POLYTECH NICE-SOPHIA

UNIVERSITÉ CÔTE D'AZUR

Rapport Drinking Factory Machine

Remis le 04/12/2020

Damien Allari - Jason Robert

FiniteStateMachine SI4

Implémentation finale du projet

Pour le projet de Finite State Machine, *"Contrôleur de machine à boisson",* nous avons réussi à implémenter toutes les fonctionnalités exigées pour la partie "MVP" ainsi que l'option sur la gestion de l'Iced Tea et la détection de gobelet.

Choix de conception

On peut séparer notre *statechart* en 3 grandes parties : la sélection et le paiement de la boisson, la préparation de la boisson et enfin le nettoyage de la machine. Le nettoyage de la machine n'est pas réellement une partie importante puisque son seul but est de remettre cette dernière dans son état initial (enlever la sélection, réinitialisé les sliders, les options etc.). Nous n'allons donc pas nous étaler là-dessus.

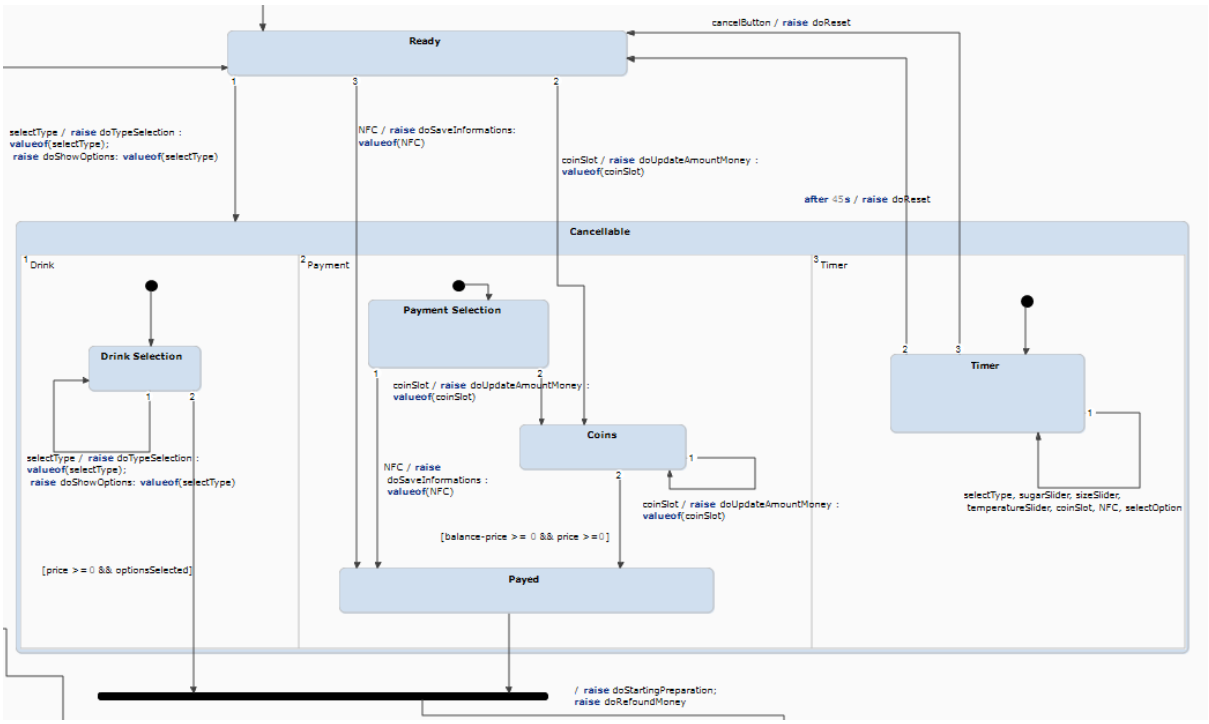
Partie Sélection

En ce qui concerne la sélection de la boisson par les utilisateurs, les exigences sont aussi nombreuses.
Afin de pouvoir sélectionner la boisson ou payer dans l'ordre que l'on veut, nous avons choisi d'utiliser un Composite State dans lequel une région est responsable de la sélection de la boisson ainsi que de ses options (que nous aborderons ultérieurement), une autre région est responsable du paiement et une dernière gérant un Timer, réinitialisé à chaque action utilisateur. Si celui-ci atteint 45 secondes, il annule la sélection.

