 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back”</p> <p>Mise en place de l’infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 1/14</b></p>
---	---	--

## Table des matières

<b>I. Introduction.....</b>	<b>3</b>
<b>II. Ajout des composants au système .....</b>	<b>4</b>
2.1. Création de la classe SourceFixe .....	4
2.2. Création de la classe SourceAleatoire .....	5
2.3. Création de la classe TransmetteurParfait .....	6
2.4. Création de la classe DestinationFinale .....	6
<b>III. Mise en fonctionnement du système .....</b>	<b>8</b>
3.1. Instanciation des composants et connexions avec les sondes .....	8
3.2. Émission du message par la source .....	9
3.3. Gestion du taux d’erreur binaire .....	9
<b>IV. Automatisation du logiciel via des scripts .....</b>	<b>10</b>
4.1. Script compile .....	10
4.2. Script genDoc .....	10
4.3. Script cleanAll .....	11
4.4. Script genDeliverable .....	12
<b>V. Tests et résultats de la première itération.....</b>	<b>13</b>
5.1. Script simulateur .....	13
5.2. Script runTests.....	13
5.3. Vérification de l’itération 1 avec le script <i>Deploiement</i> .....	14



 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back”</p> <p>Mise en place de l’infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 2/14</b></p>
---	---	--

Figure 1 – Modélisation de la chaîne de transmission à l’étape 1 .....	3
Figure 2 – Diagramme de classe correspondant à l’itération 1 .....	4
Figure 3 – Code source de la classe <b>SourceFixe</b> .....	4
Figure 4 – Code source du premier constructeur de la classe <b>SourceAleatoire</b> .....	5
Figure 5 – Code source du deuxième constructeur de la classe <b>SourceAleatoire</b> .....	5
Figure 6 – Code source de la méthode recevoir() provenant de la classe <b>TransmetteurParfait</b> .....	6
Figure 7 – Code source de la méthode emettre() provenant de la classe <b>TransmetteurParfait</b> .....	6
Figure 8 – Code source de la classe <b>DestinationFinale</b> .....	7
Figure 9 – Constructeur de la classe <b>Simulateur</b> .....	8
Figure 10 – Méthode execute() présente dans la classe <b>Simulateur</b> .....	9
Figure 11 – Méthode calculTauxErreurBinaire() présente dans la classe <b>Simulateur</b> .....	9
Figure 12 – Code source du script compile .....	10
Figure 13 – Résultat du script compile .....	10
Figure 14 – Code source du script genDoc .....	11
Figure 15 - Résultat du script genDoc .....	11
Figure 16 – Code source du script cleanAll .....	11
Figure 17 - Résultat du script cleanAll .....	12
Figure 18 – Code source du script genDeliverable .....	12
Figure 19 – Code source du script Simulateur .....	13
Figure 20 – Code source du script runTests .....	13
Figure 21 – Résultats du scripts runTests .....	14
Figure 22 – Données remontées par les deux sondes .....	14
Figure 23 - Exécution du script Deploiement .....	14

 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back »</p> <p>Mise en place de l'infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 3/14</b></p>
---	--	--

## I. Introduction

Au cours de ce projet, nous simulons différentes chaînes de transmission grâce au langage de programmation JAVA. Cette première itération a pour objectif de mettre en place la chaîne de transmission la plus simple possible, qui est représentée par la figure ci-dessous.

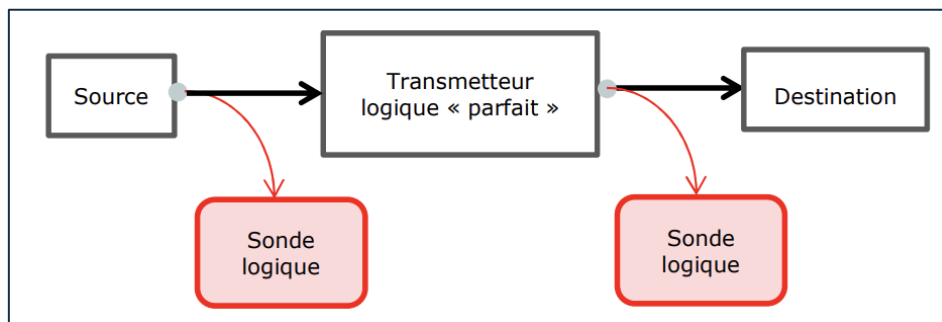


Figure 1 – Modélisation de la chaîne de transmission à l'étape 1

Cette première chaîne de transmission est composée des trois blocs suivants :

- Une **source** qui émet une séquence composée de '0' et de '1'. Cette séquence peut être fixe ou aléatoire ;
- Un **transmetteur logique « parfait »** qui réceptionne la séquence émise par la source et qui ne fait que la retransmettre vers la fin de la chaîne. Ce transmetteur logique a comme particularité d'être parfait, c'est-à-dire qu'aucune erreur ne se glisse dans la séquence booléenne entre la réception et l'émission. C'est donc le cas idéal que nous traitons durant cette première itération ;
- Enfin, la **destination** ne fait que recevoir le signal du transmetteur auquel elle est reliée.

