



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

FIP SIT 213 – Atelier logiciel : systèmes de transmission

Étape 4 : Canal à trajets multiples et bruité

Année scolaire

2024-2025

Page : 1/45



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

IMT Atlantique – Ingénieur
spécialité informatique, réseaux,
télécommunications (FIP)

Rapport SIT 213 – Atelier logiciel : simulation d'un système de transmission

Étudiants	Encadrants
<p>Jordan BAUMARD Roman GUERRY Pol GUILLOU Mathis MAQUENNE Maxime PERE</p> <p>Étudiants FIP 2A – IMT Atlantique</p>	<p>Julien MALLET Enseignant chercheur au département informatique – IMT Atlantique</p> <p>François-Xavier SOCHELEAU Enseignant chercheur au département MEE – IMT Atlantique</p>

À destination de l'équipe pédagogique FIP

5 Septembre 2024 – 3 Octobre 2024



Table des matières

I. Introduction	6
II. Itération 1 : Transmission back-to-back	7
2.1. Cahier des charges	7
2.2. Ajout des composants au système	8
2.2.1. Création de la classe SourceFixe	8
2.2.2. Création de la classe SourceAleatoire.....	9
2.2.3. Création de la classe TransmetteurParfait.....	10
2.2.4. Création de la classe DestinationFinale	11
2.3. Mise en fonctionnement du système	12
2.3.1. Instanciation des composants et connexions avec les sondes.....	12
2.3.2. Émission du message par la source	13
2.3.3. Gestion du taux d'erreur binaire	13
2.4. Automatisation du logiciel via des scripts	14
2.4.1. Script compile	14
2.4.2. Script genDoc	15
2.4.3. Script cleanAll	15
2.4.4. Script genDeliverable	16
2.5. Tests et résultats de la première itération	18
2.5.1. Script simulateur	18
2.5.2. Script runTests	18
2.5.3. Vérification de l'itération 1 avec le script <i>Deploiement</i>	20
III. Itération 2 : Transmission analogique non bruitée	21
3.1. Cahier des charges	21
3.2. Implémentation des fonctionnalités	22
3.2.2. Fonctionnement de l'émetteur et du récepteur	22
3.2.3. Création de classes dédiées à la seconde étape	23
3.2.4. Connexions des sondes à la chaîne de transmission.....	24
3.2.3. Intégration des nouveaux paramètres à la commande permettant d'utiliser le logiciel	24
3.3. Résultats obtenus	25
3.3.1. Simulation d'un signal NRZ binaire	25
3.3.2. Simulation d'un signal RZ binaire	26
3.3.3. Simulation d'un signal NRZT binaire	26
3.4. Tests effectués	28



3.4.1. Tests de la classe Emettre	28
3.4.2. Tests de la classe Recepteur	28
3.4.3. Tests de la classe SourceFixe	29
3.4.4. Tests de la classe SourceAleatoire	30
3.4.5. Tests de la classe TransmetteurParfait	30
3.4.6. Tests de la classe DestinationFinale.....	31
3.4.7. Couverture du code Java avec Emma	31
3.4.8. Modification du script runTests	32
3.5. Bilan de l'itération.....	33
IV. Itération 3 : Transmission non idéal avec canal bruité de type « gaussien »	34
4.1. Cahier des charges	34
4.2. Implémentation des fonctionnalités.....	36
4.2.1. Création de la classe TransmetteurGaussien	36
4.2.2. Nouvelle méthode de réception du signal bruité	36
4.2.3. Intégration de l'ajout de bruit à la commande permettant d'utiliser le logiciel	37
4.3. Résultats obtenus	38
4.3.1. Simulations avec les paramètres par défaut.....	38
4.3.2. Modification de la taille du message	39
4.3.2. Comparaison entre la puissance de bruit obtenue et la variance	40
4.3.3. Mise en relation du TEB et du SNR.....	40
4.3.4. Comparaison avec les études théoriques	41
4.3.5. Vérification du type de bruit ajouté	42
4.4. Tests effectués	43
4.4.1. Tests de la classe TransmetteurGaussien.....	43
4.4.2. Tests du TEB	44
4.4.3. Modification du script runTests	44
4.5. Bilan de l'itération.....	45


 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Étape 4 : Canal à trajets multiples et bruité</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 4/45</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------

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


 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Étape 4 : Canal à trajets multiples et bruité</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 5/45</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------

Figure 33 - Valeur du signal analogique capté à la sortie de l'émetteur	26
Figure 34 - Commande permettant la simulation d'un signal RZ binaire	27
Figure 35 - Résultats du signal NRZT binaire	27
Figure 36 - Résultat des tests pour la classe Emettre.....	28
Figure 37 - Résultat des tests pour la classe Emettre.....	28
Figure 38 - Résultat des tests pour la classe SourceFixe	29
Figure 39 - Résultat des tests pour la classe SourceAleatoire	30
Figure 40 - Résultat des tests pour la classe TransmetteurParfait	30
Figure 41 - Résultat des tests pour la classe DestinationFinale	31
Figure 42 - Quantité de notre code couvert grâce aux tests JUnit	32
Figure 43 - Modification du script runTests pour l'itération 2.....	32
Figure 44 - Modélisation de la chaîne de transmission à l'étape 3	34
Figure 45 - Exemple de résultats attendus à la fin de la troisième étape pour un SNR valant 10 dB	35
Figure 46 - Diagramme de classe lié à l'itération 3	36
Figure 47 - Méthode de réception d'un signal bruité	37
Figure 48 - Simulation pour un code NRZT avec un SNR de 15 dB.....	38
Figure 49 – TEB et SNR pour les codages binaires NRZ, NRZT et RZ	39
Figure 50 – Simulation pour un code RZ avec un SNR de 9 dB et un message émis de 10000 booléens	39
Figure 51 - TEB et SNR pour un message de 10000 bits utilisant le code RZ binaire.....	39
Figure 52 - Corrélation entre la variance et la puissance moyenne du bruit	40
Figure 53 - Graphe montrant le TEB en fonction du SNR pour les codes binaires NRZ, NRZT et RZ (pratique)	40
Figure 54 - Graphe montrant le TEB en fonction du SNR pour les codes binaires NRZ, NRZT et RZ (théorie)	41
Figure 55 - Histogramme des valeurs du bruit blanc gaussien additif ajouté dans le canal de transmission	42
Figure 56 - Résultats des tests pour la classe TransmetteurGaussien	43
Figure 57 - Couverture du package transmetteurs grâce à l'outil Emma	43
Figure 58 - Résultat de la classe TebTest.....	44
Figure 59 - Modification du script runTests pour l'itération 3	44

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.

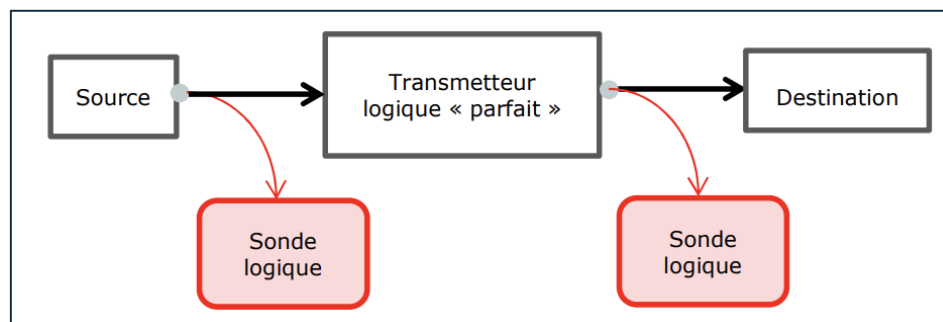


Figure 1 – Modélisation de la chaîne de transmission à l'étape 1

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.

