
Conception Avancée - Motifs de Conception

Réalisation d'une simulation d'un monde virtuel

AUGER Romain | RIO Charles

21-12-2018

Table des matières

1	Introduction	3
2	Résultats des TPs	4
2.1	Présentation générale	4
2.2	Remarques sur chaque partie	5
2.2.1	Partie 1	5
2.2.2	Partie 2	5
2.2.3	Partie 3	7
2.2.4	Partie 4	7
2.2.5	Partie 5	7
3	Partie 6 - Le partage de Monde en réseau	8
3.1	Patron de conception	9
3.1.1	Procurations de protection	9
3.1.2	Procurations à distance	10
3.2	Intégration au sein du système	10
4	Conclusion	13
5	Annexes	14
5.1	Diagramme de classe complet des parties 1 à 5	14
5.2	Résultats des tests des différentes parties	15
5.2.1	Partie 1	15
5.2.2	Partie 2	15
5.2.3	Partie 3	17
5.2.4	Partie 4	18
5.2.5	Partie 5	19
5.3	Diagramme de classe complet de la partie 6	21

1 Introduction

Le sujet de ce projet est la mise en place d'une simulation de monde composé de divers objets. On fait donc appel à la programmation objet et nous avons choisi le C++ pour l'implémentation. Les objets qui composent ce monde pourront être statiques, comme les végétaux, ou mobiles. Un certain nombre d'interactions seront possibles avec le monde : le déplacement des mobiles, le passage des saisons qui affectera les végétaux, l'agrandissement de ces derniers et enfin, l'affichage du contenu de notre monde.

L'objectif, au-delà de la mise en place de cette simulation, est de se familiariser avec les design patterns et autres bonnes méthodes de développement pour être en mesure de les reproduire dans des cas plus concrets. Cela inclut, bien sûr, la modélisation de notre application par les diagrammes de classes UML.

2 Résultats des TPs

2.1 Présentation générale

On présente ici un diagramme de classe simplifié afin d'obtenir une vision globale. Un diagramme complet contenant l'ensemble des méthodes et des attributs est disponible en annexe.

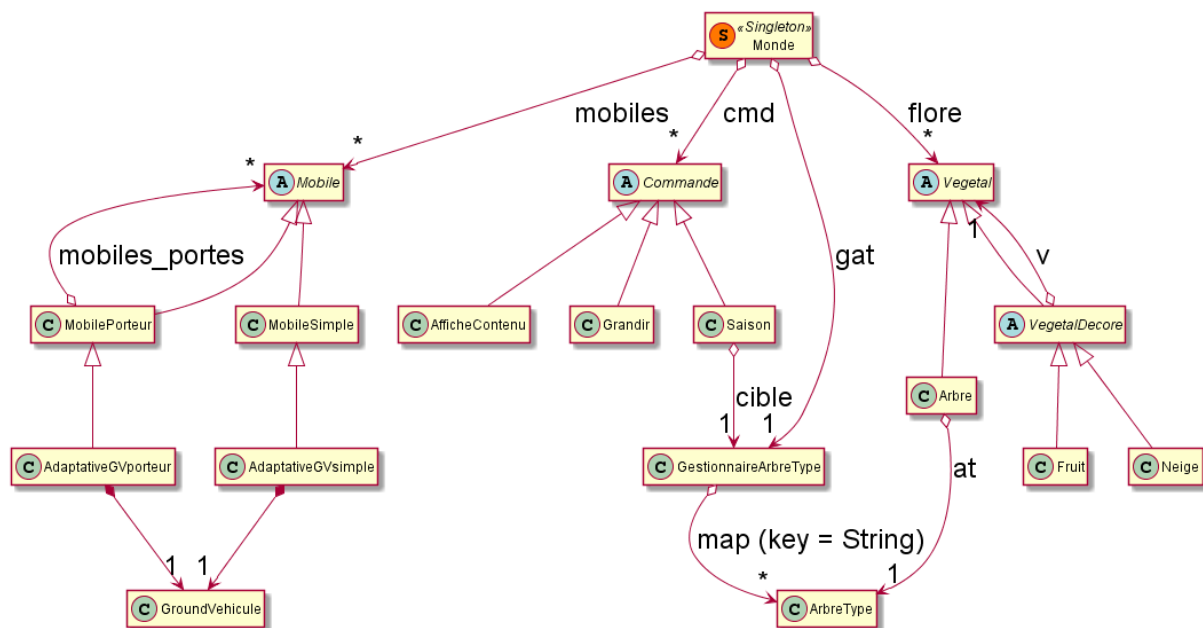


FIGURE 1 – Diagramme de classe simplifié

Une fois le monde créé comme souhaité (avec ses composants végétaux et mobiles), les interactions disponibles s'effectuent de deux façons différentes :

- Les méthodes pour grandir un arbre, pour passer une saison, et pour afficher le contenu du monde utilisent le pattern Commande. On enregistre ces Commandes au niveau du monde et on peut ensuite toutes les exécuter à la suite.
- La méthode de déplacement des mobiles n'est en revanche pas une Commande. On utilise la méthode de déplacement déjà disponible de la classe GroundVehicule et c'est donc le pattern Adaptateur qui permet cette opération.

2.2 Remarques sur chaque partie

Dans cette partie, nous expliquerons les choix que nous avons faits lorsqu'il y avait lieu d'en faire ainsi que les difficultés que nous avons pu rencontrer. Nous ne discuterons pas des résultats obtenus, qui correspondent à chaque partie aux résultats qui étaient attendus en TP. Ces résultats sont, de plus, disponibles en Annexe.

2.2.1 Partie 1

Cette partie ne nous a pas posé de problèmes liés à la conception, qui avait en plus bien été expliquée durant le TP. Il nous a par contre fallu un peu de temps pour nous ré-habituer au langage C++ avec lequel nous avons choisi de développer cette simulation.

