Grapheur

Document d'architecture et de conception

Réalisé par : GEHIER Kylian

Encadré par : Jean-Marc Perronne – Enseignant chercheur à l'ENSISA

Table des matières

Index des illustrations

1 Présentation générale du projet

Le projet présenté dans ce document d'architecture et de conception est une API ayant pour but l'affichage de Graphes (Charts en anglais).

Ce projet m'a été attribué par Jean-Marc Perronne comme projet de fin d'étude pour mon diplôme d'ingénieur informatique et réseau.

1.1 Une première conception du projet (2012)

Le projet Grapheur à vu le jour en 2012. Sa conception avait été confiée à Jean-Marc Perronne par l'entreprise INFRAL pour de la télémédecine.

Le projet avait finalement été interrompu pour des raisons qui me sont inconnues.

Avant son interruption, le Grapheur était alors capable :

- D'afficher deux types de graphes qui seront détaillés plus tard dans le document : les LineChart et les MapChart
- De lire un fichier Audio statiquement ou de simuler une acquisition en temps réel depuis ce même fichier audio
- D'enregistrer des données depuis un Micro puis de les ajouter statiquement ou dynamiquement à un Chart.

1.2 Reprise du projet (2018)

Dans le cadre d'une projet de troisième année d'école d'ingénieur, il m'a été demandé par Jean-Marc Perronne de reprendre ce Grapheur et d'y apporter diverses modifications listées dans le cahier des charges ci-dessous.

Le projet ayant été avorté en 2012, ce dernier n'avait pas été documenté. Il m'a donc fallut, dans un premier temps, me familiariser avec l'API et en comprendre les rouages.

1.3 Cahier des charges

Cahier des charges imposé :

- Dés-Infraliser le projet : nettoyer toute trace d'Infral présente dans le projet
- Généraliser le Grapheur :
  - permettre l'ajout de données statiques venant d'un CSV
  - implémenter une Factory (Usine à Graphe) pour simplifier l'utilisation du Grapheur
- Nettoyer le code (prints, commentaires) et les packages du projet
- Rédaction d'un document de conception et d'architecture (Ce document)

Mes propositions validées par M.Perronne :

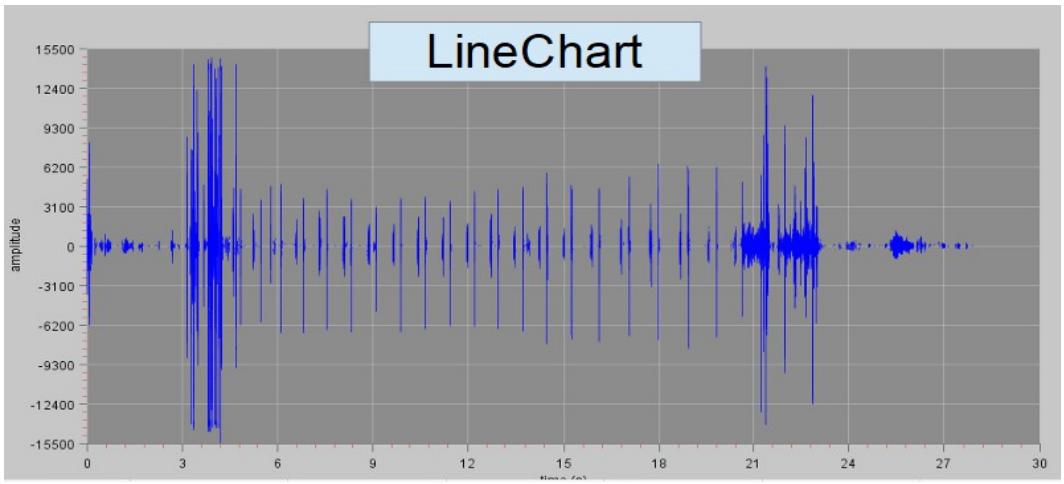
- Implémenter un système de fichier de configuration pour le paramétrage des Charts.
- Création d'un modèle d'abstraction « Entry » regroupant les différents types de données entrantes afin de faciliter l'ajout de données sur un Chart.

2 Architecture

L'une des particularités de ce Grapheur est qu'il se base sur un système d'axes. Cette particularité permet la création de différents types de Chart au sein du même projet.

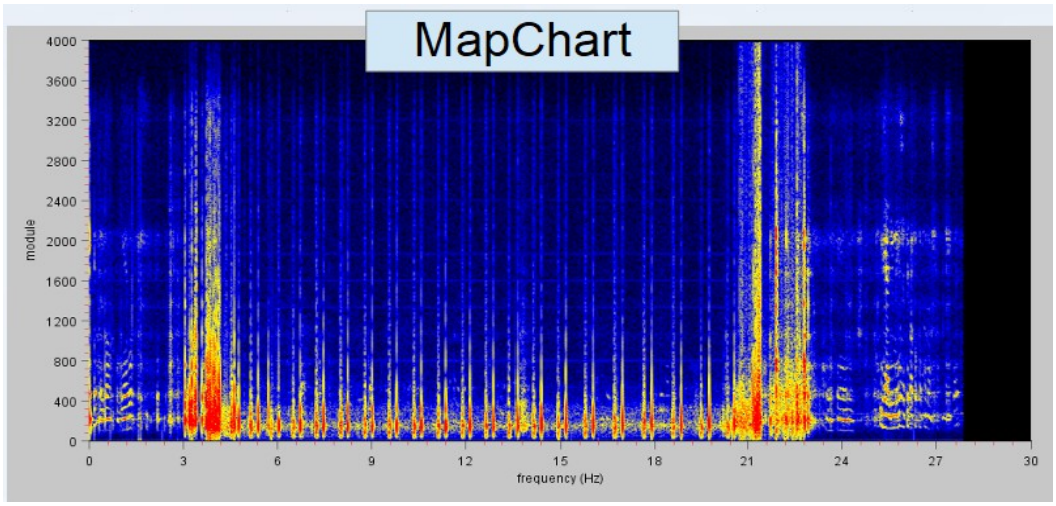
On distingue à l'heure actuelle deux types de Chart que sont les LineCharts et les MapCharts.

