amU Faculté des sciences Aix Marseille Université	Domaine Sciences et Technologies Master informatique **Génie logiciel : TP 2** **Codes UE : SINA09EL**
Année 2025-26	**Tests : Objets factices**

1 Introduction

Dans ce TP, nous allons tenter de développer et de tester une application de login sécurisé. Le rendu est demandé sur le dépôt git dédié au TP de fiabilité logicielle pour le dimanche 1er février au plus tard.

L'architecture logicielle que nous allons mettre en place est la suivante. Le login sera géré par une classe implémentant l'interface `LoginService` décrivant un service de login. La méthode associée à cette interface sera `Decision login(String id, String authData1, String authData2)` : méthode prenant en paramètre une identité `identity` et deux chaînes correspondant à des données d'identification et qui retourne une valeur du type `Decision`, qui est un type `enum` possédant deux valeurs : `GRANT` et `DENY`. La valeur `GRANT` signifie que la demande de connexion est acceptée tandis que `DENY` signifie qu'elle est refusée. Toute classe implémentant cette interface aura un constructeur prenant comme arguments deux objets de type `AuthenticationService`.

L'interface `AuthenticationService` possède deux méthodes :

— la méthode `boolean isAMatch(String identity, String authenticationData)` et
— la méthode `Strength dataStrength(String authenticationData)`.

Toute classe implémentant l'interface `AuthenticationService` possédera un constructeur prenant comme argument un objet de type `Directory` associant aux identités leurs données d'authentification. Selon le type de service d'authentification, le constructeur prendra également soit :

— un objet de type `Strength` donnant la "force" de toutes les données d'authentification du service
— un objet de type `EquivalentBitLengthEstimator` ainsi qu'un objet `AlphabetSizeEstimator` décrivant deux services dont le rôle est de mesurer l'entropie d'une donnée d'authentification.

La méthode `isAMatch` retourne vrai si l'identité et la donnée d'authentification correspondent au regard d'un objet de type `Directory`. La méthode `dataStrength(String authenticationData)` mesure la "force" de la donnée d'authentification et retourne une valeur pour le type énuméré `Strength` prise parmi les valeurs `VERY_WEAK`, `WEAK`, `REGULAR`, `STRONG`. Pour un service d'authentification biométrique, la "force" est tout simplement un attribut du service (initialisé à la construction de celui-ci) et ne dépend pas de la donnée alors qu'un service d'authentification par mot de passe utilisera un objet de type `EquivalentBitLengthEstimator` ainsi qu'un objet de type `AlphabetSizeEstimator` pour mesurer la force d'une donnée. La méthode lévera une exception de type `IllegalArgumentException` si la donnée d'authentification est `null` ou vide (chaine de longueur 0) ou bien si l'identité est `null` ou vide.

L'interface `Directory` ne possède qu'une seule méthode `String getMatch(String identity)` qui retourne pour une identité une donnée d'authentification. On peut imaginer par exemple que cette méthode utilise une base de données associant à chaque identité une chaîne de données d'identification. La méthode retourne `null` si l'identité n'existe pas dans la base de données.

Pour revenir à la méthode `dataStrength` on peut associer la force aux données de la manière suivante :

— `VERY_WEAK` à une donnée équivalente à un nombre de bits compris entre 0 et 59 ;
— `WEAK` à une donnée équivalente à un nombre de bits compris entre 60 et 89 ;
— `REGULAR` à une donnée équivalente à un nombre de bits compris entre 90 et 119 ;
— `STRONG` à une donnée équivalente à un nombre de bits supérieure ou égale à 120.

Pour mesurer l'équivalence en nombre de bits d'une donnée d'authentification, on utilisera la méthode `int equivalentBitLength(int alphabetSize, int dataSize)` de l'interface `EquivalentBitLengthEstimator` où `alphabetSize` est la taille de l'alphabet utilisé pour écrire la donnée et `dataSize` la longueur de cette donnée. Ainsi, pour un mot de passe de longueur 8 utilisant un alphabet de taille 94, on aura `equivalentBitLength(94, 8)`. Pour mesurer la taille de l'alphabet, on utilisera la méthode `int estimateAlphabetSize(String authenticationData)` de l'interface `AlphabetSizeEstimator`.

