**Conception Orientée Objet**

**Carnet de bord**

**Da Silva Goncalves André Montoya Damien**

**Sommaire**

Table des matières

[2 Introduction et rappels 2](#_Toc501818995)

[2.1 Le diagramme de cas d’utilisation 2](#_Toc501818996)

[2.2 Le diagramme de classes 2](#_Toc501818997)

[2.3 Le diagramme de séquence 2](#_Toc501818998)

[2.4 les design pattern 2](#_Toc501818999)

[3 GRASP - General Responsability Assignment Software Patterns 3](#_Toc501819000)

[3.1 Définitions 3](#_Toc501819001)

[3.2 Mise en œuvre de grasp 4](#_Toc501819002)

[4 SOLID 8](#_Toc501819003)

[4.1 Definitions 8](#_Toc501819004)

[4.2 Mise en œuvre de solid 9](#_Toc501819005)

# Introduction et rappels

Pour développer une application certaines méthodes et techniques se doivent d’être apprises. En effet la lecture et la compréhension du cahier des charges est une étape importante qui permettra de bien cerner toutes les fonctionnalités du projet et de produire une application correspondante aux attentes du client. Cet étape d’analyse est assistée par la présence d’outils et de méthodes tel que les user stories ainsi que certains diagrammes notamment les diagrammes UML et les design pattern (patrons de conception). Ses concepts permettront par la suite de produire des documents permettant de bien mettre en évidences les attentes du client ainsi que de visualiser en partie l’architecture de l’application. Au cours de ces TD nous verrons principalement l’étude des diagrammes de classes ainsi que les diagrammes de cas d’utilisation et de séquence mais d’abord nous verrons quelques rappels et explications.

## Le diagramme de cas d’utilisation

Les diagrammes de cas d’utilisation permettent de définir le système du point de vue des utilisateurs (et non depuis le nôtre, celui des développeurs !). Il permet également de définir les limites du système. Ils nous permettent enfin de structurer les besoins, grâce au cahier des charges, ainsi que tout le reste du développement.

## Le diagramme de classes

Les diagrammes de classes sont réalisé grâce à la détermination des concepts. On effectue une simple analyse grammaticale de la description textuelle des cas d’utilisation. En général, les noms représentent des concepts ou des attributs alors que les verbes représentent des comportements (opérations, méthodes).

## Le diagramme de séquence

Les diagrammes de séquence nous permettent de décrire le comportement dynamique (comportement dans le temps) d’un système dans un diagramme. Ils permettent également de montrer comment des sociétés d’objets peuvent collaborer pour réaliser les cas d’utilisation.   
On y précise le contenu d’un cas d’utilisation en déroulant les scenarios (flots d’évènements). En général, on y décrit que les scénarios les plus représentatifs

## les design pattern

En génie Logiciel, un design pattern (patron de conception en anglais) est un concept destiné à résoudre les problèmes récurrents suivant le paradigme objet. Les patrons de conception décrivent des solutions standards pour répondre à des problèmes d'architecture et de conception des logiciels (source Wikipédia)

# GRASP - General Responsability Assignment Software Patterns

## Définitions

GRASP est un design pattern se basant sur le principe de la responsabilité qui se divise en 2 parties.

* La responsabilité de faire qui peut être le déclenchement ou la réalisation d’une action ou bien de contrôler et de gérer les activités d’un autre objet.
* La responsabilité de savoir notamment connaître les valeurs de ses attributs via des accesseurs ou de connaitre les objets qui lui sont rattachés.

Ce pattern se base notamment sur 7 principes de développement orienté objet.

* **Information Expert** : Qui est responsable des actions sur les objets ?

On affecte cette responsabilité à la classe qui possède les informations de cet objet pour s’acquitter de cette tâche. Il faut donc définir dans un premier temps les informations attribuées à l’objet ainsi que savoir quels objets en sont responsables.

* **Creator** : Qui doit se charger de créer une instance d’un objet ?
  + On affecte à la classe B la responsabilité de créer une instance de la classe A si au moins :
    - B contient ou agrège des objets A
    - B enregistre des objets A
    - B utilise étroitement des objets A
    - B a les données d’initialisation qui seront transmises aux objets A lors de leur création
* **Low Coupling** : Comment minimiser les dépendances et réduire l’impact des changements ?

Le couplage peut être défini par les liens dont dépends un objet envers un autre. Plus un objet sera dépendant d’un autre, plus le couplage sera fort. La seule solution pour minimiser ce couplage et ces dépendances est d’étudier au mieux les objets utilisés pour réduire le couplage.

* **Controller** : Quel est le premier objet au-delà de l’IHM qui reçoit et coordonne une opération système ?

Le contrôleur est un objet se situant entre l’application et l’IHM. Cet objet aura pour finalité de recevoir et traiter les évènements systèmes entrainés par une interaction. Pour modéliser le contrôleur il suffit de créer un objet dans la couche application qui aura ce rôle de traitement d’évènements.

* **High Cohesion** : Comment parvenir à maintenir la complexité gérable ? Comment s’assurer que les objets restent faciles à gérer et qu’ils respectent le faible couplage ?

