

Conception du Projet : Accélérateur de Particules

Adrien Dürst, Joséphine Potdevin
adrien.durst@epfl.ch, josephine.potdevin@epfl.ch

28 mai 2019

Table des matières

0.1	Introduction	2
1	Graphique de la hiérarchie de classes	2
1.1	Graphique	2
1.2	Commentaire	3
2	La classe Vecteur3D	3
2.1	Introduction	3
2.2	Réorganisation du code	3
2.3	Utilisation	4
3	La super-classe Element et ses sous-classes	4
3.1	Introduction	4
3.2	Organisation de la hiérarchie de classe	4
3.3	Utilisation	5
4	Les faisceaux de particules	6
4.1	La classe Particule	6
4.2	Organisation de la classe Faisceau	6
4.3	La sous-classe Faisceau Circulaire	7
5	La classe Accélérateur et son dessin	7
5.1	La classe Accélérateur	7
5.2	Le Dessin d'un accélérateur	8
6	Conclusion	8

0.1 Introduction

Ce document "Conception" permet d'informer le lecteur de ce projet nos choix de conception, nos détails d'implémentation ainsi que les défis auxquels nous nous sommes confrontés au cours de ce projet.

1 Graphique de la hiérarchie de classes

1.1 Graphique

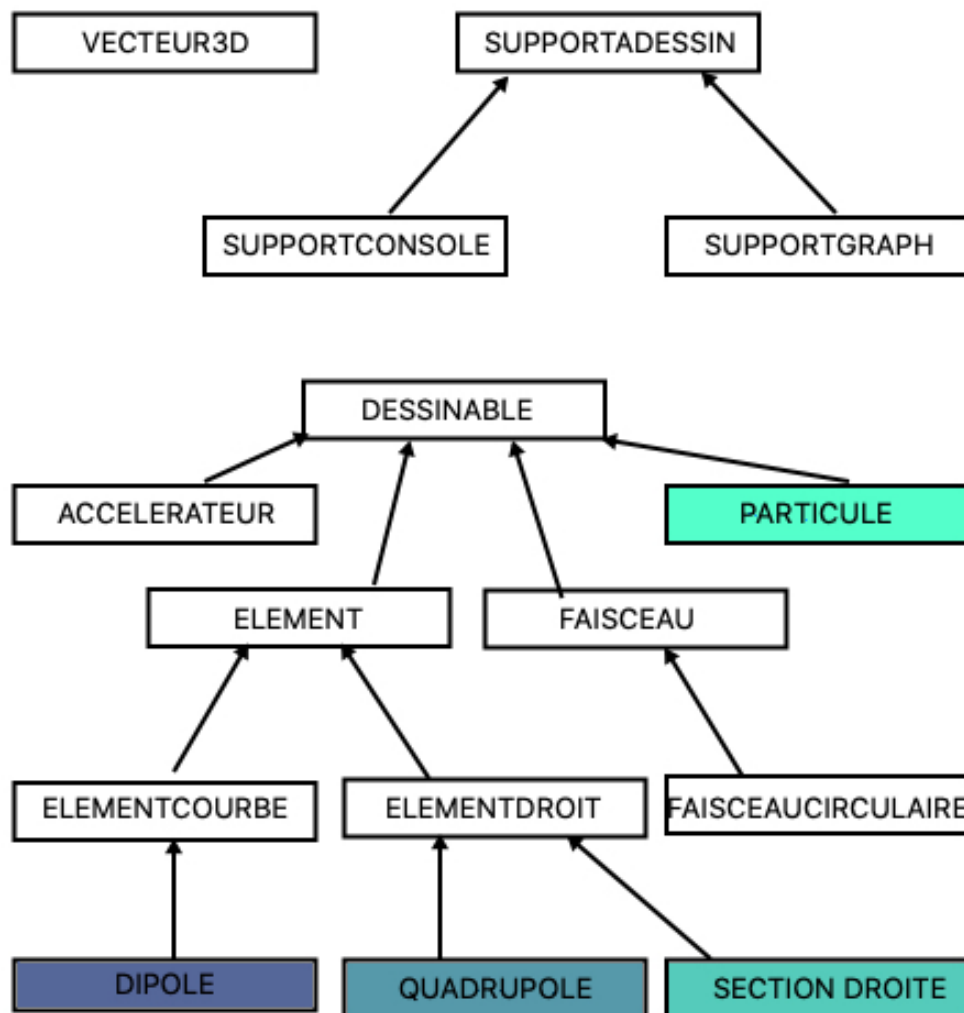


FIGURE 1 – *Hiérarchie des classes*

1.2 Commentaire

Ce graphique fait part de la hiérarchie des classes du projet à la fin de celui-ci. Chaque flèche indique les relations d'héritages d'une classe à l'autre, et donc les attributs et méthodes partagées par ces classes. Ce choix d'implémentation des classes nous paraît le plus logique et le plus facile quand aux instructions données par Mr Chappelier. Les relations de dépendances entre ces classes, les méthodes particulières ou les conceptions personnelles de l'implémentation de ces classes seront décrites dans les sections suivantes.

De plus, les couleurs des cases particules, dipôle, quadrupôle, et section droite sont les mêmes couleurs que celle de notre accélérateur (graphisme).

2 La classe Vecteur3D

2.1 Introduction

La classe Vecteur3D fut la première que nous ayons eu à implémenter. Avec le peu de connaissance sur l'orienté-objet que nous avions à ce stade, il fut difficile au début de mettre en place un code optimal, qui utilise les structures de méthodes les plus cohérentes ainsi que d'utiliser à bon escient le meilleur que puisse nous apporter le `c++11`.

2.2 Réorganisation du code

Ainsi, cette classe est une classe sur lequel nous avons remanié beaucoup ligne de code. Parmi ces remaniements :

- Division de la classe en deux fichiers (Vecteur3D.cc, Vecteur3D.h), au lieu d'un seul (Vecteur3D.cc).
- Remplacement d'un setter en constructeur
