Testabilité

1. Scénario : Testabilité 2

2. Scénario : Un développeur doit valider que l’ajout d’un module de statistique retourne les bonnes données, ne brise pas d’autres fonctionnalités et est exempt de bogue lors de l’utilisation. 5

3. Scénario : Assurance qualité sur le nombre de transactions maximum par minutes du système 8

**4.** Scénario **:** Retracer l’erreur qui a causé un crash 11

5. Scénario : Tester les options d’achats et de ventes d’actions avant de le mettre en production. 14

6. Scénario : Les développeurs doivent pouvoir vérifier si les mesures de déploiement automatisé s’effectuent correctement. 17

7. Scénario : Développement des tests pour assurer la validité des opérations possible sur un titre boursier 21

8. Scénario : Journaliser toute transaction complétée 24

9. Scénario : Exécution des tests de validation pour les transactions. 27

10. Scénario : On veut tester des opérations sur la base de données sans affecter celle en production 31

11. Scénario : Tester la création des transactions 34

12. Scénario 1 Création de tests suite à l’ajout d’une nouvelle fonctionnalité sur le système de Front End Access: 36

# Scénario : Testabilité

|  |  |
| --- | --- |
| **Objectifs d'affaires** | Assurer la couverture de 95% du code de PostTrading |
| **Source** | Les testeurs de l'organisation |
| **Stimulus** | Le code de PostTrading été modifié et doit être testé |
| **Artéfact** | Le code de PostTrading |
| **Environnement** | Temps de développement |
| **Réponse** | Exécuter les tests et capturer les résultats |
| **Mesure de la réponse** | Nombre de tests échoués |
| **Questions** | 1. L'environnement sandbox est accessible a quel client? 2. Le sandbox roule t il sur un serveur dédié ? 3. Est-ce que les bases de données de test sont tous permises ? |

## Tactique 1 : Sandbox

**Description**: Environnement virtuel isolé des autres systèmes.

**Justification**: Afin que nos utilisateurs puissent faire des simulations bancaires sans affecter notre système il est important d’offrir un environnement séparer.

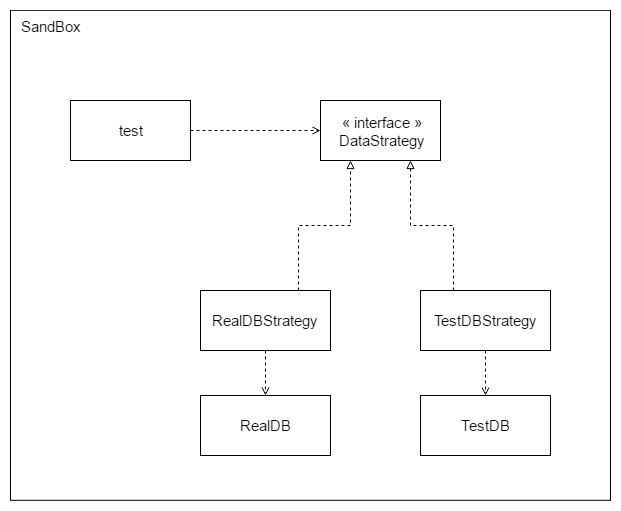
## Tactique 2 : Abstract data sources

**Description**: Capacité de contrôler la source des données du système sans avoir à modifier le code source du logiciel.

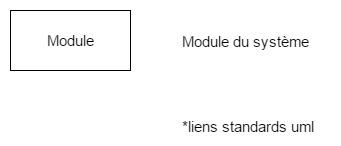
**Justification**: Afin de pouvoir mieux contrôler les cas de test, on utilise une base de données de test.

## Vue architecturale (à l’extérieur de la classe)

### Diagramme



### Légende



### Texte de description du diagramme

Dans ce diagramme nous avons deux modules strategy qui implémente l’interface DataStrategy et se faisant respectent la tactique *Abstract Data Source.* De plus, tous ces modules de test sont à l’intérieur d’un environnement de *sandbox* afin d’implémenter la tactique *Sandbox.*

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| Élément | Description |
| DataStrategy | Interface pour les accès aux bases de données |
| RealDBStrategy | strategy concrète permettant d’utiliser la base de donnée pour l’opération normale |
| TestDBStrategy | strategy concrète permettant d’utiliser la base de donnée pour les tests |
| RealDB | La base de données utilisée en opération normale |
| TestDB | La base de données utilisée pendant les tests |
| test | Les tests pour le système |
| Lien entre test et DataStrategy | Les tests font appel à une DataStrategy pour l’utilisation de différentes bases de doo |
| Lien entre DataStrategy et les strategy | Les strategy concrètes implémentent l’interface Strategy |
| Lien entre les strategy et les bases de données | Les strategy concrètes permettent l’accès à leur base de données respectives |

### Texte décrivant la relation entre les éléments et les tactiques

La suite de tests exécutée sur le système utilise le patron stratégie afin de déterminer la source des données à utiliser. De plus les tests sont effectués dans un environnement découplé à part des environnement de QA et PROD. De plus, il est important de noter qu’il y a un environnement sandbox pour les tests unitaires et de régression représenté ici dans notre graphique et un environnement sandbox pour les tests manuel.

# Scénario : Un développeur doit valider que l’ajout d’un module de statistique retourne les bonnes données, ne brise pas d’autres fonctionnalités et est exempt de bogue lors de l’utilisation.

|  |  |
| --- | --- |
| **objectifs d'affaires** | Garantir la validité des statistiques générées pour « market surveillance » |
| **Source** | Unit testers and integration testers via both manual and automated tests |
| **Stimulus** | Lors de l’intégration et de la fin du développement d’un module |
| **Artéfact** | Le nouveau module de statistique et les modules qui en dépendent |
| **Environnement** | Development time and integration time |
| **Réponse** | Execute the test with data from the test data base and capture the output data |
| **Mesure de la réponse** | Augment coverage / probability of detecting a fault |
| **Questions** |  |

## Tactique 1 : Execute assertion

**Description**: Exécution d’un code pour détecter si les valeurs de sorties sont les mêmes que les résultats avant modification du code.

**Justification**: Exécuter un code pour permettre de faire un test avec des données établies et d’avoir les mêmes résultats avant et après l’ajout du module de statistique.

## Tactique 2 : Record/Playback

**Description**: Permets d’enregistrer l’état qui a causé la faute dans le module de statistiques et de pouvoir faire des tests sur cet état.

**Justification**: Utilisé pour capter les erreurs dans le module de statistique et ensuite pouvoir faire des tests afin de découvrir ce qui a causé l’erreur dans le module de statistiques.

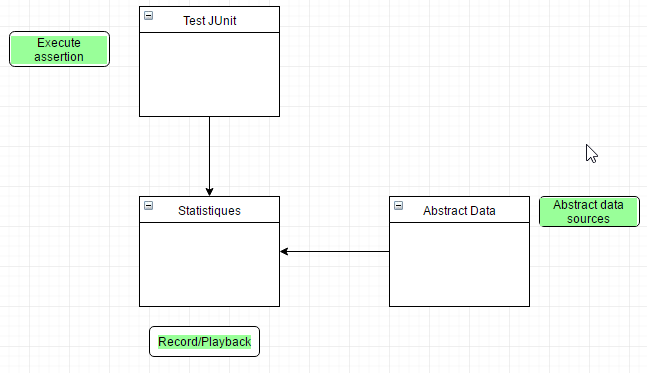
## Tactique 3 : Abstract Data Sources

**Description**: Utiliser des abstractions des données pour produire les statistiques.

**Justification**: Permets de faire des tests sur un environnement de données similaire à l'environnement en production afin de tester si l’implantation du module se fera facilement.

