UQÀM | Université du Québec à Montréal

Cours	MGL7460 – Réalisation et maintenance des logiciels
Session	Automne 2018
Nom de l'enseignant	Jacques Berger
Titre	Projet 2 – Évaluer l'impact d'un langage de programmation dynamique sur la réalisation d'un logiciel
Groupe	040
Étudiants	Olivier Charrier (CHAO25098007) Maxens Manach (MANM14019309) Hugo Sanchez (SANH18029706) François Gratton (GRAF22097405)
Adresses de courriel	charrier.olivier@courrier.uqam.ca manach.maxens@courrier.uqam.ca hugo.sanchez@viacesi.fr gratton.francois@courrier.uqam.ca
Date de remise	2018-11-08

UQÀM | Université du Québec à Montréal

Table des matières

UQÀM | Université du Québec à Montréal

Introduction

Ce projet a pour objectif d'évaluer l'impact engendré sur le développement et la maintenabilité d'un logiciel lambda par l'utilisation d'un langage de programmation dynamique. Afin d'évaluer au mieux cet impact, le projet a été découpé en phases, durant lesquelles l'équipe s'est concertée afin de réunir des informations pertinentes et des preuves empiriques.

Afin de réaliser cet exercice, deux langages ont été sélectionnés. Groovy, et Kotlin :

- **Groovy** : langage de programmation orienté objet, il a été inspiré par Python et Ruby, deux pointures reconnues en matière de langages à typage dynamique. Étendant la syntaxe Java, et complètement compatible avec ce dernier, Groovy est compilé "à la volée". Il gère aussi bien le typage statique que dynamique, et apporte son lot de spécificités qui le rendent plus polyvalent et permissif que le Java standard.

- **Kotlin** : langage développé par JetBrains, orienté objet et également fonctionnel, il entend compléter voire supplanter Java avec sa grammaire intuitive et accessible. Son typage est statique, mais aussi inféré, ce qui signifie que le compilateur peut "déduire" un type pour un objet donné en se basant sur l'utilisation qui en est faite par le programmeur. Également compatible avec Java, Kotlin réduit grandement la quantité de code redondant au sein d'une classe.

Logiciel réalisé

Ces deux langages ont été utilisés pour développer un logiciel calculant les remboursements des soins pour les usagers d'un système d'assurance. Ce logiciel se base sur des fichiers Json normalisés, qui permettent d'entrer les informations relatives aux soins d'un certain client. Après traitement, un fichier de sortie -également Json- est généré avec les remboursements afférents aux soins d'entrée.

Nos observations sont basées sur le développement de deux versions de ce logiciel, ainsi que sur les connaissances des bonnes pratiques et des spécificités de la maintenance, tirées de nos expériences personnelles et du matériel abordé en cours.

Méthodologie

Plusieurs phases se sont succédées lors de ce projet, avec à chaque fois un découpage des tâches parallélisé qui nous a permis d'amasser davantage d'informations en un temps limité. Nous avions pour idée de scinder le groupe en deux, afin de permettre à chaque binôme de travailler en autonomie sur l'implémentation. L'objectif étant qu'à la réunification du groupe, nos observations et notre synthèse pourraient se fonder sur le regard aguerri -bonne connaissance du langage et des difficultés rencontrées- des "développeurs", mais aussi sur celui -plus scrutateur et critique- des "profanes".

- **Phase 1** - Analyse du sujet, sélection des langages et du logiciel :

Lors de cette première étape, l'équipe au complet s'est concertée sur les attentes du sujet et sur la meilleure manière de mettre en valeur les différences entre l'impact de la programmation dynamique et celui de la programmation statique.

- **Phase 2** - Implémentation des programmes :

Cette seconde phase a vu le groupe se séparer en deux binômes indépendants, chargés de produire une version du logiciel dans un des deux langages choisis. Ce, afin d'offrir à chacun une expérience de développement complète, nécessaire à la bonne appréhension des langages. C'est lors de cette étape que les premières observations concernant l'apprentissage des langages ont été consignées.

- **Phase 3** - Comparaison des versions et analyse de la maintenabilité :

Lors de cette phase, chaque membre du groupe a analysé de manière indépendante les deux implémentations. L'adage veut que "deux paires d'yeux valent mieux qu'une", mais nous pensions que, pour ce projet, quatre paires d'yeux ne seraient pas de trop. Grâce à ce fonctionnement, nous avons collecté, pour chaque version du logiciel, deux expériences d'apprentissage et de développement, et quatre points de vue différents sur l'implémentation et l'analyse de maintenabilité.

- **Phase 4** - Modifications du code par les "profanes" :

Nous aurions pu nous limiter à une banale analyse de nos codes respectifs et nous arrêter à la phase 3. Mais nous souhaitions repousser les limites de l'exercice et nous mettre dans des conditions de développement "authentiques". Dans cette optique, nous avons décidé de nous concerter afin de trouver une modification applicable à nos deux versions. Après tout, la maintenabilité d'un logiciel peut être résumée à la facilité avec laquelle on peut modifier ledit logiciel pour faire face à de nouveaux besoins.