- Surcharge d'opérateurs internes et externes au lieu de méthodes pour les opérations mathématiques entre vecteurs ou sur l'instance courante

L'unique questionnement sur l'implémentation de cette classe fut à propos des corps des opérateurs internes et externes. Puisque dans la première conception des opérations mathématiques, booléennes et d'affichage sur les vecteurs (addition, soustraction, affichage, comparaison etc) nous avons implémenté celles-ci comme des méthodes aux noms de l'opération, prenant un argument pour la plupart et retournant la résultante de l'opération voulue, lors des semaines d'après, avec la connaissance des surcharges d'opérateurs, nous avons donc échangé l'utilisation de ces méthodes peu convenable à l'utilisation de cette classe (interface) par des surcharges internes et externes plus instinctives. Dès lors se pose la question du corps de ces surcharges. Fallait-il appeler les méthodes auparavant implémenter ou fallait-il reconstituer un corps identique à la méthode correspondant à l'opération demandée ?

Nous avons ainsi décidé que l'appel d'une méthode à l'intérieur d'une surcharge d'opé-

rateur n'étais pas une bonne optimisation du code mais qu'il était préférable que les corps des surcharges fassent elle-mêmes les opérations. Ainsi, les surcharges `+`; `-`; `*`(multiplication par un scalaire); produit vectoriel (et produit vectoriel=); `+=`; `-=`; `*=`(multiplication par un scalaire); `-`(opposé de l'instance) utilisent les corps des méthodes correspondantes et ces méthodes sont enlevées de la classe `Vecteur3D` puisqu'elles n'ont plus aucune utilité.

Cependant, nous avons choisi de garder les méthodes `affiche` (utiliser dans la surcharge `«`), la méthode de comparaison (appeler dans les deux surcharges `==` et `!=`), la méthode produit scalaire (appeler dans la surcharge externe `*`), ainsi que les méthodes `normes`, `normes au carré` et `rotation` utilisée dans des autres surcharges de la classe (par exemple `unitaire` qui retourne le vecteur unitaire). Nous avons gardé ces méthodes soit parce qu'elles étaient utilisées plusieurs fois (`compare`), soit parce qu'elle faisait sens dans leur existence d'après nous (`affiche`), soit parce qu'elles n'avaient pas d'opérateurs correspondant (exemple : `norme`), soit parce qu'une surcharge externe faisait son opération associée et qu'il était donc mieux selon nous d'utiliser une méthode de la classe plutôt que de redéfinir le corps de cette méthode à l'aide des getteurs, faisant donc appel à beaucoup de méthodes pour un seul calcul (produit scalaire).

