UQAC
Université du Québec
à Chicoutimi

Année universitaire : 2017/2018
Session d'été

DÉVELOPPEMENT AVANCÉ DE LOGICIELS : PATRONS ET MODÈLES

– Projet de fin de session –

Auteurs :
Matthieu LE BOUCHER
Martin ROHMER

Table des matières

1 Article étudié

L'article présenté était *When and Why Your Code Starts to Smell Bad* de TUFANO et al.[6], qui s'inscrit dans la recherche et le développement d'approches et d'outils pour détecter les *code smells* le plus tôt possible dans un projet logiciel et déclencher des opérations automatiques de ***refactoring***. Le but ultime de cet article est de permettre de réduire la dette technique.

1.1 Limites et *future work*

Cet article se veut pionnier dans le domaine de la prédiction de *code smells*. Il offre des pistes de réflexion et permet quelques observations intéressantes, néanmoins il ne s'agit que d'une première approche effectuée dans ce domaine. En tant que telle, elle a des limites évidentes. Tout d'abord, sa dépendance à la précision d'un outil externe afin d'identifier les *code smells*. Ensuite, le principal problème de l'approche proposée dans l'article se tient au niveau du temps de calcul. Ce temps résulte de plusieurs facteurs dont les principaux sont la quantité de code analysée, les formules mathématiques utilisées et le nombre de paramètres à prendre en compte. Ainsi, en limitant les *code smells* étudiés au nombre de 5, observés selon 7 métriques différentes, cela leur prit 8 semaines afin d'extraire les données utiles.

Cela induit plusieurs limites : premièrement, le temps nécessaire à cette approche ne la rend donc vraisemblablement pas applicable dans le cadre du *development stage* d'un projet. Deuxièmement, le choix des paramètres utilisés est restreint, limitant également les observations possibles.

Les intérêts des futurs travaux possibles dans ce domaine se trouveraient donc principalement dans la variation de ces paramètres, à savoir les ensembles de code utilisés, les *code smells* étudiés et les métriques prises en considération. Il résulterait de la somme de tels travaux des observations qui permettraient d'identifier de manière de plus en plus précise des méthodes de prédiction applicables à chaque *code smell*.

1.2 Technique d'adaptation de l'article

L'article propose des règles à appliquer principalement **lors des phases de conception et de développement** pour l'adaptation du logiciel par la technique de *refactoring*. Ces techniques doivent être conservées à l'esprit et appliquées **tout au long du projet** pour en garantir la pérennité. En particulier, l'article su mettre en évidence les points suivants, auxquels il nous faut particulièrement prêter attention :

— à la création du code, bien évaluer l'ensemble des métriques classiques considérées, car un *smell* apparaît souvent dès la création du code concerné ;
— lors du *refactoring* ou du débugage, **attention à ne pas introduire de *smells*** ;
— les *deadlines* — et la pression qui va avec — favorisent l'apparition de *smells* ;
— les *smells* peuvent apparaître dès le début d'un projet ;
— une charge de travail élevée favorise la capacité d'un développeur à introduire un *smell*.

1.3 Résumé solution et état de l'art

Avec comme intérêt principal la réduction de la dette technique omniprésente, l'article a basé son approche sur des techniques et outils qui font encore maintenant l'objet de travaux. Entre autres, la détection des *code smells* ainsi que les opérations de *refactoring*. Mais là où les travaux précédents se concentrent sur l'approfondissement des méthodes permettant la détection des *smells*, leur objectif ici était de s'intéresser à la prédiction et l'anticipation de ceux-ci, avec l'idée qu'*il vaut mieux prévenir que guérir*.

De leur étude a pu résulter l'extraction de connaissances générales, résumées dans la section précé-

dente. Des idées de pistes ont également été amorcées, qui permettraient à terme le développement d'outils permettant de prévenir l'apparition de *bad smells*. De tels outils, utilisés parallèlement à un système d'intégration continue par exemple — comme Jenkins, Travis ou TeamCity —, serviraient à prévenir les développeurs de la mauvaise tournure qu'est en train de prendre leur code. En étant prévenu au plus tôt, avant l'apparition réelle des *smells*, le coût nécessaire à leur correction serait moindre qu'une fois lesdits *smells* parvenus à terme.

