Modifiabilité

Table des matières

1. Scénario (Modificabilité): Rendre l’application web accessible aux clients SAIL en Chine. 2

2. Scénario : Développeur ajoute une interface statistique avancée au module “Market surveillance” 5

3. Scénario : On veut ajouter un module de SMS pour les alertes 8

4. Scénario : Réduire le temps de développement 11

5. Scénario : Le logiciel boursier doit pouvoir être adapté à plusieurs systèmes boursiers. 14

6. Scénario : Implémenter un système permettant à un utilisateur de se connecter au système bancaire avec son compte Google 17

7. Scénario : Par mesure de sécurité, nous voulons ajouter un champ «adresse courriel unique du demandeur» au niveau de la transaction. 20

8. Scénario : Fusion de systèmes boursiers 23

9. Scénario : Ajouter un module de caméra pour prendre une photo lors d’un échec de NIP 28

10. Scénario : L’équipe veut vendre le système à d’autre entreprise boursières qui ont déjà un système afin de gérer les transactions client. 31

11. Scénario : L’entreprise voudrait ajouter une nouvelle devise 34

12. Scénario : Effectuer des transactions avec différentes bourses internationales qui utilisent des protocoles de communication différents 37

# Scénario (Modificabilité): Rendre l’application web accessible aux clients SAIL en Chine.

| **Objectifs d'affaires** | Augmenter la portée de l’application et donc le nombre d’utilisateurs réguliers |
| --- | --- |
| **Source** | Les développeurs |
| **Stimulus** | Requête d’ajout d’un système de gestion des langues |
| **Artéfact** | L’interface utilisateur et l’architecture de l’application |
| **Environnement** | Design time |
| **Réponse** | Il faut modifier l’application, la tester et la déployer |
| **Mesure de la réponse** | * Le temps pour faire la modification. * Le coût d’ajout des modifications. * Le niveau d’effort mis pour modifier l’architecture. * La complexité de la modification. |
| **Questions** | 1. Combien de langue devons-nous ajouter? 2. Si on ajoute seulement une langue chinoise, laquelle? Mandarin, Cantonais? |

**Tactique 1: Reduce coupling: Code Refactor**

**Description:**

Le code refactoring est une tactique utilisée pour retirer les responsabilités communes des modules. Ces responsabilités sont ensuite placées dans un nouveau module parent. Les anciens modules deviennent des sous-modules du parent, ce qui réduit le couplage et permet d’ajouter d’autres modules similaires facilement.

**Justification**:

Puisqu’il existe déjà une langue par défaut, il faut assurer que le module de langue existant soit compatible avec le gestionnaire de langues qu’on veut ajouter. Le code refactoring permettra de centraliser les responsabilités communes entres les différents modules. De plus, cela permettra de réduire le couplage, ce qui réduira le nombre de modules affecté par de futurs changements.

**Tactique 2: Split modules**

**Description:**

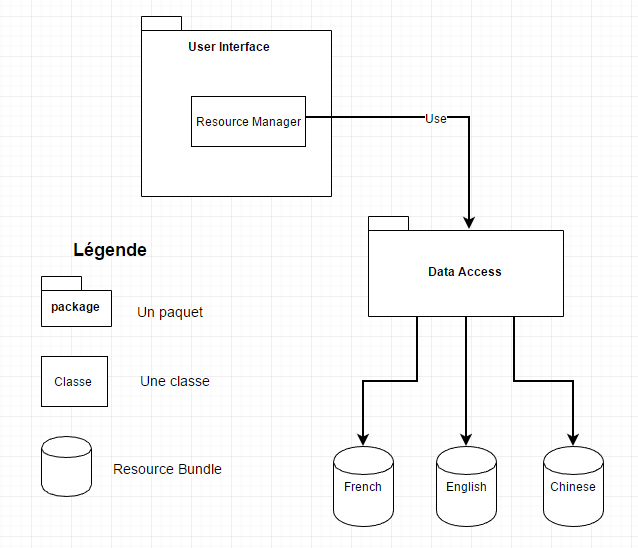
La tactique Split module permet de réduire le coût de de modifications ainsi que les dépendances.

**Justification**:

La séparation de la gestion de langue en de plus petits modules permettra de réduire le coût de modification dans le futur lorsqu’on voudra ajouter d’autres langues.

### 

### Diagramme et légende



### Texte de description du diagramme

Lors du changement de langage de l’interface usager, le système va utiliser Resource Manager qui va appeler le paquet Data Access. Ce paquet va chercher le type de ressource requis pour afficher le langage voulu. En effet,l'internationalisation du système est gérée par la classe Resource Manager qui contient la String à afficher dans l’interface usager.

### 

### 

### 

### Table de description des éléments du diagramme

| Ressource Manager | La classe qui gère la langue à afficher. |
| --- | --- |
| Data Access | Le paquet contenant les classes étant responsables de l’accès aux données. |
| Resource Bundle | Un fichier contenant toutes les constantes utilisées par la langue. |

# Scénario : Développeur ajoute une interface statistique avancée au module “Market surveillance”

| **objectifs d'affaires** | Implémenter cette interface utilisateur à moindre coût |
| --- | --- |
| **Source** | Développeur |
| **Stimulus** | Désire ajouter une interface de statistiques avancées |
| **Artéfact** | Code |
| **Environnement** | During design time |
| **Réponse** | Changement fait à l’interface de “Market surveillance” et les tests unitaires |
| **Mesure de la réponse** | Mesurer le temps d’implémentation, idéalement en 1 semaine |
| **Questions** | 1. Quel est le temps de développement normal? (1 mois) 2. Doit-on considérer l’ajout de modules semblables dans l’avenir? (oui) 3. Une interface similaire existe-t-elle déjà (non) |

## Tactique 1 : Restrict dependencies

**Description**: Restreindre le nombre de dépendance inutile à la nouvelle fenêtre

**Justification**: Permettre d’avoir une architecture en couche en ne permettant pas les appels vers les couches plus hauts

## Tactique 2 : Refactor

**Description**: Tactique utilisée pour faire le ménage et s’assurer que le code ne soit pas dupliqué ou trop compliqué.

**Justification**: Permettre d’avoir un code propre et de partager les responsabilitées communes avec un module parent et réduire le coupling.

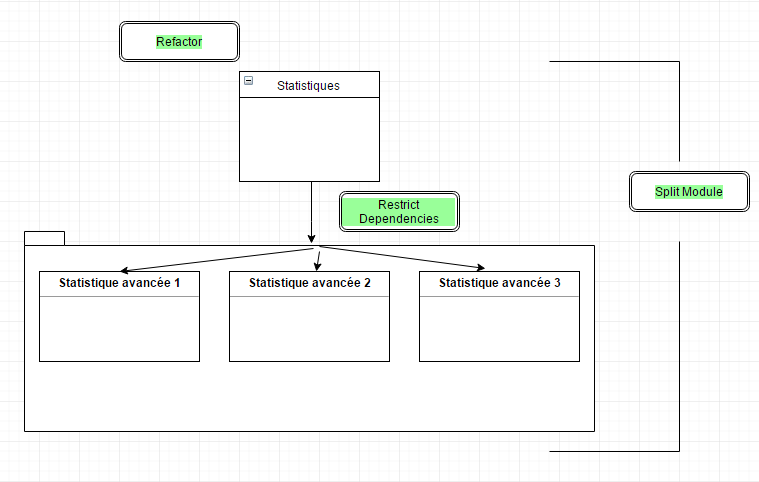
## Tactique 3 : Split module

**Description**: Réduire le module en plusieurs plus petit aux responsabilités moindres.

**Justification**: Permettre d’augmenter la modifiabilité en séparant les modules tels que statistiques et améliorer l’ajout de module additionnel ce qui améliore le coût des changements pour l’avenir.

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

### Diagramme



### Légende

Les carrés représente les classes.

Le dossier représente le(s) package(s).

Les flèches montrent le sens des appels.