2.3 Utilisation

Cette classe constitue la base sur lequel notre projet peut se fonder. Elle permet non seulement de créer un environnement pour l'existence de notre accélérateur (un espace en trois dimensions), mais aussi d'implémenter presque chacune de nos classes. En effet, les particules (et par extension les faisceaux), les éléments, l'accélérateur ont besoin de `Vecteur3D` pour représenter des positions, des vitesses, des forces, ou encore calculer des angles de dérivation des changements de positions ou de vitesse etc... En bref, la classe `Vecteur3D` est une classe indépendante dont les autres dépendent. Elle structure notre projet, et là dessus, il peut commencer à se développer.

3 La super-classe `Element` et ses sous-classes

3.1 Introduction

La hiérarchie et la structure de la classe `Element` et de ses sous-classes a pris plusieurs semaines à s'implémenter. Petit à petit nous avons construit et remodulé ces classes afin d'obtenir les trois sous-classes instanciables `dipôle`, `section droite` et `quadrupôle` dans lesquelles des particules peuvent être injectées et y subiront des forces.

3.2 Organisation de la hiérarchie de classe

Nous avons implémenté dans les classes `Element`, `Element Droit`, `Element Courbe`, `Dipôle`, `Section Droite`, `Quadrupôle`, les méthodes et attributs nécessaires à la construction de l'accélérateur. Ainsi, comme indiqué sur les énoncés de chaque semaine, nous avons créé ces classes, donné des attributs communs à tous ou spécifiques à chacun, prototypé

des méthodes et redéfinis d'autres.

Tout d'abord, pour ce qui est des attributs, nous avons fait explicitement les attributs demandés dans les énoncés du projet. C'est à dire chaque élément possède une position d'entrée, une position de sortie, le rayon de la chambre à vide et un pointeur sur l'élément suivant. Chaque élément courbe possède en plus un rayon et un centre de courbure et chaque dipôle (sous-classe d'élément courbe) a la norme de son champ magnétique. Enfin, dans la branche des élément droit, seul le quadrupole possède un autre attribut : intensité de son champs magnétique.

Enfin, grâce au polymorphisme, nous pouvons créer des méthodes virtuelles pures redéfinies dans toutes les sous-classe instanciable afin d'adapter la façon dont se comporte un pointeur sur un élément en fonction de l'adresse qu'il pointe. Ainsi, la super classe élément comporte deux sortes de méthodes :

(1) Les méthodes utilisable par tous et ne nécessitant aucune redéfinition :

- Une qui permet de savoir si une particule qui était dans l'élément courant est passé au suivant

- Une qui attache un élément après l'élément courant (construction de l'accélérateur)

- Et enfin une qui calcule la distance entre l'élément et une particule

(2) Les méthodes virtuelle pures redéfinies dans les sous-classes :

- Une méthode qui dit si oui ou non une particule a heurté les bords d'un élément. Cette méthode est redéfinie dans les sous-classes Element Courbe et Element Droit puisqu'elle peut être définie seulement avec la géométrie de l'élément en question.

- La méthode d'affichage redéfini dans les derniers maillons de la chaîne d'héritage

- Et enfin la méthode qui calcule le champ magnétique d'un élément redéfini encore une fois dans les dipôles, quadrupôles et sections droites (il ne fait rien ici)

De plus nous avons fait une maille fodo pour simplifier la construction de l'accélérateur (extension).

3.3 Utilisation

Rappelons que notre projet vise à simuler un accélérateur de particule, c'est à dire une suite circulaire d'élément dans lesquels des particules sous forme de faisceaux se déplacent et subissent des forces afin d'augmenter leurs vitesse.

Ainsi le groupe que forme les trois sous-sous-classe dipôle (héritant de élément courbe), section droite et quadrupôle (héritant de élément droit) sont la structure même de notre accélérateur. Chaque choix de conception ou de modularisation des classes Element et sous-classes ont été dans le but de visualiser et de construire au mieux un enchainement logique et fonctionnel de ces éléments les uns après les autre dans lequel des faisceau peuvent se déplacer et être accélérer.