- Un LineChart est un Graphe 2D dans un repère orthonormé :



Présentation d'un LineChart

- Un MapChart est une Graphe 2D comprenant un champ de profondeur colorimétrique donnant un aspect de Graphe 3D

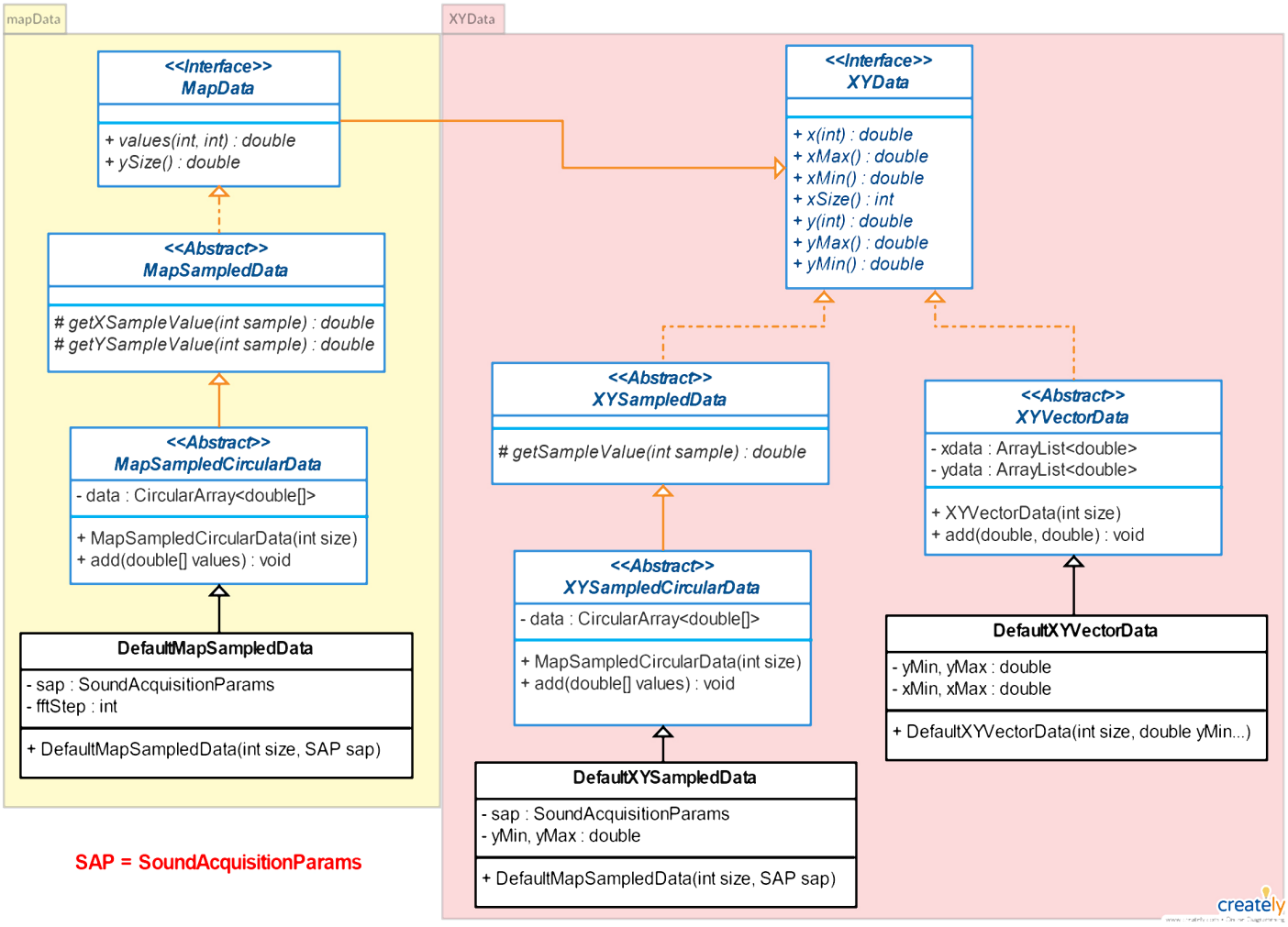


Présentation d'un MapChart

2.1 Modèle de données

Les Charts utilisent un modèle de données qui leur est propre :

Diagramme de classe – Modèle de données



Comme le montre le diagramme de classe ce-dessus, il existe deux interfaces principales :

