# Exemple d'utilisation d'UML

Jean-Yves Didier

## Table des matières



## Situation Initiale

**Le producteur et le consommateur :** *Un producteur produit des objets qui sont consommés par un consommateur . A une lointaine époque, le consommateur pouvait demander directement l'objet au producteur. Toutefois, la législation actuelle, pour des raisons d'hygiène, impose que le producteur ne soit plus en contact direct avec le consommateur. L'échange d objets se fait donc par un compartiment muni de deux trappes. Une trappe est destiné au producteur, l'autre au consommateur. Il ne peut y avoir qu'une seule trappe ouverte en même temps. Le producteur, son objet terminé au bout d'un temps plus ou moins long, ouvre la trappe et met l'objet dans le compartiment. Le consommateur, quant à lui, ouvre la trappe dès qu'il peut pour voir si l'objet de sa convoitise se trouve dans le compartiment.*

Nous nous proposons de simuler par un programme Java le comportement d'un producteur et d'un consommateur qui s'échangent des objets via un compartiment. Afin de simplifier la simulation, nous décidons que le nombre d'objets échangés sera de 10.

Le producteur ainsi que le consommateur peuvent être assimilés à des threads Java au comportement bien déterminé.

# 1 Les cas d'utilisation

## 1.1 Une première ébauche du diagramme de cas d'utilisation

Tout d'abord, dans un tel cas d'étude, il convient d'énumérer les acteurs ainsi que les objectifs auquel doit répondre le système selon le point de vue des acteurs. Une première vision de la situation nous donne la liste d'acteurs (personnes ou systèmes intervenant dans le modèles) suivants :

- Le producteur,
- Le consommateur,
- Le compartiment,

Les objectifs que doit accomplir le système sont les suivants (entre parenthèses sont indiqués les acteurs impliqués dans les objectifs) :

- Mettre des objets dans le compartiment (producteur),
- Prendre des objets dans le compartiment (consommateur),
- Assurer la synchronisation (compartiment).

Ces objectifs vont se transformer en cas d'utilisation que l'on pourra associer à chaque acteur. Pour ce qui est de la dépose ou de la prise d'objets, les cas d'utilisation feront appel à la fonctionnalité de synchronisation lorsque cette dernière s'avère nécessaire à utiliser (dépendance de type *extend*). Ceci aboutit à la réalisation du premier diagramme de cas d'utilisation (Fig. 1).