2 Projet final

2.1 Problématique

Notre objectif initial est de créer une application *context aware* et distribuée qui interagit avec d'autres interfaces, *si possible à l'aide d'un nouveau langage ou framework afin de nous former à d'autres technologies qu'Android et d'élargir notre panel de compétences.* Pour cela, nous proposons de nous intéresser à un cas concret : **une application qui fournit des données météorologiques sur la position géographique où se trouve l'appareil mobile.**

Cette étude de cas soulève deux sous-problèmes qui nous intéressent dans le cadre du cours :

1. comment obtenir une partie du contexte de l'appareil, ici sa position géographique grâce à son GPS intégré ?
2. comment communiquer avec une API externe pour obtenir d'autres informations sur ces données ?

Bien entendu, suite aux conclusions établies après l'étude de notre article — cf. section (**1**) —, on s'attachera durant tout le projet à respecter toutes les bonnes règles de programmation ainsi que les patrons architecturaux vus en cours.

2.2 Solution

L'application s'appelle CARACAS, un acronyme maladroit pour *Context Aware Application And Stuff.* Le code source est accessible sur GitHub : `https://github.com/Meight/Caracas`.

2.2.1 Technologies utilisées

Après de longues recherches, nous avons décidé de baser l'application sur Flutter, un *framework* écrit dans le langage Dart [1] et principalement développé par GOOGLE [2].

Dart est un langage lui aussi développé par Google, avec une syntaxe claire et concise proche du Java, du C ou du JavaScript et disposant de nombreuses librairies et *packages*. Il est particulièrement intéressant car il est adapté à la programmation réactive (à la manière d'autres *frameworks* récents à l'image de React Native, utilisé par Facebook, par exemple) avec la gestion simplifiée de composants à cycle de vie très court.

Remarque 1

> *Un point important qui nous intéresse est que Dart permet de gérer facilement les appels asynchrones, dont nous aurons besoin.*

Flutter est un *framework* qui tire toute la puissance du langage Dart en matière de programmation réactive. Il s'agit du *Software Development Kit* de Google pour construire des applications mobiles natives compatibles avec iOS et Android avec le même code, ce qui représente un énorme avantage en terme de coût de développement, mais aussi en terme de dette technique comme l'expliquait notre article. Son approche clé-en-main permet de créer des interfaces utilisateur très simplement — plus simplement que sous Android par exemple — en injectant facilement des *widgets* préconfigurés.

1. `https://www.dartlang.org/`

Remarque 2

Flutter offre une fonctionnalité très intéressante pour le développement : le ***hot reloading.*** *Le framework écoute les enregistrements de fichiers pour détecter les modifications, et lorsque c'est le cas, il est capable de déterminer quelles parties de l'application ont changé, d'effectuer ces modifications à la volée et de mettre l'application à jour sans interrompre son exécution. Il n'est donc pas nécessaire de recompiler toute l'application à chaque modification, ce qui fournit également un gain de temps considérable et permet d'observer ses modifications en temps réel !*

2.2.2 Obtention du contexte

Flutter fournit un accès simplifié aux capteurs du téléphone via une couche d'abstraction car iOS et Android ne fonctionnent pas de la même manière pour l'accès aux composants internes du téléphone, notamment pour la gestion des privilèges.

En ce qui nous concerne, il sera donc relativement aisé d'accéder à toutes les informations géographiques de l'appareil, à l'aide par exemple de la librairie `geolocation` de Loup Inc. [3].

Listing 1 – Obtention de la localisation de l'appareil en temps réel.

```
1    LocationResult result = await Geolocation.lastKnownLocation();
2
3    if (result.isSuccessful) {
4        double latitude = result.location.latitude;
5        double longitude = result.location.longitude;
6    }
```

2.2.3 Obtention des informations météorologiques

Pour obtenir ces informations, nous n'avons d'autre choix que de recourir à un service externe. Il en existe plusieurs mais nous avons choisi l'API d'OpenWeather [4], qui offre différentes interfaces mais une qui nous intéresse en particulier : obtention par coordonnées géographiques.

