LOG430 – Architecture logicielle

# Equipe 01

# Scénarios de qualité, tactiques et justifications des attributs produits par les équipes en classe

# < Testabilité > – Justification (en classe)

Afin de tester des changements dans la couche de traitement des données ont effectue les tests sur une base de données donc on contrôle l’état. On compare ensuite l’état final de la base de données avec une autre base de données qui a été validé auparavant et donc on s’assure de l’exactitude des données afin de s’assurer que les changements dans la couche de traitement des données donnent les mêmes résultats.

# <Testabilité> – Scénario 1 (en classe)

**Source**: Acceptance testers automatisé

**Stimuli**: La mise en production d’une nouvelle version du guichet

**Artéfact**: Couche de traitement des données

**Environnement**: Durant l’intégration

**Réponse**: Exécuter une suite de test et capturer les résultats

**Mesure de la réponse**: Faible effort pour exécuter des tests sur une nouvelle version peu importe les modifications qui ont été appliqué. Très forte probabilité de trouvé les erreurs

# <Testabilité> – Tactique 1 (en classe)

**Description**: Abstract data source

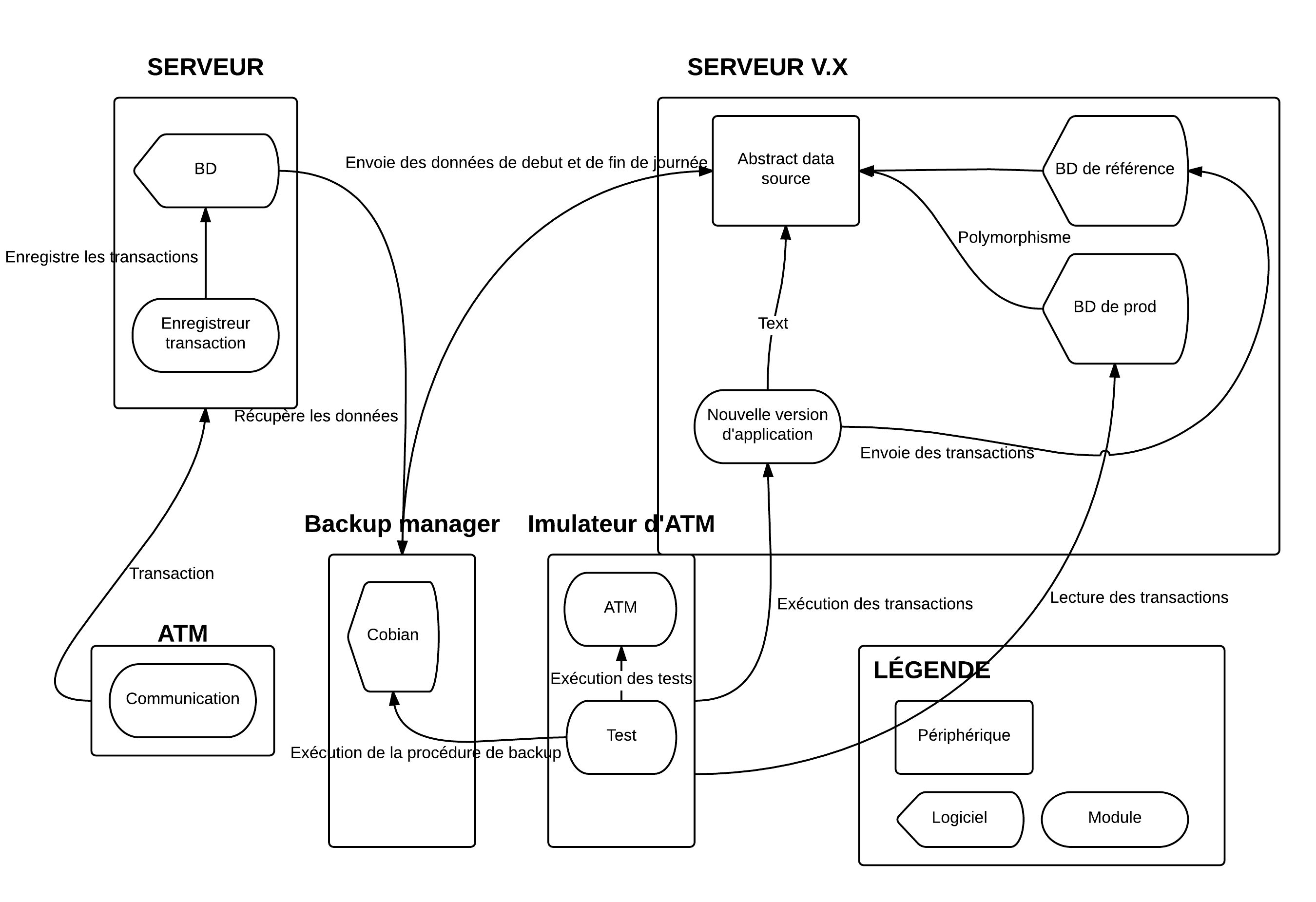
**Justification**: On utilise une base de données dont on connait l’était initial et ont peu modifié la configuration de l’application pour utiliser une base de donnée au choix