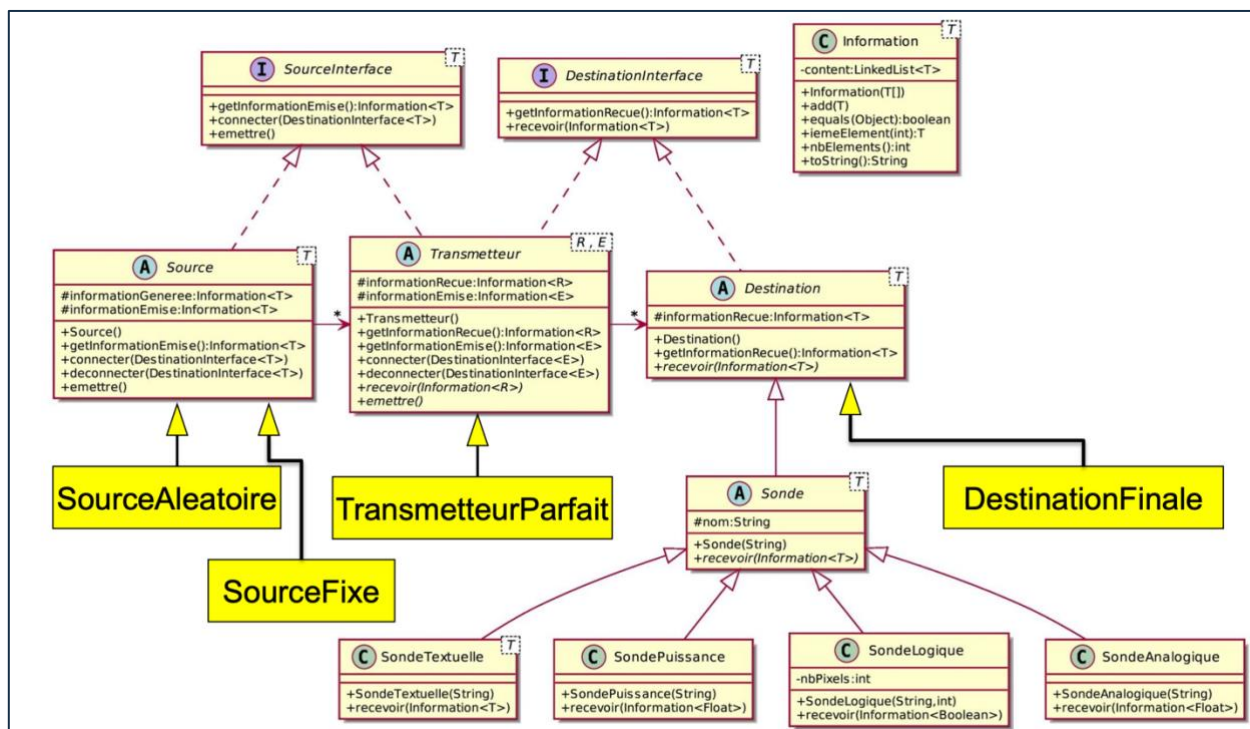


Figure 2 – Diagramme de classe correspondant à l'itération 1

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>Étape 4 : Canal à trajets multiples et bruité</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 10/45</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 passage 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**

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.
 */
// jordanbmrdr +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**

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.



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'.

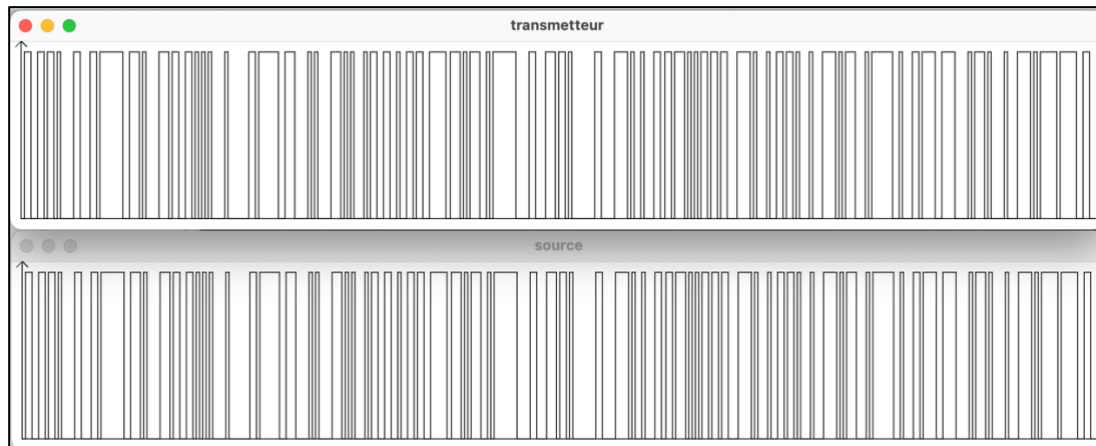


Figure 22 – Données remontées par les deux sondes

2.5.3. Vérification de l'itération 1 avec le script *Déploiement*

Pour conclure cette première itération, nous avons utilisé un nouveau script bash nommé *Déploiement*. 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 *Déploiement* nous annonce que le déploiement s'est terminé sans rencontrer d'erreur.

Figure 23 - Exécution du script *Déploiement*

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.

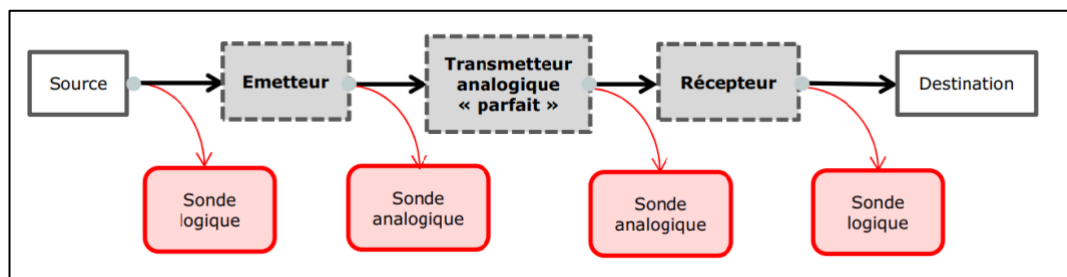


Figure 24 - Modélisation de la chaîne de transmission à l'étape 2

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.

Chaque code a ses spécificités qui nous sont également rappelées dans le cahier des charges. Une d'entre elles caractérise les amplitudes maximales et minimales que chaque code peut avoir. En effet, il faut que les règles suivantes soient appliquées :

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

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.

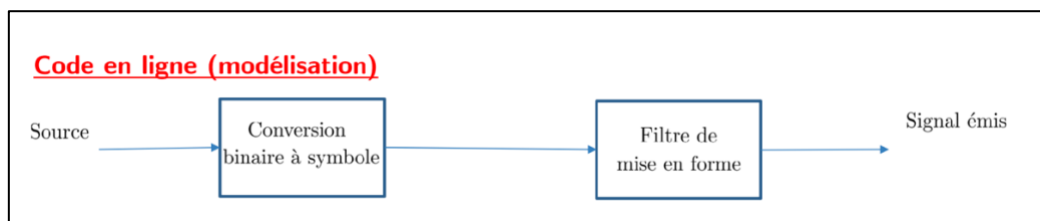


Figure 25 - Fonctionnement du bloc émetteur

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.

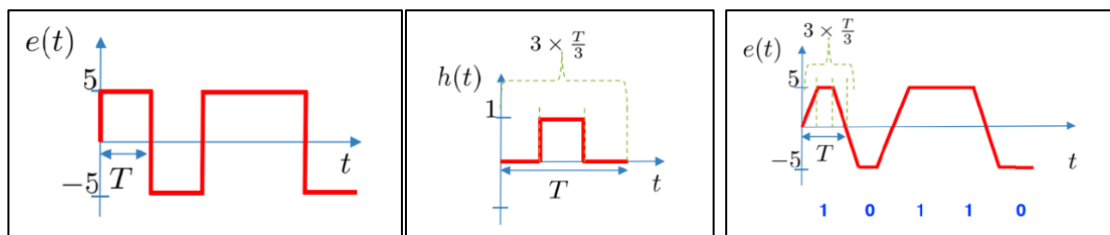



Figure 26 - Présentation des codes binaires en ligne NRZ, RZ et NRZT

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>Étape 4 : Canal à trajets multiples et bruité</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 23/45</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*.

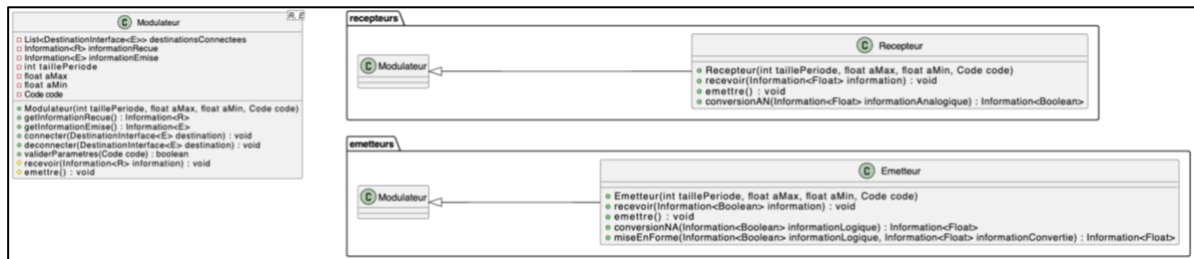


Figure 27 - Diagramme de classes lié à l'itération 2

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 décrites précédemment.