Pour vérifier que le message envoyé est le même que le message reçu, nous plaçons deux sondes sur cette chaîne de transmission. La première en sortie de la source pour mesurer le signal émis au début de la chaîne. La seconde est placée juste après le transmetteur logique. Puisque ce dernier est connecté directement à la destination, cela revient à mesurer le signal en sortie de chaîne.

Une fois que nous aurons obtenu les deux séquences booléennes perçues par les sondes, il nous suffira de les comparer pour connaître le taux d'erreur binaire (TEB) de cette première chaîne de transmission. Sachant que nous sommes dans un cas idéal avec un transmetteur logique parfait, le TEB devrait être nul pour chacun des tests effectués sur ce système.

Nous avons divisé le travail de cette première itération en quatre étapes. Nous commencerons par découvrir le code source qui nous est fourni, auquel nous ajouterons plusieurs composants. Nous les connecterons ensuite au reste de la chaîne et nous y ajouterons les sondes présentées ci-dessus. Afin de répondre aux attentes du client, nous automatiserons le logiciel afin de faciliter son utilisation. Enfin, nous effectuerons une série de test sur le système conçu afin de vérifier son bon fonctionnement.

## II. Ajout des composants au système

Bien qu'au fil des itérations, la chaîne de transmission sera modifiée, une base commune sera présente à chaque étape. En effet, le principe global du système reste le même: envoyer une suite d'informations qui passe par différents composants avant d'atteindre la destination de la chaîne. Par conséquent, la fondation de notre code JAVA sera lui aussi le même, au fur et à mesure des étapes. Cette base est composée de quatre classes abstraites qui feront office de modèle pour les classes que nous ajouterons par la suite. Cette première itération nous demande de créer quatre classes supplémentaires. Ces classes sont représentées en jaune sur la figure ci-dessous.