Une classe de forte cohésion sera une classe facile à comprendre à réutiliser à maintenir et dont les dépendances sont réduites. Une classe de forte cohésion devra être étudiée de sorte à contenir un nombre réduit de méthodes liées entre elles.

* **Polymorphisme** : Comment gérer des alternatives dépendantes des types ?

Le polymorphisme est le fait d’avoir un objet parent composé de plusieurs méthodes et attributs qui étends des objets fils. Ces objets fils permettront alors de disposer des méthodes définies dans la classe parente telles quelles ou modifiées mais encore de disposer d’autres méthodes et attributs.

* **Fabrication Pure**: Que faire quand les concepts du monde réel ne sont pas utilisables en respectant le faible couplage et la forte cohésion ?

Quand les concepts du monde réel ne peuvent s’appliquer au faible couplage et à la forte cohésion une solution est de créer une classe artificielle ne représentant pas un concept du domaine.

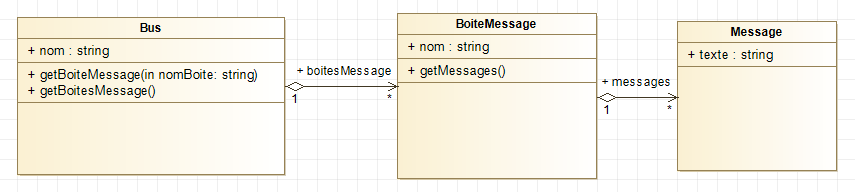
## Mise en œuvre de grasp

Durant nos séances en TP le principe du pattern GRAPS a été utilisé lors de la création d’un bus à message et de flux d’informations. Le principe est de permettre à des utilisateurs de la même manière que sur un forum ou un réseau social, de créer des bus de message identifiable via un nom, eux-mêmes composés de boites de messages contenant enfin des messages.

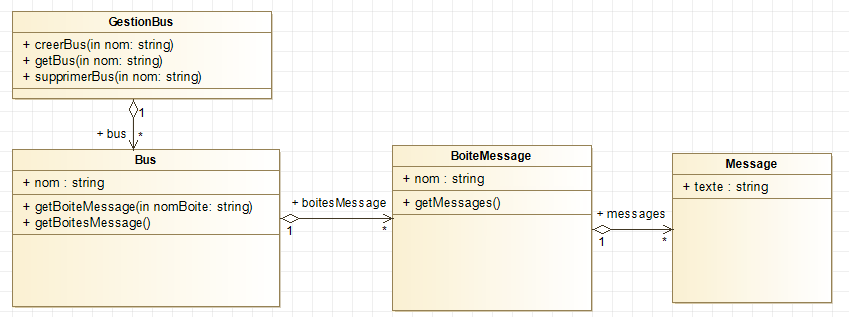
Les principes GRASP seront alors mis en place durant cet exercice indiquant les solutions à utiliser pour respecter ce pattern.

En étudiant l’énoncé nous pouvons alors nous poser plusieurs questions quant à l’utilisation de GRASP. Qui est responsable de connaitre les boites de messages ? Cette question soulève alors le principe Information Expert. En effet nous avons 2 classes, Bus ainsi que BoiteMessage. D’après l’énoncé le bus est composé de boites. Dans ce cas, c’est la classe Bus qui contiendra les méthodes permettant d’accéder aux boites. Ce principe s’applique de nouveau concernant la classe BoiteMessage, en effet celle-ci contiendra des messages c’est donc à elle de savoir quels messages lui sont associés.

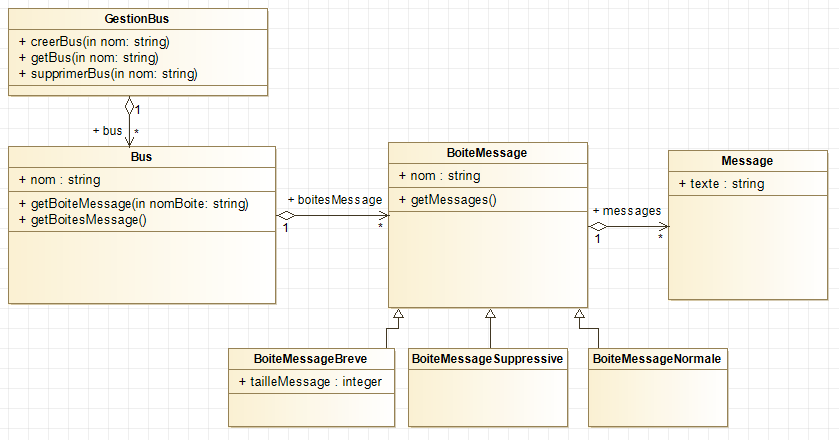
Par convention nous ne mettons pas les accesseurs dans les diagrammes de classe mais pour les besoins de ce rapport ceux-ci seront tout de même disposés.



Le problème restant est maintenant de savoir qui va créer la classe Bus ? Heureusement pour nous un des principes de GRASP est là pour répondre à la question, et ce principe est le Creator. En effet pour gérer alors les différents bus nous pouvons créer une classe qui se chargera de traiter les actions sur les bus tel un gestionnaire de bus. Ce gestionnaire contiendra dans un premier temps les méthodes de gestion des bus et sera ainsi composé de bus. C’est également ce gestionnaire qui sera en charge de la création des boites de message et de leur association au bus ainsi que de la création des messages.

