|  |  |
| --- | --- |
|  | **INF-111**  Travail pratique #2  **Groupe : 5 étudiants maximum:** (un seul rapport).  **Remise** : voir plan de cours  **Auteur :** Frédéric Simard  **Adaptation** : Pierre Bélisle  Copyright H2021 |

# Devoir2 (Partie 1)

# 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]](#footnote-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 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]](#footnote-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 est fixe et la partie se termine lorsque les individus louches ont transmis toutes leurs conversations.

# 

# 2 - Plan de développement

Pour cette phase, les éléments qui seront développés sont les suivants:

* Définition des fonctionnalités de base du jeu (Section 3, semaine 1)
  + La cartographie
  + Les objets cellulaires
  + La gestion des appels
  + Le gestionnaire de scénarios
* Définition du système de communication (Section 5, semaine 2-3)
  + La gestion des connexions
  + L’échange de messages
* Assemblage finale de l’engin de jeu (Section 6, semaine 3)

## 2.1 - Stratégie de développement

Le développement se fait d’une manière itérative et est guidé par l’implémentation des fonctionnalités. Vous devez développer des tests pour démontrer et valider les fonctionnalités développées.

Vous devez fournir le programme final, mais également les tests que vous réalisez au cours du développement.

Prenez note que la section 5 est probablement la plus difficile à réaliser. Assurez-vous de passer rapidement au travers des sections 3 et 4, pour vous donner du temps à développer la section 5.

## 2.2 - Code fourni

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

* une déclaration à compléter de la carte,
* une déclaration à compléter du gestionnaire réseau,
* un engin graphique de base qui permet de visualiser le monde, que vous avez programmé, à la fin du devoir.
* un gestionnaire de scénario, qui requiert des déclarations 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.

\*\*\* Rappelez-vous que c’est un cours de programmation, on vous demande de programmer ce qui a été conçu.

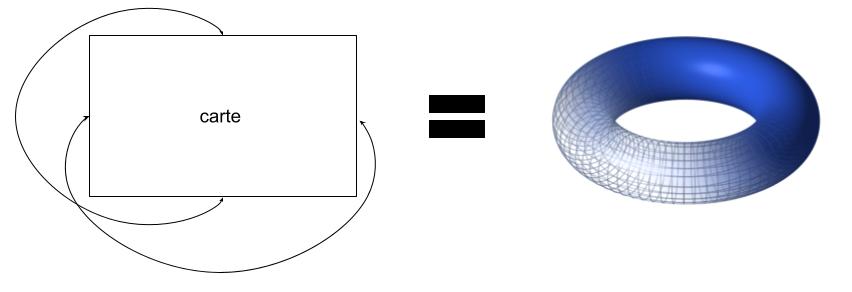
## 2.3 – Nomenclature

Tout au long de l’énoncé, l’introduction des classes se fera en suivant la nomenclature package::Classe. Vous devrez donc définir des packages là où c’est mentionné.3 – Classe simple

Cette section vous amène à développer la physique du jeu et les fonctionnalités de communications de base. Elle est décrite en approche ascendante. Cela signifie que vous décrivez les classes avant de vous en servir. Il est donc possible que vous ne voyiez l’utilité d’une méthode que plus tard dans le développement.

## 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.



### Les programmes tests doivent être rendus avec le travail, bien découpés en sous‑programmes et commentés (voir test unitaire en annexe).

### 3.1.1 – modele.physique ::Point2D

Utilisez la classe fournie à la semaine 5. Vous devez écrire un programme qui teste les méthodes qui s’y trouvent et qui démontrent par programmation qu’elles fonctionnent.

### 3.1.2 - modele.physique::Carte

Le module utilitaire carte ne contient que des éléments statiques. Elle contient comme attributs :

* une constante pour définir la taille de la carte (1920, 1080, utiliser un objet Point2D)
* une instance de Random

Elle offre les services suivants:

* un générateur de position aléatoire
  + retourne une position qui a été initialisée aléatoirement dans la carte.
* une méthode permettant d’ajuster la position. Elle 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.

