Lycée TOUCHARD WASHINGTON

académie Nantes

Modélisation

Représentation SysML / UML et leurs spécificités

OBJECTIFS

À l'issue de cette séquence, vous devrez être capable de :

- De déterminer les acteurs et les cas d'utilisation d'un systèmes
- D'établir le diagramme de séquence pour un cas d'utilisation simple
- D'interpréter un diagramme états-transitions
- De réaliser le diagramme de classes d'une application
- De prévoir le déploiement d'une application
- De comprendre les exigences pour un systèmes
- De comprendre la répartition des bloc pour un système

Niveau de maîtrise attendu pour le BTS Systèmes Numériques option Informatique et Réseaux :

S3. Modélisation		IR
S3.2. Représentation SysML / UML	Démarche d'élaboration d'un modèle, formalisme	2
	Liste des acteurs, cas d'utilisation	3
	Diagrammes de séquences	3
	Diagrammes d'états-transitions	2
S3.3. Spécificités SysML	Diagrammes d'exigences	2
	Diagrammes de blocs	2
	Diagrammes de bloc interne et/ou paramétrique	2
S3.4. Spécificités UML	Diagrammes de déploiement	3
	Diagrammes de classes et/ou d'objets	3

UNIFIED MODELING LANGUAGE™

OMG SYSTEMS MODELING LANGUAGE™

Sommaire

Modélisation

Représentation SysML / UML et leurs spécificités

1. Modélisation UML (Unified Modeling Language)

1.1. Introduction

La **modélisation** est une **représentation abstraite et simplifiée** d'un problème afin de mieux l'appréhender, le comprendre, maîtriser sa complexité et d'assurer sa cohérence.

UML, en français, langage de modélisation unifié, utilise une modélisation graphique à base de pictogrammes pour le développement de logiciel **orienté objet**. Ce langage issu d'une fusion de plusieurs méthodes est devenu la référence en termes de modélisation objet. Il est l'aboutissement de travaux d'un consortium américain à but non lucratif, l'Object Management Group (OMG).

La dernière version, en date de décembre 2017, est la 2.5.1. Ses spécifications sont disponibles sur le site officiel de l'**OMG** : https://www.omg.org/spec/UML/.

La **modélisation UML** va aboutir à un descriptif précis du système à étudier pour communiquer dans un langage commun au sein de l'équipe de développement, mais également avec le commanditaire. Elle s'inscrit dans le cycle de vie d'un logiciel.

1.2. Notion d'objet

Cette notion représente une collection d'éléments dissociés comprenant à la fois une structure de données et un comportement. Cette approche de développement logiciel orienté objet est fondée sur la modélisation des objets du monde réel.

L'objectif de cette démarche est de maîtriser la **complexité** d'applications de plus en plus volumineuses. Elle permet également de faire évoluer l'industrie du logiciel dans le sens de la **réutilisabilité** tout en réduisant le poids de la **maintenance** dans les coûts du logiciel en le structurant sous la forme de **couches logicielles**.

1.3. Cycle de vie d'un logiciel

Ce terme regroupe toutes les étapes du développement d'un logiciel. Chacune de ces étapes définit des jalons intermédiaires permettant de s'assurer de la conformité du logiciel en regard du besoin du demandeur. D'une manière générale, le cycle de vie comporte les étapes suivantes :

Analyse	C'est l'étude du problème et la formalisation des besoins du demandeur. Elle permet de recueillir les différentes contraintes et doit répondre à la question « **Quoi faire ?** ». Cette étape permet de déterminer la faisabilité du besoin exprimé et le coût que cela représente.
Conception préliminaire	Cette phase permet l'élaboration d'une solution au problème et au besoin, elle a pour objectif de répondre à la question **« Comment faire ? »**. L'architecture du logiciel et du matériel sur lequel il va être implémenté est déterminée à cette étape.
Conception détaillée	Chaque sous-ensemble est défini précisément lors de cette étape.

C'est lors de ces phases d'analyse et de conception que la modélisation UML va être utilisée.

Codage	C'est la phase de réalisation, elle consiste à traduire dans un langage de programmation les fonctionnalités attendues lors de la phase de conception.
Tests unitaires	À cette étape, on s'assure que chaque sous-ensemble est conforme aux spécifications attendues et qu'il répond donc bien au besoin.
Intégration	Chaque sous-ensemble est assemblé progressivement avec les autres en vérifiant la conformité de l'interfaçage des différents modules.
Recette	Cette étape s'assure de la conformité du système aux spécifications initiales. Elle débouche ensuite sur le déploiement sur site.

Chaque étape décrite ainsi fait l'objet de documentations pour l'utilisation du système et pour les développements ultérieurs qui peuvent apparaître lors de phases de maintenances correctives ou pour faire évoluer le produit. UML, entre autres, contribue à l'élaboration de cette documentation.

L'enchaînement de ces différentes étapes n'est pas forcement linéaire, il demande bien souvent des retours arrière et doit s'assurer de la cohérence de chacune d'entre elles. Le cycle de vie du développement utilise fréquemment le modèle en V.

Ce schéma montre le fait d'anticiper les étapes de recette, d'intégration et de tests unitaires lors des phases d'analyse et de conceptions. Un travail préparatoire est donc nécessaire en amont comme par exemple l'élaboration d'un cahier de recette et des fiches de tests. Le résultat de chaque phase doit rester cohérent avec ceux de la phase précédente. Un retour arrière peut s'avérer nécessaire en cas de modifications dans la phase courante.

L'inconvénient du modèle en V est qu'il ne permet pas une validation rapidement d'un module. D'autres modèles comme le modèle en spirale ou le modèle par incrément tentent de pallier ce problème en découpant le projet en modules développés indépendamment et intégrés à la fin. Le développement est ainsi moins complexe, progressivement des parties peuvent livrer. Le risque avec ces modèles, c'est de remettre en cause les réalisations précédentes ou ne pas pouvoir réaliser l'intégration d'un incrément.

1.4. La démarche pour la modélisation UML

UML est composé de plusieurs sous-ensembles :

- **Les vues**, elles décrivent le système d'un certain regard : organisationnel, dynamique, temporel, architectural, logique… L'ensemble de ces vues permet d'obtenir le modèle du système complet.
- **Les diagrammes** ,ils décrivent le contenu des vues. La version 2.5 d'UML propose 13 diagrammes. En fonction des situations, tous ne sont pas utilisés.
- **Les éléments**, ce sont les composants contenus dans les différents diagrammes.

La démarche UML consiste, dans un premier temps, à élaborer la **vue des cas d'utilisation**. Cette première vue est centrée sur les besoins de l'utilisateur. Elle permet de définir les frontières du système, de déterminer les sollicitations extérieures et de mettre en évidence les fonctionnalités attendues.

Dans un deuxième temps, c'est la **vue logique** qui peut être abordée. Elle décrit les objets qui vont composer le système et les interactions qu'ils vont engendrer et ainsi en définir les aspects statiques et dynamiques.

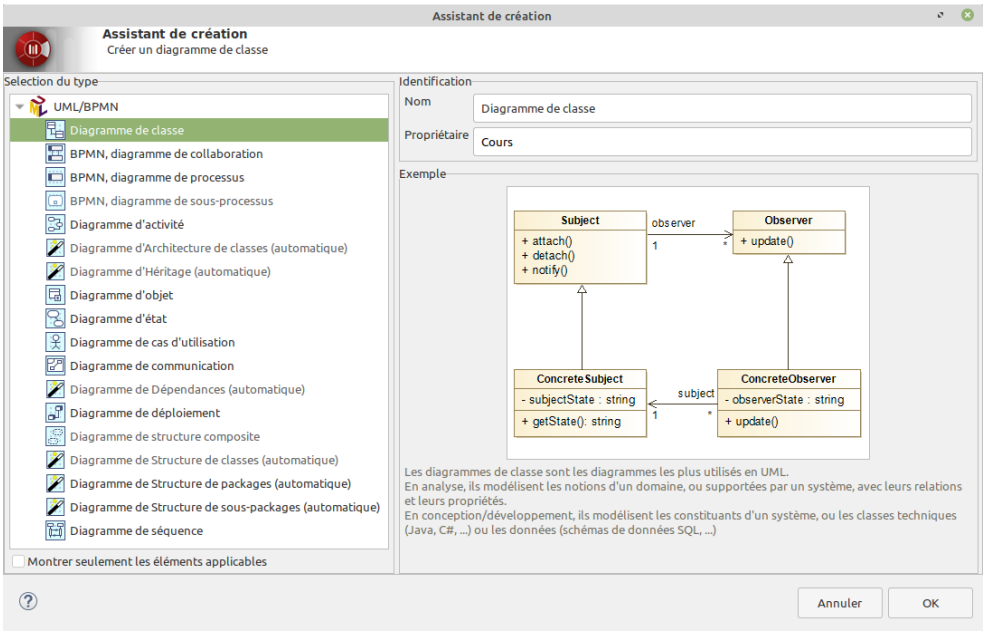
Éventuellement, la méthodologie s'intéresse à la **vue processus.** Elle met en évidence l'aspect concurrentiel et les synchronisations à mettre en place. Cette vue n'est utile que dans un système demandant des traitements parallèles.

La **vue développement** décrit l'organisation statique des modules dans environnement de développement.

Enfin, la **vue physique** montre les différents composants matériels du système et la répartition des différents modules logiciels sur ces derniers.

Le schéma suivant résume ces différentes vues et l'utilisation des différents diagrammes associés :

Tous les logiciels n'offrent pas la possibilité de réaliser l'ensemble des diagrammes. Voici par exemple ceux disponibles sous Modelio.



2. Diagramme de cas d'utilisation

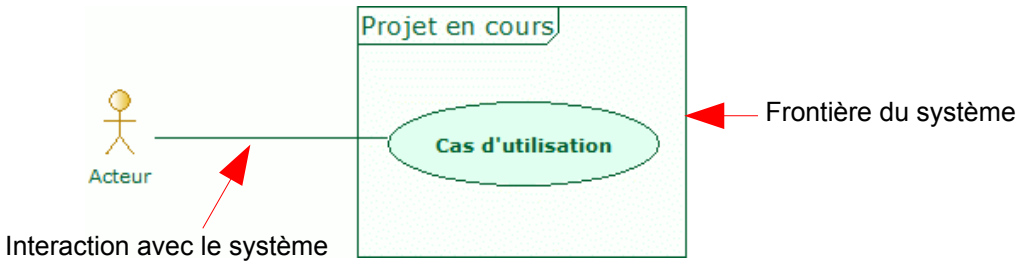
2.1. Présentation

Ce type de diagramme a pour objectif de structurer les besoins du demandeur. Il présente une vision globale du comportement fonctionnel du système, en délimite la frontière et identifie les possibilités d'interaction avec ce qui l'entoure.

On nomme, **acteur**, une personne ou un autre composant qui interagit sur le système étudié. Un acteur engendre forcément une modification de l'état du système.

Un **cas d'utilisation**, *use case,* dans la terminologie anglophone, représente une fonctionnalité du système. L'usage veut que chaque cas d'utilisation soit identifié par un verbe à l'infinitif.

Représentation graphique du diagramme de cas d'utilisation



La représentation de l'acteur est toujours schématisée par une forme humaine, même s'il s'agit d'un autre système.

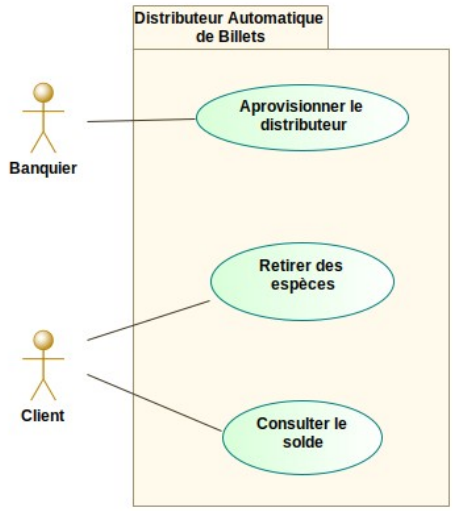
Remarque

Un cas d'utilisation représente une fonctionnalité attendue correspondant à besoin d'un acteur. Un cas d'utilisation réalise un résultat observable pour le ou les acteurs. Un acteur joue un rôle actif pour le système.

Exemple : Distributeur automatique de billets

L'exemple ci-dessous présente une version simplifiée du fonctionnement d'un distributeur automatique de billets.

Diagramme de cas d'utilisation ***Distributeur Automatique de Billets***



Le **Banquier** : Il est chargé d'approvisionner le distributeur en nouveaux billets.

Le **Client** : Il peut retirer des espèces à partir du distributeur. Il peut également consulter le solde de son compte.

La partie droite de la figure ci-dessus représente le catalogue des acteurs. Chaque cas d'utilisation doit faire maintenant l'objet d'une description détaillée.

