(double)));
  connect(manette, SIGNAL(buttonR2Changed(double)), manette,
SLOT(changerGachetteBasDroit(double)));
  connect(manette, SIGNAL(buttonSelectChanged(bool)), manette,
SLOT(changerBoutonSelect(bool)));
```

Lorsque le technicien appuie sur le bouton R2 le code du slot changerGachetteBasDroit() est exécuté.

```
void Manette::changerGachetteBasDroit(double valeur)
{
  puissanceGachetteDroite = int(valeur*100);
  if (valeur > 0)
     maManetteDeplacement.gachetteBasDroit = true;
  else
     maManetteDeplacement.gachetteBasDroit = false;
  if(modeDeplacement)
     determinerTrameDeplacementRobot();
}
```

Cette méthode va affecter la valeur donnée par la gâchette, compris entre 0 et 1, à l'attribut puissanceGachetteDroite en la multipliant par 100 pour conserver la précision à 10^{-2} de la gâchette. Puis si la valeur est supérieure à 0 on considère la gâchette comme appuyé et on modifie l'état de la variable gachetteBasDroite par l'état actuel du bouton (*true* ou *false*) dans la structure définissant l'état actuel de la manette pour le déplacement du robot (maManetteDeplacement). On vérifie ensuite si on est en mode déplacement et si c'est le cas on exécute la méthode determinerTrameDeplacement().

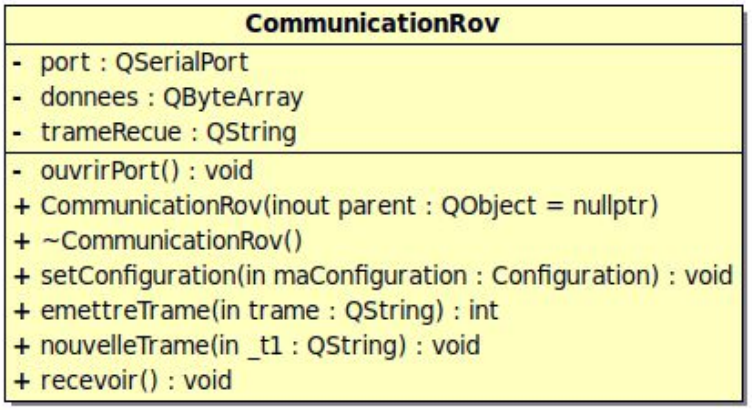
```
void Manette::determinerTrameDeplacementRobot()
{
  if(maManetteDeplacement == enAvantCas1 || maManetteDeplacement == enAvantCas2 ||
maManetteDeplacement == enAvantCas3)
     emit creationTrameDeplacement('A', puissanceGachetteDroite, '0');
  else if(maManetteDeplacement == enArriereCas1 || maManetteDeplacement ==
enArriereCas2 || maManetteDeplacement == enArriereCas3)
     emit creationTrameDeplacement('R', puissanceGachetteGauche, '0');
  else if(maManetteDeplacement == rotationAGauche)
     emit creationTrameDeplacement('0', 100, 'G');
  else if(maManetteDeplacement == rotationAGaucheDoucementCas1 ||
maManetteDeplacement == rotationAGaucheDoucementCas2)
     emit creationTrameDeplacement('0', int(100 * REDUCTION_VITESSE), 'G');
  else if(maManetteDeplacement == rotationADroite)
     emit creationTrameDeplacement('0', 100, 'D');
  else if(maManetteDeplacement == rotationADroiteDoucementCas1 ||
maManetteDeplacement == rotationADroiteDoucementCas2)
     emit creationTrameDeplacement('0', int(100 * REDUCTION_VITESSE), 'D');
  else if(maManetteDeplacement == virageAvantAGauche)
     emit creationTrameDeplacement('A', puissanceGachetteDroite, 'G');
  else if(maManetteDeplacement == virageAvantAGaucheDoucementCas1 ||
maManetteDeplacement == virageAvantAGaucheDoucementCas2)
     emit creationTrameDeplacement('A', int(puissanceGachetteDroite *
REDUCTION_VITESSE), 'G');
  else if(maManetteDeplacement == virageAvantADroite)
     emit creationTrameDeplacement('A', puissanceGachetteDroite, 'D');
```

