Le génie pour l'industrie	**INF-111** Travail pratique #2 **Groupe : 5 étudiants maximum:** (un seul rapport). **Remise** : voir plan de cours **Auteur :** Frédéric Simard

1 - Introduction

1.1 - Contexte académique

Ce second devoir vise à compléter le cours en vous amenant à développer les concepts fondamentaux associés avec la réalisation de programme de petites envergures ainsi que l'utilisation de l'API Java. Il vise l'acquisition et la consolidation des connaissances suivantes:

- Création de classes
- Organisation en packages
- Implémentation de Types de Données Abstraits (TDA),
- Utilisation des collections Java,
- Introduction à l'héritage

1.2 - Avertissement

La problématique développée dans ce devoir traite d'espionnage et d'écoute électronique. L'auteur de la problématique s'est inspiré vaguement d'un système existant et actuellement utilisé par la gendarmerie royale du Canada[1]. L'auteur tient tout même à préciser que le choix du sujet ne doit en aucun cas représenter une prise de position politique, ni même refléter une opinion professionnelle. La thématique n'a été choisie que parce qu'elle est d'actualité et se prêtait bien aux exigences académiques du cours INF-111.

1.3 - Description du problème

Vous êtes membre du studio de développement de jeu vidéo: Ubihard. Le directeur de jeu s'est inspiré de l'actualité pour inventer un nouveau jeu du style espions. Vous avez la tâche de

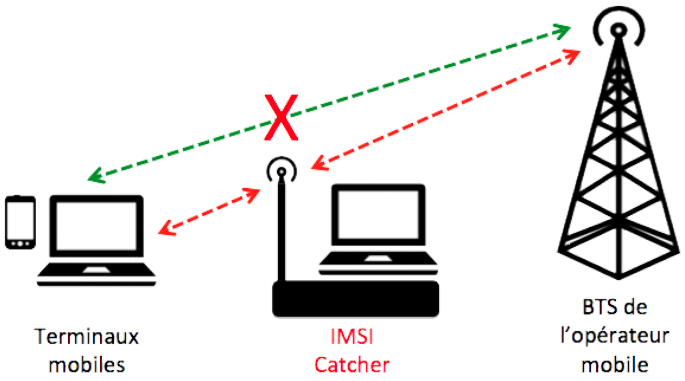
[1] https://www.lapresse.ca/actualites/justice-et-affaires-criminelles/actualites-judiciaires/201704/06/01-5085928-la-grc-admet-utiliser-des-appareils-de-surveillance-de-cellulaires.php

développer la preuve de concept permettant de tester la jouabilité, avant que l'équipe artistique ne s'embarque dans le développement de l'engin graphique.

Le projet se divise en deux phases: tout d'abord, vous allez implémenter l'engin de jeu, qui permet de tester l'interaction entre les objets (devoir 2). Ensuite, vous développerez l'interface du joueur qui permettra de tester la jouabilité à proprement dit (devoir 3).

1.4 - Description du jeu

Dans le jeu, vous êtes un membre des forces policières qui doit intercepter des communications entre des individus louches. Toutes les communications se font par cellulaire. Votre mandat vous permet d'intercepter les communications à l'aide d'un intercepteur d'IMSI[2], pour un certain nombre de numéros de téléphone, mais les individus se déplacent et sont distribués dans la carte de jeu. Vous devez donc dynamiquement positionner votre intercepteur de manière à capturer les conversations et votre pointage est basé sur le nombre de conversations que vous avez réussi à intercepter, le nombre total de conversation étant fixe et la partie se terminant lorsque les individus louches ont transmis toute leur conversation.



[2] https://fr.wikipedia.org/wiki/IMSI-catcher

2 - Plan de développement

La mécanique de jeu sera principalement implémentée au cours du devoir 3. Pour le devoir 2, vous devez développer le réseau cellulaire et le monde virtuel.

Avant de développer le programme du devoir, vous devez implémenter deux TDA

- Une file simplement chaînée
- Liste ordonnée (voir Annexe A)

Les éléments qui seront ensuite développés sont les suivants:

- Définition des fonctionnalités de base du jeu
    - La cartographie
    - Les objets cellulaires (Section 3.2)
    - Les fonctionnalités (Section 3.3)
    - La mise en action (Section 3.4)

Prenez note que la section 3.3 représente plus de 50% de la charge de travail.

2.1 - Stratégie de développement

Le développement se fait d'une manière itérative et est guidé par l'implémentation des fonctionnalités. Les fonctionnalités s'implémentent au travers de plusieurs fichiers, vous devrez compléter leur implémentation avant de les valider.