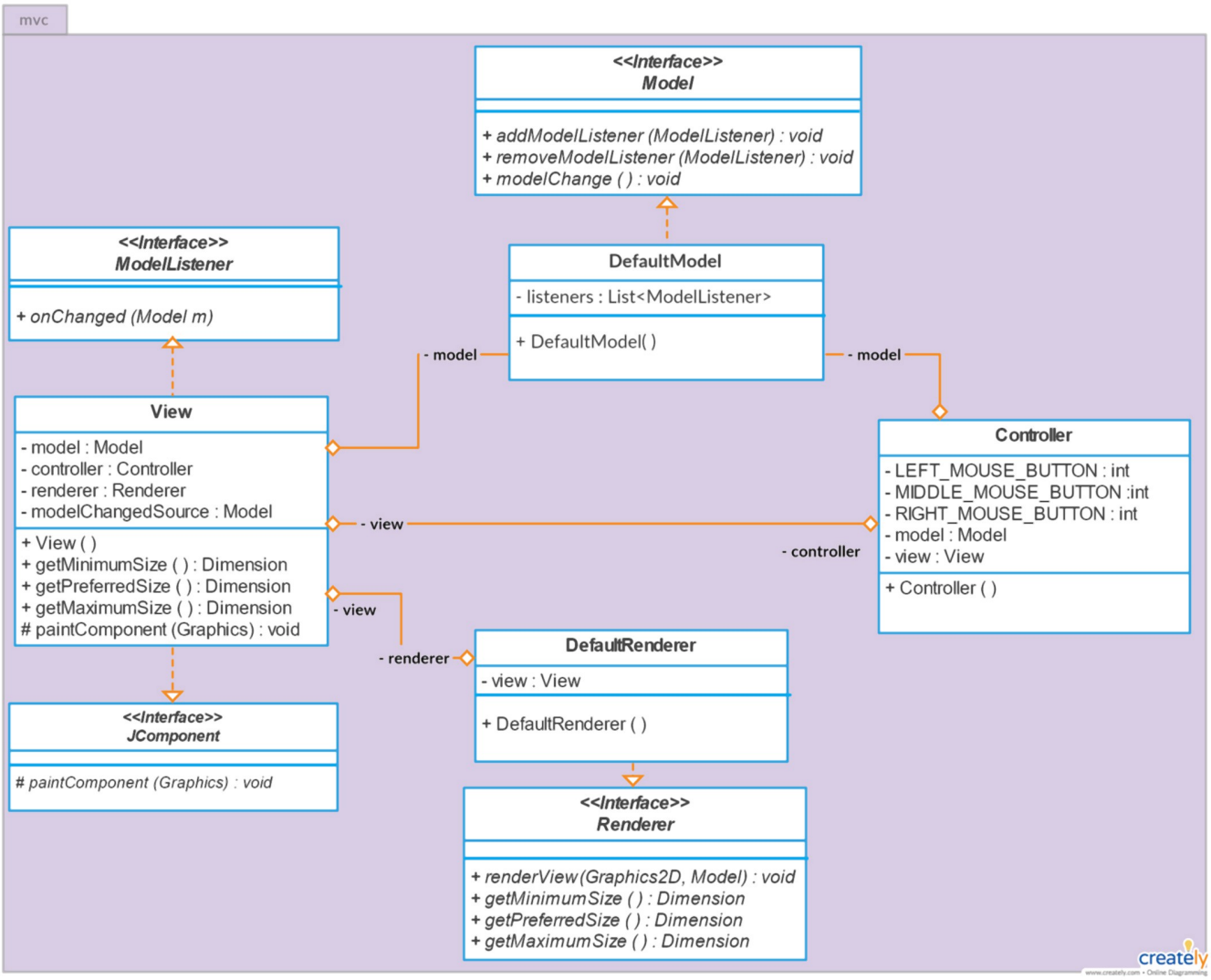
- XYData : modèle de données utilisé par les LineChart
  - XYSampledData : utilisé pour des données échantillonnées (car cela implique un intervalle constant dx)
  - XYVectorData : utilisé pour des données non-échantillonnées (ce qui implique le besoin d'utiliser un tableau de donnée en x)

- MapData : modèle de données utilisé par les MapChart
  - MapSampledData : utilisé pour des données échantillonnées
  - MapVectorData : il n'y pas de cas d'utilisation impliquant des données non-échantillonnées dans ce document. La classe n'a donc pas encore été implémentée.

2.2 MVC

Ce Grapheur repose sur une architecture MVC (Modèle – Vue – Contrôleur) Swing à laquelle vient s'ajouter un Renderer.

Diagramme de classe – MVC

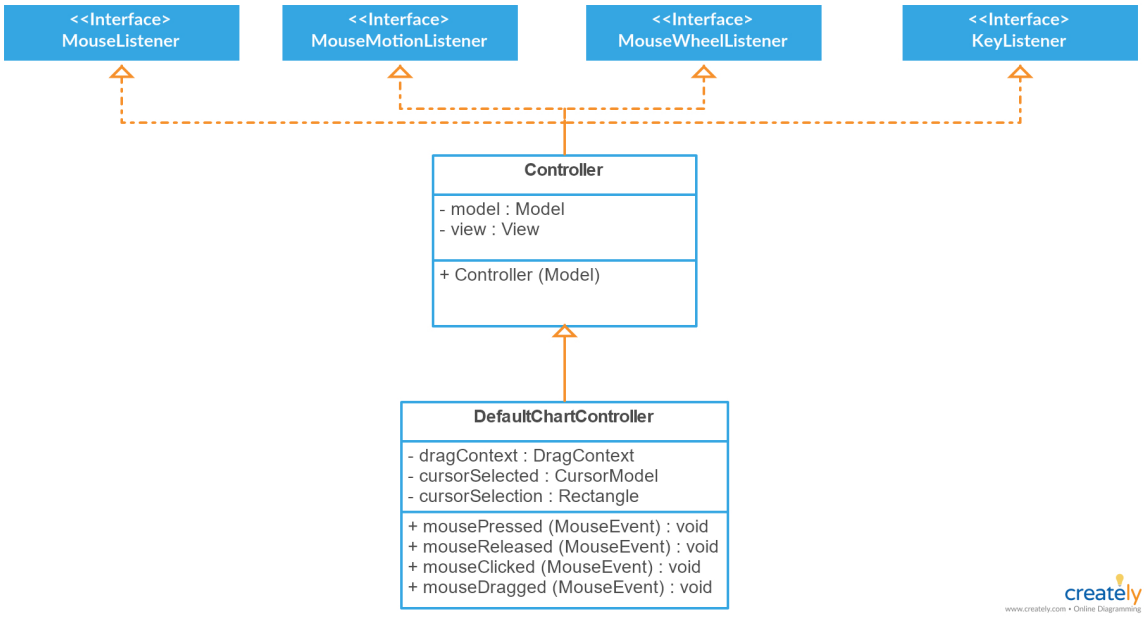


2.1 Description des différents composants du MVC

- **Model (Modèle)** : contient les données du Chart sous une forme compréhensible par la Vue et la notifie lorsque des changements lui sont appliqués.

- **Controller (contrôleur)** : réceptionne les différentes actions effectuées depuis la Vue grâce aux Listener qu'il Implémente (cf diagramme ci-dessous) puis modifie le modèle en conséquence.

