

Grapheur

Document d'architecture et de conception

Réalisé par : GEHIER Kylian



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

Table des matières

1 Présentation générale du projet	4
1.1 Une première conception du projet (2012)	4
1.2 Reprise du projet (2018)	4
1.3 Cahier des charges	5
2 Architecture	6
2.1 Modèle de données	7
2.2 MVC	8
2.1 Description des différents composants du MVC	9
2.2 Les composants d'un Chart	11
2.3 ChartView & Modèle Composite	12
2.4 Renderer	15
2.5 Factory	16
2.5.1 Le besoin d'une Factory	16
2.5.2 Création d'un modèle de fichier de configuration	17
2.6 Modèle de donnée	19
3 Use Cases (Cas d'utilisations)	21
4 Idées futures pour le projets	22
Conclusion	23

Index des illustrations

Présentation d'un LineChart.....	6
Présentation d'un MapChart.....	6
Diagramme de classe – Modèle de données.....	7
Diagramme de classe – MVC.....	8
Diagramme de classe – Controller.....	9
Diagramme de communication – Action d'un utilisateur sur la Vue.....	10
Diagramme d'objets – Composition d'un Chart.....	11
Diagramme de classe – ChartView.....	12
ChartView - Constructor.....	13
Diagramme de classe – Model Composit.....	13
LineChartView - Constructor.....	14
MapChartView – Constructor.....	14
Diagramme de classe – Renderer.....	15
Diagramme de Classe – Factory.....	16
Présentation d'un fichier de configuration.....	17
Diagramme de classe – Config.....	18
Diagramme de classe - Entry.....	19
Package – Use Cases.....	21
Illustration d'un BarChart.....	22
Illustration d'un PieChart.....	22

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 a 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'un 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.

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

1.3 Cahier des charges

Cahier des charges imposé :

- Dés-spécialisé 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 :

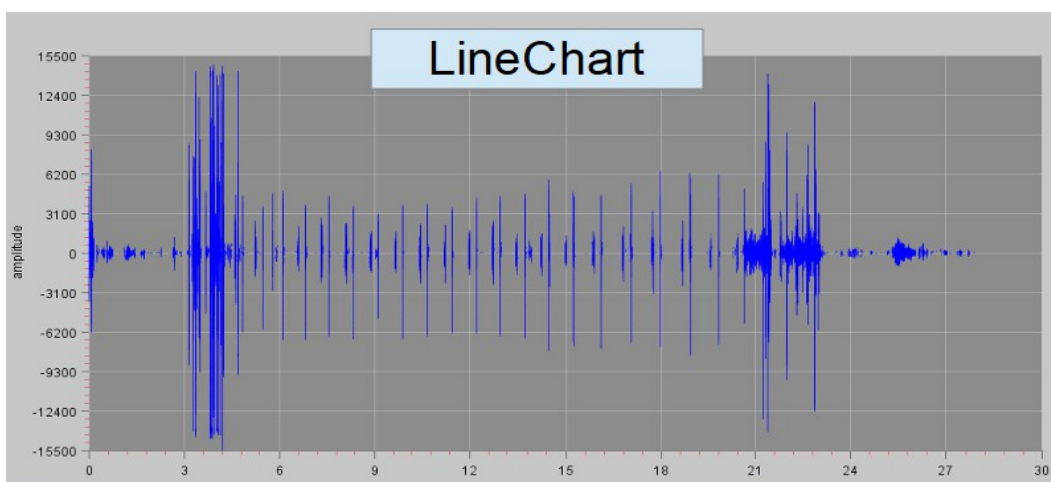
- Implémenter un système de fichier de configuration pour le paramétrage des Charts.
- Création d'un modèle d'abstraction « Entry » regroupant les différents types de données entrantes afin de faciliter l'ajout de données sur un Chart.

2 Architecture

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

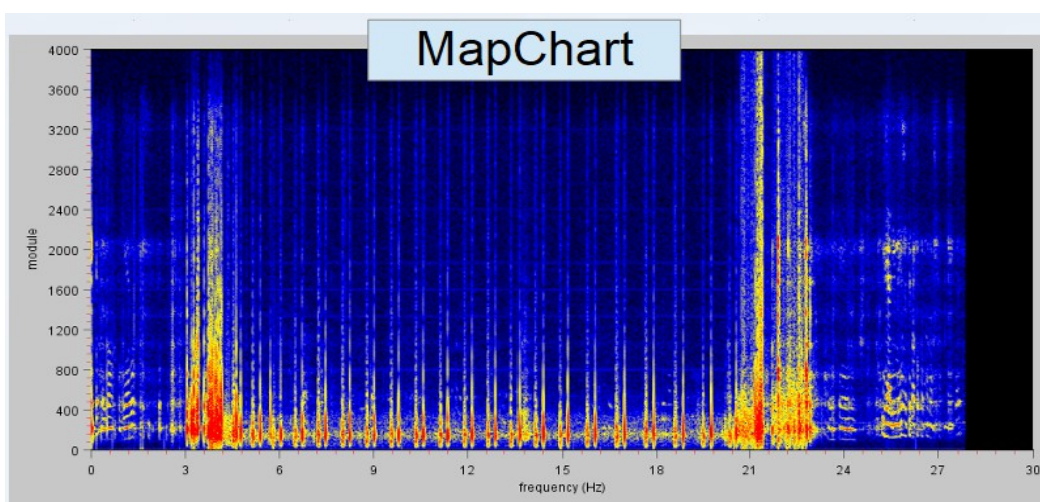
On distingue à l'heure actuelle deux types de Chart que sont les LineCharts et les MapCharts.