```
  else if(maManetteDeplacement == virageAvantADroiteDoucementCas1 ||
maManetteDeplacement == virageAvantADroiteDoucementCas2)
      emit creationTrameDeplacement('A', int(puissanceGachetteDroite *
REDUCTION_VITESSE), 'D');
  else if(maManetteDeplacement == virageArriereAGauche)
      emit creationTrameDeplacement('R', puissanceGachetteGauche, 'G');
  else if(maManetteDeplacement == virageArriereAGaucheDoucementCas1 ||
maManetteDeplacement == virageArriereAGaucheDoucementCas2)
      emit creationTrameDeplacement('R', int(puissanceGachetteGauche *
REDUCTION_VITESSE), 'G');
  else if(maManetteDeplacement == virageArriereADroite)
      emit creationTrameDeplacement('R', puissanceGachetteGauche, 'D');
  else if(maManetteDeplacement == virageArriereADroiteDoucementCas1 ||
maManetteDeplacement == virageArriereADroiteDoucementCas2)
      emit creationTrameDeplacement('R', int(puissanceGachetteGauche *
REDUCTION_VITESSE), 'D');
  else
      emit creationTrameDeplacement('0', 0, '0');
}
```

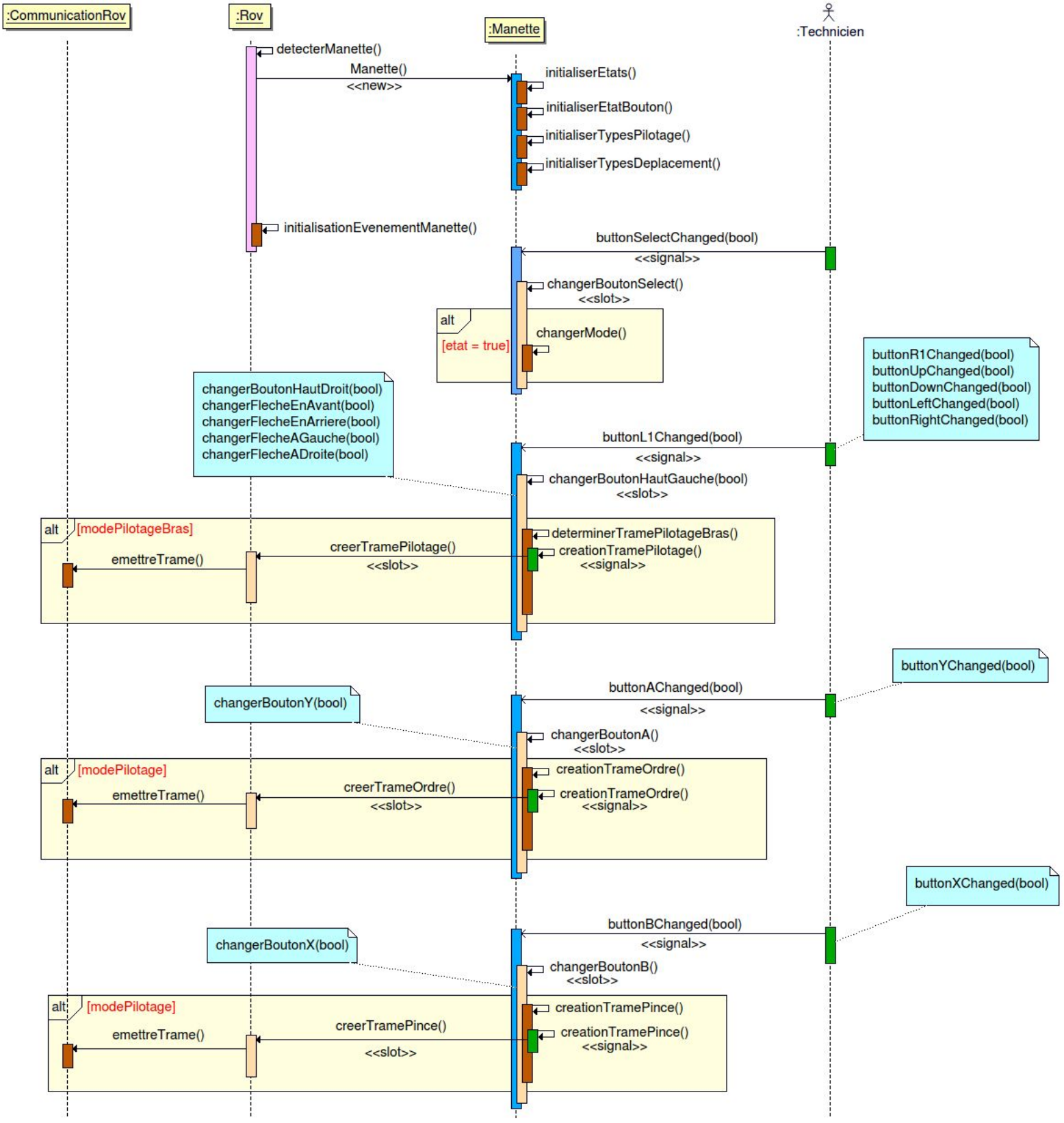
Dans la méthode determinerTrameDeplacement() on va comparer la structure actuelle de la manette pour le déplacement du robot par les différents cas possibles et si la structure ne correspond avec aucune structure de déplacement on considère qu'il s'agit d'un ordre d'arrêt du véhicule.
La classe rov prend le relai avec la méthode slot creerTrameDeplacement().

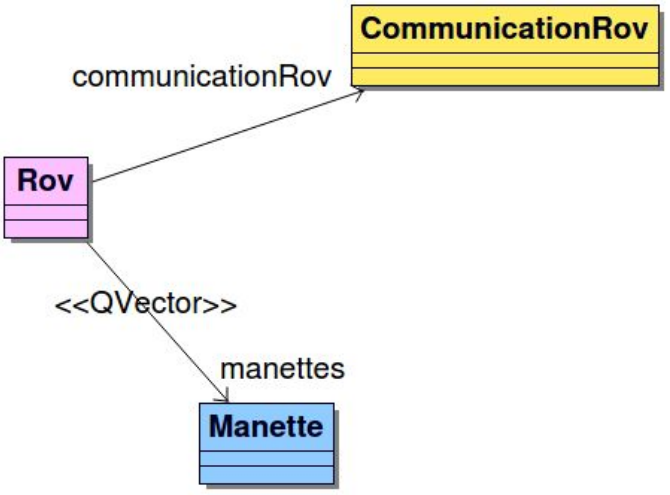
```
void Rov::creerTrameDeplacement(char deplacementAxeX, int puissance, char
deplacementAxeY)
{
  trameDeplacement = DEBUT_TRAME_DEPLACEMENT ";" + QString(deplacementAxeX)
+ ";" + QString::number(puissance) + ";" + QString(deplacementAxeY) + "\r\n";
  if(tramePrecedenteDeplacement != trameDeplacement)
  {
      qDebug() << Q_FUNC_INFO << "trameDeplacement :" << trameDeplacement;
      tramePrecedenteDeplacement = trameDeplacement;
      communicationRov->emettreTrame(trameDeplacement);
  }
}
```

Celle ci construit la trame avec les éléments reçus et l'envoie si la trame est différente de la précédente, après on fait donc appel à la méthode emmetreTrame() de l'objet communicationRov de la classe CommunicationRov.



Scénario : piloter le bras





Déroulement :

- Exécution de la méthode detecterManette()
- Création d'un objet de la classe Manette
- Exécution des méthode initialiserEtats(), initialiserEtatBouton(), initialiserTypesPilotage(), initialiserTypesDeplacement() de l'objet de la classe Manette
- Exécution de la méthode initialisationEvenementManette() de l'objet de la classe Rov

- Envoi du signal buttonSelectChanged() de la part du technicien avec la manette
- Exécution de la méthode slot changerBoutonSelect() de l'objet de la classe Manette
- Test l'état du signal
- Exécute la méthode changerMode()

- Envoi d'un signal de pilotage du bras de la part du technicien avec la manette
- Exécution d'une méthode slot de pilotage du bras de l'objet de la classe Manette

Si le modePilotageBras est vrai on exécute alors

- Exécute la méthode determinerTramePilotageBras()
- Envoi le signal creationTramePilotageBras() avec les valeurs du pilotage du bras
- Exécute la méthode slot creerTramePilotage() de l'objet de la classe Rov
- Exécute la méthode emettreTrame() de l'objet de la classe CommunicationRov

- Envoi d'un signal d'ordre de la part du technicien avec la manette
- Exécution d'une méthode slot de pilotage du bras de l'objet de la classe Manette

Si le modePilotage est vrai on exécute alors

- Exécute la méthode determinerTrameOrdre()
- Envoi le signal creationTrameOrdre() avec les valeurs du pilotage du bras
- Exécute la méthode slot creerTrameOrdre() de l'objet de la classe Rov
- Exécute la méthode emettreTrame() de l'objet de la classe CommunicationRov

- Envoi d'un signal de pilotage de la pince de la part du technicien avec la manette
- Exécution d'une méthode slot de pilotage du bras de l'objet de la classe Manette

Si le modePilotage est vrai on exécute alors

- Exécute la méthode determinerTramePince()
- Envoi le signal creationTramePince() avec les valeurs du pilotage du bras
- Exécute la méthode slot creerTramePince() de l'objet de la classe Rov
- Exécute la méthode emettreTrame() de l'objet de la classe CommunicationRov

Explications techniques

Pour chaque signal envoyé par le technicien, la manette émet un signal unique associé à chaque bouton, au niveau logiciel on créer une connexion entre le signal et la méthode slot concernée :

