

Programmation Java IV

2023-2024

Rapport de projet : HELBThermo

Table des matières

Introduction..... 2

Fonctionnalités de base..... 2

 Interface de contrôle : 2

 Les différents types de Cellules : 4

 - Cell :..... 4

 - HeatSourceCell :..... 5

 - DeadCell :..... 5

 Les différents ThermoSystem :..... 6

 - ThermoSystemManual : 7

 - ThermoSystemTarget : 8

Analyse et Application des Design Patterns 9

Limitations 10

Conclusion 11

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 lieu 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.

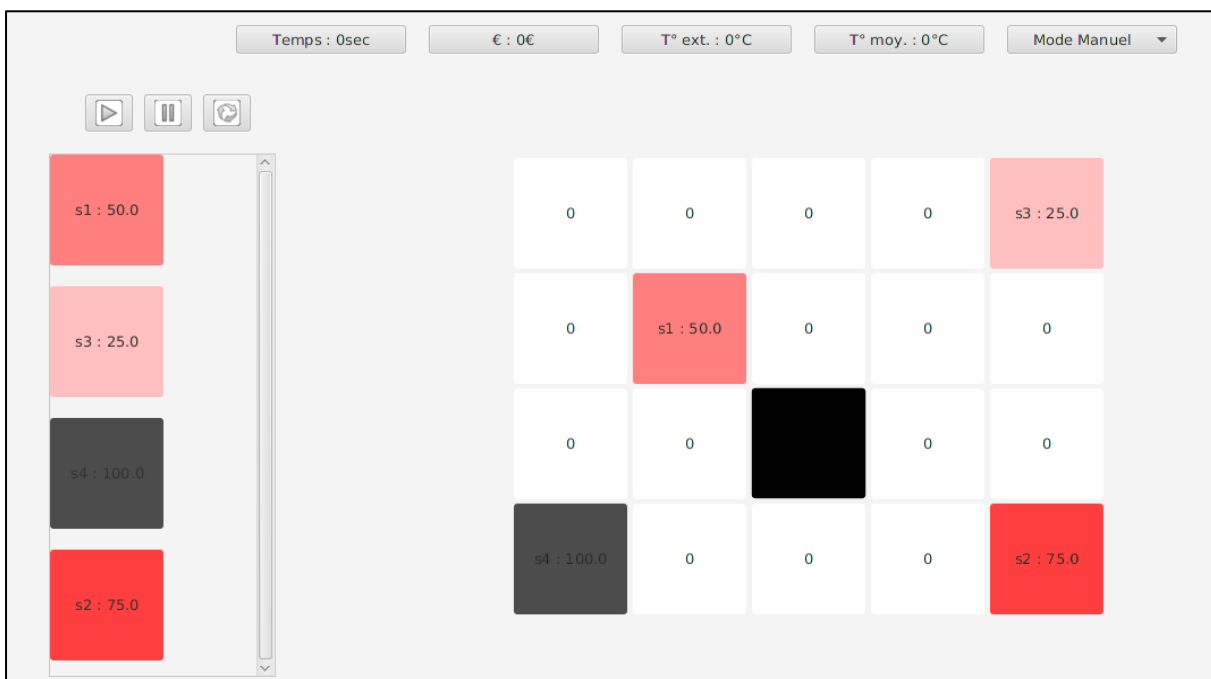
Dans ce dernier, nous allons commencer par parcourir l'ensemble des fonctionnalités qui ont été demandées d'implémenter dans ce projet ainsi que de la manière dont cela a été fait. Nous allons parler du design graphique mais aussi architecturale du projet, une analyse sera fournie afin de vérifier que 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 :

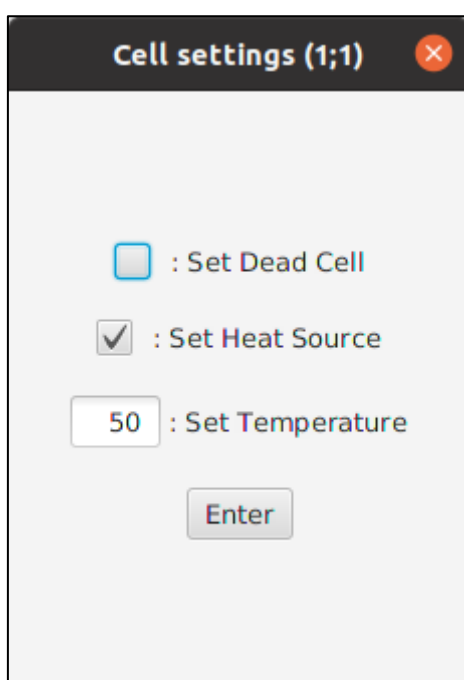
Une interface a été implémentée grâce au *Framework JavaFX*, cela a été fait en respectant le *Design Pattern MVC* cela signifie que nous avons une class *ThermoView.java* qui sera la vue principale de notre application ainsi qu'un *ThermoController.java* et différents modèles que nous verrons plus tard.