Les éléments en vert représente les tactiques utilisées.

### Texte de description du diagramme

Le diagramme donne une vue d’ensemble sur l’architecture à utiliser pour l’implémentation de l’interface de statistiques avancées. Une classe parents Statistique contenant les statistiques et certain calcul de base seras en mesure d’appeler le package Statistique avancée ou les différents calculs seront stockés dans des classes différentes pour améliorer la modifiabilitée future.

### Table de description des éléments du diagramme

| Éléments | Responsabilité |
| --- | --- |
| Classe statistique | Permet d’avoir accès aux informations sur les données et faire certain calcul statistique de base et faire les appels vers les statistiques avancées. |
| Classes statistiques avancées | Permet d’avoir une classe pour une opérations de statistiques avancées |
| Flèche | Montre la direction des appels entre les classes et packages. |

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

La tactique **refactor** pour avoir une classe maîtresse pour empêcher la duplication de code est utilisée avec la classe statistique qui contient les méthodes de base sur les données.

La tactique **split-module** permet de séparer les classes statistiques avancées en plusieurs sous-ensemble et se combine bien avec la tactique de **refactor** plus haut en ayant séparé les statistiques basic dans une autre classe.

La tactique **restrict dependencies** est ensuite utilisé pour permettre d’avoir des appels cohérent entre ses classes en focussant les appels dans une seule direction.

# Scénario : On veut ajouter un module de SMS pour les alertes

| **objectifs d'affaires** | Ajouter l’envoi des messages SMS au système d’alerte |
| --- | --- |
| **Source** | Le développeur suite à la demande du département qui supporte le système d’alerte (Technical Help Desk & Technical Operation) |
| **Stimulus** | Le client veut ajouter une nouvelle fonctionnalité d’alertes |
| **Artéfact** | System monitoring and management code |
| **Environnement** | Design time |
| **Réponse** | * Effectuer la modification * Tester la modification * Déployer la modification |
| **Mesure de la réponse** | * Nombre d’artéfacts affectés par la modification * L’effort nécessaire pour effectuer la modification * Le coût pour effectuer la modification * Les nouveaux défauts introduits avec l’ajout de la modification |
| **Questions** | 1. Quel type de service allons-nous utiliser pour faire l’envoi des SMS ? 2. Est-ce que l’envoi est configurable par utilisateur/groupe/organisation ? 3. Quels sont les artéfacts affectés par la modification 4. Est-ce que les alertes sont configurables (Répétition, sévérité, Filtres) |

## Tactique 1 : Encapsulate

**Description**: Chaque module du *system monitoring and management* est encapsulé de façon à limiter le couplage.

**Justification**: Le module de SMS sera dans son propre package. Ainsi il sera plus facile de délimiter le code et d’en ajouter de nouveau. L’ajout, la modification ou la suppression de nouveau module sera ainsi simplifié et n’inclura pas de dépendance cachée entre les fonctionnalités.

## Tactique 2 : Increase semantic coherence (Plugin)

**Description**: Séparer les responsabilités dans des modules différents.

**Justification**: En séparant les responsabilités, l’ajout du module de SMS aura moins de chance d’impacter plusieurs responsabilités.

## Tactique 3 : Intermediary (Proxy)

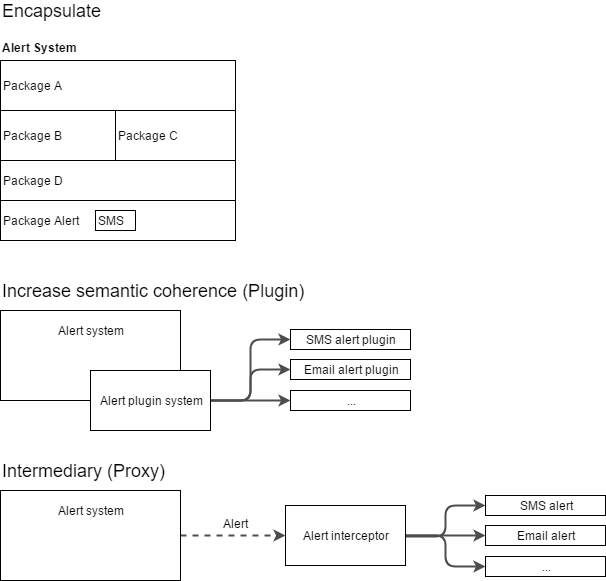
**Description**: Le proxy intercepte les alertes en place et introduit la notion des SMS aux alertes.

**Justification**: Cette tactique est transparente au système principal puisqu’il n’est pas affecté par l’ajout de fonctionnalité. Cette tactique est peu élégante, mais permet d’ajouter des fonctionnalités à un système fonctionnel sans risques d’affecter l’intégrité du système.

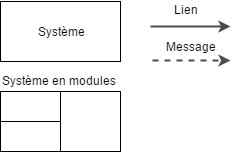
Le proxy permet de faire un nouveau départ sous une autre architecture et de créer une application spécialisée pour cette fonctionnalité.

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

### Diagramme



### Légende



### Texte de description du diagramme

Le diagramme représente les trois tactiques qui ont été proposées.

La première représente la composition d’un logiciel en modules. Chaque module possède une responsabilité précise et extensible, au besoin. Dans notre cas le nouveau module sert à l’envoi d’alerte par SMS.

La deuxième tactique ajoute un système de “plugin” pour étendre les fonctionnalités d’un logiciel existant sans avoir à modifier le code d’origine, si ce dernier possède déjà le module de gestion de plugin.

Avec un système de plugin, il est facile d’ajouter de nouveaux types d’alertes, tels que les SMS, courriel, etc.

La troisième tactique est l’interception (proxy) des messages actuellement générés par le système actuel afin de les manipuler et créer de nouveaux messages.

Ce type de système est idéal dans un scénario où il est impossible ou fortement déconseillé de modifier le code source d’un système en place. En étant une application indépendante à part entière, le proxy peut évoluer indépendamment du reste du système.

### Table de description des éléments du diagramme

| Alert System | Représente le système d’alerte en son entièreté |
| --- | --- |
| Package X | Représente des modules du système d’alerte |
| Alert Interceptor | Le proxy/intercepteur de message qui applique la nouvelle logique d’affaires |
| Plugin system | Système d’extension du système d’alerte. Des modules personnalisés peuvent s’y attacher. |
| Message | Représente un échange d’information entre deux ou plusieurs systèmes |

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

Le système d’alerte entre dans un état où il doit envoyer une alerte. Le système alors génère le contenu de l’alerte et redirige ce contenu vers le processus d’envoi. Le processus d’envoi d’alerte reste similaire entre les tactiques, avec comme différence le nombre de modules impliqués pour envoyer l’alerte.

# Scénario : Réduire le temps de développement

| **Objectifs d'affaires** | Réduire le temps (coûts) de développement |
| --- | --- |
| **Source** | Développeur |
| **Stimulus** | Ajout d’une ou plusieurs fonctionnalités |
| **Artéfact** | Code |
| **Environnement** | Temps de conception |
| **Réponse** | * Faire une modification * Déployer une modification |
| **Mesure de la \_ réponse** | * Temps requis * Effort nécessaire * Coût |
| **Questions** | 1. Combien de temps avons-nous pour faire la modification?  2. Quelle est l’ampleur des modifications?  3. Les coûts engendrés par les modifications?  4. Quels sont les risques de briser une fonctionnalité? |

## Tactique 1 : Réduire la grosseur d’un module

## Description: Réduire la grosseur des modules qui ont trop d’aptitudes en créant de plus petits module pour chaque fonctionnalité.

**Justification:** Raffiner un module en de plus petits modules permet de réduire le coût des modifications et de réduire le temps de modification dans le future.

**Tactique 2 : Augmenter la cohésion**

**Description:** Mettre les responsabilités qui se ressemblent dans le même module.

**Justification:** Réunir les modules permettent d’avoir un seul module à modifier lorsqu’on veut ajouter une fonctionnalité ou faire une modification.

