2023-2024

Une image contenant Police, Graphique, logo, graphisme

Description générée automatiquement

Programmation Java IV

Rapport de projet : HELBThermo

M. RIGGIO

SOLTAN Lenny

Table des matières

[**Introduction** 2](#_Toc167827868)

[**Fonctionnalités de base** 2](#_Toc167827869)

[Interface de contrôle : 2](#_Toc167827870)

[Les différents types de Cellules : 4](#_Toc167827871)

[- Cell : 4](#_Toc167827872)

[- HeatSourceCell : 5](#_Toc167827873)

[- DeadCell : 5](#_Toc167827874)

[Les différents ThermoSystem : 6](#_Toc167827875)

[- ThermoSystemManual : 7](#_Toc167827876)

[- ThermoSystemTarget : 8](#_Toc167827877)

[**Analyse et Application des Design Patterns** 9](#_Toc167827878)

[**Limitations** 10](#_Toc167827879)

[**Conclusion** 11](#_Toc167827880)

# **Introduction**

Dans le cadre du cours de programmation Java IV, il nous a été demandé de réaliser un projet à l’aide du *Framework* *JavaFX* sur une base *Maven* en respectant différents principes d’architectures logicielle comme certains *GRAS* patterns et *Design patterns* *GoF*. Cette application a pour but de modéliser un système thermodynamique où des échanges de chaleur ont lieux dans une simulation. Ce système contient un espace rectangulaire contenant des cellules. Parmi ces cellules, il y a des sources de chaleur et également des cellules mortes. Pour que l’on puisse interagir avec le système, nous avons implémenté une interface de contrôle de celui-ci que nous visualiserons plus tard dans ce rapport.

Dans ce dernier, nous allons commencer par parcourir l’ensemble des fonctionnalités qui ont été demandées d’implémenter dans ce projet ainsi que de la manière dont cela a été fait. Nous allons parler du design graphique mais aussi architecturale du projet, une analyse sera fournie afin de vérifier que l’application respecte bel et bien les *Design Patterns* mis en place. Nous parlerons des limitations de l’application, jusqu’où peut-elle aller, il y a-t-il des possibilités de crash à certains moments, etc. Pour finir, ce rapport se terminera par une conclusion personnelle sur l’achèvement du projet et une rétrospective dans sa globalité après la finalité de cette épreuve.

# **Fonctionnalités de base**

Dans cette section nous allons parler des fonctionnalités de base qui ont été incorporées à l’application et qui était demandé dans l’énoncé du projet ainsi que ses addendas.

## Interface de contrôle :

Une interface a été implémentée grâce au *Framework JavaFX*, cela a été fait en respectant le *Design Pattern MVC* cela signifie que nous avons une class *ThermoView.java* qui sera la vue principale de notre application ainsi qu’un *ThermoController.java* et différents modèles que nous verrons plus tard. Une image contenant texte, capture d’écran, Rectangle, carré

Description générée automatiquement

Comme nous pouvons le voir l’interface fournis un certain nombre d’informations sur la simulation du système thermodynamique.

Commençons par la partie centre droit de l’interface, celle-ci contient un tableau de 4x5 cellules qui peut être modifié à partir du code grâce aux variables *COLUMN\_CELL* et *ROW\_CELL* dans le contrôleur. Une cellule affiche sa température et prend la couleur de celle-ci, plus la température est élevée plus la cellule vire au rouge. Lorsqu’une cellule a comme texte « sn° » devant sa température c’est qu’il s’agit d’une source de chaleur sa température et sa couleur sont donc fixe. Quand la cellule est toute noire, cela signifie qu’il s’agit d’une cellule morte, elle ne possède aucune température et n’impacte pas les autres.

Une image contenant texte, capture d’écran, Police, multimédia

Description générée automatiquementLorsque l’utilisateur clique sur l’une d’elle une nouvelle interface apparait pour régler les paramètres de la cellule cliqué.

Figure 1 interfaces de configuration d'une cellule

Une image contenant texte, capture d’écran, Police, logo

Description générée automatiquementIci nous pouvons voir qu’il s’agit d’une source de chaleur avec une température réglé à 50°. Il est important de noter qu’ici si nous cochons la case ‘*Set Dead Cell*’ la partie ‘*Set Temperature*’ devient impossible à définir. Lorsque le bouton Enter sera cliqué un message de vérification sera transmis à l’utilisateur afin de vérifier qu’il souhaite réellement modifier les paramètres de la cellule.

