



Le génie pour l'industrie

LOG430

# Rapport de laboratoire

---

9 Octobre 2016

**Laboratoire** 01

**Etudiants** Benoit Lapointe <[benoit.lapointe@etsmtl.net](mailto:benoit.lapointe@etsmtl.net)> - LAPB04057905

Guillaume Larente - LARG23089406

**Cours** LOG430

**Session** AUT 2016

**Groupe** 02

**Professeur** Benoit Galarneau

**Chargé de Lab** Antoine Grenier

## Introduction

Durant ce laboratoire, nous devons améliorer l'architecture d'un guichet automatique bancaire (GAB). Le GAB était très mal conçu et comportait de nombreux défauts d'implémentations. Il y avait beaucoup de cycles, ce qui en faisait un code brouillon, qui ne répondait à aucun attribut de qualité au niveau de son architecture.

Le but de ce laboratoire consistait donc à proposer une nouvelle architecture afin que le logiciel soit plus facilement testable, modifiable et qu'on puisse en différencier les structures en plusieurs couches.

Dans sa première partie, ce rapport présente les tests et l'effort de développement que nous avons déployé pour atteindre nos objectifs. Dans sa deuxième partie, nous y présentons nos résultats d'analyse préliminaire du système suivi de l'architecture finale qui est une structure en couche. Cette architecture est accompagnée de la justification de nos choix en terme d'attributs de qualité.

## Première partie: description sommaire de votre implémentation

Cette première partie vise à décrire l'état du système au niveau de son implémentation. On y définit d'abord des tests pour se prémunir contre d'éventuelles régressions lors du travail de développement. Par la suite, on y décrit l'expérience liée à la réimplémentation du code de la nouvelle solution.

### Implémentation des tests

La consigne principale de cette tâche était de tester seulement les quatre principales fonctionnalités du système de GAB, qui sont: le dépôt, le retrait, le transfert entre comptes et l'interrogation de solde.

Avec la commande «`mvn exec:java`» on lançait facilement la simulation et on pouvait exécuter les quatre opérations l'une à la suite de l'autre. Cette méthode formait notre base de tests fonctionnels qu'on exécutait à chaque nouvelle étape d'implémentation.

Du côté des test unitaires, ils devaient s'exécuter automatiquement grâce à la commande «`mvn test`» et ils ne devaient pas échouer. Nous avons donc créé la classe de test (`AtmTestCase`) qui inclut quatre tests unitaires distincts. Ainsi, chaque test vérifie une seule fonctionnalité et chaque fonctionnalité possède deux aspects. Le premier aspect est le succès de l'opération et le deuxième, le solde du compte touché par le test. Le plus grand défi de cette tâche consistait à éviter de passer par le GUI pour exécuter les tests. Un certain travail de navigation dans le code a donc été nécessaire pour s'assurer qu'on testait bien chaque aspect de chaque fonctionnalité.

### Implémentation de la solution

Puisqu'il s'agissait de notre premier laboratoire avec ce programme, notre méthode de travail a requis un certain temps d'adaptation afin de bien comprendre le logiciel et de bien suivre toutes les dépendances.

Pour certaines classes, c'était très facile. Il suffisait d'enlever la dépendance et de passer les valeurs spécifiquement requises en paramètres, ce qui retardait l'instanciation. Cependant, dans un autre cas, c'était plus complexe. Il fallait d'abord enlever les attributs indésirables, créer des interfaces et repenser un peu la hiérarchie des classes.

Dans le cas le plus complexe, nos opinions divergeaient, puisque l'effort de développement prévu impliquait beaucoup de classes dont les dépendances étaient multiples et transitives. Nous avons donc dû choisir la meilleure solution, c'est-à-dire celle qui nous donnait le meilleur équilibre entre les attributs de qualité demandés et l'effort de développement nécessaire.

## Deuxième partie: analyse architecturale

Cette deuxième partie sert à effectuer une analyse de l'architecture avant et après les changements attribuables à ce laboratoire. On y identifie d'abord les cycles grâce à la matrice de dépendances tout en l'analysant en terme d'attributs de qualité. Ensuite, on y présente la solution de la structure en couche adoptée, une vision essentiellement statique, des nouvelles nomenclatures tout en y expliquant chaque choix.