## Vue architecturale (à l’extérieur de la classe)

### Diagramme



### Légende

|  |  |
| --- | --- |
|  | Représente un module dans l’architecture |
|  | Représente les liens et leur direction |

### Texte de description du diagramme

Le module de statistique contient la tactique d’enregistrement d’état pour permettre le test sur les états ayant des erreurs. Un module JUnit est utilisé pour faire des tests à savoir si les bonnes valeurs sont retournées après la modification du module et le module abstract Data est utilisé pour simuler le roulement du module avec des données en production en créant une abstraction de ses données.

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| Module Test JUnit | Module pour faire et exécuter les tests et savoir si les bonnes valeurs sont retournées. |
| Module Statistique | Module de statistiques qui contient le record pour enregistrer les états avec des erreurs et permettre de revoir et corriger les erreurs en testant avec les états. |
| Module Abstract Data | Module permettant de simuler l’entrée de données comme dans le système en production afin de pouvoir faciliter l’implantation du module de statistiques. |

### Texte décrivant la relation entre les éléments et les tactiques

**Execute assertion** est utilisé avec le module de statistiques pour découvrir si un changement dans le module provoque un changement à la valeur de sortie espérée à la fin de l’exécution. Les tests sont faits à l’aide de la librairie JUnit. Les erreurs dans le code seront ensuite enregistrées dans le module statistique à l’aide de la tactique **Record/Playback** qui permet de voir dans quel état l’erreur se produit et de tester avec l’état erroné enregistré. Pour faciliter l’implantation des tests seront faits avec une abstraction de la base de données en utilisant la tactique **Abstract Data** afin de pouvoir simuler le comportement du module dans l'environnement de production.

# Scénario : Assurance qualité sur le nombre de transactions maximum par minutes du système

|  |  |
| --- | --- |
| O**bjectif d'affaires** | S’assurer que le système répond aux demandes d’affaires en ce qui concerne le nombre de transactions souhaité par minutes |
| **Source** | Département de l’assurance qualité |
| **Stimulus** | Besoins d’affaires |
| **Artéfact** | Le système en entier |
| **Environnement** | Copie hors-ligne et isolé du reste du système |
| **Réponse** | * Succès ou échec des tests d’assurance qualité |
| **Mesure de la réponse** | * Temps de réponse moyen * Temps de réponse médian * Capacité du système (Transaction/minute avant dégradation des performances) |
| **Questions** | 1. Est-ce que le système est conforme aux demandes ? 2. Est-ce qu’il est réellement possible de simuler des transactions de façon dite “naturelle” 3. Quel est le seuil limite de dégradation des performances ? 4. Pouvons-nous atteindre ce seuil avec les tests ? |

## Tactique 1 : Record/Playback

**Description**: On enregistre des transactions de façon anonyme pour fin de simulation.

**Justification**: Il est difficile de produire des données représentatives d’un système courant. En utilisant de “vraies” données de production, nous sommes en mesure de tester efficacement le système.

## Tactique 2 : Sandbox

**Description**: Avec les exemples de transactions enregistrés, on simule des scénarios d’achats/ventes à haute intensité.

**Justification**: Avec un environnement de test dédié, nous sommes en mesure de rouler nos tests sans inquiétude d’impacter l’environnement de production.

## Tactique 3 : Abstract Data Sources

**Description**: On augmente la flexibilité pour l’ingestion des données afin d’offrir plus d’un canal pour effectuer des transactions.

**Justification**: Afin de pouvoir passer nos données anonymes au système de test, nous avons besoin d’un canal personnalisé afin de pouvoir ingérer au système les données anonymes de façon contrôlée.

## Vue architecturale (à l’extérieur de la classe)

### Diagramme + Légende

### Testabilite.png

### Texte de description du diagramme

Le diagramme représente, à haut niveau dans le contexte du test de performance, l’infrastructure nécessaire pour effectuer le test d’assurance qualité tel que prescrit par le scénario.

Les données enregistrées provenant de la production sont ingérées dans le système en utilisant la source de données abstraite nommée “test” afin de simuler une certaine quantité de données entrante.

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| System | Représente le système dans son ensemble, incluant Sail, Fix et les autres composantes |
| Anonymized transaction data | Les données enregistrées provenant de l’environnement de production et qui ont été rendu anonyme afin de préserver la confidentialité des clients. |
| Datasource | “Interface” à laquelle des données peuvent être ajoutées au système. |
| Sandbox | Version isolée qui permet d’effectuer des tests sans impacte sur l’environnement de production |

### Texte décrivant la relation entre les éléments et les tactiques

Les données anonymes sont envoyées à la source de données “test” (qui provient d’une source abstraite) et par la suite on envoie ces données au système “sandbox”, qui est en fait une copie du système conforme qui roule en production, mais qui est isolé de cet environnement.

Ces 3 tactiques sont complémentaires et même on pourrait dire qu'elles sont essentielles au bon fonctionnement d’un environnement de test où la simulation d’activités est requise.

# Scénario **:** Retracer l’erreur qui a causé un crash

|  |  |
| --- | --- |
| **Objectifs d'affaires** | Retracer l’erreur qui a causé un crash |
| **Source** | Testeur de système |
| **Stimulus** | Une suite de tests est exécutée sur le système en entier |
| **Artéfact** | Le système en entier |
| **Environnement** | Temps d’exécution |
| **Réponse** | * Les tests ont étés exécutés et on a récupéré les données sur le coverage et le résultat des tests |
| **Mesure de la \_ réponse** | * Coverage * Temps d’exécution * L’effort nécessaire pour trouver un bug (le temps que prend l’écriture des tests) |
| **Questions** | 1. Combien de temps avons-nous pour retracer l’erreur? 2. Quelles sont les ressources humaines permettant de retracer l’erreur? |

## Tactique 1 : Record/Playback

## Description: Réutiliser les log qui ont causé un problème et les fournir aux tests.

**Justification:** Cette tactique permet de s’assurer qu’un problème qui est survenu au niveau du système en entier a été corrigé et qu’il ne se reproduira plus.

## Tactique 2 : Sandbox

**Description:** Utiliser un environnement de développement afin d’exécuter des tests sans impacter l'environnement de production.

**Justification:** La tactique du sandbox permet de tester dans plusieurs cas différents en changeant les variables de l’environnement afin de réussir à corriger un problème. De plus, cela permet de copier l’environnement de production sans avoir à faire des modifications directement sur celui-ci.

**Tactique 3 : Limiter la complexité structurelle**

**Description:** Cette technique limite la complexité du système en réduisant les dépendances cycliques entre les composantes.

**Justification:** Diminuer les dépendances cycliques permet de mieux isoler les composantes d’un système. En isolant ces composantes, il est plus facile de tester l’application et par le fait même de repérer où le code est fautif.

## Vue architecturale (à l’extérieur de la classe)

### Diagramme



### Légende

Rectangle : Écosystème applicatif

Rectangle arrondi : Service ou environnement

### Texte de description du diagramme

Le diagramme représente un utilisateur qui utilise des services. Toutes les requêtes passent par le service de Logging afin de pouvoir être enregistrées. Cela permet de pouvoir “rejouer” des transactions, comme si c’était sur une cassette. Nous pouvons ainsi plus facilement repérer des transactions qui ne se seraient pas bien déroulées, analyser des erreurs lors de tests ou bien même cibler une attaque.

Le fait d’avoir la même application derrière chacun des environnements a plusieurs avantages. Il suffit de fournir différentes configurations et c’est le même système qui roule en arrière. Au besoin, pour les environnements plus isolés, il sera toujours possible de lier directement un service sans qu’il soit disponible dans tous les environnements. C’est en quelque sorte une abstraction qui a été faite ici afin de simplifier les choses et obtenir une certaine robustesse.

**Table de description des éléments du diagramme**

Logging : Le service qui permet d’enregistrer les requêtes et de pouvoir reproduire les requêtes.

Sandbox et Production : Deux environnements sensiblement pareils excepté le fait que l’environnement de Production représente le système en service.