- 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 défini 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 la 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.

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.

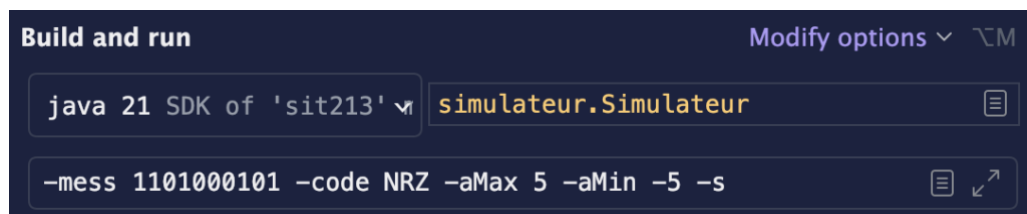
Le choix du code binaire pour l'utilisateur se fera grâce au paramètre « -form » suivi du nom du code choisit (**NRZ**, **NRZT** ou **RZ**). Pour ce qui est de l'amplitude, l'usage pourra se servir du paramètre « -ampl » suivi de la valeur des amplitudes minimales et maximales. Par défaut, le code binaire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

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.

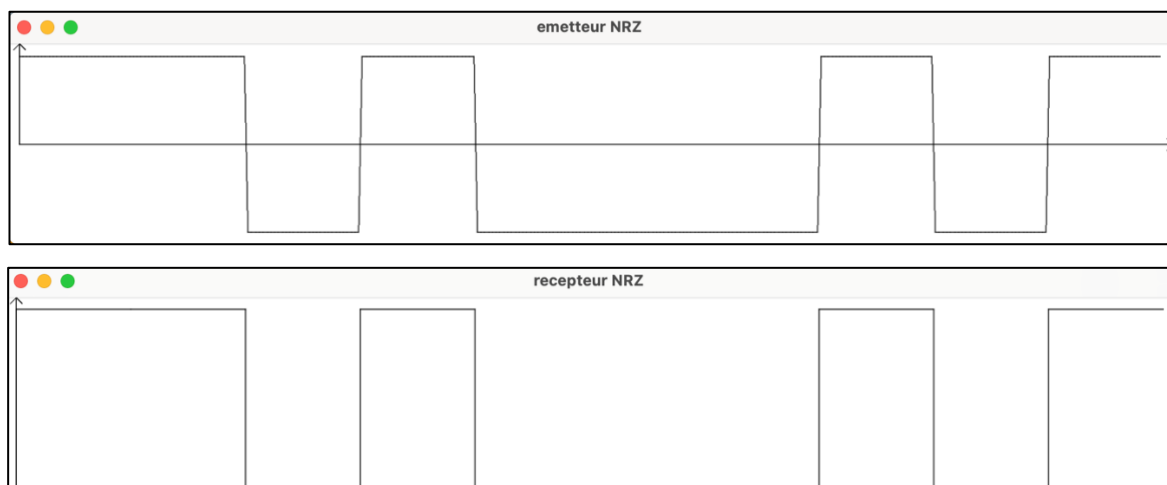


Figure 30 - Résultats du signal NRZ binaire

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.

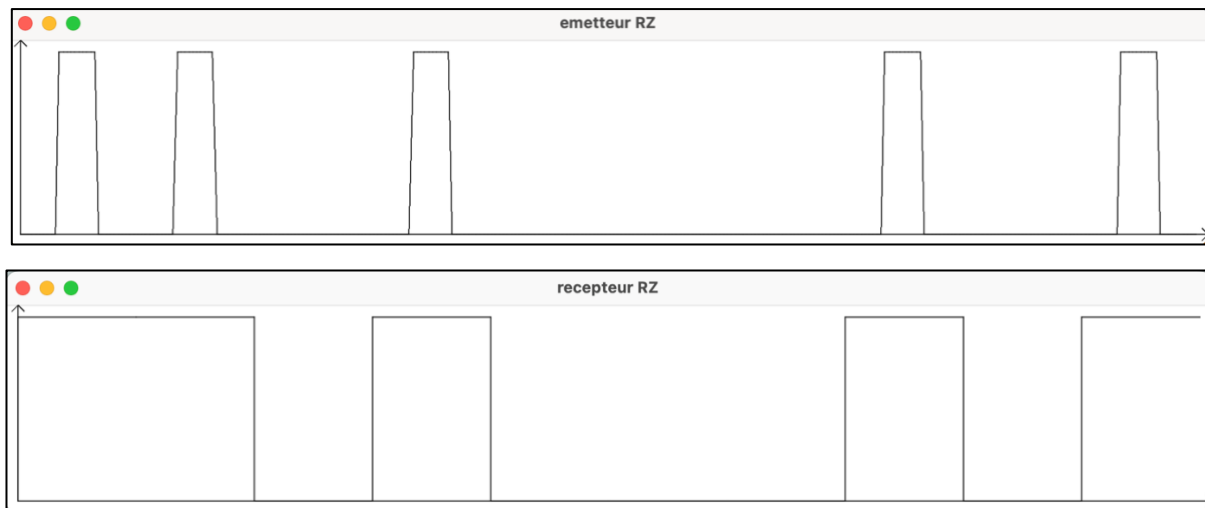


Figure 32 - Résultats du signal RZ binaire

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

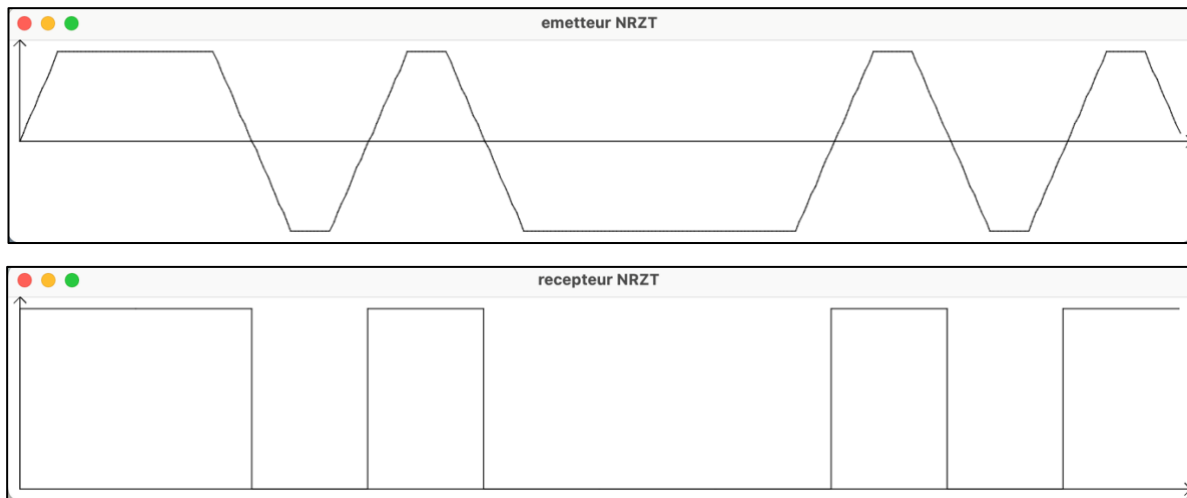


Figure 35 - Résultats du signal NRZT binaire

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.

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.

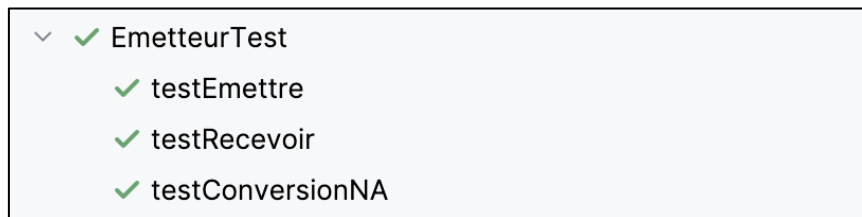


Figure 36 - Résultat des tests pour la classe Emettre

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>Étape 4 : Canal à trajets multiples et bruité</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 29/45</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.

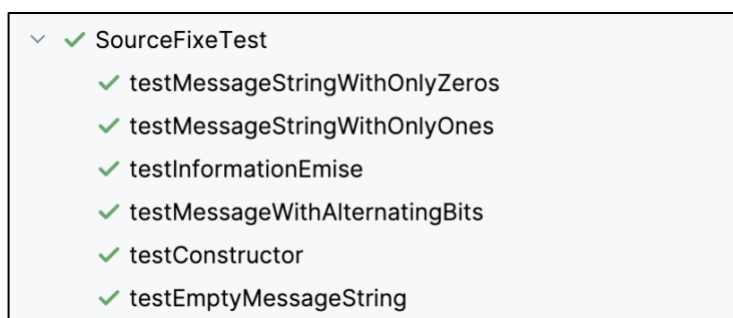


Figure 38 - Résultat des tests pour la classe **SourceFixe**

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 ;

- 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.