## 

## Tactique 3 : Diminuer le couplage

**Description:** Consiste à réduire les dépendances entre les différents modules.

**Justification:** Réduire le couplage entre les modules permettent aux développeurs de modifier ceux-ci sans avoir à effectuer des changements sur les autres modules.

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

### Diagramme

### 

### Légende

Un dossier bleu correspond à une composante.

Une ligne représente une relation.

Un carré blanc représente un module.

### Texte de description du diagramme

Le composant Fix est dépendant à la bourse de MTL. Sail est dépendant à la Bourse de Mtl. La bourse de Mtl coordonne les actions de Sail et Fix.

**Table de description des éléments du diagramme**

* Bourse de Mtl : L’entité principale du système. C’est cette composante qui coordonne le système.
* FIX: Interface standard de message pour système boursier
* Market operations : Gestion des alertes par instrument ou firme

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

On applique toutes les tactiques sur la composante *Bourse de Mtl*, puisque c’est celle-ci qui coordonne le système qu’on veut modifier. On va raffiner de cette composante en appliquant la tactique **Réduire la grosseur des modules**. Ainsi, nous nous retrouverons avec plusieurs petits modules facile à modifier.

Par la suite, pour faciliter encore plus la modification de l’application, nous allons réunir les modules semblables en un en appliquant la tactique **Augmenter la cohésion**. Ainsi, il y aura moins de modules à changer lors d’un ajout ou d’une modification de fonctionnalité.

Enfin, on va réduire les dépendances entre les différents modules, afin de réduire le nombre de retouches à faire lorsqu’on ajoute ou modifie une fonctionnalité. Ceci sera fait avec l’aide de la tactique **Diminuer le couplage**.

# Scénario : Le logiciel boursier doit pouvoir être adapté à plusieurs systèmes boursiers.

| **objectifs d'affaires** | L’entreprise désire offrir ses systèmes à d’autres bourses selon 3 échelles de performance/prix différentes |
| --- | --- |
| **Source** | L’équipe de développement ~~reçoit une demande de l’entreprise qui souhaite avoir un système “scalable”, pouvant être utilisé par plusieurs bourses.~~ |
| **Stimulus** | Une directive de rendre le système scalable, en fonction du client et de ses besoins |
| **Artéfact** | Code, resources, configurations, |
| **Environnement** | ~~Compile time, build time,~~ design time |
| **Réponse** | Implémenter, tester et déployer la modification |
| **Mesure de la réponse** | Échelle de prix de vente des configurations  Temps pour rendre le système scalable et flexible  Introduction potentielles de nouveaux problèmes |
| **Questions** | 1. Quel délai est disponible pour rendre le système scalable, considérant que la bourse de Montréal demande un temps de mise en marché rapide? 2. Au delà du nombre de requêtes par seconde, est ce que les autres bourses auraient besoin de modifications additionnelles? |

## Tactique 1 : Séparation des modules

**Description**: Tous les modules responsables de la logique d’affaire spécifique à la bourse de montréal devraient être isolés.

**Justification**: Ceci permettrait d’interchanger rapidement le module avec celui développé pour un nouveau client, réduisant le temps d’implémentation

## Tactique 2 : Abstraction du service transactionnel

**Description**: Implémentation d’un middleware couplé à un load balancer pour gérer le service transactionnel

**Justification**: En créant une abstraction pour le service transactionnel, il suffit de manipuler le nombre de ressources (serveurs) disponibles pour pouvoir vendre notre système à des clients ayant des besoins de performance différents. De plus, cette tactique s'intègre bien à notre tactique de disponibilité, la rendant plus facile à implémenter.

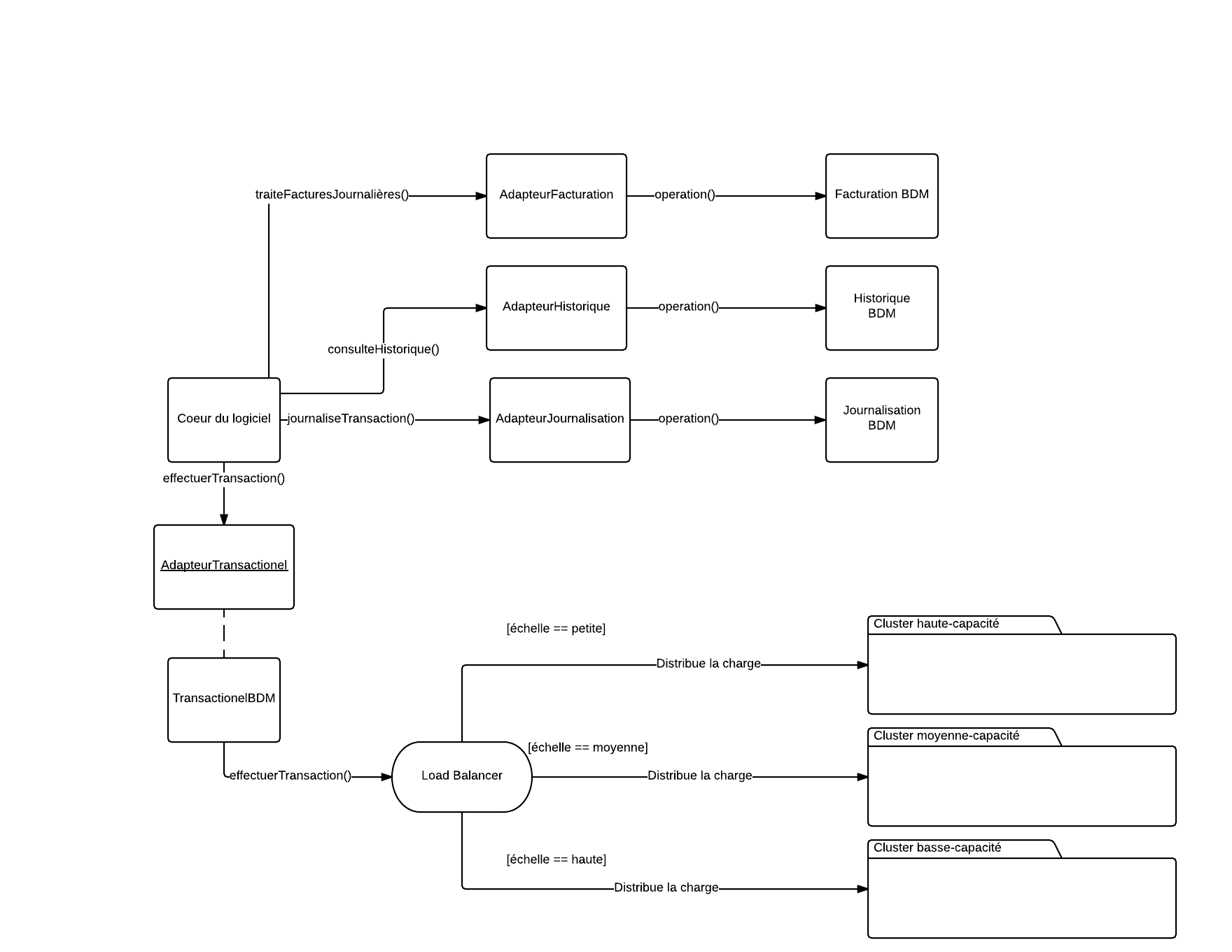
## Tactique 3 : Utilisation d’un intermédiaire (patron adaptateur)

**Description**: Un patron adaptateur devrait être utilisé afin d’accéder aux données uniques à la bourse de Montréal

**Justification**: Afin que le coeur de notre logiciel soit aussi adaptable que possible, un adaptateur devrait être utilisé pour appeler les services spécifiques internes de la bourse de Montréal. Ainsi, il faudrait seulement créer un nouvel adaptateur lors de l’implémentation avec une autre bourse.