Les différents états présents dans la région responsable du paiement nous permettent de ne pas pouvoir payer en NFC si on a inséré des pièces et inversement.

Pour passer à l'étape de préparation, il faut que plusieurs critères soient validés : boisson et options choisis, boisson payable…

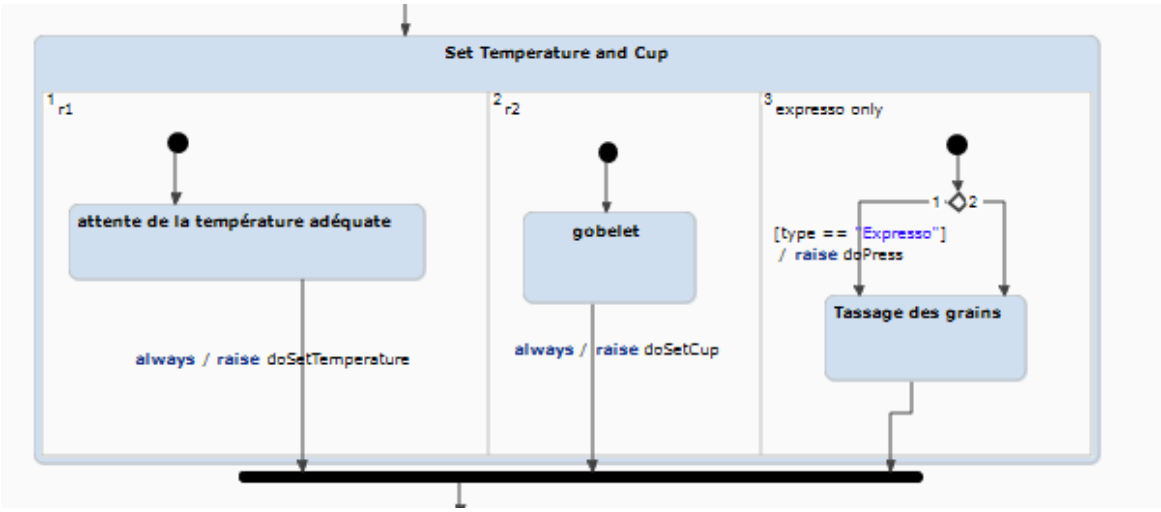
L'outil Synchronized, comparable au ET LOGIQUE, nous permet de vérifier la validation de ces critères. Le choix de son utilisation est dû au fait qu'il est très visuel, notamment lors de l'exécution du statechart, on visualise très rapidement quelle condition est validée et laquelle ne l'est pas.
Un fois la commande validée, on rend la monnaie si on a payé en pièces (ou on efface les données de paiement si on a payé en NFC).
L'état "Ready" en haut du statechart est relativement inutile car le compositeState se suffit à lui-même. Cependant, cet état nous permet de mieux visualiser quand la machine est à son état initial.



Partie Préparation

NB : Nous appellerons *groupe d'actions* l'ensemble des actions exécutables en parallèle. Exemple : (récupération et positionnement d'une dosette | démarrage du chauffage de l'eau); est le 1er groupe d'action de *coffee*.

Nous allons maintenant aborder la préparation des boissons et donc la réalisation des recettes en adéquation avec les spécificités du sujet.



Préparer une recette :

Le sujet spécifie que certaines étapes doivent être réalisées en parallèle et d'autres en séquences. L'utilisation d'états composites (voir exemple ci-dessous) nous est donc immédiatement venue à l'esprit et nous a paru être la meilleure solution. En effet, le fait de créer un *Composite State* par *groupe d'actions* dont chacun est composé d'autant de régions qu'il y a d'actions dans le *groupe d'actions*, cela nous permet de simuler parfaitement une exécution en parallèle.

