Group 22 - IS1220 - Project Report - Part 1
Emergency Department Simulator

RENAULT Thomas
VERLHAC Quentin

4 décembre 2017

Sommaire

1 Introduction

Ce document est le rapport d'un projet d'ingénierie logicielle visant à simuler le fonctionnement du département des urgences dans un hôpital. Ce projet, mené en binôme, comprend les étapes suivantes : compréhension du cahier des charges, conception de l'architecture logicielle, implémentation dans le langage Java et tests du simulateur.

2 Contexte et compréhension de l'énoncé

Le département des urgences est une entité regroupant à la fois des patients, du personnel (médecin, infirmières et transporteurs) et du matériel (différentes salles). Afin d'être traités, les patients suivent le processus suivant :

1. Enregistrement : le patient est entré dans le système informatique du département.

   Nécessite : une salle d'attente, une infirmière

2. Installation : le patient est amené dans une salle de consultation adaptée à son degré de gravité.

   Nécessite : une infirmière

3. Consultation : le patient est consulté par un médecin, qui détermine la prochaine étape.
   — Test sanguin
   — Radio
   — IRM
   — Fin de l'hospitalisation

   Nécessite : une salle de consultation, un médecin

4. Transport : le patient est amené dans une salle correspondant à l'examen prescrit.

   Nécessite : un transporteur

5. Examen : le patient est examiné selon la prescription du médecin.

   Nécessite : une salle d'examen

Une fois terminé, le patient retourne en consultation et les étapes 3 à 5 peuvent s'enchainer plusieurs fois.

Chaque patient reçoit une fiche de soin lors de son arrivée aux urgences. Il conserve cette fiche avec lui tout au long de sa prise en charge. A l'issue de chaque étape, la fiche est mise à jour avec une nouvelle entrée. Cette fiche permet notamment de calculer le prix de l'hospitalisation du patient, qui dépend de sa mutuelle.

	Design	Code	Test
EmergencyDepartment	Thomas	Thomas	Quentin
Ressources humaines	Thomas	Thomas	Quentin
Ressources matérielles	Quentin	Quentin	Thomas
Traitements du patient	Quentin	Quentin	Thomas

TABLE 1 – Répartition des tâches

Les ressources (personnel, matériel) étant limitées, certains patients peuvent se retrouver à attendre qu'un service soit disponible. Dans ce cas, ils sont placés dans une salle d'attente.

L'objectif du département est d'assurer un temps d'attente minimal pour les patients, tout en donnant la priorité aux cas les plus graves. Les indicateurs clés pour mesurer la performance de l'hôpital sont par exemple la durée de l'hospitalisation ou le temps d'attente jusqu'à la première consultation.

3 Architecture et design

Après avoir analysé les prérequis du simulateur, nous nous sommes intéressés à la répresentation des différents objets nécessaires sous forme de classe et leur articulation. Nous avons d'abord identifié plusieurs groupes d'éléments similaires : le premier regroupant toutes les ressources de l'hôpital, le second les services (correspondant aux étapes du parcours d'un client), le troisième les classes "utiles" ou secondaires utilisées par les premières citées (distributions de probabilité, niveaux de sévérité des patients, assurances maladies). Nous avons donc structuré notre projet en trois packages contenant ces éléments : *resources*, *workflow* et *utils*. Nous nous sommes ensutie aperçus qu'il fallait ajouter d'autres éléments et nous avons donc créé de nouveaux packages. Le package *core* contient la classe principale répresentant un département des urgences complet, des factories pour produire les instances dont il a besoin et des éléments plus généraux comme les interfaces *Observable* et *Observer*. Le package *processing* contient les classes servant à exécuter les actions sur les patients à l'aide du *Command Pattern* détaillé un peu plus loin. Enfin les éléments du package *resources* sont de plusieurs types : les ressources humaines (patients et employés : infirmières, médecins, transporteurs) et les ressources non humaines (salles).

Nous avons utilisé cette séparation du projet en sous-parties pour nous répartir les tâches équitablement. Nous avons également adopté le principe suivant : la personne qui modélise et implémente une classe n'est pas celle qui écrit les tests la concernant.

3.1 Diagramme UML et commentaires

3.1.1 Organisation générale

Le diagramme UML complet est fourni en annexe, pour des raisons de lisibilité. Nous fournissons ici un schéma simplifié de l'organisation générale. Celui-ci conserve l'information global en s'affranchissant des détails d'implémentation.

