# RAPPORT PROJET DSP CALCULATOR

## Programmation Orientée Objet

Ingénieur Informatique 1re année


**Enseignant :**
**FOUILHOUX Pierre**

**Étudiantes :**
**HAMILCARO Chloé**
**MOUHIB Kawtar**


Le 06 mai 2022

# SOMMAIRE

## Introduction

Dans le cadre des études d'ingénieur informatique en première année dans l'école Sup Galilée, nous avons été sollicités pour effectuer un projet dans le cadre du cours de Programmation orientée objet. Ce projet, intitulé DSP Calculator, est codé en Java, et ce pour pouvoir mettre en pratique toutes les notions apprises durant ce cours.

Les jeux de type de factory Builder tels que Dyson Sphere Program ont pour but de mettre en place des chaînes de production complexes permettant, à partir d'une ou plusieurs ressources primaires, d'effectuer une série de transformations pour obtenir un produit final spécifique.

Dans ce projet, le projet DSP calculator (Dyson Sphere Calculator), nous allons à partir de données, qui ressemblent à celles du jeu Dyson Sphere Calculator, stockées dans un fichier XML, proposer aux utilisateurs des fonctionnalités, et ce grâce à la création d'un parseur qui parcourt le fichier XML, extrait les données pertinentes et les sauvegarde dans des structures de données pour qu'elles soient manipulées plus tard.

La première partie de notre projet se base sur la conception en utilisant le diagramme de classe en UML des données de notre fichier pour avoir une vue sur l'ensemble des éléments du projet.

La deuxième partie de notre projet vise la création du corps de notre parseur, à savoir toutes les classes, interfaces et fonctions d'extraction de données.

Enfin, dans la dernière partie nous avons codé les fonctionnalités qui seront proposées à l'utilisateur en utilisant toutes les données extraites et sauvegardées.

# I. Conception en UML

A partir des données trouvées sur le fichier XML (data.xml), nous avons pu effectuer une conception en UML en utilisant le diagramme de classe pour avoir une vue de l'ensemble des éléments de notre projet. On peut à partir de notre diagrammes visualisé qu'à partir de plusieurs ressources primaires, on peut obtenir un produit final spécifique.