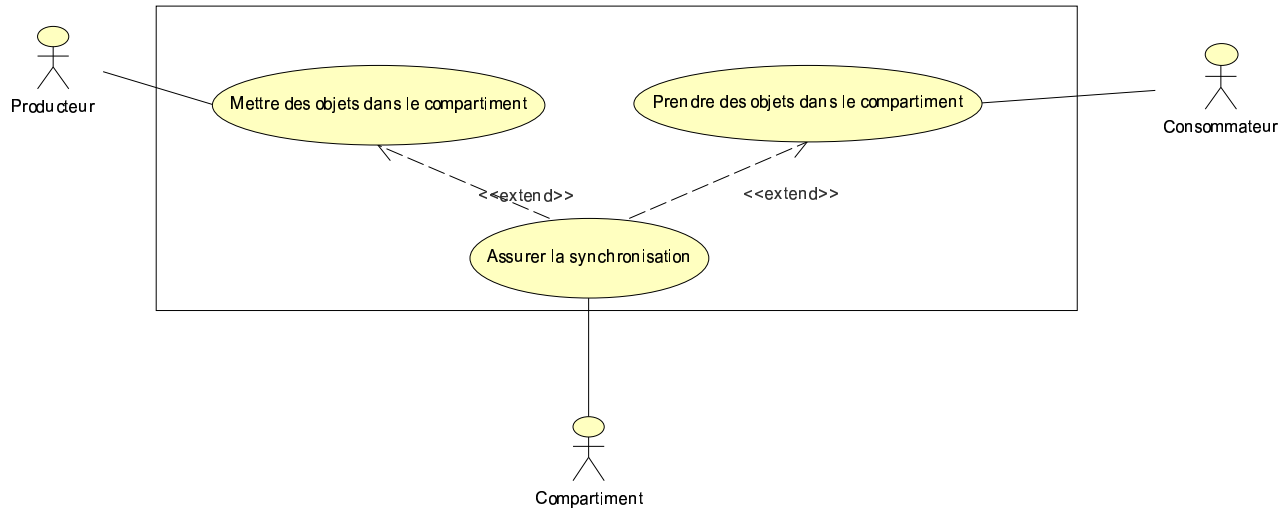



FIG. 1 – Premier diagramme de cas d'utilisation

## 1.2 Élaboration du diagramme de cas d'utilisation

Toutefois, en raison de ce qui est proposé, la modélisation de la situation implique également le fait que celle-ci est une simulation de la situation initiale. Ceci implique également de simuler le producteur, le consommateur et le compartiment. Ceci amène à créer un nouveau diagramme de cas d'utilisation (Fig. 2 page suivante).

Une fois le diagramme de cas d'utilisation élaboré, nous allons tracer les diagrammes d'activités correspondants à chaque cas d'utilisation mentionnés.

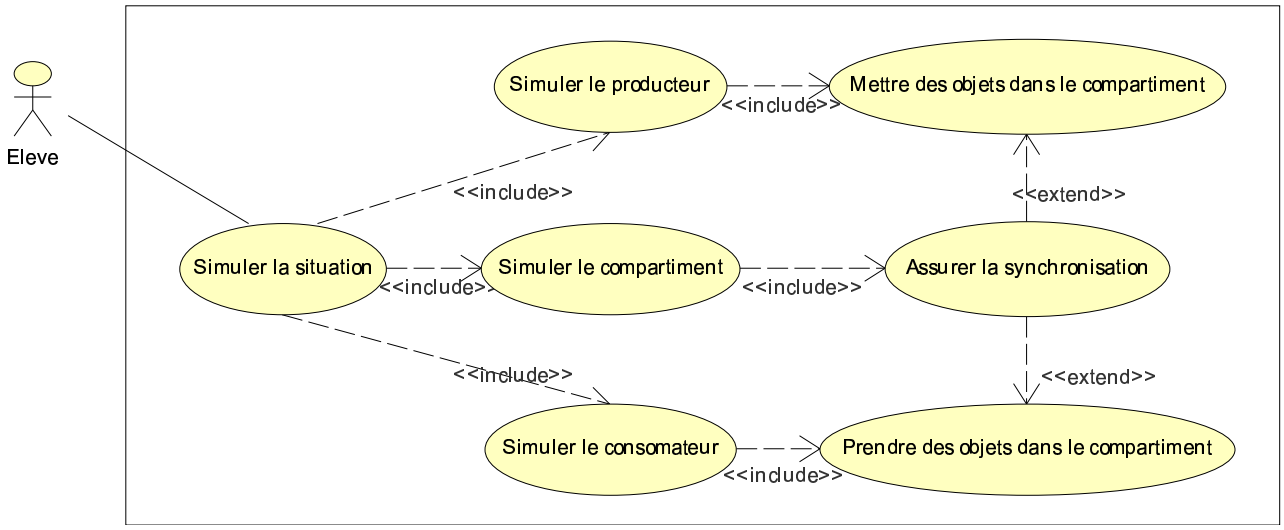



FIG. 2 – Diagramme de cas d'utilisation appliqué à la simulation

## 1.3 Scénarios de cas d'utilisation

Les scénarios de cas d'utilisation peuvent être déterminés en s'appuyant sur les diagrammes d'activité liés à la description de chaque cas d'utilisation.

### 1.3.1 Simuler la situation

(Voir Fig. 3)

Le diagramme met en évidence le lancement en parallèle de la simulation du producteur et du consommateur.