Le projet à compléter se trouve dans le répertoire TP 2 du git dédié au TP de fiabilité logicielle. Comme pour le précédent TP, il faut mettre à jour votre dépôt à partir de l'interface web de celui-ci. Les classes et interfaces principales (pas dédiées aux tests) devront être placées dans le répertoire `src/main/java` du répertoire `TP2` alors que les classes dédiées aux tests (donc les classes pour les objets fictifs créées spécifiquement pour les tests) seront à placer dans le répertoire `src/test/java`. Le lancement des tests peut se faire directement avec IntelliJ en ajoutant si nécessaire la configuration gradle du répertoire `TP2`. On peut faire cet ajout avec un clic droit sur le fichier `build.gradle` du répertoire `TP2` puis en cliquant sur `link gradle project` dans le menu venant de s'ouvrir.

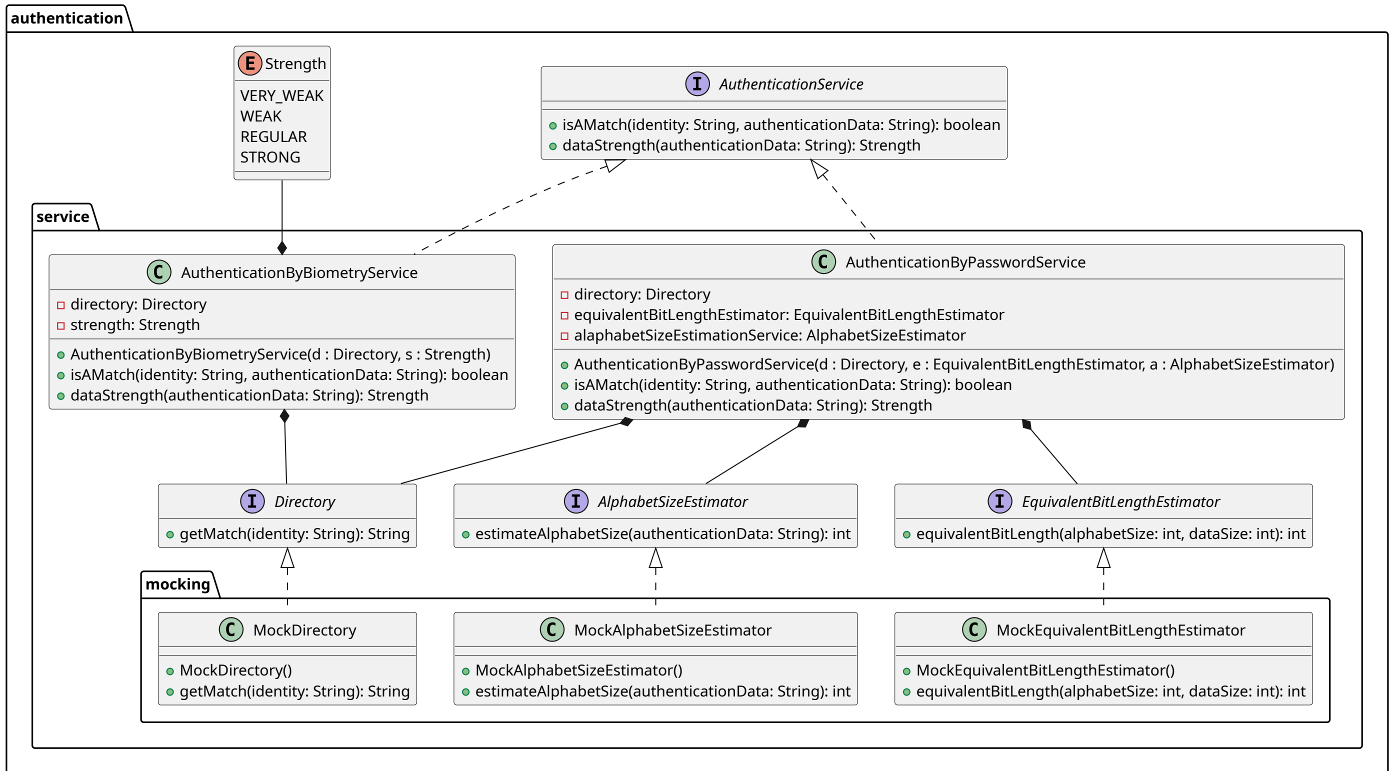
2 Implémentation d'une classe de `AuthenticationService`

