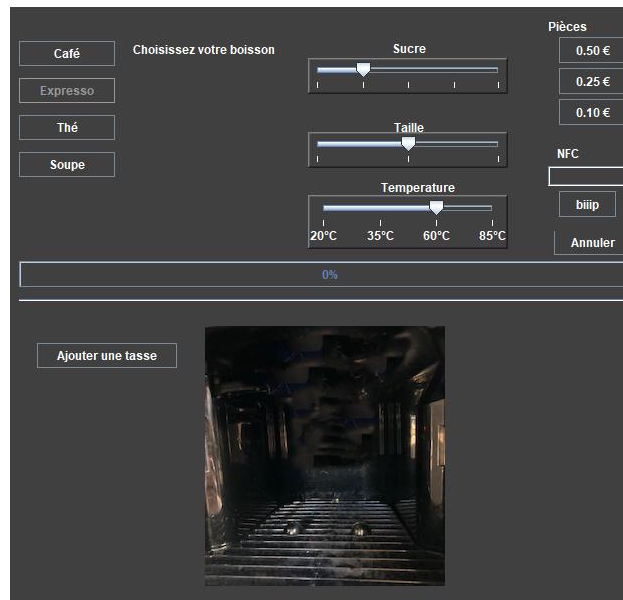


Rapport projet Finite State Machine

Drink Factory



Membres du groupe :

Sylvain Marsili

Florian Striebel

Professeur encadrant :

Julien Deantoni

Table des matières

Ce que nous avons développé	2
Choix techniques	2
Le paiement et la sélection	2
La préparation	2
La fidélisation client.....	3
La gestion de stock	3
Les options	5
La barre de progression	5
La soupe	5
Le gobelet.....	6
La V&V	6
Prise de recule.....	7

Ce que nous avons développé

Pour ce projet nous avons réalisé toutes les exigences du MVP de 1 à 17.

En ce qui concerne les extensions nous avons développé :

- La gestion de la soupe,
- La gestion de l'avancement de la préparation,
- La détection du gobelet.

Choix techniques

Le paiement et la sélection

Pour cette partie nous avons choisi d'implémenter en parallèle notre sélection et notre paiement. Cela paraissait en effet logique au vu du sujet. On doit pouvoir sélectionner et payer dans l'ordre souhaité par l'utilisateur. Nous avons également choisi de mettre un timer dans la state machine pour compter le temps d'inactivité de l'utilisateur. Nous avons décidé de le mettre à cet endroit et pas directement dans le code car ça nous semblait être une fonctionnalité interne au fonctionnement de la machine. Ce timer est en parallèle de la sélection et du paiement. Il va démarrer lorsque les manipulations de base sont effectuées (sélection, ajout gobelet et paiement). Le reset de ce timer est par contre directement liée à tous les boutons de la machine via le code, en effet s'il l'on veut capter toutes les actions des utilisateurs, nous n'avons pas le choix de le placer à ces endroits.

La préparation

Pour cette partie nous avons fait un choix important pour la structure de notre code. En effet une fois le paiement et la sélection effectués, nous avons décidé de lancer une nouvelle state machine. Ainsi toutes nos préparations sont dans des state machines différentes, le code va générer toutes nos state machines dès le début, et la state machine sélectionnée sera lancée. Une fois cette dernière finie, elle va dialoguer avec notre state machine principale pour lui dire que la préparation est finie, afin de lancer le nettoyage de la machine. Pourquoi ce choix ?

Premièrement, pour nous il y a une question de lisibilité et de maintenance, en effet en faisant plusieurs state machines, nos préparations sont séparées et nous pouvons facilement modifier l'une d'elle si quelque chose de spécifique leur est ajoutée. Cela permet donc de séparer notre code, ainsi toutes les étapes de notre préparation ne se retrouvent pas dans la même classe, cela permet une lisibilité bien plus facile. Il est aussi très facile d'ajouter de nouvelles boissons, il suffit juste de créer une nouvelle state machine et de l'appeler dans notre code. De plus, nos préparations ont certes des étapes parfois communes, mais le problème, c'est que ces étapes ne se font pas toujours au même moment. Nous avons donc du mal à implémenter dans une seule state machine le parallélisme entre ces étapes et donc de réutiliser les étapes communes. Des variables se rajoutaient en plus pour prendre les bons chemins et ces derniers partaient dans tous les sens. La séparation de state machine nous a permis de palier à ces problèmes, mais évidemment cela n'est pas sans conséquence.