```
  connect(manette, SIGNAL(buttonL1Changed(bool)), manette,
SLOT(changerBoutonHautGauche(bool)));
  connect(manette, SIGNAL(buttonR1Changed(bool)), manette,
SLOT(changerBoutonHautDroit(bool)));
  connect(manette, SIGNAL(buttonUpChanged(bool)), manette,
SLOT(changerFlecheEnAvant(bool)));
  connect(manette, SIGNAL(buttonDownChanged(bool)), manette,
SLOT(changerFlecheEnArriere(bool)));
  connect(manette, SIGNAL(buttonLeftChanged(bool)), manette,
SLOT(changerFlecheAGauche(bool)));
  connect(manette, SIGNAL(buttonRightChanged(bool)), manette,
SLOT(changerFlecheADroite(bool)));
  connect(manette, SIGNAL(buttonAChanged(bool)), manette,
SLOT(changerBoutonA(bool)));
  connect(manette, SIGNAL(buttonBChanged(bool)), manette,
SLOT(changerBoutonB(bool)));
  connect(manette, SIGNAL(buttonXChanged(bool)), manette,
SLOT(changerBoutonX(bool)));
  connect(manette, SIGNAL(buttonYChanged(bool)), manette,
SLOT(changerBoutonY(bool)));
  connect(manette, SIGNAL(buttonSelectChanged(bool)), manette,
SLOT(changerBoutonSelect(bool)));
```

Lorsque le technicien appuie sur le bouton L1 le code du slot changerBoutonHautGauche() est exécuté :