En faisant modifier la version Kotlin par l'équipe Groovy et inversement, nous faisions d'une pierre deux coups puisque nous analysions par là-même l'impact des langages (positif ou négatif) sur les tentatives des deux binômes pour refactoriser le code. Et ce, dans un système où les personnes en charge des modifications n'avaient jamais été confrontées au langage auparavant. Cette nouvelle perspective a été utile pour entériner ou infirmer nos observations ultérieures.

Réalisation en Kotlin

Évaluation de l'apprentissage

Afin de nous familiariser avec le langage et de nous préparer pour le développement de l'application, il a tout d'abord fallu trouver des ressources pertinentes. La plupart des langages proposés par de grandes compagnies, comme c'est le cas pour Kotlin (développé par JetBrains) disposent d'une documentation relativement complète. En l'occurrence, la documentation dont nous disposions pour apprendre le langage était proprement exhaustive[1] !

S'il s'agit d'un avantage certain lorsque l'on commence à maîtriser l'outil, cette source d'information profuse semblait, pour les débutants que nous étions, plus intimidante qu'accueillante. Fort heureusement, Kotlin a été développé avec l'intention de compléter, voire de supplanter Java. Aussi, les développeurs à l'origine du langage ont-ils mis en place un terrain de jeu[2] ludique et instructif pour apprendre à son rythme en se basant sur des concepts reconnus d'autres langages. Nous nous sommes en l'occurrence appuyé sur nos connaissances en Java et C#.

Koans offre une série d'exercices guidés afin de se familiariser avec les nouveaux concepts développés par l'équipe derrière Kotlin. C'est sans nul doute la meilleure façon de s'initier au langage, avec une courbe de progression très douce et une approche didactique assumée. Certes, l'accent est bien souvent mis sur des fonctionnalités qui font passer Java pour un mauvais langage, mais il n'en demeure pas moins qu'après avoir complété les différents exercices, nous étions non seulement prêts à développer en Kotlin, mais aussi capables de comprendre les innovations les plus pertinentes et comment les appliquer au projet.

Cette phase d'initiation n'a duré qu'une poignée d'heures, pour les deux développeurs concernés. Grâce aux travaux pratiques et à l'approche fondée sur des connaissances communes à la plupart des programmeurs, Koans permet de mettre rapidement le néophyte à l'aise et l'encourage à expérimenter par lui-même. De plus, en tant que langage reconnu (deuxième langage officiellement reconnu par Google pour les applications Android), Kotlin dispose d'une grande communauté qui va en s'élargissant.

Impact du langage sur la réalisation

La syntaxe intuitive et familière pour les développeurs ayant déjà pratiqué la Programmation Orientée Objet facilite grandement le développement en Kotlin. La possibilité d'écrire une classe vide dépouillé de tout son code "boilerplate" (accesseurs et mutateurs...) est un plus qui favorise énormément la lisibilité et réduit grandement le nombre de lignes redondantes écrites.

L'utilisation et la configuration d'un IDE compatible (puisque développé directement par JetBrains) a favorisé le développement en offrant des "corrections" qui mettent en avant ses concepts de simplification du code. Déduction des types, simplification des expressions mathématiques, désimbrication des boucles inutiles... Autant d'outils qui améliorent la rapidité et la fiabilité du code. De plus, l'architecture du projet a été conçue conformément aux bonnes pratiques actuellement en vigueur au sein de l'industrie. Là encore, le langage n'a pas offert de "résistance" et a même favorisé un développement consciencieux et respectueux desdites pratiques.

Réalisation en Groovy

Évaluation de l'apprentissage

Après la démonstration en classe des capacités et des possibilités offertes par le langage Groovy, la table était mise pour nous plonger dans son apprentissage. Puisque Groovy est un langage qui offre plusieurs similitudes avec Java, un langage connu par l'équipe et qu'il semble en plus être très apprécié et utilisé par les programmeur Java, nous avons estimé que son apprentissage ne devrait pas impliquer de grandes difficultés.

La première étape de l'apprentissage consistait à trouver les ressources pertinentes pour nous initier à ce nouveau langage, pour installer un environnement de développement et pour trouver la documentation. Le site officiel Groovy[3] nous a permis de facilement trouver cette information. Un des deux programmeurs a par contre eu de la difficulté à faire fonctionner Groovy sur Eclipse en suivant les étapes proposées sur le site officiel (perte d'une heure). Il a fini par suivre un tutoriel Youtube en installant une ancienne version d'Eclipse et le JDK 8.

Sur la page Documentation du site, on retrouve une liste d'un trentaine de points à lire pour démarrer en Groovy. Nous les avons donc lus pour nous familiariser avec les spécificités et les nouveaux concepts apportés par ce langage. Nous donnons une note de 8 sur 10 pour cette introduction au langage Groovy. En revanche, force est d'admettre que nous aurions apprécié un système de tutoriel via des exercices interactifs comme celui proposé par Kotlin pour débuter. Il existe bien la groovy web console (que nous avons utilisée) mais elle ne propose pas des tutoriels interactifs bien pensés comme dans Kotlin Koans. Par contre dans cette introduction, un des trente points portait sur la manipulation Json. Les explication était brèves et concises et ceci nous a permis de rapidement jouer avec les concepts et ainsi pouvoir les appliquer pour la réalisation de notre logiciel à base de Json.