Listing 2 – Exemple de réponse au format JSON de l'API pour une longitude de 139 et une latitude de 35.

```
1  {
2     "coord":{
3        "lon":139,          -- longitude --
4        "lat":35            -- latitude --
5     },
6     "sys":{
7        "country":"JP",     -- pays --
8        "sunrise":1369769524,
9        "sunset":1369821049
10    },
11    "weather":[
12       {
13          "id":804,
14          "main":"clouds",
15          "description":"overcast clouds",
16          "icon":"04n"
17       }
18    ],
19    "main":{
```

```
      "temp":289.5,      -- temperature (en kelvin) --
      "humidity":89,
      "pressure":1013,
      "temp_min":287.04,
      "temp_max":292.04
   },
   "wind":{
      "speed":7.31,      -- vitesse du vent --
      "deg":187.002
   },
   "clouds":{
      "all":92           -- pourcentage de nuages --
   },
   "dt":1369824698,
   "name":"Shuzenji",
}
```

Remarque 3

Notre objectif étant davantage de comprendre les interactions que de réaliser une application météo complète (nous n'avons pas la prétention de concurrencer les applications professionnelles), nous n'utiliserons qu'une poignée de ces informations dans notre UI.

2.2.4 Gestion des dépendances externes

Pour ce projet, nous avons décidé d'utiliser pour la gestion de dépendances l'outil d'assemblage Gradle [2] en nous basant sur l'introduction proposée dans *Gradle for Android* par PELGRIMS [5]. Gradle fournit — entre autres — un ensemble de directives pour l'injection de dépendances externes.

De plus, Dart possède son propre sur-ensemble de gestion de dépendances externes, sous la forme d'un simple fichier de configuration au format YAML appelé `pubspec.yml`.

2.2.5 Injection de dépendance par conteneurs

Dans le but de conserver une meilleure modularité, l'application vit à l'intérieur d'un conteneur qui gère les dépendances. Pour cela, on utilise la librairie `dioc` proposée par DENIEL [1]. L'idée est de décentraliser la gestion de l'API appelée pour obtenir les informations sur la météo du cœur de l'application ; on peut ainsi facilement changer d'API.

C'est utile notamment dans la phase de développement car, malgré les tests, on ne veut pas bombarder l'API réelle de requêtes parfois buguées.

Pour palier à ce problème, on définit donc plusieurs conteneurs qui vont injecter une implémentation différente de notre fournisseur de météo suivant la phase du projet. À l'intérieur du conteneur, toute référence à un `WeatherProvider` sera donc implémentée d'après cette injection. De cette façon, aucune modification du code à l'intérieur du conteneur n'est nécessaire.

Listing 3 – Déclaration des conteneurs.

```
abstract class AppBootsrapper extends Bootsrapper {
    // Conteneur pour le developpement : on utilise une API factice.
    @Provide(WeatherProvider, MockWeatherProvider)
```

2. Voir https://gradle.org/ pour en savoir plus.

```
4       Container development();
5
6       // Conteneur pour la production : on utilise la vraie API.
7       @Provide(WeatherProvider, OpenWeatherMapProvider)
8       Container production();
9   }
```

Pour effectuer l'injection, il n'y a plus qu'à enregistrer (*register*) l'instance du service associée au type voulu (cf. lignes 12 et 21), en faisant par exemple :

Listing 4 – Enregistrement des services à l'intérieur des conteneurs.

```
1       // Instanciation d'un conteneur de base pour tous les environements.
2       Container base() {
3           final container = new Container();
4           return container;
5       }
6
7       /* Enregistrement de l'instance de MockWeatherProvider sur le type
        WeatherProvider.
8       ** Valable dans tout le conteneur.
9       */
10      Container development() {
11          final container = base();
12          container.register(WeatherProvider,
13              (c) => new MockWeatherProvider());
14          return container;
15      }
16
17      /* Enregistrement de l'instance de OpenWeatherMapProvider sur le type
        WeatherProvider.
18      ** Valable dans tout le conteneur.
19      */
20      Container production() {
21          final container = base();
22          container.register(WeatherProvider,
23              (c) => new OpenWeatherMapProvider());
24          return container;
25      }
```

Enfin, il n'y a plus qu'à *build* le conteneur voulu pour la phase actuelle du projet (dans notre cas, c'est le conteneur `production` qui nous intéresse afin de tester l'API réelle).

