LOG430

Laboratoire 3

À

El Kassis, F.

PAR

Samy Lemcelli – LEMS26019103

Christopher Larivière – LARC24019005

Olivier Lemelin - LEMO18099102

Matthieu Fortier - FORM18048904

Remise: 10 juillet 2013

Table des matières

Introduction

Lors de ce laboratoire, il fut donné à l'équipe d'améliorer un système préexistant employant le style «invocation implicite » afin de lui ajouter diverses fonctionnalités prédéfinies. Ainsi, ce rapport de laboratoire a pour objectif de définir les changements apportés au système ainsi que de discuter les impacts que ces modifications ont eus sur l'architecture du projet.

A. Vues pertinentes du système

Diagramme de classes du système original

Tout d’abord, nous avons modifié un système qui consiste à attribuer des livraisons à des chauffeurs. Celui-ci a été conçu selon le style architectural d’invocation implicite. Le diagramme ci-dessous indique les liens entre chaque classe du système initial. Nous pouvons remarquer que la classe Communication adopte des liens entre plusieurs classes, puisqu'elle est la classe Parent de tous les modules pouvant envoyer et recevoir des signaux.

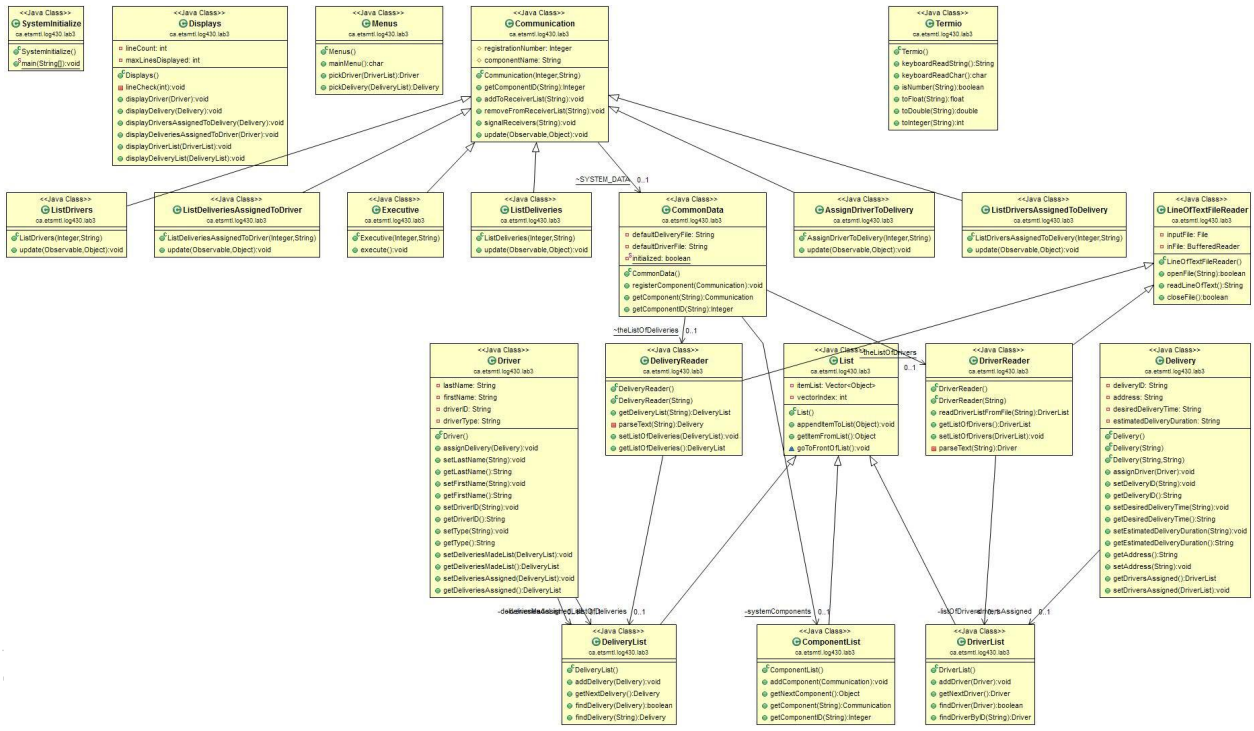

Illustration A: Diagramme de classes du système original

Légende :