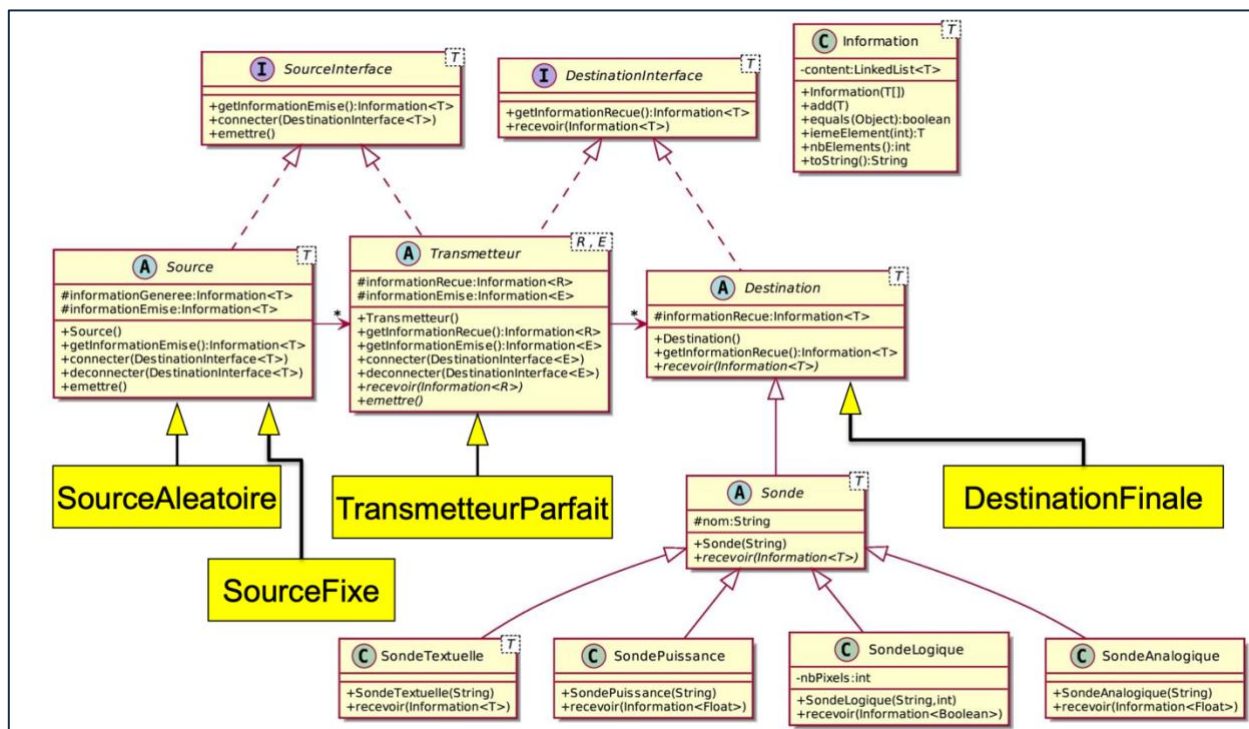


Figure 2 – Diagramme de classe correspondant à l'itération 1

### 2.1. Création de la classe SourceFixe

La première classe dont nous nous occupons est **SourceFixe**. Comme le montre la figure 2, cette classe hérite de la classe abstraite **Source**. Cette classe est appelée quand l'utilisateur donne en entrée du système la séquence booléenne qui est à transmettre.


```
public class SourceFixe extends Source<Boolean> {
    /**
     * Une source qui envoie toujours le même message
     */
    public SourceFixe (String messageString) {
        super();

        informationGeneree = new Information<Boolean>();

        for (int i = 0; i < messageString.length(); ++i) {
            boolean value = messageString.charAt(i) != '0';
            informationGeneree.add(value);
        }

        informationEmise = informationGeneree;
    }
}
```

Figure 3 – Code source de la classe **SourceFixe**

 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back”</p> <p>Mise en place de l’infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 5/14</b></p>
---	---	--

Le code de cette classe consiste juste en une boucle qui parcourt la séquence donnée par l'utilisateur. Si l'information est 0 la valeur ajoutée à la chaîne est FALSE, sinon c'est TRUE. Une fois la chaîne passée en revue, on assigne la variable informationEmise à la chaîne composée de TRUE et de FALSE.

## 2.2. Création de la classe SourceAleatoire

La classe **SourceAleatoire** est plus complexe puisqu'on peut l'appeler dans deux cas différents. Le point commun avec **SourceFixe** est qu'elle hérite également de la classe abstraite **Source**.

La première façon d'envoyer une séquence aléatoire pour l'utilisateur est de donner en paramètre un simple nombre. Ce nombre déterminera la taille de la séquence booléenne à envoyer. Dans ce cas c'est le constructeur montré en figure 4 qui sera appelé.

```
/**
 * Constructeur de la classe SourceAleatoire.
 * Génère une séquence aléatoire de bits de taille spécifiée.
 *
 * @param nbBitsMess le nombre de bits à générer dans la séquence.
 */
public SourceAleatoire(int nbBitsMess) {
    super();
    this.informationGeneree = new Information<Boolean>();

    Random random = new Random();
    // Génération de nbBitsMess bits aléatoires
    for (int i = 0; i < nbBitsMess; i++) {
        this.informationGeneree.add(random.nextBoolean());
    }

    this.informationEmise = this.informationGeneree;
}
```

Figure 4 – Code source du premier constructeur de la classe **SourceAleatoire**

Comme dans le cas d'une séquence fixe fournie par l'utilisateur, une nouvelle chaîne de booléen est créée et liée à la variable informationGeneree. Une boucle est ensuite parcourue autant de fois que la valeur passée en paramètre du constructeur. A chaque passage dans la boucle une valeur de booléen est choisie aléatoirement, puis ajoutée à la séquence. Enfin, la séquence finale est attribuée à la variable informationEmise.


La deuxième façon d'envoyer une séquence composée de '1' et de '0' de manière aléatoire est de faire appel à une graine grâce au paramètre seed. La figure 5 montre comment ce processus est utilisé.

```
/**
 * Constructeur de la classe SourceAleatoire avec graine (seed).
 * Génère une séquence aléatoire de bits de taille spécifiée avec une graine pour la reproductibilité.
 *
 * @param taille la taille de la séquence de bits à générer.
 * @param seed la graine utilisée pour initialiser le générateur aléatoire (permet la reproductibilité des séquences).
 */
public SourceAleatoire(int taille, int seed) {
    super();
    this.informationGeneree = new Information<>();

    Random random = new Random(seed);
    // Génération de taille bits aléatoires avec graine
    for (int i = 0; i < taille; i++) {
        this.informationGeneree.add(random.nextBoolean());
    }

    this.informationEmise = this.informationGeneree;
}
```

Figure 5 – Code source du deuxième constructeur de la classe **SourceAleatoire**

 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back »</p> <p>Mise en place de l'infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 6/14</b></p>
---	--	--

Une séquence de booléens est générée à nouveau et la graine donnée par l'utilisateur est également utilisée pour générer la séquence aléatoire. Cette fois, la nombre de passage dans la boucle dépend de la taille de la séquence de bits à transmettre.