Listing 5 – Obtention du conteneur de production au sein de l'application puis du service *provider* injecté dans ce conteneur.

```
1       // Construction du conteneur "production".
2       final Container container = AppBootsrapperBuilder.build().production();
3
4       // Obtention du singleton vers le service. L'instance obtenue est celle que l'on
        a enregistree ligne 23 dans le listing 2.
5       WeatherProvider _weatherProvider = container.singleton(WeatherProvider);
```

Remarque 4

> *Comme on peut le voir, à aucun moment au sein de l'application n'apparaît les implémentations de l'interface WeatherProvider :* ***ce choix est totalement découplé et injecté depuis l'extérieur par le conteneur.***

2.2.6 Fonction centrale finale

L'application tourne au final autour d'une fonction asynchrone d'obtention des données météos étant donné les coordonnées géographiques de l'appareil : `fetchWeatherData()`.

```
// Fonction asynchrone qui retourne une promesse de type WeatherData.
Future<WeatherData> fetchWeatherData() async {
    // Obtenir la localisation.
    LocationResult result = await Geolocation.lastKnownLocation();

    if (result.isSuccessful) {
        // Mise a jour de l'etat (propre a Flutter) !
        setState(() {
            _weatherProvider
                // Obtenir les donnees meteo selon l'instance du provider :
                .fetchWeatherData(
                    result.location.latitude,
                    result.location.longitude
                )
                // Lorsque la requete asynchrone est finie (fonction lambda) :
                .then((onValue) => weatherData = onValue);
        });

        return weatherData;
    }

    // Pas pu obtenir les infos de localisation.
    // Afficher un message d'erreur...

    return null;
}
```

2.3 Limites et améliorations

Notre objectif premier pour ce projet était la découverte et la mise en place de nouvelles technologies, et il est donc réussi en ce sens. En mettant de côté l'utilité moindre de l'application (compte-tenu des multiples applications météo déjà présentes et offrant beaucoup plus de fonctionnalités), nous avons tout de même relevé certains points qui pourraient être améliorés. Tout d'abord, l'utilisation d'une tierce partie qui nous rend dépendant de ses propres erreurs et limites. Les appels à l'API sont en plus limités. Ensuite, l'utilisation du GPS du téléphone est coûteux pour l'appareil. Il nous faudrait mieux gérer les cycles de mises à jour de la position et des requêtes auprès de l'API afin d'optimiser leur utilité et les ressources du smartphone.

A côté de ça, nos pistes d'amélioration seraient d'utiliser une seconde API pour proposer des prévisions des conditions météorologiques. Nous pourrions prendre toutes ces informations en considération afin d'effectuer des recommandations à l'utilisateur. Ces recommandations concerneraient par exemple les activités à proposer à l'utilisateur, une tenue adaptée au temps actuel et prévu etc.