- 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

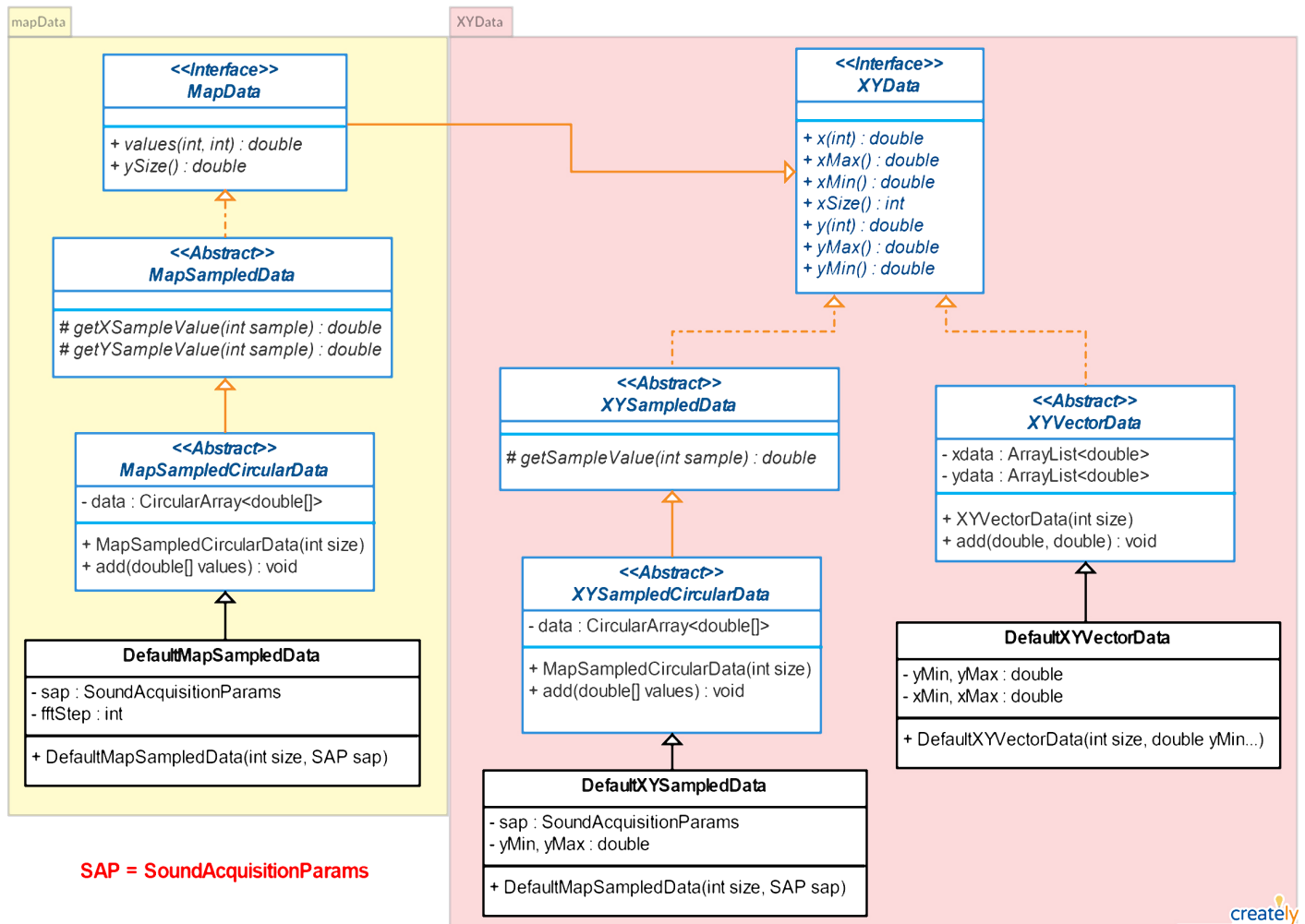


Présentation d'un MapChart

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)

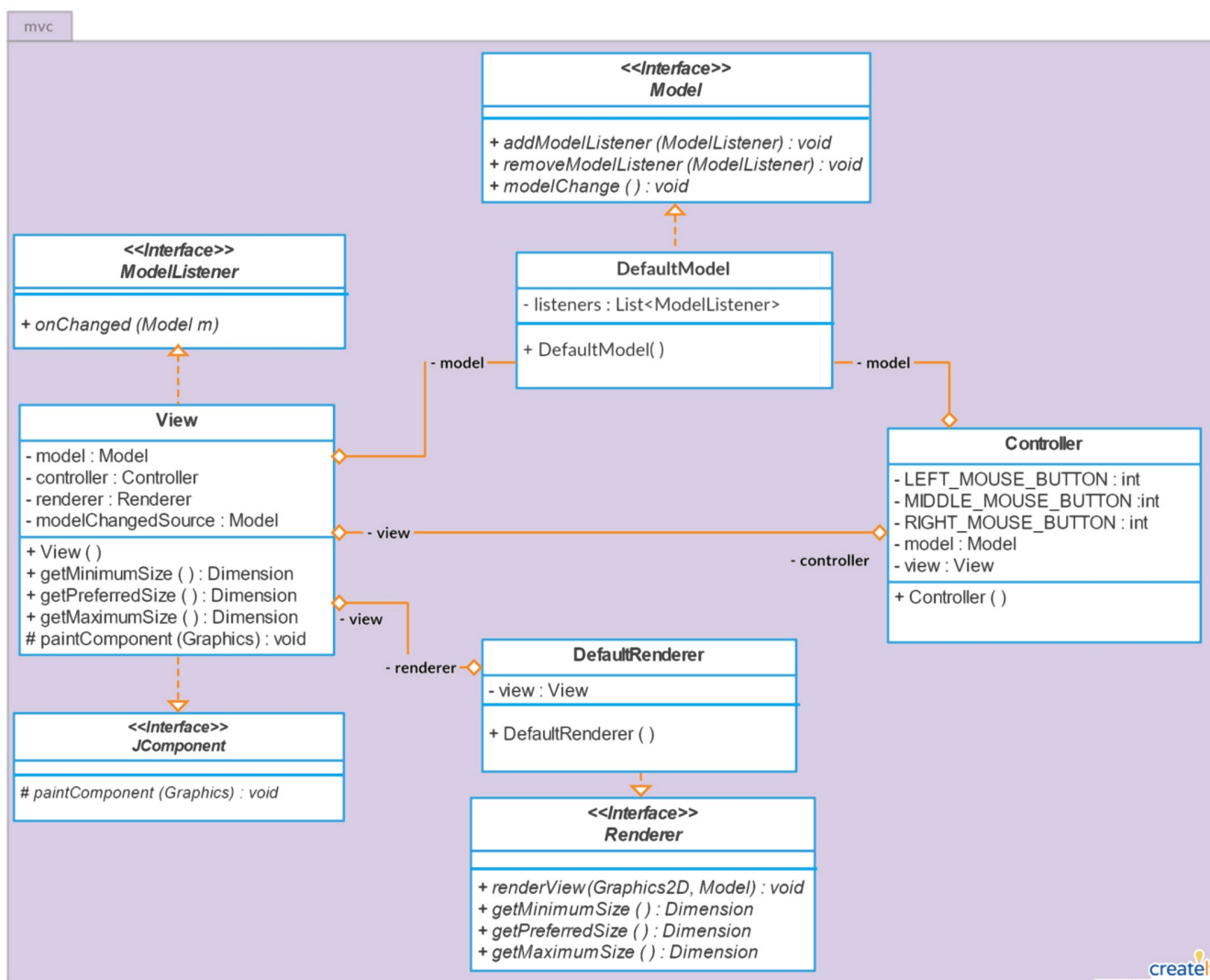
- MapData : modèle de données utilisé par les MapChart
 - MapSampledData : utilisé pour des données échantillonnées

La classe XYVectorData est une classe que j'ai implémenté pour les données venant de CSV.