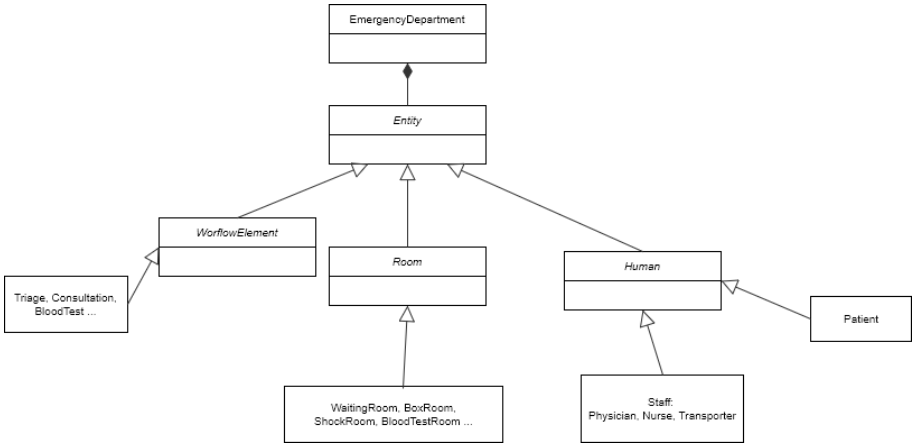

FIGURE 1 – Organisation générale des classes

3.1.2 Classe EmergencyDepartment

La classe *EmergencyDepartment* est la classe principale du simulateur, qui représente un département des urgences. Elle contient des ressources humaines et matérielles, des patients, des services et un historique des actions effectuées. Cette classe stocke également l'heure et permet de faire avancer le temps en exécutant la prochaine action.

3.1.3 Ressources humaines

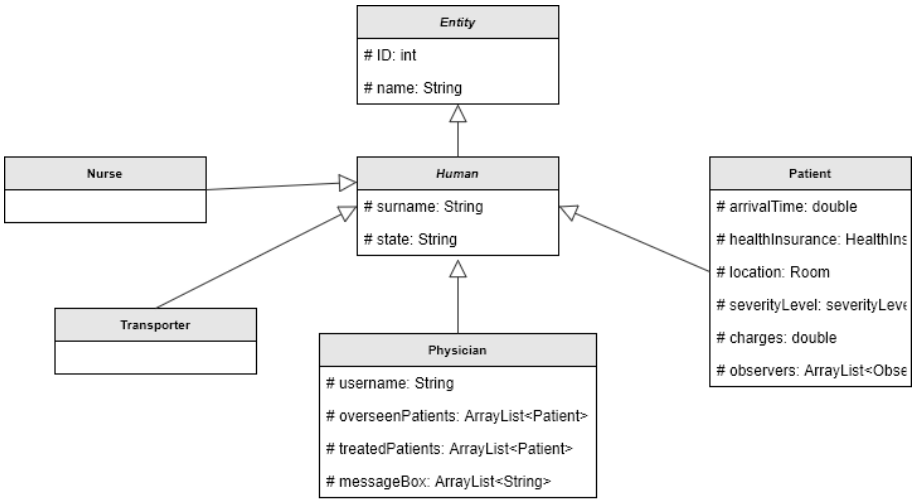

FIGURE 2 – Diagramme UML des ressources humaines

Classe Human

La classe Human est une classe abstraite héritant de la classe *Entity*. En plus de l'ID et du nom fournis par cette dernière, elle ajoute un nom de famille et un état (par exemple occupé, en attente, libre, etc) permettant de caractériser les humains présents dans la simulation.

Nurse et Transporter

Les infirmières et les transporteurs sont des employés de l'hôpital qui s'occupent des patients : les infirmières les enregistrent et les dirigent vers les salles de consultation tandis que les transporteurs emmènent les patients vers les salles d'examen. Ces deux types d'employés sont représentés par les classes *Nurse* et *Transporter* héritant simplement de la classe *Human*.

Physician et Patient

Le troisième type d'employé est le médecin, réprésenté par la classe *Physician*. La modélisation des médecins est plus complexe que les autres employés car ils doivent assurer le suivi des patients qui leurs sont attribués.

Pour cela, nous avons utilisé l'*Observer Pattern* afin que les médecins puissent être informés de la situation des patients qu'ils supervisent. Nous avons implémentés des interfaces *Observable* et *Observer*, implémentées respectivement par *Patient* et *Physician*. Ainsi, lorsque le patient subit une action, il notifie son médecin attribué qui reçoit un message. Le médecin a également accès à la liste des patients qu'il supervise ainsi que ceux qu'il a supervisés et qui sont sortis de l'hôpital.

Les patients possèdent également des attributs et fonctionnalités supplémentaires par rapport à la classe *Human* dont ils héritent. La classe patient contient des attributs stockant l'heure d'arrivée, l'assurance maladie du patient, le niveau de sévérité de sa blessure ou maladie, son emplacement physique dans l'hôpital, le total des frais dus à l'hôpital pour les actions subies (consultation, examen, etc.) ainsi qu'un historique d'évènements (constitués d'un timestamp et d'une chaîne de caractères décrivant l'évènement).