Pour enregistrer l'association entre le type d'arbre et son `ArbreType`, nous avons choisi d'utiliser une map. Celle-ci associe à un type d'arbre (une chaîne de caractères) un pointeur vers l'`ArbreType` correspondant.

2.2.2 Partie 2

Nous devons ici faire un choix concernant le calcul des coordonnées d'un mobile porté et sur le lien entre un porté et son porteur. Nous avons choisi de travailler avec des coordonnées relatives. Ainsi, avec un Mobile Vélo en (100, 100), un Mobile X s'affichant en (110, 120) sur le Velo a pour coordonnées (10, 20). De plus, nous avons choisi que le lien entre porteur et porté soit unidirectionnel. Un `MobilePorteur` n'est donc pas accessible depuis ses portés.

Cela signifie que la méthode `dessiner()` d'un `MobilePorteur` appelle la méthode `dessiner()` de ses portés en leur transmettant ses coordonnées et son nom. Les coordonnées transmises sont additionnées aux coordonnées « locales ». Les paramètres par défaut du C++ nous permettent de faire fonctionner cette méthode en toutes circonstances en fixant par défaut les coordonnées reçues à 0.

Méthode dessiner() de MobilePorteur :

```
1 void MobilePorteur::dessiner(double xx, double yy, std::string
  porteur) const
2 {
3     std::cout << "Mobile Porteur : " << this->get_nom() << " ("
      << this->get_x()+xx << ", " << this->get_y()+yy << ")";
4     if(porteur.length() > 0)
5         std::cout << " sur " << porteur;
6     std::cout << std::endl;
7
8     for(size_t i = 0; i < mobiles_portes.size(); i++)
9         mobiles_portes[i]->dessiner(this->get_x()+xx, this->
      get_y()+yy, this->get_nom());
10 }
```

Méthode dessiner() de MobileSimple :

```
1 void MobileSimple::dessiner(double xx, double yy, std::string
  porteur) const
2 {
3     std::cout << "Mobile Simple : " << this->get_nom() << " ("
      << this->get_x()+xx << ", " << this->get_y()+yy << ")";
4     if(porteur.length() > 0)
5         std::cout << " sur " << porteur;
6     std::cout << std::endl;
7 }
```

Même si nous n'en avons pas eu besoin en TP, le fait de ne pas pouvoir remonter à son porteur pourrait être un inconvénient. Si cela venait à être le cas, nous ajouterions à la classe Mobile un attribut MobilePorteur* porteur. Par exemple, le calcul de la position absolue d'un Mobile doit nécessairement passer par les porteurs successifs depuis le monde.

Nous avons également développé des constructeurs par copie de MobileSimple vers MobilePorteur et réciproquement afin de faciliter le passage de l'un à l'autre. On fait néanmoins attention à ne pas permettre la transformation de MobilePorteur vers MobileSimple s'il porte des éléments.

2.2.3 Partie 3

Telle que nous l'avons implémenté, la Commande Saison a un pointeur « cible » vers un GestionnaireArbreType. Or, celui-ci est accessible via la classe Monde, elle-même accessible par tous en tant que singleton. Ce lien entre la Commande Saison et le GestionnaireArbreType n'est donc pas indispensable.

2.2.4 Partie 4

Nous n'avons eu aucun souci ici pour mettre en place le design pattern Adaptateur et utiliser la classe GroundVehicle. On peut toutefois noter qu'avec notre implémentation, chaque Mobile (via l'adaptateur) est associé à un GroundVehicle par un lien de composition. Or, on pourrait tout à fait instancier un unique GroundVehicle lié aux mobiles par un lien d'agrégation. En effet, la méthode déplacer() s'assure de placer le GroundVehicle sur les coordonnées du Mobile avant d'effectuer le déplacement. Cette procédure est donc tout à fait compatible au passage à un unique GroundVehicle pour la gestion des déplacements de tous les Mobile.

Méthode déplacer() de la classe AdaptativeGVSimple :

```
1 void AdaptativeGVsimple::deplacer(double angleDeg)
2 {
3     this->GV.set_x(this->get_x());
4     this->GV.set_y(this->get_y());
5
6     this->GV.move(angleDeg*M_PI/180);
7
8     this->set_x(GV.get_x());
9     this->set_y(GV.get_y());
10 }
```

2.2.5 Partie 5

Nous avons une petite difficulté au niveau de l'affichage de Fruit et de Neige. En effet, ces derniers doivent afficher le type de l'objet qu'ils décorent (des arbres en l'occurrence). Or, ils ne contiennent qu'un pointeur vers la classe abstraite Vegetal, et il est impossible de remonter aux classes dérivées pour afficher le nom de la classe réellement instanciée.

Pour palier à ce problème, nous avons défini une méthode virtuelle `dessiner_simple()` dans la classe `Vegetal`. Cette méthode est (et doit être) implémentée par chacune des classes dérivées finales de `Vegetal` (dans notre cas : `Arbre`, `Fruit` et `Neige`). Cette méthode se contentera d'afficher le nom de l'objet ainsi que sa position. On appelle donc désormais la méthode `dessiner_simple()` depuis la méthode `dessiner()` des dérivées de `VegetalDecore` (`Fruit` et `Neige`). Cette méthode est appelée sur la cible qui est un `Vegetal`, la méthode est virtuelle à ce niveau, elle va donc remonter aux classes dérivées, chacune d'entre elles ayant redéfini la méthode pour afficher son type.

Méthode `dessiner()` de `Fruit` faisant appel à la méthode `dessiner_simple()` de `Vegetal` :

```
1 void Fruit::dessiner() const
2 {
3     this->v->dessiner();
4     std::cout << "+ Ajout du decor FRUITS sur ";
5     this->v->dessiner_simple();
6     std::cout << std::endl;
7 }
```

Méthode `dessiner_simple()` de la classe `Arbre` :

```
1 void Arbre::dessiner_simple() const
2 {
3     std::cout << "arbre (" << this->get_x() << ", " << this->
        get_y() << ")" << std::endl;
4 }
```

3 Partie 6 - Le partage de Monde en réseau

On détaillera, dans cette partie, les choix de modélisations effectués en réponse à la connexion de classes `Monde` en réseau. On présentera d'abord le patron de conception choisi, puis nous décrirons comment nous l'avons intégré à notre système, notamment au travers d'un schéma UML.