# <Testabilité> – Tactique 2 (en classe)

**Description**: Record play back

**Justification**: On peut conserver une trace des transactions qui sont valide et les utilisés pour s’assurer qu’avec une nouvelle version de l’application les données sont traités comme dans la version précédente de l’application

# <Testabilité> – Vue Architecturale (en classe)



* Texte de description du diagramme
* Table de description des éléments du diagramme
* Texte décrivant la relation entre les éléments et la tactique
* texte décrivant comment la tactique répond a chacun des scénarios de qualité.

# Description

L’ATM fait une transaction qui est enregistré dans la base de données de production. Un outil de backup récupère périodiquement un backup de la base de données de production et les données sur une base de données de référence et les transactions sur une autre base de données. Un outil de test se charge de récupérer les transactions sur la base de données qui contient les transactions et lance l’exécution d’un simulateur d’ATM qui va simuler chacune des transactions sur la nouvelle version de l’application. L’application va enregistrer le résultat dans la base de données de référence. On pourra par la suite faire une comparaison entre les données pour s’assurer que la nouvelle version de l’application génère le même résultat que la version antérieur.

# Tactique 1

L’abstract data source dans le diagramme permet de faire une transition entre deux base de données pour faciliter les tests et la procédure de backup. Correspond à la tactique « Abstract data source »

# Tactique 2

Le module d’enregistrement des transactions correspond à la tactique « Record play back » on conserve une trace de chaque transaction pour pouvoir par la suite exécuter les transactions sur une application de test et s’assurer que la nouvelle application produit les mêmes résultats que la version de production.

# Table de description des éléments du diagramme

|  |  |
| --- | --- |
| BD | La base de données avec les données en production |
| Enregistreur de transaction | Gère les transactions reçues de l’ATM |
| Communication | Module de communication qui communique avec le serveur pour enregistrer les transactions |
| Backup manager – Cobian | Serveur qui contient l’outil qui se charge de la procédure de backup |
| Emulateur d’ATM | Contient le module du simulateur de l’ATM permettant de simuler les tests sur la nouvelle version de l’application et le module de test qui se charge de faire les tests de façon automatisé. |
| Nouvelle version de l’application | La nouvelle version de l’application à testé |
| Abstract data source | Configuration qui permet de faire la transition entre deux bases de données |
| Serveur V.x | Serveur avec l’architecture de test |
| Bd de référence | Base de données sur laquelle seront exécutés les tests |
| Bd de prod | Base de données avec les données valide qui sera utilisé pour la comparaison de données |

# Equipe 02

LOG430 – Architecture logicielle

Scénarios de qualité, tactiques et justifications des attributs produits par les équipes en classe

# <Testabilité> – Justification

Pour éviter de perdre du temps à faire des tests manuels et pour augmenter leur confiance en leur algorithme de détection de fraude, les développeurs de la banque souhaitent tester l’algorithme qui vérifie la géolocalisation et l’horodatage lors des retraits des clients.