2.2. Description des cas d'utilisation

Cette description comporte plusieurs rubriques. Les deux premières ne sont pas systématiquement présentent :

- **Pré-conditions** : Conditions nécessaires pour pouvoir réaliser le cas d'utilisation
- **Post-conditions** : Situation, après avoir réalisé le cas d'utilisation
- **Scénario principal** : Déroulement normal des différentes étapes constituant le cas d'utilisation
- Éventuellement des **alternatives** au scénario principal : description des étapes, si toutes les conditions ne sont pas réunies lors du scénario principal, ou une variante du scénario principal.

Détail du cas d'utilisation	***Retirer des espèces***

Pré-condition : Le client dispose d'une carte bancaire

Post-condition en cas de succès : Le client récupère sa carte bancaire et reçoit des espèces.

Post-condition en cas d'échec : La carte bancaire du client est retenue par le distributeur

Détail du scénario principal :

1. le client introduit sa carte bancaire
2. le distributeur vérifie la carte bancaire
3. le client saisit son code secret
4. le distributeur vérifie le code
5. le client choisit l'opération « Retrait »
6. le client spécifie la somme à retirer
7. si le solde est suffisant, le distributeur débite le compte bancaire
8. le distributeur rend la carte
9. le client retire sa carte du distributeur
10. si le solde était suffisant, le distributeur fournit les billets
11. le client retire les éventuels billets

Alternative en cas de code incorrecte :

Suite à l'étape n°4 du scénario principal, le code est incorrect :

1. le distributeur demande à nouveau le code secret
2. le client saisit à nouveau son code
3. le distributeur vérifie à nouveau le code

Si le code est correct, le scénario se poursuit à l'étape 5 du scénario principal. Si le code est toujours incorrect, le scénario reprend à l'étape 1 de l'alternative. Au bout de la troisième tentative en échec, le scénario se termine par l'étape 4 suivante :

4. le distributeur retient la carte bancaire et avertit le client

La description du scénario peut être réalisée sous la forme d'étape comme le montre l'exemple ci-dessus ou sous une forme textuelle.

Exercice d'application

a) Réalisez la description détaillée du cas d'utilisation "Consulter le solde".

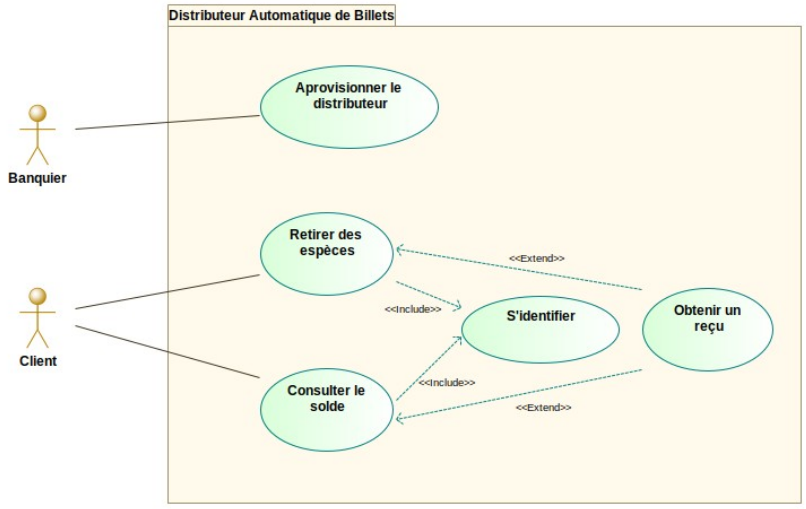
2.3. Relation entre cas d'utilisation

Lors de la description de différents scénarios, on peut éventuellement constater une certaine redondance, dans ce cas de figure, cette redondance peut-être regroupée dans un cas d'utilisation à part. L'analyse du cahier des charges peut éventuellement faire apparaître des cas d'utilisation pas nécessairement liés directement à un acteur. Deux types de relations existent entre les cas d'utilisation, une relation à caractère obligatoire ou relation à caractère facultatif.

Dans l'exemple précédent, on constate que pour les deux cas d'utilisation : le client doit introduire sa carte, le distributeur doit vérifier sa validité, le client doit saisir son code que le distributeur doit également vérifier. Toute cette séquence peut-être regroupée dans un cas d'utilisation nommé « S'identifier » . Il possède un caractère obligatoire, pour les deux cas d'utilisation, on utilisera la **relation d'inclusion** du cas d'utilisation s'identifier.

On peut imaginer un autre cas d'utilisation rencontrer avec un distributeur automatique de billets, c'est l'impression d'un reçu pour l'opération effectuée. Cette impression possède un caractère facultatif, on parle alors de **relation d'extension** pour les autres cas d'utilisation.

Ces relations entre cas d'utilisation sont représentées par des flèches en pointillés orientées.

Diagramme de cas d'utilisation, complément	***Distributeur automatique de billets***
	La relation d'inclusion se traduit par **« include »,** elle présente un caractère obligatoire. La relation d'extension se traduit par **« extend »**, elle présente un caractère facultatif. **Attention à l'orientation des flèches en pointillés** « Retirer des espèces » **inclut** « s'identifier » « Obtenir un reçu » **étend** « Retirer des espèces »

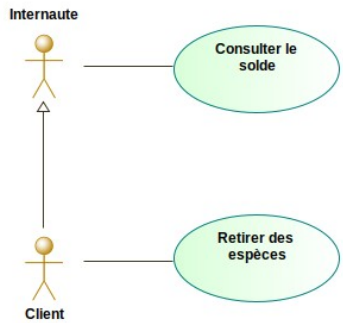
Exercice d'application (suite)

b) Sous Modelio, réalisez le diagramme de cas d'utilisation sous cette forme.

c) Réalisez la description détaillée de chaque cas d'utilisation en tenant compte des inclusions et des extensions.

2.4. Définition de profils utilisateurs

Il arrive parfois que certains acteurs disposent de fonctionnalités supplémentaires par rapport à un autre acteur.

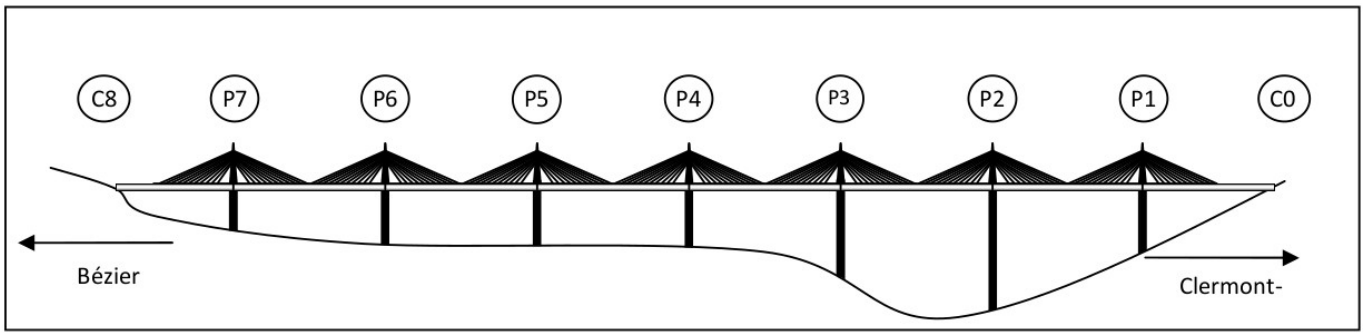
Relation entre acteurs	
	L'**Internaute** peut consulter le solde de son compte bancaire. Le **client** peut retirer des espèces et peut également consulter le solde de son compte. Dans ce cas, il joue le même rôle que l'acteur Internaute.

La relation entre les acteurs est représentée par une relation d'héritage. Le client hérite du rôle de l'internaute.

2.5. Applications

A- Viaduc de Millau

PRÉSENTATION DU SYSTÈME : extrait du sujet de BTS IRIS session 2009



Le viaduc de Millau est un pont à haubans franchissant la vallée du Tarn (Aveyron) sur l'autoroute A71 entre Clermont-Ferrand et Béziers. Il permet de contourner la ville de Millau et d'éviter ainsi les embouteillages estivaux. Son tablier de 32 m de large accueille une autoroute à 2 fois 2 voies ainsi que 2 voies de secours.

Ce pont, achevé en 2004, détient quatre records du monde :

- Les piles les plus hautes : les piles P2 et P3 mesurent respectivement 244,96 et 221,05 m.
- La flèche la plus haute : le haut du pylône de la pile P2 culmine à 343 m.
- Le tablier routier le plus haut : 270 mètres par rapport au sol.
- Le tablier suspendu par haubans le plus long : 2 460 m.

Le cahier des charges prévoit une durée d'utilisation du pont de 120 ans. La surveillance de l'ouvrage sur le long terme doit donc pouvoir respecter cette contrainte. Cette surveillance comporte trois aspects, le contrôle des conditions d'exploitation, le contrôle du comportement du viaduc et le contrôle du vieillissement de la structure.

- Contrôler les conditions d'exploitation : le but est d'assurer la sécurité des véhicules et de leurs passagers. Trois fonctionnalités sont assurées :
  - Surveiller le trafic
  - Mesurer la vitesse du vent
  - Détecter le verglas.
- Contrôler le comportement du pont et vérifier la conformité par rapport aux prévisions définies dans le cahier des charges.
- Surveiller le vieillissement : l'ouvrage est équipé de dispositifs permettant de suivre sur le long terme les fondations du pont, les haubans, les piles, les pylônes et le tablier.

Description des acteurs potentiels :

- Usagers : conducteurs de voiture, de motos ou de poids lourds.
- Opérateur GTC : personne se trouvant au péage et assurant la gestion technique centralisée (GTC)
- Technicien analyste : il va surveiller et analyser le fonctionnement des capteurs.
- Capteurs : ce sont tous les capteurs de l'ouvrage

Description des cas d'utilisation :

- Informer : les usagers peuvent être tenus au courant des conditions de circulation et météorologiques par l'intermédiaire de panneaux à message variable (PMV)
- Surveiller GTC : supervision et surveillance GTC (incendie, intrusion, trafic) ...
- Exporter les données : lors de la surveillance, le technicien peut exporter dans un fichier, les données brutes qui sont envoyées à un organisme public chargé de la surveillance sur le long terme.
- Acquérir les données : les données provenant de tous les capteurs sont récupérées et enregistrées dans une base de données.
- Surveiller ouvrage : surveiller l'instrumentation (capteurs et matériels réseau).

Les systèmes assurant les deux derniers cas d'utilisation étant totalement indépendants, il n'est donc pas systématique d'acquérir les données pour gérer la surveillance.

a) Sous Modelio, à partir des descriptions fournies, réalisez le diagramme des cas d'utilisation.

b) Justifiez l'usage de relations d'extension et/ou d'inclusion présentes dans votre diagramme.

B- Robot d'exploration

PRÉSENTATION DU SYSTÈME : Extrait du sujet de BTS IRIS – Session 2007

Les avancées technologiques dans le domaine de la robotique ont conduit la communauté européenne à mener un nouveau projet nommé **ROBOVOLC** dont le but est l'étude et la réalisation d'un robot mobile pour l'exploration volcanique. Ce projet débuté en mars 2000 rassemble plusieurs partenaires : des universités, des laboratoires de recherche et des entreprises privées. L'objectif majeur du robot étudié est de minimiser les risques pris par les vulcanologues et les techniciens impliqués dans des activités à proximité des cratères en phase éruptive. Il est à noter que les observations les plus intéressantes sont faites au cours des phases paroxysmiques des éruptions, au cours desquelles le risque est bien entendu maximum.

Le cahier des charges établi par l'ensemble des partenaires spécifie que le robot doit être capable de :

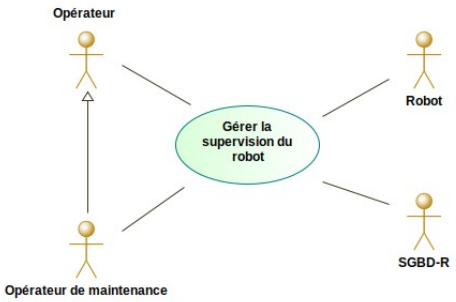
- S'approcher d'un cratère actif,
- Collecter des échantillons de rejets éruptifs,
- Collecter des échantillons gazeux,
- Collecter des données physiques et chimiques,
- Surveiller une bouche de cratère.

Lors de l'étude préliminaire, et pour atteindre ces objectifs, l'équipe de **ROBOVOLC** a décomposé le système en plusieurs sous-systèmes:

- Sous-système de déplacement,
- Sous-système de navigation,
- Sous-système de vision,
- Sous-système de prélèvement,
- Sous-système de communication,
- Sous-système de supervision.

