Composant 3 : Mineur :
Groupe : Axel Acuna, Olivier Klak, Fethi Zerara, Mohamed Hamici

## Introduction :

Dans le cadre du cours de programmation par composante, nous devons créer la composante permettant d'effectuer le minage de la block-chain, c'est à dire à partir des informations du bloc actuel, du hash du bloc précédent et d'un nonce nous devons générer le hash du bloc current.

En input nous recevons:

- l'objet bloc qu'on veut ajouter dans la bloc chaîne,
- La difficulté (nombre de 0)

En output nous devons fournir :

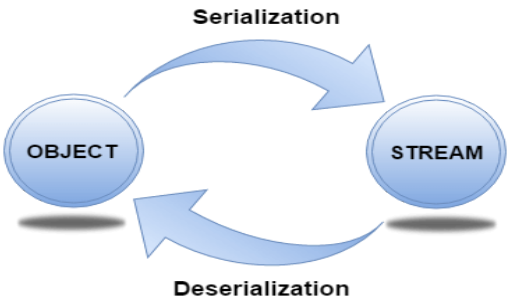
- Le hash qui respect la difficulté.

## 1 - Interface

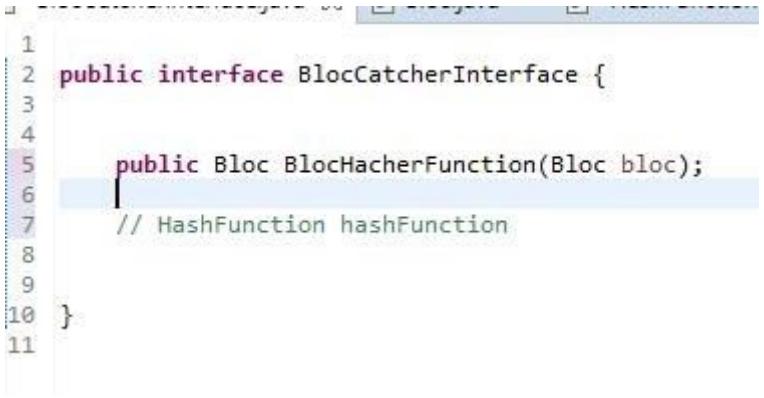
L'interface donnera accès à la fonction permettant de prendre en paramètre un bloc sans hash (et fournira en sortie un bloc avec un hash correct).

On s'est mis d'accord avec le groupe 4 afin d'assurer la bonne communication entre notre partie et la leur.

Effectivement notre interface est censée recevoir un objet BLOC qui sera défini par le groupe 4 (composant 4). Ensuite via la librairies (dlopen et dlsys) l'objet est transmis depuis le groupe 4 vers notre groupe en forme sérialisée.



Dès qu'on reçoit l'objet on le désérialise et on le passe à notre fonction de base qui s'occupera de traiter ce dernier.



```
1
2  public interface BlocCatcherInterface {
3
4
5      public Bloc BlocHacherFunction(Bloc bloc);
6
7      // HashFunction hashFunction
8
9
10 }
11
```

NB:  le transfert de l'objet sérialisé du composant 4 vers le composant 3 est détaillé dans la partie "Communication"

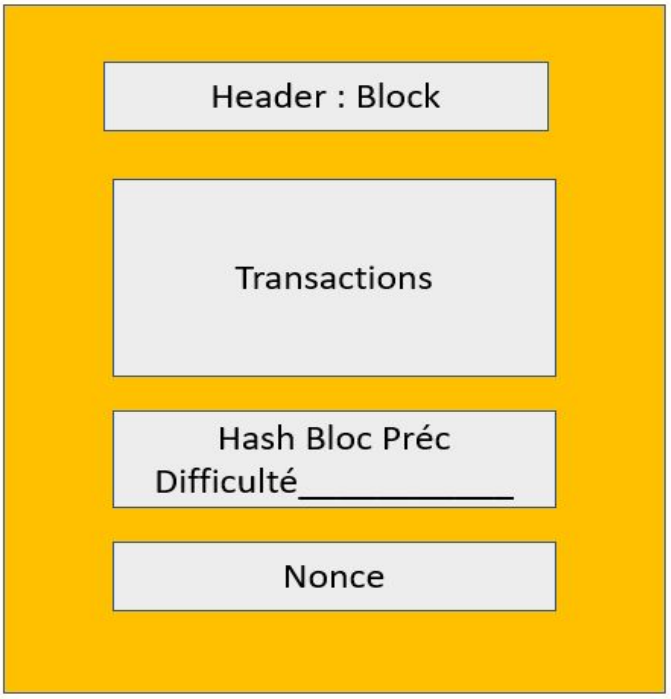
## 2 - Réception de l'Objet Bloc (Input)