Vous devez écrire un programme qui teste les méthodes qui s’y trouvent. Une courte routine de test devrait suffire:

* instancier un point aléatoirement
* afficher ce point
* déplacer le point à l’extérieur de la carte
* ajuster le point
* afficher le point à nouveau et confirmer que l’ajustement s’est déroulé tel que spécifié

# 4- Les collections

Nous aurons besoin de conserver des informations. Nous vous demandons d’implémenter une classe pour maintenir les conversations ordonnées sur le numéro de téléphone appelant.

Le tableau qui conserve les données doit être statique et d’une taille minimale de 100 cases par défaut. Un constructeur par paramètre permettra d’obtenir une instance de liste d’une taille plus grande.

Les méthodes publiques à implémenter sont :

* void inserer(Object element) // Insère l’élément à sa position dans la liste
* void retirer(Object element) // Retire l’élément en utilisant la fouille binaire
* Object getElement(int position) // Retourne null ou l’élément à la position fournie
* Ajouter toutes autres méthodes qui seront nécessaires au projet dès que vous les découvrez.

Écrivez un programme indépendant qui démontre que vos méthodes fonctionnent.

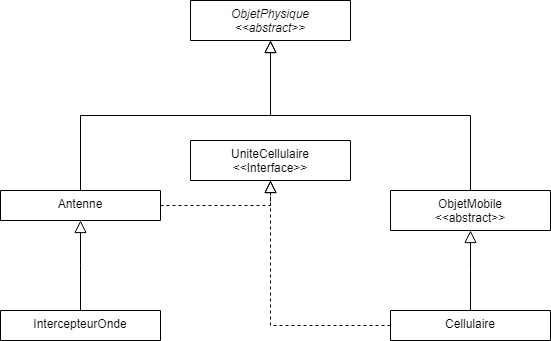
# (Partie 2)

Cette section pourra être implémenté après la présentation théorique sur l’héritage et l’implémentation d’interfaces de Java.

# 5 - La hiérachie de classes

## 5.1 - 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 (évite la répétition de code) et d’implémentation d’interface (bonne pratique). Voici l'arborescence de leur déclaration.



### 5.1.1 - modele.physique::ObjetPhysique

L’objet physique contient peu de choses mais est une classe abstraite:

* une position (L’attribut commun à tous les objets de cette hiérarchie).

et offre:

* un constructeur par paramètres
* un accesseur sur la position

### 5.1.2 - modele.physique::ObjetMobile

L’objet mobile est une classe abstraite qui ajoute une logique de mouvement dirigé aléatoirement à l’objet physique. Les attributs supplémentaires à un objetPhysique sont:

* une direction en radian (initialisé à 0)
* une vitesse (sans unité physique)
* une déviation standard de la direction

La classe offre:

* un constructeur par paramètres (la direction est exclues)
* une méthode *seDeplacer* qui implémente l’algorithme suivant:

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 déclarez une classe héritant de la classe ObjetMobile. Implémentez ensuite une courte routine de test **(à rendre)** qui instancie un objet de cette classe, puis qui valide que la position change lors de chaque appel à *seDeplacer*.

### 5.1.3 - modele.reseau::UniteCellulaire

Cette interface déclare les méthodes suivantes décrites ici mais que vous devez écrire en Java:

* **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

### 5.1.4 - 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:

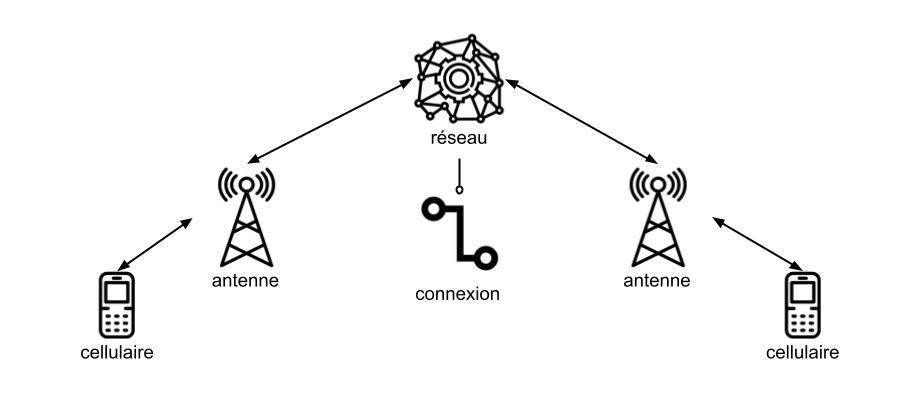
* Un cellulaire est connecté à une antenne
* l’antenne est connecté au réseau
* le réseau contient toutes les connexions, dont celle qui indique à quelle antenne ce lien cellulaire est connecté.
* l’antenne est connectée à l’autre cellulaire.

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.



### 5.1.5 - Étape 1 - déclaration initiale des classes

Cette section vous amène à écrire une déclaration de base de chacune des classes requises par le programme. Les sections suivantes vous amènent dans l’implémentation par fonctionnalités. Suivez les étapes pas à pas en prenant le temps de bien commenter au fur et à mesure pour ne pas avoir à revenir écrire des commentaires lorsque le tout fonctionnera. Il restera à lire et vérifier leur cohérence à la fin, ce qui est moins long.

**modele.reseau::Cellulaire**

* dérive de ObjetMobile et implémente l’interface UniteCellulaire
  + fournir des déclarations de méthodes vides (squelette (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 paramètres, qui reçoit:
  + le numéro local,
  + la position,
  + la vitesse
  + la déviation standard.
* Accesseurs pour les champs:
  + numeroLocal
  + numeroConnexion
* Méthode booléenne permettant de savoir si le cellulaire est connecté (opération sur numeroConnexion).
* *equals*, qui compare un numéro du **Cellulaire** en paramètre, au numéro local.
* toString qui retourne le numéro local et la position en String dans le format désiré.

**modele.reseau::Antenne**

* dérive de ObjetPhysique et implémente l’interface UniteCellulaire
  + Comme pour **Cellulaire**, fournir des déclarations de méthodes vides que vous compléterez plus tard.
* Attributs
  + une copie de l’instance du gestionnaire réseau (voir: getInstance())
  + un TDA:liste statique contenant des Cellulaires.
* Constructeur par paramètres recevant:
  + une position
* Services suivants:
  + distance, méthode exposant la méthode distance équivalente définie dans Position.
  + toString qui retourne la position en String.

**modele.communication::Connexion**

* Constantes:
  + NB\_ANTENNES= 2;
* Attributs
  + numéro de connexion (int)
  + collection d’Antenne
* Constructeur par paramètres recevant:
  + un numéro de connexion
  + deux antennes
* Services
  + accesseur du numéro de connexion
  + equals() comparant les numéros de connexion