# <Testabilité> – Scénario

**Source :** Développeurs, testeurs unitaires

**Stimuli :** Implémentation de nouvelles fonctionnalités du Proxy de géolocalisation

**Environnement :** Développement

**Artéfact :** Algorithme de détection de fraude qui dépendent du délai entre deux transactions et la localisation.

**Réponse :** Les tests sont exécutés. et les résultats sont capturés

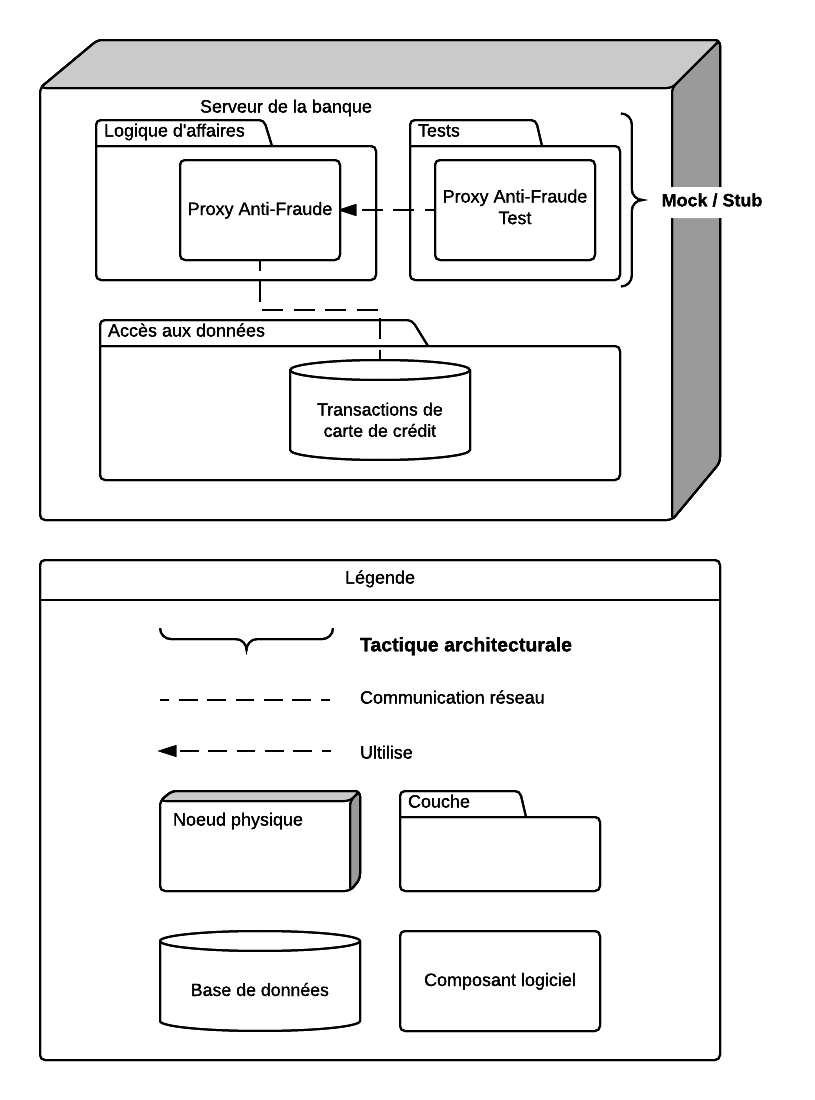
**Mesure de la réponse** : Nombre de secondes pour exécuter les tests unitaires? Effort pour trouver la faute.

# <Testabilité> – Tactique

**Description :** *[Mock / Stub]* – On utilise des tactiques, comme du couplage faible et de la haute cohésion, visant à rendre la classe testable de manière unitaire. Des fausses données sont envoyées pour rendre les tests prévisibles.

**Justification :** Pour pouvoir tester le comportement de l'algorithme dans des conditions différents, on doit faire des mocks de transactions ayant été effectuées à des moment et à des endroits précis.

# <Testabilité> – Vue architecturale



La vue architecturale montre que des tests unitaires sont implémentés pour le proxy anti-fraude. Ces tests permettent de tester en utilisant la technique de test *mock / stub*. Dans les tests, la couche d’accès aux données est mockée pour permettre d’avoir des résultats de tests déterministes avec des données prédéfinies.