```
void Manette::changerBoutonHautGauche(bool etat)
{
  maManettePilotage.boutonHautGauche = etat;
  if(modePilotageBras)
     determinerTramePilotageBras();
}
```

On change donc l'état de la variable boutonHautGauche par l'état actuel du bouton (*true* ou *false*) dans la structure définissant l'état actuel de la manette pour le pilotage du bras (maManettePilotage).
Puis on vérifie l'état du mode (modePilotageBras), si il est vrai on exécute la méthode derterminerTramePilotageBras().

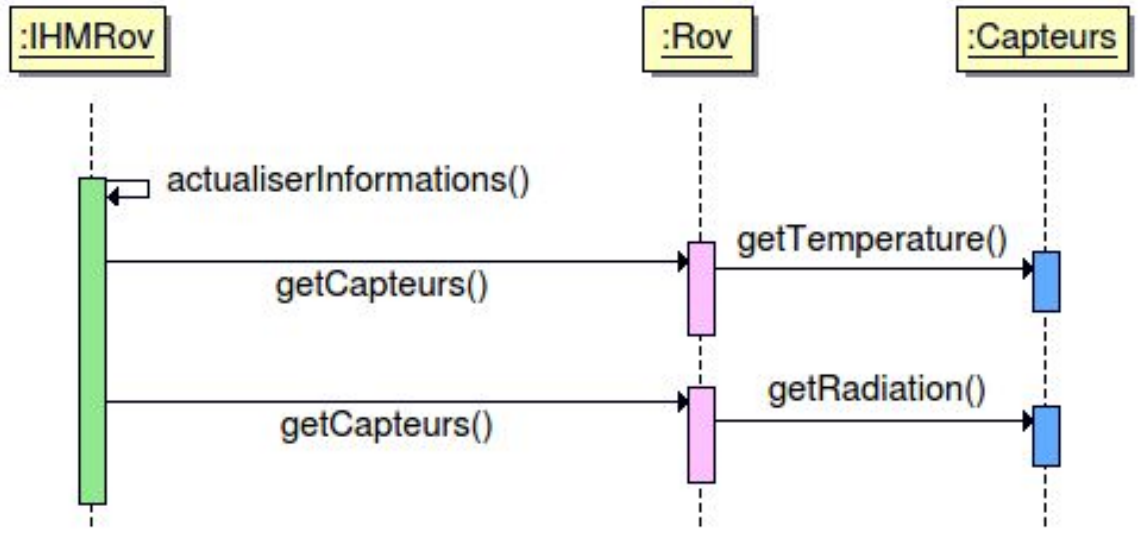
```
void Manette::determinerTramePilotageBras()
{
  if(maManettePilotage == avance)
     emit creationTramePilotage(AVANCER);
  else if(maManettePilotage == recule)
     emit creationTramePilotage(RECULER);
  else if(maManettePilotage == gauche)
     emit creationTramePilotage(GAUCHE);
  else if(maManettePilotage == droite)
     emit creationTramePilotage(DROITE);
  else if(maManettePilotage == monte)
     emit creationTramePilotage(MONTER);
  else if(maManettePilotage == descend)
     emit creationTramePilotage(DESCENDRE);
  else
     emit creationTramePilotage(IMMOBILE);
}
```

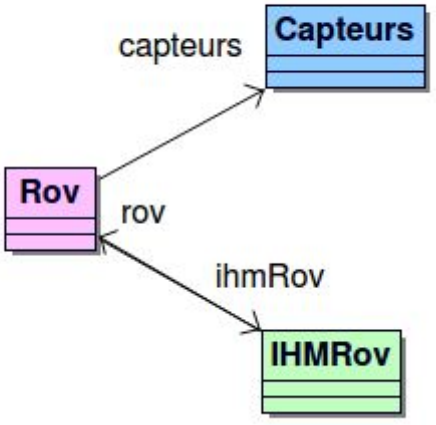
Cette méthode vérifie si la structure actuelle de la manette correspond avec une structure attendu pour un déplacement (avancer, reculer, gauche, droite, monter, descendre), si ce n'est pas le cas (exécution de 2 mouvements simultanément ou retour en position *false* du bouton) alors un signal avec la trame d'arrêt de mouvement (immobile) est envoyé.
La classe rov prend le relai avec la méthode slot creerTramePilotage().