Classe :

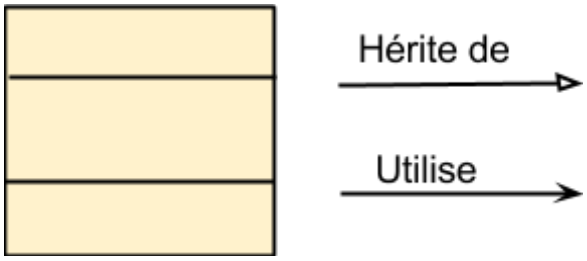

Illustration B: Légende du système original

Diagramme composant et connecteur

Dans le cadre de ce laboratoire, des ajouts et des modifications ont été nécessaires afin de répondre aux nouvelles exigences. Ainsi, des nouvelles classes ont été créées : *DeliveriesAlreadyAssignedToDriver, DeliveriesUnassigned, DeliveryAlreadyAssigned, DriverSchedule*. Aussi, les classes suivantes : *Menu, Display, SystemInitialize, Communication, Executive* ont été modifiés. Dans le diagramme C&C qui suit, l'équipe a affiché seulement les composantes importantes du système. De plus, il est important de souligner que les données partagées représentées par la classe *CommonData* sont utilisées par l'ensemble des composantes.

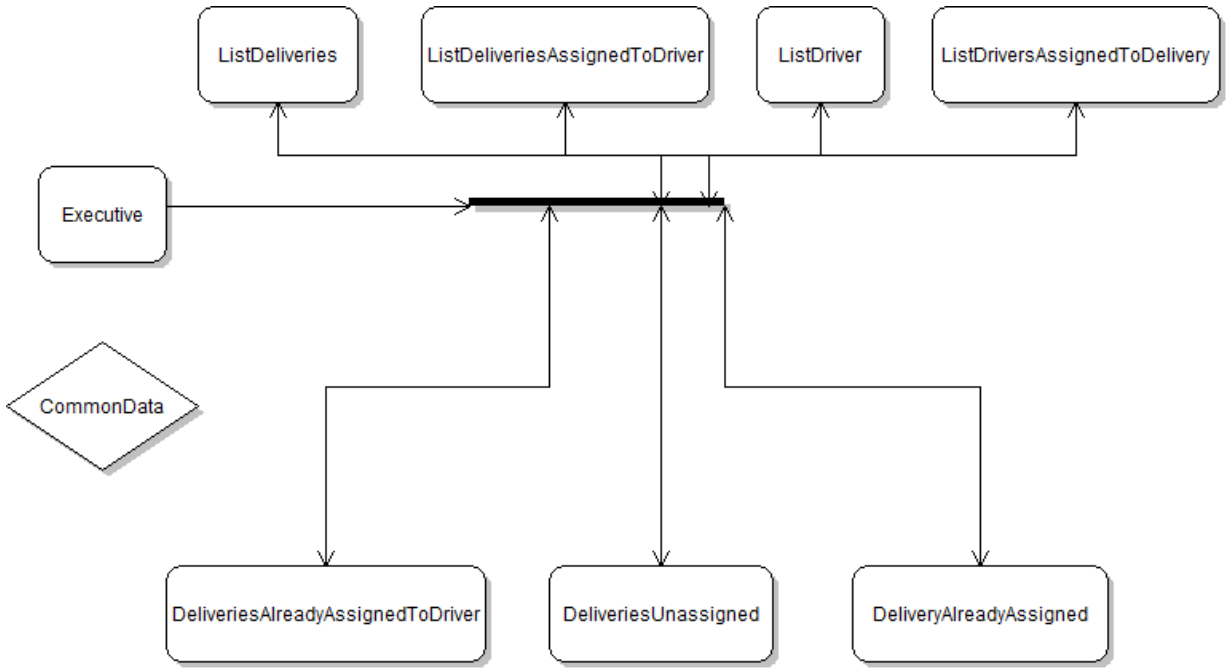

Illustration C: Diagramme composants et connecteurs

Légende :

