 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 1/32</p>
---	--	---

Table des matières

I. Introduction	5
II. Itération 1 : Transmission back-to-back	6
2.1. Cahier des charges	6
2.2. Ajout des composants au système	7
2.2.1. Création de la classe SourceFixe	7
2.2.2. Création de la classe SourceAleatoire.....	8
2.2.3. Création de la classe TransmetteurParfait.....	9
2.2.4. Création de la classe DestinationFinale	10
2.3. Mise en fonctionnement du système	11
2.3.1. Instanciation des composants et connexions avec les sondes.....	11
2.3.2. Émission du message par la source	12
2.3.3. Gestion du taux d'erreur binaire	12
2.4. Automatisation du logiciel via des scripts	13
2.4.1. Script compile	13
2.4.2. Script genDoc	14
2.4.3. Script cleanAll	14
2.4.4. Script genDeliverable	15
2.5. Tests et résultats de la première itération	17
2.5.1. Script simulateur	17
2.5.2. Script runTests	17
2.5.3. Vérification de l'itération 1 avec le script <i>Deploiement</i>	19
III. Itération 2 : Transmission analogique non bruitée	20
3.1. Cahier des charges	20
3.2. Implémentation des fonctionnalités	21
3.2.2. Fonctionnement de l'émetteur et du récepteur	21
3.2.3. Création de classes dédiées à la seconde étape	22
3.2.4. Connexions des sondes à la chaîne de transmission.....	23
3.2.3. Intégration des nouveaux paramètres à la commande permettant d'utiliser le logiciel	23
3.3. Résultats obtenus	24
3.3.1. Simulation d'un signal NRZ binaire	24
3.3.2. Simulation d'un signal RZ binaire	25
3.3.3. Simulation d'un signal NRZT binaire	25

3.4. Tests effectués	27
3.4.1. Tests de la classe Emettre	27
3.4.2. Tests de la classe Recepteur	27
3.4.3. Tests de la classe SourceFixe	28
3.4.4. Tests de la classe SourceAleatoire	29
3.4.5. Tests de la classe TransmetteurParfait	29
3.4.6. Tests de la classe DestinationFinale.....	30
3.4.7. Couverture du code Java avec Emma	30
3.4.8. Modification du script runTests	31
3.5. Bilan de l'itération	32


 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruiteé d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 3/32</p>
---	--	--

Figure 1 – Modélisation de la chaîne de transmission à l'étape 1	6
Figure 2 – Diagramme de classe correspondant à l'itération 1	7
Figure 3 – Code source de la classe SourceFixe	8
Figure 4 – Code source du premier constructeur de la classe SourceAleatoire	8
Figure 5 – Code source du deuxième constructeur de la classe SourceAleatoire	9
Figure 6 – Code source de la méthode recevoir() provenant de la classe TransmetteurParfait	9
Figure 7 – Code source de la méthode emettre() provenant de la classe TransmetteurParfait	10
Figure 8 – Code source de la classe DestinationFinale	10
Figure 9 – Constructeur de la classe Simulateur	11
Figure 10 – Méthode execute() présente dans la classe Simulateur	12
Figure 11 – Méthode calculTauxErreurBinaire() présente dans la classe Simulateur	12
Figure 12 – Code source du script compile	13
Figure 13 – Résultat du script compile	13
Figure 14 – Code source du script genDoc	14
Figure 15 - Résultat du script genDoc	14
Figure 16 – Code source du script cleanAll	15
Figure 17 - Résultat du script cleanAll	15
Figure 18 – Code source du script genDeliverable	15
Figure 19 – Code source du script Simulateur	17
Figure 20 – Code source du script runTests	18
Figure 21 – Résultats des scripts runTests	18
Figure 22 – Données remontées par les deux sondes	19
Figure 23 - Exécution du script Deploiement	19
Figure 24 - Modélisation de la chaîne de transmission à l'étape 2	20
Figure 25 - Fonctionnement du bloc émetteur	21
Figure 26 - Présentation des codes binaires en ligne NRZ, RZ et NRZT	21
Figure 27 - Diagramme de classes lié à l'itération 2	22
Figure 28 - Code permettant la connexion des sondes à notre système	23
Figure 29 - Commande permettant la simulation d'un signal NRZ binaire	24
Figure 30 - Résultats du signal NRZ binaire	24
Figure 31 - Commande permettant la simulation d'un signal RZ binaire	25
Figure 32 - Résultats du signal RZ binaire	25
Figure 33 - Valeur du signal analogique capté à la sortie de l'émetteur	25


 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruiteé d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 4/32</p>
---	--	--

Figure 34 - Commande permettant la simulation d'un signal RZ binaire	26
Figure 35 - Résultats du signal NRZT binaire	26
Figure 36 - Résultat des tests pour la classe Emettre.....	27
Figure 37 - Résultat des tests pour la classe Emettre.....	27
Figure 38 - Résultat des tests pour la classe SourceFixe	28
Figure 39 - Résultat des tests pour la classe SourceAleatoire	29
Figure 40 - Résultat des tests pour la classe TransmetteurParfait	29
Figure 41 - Résultat des tests pour la classe DestinationFinale	30
Figure 42 - Quantité de notre code couvert grâce aux tests Junit	31
Figure 43 - Modification du script runTests	31

I. Introduction

II. Itération 1 : Transmission back-to-back

2.1. Cahier des charges

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.

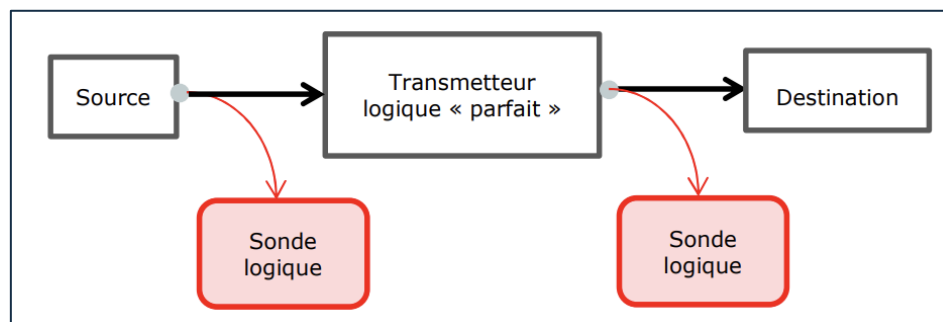


Figure 1 – Modélisation de la chaîne de transmission à l'étape 1

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.

2.2. 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.