### Matrice de dépendance

L'étude de la matrice de dépendance nous permet d'analyser la structure générale des dépendances entre les différents modules du GAB. Voici une compréhension matricielle préalable du système qu'on nous a présenté dans l'énoncé.

### DSM Report - all\_packages







		1	2	3	4	5	6
 edu.gordon (1)	1	x					
 ..gordon.atm.transaction (5)	2		x			1C	1C
 ..du.gordon.atm.physical (8)	3		11C	x		10C	
 edu.gordon.simulation (14)	4	1		8C	x		
 edu.gordon.atm (2)	5	1	10C	2C	1C	x	1C
 edu.gordon.banking (7)	6		29C	11C	13C	1C	x

Figure 1 État initial illustré par un DSM

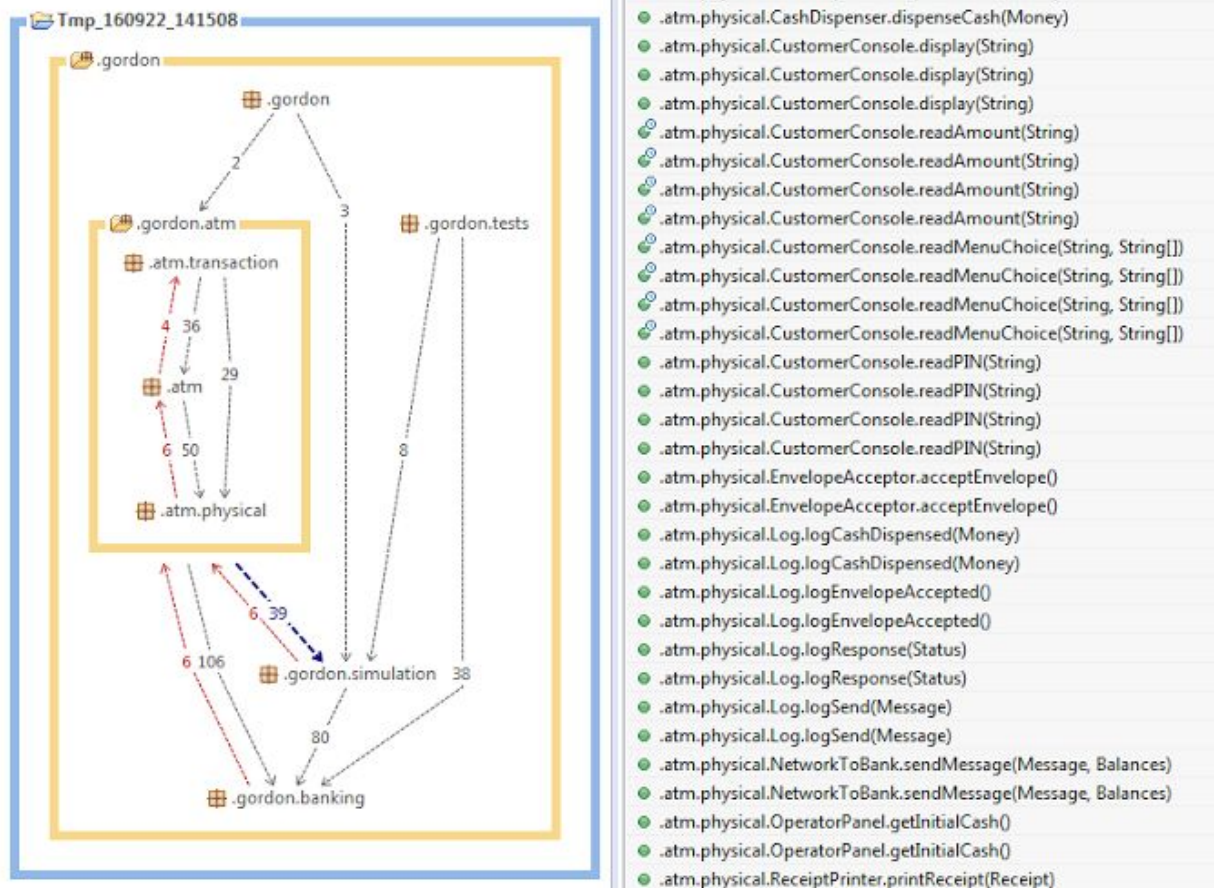
Légende 1 n=Nombre de dépendances du paquet col vers le paquet ligne; C=Dépendances cycliques