On souhaiterait que l'interface `AuthenticationService` soit implémentée par les deux classes `AuthenticationByPasswordService` et `AuthenticationByBiometryService`. Dans le premier cas, la chaîne `authenticationData` sera un mot de passe tandis que le second on peut imaginer que cela soit l'encodage Base64 d'une information biométrique (empreinte digitale, empreinte oculaire, scan 3D d'un visage, ...).

2.1 Création manuelle d'objets factices pour les tests

Cependant, nous ne disposons pas à l'heure actuelle des bases de données et donc, d'une implémentation complète de l'interface `Directory`, ni de l'accès à un *web service* pour calculer l'équivalent en nombre de bits d'une donnée d'authentification et donc d'une implémentation complète de l'interface `EquivalentBitLengthEstimator`.

Afin de pallier ce problème et pour réaliser et tester les implémentations de `AuthenticationByPasswordService` et `AuthenticationByBiometryService`, nous allons créer des objets factices se substituant aux manques actuels.



1. Rajouter le code de l'énumération `Strength` et de l'interface `AuthenticationService` dans le package `authentication`.
2. Créer des interfaces `Directory`, `AlphabetSizeEstimator` et `EquivalentBitLengthEstimator` dans le package `authentication.service`.
3. Définir une interface `AuthenticationService` et ses sous-classes `AuthenticationByPasswordService` et `AuthenticationByBiometryService` en notant que leur constructeur utilise les interfaces `Directory`, `AlphabetSizeEstimator` et `EquivalentBitLengthEstimator`.
4. Donner des cas de tests pour les méthodes `isAMatch` et `dataStrength` (Vous expliciterez ce que vous testez dans chacun des cas de test).
5. Nous allons maintenant créer des classes factices pour remplacer les implémentations absentes : définir des classes `MockDirectory`, `MockAlphabetSizeEstimator` et `MockEquivalentBitLengthEstimator` implémentant les interfaces `Directory`, `AlphabetSizeEstimator` et `EquivalentBitLengthEstimator`. Ces trois classes appartiendront au package `authentication.service.mocking` dans la partie test du dépôt (et donc dans le répertoire `src/test/java/authentication/service/mocking`). Ces trois classes posséderont chacune un constructeur sans argument. Vous devez définir le comportement de leur méthode respective `getMatch`, `estimateAlphabetSize` et `equivalentBitLength` de sorte à permettre aux tests que vous allez écrire de passer correctement.
6. Implémenter vos tests sous la forme de deux classes `AuthenticationByPasswordServiceTest` et `AuthenticationByBiometryServiceTest`. Ces deux classes utiliseront donc des objets des classes `MockDirectory`, `MockAlphabetSizeEstimator` et `MockEquivalentBitLengthEstimator`.
7. Vérifier à l'aide de JaCoCo que vos tests couvrent de manière satisfaisante (au moins 80%) les méthodes des classes `AuthenticationByPasswordService` et `AuthenticationByBiometryService`. Complétez

vos tests si nécessaire.