2.2 MVC

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

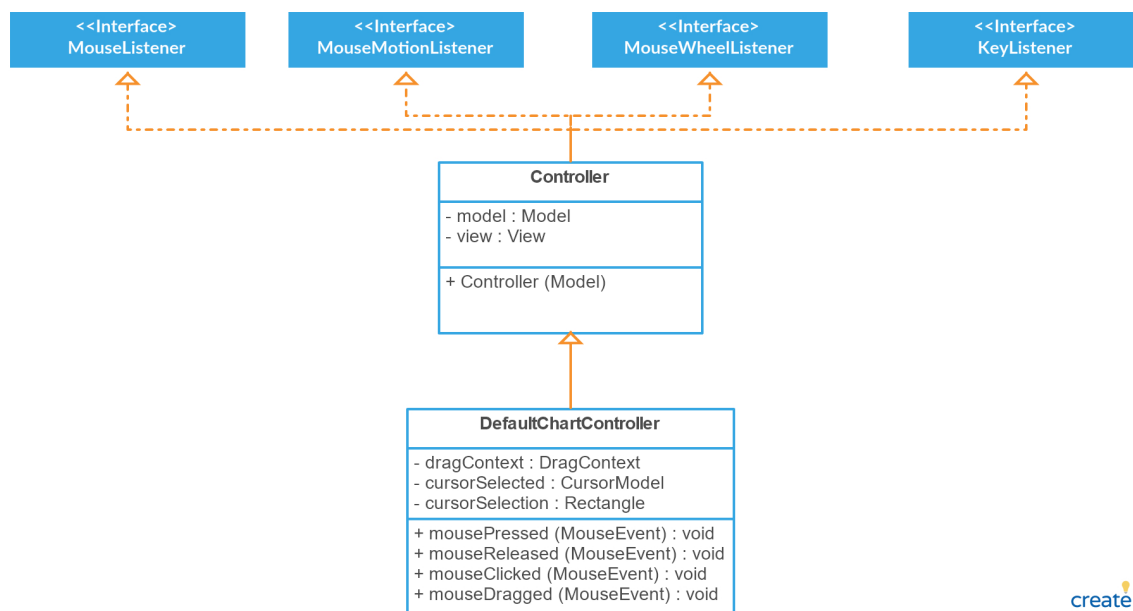
Diagramme de classe – MVC



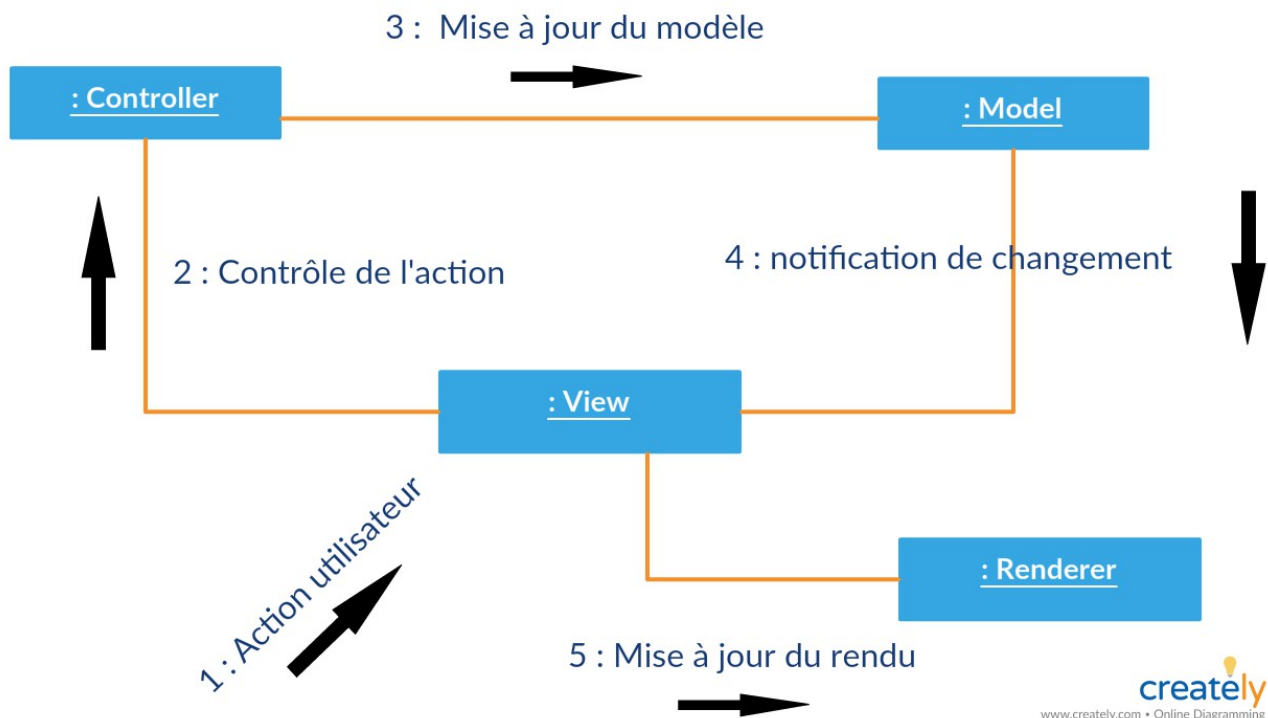
2.1 Description des différents composants du MVC

- **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(s) modèle(s) concerné(s) qui va(vont) à leur tour notifier la Vue qu'il(s) a/ont été modifié(s) via la méthode « `modelChange` » commune à tous les modèles.

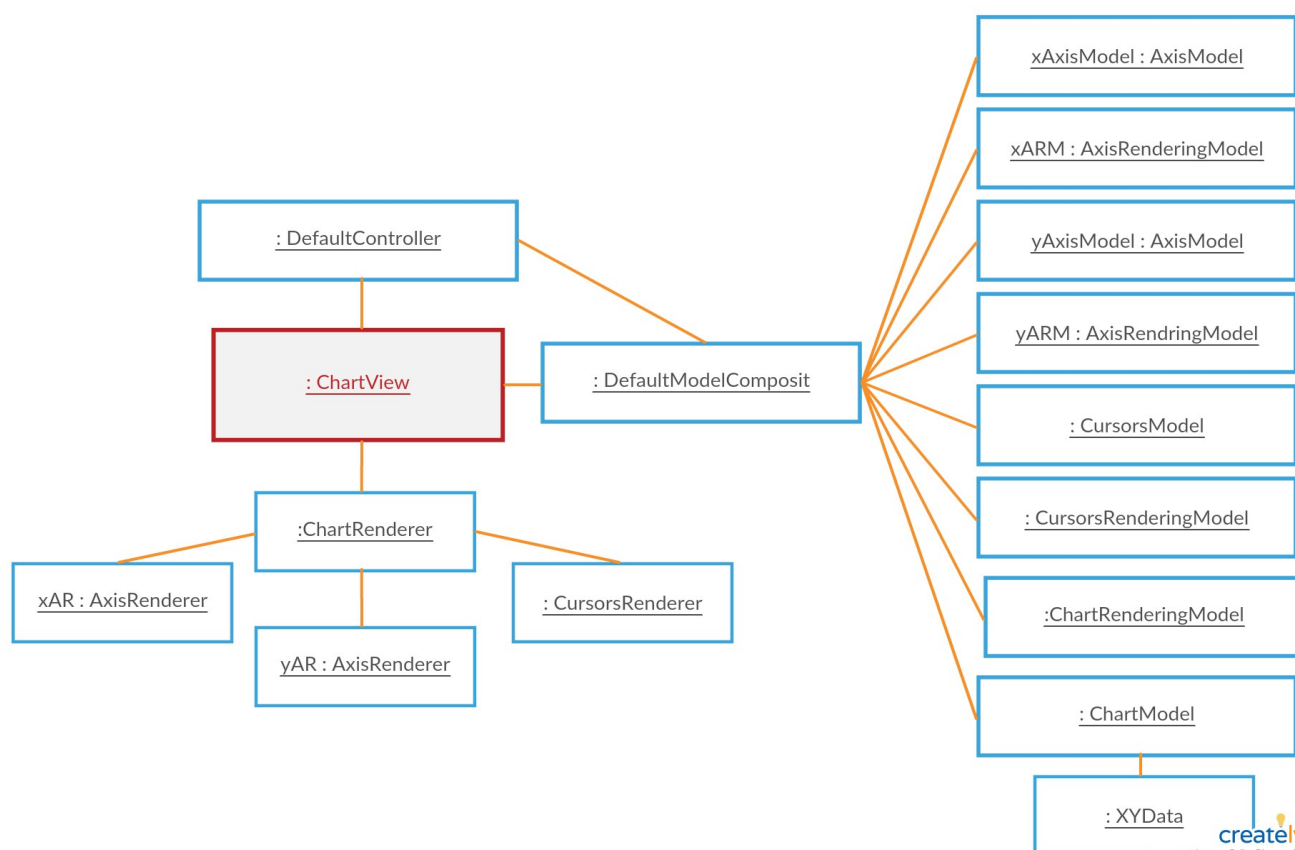
La Vue appelle alors sa méthode « `paintComponent` » ([Cf. Diagramme de classe MVC](#)) qui délègue le dessin de tous les modèles aux différents Renderer.