Fig 1 : Le ROBOVOLC

Analyse du système de supervision



Le système de supervision du robot interagit avec :

- le robot,
- un système de gestion de base de données permettant d'enregistrer les points de passage du robot,
- l'opérateur du système,
- et l'opérateur de maintenance.

Ceci est présenté dans le diagramme ci-contre.

Cette analyse des cas d'utilisation est sommaire, elle permet de définir les limites du système. Ceci implique, par exemple, que le robot est considéré comme acteur du système de supervision…

a) Que signifie la flèche reliant **Opérateur de maintenance** à **Opérateur** ?

b) La page suivante propose quatre diagrammes de cas d'utilisation d'un niveau de détail supérieur à celui-ci dessus. Deux d'entre eux sont manifestement incohérents avec le diagramme ci-dessus, indiquez lesquels et justifiez la réponse.

Diagramme	Correct (oui/non)	Justification dans le cas où le diagramme ne soit pas correct
n°1		
n°2		
n°3		
n°4		

c) Dans les diagrammes n°2 et n°3 apparaît une flèche en pointillée, indiquez le type de relation liant les deux cas d'utilisation **Prélever un échantillon** et **Piloter le robot à distance**.

d) Même question pour le diagramme n°2 pour las cas d'utilisation **Enregistrer les trajectoires réelles** et **Recevoir informations de position du robot**.

Diagramme n°1

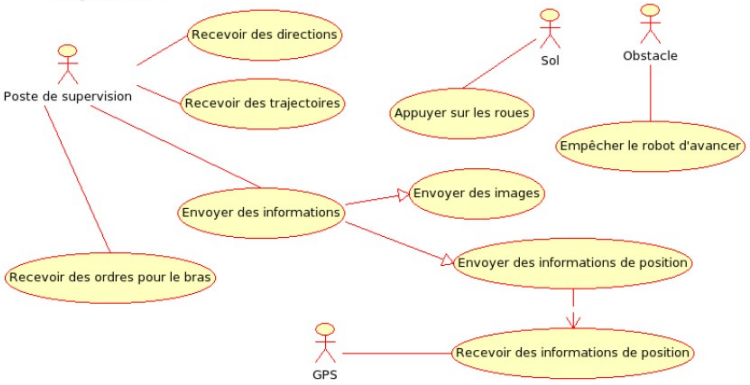


Diagramme n°2

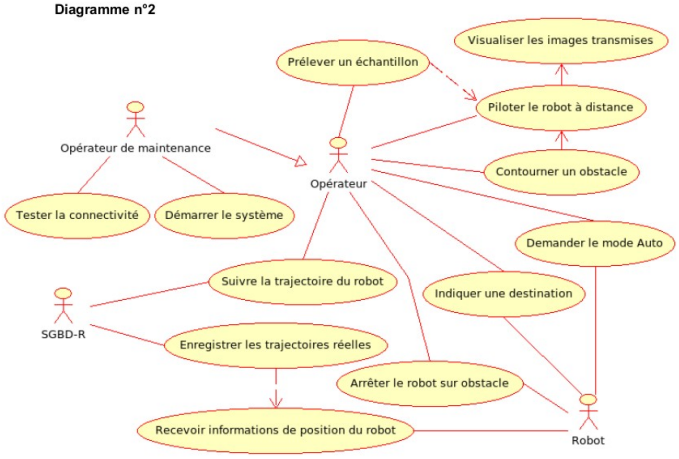


Diagramme n°3

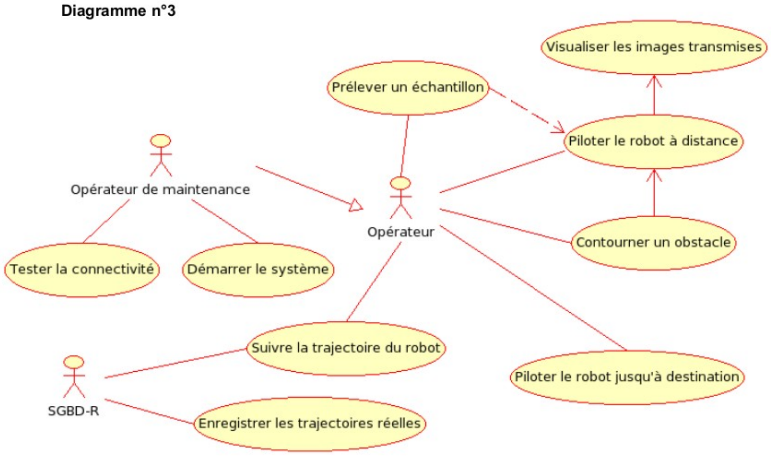
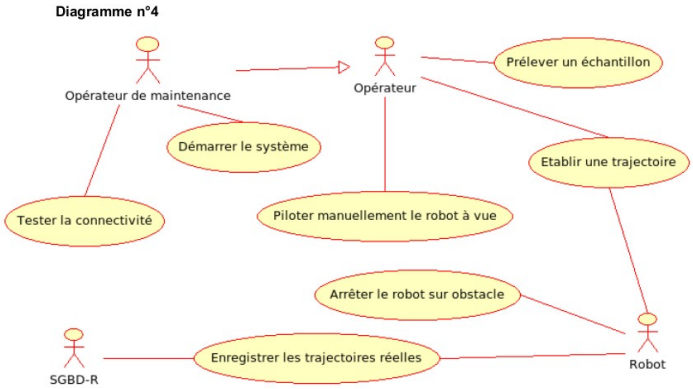


Diagramme n°4



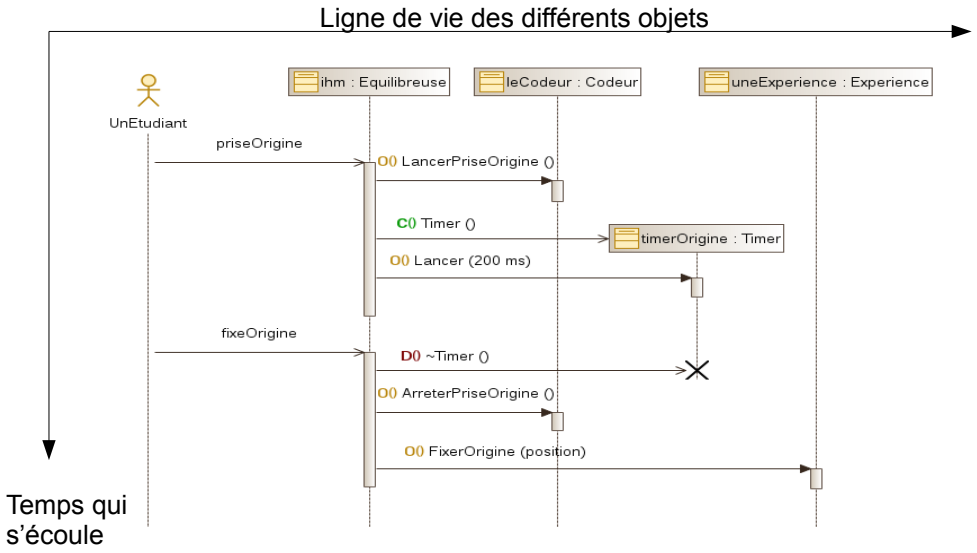
e) D'après les différents diagrammes de cas d'utilisation ci-dessus, comment justifier le fait que SGBD-R est un acteur du système ?

3. Diagramme de séquences

3.1. Présentation

Le diagramme de séquences est l'un des diagrammes UML permettant de décrire les sollicitations des acteurs envers les objets du modèle et les interactions que peuvent avoir les objets entre eux. Il représente un point de vue temporel dans le cadre d'un scénario donné, pour un cas d'utilisation particulier. Ainsi, chaque cas d'utilisation à partir de sa description textuelle peut conduire à l'élaboration d'un diagramme de séquence.

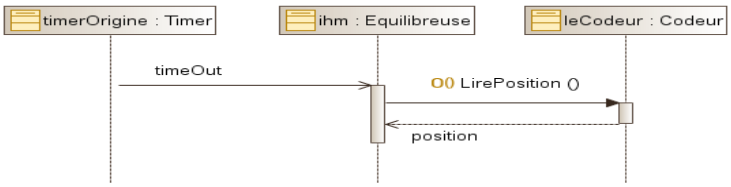
L'aspect temporel de ce diagramme est représenté par sa verticalité, le temps s'écoule du haut vers le bas. Les lignes de vie de chacun des objets donnent la dimension horizontale du diagramme.



Remarque

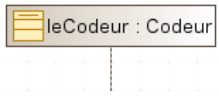
Un diagramme de séquence permet de modéliser facilement un traitement séquentiel lié à un scénario. Il est parfois nécessaire d'utiliser un deuxième diagramme pour faire apparaître un certain parallélisme.

Cela peut être le cas avec notre exemple de prise d'origine, le **timer** en parallèle des actions de l'étudiant déclenche la lecture de la position angulaire afin de rafraîchir l'affichage.



3.2. Les lignes de vie

Chaque ligne de vie représente un acteur ou un objet du modèle.

	L'acteur est représenté sous la forme d'une image. Il sollicite le système par l'envoi de messages : l'appui sur une touche, l'envoi d'un signal électrique, l'envoi d'une donnée…
	L'objet **leCodeur** est une instance de la classe **Codeur**. Il est symbolisé sous sa forme structurée. Lorsque l'on place une classe sur le diagramme de séquence, il est nécessaire d'indiquer le nom de l'instance, ou d'utiliser une instance existante dans un autre diagramme de séquence si elle existe déjà par ailleurs.

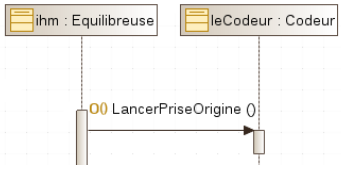
3.3. Les différents types de messages

Les messages sont échangés entre les lignes de vie, ils sont représentés par des flèches orientées de l'émetteur vers le destinataire. Ils apparaissent par ordre chronologique, de haut en bas, dans le diagramme. Il existe plusieurs types de messages : l'***envoi d'un signal***, l'***appel d'une méthode*** et la ***création*** ou la ***destruction*** d'un objet.

Les messages synchrones et asynchrones

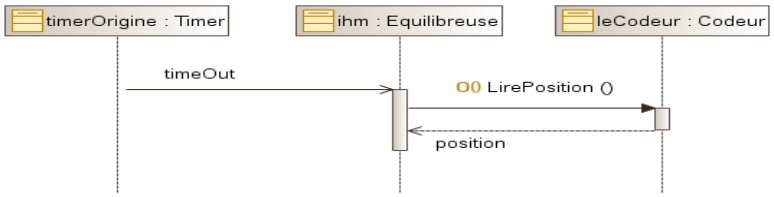
Un ***message synchrone*** bloque l'expéditeur jusqu'à l'obtention de la réponse du destinataire, ou jusqu'à la fin de l'exécution de l'appel s'il n'y a pas de retour à attendre. Ce type de message correspond à l'***appel d'une méthode***. Il est représenté par une flèche pleine ⟶.

L'instance **ihm** appelle la méthode **LancerPriseOrigine( )** de la classe **Codeur**.

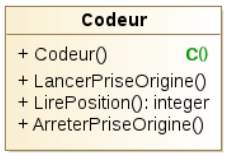


Lorsque l'exécution de la méthode est terminée, le contrôle de la séquence est repris par l'instance **ihm**. Le rectangle sur la ligne de vie représente la durée de prise de contrôle par l'objet.

Dans le cas de la méthode **LirePosition( )** dans l'extrait de diagramme ci-contre, la méthode possède un paramètre de retour, ici la position du codeur. Le traitement dans l'instance **ihm** reprend après réception de ce paramètre de retour. Il est représenté par une flèche en pointillé creuse ⟵ - - - .



Comme on peut le voir dans le premier exemple, le paramètre de retour est facultatif. Il peut donc être représenté ou pas.



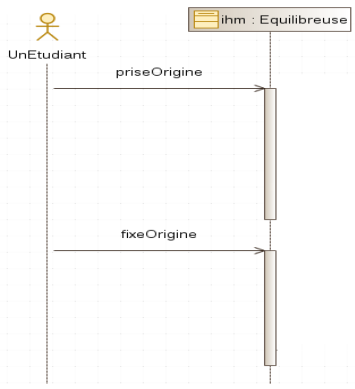
Les méthodes correspondantes aux messages doivent être déclarées dans la classe du récepteur du message. La classe **Codeur** possède ainsi au moins les méthodes **LancerPriseOrigine( )** et **LirePosition()** invoquées dans ces deux extraits de diagramme.

La deuxième méthode possède bien un paramètre de retour de type entier, il représente la position du codeur.

