­

**Département de génie logiciel et des TI**

Rapport de laboratoire

|  |  |
| --- | --- |
| **No de laboratoire** | Laboratoire 1 |
| **Étudiant** | Martin Grégoire |
| **Code permanent** | GREM31019408 |
| **Cours** | LOG121 |
| **Session** | A17 |
| **Groupe** | 01 |
| **Professeur** | Vincent Lacasse |
| **Chargé de laboratoire** | Dany Boisvert & Samir Djeffal |
| **Date de remise** | 23 Octobre 2017 |

Table of Contents

[1.Introduction 3](#_Toc496548774)

[2. Conception 4](#_Toc496548775)

[2.1 Choix et responsabilités des classes 4](#_Toc496548776)

[2.2 Diagrammes des classes 7](#_Toc496548777)

[2.3 Faiblesses de la conception 8](#_Toc496548778)

[2.4 Diagramme de séquence (uml) 9](#_Toc496548779)

[2.4.1. Exemple qui illustre la dynamique du patron observateur 9](#_Toc496548780)

[2.4.2. Production d’une composante 9](#_Toc496548781)

[3 Décisions de conception / d'implémentation 10](#_Toc496548782)

[3.1 Décision 1 : Choix de l’architecture 10](#_Toc496548783)

[3.2 Décision 2 : Implémentation de la classe Production 11](#_Toc496548784)

[4 Conclusion 12](#_Toc496548785)

[5 Références 12](#_Toc496548786)

[Annexes 13](#_Toc496548787)

[Annexe A Diagrammes des Classes du Programme 13](#_Toc496548788)

[Annexe B : Diagrammes de décisions 17](#_Toc496548789)

[Annexe C : Exemples de Code 19](#_Toc496548790)

[Annexe D : Diagrammes de Séquence 20](#_Toc496548791)

# 1.Introduction

Le but de ce laboratoire est de créer un logiciel qui simule une chaine de production. L’objectif de ce laboratoire est d’implémenter le patron Observateur, le patron Stratégie et d’utiliser le polymorphisme. La solution satisfait parfaitement les objectifs. Ce rapport est présenté en deux sections : La Conception du programme et deux exemples de décisions par rapport à la conception et l’implémentation du programme.

# 2. Conception

## 2.1 Choix et responsabilités des classes

|  |  |  |
| --- | --- | --- |
| * Classe | * Responsabilités | * Dépendances |
| * Environnement | * Classe Maitre, * S’occupe de la logique d’un tour, * Interroge les usines pour produire * Fait bouger les composantes dans le modèle * Fait la vente selon la stratégie attribuée | * Usine * IVenteStrategie * XMLHelper |
| * Simulation | * Initialise le programme * Crée les batiments | * FenetrePrincipale * BatimentMetaData * Environnement |
| * BatimentMetaData | * Transfère les informations importantes vers la Vue et les Controlleurs | * Batiment |
| * Unite | * Définit la position et le numéro d’identification d’une unité (Bâtiment ou composante) * Définit la destination de l’unité. | * Observable |
| * Batiment | * Gère l’inventaire * Gère la production | * Unite |
| * Composante | * Définit une vitesse de mouvement vers le bâtiment de destination. * Contient le type de composante | * Unite |
| * IVenteStrategie | * Interface qui permet de définir des méthodes de ventes différentes. Deux ventes sont définies : VenteAléatoire et VenteIntervalles |  |
| * Usine | * Définit la quantité de composantes en entrée nécessaire pour continuer la production. Définit aussi la nombre de tours nécessaire pour produire une composante. | * Batiment * Composante * Chemin |
| * UsineMatière | * Interroge l’entrepôt afin d’accélérer ou ralentir la production | * Observer * Entrepôt |
| * Entrepôt | * Gère la capacité * Averti ses observateurs selon son état (Vide, presque plein, plein) | * Observable |
| * FenêtrePrincipale | * Fenêtre principale de la Simulation. Elle contient des menus permettant de contrôler la simulation. * Initialise PanneauPrincipal | * MenuFenetre * PanneauPrincipal |
| * PanneauPrincipal | * Fait le dessin de la simulation | * Observer * IconeComposante * IconeBatimen |
| * XMLHelper | * Code réutilisable pour la lecture d’un fichier XML | * Fonctions IO de Java |
| * XMLSImulationParser | * S’occupe de lire le fichier xml et d’envoyer les données à la classe Simulation | * Simulation * XMLHelper * BatimentMetaBata |

## 2.2 Diagrammes des classes

Voir l’Annexe A pour les diagrammes des classes du logiciel.