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 montre le diagramme ci-dessus, un ChartView est donc composé de deux modèles d'axes (x et y), d'un 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.

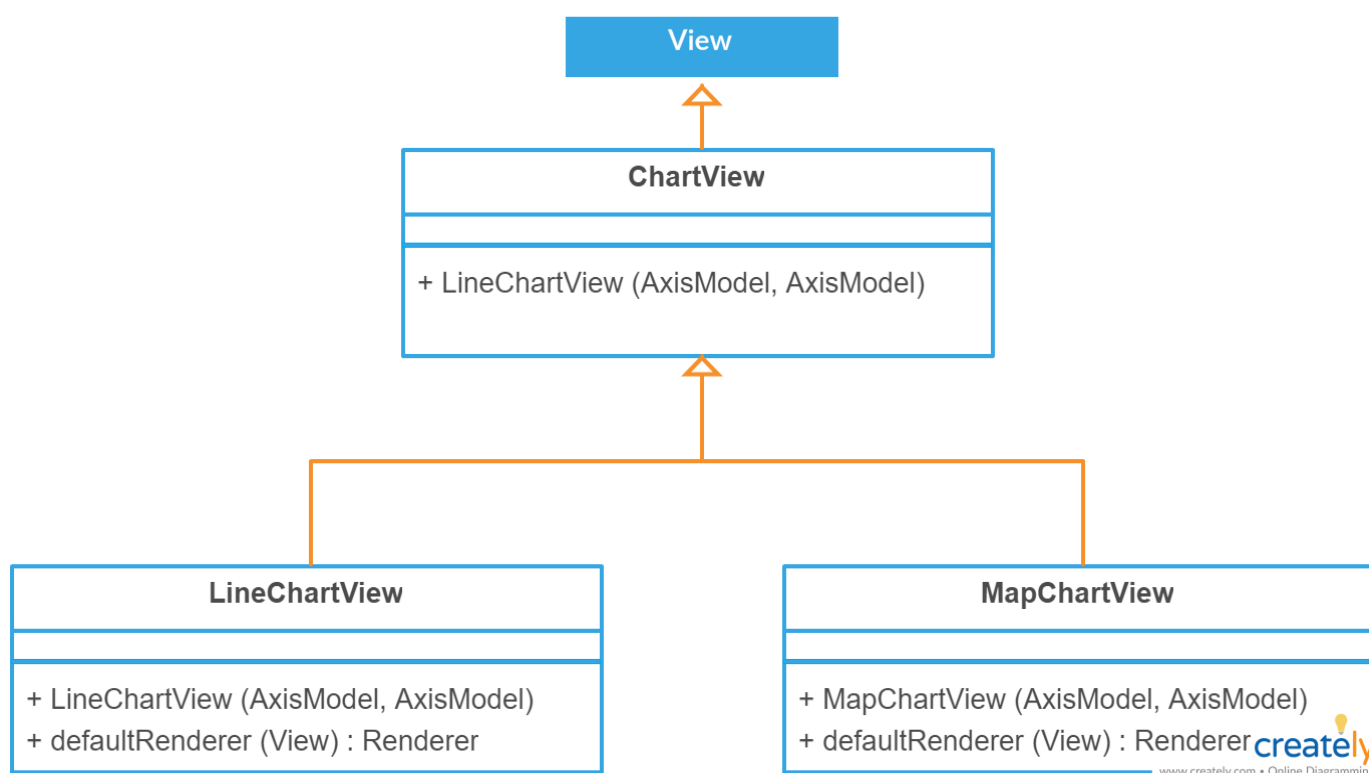
Chaque modèle se voit également attribué un Renderer qui sera chargé de le dessiner en récupérant les informations contenu dans ses Model et RenderingModel associés.

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

Nous pouvons également remarquer 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 », elles 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

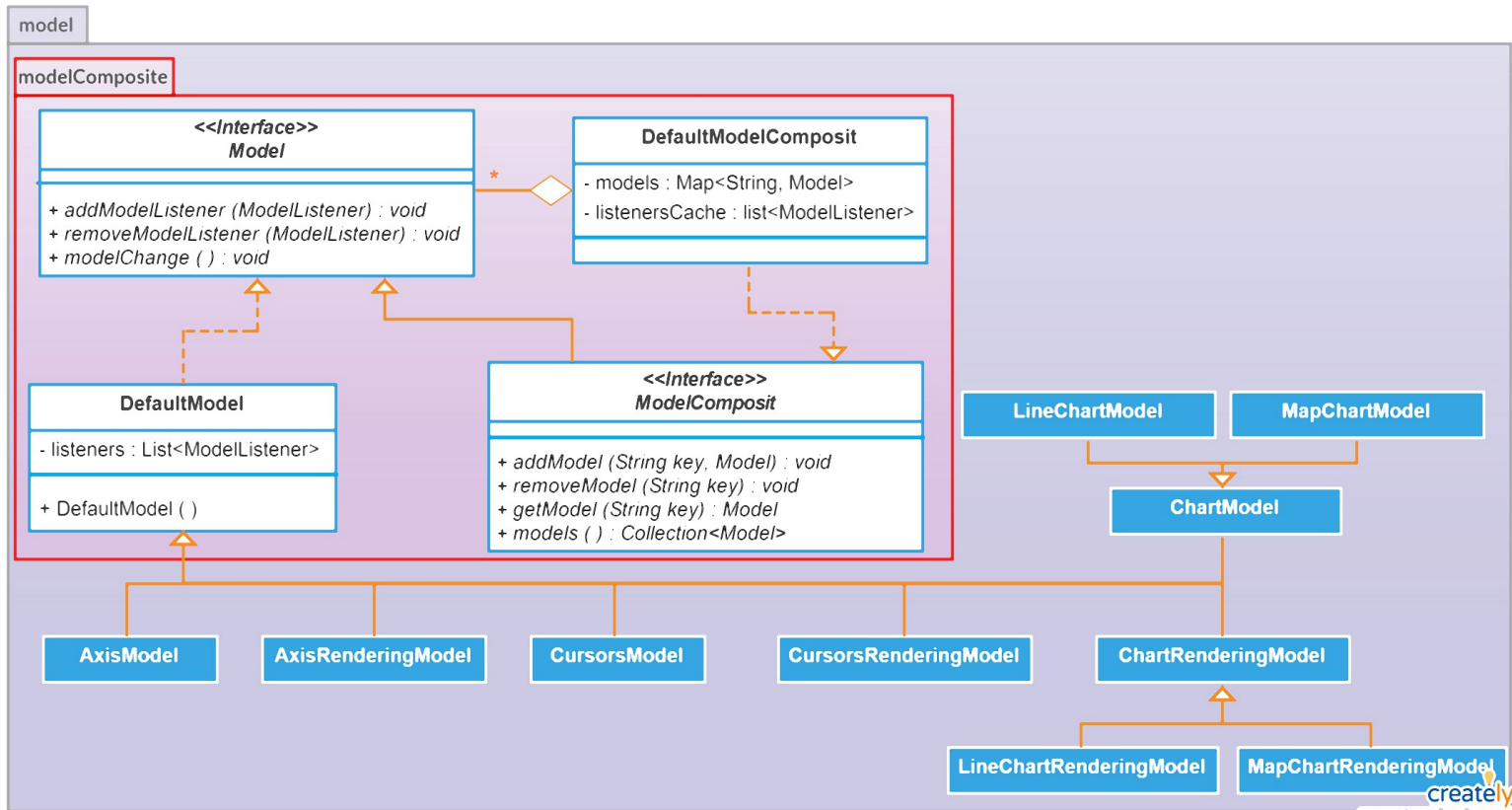
```
public ChartView(AxisModel xAxis, AxisModel yAxis)
{
    super();

    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 des classes héritant de « ChartModel ».

Lors de l'instanciation d'un « LineChartView », les modèles suivants 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());
}
```