Selon une analyse formelle de la matrice, on constate que les paquets atm et physical sont le lien direct entre l'utilisateur et le GAB. Ils formeraient donc, à notre avis, la couche supérieure et ne devraient pas être connus d'aucun autre module, sauf de la simulation qui pourra dépendre de n'importe quelle autre couche. Par contre, on remarque 9 (8 + 1) dépendances de la couche supérieure vers la simulation, ce qui produit des cycles et nuit à la séparation des préoccupations et à sa portabilité. En effet, il serait impossible d'installer le code système de l'ATM sans en installer aussi la simulation.

De même, tous les modules, excepté la racine (ATMMain), sont en dépendances cyclique entre eux. Les deux prochains niveaux de modules (niveau 2 et 3) seraient les niveaux transaction et banking.

Malheureusement, les modules transaction et banking possèdent 22 dépendances (10+11+1) vers la couche supérieure et le module banking en possède une vers transaction. Toutes ces dépendances forment des cycles qui rendent l'application très peu modulaire et, par conséquent, en font une application très difficile à tester.

Voici une vue du code original de notre application du GAB créée par le plugin STAN.



**Figure 2** État initial détaillant les dépendances illustré par STAN

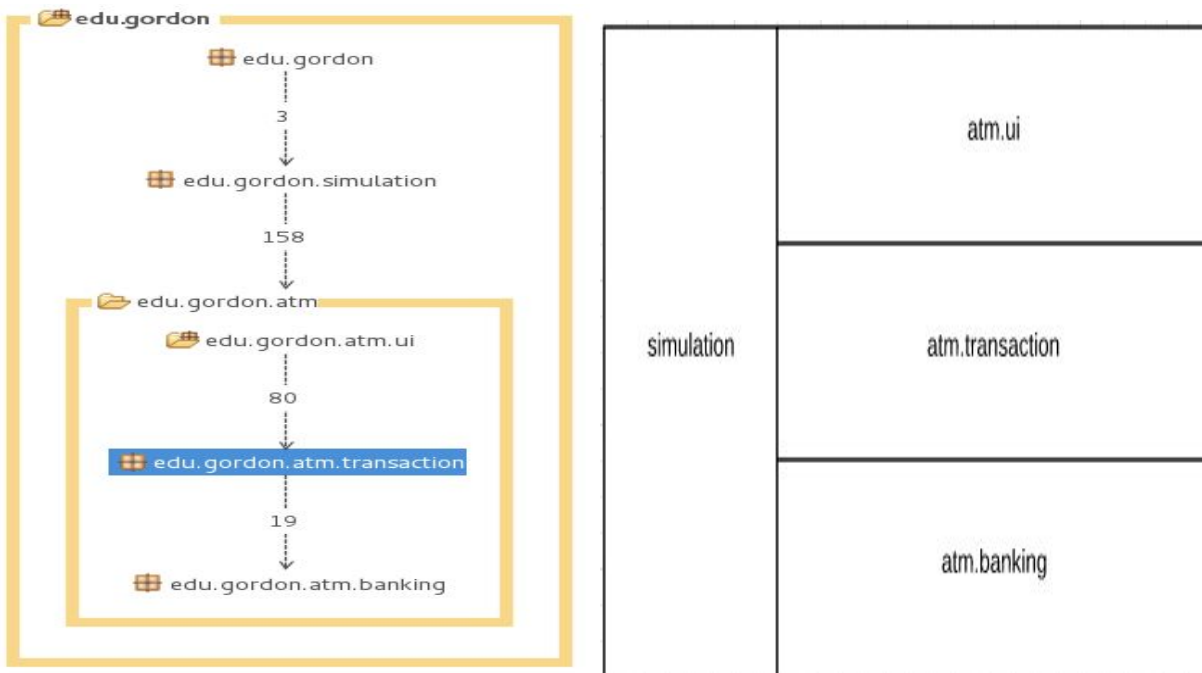
**Légende 2** ->=dépendance normale; ->=dépendance cycliques