Aucune autre option ne nous permettait de respecter parfaitement les spécificités du sujet (exécution en parallèle et en séquence) en gardant une bonne lisibilité. En effet, cette solution nous permet de visualiser toutes les étapes de la préparation directement depuis la stateMachine, en ayant une transition remontant un événement sortant (*out event*) par étape/action de la recette. Chacun de ces événements permet donc d'appeler une méthode dans le code Java responsable de la réalisation de cette action.

Préparer différentes recettes

Sachant maintenant comment préparer une recette, il reste à savoir comment représenter et choisir entre les différentes recettes que la machine propose.

Pour l'implémentation des différentes recettes, nous avons choisi de ne créer qu'une seule *statechart*. En effet, une autre option qui s'offrait à nous était de créer un nouveau *Statechart Model* pour chaque recette. Selon le type de boisson sélectionné, on bascule vers le statechart correspondant puis l'on revient vers le statechart principal une fois l'exécution terminée. Cependant, nous trouvions cette méthode lourde et comportant des redondances.
De plus, nous avons remarqué en observant les 3 recettes du MVP (*café*, *expresso*, *thé*) et de l'extension *Iced Tea* qu'elles avaient des étapes similaires et que l'ordre de ces dernières était assez similaire. Nous avons donc essayé de les factoriser afin que les recettes de ces 4 boissons soient intégrables dans le reste de notre *statechart* tout en gardant une certaine lisibilité.

Exemple :

Le premier *groupe d'actions* de ces 4 recettes possède l'action *"démarrage du chauffage de l'eau"*, nous avons donc juste à créer un seul événement de sortie (*doHeatingWater*) appelant une seule méthode utilisée pour les 4 recettes. De même pour les actions "*récupération et positionnement d'une dosette", "broyage des grains"* et *"récupération et positionnement d'un sachet"* qui concerne le composant principal de la boisson, nous les avons toutes regroupées dans la même méthode, appelée par un unique événement (*doPutProduct*).
Pour les recettes ayant plus de *groupes d'actions* (Tea et Iced Tea), on ajoute des états exclusivement pour ces recettes.

Nous avons factorisé les événements un maximum afin de d'éviter les redondances de code mais sans pour autant perdre le côté informatif de notre *statechart* en créant ce qu'on pourrait assimiler à des boîtes noires (aucune visibilité de ce qu'il se passe depuis la statechart car tout se passe dans le code Java).
Le principal inconvénient de cette méthode est qu'elle ne serait pas réalisable pour un grand nombre de recettes, ou pour des recettes aux étapes très différentes. Cet inconvénient est d'ailleurs le principal avantage de la méthode consistant à créer plusieurs *statechart*. Bien que redondante, cette méthode permet de ne pas être bridé par la composition ni par le nombre des recettes.
Les recettes présentes dans le sujet nous ont poussé à opter pour la 1ère méthode, que nous trouvions plus intéressante pour les raisons exprimées précédemment.
Cependant, si nous avions implémenté l'extension "Soupe", nous aurions opté pour cette méthode.

Explications détaillées des exigences

Exigence 7

« Lors d'un paiement par NFC les informations permettant la transaction sont détenues jusqu'au moment de la préparation de la boisson sans que la transaction soit effectuée. Cependant la transaction doit être effectuée avant la préparation de la boisson. Une fois la préparation en cours, les informations sont effacées. »

Afin de respecter cette exigence, nous avons dans un premier temps simulé le passage de carte d'un utilisateur par un TextField qui correspondrait à la valeur nous permettant de reconnaitre sa carte parmi les autres. Nous avons choisi cette option car elle nous paraissait la plus crédible et que cela ne changer par réellement de chose au fonctionnement de la state machine. Nous travaillons à partir d'une valeur donc qu'elle soit rentrée directement ou qu'elle soit détectée cela ne change pas.
Deuxièmement, lorsque l'utilisateur appuie sur le bouton « biip », cela crée automatiquement un id aléatoire composé de 10 chiffres qui sera gardé en mémoire jusqu'au moment de la préparation. Cet id aléatoire correspondrait au numéro de carte de l'utilisateur.

Exigence 15 : Options

En ce qui concerne les options, de nombreuses solutions s'offraient à nous.
Nous avions retenu 3 possibilités respectant les exigences.