2.2 - Code fourni

Du code source vous est fourni pour démarrer votre projet. Il contient:

- une définition à compléter du gestionnaire réseau
- un engin graphique de base qui vous permettra de visualiser votre monde, à la fin du devoir. (package vue)
- un gestionnaire de scénario, qui requiert la file simplement chaînée avant d'être fonctionnel.
- package observer, qui vous sera expliqué plus tard, mais qui est requis pour l'interface graphique.
- un programme principal que vous n'avez pas à modifier

2.3 - Nomenclature

Tout au long de l'énoncé, l'introduction des classes se fera en suivant la nomenclature *package::Classe*. Vous devez donc définir des packages là où nécessaire.

2.4 - Architecture logicielle

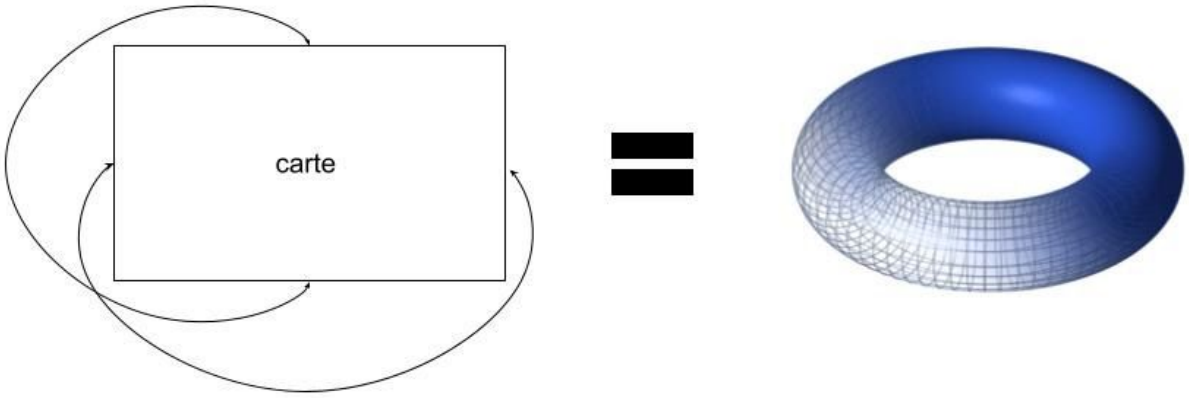
Le projet est organisé selon l'architecture Model-View-Controller (MVC), qui vous sera expliqué plus en détails lors du devoir 3.

3 - Définition du monde de jeu

Cette section vous amène à développer la physique du jeu et les fonctionnalités de communications de base.

3.1 - Physique du jeu

Les éléments physiques du jeu possèdent une position et évoluent dans une carte définie. La carte est toroïdale (a.k.a. un beigne), ce qui veut dire qu'un objet qui sort de la carte à droite, réapparaît à la gauche et pareillement pour le haut et le bas.



3.1.1 - modele.physique::Position

La classe Position contient les attributs suivants:

- position en x (double)
- position en y (double)

Elle offre les services suivants:

- constructeur par paramètre
- accesseurs informateurs et mutateurs pour les deux attributs
- une méthode retournant la distance entre le point et un autre point reçu en paramètre
- une méthode toString pour vos besoins en déverminage

Implémenter une courte routine de test qui instancie un point, puis l'affiche dans la console.

3.1.2 - modele.physique::Carte

Le module utilitaire carte ne contient que des éléments statiques. Les attributs

- une constante pour définir la taille de la carte (1920x1080, utiliser une position)
- une instance de Random

Elle offre les services suivants:

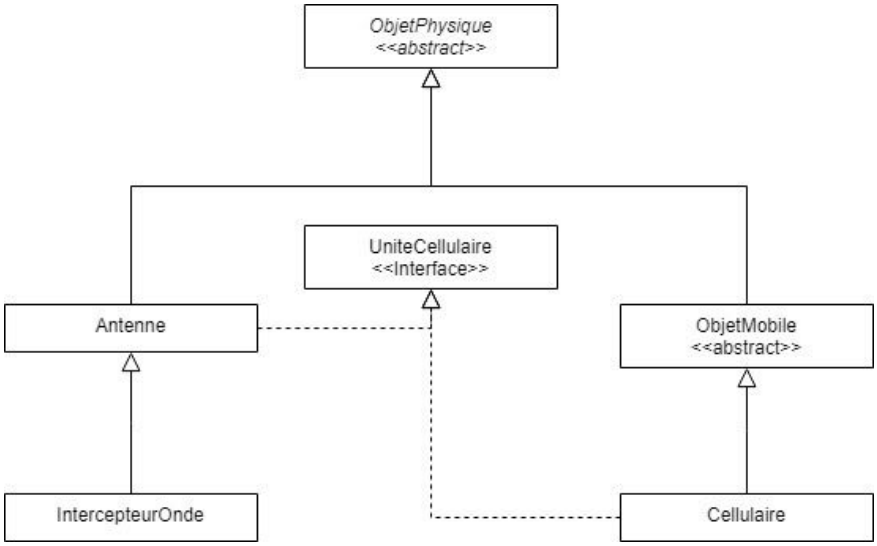
- générateur de position aléatoire
  - retourne une position qui a été initialisée aléatoirement dans la carte.
- une méthode permettant d'ajuster une position
  - reçoit une position en entrée, si la position est à l'extérieur de la carte, ses coordonnées X et/ou Y sont ajustées, par rapport à la logique du toroïde, pour la ramener dans la carte.