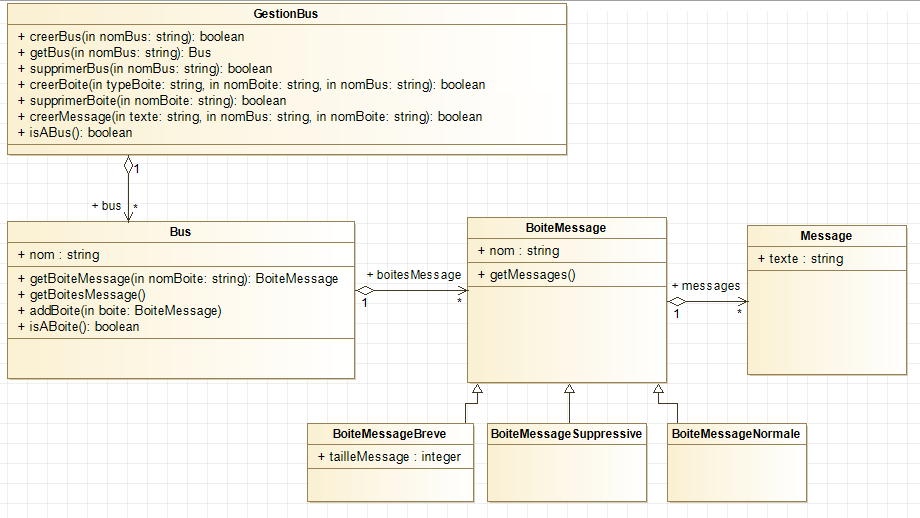


Mais si notre système de bus évolue et nous donne la possibilité de poster nos messages dans des boites différentes chacune avec une certaine spécialité, comment modéliser un tel principe ? Il faut bien évidement utiliser le principe du polymorphisme. En effet le principe du polymorphisme nous permettra de créer des classes filles qui hériterons des méthodes et attributs de leur classe mère. Pour illustrer ce principe nous allons ajouter 3 types de boites. Les boites « brèves » qui pour chaque message créé ne pourra contenir que 140 caractères. Les boites « suppressives », dès lors que nous lisons un message ceux-ci sont supprimés ainsi que le type boite message normal.



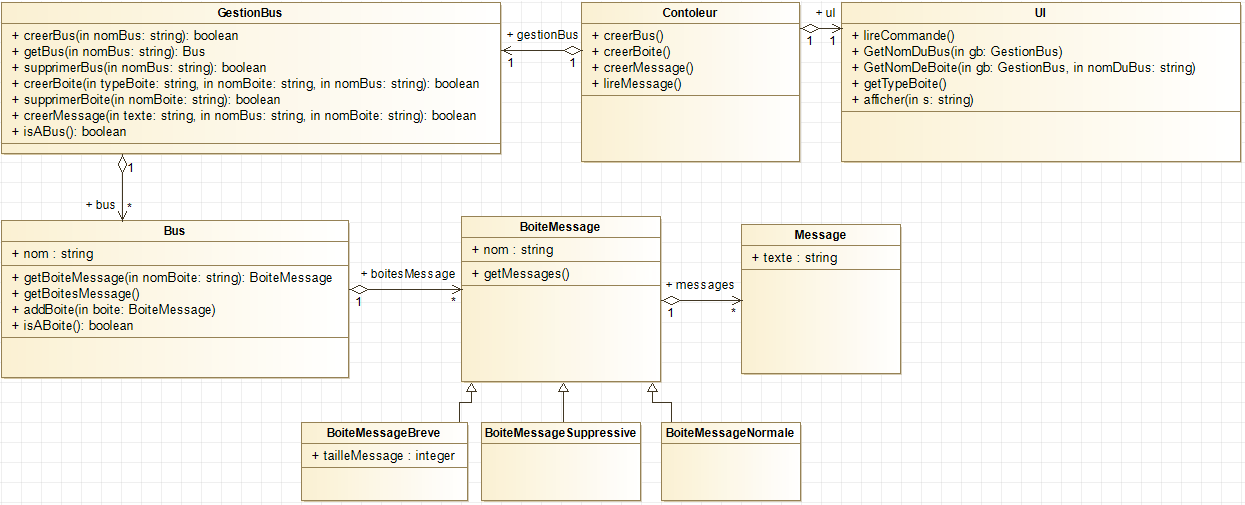
Dans le diagramme de classe nous avons ajouté un attribut tailleMessage pour les boites brèves qui permettra de valider et modifier plus aisément la taille maximale des messages par la suite.

Nous devons maintenant permettre aux utilisateurs de pouvoir créer des boites de messages et ainsi que des messages. C’est donc notre gestionnaire de bus qui va se charger de ceci.



Les méthodes isABus et isABoite permettront de vérifier que lorsque l’utilisateur voudra ajouter des messages ou des boites que les objets dont ils dépendent existent bien.