2.2 Simulation automatique avec Mockito

On peut remarquer que la création des classes manquantes pour simulation n'est réalisée que pour des tests et cela peut être fastidieux à écrire. L'automatisation de leur création serait donc bénéfique. Pour cela, il existe différents outils dont l'outil Mockito.

Vous devrez dans un premier temps lire la documentation et les exemples fournis pour étudier comme utiliser Mockito. "In a nutshell", l'outil construit les (méthodes des) objets factices à l'exécution et il n'est pas nécessaire de définir les classes factices, mais la classe de tests utilisera des objets factices créés avec *mock*. Pour le test, il suffit de définir le comportement des méthodes des objets factices. Par exemple, on peut spécifier qu'une méthode est appelée avec certains arguments, renvoie une certaine valeur, est appelée n fois, ... Les tests comportent 3 caractéristiques :

— Création d'objets factices avec `Type mockObject = mock(Type.class)` pour créer un objet factice de type `Type`,
— Spécification (`when(mockObject.method(arg1, arg2)).thenReturn(value)` qui décrit les appels factices attendus en spécifiant la valeur `value` retournée en fonction des arguments.
— Exécution des méthodes testées et vérification des assertions JUnit usuelles pour vérifier que les valeurs attendues sont celles calculées,
— Vérification `verify(mockObject.method(arg1, arg2), times(n))` qui vérifie que l'exécution a appellé la méthode `method` de l'objet factice `mockObject` avec les arguments `arg1` et `arg2` exactement `n` fois. Si les spécifications ou les vérifications ne sont pas satisfaites lors de l'exécution, le test échoue de même qu'il échoue si les assertions JUnit sont satisfaites.
— Reprendre les interfaces `Directory`, `AlphabetSizeEstimator`, `EquivalentBitLengthEstimator` et `AuthenticationService` ainsi que les classes `AuthenticationByPasswordService` et `AuthenticationByBiometryService` de l'étape précédente.
— Utiliser Mockito pour :

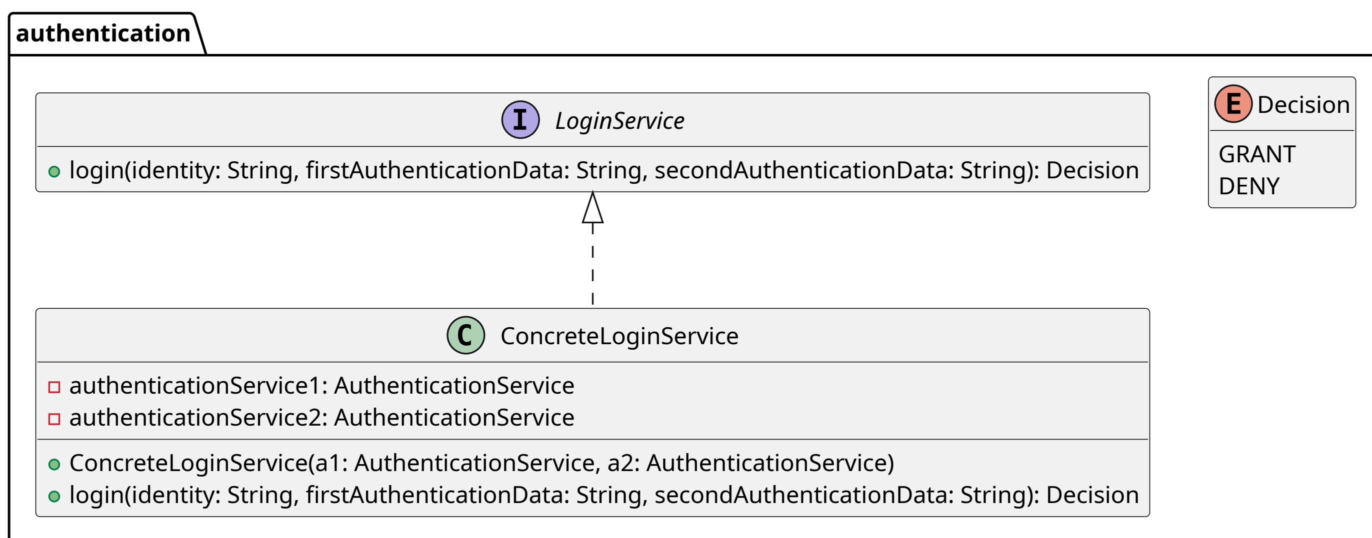
1. Écrire deux nouvelles classes de tests `AuthenticationByPasswordServiceMockitoTest` et `AuthenticationByBiometryServiceMockitoTest` qui réalisent les mêmes tests que les classes précédentes, mais en utilisant Mockito.
2. Raffiner les tests en utilisant toutes les capacités de Mockito (notamment la vérification que des appels à certaines méthodes ont bien été réalisés).
3. On suppose dorénavant que la méthode `getMatch` de l'interface `Directory` lève une exception si son paramètre de type `String` est `null`. Adapter vos tests pour tester la construction de simulation des exceptions de Mockito.