4 Les faisceaux de particules

4.1 La classe Particule

L'introduction des faisceaux dans notre projet a permis de tester l'efficacité de notre implémentation de la classe Particule. En effet, comme un faisceau est une collection de particules, nous les manipulons comme tel et avons besoin d'une classe particule maniable, fonctionnelle et logique. A la création de cette classe nous était offert deux choix : Implémentation avec des unités du système international (SI) ou bien en GeV (électron-Volt). Etant encore tôt dans le projet, nous n'avons pas réussi à nous décider face à ses deux options, quelle était la meilleure ? Pour être sûr de pouvoir revenir sur nos choix en cas de problèmes, nous avons donc implémenté une méthode privée qui transforme une masse donnée en GeV/c^2 en kg afin de pouvoir revenir sur les unités SI si besoin puisque la construction d'une particule se fait en GeV comme l'indiquent les instructions. De plus deux constructeurs étaient proposés dans l'énoncé. Pour les mêmes raisons que ci-dessus nous avons créé les deux au cas où : un par le vecteur quantité de mouvement en GeV, et l'autre par l'énergie en GeV et avec un vecteur directionnel. De plus, nous avons implémenté l'énergie et le facteur gamma comme une méthodes publiques qui seront donc accessibles par toutes les autres classes du projet (ce qui nous a été fort utile plus tard). Enfin en plus des autres méthodes et attributs proposées dans les énoncés ainsi que les getteurs nécessaires, nous utilisons une méthode privée dans la méthode publique de l'ajout de la force magnétique qui calcule l'angle de déviation que cause la force électromagnétique sur la particule. Enfin, nous avons créé l'attribut et la méthode suivante : un pointeur sur l'élément dans lequel se trouve la particule (à défaut il ne pointe sur rien), ainsi qu'une méthode qui teste si la particule passe à un autre élément et remet donc à jour l'attribut ci-dessus.

4.2 Organisation de la classe Faisceau

Nous avons implémenté la classe Faisceau selon les indications et avec notre interprétation de celles-ci de la manière suivante : un tableau de pointeurs sur des particules, une particule de référence, un nombre de particules, un facteur de macro-particules simulés, une énergie moyenne, une émittance verticale et ses trois coefficients (pas sous forme de tableau), et enfin l'émittance horizontale et ses trois coefficients. Enfin, comme méthode publique particulière nous avons donc implémenté trois méthodes différentes calculant l'énergie, calculant l'émittance verticale et ses coefficients, calculant l'émittance horizontale et ses coefficients et stockant les valeurs calculées dans les attributs correspondant. Cette conception nous paraissait la plus logique et pratique : par exemple pour l'énergie, un faisceau possède une énergie, c'est donc un attribut de la classe Faisceau, mais cette énergie change en tout temps, il faut donc la recalculer à chaque fois que l'on fait évoluer le faisceau.

De plus, nous avons fait certains ajustements dans d'autres classes pour implémenter au mieux un faisceau. Afin d'injecter dans le faisceau des particules de même vitesse et position que la particule de référence, mais de masse et charge λ fois plus grande (λ étant le coefficient des macros-particules simulées) nous avons créé

une surcharge d'opérateur $\ast=$ dans la classe particule qui, prenant un double (lambda) comme argument, change la masse et la charge de cette particule en les multipliant par le double. De plus, dans le calcul de l'émittance horizontale d'une particule dans un élément, nous avons besoin d'effectuer à un moment un produit scalaire avec un certain vecteur3D u qui n'a pas les mêmes coordonnées x,y et z en fonction de l'élément dans lequel se trouve la particule (élément droit ou élément courbe). Ainsi, nous avons créé une méthode virtuelle pure u dans élément retournant un vecteur3D et redéfini dans les sous classes élément droit et courbe.

4.3 La sous-classe Faisceau Circulaire

Durant le premier jet de l'exercice P11 dans lequel nous devions implémenter des faisceaux, nous n'avions créé qu'une seule classe indépendante Faisceau (enfin qui n'hérite que de Dessinable). Plus tard dans le projet, nous nous sommes dit qu'il était mieux de créer une sous-classe de Faisceau appelée Faisceau Circulaire. Dans cette sous-classe nous avons redéfini la méthode bouger (qui était en fait propre aux faisceaux circulaires depuis le début mais nous n'avions pas réalisé cela jusque maintenant) qui est donc devenu une méthode virtuelle pure dans Faisceau. En effet, la création de cette sous-classe amène la possibilité dans le futur ou dans une extension du projet, d'implémenter une autre sorte de Faisceau se comportant de manière différente avec peut-être d'autres attributs ou une autre façon de faire évoluer le faisceau dans l'accélérateur. Ainsi, la classe Faisceau devient ininstanciable, alors que les faisceau circulaire sont fins prêts à être injecté dans un accélérateur de particules. Cette classe amenant la dernière pièce du puzzle permet la construction finale de notre accélérateur.

