

Rapport ACO 2019-2020

1) Introduction

Le but de ce tp était de mettre en pratique nos compétences en ingénierie logiciel en implémentant le programme « CarTaylor ». Il s'agit grossièrement d'un configurateur de voiture. Nous avons rencontré plusieurs difficultés majeures, entre les conflits git, les deads lines serrées avec les autres matières et le temps que nous avons pris à bien cerner les enjeux du sujet. Ce rapport détaille la manière dont nous avons traité le problème. Bonne lecture :)

2) Architecture

1) Les enjeux de souplesse :

La majeure difficulté que nous avons rencontrée dans l'implémentation de la V2 est la direction à prendre quand à l'architecture du projet. En effet, il y avait beaucoup de manières de faire, le but étant de fabriquer un outil flexible, et possédant des données assez dynamique au point que le développeur puisse en ajouter / supprimer facilement .

En cela, l'utilisation d'enum est un bon point de départ pour définir quelles catégories sont présentes dans le programme. Seul les catégories présentes dans enums peuvent exister, et, inversement, si le développeur rajoute une catégorie dans l'enum, elle est automatiquement rajoutée aux catégories du 'configurator' :

```
for (CategoryType categoryType : CategoryType.values())  
{  
    categories.put(categoryType, new  
        CategoryImpl(categoryType.name()));  
}
```

Ensuite il était important de pouvoir lier un Part et son PartType simplement, ainsi, nous avons modifié la fonction `newInstance()` de `PartTypeImpl` afin de réaliser cette association :

```
public PartImpl newInstance()  
{  
    Constructor<? extends PartImpl> constructor;  
    try  
    {  
        constructor = classRef.getConstructor();  
        PartImpl part = (PartImpl)constructor.newInstance();  
        part.setType(this);  
        return part;  
    }  
    catch (Exception e)  
    {  
        Logger.getGlobal().log(Level.SEVERE, "constructor call failed", e);  
        System.exit(-1);  
    }  
    return null;  
}
```

Nous avons pensé en premier lieu à utiliser MongoDB afin de pouvoir gérer la structure des types de l'extérieur. Pouvoir éditer les données et en rajouter sans même avoir à toucher au code source ! Cependant, par soucis de temps, nous avons dû renoncer à cette implémentation .

Nous avons donc conçu Car Taylor comme un gestionnaire de données. Il n'y a pas de GUI dans l'implémentation JAVA. Seulement une console qui permet de gérer les données, (modifier la configuration courante, modifier les incompatibilités, exporter les données en HTML). Notre implémentation contenait donc toute la logique de Car Taylor.... mais ce n'était pas très 'user-friendly' ...

2) Le pattern command

Afin d'apporter plus de souplesse à notre implémentation, nous avons utilisé le pattern command.

```
@Command(name ="SELECT", role="User")
public static void SelectPart(String param) throws InvalidParameterException
```

Voici un exemple de commande. La fonction SelectPart est associée à la commande SELECT via le biais des annotations java. L'annotation contient également le rôle minimum requis pour pouvoir exécuter la commande. A l'aide de la reflection, nous avons créé un mappage entre la fonction et la commande. Ainsi, l'utilisateur peut interagir avec le programme par le biais de la console :

```
SELECT EG100
|déc. 18, 2020 8:34:56 PM commands.Commands SelectPart
INFO: Part EG100 selected.
```

Toujours pas très user friendly hein ...

Cependant, si un utilisateur peut taper une commande par le biais de System.in , une application peut le faire aussi ! Ainsi nous avons développé un GUI en C# , qui n'implémente **AUCUNE** logique ! L'application tiens en une cinquantaine de ligne. Il sert juste d'interface et exécute simplement les commandes que nous avons défini dans Car Taylor !

```
/*
 * Ecrit 'CATEGORIES' dans le flux d'entrée du processus CarTaylor en cours d'exécution.
 * CarTaylor traite la commande et écrit dans System.out !
 * Nous récupérons ensuite cette valeur ... et le tour est joué !
 */
string result = CarTaylorHooks.Get("CATEGORIES");
```

Cette implémentation a deux avantages : en premier lieu elle définit clairement les fonctionnalités de l'application Car Taylor à travers la liste des commandes (qui constitue une sorte d'API de CarTaylor). Et ensuite , elle dissocie totalement la logique de la vue utilisateur ! On peut refaire une interface utilisateur , quelque soit le langage, en utilisant le jar de Car Taylor. Et réciproquement, on peut modifier l'implémentation de CarTaylor sans modifier le GUI ! Voila un exemple de rendu graphique en C# (les source sont dans GUI/Sources/ , l'exécutable pour Windows dans GUI/Binaries/, lire le README !)

The screenshot displays the CarTaylor GUI interface. It features a 'Categories' list on the left with options: Engine, Exterior, Transmission, and Interior. To the right is a 'Variants' list with options: EH120, ED180, EG100 (highlighted), EG133, EG210, and ED110. Further right are three buttons: 'Select part', 'Is Valid', and 'Is Complete'. Below these is a 'Current Configuration' section showing a list of selected parts: EH120 (Engine), IH (Interior), XS (Exterior), and TM6 (Transmission). To the right of this list is a 'Remove Part' button. At the bottom right is an 'Export Configuration' button.

