Compte rendu du TP CarTaylor

Akoto Yao-Arnaud,, Amiard Anthony

18 décembre 2020

Table des matières

1 Présentation du projet

Le but de ce projet est de créer un configurateur d'automobile, plus précisément son *back-end* en Java.

Le configurateur permet à l'utilisateur, pour un ensemble de catégories, de choisir parmi plusieurs variantes. Par exemple, pour la catégorie moteur, l'utilisateur peut choisir entre un moteur essence, un moteur diesel, ou un moteur électrique ; pour la catégorie transmission, entre une boîte manuelle, une boîte automatique où une transmission adaptée à un moteur électrique.

Les variantes peuvent avoir des incompatibilités entre elles et une variante peut en requérir d'autres. Le configurateur gère des notions de compatibilités et de *requirements* entre les différentes parties, et permet à l'utilisateur de savoir si sa configuration est valide, ou non.

Dans une deuxième version, l'utilisateur peut également modifier certaines propriétés des différentes variantes, comme sa couleur par exemple.

L'architecture de ce projet est indépendante du jeu de données qu'elle manipule, et permet donc une plus grande souplesse dans son cadre d'utilisation. On peut même imaginer l'application être utilisée en dehors du cadre automobile.

2 Organisation de l'application

Le projet est séparé en deux grandes parties : une interface publique `fr.istic.cartaylor.api`, utilisable pour le *front-end*, et un ensemble de classes d'implémentation `fr.istic.cartylor.implementation`.

2.1 Interface publique

L'interface publique contient différents types qui permettent de représenter les données de l'application :

- `Configurator`, le point d'entrée de l'interface qui permet d'accéder à la configuration de l'utilisateur et au catalogue de parties disponibles pour la configuration,
- `Configuration`, l'interface qui permet d'accéder et de modifier la configuration de l'utilisateur,
- `Category`, l'interface qui représente une catégorie de parties,
- `PartType`, l'interface qui représente une variante d'une catégorie de parties,
- `CompatibilityChecker`, une interface qui permet d'accéder aux liens de compatibilités et de *requirements* existants entre les parties,
- `CompatibilityManager`, une extension de `CompatibilityChecker` destinée aux administrateurs, qui permet de définir les incompatibilités et les *requirements* entre les différentes parties.

La deuxième version ajoute également ces interfaces :

- `Part`, la représentation d'une partie dont l'utilisateur peut modifier les propriétés,
- `PropertyManager`, l'interface utilisée par `Part` pour modifier ses propriétés.

2.2 Présentation de l'implémentation

L'implémentation de cette interface se divise en plusieurs classes :

- `ConfiguratorImpl`, qui implémente l'interface `Configurator`,
- `ConfigurationImpl`, qui implémente l'interface `Configuration`,
- `CategoryImpl`, qui implémente l'interface `Category`,
- `PartTypeImpl`, qui implémente l'interface `PartType`,
- `CompatibilityManagerImpl`, qui implémente `CompatibilityChecker` et `CompatibilityManager`.

La deuxième version ajoute également la classe `PartImpl`, qui implémente `PropertyManager` et `Part`.

L'implémentation définit une interface `Initializer` qui permet de définir le catalogue de parties utilisées par l'application, et les relations de compatibilité et de *requirements* entre ces parties.

3 Détails sur l'implémentation

3.1 `ConfiguratorImpl` et `Initializer`

Pour initialiser le configurateur, on doit passer en paramètre du constructeur `public ConfiguratorImpl(Initializer initializer)` un objet de type `Initializer` qui permet de stocker toutes les données du configurateur : catégories, parties ainsi que leurs incompatibilités, leurs *requirements* et leurs propriétés pour la deuxième version.