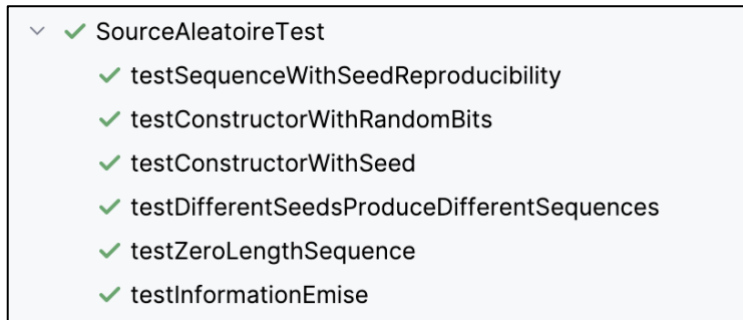


Figure 39 - Résultat des tests pour la classe SourceAleatoire

- 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.

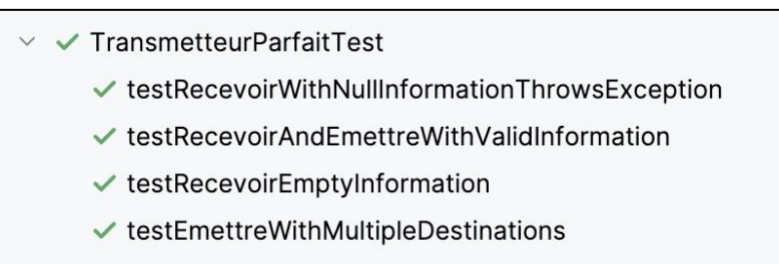


Figure 40 - Résultat des tests pour la classe TransmetteurParfait

- 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.

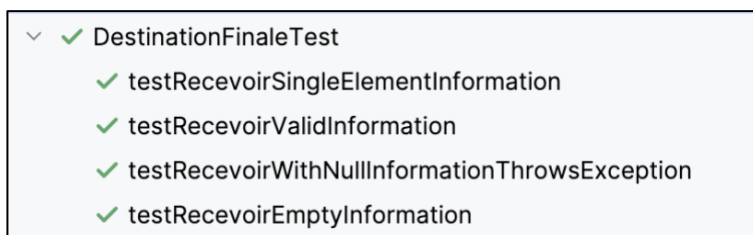


Figure 41 - Résultat des tests pour la classe DestinationFinale

- 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)

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 100111001110001111000011100000111000001110000111100001111000000000" # 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 pour l'itération 2

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.



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.

IV. Itération 3 : Transmission non idéal avec canal bruité de type « gaussien »

4.1. Cahier des charges

Cette troisième itération ajoute un changement majeur dans notre chaîne de transmission. En effet, jusqu'à présent nous étudions une transmission parfaite sans qu'aucun bruit ne vienne perturber le signal émis depuis la source vers la destination. Désormais, notre client demande que l'ajout de bruit au sein du canal de transmission, comme le montre la figure 44.

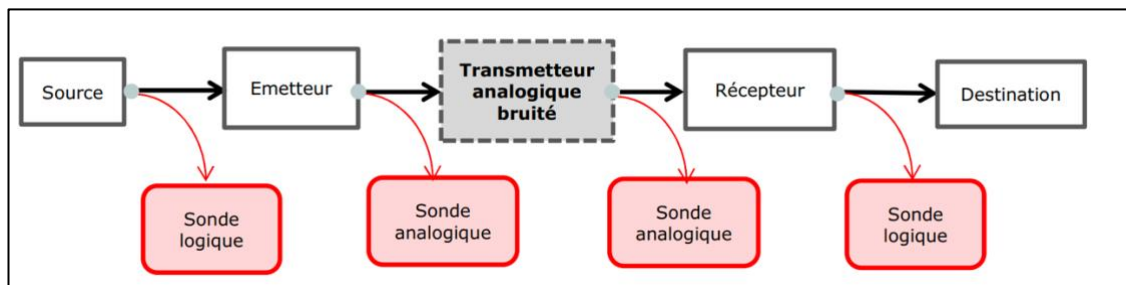


Figure 44 - Modélisation de la chaîne de transmission à l'étape 3

Nous pouvons remarquer sur la figure ci-dessus qu'à part le transmetteur analogique qui n'est plus parfait, notre chaîne de transmission reste la même en tous points. Mais l'étude de ce nouveau transmetteur est cruciale pour que notre logiciel réponde aux nouvelles demandes du client.

Dans un premier temps il nous est demandé que le bruit qui perturbe le signal soit un bruit **blanc additif gaussien**. Chacun de ces mots a une signification bien particulière :

- Un bruit **blanc** signifie que toutes les valeurs de ce bruit sont indépendantes et ont une puissance égale sur tout le spectre de fréquence ;
- Le fait qu'il soit **additif** indique simplement que ce bruit viendra s'ajouter au signal d'origine qui part de la source de notre structure ;
- Enfin, un bruit dit « **gaussien** » a pour particularité que la majorité des valeurs se situent autour de la moyenne, tout en suivant une distribution normale (ou gaussienne).

Maintenant que nous avons étudié le type de bruit à intégrer à notre système, il nous reste à déterminer comment le générer. Pour ce faire, nous utilisons la formule suivante :

$$b(n) = \sigma_b \sqrt{-2\ln(1 - a_1(n))} \cos(2\pi a_2(n)) \quad \begin{array}{l} a_1(n) \sim \mathcal{U}[0, 1[\text{ (loi uniforme)} \\ a_2(n) \sim \mathcal{U}[0, 1[\end{array}$$

Dans la formule affichée ci-dessus :

- $b(n)$ est la valeur du bruit blanc gaussien à l'indice n ;
- σ_b représente l'écart-type du bruit gaussien ;
- $a_1(n)$ et $a_2(n)$ les variables aléatoires uniformes comprise entre 0 et 1 qui permettent de créer les valeurs suivant une distribution normale.

L'objectif de cette troisième étape est donc d'ajouter le bruit blanc gaussien présenté précédemment à notre signal d'origine provenant du bloc émetteur de notre système. Grâce à cela, chaîne de transmission ressemblera plus à ce que l'on peut retrouver en situation réelle. Le rôle du récepteur placé au bout du canal sera donc plus important puisqu'il devra, en plus de décoder le signal analogique, supprimer le bruit qui aura été ajouté afin de retrouver le signal d'origine, avec le moins d'erreur possible.

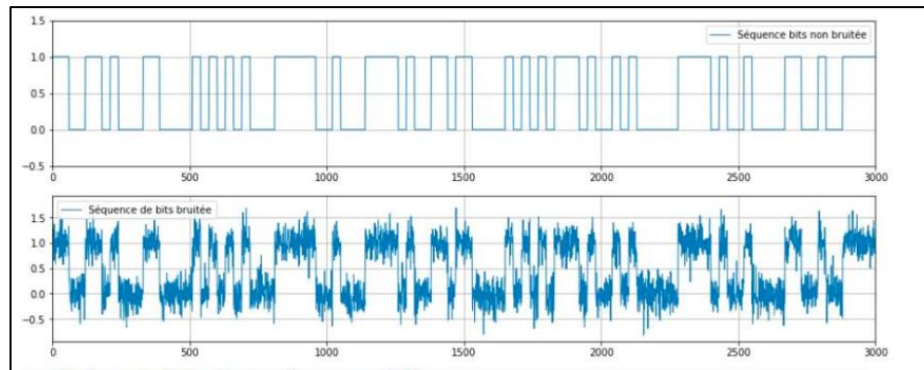


Figure 45 - Exemple de résultats attendus à la fin de la troisième étape pour un SNR valant 10 dB

Afin de réaliser différentes simulations, l'utilisateur final pourra préciser, en dB, le niveau de bruit qu'il souhaite ajouter au canal de transmission. Par défaut ce sera un transmetteur analogique parfait qui sera utilisé. La valeur qui nous intéressera le plus sera bien sur le taux d'erreur binaire (TEB). Pour rappel, ce calcul compare la chaîne émise par la source, et celle reçu par la destination. Dans le cas d'un transmetteur parfait ou quasi-parfait le TEB vaut 0 puisqu'aucune erreur n'est détecté. Il sera donc intéressant de voir, pour chaque code binaire, à partir de quel seuil de bruit des erreurs commencent à apparaître.