Commandes de CarTaylor

Nom	Description	Rôle minimum requis
SELECT	Sélectionne une partie dans la configuration courante.	Utilisateur
COMPLETE	Écrit 'true' dans Sytem.out si la configuration est complète, 'false' sinon.	Utilisateur
VALID	Écrit 'true' dans Sytem.out si la configuration est valide, 'false' sinon.	Utilisateur
EXPORT	Exporte la configuration courante dans un fichier HTML nommé Commands.ExportFilename	Utilisateur
CATEGORIES	Ecrit toute les categories de part possible dans System.out	Utilisateur

VARIANTS	Obtiens les parts possible en fonction d'une catégorie	Utilisateur
VIEW	Ecrit la configuration courante dans System.out	Utilisateur
UNSELECT	Supprime une partie en fonction de sa catégorie dans la configuration courante	Utilisateur
ADDICMP	Ajoute une incompatibilité entre deux parties.	Admin
RMICMP	Supprime une incompatibilité entre deux parties	Admin
ADDREQ	Ajoute un prérequis entre deux parties	Admin
RMREQ	Supprime un prérequis entre deux parties	Admin

3) Tests

Pour les tests nous avons procédé avec une stratégie simple : Nous avons d'abord testé nos classes et méthodes les plus simples, pour aller vers les plus compliquées. En procédant de la sorte, on avance pas à pas.

Afin d'assurer la fiabilité de nos tests, nous avons utilisé l'outil coverage qui permet d'afficher la quantité de code testé. Les parties en vert indiquent que ces parties du code ont été testées. Celles en rouge que le test n'est jamais passé par là ! Les lignes jaunes indiquent qu'on n'a pas parcouru l'intégralité de l'expression conditionnelle (ce qui explique que l'exception soit soulignée en rouge ci dessous !)

```
@Override
public void selectPart(Part chosenPart) throws InvalidParameterException
{
    if(this.configurator.getVariants(chosenPart.getCategory()).contains(chosenPart.getType()))
    {
        if (!parts.containsKey(chosenPart.getCategory()))
        {
            parts.put(chosenPart.getCategory(), chosenPart);
        }
        else
        {
            parts.replace(chosenPart.getCategory(), chosenPart);
        }
    }
    else
    {
        throw new InvalidParameterException("The Part " + chosenPart + " is not refer in our catalog.");
    }
}
```

element	Coverage	Covered Instru...
▼ ACOTP2	59,7 %	1341
▼ src	59,7 %	1341
▼ impl	57,5 %	465
> ConfigurationImpl.java	51,6 %	127
> CompatibilityManagerImpl.java	12,3 %	13
> PartImpl.java	38,8 %	50
> ConfiguratorImpl.java	74,7 %	109
> PartTypeImpl.java	71,1 %	32
> Session.java	97,6 %	122
> CategoryImpl.java	100,0 %	12

L'outil coverage nous fournit également une interface graphique contenant des informations plus générale pour voir si une classe a été entièrement testée ou non. On voit que CategoryImpl.java l'est contrairement aux autres.

▼ commands	50,6 %	280
> Commands.java	41,7 %	175
> CommandsManager.java	77,4 %	96
> CommandInfo.java	100,0 %	9
▼ impl	81,4 %	1012
> PartImpl.java	38,8 %	50
> ConfigurationImpl.java	76,5 %	208
> ConfiguratorImpl.java	76,5 %	127
> CompatibilityManagerImpl.java	83,8 %	192
> PartTypeImpl.java	77,8 %	35
> Session.java	99,2 %	388
> CategoryImpl.java	100,0 %	12

Malheureusement, nous n'avons pas pu tester toutes nos classes. Les classes relatives aux commandes par exemple : certaines des commandes écrivent le résultat dans le flux de données de sortie de la console Eclipse, auquel JUnit n'a pas accès. Cependant, ces commandes font un appel (presque direct) aux classes ConfigurationImpl, ConfiguratorImpl etc. Ces classes ont été testées, donc on suppose qu'elles devraient se comporter correctement ! Nous avons également ignoré les fonctions qui nous paraissaient trop triviales (toString(), getName() { return name ; } etc.

En outre, le GUI en C# nous a permis de réaliser des tests complémentaires manuellement. Grâce aux tests nous avons pu repérer plusieurs erreurs dans notre implémentation (parfois subtiles!)

4) Pour aller plus loin ... Il faut avancer

- Nous pourrions améliorer CarTaylor de la manière suivante : type de Part stockés dans une base de données NoSQL (nous pourrions utiliser les méthodes de PropertyManager pour créer nos attributs dynamiquement (en fonction des résultats dans la base de données), donc créer un mappage entre les documents NoSQL et les Part. De cette manière, nous pourrions étendre la liste des commandes. exemple : `'ADDPART field1 field2 field3 field4 field4 ...'`. Cela éviterait de spécifier dans le code source les propriétés de chaque parties et leur valeurs par défaut ! Ex :

```
super.addProperty("gas", getter,setter,possibleValues);

getter = () -> getPower();
setter = (String value) -> setPower(value);
possibleValues = new HashSet<String>();
super.addProperty("consumption", getter,setter,possibleValues);

@Override
protected void initialize()
{
    this.gasType = GasType.Diesel; // aled
    this.power = "110kW";
}
```

- Nous pourrions ensuite créer un GUI pour l'utilisateur et un GUI pour l'administrateur, et mettre en place un petit système d'authentification via une commande.
- Quitte à utiliser une base de données... Quitte à utiliser des commandes ... Pourquoi ne pas faire de CarTaylor un serveur ? Les clients (GUI) pourraient communiquer avec le serveur (CarTaylor) via l'API (Commands). Cela imposerait néanmoins une bonne gestion de la concurrence pour éviter les soucis lors de l'accès à nos données !
- Réécrire le HtmlWriter afin de générer un HTML plus beau, plus propre, en utilisant du CSS.

Bonne journée / soirée !