Ce choix comporte un inconvénient majeur, la duplication de code. En effet comme cité plus haut, certaines étapes sont communes, donc forcément elles comportent un code similaire, et la state chart est formé de la même façon (par exemple le fait de chauffer est pareil dans toutes les state chart). Mais nous trouvons cela raisonnable, car le code dupliqué n'est pas lourd et de plus nous avons essayé de le factoriser au maximum. Par exemple nous avons fait une classe préparation où toutes les fonctions de temps comme le temps de chauffe, ou encore d'écoulement, sont présentes ; et cette classe est étendue par toutes nos préparations. Nous avons également mis en place une sous state machine pour le versement des ingrédients., car les étapes du versement des ingrédients sont exactement les mêmes pour toutes les boissons de base. Cette sous state machine est donc appelée dans chaque state chart des préparations de base, factorisant ainsi ces étapes.

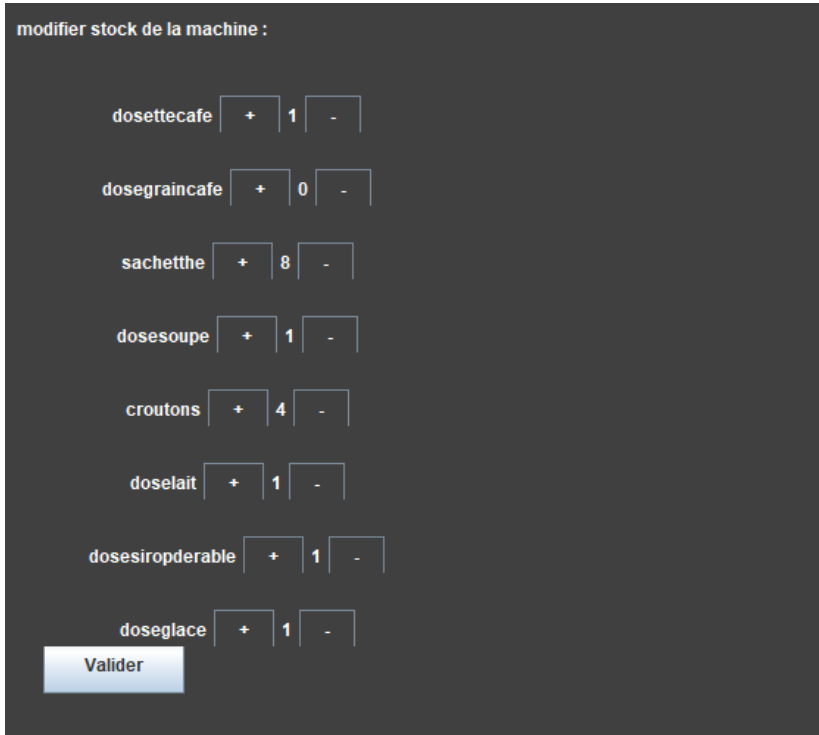
La fidélisation client

Le système de fidélisation client s'enclenche chaque fois qu'un client paie par NFC, lors de son 11ème achat sa boisson lui est offerte dans la limite de prix de la moyenne des 10 boissons précédentes si la boisson est moins chère alors elle est offerte sinon on retranche le prix moyen des achats précédents à la boisson. Pour cette partie fidélisation nous avons fait le choix pour différencier chaque client d'utiliser un champ texte comme nous ne pouvons pas vraiment détecter les cartes. Il suffit de rentrer une chaîne différente pour chaque client. Nous avons dû gérer le fait de rendre impossible de modifier le champ NFC après avoir validé la carte et pendant la préparation pour ne pas que, par exemple ce soit une carte qui paie mais une autre qui profite du programme. Celui-ci n'est pris en compte qu'une fois la préparation commencée ce qui représenterait dans le monde réel l'impossibilité pour quelqu'un de mettre discrètement sa carte juste après l'étape de paiement et juste avant la préparation pour profiter du point de fidélisation que va gagner le client précédent. Nous pouvions aussi faire le choix du menu déroulant mais cela nous aurait limité dans le nombre de client, mais aurait rendu la fidélisation plus simple à développer en créant une ArrayList par item du menu déroulant. Afin de stocker les id de carte et les prix associés, nous avons stocké toutes ces informations dans un fichier. Pour le format de stockage des données nous avons deux solutions qui s'offraient à nous soit nous gérons nous même le format des données soit nous stockons nos données aux formats json. Les 2 ont leurs avantages, la gestion de l'écriture lecture est plus difficile à coder mais permet de ne pas importer une librairie complète pour des objets simples. Alors qu'en json ça nous permettait de ne pas développer tout le système de lecture/écriture pour notre HashMap. Et de plus cela nous permet facilement de réutiliser pour stocker d'autres données qui ne seront pas au même format. Pour ces deux raisons, nous avons choisi la deuxième solution.