5 La classe Accélérateur et son dessin

5.1 La classe Accélérateur

Nous avons maintenant tout les objets nécessaires à l'implémentation de notre accélérateur de particules : des éléments et des faisceaux de particules aux méthodes et attributs adéquates.

Nous avons implémenté cette classe exactement comme l'indiquait l'énoncé : des tableaux dynamiques contenant des pointeurs sur des éléments et sur des faisceaux, des méthodes qui ajoutent des éléments ou des faisceaux, des méthodes qui suppriment ces collections, et enfin la méthode évolue qui fait marcher notre accélérateur. Nous avons seulement ajouté une méthode qui trouve l'élément dans lequel se trouve une particule. Cette méthode permet d'injecter les faisceaux dans un élément notamment.

Enfin, les dernières modifications que nous avons apporté à notre projet afin de le rendre plus précis sont les "meilleurs voisins". Après avoir pris en compte les interactions entre les particules (implémenté dans la méthode bouger des faisceaux), l'implémentation de l'exercice final p14 permet de réduire le nombre de calculs quant à l'interaction entre les particules. On a donc segmenté l'accélérateur en cases circulaires à l'aide de la méthode segmenterAccelerateur dans l'accélérateur. Ainsi les coordonnées de chaque case

est simplement représentée par un entier, permettant ainsi chaque particule de savoir dans quelle case elle appartient.

5.2 Le Dessin d'un accélérateur

Enfin, afin de dessiner un accélérateur, nous avons implémenté, comme indiqué, une super-classe Dessinable. Tout les objets que l'on souhaite dessiner en hérite donc : Element et ses sous-classes, Faisceau et sa sous-classe, Particule et surtout Accélérateur sont des dessinable. C'est à dire que pour chacune de ses classes il a fallut changer les constructeurs afin de rajouter un support à dessin (attribut de Dessinable). De plus, chacune des classes suivantes à du redéfinir la méthode dessine qui est virtuelle pure dans Dessinable : Particule, Faisceau Circulaire, Dipôle, Section Droite, Quadrupôle, et Accélérateur. Cette méthode appelle la méthode dessine spécifique à l'objet sur lequel elle s'applique du support à dessin de celui-ci.

Ainsi nous avons aussi créé la super classe Support à Dessin ainsi que ces sous classe Support Graphique et Support Console qui respectivement dessinent l'accélérateur (c'est à dire tout les éléments et faisceaux qu'il contient) de façon textuelle et de façon graphique. Il n'y a pas grand chose à dire sur la classe Support Console puisqu'on n'avons fait que de suivre les instructions sans diverger des consignes. Cependant pour le Support Graphique nous avons en plus construit nos propres méthodes Cercle, Cylindre et Cylindre Incurvé pour simplifier le dessin graphique de l'accélérateur. Un cercle est conçu d'une multitude de lignes infinitésimalement petites misent ensemble pour former un cercle, et les cylindres sont contruit en empilant des cercles.

Ainsi, l'ajout de ses dernières classes permettent la visualisation de notre accélérateur maintenant qu'il est construit et fonctionnel.

6 Conclusion

En conclusion, notre conception de ce projet ne semble pas diverger de la conception conseillée. Nous nous sommes basés sur les instructions de chaque étapes afin d'implémenter ou de réorganiser le code de chacune des classes semaines après semaines selon les besoins et en essayant toujours d'optimiser au maximum celui-ci.

Enfin, pour chaque doutes conceptuels rencontrés nous avons du faire des choix d'implémentations. Nous avons pris systématiquement les options qui nous paraissaient les plus intuitives et les plus logiques, selon nous. Ils ne sont peut-être pas les choix que les meilleurs programmeurs utiliseraient, mais ils reflètent de notre conception du projet et permettent une utilisation facile instinctives de l'interface.