Couche Applicative : Englobe toutes les composantes du système. Par exemple, dans le graphique on peut voir les composantes SAIL et FIX.

### Texte décrivant la relation entre les éléments et les tactiques

Pour la tactique de Record/Playback, le service de Logging permet d’enregistrer toutes les requêtes et par la suite pourra les reproduire dépendamment de l'environnement.

La tactique de Sandbox est représentée par l’environnement Sandbox qui est une réplique de l’environnement de production. Le sandbox permet d'exécuter les requêtes enregistrées par le service de logging afin de retracer l’erreur.

Les composantes SAIL et FIX sont liées par la couche applicative permettant de réduire les dépendances circulaires entre elles. Ceci représente la tactique visant à limiter la complexité structurelle.

# Scénario : Tester les options d’achats et de ventes d’actions avant de le mettre en production.

|  |  |
| --- | --- |
| **objectifs d'affaires** | Améliorer les tests pour rendre le système sans failles.  Tester les options d’achats  (99% code coverage?) |
| **Source** | Unit Tester |
| **Stimulus** | Unit tests |
| **Artéfact** | Module de transaction des actions |
| **Environnement** | Development Time |
| **Réponse** | Results Captured |
| **Mesure de la réponse** | 2 semaines de tests (Implémentation des tests et exécution des tests) |
| **Questions** | 1. Est-ce que la fonctionnalité est bien implémenté ? 2. Pouvons-nous automatiser les tests? 3. Avons-nous un environnement de tests déjà implémenté? 4. Doit-on définir des périodes de tests? |

## Tactique 1 : Executable Assertions

**Description**: Indiquer à quel endroit le système représente une faille.

**Justification**: Couvrir tous les cas de tests et voir si les assertions sont correctes et bien implémentées.

## Tactique 2 : Limit Structural Complexity

**Description**: Limiter la complexité de la structure du logiciel et permet d’éviter les dépendances entre les composantes

**Justification**: Limiter la complexité de la structure permet d’avoir un logiciel plus facilement testable, de sa complexité limité et réduit le temps des exécutions de tests. Aussi, la non-dépendances entre les composantes permet de

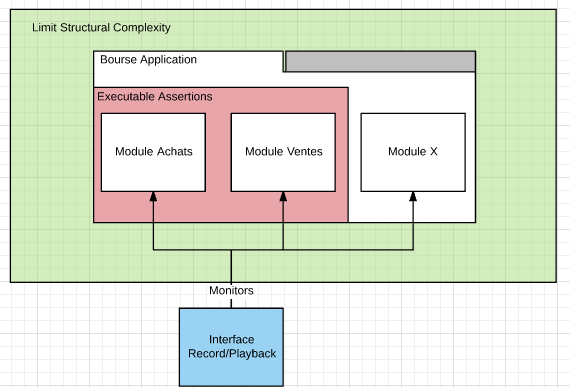
## Tactique 3 :Record/Playback

**Description**: Permet de capturer l’information qui est passer par un interface afin de l’utiliser pour des prochains tests, lors de l’échec de ceux-ci

**Justification**: Cela permet d’enregistrer les données lors de configurations et états difficiles à recréer pour les utiliser dans de futurs tests.

## Vue architecturale (à l’extérieur de la classe)

### Diagramme



### Légende

-Les Module à l’intérieur de l’application représente les modules dans l’application

-Le Bourse Application est l’application contenant les modules

-Les cases de couleurs représentés l’utilisation des tactiques énumérées

### Texte de description du diagramme

L’Application Bourse contient des sous-modules séparées selon les fonctionnalités

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| **Tactique/Élément** | **Description** |
| Module Achats | Le module de l’achat |
| Module Ventes | Le module de vente |
| Module X | Les autres modules de l’application, non-énumérer car ils ne sont pas nécessaires pour le scénario |
| Bourse Application | L’application de la Bourse, avec tous ses sous-modules |
| Exécutables Assertions | La tactique de l’assertion exécutable |
| Limit Structural Complexity | La tactique de limitation de la complexité de la structure |
| Interface Record/Playback | La tactique de Record/PlayBack |

### Texte décrivant la relation entre les éléments et les tactiques

L’application Bourse contient des modules séparées (Achat, Vente, etc.) afin de mieux cerner les modules défectueux (Executable assertions) et limiter le couplage et éviter les dépendances entre les modules (Limit structural complexity). Une interface permet de capturer les informations qui passe pour les utiliser lors des prochains tests (Record/PlayBack).

# Scénario : Les développeurs doivent pouvoir vérifier si les mesures de déploiement automatisé s’effectuent correctement.

|  |  |
| --- | --- |
| **objectifs d'affaires** | Le système est correctement déployé. |
| **Source** | Testeurs systèmes |
| **Stimulus** | Une série de tests est lancée lors du déploiement du système. |
| **Artéfact** | La partie déployée du système. |
| **Environnement** | Le temps du déploiement |
| **Réponse** | Surveillance du système lors du déploiement. |
| **Mesure de la réponse** | Le nombre d’erreur détectées ainsi que le temps nécessaire pour les corriger. |
| **Questions** | 1. Est-ce que les tests sont pertinents ? 2. Comment gère-t-on les tests échoués ? 3. Quel type de fautes sont détectées ? 4. Quels tests sont critiques ? |

## Tactique 1 : Record/Playback

**Description**: Capturer l’information utilisant l’interface pour la passer comme entrée pour les prochains tests.

**Justification**: Lancer le déploiement et capturer les données générées par les instructions afin de tester les prochains déploiements avec ces données comme entrées.

## Tactique 2 : Sandbox

**Description**: Isoler une instance du système du vrai monde pour permettre l’expérimentation sans avoir à renverser les conséquences d’une expérience.