```
void Rov::creerTramePilotage(QString deplacement)
{
  tramePilotage = DEBUT_TRAME_PILOTAGE ";" + deplacement + "\r\n";
  if(tramePrecedentePilotage != tramePilotage && tramePince ==
TRAME_PINCE_IMMOBILE)
  {
     qDebug() << Q_FUNC_INFO << "tramePilotage :" << tramePilotage;
     tramePrecedentePilotage = tramePilotage;
     communicationRov->emettreTrame(tramePilotage);
  }
}
```

Celle ci construit la trame avec les éléments reçus et l'envoie si la trame est différente de la précédente et si aucune trame de la pince est en cours, après on fait donc appel à la méthode emettreTrame() de l'objet communicationRov de la classe CommunicationRov.

Scénario : VisualiserMesures





Déroulement

- Exécution la méthode actualiserInformations() de l'objet de la classe IHMRov
- Exécution de la méthode getCateurs() de l'objet de la classe Rov
- Exécution de la méthode getTemperature() de l'objet de la classe Capteurs
- Exécution de la méthode getCateurs() de l'objet de la classe Rov
- Exécution de la méthode getRadiation() de l'objet de la classe Capteurs

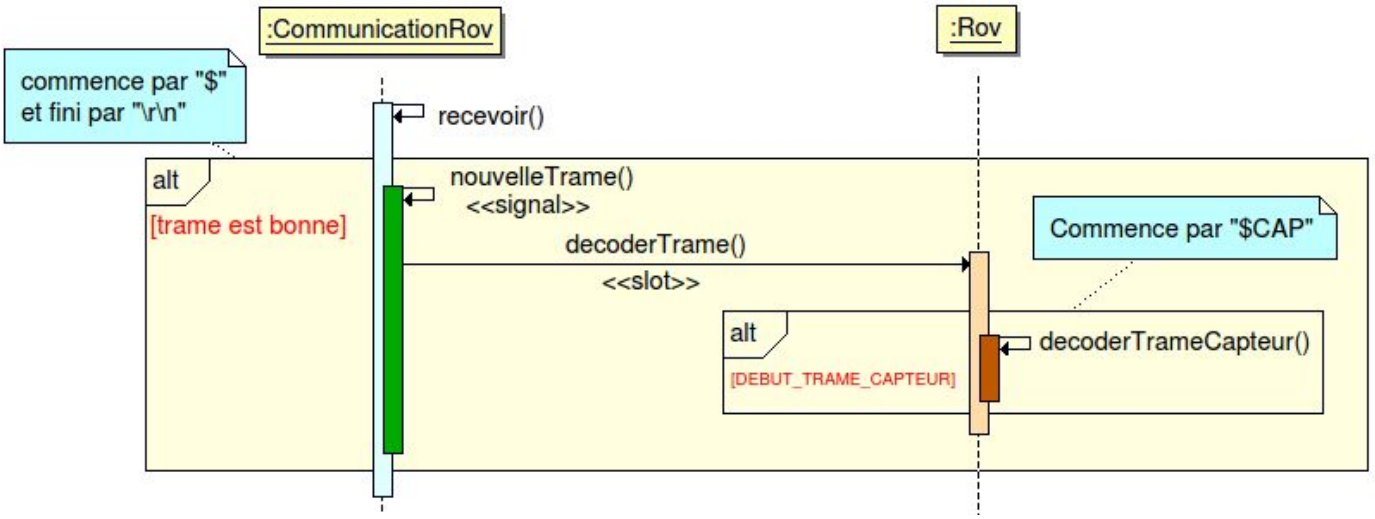
Explications techniques

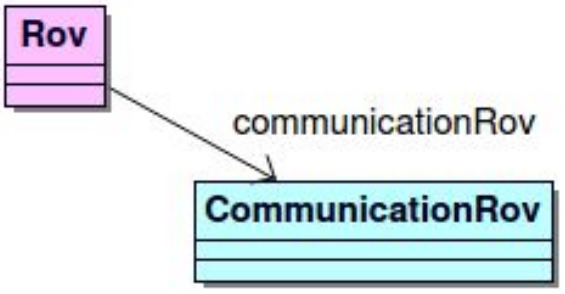
La méthode actualiserInformation() est appelée depuis la méthode actualiserImage() et dont l'appel est déclenché par le thread de la Caméra.