1) Afficher toutes les options directement sous forme de boutons :
Selon les options que l'on coche, il ne reste que les options et les boissons compatibles.
Afin que l'utilisateur ne soit pas surpris : afficher dans la console : "n'oubliez pas de cocher les options avant de sélectionner la boisson ou avant de payer".

2) Choix sous forme de checkbox "Oui/Non" :
Pour toutes les options on affiche oui / non via 2 checkbox plus un bouton "aucune option". La préparation se lance quand un choix est fait pour chaque option, les options disponibles apparaissent une fois qu'on a sélectionné la boisson.

3) Une fois qu'on a sélectionné la boisson, les options s'affichent sous forme de pop-up (en gros) et on doit choisir.

L'alternative 3 nous paraissait trop "agressive". En effet, si beaucoup d'options sont disponibles et que l'utilisateur n'en veut aucune, il pourrait être relativement pénible de devoir choisir "non" pour chaque option.

L'alternative 1 comportait elle aussi des défauts car la préparation de la boisson pouvait se déclencher sans que l'utilisateur n'ai spécifié les options qu'il voulait.

C'est pour ces raisons que nous avons opté pour la seconde alternative, qui obligeait l'utilisateur à spécifier ses choix tout en lui permettant de tout refuser rapidement via le bouton "No options". Les options affichées correspondent à celles disponibles pour chaque boisson et dans la limite des stocks disponibles.
On vérifie qu'elles sont sélectionnées (oui ou non pour chacune) via un booléen dans le statechart (*optionsSelected*).

Prémices exigences 16 et 17

Comme vous allez le voir par la suite, nous utilisons par deux fois la lecture et écriture d'un fichier .txt pour gérer des informations.
Notre classe "FileExtern.java" nous permet de faire la liaison entre les fichiers .txt et les classes que nous présenterons par la suite qui sont "Stock.java" et "InformationNFC.java".
Elle prend en paramètre un chemin vers un fichier, dans notre cas ce sera des .txt.

- read() → renvoie une ArrayList de String qui contient chaque ligne du fichier de lecture
- write(String modification) → cette fonction réécrit le contenu du document à l'aide de la String modification.

Que ce soit "Stock.java" ou "InformationNFC.java", ces deux classes utilisent la méthode read() pour lire et affecter les données à leur Map à la construction de l'objet.
Elles utilisent aussi la méthode write(String modification) à chaque fois qu'elle modifier une donnée.

Exigence 16

« Il est maintenant nécessaire de savoir combien de doses sont disponibles pour chaque ingrédient et de désactiver les recettes/options pour lesquelles les ingrédients ne sont pas disponibles »

Pour ce qui est de savoir le nombre de doses qu'il reste pour chaque ingrédient afin d'exécuter les actions adéquates, nous avions deux solutions possibles qui se présentaient à nous. Stocker les informations dans un fichier externe ou créer tout simplement une enum qui selon l'ingrédient nous renvoyait son stock.
Nous avons opté pour la première solution car elle permettait tout simplement de stocker ces informations à l'extérieur de la machine. Le statut de la state machine (allumé, eteinte) n'influe donc pas sur les stocks.
Pour cela nous avons créé une classe "Stock.java" qui, à chaque lancement, la machine va lire un fichier text "stock.txt" et initialisé une HashMap<String, Integer> avec les données récupérées (String -> nom de l'ingrédient, Integer -> dose de l'ingrédient restant).
A chaque fois qu'une commande est passé par un utilisateur, les ingrédients nécessaire à la recette sont enlevés du stock à l'aide d'une méthode decrementeStock(String name, Int value) qui va enlever "value" doses à l'ingrédient "name".