Enfin, un aspect essentiel de l'analyse du canal bruité est le rapport signal-bruit (SNR), défini ici comme $\frac{E_b}{N_0}$, où E_b représente l'énergie par bit, et N_0 la densité spectrale du bruit. Ce rapport est crucial pour évaluer les performances de la transmission car il détermine directement le taux d'erreur binaire (TEB). En effet, plus le SNR est élevé, moins le bruit a d'influence sur le signal, réduisant ainsi les erreurs lors de la transmission. Inversement, un faible SNR augmente le risque d'erreurs. Ce rapport est particulièrement important pour ajuster la puissance du signal en fonction du niveau de bruit, permettant d'optimiser la qualité de la transmission dans des conditions bruitées. Dans notre simulation, le SNR sera contrôlé pour observer l'impact du bruit blanc gaussien sur le taux d'erreur binaire. Pour obtenir ce SNR en dB nous utilisons la formule suivante : $(\frac{E_b}{N_0})_{dB} = 10 \log_{10}(\frac{P_s \times N}{2\sigma_b^2})$ avec :

- P_s la puissance du signal ;
- N le nombre d'échantillons par bit, qui est un des paramètres intégrés à notre logiciel lors de l'étape précédente ;
- σ_b^2 la puissance du bruit.

4.2. Implémentation des fonctionnalités

Comme lors de la seconde itération, les nouvelles fonctionnalités présentées dans la partie précédente nous obligent à modifier notre code Java afin que notre logiciel les prenne en compte. L'élément de notre système qui gère l'addition de bruit à notre signal d'origine est le transmetteur. Il nous faudra donc ajouter une classe dédiée à ce transmetteur gaussien. L'étude du cahier des charges nous a également montrée que le récepteur aura pour nouveau rôle la suppression de ce même bruit, afin de retransmettre à la destination le signal d'origine. Enfin, la classe mère de notre logiciel nommée **Simulateur** devra elle aussi être revue afin de traiter les nouveaux paramètres liés à la troisième étape de ce projet.

4.2.1. Création de la classe TransmetteurGaussien

Pour rappel, notre logiciel contenait une classe **TransmetteurParfait** qui s'occupait simplement de retransmettre le signal qu'il recevait de l'émetteur. Cette classe hérite de la classe abstraite *Transmetteur*. Nous avons donc simplement ajouté une nouvelle classe **TransmetteurGaussien**, qui hérite elle aussi de *Transmetteur*. Le diagramme de classe regroupant ces trois classes et affiché ci-dessous.



Figure 46 - Diagramme de classe lié à l'itération 3

La figure 46 nous permet de remarquer que la nouvelle classe **TransmetteurGaussien** est bien plus complexe que son homonyme **TransmetteurParfait**. En effet, en plus du fait d'émettre et de recevoir des informations, notre nouvelle classe doit également ajouter un signal de type bruit blanc gaussien au signal émis par l'émetteur. Pour faire cela, on génère un signal correspondant au bruit blanc gaussien. On parcourt ensuite le signal reçu et, pour chaque échantillon de ce signal, on ajoute une valeur du signal bruité. Ainsi, on se retrouve avec notre signal d'origine mélangé au bruit blanc gaussien. C'est cette addition de signal que le nouveau transmetteur envoie vers le récepteur du système.

4.2.2. Nouvelle méthode de réception du signal bruité

Le second changement important que nous avons implémenté à notre code Java concerne le récepteur. Lors de la seconde étape, nous avons décidé de regarder pour chaque bit les valeurs minimales et maximales présentes. Nous les comparons ensuite à l'amplitude déterminée par l'utilisateur du logiciel et cela nous permettait de déterminer si le bit passé en entrée de la chaîne de transmission était un « 0 » ou un « 1 ». Or, l'ajout de bruit bloque

totalemment cette méthode puisque qu'une valeur déjà maximale peut être encore augmentée, et une valeur correspondant à l'amplitude minimale peut se retrouver diminuer.

Nous avons donc dû opter pour une toute nouvelle méthode, basée cette fois sur la valeur moyenne de chaque bit. Pour les codages binaires NRZT et RZT nous incrémentons un compteur avec toutes les valeurs du signal bruité qui compose le bit étudié. Une fois que nous avons parcouru le bit en entier nous calculons la puissance moyenne du bit.

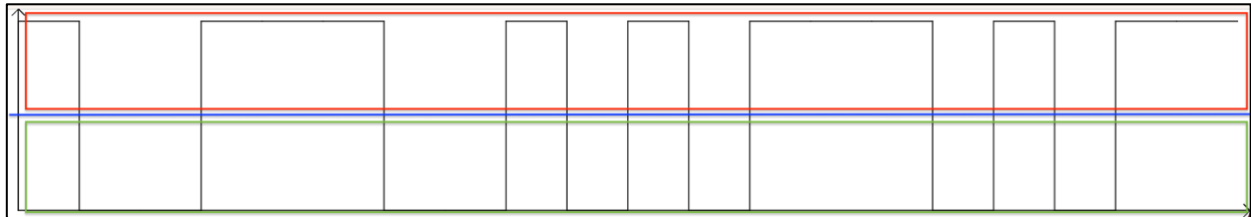


Figure 47 - Méthode de réception d'un signal bruité

Si la puissance moyenne calculé se situe dans la zone rouge, c'est-à-dire que

$P \geq \frac{aMin+aMax}{2}$, le signal débruité vaudra « 1 ». Si, à l'inverse, la puissance se situe dans la zone verte, c'est-à-dire que $P \leq \frac{aMin+aMax}{2}$, la valeur finale vaudra « 0 ». Pour le codage RZ binaire nous ne prenons que les échantillons présents dans le deuxième tier du bit, puisqu'il vaut 0 aux deux extrémités.

4.2.3. Intégration de l'ajout de bruit à la commande permettant d'utiliser le logiciel

Pour cette troisième étape, la modification de la classe **Simulateur** consistait simplement à prendre en compte du nouveau paramètre snrpb spécifié par notre client.

- Si le paramètre est spécifié, nous utilisons une transmission analogique bruitée. La valeur donnée par l'utilisateur correspond au rapport signal à bruit par bit en dB. Pour ce faire nous utilisons la classe **TransmetteurGaussien** présentée précédemment ;
- Si aucun bruit n'est spécifié, le signal transitant dans la chaîne de transmission n'est pas bruité. Dans ce cas, nous nous servons de la classe **TransmetteurParfait**.

4.3. Résultats obtenus

Maintenant que les fonctionnalités incluses dans cette troisième étape font partie intégrante de notre code, nous pouvons lancer des simulations qui ajoutent du bruit au signal d'origine au sein du canal de propagation. Pour faire cela, il nous suffit d'exécuter la classe Simulateur en faisant varier l'amplitude de notre signal, le bruit que nous ajoutons, le type de codage binaire utilisé...

4.3.1. Simulations avec les paramètres par défaut

Pour commencer nous décidons d'utiliser les paramètres par défaut afin de comparer seulement les trois codes binaires mis à notre disposition. Nous nous retrouvons donc avec les valeurs suivantes :

- Une amplitude allant de 0.0 à 1.0
- Chaque bit est composé de 30 échantillons
- Le message envoyé est un message aléatoire d'une longueur de 100 bits

Comme attendu, quand nous utilisons un SNR relativement élevé (15 dB ou plus), aucune erreur n'est détectée à la sortie de la chaîne de transmission, quel que soit le codage binaire utilisé.