La phase d'apprentissage qui nous a permis de nous mettre en confiance et de passer à la réalisation à proprement parler de notre logiciel a été rapide et n'a durée que 2 ou 3 heures. Elle s'est avérée plus courte que pour le binôme travaillant en Kotlin.

La phase d'apprentissage suivante, qui a eu lieu tout au long de la réalisation du logiciel, s'est elle aussi avérée relativement facile. La documentation est présente et Groovy dispose

d'une communauté qui échange et répond aux questions posées par les programmeurs. La communauté est tout de même nettement plus petite que pour Python par exemple, un autre langage dynamique. Nous avons tout de même trouvé les réponses à toutes nos questions, au vu de la simplicité relative du logiciel développé. Puisque le code Java fonctionne normalement sous Groovy, il est aussi possible d'utiliser des solutions pour Java et d'adapter le code de ces exemples.

Impact du langage sur la réalisation

La syntaxe moderne, simple, intuitive et familière du langage Groovy apporte une facilité à la réalisation. Si le développeur connait Java, un des langages les plus populaires de tous les temps, et qu'il maîtrise les bases de la programmation objet, on peut estimer que le codage en tant que tel sera très rapide pour un nouveau langage. Nous avons profité de plusieurs propriétés du langage lors de la réalisation du logiciel:

- Prise en charge du typage statique et dynamique
- Syntaxe simplifiée, code plus court pour une même fonctionnalité, plus besoin de point-virgule par exemple...
- Classes courtes, accesseurs et mutateurs autogénérés

Groovy peut être considéré comme est un langage puissant au niveau des tests. Ceci est important pour mitiger les possibles dangers inhérents à son typage dynamique. Il utilise une syntaxe qui prend en charge l'exécution de tests dans les IDE, dans Ant et dans Maven[4] par exemple. Groovy intègre également des bibliothèques et des frameworks de test de pointe comme le traditionnel JUnit mais aussi Spock ou Geb qui tire profit des fonctionnalités de Groovy. Groovy propose et implémente des notions puissantes comme les power asserts, la création facile de mock et de stubs personnalisés[3].

Eclipse est l'IDE de choix pour programmer en Groovy. Le plugin Groovy Development Tools apporte les supports habituels pour un langage comme l'assistance de contenu, l'autocompletion et le débogage. Les plug-ins existe pour les frameworks et les autres outils externes. Malheureusement nous ne croyons pas que l'environnement Groovy fait et fera le poids (dans les prochaines années) face à l'environnement IntelliJ + Kotlin qui sont tous les deux produit par le même éditeur. Kotlin couplé avec IntelliJ offre des concepts de simplification du code, de déduction des types, de simplification des expressions mathématiques et de désimbrication des boucles inutiles par exemple.

Le typage dynamique de Groovy a simplifié la réalisation du programme. Groovy nous a permis d'essayer à la volée toutes sortes de choses pour réussir nos manipulations. On ne s'occupait pas des types des variables. On essayait toutes sortes des choses, on mélangeait les types, on ne typait plus, on profitait de l'intelligence du langage qui découvrait ce que l'on tentait de faire et on roulait le programme a répétitions jusqu'à ce que l'on obtienne le résultat désiré. Ensuite on passait au prochain problème mais le code demeurait sale.

Voici un exemple du style intelligent mais laxiste de Groovy: Nous avons modifié le nom d'un objet Json de "réclamation" à "remboursement" mais on l'appelait plus loin dans le code en utilisant son ancien nom et tout fonctionnait ! Un comportement qui, certes, nous facilite la vie mais qui pose également un problème flagrant au niveau de la maintenance et de la compréhension du code...

Comparaison des deux produits

En préambule de cette partie, nous tenons à préciser qu'il s'agira dans un premier temps ici de récupérer les analyses personnelles des quatre membres du groupe qui ont chacun eu une expérience différente de ce TP, par le choix que nous avons fait de séparer le travail en deux groupes indépendants comme précisé en début de rapport.

Ainsi au-delà des différences propres aux langages, il y a aussi une différence dans la conception des programmes. Nous tenterons d'en faire abstraction, autant que possible, en nous concentrant sur la comparaison des langages, les deux programmes étant fonctionnels. Pour préciser, le programme Kotlin à été conçu en partie par un expert dans la conception qui avait comme objectif de créer un logiciel réutilisable et modifiable facilement en dehors du cadre du projet. Pour ce qui est du programme Groovy, il a été conçu plus simplement avec l'objectif en tête et est composé de bien moins de classes (trois, une pour l'import, une pour l'export et une pour le traitement de l'information) qu'en Kotlin.

Kotlin

Dans un premier temps, évoquons le langage Kotlin. Le binôme attitré au langage à pu avec le développement dresser un portrait des bons points du langage favorisant la maintenance :

Une idée intéressante est que le langage a instauré une règle visant à réduire une des erreurs d'exécution la plus courante: l'exception nulle. Cela est fait en ne permettant pas l'assignation d'une valeur nulle à une variable régulière, cependant il demeure possible d'assigner un "null" lorsque la situation l'oblige, mais pour ce faire, il faut définir une variable spéciale acceptant la valeur nulle.