**Justification**: Déployer différents composants du système afin de voir comment ils réagissent.

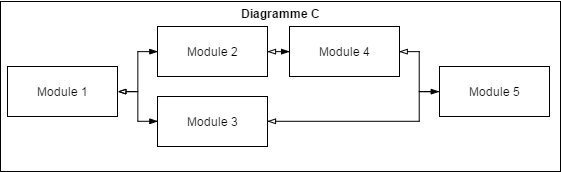
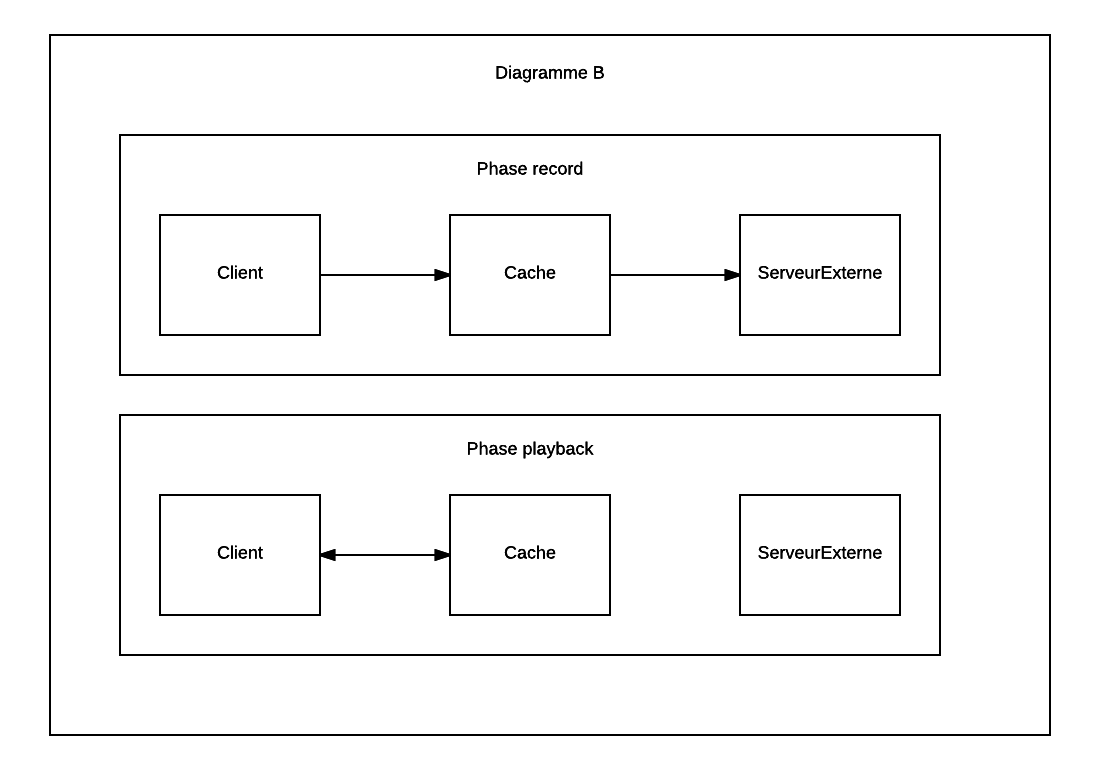
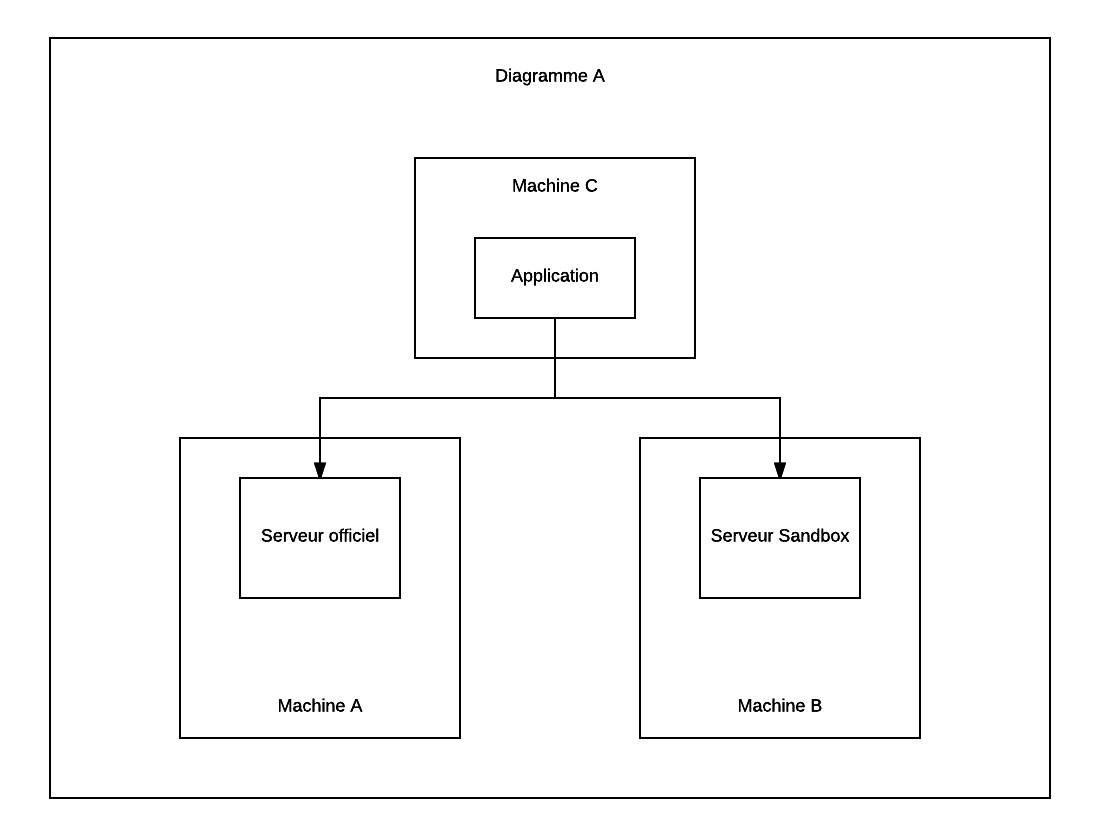
## Tactique 3 : Limit nondeterminism

**Description**: Trouver et documenter toutes les sources non déterministes.

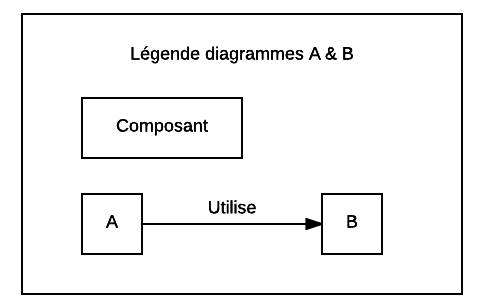
**Justification**: Définir l’ordre de déploiement, ainsi que les dépendances entre les différents sous-systèmes afin d'empêcher les erreurs provenant de sources non déterministes. ex. déploiement des services en parallèle.

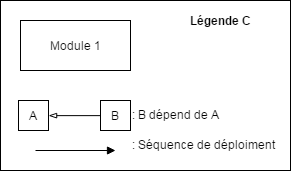
## Vue architecturale (à l’extérieur de la classe)

### Diagramme



### Légende





### Texte de description du diagramme

|  |  |
| --- | --- |
| **Diagramme A** |  |
| Ce diagramme montre l’utilisation d’un serveur de Sandbox sur une autre machine où une copie de l’application est déployée pour tester des changements. | |

|  |  |
| --- | --- |
| **Diagramme B** |  |
| Ce diagramme montre l’utilisation d’une cache qui garde en mémoire les résultats de requêtes à un serveur externe pour qu’au prochain test, la cache renvoie plus rapidement une réponse. | |

|  |  |
| --- | --- |
| **Diagramme C** |  |
| Ce diagramme montre l’ordre de séquencement des différents composants et modules de l’application.  Il montre aussi les dépendances entre ceux-ci. | |

### Table de description des éléments du diagramme

Diagramme A

Application: Code source sur une machine de développement

Machine(A,B,C): Machine physique(Serveur ou machine de développement)

Serveur officiel: Serveur roulant l’application en mode production

Serveur Sandbox: Serveur roulant l’application en mode test

Diagramme B

Phase record: Phase durant laquelle on construit la cache en faisant une requête au serveur externe

Phase replay: Phase durant laquelle on utilise la cache pour accélérer la vitesse des tests

Client: Utilise la cache pour exécuter les tests

Cache: Garde en mémoire les résultats des requêtes au serveur externe

Serveur externe: Serveur retournant des résultats basés sur des requêtes faites par un client ou une cache

Diagramme C:

Module X: module quelconque de l’application.

### Texte décrivant la relation entre les éléments et les tactiques

Le diagramme A représente les éléments nécessaires à l’implémentation d’un serveur Sandbox (Tactique 2).

Le diagramme B montre la logique, ainsi que la structure derrière le processus Record/Playback (Tactique 1). Le dernier diagramme montre un ordre de séquencement général pouvant être utilisée lors de la tactique 3, afin d’éviter les problèmes de dépendances liés au déploiement en parallèle.