Factories

Nous avons utilisé le *Factory pattern* pour les humains afin de pouvoir les générer plus facilement, notamment avec des noms aléatoires mais réalistes. Nous avons pour cela défini une classe abstraite *Human factory* exposant une fonction pour générer un prénom et un nom tirés aléatoirement dans une liste de noms courants fournie. Cette classe définit également une méthode abstraite *create* implémentée par les classes filles définies pour chaque types d'humain à générer (patient, médecin, infirmier, transporteur).

HealthInsurance et SeverityLevel

Les assurances maladies que peuvent avoir les patients sont représentées par des classes héritant d'une classe mère abstraite *HealthInsurance* implémentant un attribut stockant le pourcentage des frais remboursé par l'assurance et une méthode *computeDiscountedPrice* permettant de calculer le prix que doit payer le patient en prenant en compte le remboursement de son assurance. Des classes *NoInsurance*, *SilverInsurance* et *GoldInsurance* symbolisent les trois types d'assurances fournies de base par le simulateur, avec un pourcentage de remboursement respectivement à 0%, 50% et 80%.

3.1.4 Salles

Les différents types de salles sont représentés par des classes héritant de la classe mère abstraite *Room*. Celle-ci possède un attribut représentant sa capacité ainsi qu'une liste de patients qui sont présents dans la salle. Les différents types de salles sont : *BoxRoom* et *ShockRoom* (salles de consultation) ainsi que *BloodTestRoom*, *XRayRoom* et *MRIRoom* (salles d'examen). La classe *Corridor* est une salle de capacité infinie permettant de réprésenter les couloirs de l'hôpital, où vont les patients lorsqu'il n'y a plus de place dans les salles d'attente. Chaque hôpital possède une instance de cette classe.

3.1.5 Workflow

Le package workflow contient toutes les classes qui gèrent le parcours du patient dans le département des urgences. Chaque étape présentée dans la section Contexte et compréhension de l'énoncé est représentée par une classe éponyme, qui hérite de la classe abstraite WorkflowElement.

Cette classe abstraite implémente des attributs et méthodes communs à l'ensemble des étapes du workflow, notamment pour la Gestion du temps avec le design pattern.

Chaque étape du workflow est capable de déterminer la prochaine tâche qu'elle doit executer, en fonction des patients en cours de traitement, des patients dans la file d'attente et de la disponibilité des ressources. Actuellement, les patients graves (niveau 1 ou 2) ont la priorité absolue sur les patients légers (niveaux 3 à 5). A terme, cette gestion des priorité sera affinée pour éviter des situations de famine pour les patients légers. Des précisions supplémentaires sur le workflow peuvent être trouvées dans la section Gestion du temps pour son fonctionnement global et dans la Javadoc pour le détail de ses classes.

3.1.6 Aléatoire

Les distributions aléatoires sont un aspect important du projet puisqu'elles sont utilisées à la fois pour générer l'arrivée des patients et pour déterminer la durée des actions effectuées sur les patients. Nous avons choisi d'utiliser le *Strategy pattern* afin de pouvoir facilement et indifféremment utiliser la distribution de probabilité souhaitée. Une interface *ProbabilityDistribution* définit la fonction *generateSample()* qui est appelée lorsque l'on souhaite générer un échantillon d'une distribution de probabilité. Nous avons implémenté différentes classes représentant diverses distributions de probabilité : déterministe, exponentielle et uniforme. Ces classes contiennent lorsque nécessaire les paramètres des lois comme attributs et implémentent donc la fonction *generateSample()* qui générère des échantillons. Pour cela nous utilisons la fonction *Math.random()* du package *java.lang* qui permet de générer un nombre flottant aléatoire compris entre 0 et 1 et le fait que les distributions de probabilité implémentées peuvent être obtenues à partir d'une loi uniforme entre 0 et 1 par les formules suivants :

— Loi uniforme entre a et b : $U([a,b]) = a + U([0,1]) * (b-a)$
— Loi exponentielle entre a et b : $\varepsilon(\lambda) = -\frac{ln(U([0,1])}{\lambda}$