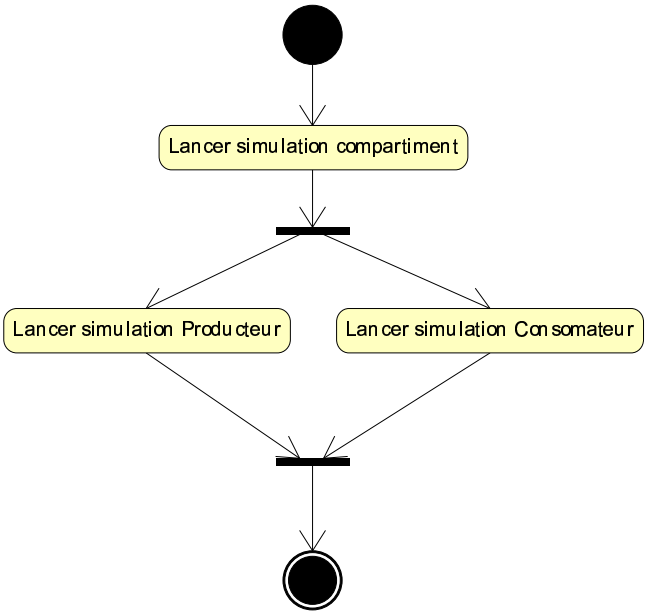
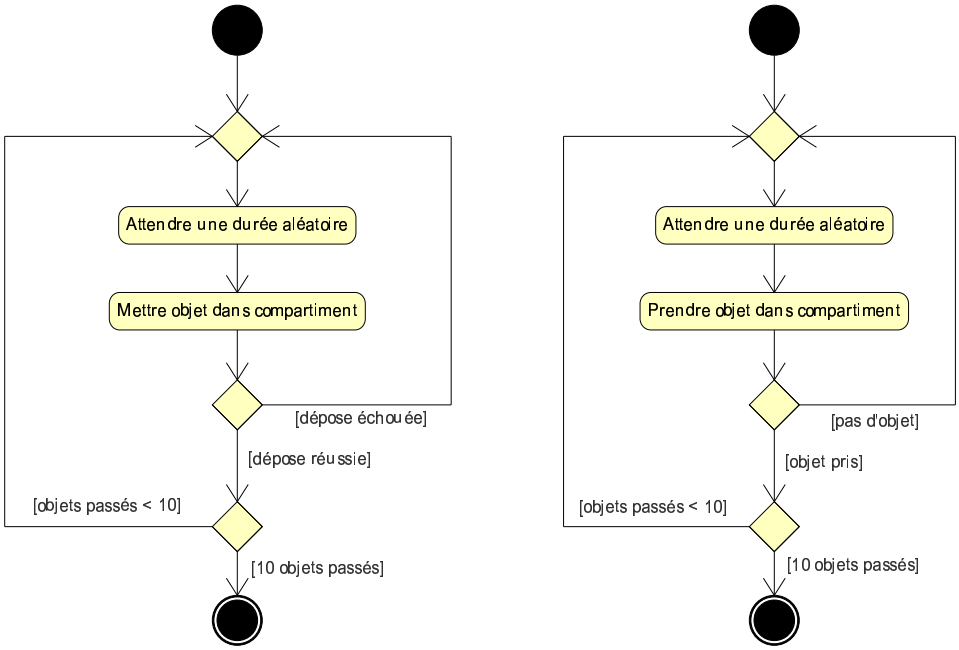



FIG. 3 – Diagramme d'activité lié au cas d'utilisation *Simuler la situation*

### 1.3.2 Simuler le producteur et le consommateur

(Voir Fig. 4(a) et 4(b))

Dans le cadre de la situation initiale, nous ne savons pas quand le producteur va mettre un objet et quand le consommateur va prendre un objet dans le compartiment. Ceci sera matérialisé par un temps d'attente aléatoire. Ensuite, le producteur et le consommateur vont tenter d'effectuer leurs actions respectives. Si celles-ci échouent, alors ils se remettent à attendre un temps aléatoire avant de refaire une tentative.




(a) Simuler le producteur (b) Simuler le consommateur

FIG. 4 – Diagrammes d'activités associés aux cas d'utilisations simulant producteur et consommateur