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

### Diagramme



### Légende

Les paquetages UML sont ici utilisés pour représenter des cluster de serveurs

Les flèches sont des interactions entre les composants du systèmes

Les balises [] encadrent une condition

Les carrés arrondis sont des modules du logiciel

### Texte de description du diagramme

Le diagramme représente les trois tactiques. On remarque que chaque fonctionnalité du logiciel (Journalisation, Historique, Facturation et transactionnel) ont été séparés en modules. Les modules, bien que génériques, ont des spécificité en fonction de la législation de l’endroit de la bourse. Ainsi, ils sont aisément interchangeables pour un autre module, si le besoin se présente. Ceux-ci sont utilisés à partir d’un adapteur: ainsi, dans le cas ou le module serait un module maison, nous n’aurions qu'à créer un adaptateur pour connecter ce module au coeur du logiciel. De plus, l’utilisation d’un adaptateur fait en sorte que nous n’avons pas a modifier le coeur du logiciel si un module alternatif était nécessaire. Finalement, la partie du bas démontre l’abstraction transactionnelle. En effet, le coeur n’a pas besoin de savoir l’échelle du logiciel, il n’a qu'à acheminer les requêtes vers le système transactionnel, qui utilise un load balancer pour distribuer la charge entre les serveurs du cluster approprié. Nous avons représenté les trois types de cluster sur ce graphique, mais chaque configuration du système n’a qu’un seul type de cluster

### Table de description des éléments du diagramme

| Élément | Description |
| --- | --- |
| Coeur du logiciel | Il s’agit du noyau du logiciel, auquel sont connectés les différents modules de chaque bourse. |
| AdapteurX | Sert à adapter les requêtes du coeur en une requête pour un module X |
| FacturationBDM | Gère la facturation pour la bourse de Montréal. Interchangeable avec un module de facturation d’une autre bourse. |
| HistoriqueBDM | Gère la conservation de l’historique transactionnel de la bourse de Montréal. Interchangeable avec un module d’historiques d’une autre bourse. |
| JournalisationBDM | Gère la journalisation des communications entrantes et sortantes de la Bourse de Montréal. Interchangeable avec un module de journalisation d’une autre bourse. |
| TransactionelBDM | Gère les transactions de la bourse de Montréal. Interchangeable avec un module transaction d’une autre bourse, si besoin est, |
| LoadBalancer | Gère la répartition des requêtes entre les différents serveurs de son cluster. |
| Cluster haute-capacité | Contient une configuration de serveurs capable de servir 200k ordres par seconde. Seulement présent dans les configurations à Grande échelle. |
| Cluster moyenne-capacité | Contient une configuration de serveurs capable de servir 40k ordres par seconde. Seulement présent dans les configurations à moyenne échelle. |
| Cluster basse-capacité | Contient une configuration de serveurs capable de servir 20k ordres par seconde. Seulement présent dans les configurations à petite échelle. |

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

Les modules FacturationBDM, HistoriqueBDM, JournalisationBDM et TransactionelBDM sont associés à la première tactique. L’élément AdaptateurX, qui englobe les adaptateurs présents dans notre architecture, sont associés à notre seconde tactiques, car ils s’assurent de la compatibilité entre le coeur et les différents modules. Enfin, les cluster ainsi que le load balancer permettent de gérer des quantités de requêtes variables en utilisant la quantité minimale de ressources, et s’associent donc à notre troisième tactique.

# Scénario : Implémenter un système permettant à un utilisateur de se connecter au système bancaire avec son compte Google

| **objectifs d'affaires** | Modifier l’interface utilisateur pour améliorer l’utilisabilité |
| --- | --- |
| **Source** | Développeur |
| **Stimulus** | Désire modifier le système d'authentification pour permettre de se connecter avec un compte google. |
| **Artéfact** | Système d’authentification |
| **Environnement** | Runtime |
| **Réponse** | Faire la modification  Tester la modification  Déployer la modification |
| **Mesure de la réponse** | Front-end et back-end affectés  1 journée  1 personne  Affecte le système d’authentification  Peut possiblement briser le système initial d’authentification |
| **Questions** | 1. Comment les informations sur les comptes Google sont accessibles publiquement (API OAuth2)? 2. Comment informer les utilisateurs qu’ils peuvent utiliser la nouvelle authentification? |

## Tactique 1 : Abstraire les services communs

**Description**: Lorsque deux modules ont des services similaires, il faut abstraire le service en une forme plus générale.

**Justification**: Se connecter à la banque est similaire autant pour l’authentification régulière que pour celle avec Google. On peut donc créer un service de connexion à la banque.

## Tactique 2 : Encapsuler

**Description**: Introduire une interface explicite sur un module

**Justification**: Pour utiliser les différentes tactiques d’authentification, il faudra créer une interface commune pour une meilleure modifiabilité. Le serveur n’a pas à savoir avec quel type de compte l’utilisateur est connecté.

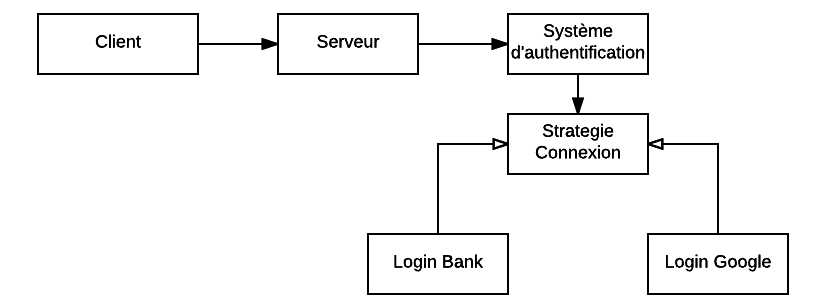
## Tactique 3 : Ré-usiner (Refactor)

**Description**: Tactique utilisée lorsque deux modules sont affectés par le même changement puisqu’ils sont (au moins partiellement) des dupliqués.

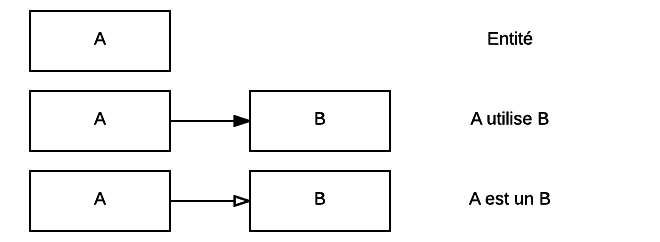
**Justification**: Il faudra ré-usiner le code afin d’extraire la logique d’authentification dans deux classes séparées utilisables grâce à une interface.

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

### Diagramme



### Légende



### Texte de description du diagramme

Le diagramme montre les différentes entités qui servent à authentifier un utilisateur. On utilise la tactique ‘**abstraire les services communs**’ pour abstraire la méthode de s’authentifier auprès du serveur. De cette façon il est aisé d’ajouter de nouvelles façons de s’authentifier ultérieurement. On **encapsule** chacune de ces méthodes de connexion dans des modules différents pour augmenter la modifiabilité. Au final, on a dû **ré-usiner** l’application pour insérer ce mécanisme de modifiabilité.

### Table de description des éléments du diagramme

| Client | Le client représente la machine que l’utilisateur utilise pour s’authentifier |
| --- | --- |
| Serveur | Le serveur représente l’entité qui recevra la requête d’authentification de la part du client. |
| Système d’authentification | Le serveur utilisera le système d’authentification pour effectuer l’authentification de l’utilisateur |
| Stratégie connexion | Abstraction des différentes méthodes permises pour s’authentifier. |
| Login Bank | Authentification avec un compte chez la banque |
| Login Google | Authentification avec un compte chez google. |

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

L’entité abstraite ‘Stratégie de connexion’ sert à abstraire les services communs des différentes manières de s’authentifier. Les entités ‘Login Bank’ et ‘Login Google’ représentent les deux façons permises de s’authentifier en les encapsulant. Le système d’authentification a été ré-usiné pour permettre ces modifications.