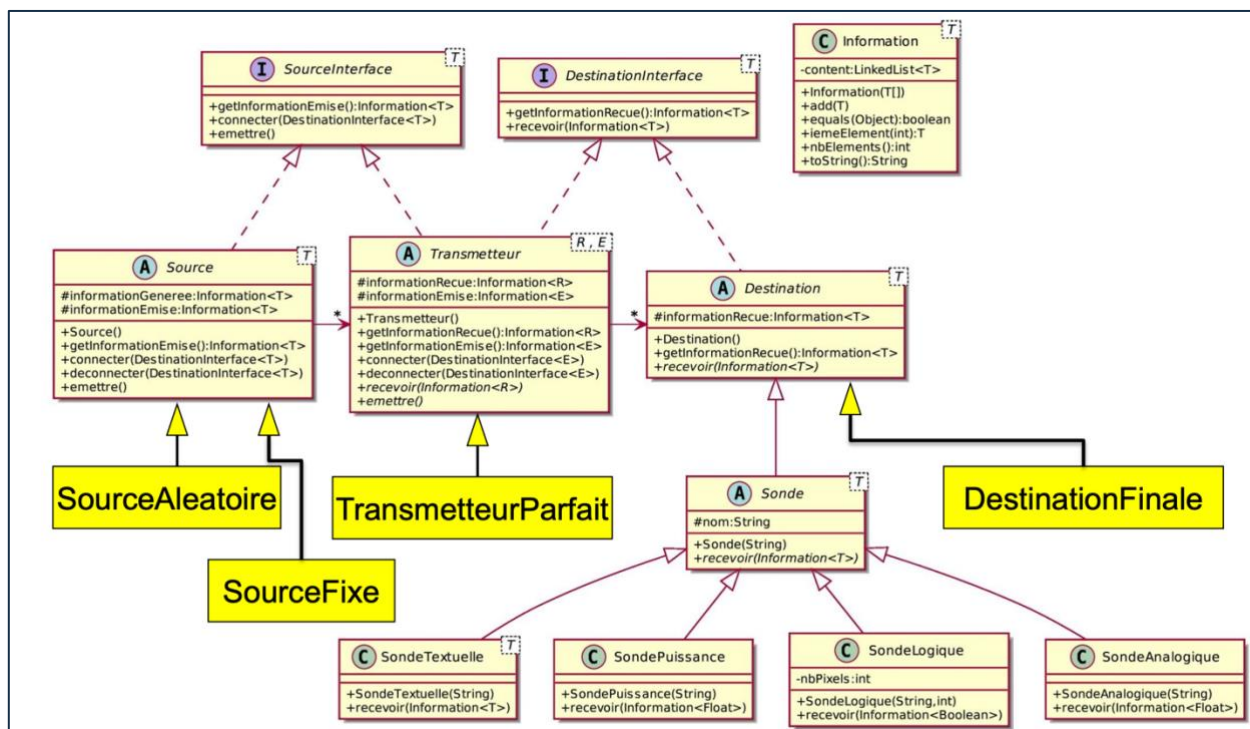


Figure 2 – Diagramme de classe correspondant à l'itération 1

2.2.1. 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**

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.

2.2.2. 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

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 9/32</p>
---	--	---

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**

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, le nombre de passages dans la boucle dépend de la taille de la séquence de bits à transmettre.

2.2.3. 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**

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 10/32</p>
---	--	---

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.

2.2.4. 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).



2.3. 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.

2.3.1. 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 réponde 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.

2.3.2. 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**.

2.3.3. 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>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 13/32</p>
---	--	---

2.4. 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.

2.4.1. 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.

```
[polguillou@pol Projet % ./compile
Compiling into bin/ folder
Done!]
```

bin			
Nom	Date de modification	Taille	Type
▼ destinations	aujourd'hui à 14:27	--	Dossier
Destination.class	aujourd'hui à 14:26	776 octets	Fichier...se Java
DestinationFinale.class	aujourd'hui à 14:26	578 octets	Fichier...se Java
DestinationInterface.class	aujourd'hui à 14:26	466 octets	Fichier...se Java
▼ information	aujourd'hui à 14:27	--	Dossier
Information.class	aujourd'hui à 14:26	2 ko	Fichier...se Java
InformationNonConformeException.class	aujourd'hui à 14:26	401 octets	Fichier...se Java
▼ simulateur	aujourd'hui à 14:27	--	Dossier
ArgumentsException.class	aujourd'hui à 14:26	311 octets	Fichier...se Java
Simulateur.class	aujourd'hui à 14:26	4 ko	Fichier...se Java
▼ sources	aujourd'hui à 14:27	--	Dossier
Source.class	aujourd'hui à 14:26	2 ko	Fichier...se Java

Figure 13 – Résultat du script compile

2.4.2. 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*

2.4.3. 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

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

Figure 17 - Résultat du script cleanAll

2.4.4. 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>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 17/32</p>
---	--	---

2.5. 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.

2.5.1. 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.

2.5.2. 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
```

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

Figure 20 – Code source du script runTests

```
[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'.

