Génie logiciel et gestion de projet

Groupe 01 - Release notes

David Abraham, Rémy Detobel, Stanislas Gueniffey, Denis Hoornaert,
Nathan Liccardo, Robin Petit, Loan Sens, Théo Verhelst

Itération 1

Planification du projet

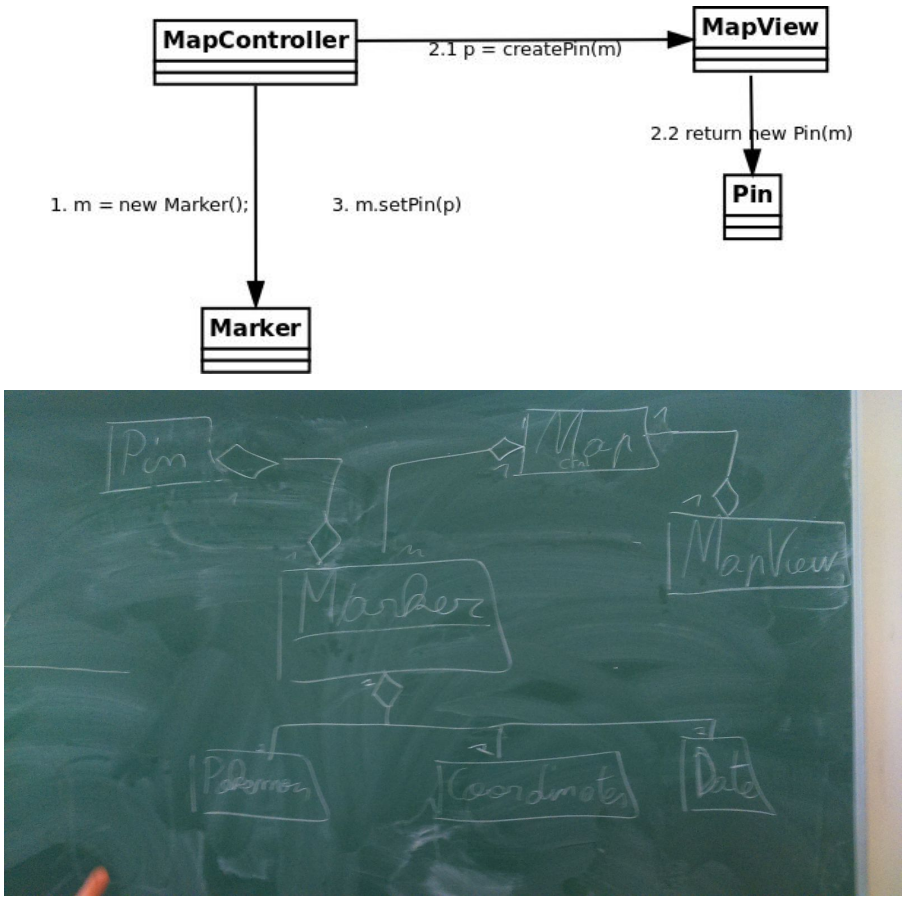
Dans l'histoire une, toute les sous-histoires ont été implémentées sauf le *grouping* lors du "dézoom" (comme convenu avec le client). De plus, la fonctionnalité de "zoom/dézoom" n'est, à l'heure actuelle, pas fonctionnelle.

Les tâches à accomplir étaient les suivantes :

- Afficher des épingles
- Pouvoir sélectionner des épingles pour voir leurs informations (nom du pokémon et date de parution du pokémon).
- Ajout d'épingle sur la carte (avec complétion des informations : nom et date de parution)
- Système de "zoom/dézoom"

Structure du code et choix techniques

Pour le moment, l'architecture générale suit le modèle de classe suivant :



Le modèle ci-dessus ne contient pas tous les widgets ayant été créés.

Le modèle suivant le schéma MVC, nous avons réparti les classes de la manière suivante :

Modèle	Vue	Controller
	MapView	MapController
	Pin	Marker
		Pokemon
		Coordinates
		Date

De manière générale, il a été décidé que les classes faisant parties de la vue ne contiendraient aucune informations et que ces dernières seraient contenues dans les contrôleurs qui leur sont associés.
À titre d'exemple, la *MapView* affiche bien l'image désirée mais ne connais aucunement le chemin menant à l'image. En conséquence, MapView doit faire appel à une méthode de MapController pour obtenir le chemin vers l'image.
De plus, il a été décidé pour des raisons de mise à jour graphiques que ce sont les contrôleurs qui vont indiqué aux vues quand et où dessiner les widgets. Pour cela, les vues possèdent des attributs et des méthodes static.