## 2.3 Faiblesses de la conception

La décision de séparer les types d’usines par les composantes qu’elles produisent est une mauvaise idée. Il n’y a pas vraiment de différence entre le fonctionnement de l’usine d’ailes et l’usine de moteurs. De plus, cette façon de séparer les classes empêche d’ajouter de nouveaux types d’usines avec le fichier xml. Pour régler ce problème, il aurait fallu séparer les usines par le fonctionnement. Par exemple, avoir une classe pour une usine qui produit sans avoir besoin de composantes en entrée et une classe qui produit en prenant des composantes en entrées. Pour assigner le type de composantes en sortie et en entrée, il suffit d’assigner une variable à la classe qui garde en mémoire le type produit et les types requis en entrée.

Lors de l’avancement d’un tour de chaque usine, je regarde si sa production est arrêtée et si elle peut produire. Cela résulte en un code lourd rempli de booléens. Alternativement, il aurait fallu utiliser le patron State. Le patron State peut déterminer qu’elles actions sont possibles dans certaines conditions, ce qui aurait allégé le code.

## 2.4 Diagramme de séquence (uml)

### 2.4.1. Exemple qui illustre la dynamique du patron observateur

Voir le diagramme de séquence à l’annexe D : Diagramme de Séquence 1 Observation de l'entrepôt

Lorsque l’entrepôt doit vendre ou lorsqu’un avion est ajouté à son inventaire, il avertit ses observateurs. Ses observateurs sont les usines matières de la chaîne de production. Lorsque la méthode update des instances de UsineMatiere est appelée, les instances interrogent l’entrepôt afin de connaître le statut de l’inventaire. L’entrepôt interroge ensuite la classe Production pour connaître son inventaire et retourne une réponse à l’usine sous forme de pourcentage. Selon le statut, les usines vont ajuster leur cadence de production et arrêter complétement la production si nécessaire.

### 2.4.2. Production d’une composante

Voir le diagramme de séquence à l’annexe D : Diagramme de Séquence 2 Extraction d'une Composante

Lorsqu’une usine reçoit une requête de production de composante, elle regarde d’abord si elle peut produire en vérifiant si elle a suffisamment de composantes en entrées et si c’est le bon tour pour produire. Si c’est le cas, elle ajuste son inventaire et retourne une nouvelle composante qu’elle a créée. Sinon, elle retourne null.

# 3 Décisions de conception / d'implémentation

## 3.1 Décision 1 : Choix de l’architecture

**Contexte**: Il existe plusieurs façons d’écrire un logiciel. Plusieurs architectures existent qui permettent d’organiser les classes d’un programme de manières différentes. Deux solutions sont envisageables ici : l’architecture MVC (Modèle-Vue-Contrôleur) ou simplement aucune architecture.

**Aucune architecture:** Il n’est pas nécessaire de suivre une architecture spéciale pour réaliser ce programme. Le point fort de cette solution est la quantité peu élevée de classes nécessaires, car les classes contiennent plus d’information qui peuvent servir dans plusieurs contextes. Par exemple, se référer au Diagramme de Classe 5. Dans cette exemple, l’usine contient une icône permettant de la représenter. Le panneau principal peut donc interroger directement la classe usine pour la dessine. Cependant, une telle architecture introduit beaucoup de couplage et il devient donc difficile de changer certaines parties. Par exemple, si la logique d’affichage a besoin d’un changement considérable, toutes les classes du programme risquent d’être affectées.

**Architecture MVC:** Cette architecture consiste à séparer les classes du logiciel en trois groupes de fonctionnalités différentes : le Modèle, qui contient les données, le Contrôleur, qui effectue la logique et sert au déroulement général du programme et la Vue, qui s’occupe de l’affichage. Une telle séparation diminue grandement le couplage entre les classes. Généralement, les classes du modèle n’ont pas de couplage avec les classes du contrôleur et de la vue, ce qui permet de changer la vue d’un programme facilement. Cependant, il n’est pas facile d’implémenter cette architecture pour les débutants. Si elle n’est pas implémentée correctement, il y aura un grand nombre de couplage et une très mauvaise cohésion dans le programme. Cette architecture est aussi moins performante car plus de code est nécessaire pour atteindre le même résultat que si aucune architecture est implémentée. Se référer au Diagramme de Classe 1 pour un exemple de l’architecture MVC.

**Choix de la solution et justification :** L’architecture MVC est la solution choisie pour cette décision, car elle permet de mieux organiser le code. De plus, puisque l’objectif de ce laboratoire est d’implémenter les patrons Observateur et le polymorphisme, il est mieux d’organiser le code le plus possible afin que l’application des patrons puisse se faire le plus facilement possible.

## 3.2 Décision 2 : Implémentation de la classe Production

**Contexte:** La classe Production contient la logique nécessaire à la gestion de l’inventaire d’un bâtiment. Cette classe peut être implémentée comme classe interne ou comme classe externe.