3.1 Patron de conception

Parmi les différents design patterns auxquels nous avons pensé, un s'est clairement distingué : le proxy.

Dans sa forme la plus générale, le proxy est une classe fonctionnant comme une interface vers autre chose. Toute interaction avec un objet se fera de façon indirecte via le proxy. Le passage par cet intermédiaire est totalement transparent pour l'utilisateur, car les classes du proxy et de l'objet réel vont implémenter la même classe abstraite qui sert d'interface.

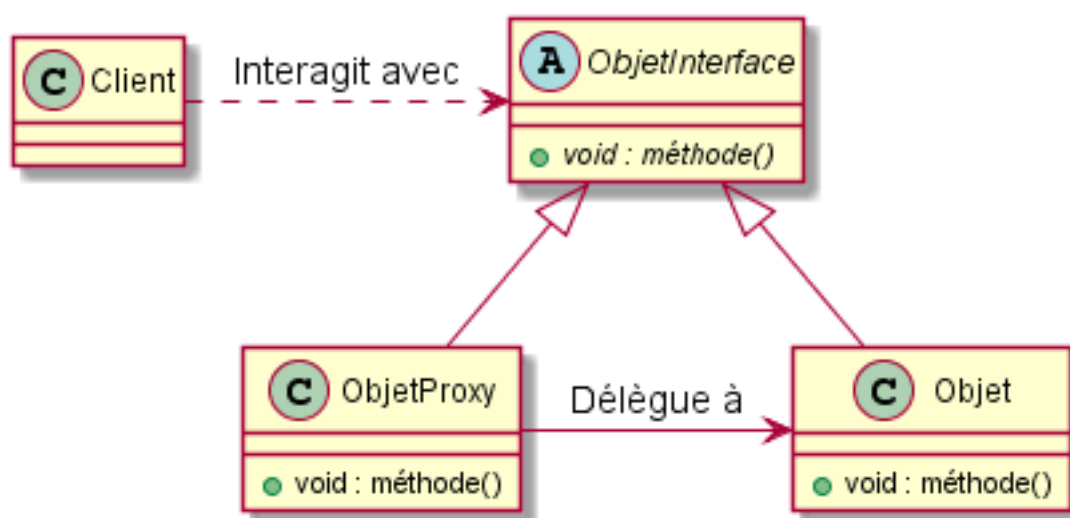


FIGURE 2 – Schéma (diagramme de classe) du patron de conception « proxy »

Comme on le voit sur ce schéma, les clients (utilisateurs) vont uniquement interagir via une classe *ObjetInterface* dont héritent les deux classes *ObjetProxy* et *Objet*. Les appels à l'interface sont redirigés vers la classe *ObjetProxy*, dont tous les objets du client local comme des clients distants ont une instance. Si les droits d'accès sont confirmés, les modifications sont acceptées et, si besoin, redirigées vers l'objet réel (classe *Objet*).

Ce pattern répond extrêmement bien à notre problématique car il va permettre de résoudre deux importants problèmes liés au partage du Monde :

3.1.1 Procurations de protection

Tout d'abord, il permet de contrôler l'accès aux méthodes de la classe substituée. Les appels des méthodes de *ObjetInterface* sont redirigés vers l'*ObjetProxy* correspondant. Si le

client est l'utilisateur local, on lui autorise l'accès aux objets ayant une instance Objet, qui sont les objets locaux. Les modifications sont alors acceptées sur l'ObjetProxy et transmises à la classe Objet. En revanche, l'utilisateur local se verra refuser l'accès aux ObjetProxy non instanciés comme Objet en local. Seul le serveur, en transmettant les informations des utilisateurs distants, pourra modifier ces ObjetsProxy qui représentent les Objets des autres utilisateurs. Enfin, les utilisateurs distants, à l'inverse de l'utilisateur local, ne pourront modifier que les ObjetProxy non instanciés comme Objet en local.

3.1.2 Procurations à distance

De plus, il permettra à notre programme de communiquer à travers le réseau sans que cela soit visible pour l'utilisateur. Ainsi, s'il appelle une méthode nécessitant des informations que seul le serveur ou un autre utilisateur possède, cette requête pourra être transmise à ce dernier par la classe proxy et ainsi permettre la récupération de cette information sans que l'utilisateur n'en ait conscience.

3.2 Intégration au sein du système

Dans notre cas, le design pattern proxy sera mis en place 2 fois. En effet, les classes Vegetal et Mobile, qui sont à la base de tous les objets instanciés dans le monde, auront chacune leur proxy. Ainsi, les classes VegetalProxy et Vegetal hériteront toutes les deux d'une même classe abstraite VegetalInterface. Il en va de même pour les Mobile, avec les classes MobileProxy et MobileInterface. Les utilisateurs n'interagiront qu'avec les classes VegetalInterface et MobileInterface.