# Scénario : Par mesure de sécurité, nous voulons ajouter un champ «adresse courriel unique du demandeur» au niveau de la transaction.

| **Objectif d'affaires** | Assurer la sécurité des transactions |
| --- | --- |
| **Source** | Équipe de sécurité du système boursier (Développeurs) |
| **Stimulus** | Ajouter un nouveau champ à une transaction (adresse courriel unique du demandeur) |
| **Artéfact** | Code de traitement des transactions (base de données transactionnelles) |
| **Environnement** | Design Time |
| **Réponse** | Effectuer la modification  Tester la modification  Déployer la modification |
| **Mesure de la réponse** | Le temps de traitement pour chaque transaction doit rester le même.  L’effort pour effectuer les changements (jours/personne).  Coûts pour réaliser la modification. |
| **Questions** | 1. Combien de ressources sont nécessaires? 2. Faut-il développer un sous-système pour s’assurer de l’unicité des courriels enregistrés? 3. Quel format d’adresse courriel est toléré? |

## Tactique 1 : Encapsuler

S’assurer que tous les détails d’une transaction sont regroupées et accessibles de la même manière. Empêcher la modification de ces détails pour éviter tout conflit de traçabilité des transactions.

## Tactique 2 : Refactor

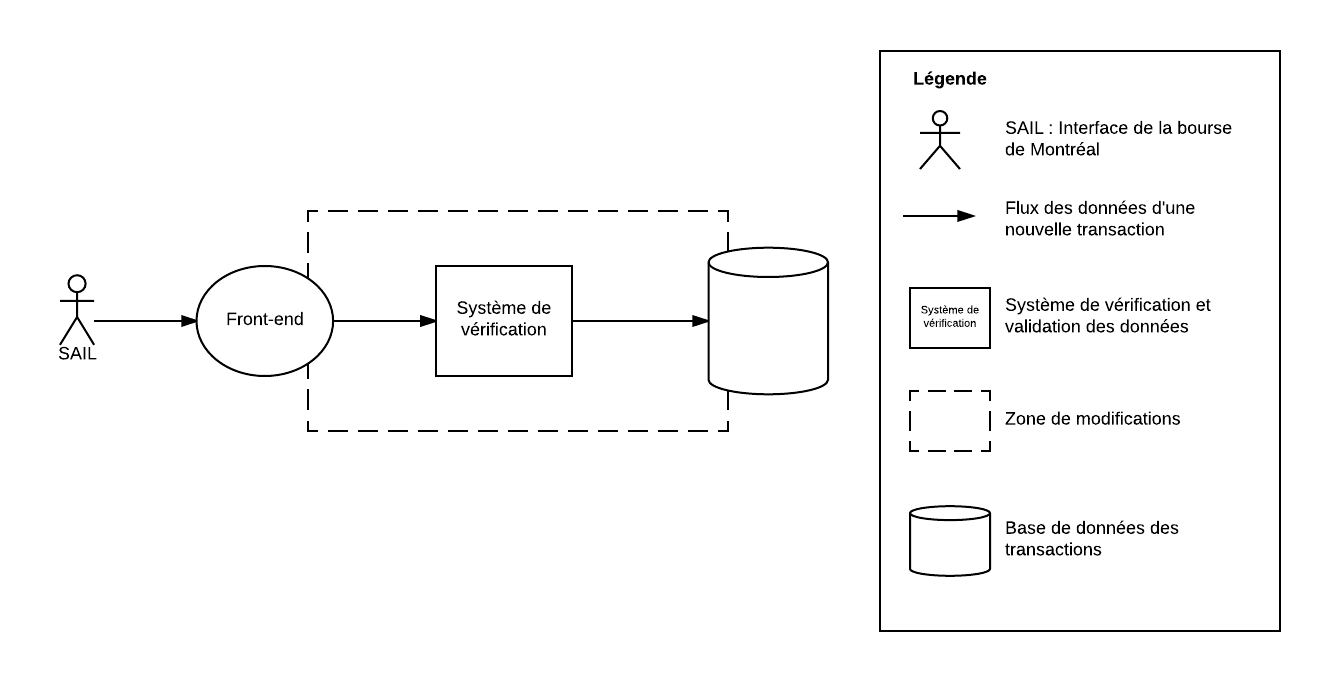
Effectuer les modifications nécessaires pour que les modules utilisant les transactions tiennent compte du nouveau champ. Si cette modification n’est pas faite, le nouveau champ ne sera pas utilisé par l’application.

## Tactique 3 : Restrict dependencies

Les modifications apportées à l'artefact ne devraient pas affecter les autres composantes du système. Avec cette tactique, nous diminuons les risques d’erreurs qui peuvent se produire une fois que la transaction est en traitement ou complétée.

## Vue architecturale

### Diagramme



### Texte de description du diagramme

Le diagramme présenté représente les modifications à faire dans le système boursier courant. Nous pouvons décomposer le travail à faire en trois partie. La première partie serait de modifier le front-end pour permettre à l’utilisateur de rentrer son adresse électronique. Deuxièmement, le nouveau système de vérification de données de transactions devra s’assurer que le nouveau champ “adresse électronique” est présent et valide (une requête à la base de données devra être faite pour s’assurer de la validité et unicité de ce champ). Troisièmement, la base de données devra avoir une colonne de type unique pour ce nouveau champ.

### Table de description des éléments du diagramme

| **Élément** | **Description** |
| --- | --- |
| Utilisateur SAIL | Partie prenante et utilisateur principal de l’accès Front-End. Il initialise les transactions à être vérifiées |
| Flux des données | Dans le flux des données de la transaction on retrouve tous les champs nécessaire pour compléter une transaction (incluant le nouvel champ unique) |
| Système de vérification | Module à implémenter qui permettra de vérifier si tous les champs sont inclus et valides. De plus, il sera capable de valider l’unicité du champ “adresse électronique” |
| Zone de modifications | Cette zone touches aux 3 éléments qui doivent être modifiés pour implémenter notre architecture de modifiabilité |
| Base de données des transactions | La colonne unique “adresse courriel” doit être ajoutée dans la table comprenant les transactions. |

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

Ce schéma présente bien les trois tactiques utilisées en modifiabilité. Tout d’abord, l’encapsulation est présentée dans ce schéma au niveau du flux de la transaction. Les données pertinentes d’une transaction sont sauvegardées dans cet objet et les champs requis de ce dernier ont une très grande modifiabilité. De plus, le nouveau système de validation est lié à la troisième tactique présentée dans ce document, la restriction des dépendances (« Restrict dependencies »). Puisque toutes les données sont validées et vérifiées à cet endroit, cet objet possède un très faible couplage, un élément important pour la modifiabilité. Finalement, la tactique « refactor » est présente dans tous les éléments de ce schéma puisque le front-end et la base de données des transactions devront inclure ce nouveau champ dans leurs considérations.

# Scénario : Fusion de systèmes boursiers

| Objectifs d'affaires | Fusionner deux systèmes boursiers sous une interface commune, ou bien abandonner un système et transférer les clients vers le système choisi |
| --- | --- |
| Source | Les développeurs de la solution retenue |
| Stimulus | * Réception d’une demande pour élargir les systèmes déjà mis en place * Expansion des compagnies participantes |
| Artéfact | Les interfaces utilisées dans les deux systèmes SAILS |
| Environnement | (conception/organisation) Design time |
| Réponse | Implémentation, tests et déploiement |
| Mesure de la réponse | * Temps de complexité : 3 semaines * Déployer les ajouts/modifications * Tester les ajouts/modifications |
| Questions | 1. Lequel des systèmes sera sauvegardé ? 2. Quelles solutions doit-on préférer (mesure de coût) ? 3. Quel sont les différences entre les deux systèmes boursiers? 4. Quand est-ce que la fusion peut-elle être faite ? |

## Tactiques

## Tactique 1 : Reporter l’instanciation - *Defer Binding*

Description : Une architecture possède un coût moins élevé si elle accommode les modifications tardives dans le cycle de vie d’un logiciel ou d’un système contrairement à une architecture qui force plutôt une modification plus tôt dans le cycle.