### 1.3.3 Mettre, prendre un objet et assurer la synchronisation

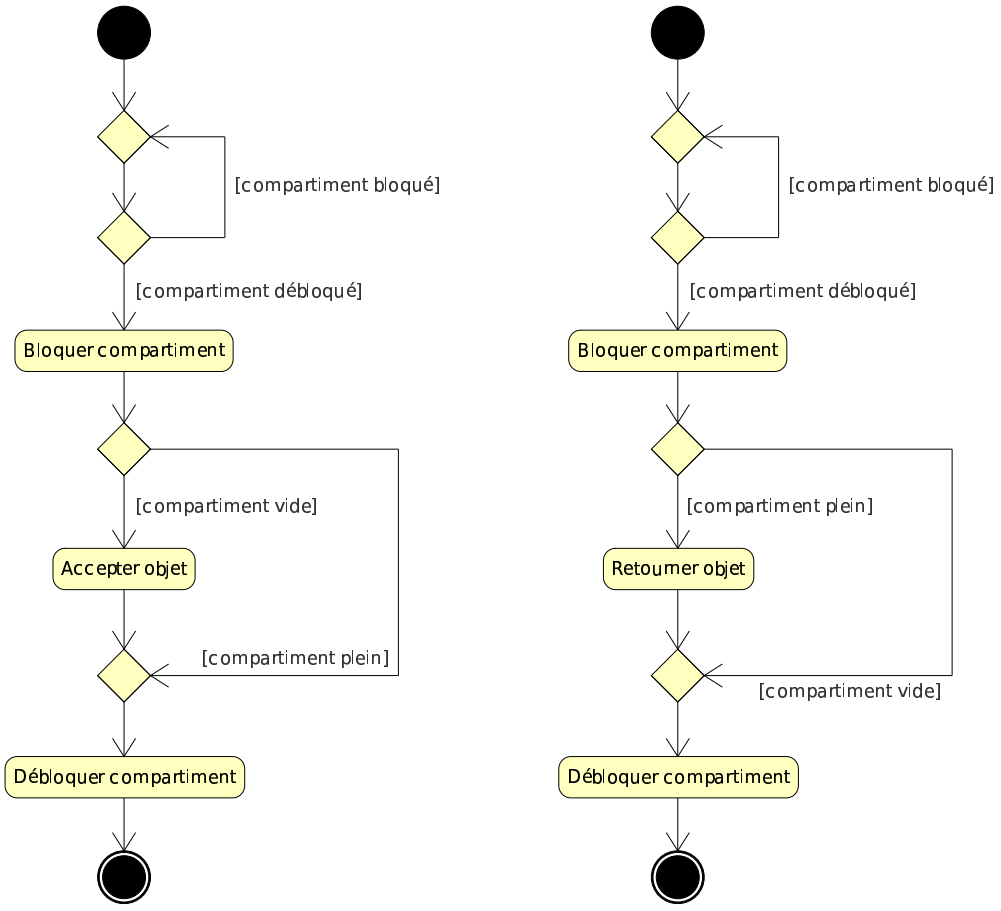
(Voir Fig. 5(a) et 5(b) page suivante)

Pour chaque cas d'utilisation élémentaire, l'idée est de bloquer le compartiment si celui-ci n'est pas déjà bloqué, vérifier l'état de ce dernier (vide ou plein), puis effectuer l'opération associé à la prise de l'objet (*retourner l'objet*) ou à la mise de l'objet (*Accepter l'objet*) du point de vue du compartiment. Le cas d'utilisation n'est pas représenté, puisqu'il est en réalité composé des opérations de blocage et de déblocage du compartiment.

---

**Remarque :** Lorsqu'il y a une dépendance entre deux cas d'utilisation, ceci se traduit par la présence d'une activité nommée comme l'un des cas d'utilisation dans le diagramme d'activités de l'autre cas d'utilisation. Cf le diagramme d'activités de la simulation du producteur (Fig. 4(a)) qui contient une activité *Mettre objet dans compartiment*. Dans le même ordre d'idée, il serait possible de substituer à cette activité le diagramme qui décrit le cas d'utilisation du même nom et de l'inclure dans le diagramme de la simulation du producteur. Toutefois, ceci est à éviter puisque cela surchargerait le diagramme en question.

---




(a) Mettre un objet (b) Prendre un objet

FIG. 5 – Diagrammes d'activités associés aux cas d'utilisations mettre et prendre un objet dans le compartiment

Une fois les diagrammes d'activités réalisés, chaque cheminement possible dans un diagramme d'activité se traduit sous la forme d'un scénario de cas d'utilisation. Dans le cadre d'une application complexe, étudier de manières exhaustive tous les cheminements est impensable. L'idée est de sélectionner les cheminements qui vont mettre en évidence les interactions entre les différents objets du système en vue de déterminer les interfaces des classes à l'aide des diagrammes de collaboration.