**Classe Interne :** Implémenter la classe production comme classe interne permet de l’encapsuler. Les autres classes du programme n’ont donc pas connaissance de l’implémentation de la gestion de l’inventaire d’une usine. Elle permet aussi de diminuer le couplage. Cependant, il est plus difficile de gérer l’inventaire car il faut passer par la classe Bâtiment, ce qui peut diminuer la cohésion, comme le montre l’Exemple de Code 1. Se référer au Diagramme de Classe 2 pour un exemple d’implémentation de la classe Production comme classe interne dans la classe Batiment.

**Classe Externe :** Implémenter la classe Production comme classe externe permet aux classes enfantes de Batiment de gérer leur inventaire sans avoir à passer par leur parente. Ça permet une utilisation plus flexible de la classe Production. Cependant, une telle utilisation augmente le couple entre les classes enfantes de la classe Batiment et la classe Production, comme le montre le Diagramme de Classe 6. Tel que montré dans cet exemple, un changement à la classe Production peut entrainer un changement dans la classe Entrepôt, la classe Usine, la classe UsineMatiere et la classe Bâtiment.

**Choix de la solution et justification :** Il est mieux d’implémenter la classe Production comme classe Interne de la classe Batiment, car il n’est pas nécessaire pour les classes enfantes de connaitre le fonctionnement de l’inventaire. Le problème de diminution de la cohésion peut être réglé par des méthodes génériques.

# 4 Conclusion

Pour conclure, l’implémentation des patrons Observateur et Stratégie ainsi que l’implémentation du polymorphisme a été réussie avec succès. Le programme a été organisé sous l’architecture MVC, ce qui a permis de réduire le couplage en les classes et de donner des responsabilités claires à chacune d’elles. Ce programme arrive à faire l’optimisation de la production des usines de manière précise et peut fonctionner pendant des heures sans problème. La conception initiale des classes d’usines et de composantes s’est avérée faible et devrait être changée dans la prochaine version du logiciel. Ce programme devrait permettre d’ajouter n’importe quel type d’usine produisant n’importe quel type de composante dans une version future. Le fichier xml devrait aussi être validé, préférablement à l’aide d’un fichier XSD.

# 5 Références

* Mettez, au besoin, les références

# Annexes

Une copie de chaque figure est disponible dans le dépôt sur GitLab au besoin.