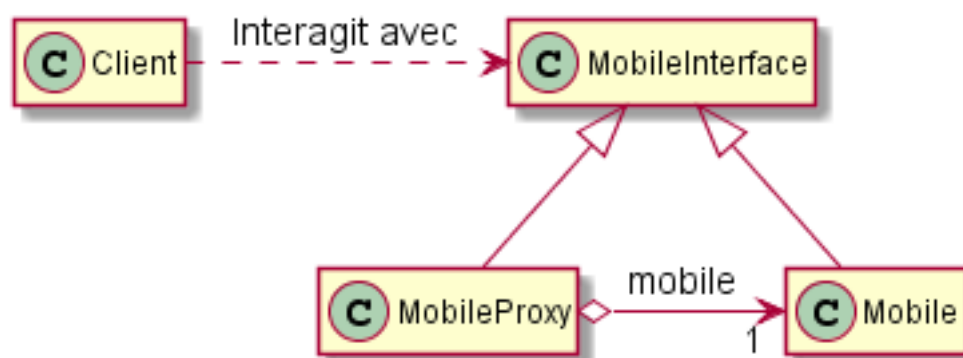


FIGURE 3 – Le pattern proxy adapté pour la classe Mobile

Les méthodes des classes proxy feront appel aux méthodes correspondantes dans les classes « réelles » (lorsque les droits sont confirmés). Elles effectueront toutefois des actions supplémentaires : d'abord un contrôle du droit d'accès, et ensuite, le cas échéant, une communication de mise à jour vers le serveur. Par exemple, après l'appel à la méthode `deplacer()` de la classe réelle, la méthode `deplacer()` du proxy devra communiquer avec le serveur pour transmettre les nouvelles coordonnées aux autres clients.

La transmission des mises à jour de l'état des objets au serveur est une action récurrente qui doit être effectuée par toutes les méthodes qui modifient l'état d'un objet. Nous avons peu de méthodes concernées ici mais en prévision de l'intégration à un cas plus concret, on peut implémenter une méthode `update()`. Cette méthode serait chargée de la transmission de l'état de l'objet au serveur et serait appelée par exemple à la fin de la méthode `deplacer()`. De la même manière, les contrôles de droits d'accès seront récurrents et on mettra en place une méthode dédiée.

On aura donc besoin d'un identifiant unique pour distinguer les utilisateurs et vérifier leurs droits d'accès à un objet. Cet ID utilisateur sera stocké comme attribut du Monde et sera initialisé par le serveur lorsque le Monde se connecte en réseau. Ainsi, on pourra s'assurer que l'identifiant est unique. De plus, la classe `ObjetInterface` aura elle aussi un attribut ID correspondant à celui de son Monde propriétaire.

Enfin, les `ObjetProxy` auront un pointeur vers leur équivalent réel. Ce pointeur indiquera la cible vers laquelle rediriger les appels de méthodes et sera initialisé à `NULL` (`nullptr`) pour les objets d'utilisateurs distants.

Pour finaliser la mise en place du pattern, on pensera à supprimer les méthodes d'accès aux listes des objets réels du Monde pour les remplacer par des accès aux listes d'`ObjetInterface`. En effet, toutes les interactions avec les objets réels (`Mobile` et `Vegetal`) doivent maintenant passer par les contrôles mis en place dans les classes Proxy puis, si autorisées, être redirigées vers la cible réelle.

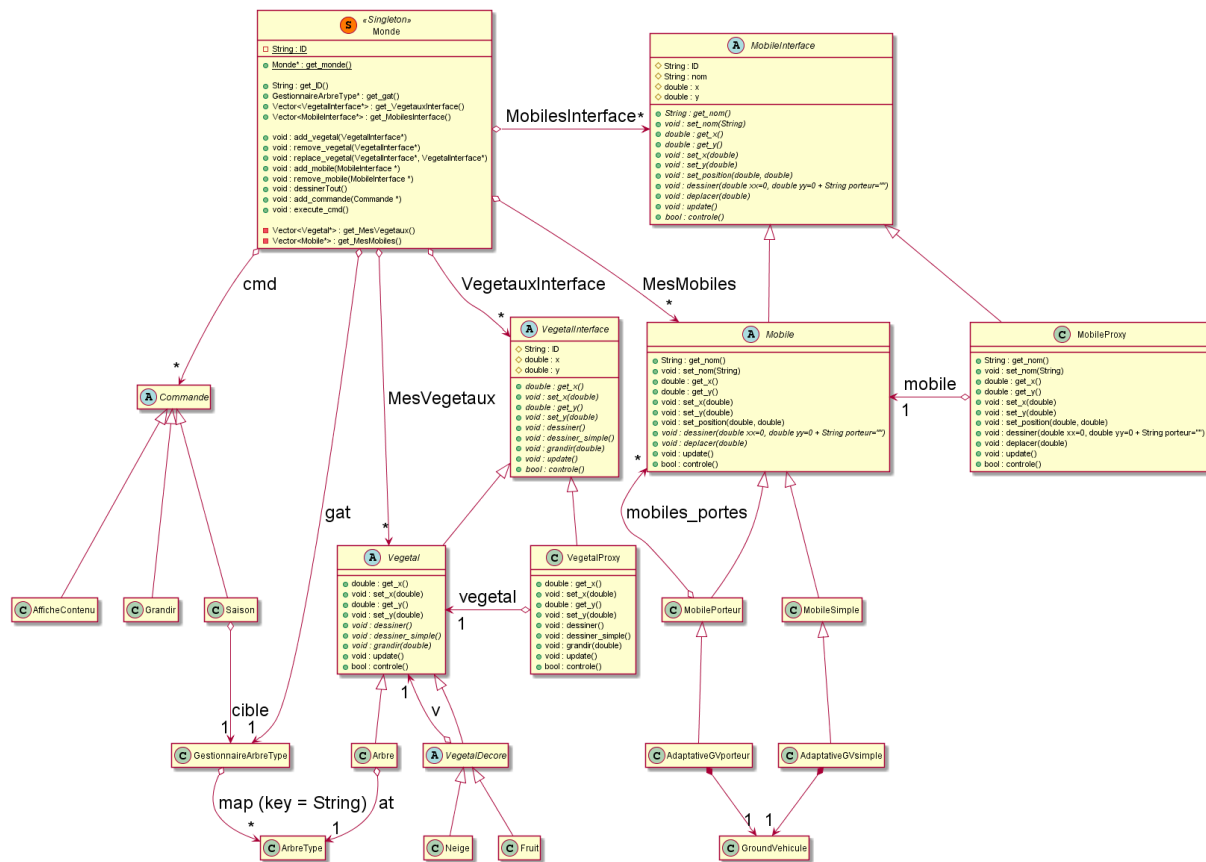


FIGURE 4 – Diagramme de classe simplifié de la partie 6

4 Conclusion

Le travail fourni permet le fonctionnement de ce simulateur. Néanmoins, comme il l'est précisé dans l'introduction, l'objectif principal n'est pas le développement mais de s'interroger sur la modélisation de notre système. En cela, cet enseignement nous offre de bonnes pratiques de programmation.