**modele.communication::Message**

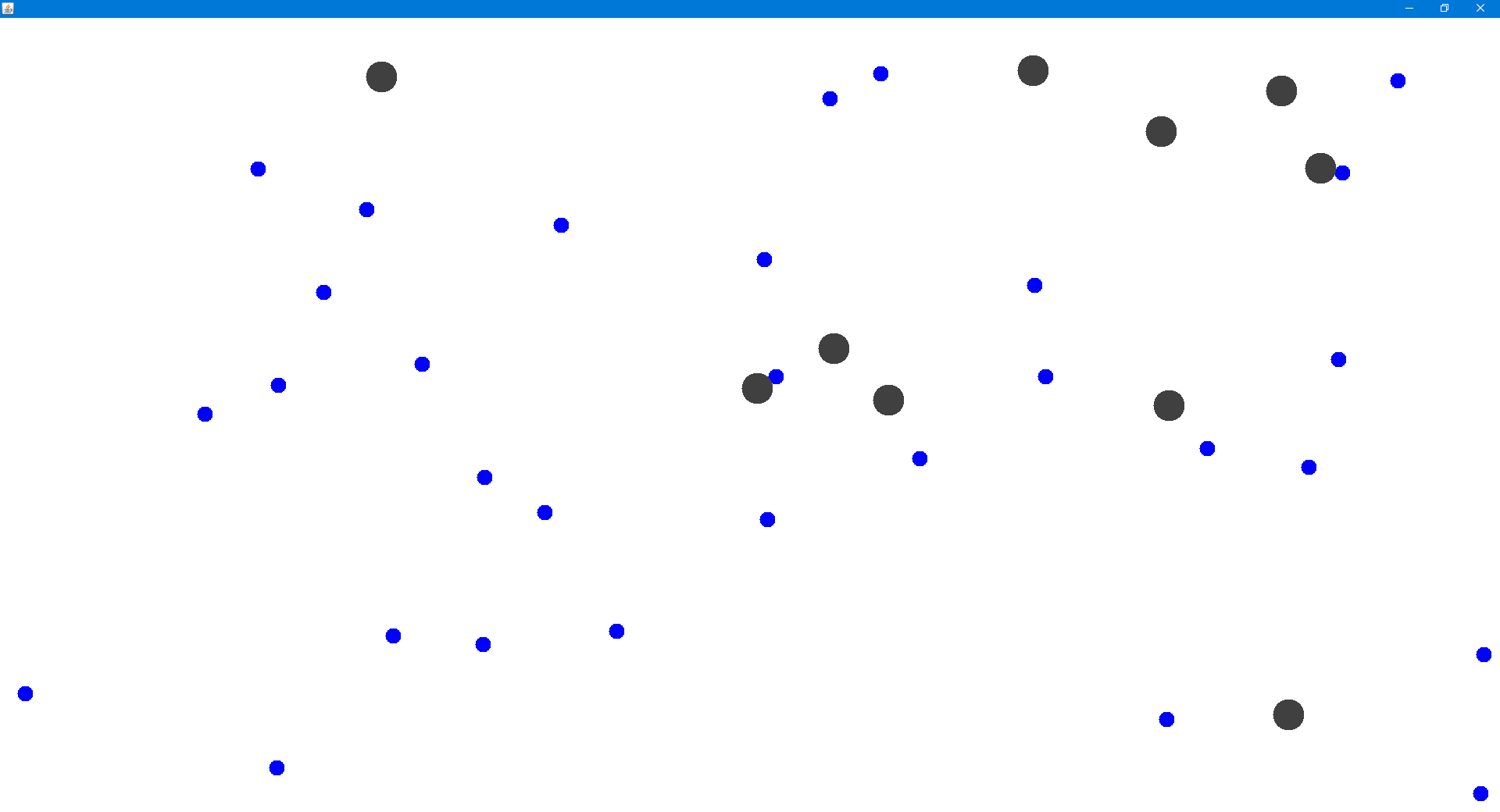
* Attributs
  + numéro de destination (String)
  + message (String)
* Constructeur par paramètres recevant:
  + numéro de destination
  + message
* Services
  + Accesseurs pour les deux attributs

**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 contenant des connexions (un objet de la classe que vous avez écrites).
* 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 la référence, tout simplement).
  + get sur la collection de cellulaires (retourne la référence, tout simplement)..

Pour valider, activez 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.