```
void IHMRov::actualiserInformations(QPixmap &image)
{
  QPainter p(&image);
  p.setPen(Qt::darkRed);
  p.setFont(QFont("Arial", 20));

...

  QImage logoTemperature(qApp->applicationDirPath() + "/images/logo_temperature.png");
  p.drawImage(5, 430, logoTemperature.scaled(20,40));
  p.drawText(35, 460, rov->getCapteurs()->getTemperature() + " °C");

  QImage logoRadiation(qApp->applicationDirPath() + "/images/logo_radiation.png");
  p.drawImage(160, 430, logoRadiation.scaled(40,40));
  p.drawText(210, 460, rov->getCapteurs()->getRadiation() + " uS/h");

  QImage logoObstacle(qApp->applicationDirPath() + "/images/logo_telemetrie.png");
  p.drawImage(400, 430, logoObstacle.scaled(40,40));
  p.drawText(450, 460, "Obstacle : " + rov->getCapteurs()->getTelemetrie() + " cm");

  p.end();
}
```

Dans cette méthode on va incruster dans l'image du flux vidéo la valeur de la mesure ainsi que son unité. Pour cela, on va donc utiliser une classe Qt QPainter qui va permettre de "dessiner" sur l'image du flux en utilisant les méthodes drawImage() pour les images et drawText() pour le texte. Dans ces méthodes, il faut fournir les coordonnées en pixel et un contenu (image ou texte).

Scénario : RecevoirMesures





Déroulement

- Exécution de la méthode recevoir() de l'objet de la classe CommunicationRov

Si la trame commence par "$" et fini par "\r\n" alors

- Envoi du signal nouvelleTrame() avec le contenu de la trame
- Exécution de la méthode slot decoderTrame() de l'objet de la classe Rov

  Si la trame commence par "$CAP" alors
  - Exécution de la méthode decoderTrameCapteur() de l'objet de la classe Rov

Explications techniques

La méthode recevoir() est appelée lorsque le signal readyRead() est envoyé :

```
void CommunicationRov::recevoir()
{
  while(port->bytesAvailable())
  {
     donnees += port->readAll();
     qDebug() << Q_FUNC_INFO << "bytesAvailable" << port->bytesAvailable() <<
"donnees" << donnees;
  }

  if(donnees.startsWith("$") && donnees.endsWith("\r\n"))
  {
     trameRecue = QString(donnees.data());
     qDebug() << Q_FUNC_INFO << "trameRecue" << trameRecue;
     emit nouvelleTrame(trameRecue);
     donnees.clear();
  }
}
```

La méthode va alors lire les données puis tester si la trame est valide (commence par “$” et fini par “\r\n” selon le protocole) et va envoyer le signal nouvelletrame() puis “vider” donnees afin d’assurer une prochaine réception.

```
connect(communicationRov, SIGNAL(nouvelleTrame(QString)), this,
SLOT(decoderTrame(QString)));
```

Le signal est relié à la méthode slot decoderTrame().

```
void Rov::decoderTrame(QString trame)
{
  QString trameTelemetrie, trameCapteur;

  QStringList trames = trame.split("\r\n");
  qDebug() << Q_FUNC_INFO << "trame" << trames.size();
  for(int i = 0; i < trames.size(); i++)
  {
     if(!trames[i].isEmpty())
     {
        if(trames[i].startsWith(DEBUT_TRAME_TELEMETRIE))
        {
           trameTelemetrie = trames[i];
           decoderTrameTelemetrie(trameTelemetrie);
```

```
        }
        else if(trames[i].startsWith(DEBUT_TRAME_CAPTEUR))
        {
            trameCapteur = trames[i];
            decoderTrameCapteur(trameCapteur);
        }
    }
    else
    {
        qDebug() << Q_FUNC_INFO << "Trame inconnue !";
    }
  }
}
```

Cette méthode va découper la trame reçue à l'aide du délimiteur de fin de trame "\r\n" et va determiner s'il s'agit d'une trame télémétrie ou d'une trame capteur puis va exécuter la méthode decoderTrame… associée (ici decoderTrameCapteur).