## Annexe A Diagrammes des Classes du Programme

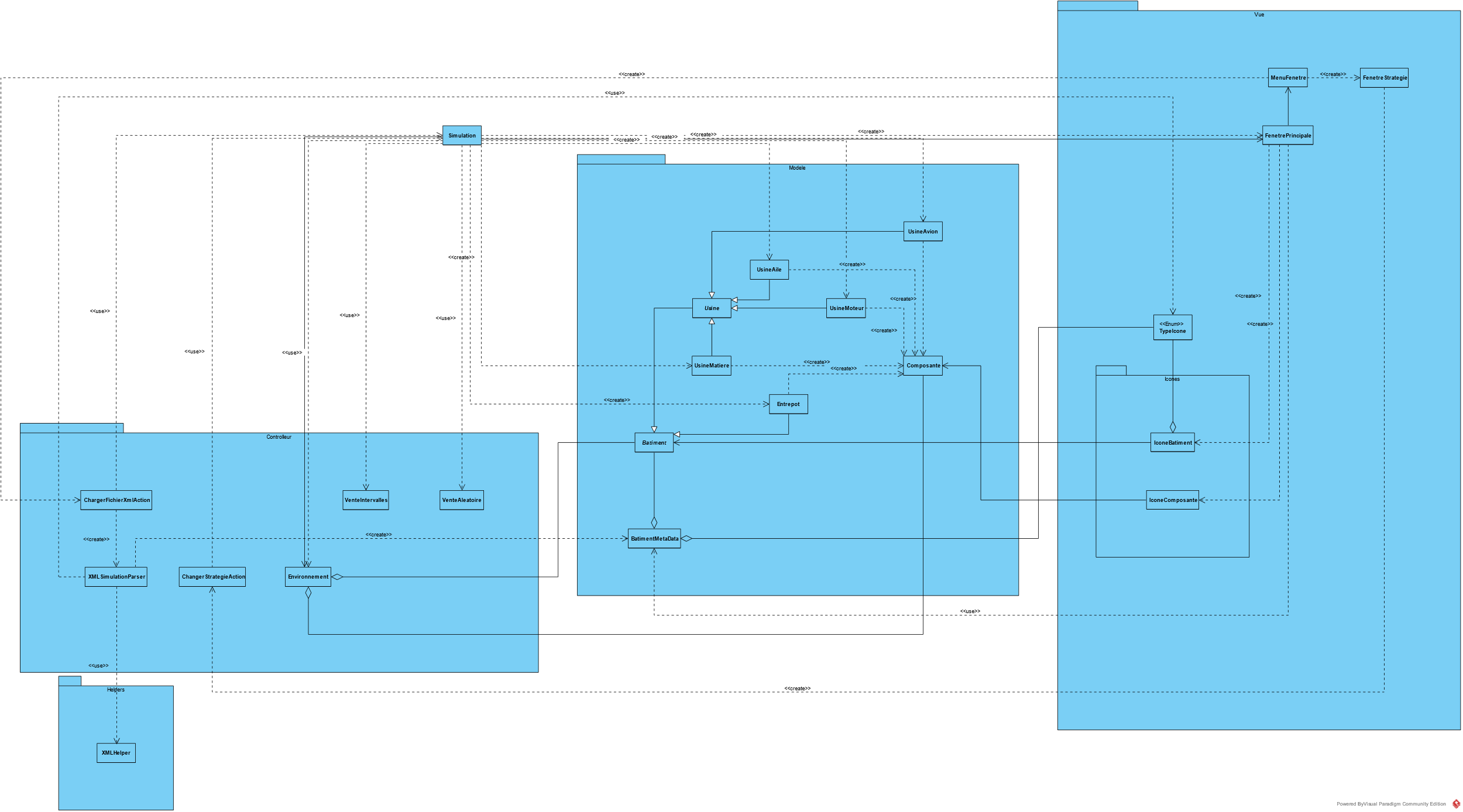


Diagramme de Classe Interactions entre les packages

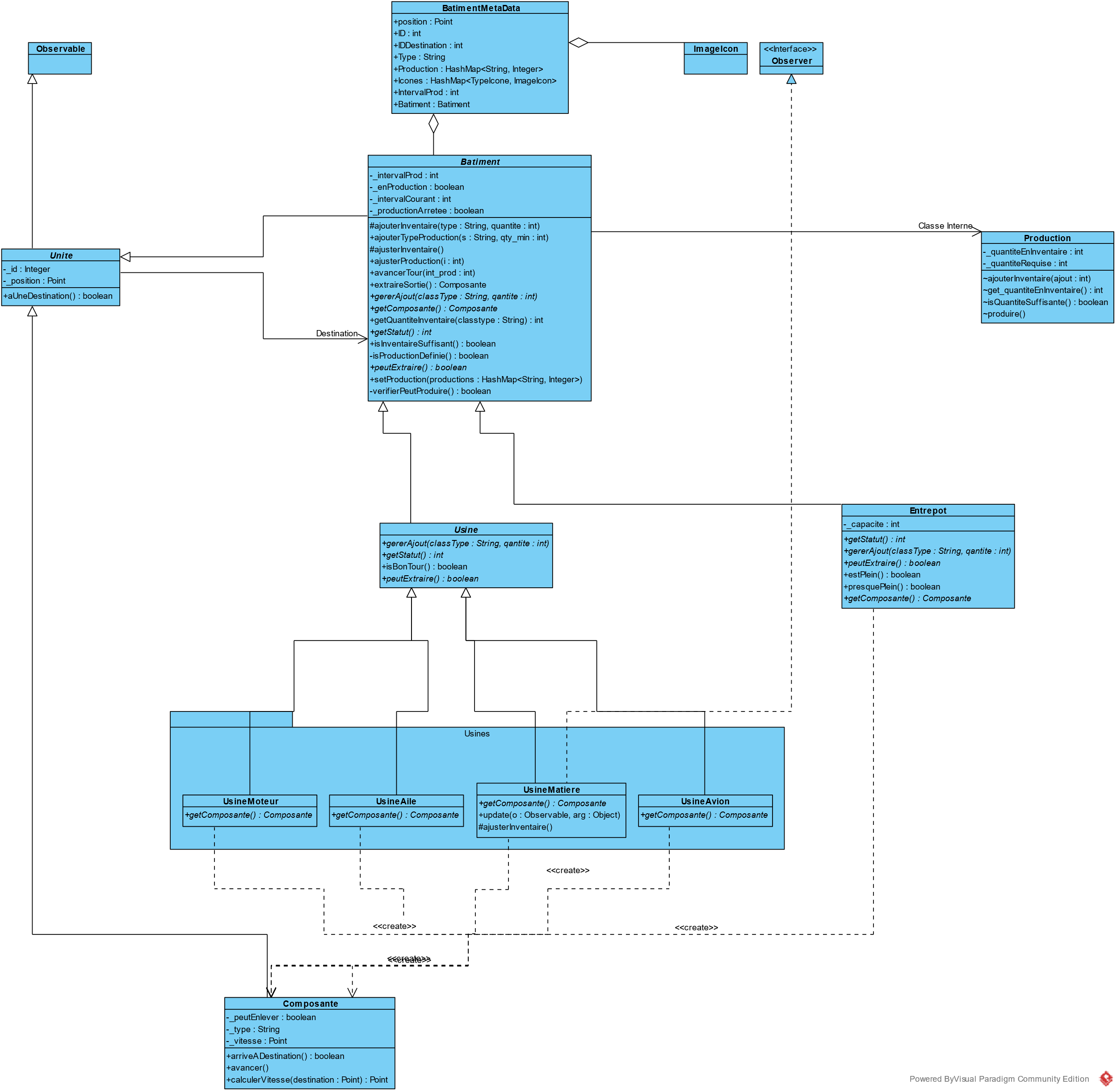