## 2.3. Création de la classe TransmetteurParfait

Comme précisé auparavant, le transmetteur logique ne s'occupe, pour cette première itération, que de réceptionner le message émis par la source, puis de le transmettre à la destination. Par conséquent la classe **TransmetteurParfait**, qui hérite de la classe abstraite *Transmetteur*, ne contient que deux méthodes : recevoir() et emettre().

```
/**
 * reçoit une information. Cette méthode, en fin d'exécution,
 * appelle la méthode émettre.
 *
 * @param information l'information reçue
 * @throws InformationNonConformeException si l'Information comporte une anomalie
 */
@Override
public void recevoir(Information<Boolean> information) throws InformationNonConformeException {
    this.informationRecue = information;
    emettre();
}
```

Figure 6 – Code source de la méthode recevoir() provenant de la classe **TransmetteurParfait**

Ce transmetteur devant être parfait, le message reçu en paramètre de la méthode n'a qu'à être émise. Pour faire cela la méthode recevoir() fait appelle à la deuxième méthode de cette classe : emettre().

```
/**
 * émet l'information construite par le transmetteur
 *
 * @throws InformationNonConformeException si l'Information comporte une anomalie
 */
@Override
public void emettre() throws InformationNonConformeException {
    this.informationEmise = this.informationRecue;

    // Émission vers les composants connectés
    for (DestinationInterface<Boolean> destinationConnectee : destinationsConnectees) {
        destinationConnectee.recevoir(this.informationEmise);
    }
}
```

Figure 7 – Code source de la méthode emettre() provenant de la classe **TransmetteurParfait**

Cette seconde méthode se contente de parcourir la liste des composants auquel le transmetteur est connecté, et de transmettre à chaque composant la séquence d'informations reçue.

## 2.4. Création de la classe DestinationFinale

Cette dernière classe **DestinationFinale** est extrêmement simple puisqu'elle ne fait que recevoir la séquence finale, et l'assigner à l'attribut de classe **informationRecue**.



```
public class DestinationFinale extends Destination<Boolean> {  
    /**  
     * reçoit une information  
     *  
     * @param information l'information à recevoir  
     * @throws InformationNonConformeException si l'Information comporte une anomalie  
     */  
    @Override  
    public void recevoir(Information<Boolean> information) throws InformationNonConformeException {  
        this.informationRecue = information;  
    }  
}
```

Figure 8 – Code source de la classe ***DestinationFinale***

Maintenant que nous avons instancié l'ensemble des classes utiles pour la première étape de ce projet, nous devons connecter les différents composants du système entre eux. Le message rentré par l'utilisateur passera d'un bout à l'autre de la chaîne, avec un taux d'erreur binaire nul (transmetteur parfait).

### III. Mise en fonctionnement du système

La mise en fonctionnement du système passe par la modification de la classe **Simulateur**, qui fait office de chef d'orchestre de notre logiciel. Nous commençons par instancier les différents composants puis nous les connectons entre eux. Une fois cela fait, nous intégrons les sondes dans notre chaîne de transmission. Enfin, il nous reste à lancer l'émission du message par la source, et à gérer le calcul du taux d'erreur binaire.

#### 3.1. Instanciation des composants et connexions avec les sondes

Nous nous intéressons dans un premier temps au constructeur de la classe **Simulateur**. La figure 8 ci-dessous montre notre code.

```
public Simulateur(String[] args) throws ArgumentsException {
    // Analyser et récupérer les arguments
    analyseArguments(args);

    // Choix de la source en fonction des paramètres
    if (messageAleatoire) {
        if (aleatoireAvecGerme) {
            this.source = new SourceAleatoire(nbBitsMess, seed);
        } else {
            this.source = new SourceAleatoire(nbBitsMess);
        }
    } else {
        this.source = new SourceFixe(messageString);
    }

    // Instanciation des composants
    this.transmetteurLogique = new TransmetteurParfait();
    this.destination = new DestinationFinale();

    // Connexion des différents composants
    this.source.connecter(this.transmetteurLogique);
    this.transmetteurLogique.connecter(destination);


    // Connexion des sondes (si l'option -s est pas utilisée)
    if (affichage) {
        this.source.connecter(new SondeLogique( nom: "source", nbPixels: 200));
        this.transmetteurLogique.connecter(new SondeLogique( nom: "transmetteur", nbPixels: 200));
    }
}
```