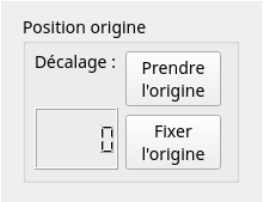
Remarque

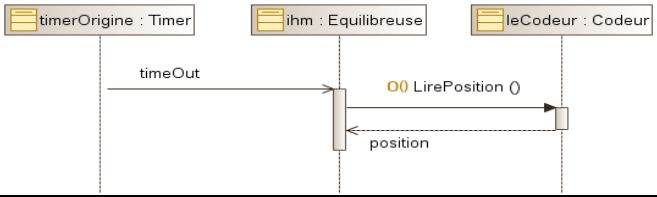
Un retour implicite ne peut revenir que sur l'objet qui a donné naissance à l'envoi du message

Un ***message asynchrone*** ne bloque pas l'expéditeur, il peut être pris en compte ou ignoré par le récepteur. Ce type de message est représenté par une flèche creuse ⟶. Il correspond à l'***envoi d'un signal***.



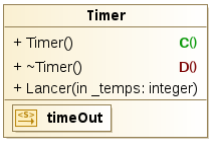
Ici, l'acteur **unEtudiant** dispose par exemple de deux boutons sur son Interface Homme Machine. L'appui sur un de ces boutons engendre l'envoi d'un signal qui sera éventuellement intercepté par l'instance **ihm** de la classe **Equilibreuse**. Deux méthodes différentes pourront être associées à ces signaux pour effectuer le traitement correspondant. Le nom de ces méthodes n'apparaît pas sur le diagramme de séquence.





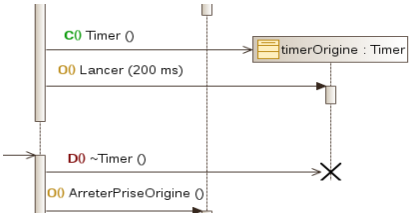
Un **message asynchrone** peut également être un signal produit par un autre objet, ici, le **timerOrigine** indique à l'objet **ihm** que le temps est écoulé à travers le signal **timeOut**. Il faut aller lire la position du codeur pour rafraîchir l'affichage.

Le signal apparaît cette fois-ci, dans la classe expéditrice du message, ici la classe **Timer**.



Les messages de construction et de destruction d'objet

La création se fait par un **message asynchrone** pointant vers le rectangle en tête de ligne de vie, son stéréotype est **<<create>>**. La destruction est également réalisée par un **message asynchrone** pointant une croix sur la ligne de vie, le stéréotype du message est cette fois **<<destroy>>**. La ligne de vie est réduite entre la création et la destruction de l'objet. Elle pourra se traduire par exemple avec une allocation dynamique de l'objet dans le futur programme et la libération de la mémoire.



L'extrait du diagramme de séquence ci-contre fait apparaître les messages de construction et destruction. Ici, les stéréotypes ont été remplacés par le constructeur et le destructeur de la classe, qui est une autre manière de représenter l'action de construire et de détruire un objet.

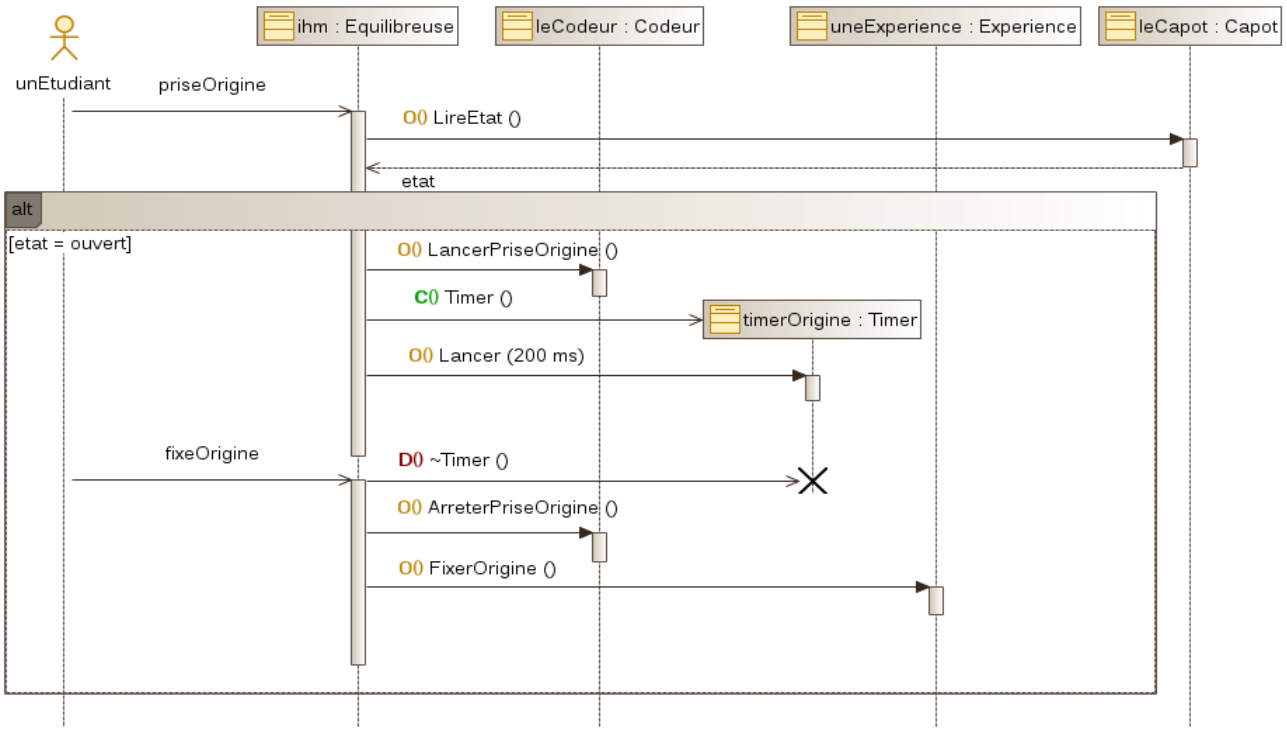
3.4. Les fragments combinés

Un fragment combiné permet d'exprimer de manière plus simple une séquence qui semble complexe au premier abord. Il est représenté par un rectangle dont le coin supérieur gauche contient un pentagone ou figure un opérateur d'interaction. Un fragment combiné peut avoir des contraintes d'intégration également appelé **gardes** ou **guard** en anglais. Elles spécifient une condition et sont représentées entre crochets dans le fragment combiné.

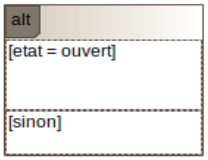
Fragment *alt* : l'opérateur d'interaction alternatif

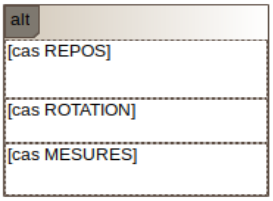
L'opérateur d'interaction **alt** signifie que le fragment combiné représente un choix parmi des alternatives de comportement. Au plus, un comportement sera choisi. Pour être choisie, l'expression de garde associée doit être évaluée à vrai à ce stade du scénario.

Si dans l'exemple précédent on tient compte du fait que le capot soit ouvert pour réaliser la prise d'origine le diagramme de séquence devient :



Si nécessaire, il est possible d'utiliser la close [sinon] ou de généraliser aux différents cas parmi plusieurs chaque cas forme un nouveau fragment combiné.



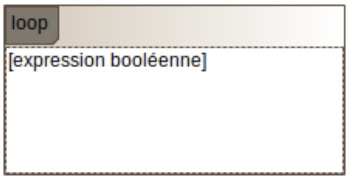


Fragment *opt* : l'opérateur d'interaction optionnel

Ce fragment contient une séquence qui peut ou non se produire. C'est l'équivalent d'un fragment combiné avec l'opérateur **alt** sans possibilité d'avoir un sinon.

Fragment *loop* : l'opérateur d'interaction itératif

Le fragment est répété un certain nombre de fois en fonction de la condition spécifiée par l'expression booléenne entre crochets.



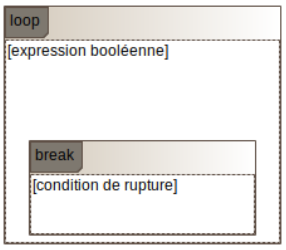
Remarque

Les deux fragments ***alt*** et ***loop*** sont les principaux fragments rencontrés en BTS. Les autres sont donnés à titre indicatif et peuvent être utilisés au besoin.

Fragment *break* : l'opérateur d'interaction d'arrêt

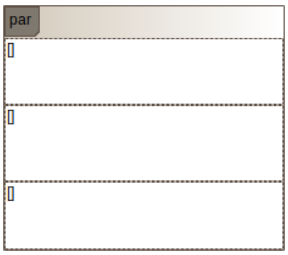
Si ce fragment est exécuté, le reste de la séquence est abandonné. La condition de rupture est indiquée entre crochets.

Un fragment combiné avec l'opérateur ***break*** doit couvrir toutes les lignes de vie du fragment d'interaction englobant. Lorsque la condition de rupture est vraie, le reste du fragment englobant est ignoré.



Fragment *par* : l'opérateur d'interaction de parallélisation

L'opérateur d'interaction **par** définit une exécution potentiellement parallèle des comportements des différentes parties du fragment combiné. Il n'y a pas forcement d'expression entre les crochets.



Fragment *seq* : l'opérateur d'interaction de séquençage faible

Il existe au moins deux fragments d'opérande. Les messages impliquant la même ligne de vie doivent se produire dans l'ordre des fragments. Lorsqu'ils n'impliquent pas les mêmes lignes de vie, les messages des différents fragments peuvent être entrelacés en parallèle.

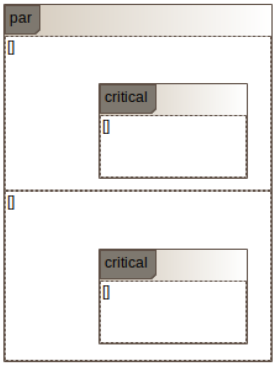
Fragment *strict* : l'opérateur d'interaction de séquençage strict

Il existe au moins deux fragments d'opérande. Les fragments doivent se produire dans l'ordre donné.

Fragment *critical* : l'opérateur d'interaction de section critique

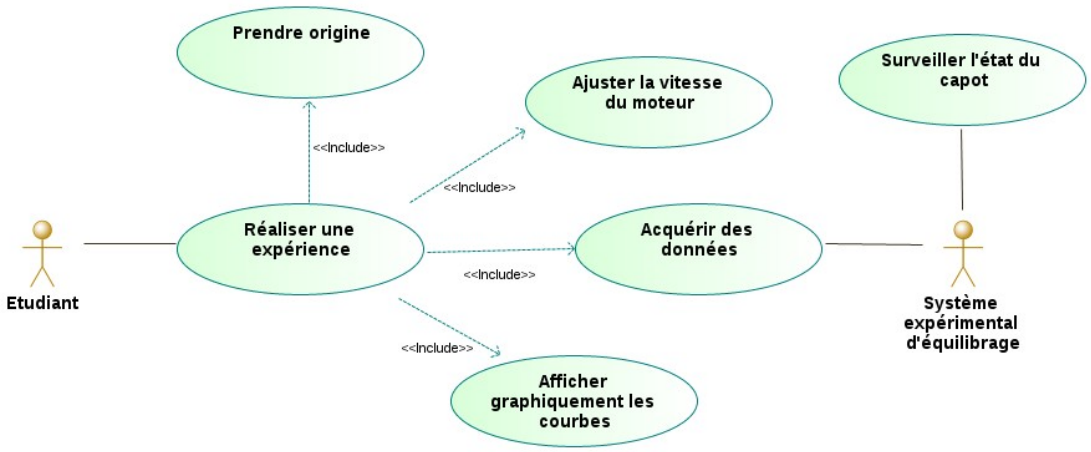
Utilisé dans un fragment ***par*** ou ***seq***. Indique que les messages de fragment ne doivent pas être entrelacés avec d'autres messages.

Cela signifie que la section critique est traitée de manière **atomique** par le fragment englobant.



3.5. Application

Le diagramme de cas d'utilisation de l'équilibreuse est présenté à la figure suivante.



Après avoir étudié le cas d'utilisation « **Prendre origine** » précédemment, on propose de faire l'étude du cas d'utilisation « **Ajuster la vitesse du moteur** ».

Détail du cas d'utilisation	***Ajuster la vitesse du moteur***

Pré-condition : L'étudiant fixe un pourcentage de la vitesse maximale du moteur à l'aide du potentiomètre mis à sa disposition sur l'ihm.

Détail du scénario principal :

Lorsque l'étudiant applique la consigne de vitesse, si le capot est fermé, la vitesse maximale du moteur est obtenue dans les caractéristiques du système. Puis après calcul du pourcentage, la consigne de vitesse est fournie au moteur. En revanche, si le capot est ouvert, la consigne de vitesse est fixée à 0 pour éviter que le moteur démarre lors du prochain changement d'état du capot.
Lorsque l'étudiant décide d'arrêter le moteur, la consigne de vitesse est fixée à 0 pour le moteur.
Au cours de l'expérience, l'étudiant peut modifier la consigne de vitesse.