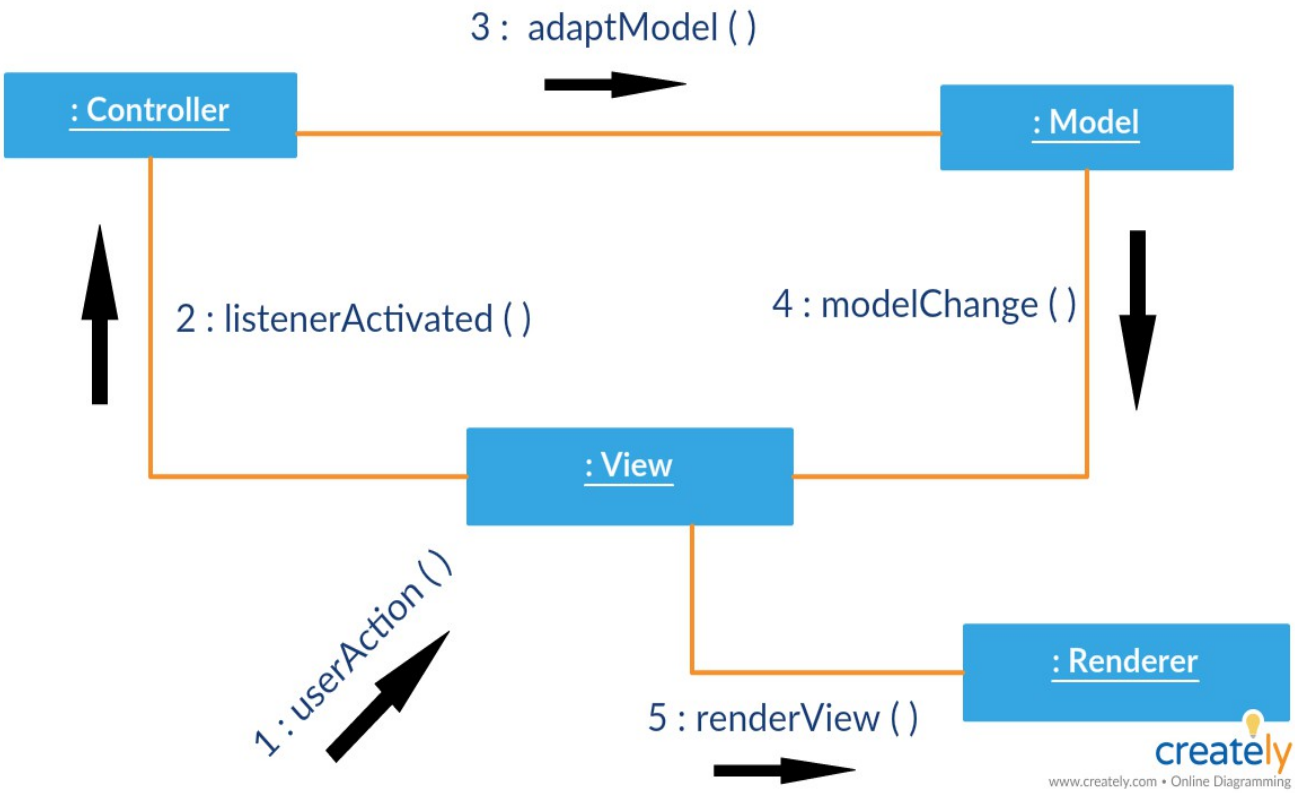
Diagramme de classe – Controller



- **Renderer** : chargé par la Vue de dessiner les modèles en récupérant leur données et leurs attributs.

- **View (Vue)** : c'est l'IHM (Interface Humain Machine). Les modèles y sont dessinés par les renderers associés à chaque modèle. La Vue est donc le Chart.

Diagramme de communication – Action d'un utilisateur sur la Vue



Comme le montre le diagramme ci-dessus, lorsqu'un utilisateur effectue une action depuis la Vue (ex : click de souris), cette dernière active un ou plusieurs Listener(s) (écouteur) implémenté(s) par le contrôleur.

Le contrôleur modifie alors le ou les modèle(s) concerné(s) qui va/vont a leur tour notifier la Vue qu'il(s) a/ont été modifié(s) via la méthode « modelChange » commune à tous les modèles.

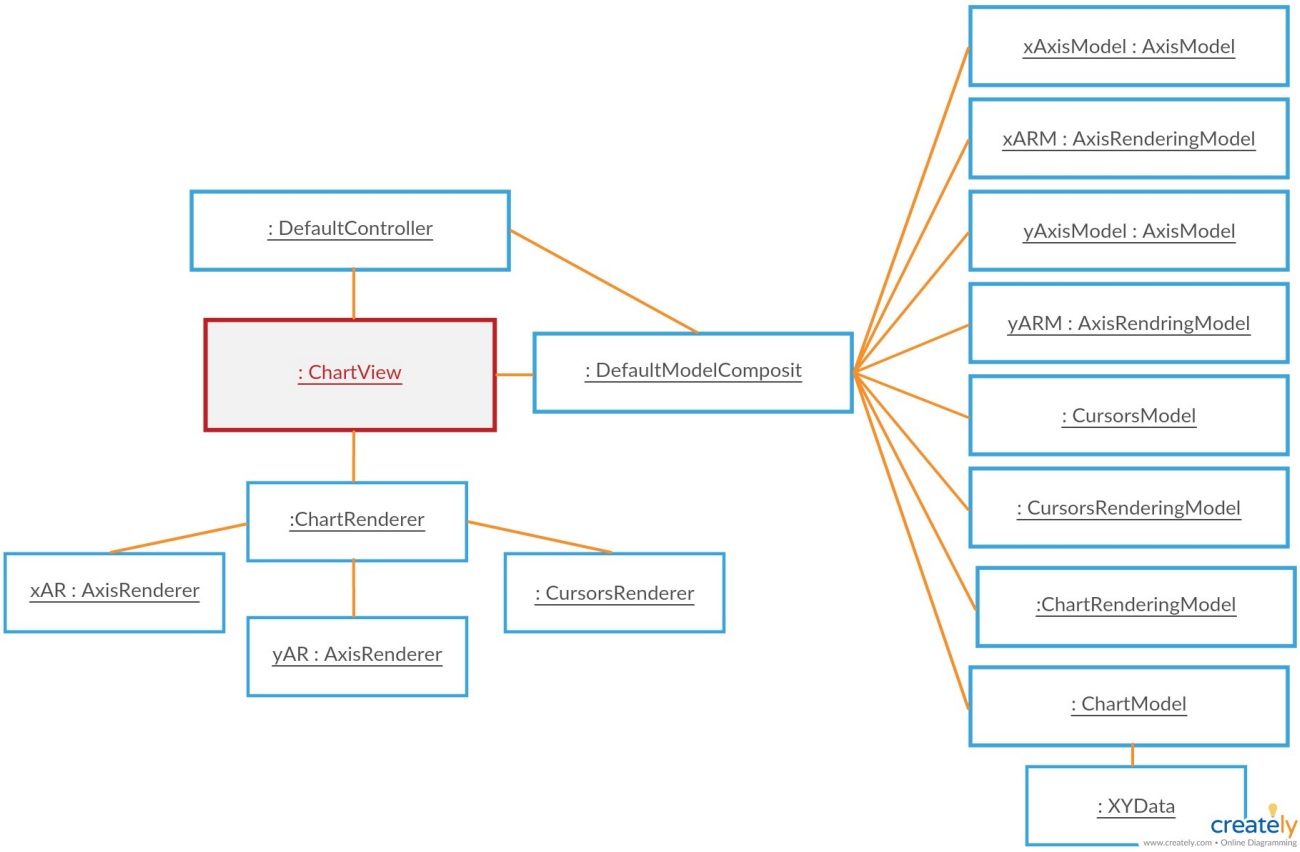
La Vue appel alors sa méthode « paintComponent » (Cf. Diagramme de classe MVC) qui délégue le dessin de tous les modèles aux différents renderes.