Figure 2 Message de confirmation

Une fois cliqué sur *Yes* les deux fenêtres se ferment et une nouvelle cellule est créé pour remplacer l’ancienne. Nous verrons plus en profondeur la création des cellules plus loin dans ce rapport.

Une image contenant texte, capture d’écran

Description générée automatiquementDans la partie gauche de l’interface nous pouvons retrouver les différentes sources de chaleur, à partir de là l’utilisateur peux activer/désactiver ces dernières, lorsqu’une cellule est désactivée sa couleur devient grisée. Juste au-dessus de ces cellules nous retrouvons 3 boutons pour gérer la *timeline* de la simulation le bouton *start* permet de démarrer/reprendre la simulation, le bouton pause permet de mettre en pause la *timeline* et le *reset* de remettre à 0 le temps et le coût.

Ces deux derniers éléments peuvent être retrouvé dans la partie supérieur de l’interface, nous y retrouvons également la température extérieur et moyenne mais aussi le mode de chauffe qui sera développer ultérieurement. A chaque secondes le coût est calculé en fonction de la température de chaque source de chaleur : « s1.température² + s2.température² + s3.temperature² + etc… ».

## Les différents types de Cellules :

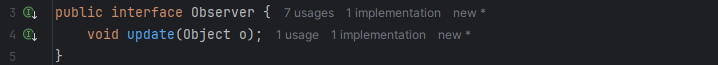
### Cell :

Une image contenant capture d’écran, texte

Description générée automatiquementLa class *Cell.java* représente une cellule de notre système thermodynamique. Celle-ci contient deux *int* x et y, un *double* *temperature* et un *String id*. Voici son constructeur :

Nous pouvons voir que son id est directement initialisé aux deux nombres de la position de celle-ci étant donné que les cellules ne sont pas mobiles ces deux nombres sont constant et unique étant donné qu’il ne peut y avoir deux cellules à la même position. La position est transmise en paramètre du constructeur, la temperature est établie de base à celle de l’extérieur. Une cellule implémente également l’interface *Obserbable.java*.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Ces interfaces sont en lien avec le *Design Pattern Observer*, puisque les cellules doivent être actualisé à cycle de la *timeline* avec une nouvelle température il fût primordial d’incorporer ce *pattern* dans notre projet. Grâce à la méthode *notifyObserver()* la vue pourra s’actualiser en conséquence à chaque appel de cette méthode puisque *ThermoView.java* implémente l’interface *Observer.java.*

### HeatSourceCell :

La classe *HeatSourceCell.java* représente une cellule devenue une source de chaleur donc celle-ci hérite de la classe Cell. En plus des attributs de sa class mère, elle contient un double heatTemperature qui sera transmis lorsque la source est activée et un *boolean* *isActivated* initialisé à *true*. Voici son constructeur :

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Dans celui-ci, nous faisons appel à celui de la classe mère avec la position et la transmise en paramètre. *isActivated* est initialisé à *true* et *heatTemperature* à la température transmise en paramètre.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Cette class contient une réécriture du getter de la température pour renvoyer la température de chauffe lorsque la cellule est activée. Cela permet de renvoyer la température adéquate aux autres cellules.

### DeadCell :

La class DeadCell.java représente une cellule devenue une cellule morte c’est pour cela qu’elle hérite de la classe Cell.java. Elle n’apporte aucun attribut en plus, cependant son constructeur contient une particularité et la méthode *setTemperature()* est réécrite.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

La température est set à 0 pour faire en sorte que la cellule ne contienne plus de température.

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Dans la classe *ThermoView.java* qui pour rappel implémente *Observer.java*, nous avons implémenter la méthode update de cette manière. Celle-ci gère l’affichage de chaque cellule différemment en fonction de leur type.

## Les différents ThermoSystem :

Plus tôt, nous avons vu que le bouton start faisait appel à une méthode *startSimulation()*, celle-ci aura pour but de lancer la timeline qui elle-même aura pour but d’exécuter certaines actions à chaque seconde de la simulation. Tout d’abord nous mettons à jour le temps avec le timer actuel, ensuite nous mettons à jour la température extérieure à jour grâce au Parser qui a été chercher toutes les températures dans un fichier fourni au préalable. Après ça, nous pouvons exécuter la méthode de simulation adéquate en fonction du mode chauffe que nous avons sélectionnée, c’est ici que le Design Pattern Strategy entre en jeu. Il a été primordial d’inclure ce Design Pattern ici car comme dit plus haut en fonction du mode chauffe sélectionné nous souhaitons un comportement différent de la part de l’algorithme.