Implémenter une courte routine de test qui:

- instancie un point aléatoirement
- affiche ce point
- déplace le point à l'extérieur de la carte
- ajuste le point
- affiche le point à nouveau et confirme que l'ajustement s'est déroulé tel que spécifié

3.2 - Objets cellulaires

Les objets cellulaires utilisés dans le jeu sont les suivants: Cellulaire, Antenne et IntercepteurOnde. Par contre, leur implémentation tire profit du concept d'héritage. Voici l'arborescence de leur définition.



3.2.1 - modele.physique::ObjetPhysique

L'objet physique contient:

- une position

et offre:

- un constructeur par paramètre

- un accesseur informateur sur la position

3.2.2 - modele.physique::ObjetMobile

L'objet mobile ajoute une logique de mouvement dirigé aléatoirement à l'objet physique. Les attributs supplémentaires sont:

- direction en radian (initialisé à 0)
- vitesse (en pixels par itération)
- déviation standard de la direction (en radian)

La classe offre:

- un constructeur par paramètre (la direction est exclue)
- une méthode *seDeplacer* qui implémente l'algorithme suivant:

$$direction(t+1) = direction(t) + gaussienne() * déviation\ standard$$
$$pos_x(t+1) = pos_x(t) + vitesse * cos(direction(t))$$
$$pos_y(t+1) = pos_y(t) + vitesse * sin(direction(t))$$
$$Carte -> ajustement\ position(pos)$$

NOTE: gaussienne() est un nombre tiré aléatoirement selon une distribution gaussienne. À vous de trouver quelle méthode de la librairie Java utiliser.

Dans votre package de test définissez une classe héritant de ObjetMobile. Implémenter ensuite une courte routine de test qui instancie cet objet, puis qui valide que la position change lors de chaque appel à *seDeplacer*.

3.2.3 - modele.reseau::UniteCellulaire

Cette interface déclare les méthodes suivantes (NOTE: certains types n'existent pas encore):

- **appeler**
  - Entrées
    - numéro appelé, String
    - numéro appelant, String
    - antenne connecté, Antenne
  - Sortie
    - entier, indiquant le numéro de connexion

- **repondre**
  - Entrées
    - numéro appelé, String
    - numéro appelant, String
    - numéro de connexion, entier
  - Sortie
    - Cellulaire, référence au cellulaire qui répond

- **finAppelLocal**
  - Entrées
    - numéro appelé, String
    - numéro de connexion, entier
  - Sortie
    - aucune

- **finAppelDistant**
  - Entrées
    - numéro appelé, String
    - numéro de connexion, entier
  - Sortie
    - aucune

- **envoyer**
  - Entrées
    - message, Message
    - numéro de connexion, entier
  - Sortie
    - aucune

- **recevoir**
  - Entrées
    - message, Message
  - Sortie
    - aucune

3.2.3 - Description d'un lien cellulaire

NOTE: Cette section est descriptive et ne vous demande aucune implémentation.

Un lien cellulaire s'établit de la manière suivante:

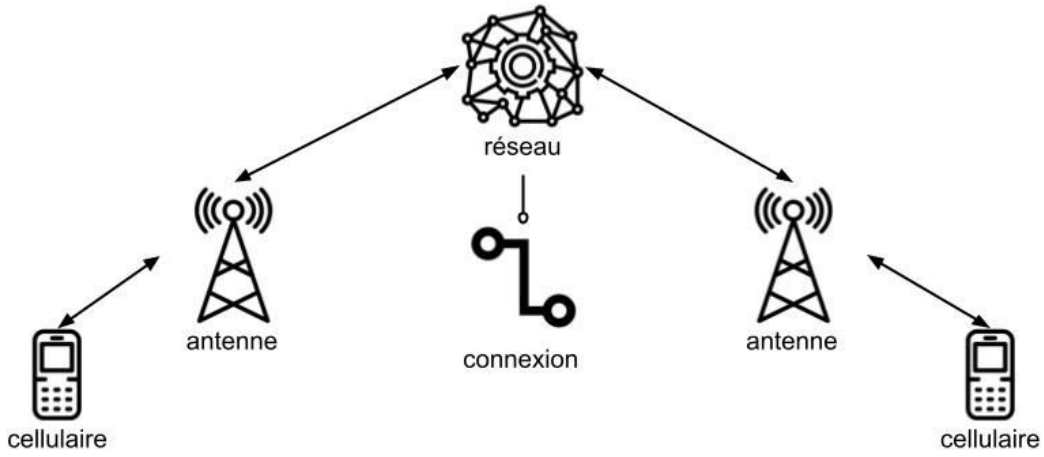
- Un cellulaire est connecté à une antenne
- l'antenne est connecté au réseau
- le réseau contient toutes les connexions, dont celle qui indique à quelle antenne ce lien cellulaire est connecté
- l'antenne est connectée à l'autre cellulaire.

Une fois connecté, le lien est bidirectionnel. Les cellulaires peuvent tous deux envoyer et recevoir des messages et tous deux peuvent interrompre la connexion.

Les étapes sont donc:

- établir un lien
- échanger des messages
- raccrocher

Les sous-sections suivantes vous amènent à implémenter toutes ces opérations.



3.2.4 - Définition initiale des classes

Cette section vous amène à écrire une définition de base de chacune des classes requises par le programme. Les sections suivantes vous amènent dans l'implémentation par fonctionnalités.

modele.reseau::Cellulaire

- dérive de ObjetMobile et implémente l'interface UniteCellulaire
  - fournir des définitions de méthodes vides (stub), vous compléterez plus tard.
- Constantes:
  - NON_CONNECTE = -1;
  - PROB_APPELER = 0.05;
  - PROB_ENVOI_MSG = 0.2;
  - PROB_DECONNEXION = 0.1;
- Attributs
  - numeroLocal, String
  - numeroConnexion, entier (par défaut: NON_CONNECTE)
  - numeroConnecte, String (null)
  - antenneConnecte, Antenne
  - une instance de Random

    - une copie de l'instance du gestionnaire réseau (voir: getInstance())
- Constructeur par paramètre, qui reçoit:
    - le numéro local,
    - la position,
    - la vitesse
    - la déviation standard.
- Accesseurs informateurs pour les champs:
    - numeroLocal
    - numeroConnexion
- Méthode permettant de savoir si le cellulaire est connecté (opération sur numeroConnexion)
- *comparerNumero*, qui compare un numéro reçu en paramètre au numéro local.
- toString qui indique le numéro local et la position

modele.reseau::Antenne

- dérive de ObjetPhysique et implémente l'interface UniteCellulaire
    - fournir des définitions de méthodes vides (stub), vous compléterez plus tard.
- Attributs
    - une copie de l'instance du gestionnaire réseau (voir: getInstance())
    - un TDA:liste statique contenant des Cellulaires
- Constructeur par paramètre recevant:
    - une position
- Services suivants:
    - distance, méthode exposant la méthode distance équivalente définie dans Position
    - toString qui indique la position

modele.communication::Connexion

- Constantes:
    - NB_ANTENNES= 2;
- Attributs
    - numéro de connexion (int)
    - tableau d'Antenne
- Constructeur par paramètre recevant:
    - un numéro de connexion
    - une antenne
    - une autre antenne
- Services
    - accesseur informateur numéro de connexion
    - méthode equals() comparant les numéros de connexion