Kotlin offre également une syntaxe de déclaration plus concise. Par exemple, la définition d'une classe, de ses attributs, ses accesseurs et du constructeur peut se faire sur une seule ligne. La lisibilité du code est priorisée, en réduisant le bruit redondant (accesseurs et mutateurs), mais cela permet surtout aux développeurs d'investir leur temps et énergie dans l'écriture du code de qualité plutôt que de s'étaler sur l'écriture de code "boilerplate".

Enfin le langage offre l'immuabilité: comme en programmation fonctionnelle, le langage offre la possibilité de déclarer une variable de façon à ce qu'elle soit immuable. Les variables de classes peuvent êtres déclaré d'une façon variable (var) ou invariable (val), les collections de bases sont immuables (List, Set, etc), mais il est possible d'utiliser leurs équivalents

mutables (MutableList, MutableSet, etc). Cette notion d'immuabilité permet de réduire les bogues associés aux effets de bords, aux données partagées en traitement parallèle, etc.

Les observations du binôme Groovy sur le langage Kotlin, sans être logiquement aussi poussées, vont dans le sens d'un langage qui paraît relativement maintenable également. Les raisons derrière cette impression sont que dans un premier temps, le langage est assez similaire syntaxiquement aux langages déjà très répandus dans l'Industrie, avec quelques différences dans la manière de définir certains éléments (par exemple les fonctions qui se définissent grâce à la syntaxe « fun nom(arg) : type ». Ajoutez à cela la conception détaillée du programme, et on obtient un logiciel aisément compréhensible, premier pas vers la maintenabilité.

Groovy

Nous avons été impressionnés par la capacité du langage à parser le JSON. C'est une chose que nous avions lu sur divers forums en nous renseignant sur le langage, notamment de la part de certains développeurs qui aimaient à avoir des programmes en Groovy, qui en réalité étaient entièrement codés en Java, à l'exception du parsing de XML ou JSON qu'ils réalisaient en Groovy. Dans les faits, cela nous permet en tout cas d'avoir un programme hautement maintenable au niveau des règles de remboursement des soins. Aucun besoin de créer de classe ou de faire de la disjonction de cas en fonction de tel contrat ou tel soin pour avoir un traitement, on peut simplement stipuler les règles dans un fichier JSON puis parcourir l'ensemble des règles pour appliquer celles qui correspondent.

En parlant de mélanger Java et Groovy, nous y avons vu un danger également en tant que développeurs à la fois novices du langage et confirmés en Java. En effet, il n'est pas rare (et surtout, pas défendu par le compilateur) de laisser trainer un élément syntaxique de Java (par exemple le point virgule en fin d'instruction qui accroche l'œil), donnant un code au style non-consistant.

Toutefois cette proximité syntaxique avec le Java, tout comme Kotlin, donne un aspect immédiatement compréhensible au Groovy. D'ailleurs il existe dans les IDEs Java des extensions pour Groovy. Sous Eclipse, l'intégration de tests unitaires JUnit en Groovy se fait très facilement, ce qui est également un plus.

Différences

Maintenant que l'analyse indépendante des deux programmes et langage est effectuée, comparons ce qui les différencie :

Le langage dynamique Groovy offre les particularités suivantes :

- Paradigme orienté objet, où tout est un objet
- "Closure", un bloc de code peut être déplacé et exécuté ailleur dans le programme
- Le protocol MOP (meta object protocol) est supporté pour offrir la métaprogrammation
- Supporte nativement la lecture et l'écriture de contenu json
- Supporte le duck typing

De ces particularités techniques notamment, il apparaît que Groovy – langage dynamique – est un langage qui offre plus de possibilités et de flexibilités que son rival, Kotlin. La possibilité d'avoir un langage complètement objet, permet d'avoir des méthodes sur les primitives objet, mais aussi d'offrir la surcharge d'opérateurs. Le protocol MOP permet de reprogrammer le code au runtime, pour faciliter les tests, mais également pour d'autre utilisations plus poussées. L'assignation de différents types de données à une variable, et la découverte de son type selon l'utilisation (duck typing), confère au développeur une flexibilité incroyable dans la conception et la mise en oeuvre de ses idées.

Finalement, en prenant en compte que le langage dynamique Groovy nous offre plusieurs moyens pour écrire du code de façon plus flexible (variables à type dynamique, typage des variables, surcharge d'opérateurs, duck typing, réduction du code "boilerplate", etc), cette mentalité favorise alors une écriture désinvolte du code et se traduit par un code plus difficile à maintenir.

C'est une observation qui à été confirmée pendant le développement par notre binôme Groovy qui, étant assez peu familier avec les langages dynamique, a parfois eu tendance à mal utiliser les types non-définis par facilité. Toutefois c'est une mauvaise pratique qui se ressent au moment de l'exécution avec l'apparition des bogues. Et pour éviter de perdre trop de temps en maintenance, il faut employer ce même temps à être très rigoureux dans les tests afin de couvrir les errements du développeur. Plus facile à réaliser donc en théorie, mais qui demande une meilleure rigueur et une bonne connaissance du langage pour éviter de commettre des impairs.