Exigence 17

« fidélisation NFC: lors du paiement par NFC, le 11*eme* achat est gratuit dans la limite de la valeur moyenne des 10 premiers achats. (un *hash* des infos de la carte peut être gardé en mémoire) »

Pour ce qui est de l'exigence 17, nous avions la même problématique que pour les stocks, le but était de pouvoir garder les informations de la carte afin d'appliquer ou non l'option de la commande gratuite. Nous avons donc eu la même réflexion que pour le cas précédent et avons aussi enregistré les informations dans un fichier texte externe "nfc.txt".
La classe "InformationNFC.java" s'assurait de lire le fichier à chaque lancement de la machine et de répertorier les données dans une HashMap<Long, Order>.(Long → Le code rentré par l'utilisateur pour le paiement par NFC, Order → voir explications ci-dessous)
La classe Order.java nous servait de deuxième argument dans notre map et possédait deux attributs, un nombre de commandes et un prix moyen.

Nous allons à présent vous présenter les différentes fonctions présentes dans la classe "Informations.java" et leur fonctionnalités:
incrementeNFC(Long id, double price) → elle permet d'incrémenter le nombre de commande de un, à chaque fois qu'un utilisateur déjà connu de notre système passe une commande
addNFC(Long id, double price) → elle permet d'ajouter un nouvel utilisateur à nos données
remove(Long id) → elle permet de retirer la carte de nos données.
Lors de l'envoie de notre préparation, nous effectuons une condition nous permettant de vérifier si le nombre de commandes de l'utilisateur est supérieur à 10, si cela est le cas la commande devient alors gratuite et l'utilisateur est supprimé de nos données (à l'aide de la fonction remove). Nous avons pris le choix de le supprimer de nos données et de le considérer comme un nouvel utilisateur lors de son prochain achat.

Explications Supplémentaires

Calcul du prix

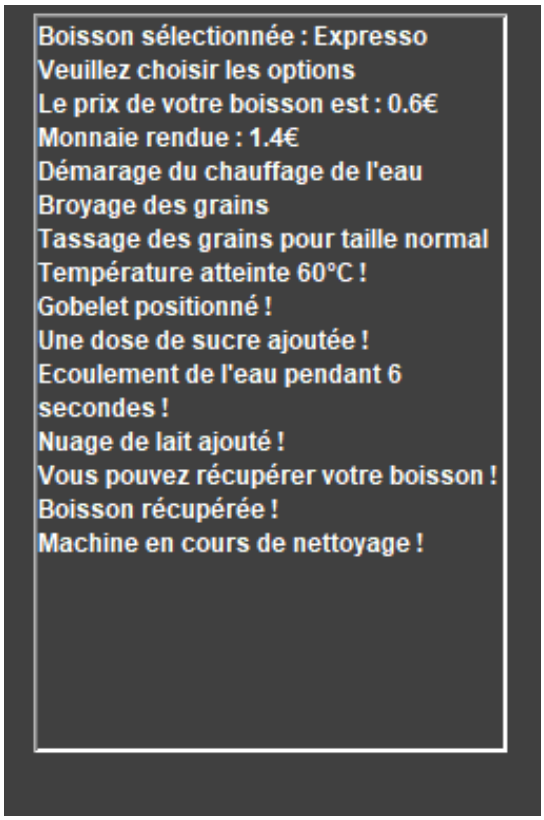
Les calculs entre nombres entiers étant moins “lourds” que les calculs entre flottants, nous avons préféré utiliser des entiers pour la monnaie (on compte donc en centimes d’euros).
Ce n’est que lors de l’encaissement et du remboursement que nous passons par des nombres flottants pour pouvoir afficher les prix en € (à virgule donc).

On utilise des variables (entiers) pour stocker la valeur du solde (si on utilise les pièces) et le prix de la boisson. Cela sert à créer des conditions, internes au statechart, nécessaires pour passer à la phase de préparation.
On initialise le prix de la boisson à -1. Cela nous permet de vérifier si une boisson a été sélectionnée, puisque son prix sera au minimum de 0. Nous pourrions aussi faire cette vérification avec la variable “*type*” mais comparer une chaîne de caractères est plus complexe qu’un entier.
Le prix final est mis à jour quand toutes les options disponibles sont sélectionnées afin de vérifier si le prix de la boisson est inférieur ou non au solde de l’utilisateur (price < balance).
Ces variables ont aussi l’avantage de pouvoir être récupérées et modifiées facilement.