La gestion de stock

Nous avons fait le choix que chaque ingrédient était géré avec des doses, et non pas des quantités (poids etc....), l'eau est également présente en quantité infinie (branchée directement au réseau d'eau).

Pour cette exigence de gestion des stocks nous avons plusieurs possibilités la première, la plus simple, c'est dire qu'à chaque fois que le technicien viendra il va devoir éteindre la machine puis il rechargera la machine au maximum. Donc qu'à chaque redémarrage de la machine on part du principe que celle-ci est pleine. L'autre possibilité est de permettre à la machine, au travers de capteur, d'elle-même connaître ses stocks. Comme nous ne pouvons pas utiliser de capteur dans notre cas, la machine lit un fichier où chaque ingrédient est lié à un niveau de stock quand celui-ci atteint 0 elle le détecte donc est rend ensuite l'item correspondant à cet ingrédient indisponible. Ce cas est bien plus réaliste mais le problème, c'est que le fichier doit être manipulé pour y insérer les valeurs nous ne sommes pas à l'abri d'une erreur du technicien lorsque celui-ci le modifiera. Comme dernière solution nous avons la possibilité de permettre au technicien de ne pas devoir toucher à ce fichier en lui offrant la possibilité de pouvoir changé les stocks depuis un affichage graphique. **Pour afficher cette interface il faut cliquer sur CTRL** avant le début d'une préparation. Nous avons d'abord développé la version plus simple pour montrer quelque chose de fonctionnel puis après nous avons développé la dernière version avec l'affichage graphique pour notre utilisateur. Une fois le stock géré il faut maintenant trouver un moyen de montrer aux clients quand il n'y a plus d'ingrédients pour faire une boisson. Pour ça il existe différente possibilité soit faire disparaître les boutons des boissons et/ou option en rupture de stock le problème ici est que l'utilisateur ne peut pas savoir si la boisson a été enlevé car il n'y a plus de stock ou car elle n'est plus vendue. La deuxième possibilité que nous avons implémentée est de désactiver le bouton concerné et de changer sa couleur pour montrer que celui-ci n'est plus cliquable.



modifier stock de la machine :

dosettecafe	+	1	-
dosegraincafe	+	0	-
sachetthe	+	8	-
dosesoupe	+	1	-
croutons	+	4	-
doselait	+	1	-
dosesiroperable	+	1	-
doseglace	+	1	-