Analyse de la maintenabilité des logiciels finis

Dans cette partie du rapport, en complément de la partie précédente, nous allons utiliser des métriques afin de pouvoir évaluer la maintenabilité des deux produits et les comparer.

Nous tenons à préciser qu'en raison de nos conceptions différentes sur les deux produits, nous avons essayé de sélectionner des métriques davantage orientées sur les différences entre les langages que sur la conception. Il a été fait abstraction autant que faire se peut des différences de style entre binômes.

Lines of Code

Puisque l'implémentation est différente, il s'agit d'une des mesures de maintenabilité du code les plus pertinentes. En théorie, la différence ne devrait pas être trop élevée, puisque, indépendamment de l'implémentation, la logique reste la même.

Nous observons cependant une différence majeure, puisque 120 lignes de code ont été produites au total en Kotlin contre seulement 53 en Groovy. Une différence, du simple au double, que ne peut expliquer seule une conception plus soignée en Kotlin.

Nous expliquons ces différences par la manière de vérifier le format des informations qui est très bref sous Groovy. L'autre différence majeure est la manière non seulement de récupérer le JSON en Groovy, mais surtout de le traiter. Dans le programme Groovy, une double boucle FOR passe simultanément à travers les soins et les couvertures et modifie ce même Json à la volée pour obtenir le fichier de sortie.

Force est d'admettre que Kotlin, malgré sa volonté affichée de réduire le code redondant est tout de même un langage un peu plus verbeux que le Groovy, ce qui le rendrait de facto plus difficile à lire, et donc à comprendre. Dans les faits, cette analyse est à nuancer, au vu de la syntaxe extrêmement intuitive du Kotlin et de la limpidité observée dans les concepts affichés.

Niveaux d'imbrication des structures de contrôle

En développement logiciel, les structures de contrôle sont omniprésentes et nécessaires à la réalisation d'un code un tant soit peu complexe. Pour autant, leur multiplication et leur entrelacement au sein d'une même fonction peuvent mener à une logique obscurcie, voire carrément à une portion de code non-maintenable, lorsque l'imbrication massive est couplée avec de mauvaises pratiques telles qu'une nomenclature irrégulière et un style dépareillé. Il est généralement pertinent de chercher à limiter autant que possible cette métrique, afin de conserver un code lisible et compréhensible.

Au sein de nos deux projets, un niveau d'imbrication maximum et moyen, par fonction, a été calculé. Ce, afin de départager les langages sur leur complexité effective face à une même tâche. La métrique a été analysée de manière globale plutôt que pour chaque fonction, afin d'offrir un retour plus pertinent étant donné les disparités des deux programmes.

Kotlin :

- Niveau d'imbrication maximum : 2 boucles imbriquées
- Niveau d'imbrication moyen : 0.78 boucle imbriquée par fonction

Groovy :

- Niveau d'imbrication maximum : 3 boucles imbriquées
- Niveau d'imbrication moyen : 0.88 boucle imbriquée par fonction

On peut observer avec ces résultats que pour un même projet, les structures de contrôle Groovy tendent à être plus complexes en moyenne que les structures Kotlin. Du point de vue de la maintenabilité, on peut en déduire qu'une méthode Groovy sera possiblement plus difficile à comprendre et à modifier.

Nombre de jeton

Selon Halstead, un programme informatique est la mise en œuvre d'un algorithme considéré comme une collection de jetons qui peuvent être classés soit comme des opérateurs, soit comme des opérandes. En d'autres termes, un programme peut être considéré comme une séquence d'opérateurs et de leurs opérandes associés(9). Les métriques de Halstead utilisent le compte de ces jetons dans plusieurs formules pour estimer par exemple le temps requis et la difficulté pour programmer un logiciel. Pour le travail, nous avons compté

manuellement, pour chacun des logiciels, les jetons sans différencier les opérandes des opérateurs. Le but ici est de comparer grossièrement les résultats. Dans les petits logiciels, l'effort, le temps requis et même la maintenabilité tendent à être proportionnels au nombre de jetons du logiciel. Il y a plusieurs écoles de pensée sur ce qui devrait être considéré ou non comme des jetons. Nous nous sommes entendu sur un style et avons suivi ces mêmes règles pour les deux logiciels. Voici les résultats :

Langage	Kotlin	Groovy
Nombre de jetons	533	266

Nous obtenons sensiblement la même proportion, la même différence entre les deux logiciels qu'avec la métrique LOC discutée plus haut. Est ce que Groovy permet de coder un même logiciel que Kotlin avec 2 fois moins de jetons? Nous ne pensons pas. Groovy et Kotlin sont des langages modernes et pensés pour coder court. Cette différence est plutôt attribuable à la division en de multiples classes (Single Responsability Principle) et à l'architecture plus étendue mais de meilleure qualité et suivant les pratiques de l'industrie côté Kotlin.

Évolution des logiciels par un binôme novice :

Enfin, comme précisé dans la partie Méthodologie, nous avons décidé de profiter de la structure atypique de notre groupe pour voir s'il était simple de faire évoluer un programme pour une personne venue de l'extérieur. Nous avons voulu adopter le point de vue d'une équipe de développement qui arrive dans une société et doit modifier un système patrimonial dont elle ne connaît pas le langage. Ceci nous a à la fois permis de prendre connaissance du langage, et surtout d'évaluer la maintenabilité du produit du binôme opposé. Car autant il ne serait pas honteux de ne pouvoir faire évoluer le logiciel, autant si on en est capable alors cela témoigne d'un langage particulièrement accueillant pour les novices.