## Éléments du diagramme

|  |  |
| --- | --- |
| **Élément** | **Description** |
| Système bancaire | Le noeud physique correspondant de manière large au système bancaire qui traite les transactions demandées dans les guichets. |
| Système bancaire / Logique d’affaires | La couche logicielle contenant toute la logique d’affaires des systèmes bancaires. |
| Système bancaire / Logique d’affaires  Proxy Anti-fraude | Un proxy qui va vérifier le lieu et l’heure aux quelles ont été faites les dernières transactions reliées à une carte de crédit (utilisation d’une carte de crédit dans un guichet de succursale). S’il n’est pas possible qu’un client ait pu faire deux transactions dans des lieux différents avec l’écart de temps calculé, la banque va investiguer le cas pour vérifier qu’il ne s’agisse pas d’une tentative de piratage de carte de crédit. |
| Système bancaire / Accès aux données | La couche logicielle qui permet d’interagir avec les basses de données de la banque. |
| Système bancaire / Accès aux données  Transactions de carte de crédit | La base de données où est enregistrée chaque transaction. |
| Système bancaire / Tests | La couche logicielle contenant toutes les classes de tests. |
| Système bancaire / Tests Proxy Anti-Fraude Test | La classe contenant les tests unitaires du proxy anti-fraude à l’aide de mock. |

La couche de test est directement liée à la tactique *mock / stub* utilisée. Les classes de proxy et la base de données sont nécessaires car l’une est testée et l’autre est mockée pour le bien des tests.

La tactique est une bonne solution au problème de devoir tester de manière efficace et précise le module de proxy qui permet de réduire l’impact des fraudes bancaires. On évite les tests manuels ou les tests qui requièrent beaucoup de ressources (environnement de test / sandbox) en utilisant un test unitaire avec un focus sur l’algorithme de détection.

# Equipe 05

LOG430 – Architecture logicielle

Scénarios de qualité, tactiques et justifications des attributs produits par les équipes en classe

< Testabilité > – Justification (en classe)

Difficulté à tester un GAB sans effectuer de réelles transactions avec des comptes clients.

<Testabilité> – Scénario 1 (en classe)

**Source**: Testeurs du système

**Stimuli**: La délivrance du GAB à la banque.

**Artéfact**: Les communications entre le GAB et les serveurs bancaire

**Environnement**: A la fin de la période d'intégration

**Réponse**: Exécuter les tests sur le système et capturer les résultats pour les comparer avec ceux attendus.

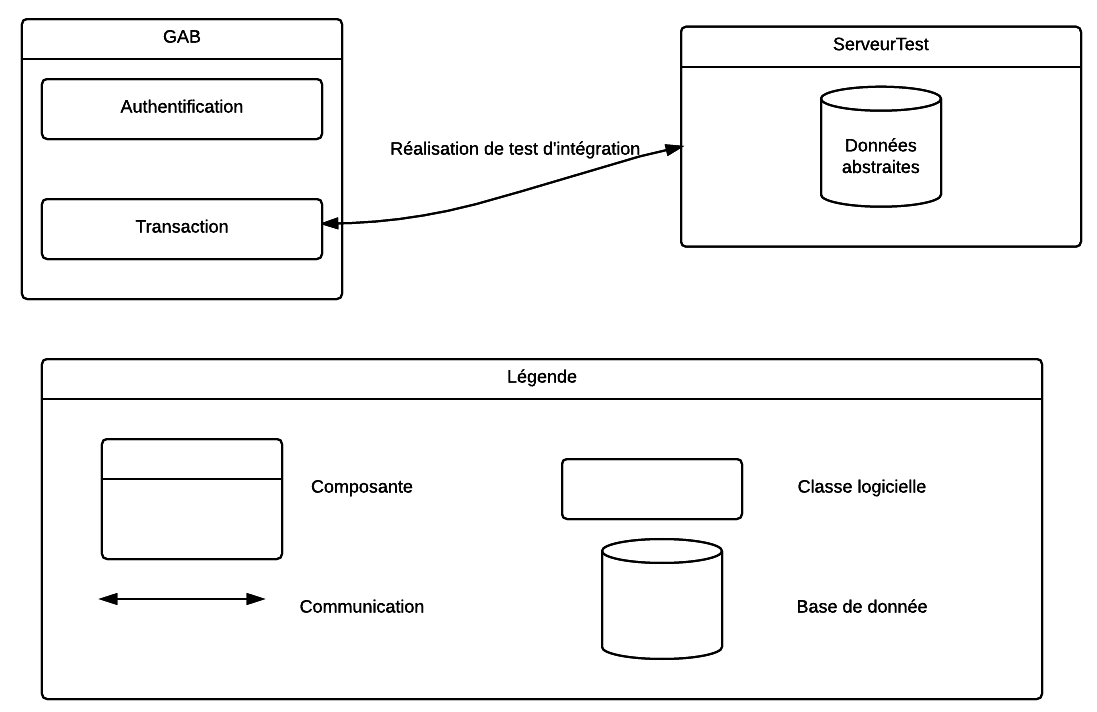
**Mesure de la réponse**: Effort pour trouver l'erreur dans les classes concernées.