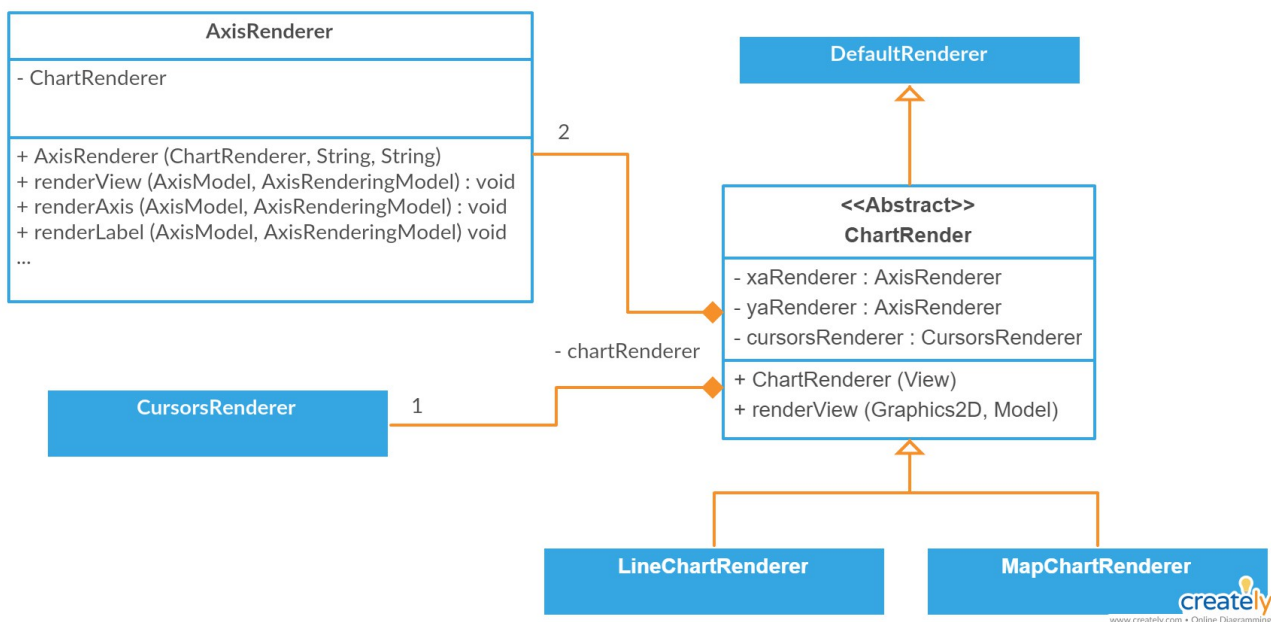
2.4 Renderer

Dans cette partie, je souhaiterais simplement clarifier le fonctionnement des Renderer.

Précédemment, nous avons parlé de couple Model/RenderingModel. A chaque'un de ces couples existe un Renderer associé (Ex : AxisModel/AxisRenderingModel → AxisRenderer) dont le constructeur prend en paramètres le Model et le RenderingModel associé permettant l'accès aux informations de rendus des différents composants du Chart.

La méthode « paintComponent » de la classe « View » est appelée à chaque changement du Model car celui-ci doit être ressiné. Lorsqu'elle est appelée, elle fait appel à la méthode « renderVlew » de la classe « ChartRenderer » du diagramme ci-dessous.

Diagramme de classe – Renderer



La méthode « renderView » de la classe **ChartRenderer** va donc appeler les méthodes « renderView » de chacun des **Renderer** qu'elle a instanciée dans son constructeur (2 **AxisRenderer** et 1 **CursorsRenderer**).

La méthode « renderView » des **Renderer** instanciés (Ex : celle de **AxisRenderer**) va appeler une cascade de méthode **renderXX** (Ex : **renderAxis**) afin de dessiner les éléments de chaque modèle composant le Chart.