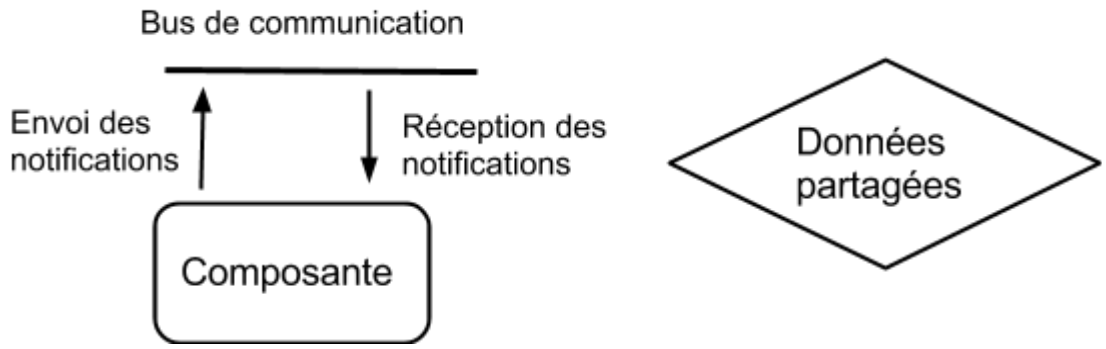

Illustration D: Légende

Diagramme de classe final

Tel que mentionné plus haut, le système déjà implémenté est de type invocation implicite, ce qui implique que chaque composant est indépendant de la file d'exécution. Le bus de communication a comme objectif de donner

le droit à chaque composant d'envoyer ou de recevoir des notifications. De plus, la classe *CommonData*, qui est utilisée par tous les composants du système, contient les données partagées nécessaires à son bon fonctionnement. Entre autres, elle localise les fichiers deliveries.txt et driver.txt. La classe *Executive* est une autre classe qui joue un rôle dominant dans le principe d'invocation implicite puisqu'elle diffuse les événements aux autres composants.