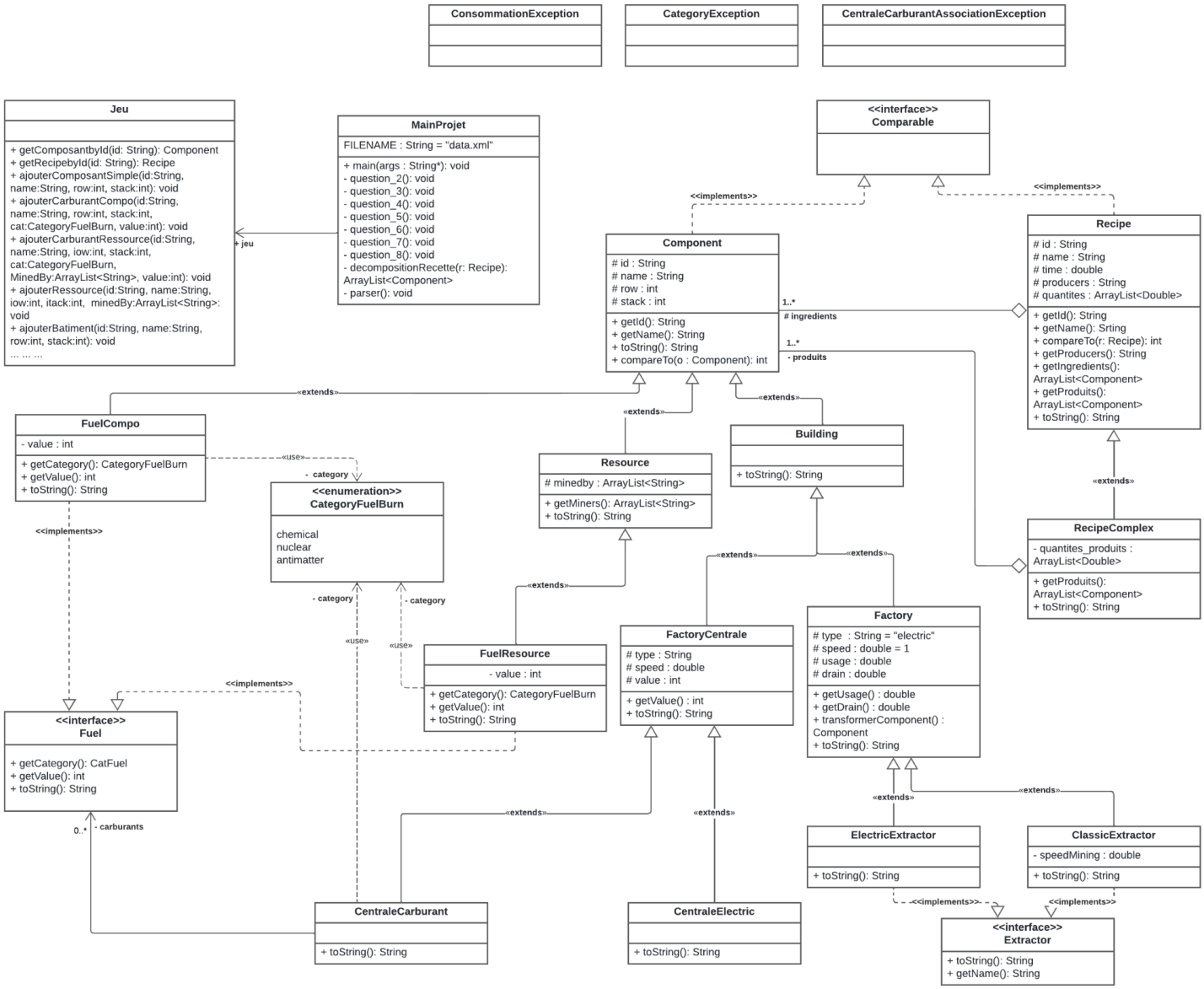


figure 1:Conception en UML en diagramme de classe

Une version png de ce diagramme se trouve dans le répertoire du projet.

Diagramme de classe

La classe Jeu est associée à toutes les autres classes du projet, exceptés l'énumération CategoryFuelBurn et les classes d'exception ConsommationException, CategoryException et CentraleCarburantAssociationException. Nous n'avons pas représenté ces associations par souci de lisibilité du diagramme, mais elles sont bien présentes et expliquées plus en détail dans la suite de ce rapport.

# II. Éléments du Projet

## A. Les classes du diagramme

Dans cette partie, nous allons vous expliquer les choix de modélisation qui nous paraissent importants à la compréhension de notre diagramme de classe.

Classe Component:
C'est la classe mère de toutes les classes de notre projet qui ne sont pas des recettes. Tout élément dans notre projet est un composant (Component), hormis les recettes, la classe Jeu et la classe MainProjet. La classe implémente l'interface Comparable<Component>, ce qui permet, par exemple, de trier une liste de composants en comparant chaque instance avec un autre composant.

Classe Building

C'est la classe qui représente les bâtiments du jeu. Ils sont soit des usines classiques, des centrales, des extracteurs ou bien tout simplement des bâtiments simples.

Classe Factory

Classe englobant tous les extracteurs : les extracteurs classiques (ClassicExtractor) et électriques (ElectricExtractor). C'est une classe instanciable et non abstraite, car certaines usines ne sont pas des extracteurs.

Classe FactoryCentrale

C'est la classe mère des centrales : électriques (CentraleElectrique) et à carburant (CentraleCarburant)

Classe Recipe

Cette classe est la classe des recettes simples, qui ne produisent qu'un seul élément, et la classe mère des recettes complexes, qui en produisent plusieurs (RecipeComplex). Elle implémente l'interface Comparable<Recipe>, pour permettre par la suite de comparer deux recettes entre elles.

Classe Resource

Il s'agit de la classe représentant les ressources. Une ressource est extraite par un extracteur. Cependant, notre parser lit le fichier xml dans l'ordre d'apparition des éléments. Par conséquent, nous avons décidé d'inclure dans la classe Resource une liste de chaînes de caractères représentant les identifiants des mineurs de cette ressource plutôt qu'une liste d'objets implémentant l'interface Extractor. C'est pourquoi il n'existe pas d'association visible entre la classe Resource et l'interface Extractor.