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>

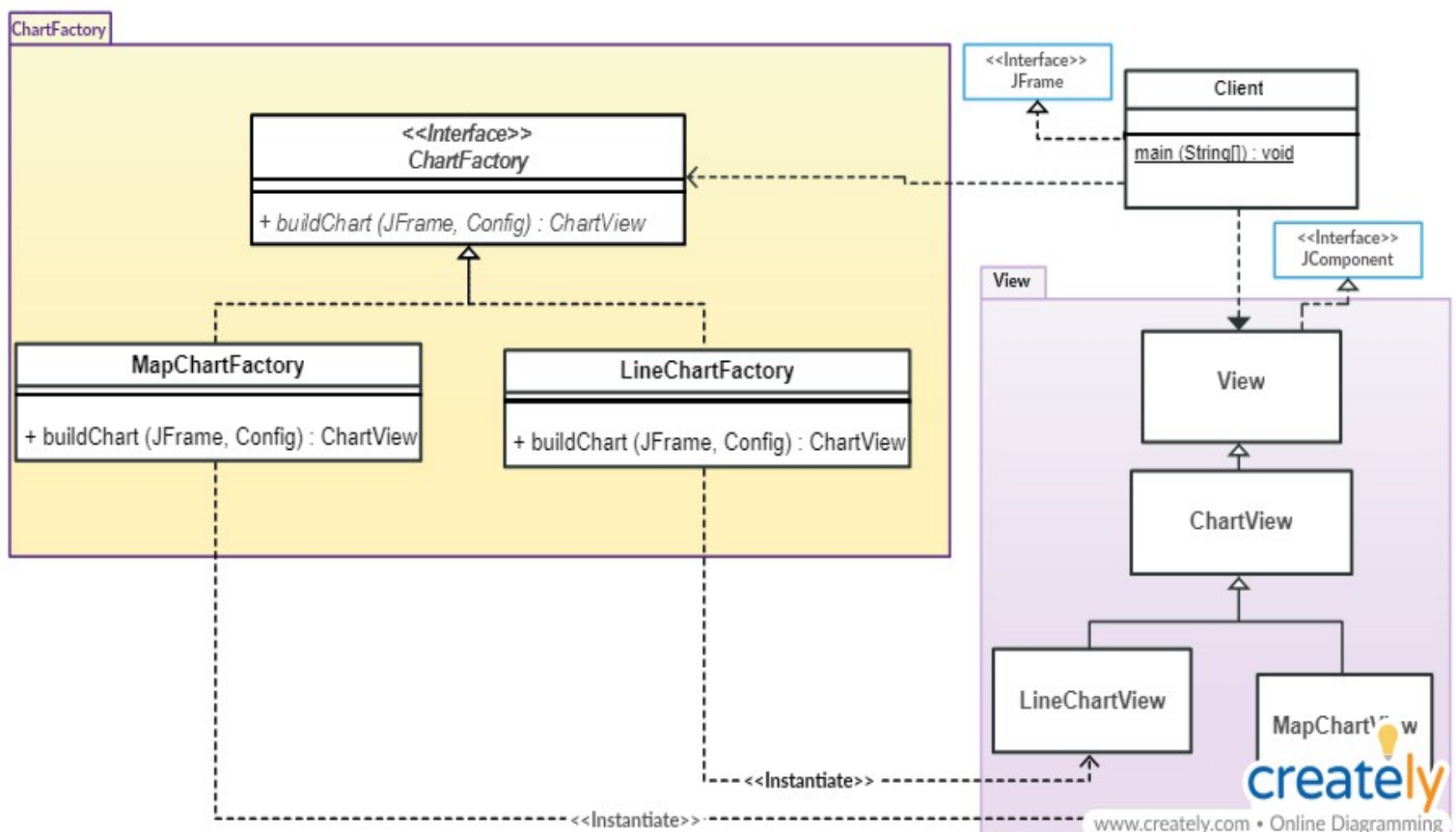
Nous pouvons à présent afficher des Graphes sous forme de « LineChart » ou de « MapChart ». Bien qu'ils soient vides de données pour le moment, un premier problème se pose déjà.

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

A la vue de la quantité 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 amélioré par la suite) et possédant son propre « Parseur ».

Présentation d'un fichier de configuration

```
3
4 // Boolean values
5
6 X_AXIS_MINORTICK_VISIBLE = true
7 Y_AXIS_MINORTICK_VISIBLE = true
8 HORIZONTAL_GRIDLINE_VISIBLE = true
9 VERTICAL_GRIDLINE_VISIBLE = true
10
11 // Choose between : Bottom, Left, Right and Top
12
13 MARGIN_SIDE = Left
14
15 // Double values
16
17 MARGIN_VALUE = 80
18
19 // Integer values
20 CHART_WIDTH = 1000
21 CHART_HEIGHT = 500
22
23 // String values
24 X_AXIS_LABEL = xLabel
25 Y_AXIS_LABEL = yLabel
26
27 // Color values (red, black, magenta,...)
28
29 X_AXIS_MINORTICK_COLOR = red
30 Y_AXIS_MINORTICK_COLOR = red
31
32 //Color
33 BACKGROUND_COLOR = black
```

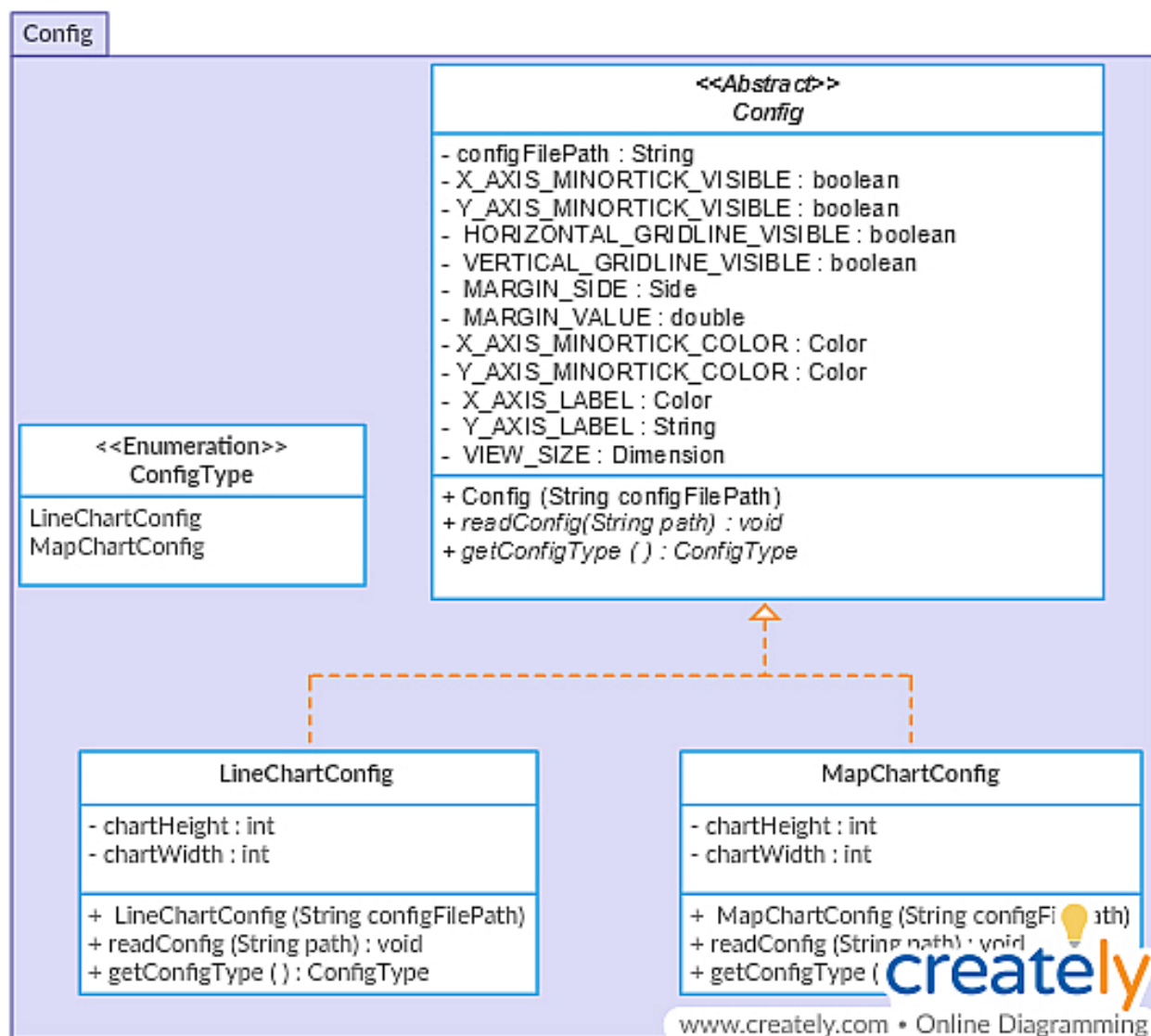
Ce fichier de configuration représente l'ensemble des paramètres d'un Graphe que le Parseur est actuellement capable de lire.