3 Implémentation d'une classe implémentant `LoginService`

On souhaite maintenant coder une classe `ConcreteLoginService` qui implémentera l'interface `LoginService`. Cette classe aura un constructeur de la forme `ConcreteLoginService(AuthenticationService`

firstAuthenticationService, AuthenticationService secondAuthenticationService). On souhaite réaliser des tests pour s'assurer que cette implémentation est correcte. La spécification de la méthode Decision login(String identity, String firstAuthenticationData, String secondAuthenticationData) de cette classe est la suivante :

— dans le cas où la donnée firstAuthenticationData est forte, si cette donnée correspond à l'identité alors la méthode retourne GRANT dans le cas contraire, la méthode retourne DENY.
— dans le cas où la donnée firstAuthenticationData est très faible ou faible alors la méthode retourne DENY
— dans le cas où la donnée firstAuthenticationData est normale ("regular") et secondAuthenticationData n'est ni faible, ni très faible, si les deux données correspondent à l'identité alors la méthode retourne GRANT dans le cas contraire, la méthode retourne DENY.



1. Rajouter le code de l'interface LoginService, de l'énumération Decision et de la classe ConcreteLoginService. Toutes ces classes et interfaces appartiennent au package authentication.
2. Produire des cas de tests pour la méthode login de ConcreteLoginService.
3. Réaliser l'implémentation de la classe de tests ConcreteLoginServiceTest pour la classe ConcreteLoginService qui utilisera Mockito pour simuler les dépendances.
4. Vérifier à l'aide de JaCoCo que vos tests couvrent de manière satisfaisante le code de la classe ConcreteLoginService. Complétez vos tests si nécessaire.

4 Test d'intégration

On va finir le TP sur un test d'intégration. On souhaite tester l'intégration de la classe ConcreteLoginService avec les classes AuthenticationByPasswordService et AuthenticationByBiometryService. Pour cela, on va créer une classe de test ConcreteLoginServiceIntegrationTest dans laquelle on va créer un objet ConcreteLoginService utilisant une instance de AuthenticationByPasswordService et une instance de AuthenticationByBiometryService.

On utilisera pour cela des objets concrets (non-mockés) des classes Directory, EquivalentBitLengthEstimator