ConsommationException, CategoryException et CentraleCarburantAssociationException

Ces classes d'exception nous permettent de proposer la première fonctionnalité du projet : générer une exception dans le cas où un bâtiment électrique consommerait plus "au repos" qu'"au travail", dans le cas où une centrale à carburant ou un carburant n'aurait pas l'une des catégorie attendues et dans le cas où une centrale à carburant serait associée à un carburant d'un autre type que la centrale. Avoir trois sortes d'exceptions différentes permet par la suite

de gérer ces exceptions différemment. Ce n'est pas le cas actuellement dans notre projet, mais, par exemple, il se peut que l'une d'entre elles doit absolument faire s'arrêter le programme alors qu'une autre peut être ignorée ou traitée différemment. Ainsi, pour prévoir ces cas de figure, nous avons créé trois classes d'exception.

## B. Les interfaces du diagramme

Interface Fuel

L'interface Fuel définit les méthodes communes aux carburants (FuelCompo et FuelResource), qui implémentent cette interface. En effet, les deux types de carburants de notre application ne divergent que par leur classe mère. Pour maintenir une certaine cohérence entre ces deux objets, l'héritage multiple n'étant pas réalisable en Java, il a fallu créer une interface. De plus, cela permet de regrouper les deux sortes de carburant, ce qui est utile pour implémenter les fonctionnalités du projet.

## C. Les énumérations

Enumeration CatFuelBurn

Cette énumération est associée aux carburants qui héritent directement de la classe Component (FuelCompo), aux carburants qui sont également des ressources (FuelResource) et aux centrales à carburant (CentraleCarburant). Elle permet de définir la catégorie de ces objets, qui peut prendre trois valeurs : chemical, nuclear ou antimatter.

## D. Les classes Jeu et MainProjet

Classe Jeu

La classe Jeu est une classe qui contient les listes d'objets extraits par le parser. Nous avons conçu cette classe comme si nous ne savions pas à l'avance quelles listes nous seraient utiles, afin de pouvoir regrouper les objets qui peuvent l'être et distinguer ceux qui le doivent. Ainsi, la classe possède une ArrayList pour chaque classe du projet, et plus encore. Tout d'abord, il y a une liste qui regroupe toutes les instances de composants (Component), y compris les classes filles, et celles encore plus bas dans la hiérarchie visible sur le diagramme de classes. Puis, nous avons une liste composants_simples qui regroupe les instances de composants (Component) qui sont uniquement des instances de Component (et de Object, par définition), et n'héritent d'aucune autre classe. Il y a également une liste de carburants qui contient les objets implémentant l'interface Fuel, etc.

Une fois ces listes créées et initialisées dans le constructeur, il a fallu se demander comment les remplir. Sachant qu'il n'est possible de les remplir qu'au moment où le parser est lancé, nous avions deux choix ; par exemple, pour ajouter un carburant un tant que ressource (FuelResource), nous pouvions mettre dans le parser les deux instructions suivantes :

```
FuelResource f = new FuelResource(id, name, row, stack, MinedBy, value, cat);
jeu.carburants_ressources.add(f);
```

Cependant, une instance de FuelResource est également une implémentation de Fuel, une instance de Resource et une instance de Component. Or, la classe Jeu possède trois tableaux

pour ces trois cas de figure, il aurait donc fallu cinq instructions et non deux : cela représente un alourdissement conséquent du code du parser. De plus, nous souhaitions que l'instance jeu de la classe Jeu soit assez invisible du parser et que le fonctionnement code de celui-ci ne dépende pas énormément de la classe Jeu. Ainsi, nous avons choisi de créer une méthode pour chaque ajout d'un élément dans une liste, qui prend en paramètre tous les éléments nécessaires à son succès. Dans le cas de FuelResource, cette méthode commence par créer un un carburant en tant que ressource (FuelResource) qui est un composant (Component). Puis, l'objet créé est ajouté dans les listes adéquates, avec les transtypages nécessaires. Nous avons choisi de passer l'objet par référence : il s'agit donc du même objet dans chacune des listes. Il n'est en effet pas nécessaire de cloner l'objet à chaque fois pour l'ajouter dans une autre liste: il s'agit du même objet, catégorisé dans plusieurs listes.