3.1.7 Gestion du temps

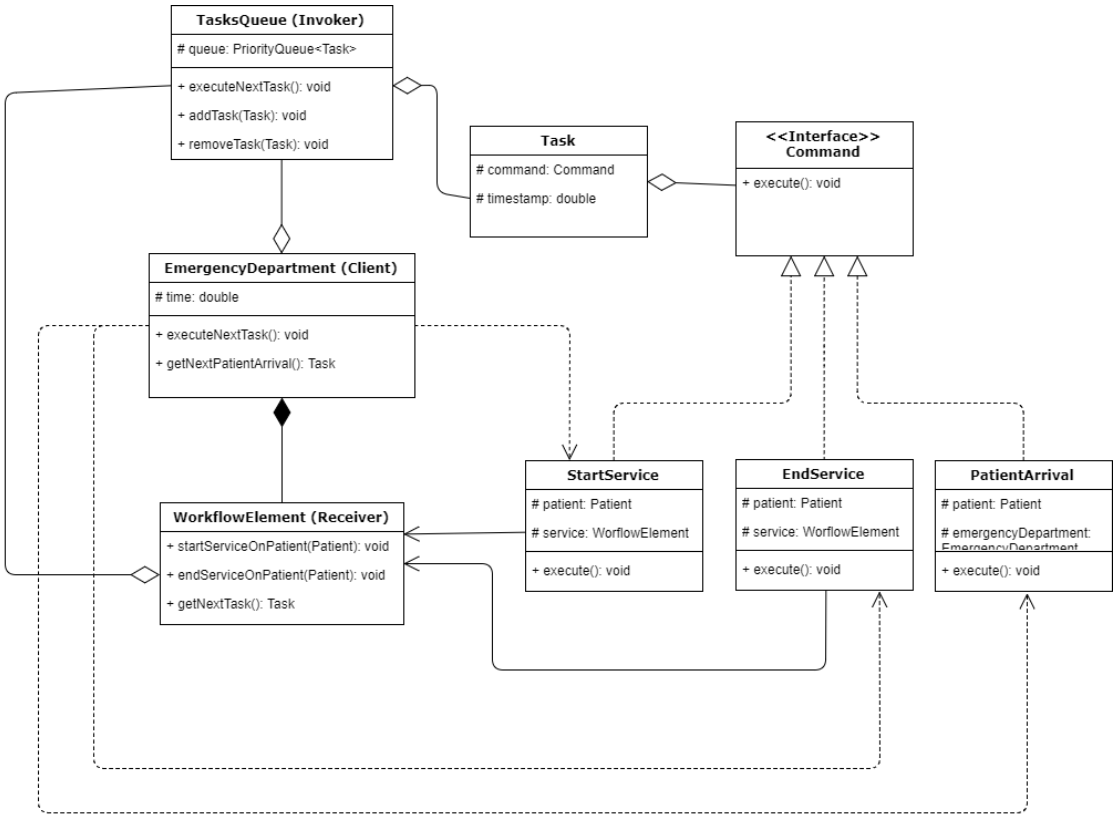

FIGURE 3 – Implémentation du "Command pattern"

La gestion du temps est effectuée par une simulation à évènements discrets. Avant chaque action, on regarde les prochaines evènements possibles puis on exécute l'action associée au plus proche dans le temps.

Le stockage des évènements et des actions associées est implémenté à l'aide du *Command pattern*. Ce pattern permet de stocker dans un objet *Command* une méthode d'un autre objet (*Receiver*) à exécuter et les paramètres à lui passer. De cette manière, on peut stocker les différentes actions à effectuer sur les patients, ainsi que le temps auquel elles doivent être exécutées, dans des objets *Task*. Ces tâches sont elles-mêmes stockées dans une file d'attente, triées par le temps auquel elles doivent être exécutées. La file d'attente *TasksQueue* permet de récupérer la prochaine tâche à exécuter et de l'exécuter.

Dans l'hôpital, chaque service (correspondant aux différentes étapes de la visite d'un patient) peut effectuer deux actions principales : commencer à traiter un patient et finir de le traiter. Ces deux actions sont représentées par deux types de *Command* : *StartService* et *EndService* qui stockent le patient et le service et peuvent exécuter la méthode du service correspondante. Chaque service possède sa propre *TasksQueue* pour stocker les prochaines tâches à effectuer et une méthode qui renvoie la prochaine tâche. Lors de chaque itération, l'instance de la classe *EmergencyDepartment* va créer une *TasksQueue*, demander à chacun de ses services sa prochaine tâche et générer une tâche correspondante à l'arrivée du prochain patient. Il exécute ensuite la première tâche de la queue, puis recommence.

3.2 Desaccords et consensus

Notre modélisation, tout d'abord relativement simple et peu détaillée, s'est étoffée au fur et à mesure de notre compréhension du sujet, amenant certaines discussions sur les choix à faire.

3.2.1 Gestion du temps

Nous avions identifié deux possibilités concernant la gestion du temps :

— La première était le multithreading, permettant aux workflow elements de tourner en parallèle. Cette méthode a pour avantage de gérer le temps naturellement. En revanche, elle nécessite une compréhension fine des échanges d'informations entre étapes du workflow, ainsi qu'un réglage précis des paramètres temporels (de l'ordre de la dizaine de millisecondes) pour avoir une simulation suffisamment rapide.

— La seconde était un processus à événements discrets, executant les étapes une par une. Cette méthode a pour avantage de simplifier la compréhension du processus, les étapes s'enchainant en série. Elle permet aussi de simuler

aussi vite et simplement que possible la simulation sur des grandes durées, puisqu'elle va de tâche en tâche sans attendre. Par ailleurs, nous avons trouvé un design pattern permettant de mettre en place facilement ce processus discret.