# Scénario : Développement des tests pour assurer la validité des opérations possible sur un titre boursier

|  |  |
| --- | --- |
| **Objectifs d'affaires** | Obtenir une couverture de tests de 100% pour tous les différentes opérations possible sur un compte |
| **Source** | Tests automatiques |
| **Stimulus** | Implémentation du système |
| **Artéfact** | Le backend du système de transaction boursière |
| **Environnement** | Lors du développement de l’application |
| **Réponse** | Exécution des tests sont tous réussis et enregistrer les résultats |
| **Mesure de la réponse** | Obtenir une couverture de 100% pour les modules de l’application des transaction boursières. Chaque scénario est couvert via les tests. |
| **Questions** | 1. Quels sont les types de tests possibles dans l’application ? 2. Quoi faire lors d’un échec de tests ? 3. Quels sont les différents outils possibles pour automatiser les tests ? 4. Devrait-on avoir des testeurs externes pour éviter une faute des outils d’automatisation de tests ? |

## Tactique 1 : Enregistrement / Lecture

**Description**: Le but étant d'enregistrer la saisie d’information traversant l’interface et à l'utiliser ensuite comme paramètre d’entré lors des tests

**Justification**: L'état du système lors d’une faute est souvent difficile à recréer. Utiliser le principe d’enregistrement/lecture servira à capturer l’état du système lors de la faute pour ensuite pouvoir recréer exactement le même état et voir où est le problème.

Tactique 2 : Avoir une interface spécialisé**e**

**Description**: Avoir une interface spécialisée pour tester notre application.

**Justification**: Une interface spécialisée pourrait être utilisée pour tester les fonctionnalités des modules que l’on voudrait tester. On pourrait aussi avoir des tests automatisés avec cette interface. Les tests automatisés seraient utiles, car l’on pourrait avoir en temps réel la source des problèmes lorsque ceux-ci sont détectés par les tests.

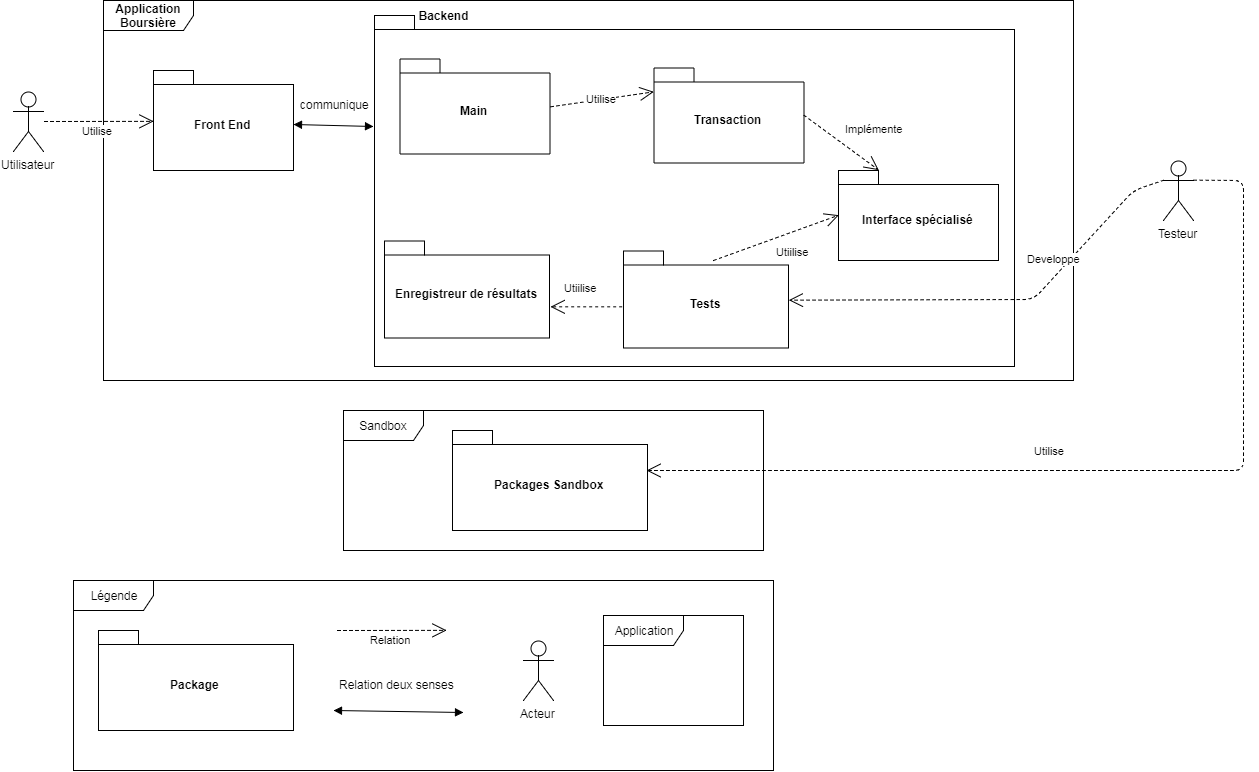
## Tactique 3 : Utiliser une sandbox

**Description**: Avoir une sandbox pour faire des tests de toute sorte sur les modules sans avoir de conséquences sur le vrai système.

**Justification**: On veut pouvoir tester tous les modules au maximum. L’utilisation d’une sandbox va nous permettre d’avoir chaque module dans un état désiré et s’il y a un problème cela ne va pas impacter le vrai système. On pourra aussi utiliser la sandbox pour faire des scénarios d’analyse, de la formation et de la simulation.

## Vue architecturale (à l’extérieur de la classe)

### Diagramme



### Texte de description du diagramme

Dans le diagramme ci-dessus on peut voir comment l’on applique les différentes stratégies à notre application. En premiers lieux on peut voir qu’un utilisateur va utiliser l’application boursière au travers du front end. Ensuite le front end va communiquer au back end avec le main. Le main envoie ensuite la requête au package des transactions. En traitant la transaction, celle-ci passe dans l’interface spécialisée. Ce qui déclenche aussi les tests automatisés et on s’assure que tout fonctionne encore bien. Pour ensuite finir la transaction, envoyer le retour à l’utilisateur et enregistrer le résultat de la transaction et des tests. C’est le testeur qui développe les tests qui seront utilisés. C’est aussi le testeur qui utilise la sandbox pour faire des tests comme il le veut sans avoir peur de briser l’infrastructure déjà mise en place.

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| **Nom de l’élément** | **Description** |
| Utilisateur | Représente l’utilisateur qui accède à l’application |
| Front End | Représente le package contenant la logique derrière le front end de l’application |
| Main | Représente le package contenant les modules de l’application |
| Transaction | Représente le package contenant la logique des transactions effectuées dans l’application |
| Interface spécialisé | Représente l’interface spécialisée de la tactique deux. Aide à tester les modules |
| Tests | Représente l’ensemble des tests de l’application développée par les testeurs |
| Enregistreur de résultats | Représente la classe qui enregistre les résultats entrés à travers l’application pour être utilisé lors des tests |
| Testeur | Représente une personne responsable de tester l’application |
| Package Sandbox | Représente le package correspondant à la tactique trois que le testeur va pouvoir utiliser |
| Application boursière | Représente le package de l’application boursière |

### Texte décrivant la relation entre les éléments et les tactiques

La première tactique est représentée par le package enregistreur de résultats. Il enregistre tout ce qui se rapporte aux tests et aux transactions. Cette information est ensuite accessible et utilisable pour reproduire toute faute durant l’utilisation de l'application. Ensuite, la deuxième tactique est toujours utilisée lors de l’utilisation de l’application. Cela permet au testeur de vouloir tester n’importe quelle transaction qui est faite avec notre application. Il peut aussi utiliser l’interface spécialisée pour faire des tests automatisés sur certaines transactions à un certain temps donné. Et finalement pour la dernière tactique, celle de la sandbox. Celle-ci est à part du reste de l’application, car on ne veut pas qu’elle ait de liens avec celle-ci. Cela nous donne une plus grande ouverture de jeux au niveau de nos tests que l'on peut effectuer. Avec la sandbox on peut décider de l'environnement dans lequel l’on veut tester des transactions et on peut voir en temps réel les résultats sans même impacter l’application originale qui peut toujours rester en service même si l’on fait planter la sandbox. Ce sera le testeur qui sera responsable d’utiliser la sandbox pour faire les tests désirés.