# 2 Diagrammes de collaboration

## 2.1 Lancement de la simulation

La simulation doit d'abord créer les entités *Producer* (Producteur), *Consumer* (Consommateur) et *CubbyHole* (Compartiment). De toute évidence, le producteur et le consommateur doivent avoir connaissance du compartiment par lequel ils doivent communiquer. C'est pourquoi, le compartiment créé est passé en paramètre aux opérations chargées de créer le Producteur et le Consommateur. Enfin, la simulation doit démarrer les fonctions principales (*start*) des deux threads *Producer* et *Consumer*. C'est ce que nous pouvons voir sur le diagramme d'activité correspondant (Fig. 6).

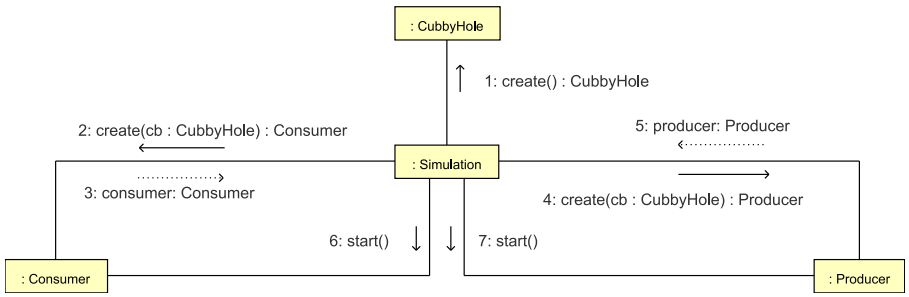



FIG. 6 – Diagramme de collaboration décrivant les messages échangés au lancement de la simulation

### 2.1.1 Passage au diagramme de séquence

Comme évoqué en cours, il est facile de passer du diagramme de collaboration au diagramme de séquences (Fig. 7 page suivante). Ce dernier a l'avantage de montrer l'aspect temporel des échanges de messages. A titre d'exemple, nous montrons ici le diagramme de séquence résultant pour la simulation. Ce dernier met en évidence la création des objets de type compartiment, producteur et consommateur par rapport à la simulation.

## 2.2 Mettre et prendre un objet

Avant de tracer le diagramme de collaboration portant sur la mise d'un objet dans le compartiment, nous allons nous intéresser diagramme d'activités *Mettre un objet* (Fig. 5(a) page précédente). L'idée est de trouver un ou plusieurs cheminements dans ce diagramme nous permettant de passer par toutes les activités de ce diagramme. Ici, le cas étant simple, il existe un cheminement permettant de couvrir l'intégralité des activités : celui qui est à la verticale du point d'entrée au point de sortie.

Il reste donc à traduire ces opérations en messages entre les entités intervenant dans le diagramme d"activités : le producteur et le compartiment. Les messages sont :

- la mise (*put*) dans le compartiment,
- le blocage ou vérouillage (*lock*) du compartiment,
- le déblocage ou déverrouillage (*unlock*),
- l'acceptation ou stockage (*store*) effectif de l'objet.