Maintenant que nous avons notre diagramme de classes sur le fonctionnement métier de l’application nous nous intéressons à l’interface. D’après le td grâce à celle-ci nous devrons être en mesure de pouvoir créer des bus, des boites ainsi que des messages et lister ceux-ci. Il faut savoir que l’interface n’interagit jamais directement avec la partie métier, c’est un contrôleur qui prend ce rôle et donc qui nous permet d’établir un lien avec le principe de GRASP et du contrôleur. Nous aurons donc une classe qui se chargera d’interagir avec l’interface et d’utiliser la partie métier de l’application. Pour l’application de ce td notre interface sera utilisée en ligne de commandes. Les modifications précédentes nous permettent alors d’obtenir le diagramme de classe suivant :



Grâce à ce diagramme nous pouvons dès à présent commencer la création du code. Dans notre cas nous nous attarderons sur la classe Controleur ainsi que UI qui sont les éléments les plus compliqués de ce principe à prendre en main.

public class Controleur{

GestionBus gestionnaireBus = new GestionBus();

UI ui = new UI();

//le programme principal qui permet grâce à l'interface de créer nos objets

public void start() throws Exception {

String commande = ui.lireCommande();

switch (commande.charAt(0)) {

case 'c':

creerBus();

start();

;break;

case 'b':

creerBoite();

start();

;break;

case 'l':

lireMessage();

start();

;break;

case 'e':

creerMessage();

start();

;break;

case 's' :

ui.afficher("Au revoir !");

; break;

default :

ui.afficher("Erreur de commande");

start();

; break;

}

}

private void creerBus() {

String nomDuBus = ui.getNomDuBus(gestionnaireBus);

boolean reussi = gestionnaireBus.creerBus(nomDuBus);

if (!reussi)

ui.afficher("Erreur le bus existe déjà");

else

ui.afficher("Bus " + nomDuBus + " créé");

}

Les méthodes creerBoire, lireMessage, creerMessage fonctionneront de la même amnière que creerBus, via des vérifications.

# SOLID

## Definitions

* **Single Responsability Principe (SRP) :**

On ne doit pas donner à une classe plusieurs responsabilités. Elle ne doit en avoir qu’une seule. Pourquoi ? Car en tant que programmeur on change toujours nos classes afin de résoudre les différents problèmes que l’on a (non ce n’est pas un bug de la machine ou d’Eclipse). Afin de posséder un code plus robuste et d’éviter la recherche d’erreurs il faut absolument associer une classe à une responsabilité qu’elle sera la seule à avoir.

* **Open/closed principle (OCP) :**

La plupart des informaticiens considèrent ce principe comme l’un des plus important. Après avoir coder et tester une classe celle-ci est considérer comme Open donc ouverte aux extensions. Ce que nous entendons par extensions ce sont par exemple les différents « Extends » que l’on peut ajouter à une classe.

En revanche si la class marche bien on a aucune raison de la modifier d’où l’idée de fermeture (closed). Par exemple pour passer des attributs qui étaient en public à private.

La petite astuce : Ce principe est souvent impossible à appliquer partout mais essayer de le faire au maximum.

* **Liskov subtition principle (LSP) :**

Dans votre programme vous avez surement plusieurs objets qui font partie d’une class. Ces différents objets instanciés par votre class peuvent tous être instancier par des sous classes sans pour autant casser le programme.

Prenons l’exemple du carré et du rectangle. Le carré étant un rectangle on est sensé pouvoir remplacer tous les objets rectangles par des objets de type carré sans pour autant créer des erreurs.

* **Interface segregation principle (ISP) :**

Ici il s’agirait plutôt d’un principe qui affecte le client. Le principe de la ségrégation des interfaces consisterait à limiter les informations sur l’interface afin de pouvoir guider le client et empêcher le surplus d’informations. Il ne doit voir que des méthodes qui lui sont utiles.

* La petite astuce : Au lieu d’avoir qu’une seule interface globale pensez à la séparer en plusieurs éléments.
* **Dependency inversion principle (DIP) :**

Les class de doivent pas dépendre des implémentations mais des abstractions (Abstracts). Pourquoi ? Parce que les classe de haut nieau PAS FINI

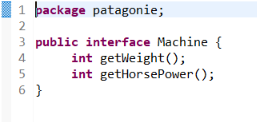
## Mise en œuvre de solid

Le TD numéro 3 fut l’un des plus court, le sujet portait sur l’élaboration d’un nouveau moteur. L’entreprise qui mit au point ce moteur voulait faire des simulations qui porte sur le comportement des véhicules par exemple calculer la vitesse maximale.

Le polymorphisme :

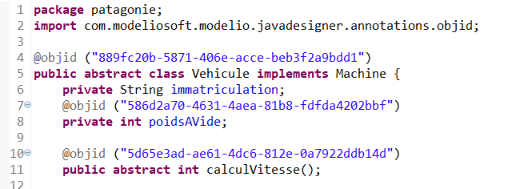
Le fait que Petitbus, camionCiterne ou encore CamionBache soit un vechicule crée un polymorphisme.