Dans le cadre du composant que nous allons développer, nous allons avoir besoin en entrée d'un bloc de la chaîne.

Un bloc contient :

- La difficulté
- Le nonce
- Les transactions
- Le hash du bloc précédent

Dans notre cas, nous recevrons la difficulté et la transaction, et nous produirons le Hash du bloc grâce à un hachage SHA-256 ainsi que le nonce. Nous précisons que dans notre cas, le bloc reçu dont nous calculons le Hash est le dernier bloc à rajouter à la chaîne, qu'on nomme le "Target Bloc".




## 3 - Analyse de la difficulté

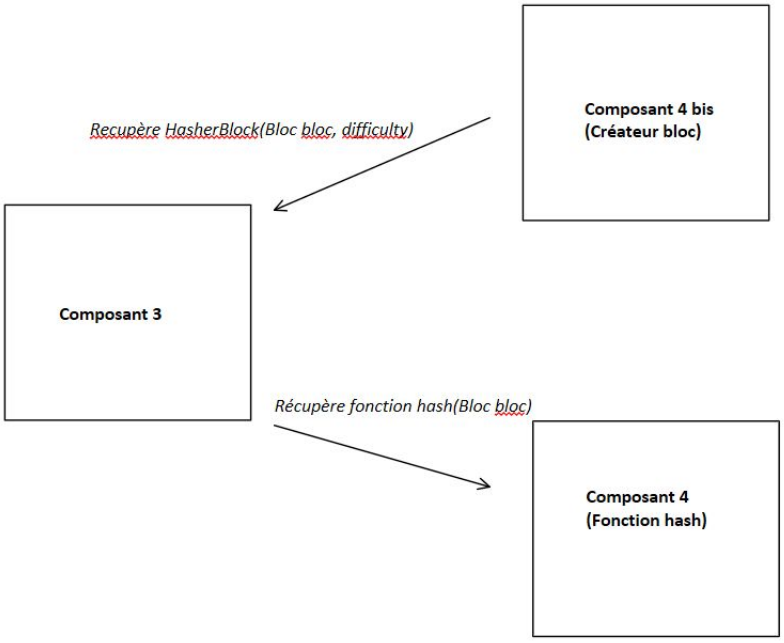
Développons ce qu'est la difficulté. Elle permet de rendre plus difficile le minage d'un bloc. Le niveau de difficulté représente un nombre de zéros obligatoires en début de hachage. Dans le cas du bitcoin par exemple, la difficulté évolue en fonction du nombre de mineurs afin de ramener le temps moyen de minage à 10 minutes, donc, plus il y a de mineur, plus la difficulté augmente, et inversement, la difficulté diminue si le nombre de mineurs diminue.

## 4 - Appel de la fonction SHA256

Cette fonction sera gérée par un autre groupe responsable de produire les interfaces nécessaires pour le hachage.
La fonction de hachage *SHA-256* est souvent utilisée dans tout ce qui concerne la génération de Bitcoin.C'est une des fonctions mathématiques qui permettent de transformer une chaîne de caractères de longueur indifférente en une autre chaîne de longueur fixe (256 bits, soit 64 caractères en notation hexadécimale complète pour *SHA-256*). Caractéristique : le changement le plus insignifiant dans la chaîne d'entrée provoque un grand changement dans la chaîne de sortie.