Difficultés techniques rencontrées

Lors de cette partie, les principales difficultés qui ont été rencontrées étaient liées à l'utilisation correcte de l'API JavaFX. Typiquement, beaucoup de temps a été perdu lors de la conception des fenêtres *pop-up* car l'équipe ignorait qu'elles étaient les widgets les plus adaptées ainsi que les fonctionnalités déjà disponible.

Du point de vue de la conception des classes, des modifications tardives ont dû être effectuées pour permettre d'incorporer les éléments de zoom au MapView. De plus la zone du plan à afficher est difficile à modifier en fonction du facteur de zoom et du point du plan à conserver lors de la translation.

Itération 2

Planification du projet

Objectifs :

- Regroupement de pins ;
- introduire Google maps ;
- Ajout du système de réputation avec un utilisateur bidon
- Ajout d'informations sur les pokémons (PV, attaque, défense, type(s))
- Remplacer épingle par l'image du pokémon correspondant
- Ajout d'un pop-up lors du click sur un groupement d'épingles

Typiquement, l'ensemble de ces points ont été respectés.

Structure du code et choix techniques

Modèle	Vue	Controller
CoordinateModel	MapView	MapController
DataBaseModel	PinPopUp	MarkerController
MarkerDataBaseModel	ClusterPopUp	AuthetificationConroller
MarkerModel	LoginPopUp	ClusterPopUpController
PokemonDataBaseModel	NewMarkerPopUp	NewmarkerPopUpControll er
PokemonModel	PokemonLabel	PinPopUpController
PokemonTypeDataBaseM odel	PopUp	
PokemonTypeModel	PokemonPanel	
ReputationScoreModel		

De manière générale, il a été décidé que les classes faisant parties de la vue ne contiendraient aucune informations et que ces dernières seraient contenues dans les contrôleurs qui leur sont associés.

À titre d'exemple, la *MapView* affiche bien l'image désirée mais ne connais aucunement le chemin menant à l'image. En conséquence, MapView doit faire appel à une méthode de MapController pour obtenir le chemin vers l'image.
De plus, il a été décidé pour des raisons de mise à jour graphiques que ce sont les contrôleurs qui vont indiqué aux vues quand et où dessiner les widgets. Pour cela, les vues possèdent des attributs et des méthodes static.
Depuis le début de l'itération deux, des efforts ont été fait suite à une remarque. Celle-ci disait que les class contenant des données doivent belle et bien se trouver dans la catégorie "Modèle".

Difficultés techniques rencontrées

Refactoring MVC
On a décidé que les controllers créés les modèles (déjà dis mais confirmation). Marker à été refactoré (split entre MarkerModel et MarkerController). Constatation : les contrôleurs ne servent pour le moment que de relais entre les Views et modèles. De plus, on a supprimé certain getters pour les remplacer par d'autres.

Ajout d'input dans le NewMarkerPopUp
Refactor de la gestion du "newMarkerPopUp". En effet, celui-ci ne possédait pas de contrôler propre lors de la première itération (C'était le MapController qui le gérait). Du coup, un contrôleur nommé "NewMarkerController" a été créé.

Base de donnée concernant les types des pokémons
Le type NONE ne peut jamais être enregistré en base de donnés (il ne peut donc pas être assigné à un Pokémon qui sera lui même enregistré dans la base de données).

Chargement des Pokémons depuis la Base de données
Pour question de simplicité, les Pokémons seront tous chargé au démarrage de l'application. Il en va de même pour les Type de Pokémon.

Utilisation de Google Maps
Changements effectués de manière satisfaisante.
Difficultés principalement pour communiquer entre la page Web reprenant le module Google Maps et la classe Java l'utilisante (MapView). Difficultés également pour configurer le chemin vers les assets pour le pin clustering (plus précisément, les images représentant un cluster).

Ajout d'une autocomplémtion sur la combobox/entry de newMarkerPopUp
Création d'un contrôleur dédié à la newMarkerPopUp. Avant, la map controller gérait aussi le newmarkerPopUp. L'autocomplétion devait démarrer une fois que 3 caractères sont rentrés. La plupart du temps de programmation a été passé à rechercher l'event JavaFX déclenché à

chaque entrée d'un caractère. Malheureusement, même après avoir trouvé l'event, d'autre problème sont survenus. Ceux-ci survenaient lors de la mise à jour du contenu de la combo-box. Pour finir, l'idée a été abandonnée de manière à remettre un prototype propre.

PinPopUp interface + reliage controller
Ajoute de widgets affichant les données relative au marker correspondant. Il a donc fallu ajouter des getters ainsi que des widgets (principalement label). Pas de remarque ou de décision particulière à mentionner. Cependant, il faudra négocier la création d'un contrôleur dédié (p-e). Il reste a relier l'interface avec le controller pour mettre à jour l'interface lors d'un vote ainsi qu'à ajouter des action/event sur les buttons.