Une interface :

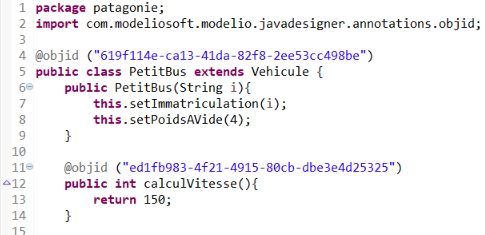
Sur l’image de droite vous pouvez voir l’interface que nous avons utilisé.

A quoi sert une interface ? Une interface est en quelque sorte le père de toute les classes qui l’implémente. Dans ce cas présent, la classe machine est celle qui donne des ordres, elle oblige toutes les autres classes comme Vehicule à implémenter les méthodes getWeight() et getHorsePower().

Une classe abstraite :

Il est impossible d’instancier une classe abstraite ce qui veut dire que nous ne pouvons pas la définir. Plus précisément si vous avez une méthode celle-ci n’y sera pas défini, il faudra simplement y laisser son nom et ses attributs. En effet cette méthode sera alors définie dans les classes filles.

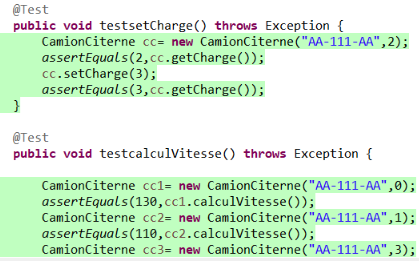
Mais alors à quoi sert une classe abstraite ?



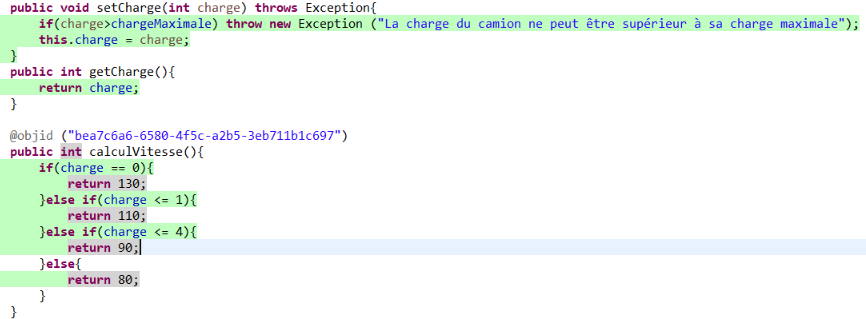
A dire que les classes ont un lien entre elle pardi ! Il nous arrive de parler de factorisation de code lorsque l’on souhaite parler de classe abstraite. Dans notre TD ce sont des classes comme calculVitesse () qui sont factorisées.

Lors de cette séance nous avons appris à faire des tests. Il nous a bien fallu plus d’un an pour apprendre que les tests ce n’est pas de simples comparaisons entre deux chaines de caractères. Sans oublier que Eclipse nous permet de vérifier si toutes les classes sont bien tester. Si elles sont bien testé elles seront surlignées en vert.

Le test



Le code tester :