Justification : En rendant le système plus flexible, nous favorisons un coût moins élevé quant à la modifiabilité et la maintenabilité. C’est aussi plus facile d’ajouter des fonctionnalités et c’est beaucoup plus portable. En terme de fusion, la portabilité et la capacité d’adaptation sont des éléments clés à considérer puisque si un troisième système viendrait se joindre au nouveau système fusionner, alors la nouvelle fusion serait facilitée.

## Tactique 2 : Séparer en module - *Size of a module*

Description : Séparer les parties existantes en modules.

Justification : Cette tactique permettra de garder les deux logiciels intacts tout en les fusionnant. Ainsi, il sera beaucoup plus simple de faire les modifications. De plus, les coûts de modifications seront réduits.

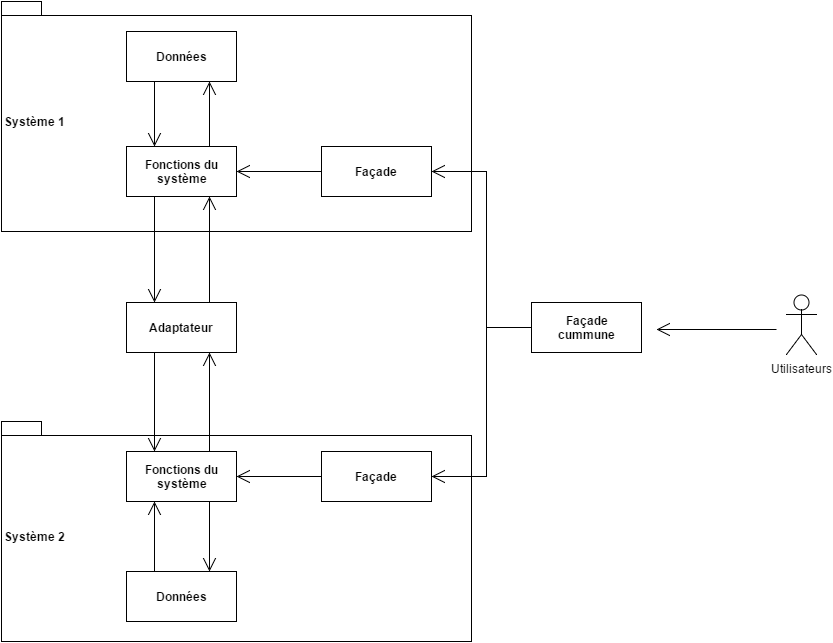
## Tactique 3 : (Couplage) Utiliser un intermédiaire - (Coupling) : Use an Intermediary.

Description : L’usage d’un intermédiaire brise des dépendances souvent jugées inutiles entre des éléments d’un système ou des modules. L’intermédiaire sera adaptée en fonction des dépendances à découpler.

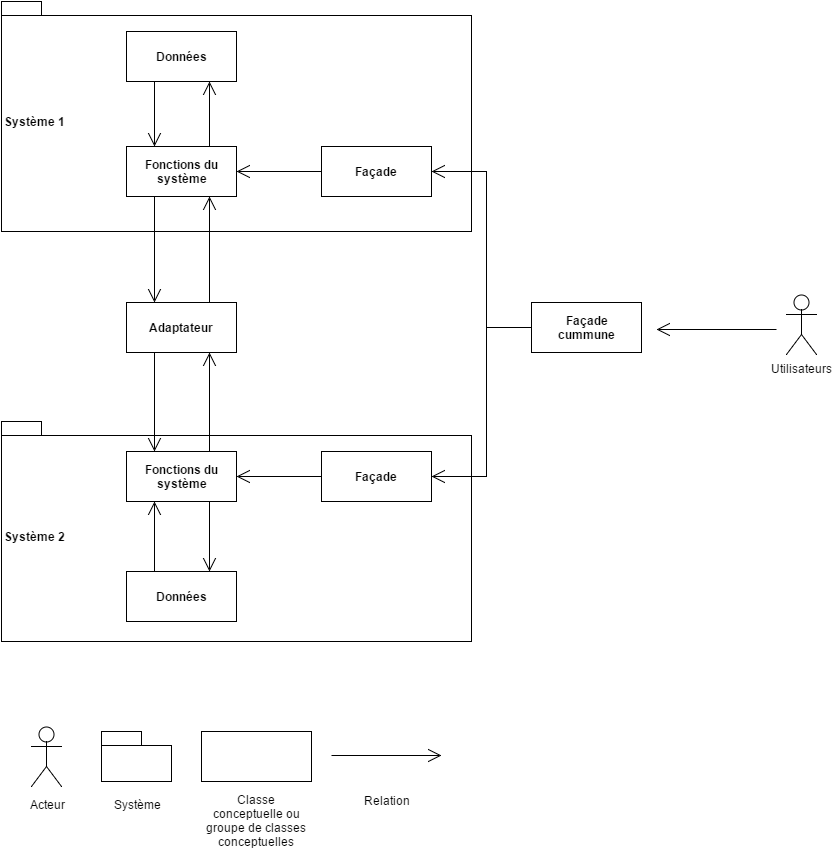
Justification : La réduction de couplage est un aspect important relié au portrait de la fusion que nous essayons de dresser. En réduisant le couplage, il est plus facile de comprendre le code en réduisant les liens de dépendances inutiles. Il est aussi plus facilement portable vers de nouveaux systèmes, mais aussi plus facilement modifiable et testable, car nous avons simplifié le code de manière générale.

## Vue architecturale

## Diagramme



## Légende



## Texte de description du diagramme

Dans ce diagramme, chaque système propose une interface afin de pouvoir être utilisé par un utilisateur tiers. Cet utilisateur peut soit être un acteur (humain) ou un autre système. Cette interface est sous la forme du patron de conception « Façade ». Dans le cas où l’on voudrait fusionner deux ou plusieurs systèmes, il est possible de créer une classe intermédiaire permettant de communiquer avec les deux systèmes en utilisant une seule interface. De cette façon, seules les classes de « façade » de chaque système auront besoin d’être « fusionnées » et non les systèmes tout entiers. L’utilisation d’une façade permet de rendre le système très portatif, ce qui lui permet d’être utilisé facilement par d’autres systèmes. Le patron adaptateur permet aux deux systèmes de communiquer entre eux dans le cas où ils voudraient partager leurs données par exemple.

## Table de description des éléments du diagramme

| **Élément** | **Responsabilité** |
| --- | --- |
| Système 1 | Systèmes boursiers des compagnies participantes |
| Système 2 |
| Données | Données appartenant à un système |
| Fonctions du système | Groupe de fonctions permettant de gérer les données du système |
| Adaptateur | Permet aux diverses fonctions de deux systèmes de communiquer entre elles par l’intermédiaire du patron adaptateur. |
| Façade | Permet à l’un des systèmes de pouvoir être utilisé par un acteur ou par un autre système |
| Façade commune | Unique point d’entrée qui joint entre les façades des systèmes afin de communiquer |
| Utilisateurs | Acteurs utilisant le ou les systèmes |

### 

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

| **Tactique** | **Description** |
| --- | --- |
| Defer Binding | Permet une plus grande adaptabilité suite à l’ajout de nouveaux systèmes externes |
| Size of a module | La séparation des modules en sous-modules permet la réduction les coûts de modification. |
| Coupling | Permet de retirer les dépendances entre les modules internes et les autres systèmes boursiers. |

## Description des relations entre les éléments

La vue architecturale (diagramme ci-haut) utilise des flèches bidirectionnelles. En effet, chaque module est en mesure de s’échanger des flux de messages.