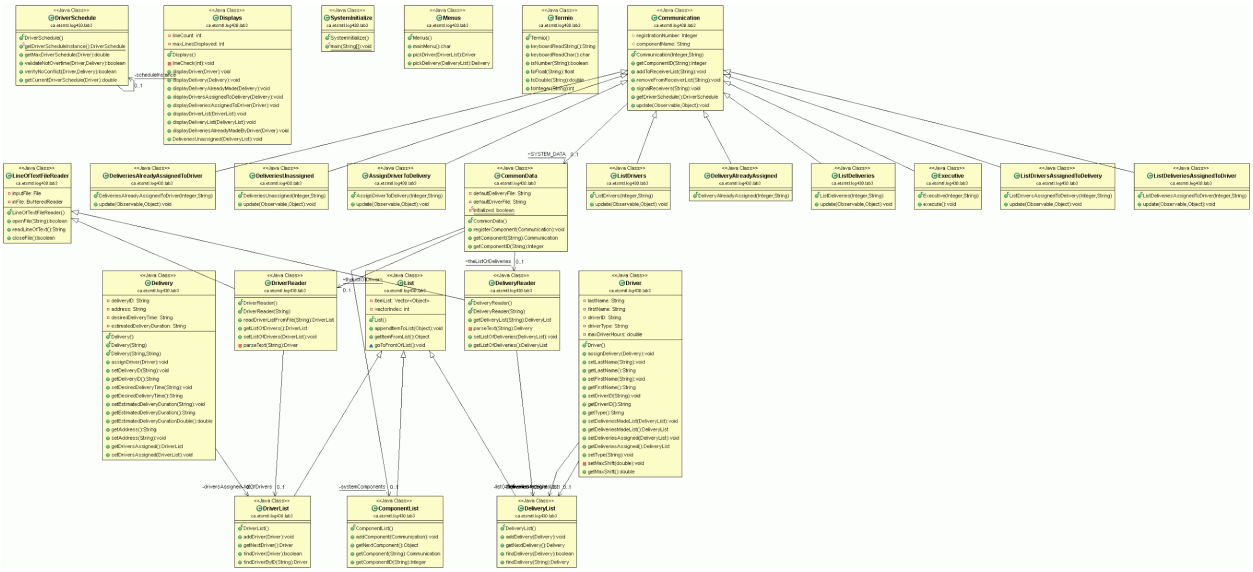

Illustration E: Diagramme de classes du système final

Légende :

Classe :

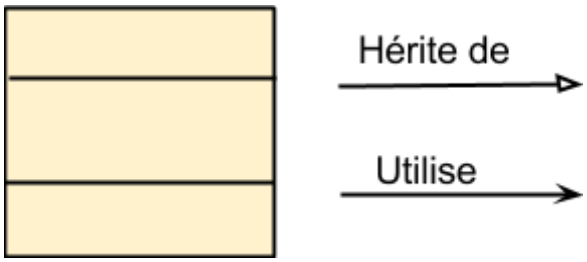

Illustration F: Légende

B. Déviation du système

Le système produit dévie du style «invocation implicite» sur un nombre minimal de points. Tout d'abord, la classe DriverSchedule, qui est une classe utilitaire visant à gérer les horaires des conducteurs, est utilisée par quelques modules communicants, ce qui crée un certain couplage commun. De ce fait, on externalise une partie du traitement des données de quelques modules, ce qui semble aller à l'encontre du fait que les modules, jusqu'à maintenant, étaient complètement indépendants. Ensuite, on peut observer la méthode *getDriverSchedule* à l'intérieur de la classe Communication. On assigne alors une fonctionnalité de traitement de données à une classe de gestion des signaux, une responsabilité qui jusqu'à maintenant ne lui était pas assignée.
Par contre, ces ajouts permettent un accès plus simple aux données des horaires, ce qui simplifie grandement le traitement à effectuer pour les modifications à apporter. Au court terme, ces ajouts permettent d'obtenir rapidement et aisément un accès aux données nécessaires malgré une légère entorse au style en invocation implicite.