# Scénario : Journaliser toute transaction complétée

|  |  |
| --- | --- |
| **Objectif d'affaires** | Assurer la journalisation des transactions |
| **Source** | Testeur système |
| **Stimulus** | Journalisation des événements systèmes |
| **Artéfact** | Journal des transactions |
| **Environnement** | Opération normale |
| **Réponse** | S’assurer que chaque transaction possède son entrée dans le journal des transactions |
| **Mesure de la réponse** | Effort requis pour détecter les erreurs de journalisation  Temps requis pour tester la journalisation |
| **Questions** | 1. À quelle fréquence les transactions sont-elles complétées par seconde? 2. Quelle est la longueur maximale d’une entrée au journal? 3. Que se passe t-il en cas d’incapacité de journaliser une transaction ou d’une panne de service pendant la journalisation? |

## Tactique 1 : Interfaces spécialisées

**Description**: Définir une interface de journalisation.

**Justification**: Chaque élément qui implémente cette interface journalisera automatiquement ses résultats à la fin de chacune de ses actions. Ainsi, toute action pourra être journalisée et facilitera l’identification de problèmes en retraçant n’importe quelle séquence d’actions.

## Tactique 2 : Record / Playback

**Description**: Enregistrer une séquence d’actions et pouvoir la répéter.

**Justification**: Pouvoir enregistrer une séquence d’action et la répéter autant de fois que désiré pour s’assurer que le résultats final est le même chaque fois. Dans ce contexte, on vérifiera l’ajout d’une entrée au journal des transactions pour chaque transaction complétée.

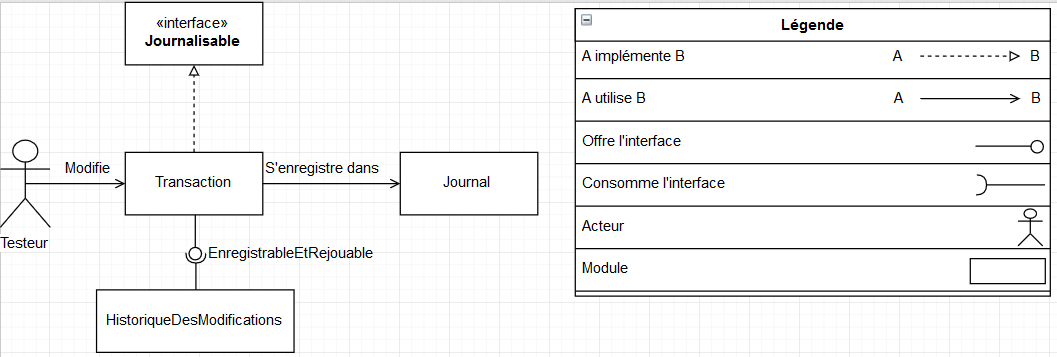
## Tactique 3 : Limiter la complexité structurelle

**Description**: Limiter la complexité de la récupération des informations liées aux transactions.

**Justification**: Les informations des transactions devront être journalisées chaque fois que ces dernières seront modifiées. Par conséquent, il faut assurer que les informations de ces transactions sont simples et testables à la suite d’une ou plusieur modifications. Les transactions devraient donc avoir une très forte cohésion et un faible couplage pour les tester plus facilement.

## Vue architecturale

### Diagramme



### Texte de description du diagramme

Ce diagramme présente une vue module de la journalisation des modifications. Dans ce diagramme, un testeur de l’application modifie les informations d’une transaction. La transaction, lorsqu'elle est modifiée, lance une notification via l’interface EnregistrableEtRejouable qui est détectée par l’historique des modifications. L’historique des modifications enregistre la modification effectuée et offre l’option de la défaire ou de la refaire. Suite à la modification, la transaction enregistre son nouvel état dans le Journal grâce aux méthodes définies dans l’interface Journalisable.

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| **Élément du diagramme** | **Description** |
| Testeur | Un acteur chargé de tester la modification des transactions. |
| Transaction | Un élément logiciel contenant diverses informations sur une transaction entre deux parties. |
| Historique des modifications | Un agrégat de commandes permettant de défaire et de refaire les différentes modifications apportées à une transaction. |
| Journal | Liste des états d’une transaction à chaque étape de son traitement. |
| EnregistrableEtRejouable | Interface offerte par les transactions pour permettre à son consommateur de rejouer les divers événements liés à une transaction. |
| Journalisable | Une interface permettant à une transaction d’être enregistrée dans le journal. |

### Texte décrivant la relation entre les éléments et les tactiques

L’interface « Journalisable » et la relation entre Transaction et HistoriqueDesModifications représentent tous deux des applications de la tactique **Interfaces Spécialisées**. Ces interfaces permettant de donner un comportement particulier aux transactions pour augmenter leur testabilité. Ces interfaces aident aussi à **limiter la complexité structurelle** d’une transaction, puisque les comportements plus complexes d’une transaction sont encapsulé dans ces interfaces. Cette séparation ne modifie pas le couplage et augmente la cohésion de la classe. Finalement, l’historique des modifications permet de refaire et de défaire les actions apportées à une transaction autant de fois que nécessaire, ce qui est l’utilité première de la tactique **Record/Playback**.

# Scénario : Exécution des tests de validation pour les transactions.

|  |  |
| --- | --- |
| Objectifs d'affaires | S’assurer que tous les tests de transactions sont valides. |
| Source | Développeurs des tests d’intégration |
| Stimulus | Tests d’intégration au niveau des transactions |
| Artéfact | Le processus de transaction |
| Environnement | Environnement d’intégration |
| Réponse | Exécuter les tests et capturer les résultats |
| Mesure de la réponse | Pourcentage de couverture des tests |
| Questions | 1. Quel est le pourcentage de couverture? 2. Est-ce que les tests sont fonctionnels? |

## Tactique 1 : Specialized interfaces

Description: Cette tactique consiste à créer des interfaces de tests spécialisés. Elle a pour objectif de permettre le contrôle et la capture de variables pendant l’exécution normale des tests.

Justification: Les méthodes *get*, *set*, *reset* et *report* ont pour objectif de retourner l’état complet des objets du système. C’est grâce aux interfaces que le comportement du système boursier sera vérifié particulièrement lors de transactions bancaires. En faisant usage de cette tactique, il sera possible de détecter les comportements anormaux et d’en trouver la cause afin d’y remédier.

## Tactique 2 : Record/playback

Description: Cette tactique propose le contrôle et la récupération des variables des composantes du système et l’état du système fautif. L’objectif primaire est d’enregistrer l’état du système afin de pouvoir rejouer les événements.

Justification: En rejouant les événements, il sera possible de localiser la ou les transactions fautives., puis par la suite, consolider les tests.

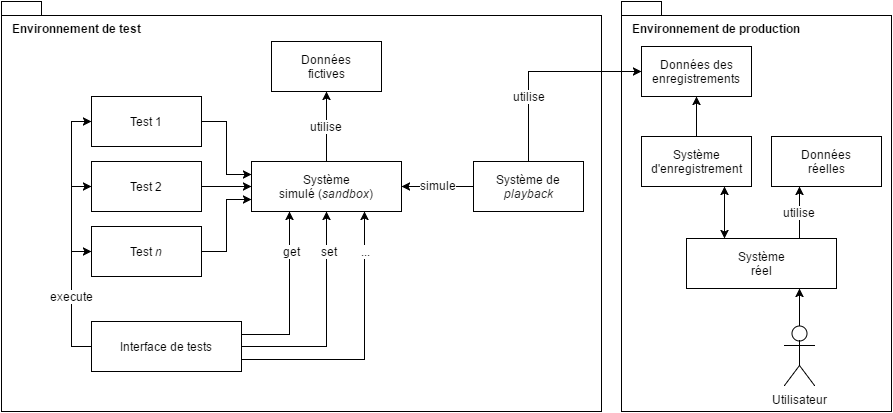
## Tactique 3 : Sandbox

Description: La tactique de *Sandboxing* consiste à simuler un environnement réel de transactions afin d’y effectuer des tests sans affecter les comptes utilisateurs du système boursier.