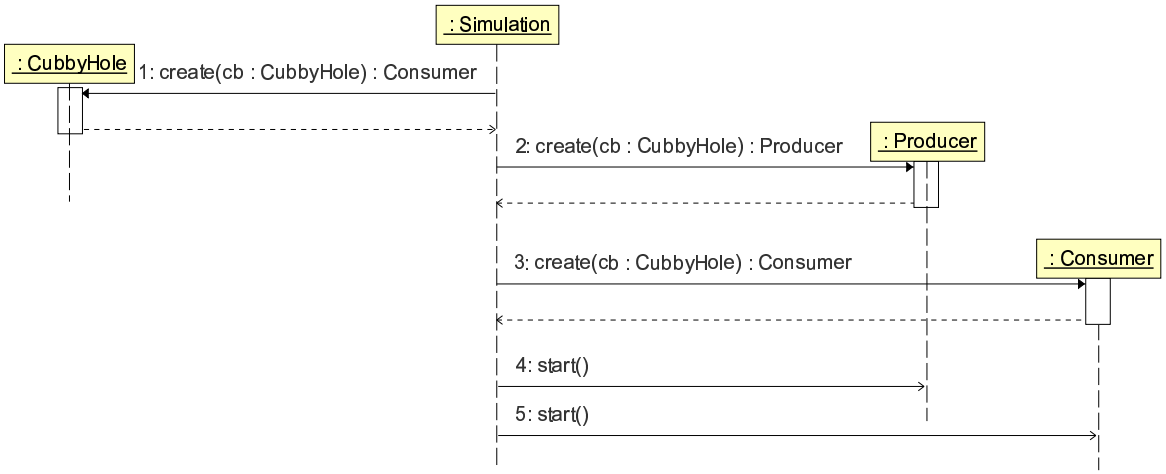
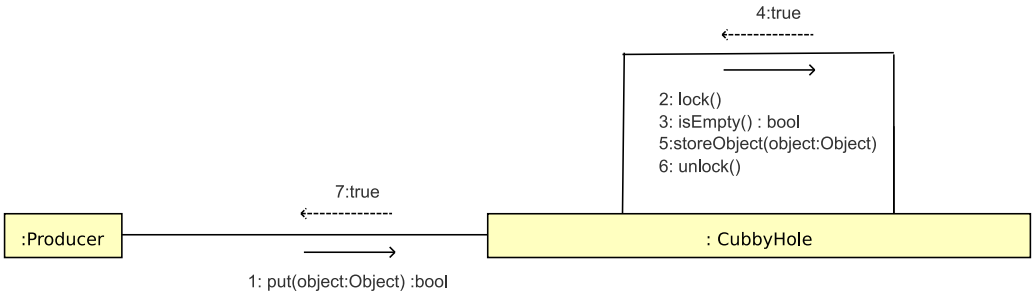


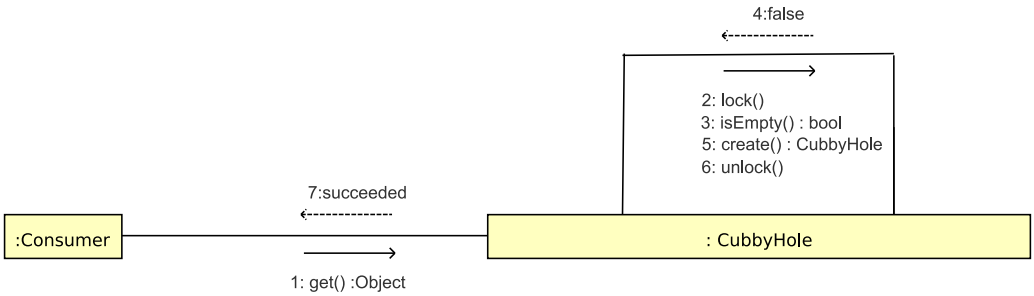

FIG. 7 – Diagramme de séquences associé à la simulation

Cette traduction aboutit au diagramme de collaboration *Mise d'un objet dans le compartiment* (Fig. 8(a)).

La même logique de construction s'applique au diagramme de collaboration concernant la *prise* (get) *d'un objet dans le compartiment* (Fig. 8(b)).




(a) Mise d'un objet dans le compartiment




(b) Prise d'un objet dans le compartiment

FIG. 8 – Diagrammes de collaboration décrivant la mise et la prise d'un objet dans le compartiment

---

**Remarque :** Afin de ne pas surcharger les diagrammes de collaboration, lorsqu'un appel sans type de retour (*void*) est réalisé, le diagramme ne comporte pas de message de retour associé à ces appels.

---

Lorsque les diagrammes de collaborations ont été tracés, il est possible de déterminer les interfaces entre les divers objets. Les interfaces sont en réalité les messages échangés. Dans le cadre de cet exemple, les messages sont en réalité les opérations contenues par nos classes. Il est dès lors, possible de créer le diagramme des classes de notre application.

# 3 Diagramme des classes

Le diagramme des classes fait également intervenir la classe *Thread* car il est précisé dans l'énoncé que le producteur et le consommateur sont simulés par des threads. Nous pouvons remarquer la relation de généralisation qui relie les trois classes dans le diagramme des classes (Fig. 9).

Il faut spécifier que la classe *Simulation* est le point d'entrée du programme (*main*) et qu'elle contient, pour les besoins de la simulation, un producteur, un consommateur et un compartiment.

De même, le producteur et le consommateur contiennent une référence au compartiment qu'ils vont utiliser pour se passer les objets.