Tous les paramétrages possibles ne sont actuellement pas « parsable » pour des raisons de manque de temps associé à cette partie.

Néanmoins si l'on souhaitait ajouté de nouveaux paramètres que possèdent les Graphes, la classe Config a été codées de façon suffisamment explicite les lui implémenter.

(Cf . 5.Améliorations)

Diagramme de classe – Config



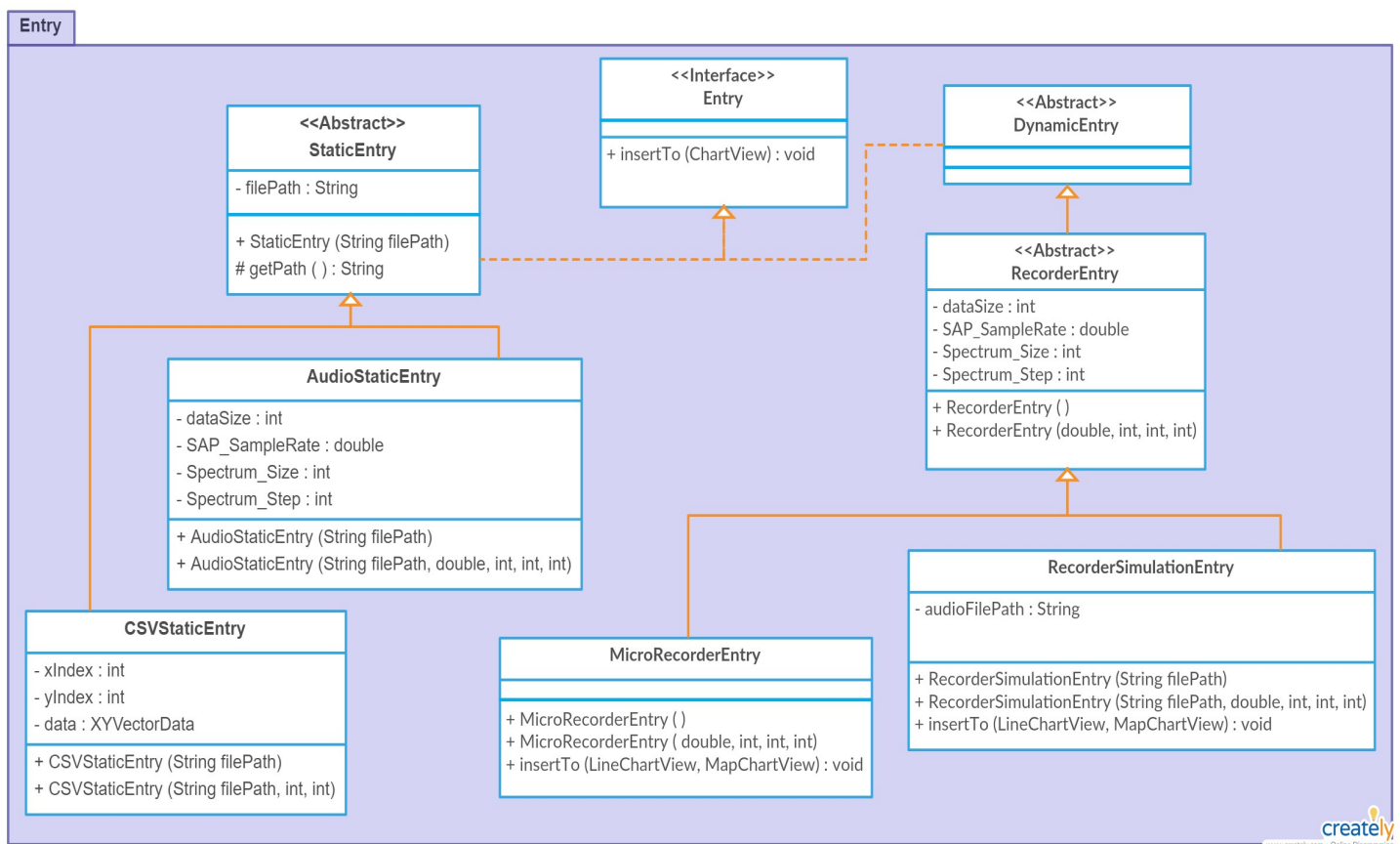
La classe « Config » et ses abstractions contiennent le Parser nécessaire à la lecture des fichiers de configurations. Le code de ce Parser est identique pour la classe « LineChartConfig » et « MapChartConfig » (On pourrait donc les regrouper en une pour les cas de paramètres que j'ai traité) néanmoins j'ai préféré les dissocier dans le cas où de futurs changements sur ces deux types de Chart viendraient à rendre nécessaire cette séparation.

2.6 Modèle de donnée

Nous avons vu en tout début de document (Cf . [2.1 Modèle de données](#)) que les Chart possédaient leur propre modèle de données. Lorsque nous voulons ajouter des données entrantes qualifiées « d'Entry » dans ce document, il est donc nécessaire de les convertir en données utilisables par le Chart : « XYData ».

Voici donc le modèle de données Entry que j'ai conçu afin de centraliser tous les types de données entrantes que l'ont souhaiterait afficher dans un Chart.

Diagramme de classe - Entry



Les Entry sont déclinées en deux abstractions :

- **StaticEntry** : Données entrantes que l'on souhaite afficher statiquement sur les Charts. Ces données sont stockées dans un fichier (Csv, audio, ...) dont le chemin d'accès est pris en paramètre du constructeur de la classe abstraite StaticEntry.
- **DynamicEntry** : Données entrantes que l'on souhaite afficher dynamiquement

Concernant les StaticEntry, il existe pour le moment deux classes concrètes :

- CSVStaticEntry : Données provenant d'un fichier CSV. Si plus de deux colonnes sont présentes dans le fichier CSV et que le constructeur à trois paramètres est appelé (String filePath, int xIndex, int yIndex) la colonne xIndex (respectivement yIndex) sera lue et assignée à l'AxisModel de l'axe des x (respectivement y). Dans le cas contraire, les valeurs de xIndex et yIndex seront celles des deux premières colonnes.
- AudioStaticEntry : Données provenant d'un fichier Audio. Les paramètres du deuxième constructeur servent à initialiser le SpectrumModel du MapChart si un MapChart est passé en paramètre de la méthode insertTo(ChartView). Un RecorderSimulator (créé par M.Perronne pour Infral à l'époque) prenant en paramètre le chemin d'accès du fichier Audio qui lira les données afin de les afficher statiquement sur le Chart.

Pour les DynamicEntry, je n'ai utilisé que des données Audio enregistrées depuis un micro (via un MicroRecorder implémenté par M.Perronne) ou simulées à l'aide du RecorderSimulator précédemment décrits. La seule abstraction que j'ai écrite est celle des RecorderEntry se divisant en :

- MicroRecorderEntry : pour les Acquisitions en temps réelle
- RecorderSimulationEntry : pour les Acquisitions simulées

Un point important concernant toutes les Entry dynamiques actuelles est qu'on ne peut utiliser les méthodes d'insertion de données « insertTo(ChartView) » propre à chaque classe des DynamicEntry qu'une seule fois par DynamicEntry instanciée sans quoi une erreur est relevée.