2.2 Les composants d'un Chart

Nous venons de voir qu'un Chart était en fait une Vue déclinée du MVC.

Nous parlerons donc maintenant de « ChartView » pour désigner un Chart et inversement.

Diagramme d'objets – Composition d'un Chart



Comme nous pouvons le voir sur le diagramme ce-dessus, un ChartView est donc composé de deux modèles d'axes (x et y), d'une modèle de curseurs et de son modèle de Graphe. Ces différents modèles seront détaillés plus loin.

Il est également composé d'un modèle de rendu (RenderingModel) pour chaque modèle contenant les données nécessaires à l'affichage graphique de chacun.

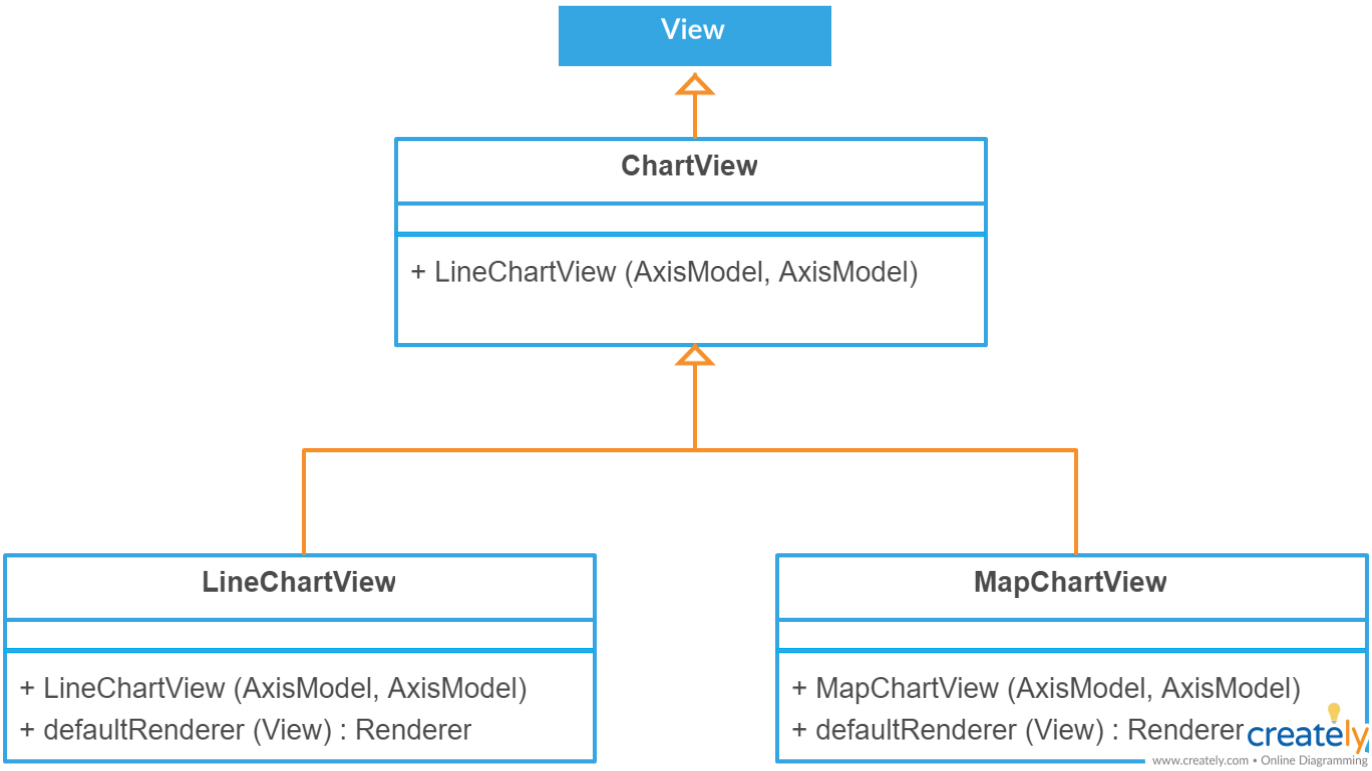
Chaque modèle se voit également attribué un Renderer qui sera chargé de le dessiner en récupérant les informations contenu dans ses Model et RenderingModel associés.

Un model peut donc être considéré comme un triplet : (Model, RenderingModel, Renderer).

Nous pouvons également remarqué la présence d'un pattern Composite qui sera également détaillé plus loin dans le document.

2.3 ChartView & Modèle Composite

Diagramme de classe – ChartView



Nous pouvons voir d'après ce diagramme qu'il existe deux types de vues implémentées pour le moment, une pour les LineChart et une autre pour les MapChart. Si elles semblent similaires d'un point de vue « UML : diagramme de classe », elle sont bien différentes au niveau de leur constructeur et de leur méthodes « defaultRenderer » respective.

La différence au niveau des méthodes « defaultRenderer » n'est autre que le sur type de Renderer retourné par cette dernière.