On propose l'Interface Homme Machine suivante pour ajuster la vitesse du moteur.

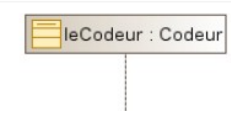
a) En vous inspirant des exemples du cours, à l'aide de l'outil Modelio, réalisez le diagramme de séquence permettant d'ajuster la vitesse du moteur.

b) Reproduisez le diagramme de cas d'utilisation donné ci-dessus. Puis déplacez le diagramme de séquence que vous venez de réaliser pour qu'il soit en relation avec le cas d'utilisation lui correspondant. Renommez l'interaction.

c) Reproduisez les diagrammes de séquence donnés en exemple du cours relatif au cas « Prendre l'origine » en tenant compte des instances existantes dans le diagramme de séquence que vous avez réalisé précédemment.

4. Diagramme de classes

4.1. Différences entre classe et instance.



Une classe décrit un ensemble d'objets. Ces objets sont apparus dans le diagramme de séquence étudié précédemment, ce sont les différentes lignes de vie. Exemple, l'objet **leCodeur** est un exemplaire, ou instance, de la classe **Codeur**. Les classes sont constituées d'attributs et d'opérations, elles représentent un concept abstrait qui s'appuie sur le monde réel, ce sont des modèles.

Une instance est une vision concrète d'une classe. Dans les exemples précédents, la classe **Experience** peut contenir des données mesurées, une vitesse de rotation du moteur, un nom pour l'expérimentateur… L'instance, **une**E**xperience,** désigne l'expérience d'un étudiant en particulier contenant les mesures qu'il a réalisées à une certaine vitesse de rotation.

Remarque

Une classe représente une description formelle d'un ensemble d'objets ayant des caractéristiques semblables et des comportements identiques.

Sous un identifiant unique, une classe regroupe un ensemble fortement couplé de données et d'opérations permettant à l'utilisateur de cette classe de manipuler ces données.

On parle d'opération ou de spécification lors de la déclaration d'une méthode. Le nom **méthode** est normalement réservé à son implémentation ou définition.

4.2. Notion d'encapsulation

La notion d'encapsulation représente un mécanisme qui permet de rassembler au sein d'une même entité des données et des opérations en cachant l'implémentation. L'intégrité des données est ainsi garantie par une visibilité restreinte des données et des services offrant aux utilisateurs de l'objet les moyens de les manipuler indirectement. Trois niveaux de visibilité apparaissent dans les diagrammes UML :public, protégé et privé.

D'une manière générale, les données sont privées, voire protégées, elles peuvent être manipulées que par les services proposés dans la classe. Les opérations ou services appliqués sur ces données sont publics pour être accessible.

4.3. Représentation des classes

Les classes sont représentées par un rectangle comportant plusieurs subdivisions :

- Le premier précise le nom de la classe, par convention, il commence par une majuscule.
- La seconde subdivision est réservée aux attributs, outre le nom des attributs, leur type et leur visibilité sont également indiqués. Le nom de l'attribut commence par convention par une minuscule. Il existe trois sortes de visibilité. Elle est indiquée par le signe qui précède l'attribut :

  + pour **Public** **:** Tout élément ayant accès à la classe, accède également à l'attribut.
  # pour **Protégé :** Seul un élément situé dans la classe ou un de ses descendants accède à l'attribut.
  - pour **Privé** **:** Seul un élément situé dans la classe accède à l'attribut.

- La troisième subdivision contient les différentes opérations de la classe. Le nom des opérations est également précédé d'un signe, **+**, **-** ou **#**, pour indiquer la visibilité comme avec les attributs. Le nom est généralement composé d'un verbe à l'infinitif commençant par une majuscule suivi éventuellement par un complément commençant également par une majuscule afin d'améliorer la lisibilité. Entre parenthèses, se trouvent ensuite les paramètres fournis à la fonction. Pour chaque paramètre, on trouve sa direction, son nom écrit en minuscule et son type. Généralement, le nom du paramètre est précédé du signe souligné _ afin de le différencier d'un attribut. La direction prend l'une des trois valeurs :

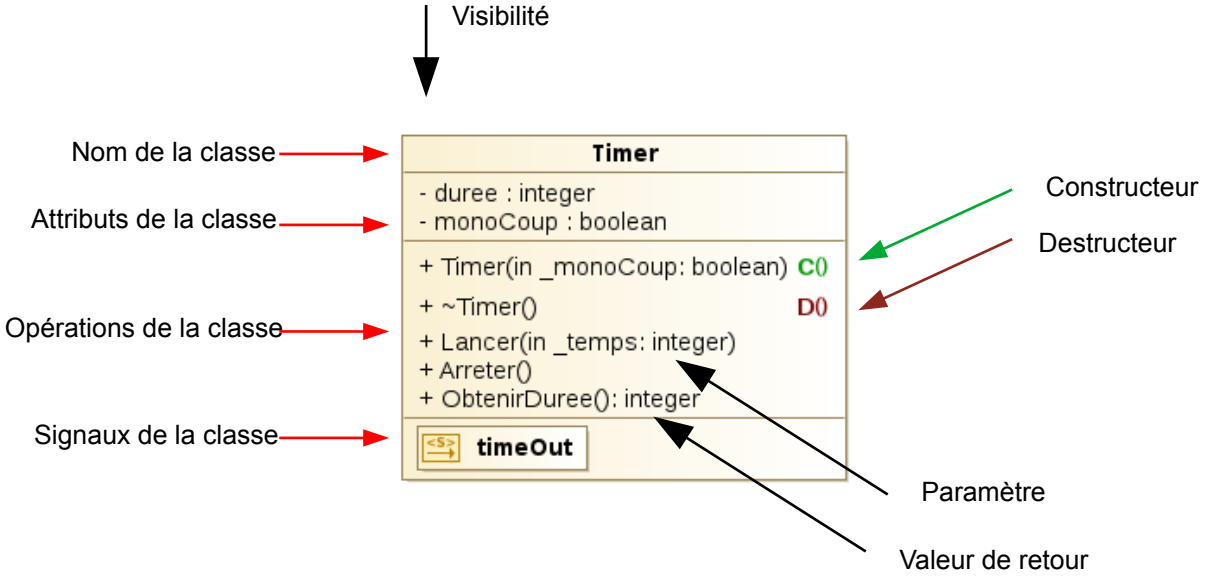
  in : Le paramètre est uniquement en entrée. Il n'est pas modifiable dans la fonction. Sa valeur n'est pas affectée dans l'appelant au retour de la fonction.
  out : Le paramètre est uniquement en sortie. Il ne possède pas de valeur en entrée. Sa valeur, fixée dans la fonction, est disponible pour l'appelant au retour de la fonction.
  inout : Le paramètre est en entrée / sortie. L'appelant transmet une valeur qui est modifiée par la fonction. La valeur finale est disponible pour l'appelant au retour de la fonction.

Enfin, la spécification de la méthode se termine éventuellement par son type de retour, rien si celui-ci n'est pas nécessaire.

Généralement, il existe deux méthodes particulières, le constructeur et le destructeur, qui font référence aux messages de construction et de destruction découverts lors de la présentation des diagrammes de séquence. Il porte tous deux le nom de la classe, le second est précédé du signe **~** pour le différencier. Le constructeur sert à initialiser les attributs de la classe et réaliser de l'allocation mémoire si nécessaire. Le destructeur est utilisé pour restituer un certain contexte mémoire, ou un certain état à la fin de vie de l'objet. Ils peuvent tous deux être implicites, ne pas apparaître, si l'objet ne nécessite pas d'initialisation particulière. Tous deux n'ont jamais de valeur de retour. Le destructeur ne possède jamais de paramètre, la liste entre parenthèses est vide.

Le nom des différentes méthodes et du constructeur peut apparaître plusieurs fois pour une même classe. Dans ce cas, les paramètres sont différents, cette spécificité se nomme **surcharge d'opération**. Cette surcharge permet d'appeler une méthode dans différents contextes afin d'adapter le code en fonction des données à traiter. Ces surcharges feront l'objet d'implémentations spécifiques dans le codage.

- La quatrième subdivision est réservée à énoncer les énumérations propres à la classe et les signaux émis.
- Une éventuelle cinquième subdivision est réservée pour faire apparaître les traitements d'exception gérés par la classe.



4.4. Particularités, attributs et méthodes de classe

Un attribut de classe à la particularité d'être partagé par toute les instances de la classe contrairement aux autres attributs qui eux sont différenciés. Ce type d'attribut peut s'avérer utile lorsque l'on a besoin de compter le nombre d'exemplaires de la classe par exemple, ou de mettre en place un partage de ressource… **Dans sa représentation UML, le nom de l'attribut est souligné**.

De même, il existe des méthodes de classe. Ce type de méthode ne peut en aucun cas manipuler des attributs autres que des attributs de classe. Elle est parfois utilisée pour implémenter une routine d'interruption qui ne peut exister qu'en un seul exemplaire, sinon, elle permet d'implémenter des fonctions qui ne nécessitent pas l'instanciation de la classe. De la même manière, l**a représentation UML est le soulignement du nom de l'opération**.

4.5. Méthodes et classes abstraites

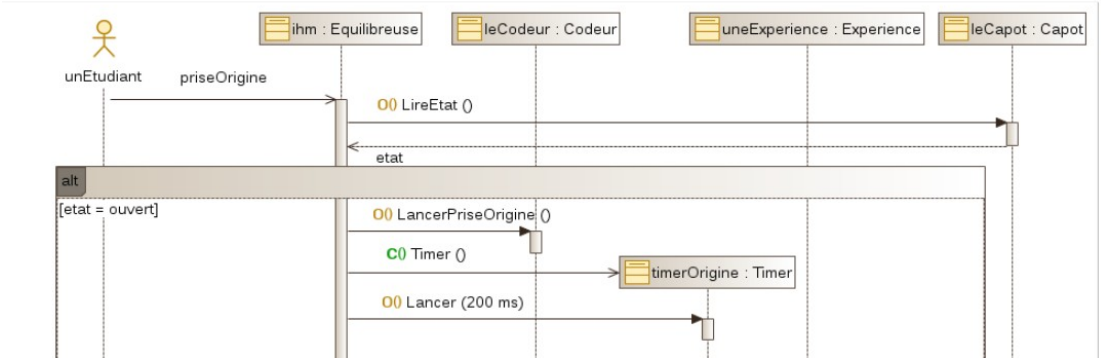
Une méthode est dite abstraite lorsque son implémentation n'est pas connue. Ce mécanisme permet de fixer un cadre à des méthodes dont l'implémentation dépend de l'exécution du programme. On parle également de méthode virtuelle pure dans certains langages comme le C++.

Une classe abstraite contient au moins une méthode abstraite. La particularité de cette classe est qu'elle ne peut pas être instanciée directement, il est nécessaire de la dériver et de redéfinir la ou les méthodes abstraites.

Dans la représentation UML, le nom de classes ou de méthodes abstraites est en italique.

4.6. Relations entre classes

La nécessité de mettre en place des relations entre les classes apparaît lors de la construction du diagramme de séquence.



La classe **Equilibreuse** fait appel à une méthode de la classe **Capot**, **LireEtat()** ou elle fait appelle à une méthode de la classe **Codeur**, **LancerPriseOrigine()**. L'instance **ihm** doit nécessairement connaître l'instance **leCapot** ou **leCodeur** dans le cas présent.

Cette relation entre objets peut prendre différentes formes en fonction du degré d'imbrication des classes entre elles.

Notion d'association

Une association est un premier type de relation entre classes, elle décrit des connexions structurelles entre leurs instances. Ces instances ont une durée de vie complètement indépendante les unes des autres.

Une terminaison d'association est un attribut désignant l'objet à l'extrémité opposée de l'association. Dans l'exemple ci-dessous, l'association possède deux terminaisons.

Une terminaison possède plusieurs propriétés, toutes ne sont pas systématiquement renseignées. Elles interviendront sur l'implémentation ou non de l'attribut par la suite.

Nom : Comme tout attribut, une terminaison d'association est nommée, le nom est situé à proximité de la terminaison opposée. Il s'appelle également dans la terminologie UML nom de **rôle**.

Visibilité : Comme pour un attribut, elle indique le degré d'encapsulation, privé, protégé ou public de la terminaison. Elle est placée éventuellement devant le nom de rôle.