<Testabilité> – Tactique 1 (en classe)

**Description**: Source de données abstraite

**Justification**: Création d'un environnement externe au système bancaire avec un contrôle total des données pour prévoir les résultats de tests et ainsi éviter tout risque de contamination des données de production.

<Testabilité> – Vue Architecturale (en classe)



Texte de description du diagramme

Le diagramme précédent représente l’interaction qu’il y a entre un guichet automatique bancaire et un serveur de test. Le serveur contient des données fictives que les testeurs connaissent. Ils peuvent alors prévoir le résultat de l'action du GAB lors des phases de test d'intégration.

Table de description des éléments du diagramme

* **GAB**: Guichet automatique bancaire que le client utilise pour effectuer une transaction.
* **ServeurTest**: Serveur de test pour simuler les comptes clients
* **Authentification**: Classe logicielle récolte les informations du client
* **Transaction**: Classe logicielle qui effectue une transaction bancaire
* **Donnée Abstraites**: Base de donnée de test.
* **Réalisation de tests d'intégration**: Lien de communication entre le GAB et le serveur de tests..

**Équipe 7**

**< Testabilité > – Justification (en classe)**

On veut un moyen de tester les fonctionnalités du GAB avec des valeurs de tests réalistes sans affecter les comptes clients, pour s’assurer que le comportement normal du GAB respecte l’intégrité des données.