Le nombre de cycles diffère du DSM, mais leur nature reste la même. Le plugin STAN nous permet d'analyser les cycles en groupes ou individuellement et nous permet de mesurer l'impact de nos changements en temps réel. Il constitue donc un net avantage sur le DSM en terme de visualisation statique et c'est pour cette raison qu'on s'est servi de STAN presque exclusivement lors du développement.

## Vue architecturale

L'objectif de ce laboratoire était d'atteindre une structure en couche pour l'architecture du GAB. L'essentiel de cette tâche consiste à bien regrouper les responsabilités du système bancaire tout en brisant ses cycles.

Une compréhension sommaire d'un système bancaire a guidé notre raisonnement de séparation des modules. Cette différenciation s'illustre par un chemin à sens unique qui débute à l'utilisateur du guichet et se termine à la banque. En réorganisant le tout conformément à notre logique, on a obtenu une nomenclature claire et uniforme qui permet de regrouper les modules ou certains cycles dans une même couche. De cette façon, on a réussi à minimiser la charge de travail, même si elle s'est un peu multipliée lors du traitement de dépendances transitives. Voici la structure en couche que nous avons adopté.

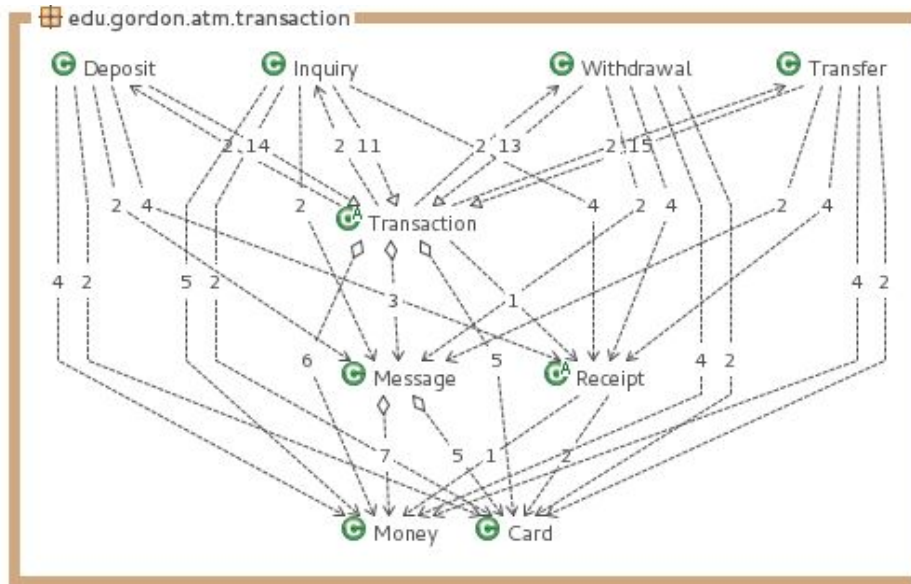


**Figure 3** Vue en couche de la nouvelle architecture

**Légende 3** ->=dépendance normale

L'utilisateur étant l'initiateur de toutes les actions, on le situe au sommet de la hiérarchie du GAB (`atm.ui`) et plus on se rapproche des données bancaires plus on touche au niveau le plus bas (`atm.banking`). Normalement il y aurait eu une couche de données à la toute fin, mais ce n'était pas nécessaire dans le cadre de ce laboratoire puisqu'il n'y avait aucune base de données. Il n'y avait que des structures de données de tests liées à la simulation.

La couche intermédiaire (atm.transaction) regroupe toutes les opérations qu'un utilisateur peut effectuer ainsi que tous les outils de transaction.