Figure 9 – Constructeur de la classe **Simulateur**

Le code de ce constructeur est divisé en plusieurs étapes :

1. Tout d'abord, nous faisons appel à la méthode `analyseArguments()` pour vérifier que les paramètres fournis par l'utilisateur répondent aux attentes du logiciel. Si ce n'est pas le cas, l'exception `ArgumentsException` est levée ;
2. Une fois que les paramètres sont validés, nous choisissons la source à utiliser. Pour faire cela, nous regardons si une graine a été donnée par l'utilisateur, s'il utilise une séquence fixe ou encore s'il veut créer une séquence booléenne à partir d'un message ;
3. Nous instancions ensuite le transmetteur logique ainsi que le composant final de notre système ;
4. Ensuite, nous connectons entre eux les composants. Pour cette première étape, la source est connectée au transmetteur logique qui est lui-même connecté à la destination ;
5. Enfin, nous ajoutons à cette structure les deux sondes représentées sur la figure 1.



 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back »</p> <p>Mise en place de l'infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 9/14</b></p>
---	--	--

### 3.2. Émission du message par la source

Maintenant que la chaîne de transmission est formée, il nous suffit d'émettre la séquence de booléens pour qu'elle passe d'un bout à l'autre du système. Pour faire cela nous programmons la méthode `execute()` présente dans la classe **Simulateur**.

```
public void execute() throws Exception {
    // La source émet le message
    source.emettre();
}
```

Figure 10 – Méthode `execute()` présente dans la classe **Simulateur**

Cette simple méthode utilise la source choisie précédemment (fixe ou aléatoire) ainsi que la méthode `emettre()` présente dans la classe abstraite **Source**.

### 3.3. Gestion du taux d'erreur binaire

Pour rappel, le taux d'erreur binaire (TEB) est calculé après que la séquence booléenne soit passée dans l'ensemble des composants du système. Pour obtenir ce TEB, nous avons créé une méthode `calculTauxErreurBinaire()` au sein de la classe **Simulateur**.

```
/**
 * La méthode qui calcule le taux d'erreur binaire en comparant
 * les bits du message émis avec ceux du message reçu.
 *
 * @return La valeur du Taux d'Erreur Binaire.
 */
// jordanbmrd +1
public float calculTauxErreurBinaire() {
    Information messageEmis = this.source.getInformationEmise();
    Information messageReçu = this.destination.getInformationRecue();

    // Vérification de la taille des messages
    if (messageEmis.nbElements() != messageReçu.nbElements()) {
        throw new IllegalArgumentException("La taille du message émis est différente de celle du message reçu.");
    }


    int nbBits = messageReçu.nbElements();
    if (nbBits == 0) {
        return 0f;
    }

    float nbErreurs = 0f;

    // Parcours des éléments pour comparer bit par bit
    for (int i = 0; i < nbBits; ++i) {
        if (messageEmis.iemeElement(i) != messageReçu.iemeElement(i)) {
            nbErreurs++;
        }
    }

    // Calcul du taux d'erreur
    return nbErreurs / nbBits;
}
```

Figure 11 – Méthode `calculTauxErreurBinaire()` présente dans la classe **Simulateur**

 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back »</p> <p>Mise en place de l'infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 10/14</b></p>
---	--	---

## IV. Automatisation du logiciel via des scripts

Notre chaîne de transmission composée d'une source, d'un transmetteur logique « parfait », d'une destination ainsi que de deux sondes est désormais opérationnelle. Il nous reste désormais à simplifier la gestion de notre logiciel. Pour faire cela, l'utilisateur final nous demande de fournir plusieurs scripts. Cette quatrième partie est consacrée au développement de ces scripts.

### 4.1. Script compile

Le premier script bash nommé *compile* permet de compiler l'ensemble du code source permettant à notre logiciel de fonctionner.

```
#!/bin/bash

# Suppression du dossier bin/
rm -rf bin/

# Compilation du projet
echo "Compiling into bin/ folder"
javac -d bin/ src/**/*.java
echo "Done!"
```

Figure 12 – Code source du script compile

Ce programme commence par supprimer le contenu du fichier bin/ au cas où l'utilisateur aurait déjà compilé le programme JAVA auparavant. Il utilise ensuite la commande javac afin de compiler l'ensemble des fichiers JAVA contenus dans le dossier src du projet SIT\_213. L'affichage de commentaires via la commande echo permet de vérifier que la compilation s'est bien passée comme le montre la figure ci-dessous.