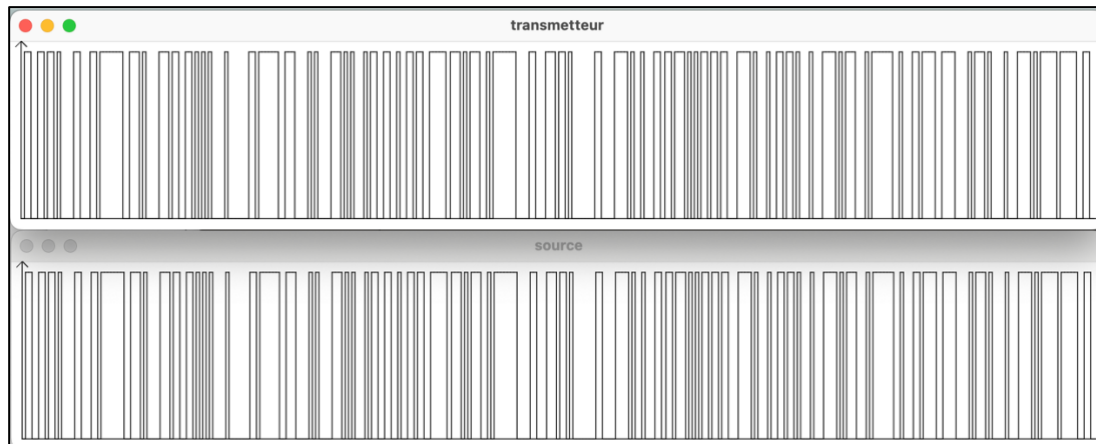


Figure 22 – Données remontées par les deux sondes

2.5.3. 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 ! ---
```

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.

Figure 23 - Exécution du script *Deploiement*

III. Itération 2 : Transmission analogique non bruitée

3.1. Cahier des charges

Pour cette seconde itération, nous gardons le principe d'une chaîne de transmission parfaite. Pour rappel, nous simulons cela grâce à un transmetteur dit « parfait », c'est-à-dire qu'il n'ajoute pas de bruit au signal qu'il transmet. Même si cela n'arrive jamais en pratique, il est très pratique de commencer à implémenter notre logiciel dans cette situation puisque le taux d'erreur binaire est nul en sortie du système.

Comme lors de la précédente itération, notre chaîne comporte en plus du transmetteur, une source et une destination. Ces deux blocs sont situés aux extrémités de la chaîne de transmission et travaillent exclusivement avec des suites de booléens.

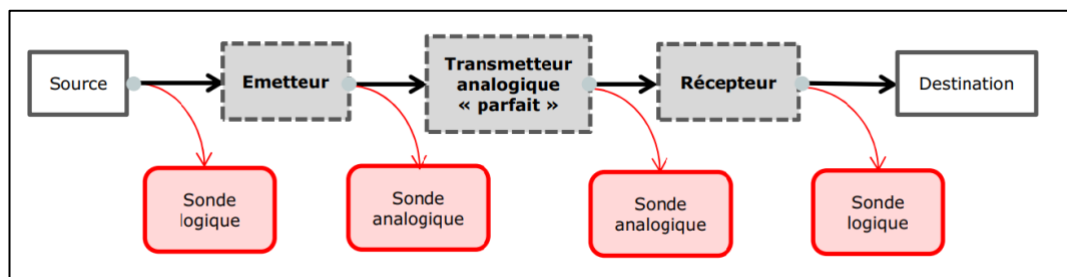


Figure 24 - Modélisation de la chaîne de transmission à l'étape 2

La figure 24 montre la chaîne de transmission complète dédiée à la seconde itération. Nous pouvons remarquer qu'en plus des trois blocs utilisés précédemment, un bloc **Émetteur** ainsi qu'un bloc **Récepteur** sont introduits dans le système :

- Le bloc **Émetteur** reçoit un signal numérique, c'est-à-dire une suite de booléens, depuis la source de la chaîne. Son rôle est de transformer ce signal numérique en signal analogique avant de l'envoyer au transmetteur analogique ;
- Le **Récepteur** s'occupe quant à lui de recevoir le signal analogique venant du transmetteur, et de le retransformer en signal numérique, avant de le transmettre à la destination.

Passer d'un type de signal à l'autre au milieu de la chaîne de transmission permet de mieux adapter le signal aux canaux de transmission. Cette manipulation a un réel intérêt dans des cas pratiques où du bruit est ajouté par le canal de transmission. Cette opération facilitera la modulation et la démodulation du signal, afin de récupérer en sortie du système les données malgré les perturbations du canal. Cela prendra donc tout son sens lors des itérations suivantes où notre signal subira du bruit de manière aléatoire.

Cette seconde itération introduit également dans notre logiciel différents types de signaux en entrée de notre système. Ces signaux (appelés également code) peuvent être sous trois formes différentes :

- **NRZ** Binaire ;
- **NRZT** Binaire ;
- **RZ** Binaire.

Les particularités de ces codes seront présentées dans la partie suivante, avec les justifications qui expliquent la manière dont nous avons implémenté ces nouveautés au sein de notre code Java.

3.2. Implémentation des fonctionnalités

Sachant que ce logiciel est construit de manière itérative, c'est-à-dire que chaque semaine un nouveau bloc de fonctionnalités est introduit, la base de notre code Java reste la même. Nous devons en revanche la modifier pour y inclure, entre autres, l'émetteur et le récepteur dont nous avons parlé auparavant.

3.2.2. Fonctionnement de l'émetteur et du récepteur

Avant de passer à la partie technique et de développer en Java, il est important de comprendre comment fonctionnent ces deux éléments désormais essentiels pour transmettre l'information utile. Entre la réception et l'émission du message par le récepteur ce dernier effectue deux actions décrites par la figure ci-dessous.

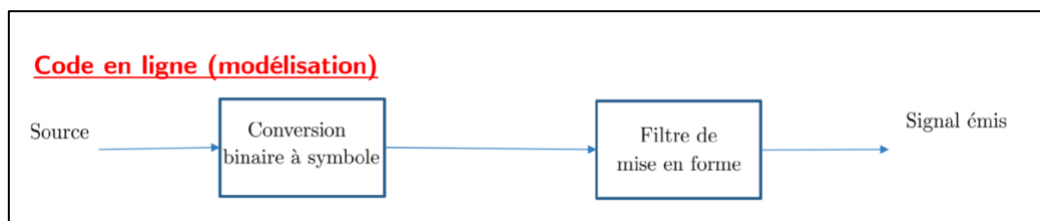


Figure 25 - Fonctionnement du bloc émetteur

L'émetteur commence par convertir, bit à bit, le signal qu'il reçoit de la source sous forme de symbole, afin qu'il devienne un signal analogique. Une fois cela fait, l'élément de notre système applique au signal analogique un filtre de mise en forme. C'est ce filtre qui définit le code de notre signal qui transite par le canal d'information. Dans notre cas nous avons affaire à trois codes binaires différents : **NRZ**, **RZ** et **NRZT**. La forme de ces trois signaux est présentée par la figure 26.

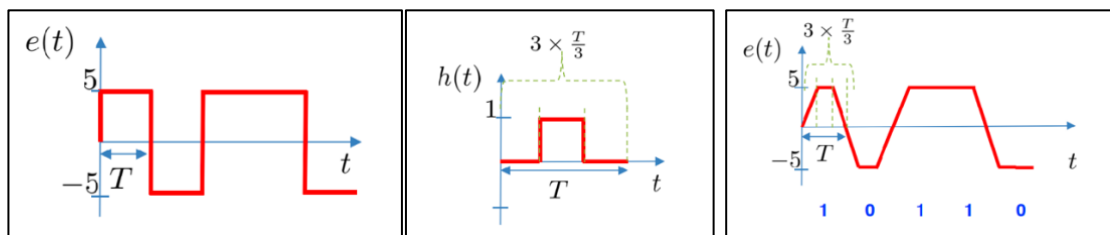



Figure 26 - Présentation des codes binaires en ligne NRZ, RZ et NRZT

La figure 26 nous permet d'observer que ces trois codes ont au moins un point commun : chaque bit peut être divisé en trois parties équitables. La différence notable se trouve dans la différence de forme entre ces trois tiers de période. Dans le cas du code NRZ binaire la valeur du signal reste constante. Pour un code RZ binaire l'amplitude ne vaut a_{\min} ou a_{\max} qu'au milieu du temps bit. Enfin, pour un signal du type NRZT binaire l'amplitude varie progressivement pour atteindre un extremum, avant de rejoindre la valeur du prochain bit de manière progressive également. Nous pouvons d'ailleurs noter que c'est ce troisième exemple qui semble le plus proche de la réalité, puisque dans des cas réels l'amplitude du signal ne peut pas passer d'un extrême à l'autre en un instant t .

Dans le cas du récepteur, situé après le transmetteur parfait, ce sont les actions inverses qui sont effectuées. En effet, le signal analogique est d'abord remis sous sa forme d'origine, avant de repasser en un signal numérique.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 22/32</p>
---	--	--

Maintenant que nous avons bien compris comment chaque nouvel élément de la chaîne de transmission fonctionne, ainsi que les particularités de chaque code traité par notre logiciel, nous pouvons passer à la partie programmation Java.

3.2.3. Création de classes dédiées à la seconde étape

Pour rappel, lors de la première itération chaque élément de la chaîne de transmissions avait une classe Java qui lui était dédié. Nous avons donc continué sur cette lancée en créant une classe **Emetteur**, une classe **Recepteur** ainsi qu'une classe abstraite *Modulateur*.

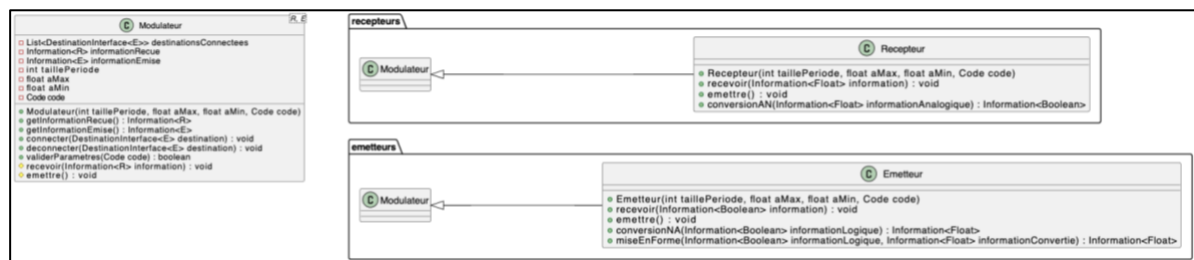



Figure 27 - Diagramme de classes lié à l'itération 2

Nous avons vu précédemment qu'au sein de la chaîne de transmission, l'émetteur et le récepteur ont beaucoup de similitudes comme le fait de recevoir et d'émettre des informations ou d'être connecter à différents éléments du système. Les informations manipulées par ces deux blocs sont également semblables (le type de code, l'amplitude minimum et maximum du signal à transmettre...). C'est pour ces raisons que nous avons intégré à notre programme une classe abstraite *Modulateur*. Grâce à cela les classes **Emetteur** et **Recepteur** n'ont qu'à venir piocher dans les méthodes décrites dans *Modulateur* et nous ne nous retrouvons pas avec des portions de code en double. La classe abstraite *Modulateur* contient également la méthode `validerParametre()` qui nous permet de vérifier que les valeurs d'amplitudes rentrées par l'utilisateur respecte les règles suivantes :

- Pour **NRZ** et **NRZT** : $A_{max} \geq 0 \ \& \ A_{min} \leq 0 \ \& \ A_{min} < A_{max}$
- Pour **RZ** : $A_{max} \geq 0 \ \& \ A_{min} = 0 \ \& \ A_{min} < A_{max}$

La figure 27 montre également que notre code ne possède qu'une classe pour représenter l'émetteur de notre chaîne de transmission, ainsi qu'une classe pour simuler le récepteur. Sachant que la première étape (conversion binaire → symbole) est la même pour les trois codes binaires, cela ne nous semblait pas optimal de coder une classe par code binaire. La différence majeure se trouve dans l'application ou non d'un filtre de mise en forme. Une simple condition dans notre code suffit à déterminer si le signal qui arrive en entrée de l'émetteur doit être filtré ou non. Si c'est le cas, une méthode `miseEnForme()` est appelée. Cette méthode analyse chaque bit et, à partir des tiers de périodes définit précédemment, peut appliquer un filtre différent en fonction du moment du bit qui est analysé.

Il nous a semblé encore plus logique de n'utiliser qu'une classe pour la partie réceptrice du système. En effet, pour les codes binaires NRZ, RZ et NRZT qui arrivent sous forme analogique il suffit de regarder la valeur extrême pour chaque bit. Nous le comparons ensuite avec les valeurs d'amplitude maximale et minimale données en entrées de la chaîne de transmission et cela nous permet de reproduire le signal numérique original.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruiteé d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 23/32</p>
---	--	--

3.2.4. Connexions des sondes à la chaîne de transmission

Un moyen très fiable de vérifier que le signal passe bien à travers notre chaîne de transmission est de visualiser le signal qui sort de chaque élément du système (la source, l'émetteur, le transmetteur, le récepteur et la destination). Pour faire cela nous utilisons des sondes que nous connectons aux différents blocs de notre structure. Lors de la première itération nous n'avions utilisé que des sondes logiques puisque le signal était numérique d'un bout à l'autre du système. Maintenant que le signal est sous forme analogique pendant une partie de son trajet, nous introduisons également dans notre code des sondes du même type que le signal. La figure 24 présentée dans la partie 3.1. montre les différentes sondes que nous plaçons tout le long de notre chaîne. Leur emplacement relève simplement de la logique :

- Si le signal qui sort du bloc est **numérique** nous utilisons une **sonde logique** ;
- Si le signal qui sort du bloc est **analogique** nous utilisons une **sonde analogique**.