**<Testabilité> – Scénario 1 (en classe)**

**Source**: Intégrateur de tests

**Stimuli**: Scénarios de tests de transactions bancaires (retrait, dépôt, etc.)

**Artéfact**: GAB en mode transaction

**Environnement**: Integration time

**Réponse**: Exécuter la suite de tests et monitorer le résultats

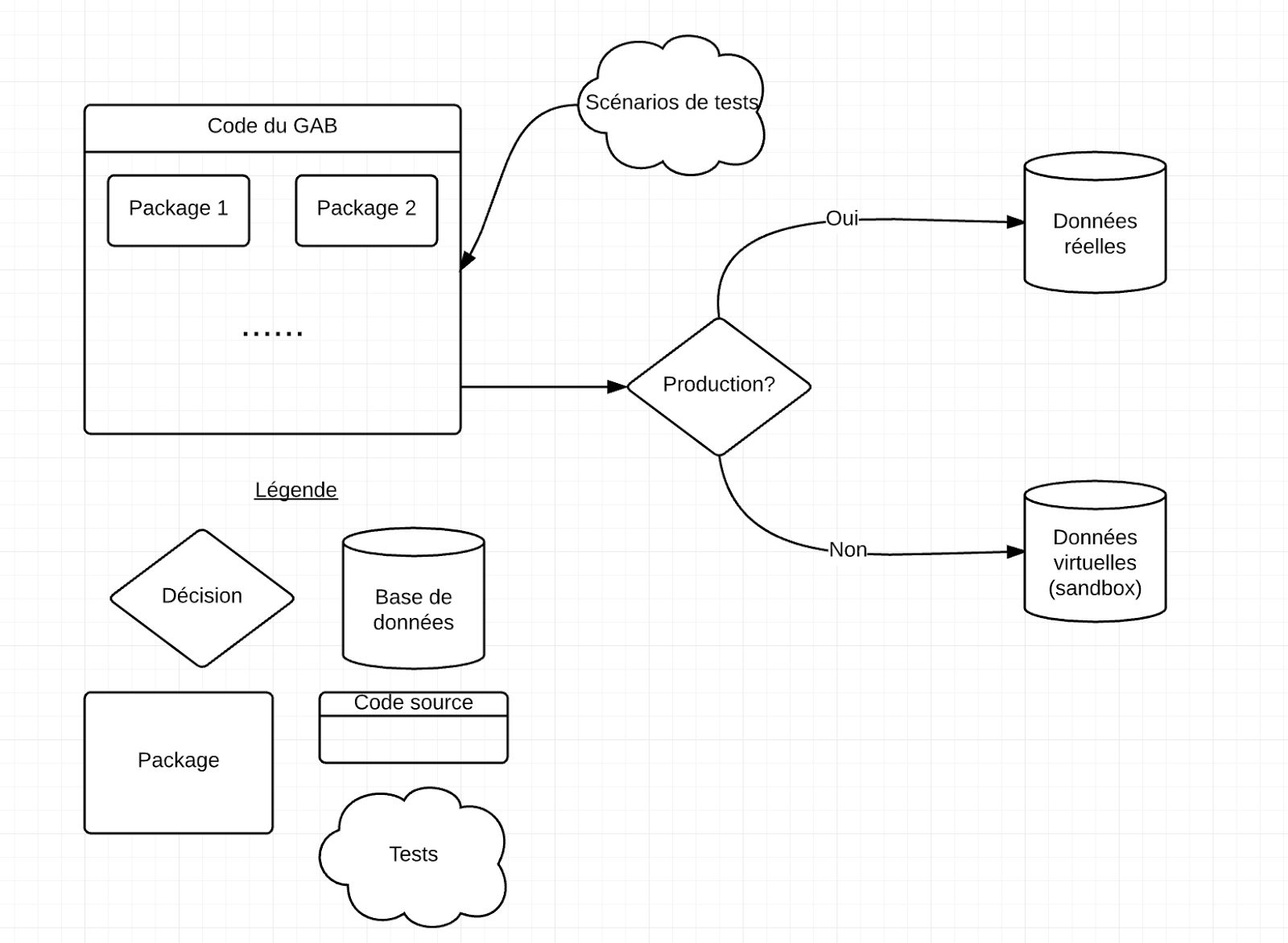
**Mesure de la réponse**: Pourcentage de tests réussis, couverture des fonctionalités de transaction

**<Testabilité> – Tactique 1 (en classe)**

**Description**: *Sandbox*

**Justification**: Avec un environnement de test virtuel, nous voulons tester toutes les fonctionalités de transaction du GAB sans affecter de vrais comptes clients, tout en ayant des valeurs de tests qui s’approche d’une utilisation normale.

**<Testabilité> – Vue Architecturale (en classe)**



Le diagramme décrit la façon dont les tests se dérouleront. Les tests effectuent des actions sur le code du GAB qui communiquera avec le serveur bancaire pour faire les transactions. Si le GAB est en production, il modifiera les données réelles, mais s’il est en test il utilisera les données virtuelles dans un *sandbox*.

**Table de description des éléments du diagramme**

|  |  |
| --- | --- |
| **Élément** | **Description** |
| Code du GAB | Dépôt pour le code source du GAB |
| Scénarios de tests | Batterie de tests pour les transactions bancaires du GAB |
| Package | Modules du code sources du GAB |
| Données réelles | Serveur hébergeant les données de transaction des clients |
| Données virtuelles (sandbox) | Serveur hébergeant une simulation des données de transaction des clients |

Le *sandbox* est le serveur où les fausses données virtuelles sont stockées . C’est le serveur avec lequelle GAB communiquera pendant la réalisation des scénarios de tests.

Les tests peuvent donc se dérouler sans effectuer les données réelles des clients, mais tout en utilisant des données qui se rapprochent fortement des données réelles.