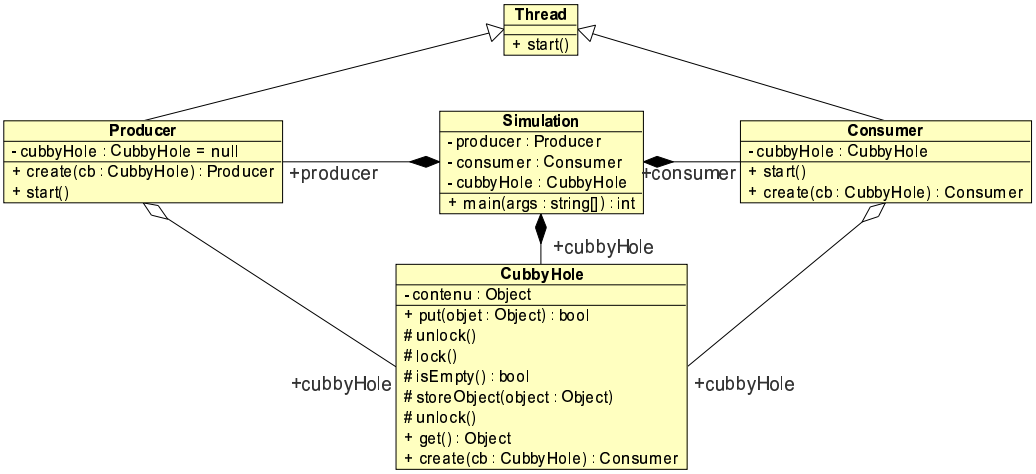



FIG. 9 – Diagramme des classes

Une fois le diagramme des classes tracé, nous pouvons compléter ce dernier avec des diagrammes d'activité et des diagrammes de séquence décrivant les diverses opérations à effectuer.

Toutefois, compte tenu de la complexité faible de l'exemple présenté, les diverses opérations effectuées (le point d'entrée, la mise et la prise d'objets) sont déjà documentés en détails.

En revanche, il convient de faire quelques remarques sur la programmation des classes résultantes en Java par rapport à ce qui est exposé dans les diagrammes UML ici présents.

## 3.1 Discussion sur les spécificité Java et le développement de classes

### 3.1.1 Les constructeurs

En UML, le constructeur d'une classe est une opération comme une autre. Ainsi, les opérations *create* évoquées pour les classes *Consumer, Producer* et *CubbyHole* sont en réalité des constructeurs des classes concernées et pourraient être nommés en tant que tel.

### 3.1.2 Le verrouillage et déverrouillage du compartiment

Si l'on regarde les diagrammes de collaboration concernant la mise et la prise d'un objet dans le compartiment (Fig. 8(a) et 8(b) page précédente), ainsi que les diagrammes d'activité reflétant les mêmes processus (Fig. 5(a) et 5(b) page 5), nous remarquons que la première activité traversée est le verrouillage et la dernière le déverrouillage du compartiment quelque soit le parcours choisi dans les diagrammes en question. Ceci est géré spécifiquement en Java par le mot clé *synchronized*. En somme, il est possible de simplifier les diagrammes de collaborations associés. Ceci est ainsi réalisé

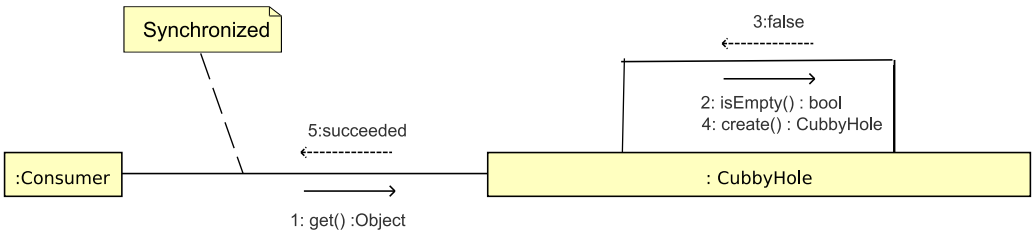



FIG. 10 – Diagramme de collaboration simplifié pour la prise d'objets

pour la prise d'objet (Fig. 10).

La discussion concernant les spécificités de Java par rapport à UML étant terminées. Nous pouvons supposer que le code est prêt à être généré. Ce code sera entreposé sous forme de fichiers qui seront ensuite compilés pour exécuter la simulation. Nous allons voir quels sont les composants que ceci génère pour la simulation.