```
// Connexion des sondes
if (affichage) {
    this.source.connecter(new SondeLogique( nom: "source " + code, nbPixels: 200));
    this.emetteur.connecter(new SondeAnalogique( nom: "emetteur " + code));
    this.transmetteurAnalogique.connecter(new SondeAnalogique( nom: "transmetteur " + code));
    this.recepteur.connecter(new SondeLogique( nom: "recepteur " + code, nbPixels: 200));
}
```

Figure 28 - Code permettant la connexion des sondes à notre système

Une fois que nous savons où placée telle ou telle sonde, il nous suffit d'utiliser la méthode `connecter()` pour relier chaque sonde aux éléments de notre chaîne de transmission, comme le montre la figure 28.


3.2.3. Intégration des nouveaux paramètres à la commande permettant d'utiliser le logiciel

La dernière chose qu'il reste à intégrer à notre code concerne l'utilisation du logiciel. Pour utiliser cette chaîne de transmission, l'utilisateur doit appeler le script *Simulateur* suivi de plusieurs paramètres qui décrivent le signal passé en entrée du système. Jusqu'à présent l'utilisateur pouvait :

- Préciser le message ou la longueur du message à émettre ;
- Utiliser ou non les sondes intégrées à la structure ;
- Générer un signal aléatoire à partir d'une graine donnée en paramètre.

Cette seconde itération doit désormais permettre à l'utilisateur de notre logiciel de choisir quel code binaire utiliser, ainsi que l'amplitude minimum et maximum que le signal aura. Pour faire cela, nous nous intéressons à classe mère de notre projet : **Simulateur**. C'est cette classe qui est exécutée lors de l'appel du script qui porte le même nom. De plus, c'est dans cette classe que les autres paramètres (`-mess`, `-s`, `-seed...`) sont gérés. Cela nous semblait donc logique d'intégrer les trois nouveaux paramètres au même endroit.

Le choix du code binaire pour l'utilisateur se fera grâce au paramètre « `-code` » suivi du nom du code choisit (**NRZ**, **NRZT** ou **RZ**). Pour ce qui est de l'amplitude, l'usage pourra se servir des paramètres « `-aMax` » et « `-aMin` » suivis de la valeur des amplitudes minimales et maximales. Par défaut, le code binaire utilisé est NRZ, l'amplitude minimale vaut 0 et l'amplitude maximale vaut 1.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 24/32</p>
---	--	--

3.3. Résultats obtenus

Après avoir implémenté les fonctionnalités liées à l'itération 2, il nous reste à appeler la classe **Simulateur** avec les nouveaux paramètres. Cela nous permet de vérifier si les résultats retournés par les sondes sont cohérents avec ce que l'on attend en théorie. Pour l'instant nous sommes dans le cas d'un transmetteur parfait qui n'ajoute pas de bruit à notre signal. Par conséquent, les deux sondes logiques donnent le même résultat, tout comme les deux sondes analogiques. Nous avons choisi de nous concentrer sur les sondes postées en sortie de l'émetteur et du récepteur puisque ce sont les deux éléments cruciaux de cette chaîne de transmission adaptée à la seconde itération. Pour chaque code nous décidons de simuler le signal correspondant à la chaîne booléenne suivante : « 1101000101 ». Nous avons choisi cette suite puisqu'elle contient une alternance entre les 0 et 1, mais également au moins deux bits d'affilée qui sont les mêmes.

3.3.1. Simulation d'un signal NRZ binaire

Le premier type de signal que nous passons en entrée de la chaîne de transmission utilise le code binaire NRZ dont le résultat théorique est présenté dans la partie 3.2.2.