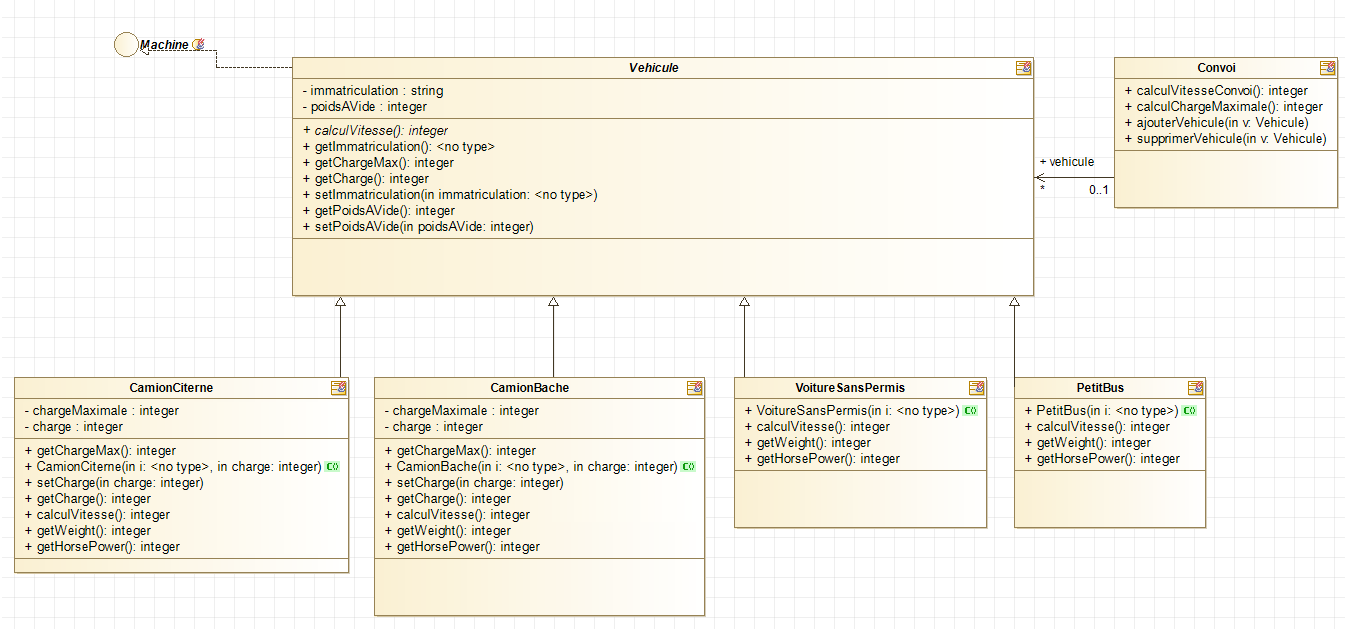


Et tout cela grâce à une petite astuce ce fameux bouton d’exécution



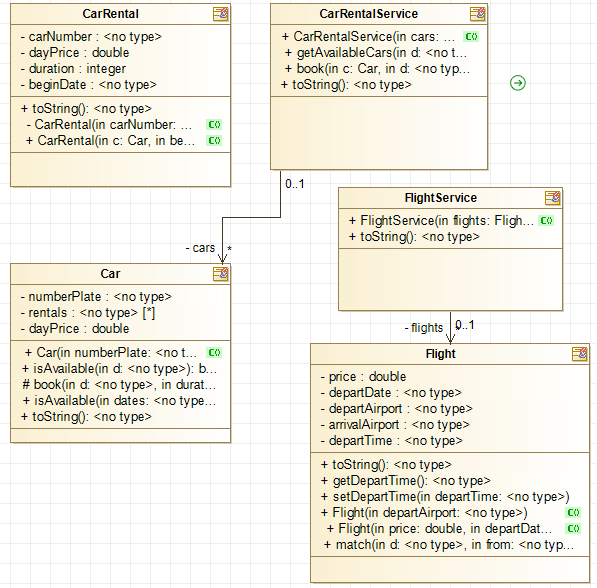
Le plus dur fut de comprendre la différence entre la classe abstraite et les interfaces qui permettent la création d’un polymorphisme.

La différence entre une classe abstraite et une interface est simple : Avec une interface on parle d’une implémentation alors qu’une classe abstraire on parle d’extensions. Il n’est possible pour une classe d’hériter que d’un seul parent en revanche il est possible d’implémenter plusieurs interfaces dans une seul et même classes.

LE DIAGRAMME DE CLASSE QUI VAS AVEC

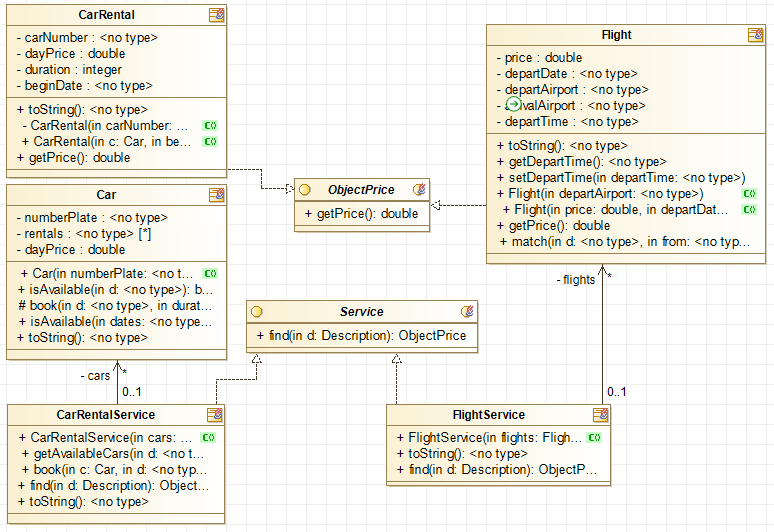
Le principe open/closed est un principe important dans le pattern SOLID c’est donc pour cela que nous allons en parler plus en détail avec un exemple concret. Dans notre cas la société Amadeum nous demande d’améliorer la gestion de vols et de locations de voitures. Elle nous demande pour faire ceci d’améliorer un service de préparation de voyages permettant en fonction d’une description d’un voyage, des lieux et dates de voyages de proposer au client le moyen le moins cher d’effectuer son trajet.

Nous avons donc au départ après analyse des codes les voitures et avions créés. De plus nous avons un service permettant de gérer les voitures ainsi qu’un service permettant de gérer les avions séparément nous permettant alors d’établir le diagramme de classes suivant :