Évolutions proposées

Comme pour le sujet du programme, nous nous sommes conformés au document de demande de changement. Il s'agissait de demandes d'évolutions du logiciel de base. Nous sommes partis sur l'idée d'implémenter des améliorations demandées dans le document DDC1. Toutefois elles étaient très nombreuses, et nous avons donc décidé de n'implémenter que cinq fonctionnalités de la liste (voir tableau ci-dessous) car le but de cet exercice est avant de juger de la maintenabilité.

Fonctionnalités recherchées :

- Notre client rembourse maintenant un soin supplémentaire : «Kinésithérapie», qui possède le numéro de soin 150.
- La psychologie individuelle pour le contrat de type B est maintenant couverte à 100% sans montant maximal.
- Dans le fichier d'entrée, les propriétés «client» et «contrat» n'existe plus. Elles ont été remplacées par une nouvelle propriété «dossier» qui contiendra la lettre du contrat suivi du numéro du client.
- Dans le fichier de sortie, la propriété «client» a également été remplacée par la propriété «dossier» qui doit contenir exactement la même valeur que dans le fichier d'entrée.
- Un nouveau type de contrat a été créé. Il possède la lettre «E».

Evolution du programme Kotlin par le binôme Groovy

Avant de commencer à parler de notre approche de Kotlin en tant que débutants, nous tenons à préciser qu'à l'inverse de notre apprentissage du Groovy, nous ne sommes passés par la documentation et la communauté que lorsque c'était nécessaire, devant un bogue persistant par exemple. L'idée étant de voir s'il est possible de saisir à la fois le code, et les bases du langage en reproduisant/modifiant des bouts du code en s'inspirant de ce qui a déjà été fait par nos collègues.

Premièrement, nous allons brièvement évoquer la conception. La précision de cette dernière nous permet instinctivement de nous retrouver dans le code et de connaître le fonctionnement de ce dernier (d'autant plus que nous sommes aidés par le diagramme de classe de nos collègues, qui constitue dans ce genre d'exercice un point non négligeable de maintenabilité). Toutefois il apparaît également qu'une telle complexité dans la conception peut également être considérée « trop » pour un projet de cette taille, bien que la scission en nombreuses classes avec un fort accent sur la généricité semble permettre une bonne maintenabilité, s'il fallait apporter beaucoup de modifications ou si le logiciel devait grandement évoluer par la suite.

Concernant le développement des améliorations, une fois le fonctionnement de leur architecture en tête, nous avons été surpris par la facilité d'implémenter de nouvelles fonctionnalités.

Cela peut également être dû à notre développement Groovy qui nous permet de savoir clairement ce que doit faire le programme et où s'insèreraient les améliorations, toutefois ici la différence dans la conception est un avantage puisque nous ne pouvons appliquer ce qui nous viendrait à l'instinct, ce qui nous force à approcher le programme comme un véritable système patrimonial.

Un des détails qui nous a séduit dans le développement (et que l'on voyait pourtant comme un gadget lors de l'observation du code), est la présence d'annotations de la part de l'IDE pour indiquer des types non précisés normalement.

```
couvertures.forEach { it: Couverture
    if(it.soinId == soinId && it.typeContrat == typeContrat)
        return it
}

return Couverture( soinId: 0,  typeContrat: "",  pourcentage: 0)
```

En effet, ces sortes de commentaires automatiques nous ont bien aidé dans la compréhension du code, nous permettant d'éviter des allers-retours dans les classes pour comprendre ce qu'était censé être "Couverture (0, '', 0)" par exemple.

Comme nous l'avions remarqué lors de la comparaison des deux programmes, le Kotlin est effectivement un langage facilement approchable puisque ressemblant à ce que nous connaissons. La possibilité d'avoir des classes « vides » en termes de code car beaucoup d'implicite mais pourtant bien utiles et remplies est également un plus niveau compréhension/maintenabilité.

```
package Model

class Contrat(var type:String) {
}
```

Concernant les améliorations, tout ce qui touchait à l'ajout/modification du modèle de données (ajouter un type de contrat, modifier un type de soin, etc ...) était très simple puisque nous n'avions pas vraiment à toucher le Kotlin. Pour ce qui est de modification plus en profondeur (par exemple fusionner les champs Client et Contrat en un seul champ Dossier comme demandé) n'était pas bien plus difficile en nous inspirant de ce qui était fait. Pour pousser la réflexion nous avons également regardé une demande d'amélioration plus complexe dans la DDC1 : L'ajout dans l'output d'une ligne total pour la somme de tous les remboursements. Cette fonctionnalité nous à demandé de nous frotter un peu plus au compilateur Kotlin et à effectuer quelques recherches sur Internet, mais la aussi dans l'ensemble, les novices que nous sommes sont parvenus à implémenter cette fonction grâce à la clarté offerte par le Kotlin qui est particulièrement lisible selon nous.