```
Build and run Modify options ▾ ↗
java 21 SDK of 'sit213' ▾ simulateur.Simulateur
-mess 1101000101 -code NRZ -aMax 5 -aMin -5 -s
```

Figure 29 - Commande permettant la simulation d'un signal NRZ binaire

Le code NRZ binaire permettant d'avoir une amplitude minimale inférieure à 0, nous en profitons pour avoir un signal centré en 0.

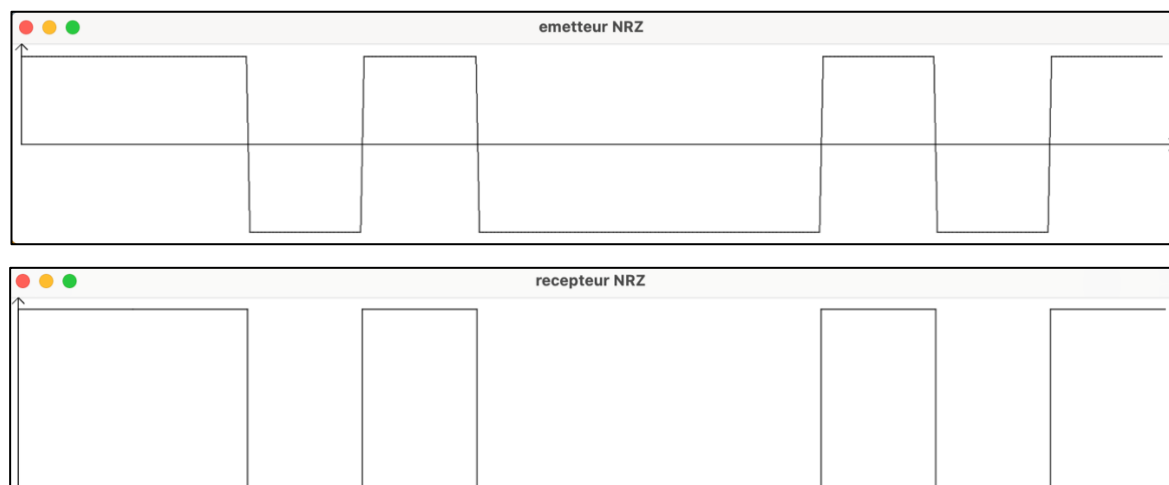


Figure 30 - Résultats du signal NRZ binaire

Le code NRZ binaire étant le seul code sur lequel aucun filtre n'est appliqué, il est normal que les signaux captés par les sondes logiques et analogiques sont très similaires. On remarque tout de même que le signal analogique capturé par l'émetteur ne passe pas d'un extrémum à l'autre. C'est tout à fait normal puisqu'il n'est pas possible dans un cas réel de passer en un instant d'une valeur x à une valeur $-x$. Les résultats obtenus ici sont donc ceux attendus.

3.3.2. Simulation d'un signal RZ binaire

Une des particularités du code RZ binaire est que son amplitude minimale doit forcément être égale. Modifions donc notre ligne de commande pour s'adapter à ce second type de signal.

```
Build and run Modify options
java 21 SDK of 'sit213' simulateur.Simulateur
-mess 1101000101 -code RZ -aMax 3 -aMin 0 -s
```

Figure 31 - Commande permettant la simulation d'un signal RZ binaire

Cette fois le signal qui passe arrive en entrée du récepteur est soumis à un filtre de mise en forme. Ce dernier est présenté sur la figure 26.

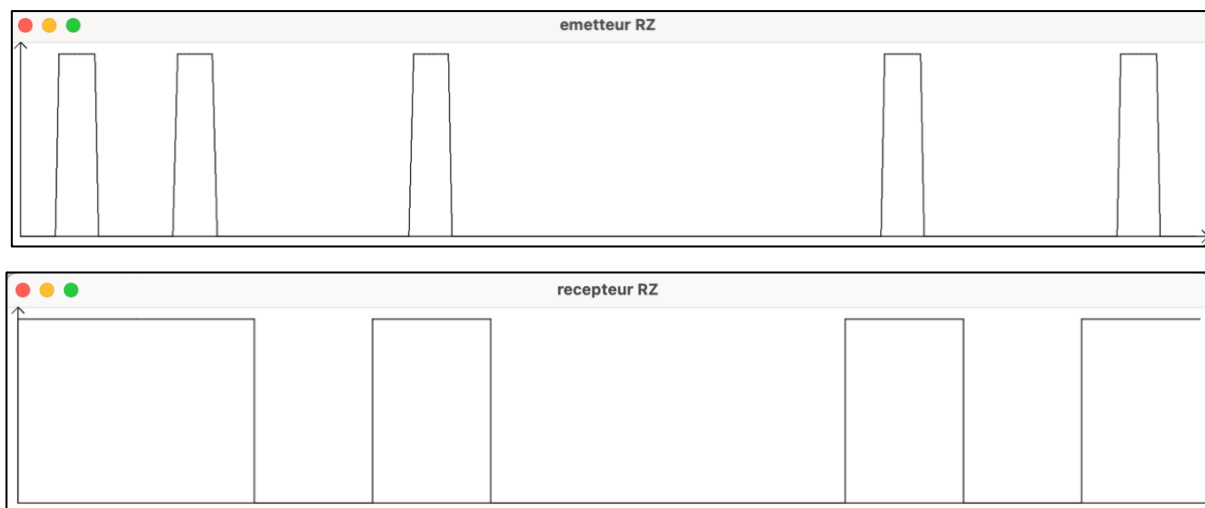


Figure 32 - Résultats du signal RZ binaire

La figure 32 montre que le résultat retourné par les sondes analogiques sont extrêmement proches du résultat théorique présenté précédemment. Pour chaque bit logique le signal est bien décomposé en trois parties égales. Aux extrémités la valeur vaut toujours 0, et au milieu du bit elle vaut aMax ou aMin en fonction du booléen passé en entrée du système. Comme pour le code NRZ binaire nous pouvons observer que lorsque l'amplitude passe progressivement d'une valeur à l'autre dans le signal analogique, contrairement au résultat fourni par la sonde logique. Cela est dû uniquement aux sondes analogiques, puisqu'on ne retrouve aucune valeur intermédiaire quand on observe précisément les valeurs à la sortie de l'émetteur, comme le montre la figure 33.