1. **Communication**

   Afin de pouvoir appeler la fonction hash du composant 4, nous devons importer dans un premier temps la librairie *“composant4.so”*, puis importer la fonction *“hash(bloc)”* et enfin nous devons l'exécuter sur le *bloc* entré en paramètre.




   L’importation des composants.so se fera par l'intermédiaire des dlopen et l’importation de la fonction par dlsys. En effet, dlopen et dlsym permettent de charger dynamiquement une librairie partageable. Donc, nous chargerons la fonction *“hash(Bloc bloc)”* du composant 4 et la fonction ”verification_bloc(Bloc bloc)” du composant 5.

   A l’issue de l'exécution la fonction renverra un bloc disposant d’un hash ayant vérifié la difficulté.

## 5 - Boucle de Validation

Cette boucle itérative permet de vérifier si le hash, que nous obtenons pour ce bloc, respecte les contraintes de difficulté.

Dans un premier temps nous devons initialiser le nonce et appliquer à la fonction hash la concaténation des données suivantes :

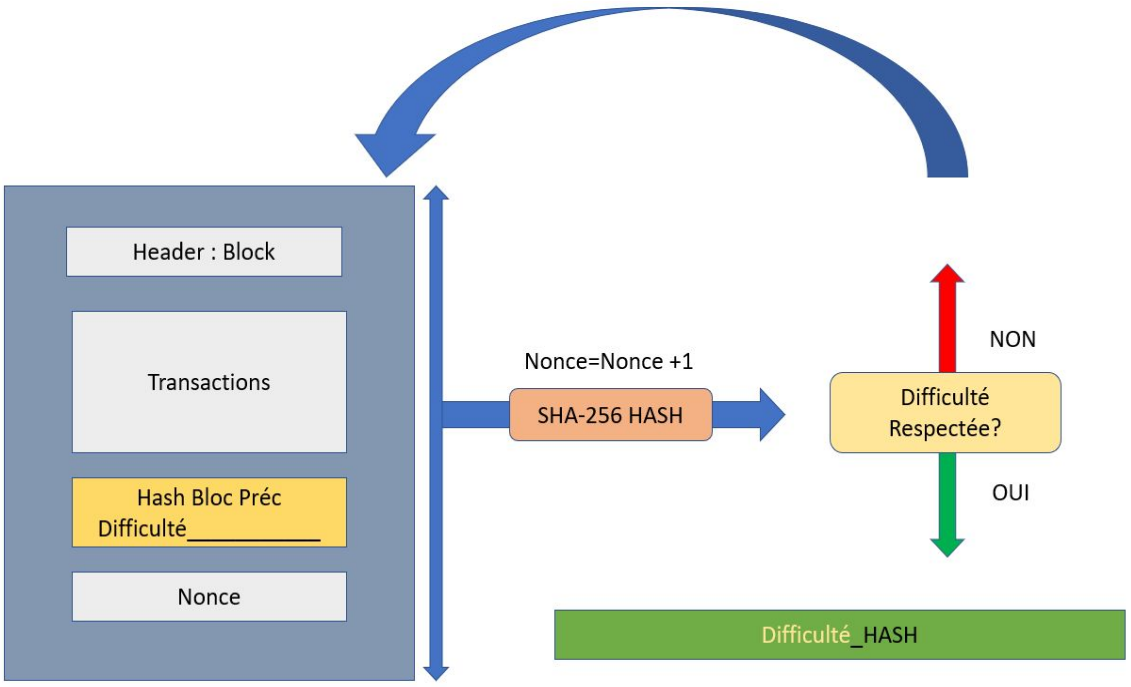
- Id du bloc,
- Transactions,
- Hash du bloc précédent,
- Nonce (le nonce est initialisé à zéro en début de boucle)