# (Partie 3)

# *6 - Implémentation par fonctionnalités*

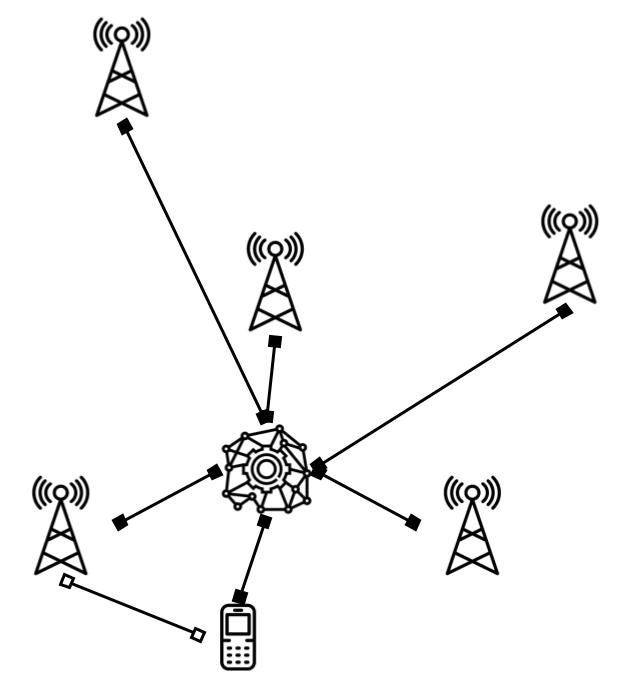
Cette section vous guide dans l’implémentation des fonctionnalités suivantes:

* connecter dynamiquement Cellulaire et Antenne
* établir une connexion
* mise à jour des connexions, suivant les 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.

### 6.1 - Étape 1 - lien cellulaire-antenne

La première étape 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:

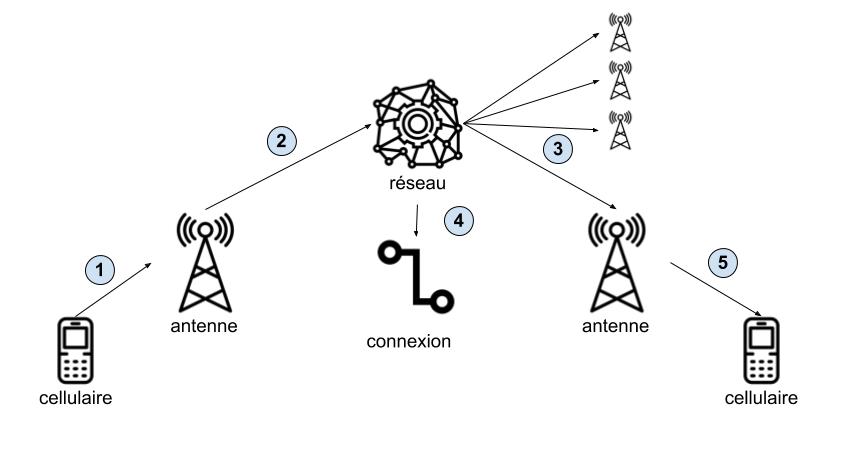
* Le **réseau** doit offrir un service permettant d’obtenir une référence sur l’antenne la plus proche (à partir d’une position).
* L’**Antenne** doit avoir deux services supplémentaires: un permettant d’ajouter un cellulaire à la collection et l’autre permettant d’enlever un cellulaire à la collection.
* Le **Cellulaire:**
  + déclenche 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 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.

### 6.2 - 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.



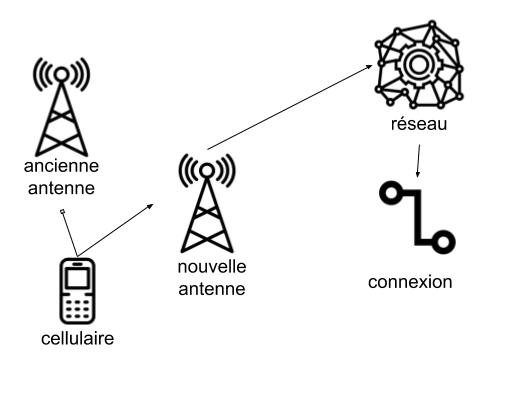
La fonctionnalité utilise les méthodes déclarées par l’interface de l’unité Cellulaire.