Finalement, la solution que nous avons retenue est celle du processus à événements discrets, par comparaison des avantages et inconvénients de chaque solution.

3.2.2 Héritage des classes répresentant les ressources

L'implémentation des différentes ressources pouvait se faire de plusieurs manières différentes, à travers l'héritage de classe abstraite. A titre d'exemple, nous nous sommes demandés s'il fallait un ID propre à chaque type de ressource, incrémenté dans le constructeur de la classe correspondant. Par simplicité, nous avons finalement préféré implémenter un ID unique pour toutes les classes, en l'incrémentant dans la classe abstraite Entity, dont héritent toutes les classes ressources.

Il y a aussi des conflits entre la simplicité d'implémentation et le sens réel des objets. Par exemple, nous avons choisi d'implémenter le Patient comme une ressource héritant de la classe abstraite Human, au même titre que le Physician ou la Nurse. Cependant, un patient n'est pas considéré comme une ressource au même titre d'un médecin, dans un hôpital.

Enfin, nous avons parfois rencontré des incompatibilités avec les bibliothèques natives de Java. C'est le cas pour le pattern observable/observer. La classe *Observable* implémentée dans *java.util* ne pouvait pas être utilisée puisque *Physician* et *Patient* héritaient déjà de *Human*. Nous avons donc recodé le pattern en utilisant des interfaces.

3.2.3 Modélisation des files d'attente

Concernant les étapes du workflow, nous avons hésité entre l'implémentation des files d'attente comme étant des salles physiques (ie objets de type Room) ou comme étant des files virtuelles (ie des array de patients), indépendamment de la salle où les patients se trouvent. Nous avons opté pour la deuxième solution. Elle a le désavantage de complexifier l'implémentation du workflow, mais a le mérite d'autoriser une étape du workflow a traiter des patients dans plusieurs salles (par exemple pour la consultation, le patient peut être en BoxRoom ou ShockRoom). Cette solution permet également d'avoir plusieurs patients pour différentes tâches dans la même salle.

4 Implementation

4.1 Démarche

Pour l'implémentation, nous avons commencé par créer l'ensemble des classes présentes dans le diagramme UML. Nous avons ensuite écrit le contenu de ces classes, en commençant par les ressources. Ensuite, nous avons poursuivi avec l'*EmergencyDepartment* pour Thomas et le *Workflow* pour Quentin. Les tests unitaires ont été implémentés en parallèle, en alternance avec l'avancement du code.

Nous avons suivi un processus itératif : en codant, nous nous rendions compte que certaines parties de notre architecture étaient inapropriées ou incomplète. Nous modifiions alors le diagramme et nous reprenions le processus. Nous allons présenter ces itérations dans la partie suivante.

4.2 Difficultés

Une des itérations que nous avons effectuées est le réarrangement de la structure du code. Nous avons à deux reprises réorganisé les packages du projet, pour avoir une hierarchie plus pertinente.

Nous nous sommes également rendus compte que certaines instances avaient besoin d'accéder à d'autres pour le bon fonctionnement de la simulation. Dans les classes correspondantes, nous avons donc rajouté une référence à l'instance d'*EmergencyDepartment*. Celle-ci permet d'accéder à toutes les instances utiles à partir de l'*EmergencyDepartment*.

L'implémentation de la temporalité à également apporté de grands changements dans l'architecture initiale. Nous avions initialement prévu des méthodes telles que *executeServiceOnPatient* pour les étapes du Workflow. Cette méthode effectuait l'intégralité du traitement, du début à la fin. En mettant en place le design pattern, nous avons remplacé cette méthode par les méthodes *startServiceOnPatient* et *endServiceOnPatient*. Ces méthodes, qui intéragissent avec les classes *StartService* et *EndService* du Command Patern, permettent de séparer le début du traitement et la fin, pour que le service puisse lancer plusieurs traitements en parallèle (lorsque plusieurs ressources sont disponibles par exemple).

Toujours à propos du Workflow, des difficultés plus précises ont été rencontrées tout le long du projet. C'est effectivement la partie la plus critique du système. Ne pouvant lister de manière exhaustive toutes ces difficultés, nous allons présenter les plus délicates :

— Lorsqu'un patient doit être transporté dans une salle, la salle ne doit pas devenir occupée pendant le transport du patient. La solution que nous avons trouvé est la suivante : l'étape de transport ne commence que si le personnel

et la salle sont disponibles. Dans ce cas, le patient est ajouté à la salle même s'il ne s'y trouve physiquement pas encore. Ceci permet d'éviter d'attribuer la salle à quelqu'un d'autre pendant le transport, ou que la salle ait précédemment été attribuée à un autre patient qui n'est pas encore arrivé dedans au moment de la vérification. En revanche, nous avons choisi de ne mettre à jour la localisation du patient (attribut de *Patient*, différent de la liste des patients, attribut de *Room*) qu'à l'arrivée du patient, notamment pour avoir un historique cohérent.