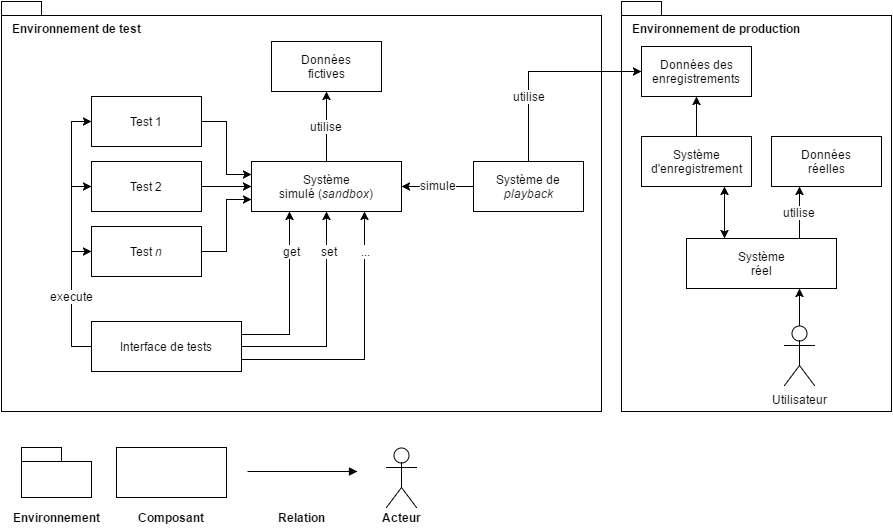
Justification: Étant donné que l’environnement comporte des données sensibles comme de l’argent et des informations personnelles en provenance des clients. Il s’avère fort utile d’avoir un environnement de tests qui simule un comportement identique à celui du système boursier actuel sans toutefois venir modifier les informations réelles utilisées lors des transactions.

## Vue architecturale

### Diagramme



### Légende



### Texte de description du diagramme

D’une part, dans un environnement de production, l’utilisateur se sert d’un système réel qui est relié à un système d’enregistrement et à plusieurs données réelles pour assurer son fonctionnement. Ce système d’enregistrement utilise, à son tour, des données d’enregistrements qui sont utilisées dans l’environnement de test. Dans ce second environnement, nous gérons les tests effectués. Via le système de playback, on simule un système afin de pouvoir exécuter une certaine quantité de tests par une interface qui les relie tous. On conserve ensuite des données reliées à ces tests.

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| **Élément** | **Responsabilité** |
| Test 1,2,3 | Tests exécutés par l’environnement de test |
| Interface de tests | Interface pour les tests |
| Système simulé (sandbox) | Simulation de système pour effectuer les tests |
| Données fictives | Données fictives liées aux tests |
| Système de playback | Système de recouvrement de composantes du système |
| Données des enregistrements | Données liées aux enregistrements |
| Système d’enregistrement | Gère l’enregistrement des états du système |
| Système réel | Système de référence dans le monde réel |
| Données réelles | Données utilisées par le système réel |

### Texte décrivant la relation entre les éléments et les tactiques

|  |  |
| --- | --- |
| **Tactique** | **Description** |
| Specialized interface | Une partie *Environnement de test* possède des interfaces qui sont appelées par le système simulé. Des tests sont ensuite exécutés via ~~l’unique ou~~ les ~~plusieurs~~ interfaces. On permet ainsi la testabilité du système. Avec plusieurs méthodes telles que *get*, *set*, *reset* et autres, il est possible de gérer les interactions entre le système simulé et l’interface de test. |
| Record/playback | Toujours dans l’environnement de test, on peut trouver un système de playback et d’enregistrement. Le système est simulé par le système de playback et, à l’aide de ce dernier, on peut permettre le contrôle de variables dans les plusieurs composantes.  Le système de playback utilise les données d’enregistrement recueillies dans l’environnement de production afin d’établir les grandes lignes de base des composantes. |
| Sandbox | L’environnement de test contient aussi ce qu’on appelle un environnement simulé basé sur le réel. On y effectue les tests. L’environnement de *Sandboxing* utilise les données fictives pour effectuer ses tests regroupés avec la première tactique sous forme d’interface. |

# Scénario : On veut tester des opérations sur la base de données sans affecter celle en production

|  |  |
| --- | --- |
| **Objectifs d'affaires** | Création d’un “sandbox” permettant d’effectuer des transactions fictives qui n'affectent pas les données en production. |
| **Source** | Un testeur unitaire voulant effectuer des tests de transactions. |
| **Stimulus** | Une série de tests pour les transactions grâce au “sandbox”. |
| **Artéfact** | Transactions |
| **Environnement** | Lors du développement ~~et la compilation~~. |
| **Réponse** | Tests des transactions |
| **Mesure de la réponse** | 100% des tests de transactions ont passés et la base de données en production n’a pas été affecté. |
| **Questions** | 1. Sur quelle base de données veut-on tester les transactions (“sandbox”) ? 2. Comment va-t-on configurer l’accès au “sandbox” ? 3. À quelle fréquence rafraîchie-t-on la “sandbox” ? |

## Tactique 1 : Sandbox

**Description** : Le “sandbox” est un concept permettant d’effectuer des actions fictives qui modifierait normalement une base de donnée en production.

**Justification** : Grâce à ce concept, les testeurs seront en mesure d’effectuer des transactions dans le système sans affecter les données courantes dans la base de données en production. L’entreprise ne peut pas se permettre d’effectuer des tests sur de l’argent réel appartenant aux clients.

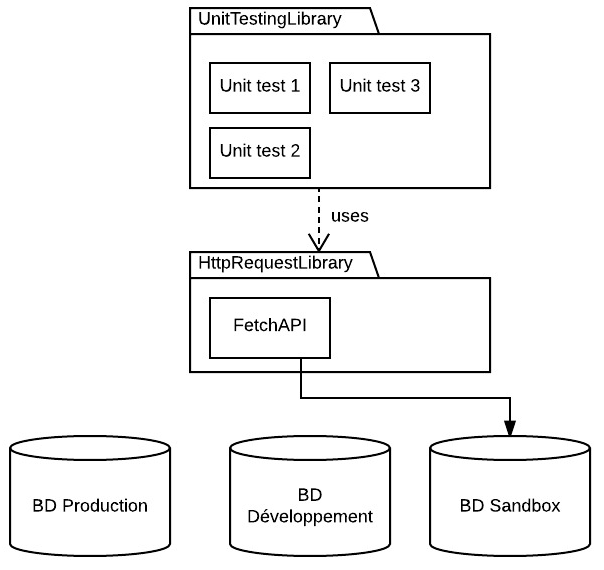
## Tactique 2 : Executable assertions

**Description** : L’exécution de tests unitaires vont nous permettre de tester les transactions possibles pour un utilisateur.

**Justification** : Puisque nous travaillons dans un système bancaire, il est important d’effectuer des assertions pour comparer l’état avant et après la transaction avec la valeur désirée afin d’assurer la validité des transactions. Sans ces tests, des transactions incomplètes pourraient affecter des milliers d’utilisateur de votre système.

Vue architecturale

## Diagramme



## Description du diagramme

Le module des tests unitaires utilise celui de la bibliothèque à tester. Les requêtes traitées par l’API vont effectuer des modifications dans la base de données de test, qui est séparée de celle de production afin d’éviter de corrompre les données actuelles. Cela permet de tester toutes les transactions possibles en toute sécurité, ce qui rendra le système plus robuste grâce à une couverture de test plus élevée.