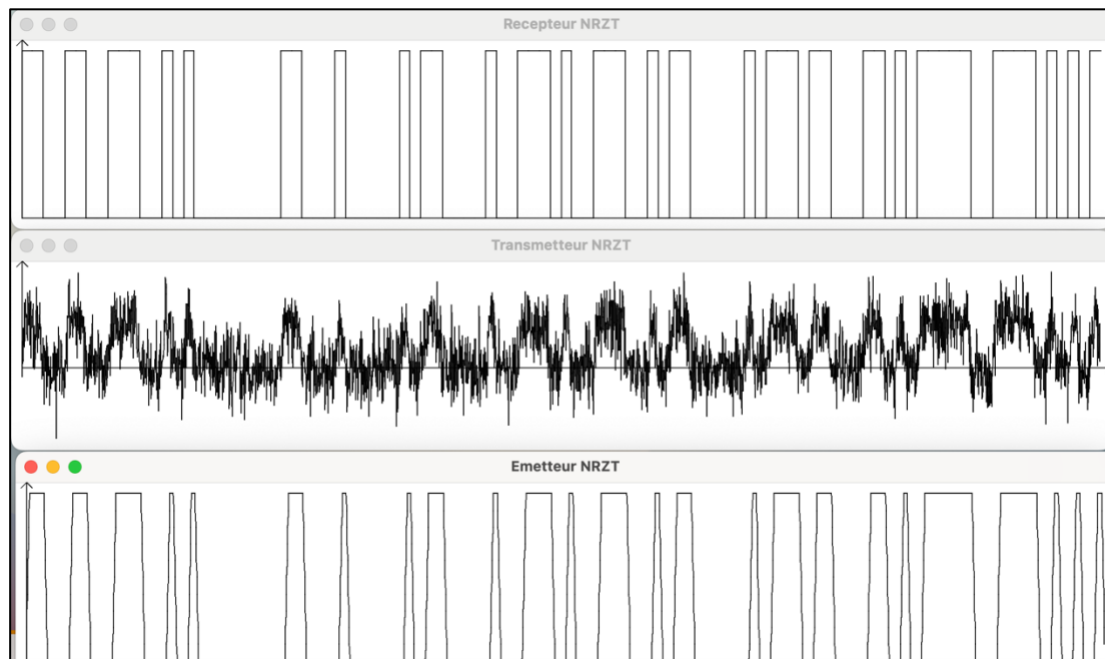


Figure 48 - Simulation pour un code NRZT avec un SNR de 15 dB

La figure ci-dessus montre le résultat pour le codage NRZT binaire, mais les sondes renvoient des résultats très similaires pour les deux autres codages, à l'exception de la mise en forme qui varie. Le seul paramètre que nous modifions pour cette première batterie de résultats est le rapport signal à bruit. En diminuant cette valeur, nous observons que des erreurs sont introduites dès que le SNR atteint environ 12 dB pour les codages binaires NRZT et NRZ. Le code RZ binaire réagit différemment puisqu'avec un SNR valant 9 dB ou plus le TEB vaut toujours 0. Les figures ci-dessous illustrent ce commentaire. Elles montrent également que le SNR par bit détecté par le logiciel grâce à la formule $\left(\frac{E_b}{N_0}\right)_{dB} = 10 \log_{10}\left(\frac{P_s \times N}{\sigma_B^2}\right)$ est très proche de la valeur rentrée en paramètre.


```
java Simulateur -mess 100 -form NRZ -snrpb 12 java Simulateur -mess 100 -form NRZT -snrpb 12 java Simulateur -mess 100 -form RZ -snrpb 9
=> TEB : 0.05                               => TEB : 0.03                               => TEB : 0.0
=> SNR par bit demandé : 12.0                 => SNR par bit demandé : 12.0                 => SNR par bit demandé : 9.0
=> SNR par bit réel obtenu : 11.761269         => SNR par bit réel obtenu : 11.769749         => SNR par bit réel obtenu : 8.913521
```

Figure 49 – TEB et SNR pour les codages binaires NRZ, NRZT et RZ

Pour rappel, lorsque le récepteur a affaire à un signal utilisant le codage RZ, il ne regarde que le tiers central de chaque bit. Ainsi, il y a trois fois moins d'échantillons qui sont analysés comparés aux deux autres codages. Nous pouvons donc penser qu'avec un nombre d'échantillons réduit à analyser les performances seraient meilleures ou, à l'inverse, utiliser un codage RZ dans des conditions moins favorables augmenterait le TEB pour un même rapport signal à bruit.

Par ailleurs, nous restons très loin du pire TEB théorique que l'on pourrait avoir qui est 0.5. En effet, avoir un taux d'erreur binaire valant 0.5 signifie que la moitié des bits du signal reçu sont faux mais nous ne pouvons pas savoir lesquels. Il est même préférable d'avoir un TEB valant par exemple 0.7, puisqu'il nous suffirait d'inverser le message reçu. Ainsi, le nouveau TEB vaudrait $1 - 0.7 = 0.3$.

4.3.2. Modification de la taille du message

La première façon de vérifier notre hypothèse est de changer la taille du message émis par la source de notre système. Nous augmentons la taille de ce dernier pour avoir une chaîne de 10000 booléens en entrée de la chaîne de transmission. Cette taille correspond à un e-mail qui serait envoyé sans pièce jointe, donc un exemple typique de signal qui transite en permanence.

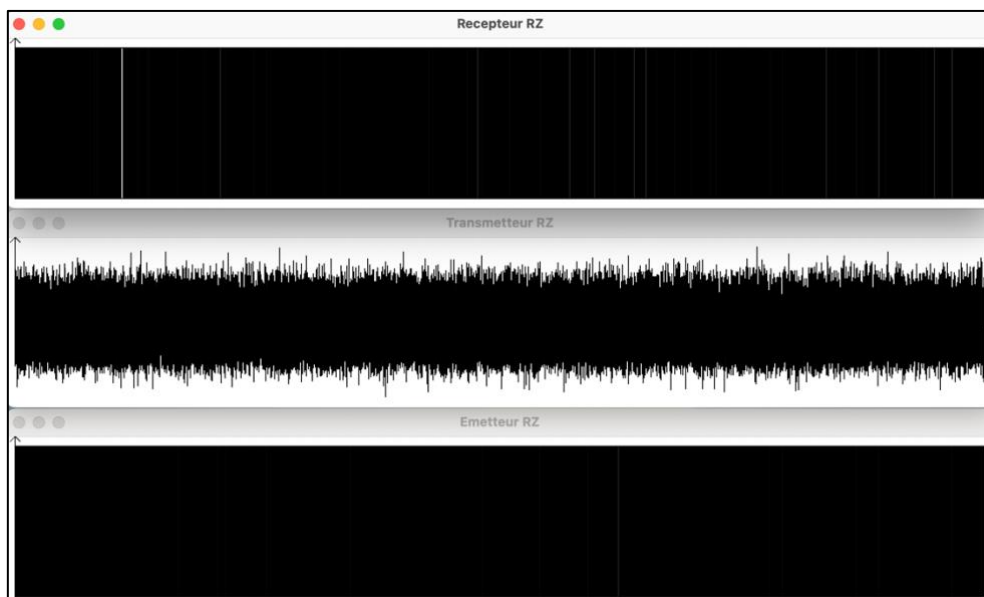


Figure 50 – Simulation pour un code RZ avec un SNR de 9 dB et un message émis de 10000 booléens

```
java Simulateur -mess 10000 -form RZ -snrpb 9 -s
=> TEB : 0.003
=> SNR par bit demandé : 9.0
=> SNR par bit réel obtenu : 8.994322
```

Figure 51 - TEB et SNR pour un message de 10000 bits utilisant le code RZ binaire

Bien que le code RZ reste dans ce cas encore plus efficace que NRZ et NRZT pour un même SNR (environ 0.07 pour un SNR de 9 dB), nous pouvons remarquer que son taux d'erreur binaire n'est plus nul. La théorie présentée dans le chapitre précédent s'avère donc réelle.

4.3.2. Comparaison entre la puissance de bruit obtenue et la variance

Le deuxième point que nous pouvons analyser concerne la puissance moyenne du bruit que nous ajoutons au signal d'origine et son lien avec la variance. En effet, nous pouvons remarquer sur les figures ci-dessous que, pour l'ensemble des simulations que nous exécutons, ces deux attributs ont des valeurs extrêmement proches.

```
java Simulateur -mess 500 -form RZ -snrpb 3 -ampl 0 5
=> TEB : 0.098
- Nombre de bits de la séquence : 500
- Nombre d'échantillons par bit : 30
=> Puissance moyenne du bruit : 32.3138
=> Variance : 32.827763
=> Rapport signal-sur-bruit (S/N, en dB) : 3.0685327

java Simulateur -seed 127834 -form NRZT -snrpb -9 -ampl -4 4
=> TEB : 0.35
- Nombre de bits de la séquence : 100
- Nombre d'échantillons par bit : 30
=> Puissance moyenne du bruit : 1503.0925
=> Variance : 1492.2595
=> Rapport signal-sur-bruit (S/N, en dB) : -9.031413
```

Figure 52 - Corrélation entre la variance et la puissance moyenne du bruit

Notre bruit étant un signal de type bruit blanc gaussien, sa moyenne est nulle. Dans ce cas l'espérance de x^2 (le carré de l'amplitude) est directement égale à la variance σ^2 . Autrement dit, la puissance moyenne du bruit est égale à sa variance, car la moyenne est nulle et la variance mesure justement l'écart au carré des valeurs autour de cette moyenne. C'est pour cette raison que ces deux valeurs sont très proches.