C. Comparaison des matrices de dépendances originales et finales

IMAGE DSM ORIGINALE

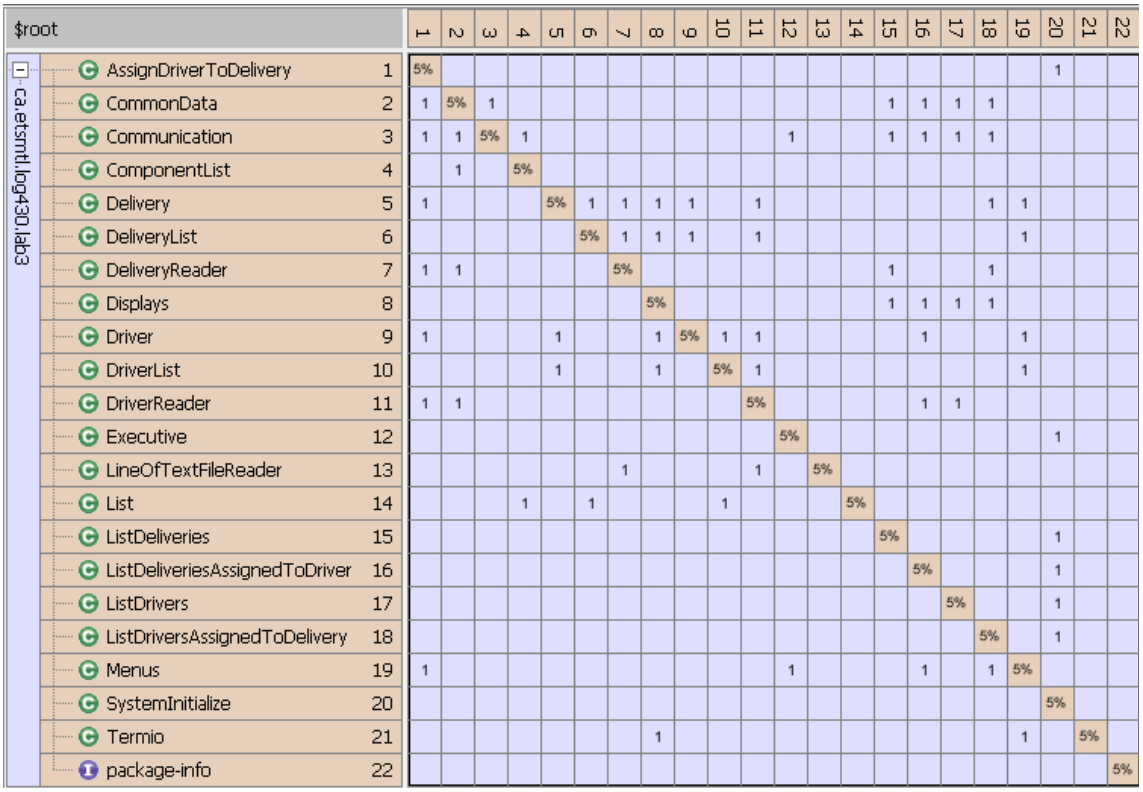

Illustration G: Image du DSM Original

IMAGE DSM FINALE

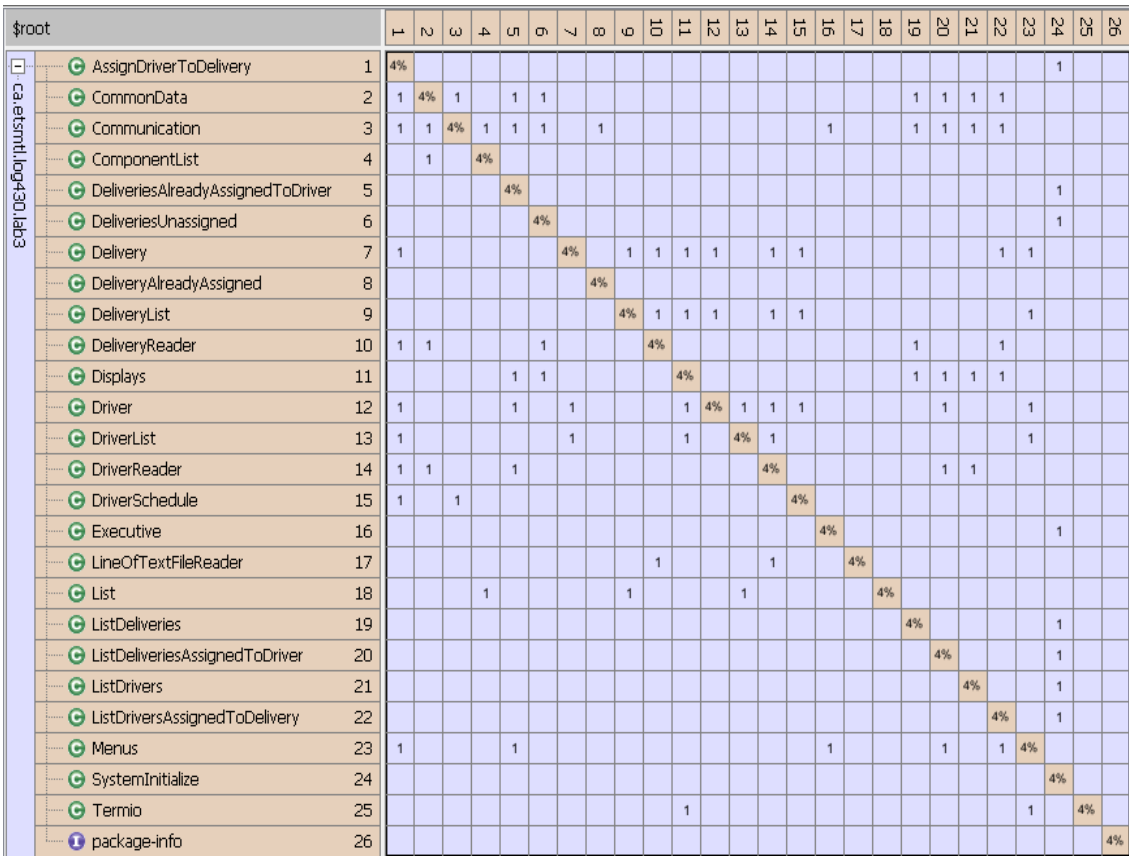

Illustration H: Image du DSM final

Comme on peut l'observer sur les deux images du DSM, quelques ajouts ont été effectués lors de la modification du système. Tout d'abord, la classe DriverSchedule a été ajoutée au système, ajoutant quelques dépendances simples au projet. Ensuite, quelques acteurs réagissant aux événements ont été ajoutés, ajoutant des dépendances envers la classe Communication. En somme, les modifications apportées ont bel et bien changé les dépendances, mais pas de façon significative.

D. Discussion des modifications apportées

4.1 Discussion sur la modificabilité de l'architecture en invocation implicite versus l'architecture en couches

Dans l'architecture de style «invocation implicite», chaque module est potentiellement indépendant des autres, alors qu'une couche, dans l'architecture de style «en couches» est un composé cohésif dépendant souvent d'une couche inférieure et de son interface.

De cette façon, avec le style «invocation implicite», il est plus simple de ne changer qu'un seul module, alors que dans l'architecture en couche, il est parfois plus simple de remplacer une couche complète, vu sa cohésion et l'interdépendance des modules internes à la couche. De plus, le style «invocation implicite» permet une plus grande flexibilité au niveau de la communication entre les modules: plutôt que de créer des dépendances entre plusieurs modules, les modules ne dépendent que du bus de données, qui lui aura la responsabilité d'acheminer des messages aux bons composants. Avec l'architecture