# Scénario : Ajouter un module de caméra pour prendre une photo lors d’un échec de NIP

| **objectifs d'affaires** | Retracer un voleur potentiel de carte (Vues modules) |
| --- | --- |
| **Source** | Développeur |
| **Stimulus** | Ajouter une fonctionnalité |
| **Artéfact** | Code |
| **Environnement** | Run time |
| **Réponse** | Faire une modification |
| **Mesure de la réponse** | 3 semaines d’effort avec un budget de 8000$  Le temps d’ajout du module et le coût d’ajout des modifications. |
| **Questions** | 1. Quels sont les modules de caméra standards qui s’adaptent bien avec le GAB? 2. La photo doit-elle être envoyée directement aux autorités pour qu’ils soient notifiés ? 3. Faut-il offrir à l’utilisateur un maximum d’échecs de NIP avant de prendre la photo? 4. Par quel protocole doivent transiter les données ? 5. Est-ce qu’il faut réduire le facteur de qualité de l’image prise par la caméra pour réduire la taille de l’image et accélérer l’envoi ? |

## Tactique 1 : Split Module

**Description**: Raffiner le module en plusieurs petits modules divisés.

**Justification**: Cela devrait réduire le coût moyen des changements futurs. En l'occurrence le module de caméra qu’on va éventuellement ajouter pour prendre une photo lors d’un échec de NIP.

## Tactique 2 : Reduce Coupling (Encapsulate)

**Description**: Encapsuler le code du module de caméra qui va inclure des méthodes exposées via une interface de programmation applicative.

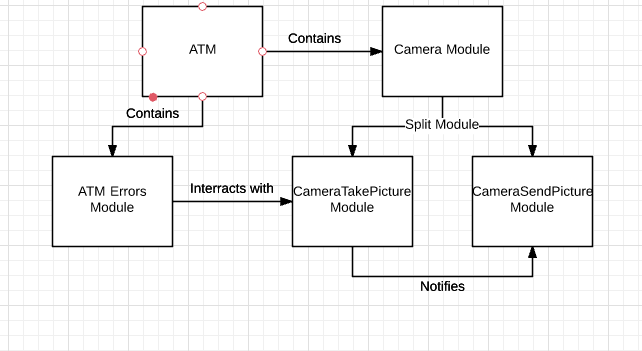
**Justification**: En faisant l’encapsulation du code du module de caméra, ça va permettre de modifier la vue architecturale sans affecter les autres modules du système.

## Tactique 3 : Increase semantic coherence

**Description**: Déplacer les responsabilités du module de caméra des autres composants du système.

**Justification**: Il sera plus cohérent et plus facile de séparer les responsabilités du module de caméra affecté.

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



### Diagramme

### Légende

Les flèches symbolisent les relations entre les différents module de l’application. Les noms des relations sont indiqués sur le diagramme

### Texte de description du diagramme

L’ATM contient différents modules (les modules nécessaires sont illustrés dans le diagramme, soit le Errors module et le Camera Module) afin de séparer l’impact des changements fait sur le module de la caméra. Dans le module de la caméra, des sous-modules sont créés en utilisant la tactique du split module, ce qui va permettre encore de réduire l’impact lors de la modification du module d’envoi de données ou bien du module de la prise de la photo. Le module ATM Errors notifie le module de la prise de la photo lors d’un certain nombre de NIP erronés

### Table de description des éléments du diagramme

| **Tactiques/Élements** | **Description** |
| --- | --- |
| Split Module | Le module de caméra est séparé en plusieurs modules afin d’en réduire l’impact/le coût des changements futur |
| ATM | Le module de l’ATM dans sa globalité |
| ATM Errors Module | Le module de la gestion des erreurs de l’ATM |
| CameraTakePicture Module | Le module qui va s’occuper de prendre une photo Et d’avertir le module d’envoie |
| Camera Module | Le module qui contient tout ce qui a rapport à la Camera. Issue de la tactique Encapsulating |
| CameraSendPicture Module | Le module de la gestion de l’envoie des photos |

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

Un développeur décide de modifier le nombre d’erreur de NIP requis pour envoyer des photos et aussi les destinataires desdites photos. Le module de Caméra est séparé en deux modules (Split module), ce qui permet de mieux cerner les problèmes lors de l’exécution des tests après l’implémentation des nouvelles modifications. Le module d’erreurs, lors d’entrée de NIP non valide, peut notifier le module de Caméra afin qu’il puisse prendre une photo du supposé malfaiteur. Ensuite, le module de prise de photo peut avertir le module d’envoie de la photo afin de faire les actions nécessaires et implémentées par le développeur.

# Scénario : L’équipe veut vendre le système à d’autre entreprise boursières qui ont déjà un système afin de gérer les transactions client.

| **objectifs d'affaires** | Le système doit être fonctionnel sans SAIL |
| --- | --- |
| **Source** | Utilisateurs finaux, développeurs, administrateurs de systèmes |
| **Stimulus** | Capacité à fonctionner indépendamment de SAIL |
| **Artéfact** | Code |
| **Environnement** | Au moment du conception de la solution. |
| **Réponse** | Faire et tester les modifications avant de faire le déploiement. |
| **Mesure de la réponse** | Vérifier que les modifications n’affecte pas les autres fonctionnalités et attributs de qualité via le résultat des tests. Il faut également faire le suivi des défauts décelés par les tests. |
| **Questions** | 1. Est-ce possible d’introduire mon logiciel de transaction client dans votre solution? 2. Est-ce possible de continuer à utiliser SAIL? 3. Est-ce que le programme est toujours cohérent? |

## Tactique 1 : Abstraction du frontend client (abstraction des services communs)

**Description**: Créer une interface pour le frontend client, qui serait implémenté par défaut par SAIL

**Justification**: Ceci permettra au logiciel de fonctionner avec différentes frontend pouvant remplacer SAIL, sans pour autant changer le fonctionnement du backend.

## Tactique 2 : Split module

**Description**: Séparer SAIL en différent modules.

**Justification**: Ceci permettra aux acheteurs de garder certaines fonctionnalités de SAIL, tout en leur permettant d’utiliser des modules personnalisés pour ces dernières.

## Tactique 3 : Réduire le couplage entre les modules (restreindre les dépendances )

**Description**: Réduction du couplage entre les différents modules avec une architecture en couche.

**Justification**: Afin de rendre les futurs modifications moins coûteuse en temps et argent, une architecture en couche permettrait de diminuer le couplage ce qui rendrait la modifiabilité plus facile.

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

### Diagramme et légende

### 

### Texte de description du diagramme

Pour commencer, le package SAIL comprenant les multiples modules de son système qui implémente tous les différentes interfaces des fonctionnalités. Par la suite, on a l’implémentation du client SAIL qui utilise les multiples fonctionnalités de son système en passant, bien sûr, par les interfaces de ces dernières.

Ensuite, le package FIX est séparé du système SAIL afin de pouvoir réussir notre objectif d’affaire. Le package Backend s’occupe de faire la liaison entre les deux systèmes et comprenant toute la logique nécessaire afin de bien coordonner les activités des système.

### Table de description des éléments du diagramme

| Nom | Description |
| --- | --- |
| SAIL | Package représentent le système SAIL |
| FIX | Package représentent le système FIX |
| Backend | Package représentent le backend du système |
| Client interface | Représente l’interface client du système SAIL |
| SAIL Client Implémentation | Implémentation de l’interface client afin que le backend puisse utiliser SAIL |
| Interfaces des fonctionnalités | Contient les multiples interfaces des packages représentant les fonctionnalités du système SAIL |
| Gestion des communications | Package gérant les communications entre le client et l’interface |
| Traitement de transaction | Package gérant le traitement des transactions du système SAIL |
| Facturation | Package gérant les différents composants d’une facturation |
| Journalisation | Package comprenant les composants de la journalisation du système |
| Opérations système | Package contenant les différentes parties des opérations effectués par le système outre les autres packages. |

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

Si on regarde la première stratégie, qui est l’abstraction du front end client, on peut voir son utilisation avec l’élément Client interface qui représente l’interface du client qui doit être implémenté par le système SAIL afin de pouvoir faire fonctionner l’application sans ce dernier. L’implémentation dans le système SAIL est représentée par l’élément SAIL Client Implémentation.