Diagramme de Classe Modèle

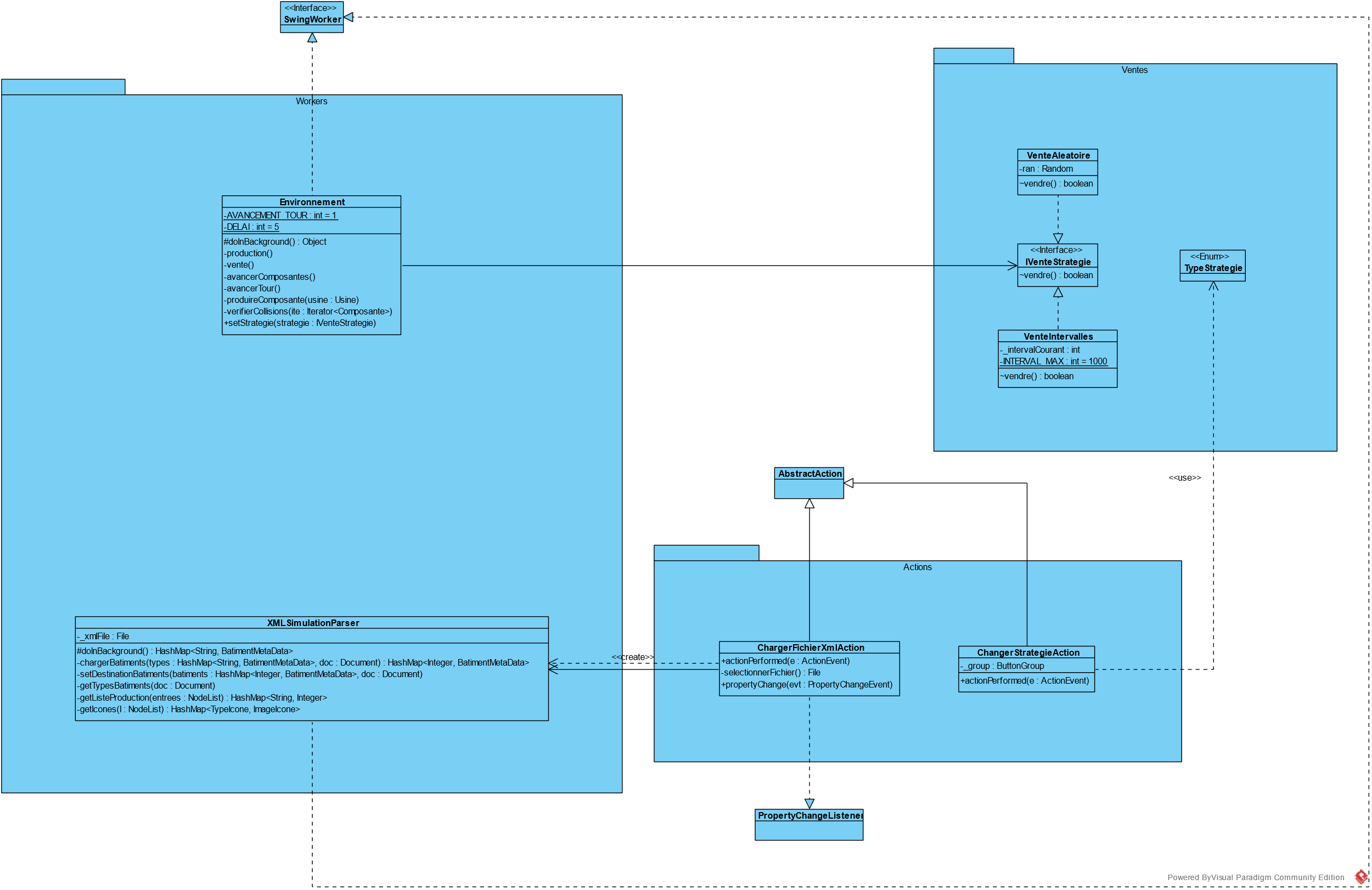


Diagramme de Classe Contrôleur

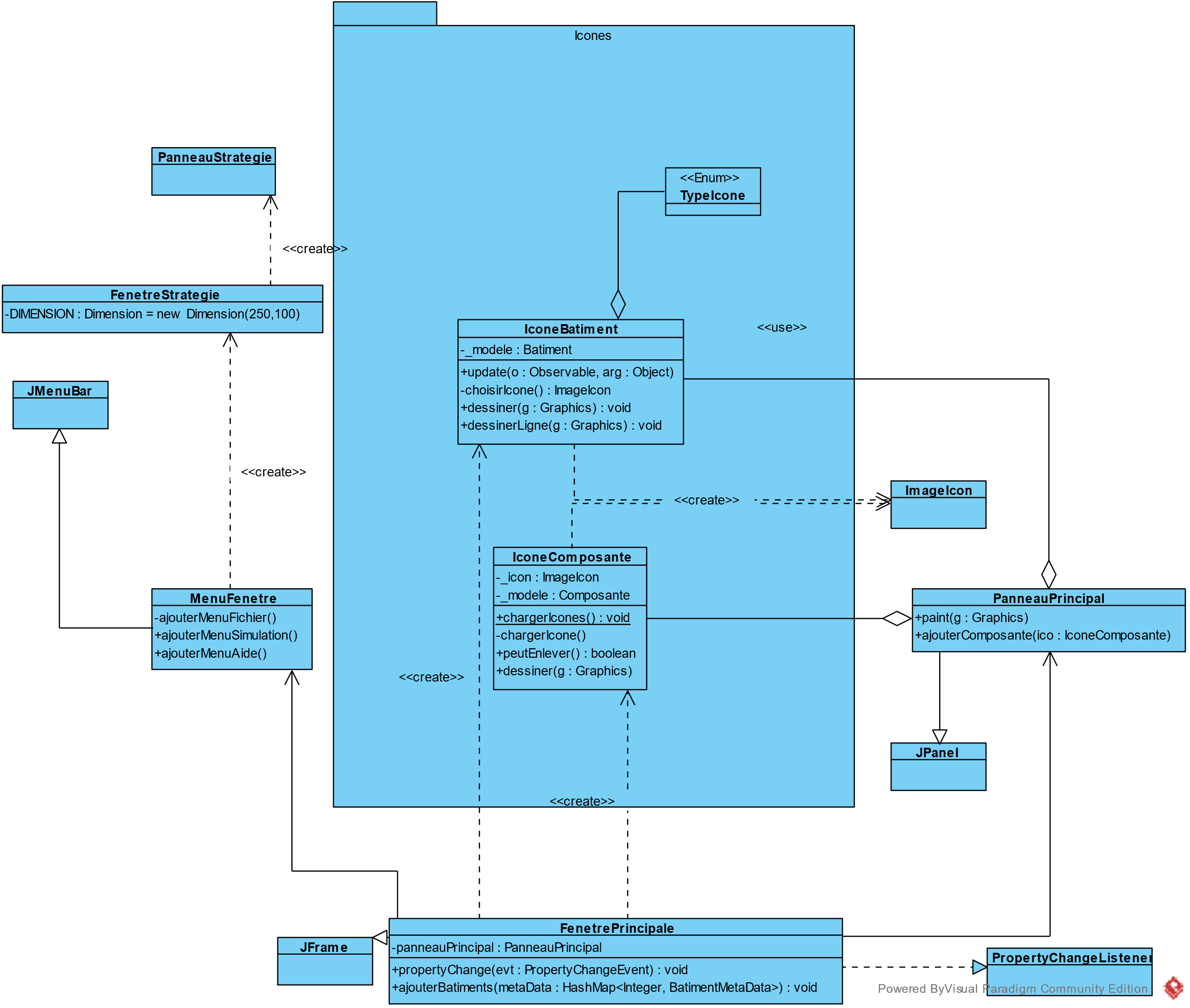


Diagramme de Classe Vue

## Annexe B : Diagrammes de