```
void Rov::decoderTrameCapteur(QString trameCapteur)
{
  QString temperature, humidite, radiation;

  temperature = trameCapteur.section(";",1,1);
  humidite = trameCapteur.section(";",2,2);
  radiation = trameCapteur.section(";",3,3);

  capteurs->setTemperature(temperature);
  capteurs->setHumidite(humidite);
  capteurs->setRadiation(radiation);

  Mesure mesure;
  mesure.dateheure = QDateTime::currentDateTime();
  mesure.temperature = temperature;
  mesure.humidite = humidite;
  mesure.radiation = radiation;

  QDateTime date;

  ihmRov->getCampagne()->ajouterMesure(mesure);

  emit enregistrerMesures(temperature, humidite, radiation);
}
```

La méthode va découper la trame et extraire les valeurs à l'aide du délimiteur de champs : '**;**', puis va affecter les valeurs de mesures , les ajouter dans une structure, puis envoyer le signal enregistrerMesures().

```
connect(rov, SIGNAL(enregistrerMesures(QString, QString, QString)), ihmAccueil,
SLOT(enregisterMesureBDD(QString, QString, QString)));
```

Celui ci connecté à la méthode slot enregistrerMesureBDD().

```
void IHMAccueil::enregisterMesureBDD(QString temperature, QString humidite, QString
radiation)
{
  baseDeDonnees->ouvrir("campagnes.sqlite");
  bool retourRequeteEnregistrementMesure;

  QString requeteEnregistrementMesure = "INSERT INTO mesure (idCampagne, heure,
temperature, radiation, humidite) VALUES (" + recupererIdCampagne() + "," +
QDateTime::currentDateTime().toString() + "," + temperature + "," + humidite + "," + radiation
+ ")";

  retourRequeteEnregistrementMesure =
baseDeDonnees->executer(requeteEnregistrementMesure);