Naturellement, la méthode « defaultRenderer » de la classe « LineChartView » renverra un « LineChartRenderer », tandis que celle de la classe « MapChartView », un « MapChartRenderer ».

L'arborescence des classes héritant de « defaultRenderer » n'a pas encore été présentée et sera présentée un peu plus loin dans le document (Cf . 2.4 Renderer)

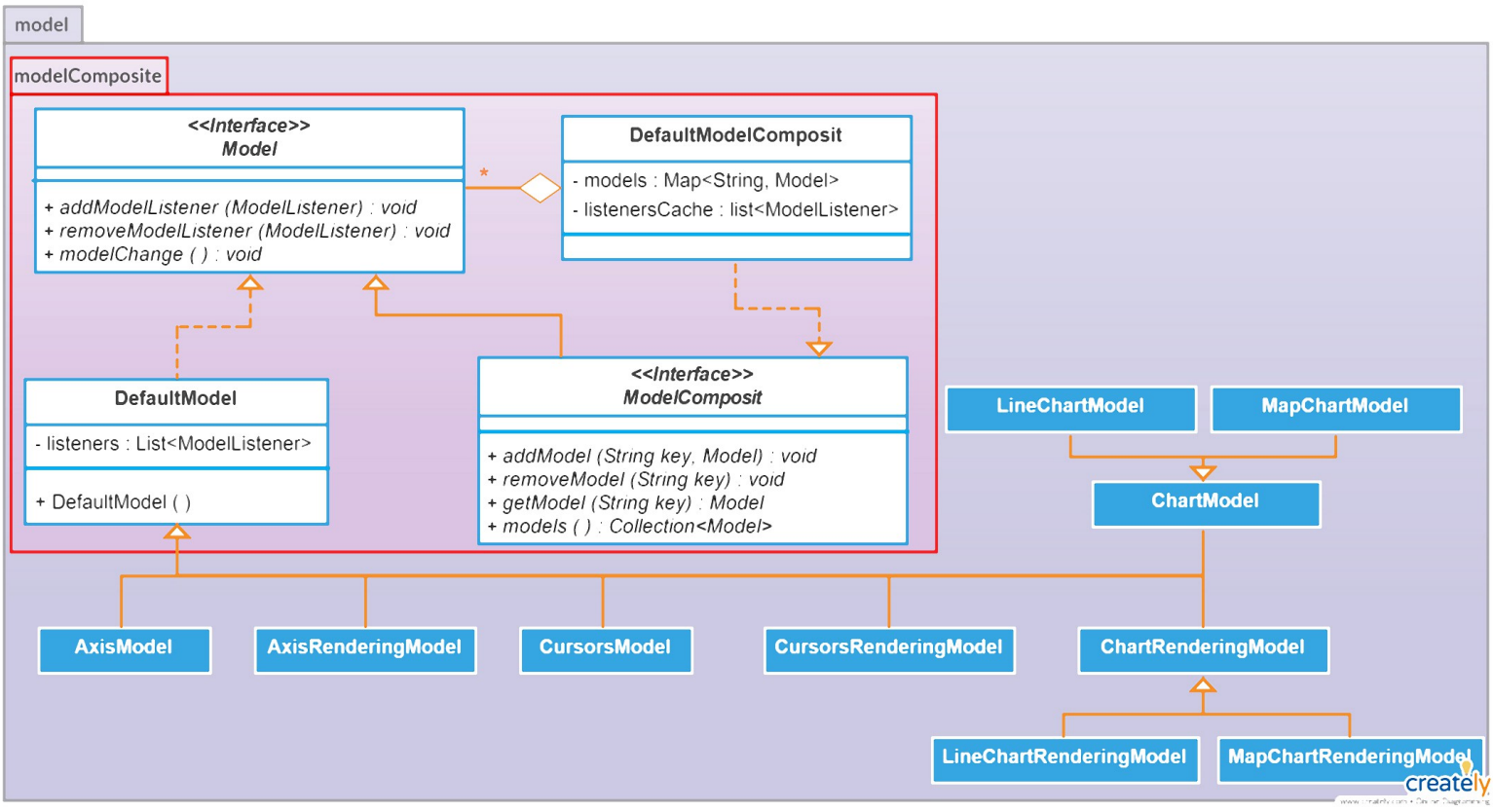
Regardons maintenant les différences au niveau des constructeurs.

ChartView - Constructor

```
public ChartView(AxisModel xAxis, AxisModel yAxis)
{
    super();

    xAxis.setSide(Side.Bottom);
    yAxis.setSide(Side.Left);

    ModelComposit m = new DefaultModelComposit();
    m.addModel(X_AXIS_MODEL, xAxis);
    m.addModel(Y_AXIS_MODEL, yAxis);
    m.addModel(X_AXIS_RENDERING_MODEL, new AxisRenderingModel());
    m.addModel(Y_AXIS_RENDERING_MODEL, new AxisRenderingModel());
    m.addModel(CURSORS_MODEL, new CursorsModel());
    m.addModel(CURSORS_RENDERING_MODEL, new CursorsRenderingModel());
    this.setModel(m);
    //in order to update display rectangle
    this.addComponentListener((ChartRenderer)this.renderer());
}
```

Le constructeur de la classe « ChartView » instancie un modèle dit composite.

Diagramme de classe – Model Composit



Ce modèle composite est instancié pour stocker les différents modèles contenus dans un « ChartModel » et ainsi être assigné comme étant le nouveau Model contenu en donnée membre dans la classe « View » du MVC (Cf .

Ces modèles sont stockés dans ce « ModelComposit » afin que ce dernier puisse être appelé par les constructeurs deux classes héritant de « ChartModel ».