modele.communication::Message

- Attributs

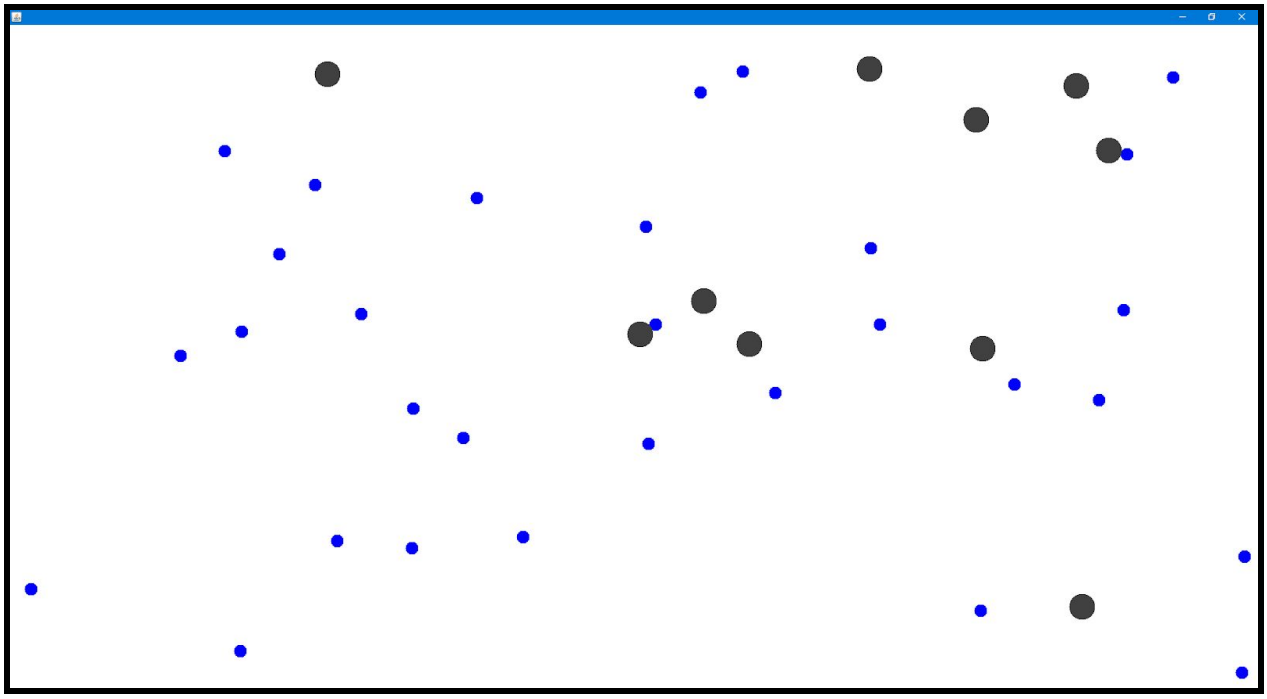
    - numéro de destination (String)
    - message (String)
- Constructeur par paramètre recevant:
    - numéro de destination
    - message
- Services
    - accesseur informateur pour les deux attributs

modele.reseau::Reseau

Un gestionnaire réseau incomplet vous est fourni, vous devez y ajouter les éléments suivants:

- constantes:
    - PERIODE_SIMULATION_MS = 100
    - VITESSE = 10.0
    - DEVIATION_STANDARD = 0.05
    - NB_CELLULAIRES = 30
    - NB_ANTENNES = 10
    - CODE_NON_CONNECTE = -1
- attributs:
    - instance de Random
    - Collection d'antennes
    - Collection de cellulaires
    - Une liste ordonnée (voir annexe A) contenant des connexions
- méthodes privées
    - une méthode permettant de créer des Antennes et de les ajouter dans la collection
    - une méthode permettant de créer des Cellulaires et de les ajouter dans la collection
        - NOTE: utiliser GestionnaireScenario::obtenirNouveauNumeroStandard(), pour obtenir des numéros de téléphones
- Services
    - get sur la collection d'antennes (retourne référence, tout simplement)
    - get sur la collection de cellulaires (retourne référence, tout simplement)

Activé le code en commentaires dans la méthode run() et lancez le programme principal. Vous devriez voir apparaître le monde du jeu avec les antennes (cercles gris) et les cellulaires (cercles bleus) qui se déplacent.



3.3 - Implémentation par fonctionnalitées

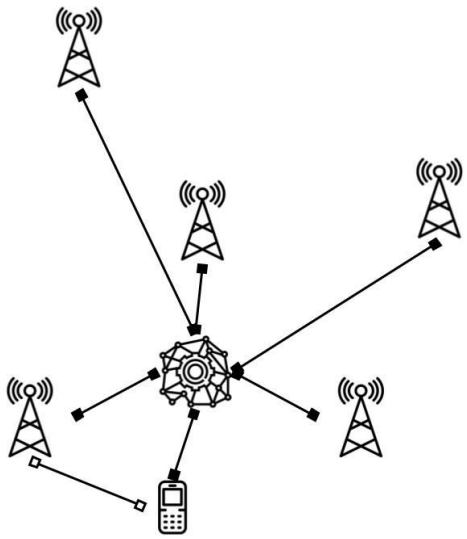
Cette section vous guide dans l'implémentation des fonctionnalités suivantes:

- connecter dynamiquement Cellulaire et Antenne
- établir une connexion
- mise à jours des connexions, suivant changements d'antennes
- échanger des messages
- terminer un appel

L'implémentation de ces fonctionnalités implique d'apporter des modifications à chacun des éléments de la chaîne de communication.