* (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() appelle **GestionnaireReseau**::relayerAppel.
* la méthode **GestionnaireReseau**::relayerAppel effectue les opérations suivantes:
  + obtient un numéro de connexion unique (à vous de le faire)
  + (3) parcourt 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:
  + parcourt tous les cellulaires enregistrés à cette antenne
    - compare le numéro cherché avec le numéro des cellulaires
    - s’il y a une correspondance
      * appelle 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 actuel (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.

### 6.3 - Étape 3 - Mise à jour des connexions suivant un changement d’antenne

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.

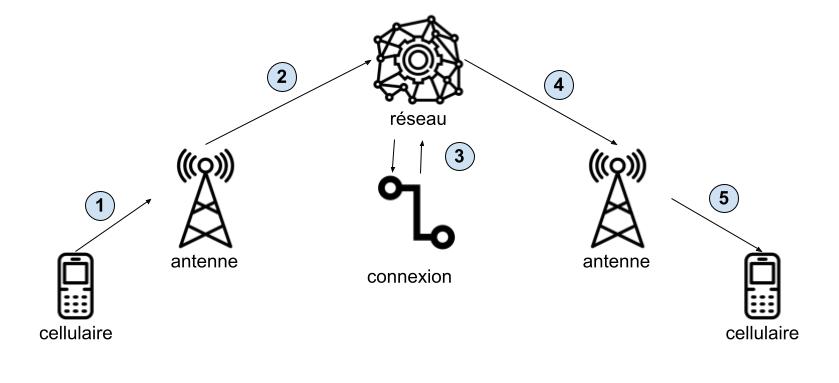


* Dans la classe Connexion, ajoutez une méthode Connexion::miseAJourAntenne() permettant de remplacer une ancienne antenne par une nouvelle.
* Dans la classe gestionnaire réseau, ajoutez une méthode privée qui permet d’obtenir une référence à une Connexion à partir de son numéro.
* ajouter une méthode mettreAJourConnexion() aux classes 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 met à jour 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.

### 6.4 - 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:
  + crée un **Message** et utilise **GestionnaireScenario**.obtenirMessage(numeroLocal), pour obtenir un message aléatoire.
  + la méthode **Cellulaire ::**appeler() appelle la 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 appelle Antenne::recevoir()
  + la méthode Antenne::recevoir() trouve le Cellulaire puis appelle Cellulaire::recevoir()
  + Cellulaire::recevoir peut afficher 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.

### 6.5 - 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.

## 7 - Mise en action, Cellulaire

À chaque tour, chaque cellulaire effectue les opérations suivantes:

* se déplace
* met à jour l’antenne connecté (quelques instructions quand même).
* si connecté à un autre cellulaire déjà
  + probabilité d’envoyer un message
  + sinon, probabilité de raccrocher
* sinon, probabilité d‘effectuer un appel

# Barème d’évaluation

Simulateur et programmes tests 40%

Qualité du code et optimisation 60%

* La qualité professionnelle du code qui vous a été enseignée est attendue

# Conclusion

Dans ce travail vous apprendrez à programmer des tests unitaires. C’est un très bon exercice de programmation et cela aide à développer son sens algorithmique.

Vous mettrez l’écriture de classe simple et l’écriture de collection dans un contexte réaliste en plus d’utiliser les collections de base de Java (ArrayList et Vector).

Vous implémenterez une petite hiérarchie de classes qui utilise les classes abstraites et l’implémentation d’interfaces et enfin vous écrirez les méthodes qui représente le modèle du jeu et de ses connexions.

On vous demande de continuer à programmer en divisant vos problèmes en sous-programmes qui n’effectuent qu’une tâche et qui évite la répétition de code avec des considérations de qualité.







Bon apprentissage**!**

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 [↑](#footnote-ref-1)
2. https://fr.wikipedia.org/wiki/IMSI-catcher [↑](#footnote-ref-2)