```
Signal analogique à la sortie de l'émetteur pour un signal RZ binaire : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0
```

Figure 33 - Valeur du signal analogique capté à la sortie de l'émetteur

3.3.3. Simulation d'un signal NRZT binaire

Le troisième et dernier type de signal que nous simulons est le code NRZT binaire. Comme pour le code RZ binaire, l'émetteur applique au signal numérique un filtre de mise en forme dont parlé auparavant. Nous décidons d'utiliser des valeurs d'amplitude différentes pour ce code puisqu'il a moins de contraintes que celui traité précédemment.

```
Build and run Modify options
java 21 SDK of 'sit213' simulateur.Simulateur
-mess 1101000101 -code NRZT -aMax 8 -aMin -8 -s
```

Figure 34 - Commande permettant la simulation d'un signal RZ binaire

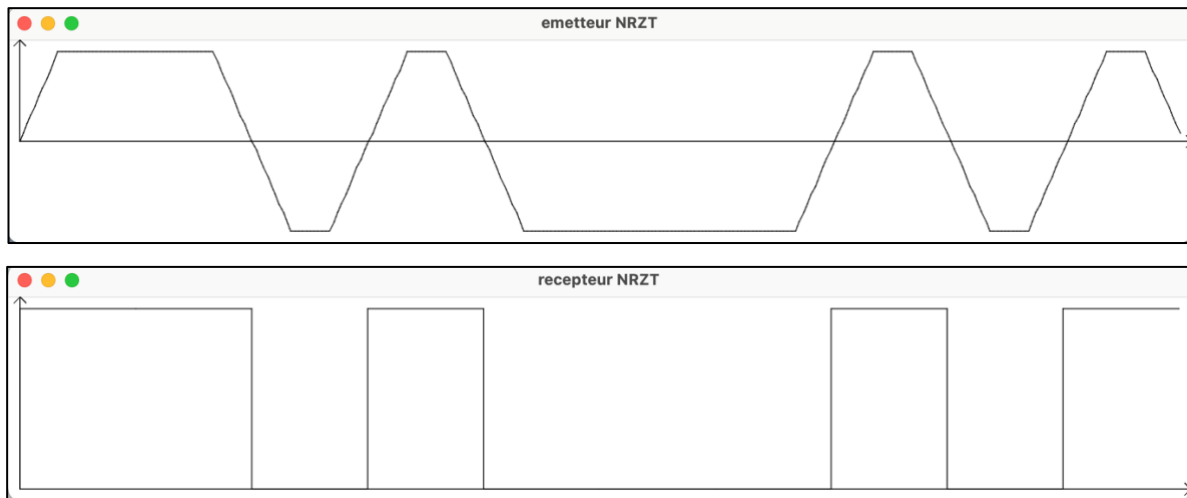



Figure 35 - Résultats du signal NRZT binaire

La complexité de ce filtre réside dans le fait qu'il ne réagit pas de la même manière en fonction du bit suivant. En effet, si dans la chaîne de booléens deux TRUE sont l'un après l'autre l'amplitude du signal restera la même pendant toute la suite de bits similaires. Par ailleurs, la transition entre deux bits différents (pour un signal numérique TRUE → FALSE ou FALSE → TRUE) est lissée directement grâce au filtre de mise en forme. Dans un cas réel cela permet une meilleure compatibilité entre les systèmes analogiques, notamment pour certains systèmes de transmission qui peuvent être sensibles aux changements rapides de signal.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 27/32</p>
---	--	--

3.4. Tests effectués

Nous avons mis en place une série de tests pour valider le bon fonctionnement des différentes composantes du système. Nous avons réalisé les tests unitaires à l'aide de JUnit 4. Nous avons également utilisé Emma afin d'évaluer la couverture de code et nous assurer que les tests couvrent toutes les parties de l'application. Pour simuler certains composants, nous avons utilisé EasyMock, qui permet de mocker et tester des fonctions.

Les tests que nous avons effectués couvrent à la fois les performances du projet ainsi que l'analyse des résultats (TEB). Nous avons décidé de mettre en place ces tests puisque cette itération le permet du fait que le transmetteur n'ajoute pas de bruit au signal.

3.4.1. Tests de la classe Emettre

Les tests liés à cette classe nous permettent de vérifier le bon fonctionnement du passage SIGNAL NUMÉRIQUE → ANALOGIQUE pour les codes binaires NRZ, NRZT et RZ. Nous pouvons voir sur la figure 36 que les trois tests que nous avons effectués sur cette classe sont exécutés et renvoient les résultats attendus.

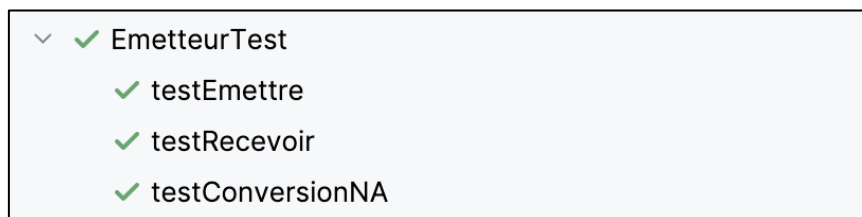


Figure 36 - Résultat des tests pour la classe Emettre

La figure ci-dessus montre la liste des tests qui sont effectués sur la classe Emetteur :


- testEmettre s'assure que la méthode `emettre()` génère bien une information émise pour différents types de codages binaires (NRZ, RZ, NRZT) ;
- testRecevoir vérifie que la méthode `recevoir()` de chaque émetteur reçoit correctement l'information avant de l'émettre ;
- testConversionNA examine la conversion numérique-analogique (NA) pour chaque type de codage. Il vérifie que le nombre d'éléments et les valeurs des signaux convertis sont corrects (ex. : valeur maximale atteinte dans le cas du NRZT) ;

3.4.2. Tests de la classe Recepteur

Les tests de la classe **Recepteur** visent à valider la réception, la conversion SIGNAL ANALOGIQUE → NUMÉRIQUE ainsi que la gestion des erreurs. La figure 37 nous permet de vérifier que l'ensemble des tests passés sur cette classe se sont déroulés sans erreur.



Figure 37 - Résultat des tests pour la classe Emettre

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruiteé d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 28/32</p>
---	--	--

Cette classe de tests nous sert à vérifier le bon fonctionnement de l'ensemble des méthodes incluses dans la classe **Recepteur** :

- testRecevoirInformationValide vérifie que le récepteur reçoit correctement une information analogique valide et la transmet à la destination connectée après l'avoir convertie ;
- testRecevoirInformationInvalide s'assure que la réception d'une information nulle provoque une exception `InformationNonConformeException` ;
- testRecevoirInformationVide vérifie que la réception d'une information vide déclenche également une exception `InformationNonConformeException` ;
- testEmettre valide que le récepteur émet une information correcte après réception d'une information analogique valide. La taille de l'information émise est vérifiée après conversion en fonction de la période définie (ici, période de 2) ;
- testValiderParametres s'assure que la méthode `validerParametres` valide correctement les paramètres (type de codage, `aMin`, `aMax`...) ;
- testValiderParametresInvalide vérifie que des paramètres invalides, comme un intervalle incohérent où `aMin >= aMax`, déclenchent une exception ;
- testConversionAN s'assure que la conversion d'une information analogique valide en binaire est effectuée correctement. La taille de l'information binaire est vérifiée après la conversion ;
- testConversionANInformationInvalide s'assure que la tentative de conversion d'une information nulle en binaire lève une exception `InformationNonConformeException` ;
- testConversionANInformationVide vérifie que la conversion d'une information vide en binaire lève également une exception `InformationNonConformeException`.

3.4.3. Tests de la classe **SourceFixe**

Les tests effectués en lien avec la classe **SourceFixe** vérifient que les messages fournis par l'utilisateur, lors de l'utilisation du logiciel, sont générés et émis correctement.

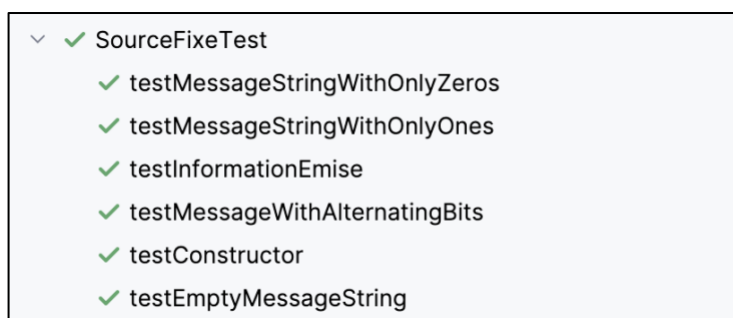



Figure 38 - Résultat des tests pour la classe **SourceFixe**

Tout comme les précédentes classes, les méthodes de test suivantes ont été exécutées sans problème :

- testConstructor s'assure que le constructeur de **SourceFixe** génère correctement une séquence d'informations à partir d'un message binaire donné en paramètre ;
- testInformationEmise vérifie que l'information émise est identique à l'information générée ;
