

# MCR, Projet Monteur

Par J.Châtillon, B.Schopfer, J.Smith, A.Rochat  
Juin 2018

## Introduction :

Dans le cadre du projet de MCR, nous avons réalisé un jeu en Java implémentant le patron de conception « Monteur ». Dans ce jeu, l'utilisateur tient un restaurant de burger. Il est chargé de créer les burgers correspondant aux commandes des clients et de les terminer avant que le client ne s'impatiente. Il faut également faire attention à ce que le burger corresponde à la demande du client (mêmes ingrédients et même ordre) sinon le client vomira.

Le but est d'amasser autant d'argent que possible, servir un maximum de client et tenir le restaurant le plus longtemps. Mais attention, si 10 personnes quittent le restaurant sans leur commande ou le quittent en étant malades, c'est perdu !

## Interface :

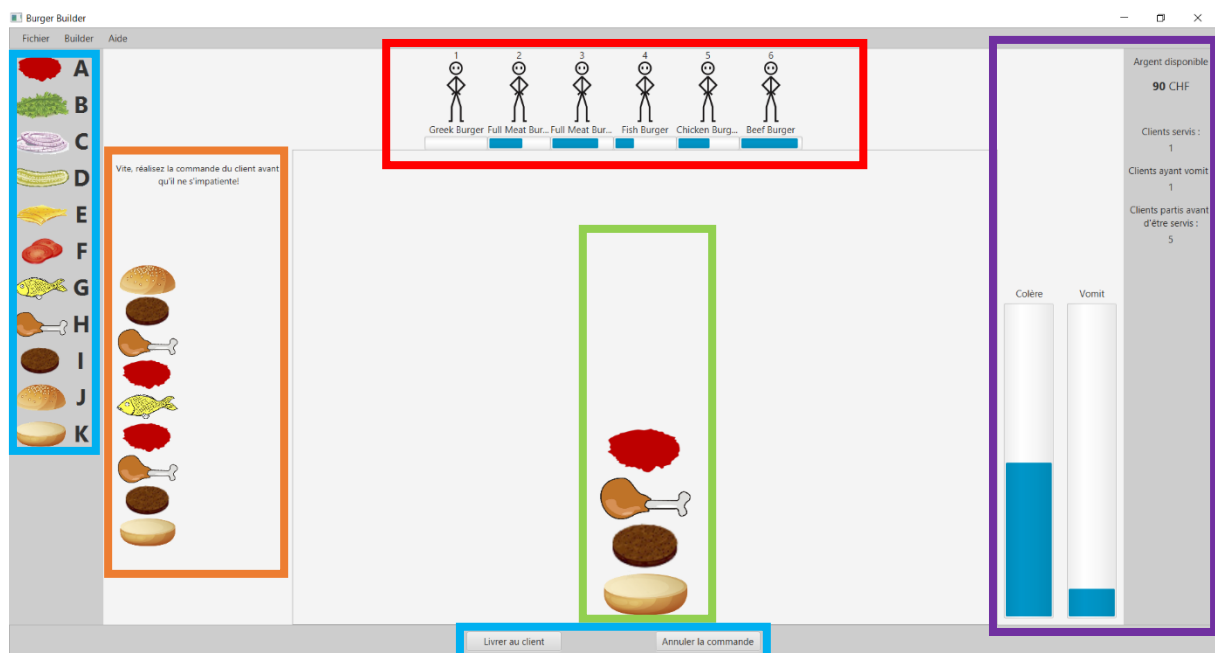


Figure 1: L'interface est découpée en plusieurs parties, encadrées dans cette image.

À gauche de l'interface, dans le cadre bleu clair, se trouve la liste des condiments qui composent les différents burgers proposés par le restaurant. La lettre apposée à chaque condiment correspond à la touche du clavier à utiliser pour ajouter ce condiment au burger en cours de fabrication. Les touches ont volontairement été choisies pour ne pas être instinctives.

Dans le cadre orange, se trouve le burger exemple tel qu'il est défini dans le menu du restaurant. C'est cet exemple que l'utilisateur doit reproduire afin de livrer le bon burger au client qu'il a sélectionné.

Dans le cadre rouge, au centre de l'interface, se trouve la file d'attente des clients. Chaque client possède un timer correspondant au temps restant avant qu'il ne s'impatiente et reparte en colère en incrémentant la jauge correspondante. Les chiffres situés au-dessus des clients correspondent au

numéro de la touche du clavier que l'utilisateur doit presser pour sélectionner le client désiré. Le texte correspond au nom du burger que le client souhaite commander.

Le cadre vert contient le burger que l'utilisateur est en train de cuisiner. C'est ici qu'apparaissent les condiments ajoutés.

Si l'utilisateur s'est trompé, le bouton « Annuler la commande » permet de réinitialiser le builder. Ce bouton est également activable à l'aide de la touche *effacer* du clavier.

Lorsque le burger correspond à celui du menu, le bouton « Livrer au client » permet de le valider et de le lui donner. Ce bouton est également activable à l'aide de la touche *Enter* du clavier.