4.3.3. Mise en relation du TEB et du SNR

Nous avons vu précédemment que, pour les trois codes binaires étudiés, plus le SNR en dB était faible, plus le nombre d'erreurs détectées à la sortie du système était élevé. Afin de vérifier cela, nous avons tracé le taux d'erreur binaire retourné par la chaîne de transmission pour un SNR différent à chaque itération. Nous avons lancé 75 simulations avec un SNR qui variait de 15 à -10 dB et un message de 100000 bits à chaque itération.

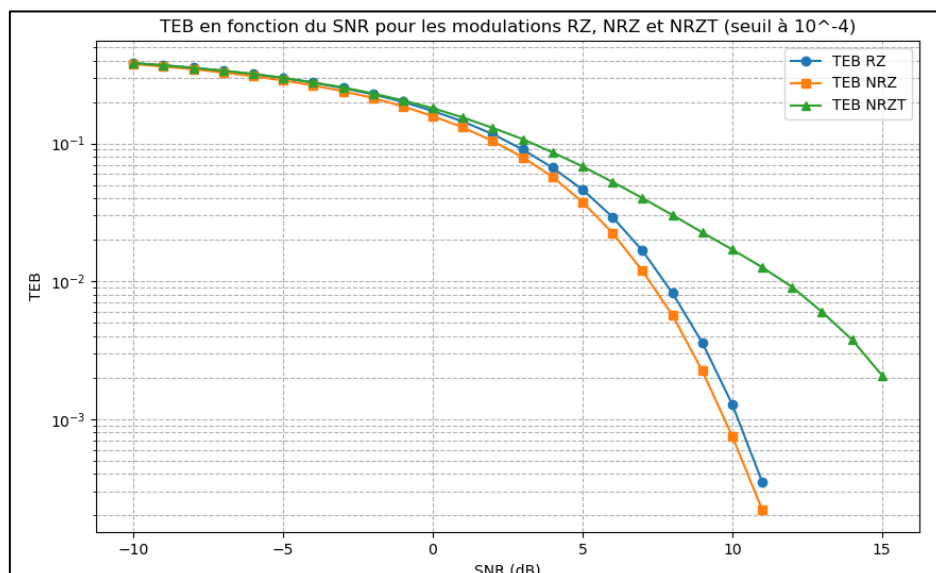


Figure 53 - Graphe montrant le TEB en fonction du SNR pour les codes binaires NRZ, NRZT et RZ (pratique)

Nous avons décidé de représenter le TEB par rapport au SNR sur une échelle semi-logarithmique, afin de pouvoir observer la variation du TEB sur plusieurs ordres de grandeur. De plus, cela nous permet de comparer nos résultats avec ceux connus dans des exemples de la vie courante. Par exemple, dans le cas d'un signal vidéo le TEB demandé est d'environ 10^{-9} , c'est-à-dire une erreur sur un bit pour un milliard de bit envoyé par la source.

En étudiant le graphe de la figure 53 nous pouvons remarquer que :

- Comme nous l'avons remarqué, le TEB diminue au fur et à mesure que le SNR augmente pour les trois types de modulation ;
- Le code NRZT semble être le moins performant pour un rapport signal à bruit élevé (de 5dB à 15dB). En effet, nous pouvons remarquer que le taux d'erreur binaire n'est jamais mieux que $2 \cdot 10^{-3}$;
- Pour ces mêmes valeurs de SNR, les codes binaires NRZ et RZ ont des performances très similaires, bien que le code NRZ semble un tout petit peu mieux ;
- Cependant, pour des valeurs de SNR en dB négatives, nous pouvons observer que les trois codes binaires se valent.

Ces différences entre les codes binaires peuvent être dû à la mise en forme qui n'est pas la même ou au fait que nous effectuons nos tests avec un nombre de bits relativement limité (100000 dans notre cas contre parfois des millions, voir des milliards de bits, dans des systèmes de communications modernes).

4.3.4. Comparaison avec les études théoriques

La partie précédente nous a montré le taux d'erreur binaire en fonction du SNR avec notre logiciel. Pour valider nos résultats, nous devons comparer ceux obtenus sur la figure 53 avec les courbes TEB par rapport au SNR théoriques pour chacun des codes binaires exécutés. Avec l'aide des équipes pédagogiques et de nos recherches personnelles nous avons déterminé les formules suivantes pour obtenir le TEB théorique en fonction du SNR :

- NRZ binaire et NRZT binaire $\rightarrow TEB = \frac{1}{2} \times \text{erfc}\left(\sqrt{\frac{Eb}{N0}}\right)$
- RZ binaire $\rightarrow TEB = \frac{1}{2} \times \text{erfc}\left(\frac{1}{\sqrt{2}} \times \sqrt{\frac{Eb}{N0}}\right)$

Nous visualisons ensuite les courbes résultantes sur le graphique ci-dessous. Nous avons choisi d'utiliser également une échelle semi-logarithmique afin de pouvoir comparer plus simplement les résultats théoriques avec ceux obtenus auparavant.

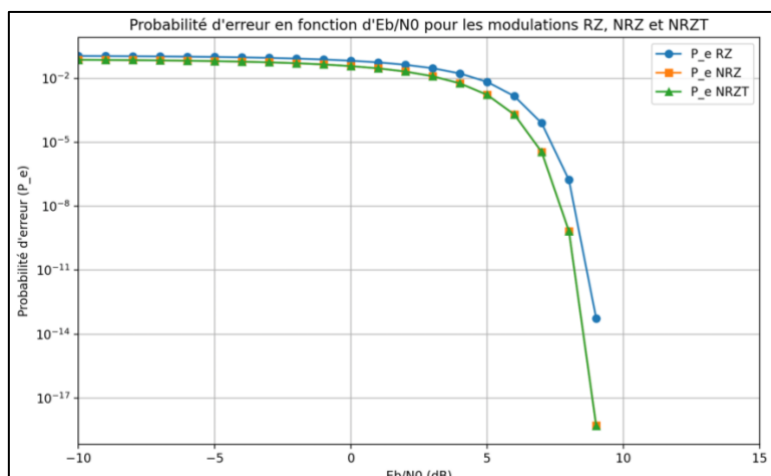


Figure 54 - Graphe montrant le TEB en fonction du SNR pour les codes binaires NRZ, NRZT et RZ (théorie)

Nous pouvons remarquer sur la figure ci-dessus que les signaux RZ et NRZ binaire sont très semblables à ce que nous obtenons avec notre logiciel. Cela confirme le fait que les résultats obtenus avec notre système de communication sont bons. En revanche, le TEB / SNR pour le code binaire NRZT est le même que pour le code NRZ puisque nous avons utilisé la même formule. Cela est dû au fait que nous n'avons pas réussi à trouver la bonne formule théorique pour ce troisième type de codage.

4.3.5. Vérification du type de bruit ajouté

Enfin, nous avons voulu vérifier que le bruit ajouté par le canal de transmission est bien gaussien. Pour rappel, les valeurs d'un bruit gaussien doivent suivre une loi normale (ou gaussienne) et être, pour la majorité d'entre elles, situées autour de la moyenne. Pour vérifier cela nous avons pris toutes les valeurs de bruit pour tous les échantillons d'un signal (plusieurs dizaines de milliers) et nous les avons mis sous la forme d'un histogramme. Cet histogramme est affiché en figure 54.

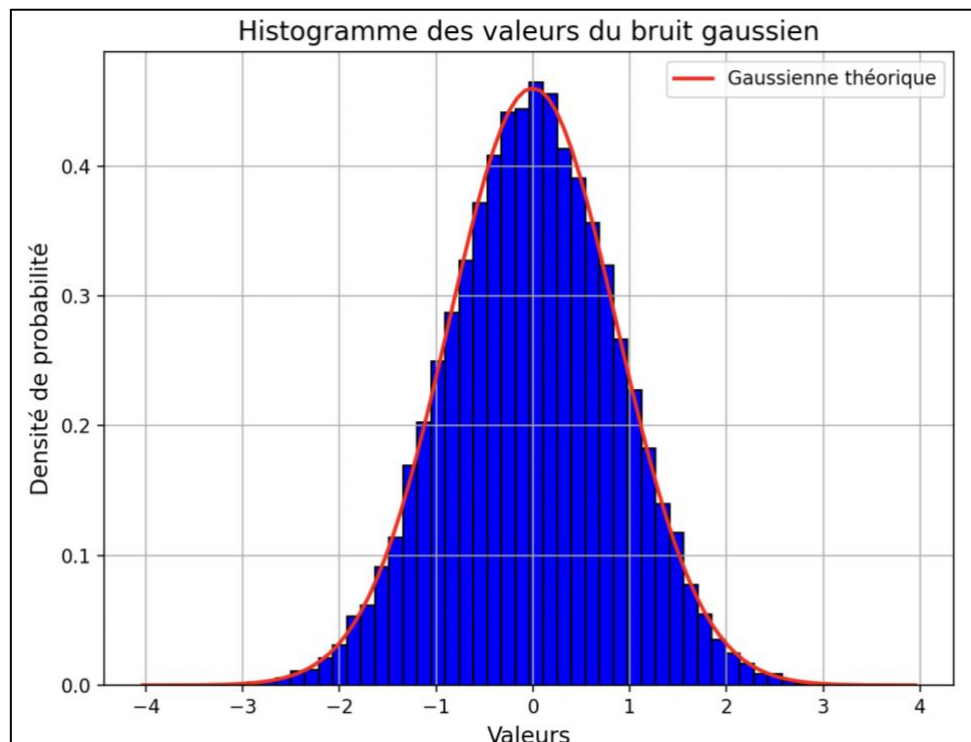


Figure 55 - Histogramme des valeurs du bruit blanc gaussien additif ajouté dans le canal de transmission