La classe Jeu possède également une méthode `getComposantbyId()` et `getRecipebyId()` qui renvoient respectivement le composant et la recette dont l'identifiant est passé en paramètre. Ce sont des fonctions qui sont utilisées dans d'autres classes du projet. Enfin, la classe contient deux listes et méthodes utiles pour répondre aux questions du sujet.

Classe MainProjet

La classe MainProjet est la classe principale du projet. Elle contient la méthode main qui sert à lancer le menu des fonctionnalités que l'utilisateur peut utiliser, ainsi qu'un attribut statique `jeu` de type `Jeu`, qui représente tous les éléments du jeu (composants et recettes). Cet attribut est statique et publique, pour que d'autres classes du projet puissent y accéder : par exemple, la méthode `toString()` de la classe `Resource` a besoin de trouver un composant par son identifiant. Pour cela, elle fait appel à la fonction `getComposantbyId(String id)` de la classe `Jeu`, via l'attribut statique `jeu` de la classe MainProjet.

La classe contient exclusivement des méthodes statiques, dont certaines sont privées, pour pouvoir être utilisées uniquement par cette classe comme des fonctions en programmation impérative. En effet, la méthode main étant une méthode statique, il est impossible d'utiliser `this.nom_de_la_methode()` pour appeler une méthode de la classe MainProjet ; nous avons donc fait de toutes les méthodes des méthodes de classe.

Cette classe contient également le parser, dont le fonctionnement est décrit dans la partie suivante.

## III. Explication du parser

Le parser est la partie algorithmique la plus complexe du projet. Il s'agit d' une méthode qui permet d'extraire les données d'un fichier xml. Dans notre cas, cette méthode est une méthode `static` et se trouve dans la classe `MainProjet`, et permet de récupérer les données du fichier data.xml. Le nom du fichier est un attribut static et final de la classe : le programme ne peut pas le modifier par la suite. Cela est important car le parseur ne fonctionnera qu'avec les données qui viennent de ce fichier, qui nous a été fourni. La méthode, quant-à-elle, est private: on ne veut pas que d'autres classes puissent utiliser le parser. Le but de cette méthode est d'extraire et de stocker toutes les données avant que l'utilisateur ne puisse faire quoi que ce soit, et ne plus accéder au fichier par la suite.

Le parser peut soulever deux types d'exceptions, qui seront traitées par la méthode appelante: ConsommationException et CategoryException. Les exceptions ParserConfigurationException, SAXException et IOException sont gérées en interne par la méthode et capturées à l'aide d'un bloc try/catch.

Le fonctionnement du parser est le suivant : il récupère tout d'abord les éléments dont l'étiquette la plus haute est items, sous forme de nœuds. Ces éléments sont tous des instances de composants, comme précisé dans l'énoncé.

Les éléments entre les étiquettes items ont ensuite des syntaxes spécifiques, qui déterminent leur sous-classe : `Resource, Building, CentraleElectrique`... Cependant, le parser n'a aucun moyen de déterminer à l'avance si oui ou non certains tags, comme le tag fuel, sera présent dans l'élément qu'il est en train de traiter : ainsi, il faut s'occuper d'implémenter chaque cas de figure. Par exemple, si l'élément possède un tag fuel, nous savons qu'entre ce tag se trouvera d'autres tags : `category` et `value`. Nous ne pouvons cependant pas essayer de chercher le tag category dans le tag `fuel` si ce dernier n'est pas présent, cela n'a pas de sens et générera une erreur. Ainsi, il faut tester en avance si l'élément fuel est bien présent.

```
if ((fuel = (Element) element.getElementsByTagName("fuel").item(0)) !=
null) {
      category_fuel = fuel.getElementsByTagName("category").item(0)
.getTextContent();
      value = Integer.parseInt(fuel.getElementsByTagName("value").item(0)
.getTextContent());
}
```