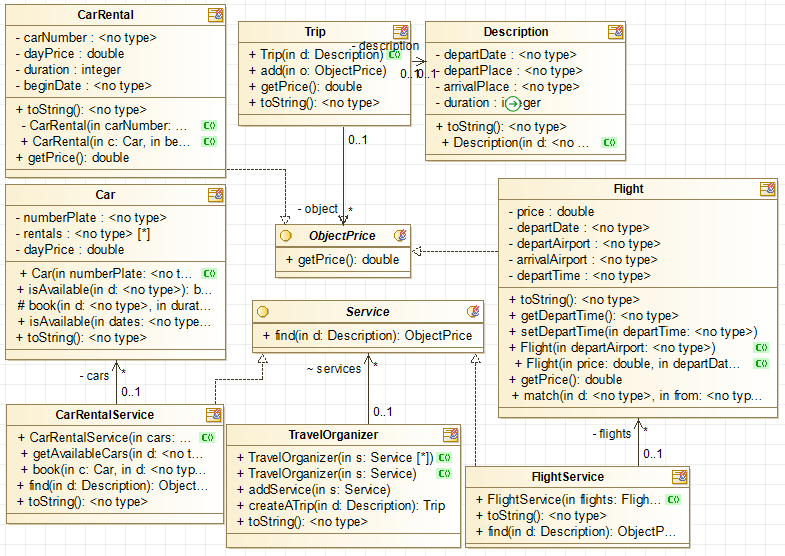
Mais comment faire pour organiser des voyages en calculant le prix minimum que ce soit par avion ou voiture ? Et bien tout simplement en rendant notre code ouvert et en lui permettant d’être extensible. Pour implémenter un tel principe nous pouvons utiliser comme solution l’implémentation. En effet, en créant des interfaces et en les implémentant par ces classes nous serons en mesure de définir les méthodes même méthodes pour chaque classe permettant par la suite de les utiliser lors de la création des voyages par un gestionnaire de voyage.

Nous aurons donc une interface nommée ObjectPrice qui permettra de redéfinir la méthode getPrice() et qui sera implémentée par Flight ainsi que CarRental. Cette méthode permettra alors a la classe gérant chacun des moyens de locomotion soit CarRentalService ainsi que FlightService d’obtenir le prix de chaque véhicule sachant que ceux-ci peuvent varier en fonction de divers critères. Notre but étant de retourner le moyen le moins cher de faire un voyage nous pouvons également créer une interface services qui sera implémentée par les gestionnaires permettant ainsi de trouver les moyens de déplacement les moins chers, interface qui retournera un objet de type ObjectPrice créé précédemment.



La méthode find maintenant mise en place devra être implémentée et utilisée par le type ObjectPrice. Pour trouver donc les meilleurs moyens de déplacements aux meilleurs prix nous pouvons imaginer une classe abstraite, CompareTools qui pourra trier les divers moyens de transport et retourner le moins cher. Cette classe sera définie en tant que classe abstraite car celle-ci n’aura pas besoin d’être instanciée, en effet nous pourrons utiliser directement des méthodes propres à la classe grâce aux arguments. Maintenant que nous avons la structure permettant de récupérer les meilleurs moyens de déplacement il nous suffit de créer un gestionnaire de voyages. Nous aurons donc un gestionnaire de voyage qui permettra de créer des voyages représentés ici par la classe Trip permettant de stocker dans cette classe les différents moyens de locomotion à prendre.

Cela nous amène donc au diagramme de classe qui suit :



Le diagramme de classe nous permet alors de produire plus facilement le code. Nous n’avons pas mis les accesseurs dans ce diagramme pour ne pas insérer trop d’informations superflues.

La comparaison des prix étant importante voici donc la classe CompareTools une fois implémentée :

public class CompareTools {

public static void sortedByPrice(ArrayList<ObjectPrice> ob){

Collections.*sort*(ob, new Comparator<ObjectPrice>(){

*@Override*

public int compare(ObjectPrice f1, ObjectPrice f2){

return (int)(f1.getPrice() - f2.getPrice());

}

});

}

public static ObjectPrice cheaper(ArrayList<ObjectPrice> ob){

*sortedByPrice*(ob);

return ob.get(0);

}

}

Voici également ce que l’on peut obtenir pour trouver les vols les moins chers :

**public** ObjectPrice **find**(**Description** d) {

**ArrayList**<Flight> **o** = getFlights(d.getDepartDate(), d.getDepartPlace(), d.getArrivalPlace());

**ArrayList**<ObjectPrice> **op** = **new** ArrayList<ObjectPrice>();

**for**(**int** **i**=0;i<o.size();i++){

op.add(o.get(i));

}

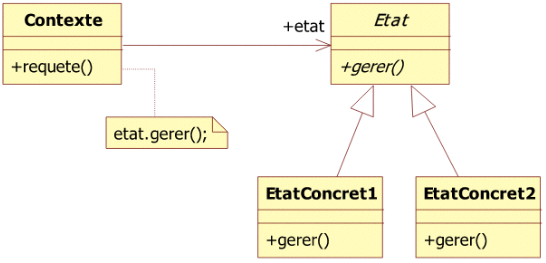
**return** **CompareTools**.*cheaper*(op);

}

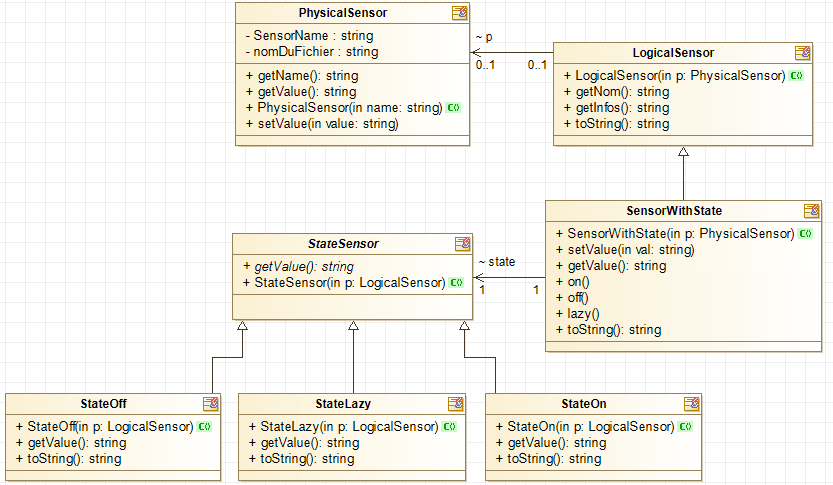
Grâce à la boucle nous récupérons nos Flight en tant qu’ObjectPrice et nous retournons simplement le moins cher.