Comme nous pouvons le voir l'interface fournit 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.

Lorsque l'utilisateur clique sur l'une d'elle une nouvelle interface apparaît pour régler les paramètres de la cellule cliqué.



Cell settings (1;1)

☐ : Set Dead Cell

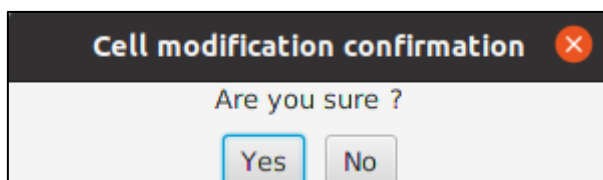
☒ : Set Heat Source

50 : Set Temperature

Enter

Figure 1 interfaces de configuration d'une cellule

Ici 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.



Cell modification confirmation

Are you sure ?

Yes No

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.

Dans la partie gauche de l'interface nous pouvons retrouver les différentes sources de chaleur, à partir de là l'utilisateur peut 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.

```

76     private void setLeftButtonsActions() { 1 usage new *
77         view.getPlayButton().setOnAction(event -> {
78             if (!isSimulationStarted) {
79                 startSimulation();
80                 isSimulationStarted = true;
81             } else {
82                 timeline.play();
83             }
84         });
85
86         view.getPauseButton().setOnAction(event -> {
87             timeline.pause();
88             isSimulationStarted = false;
89         });
90
91         view.getResetButton().setOnAction(event -> {
92             timer = 0;
93             cost = 0;
94             view.resetView();
95         });
96     }

```

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 : « $s1.temperature^2 + s2.temperature^2 + s3.temperature^2 + \text{etc...}$ ».

Les différents types de Cellules :

- Cell :

La 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 :

```

15     public Cell(int x, int y) { new *
16         this.id = ""+x+y;
17         this.x = x;
18         this.y = y;
19         this.temperature = ThermoController.tempExt;
20         this.observers = new ArrayList<>();
21     }

```

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 température est établie de base à celle de l'extérieur. Une cellule implémente également l'interface *Observable.java*.

```

3 public interface Observable { 1 usage 3 implementations new *
4     void attach(Observer o); 3 usages 1 implementation new *
5     void detach(Observer o); no usages 1 implementation new *
6     void notifyObserver(); 6 usages 1 implementation new *
7 }

3 public interface Observer { 7 usages 1 implementation new *
4     void update(Object o); 1 usage 1 implementation new *
5 }

```

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 :

```

8 public HeatSourceCell(int x, int y, double heatTemperature) { 1 usage new *
9     super(x, y);
10    setTemperature(heatTemperature);
11    this.isActivated = true;
12    this.heatTemperature = heatTemperature;
13 }

```

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.

```

17 @Override 13 usages new *
18 public double getTemperature() {
19     if (isActivated) {
20         return heatTemperature;
21     } else {
22         return super.getTemperature();
23     }
24 }

```

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.

```

3 public class DeadCell extends Cell{ 11 usages new *
4
5     public DeadCell(int x, int y) { 1 usage new *
6         super(x, y);
7         setTemperature(0);
8     }
9
10    // Sécurité pour ne pas changé de temperature
11    @Override 4 usages new *
12    public void setTemperature(double temperature) { super.setTemperature(0.0); }
15 }