Valider

Figure 1-Affichage gestion des stocks

Les options

Le choix des options est entièrement géré par le code java. Nous avons d'abord pensé à rajouter des check boxes sur le simulateur et permettre à l'utilisateur de les cocher quand il veut ; et faire disparaître ou réapparaître certaines selon la sélection. Mais ce choix était un peu lourd, ça prenait de la place en permanence, l'utilisateur n'était pas forcément interpellé par les options. Mais surtout nous avions du mal à réactualiser le prix selon les options choisis. Si l'utilisateur a payé d'avance et qu'il sélectionne une boisson, une option pouvait disparaître car elle n'était pas présente pour cette boisson, et la préparation se lançait quand même. Nous pouvions donc faire un système similaire à celui implémenté pour la soupe avec un bouton validé, mais cela ne semblait pas très orienté utilisateur. Nous avons donc fait le choix de mettre une fenêtre pop-up à chaque sélection de boissons pour obliger l'utilisateur à faire un choix d'option ou au contraire ne mettre aucune option. Ainsi dès que le choix est fait, le prix est directement mis à jour, et il peut par exemple rajouter des pièces s'il a payé en espèces. Le seul inconvénient de ce choix c'est que nous ne permettons pas directement à l'utilisateur de changer ses options, pour cela il faut qu'il resélectionne la boisson, c'est un inconvénient très léger. Les choix seront ensuite transmis aux state machines de chaque boisson lors du début de leur préparations.

La barre de progression

Plusieurs choix s'offraient à nous, implémenter avec des threads dans le code java, le mettre dans la state machine de chaque préparation ou dans la state machine principale. Etant donné que cette barre était liée au fonctionnement direct de la machine nous avons éliminé le premier choix. Le deuxième était éliminé car il impliquait une nouvelle duplication de code facilement évitable avec le troisième choix. Ainsi un état parallèle à la préparation dans la state machine principale, met automatiquement la barre à jour de 1% tous les temps $\text{préparation}(\text{ms})/100 \text{ ms}$. Il suffit juste que chaque préparation nous remonte le temps de préparation en fonction de tous leurs paramètres (options, gobelet ...).

La soupe

Le plus gros problème de la soupe c'est qu'il faut absolument choisir la quantité d'épice et la faire valider par l'utilisateur. Et pour le coup nous n'avons pas trouvé beaucoup de solution. Nous avons juste signalé à l'utilisateur qu'il fallait absolument choisir une épice et ensuite la valider. Le changement d'affichage n'est pas géré par la state machine mais par le code, nous ne trouvons aucun intérêt que cela se fasse par cette dernière. Il suffit juste de changer l'affichage lors d'un clic sur le bouton soupe au niveau du code. Pour la state machine nous avons simplement ajouté une variable "validé" pour lancer la préparation, toujours vraie lors de la sélection des autres boissons, et qui se met vraie pour la soupe lorsque l'utilisateur valide la quantité d'épices.

Le gobelet

Le gobelet c'est fait assez simplement au niveau de la state machine. Seul problème, la gestion des prix. En effet, on peut rajouter le gobelet à tout moment avant la préparation, un problème se présente notamment au niveau de la sélection. Vu qu'à chaque sélection on change de prix, il faut qu'à chaque sélection le fait que le gobelet est présent soit reconnu. Ainsi lors de l'ajout du gobelet on va mettre à jour le prix s'il y a un, ou on va mettre un jour un booléen qui pourra indiquer en cas de nouveau prix (une autre sélection), que la réduction doit s'ajouter à ce prix.

La V&V

Nous avons eu quelques difficultés à utiliser Itsa pour les tests car nous avons quelques problèmes avec le logiciel qui avait souvent des crashes lors de la compilation. Pour notre activité Itsa, nous avons choisi de faire une V&V uniquement sur la partie paiement de notre projet. Nous avons eu du mal à mettre en place le code Itsa, surtout pour créer une state machine similaire à la nôtre.

```
assert Prop1= [] (addFirstCoin -> (<>!back))
assert Prop2 = [] (paidNFC -> (<>!change))
```