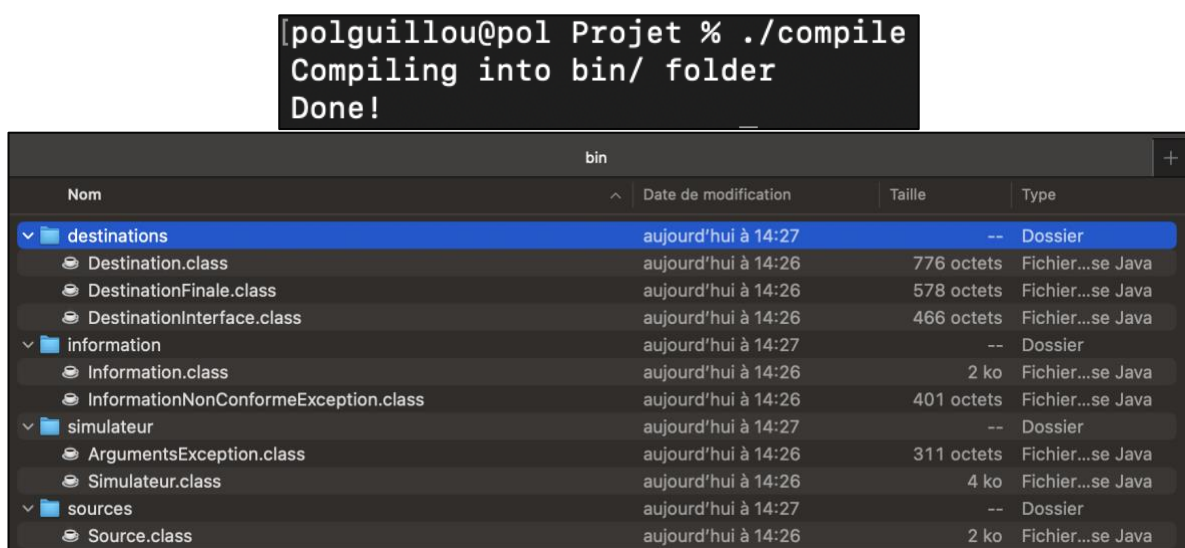


Figure 13 – Résultat du script compile

### 4.2. Script genDoc

Le second script *genDoc* permet à l'utilisateur qui l'exécute de générer la documentation de l'ensemble du projet.

```
#!/bin/bash

# Suppression du dossier docs/
rm -rf docs/

# Génération de la Javadoc
echo "Generating javadoc into /docs folder"
javadoc -d ./docs -quiet ./src/**/*.java -Xdoclint:none
echo "Done!"
```

Figure 14 – Code source du script genDoc

Comme pour le script *compile*, genDoc commence par supprimer le contenu du dossier docs/ où est stockée la documentation du projet, afin de ne pas avoir de doublon ou des documentations obsolètes. Une fois que les anciennes versions sont supprimées, genDoc se sert de la commande javadoc pour insérer dans le dossier docs/ la javadoc.

```
polguillou@pol Projet % ./genDoc
Generating javadoc into /docs folder
Done!
```

docs				
Nom	Date de modification	Taille	Type	
visualisations	aujourd'hui à 14:40	--	Dossier	
VueValeur.html	aujourd'hui à 14:40	101 ko	Texte HTML	
VueCourbe.html	aujourd'hui à 14:40	106 ko	Texte HTML	
Vue.html	aujourd'hui à 14:40	106 ko	Texte HTML	
SondeTextuelle.html	aujourd'hui à 14:40	14 ko	Texte HTML	
SondePuissance.html	aujourd'hui à 14:40	15 ko	Texte HTML	
SondeLogique.html	aujourd'hui à 14:40	15 ko	Texte HTML	
SondeAnalogique.html	aujourd'hui à 14:40	15 ko	Texte HTML	
Sonde.html	aujourd'hui à 14:40	15 ko	Texte HTML	
package-tree.html	aujourd'hui à 14:40	7 ko	Texte HTML	
package-summary.html	aujourd'hui à 14:40	6 ko	Texte HTML	
type-search-index.js	aujourd'hui à 14:40	961 octets	text document	
transmetteurs	aujourd'hui à 14:40	--	Dossier	
TransmetteurParfait.html	aujourd'hui à 14:40	17 ko	Texte HTML	
Transmetteur.html	aujourd'hui à 14:40	26 ko	Texte HTML	