Base de donnée concernant les Pokemons - Rémy-David 27/03/17
Problème de merge avec Pokemon_Stats… (perte de temps) Suite de l'implémentation de la base de donnée, les fonctions pour charger les Types et les Pokemons ont été ajoutés.

Corrections/Débug pour l'affichage de la pop-up du pin - Stanislas & Loan 28/03/17
Les pop-ups n'apparaissant plus, une erreur ambiguë provenant du JavaScript indiquant qu'on ajoute un "null" comme "node", mais aucune information concernant la ligne en question, mais grâce au débogueur (et beaucoup de patience) on à pu remonter jusqu'à la source de l'erreur.
La création des spinner en référence ne se réalisait pas, on a donc modifié pour renvoyer un Spinner (au lieu de modifier la référence) pour résoudre le bug

Itération 3

Planification du projet

Les objectifs de cette itération étaient les suivants :

● Histoire 5 sans implémenter d'avatar

● Ne pas implémenter la possibilité pour un user de modifier ses informations

● Interface de modification visible directement dans le popup

● Barre d'option qui apparaît avec un bouton, comme sur interfaces mobiles

● Un visiteur peut tout utiliser en local, et se synchronise si il décide de créer un compte. Cependant il ne peut pas voter sur les marqueurs

● Pas de gestion des conflits de marqueurs

● Mise à jour des informations au lancement du programme seulement

● Compte non utilisable avant la confirmation de l'utilisateur par e-mail

L'ensemble de ces points ont été abordés, cependant des difficultés liées à la configuration client/serveur nous empêchent au moment de la release de les tester.

Structure du code et choix techniques

La séparation du projet en client/serveur (ainsi que common, la partie du code partagée) implique également l'apparition de nouvelles classes. En effet les classes situées dans common et dont les instances sont échangées à travers le réseau doivent à présent implémenter une interface correspondante, et un système d'héritage est parfois requis pour séparer des fonctionnalités qui elles ne sont pas communes. Les classes situées dans common sont qualifiées de sendable model et leurs interfaces sont dites des query model. La plupart des contrôleurs font le lien entre un modèle et une vue et se trouvent du côté client, mais des fonctionnalités plus complexes que de simples requêtes existent également côté serveur, comme l'envoi d'e-mails.

La séparation entre "Modèle", "Vue" et "Contrôleur" existe toujours et les choix effectués à propos de leurs utilités et portées restent vrais. Chaque type de classe se trouvant désormais dans un package propre à ce type, il est redondant de reprendre la liste

extensive dans un tableau. Depuis le début de l'itération deux, des efforts ont été faits pour que les classes contenant des données se trouvent bel et bien dans la catégorie "Modèle".

Difficultés techniques rencontrées

Initialisation de communications RESTFul

De grosses difficultés ont été éprouvées à mettre sur pied le système de communications et à décider quelle librairie utiliser. Au final, notre choix s'est porté sur Jersey. La compréhension des mécanismes de fonctionnement de celle-ci a demandé du temps. Et son déploiement encore plus.

Refactoring : séparation des classes entre client et serveur

L'introduction d'un système client-serveur survenant tardivement dans le projet, le refactoring ainsi que la séparation claire et précise des classes entre le client et le serveur a été long, laborieux et a demandé une concentration ainsi qu'une organisation toute particulière.

Interface d'édition de pokémon

L'ajout de cette option a demandé un peu de temps mais s'est vite faite via le refactoring de classes déjà existantes et grâce à un héritage de contrôleurs et de vues.

Interface de login/logon et panel

Cela s'est fait sans gros problèmes et a permis de refactorer le main et d'autres parties du code pour le rendre (de notre point de vue) plus lisible et léger.

Système de réputation : finitions

Le système de login complet s'est accompagné d'une restructuration des tables de la base de donnée, permettant de stocker la relation entre un "vote" (une appréciation d'un utilisateur sur un marqueur) et un utilisateur, et évitant ainsi toute fraude (votes multiples entre le même utilisateur et un marqueur, entre un visiteur et un marqueur...)

Itération 4

Planification du projet

Les objectifs de l'itération 3 n'ayant pas été complètement atteints à cause de difficultés techniques rencontrées, concernant principalement la mise en place d'une architecture client/serveur et la configuration des ressources dans les environnements de développement/release, une réestimation des points pour l'itération 4 a été nécessaire.