Nous pouvons observer sur la figure 54 que le bruit additif qui est produit par notre système ressemble parfaitement à une distribution gaussienne. Pour montrer cela nous avons affiché en rouge la densité spectrale de puissance (DSP) de notre bruit. Puisqu'il est gaussien, l'aire sous cette fonction vaut 1. Nous avons ensuite dimensionné notre histogramme pour que les valeurs maximales atteignent le maximum de la DSP. Nous pouvons observer que les valeurs du bruit remplissent parfaitement l'aire définie par la DSP. Nous en déduisons que le bruit créé aléatoirement par le canal de transmission suit bien une loi gaussienne.

4.4. Tests effectués

Notre logiciel incluant une nouvelle classe et de nouvelles fonctionnalités, nous avons ajouté de nouveaux tests en utilisant les mêmes outils que précédemment (JUnit 4, Emma et EasyMock).

4.4.1. Tests de la classe TransmetteurGaussien

Les tests effectués sur cette classe permettent de vérifier que la fonction de bruitage du signal, intégrée dans le transmetteur de notre système, fonctionne comme prévu. La figure ci-dessous montre que les quatre tests ont été exécutés sans problème.



Figure 56 - Résultats des tests pour la classe TransmetteurGaussien

La figure 55 liste les tests qui sont effectués sur la classe **TransmetteurGaussien** :

- testAjoutBruit vérifie que l'information bruitée est correctement reçue par la destination ;
- testEmettreAvecValeursNegatives teste la réception d'un signal bruité contenant des valeurs négatives ;
- testRecevoirInformationVide s'assure que lorsqu'un signal vide est émis, un signal vide est également réceptionné à la sortie du système.

En plus des tests JUnit 4 présentés ci-dessus, l'outil de couverture Emma nous permet de regarder à quel point la classe **TransmetteurGaussien** est couverte par des tests.

Element ^	Class, %	Method, %	Line, %	Branch, %
▼ [icon] transmetteurs	66% (2/3)	63% (12/19)	72% (42/58)	60% (12/20)
[icon] Transmetteur	100% (1/1)	50% (3/6)	66% (6/9)	100% (0/0)
[icon] TransmetteurGaussien	100% (1/1)	81% (9/11)	85% (36/42)	75% (12/16)
[icon] TransmetteurParfait	0% (0/1)	0% (0/2)	0% (0/7)	0% (0/4)

Figure 57 - Couverture du package transmetteurs grâce à l'outil Emma

Nous pouvons remarquer grâce à la figure 56 que plus de 80% des méthodes de cette classe sont couvertes par les tests JUnit. Les deux méthodes qui ne sont pas testées sont :

- une méthode qui ne fait que retourner des valeurs ;
- un constructeur simplifié dont la version complexe a déjà été testée.

4.4.2. Tests du TEB

La seconde série de tests dédiée à cette itération ne concerne cette fois pas une classe en particulier, mais le calcul du taux d'erreur binaire. Ce résultat étant essentiel pour analyser nos résultats, nous devons être sûrs qu'il est bien renvoyé. Pour tester cela nous avons créé une classe `TebTest` qui effectue le test suivant :

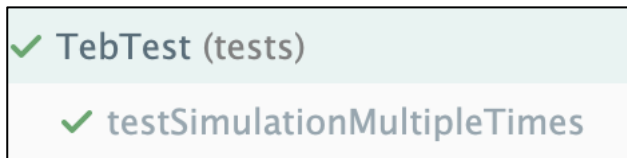


Figure 58 - Résultat de la classe `TebTest`

Le test `testSimulationMultipleTimes` lance une simulation pour les codages binaires NRZT, RZ et NRZ et vérifie qu'un résultat est correctement renvoyé à chaque fois que le signal a atteint sa destination.

4.4.3. Modification du script `runTests`

Comme pour les étapes précédentes, nous effectuons toute une série de tests grâce au script `runTests.sh`. Les nouveaux tests que ce script comprend nous permettent d'effectuer des simulations en combinant tous les paramètres que notre logiciel gère. La partie du script qui comprend les tests associés à la troisième étape de notre projet est présentée 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 0010011 -form NRZ" # Test avec codage NRZ
  "-mess 0010011 -form RZ" # Test avec codage RZ
  "-mess 0010011 -form NRZT" # Test avec codage NRZT
  "-ampl -5 5 -form NRZT" # Test avec des amplitudes
  "-mess 50 -seed 1234" # Test avec une seed spécifique
  "-mess 1001110011001100001111000011100000111000001110000111100000000000" # Message long
  "-mess 1000000" # Test avec un message d'un million de bits

  "-mess 1010101 -snrpb -20" # SNR par bit faible
  "-mess 110011 -snrpb 50" # SNR par bit élevé


  "-mess 200 -form NRZ -snrpb 30" # SNR par bit avec format NRZ
  "-mess 100 -form NRZ -snrpb 0" # SNR par bit avec format NRZ
  "-mess 101010100 -form NRZ -snrpb -30" # SNR par bit avec format NRZ
  "-mess 1000 -form RZ -snrpb -30" # SNR par bit avec format RZ
  "-seed 1234 -form RZ -snrpb 0" # SNR par bit avec format RZ
  "-seed 812873 -form RZ -snrpb 30" # SNR par bit avec format RZ
  "-mess 23 -form NRZT -snrpb -30" # SNR par bit avec format NRZT
  "-mess 2000 -form NRZT -snrpb 0" # SNR par bit avec format NRZT
  "-mess 1010000101 -form NRZT -snrpb 30" # SNR par bit avec format NRZT

  "-ampl -3 3 -form NRZT -snrpb 30" # SNR par bit avec amplitudes personnalisées
)

# 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 -form NRZT -ampl -4 4"
    "-s -seed 27 -form RZ -ampl 0 6"
    "-s -mess 12 -seed 1234 -form NRZ -ampl -5 5"
  )
fi
```

Figure 59 - Modification du script `runTests` pour l'itération 3

Comme nous l'avons présenté dans la partie 4.3, nous effectuons également des tests visuels à l'aide des différentes sondes connectées à notre chaîne de transmission.

 <p>IMT Atlantique Bretagne-Pays de la Loire École Mines-Télécom</p>	<p>FIP SIT 213 – Atelier logiciel : systèmes de transmission</p> <p>Étape 4 : Canal à trajets multiples et bruité</p>	<p>Année scolaire</p> <p>2024-2025</p> <p>Page : 45/45</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------

4.5. Bilan de l'itération

Cette troisième itération a abordé la simulation d'une transmission non idéale via un canal bruité de type « gaussien ». La gestion de notre système s'est donc révélée plus complexe par rapport aux itérations précédentes qui concernaient des transmissions idéales ou analogiques non bruitées.

Les modifications apportées pour gérer le bruit gaussien ont été testées rigoureusement, montrant que notre simulateur peut maintenir un taux d'erreur binaire contrôlé. Ces tests ont confirmé l'efficacité des ajustements réalisés, conformément aux attentes théoriques.

Les résultats obtenus dans la partie 4.3. nous ont montré qu'en fonction de la valeur du SNR (rapport signal à bruit) certains types de codages binaires obtenaient un meilleur TEB que les autres. C'était notamment le cas pour le codage RZ binaire.

Cet ajout de complexité a exigé une compréhension approfondie de l'impact du bruit sur les performances du système. Les améliorations apportées aux scripts de tests et de simulation nous ont permis d'améliorer l'efficacité de notre logiciel, notamment avec une gestion différente de la réception des messages.

En conclusion, cette itération a non seulement testé la capacité de notre système à gérer des conditions de transmission réalistes. Elle nous a également permis d'approfondir nos connaissances en matière de signaux présents au sein d'environnements bruités, et des paramètres qui les accompagnent.