```

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

```

279 @Override 1 usage new *
280 public void update(Object o) {
281     if (o instanceof HeatSourceCell) {
282         HeatSourceCell heatSourceCell = (HeatSourceCell) o;
283         if (!buttonHeatCellMap.containsKey(heatSourceCell.getId())) {
284             createHeatCell(heatSourceCell.getId(), heatSourceCell.getTemperature());
285         } else {
286             updateHeatCell(heatSourceCell.getId(), heatSourceCell.getTemperature(), heatSourceCell.isActivated());
287         }
288
289         if (!heatSourceCell.isActivated()) {
290             disableHeatSourceCell(heatSourceCell.getId());
291         } else {
292             enableHeatSourceCell(heatSourceCell.getId(), heatSourceCell.getTemperature());
293         }
294     } else if (o instanceof DeadCell) {
295         DeadCell deadCell = (DeadCell) o;
296         removeHeatCell(deadCell.getId());
297         createDeadCell(deadCell.getId());
298     } else {
299         Cell cell = (Cell) o;
300         removeHeatCell(cell.getId());
301         updateCell(cell.getId(), cell.getTemperature());
302     }
303 }

```

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 :

```

12      @Override 2 usages 1 override new *
13      public void simulation(HashMap<String, Cell> cells) {
14          double newTemp;
15
16          for (Cell cell : cells.values()) {
17              int x = cell.getX();
18              int y = cell.getY();
19              outOfRangeCellNbr = 0;
20              deadCellNbr = 0;
21
22              newTemp = cell.getTemperature();
23              newTemp += getTemperature(cells, x: x - 1, y); // Top
24              newTemp += getTemperature(cells, x: x + 1, y); // Bottom
25              newTemp += getTemperature(cells, x, y: y - 1); // Left
26              newTemp += getTemperature(cells, x, y: y + 1); // Right
27              newTemp += getTemperature(cells, x: x - 1, y: y - 1); // Top Left
28              newTemp += getTemperature(cells, x: x - 1, y: y + 1); // Top right
29              newTemp += getTemperature(cells, x: x + 1, y: y - 1); // Bottom left
30              newTemp += getTemperature(cells, x: x + 1, y: y + 1); // Bottom Right
31              newTemp += (ThermoController.tempExt * outOfRangeCellNbr); // Ext Cell
32              newTemp /= maxCellNeighbors - deadCellNbr; // DeadCells
33
34              if (cell instanceof HeatSourceCell) {
35                  if (((HeatSourceCell) cell).isActivated()) newTemp = cell.getTemperature();
36              }
37
38              cell.setTemperature(newTemp);
39              cell.notifyObserver();
40          }
41      }

```

```

43      @
44      private double getTemperature(HashMap<String, Cell> cells, int x, int y) { 8 usages new *
45          String cellKey = x + " " + y;
46          Cell neighborCell = cells.get(cellKey);
47          if (neighborCell != null) {
48              if (neighborCell instanceof DeadCell) {
49                  deadCellNbr++;
50              }
51              return neighborCell.getTemperature();
52          } else {
53              outOfRangeCellNbr++;
54              return ThermoController.tempExt;
55          }
56      }
57  }

```

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 calcul 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.

```

10      @Override 2 usages new *
11      public void simulation(HashMap<String, Cell> cells) {
12          super.simulation(cells);
13          int deadCells = 0;
14
15          for (Cell c : cells.values()) {
16              if (c instanceof DeadCell) {
17                  deadCells++;
18              }
19              avgTemp += c.getTemperature();
20          }
21
22          avgTemp /= cells.size() - deadCells;
23
24          for (Cell c : cells.values()) {
25              if (c instanceof HeatSourceCell) {
26                  HeatSourceCell heatSourceCell = (HeatSourceCell) c;
27                  if (avgTemp < targetTemperature) {
28                      heatSourceCell.activate();
29                  } else {
30                      heatSourceCell.deactivate();
31                  }
32                  heatSourceCell.notifyObserver();
33              }
34          }
35      }