Figure 2-propriétés rendu de monnaie

Prop1 et Prop2 sont des propriétés safety et de vérification. On vérifie bien qu'on ne peut pas être remboursé de la même façon si l'on paye avec de la monnaie, ou si on rajoute des pièces une fois avoir payé par NFC

```
assert Prop3 = [] (addFirstCoin -> (<>!paidNFC))
assert Prop4 = [] (paidNFC -> (<>!addFirstCoin))
```

Figure 3-propriétés de paiement

Pour ces propriétés nous sommes encore dans le même cas qu'au-dessus, nous vérifions bien que nous ne pouvons pas payer de deux façons différentes à la fois.

```
assert Prop5 = [] (addFirstCoin -> (<>addCoin))
```

Figure 4-propriété paiement en pièce

Pour cette propriété, nous sommes dans un cas de liveness et vérification. On est on s'assure que globalement, si on ajoute la première pièce, alors nous pouvons quand ajouter une nouvelle pièce.

```
assert Prop6 = [](((paidNFC || addFirstCoin) -> (<>selected)) || (selected -> <>(paidNFC || addFirstCoin)))
```

Figure 5-propriété 6

Pour cette propriété, nous sommes dans un cas de liveness et validation, conformément au MVP, on peut d'abord payer comme on le souhaite puis on pourra sélectionner la boisson ; ou alors on pourra d'abord sélectionner la boisson puis on pourra payer de la manière que l'on veut.

Prise de recule

Tout d'abord nous nous sommes heurtés à beaucoup de problème avec Yakindu.

La synchronisation entre les états parallèles ne se faisait pas correctement, nous avons dû ajouter à chaque fois des every 100ms pour une des transitions vers la synchronisation pour que celle-ci soit s'effectue correctement.

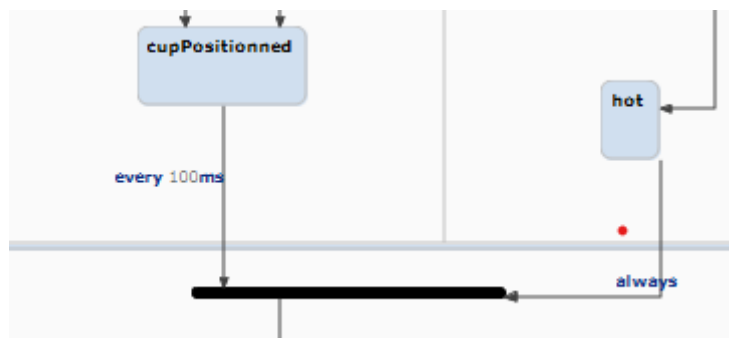


Figure 6- synchronisation sous Yakindu

De plus lorsque l'on changeait un booléen de la state machine via notre code, ce changement était reconnu par la state machine mais aucune action ne se lançait. Nous avons dû régler le problème également avec des every, ou dans le cas du gobelet avec un appelle raise après la mise à jour du booléen. Peut-être nous n'étions pas assez expérimentés pour la maîtrise de l'outil sur certain point. Pourtant nous avons beaucoup cherché, même si la documentation est assez fournie nous ne trouvions pas toujours la solution à nos problèmes.

Si nous devions refaire le projet, nous améliorerions certainement certaines parties du code. Nous nous sommes principalement concentrés sur l'aspects fonctionnelle de notre projet. Nous avons fait en sorte que tout marche correctement, au détriment de la qualité de certaines parties du code. Certaines classes sont peut-être un peu trop longues, notamment les classes où se font toute la partie visuelle du projet (OptionPanel, DrinkFactoryMachine). Par exemple mettre tous les boutons visible ou enabled pour la partie option, est assez fastidieuse.

Nous pensons que notre choix pour la mise en place des différentes state machine est une bonne option parmi toutes les solutions possibles, nous avons vraiment pesé les pors et les contres, et c'est un choix que nous referions.

Pour finir, nous avons apprécié faire ce projet. L'utilisation d'outil comme Yakindu pour construire des state machines est vraiment pratique. Il aurait été intéressant de voir comment nous l'aurions implémenté si nous devions entièrement le codé à la main.