Figure 15 - Résultat du script genDoc

### 4.3. Script cleanAll


Le script bash *cleanAll* est très simple, puisqu'il ne fait que supprimer le contenu des dossiers bin/ et docs/. Grâce à cela, un projet peut être rendu comme étant « vierge » sans contenir les fichiers compilés ou la Javadoc.

```
#!/bin/bash

# Suppression :
# - des archives générées
# - des fichiers compilés
# - de la Javadoc générée

echo "Cleaning..."
rm -f *.tar.gz
rm -rf bin/*
rm -rf docs/*
echo "Done!"
```

Figure 16 – Code source du script cleanAll

 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back »</p> <p>Mise en place de l'infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 12/14</b></p>
---	--	---

```
[polguillou@pol Projet % ./cleanAll
Cleaning...
Done!
[polguillou@pol Projet % ls bin/
[polguillou@pol Projet % ls docs/
```

Figure 17 - Résultat du script *cleanAll*

#### 4.4. Script *genDeliverable*

Le dernier script de la catégorie « gestion du projet » se nomme *genDeliverable*. Il permet de créer une archive du projet complet au format .tar.gz.

```
#!/bin/bash


# Clean le projet
./cleanAll

# Récupération des dernières sources du projet
# echo "Récupération de la dernière version du projet"
# git pull --quiet

# Création de l'archive
echo "Creating archive..."
tar --exclude='genDeliverable' --exclude='Deploiement' --exclude='Parametrage' --exclude='.git'
--exclude='Livrables/' -czf GUILLOU-MAQUENNE-BAUMARD.SIT213.Étape1.tar.gz *
echo "Archive created with success!"
```

Figure 18 – Code source du script *genDeliverable*

Le programme *genDeliverable* commence par appeler le script *cleanAll* présenté ci-dessus. On se sert ensuite de la commande *tar* avec plusieurs paramètres pour archiver le dossier contenant l'ensemble du projet, à l'exception de certains fichiers dont l'utilisateur final n'a pas besoin. Quand nous exécutons ce script une archive est créée. C'est cette dernière que nous envoyons à l'utilisateur final.

 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back »</p> <p>Mise en place de l'infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 13/14</b></p>
---	--	---

## V. Tests et résultats de la première itération

En supplément des quatre scripts présentés précédemment, nous avons utilisé trois programmes supplémentaires afin de tester la première itération de ce projet à travers différents exemples.

### 5.1. Script simulateur

Pour rappel, c'est la classe java **Simulateur** qui permet de mettre en route notre logiciel et de démarrer la chaîne de transmission. Or, notre client souhaite pouvoir utiliser ce système grâce à l'appel d'un script *simulateur* suivi de plusieurs paramètres que nous avons étudiés auparavant (ajout de sondes, utilisation d'une graine...). Nous avons donc répondu à sa demande en confectionnant le script bash suivant :

```
#!/bin/bash

# shellcheck disable=SC2068
java -cp bin/ simulateur.Simulateur $@
```

Figure 19 – Code source du script *Simulateur*

Ce script lance simplement l'exécution de la classe *Simulateur* puis prend en compte les différents paramètres que l'utilisateur final veut prendre en compte dans la chaîne de transmission.

### 5.2. Script runTests

Pour cette première itération, notre client voulait également que nous passions une batterie de test à notre logiciel grâce à un dernier script. Ce script *runTests*, dont le code source est affiché ci-dessous, commence par compiler notre logiciel en faisant appel au script bash *compile*.

```
#!/bin/bash

# Compilation
./compile

# Liste des scénarios de tests à exécuter
test_cases=(
    "-mess 101000101"
    "-mess 325 -s"
    "-seed 1234"
)

# Initialisation du compteur d'échecs
failed_tests=0


# Boucle sur les scénarios de tests
for test_case in "${test_cases[@]"; do
    ./simulateur "$test_case"
    if [ $? -ne 0 ]; then
        echo "Échec du test : $test_case"
        ((failed_tests++))
    fi
done

# Résumé des résultats des tests
if [ "$failed_tests" -ne 0 ]; then
    echo -e "\n$failed_tests tests échoués sur ${#test_cases[@]}"
    exit 1
else
    echo "Tous les tests sont passés avec succès"
fi
```

Figure 20 – Code source du script *runTests*

Le corps de ce programme est décomposé en trois parties :

- Premièrement, la liste des paramètres qui suivent l'appel du script *simulateur*. Nous avons déterminé les trois tests ci-contre puisqu'ils représentent les différentes utilisations que l'utilisateur pourrait faire du logiciel ;
- Ensuite, une boucle qui exécute un par un les tests configurés en première partie du script ;
- Enfin, des messages de sortie qui indiquent à l'utilisateur si les tests se sont bien déroulés ou certains n'ont pas pu être exécutés.