2. **Cas d'erreurs et incrémentation du nonce.**

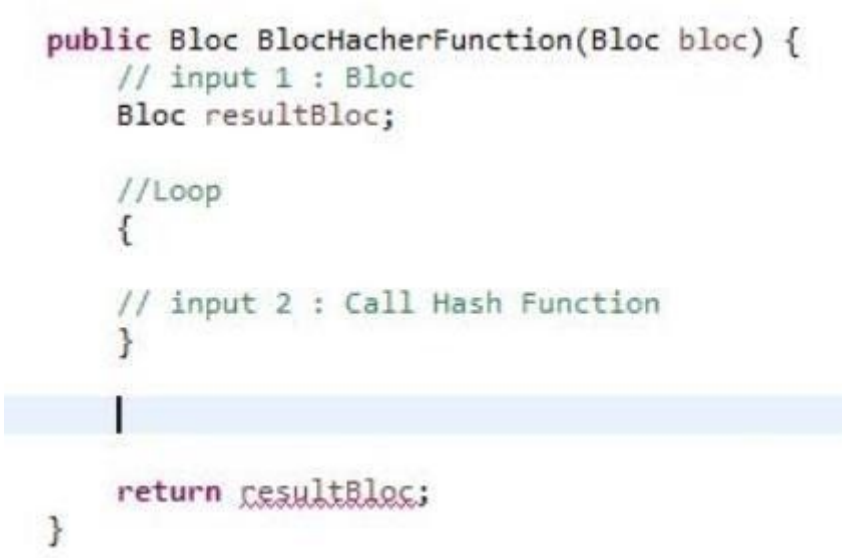
   Il se peut que le résultat du hash pour ces données ne soit pas conforme à la contrainte de difficulté. En effet, si celui ci ne respect pas le nombre de zéro en début de hash pour ce résultat alors le bloc n'est pas validé. Dans ce cas, le nonce est incrémenté et la boucle est réitérée avec le nouveau nonce.

3. **Cas de validation**

   Si le hash renvoyé par la fonction respecte la contrainte de difficulté, c'est à dire que le nombre de 0 en début de hash correspond à la difficulté alors le hash pour ce bloc est validé. La composante fournira alors le hash qui a été calculé par le mineur.