Pour définir un jeu donné, il faut donc implémenter l'interface *Initializer*. Cette interface comporte deux méthodes à implémenter :

- `Map<Category, Set<PartType>> getCatalog()`, qui retourne une table associant aux différentes catégories du configurateur l'ensemble des variantes disponibles,
- `void initCompatibilityManager(CompatibilityManager compatibilityManager)`, qui initialise le `CompatibilityManager` donné en appelant ses méthodes `addIncompatibilities` et `addRequirements`.

Exemple d'implémentation d'`Initializer` pour un catalogue contenant deux catégories `categoryA` et `categoryB`, avec deux variantes par catégories, respectivement `partTypeAA` et `partTypeAB`, et `partTypeBA` et `partTypeBB`. `partTypeAA` requiert `partTypeBA` et `partTypeAB` et `partTypeBA` sont incompatibles :

```
class Initialization implements Initializer {
        @Override
        public Map<Category, Set<PartType>> getCatalog() {
                return new HashMap<>() {{
                        put(categoryA, new HashSet<>() {{
                                add(partTypeAA);
                                add(partTypeAB);
                        }});
                        put(categoryB, new HashSet<>() {{
                                add(partTypeBA);
                                add(partTypeBB);
                        }});
                }};
        }

        @Override
        public void
        initCompatibilityManager(CompatibilityManager compatibilityManager){
                compatibilityManager.addRequirements(
                                partTypeAA,
                                new HashSet<>() {{ add(partTypeBA); }}
                );
                compatibilityManager.addIncompatibilities(
                                partTypeAB,
                                new HashSet<>() {{ add(partTypeBA); }}
                );
        }
}
```

3.2 `CompatibilityManagerImpl`

La classe `CompatibilityManagerImpl` implémente les interfaces `CompatibilityChecker` et `CompatibilityManager`, et permet de stocker les incompatibilités et les contraintes entre les variantes de différentes catégories.

Pour gérer les incompatibilités, une classe privée `Incompatibility` représente un couple entre deux inscances de `PartType` incomptatibles entre elles. L'incompatibilité étant réflective, la méthode `equals` fonctionne entre deux couples de types identiques quelque soit leur ordre.

Exemple :

```
(new Incompatibility(
        partTypeAB,
        partTypeBA
).equals(new Incompatibility(
        partTypeBA,
        pratTypeAB
))) // -> true
```

Cela permet de stocker les `Incompatibility` dans un objet de type `Set<Incompatibility>`. Ainsi une incompatibilité entre deux parties n'est stockée qu'une seule fois. Néanmoins, pour retrouver une incompatibilité pour un `PartType` donné, il faut vérifier les deux composants du couple. Exemple avec un extrait de la méthode `getIncompatibilities` qui utilise les méthodes fonctionnelles de Java pour récupérer les incompatibilités avec `reference` dans un tableau :

```
PartType[] array = incompatibilities.stream().filter(
        // Recherche du couple contenant reference
        (pair) -> pair.part1.equals(reference) || pair.part2.equals((reference))
).map(
        // Récupération du PartType autre que reference dans le couple
        (pair) -> pair.part1.equals(reference) ? pair.part2 : pair.part1
).toArray(PartType[]::new);
```

Pour les contraintes de partier en requiérant d'autres, ces relations sont stockées dans des couples de la classe `Requirement`. Cette fois-ci, comme les *requirements* sont à sens unique (une partie a besoin d'une autre, mais cette autre partie n'a pas nécessairement besoin de la première), l'ordre a une importance dans l'égalité entre deux couples :

```
(new Requirement(
        partTypeAA,
        partTypeBA
).equals(new Requirement(
        partTypeBA,
        pratTypeAA
))) // -> false
```

Récuperer la liste des *requirements* pour un `PartType` est donc beaucoup plus simple :

```
PartType[] array = requirements.stream().filter(
        // Recherche du couple ayant reference comme reference
        (req) -> req.reference.equals(reference)
).map(
        // Récupératino du PartType requis par reference
        (req) -> req.required
).toArray(PartType[]::new);
```

3.3 `ConfigurationImpl`

Pour stocker la configuration de l'utilisateur, une table de type `Map<Category, Part>` associe, à chaque catégorie, la partie sélectionnée par l'utilisateur.