### ThermoSystemManual :

La class ThermoSystemManual.java représente le système de chauffe en mode manuel à chaque itération de la timeline sa méthode simulation() est appelé dans le contrôleur :

Une image contenant texte, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran

Description générée automatiquement

Pour chaque cellule du système leur température va être recalculé en fonction de toutes les cellules autour de celle-ci en plus d’elle-même diviser par le nombre de cellule voisine moins le nombre de cellule morte car elle ne rentre pas en compte dans le calcul. Si une cellule est collée à une bordure c’est la température extérieure que nous prenons en compte. Si la cellule est une source de chaleur c’est la température de chauffe que nous gardons. A la fin de ce calcule la cellule notifie la vue pour modifier l’affichage de celle-ci. Voilà la simulation par défaut de l’application dans ce mode l’utilisateur peut gérer comme bon lui semble les cellules, le système n’interviendra pas.

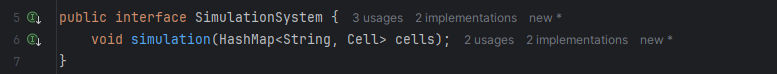
### ThermoSystemTarget :

La class ThermoSystemTarget.java représente le mode de chauffe Target du système celui-ci a pour but de viser la température moyenne de 20°. Lorsque le système est en dessous il va activer toutes les sources de chaleurs de celui-ci pour atteindre ce niveau, lorsque c’est l’inverse il va toutes les désactiver. Cette class hérite de *ThermoSystemManuel* étant donné que nous souhaitons garder la même simulation mais y rajouter un nouveau comportement qui est la gestion des sources de chaleur.

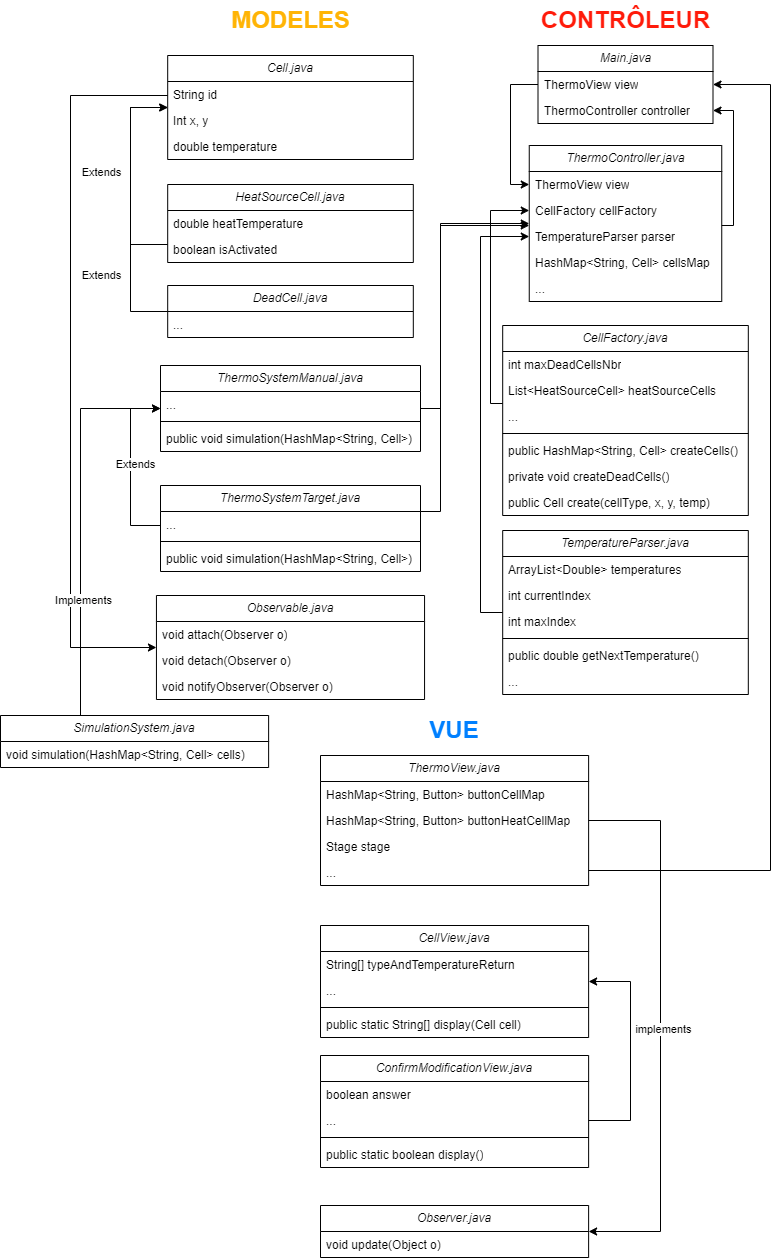
Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