**Figure 3** Vue des dépendances des classes du paquet transaction

**Légende 3** ->=dépendance normale; C=Classe

Le paquet original regroupait les quatre opérations d'un GAB, soit les Transactions de retrait (Withdrawal), de transfert (Transfer), de dépôt (Deposit) et d'interrogation de solde (Inquiry). Nous avons d'abord transféré les fonctionnalités liées à performTransaction vers Session, car elles caractérisaient mieux le déroulement de la session qu'une transaction. Cette action a réglé 90% du problème cyclique entre atm.ui et atm.transaction.

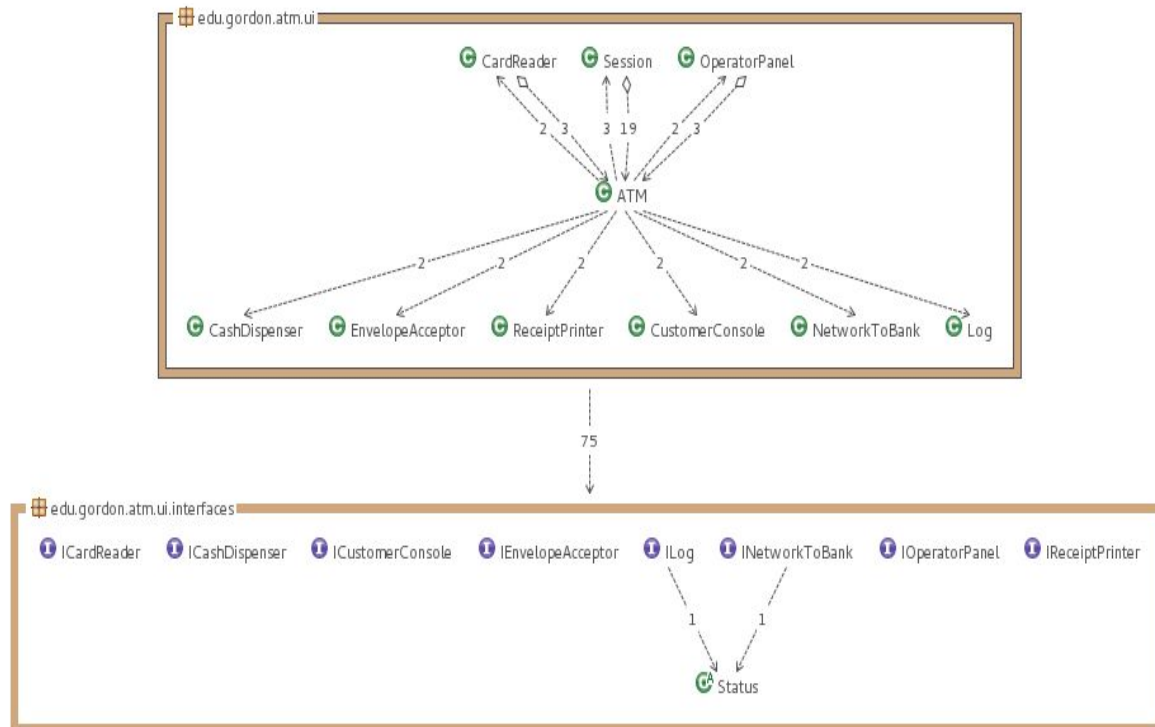
Nous y avons ensuite ajouté les outils de transaction (Message, Receipt, Card, Money), qui sont fortement liés aux transactions et indépendants des composantes physique du GAB. Ces outils ont donc été transférés de atm.banking vers atm.transaction. Ce qui réglait aussi 90% du problème entre atm.banking et atm.transaction et entre atm.banking et atm.ui.

Nous sommes conscient que dans le monde réel, avec les conversions de monnaie, il aurait été problématique d'extraire Money de atm.banking. Heureusement, il ne s'agissait pas d'une contrainte de ce laboratoire et comme ce changement simplifiait notre travail, nous avons choisi cette option. La couche banking ne contient donc plus que AccountInformation et Balances qui sont conceptuellement indissociables de la banque.