# 4 Le diagramme de composants

Chaque classe développée est écrite en java. Ceci résulte en un fichier à l'extension *.java*. Ces fichiers sont ensuite compilés pour fournir les fichiers à l'extension *.class* contenant le bytecode nécessaire à l'exécution. Ensuite, pour l'exécution, il faut lancer *Simulation.class* en utilisant la machine virtuelle Java. L'ensemble des composants mis en jeu ainsi que leurs dépendances sont présentes dans le diagramme de composants (Fig. 11).

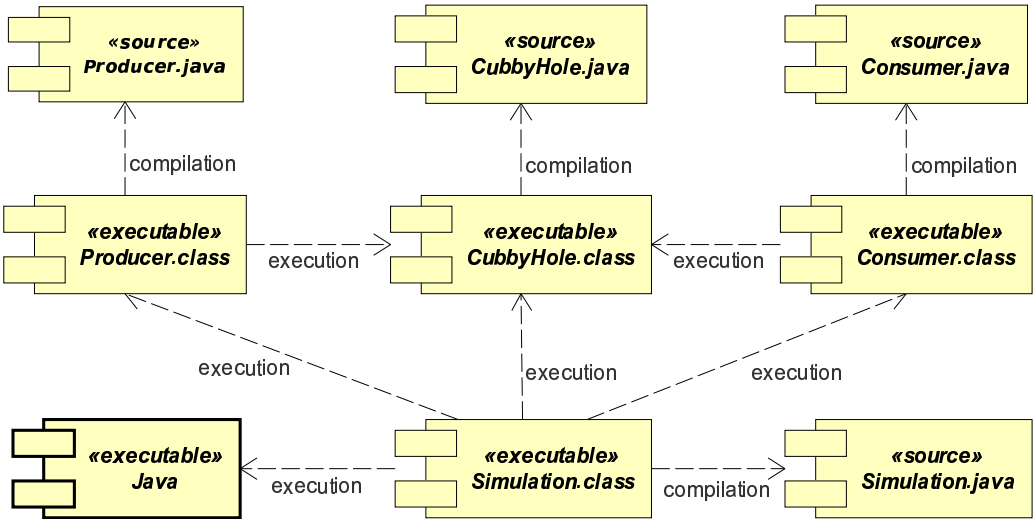



FIG. 11 – Diagramme de composants

Pour parachever la vue de déploiement, il convient également de tracer le diagramme de déploiement.

# 5 Le diagramme de déploiement

Le diagramme de déploiement (Fig. 12 page suivante) est ici très simple puisqu'il s'agit d'exécuter la simulation sur une seule machine capable de supporter le poids d'une machine virtuelle java. La version 1.5.0 du java development kit nécessite au minimum 64 Mo de mémoire vive pour fonctionner. Ici les développements et l'exécution sont supposés se dérouler sur un système de type Linux.

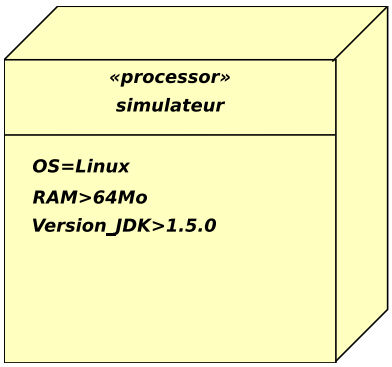



FIG. 12 – Diagramme de déploiement

# Conclusion

Cet exemple met en oeuvre les principaux diagrammes employés dans le cadre de la notation UML :

- diagramme de cas d'utilisation,
- diagramme d'activités,
- diagramme de collaborations,
- diagramme de séquences,
- diagramme de classes,
- diagramme de composants,
- diagramme de déploiement.

Comme nous pouvons le voir, il n'est pas nécessaire, à chaque fois, d'utiliser l'ensemble des diagrammes disponibles pour effectuer l'analyse complète d'un système. Les autres diagrammes serviront éventuellement pour d'autres types de programmes ou de systèmes d'information.

L'objectif consiste simplement à établir le support documentaire nécessaire pour comprendre le système décrit sous ses divers aspects.

---

Diagrammes dessinés avec Umbrello et retouchés à l'aide d'Inkscape, document généré à l'aide de pdflatex.