```

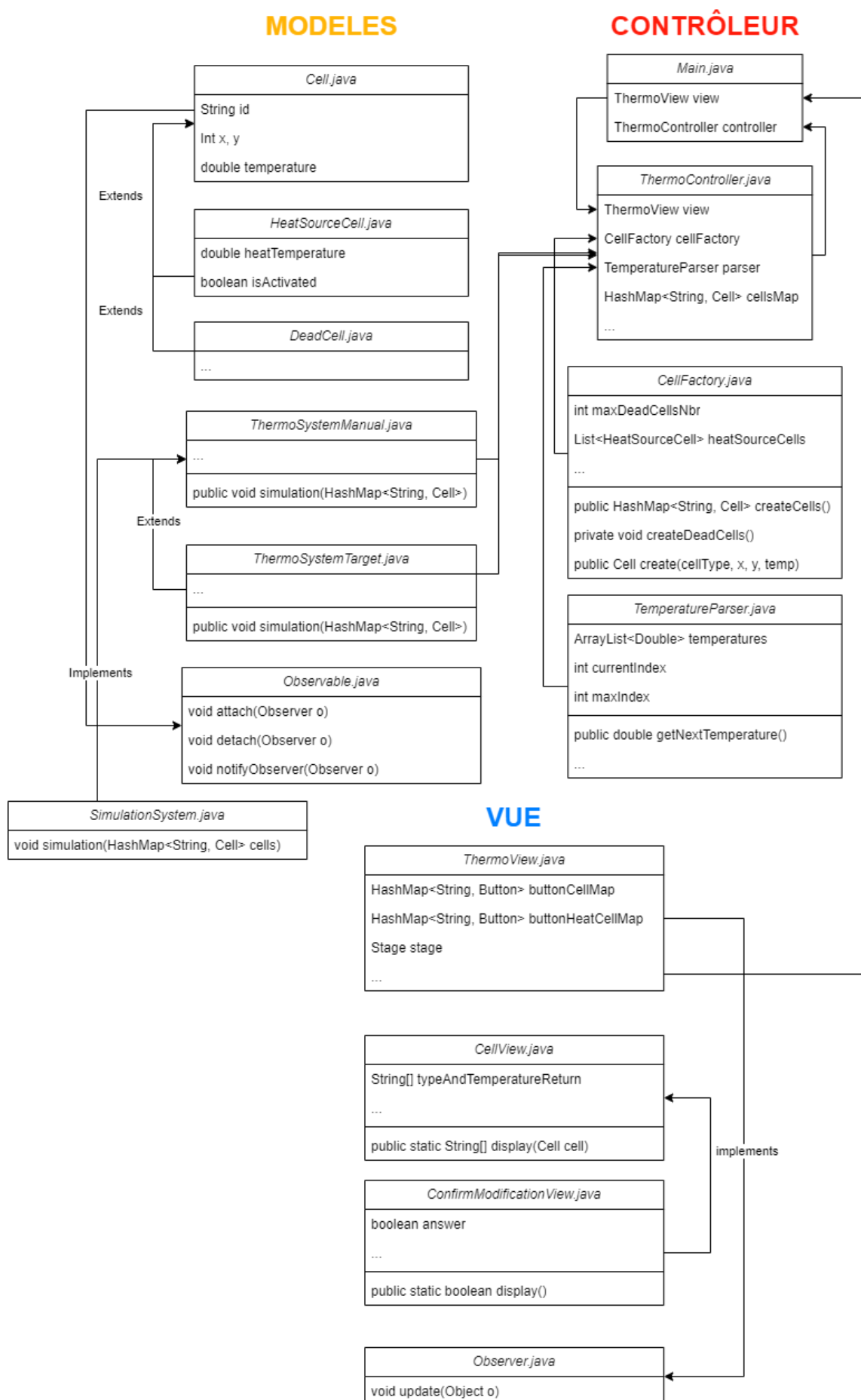
D'abord nous calculons la température moyenne du système pour ensuite parcourir toute les cellules source de chaleur et les activer ou désactiver en conséquence. Voici l'interface qui a été implémenter pour pouvoir changer de Strategy quand il le faut :

```

5      public interface SimulationSystem { 3 usages 2 implementations new *
6          void simulation(HashMap<String, Cell> cells); 2 usages 2 implementations new *
7      }

```

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

Lors de la modification de la température d'une cellule source de chaleur il est obligatoire d'utiliser une valeur positive et inférieure 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 :

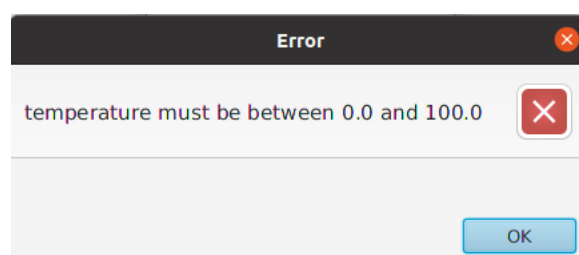


Figure 3 message d'erreur de la cellule

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.