Ensuite, avec la deuxième stratégie, qui constitue la séparation des différentes parties du système SAIL en modules, on peut voir son utilisation avec les différents éléments compris dans le package SAIL (Journalisation, Opération système, Facturation, Traitement de transaction et Gestion des communications). Ceci permettant l'utilisation de certaines fonctionnalités sans être dépendant d’autrui.

Pour la dernière stratégie, on utilise celle de réduction de couplage afin de justifier notre choix d’architecture en couche. On voit l’utilisation de ce style architectural à travers les différents éléments et les différentes relations de l’application. Comme on peut voir dans le diagramme, il n’a aucune relation cyclique entre les différents packages. Ce qui représente effectivement une architecture en couche.

# Scénario : L’entreprise voudrait ajouter une nouvelle devise

| **Objectifs d'affaires** | L’entreprise veut permettre aux utilisateurs de l’application de choisir en quel devise ils veulent effectuer leur transactions |
| --- | --- |
| **Source** | Product Manager |
| **Stimulus** | Ajout d’une nouvelle devise |
| **Artéfact** | Configuration, code |
| **Environnement** | Runtime, initiation time |
| **Réponse** | Modifier l’affichage |
| **Mesure de la réponse** | Doit supporter minimum la devise du Canada, des États-Unis, de l’Euro, et de Hong Kong |
| **Questions** | 1- Comment on va implémenter l’internationalisation  2- Comment on procède au niveau des conversions (dynamique) ?  3- Comment va-t-on garder l’historique des conversions de devises?  4- Devrions nous afficher la devise d’origine de l’action en plus de la devise désiré?  5- Stocke t-on les montants en une devise pour ensuite les reconvertir pour l’utilisateur ? |

## Tactique 1 : Abstraction des devises

**Description**: Une classe abstraite “devise” qui définit ce qu’une devise et en général.

**Justification**: L’abstraction des devises nous permet d’ajouter de nouvelles devises au besoin. Chaque devise a son propre taux de change et donc concrétise l’abstraction de la devise. Cette abstraction rend donc le logiciel modifiable au niveau des ajouts de devises.

## Tactique 2 : Généralisation du module

**Description**: Le module de conversion ne suppose pas quel sera le moyen d'obtentions des taux de conversion courant

**Justification**: Si pour une raison quelconque l’API que le module de conversion utilisé n’est pas disponible, il est possible de spécifier d’autres sources de données (autres API, cache, défauts) en ayant aucun impact sur le code dépendant du module

## Vue architecturale

### DiagrammeDevoir 2 - Modifiabilité

### Description du diagramme

Dans le diagramme ci-dessus, nous avons deux vues. Tout d’abord, nous avons un module business logic qui contient tous les composantes du logiciel relié aux opérations et aux définitions financières. Dans ce module, on retrouve le module de devise ainsi qu’un module de conversion. Ces modules communiquent entre eux pour la conversion des devises. Ensuite, notre deuxième vue représente le contenue du module devise. Celui-ci contient une abstraction de la classe devise. Cette abstraction est redéfinie en 4 devises concrètes. Cette architecture permet donc de rajouter de nouvelles devises à n’importe quel moment. Pour terminer, le module de conversion s’occuperait d’offrir une abstraction pour le client qui lui ne fera que communiquer avec l’API offert par le système pour avoir accès à la conversion des devises.

### Table d’éléments

| Module | Ces composantes appartiennent à la vue architecturale en modules. Elles permettent de représenter plusieurs parties du système à différents niveaux. |
| --- | --- |
| Classe Abstraite | Cette classe crée une abstraction de la devise afin que le système soit en mesure de gérer tous les devises de la même manière mais que chaque devise soit définie en fonction de leur taux de conversions. |
| Classe Concrète | La concrétisation des devises est définie par leur taux de conversion. |

### Liens entre les vues et les tactiques

La tactique de l’abstraction des devises est représenté par le module “Devises”. Celui-ci contient une classe abstraite qui permet de diminuer le couplage entre les devises concrètes et les autres modules du système en offrant un interface commun. De plus, cela permet d’ajouter ou d’enlever plusieurs différentes devises au système en diminuant l’impact sur les autres modules. L’ajout devrait être transparent, puisque les autres modules n’ont pas besoin de connaître les types concrets.

Le module “Conversion” utilise la tactique de généralisation. Il offre plusieurs façons de convertir une devise vers une autre en utilisant plus d’un API. Si un de ceux-ci n’est pas disponible, un autre peut être utilisé à sa place.

# Scénario : Effectuer des transactions avec différentes bourses internationales qui utilisent des protocoles de communication différents

| **objectifs d'affaires** | Effectuer des transactions avec différentes bourses internationales qui utilisent des protocoles de communication différents. |
| --- | --- |
| **Source** | Développeur qui est chargé de faire la modification |
| **Stimulus** | Rendre le système plus polyvalent en ajoutant l’option de supporter d’autres protocoles de communications aisément. |
| **Artéfact** | Front end access |
| **Environnement** | Période de design d’une nouvelle fonctionnalité |
| **Réponse** | Évaluer les options pour effectuer la modification  Documenter les modifications nécessaire  Effectuer les modifications  Tester les modifications  Déployer les modifications |
| **Mesure de la réponse** | Il est possible d’implémenter un nouveau protocole de communication sans modification de code |
| **Questions** | 1. Combien se protocoles supplémentaires devraient être supportés? 2. Est-ce que certain protocoles vont être bloqué? |

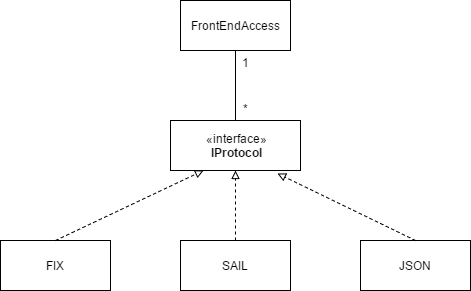
## Tactique 1 : Encapsulation

**Description**: En créant une interface explicite qui définit les actions possibles, on peut découpler deux modules au profit d'un couplage avec une interface abstraite indépendante de toute implémentation.

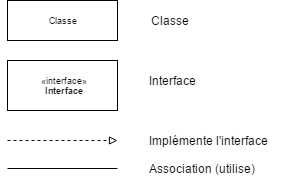
**Justification**: En encapsulant les différentes transactions possibles dans une interface indépendante des protocoles de communication, on peut supporter tous les différents protocoles sans avoir à modifier le reste du système puisque ce dernier est uniquement couplé avec l'interface abstraite.

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

### Diagramme



### Légende



### Texte de description du diagramme

FrontEndAccess utilise des modules de communication qui implémentent l'interface IProtocol.

### Table de description des éléments du diagramme

| **Élément** | **Description** |
| --- | --- |
| FrontEndAccess | Le module de communication qui utilise les différents protocoles. |
| IProtocol | L'interface de protocole. Paradigme orienté objet. |
| FIX, SAIL et JSON | Exemples de protocoles qui implémentent IProtocol. |
| Lien entre FrontEndAccess et IProtocol | FrontEndAccess utilise n'importe quelle quantité de protocoles de communication qui implémentent IProtocol. |
| Lien entre IProtocol et FIX, SAIL et JSON | FIX, SAIL et JSON sont des exemples de protocoles de communication qui implémentent IProtocol. |

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

La tactique d'encapsulation est concrétisée avec IProtocol, qui est une interface orientée objet. Cette interface définit les différentes responsabilités et actions que chaque protocole doit implémenter. FIX, SAIL et JSON sont des exemples de modules qui implémentent leur protocole de communication respectif en cachant leurs détails d'implémentation grâce à l'encapsulation. FrontEndAccess est donc découplé des protocoles spécifiques et il n'est pas nécessaire de le modifier pour supporter un nouveau protocole.