D’abord nous calculons la température moyenne du système pour ensuite parcourir toute les cellules source de chaleur et les activer ou désactiver en conséquence. Voici l’interface qui a été implémenter pour pouvoir changer de Strategy quand il le faut :



# **Analyse et Application des Design Patterns**



L’application est basée sur une architecture MVC (Modèle-Vue-Contrôleur), ce qui en fait un projet plus volumineux mais mieux maintenable. Le point d’entrée de notre programme se situe dans la méthode *start()* de la classe *Main.java*. Cette dernière contient un attribut *controller* et *view*. Le constructeur de la classe *ThermoController.java* reçoit l’attribut *view* de la classe *Main.java*. Ensuite, dans le constructeur du contrôleur, nous instancions la classe *TemperatureParser.java* afin d’établir la température extérieure de notre système dans une variable *public* *static*. Par la suite, nous appelons la méthode *initView()* de la vue pour créer et afficher la fenêtre principale de notre application, à savoir l’interface de contrôle. Juste après, c’est la classe *CellFactory.java* qui est initialisée pour ensuite appeler sa méthode *createCells()* et créer la *HashMap* des cellules de notre application. Pour terminer, ce sont les méthodes *setCellButtonsActions()*, *setLeftButtonsActions()*, *setModeMenuActions()* et *setStageCloseEvent()* qui sont appelées pour établir le comportement de chaque bouton de notre interface. Pour cela, nous utilisons de nouvelles « sous-vues » pour gérer l’affichage du formulaire de paramètres d’une cellule grâce à la classe *CellView.java*, ainsi que l’affichage d’un pop-up de vérification avec *ConfirmModificationView.java*.

Du côté des modèles, nous avons une classe *Cell.java* qui contient les différents attributs d’une cellule de notre application en plus de ceux nécessaire à l’implémentation de l’interface *Observable.java* qui contient les méthodes nécessaires au bon fonctionnement du *pattern* *Observer*. Pour cela, la classe *ThermoView.java* implémente l’interface *Observer.java* qui contient la méthode *update()* pour mettre à jour l’affichage de certains objets modèle. La classe *Cell.java* est la classe mère de deux classes filles : *HeatSourceCell.java* *et DeadCell.java*. De plus, nous avons *ThermoSystemManual.java*, qui représente le mode de chauffe manuel de notre application. Cette dernière contient une seule méthode *static* *simulation()* qui a pour but de calculer la nouvelle température et de l’appliquer à l’ensemble des cellules du tableau. Il s’agit de la classe mère de la classe *ThermoSystemTarget.java*, qui représente le mode de chauffe *target*. Cette classe réutilise la méthode *simulation()* de sa classe mère pour rajouter une logique qui lui est propre. Cependant, ce comportement est variable puisqu’il dépend du choix de l’utilisateur dans le mode de chauffe. C’est pour cela que ces deux classes implémentent l’interface *SystemSimulation.java* pour répondre à la problématique du *design pattern Strategy*.

# **Limitations**

Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquementLors de la modification de la température d’une cellule source de chaleur il est obligatoire d’utiliser une valeur positive et inférieur à 100 car si ce n’est pas le cas l’application crash. Pour éviter cela, nous avons incorporé un message d’erreur pour s’assurer que l’utilisateur entre bel et bien une valeur adéquate :

Figure 3 message d'erreur de la cellule

# **Conclusion**

Pour en conclure avec ce rapport, nous avons vu que chaque fonctionnalité de ce projet a été implémenté sur une base de structure MVC tout en respectant différents *design patterns* tel qu’un *Observer*, deux *Strategies* et une *Factory*. Cet aspect de développement fût riche en apprentissages non seulement dans le domaine des *designs patterns* mais également dans le développement d’application Java avec interface graphique. *JavaFX* est un *Framework* riche et puissant en termes d’application de bureau et nous sommes persuadé d’avoir encore pleins de choses à apprendre de ce dernier. Nous espérons que cette release de l’application contentera toutes les demandes clientes aux vues du travail qui a été fournis.