3.3.1 - Étape 1 - lien cellulaire-antenne

La première étape à programme le système qui connecte les Cellulaires et les Antennes. La logique est simple, chaque cellulaire doit toujours être connecté à l'antenne la plus proche.



La fonctionnalité se réalise de la manière suivante:

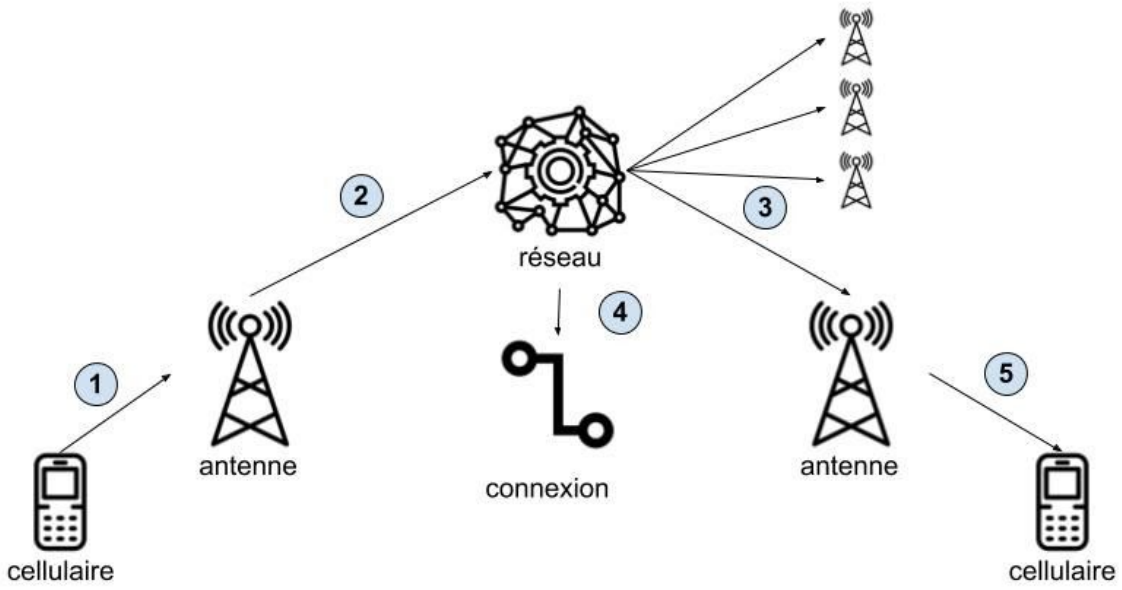
- Le **réseau** doit offrir un service permettant d'obtenir une référence sur l'antenne la plus proche (à partir d'une position)
- L'**Antenne** doit avoir deux services supplémentaire: un permettant d'ajouter un cellulaire à la collection et l'autre permettant d'enlever un cellulaire à la collection
- Le **Cellulaire:**
    - appelle la méthode pour obtenir l'antenne la plus proche à chaque tour (voir méthode Cellulaire:effectuerTour()).
    - Si l'antenne la plus proche est différente de l'antenne connecté, le Cellulaire se s'enlève de l'antenne connecté et s'ajoute à la nouvelle antenne connecté. Le cellulaire met également sa référence à l'antenne connectée à jour.

NOTE: l'antenne connectée doit également être initiée dans le constructeur.

Pour valider, changer les valeurs des constantes pour avoir 1 seul Cellulaire, mais un grand nombre d'antennes. Afficher le cellulaire et l'antenne connecté à répétition dans la console et valider que le Cellulaire se connecte à une antenne différente et plus proche fréquemment.

3.3.2 - Étape 2 - établissement d'une connexion

Établir une connexion consiste à connecter un Cellulaire à un autre, tout en ajoutant une Connexion permettant de facilement transmettre les messages par la suite.



La fonctionnalité utilise les méthodes déclarées par l'interface de l'unité Cellulaire.