En effet, les motifs de conceptions nous permettent de répondre facilement et efficacement à des problèmes courant rencontrés lors du développement. Ils offrent également une communication plus facile entre développeurs puisqu'ils permettent d'échanger l'architecture globale d'un projet sans avoir à rentrer dans les détails de chaque classe.

De plus, il nous a fait prendre conscience de l'importance de la modélisation et de lui accorder du temps. Elle permet non seulement de réfléchir à l'architecture globale afin de la perfectionner, mais surtout, elle évite de se retrouver bloquer en cours de développement en raison d'un mauvais choix d'architecture. Enfin, tout comme les motifs de conception, elle permet une meilleure collaboration entre développeurs puisqu'elle offre, généralement au travers de diagrammes UML, une vue globale du projet.

5 Annexes

5.1 Diagramme de classe complet des parties 1 à 5

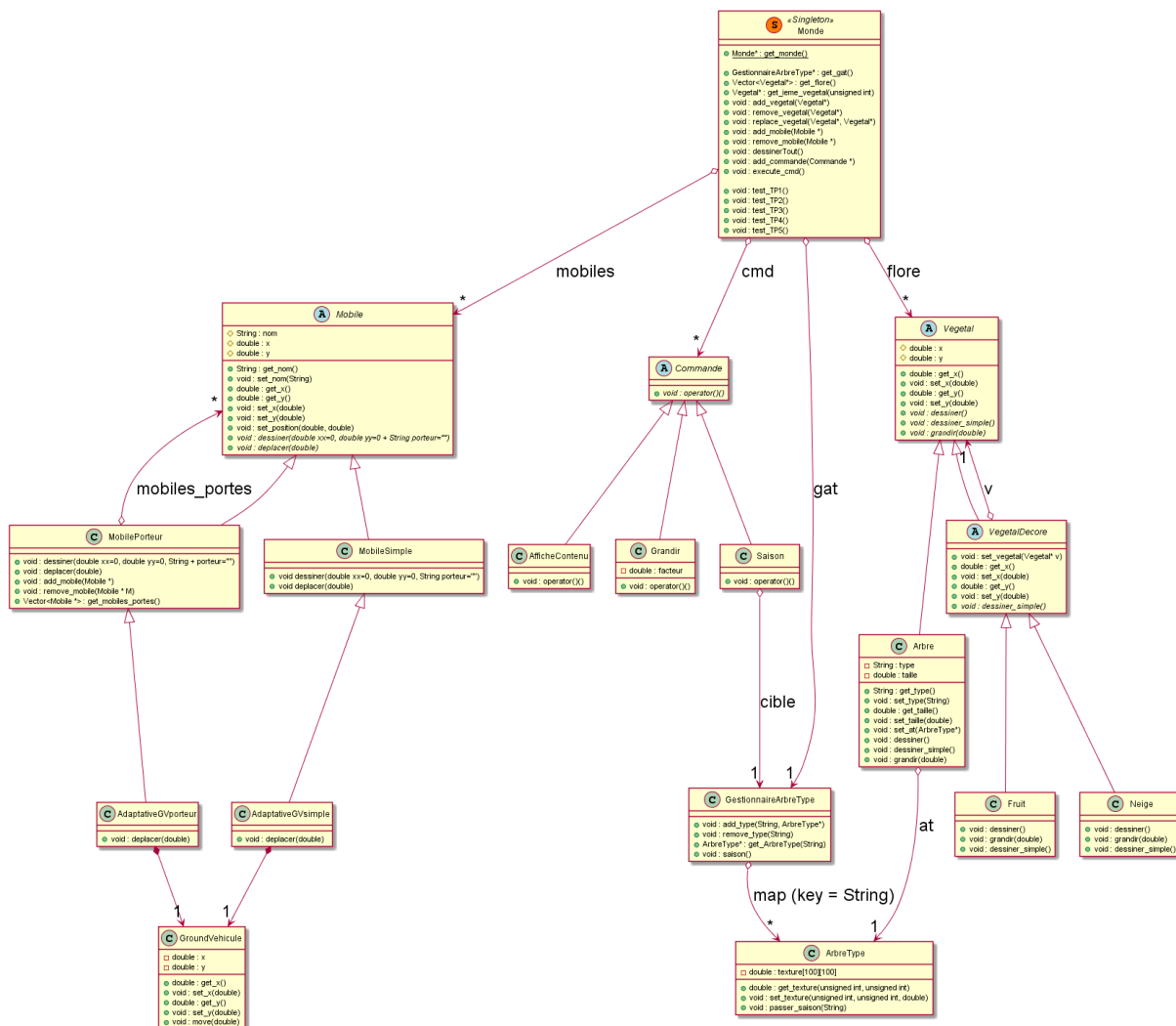


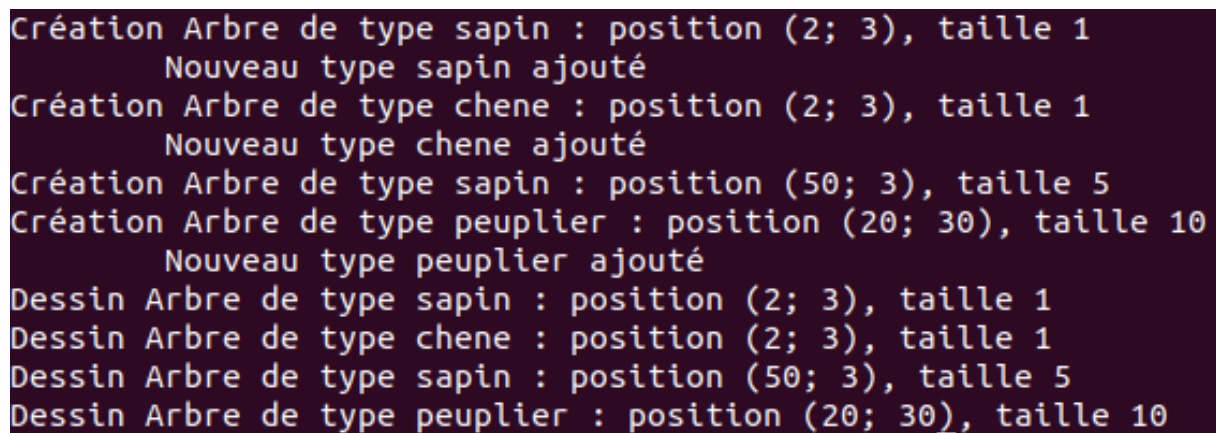
FIGURE 5 – Diagramme de classe des parties 1 à 5

5.2 Résultats des tests des différentes parties

Dans cette section vous trouverez les tests utilisés pour chacune des parties ainsi que les résultats obtenus.

5.2.1 Partie 1

```
1 void Monde::test_TP1()
2 {
3     this->add_vegetal(new Arbre("sapin", 1, 2, 3));
4     this->add_vegetal(new Arbre("chene", 1, 2, 3));
5     this->add_vegetal(new Arbre("sapin", 5, 50, 3));
6     this->add_vegetal(new Arbre("peuplier", 10, 20, 30));
7
8     this->dessinerTout();
9 }
```



```
Création Arbre de type sapin : position (2; 3), taille 1
    Nouveau type sapin ajouté
Création Arbre de type chene : position (2; 3), taille 1
    Nouveau type chene ajouté
Création Arbre de type sapin : position (50; 3), taille 5
Création Arbre de type peuplier : position (20; 30), taille 10
    Nouveau type peuplier ajouté
Dessin Arbre de type sapin : position (2; 3), taille 1
Dessin Arbre de type chene : position (2; 3), taille 1
Dessin Arbre de type sapin : position (50; 3), taille 5
Dessin Arbre de type peuplier : position (20; 30), taille 10