Cela provient d'un problème de pluralité de Recorder instancié lors de l'appel répété de cette méthode. Problème que je n'ai pas encore réussi à corriger par manque de temps. C'est pourquoi, il existe deux possibilités afin d'afficher les mêmes données sur deux Chart différents :

- Instancier deux DynamicEntry E1 et E2 et faire appel à insertTo(chartView) de E1 sur le premier Chart et insertTo(chartView) de E2 sur le deuxième Chart. Cela est montré dans les un des nombreux « use cases » (Cas d'utilisations) présentés la page suivante : notamment dans le use case « LineMapChartAudioSimulatedDynamicData ».
- Utiliser la méthode insertTo(LineChartView, MapChartView) permettant d'afficher les mêmes données sur un LineChart et un MapChart

3 Use Cases (Cas d'utilisations)

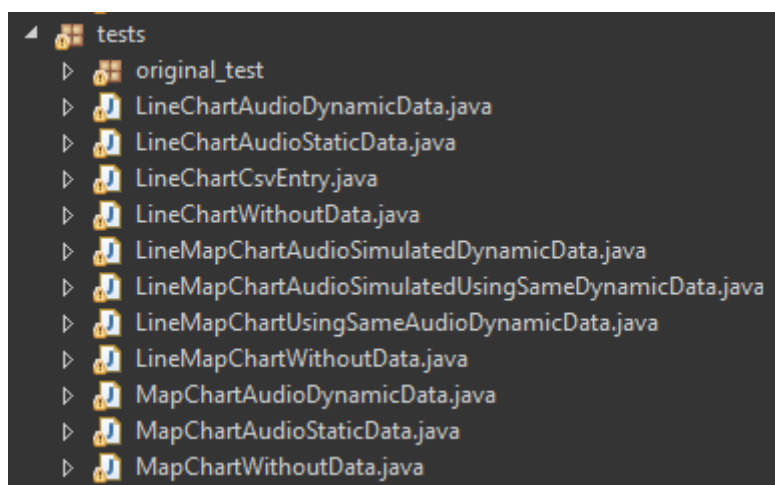
Afin de faciliter la compréhension sur le fonctionnement de l'API, 11 use cases ont été écrits dans le package « tests ». Les précédents tests écrits par M.Perronne à l'époque et qui me semblaient importants de garder sont également conservés dans un sous-package « original_test » de ce package.

Voici un listing rapide de quelques cas d'utilisations :

1. LineChartWithoutData : instantiation d'un LineChart vide de données
2. MapChartWithoutData : instantiation d'un MapChart vide de données
3. LineMapChartWithoutData : instantiation d'un LineChart et d'un MapChart vides de données. L'intérêt de ce test est de montrer comment superposer des Chart en utilisant un « BoxLayout »
4. LineChartCsvEntry : instantiation d'un LineChart dans lequel sont insérées les données d'un fichier CSV (ATTENTION : les chemins d'accès des fichiers utilisés pour les tests sont contenus dans une classe Constants et sont écrits en Absolu, ce qui implique de les changer par votre chemin absolu)

Je n'explicitai pas les 11 use cases présents dans le package de test, les noms donnés aux tests sont suffisamment clairs. Néanmoins voici une capture du package :

Package – Use Cases



4 Idées futures pour le projets

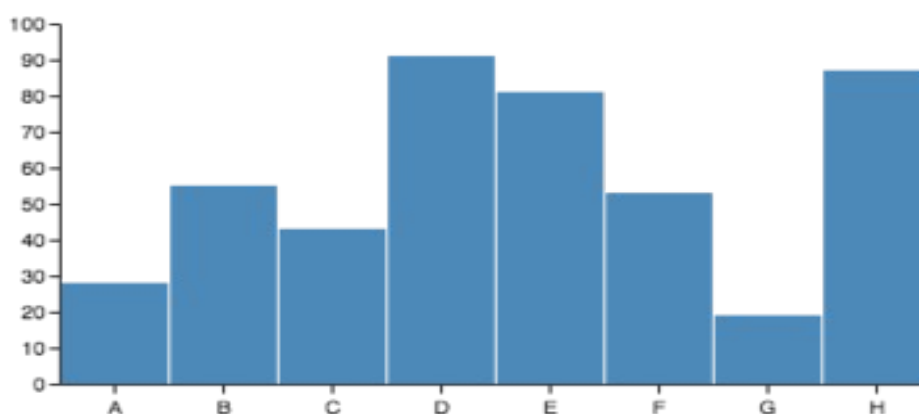
Je n'ai malheureusement pas pu implémenter toutes les idées que j'ai eu pendant la durée de ce projet, notamment à cause du temps passé à comprendre le projet et celui consacré à rédiger ce document de conception ainsi que ses diagrammes.

Je me permet donc d'exposer mes idées qui pourront être utiles aux étudiants qui le reprendront.

1) Je pense notamment à la création de deux nouveaux types de Chart

- Le BarChart (Diagramme en baton) : ce Chart ne devrait pas poser trop de difficultés en terme d'implémentation.

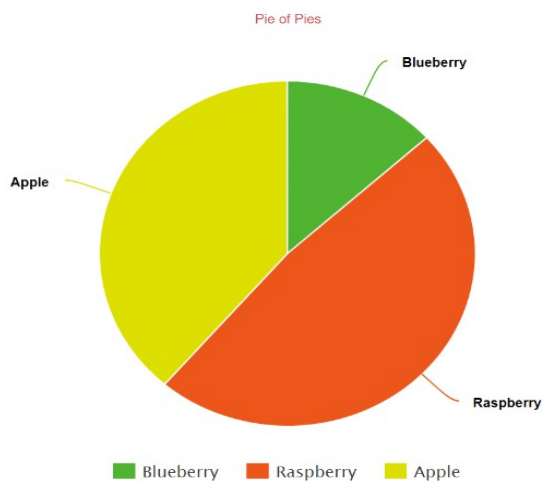
Illustration d'un BarChart



1.

- le PieChart (Diagramme en Camembert) : celui ci devrait être plus compliqué à implémenter car il demande de passer en coordonnées polaires et donc potentiellement implémenter de nouveaux « Model ».

Illustration d'un PieChart



- 2) Une deuxième idée serait de remplacer le système de fichier de configuration (C'est également une des raisons qui fait qu'il n'est pas complet à 100%) par un l'ajout d'un volet de paramètre attaché à chaque instanciation de Chart. Ce volet de paramètre permettrait donc à l'utilisateur de modifier les paramètres du Chart Avant/Pendant l'affichage du Chart selon comment il sera codé.
- 3) Ajout dans les StaticEntry de la possibilité de lire du JSON ou XML éventuellement à condition que les données soient bien de type numérique.

Conclusion

Afin de conclure ce document de conception , je voudrais remercier les lecteurs pour m'avoir lu !

Je tiens également à remercié M. Perronne de m'avoir confié ce projet car il m'a permis de faire face à quelques lacunes accumulées lors de mes premières années à l'ENSISA et ainsi de les surmonter.

Je n'ai pas de source spécifiques à citer étant donné que mes recherches se sont principalement effectuées sur StackOverFlow.

Je laisse mes coordonnées Mail pour les élèves qui reprendront ce projet afin de pouvoir répondre à d'éventuelles questions : kylian.gehier@gmail.com