Curieusement, nous avons finalement mis plus de temps à installer et configurer un IDE pour faire tourner Kotlin que nous n'avons mis de temps à réellement comprendre et modifier le code.

En conclusion, nous avons vu dans ce programme Kotlin un logiciel particulièrement bien maintenable et évolutif. Le mérite en revient partiellement à la conception claire du binôme Kotlin, mais le langage à également ses mérites, notamment par des possibilité que lui et IntelliJ offrent niveau affichage et une proximité claire avec le Java. Enfin, à l'inverse de notre développement en Groovy nous n'avons pas de remarque à faire sur le rôle que le langage statique joue dans la maintenabilité puisque c'est quelque chose que nous connaissions déjà et qui n'a pas demandé d'efforts supplémentaires à l'inverse d'un langage dynamique.

Evolution du programme Groovy par le binôme Kotlin

La modification du logiciel Groovy a été une expérience en demi-teinte. En effet, si l'ajout de fonctionnalités paraissait assez simple de prime abord, il a fallu batailler longtemps avec le casse tête invisible que représentait ce que nous nommerons ici "la logique cachée du code". Les modifications relativement aisées à mettre en place comme l'ajout d'un nouveau type de soin ont été rapidement implémentées. En revanche, le passage de deux valeurs "client" et "contrat" à une seule valeur "dossier" de composition [LettreContrat-NuméroClient] s'est avéré particulièrement ardu.

Il a fallu faire un usage intensif du débogueur afin de détecter des problèmes engendrés par les modifications qui n'étaient pas remontés par le compilateur. La flexibilité d'un langage dynamique tel que Groovy nous desservait puisqu'elle nous autorisait à briser la logique cachée du code sans pour autant lever une quelconque alarme, contrairement au développement en Kotlin, qui lui permettait de remonter rapidement les rares bogues générés.

Ainsi, l'ajout de fonctionnalités a demandé bien plus de temps et d'investissement pour un débutant confronté à ce nouveau langage. Et dans ce cas de figure, les aspects de Groovy qui pourraient lui donner un avantage sur Kotlin, comme la métaprogrammation, ne sont d'aucun secours.

Temps nécessaire au développement

	Logiciel Kotlin	Logiciel Groovy
Apprentissage du langage	3-4 heures	2-3 heures
Réalisation du logiciel	6 heures	5 heures
Demande de changement	3 heures	1-2 heures

Conclusion

À partir des informations développées dans le présent rapport, nous sommes en mesure de dire que Kotlin comme Groovy sont des langages faciles à prendre en main pour tout développeur ayant pratiqué la programmation orientée objet. La courbe d'apprentissage étant particulièrement douce dans les deux cas, l'impact du langage sur la vélocité initiale d'une équipe de développement n'ayant jamais utilisé ces technologies n'est pas important.

La réalisation, en revanche, est impactée par les concepts nouveaux développés dans les deux langages. On notera particulièrement que le binôme développant en Groovy a rencontré certains bogues tenaces lors du développement, ce qui n'a pas été le cas en Kotlin. Le développement en lui même a été facilité par la syntaxe intuitive et directe dans les deux cas. Le passage de l'idée de conception initiale au code s'est fait très naturellement. Ce qui laisse à penser que le secret de la simplicité de réalisation ne réside pas dans les attributs propres à la programmation dynamique.

L'hypothèse avancée ici, se basant sur l'observation des deux programmes, est que la réduction, voire la suppression du code sans logique d'affaire intrinsèque (par exemple, les accesseurs et les mutateurs), couplée à une syntaxe claire et à des concepts de programmation comme l'inférence du type est ce qui permet réellement de faciliter la production de code.

En ce qui concerne la maintenabilité, nous avons observé que même si les langages de programmation dynamique favorisent la rapidité et le développement à court terme ils donnent naissance à un code disposant d'une nouvelle dimension de complexité. Code qui a peu de chances d'être maintenable à long terme en dehors du cercle des développeurs initialement présents sur le projet. On notera ici la différence avec le Kotlin, qui, bien que légèrement plus exigeant lors du développement, engendre un code clair, exempt de bogues récurrents et facilement maintenable même après un changement au sein de l'équipe.

Nous conclurons donc cette analyse sur les constatations suivantes :

Si Groovy se prête bien au développement de preuves de concepts rapides et efficaces, et à la programmation multi-langage, on peut en dire autant de Kotlin. Kotlin présente les mêmes concepts de simplification et d'abstraction poussée du code, ce qui facilite et la réalisation, et la maintenance. De plus, Kotlin semble offrir davantage de robustesse naturelle et de protection contre les erreurs de programmation et les anti-patrons les plus communs. Nous estimons donc que Kotlin est un langage intrinsèquement plus maintenable que Groovy et que les applications développées avec ce langage ont plus de chance d'être lisibles et modifiables sur le long terme.

Groovy conserve quelques avantages majeurs, comme des éléments de meta-programmation et d'intégration facilitée du Json, mais en dehors des projets où ces concepts sont nécessaires et bien mis à profit, Kotlin garde une longueur d'avance. Nous recommandons donc l'utilisation de Groovy dans le cas où l'équipe connaît déjà le langage, ou bien si elle souhaite produire une application nécessitant explicitement et extensivement des éléments de meta-programmation. Nous recommandons Kotlin dans tous les autres cas.