- testMessageStringWithOnlyOnes valide que le constructeur de **SourceFixe** génère correctement une séquence de `true` pour un message binaire composé uniquement de 1 ;
- testMessageStringWithOnlyZeros s'assure que le constructeur génère une séquence de `false` pour un message binaire composé uniquement de 0 ;
- testMessageWithAlternatingBits vérifie que la génération de séquences avec des bits alternés (ici 101010) est correcte ;

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 29/32</p>
---	--	--

- testEmptyMessageString s'assure que la génération d'une séquence à partir d'un message vide produit une information vide.

3.4.4. Tests de la classe SourceAleatoire

Les tests liés à la classe **SourceAleatoire** visent à vérifier la génération correcte de séquences aléatoires d'informations. Encore une fois, tous les tests décrits ci-dessous se sont déroulés sans accroc.

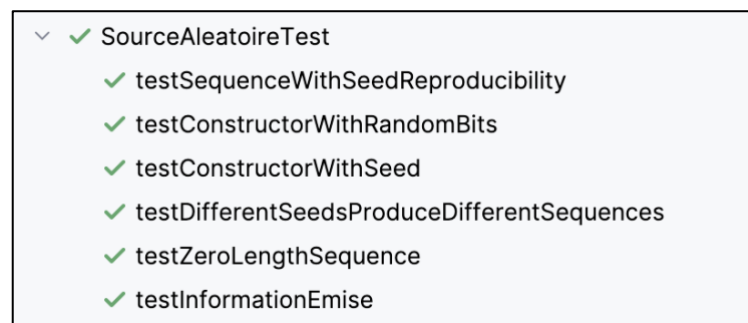


Figure 39 - Résultat des tests pour la classe SourceAleatoire

- testConstructorWithRandomBits vérifie que le constructeur de **SourceAleatoire**, lorsqu'il est appelé sans seed, génère une séquence aléatoire de bits de la bonne longueur ;
- testConstructorWithSeed vérifie que l'utilisation d'un seed permet de générer une séquence aléatoire reproductible ;
- testInformationEmise vérifie que la séquence d'informations générée par **SourceAleatoire** est correctement émise ;
- testZeroLengthSequence s'assure que la génération d'une séquence de longueur zéro ne produit pas d'erreur et retourne une séquence vide ;
- testSequenceWithSeedReproducibility valide que deux instances de **SourceAleatoire** initialisées avec le même seed produisent des séquences identiques ;
- testDifferentSeedsProduceDifferentSequences vérifie que l'utilisation de seeds différents produit des séquences différentes.

3.4.5. Tests de la classe TransmetteurParfait

Les tests exécutés dans la classe **TransmetteurParfaitTest** valident le fait que la transmission des informations se fait sans altération. La figure 40 montre la liste des méthodes de tests dont nous nous servons pour vérifier cela. Les objectifs de chacun de ces tests sont définis ci-dessous.

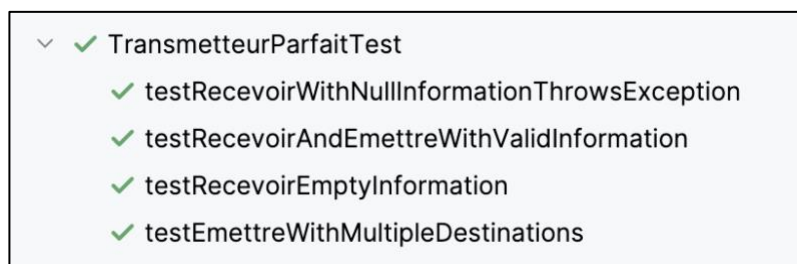


Figure 40 - Résultat des tests pour la classe TransmetteurParfait

- testRecevoirAndEmettreWithValidInformation vérifie que la méthode recevoir() traite correctement les informations et que la méthode emettre() transmet ces informations de manière parfaite (sans altération) vers le récepteur ;
- testEmettreWithMultipleDestinations s'assure que la méthode emettre() fonctionne correctement avec plusieurs destinations connectées (récepteur + sondes), en envoyant la même information à toutes les destinations ;

- testRecevoirWithNullInformationThrowsException vérifie que l'envoi d'une information nulle à la méthode recevoir() lève une exception InformationNonConformeException ;
- testRecevoirEmptyInformation s'assure que l'émission d'une information vide ne provoque pas d'erreur et que les destinations reçoivent une information vide.

3.4.6. Tests de la classe DestinationFinale

Cette sixième et dernière classe de tests a pour objectif de vérifier la capacité de la destination de notre chaîne de transmission à recevoir et stocker correctement les informations, ainsi que de gérer les erreurs potentielles. Nous pouvons voir sur la figure 41 que les quatre types de tests que nous exécutons pour valider le bon fonctionnement de **DestinationFinale** passent tous sans lever d'erreurs inattendues.

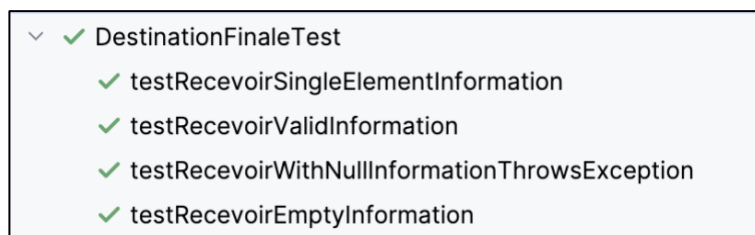


Figure 41 - Résultat des tests pour la classe DestinationFinale

- testRecevoirValidInformation vérifie que la méthode recevoir() de la classe DestinationFinale stocke correctement une séquence d'informations valides. Le test compare les informations reçues avec celles envoyées par l'élément d'avant pour s'assurer qu'elles sont identiques ;
- testRecevoirWithNullInformationThrowsException s'assure que l'envoi d'une information nulle à la méthode recevoir déclenche une exception InformationNonConformeException;
- testRecevoirEmptyInformation valide que la réception d'une information vide ne déclenche pas d'erreur, et que l'information stockée est bien vide ;
- testRecevoirSingleElementInformation vérifie que la méthode recevoir() gère correctement une séquence contenant un seul élément.


3.4.7. Couverture du code Java avec Emma

Comme expliqué précédemment, nous nous servons de l'outil Emma afin de voir le pourcentage de notre code qui est couvert par les tests présentés dans les sous-parties précédentes. La figure 42 affichée ci-dessous montre, pour chaque classe de tests, le nombre de méthodes couvertes par les tests Junit.

Element ^	Class, %	Method, %	Line, %
sources	100% (1/1)	40% (2/5)	50% (5/10)
Source	100% (1/1)	40% (2/5)	50% (5/10)

Element ^	Class, %	Method, %	Line, %
transmetteurs	100% (1/1)	100% (2/2)	100% (7/7)
TransmetteurParfait	100% (1/1)	100% (2/2)	100% (7/7)

Element ^	Class, %	Method, %	Line, %
destinations	100% (1/1)	100% (1/1)	100% (3/3)
DestinationFinale	100% (1/1)	100% (1/1)	100% (3/3)

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 31/32</p>
---	--	---

Element ^	Class, %	Method, %	Line, %
modulation	100% (4/4)	93% (15/16)	90% (87/96)
emetteurs	100% (2/2)	100% (6/6)	92% (47/51)
Emetteur	100% (2/2)	100% (6/6)	92% (47/51)
recepteurs	100% (1/1)	100% (4/4)	95% (23/24)
Recepteur	100% (1/1)	100% (4/4)	95% (23/24)

Figure 42 - Quantité de notre code couvert grâce aux tests Junit

Nous pouvons remarquer que sur les 28 méthodes contenues dans les principales classes de notre projet 24 sont concernées par les tests Junit. Celles qui ne le sont pas sont souvent des méthodes qui ne servent qu'à afficher ou renvoyer des valeurs, donc effectuer des tests sur ces dernières présente peu d'intérêt.

3.4.8. Modification du script runTests


Nous avons également des tests en ligne de commande qui s'exécutent depuis notre script *runTests*. Ces tests nous permettent de nous assurer que le programme fonctionne correctement avec des paramètres différents. La partie du script *runTests* modifiée pour accueillir ces nouveaux tests est présentée dans la figure ci-dessous.