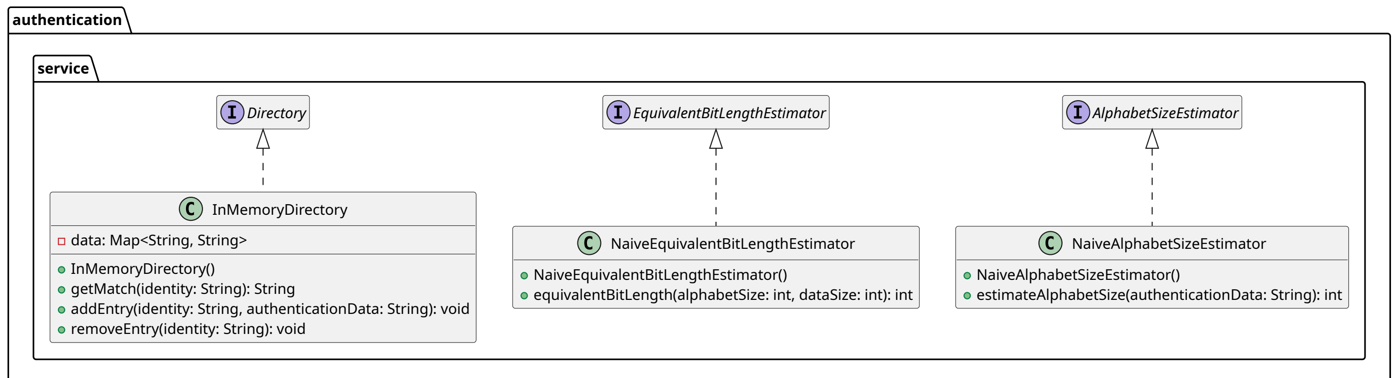
et AlphabetSizeEstimator en utilisant les spécifications suivantes :

— L'implémentation de l'interface EquivalentBitLengthEstimator retournera pour une taille d'alphabet A et une longueur de donnée L la valeur entière correspondant à $\lfloor L \times \log_2(A) \rfloor$ (arrondi inférieur de $L \times \log_2(A)$).
— L'implémentation de l'interface AlphabetSizeEstimator retournera pour une chaîne de caractères la taille de l'alphabet suivant les règles suivantes. Pour la taille de l'alphabet, on supposera que le mot de passe est écrit avec les caractères suivants :

  — lettres majuscules A à Z : 26 symboles
  — lettres minuscules a à z : 26 symboles
  — chiffres 0 à 9 : 10 symboles
  — caractères spéciaux ~ ! @ # $ % ^ & * ( ) - _ = + [ ] { } \ | ; : ' " , < > / ? : 32 symboles

  On considérera qu'un mot de passe utilisant un caractère de chaque catégorie utilise un alphabet de taille 94 (26 + 26 + 10 + 32). Si une catégorie n'est pas utilisée, on soustraira sa taille de 94 pour obtenir la taille de l'alphabet. Par exemple, un mot de passe utilisant uniquement des lettres minuscules et des chiffres utilisera un alphabet de taille 36 (94 - 26 - 32, car il ne contient pas de lettres majuscules ni de caractères spéciaux).
— L'implémentation de l'interface Directory utilisera une table de hachage (HashMap) pour associer des identités à leurs données d'authentification. On rajoutera des méthodes pour ajouter ou supprimer des associations identité/donnée d'authentification dans cette table de hachage.

Le diagramme suivant résume l'architecture logicielle demandée (pas besoin d'implémenter les classes qui le sont déjà).



1. Implémenter les classes InMemoryDirectory, NaiveEquivalentBitLengthEstimator et NaiveAlphabetSizeEstimator selon les spécifications données ci-dessus. Ces classes appartiendront au package authentication.service.
2. Implémenter la classe de test ConcreteLoginServiceIntegrationTest qui testera l'intégration de la classe ConcreteLoginService avec les classes AuthenticationByPasswordService et AuthenticationByBiometryService en utilisant les classes InMemoryDirectory, NaiveEquivalentBitLengthEstimator et NaiveAlphabetSizeEstimator.

5 Rendu

Le rendu du TP se fera sur le dépôt git dédié au TP de fiabilité logicielle. On veillera à bien organiser les fichiers sources et de tests dans les répertoires `src/main/java` et `src/test/java` du répertoire TP2 du dépôt. Le rendu est demandé pour le dimanche 1er février au plus tard. Il vous faudra également fournir un compte rendu de TP (format PDF) expliquant votre démarche, les tests que vous avez réalisés ainsi que les résultats obtenus (taux de couverture des tests, difficultés rencontrées, ...).