```

FIGURE 6 – Résultat de test_TP1

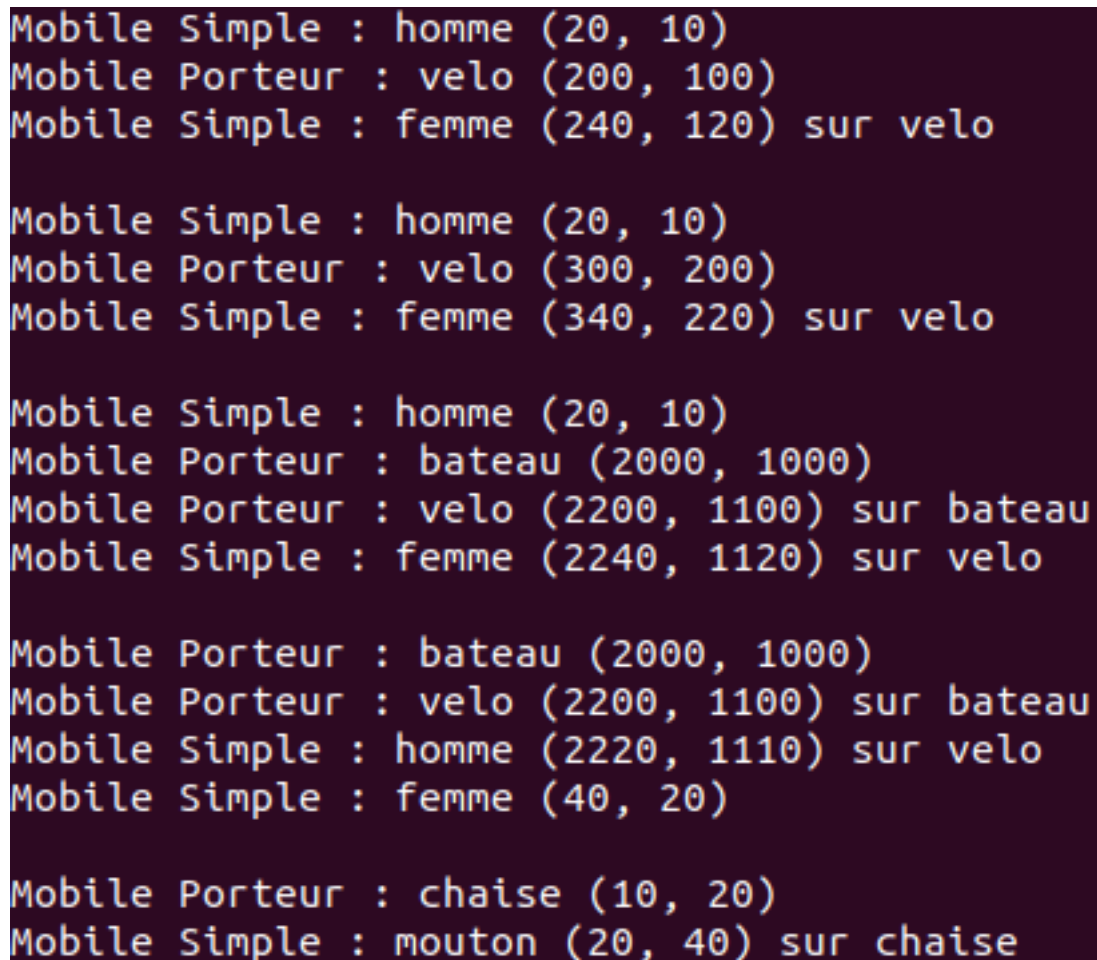
5.2.2 Partie 2

```
1 void Monde::test_TP2()
2 {
3     MobileSimple * homme = new MobileSimple("homme", 20, 10);
4     MobileSimple * femme = new MobileSimple("femme", 40, 20);
```

```
5     MobilePorteur * velo = new MobilePorteur("velo", 200, 100);
6     MobilePorteur * bateau = new MobilePorteur("bateau", 2000,
          1000);
7
8     this->add_mobile(homme);
9     velo->add_mobile(femme);
10    this->add_mobile(velo);
11    this->dessinerTout();
12    std::cout << std::endl;
13
14    velo->set_position(300, 200);
15    this->dessinerTout();
16    std::cout << std::endl;
17
18    velo->set_position(200, 100);
19    this->remove_mobile(velo);
20    this->add_mobile(bateau);
21    bateau->add_mobile(velo);
22    this->dessinerTout();
23    std::cout << std::endl;
24
25    this->remove_mobile(homme);
26    velo->remove_mobile(femme);
27    velo->add_mobile(homme);
28    this->add_mobile(femme);
29    this->dessinerTout();
30    std::cout << std::endl;
31
32    MobileSimple * chaise = new MobileSimple("chaise");
33    MobilePorteur * mouton = new MobilePorteur("mouton");
34
35    MobilePorteur * MP = new MobilePorteur(*chaise);
36    MobileSimple * MS = new MobileSimple(*mouton);
37
38    MP->add_mobile(MS);
39    MS->set_position(10, 20);
40    MP->set_position(10, 20);
```



```
41     MP->dessiner();  
42 }
```



```
Mobile Simple : homme (20, 10)  
Mobile Porteur : velo (200, 100)  
Mobile Simple : femme (240, 120) sur velo  
  
Mobile Simple : homme (20, 10)  
Mobile Porteur : velo (300, 200)  
Mobile Simple : femme (340, 220) sur velo  
  
Mobile Simple : homme (20, 10)  
Mobile Porteur : bateau (2000, 1000)  
Mobile Porteur : velo (2200, 1100) sur bateau  
Mobile Simple : femme (2240, 1120) sur velo  
  
Mobile Porteur : bateau (2000, 1000)  
Mobile Porteur : velo (2200, 1100) sur bateau  
Mobile Simple : homme (2220, 1110) sur velo  
Mobile Simple : femme (40, 20)  
  
Mobile Porteur : chaise (10, 20)  
Mobile Simple : mouton (20, 40) sur chaise
```