Affichage

Afin d'informer l'utilisateur du bon déroulement de sa commande nous avons choisis d'afficher les différentes informations correspondantes aux différentes étapes de l'avancement de la recette. Lors de la sélection de sa boisson, l'utilisateur est informé en temps réel sur l'état de sa commande, prenons pour exemple le cas où il demanderait 3 sucres et que la machine n'en possède plus que deux en stock, un message lui indiquant cela s'affiche et cela permet à l'utilisateur de changer sa commande s'il le souhaite.
Nous aurions pu choisir de bloquer le slider à deux dans le cas précedeent, mais cela nous paraissait un peu trop brutal pour l'utilisateur qui pourrait se sentir bloquer dans cette situation.

Nous aurions aussi pu intégrer les sliders au statechart car ils ne nous servent qu'à réinitialiser le Timer, tout ce qui concerne les sliders est interne au code Java ce qui est un peu dommage. Nous nous en sommes malheureusement rendu compte trop tard.



Timers pour chaque action

Nous avons essayé de trouver un compromis entre réalisme et testabilité. En effet, autant pour le testeur que pour nous, le fait d'avoir un temps de préparation total trop long aurait été problématique dans le cas où l'on veut tester beaucoup de recettes à la suite. Pour cette raison, nous avons revu légèrement à la baisse les temps d'exécutions réels des actions (en moyenne une préparation dure une vingtaine de secondes).
Nous avons cependant essayé de garder un ordre de grandeur cohérent, c'est-à-dire que les 2 actions les plus longues sont celles du chauffage de l'eau et du versement de l'eau, ce qui nous semble être les 2 étapes les plus longues dans la réalité.

Extension soupe:

La partie sélection de la soupe est implémentée. En effet, il est possible de sélectionner cette boisson ainsi que lui ajouter les options disponibles. La gestion du prix de ces dernières est également implémentée dans notre projet. Cependant, la partie préparation n'est pas implémentée. Comme expliqué précédemment, nous l'aurions fait via un *statechart* annexe.
Nous avons donc désactivé le *Listener* sur le bouton Soupe, vous pouvez le décommenter pour vérifier nos dires.

Explication des options

Gestion de l'ice tea

Pour l'extension de l'Iced Tea, elle était relativement simple à implémenter. En effet, sa recette étant semblable à celles du MVP, il suffisait de rajouter un *case "Iced Tea"* dans nos switch. Pour les étapes de fin de préparation, nous avons fait de la même manière que pour le Thé. Il a ensuite fallu modifier la valeur du slider de température. Pour la taille, si le slider est sur small ou normal, cela compte pour une taille normale et s'il est sur long, la boisson sera longue.

Détection du gobelet

Pour la détection de la tasse, on crée une nouvelle région en parallèle dans le Composite State de sélection. Cela nous permet de pouvoir ajouter la tasse à n'importe quel moment avant le début de la préparation. C'est le meilleur moyen que nous avons trouvé.Lorsqu'on clique sur le bouton addCup, on met un booléen à *true* qui nous servira pour modifier le prix et ne pas mettre de gobelet.

Rétrospective

Nous allons à présent vous faire une rétrospective de notre projet et ainsi vous expliquer les choix que nous avons fait et ce que nous aurions pu améliorer.
Le point le plus important, et celui que nous avons le moins bien géré, était la gestion du temps du projet. Nous avions pris énormément de retard dès le début du projet car nous n'étions pas partis sur la bonne voie. Nous étions beaucoup trop concentrés sur la partie sélection de la boisson et paiement en mettant totalement de côté la partie préparation qui était pourtant tout aussi importante.
Malheureusement nous n'avons jamais réellement rattrapé ce retard et c'est notre plus grand regret.
De plus, nous avons finalement réussi à rendre un MVP et réaliser 2 options. Nous sommes convaincues que la réalisation d'autres options aurait été totalement possible si nous avions mieux géré l'avancement du projet.
Deuxièmement, il est vrai que nous n'avions pas réalisé de vérifications concernant nos choix de conception avec le professeur Deantoni. Nous avons préféré effectuer ces vérifications avec des personnes totalement externes aux mondes des states machines, que ce soit nos parents, sœurs etc··· afin d'avoir un avis neutre et concret.