2.4 Captures d'écran code commenté

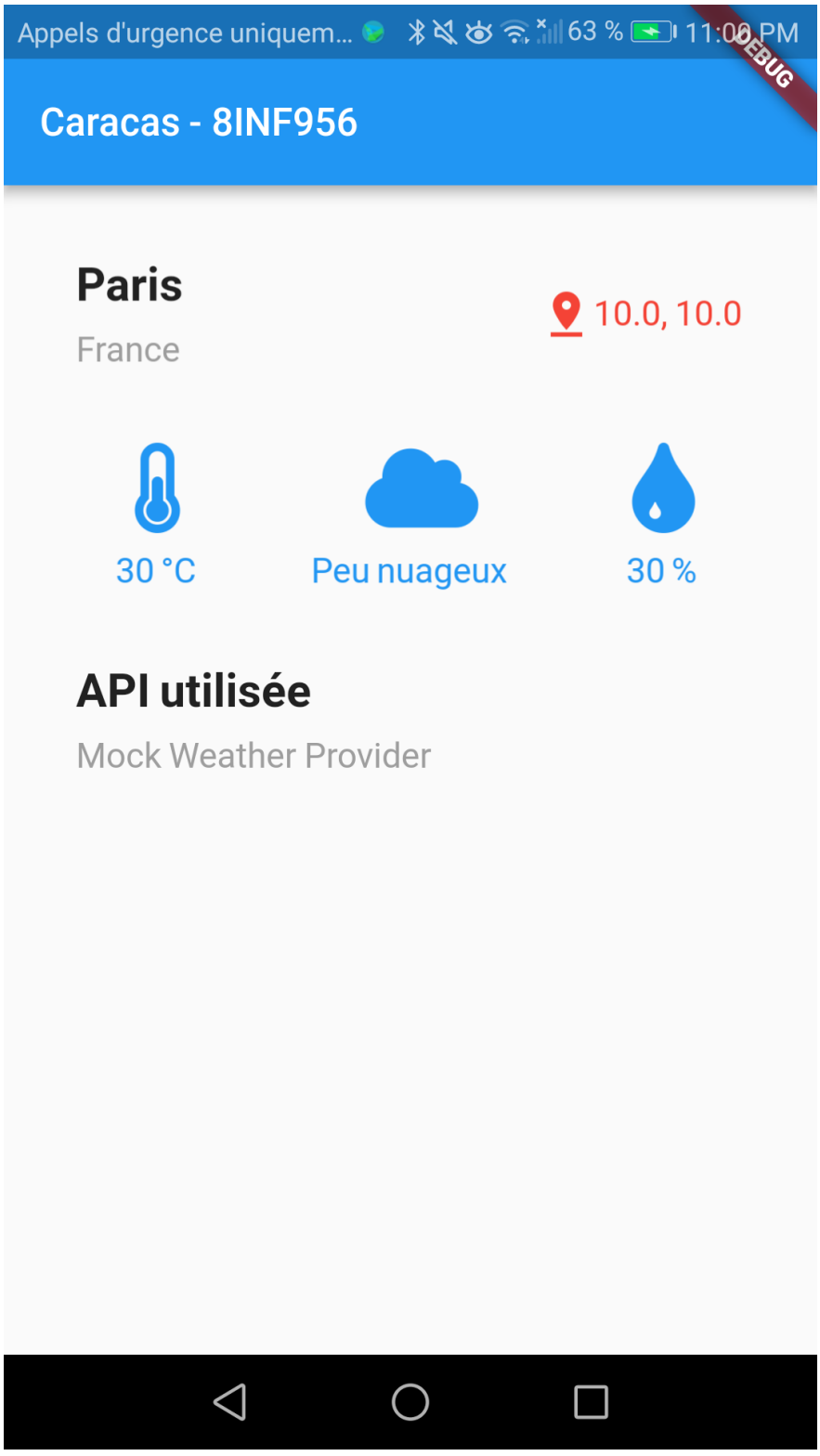

FIGURE 1 – Capture d'écran de l'application en mode ***development***, utilisée à Chicoutimi. L'API factice injectée par ce conteneur, appelée `MockWeatherProvider`, renvoie toujours la même réponse (une météo ydillique à Paris).

FIGURE 2 – Capture d'écran de l'application en mode ***production***, utilisée à Chicoutimi. L'API injectée par ce conteneur, appelée `OpenWeatherMapProvider`, utilise les coordonnées réelles actuelles.

Références

[1] Aloïs DENIEL. *Inversion of control based on dependency injection through containers*. 2018. URL : https://pub.dartlang.org/packages/dioc (visité le 17/06/2018).

[2] GOOGLE. *Build beautiful native apps in record time*. 2018. URL : https://flutter.io/docs/ (visité le 17/06/2018).

[3] LOUP INC. *Flutter geolocation plugin for Android API 16+ and iOS 9+*. 2018. URL : https://pub.dartlang.org/packages/geolocation (visité le 17/06/2018).

[4] OPENWEATHER. *Current weather and forecasts in your city*. 2018. URL : https://openweathermap.org/ (visité le 17/06/2018).

[5] Kevin PELGRIMS. *Gradle for Android*. Packt Publishing, 2015. ISBN : 978-1-78398-682-8.

[6] Michele TUFANO et al. "When and Why Your Code Starts to Smell Bad". In : *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy : IEEE Press, 2015, p. 403–414. ISBN : 978-1-4799-1934-5. URL : http://dl.acm.org/citation.cfm?id=2818754.2818805.