  if(!retourRequeteEnregistrementMesure)
  {
    #ifdef DEBUG_BASEDEDONNEES
      qDebug() << Q_FUNC_INFO << QString::fromUtf8("erreur !");
    #endif
  }
}
```

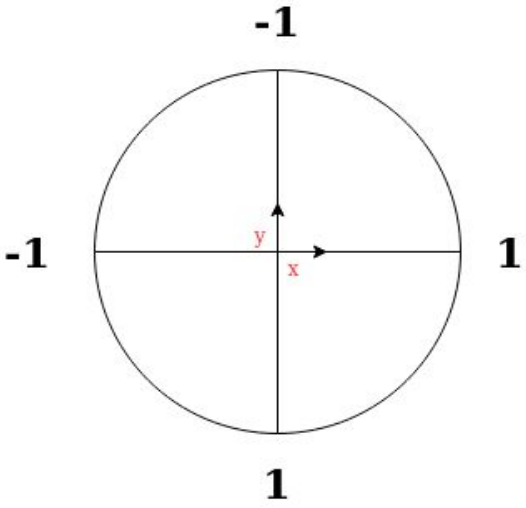
La méthode slot enregistrerMesureBDD() va créer une requête qui va ajouter les mesures dans la base de données et exécuter cette requête.

Fonctionnement Manette :

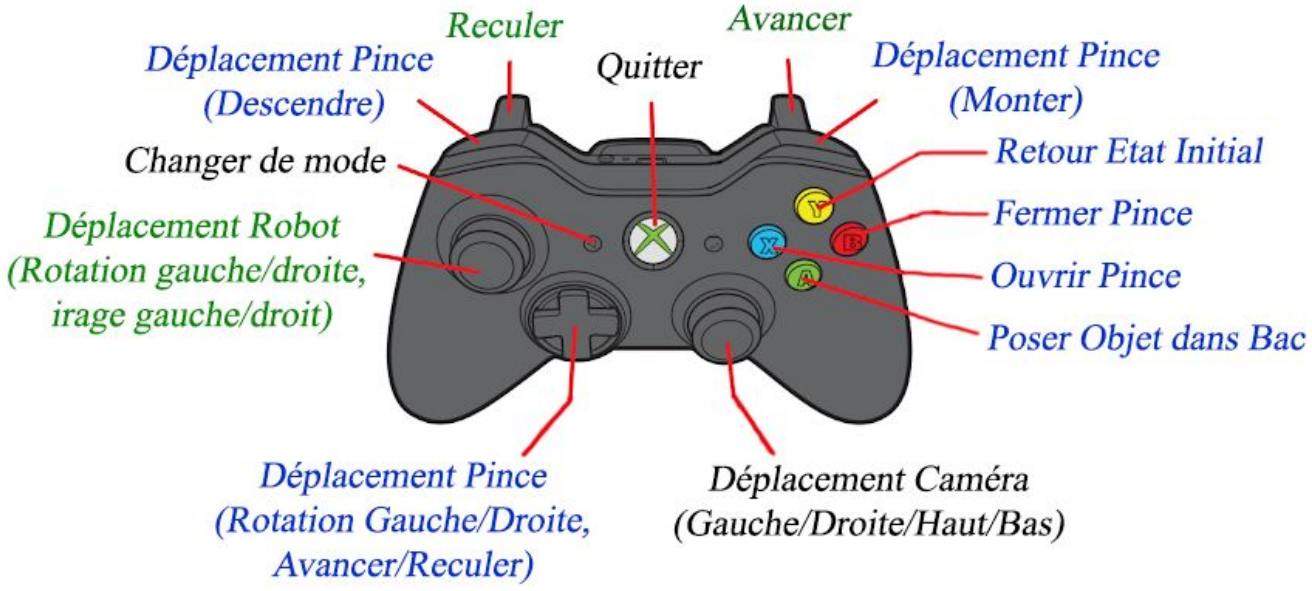
Explication signaux manette

Une manette de jeu est un contrôleur de jeu tenu à deux mains où les doigts sont utilisés pour fournir une entrée. On y trouve différents types de signaux:

- les boutons (X, Y, B, A, L2, L3, R2, R3, Flèche Gauche/Droite/Haut/Bas) qui fournissent un état de bonton *true* ou *false* (0 ou 1).
- les gâchettes (L1, R1) qui fournissent une valeur décimal entre 0 et 1
- les joysticks (Gauche, Droit) qui fournissent une valeur Y et une valeur X décimale entre -1 et 1, plus la valeur est proche de 0 et plus le joystick est au centre

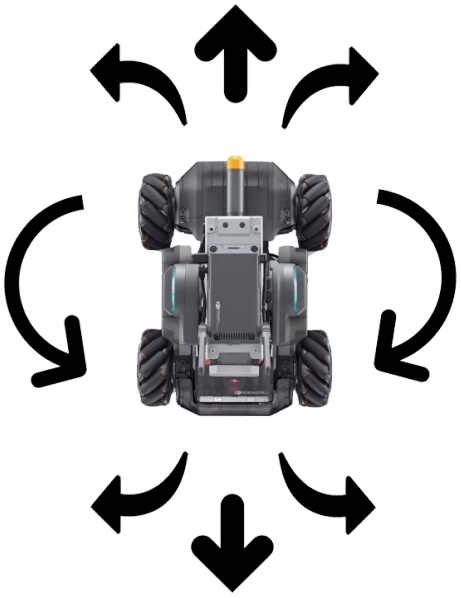


Explication choix des boutons



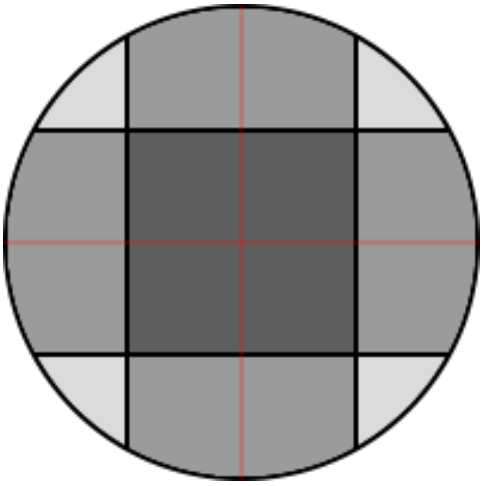
La manette de jeu étant issu du monde des jeux vidéos nous avons choisi de conserver un maximum l'utilisation de boutons communément utilisé pour la conduite de “véhicule”.

- le Joystick gauche symbolisant le volant du véhicule :
    - Rotation à gauche (sur place) : joystick à gauche + Frein
    - Rotation à droite (sur place) : joystick à droite + Frein
    - Virage avant/arrière gauche: joystick à gauche + Avancer/Reculer
    - Virage avant droite : joystick à droite + Avancer/Reculer
    - Virage/Rotation [direction] doucement : joystick en diagonale haut/bas vers [direction]
- les gâchettes symbolisant les pédales du véhicule :
    - Avancer : gâchette de droite (R1)
    - Reculer : gâchette de gauche (L1)
    - Frein : combinaison des deux gâchette (L1 + R1) ou aucune gâchette



Ce qui fait que nous avons un joystick séquencé en différentes parties:

: Zone du joystick où le joystick est considéré comme immobile
: Zone du joystick où l’utilisateur effectue un déplacement direct
: Zone du joystick où l’utilisateur effectue un déplacement en diagonale
__ : Axes X et Y du joystick



Explication création de trame

Pour créer les trames de déplacement il fallait trouver une solution qui empêche l'utilisateur de faire des ordres contradictoires :

- Aller à gauche et à droite en même temps : pour éviter le problème nous avons choisi d'utiliser le joystick pour choisir la direction car celui ci empêche le problème via une impossibilité physique.
- Avancer et reculer : un ordre comme celui ci symboliserait un arrêt du véhicule donc l'ordre sera de freiner (équivalent à aucun ordre).