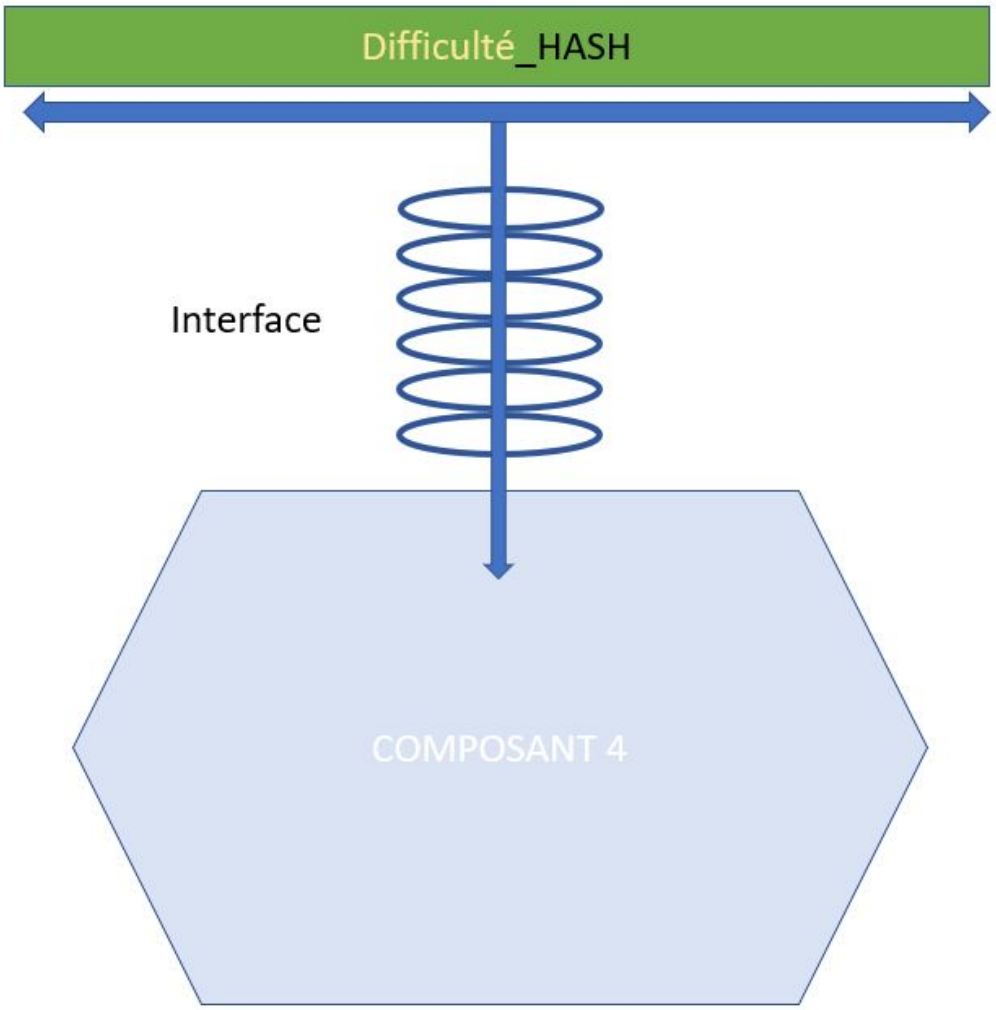

4. **Squelette de la boucle de validation**




## 6 - La sortie (Output)

En sortie de notre bloc nous allons trouver le hash du flot de données arrivés en entrée, c'est le groupe 2 qui nous fournira le flot de données. La fonction de hachage ne sera pas réalisée dans le bloc 3, nous utiliserons le hachage SHA-265 mais nous aurions pu utiliser d'autres fonctions de hachage. Notre Hash aura pour résultat d'être une empreinte numérique servant à identifier rapidement la donnée initiale, au même titre qu'une signature pour identifier une personne. La partie signature sera une part importante du composant 6 réalisé et documenté par l'équipe en charge de cette partie.

Le hash est une part importante de la sécurisation des données dans le monde du numérique. Avec le hash nous garantissons l'intégrité des données et la répudiation, sans les autres éléments les informations seront indéchiffrables il faudrait une puissance de calcul inimaginable pour essayer de craquer une fonction dites de hachage.




(olivier)

Comment communiquer avec les autres composants, Git commun, même langage de développement, voir avec les autres groupes concernant l'appel et l'envoi des fonctions

## 7 - Tests Unitaires

Le test unitaire permet au développeur de tester un module indépendamment du reste du programme, afin de valider qu'il répond au spécifications fonctionnelles attendues.
Dans notre cas, nous attendons de notre composant qu'il puisse :
- Etre appelé par le composant 4 et récupérer le bloc envoyé ainsi que la difficulté
- Appeler la fonction de hachage afin de hacher le bloc et incrémenter le nonce
- Retourner le bloc avec la bonne valeur de nonce

Dans le cadre de ces attentes, nous devons donc définir les tests suivants:
- Vérifier que notre interface est utilisable et notre composant joignable avec les bons paramètres.
- Vérifier que nous recevons bien le hash du bloc en retour du composant 4

- Vérifier que le nonce est bien incrémenté dans le bloc avant le return.

En règle générale, les tests unitaires se font suivant les étapes suivantes :
- L'initialisation qui consiste à créer un environnement de tests
- L'exercice qui est l'exécution du module à tester
- La vérification qui consiste à comparer les résultats avec un vecteur de résultats prédéfinis
- La désactivation, soit la désinstallation de l'environnement de tests.

Dans notre cas nous pensons procéder comme suit :
- Définir un appel à notre fonction en fournissant un vecteurs de blocs ainsi que des difficultés
- Exécuter notre fonction sur ces différents paramètres
- Vérifier le résultat:
    -En vérifiant que pour chaque bloc, le hashage respecte la difficulté
    -En vérifiant le nonce de chaque bloc grâce à une fonction Verification() fournie par le composant 5