# Pattern state et composite

Dans ce tp nous avons vu l’utilisation de deux pattern intéressants par leur optimisation du code. En effet le pattern state nous permet d’affecter à un objet un état divisé en sous états permettant alors une utilisation et une modification facile de l’objet. Pour avoir un aperçu de ce pattern voici un diagramme de classe explicite :



Par exemple nous avons des capteurs. Ces capteurs peuvent être sur On, sur Off ou sur l’état Lazy. Comment modéliser une telle chose ? et bien tout simplement en utilisant le pattern state qui permettra à un capteur de disposer d’un objet StateSensor ou ses classes filles seront alors On, Off et Lazy. Ces classes filles contiendrons alors des méthodes communes qui pourront s’exécuter de manière différente en fonction des états. Dans le cadre de notre TD, un LogicalSensor est associé à un PhysicalSensor. De plus nous avons une classe SensorWithState permettant d’affecter à un LogicalSensor un état. Cela nous permet donc d’établir un premier diagramme de classe :

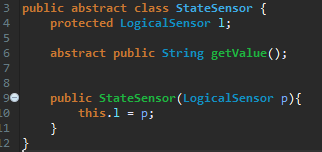


Nous pouvons donc voir sur le diagramme de classes ci-dessus que le design pattern state est bien appliqué, en effet nous avons différents états descendant d’une classe mère StateSensor.

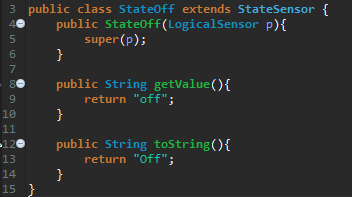
Voilà donc le code associé à ce diagramme de classe :



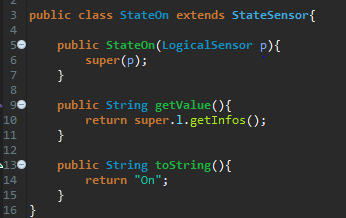
Nous aurons donc en premier temps la classe permettant de définir un capteur associé à un état.



Nous aurons ensuite la classe StateSensor qui sera l’état mère et qui donc ne pourra être instancié. La méthode abstract getValue() sera donc à redéfinir pour chaque classe fille.

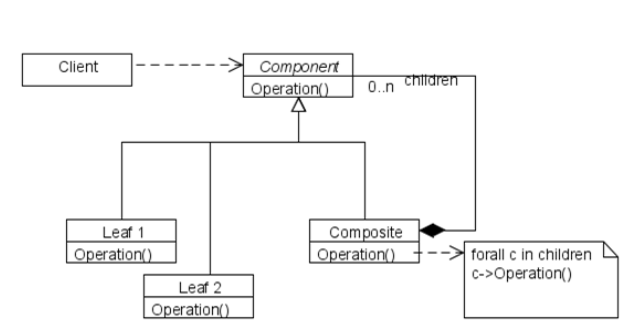


Dans le cas où le capteur est sur off la méthode getValue retournera « Off »

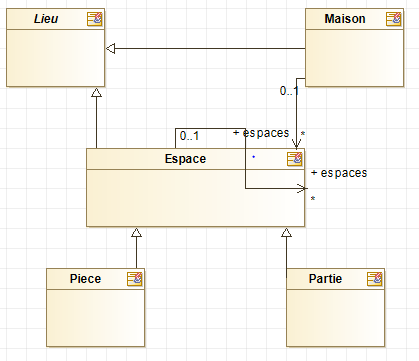


Dans le cas où le capteur est sur on la méthode getValue retournera alors la valeur du capteur.

Mais notre travail ne s’arrête pas là, en effet des capteurs doivent maintenant être disposés dans une maison. Dans une maison se trouve des parties comme par exemple l’étage, le rez de chaussé mais dans ces parties il existe également des pièces. Une partie peut contenir des capteurs mais une pièce aussi et la maison également peut obtenir un capteur central. Un moyen simple et optimisé existe alors pour permettre d’avoir les méthodes concernant les capteurs pour chaque lieu, le pattern Composite.



D’après ce diagramme, nous pouvons alors visualiser le principe de ce pattern. En effet, un Composite peut se composer de plusieurs components qui peuvent alors être des composites et donc peuvent contenir à nouveau des components. Les méthodes pourront alors s’exécuter sur chaque objet. Si l’on suit ce principe, voilà le diagramme de classes que nous pouvons obtenir pour notre maison :



Le lieu est donc le composant qui contiendra des capteurs. En effet, nous pouvons avoir des capteurs dans la Maison mais aussi dans les Pieces et Parties.