 <p><b>IMT Atlantique</b> Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 1 : Transmission « back-to-back”</p> <p>Mise en place de l’infrastructure</p>	<p>Année scolaire</p> <p>2024-2025</p> <p><b>Page : 14/14</b></p>
---	---	---

```
polguillou@pol sit213 % ./runTests
Compiling into bin/ folder
Done!
java Simulateur -mess 101000101 => TEB : 0.0
java Simulateur -mess 325 -s => TEB : 0.0
java Simulateur -seed 1234 => TEB : 0.0
Tous les tests sont passés avec succès
```

Figure 21 – Résultats du scripts runTests

La figure 20 nous montre que les trois tests que nous avons effectués ont été exécutés sans problème. Par ailleurs, cette première itération simplifie les tests que nous avons effectués car nous sommes ici dans le cas d'un transmetteur parfait. Par conséquent, il nous suffit de vérifier que les tests sont arrivés à terme et que le Taux d'erreur vaut 0 à chaque fois. Enfin, nous pouvons observer sur la figure 21 les données remontées par les deux sondes installées sur la chaîne de transmission et utilisée lors du second test grâce au paramètre '-s'.

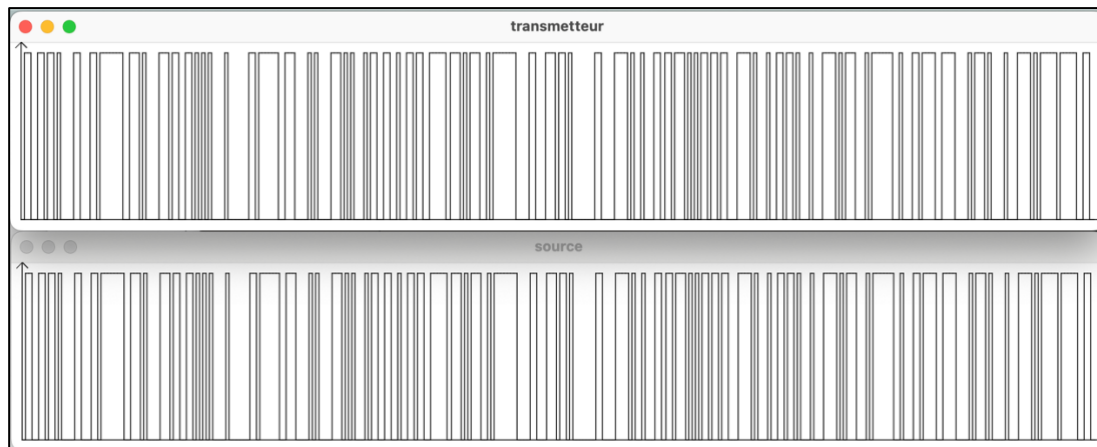


Figure 22 – Données remontées par les deux sondes

### 5.3. Vérification de l'itération 1 avec le script *Deploiement*

Pour conclure cette première itération, nous avons utilisé un nouveau script bash nommé *Deploiement*. A l'inverse des autres scripts, ce dernier nous a été fourni par le client lui-même. Il nous a conseillé de s'en servir sur l'archive que nous voulions lui rendre. En effet, ce programme passe en revue toutes les fonctionnalités que la première étape de ce projet doit contenir. Nous l'avons donc exécuté avec l'archive GUILLOU-MAQUENNE-BAUMARD.SIT213.Étape1.tar.gz.

```
Génération de la javadoc :
Generating javadoc into /docs folder
Done!
-n
-n

Lancement de l'autotest :
Compiling into bin/ folder
Done!
java Simulateur -mess 101000101 => TEB : 0.0
java Simulateur -mess 325 -s => TEB : 0.0
java Simulateur -seed 1234 => TEB : 0.0
Tous les tests sont passés avec succès
--- Déploiement terminé ! ---
```

Figure 23 - Exécution du script Deploiement

Après avoir effectué une série de tests sur l'archive passée en paramètre, le script *Deploiement* nous annonce que le déploiement s'est terminé sans rencontrer d'erreur.