Le cadre violet contient deux jauges. La première, celle de colère se remplit lorsque l'utilisateur n'a pas été assez rapide pour servir un client. La seconde, celle de vomit, se remplit lorsque l'utilisateur livre à un client un burger incorrect. Également dans le cadre violet, tout à droite de l'interface, se trouve la barre de score. Celle-ci indique l'argent restant dans la caisse du restaurant ainsi que le nombre de clients servis, le nombre de clients partis en colères et le nombre de clients ayant vomis.

## GamePlay :

Ce jeu est conçu pour pouvoir être joué entièrement au clavier et ainsi être aussi rapide que possible. Toutefois, toutes les actions peuvent également être effectuées à l'aide de la souris.

Tous les certains temps<sup>1</sup>, un nouveau client vient s'ajouter à la file d'attente.

L'utilisateur doit tout d'abord sélectionner un client à l'aide de la touche numérotée correspondante au chiffre situé au-dessus de l'utilisateur, ou en cliquant dessus à l'aide de la souris. Une fois sélectionné, le burger à créer sera affiché à gauche de l'interface (cadre orange).

L'utilisateur doit alors créer ce burger en empilant dans le bon ordre les condiments (cadre bleu). Attention, les condiments ont des coûts ! Ajouter un condiment au burger en construction va donc faire baisser le fond de caisse du restaurant. En cas d'erreur, l'utilisateur ne peut pas enlever un seul condiment. Il est obligé d'annuler la commande, re-sélectionner le client et recommencer. L'argent investi dans les condiments ainsi gaspillés est donc perdu.

Une fois le burger terminé, il faut le livrer au client à l'aide du bouton « Livrer au client » ou en pressant la touche *Enter* du clavier. Le client payera sa commande (prix du condiment + 2 pour chaque condiment du burger commandé). Cet argent sera directement ajouté à la caisse du restaurant. Si le burger ne correspond pas à la demande, non seulement le client ne payera pas, mais en plus il vomira sur le plancher et partira, ce qui incrémentera la jauge de vomit.

Si l'utilisateur met trop de temps à servir un client, l'utilisateur s'impatient et quitte le restaurant en colère, ce qui incrémente la jauge de colère. Cependant lorsqu'un client est sélectionné, son timer s'arrête et celui-ci attendra le temps qu'il faut afin de recevoir sa commande.

Le but du jeu est de servir le plus de clients et de générer un maximum d'argent.

Il y a trois événements qui mettent un terme à la partie :

- La barre de colère atteint son maximum.
- La barre de vomit atteint son maximum.

---

<sup>1</sup> Le temps entre l'arrivée de 2 clients varie entre 2 et 5 secondes. Tous les temps sont définis à l'aide de constantes dans la classe *Rules* où ils peuvent être facilement adaptés en fonction de la difficulté souhaitée.

- Le restaurant n'a plus d'argent.

## Implémentation globale :

Langue : Java

Librairie graphique : JavaFX

Afin de pouvoir réaliser cette application, nous avons décidé de choisir la librairie graphique JavaFX. Le fichier *mainView.fxml* définit le contenu graphique statique. Ce contenu est l'ensemble de la vue mis à part la vue du tableau de clients qui est générée dynamiquement (aussi avec la librairie JavaFX).

Le cœur du programme est la classe *MainViewController* qui gère l'ensemble de l'affichage et du fonctionnement du programme. Elle contient tous les « *EventHandler()* » dont les clics et actions claviers. Elle gère aussi la logique du jeu mis à part la création des burger et des clients ; c'est elle qui définit quand la partie est terminée, quand un client n'est pas content, l'argent disponible etc..

La classe *ClientsManager* gère la file d'attente des clients. Elle en ajoute un certain temps et les mets dans la tableau de client jusqu'à ce que celui-ci soit pleins. Elle contient aussi la référence sur le client qui est entrain d'être servit. Nous avons une relation forte entre le *ClientsManager* et le *MainViewController*. Entre le *ClientsManager* a besoin d'avoir un lien pour pouvoir afficher le tableau et le *MainViewController* doit pouvoir transmettre l'événement quand le joueur sélectionne un client.

Un client (de la classe *Client*) est un objet qui a sa propre représentation graphique. Elle contient un *Timer* qui, une fois terminé, appelle une fonction le *MainViewController* pour pouvoir gérer le fait qu'il soit parti. Chacun des client possède une référence sur un des menu. Le menu est choisi aléatoirement parmi les menus disponibles.

La classe *Menu* contient une Liste statique de tous nos menus. Ils sont composés d'un *BurgerBuilder*, d'un *Burger* et d'un nom.

Afin de simplifier et rendre le jeu plus fonctionnel, nous avons implémenté la classe *Rules* qui contient des données statiques comme le nombre maximum de client dans la file d'attente ou l'argent de départ par exemple. Nous avons mis que les éléments qui nous semblaient vraiment important dans cette classe.