en couche, contrairemetn à l'architecture en invocation implicite, on tend à restreindre les communications entre les modules, dans ce cas-ci aux couches inférieures, ce qui restreint les possibilités de communication dans les situations où plusieurs modules doivent communiquer d'égal à égal.

Le style d'architecture invocation implicite permet plus facilement de découpler et de rendre les événements autonomes : les modifications sont donc plus faciles à réaliser dans un environnements où on retrouverait divers acteurs pouvant agir de façon autonome ou semi-autonomes. Il est aisé, alors, d'ajouter, d'enlever ou de modifier des acteurs. Par contre, il serait difficile de modifier un système afin de modifier ou de supprimer des évènements : en effet, les acteurs, indépendamment, dépendent des signals

qu'ils reçoivent, et ne modifier ne serait-ce qu'un seul signal au cœur d'un vaste système où ce signal est utilisé pourrait être complexe.

Conclusion

En conclusion, avec les notions apprises en classe ainsi qu'avec le livre d'architecture logicielle obligatoire pour ce cours, nous avons eu la chance d'appliquer nos nouvelles connaissances en implémentant et en analysant un système de livraison selon un style architectural à invocation implicite. La partie analyse a permis à l'équipe d'approfondir davantage les notions acquises, mais nous a aussi permis de nous former une opinion plus éclairée sur ce style architectural. Avec ce rapport de laboratoire, l'équipe a expliqué brièvement les modifications apportées au système original de gestion des livraisons ainsi que les impacts que celles-ci ont eus sur l'architecture du logiciel. Le style invocation tel qu'indiqué fut conservé du logiciel initial avec très peu de déviations.