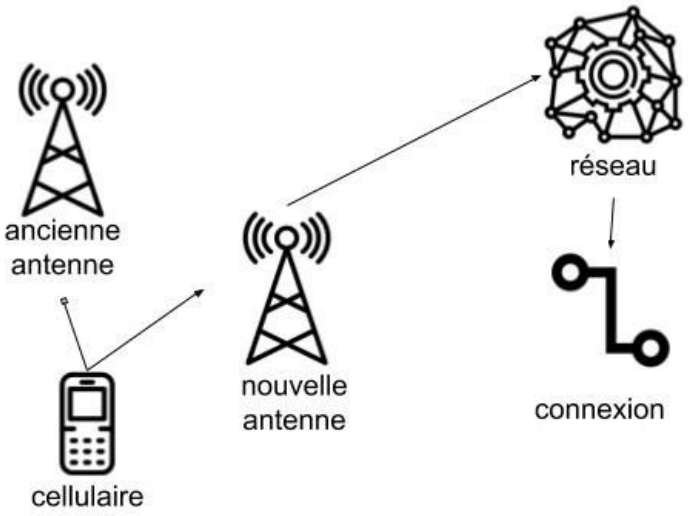
- (1) Lorsque la méthode **Cellulaire**::appeler est lancé, le cellulaire appel la méthode Antenne::appeler() appartenant à son antenne connecté.
    - NOTE: utiliser GestionnaireScenario.obtenirNumeroStandardAlea(), pour obtenir un numéro de destination valide
- (2) La méthode **Antenne**::appeler() appel la méthode **GestionnaireReseau**::relayerAppel (à définir)
- la méthode **GestionnaireReseau**::relayerAppel effectue les opérations suivantes:
    - obtient un numéro de connexion unique (à vous de le faire)
    - (3) parcours toutes les Antennes en appelant leur méthode répondre
    - si l'une d'entre elle retourne une référence valide. (5)
        - (4) ajoute une nouvelle connexion à la liste ordonnée
            - la connexion obtient des références à l'antenne source et l'antenne destination
        - retourne le numéro de connexion
    - sinon, retourne le CODE_NON_CONNECTE
- (5a) la méthode **Antenne**::repondre effectue les opérations suivantes:
    - parcours tous les cellulaires enregistré à cette antenne
        - compare le numéro cherché avec le numéro des cellulaire
        - s'il y a une correspondance
            - appel la méthode Cellulaire::repondre
        - si aucun cellulaire ne répond, retourne une référence nulle.
- (5b) la méthode **Cellulaire**::repondre effectue les opérations suivantes:

- Si le Cellulaire n'est pas connecté déjà
  - enregistre le numéro appelant
  - enregistre le numéro de connexion
  - retourne une référence au cellulaire (this)
- Sinon, retourne une référence nulle

Pour valider, changer les valeurs des constantes pour avoir 2 Cellulaires, et plusieurs antennes. Valider qu'une connexion est créée et établie entre les cellulaires en affichant des messages dans les méthodes qui s'exécutent dans les différentes classes de la chaîne.

3.3.3 - Étape 3 - Mise à jours des connexions suivant changement d'antennes

Les connexions ont des références aux deux antennes d'un lien cellulaire. Par contre, les liens cellulaire-antenne changent au fil du temps. Lorsque cela arrive, il faut mettre à jour la connexion.



- Dans la classe Connexion, ajoutez une méthode Connexion::miseAJourAntenne permettant de remplacer une ancienne antenne par une nouvelle.
- Dans la classe gestionnaire réseau, ajoutez une méthode privée qui permet d'obtenir une référence à une Connexion à partir de son numéro.

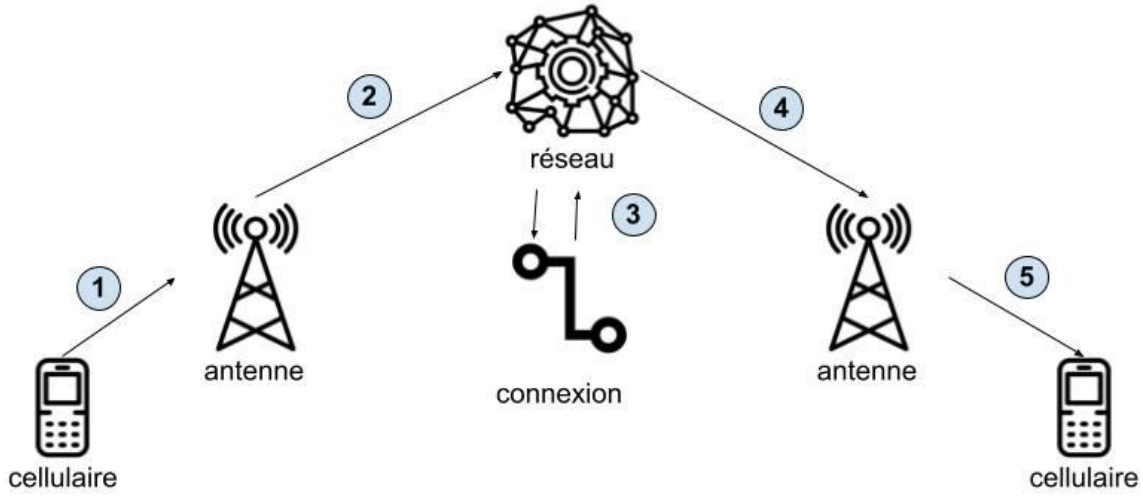
- ajouter une méthode mettreAJourConnexion à Antenne et GestionnaireRéseau
  - Celle d'Antenne reçoit le numéro de connexion et la référence à l'ancienne Antenne et appelle la méthode GestionnaireRéseau::mettreAJourConnexion, en utilisant *this*, comme nouvelle antenne.