Cardinalité : Également nommée multiplicités, cette propriété indique le nombre d'instances mises en jeu pour cette association. Elle est représentée par un chiffre ou une suite de symboles à proximité de la terminaison de l'association. Exemple de cardinalité :

- exactement un : 1
- plusieurs : * (éventuellement 0)
- entre n et m : 1..5
- au moins 3 : 3..*

Navigabilité : Lorsqu'elle n'est pas précisée, la navigabilité peut être réalisée dans les deux sens. Pour la limiter, une des extrémités comporte une flèche creuse pour indiquer le sens. Cela conduit à l'implémentation d'un seul rôle en tant qu'attribut, celui à proximité de la flèche. Il est implémenté dans la classe opposée.

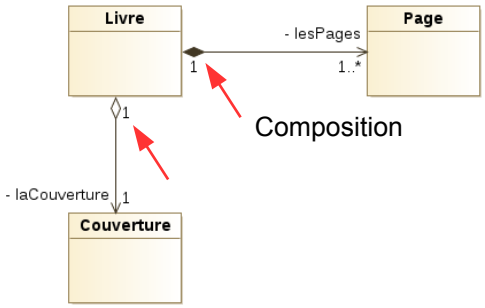
Dans l'exemple ci-dessus, un polygone possède au moins 3 sommets. Les sommets sont regroupés dans l'attribut privé **lesSommets** dans la classe **Polygone**. Un sommet fait partie d'un et un seul polygone. Dans la classe **Sommet,** c'est l'attribut **unPolygone** qui désigne le polygone auquel il appartient. La navigabilité est bidirectionnelle. Si un seul sens avait été retenu, par exemple de **Polygone** vers **Sommet**, une flèche aurait été représentée dans le sens **Polygone** – **Sommet**. Un seul attribut serait implémenté, celui de la classe **Polygone**.

Notions d'agrégation et de composition

Une agrégation est une association qui désigne une relation structurelle ou comportementale d'une classe dans un ensemble, sans pour autant contraindre la durée de vie des objets en relation. Cette notion est purement conceptuelle. Elle est représentée par un losange creux : ◇

Une composition est une agrégation forte. Contrairement à la forme précédente, la durée de vie des éléments conduisant à cette relation est liée. La création de l'objet contenant entraîne la création des objets contenus, de même sa destruction entraîne leur destruction. La représentation **UML** est le losange plein : ◆

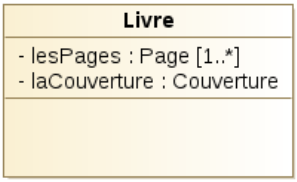
Pour ces deux notions, on parle d'**agrégat** pour l'objet contenant, et de **composants** pour les objets associés.



Un livre est composé d'au moins une page. En revanche s'il ne possède pas de couverture ce n'est pas très grave, le livre reste utilisable. Cet exemple est purement conceptuel, il illustre juste un degré d'importance entre les pages d'un livre et sa couverture.

Dans cet exemple, le nom de rôle n'apparaît que du côté qui nous intéresse, celui exprimé du côté de la navigabilité. La cardinalité est reportée des deux côtés, mais n'est utile que du côté du nom de rôle. Elle servira à l'implémentation des attributs dans la classe côté opposé.

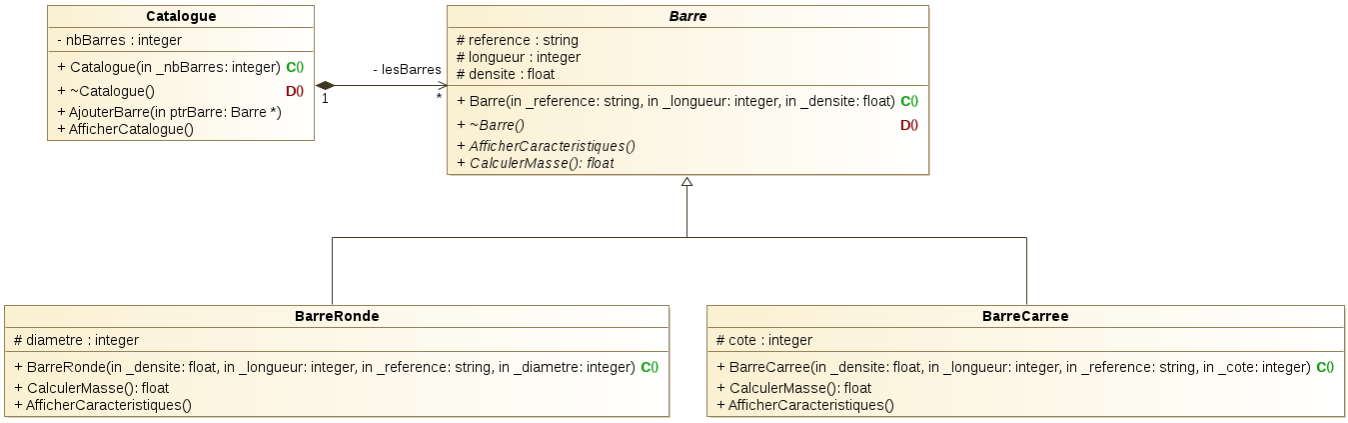
Autre représentation possible du diagramme précédent :



Notion d'héritage

Le concept d'**héritage** ou de **généralisation** décrit une relation entre une **classe de base** et une ou plusieurs classes **spécialisées**, les **sous-classes**. La classe spécialisée bénéficie de toutes les propriétés qui ne sont pas privées de la classe de base, d'où la notion de **protégé** introduite dans le paragraphe sur la représentation des classes. Cette sous-classe peut apporter de nouveaux comportements et de nouvelles données au besoin, elle reste cohérente par rapport à la classe de base. Une instance d'une sous-classe peut être utilisée partout ou une instance de la classe de base est autorisée.

La représentation de l'héritage est un triangle creux dont le sommet est orienté vers la classe de base : △



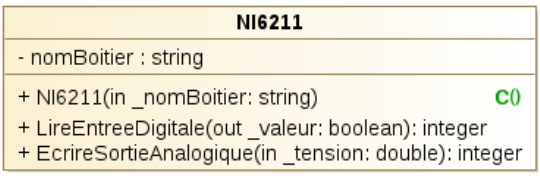
Dans l'exemple ci-dessus, la classe **BarreRonde** hérite de la classe **Barre**, de même pour la **BarreCarree**. On constate en effet que **BarreRonde** et **BarreCarree** spécialisent la classe **Barre**. En redéfinissant chacune à leur manière l'opération **CalculerMasse()** qui n'utilise pas les mêmes formules mathématiques pour calculer la section d'un disque ou d'un carré.

La notion de classe abstraite évoquée précédemment prend ici tout son sens. La classe **Barre** définit un cadre pour toutes les formes de barres qui pourraient en hériter, mais ne peut pas être instanciée. En effet, dans la classe **Barre**, aucun attribut ne permet de déterminer la section de la barre.

Cette représentation met en évidence une hiérarchie de classes.

4.7. Application

1. En utilisant les diagrammes de séquences sur l'équilibreuse illustrant le chapitre précédent, élaborez une ébauche du diagramme de classes.
2. Faites apparaître une nouvelle classe NI6211 comme le montre la représentation suivante :



Les deux méthodes sont appelées respectivement par la lecture de l'état du capot et pour l'envoi de la consigne de vitesse du moteur.

Pour l'ensemble de l'application, il ne peut y avoir qu'une seule instance de la carte USB NI6211, complétez le diagramme de classes pour que toutes les relations apparaissent.

3. Complétez les diagrammes de séquences pour montrer l'utilisation des méthodes de la classe NI6211.

5. Diagramme états-transitions

5.1. Présentation

Le diagramme états-transitions ou diagramme de machine à états décrit le comportement d'un objet sous la forme d'un automate d'états. Ils présentent les suites d'états pris par l'instance d'une classe au cours de son cycle de vie. Pour passer d'un état à l'autre, l'objet est sollicité par divers signaux ou messages asynchrones rencontrés dans le diagramme de séquence. En réaction à ces sollicitations, il est nécessaire d'invoquer des appels de méthodes de la classe, également appelés activités.

5.2. Les états

État simple

Ils sont représentés par des rectangles avec des coins arrondis. Le nom de l'état est spécifié dans le rectangle, il doit être unique dans le diagramme.

Dans le cadre d'état simple, il ne peut y avoir qu'un seul état actif à la fois. Ici, l'état représente un mode de fonctionnement ou le système est au repos, il attend des sollicitations de l'utilisateur.

État initial

L'état initial représente le point de départ du diagramme. C'est l'état dans lequel se trouve l'objet lors de sa création. Il est représenté par un cercle noir.

Un nom comme **début** peut lui être juxtaposé.

État final

L'état final représente la fin du diagramme. Lorsque l'enchaînement des différents états se déroule normalement, le dernier état est l'état final. Il est représenté par un double cercle, celui à l'intérieur est plein.

Un nom comme **fin** peut lui être juxtaposé.

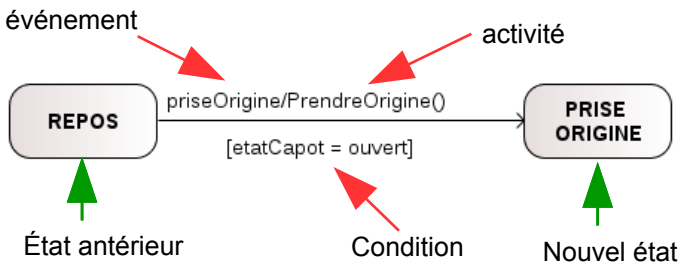
5.3. Les transitions

Les transitions sont les réactions aux sollicitations permettant le changement d'état d'un objet. Ils entraînent l'exécution d'activités particulières liées à ce changement d'état. Il peut également être soumis à une condition de garde spécifiée entre crochets comme dans les diagrammes de séquence.

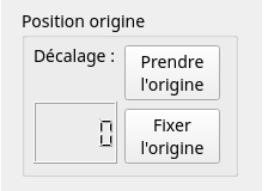
Transitions externes

Ce sont des transitions placées entre deux états de l'objet. Elles sont produites par une sollicitation extérieure à l'objet, comme une demande en provenance de l'utilisateur à travers l'IHM, ou en provenance d'un autre objet, comme le signal **timeOut** de la classe **Timer**.

Exemple :



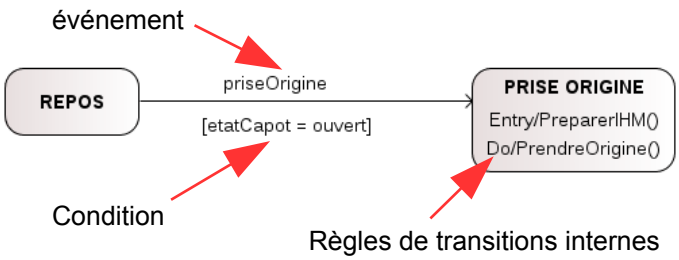
Le passage de l'état REPOS à l'état PRISE_ORIGINE est conditionné par l'événement **priseOrigine** issu du bouton de l'IHM. Cet événement engendre l'appel de la méthode **PrendreOrigine()** de la classe **Equilibreuse**. Le changement d'état est conditionné au fait que le capot soit ouvert.



La flèche représente la transition, elle est orientée dans le sens de l'enchaînement des différents états.

Transitions internes

Les transitions internes représentent des règles de déclenchement permettant de lancer diverses activités en fonction de la règle.



Cet extrait de diagramme états-transitions est équivalent à celui présenté précédemment, hors mis le fait que l'activité ai été déportée à l'intérieur de l'état derrière la règle **Do**. Elle fait suite à une règle **Entry** dont l'activité associée, la méthode **PreparerIHM()** rend accessible ou pas certains boutons de l'IHM en fonction de l'état.

Entry : est une règle qui permet de spécifier une activité exécutée lorsque l'on entre dans l'état.

Do : est une règle qui permet de réaliser une activité pendant la durée de l'état. Elle est exécutée juste après l'activité lancée par **Entry**.

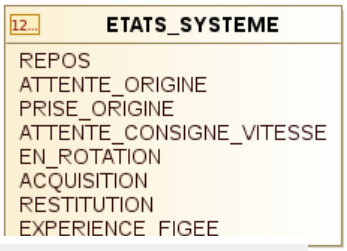
Exit : est une règle qui permet de spécifier une activité effectuée à la sortie d'un état. Elle interrompt l'activité lancée par **Do,** et est exécutée juste avant le changement d'état.

Chacune de ces règles peut également faire l'objet d'une condition de garde exprimée entre crochets.