La méthode `void isValid()` vérifie donc pour chaque partie sélectionnée :

— qu'aucune des parties incompatibles avec elle ne soit présente dans la sélection,

— que toutes les parties requises le sont.

4 Tests

Nous avons testé les trois classes qui effectuent un traitement sur des données (en comparaison aux autres classes qui stockent juste des informations ou un état) : `ConfigurationImpl`, `CompatibilityManagerImpl` et `PartImpl` pour la deuxième version.

4.1 `ConfigurationImpl`

Les comportements suivant sont testés sur la classe `ConfigurationImpl` :

— `isComplete()` sur une configuration vide renvoie `false`,

— `isValid()` sur une configuration vide renvoie `false`,

— `selectPart()` sur une configuration vide ajoute bien la partie sélectionnée à la configuration,

— `selectPart()` pour une catégorie déjà configurée remplace bien la partie précédemment sélectionnée par la nouvelle partie,

— `unselectPartType()` réinitialise bien la catégorie donnée,

— `isComplete()` sur une configuration non complète retourne `false`,

— `isValid()` sur une configuration non complète retourne `false`,

— `isComplete()` sur une configuration complète contenant une incompatibilité retourne `true`,

— `isValid()` sur une configuration complète contenant une incompatibilité retourne `false`,

— `isComplete()` sur une configuration complète contenant un *requirement* non satisfait retourne `true`,

— `isValid()` sur une configuration complète contenant un *requirement* non satisfait retourne `false`,

— `isComplete()` sur une configuration complète et valide retourne `true`,

— `isValid()` sur une configuration complète et valide retourne `true`,

— `clear()` réinitialise bien la sélection de chaque catégorie.

Le taux de couverture des lignes de code de la classe `ConfigurationImpl` est de 100% et le taux de réussite des tests est de 100%.

4.2 `CompatibilityManagerImpl`

Les comportements suivants ont été testés sur la classe `CompatibilityManagerImpl` :

— `addIncompatibilities` sur deux parties de catégories différentes ajoute la contrainte,

— `addIncompatibilities` sur deux parties de même catégorie lève une `IllegalArgumentException`,

— `addIncompatibilities` sur une même partie lève une `IllegalArgumentException`,

— `addIncompatibilities` sur deux parties liées par une contrainte de *requirement* lève une `IllegalArgumentException`,

— `addIncompatibilities` est réflexive,

— `removeIncompatibilities` supprime bien la contrainte d'incompatibilité,

— `addRequirements` sur deux parties de catégories différentes ajoute la contrainte,

— `addRequirements` sur deux parties de même catégorie lève une `IllegalArgumentException`,

— `addRequirements` sur une même partie lève une `IllegalArgumentException`,

— `addRequirements` sur deux parties liées par une contrainte d'incompatibilité lève une `IllegalArgumentException`,
— `addRequirements` n'est réflexive,
— `removeRequirement` supprime bien la contrainte de *requirement*.

Le taux de couverture des lignes de code de la classe `CompatibilityManagerImpl` est de 100% et le taux de réussite des tests est de 100%.

4.3 `PartImpl`

Les comportements suivants sont testés sur la classe `PartImpl` :
— les différents accesseurs retournent les valeurs attendues,
— `setProperty` pour une valeur discrète valide modifie bien la propriété,
— `setProperty` pour une valeur continue modifie bien la propriété,
— `setProperty` pour une valeur discrète invalide lève une `IllegalArgumentException`,
— `setProperty` sur une propriété en lecture seule lève une `IllegalArgumentException`,
— `setProperty` sur une propriété inexistante lève une `IllegalArgumentException`.

Le taux de couverture des lignes de code de la classe `PartImpl` est de 100% et le taux de réussite des tests est de 100%.