    - Celle de gestionnaire réseau reçoit le numéro de connexion, la référence à l'ancienne Antenne et la référence à la nouvelle antenne.
        - la méthode obtient la connexion et mets à jours l'antenne
- Le Cellulaire doit appeler Antenne::mettreAJourConnexion à chaque changement d'antenne (voir effectuerTour()). La méthode à appeler est celle associée à l'instance de la nouvelle antenne, et le cellulaire doit passer la référence à l'ancienne antenne au moment de l'appel.

Pour valider, changer les valeurs des constantes pour avoir 2 Cellulaires, et plusieurs antennes. Valider qu'après qu'une connexion soit créé, les messages envoyés par un cellulaire soient bien reçus par l'autre cellulaire.

3.3.4 - Étape 4 - Échange de messages

L'échange de message se propage d'une manière similaire à l'établissement d'une connexion, mais utilise la connexion pour éviter d'avoir à chercher à quelle antenne appartient le Cellulaire appelé.



- Avant d'envoyer un message, un **Cellulaire** doit avoir établi une connexion.
- Pour envoyer un message, le cellulaire:
    - créer un **Message**, utiliser **GestionnaireScenario**.obtenirMessage(numeroLocal), pour obtenir un message aléatoire.
    - la méthode appeler de **Cellulaire**, appel à méthode **Antenne**::envoyer
    - la méthode **Antenne**::envoyer appelle la méthode GestionnaireReseau::relayerMessage
    - la méthode relayerMessage utilise le registre de connexions pour déterminer l'antenne destination, puis appel Antenne::recevoir

    - Antenne::recevoir trouve le Cellulaire, puis appelle Cellulaire::recevoir
    - Cellulaire::recevoir peut affiche le message à l'écran pour fin de déverminage

Pour valider, changer les valeurs des constantes pour avoir 2 Cellulaires, et plusieurs antennes. Valider qu'une connexion est créée et établie entre les cellulaires en affichant des messages dans les méthodes qui s'exécutent dans les différentes classes de la chaîne.

3.3.5 - Étape 5 - Raccrocher

À ce moment-ci, vous devriez avoir une idée de comment le réseau fonctionne. Pour terminer un appel, suivre le chemin suivant:

- Cellulaire::finAppelLocal
- Antenne::finAppelLocal
- GestionnaireReseau::relayerFinAppel
- Antenne::finAppelDistant
- Cellulaire::finAppelDistant

Il ne faut pas oublier d'enlever la connexion.

Pour valider, vérifier que les cellulaires peuvent se connecter et se déconnecter à répétition.

3.4 - Mise en action, Cellulaire

À chaque tour, chaque cellulaire effectue les opérations suivantes:

- se déplace
- mets à jours l'antenne connecté
- si connecté à un autre cellulaire déjà
    - probabilité d'envoyer un message
    - sinon, probabilité de raccrocher
- sinon, probabilité de faire un appel

4 - Conclusion et Remise

Faites le ménage de votre code, tous les sous-programme de validation doivent être fournis avec le programme, mais seul le dernier sur lequel vous avez travaillé doit être actif dans le programme principal.

Attention, prenez bien soin de limiter les println présent dans le code à ceux qui servent à valider la dernière fonctionnalité sur laquelle vous avez travaillé.

Annexe A - Liste ordonnée

Une liste ordonnée est une liste telle que nous l'avons vue en classe, mais pour laquelle les éléments sont gardés en ordre.

Pour le devoir, vous devez implémenter une liste statique, pour laquelle les entrées sont insérés de manière à ce qu'elle soit toujours en ordre. Votre liste contient des Connexions et leur ordre doit être définie par leur numéro de connexion.

La méthode doit également offrir une méthode permettant d'obtenir une connexion, à partir du numéro. Cette méthode doit implémenter une recherche binaire.

Bonne chance!