L'application prévoit aussi un code de simulation et celui-ci devait être invisible au code de l'atm. Il pouvait toutefois utiliser n'importe quelle couche. La couche simulation a été extraite du paquet atm et est maintenant utilisée en nacelle (**Figure 3**). La tâche reliée à cet objectif consistait à éliminer le cycle entre

atm.ui et simulation. Le paquet atm a été renommé en atm.ui et toutes les classes présentes dans atm.physical y ont été déplacées.

Ensuite, nous avons déterminé que Simulation hériterait de ATM, ce qui est logiquement vrai et qui permet à simulation de ne plus agréger ATM, mais seulement les composantes physiques simulés par l'entremise d'interfaces (polymorphisme). Ces interfaces sont toutes situées dans le paquet atm.ui.interfaces.



**Figure 4** Vue sur la couche ui et ui.interfaces

**Légende 4** ->=dépendances normales; <->=agrégation; C=Classes; I=Interface

Les interfaces sont aussi implémentées par les classes physiques, mais tout le code qui était relié à la simulation a été transféré. Normalement, il aurait dû être remplacé par le code de la machine physique, mais il nous est impossible de la représenter sans connaître l'API du matériel du guichet. Nous avons donc ajouté des exceptions mentionnant l'absence de support pour ces fonctionnalités. Outre tout ce transfert de code, seule la classe SimCustomerConsole a été créée et ajoutée au paquet simulation pour contenir le code lié à l'écran et au clavier bancaire. Par la suite, le singleton dans simulation a dû être retiré, ce qui a relevé de nouvelles dépendances transitives dans simulation qui ont dû être traitées.

Grâce à toutes ces actions, la différenciation entre atm.simulation et atm.ui est maintenant beaucoup plus explicite et nette, ce qui en fait un code plus modulaire et plus facile à tester. Le fait que les cycles résultants ont été relativement simples à traiter par la suite nous convainc de la justesse de notre travail de réorganisation.



## Considérations architecturales

Après toutes les modifications architecturales, le code du GAB est beaucoup plus facilement maintenable. L'interface utilisateur est séparé de la simulation et les liens de transaction vers l'interface utilisateur ont été inversés. De cette manière, puisque les liens pointent seulement vers le bas de cette architecture en couche sans pont, on peut modifier l'interface utilisateur sans aucun problème. On pourrait même remplacer l'interface utilisateur par un autre type d'interface. Par exemple, on pourrait ajouter un téléphone intelligent et, en principe, tout fonctionnerait encore. De plus, en enlevant les dépendances vers l'interface utilisateur, le logiciel est beaucoup plus facilement testable. En effet, il sera plus facile de tester une classe de la couche transaction ou banking puisqu'elle n'appelle plus aucune classe de l'interface utilisateur.

Grâce à notre environnement de développement qui est Eclipse, nous avons activé des options de réusinage automatique de code et des actions de sauvegarde, ce qui a aussi grandement amélioré la lisibilité du code.

## Conclusion

En résumé, l'architecture de ce logiciel a grandement été améliorée. Nous l'avons séparée en trois couches et nous avons séparé l'interface utilisateur de la simulation. De plus, l'interface utilisateur utilise maintenant beaucoup mieux la couche transaction qui, elle, se sert uniquement de la couche banking. La couche simulation en nacelle utilise beaucoup mieux toutes les autres couches, sans cycles. Malgré toutes ces modifications, l'application est toujours pleinement fonctionnelle et tous nos tests passent à 100%.

La problème potentiel de ces modifications réside dans des cas de conversion de devises au niveau des informations bancaires. Il faudrait nécessairement remettre la classe Money dans banking et rechercher une solution qui retarderait son initialisation pour éviter que atm.ui l'utilise directement.

Malgré cela, on peut affirmer avec assurance que cette application s'est nettement améliorée et qu'il s'agirait d'un changement mineur en comparaison avec le travail d'élimination des cycles et de différenciation des couches qui a été effectué. Ce qui confirme bien la grande amélioration de sa modifiabilité.