Lors de l'instanciation d'un « LineChartView », les modèles suivant composeront donc son « ModelComposit » :

- **Modèles communs** (CF . ChartView -Constructor) aux deux types de charts
  - Deux « AxisModel » pour l'axe des abscisses et celui des ordonnées
  - Deux « AxisRenderingModel » pour le rendu des axes
  - Un « CursorsModel » pour stocker les curseurs pouvant être ajoutés par un utilisateur depuis la vue
  - un « CursorsRenderingModel » pour le rendu des curseurs
  -
- **Modèles propre aux LineChart**
  - un « LineChartModel » et son « LineChartRenderingModel »

MapChartView – Constructor

```
public LineChartView(AxisModel xAxis, AxisModel yAxis)
{
    super(xAxis, yAxis);

    ModelComposit m = (ModelComposit) this.getModel();
    m.addModel(CHART_MODEL, new LineChartModel());
    m.addModel(CHART_RENDERING_MODEL, new LineChartRenderingModel());

}
```

- **Modèles propre aux MapChart**
  - un « MapChartModel » et son « MapChartRenderingModel »

MapChartView – Constructor

```
public MapChartView(AxisModel xAxis, AxisModel yAxis)
{
    super(xAxis, yAxis);

    ModelComposit m = (ModelComposit) this.getModel();
    m.addModel(CHART_MODEL, new MapChartModel());
    m.addModel(CHART_RENDERING_MODEL, new MapChartRenderingModel());
}
```

2.4 Renderer

TO -DO :

- ✗ Diagramme de classe
- ✗ Explication de la cascade d'appel des différentes méthode renderX depuis renderView de View

2.5 Factory

Dans cette partie je ne redéfinirai pas la notion de « Pattern Factory ». Celle-ci est relativement bien expliquée dans ce cours : https://www.jmdoudoux.fr/java/dej/chap-design-patterns.htm

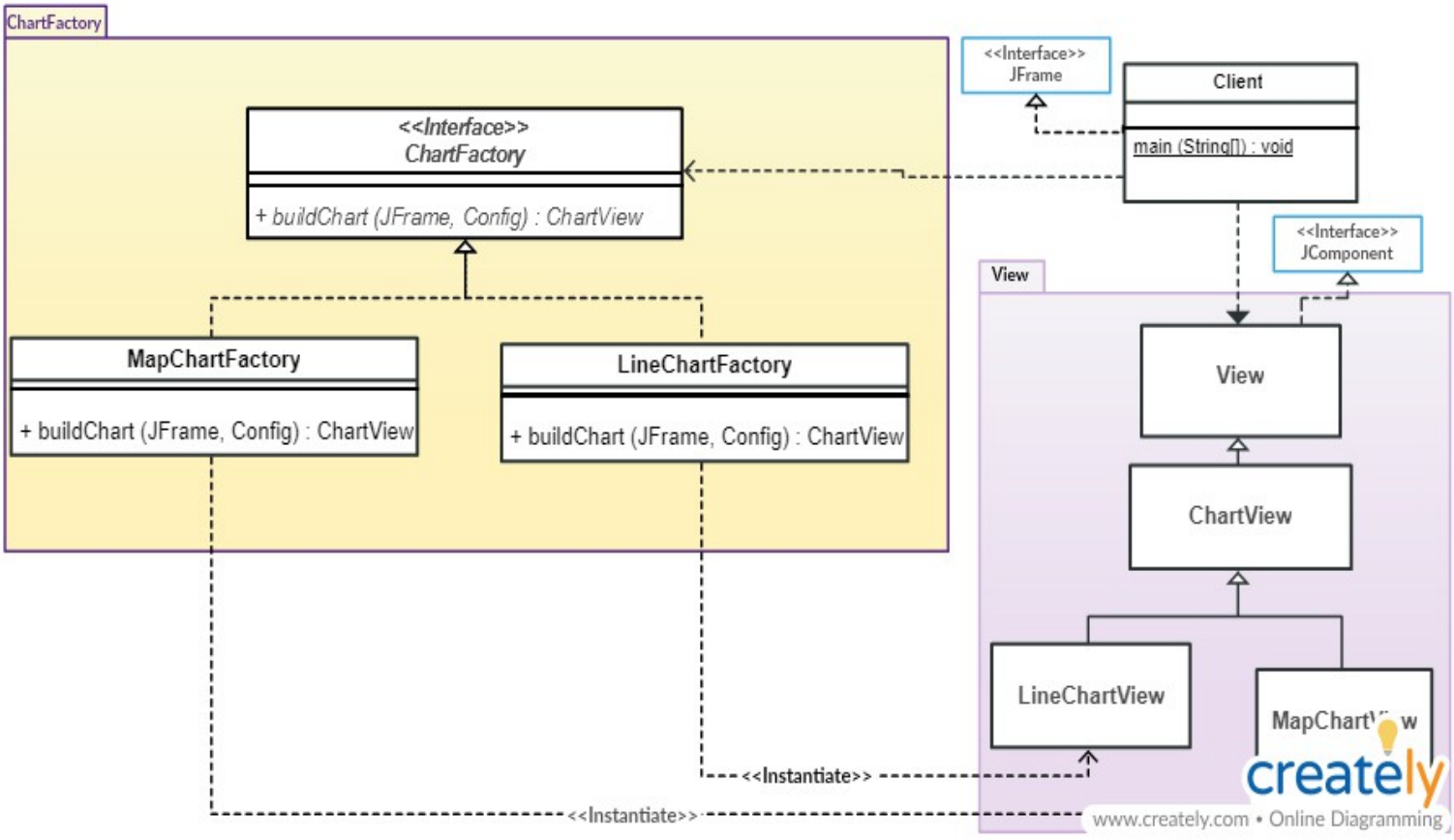
Nous pouvons à présent afficher des Graphes sous forme de « LineChart » ou de « MapChart ». Bien qu'il soient vides de données pour le moment, une premier problème se pose déjà.

2.5.1 Le besoin d'une Factory

Devant l'importance de la taille du code nécessaire à l'affichage d'un Graphe (une centaine de lignes de code effectives), il m'a semblé évident que l'implémentation d'une « Factory » était nécessaire.

Nous passons alors d'une centaine de lignes à une vingtaine de lignes de codes effectives pour l'affichage d'une Graphe vide de données.

Diagramme de Classe – Factory



2.5.2 Création d'un modèle de fichier de configuration

Au vu du nombres de paramètres (Visibilité des ticks, couleurs des axes, couleur du fond etc...) qu'un « Chart » peut posséder, j'ai donc décidé de créer un modèle de fichier de configuration « Config », relativement basique (il pourra être améliorer par la suite) et possédant son propre « Parseur ».