```
# Liste des scénarios de tests à exécuter
test_cases=(
  "-mess 10"                # Test avec un message aléatoire de 10 bits
  "-mess 1010101"          # Test avec un message fixe
  "-mess 110011 -code NRZ"  # Test avec codage NRZ
  "-mess 0010011 -code RZ"  # Test avec codage RZ
  "-aMax 5 -aMin -5 -code NRZT" # Test avec des amplitudes extrêmes
  "-mess 50 -seed 1234"      # Test avec une seed spécifique
  "-mess 10011100111001100011110000111000011100001110000111100001111000000000" # Message long
  "-mess 1000000"           # Test avec un message d'un million de bits
)

# Si on n'est pas dans un pipeline GitLab (la variable d'environnement CI n'est pas définie)
if [ -z "$CI" ]; then
  test_cases+=(
    "-s -code NRZT -aMin -4 -aMax 4"
    "-s -seed 27 -code RZ -aMin 0 -aMax 6"
    "-s -mess 12 -seed 1234 -code NRZ -aMin -5 -aMax 5"
    "-s -mess 01101101110 -code NRZT"
  )
fi
```

Figure 43 - Modification du script runTests

Nous avons séparé les tests visuels (avec le paramètre -s) des tests non visuels afin que la pipeline GitLab puisse s'exécuter correctement. Ainsi, l'exécution du script ne provoque pas un crash du logiciel ou une erreur lors du push vers le repository.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Etape 2 : Transmission non-bruitee d'un signal</p> <p>analogique</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 32/32</p>
---	--	--

3.5. Bilan de l'itération

Cette seconde itération nous a permis de franchir une étape importante dans la simulation de la transmission d'un signal à travers une chaîne de transmission dite « parfaite ». Les principaux objectifs étaient de mettre en place un émetteur et un récepteur capables de convertir des signaux numériques en signaux analogiques, et vice versa, tout en intégrant trois types de codage binaire : NRZ, RZ, et NRZT.

Nous avons réussi à implémenter les différentes classes nécessaires, notamment celles de l'émetteur, du récepteur, ainsi que la classe abstraite *Modulateur*, qui permet une gestion optimisée des paramètres communs à ces blocs. Cette abstraction a simplifié la gestion des codes binaires et a permis d'éviter la duplication de code.

Les résultats obtenus lors des simulations montrent que notre système fonctionne comme attendu, sans erreurs de transmission, puisque cette seconde étape comprenait l'absence de bruit au sein du canal de transmission. Nous avons validé les différentes simulations avec les sondes analogiques et logiques, et les signaux captés correspondent aux théories liées à chaque type de codage. Le code NRZT, avec ses transitions douces, se montre particulièrement adapté pour les transmissions analogiques.

Les tests effectués à l'aide de **JUnit** et la vérification de la couverture de code avec **Emma** ont confirmé la robustesse de notre implémentation. Nous avons également utilisé **EasyMock** pour simuler certaines composantes, ce qui a permis d'effectuer des tests plus exhaustifs.

En résumé, cette itération a non seulement permis de passer à une gestion des signaux analogiques, mais aussi de poser les bases pour la prochaine étape, où nous introduirons du bruit dans le canal de transmission. Le système est désormais prêt pour simuler des environnements de transmission plus réalistes et complexes.