Finalement, en prenant en compte que les langages dynamiques nous offre des moyens qui nous permettre d'écrire le code d'une façon plus rapide avec par exemple la flexibilité de

typage des variables, la disparition de code structurel comme les accesseurs et mutateurs, nous croyons que cette mentalité favorise une écriture désinvolte du code et cela transférerait ainsi une dette technique plus importante aux mainteneurs. On peut bien sûr mitiger ce problème si les programmeurs font vraiment preuve de rigueur ou si le logiciel est de petite taille. Dans le même ordre d'idée et en général, les langages statiques nous forcent à n'avoir qu'un type par variable et transmettent donc un code plus explicite aux prochains qui travailleront sur le code, favorisant ainsi sa maintenance.

Références

[1] Documentation Kotlin : https://kotlinlang.org/docs/reference

[2] Koans, apprentissage au travers d'exercices guidés : https://play.kotlinlang.org/koans/overview

[3] Apache Groovy : http://groovy-lang.org

[4] Groovy gaining popularity : https://https://bbvaopen4u.com/en/actualidad/why-groovy-gaining-popularity-among-java-developers

Annexes

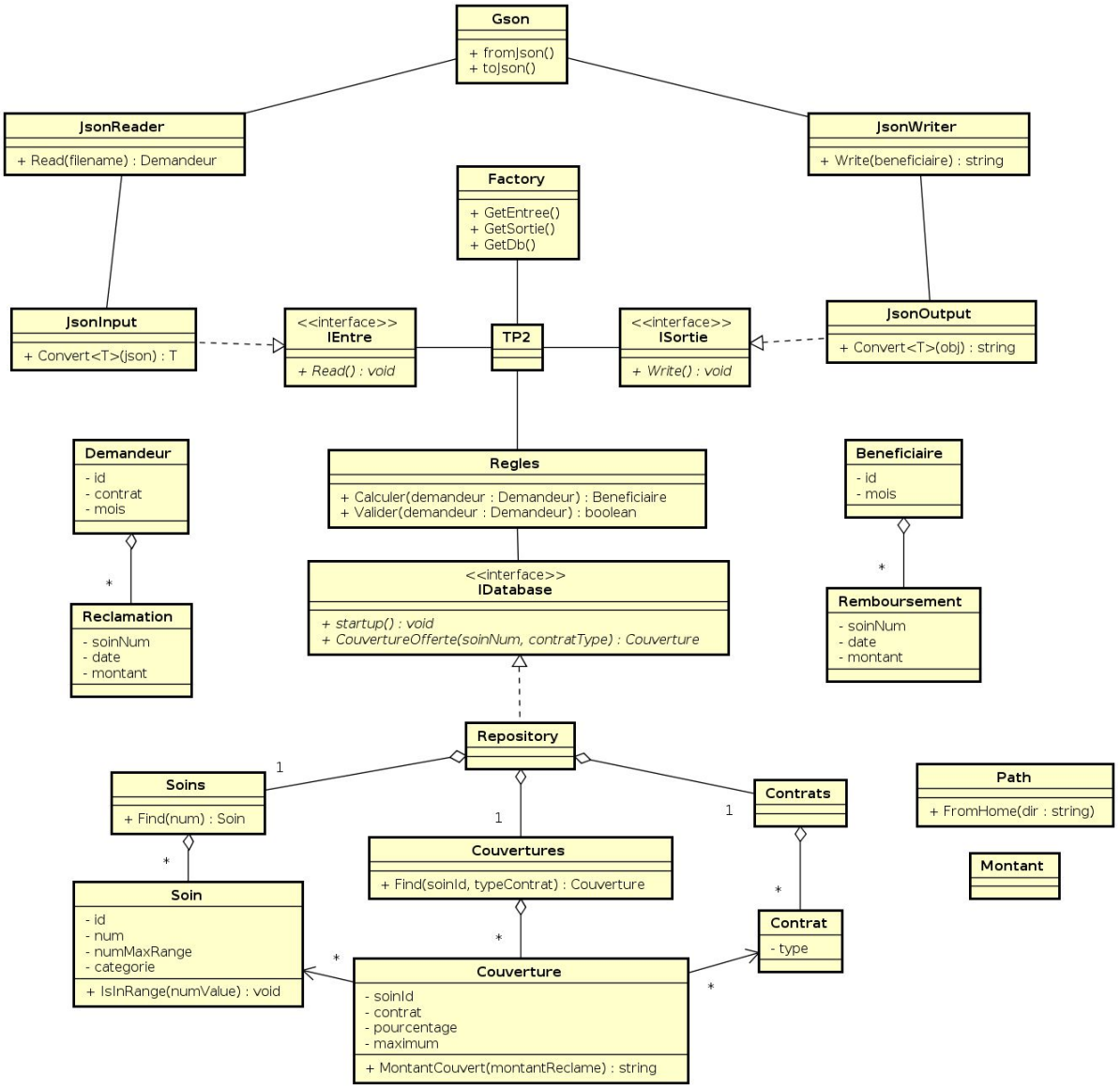

Diagramme de classes du programme Kotlin