Voici la liste des objectifs pour l'itération 4 :

- Complétion des objectifs de l'itération 3, révision de l'architecture pour éviter de nouvelles erreurs d'estimation.
- Correction des erreurs survenues lors de la démonstration (apparition des popup sur l'écran secondaire en mode projecteur, problèmes de connexion avec un compte nouvellement créé).
- Partage des informations d'un marqueur sur Twitter, avec lien vers les coordonnées du marqueur sur Google Map.
- Recherche et filtrage des marqueurs sur la carte (basées sur les caractéristiques de nom et de type des pokémons concernés, conditions combinées avec opérateurs ET, OU, NON).
- Sauvegarde des filtres par les utilisateurs ayant un compte, utilisation de filtres sauvegardés par n'importe quel utilisateur.

Le premier point a été résolu sans problème, une fois les solutions aux problèmes techniques rencontrés l'architecture du code s'est révélée correcte, nous avons tout de même veillé à effectuer certaines restructurations pour assurer plus de modularité dans le code.

Une fois cette difficulté surmontée, les nouveaux objectifs ont tous été rencontrés de façon satisfaisante.

Structure du code et choix techniques

La séparation du projet en *client*/*serveur* (ainsi que *common*, la partie du code partagée) implique également l'apparition de nouvelles classes. En effet les classes situées dans *common* et dont les instances sont échangées à travers le réseau doivent à présent implémenter une *interface* correspondante, et un système d'héritage est

parfois requis pour séparer des fonctionnalités qui elles ne sont pas communes. Les classes situées dans *common* sont qualifiées de *sendable model* et leurs interfaces sont dites des *query model*. La plupart des contrôleurs font le lien entre un modèle et une vue et se trouvent du côté client, mais des fonctionnalités plus complexes que de simples requêtes existent également côté serveur, comme l'envoi d'e-mails.

La séparation entre "Modèle", "Vue" et "Contrôleur" existe toujours et les choix effectués à propos de leurs utilités et portées restent vrais. Chaque type de classe se trouvant désormais dans un *package* propre à ce type, il est redondant de reprendre la liste extensive dans un tableau. Depuis le début de l'itération deux, des efforts ont été faits pour que les classes contenant des données se trouvent bel et bien dans la catégorie "Modèle".
(parler des sous "catégorie" dans les packages)

Difficultés techniques rencontrées

***Refactoring* du code**
Une vérification du code a été faite avec les outils présentés en cours. Dans un premier temps, l'ensemble du code a été revu à l'aide de PMD et des modifications et améliorations ont été apportées lorsque celles-ci ont été considérées judicieuses. Ensuite, de manière à estimer la qualité de l'architecture ainsi que les redondances dans le code, CodeCity a été utilisé. Peu d'incohérences ou de situations alarmantes ont été trouvées. Cependant, il nous est apparu clairement que les contrôleurs sont fortement liés (ce constat a déjà mis en avant lors de la troisième code review via Moose). Au vu de la deadline de la quatrième itération et de la date de la troisième code review, il a été décidé de ne pas se lancer dans un refactor complet de l'architecture car celle-ci aurait été trop chronophage.

(Wontfix) - bug d'affichage des marqueurs sur la map
Depuis la première itération, le système de clustering utilisé a causé des problèmes d'affichage des marqueurs : après un declustering, et parfois après un changement du niveau de zoom, les images représentant les marqueurs ne correspondent pas toujours au pokemon indiqué. Ce *glitch* est uniquement graphique et un simple zoom/dézoom suffit à retrouver l'apparence correcte, mais nous avons évidemment déployé des moyens pour y remédier.

- Une issue a été ouverte sur le *repo* officiel de google maps - marker clusterer où se trouve le code JavaScript employé. Il s'est avéré que ce repo n'était plus maintenu de façon active, mais que son auteur avait créé un fork pour le maintenir lui même. Après s'être entretenus avec cet employé de Google, nous avons pu déterminer que le glitch était entièrement lié à l'environnement WebView dans JavaFX, et donc que le code utilisé n'était pas en tort.

- Pour l’instant, le seul moyen trouvé pour arranger l’affichage est de recréer toutes les instances de marqueurs à chaque changement de zoom, ce qui n’a finalement pas été merge sur master par soucis de propreté et d’efficacité du code.

Bug affichage-chargement du css

Lors de la dernière itération, un “bug” concernant le thème css est apparu. Par déduction, nous avons déterminé qu’il provient d’un conflit lors du chargement du css de l’application javafx et de la google map (il est difficile d’apporter une preuve formelle). Nous nous sommes rendu compte que ce “bug” ne survenait pas lorsqu’on active le bouton du panel avant le chargement de la map. Nous avons donc créé une méthode activant par défaut le bouton. Ainsi, le “bug” ne survient plus.