décisions

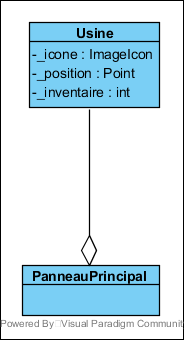


Diagramme de Classe Exemple Aucune Architecture

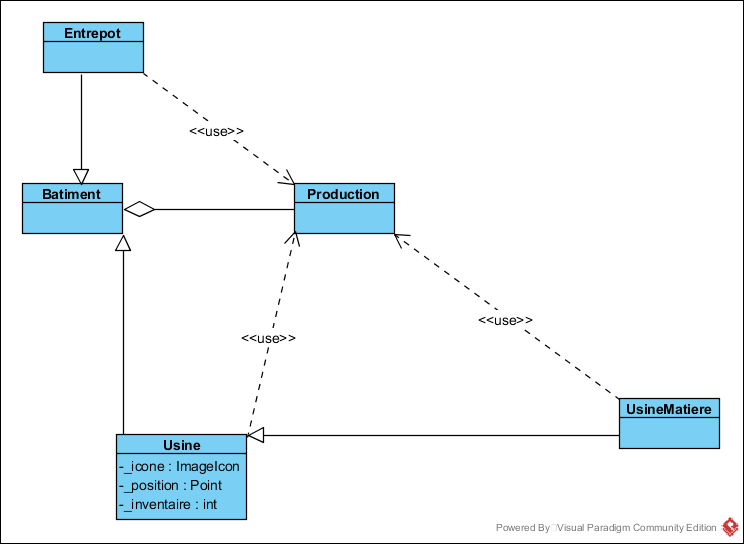


Diagramme de Classe Classe Production Comme Classe Externe

## Annexe C : Exemples de Code

Classe Batiment{

Public ajouterUneAileAInventaire(){

Production.ajouter(“aile”);

}

Public ajouterUnMoteurAInventaire(){

Production.ajouter(“moteur”);