*Extrait de code : test de la présence d'un élément "fuel"*

Une fois que les éléments permettant d'instancier une classe ont été récupérés, la méthode regarde, à l'aide de la méthode equals, quelle est la catégorie du composant extrait : est-ce un composant que l'on peut qualifier de "simple", est-ce une ressource ou bien un bâtiment ? Si le tag "category" vaut "component", la fonction regarde si une valeur correspondant au tag "fuel" a été récupérée. Si ce n'est pas le cas, il s'agit d'un composant simple. Si c'est le cas, il s'agit d'un carburant qui est également un composant. Cela signifie également que la variable `category_fuel`, qui détermine la catégorie du carburant, a été initialisée : la fonction associe cette chaîne de caractères à un élément de l'énumération `CategoryFuelBurn`, avant d'appeler la méthode de l'instance jeu permettant d'ajouter un carburant en tant que composant. L'énoncé

nous impose de soulever une exception si le carburant est associé à une catégorie qui n'est pas attendue. Ainsi, si la catégorie n'a pas l'une des trois valeurs attendues, le parser renvoie une erreur, qui est gérée par la méthode appelante.

```
if (category_fuel.equals("chemical"))
                    jeu.ajouterCarburantCompo(id, name, row, stack,
CategoryFuelBurn.chemical, value);
else if (category_fuel.equals("nuclear"))
              jeu.ajouterCarburantCompo(id, name, row, stack,
CategoryFuelBurn.nuclear, value);
else if (category_fuel.equals("antimatter"))
              jeu.ajouterCarburantCompo(id, name, row, stack,
CategoryFuelBurn.antimatter, value);
else
                    throw new CategoryException("Le carburant " + id + "
n'a pas une des categories attendues.");
```

*Extrait de code : génération d'exception*

L'exemple d'un carburant en tant que composant permet d'illustrer simplement le fonctionnement du parser, qui fonctionne de la sorte pour tous les autres composants, en adaptant les tests en fonction des tags présents.

Après avoir traité tous les composants du fichier, le parser traite les données dont l'étiquette la plus haute est recipe, c'est-à-dire les recettes. L'extraction des données des recettes suit les explications décrites précédemment. Si un tag "out" est présent, le parser ajoute au jeu une recette complexe, sinon, une recette simple est ajoutée.

## Conclusion

Le projet "DSP Calculator" conclut ainsi le cours de Programmation Orientée Objet, nous permettant de mettre en pratique tous les savoirs que nous avons acquis au cours de ces dernières semaines, ainsi que de découvrir de nouveaux éléments comme les parsers. Nous avons pour la première fois pu utiliser les listes chaînées en Java, plus particulièrement les ArrayList, et déterminer combien elles sont bien plus pratiques à manipuler que les tableaux. Nous avons également appliqué les notions d'héritage, d'implémentation. Cela nous a permis de voir les limites de Java, qui ne permet pas de faire de l'héritage multiple et nous oblige à utiliser les interfaces pour pallier cette faiblesse, entraînant parfois de la redondance dans le code du projet. Enfin, nous avons utilisé les énumérations et les exceptions.

Ce projet nous aura permis de comprendre l'importance de la modélisation lors du développement d'un projet, faisant ainsi le lien avec le cours de Modélisation des Systèmes Informatiques. En effet, réaliser un diagramme de classes avant de se mettre à programmer nous a permis de gagner du temps sur la partie structuration des données, puisque nous avions déjà une idée claire des attributs et méthodes nécessaires au sein d'une classe. La construction du diagramme de classe nous a également permis de nous représenter plus facilement les liens entre les différents objets, afin de déterminer où placer efficacement les méthodes des classes. Au cours du projet, nous avons parfois eu à modifier les classes et revoir notre modélisation, lorsque nous avons rencontré des problèmes que nous n'avions pas anticipés. Par exemple, l'attribut drain d'une usine était pour nous un entier, alors qu'il s'agissait en réalité d'un double : modifier le type de cet attribut a engendré plusieurs autres modifications, auxquelles il a fallu faire particulièrement attention.

Nous avons également pu découvrir comment extraire les données d'un fichier XML à l'aide d'un parser. Cette partie, bien que très technique puisque qu'il s'agissait de quelque-chose de très algorithmique que l'on n'avait pas vu en cours, s'est révélée assez simple une fois le mécanisme du parser compris grâce aux exemples fournis en annexe du projet.

Enfin, ce projet a été l'occasion de travailler en groupe, d'échanger nos interrogations sur le sujet, et de confronter nos idées pour faire les choix qui nous paraissaient les plus justes. Ainsi, nous avons acquis de nouvelles compétences techniques et notions en programmation orientée objet. Ces compétences acquises sont également des compétences humaines : la communication est un élément essentiel de la réussite de ce projet.