— Les processus de vérification du workflow sont assez complexes :
  — En *Consultation*, si le patient est déjà suivi par un médecin, il faut vérifier que celui-ci est disponible. Sinon, il faut vérifier qu'un médecin quelconque est disponible.
  — En *Transportation*, il faut récupérer l'examen attribué lors de la Consultation par le médecin, puis vérifier qu'une salle correspondant et qu'un transporteur sont disponibles.

  Pourtant, le reste de la phase de vérification est toujours identique : si la condition est remplie, on indique à la *TaskQueue* que la prochaine tâche de ce *WorkflowElement* est le début de traitement de ce patient. Sinon, on indique que c'est la prochaine fin de traitement. Après quelques essais et erreurs, nous avons affiné l'architecture. La classe abstraite *WorkflowElement* implémente une méthode *getNextTask*, qui retourne la prochaine tâche selon la méthode de décision définie juste au dessus. Cette méthode appelle une autre méthode *canTreatPatient*, pour vérifier s'il est possible de commencer le traitement du patient passé en argument. Cette dernière méthode est abstraite. Elle est implémentée dans chaque classe héritée de *WorkflowElement* pour traduire la complexité et la variété des verifications, comme introduit au début de ce paragraphe.
— A l'issue de son examen, le patient doit être ramené en consultation. L'énoncé semble ignorer cette étape de transport pour le retour en consultation. Nous nous sommes rendu compte que notre Workflow n'était pas compatible avec un retour direct en consultation puisque les vérifications de disponibilité des salles sont faites dans l'étape d'Installation. Nous avons alors prit l'initiative de renvoyer le patient en début d'étape d'Installation, pour le réinjecter sans erreur dans le Workflow. Ceci a également l'avantage d'ajouter plus de réalisme à la simulation. D'un point de vue performance simulée, notre département des urgences sera moins performant (au sens des KPI) avec cette nouvelle étape puisqu'elle nécessite une infirmière et rajoute du temps d'attente.

4.2.1 Chaines de caractères ou classes

Cette section traite d'un choix que nous avons fait, toujours par souci de simplicité. Il s'agit de représenter certaines informations en temps que chaines de caractères, plutôt que de créer des classes. Il s'agit par exemple de l'état des patients, des médecins, ou encore des messages dans la *messageBox* des médecins. Nous aurions pu créer des classes pour chaque, mais celles-ci auraient demandé plus de complexité dans le diagramme UML pour un apport nul en fonctionnalité. Nous avons conscience qu'il serait plus rigoureux de créer des classes, notamment pour l'état des humains. Ici, il est possible d'entrer n'importe quelle chaîne de caractère, alors que nous pouvons contrôler les états possible en créant des classes (à la manière de l'état de gravité d'un patient). Toutefois, l'état n'est modifié que par des méthodes, et non par la main de l'homme. A notre niveau, nous considérons que ceci est suffisant en terme de sécurité. Ayant pleine conscience de ces aspects, nous avons prit délibéremment le choix de la simplicité avec les chaînes de caractères.

5 Tests Junit

5.1 Présentation de la démarche

Nous avons mis en place les tests unitaires en parallèle du développement du système. Nous avons respecté le principe suivant : celui qui code n'est pas celui qui teste. Selon les classes testées, les tests ont des dimensions considérablement différentes. Nous avons prit le parti de ne pas tester les getters et les setters pour chaque méthode lorsqu'il n'y avait pas de raison précise de le faire. En effet, ceux-ci sont générés automatiquement par Eclipse, donc ne laissent pas de place à l'erreur humaine. Ainsi certaines classes comme les *Severity Levels* n'ont pas de tests unitaires car elles n'ont pas d'autre contenu que des getters et setters. Il existe également d'autres classes qui n'ont pas réellement été testées unitairement car interagissent beaucoup trop avec d'autres classes. C'est le cas pour les classes héritant de *WorkflowElement* ou de l'*EmergencyDepartment*. Dans ce cas, nous les testons plutôt dans un test d'intégration.

Nous avons les cas particuliers suivants pour les tests :

— Les tests des factories se font dans l'initialisation des tests d'humains.
— Les tests des distributions de probabilité se font par comparaison de la moyenne empirique sur 100 000 échantillons avec le paramètre théorique, en incorporant une marge de +/-5%
— Les tests de méthodes d'interfaces se font en utilisant une des classes héritées.
— Le test des *WorkflowElements* est complexe car beaucoup d'interdépendance entre les classes et avec les autres classes. Il y a beaucoup d'objets à initialiser

et il faut simuler le début du service pour tester sa fin. Cela peut provoquer des erreurs potentielles non dues à la fonction testée. Nous effectuons des tests d'intégration.