FIGURE 7 – Résultat de test_TP2

5.2.3 Partie 3

```
1 void Monde::test_TP3()  
2 {  
3     this->add_vegetal(new Arbre("sapin", 1, 2, 3));  
4     this->add_vegetal(new Arbre("chene", 10, 20, 30));  
5     this->add_vegetal(new Arbre("sapin", 15, 15, 15));  
6 }
```

```
7     this->add_commande(new AfficheContenu());
8
9     this->add_commande(new Grandir(2.0));
10    this->add_commande(new Saison(this->gat));
11
12    this->add_commande(new AfficheContenu());
13
14    this->execute_cmd();
15 }
```

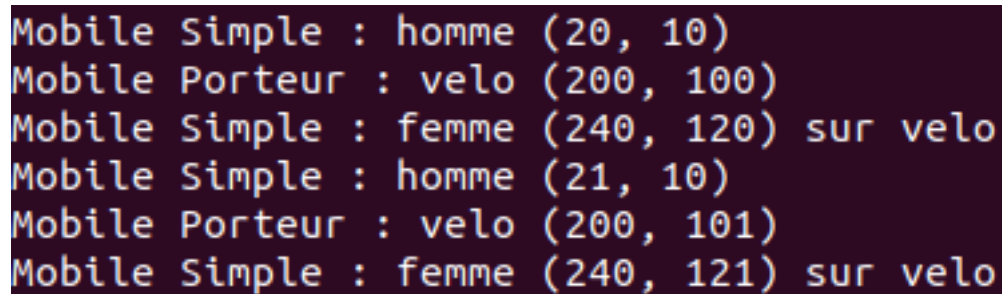
```
Création Arbre de type sapin : position (2; 3), taille 1
Nouveau type sapin ajouté
Création Arbre de type chene : position (20; 30), taille 10
Nouveau type chene ajouté
Création Arbre de type sapin : position (15; 15), taille 15
Dessin Arbre de type sapin : position (2; 3), taille 1
Dessin Arbre de type chene : position (20; 30), taille 10
Dessin Arbre de type sapin : position (15; 15), taille 15
Les arbres de type chene passent la saison.
Les arbres de type sapin passent la saison.
Dessin Arbre de type sapin : position (2; 3), taille 2
Dessin Arbre de type chene : position (20; 30), taille 20
Dessin Arbre de type sapin : position (15; 15), taille 30
```

FIGURE 8 – Résultat de test_TP3

5.2.4 Partie 4

```
1 void Monde::test_TP4()
2 {
3     MobileSimple * homme = new AdaptativeGVsimple("homme", 20,
4         10);
5     MobileSimple * femme = new MobileSimple("femme", 40, 20);
6     MobilePorteur * velo = new AdaptativeGVporteur("velo", 200,
7         100);
8
9     this->add_mobile(homme);
```

```
8     velo->add_mobile(femme);
9     this->add_mobile(velo);
10    this->dessinerTout();
11
12    homme->deplacer(0);
13    velo->deplacer(90);
14    this->dessinerTout();
15 }
```



Mobile Simple : homme (20, 10)
Mobile Porteur : velo (200, 100)
Mobile Simple : femme (240, 120) sur velo
Mobile Simple : homme (21, 10)
Mobile Porteur : velo (200, 101)
Mobile Simple : femme (240, 121) sur velo