Pour la modélisation des classes, nous avons généré 2 UML :

- Un complet qui contient toutes les classes et dépendances généré depuis IntelliJ. ( Cf : *MRC\_ProjetMonteur\_UML.png*)
- Un résumé plus lisible qui contient les informations et les relations importantes pour bien comprendre la structure du projet. (Cf : *MRC\_ProjetMonteur\_SmartUML.png*)

## Implémentation du modèle :

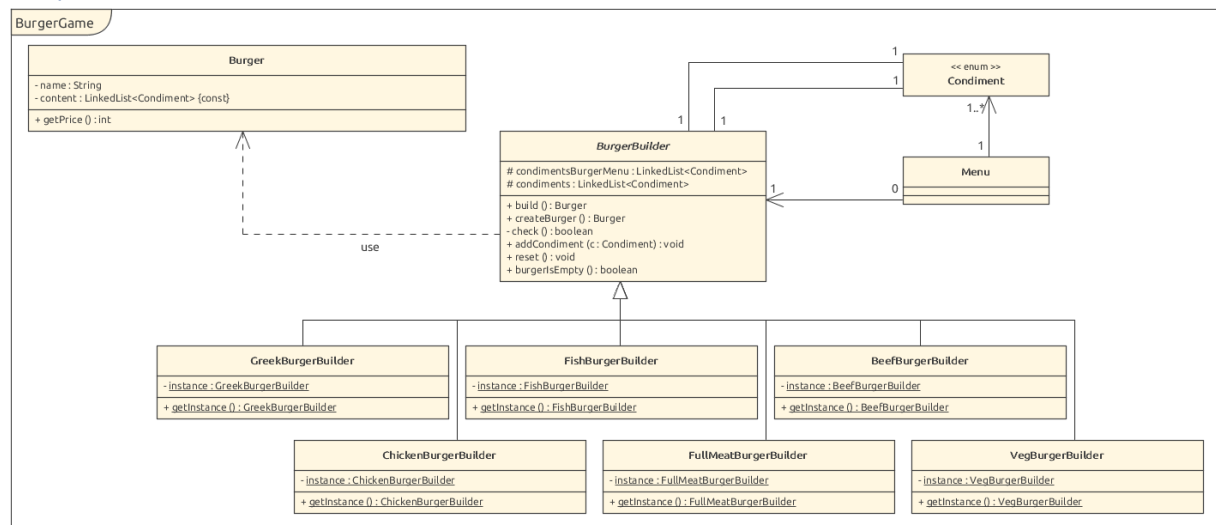


Figure 2: Diagramme de classe de la partie de notre application implémentant le modèle Monteur.

Nous avons vu pendant notre présentation du modèle Monteur qu'il pouvait être utilisé de nombreuses façons en fonction des situations rencontrées. Pour illustrer cela, nous avons implémenté deux façons d'utiliser le modèle Monteur, qui sont, selon nous, les plus fréquemment utilisées.

La première façon d'utiliser le modèle Monteur est de lui demander de nous fournir un objet « tout fait ». Nous avons implémenté cela dans la méthode `build()` du monteur abstrait **BurgerBuilder**. Cette méthode construit toute seule un objet de type **Burger** et nous le retourne. Dans notre application, nous utilisons ce monteur afin de créer les burger exemples correspondants aux différents menus proposés par le restaurant. Un Burger est ainsi créé pour chaque menu lors du chargement du jeu.

La seconde façon d'utiliser le modèle Monteur est de créer un monteur concret vide, puis lui donner petit à petit les éléments permettant de construire l'objet final. Une fois tous les éléments réunis dans le monteur, on lui demande de vérifier son contenu et de construire l'objet final si possible. Cette idée est le centre névralgique de notre jeu. En effet, le monteur est représenté par le burger que l'utilisateur doit construire pour le vendre au client. Chaque menu du restaurant possède une référence vers un monteur concret. Tous les monteurs concrets héritent du monteur abstrait **BurgerBuilder** et implémentent en plus le modèle du Singleton. Chaque menu possède donc une référence vers une instance unique du monteur concret permettant de créer le burger souhaité.

Ainsi, lorsque l'utilisateur sélectionne un client, on récupère l'instance du monteur concret correspondant au menu désiré par le client. Ce monteur est vide et l'utilisateur y ajoute des condiments à chaque fois qu'il presse la touche correspondant à un condiment de la barre de gauche de l'interface graphique. Lorsque l'utilisateur active le bouton « Livrer au client », les condiments contenus dans le monteur (*condiments*) sont comparés à ceux du burger exemple du menu (*condimentsBurgerMenu*). Si les condiments sont les mêmes et qu'ils sont dans le même ordre, alors un burger correspondant est créé (via l'appel à la méthode `build()`) et le client s'en va avec sa commande après l'avoir payée. Sinon, une erreur est levée et catchée dans le `mainViewController`. Dans ce cas, le client vomit puis s'en va sans payer.

Dans notre application, la vérification (méthode `check()`) est la même pour tous les monteurs concrets. C'est pourquoi, elle est faite directement dans le monteur abstrait. Si ça n'était pas le cas, il suffirait de déclarer cette méthode abstraite dans **BurgerBuilder** et de la définir dans chacun des monteurs concrets en fonction des vérifications à effectuer pour chacun.