5.2 Difficultés, tests ayant échoué et difficultés à corriger

Parmi les tests que nous avons voulu mettre en oeuvre, nous avons parfois rencontré des difficultés à les implémenter ou à corriger les erreurs lorsque ceux-ci ont échoué. Cependant, la plupart des erreurs ont été corrigées facilement. Il s'agissait d'oublis. Par exemple, en voulant tester les *ArrayLists* du workflow, nous avons rencontré des erreurs sur les patients. La cause était que nous n'avions pas implémenté de méthode *equals* pour les patients.

Parmi les problèmes qui ont demandé un peu plus de recherches avant d'être corrigé, il y avait :

- Des patients traités n'étaient pas retirés de la salle correspondant. Il y avait possibilité d'avoir un blocage du système à cause de cet oubli.
- Dans le workflow (installation), une condition ne vérifiait pas si le patient était non nul. Ceci pouvait aussi faire crasher la simulation de manière aléatoire.
- Lors de l'envoi des patients dans les différentes salles d'examen, ceux-ci allaient toujours en MRI. La cause était simplement le traitement de la fin d'event qui été effectué avant d'assigner la salle où aller. En conséquence, la salle attribuée était null. Donc les conditions testées étaient toutes fausses, et le patient allait en MRI par défaut.

6 Scénario d'évaluation

6.1 Description

Nous avons imaginé et implémenté différents scénarios afin de tester le simulateur.

Le premier est un scénario semi-automatique, qui permet de tester la simulation à évènements discrets en choisissant le nombre d'étapes à simuler et en suivant le premier patient arrivé. Il se trouve dans la méthode **main() de la classe SemiAutomaticUseCase du package test**. L'hôpital est d'abord initialisé avec :

- 5 salles d'attente d'une capacité de 10 personnes
- 1 salle de chaque type autre que les salles d'attente, d'une capacité d'une personne
- 4 médecins
- 4 infirmiers
- 4 transporteurs

— Les distributions de probabilité et les coûts par défaut de la classe *EmergencyDepartment* pour l'arrivée des patients, la durée des étapes et la répartition des niveaux de sévérité.

A la fin de la simulation le rapport du premier patient et de l'hôpital sont affichés pour vérifier que tout s'est déroulé comme prévu.

Le deuxième scénario est manuel afin de pouvoir choisir les actions effectuées et leur ordre pour tester plus finement le déroulement du traitement d'un seul patient. Il est contenu dans la méthode **main() de la classe ManualUseCase du package test**. L'hôpital est initialisé comme précédemment. On fait arriver le patient puis on le fait passer dans les différentes étapes successives du workflow, ce qui permet d'afficher son rapport à tout moment si on le souhaite (la version présente de base dans le projet ne l'affiche cependant qu'à la fin pour plus de lisibilité).

6.2 Problèmes rencontrés

Une des difficultés pour l'implémentation de ces scénarios a été de gérer l'automatisation de la simulation de l'hôpital. Nous avions en effet réalisé un premier scénario manuel avant de refondre la gestion des évènements avec l'implémentation d'une simulation à évènements discrets pour permettre de l'automatiser. Nous avons donc réécrit ce scénario pour tester la simulation, mais cela était moins flexible quant au choix des actions à effectuer (par exemple les arrivées de patient) et pour vérifier que les actions se déroulaient comme souhaité. Nous avons donc récupéré toutes les tâches effectuées automatiquement à chaque étape de la simulation, puis nous les avons regroupé dans la méthode *executeService* de la classe *ManualUseCase* afin de pouvoir faire passer un patient dans un service choisi. Cela nous a permis de réécrire un scénario en choisissant les différentes étapes pour tester plus en profondeur le bon fonctionnement des mécanismes de la simulation.

6.3 Lancer les tests

Voici la démarche à suivre pour lancer les scénarios d'évaluation décrit dans la partie 6.1 :

1. Importer l'archive du projet dans Papyrus
2. Se rendre dans le dossier src/test
3. Executer les fichiers SemiAutomaticUseCase.java et ManualUseCase.java

7 Conclusion

Nous avons donc réalisé un système de simulation d'urgences médicales. En partant d'un dossier présentant les exigences du simulateur, nous avons construit une architecture logicielle. Nous avons ensuite développé le logiciel en suivant l'architecture construite. Nous avons mené un processus itératif, revenant régulièrement sur l'architecture lorsque nous rencontrions des problèmes ou des imprévus dans le développement.

Nous avons maintenant un système entièrement fonctionnel, qui gère toutes les ressources nécessaires de manière automatique, à travers le workflow. Il reste cependant des fonctionnalités à implémenter.