FIGURE 9 – Résultat de test_TP4

5.2.5 Partie 5

```
1 void Monde::test_TP5()
2 {
3     Arbre* pommier1 = new Arbre("pommier", 50, 1000, 500);
4     this->add_vegetal(pommier1);
5     Arbre* pommier2 = new Arbre("pommier", 20, 2000, 1500);
6     this->add_vegetal(pommier2);
7     Arbre* peuplier = new Arbre("peuplier", 5, 3, 3);
8     this->add_vegetal(peuplier);
9
10    this->dessinerTout();
11
12    std::cout << "\n[Ajout décors]\n" << std::endl;
13
14    Fruit* pomf = new Fruit(pommier1);
15    this->replace_vegetal(pommier1, pomf);
16 }
```

```
17     Fruit* poirf = new Fruit(pommier2);
18     this->replace_vegetal(pommier2, poirf);
19
20     Neige* peun = new Neige(peuplier);
21     this->replace_vegetal(peuplier, peun);
22
23     this->dessinerTout();
24 }
```

```
Création Arbre de type pommier : position (1000; 500), taille 50
    Nouveau type pommier ajouté
Création Arbre de type pommier : position (2000; 1500), taille 20
Création Arbre de type peuplier : position (3; 3), taille 5
    Nouveau type peuplier ajouté
Dessin Arbre de type pommier : position (1000; 500), taille 50
Dessin Arbre de type pommier : position (2000; 1500), taille 20
Dessin Arbre de type peuplier : position (3; 3), taille 5

[Ajout décors]

Dessin Arbre de type pommier : position (1000; 500), taille 50
+ Ajout du decor FRUITS sur arbre (1000, 500)

Dessin Arbre de type pommier : position (2000; 1500), taille 20
+ Ajout du decor FRUITS sur arbre (2000, 1500)

+ Ajout du decor NEIGE sur arbre (3, 3)
```

FIGURE 10 – Résultat de test_TP5

5.3 Diagramme de classe complet de la partie 6

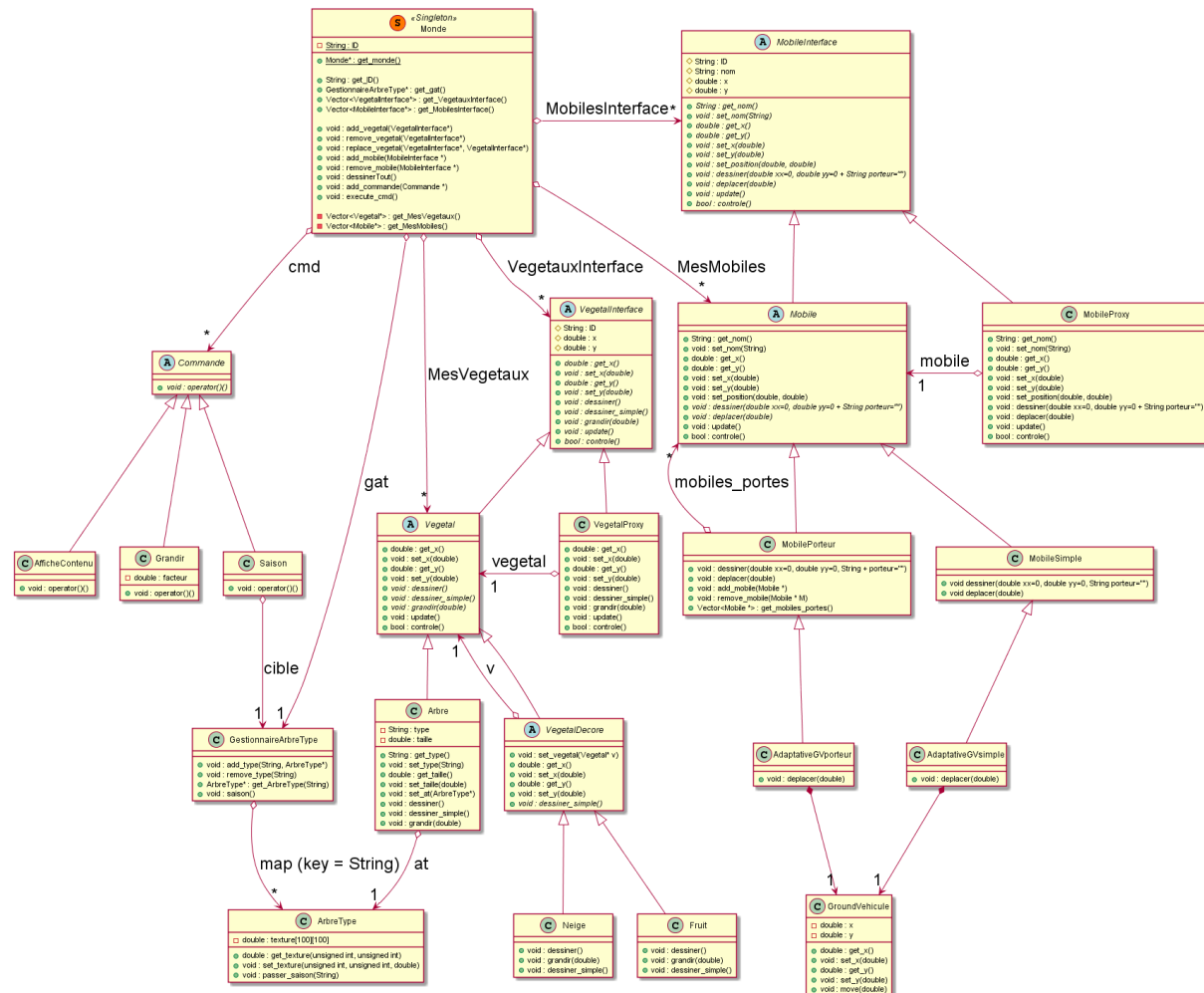


FIGURE 11 – Diagramme de classe de la partie 6