}

}

Exemple de Code Production comme classe interne

## Annexe D : Diagrammes de Séquence

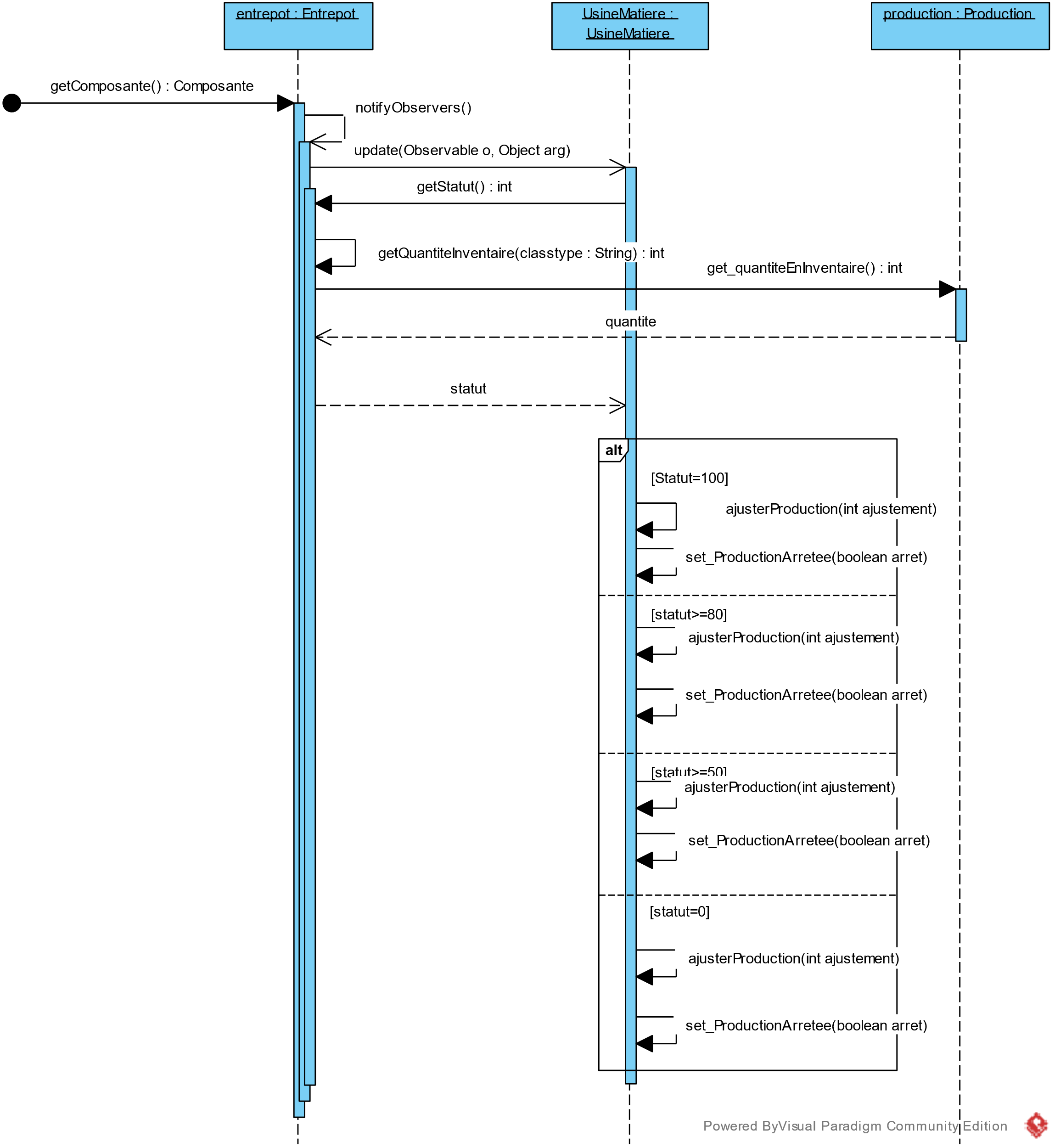


Diagramme de Séquence Observation de l'entrepôt

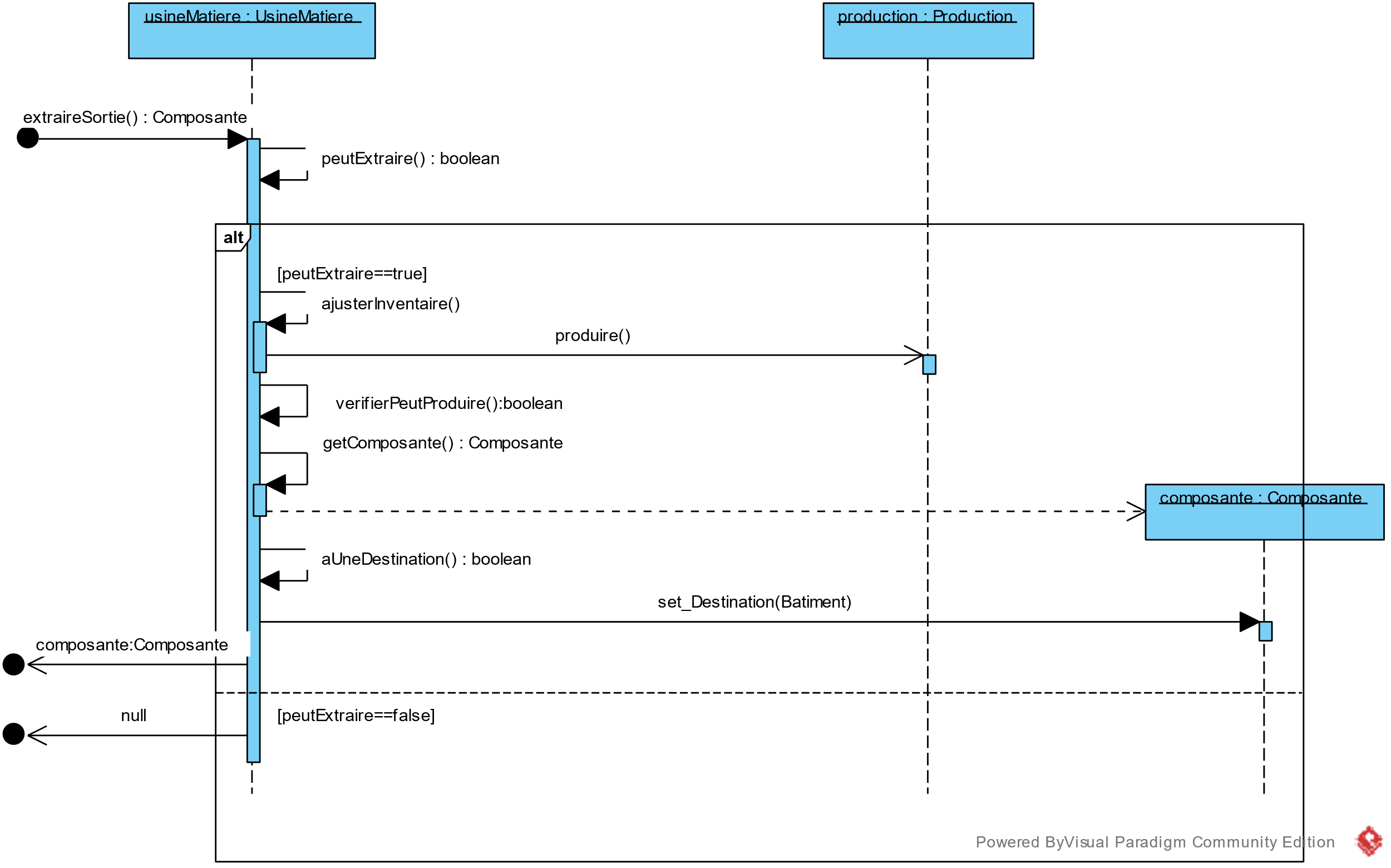


Diagramme de Séquence Extraction d'une Composante