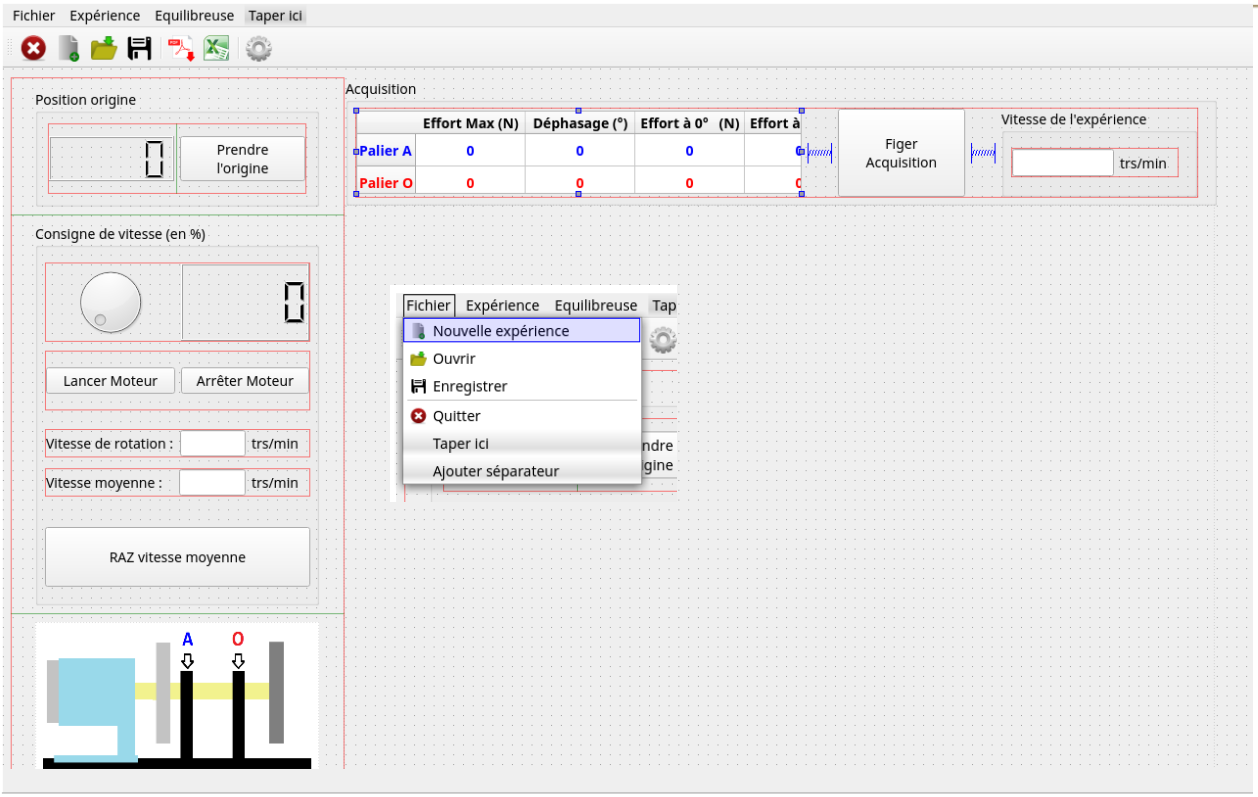
L'arrivée d'un événement peut également engendrer l'envoi d'un signal vers un autre objet par exemple.

5.4. Application

1. À partir de l'écran principal de l'Interface Homme Machine de l'équilibreuse et des états du système présentés figure ci-contre, proposez le diagramme état transition de la classe **Equilibreuse**.
2. Complétez la classe **Equilibreuse** avec les différentes méthodes et les attributs que vous trouverez au fur et à mesure de votre analyse.



IHM de l'équilibreuse



6. Diagramme de déploiement

6.1. Présentation

Les diagrammes de déploiement UML modélisent l'architecture physique d'un système, en montrant comment les composants logiciels s'exécutent sur des nœuds matériels ou virtuels. Ils sont particulièrement utiles pour représenter la disposition des systèmes dans des environnements distribués, mettant en évidence les relations entre le matériel, les logiciels, et leurs interconnexions.

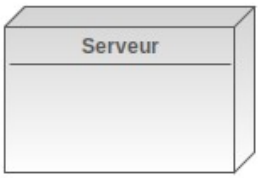
Ces diagrammes sont largement utilisés dans la documentation technique pour :

- **Planifier** l'infrastructure nécessaire à un système distribué.
- **Collaborer** entre développeurs et administrateurs système.
- **Analyser** les performances, la capacité du système à continuer de fonctionner correctement en cas de panne partielle ou d'erreur ou la sécurité d'un système.

Le diagramme de déploiement aide à visualiser les dépendances et les interactions dans le système, offrant une base solide pour identifier les faiblesses potentielles et anticiper les solutions nécessaires.

6.2. Principaux éléments d'un diagramme de déploiement

6.2.1. Les Nœuds



Un nœud représente une unité matérielle ou virtuelle capable de mémoriser et de traiter des informations.

- **Nœuds matériels** : ordinateurs, serveurs, routeurs, périphériques IoT, etc.
- **Nœuds logiciels** : conteneurs (Docker), machines virtuelles (VMWare), environnements cloud (AWS, Azure), etc.

Dans un diagramme, un nœud est souvent nommé en fonction du dispositif qu'il représente. Les nœuds peuvent également contenir d'autres nœuds, par exemple, un serveur physique hébergeant plusieurs machines virtuelles.

Exemple d'attributs pour un nœud matériel :

- Nom du matériel.
- Spécifications techniques (CPU, RAM, système d'exploitation).

6.2.2. Les artefacts

Un artefact est une unité déployable d'un logiciel, comme :

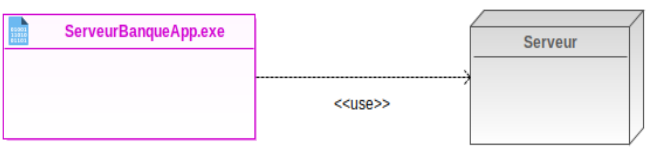
- Un exécutable (ex. : fichier `.exe`, `.jar`).
- Une bibliothèque dynamique (DLL).
- Une base de données ou un fichier de configuration.

Les artefacts sont souvent reliés à des composants logiciels abstraits qu'ils concrétisent via une **relation de manifestation**. Ces artefacts sont ensuite déployés sur des nœuds pour leur exécution.

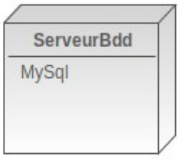
6.2.3. Les relations de déploiement

Les relations de déploiement indiquent où un artefact est déployé. Elles relient un artefact à un nœud, représentées par une flèche en pointillé avec l'étiquette « use ».

- **Exemple** : Un fichier exécutable `ServeurBanqueApp.exe` déployé sur un serveur peut être modélisé ainsi :



Les artefacts peuvent être également représentés à l'intérieur du nœud.



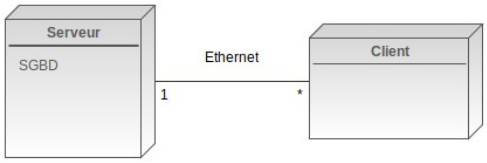
Cette relation explicite que le nœud "héberge" l'artefact, mais ne fournit pas nécessairement des services actifs à celui-ci.

6.2.4. Les relations entre nœuds

Ces relations modélisent les canaux de communication (réseaux, bus) entre différents nœuds matériels ou logiciels. Elles décrivent comment les systèmes interagissent, en ajoutant parfois des détails techniques :

- La cardinalité aux extrémités du lien indique combien de connexions ou de liens existent entre les nœuds.
  `1..*` : Un serveur connecté à un ou plusieurs clients.
  `1..1` : Une connexion unique entre deux nœuds.
  `*..*` : Une connexion possible entre un nombre indéterminé de nœuds.
- La description du type de support physique ou logique utilisé pour la communication entre les nœuds. Cela peut inclure des détails comme :

  Les technologies physiques : Ethernet, USB, Bluetooth, WiFi...
  Les protocoles de communication : TCP/IP, HTTP, HTTPS…

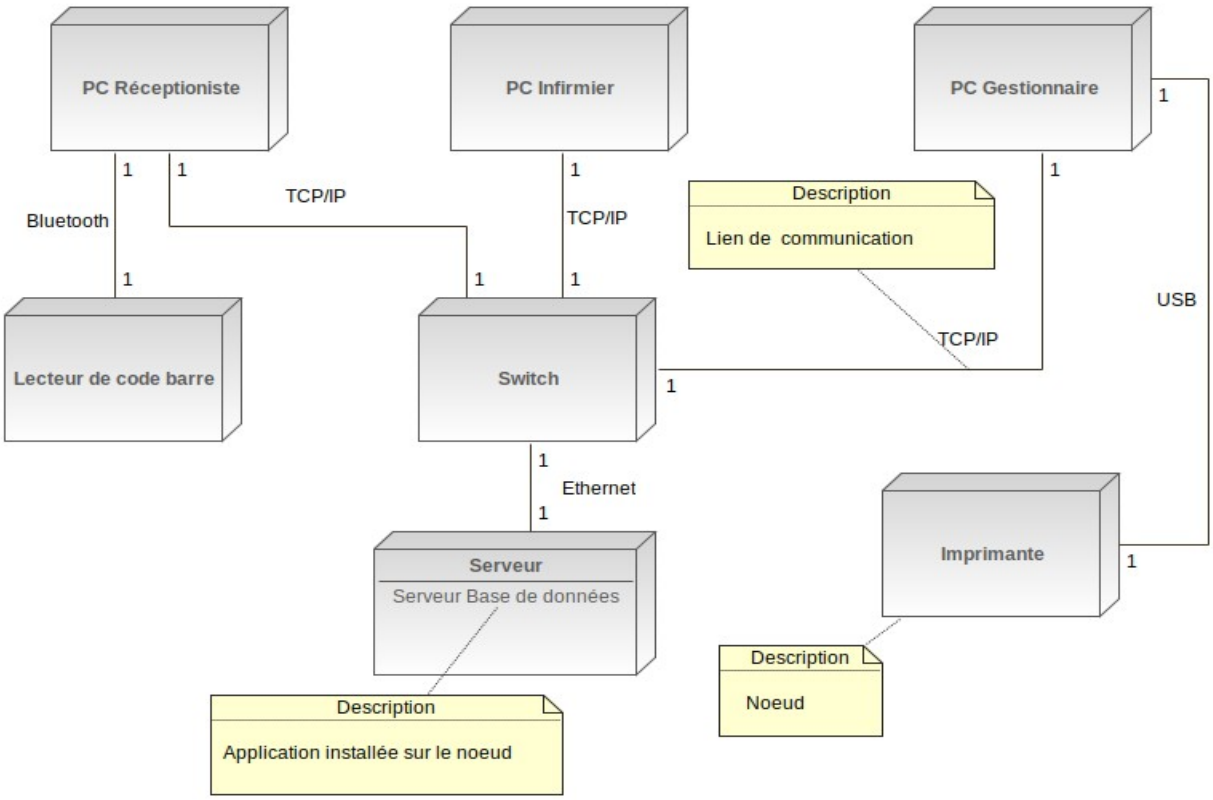


Ces liens peuvent également inclure des informations supplémentaires, telles que le débit (`10 Mbps`), la latence, ou des contraintes spécifiques (par ex., qualité de service).

6.3. Recommandations et bonnes pratiques :

- **Simplification** : Si un nœud héberge un seul artefact, le nom de l'application peut apparaître directement dans le nœud pour une meilleure lisibilité.
- **Conventions claires** : L'Utilisation de légendes ou d'annotations peut permettre de décrire facilement les liens et les technologies utilisées.
- **Modularité** : En cas de systèmes complexes, il est utile de diviser le diagramme en parties logiques (ex. : réseau interne, interactions cloud, etc.).

6.4. Exemple :



7. Modelisation SYSML

7.1. Introduction

SysML est une extension d'UML spécialement conçue pour répondre aux besoins de **modélisation des systèmes complexes**. Il permet de capturer non seulement les aspects logiciels, mais aussi les exigences, les composants matériels, les flux d'informations, et même les contraintes physiques ou de performance. Son objectif est de faciliter la conception des **systèmes multidisciplinaires**, comme dans les domaines de l'aérospatiale, l'automobile, les systèmes embarqués ou encore les systèmes médicaux.

7.2. Apport de SysML

1. **Prise en compte des exigences** : SysML propose des diagrammes spécifiques pour capturer et organiser les exigences, facilitant la traçabilité et l'alignement avec les besoins des parties prenantes. UML n'a pas de mécanismes natifs pour gérer ce type d'informations.
2. **Modélisation des systèmes physiques** : Contrairement à UML, qui est orienté principalement vers le logiciel, SysML permet de modéliser les éléments physiques d'un système comme les capteurs, les actionneurs, et même les ressources humaines.
3. **Flux d'énergie et de matière** : En plus des flux d'informations gérés par UML, SysML permet de modéliser les flux d'énergie, de matière, ou de ressources à travers un système, ce qui est essentiel pour des systèmes industriels, électriques ou mécaniques.
4. **Souplesse dans la modélisation** : SysML conserve une partie des diagrammes d'UML, comme les diagrammes de séquence, d'états ou de classes, mais il introduit également des diagrammes spécifiques aux systèmes complexes, comme les **diagrammes de blocs internes** et les **diagrammes de paramétrisation**.