Parmi celles-ci, nous avons pour objectif de proposer une interface utilisateur par lignes de commandes. Celle-ci permettra à l'utilisateur de lancer des scénarios de tests, des simulations et d'obtenir des informations sur le système d'urgence simulé. Nous devons en particulier implémenter le calcul d'indicateurs de performances (KPI) pour que l'utilisateur puisse mesurer les performances du système d'urgences en fonction des ressources allouées. Nous prévoyons également d'écrire plus de scénarios de tests, notamment pour vérifier l'allocation des ressources lorsque certaines sont indisponibles (médecin, infirmière, salles...). Nous souhaitons aussi améliorer la préemption pour les patients prioritaires (pour éviter la famine des patients les moins grave). De même, nous pouvons améliorer la sélection de tâche en Consultation. Si le médecin du premier patient est occupé, nous pouvons chercher un autre patient, qui n'a pas encore de médecin, et le prendre en charge directement.

Annexe

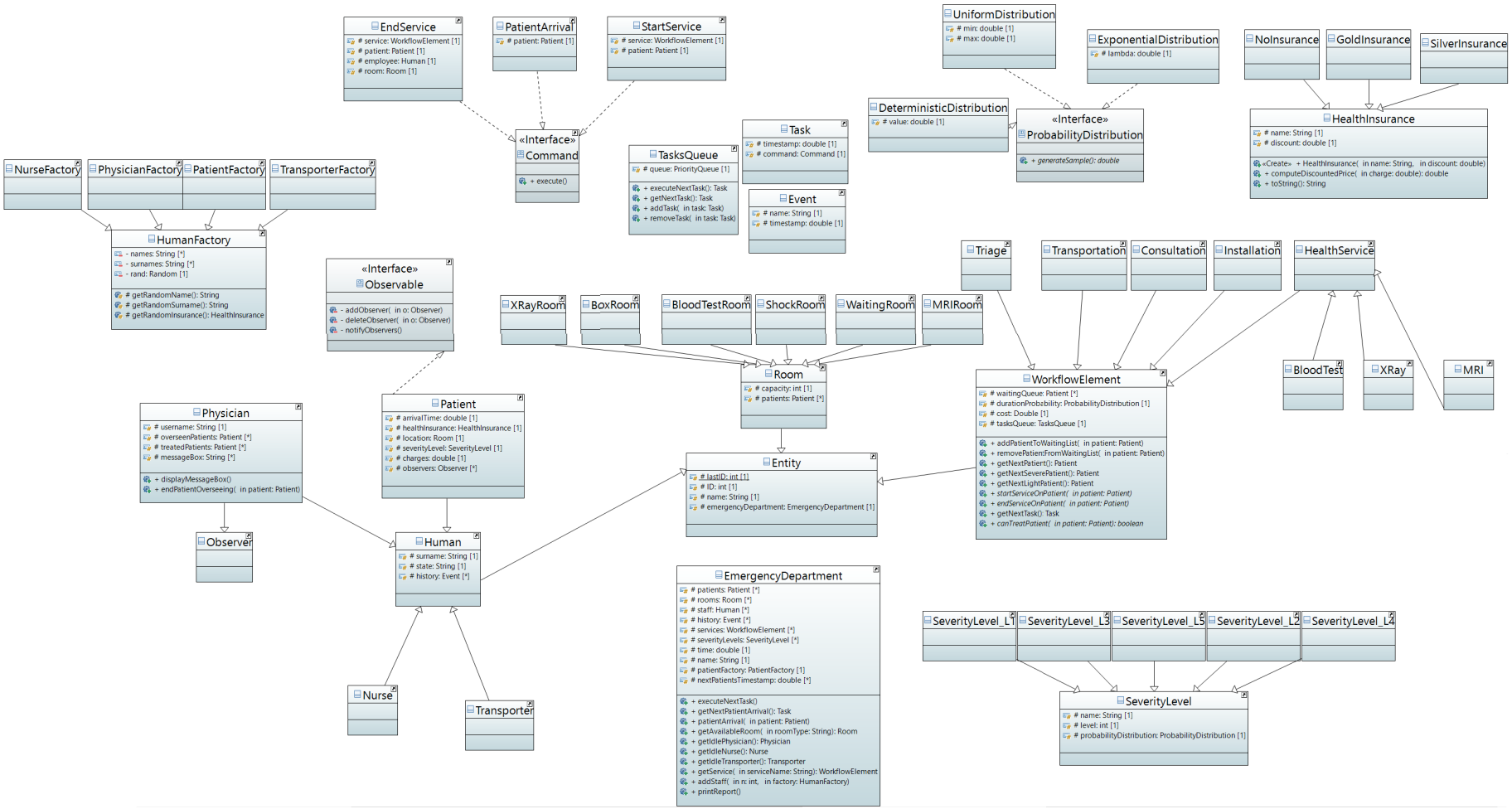

FIGURE 4 – Diagramme UML du projet