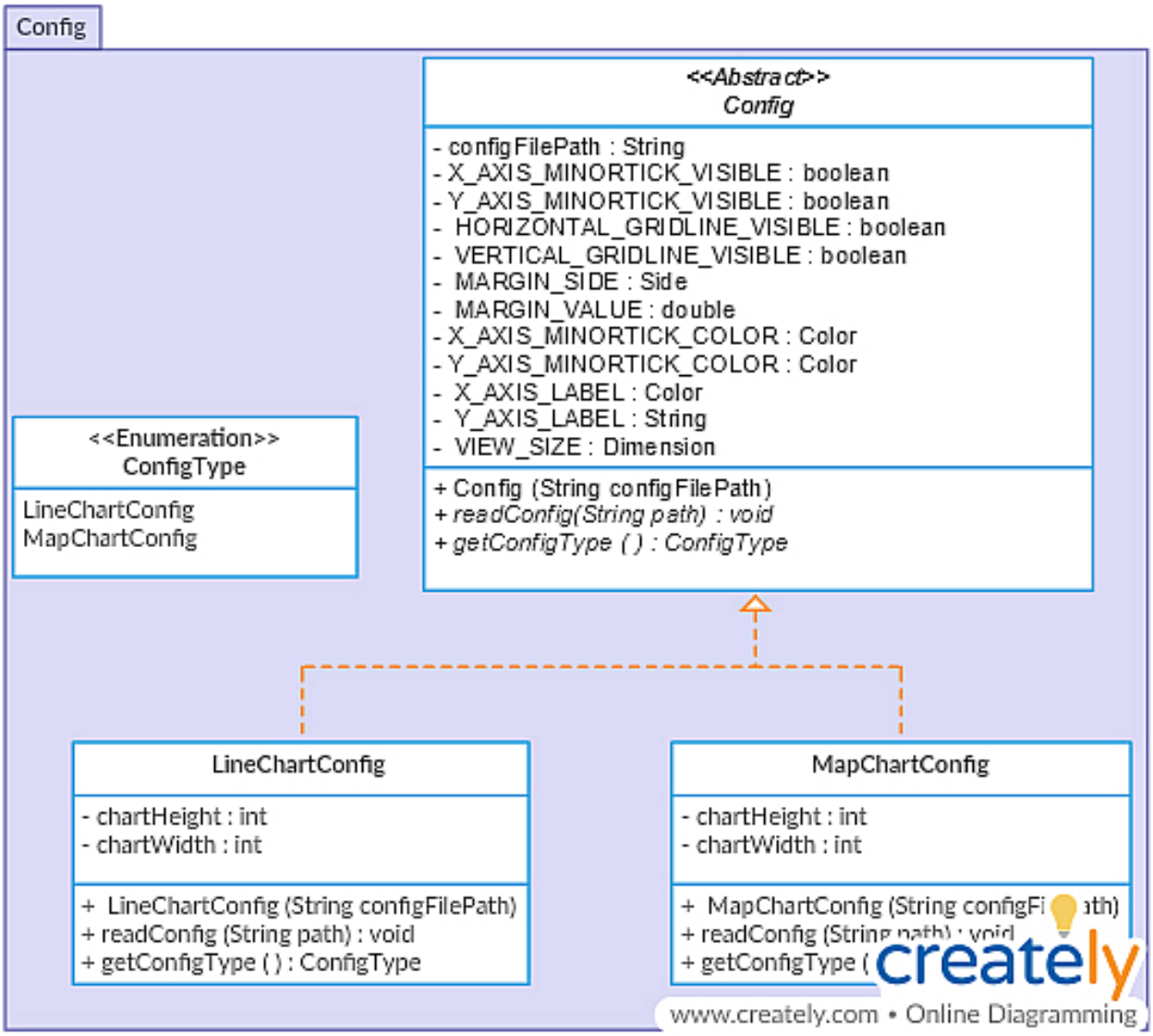
Présentation d'un fichier de configuration

```
// Boolean values

X_AXIS_MINORTICK_VISIBLE = true
Y_AXIS_MINORTICK_VISIBLE = true
HORIZONTAL_GRIDLINE_VISIBLE = true
VERTICAL_GRIDLINE_VISIBLE = true

// Choose between : Bottom,Left, Right and Top

MARGIN_SIDE = Left

// Double values

MARGIN_VALUE = 80

// Integer values
CHART_WIDTH = 1000
CHART_HEIGHT = 500

// String values
X_AXIS_LABEL = xLabel
Y_AXIS_LABEL = yLabel

// Color values (red, black, magenta,...)

X_AXIS_MINORTICK_COLOR = red
Y_AXIS_MINORTICK_COLOR = red

//
// Map : Spectrum configuration
//

//Integer
DATA_SIZE = 300000
SPECTRUM_SIZE = 1024
SPECTRUM_STEP = 100

//Color
BACKGROUND_COLOR = black
```

Ce fichier de configuration représente l'ensemble des paramètres d'un Graphe que le Parser est actuellement capable de lire.
Tous les paramétrages possibles ne sont actuellement pas « parsable » pour des raisons de manque de temps associé à cette partie.
Néanmoins si l'ont souhaitait ajouté de nouveaux paramètres que possèdent les Graphes, la classe Config a été codées de façon suffisamment explicite les lui implémenter.

Partie concernant le SpectrumModel détaillé dans la partie (UNKNOW yet)

Diagramme de classe – Config



Idée d'amélioration future :

Une des fonctionnalités que j'aurais aimé implémenter aurait été de pouvoir créer une méthode dans la classe « Config » permettant l 'écriture d'un fichier de configuration à partir des paramètres d'un « Chart » instancié par un utilisateur.

Cela pourrait permettre à cet utilisateur de tester différentes configurations en utilisant les « Setters » présents dans les classes des modèles (Model) et modèles de rendu (RenderingModel) et de générer un fichier de configuration ce paramétrage.

TO – DO :

- ✗ Model composit
- ✗ Renderer
  - Diagramme de classe
  - Explication Rendering en cascade
- ✗ 2.5.5 : Changé « UNKNOW yet » par le paragraphe associé au SpectrumModel
- ✗

3 Cas d'utilisations