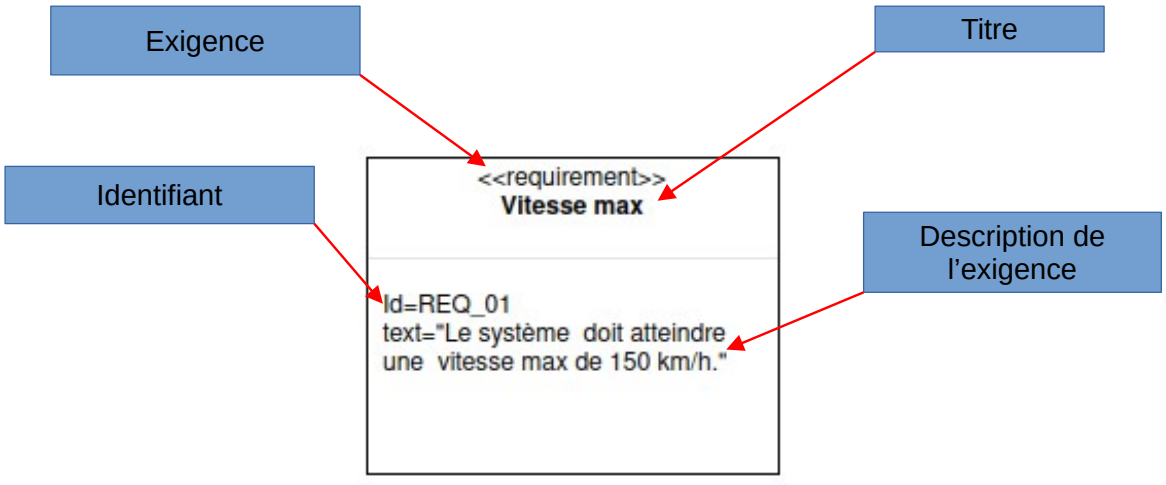
7.3. Diagramme d'exigences

Un diagramme d'exigences est une représentation graphique des exigences et de leurs relations avec d'autres exigences ou éléments du modèle système. Il décrit les fonctionnalités, les performances, les contraintes ou les interfaces du système, selon les attentes des parties prenantes. Ce type de diagramme aide à identifier les exigences à chaque étape du développement du système.

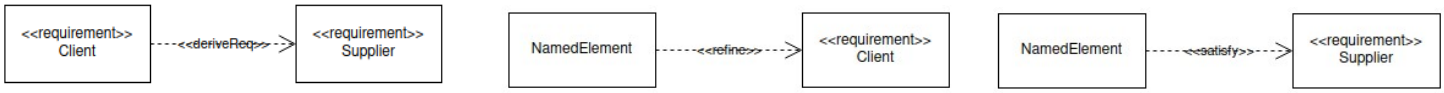
7.3.1. Éléments d'un diagramme d'exigences

Les principaux éléments utilisés dans un diagramme d'exigences SysML sont :

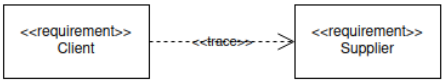
- **Exigence (Requirement)** : représente une exigence ou un besoin à satisfaire par le système. Chaque exigence est souvent décrite par un texte, un identifiant unique et d'autres attributs comme le niveau de priorité ou le type (fonctionnel, non fonctionnel, etc.). Elle est représentée par un rectangle avec le mot clé « requirement ».



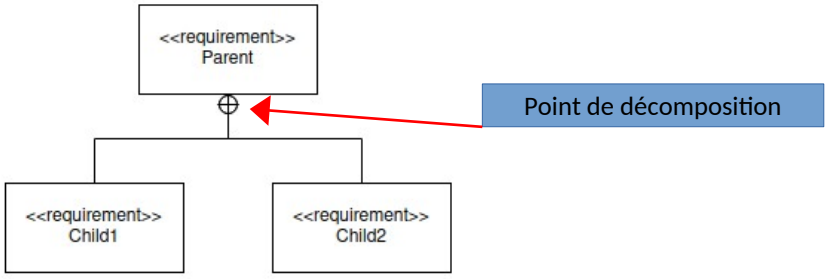
- **Décomposition (DeriveReqt, Refine, Satisfy)** : décrit les relations entre les exigences et d'autres éléments du modèle SysML. Ces relations permettent de lier des exigences de haut niveau à des sous-exigences ou de montrer comment un élément (comme une partie du système) satisfait une exigence.
    - **DeriveReqt** : une exigence est dérivée d'une autre (exigence parent → exigence enfant).
    - **Refine** : une exigence est affinée ou détaillée.
    - **Satisfy** : un élément du système satisfait l'exigence.



- **Relation de traçabilité (Trace)** : permet de tracer l'origine ou l'impact d'une exigence. Par exemple, elle peut montrer comment une exigence est liée à un test ou à une autre exigence.



- **Exigence bloquée (Containment)** : permet de grouper des exigences ou de définir une hiérarchie dans les exigences (par exemple, des exigences de niveau supérieur et leurs sous-exigences).



Ce mécanisme permet de clarifier la relation hiérarchique entre une exigence de haut niveau et ses sous-exigences. Il est particulièrement utile pour :

- **Décomposer** une exigence complexe en plusieurs sous-exigences plus simples.
- **Structurer** les exigences en plusieurs niveaux, facilitant ainsi la traçabilité et la gestion de ces dernières.

7.3.2. Types d'exigences

Les exigences peuvent être classées en plusieurs catégories, en fonction de leur nature :

- **Exigences fonctionnelles** : décrivent les fonctionnalités que le système doit fournir (exemple : "Le système doit permettre la gestion des comptes utilisateurs").
- **Exigences non fonctionnelles** : spécifient des contraintes de performance, de sécurité, d'efficacité, etc. (exemple : "Le système doit supporter jusqu'à 1000 utilisateurs simultanés").
- **Exigences de performance** : se concentrent sur les métriques de performance telles que la vitesse, la capacité, la fiabilité (exemple : "Le système doit répondre en moins de 2 secondes").
- **Exigences de sécurité** : concernent les exigences de confidentialité, d'intégrité et de disponibilité des informations.

Le type d'exigence peut-être représenté sous la forme d'un attribut, comme le texte ou identifiant dans le rectangle représentant l'exigence.

7.3.3. Relations entre les exigences

Les relations permettent de structurer et d'organiser les exigences de manière hiérarchique ou logique :

- **Include** : une exigence inclut une autre. Cela montre que l'une des exigences fait partie du comportement de base d'une autre.
- **Extend** : une exigence peut être étendue par une autre. Cela introduit un comportement optionnel sous certaines conditions.
- **Copy** : permet de créer une copie d'une exigence pour un usage dans un autre contexte tout en maintenant la relation entre les deux.

7.3.4. Outils de traçabilité des exigences

Le diagramme d'exigences en SysML aide à maintenir la traçabilité, c'est-à-dire à suivre l'évolution des exigences et leurs relations avec d'autres éléments du modèle. Cela garantit que toutes les exigences sont couvertes par le système et que leur implémentation peut être vérifiée à chaque étape.

Les relations de traçabilité peuvent inclure des liens vers :

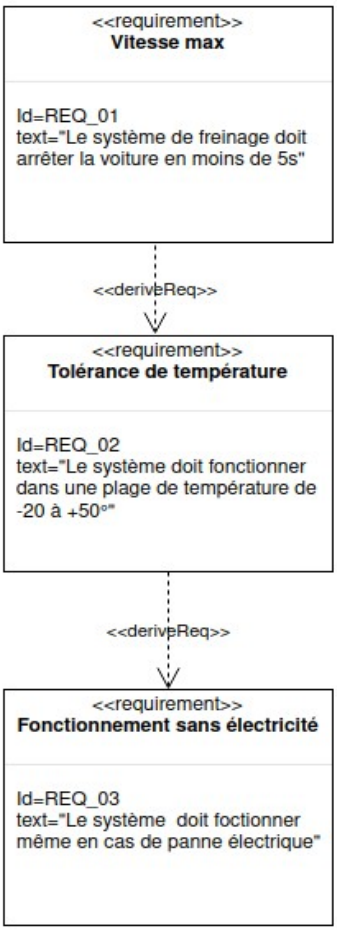
- Des cas d'utilisation (UseCase)
- Des tests de vérification
- Des architectures ou composants du système

7.3.5. Exemples de diagramme d'exigences

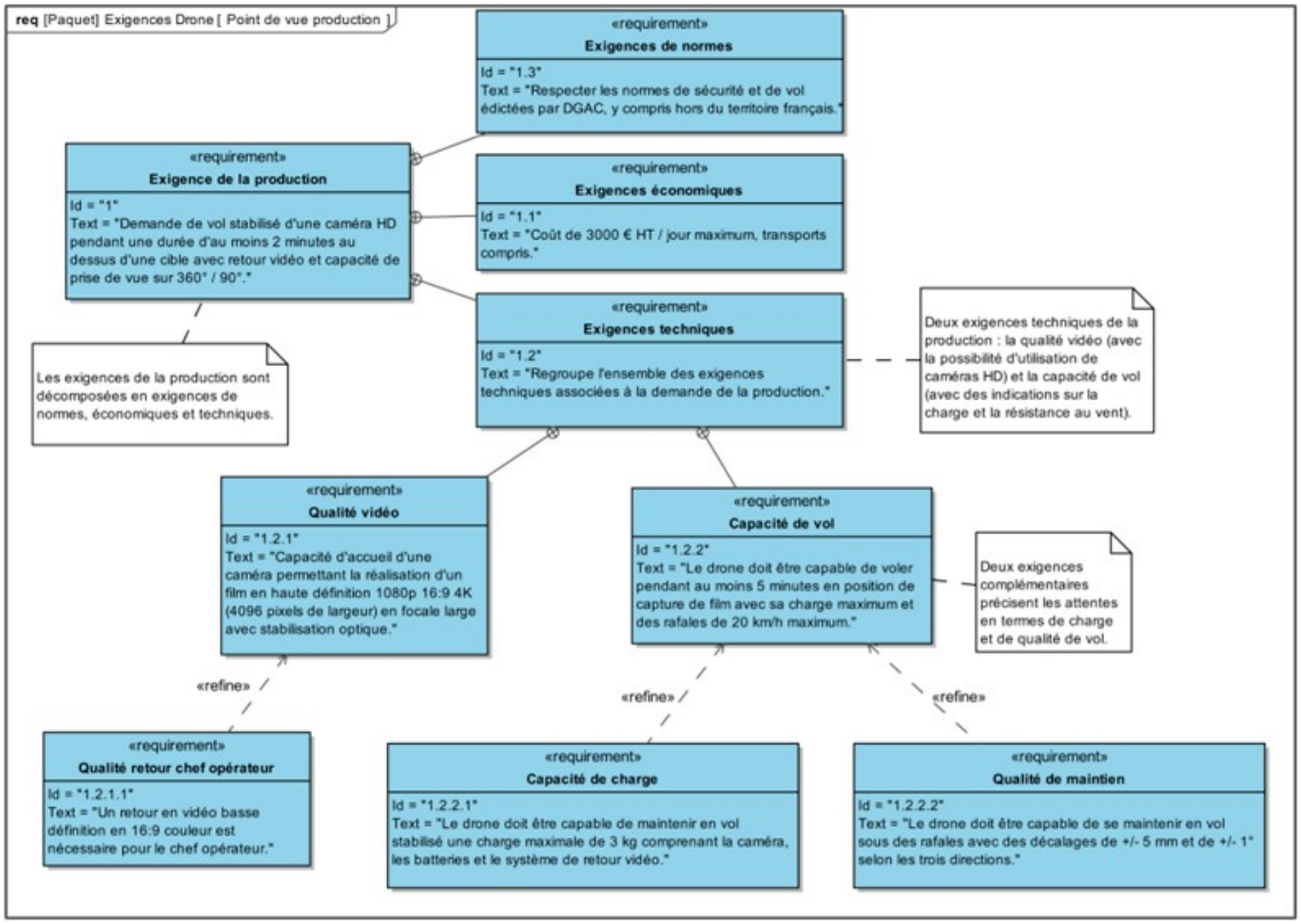
Imaginons un système de freinage pour une voiture avec les exigences suivantes :

- **Exigence principale** : "Le système de freinage doit arrêter la voiture en moins de 5 secondes".
- **Sous-exigence 1** : "Le système doit fonctionner dans une plage de température de -20 à +50°C".
- **Sous-exigence 2** : "Le système doit fonctionner même en cas de panne électrique".

Ces exigences peuvent être représentées dans un diagramme d'exigences avec des liens de dérivation (DeriveReqt) entre l'exigence principale et les sous-exigences.



Exemple 2 : Système de prise vidéo par l'intermédiaire d'un drone.



Exemple 3 :