## Table d’éléments

|  |  |
| --- | --- |
| **UnitTestingLibrary** | La bibliothèque qui contient les outils nécessaires pour effectuer des tests, ainsi que les tests eux-mêmes. |
| **HttpRequestLibrary** | C’est là où les requêtes vers la base de données s’effectuent. Cette librairie ne connaît pas à la base quelle base de données utiliser. On doit lui passer la connexion en paramètre. |
| **BD Sandbox** | La base de données “sandbox”, contenant des données de tests, telle qu’une sauvegarde antérieure de la base de données réelles. Elle est utilisée par les tests. |
| **BD Développement** | La base de données utilisée par les développeurs qui est, encore une fois, distincte de celle en production. Par contre, elle est séparée de celle utilisée par les tests. |
| **BD Production** | La base de données utilisée en temps réel par le système. |

## Liens entre les vues et les tactiques

Une librairie de tests permet de vérifier si les fonctionnalitées de base sont toujours fonctionnelles. Afin de pouvoir effectuer des tests plus poussés sur des données plus représentatives, le programme emploiera une BD Sandbox durant les tests. La Sandbox peut être une sauvegarde antérieure de la BD réelle.

# Scénario : Tester la création des transactions

|  |  |
| --- | --- |
| **Objectifs d’affaires** | Être capable de détecter des anomalies dans le processus de création de transactions. |
| **Source** | Testeurs |
| **Stimulus** | Des tests aux niveaux de service de création de transactions. |
| **Artéfact** | Système de création de transactions |
| **Environnement** | Développement |
| **Réponse** | Capturer les résultats des tests exécutés. |
| **Mesure de la réponse** | * Avoir un taux d’au moins 85 % de couverture du code. * Les tests s’exécutent en moins de 10 minutes. * Réduire l’exposition aux risques. |
| **Questions** | * Quelles répercussions peut-il y avoir si un ou plusieurs tests échouent? * Y a-t-il des différences entre l’environnement de développement et l’environnement de production? * Quels types de données devrait-on tester? |

## Tactique 1 : Sandbox

**Description**: Réplique exacte de l'environnement de production qui permet de tester tout ajouts de fonctionnalité afin d’observer les impacts sur le système réel, sans toutefois endommager ce dernier.

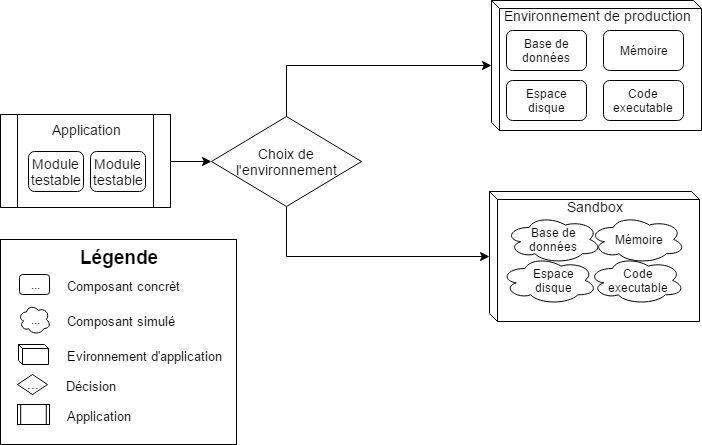
**Justification**: Cette tactique nous permet d’avoir une idée du comportement du système de production lorsque confronté à n’importe quelle situation.

**Tactique 2 : Limit Structural Complexity**

**Description**: En limitant la complexité de l’architecture, il devient beaucoup plus simple de mettre en place une structure de tests. Dans ce sens, il est plus facile de cibler un composant à tester.

**Justification**: Cette tactique nous permettrait de seulement tester la création des transactions si on le souhaite sans avoir à tout tester.

**Vue architecturale**



### Texte de description du diagramme

L’application est composée de modules séparés. Dans sa configuration, un environnement d’exécution est spécifié (Production ou Sandbox). L’environnement Sandbox est composé de répliques simulées des ressources de l’environnement de production.

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| **Application** | Modules testables |
| **Choix de l’environnement** | Détermine si on test dans un environnement de production ou le sandbox. |
| **Environnement de Production** | Environnement officiel ou l’application est pleinement en fonction. |
| **Sandbox** | Environnement d’application où tous les composants sont simulés. |
| **Composant** | Service requis au fonctionnement de l’application. |
| **Composant (simulé)** | Composant clonée à partir des composant non simulé. |
| **Décision** | Noeud qui choisit quel environnement l’application devra accéder. |

### Texte décrivant la relation entre les éléments et les tactiques

L’environnement “Sandbox” est une copie presque exact de l’environnement de production. Son rôle est de fournir un système qui réagira aussi fidèlement que possible aux actions de l’application, ce qui améliore la testabilité du système. Les modules testables qui composent l’application représentent notre implémentation de la tactique “Limit Structural Complexity”. Comme mentionné dans la justification de cette tactique, cette implémentation nous permet de simplifier les interactions (notamment les tests) dans l’application.

# Scénario 1 Création de tests suite à l’ajout d’une nouvelle fonctionnalité sur le système de Front End Access:

|  |  |
| --- | --- |
| **Objectifs d'affaires** | Atteindre 95% de couverture de test sur le système. |
| **Source** | Tests unitaires et test d’intégration. |
| **Stimulus** | L’ajout d’une nouvelle fonctionnalité de gestionnaire de langue. |
| **Artéfact** | Le système de Front End Access. |
| **Environnement** | Dans l’environnement de développement. |
| **Réponse** | * Exécuter les tests unitaires et les tests d’intégration et évaluer les mesures. |
| **Mesure de la réponse** | Le niveau d’effort requis pour écrire les tests (Exemple: Analyse d’impact).  Le % de couverture atteint par les tests. |
| **Questions** |  |

**Tactique 1: Interfaces spécialisées**

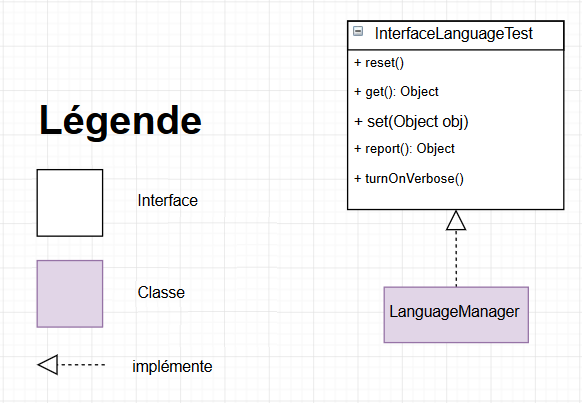
**Description:**

Les interfaces spécialisées permettent de contrôler ou de capturer les valeurs des variables d’un composant dans les tests.

**Justification**:

Intégrer la tactique d’interfaces spécialisées permettra de vérifier si le changement de langage a vraiment modifié les messages affichés pour les usagers. En effet, en utilisant les interfaces spécialisées, on pourra créer des méthodes de set et get afin de bien voir les valeurs données lorsque nous aurons changé le paramètre de langue.

### Diagramme et légende



### Texte de description du diagramme

Le diagramme illustre l’implémentation d’une interface spécialisée par la classe LanguageManager. L’interface spécialisée, qui s’appelle InterfaceLanguageTest, définit des méthodes qui serviront dans les tests.

### Table de description des éléments du diagramme

|  |  |
| --- | --- |
| L’interface | L’interface spécialisée qui définit des méthodes afin de pouvoir faire des tests |
| Classe | Une des classes qui implémente l’interface spécialisée |
| Flèche | Lien entre deux composants |

### 

### Texte décrivant la relation entre les éléments et les tactiques

L’interface InterfaceLanguageTest définit des méthodes qui serviront à tester les classes enfants, c’est-à-dire les classes qui héritent d’elle. Ces méthodes permettent de contrôler et de capturer l’état de classe enfant.