IS1220 Object Oriented Software Design

Rapport de projet

SimErgy

Donatien CRIAUD
21/11/2017

Préface

Ce projet s'inscrit dans le cadre du cours IS1220 – Object Oriented Software Design dispensé à CentraleSupélec dans le cursus Ingénieur Centralien. Je suis actuellement en stage en entreprise à temps complet. Compte tenu de cette situation particulière, je ne n'ai pas pu suivre les cours et les séances de travaux dirigés dispensés sur le campus de l'Ecole mais j'ai fait en sorte de pouvoir suivre ce cours à l'aide notamment des documents mis à disposition sur la plateforme Claroline. Il a été convenu avec l'enseignant M. Ballarini que j'effectue la totalité du projet seul. Ne pouvant bien évidemment pas décrire l'organisation de la réalisation du projet en binôme, je détaillerai tout de même dans ce rapport les détails de mon organisation personnelle pour réaliser ce projet. Je remercie tout lecteur de la compréhension qu'il pourra avoir de ma situation.

Les diagrammes de classe représentés dans ces documents sont présents dans le dossier*. /model* **et il est possible de les ouvrir en cliquant les images** de ce dossier (chaque diagramme est linké à son document image). J'espère que cela permettra une plus grande lisibilité du rapport.

Enfin, j'utiliserai le français pour rédiger ce rapport mais j'emploierai l'anglais pour désigner les termes techniques relatifs au langage JAVA, la programmation orientée objet ou encore le nom des entités présentes dans SimErgy. J'écrirai en italique ces mots afin de donner une plus grande fluidité au rapport.

Table des matières

1 Introduction générale

Les centres hospitaliers sont actuellement sujets à la polémique. ARTE diffusait le 26 octobre 2017 un documentaire dans lequel un professionnel de santé en activité à l'hôpital Saint-Louis décrit « un système complètement fou ». Ce dernier parle d'une « pure gestion financière » des hôpitaux, délaissant l'humain. Dans un contexte où le gouvernement français affiche une volonté de redéfinir la politique d'attribution des subventions aux hôpitaux en favorisant l'efficacité des établissements, il paraît important d'étudier l'impact d'une telle politique et du fonctionnement d'un hôpital de manière générale. Un premier moyen d'analyse des performances d'un hôpital est la simulation, par laquelle il est possible de représenter un service et ses différents attributs. Outre ses médecins, un service comprend aussi une multitude de personnes et d'équipements médicaux. Simuler toutes ces parties prenantes pourrait permettre de prononcer des estimations de critères de performance, tout en cherchant à minimiser le coût du service et à alléger la charge de travail du personnel soignant. L'ingénieur a dans cette situation une position idéale pour répondre aux problématiques actuelles concernant les hôpitaux français.

Le projet *SimErgy* a pour prétention de développer un outil de simulation d'un service d'urgences. Ce service joue un rôle clé dans un hôpital dans le sens où il est le premier contact entre un patient et les services de santé en cas d'un accident. Ainsi, il doit être actif en permanence et apte à traiter tout patient arrivant de manière imprévue. L'outil *SimErgy* doit donc être en mesure d'évaluer des critères de performance noyés dans un processus complexe. Pour cela, *SimErgy* doit représenter un service d'urgence et toutes les parties le composant à n'importe quel instant. L'interaction entre les patients, médecins, infirmiers, ressources etc… doit donc être implémentée.

Ce projet fut très intéressant d'un point de vue de la réflexion autour de l'architecture de ma solution. Il m'a été indispensable d'utiliser certains design patterns pour implémenter une solution qui soit aussi bien simple qu'efficace. En effet, j'ai gardé tout au long de ce projet la volonté d'encapsuler au maximum mes classes (*open-close principle*). Cela me fut notamment utile lorsque j'ai dû rajouter une classe de ressource oubliée lors d'une première implémentation. J'ai porté une attention toute particulière à l'héritage et au polymorphisme de mes classes pour pouvoir par la suite designer le simulateur de la manière la plus générique possible. Cela m'a amené à modifier une première structure événementielle de *SimErgy* car je jugeais sa conception trop « ouverte » et insuffisamment encapsulée.

2 Généralités

2.1 Architecture du *core* de SimErgy

SimErgy possède un *core* composé de 5 packages :

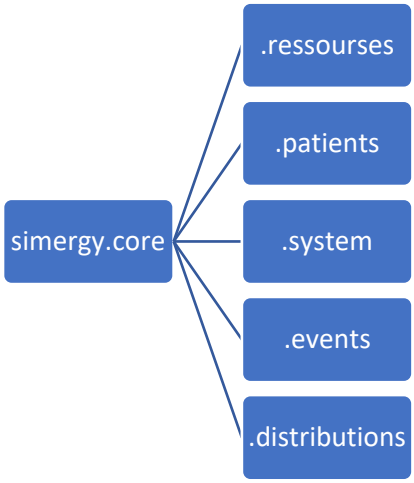

FIGURE 1 : ORGANISATION DE SIMERGY.CORE

2.1.1 Ressources

Le package *simergy.core.ressources* contient les classes relatives à la représentation de toute ressource présente dans un service d'urgences. J'ai considéré comme une *Resource* toute entité intervenant dans l'*ED* et qui ne soit pas un patient. Pour créer ce package, j'ai choisi d'utiliser au maximum la généricité. Comme *Resource* est un type très générique qui peut être étendu (humains et non humains) et que chacun de ceux-ci possède des propriétés relativement semblables, il apparait logique d'utiliser un ***design factory pattern***. Cela m'a notamment permis par la suite de gérer les événements du service d'urgence indépendamment du type de ressource nécessaire pour la réalisation de ceux-ci.

HealthService : Ces services possède une file d'attente, un coût et une distribution de probabilité. Ils sont utilisés pour représenter les équipements d'examen d'un *ED* (***Radiography***, ***MRI***, ***BloodTest*** etc…)

HumanResource : Ces ressources représentent les employés de l'ED. Ils partagent des caractéristiques telles que name et surname.

Nurse : Les *Nurses* sont chargées de la *Registration* d'un patient puis de la *Transportation* de celui-ci vers une *BoxRoom* ou une *ShockRoom* (selon l'état de santé du patient).

Transporter : La structure de cette classe est identique à la précédente, seul le rôle du Transporter diffère. Celui-ci amène le Patient à un service d'examen après sa consultation par un *Physician*.

Physician : Le *Physician* est chargé de visiter un patient pendant une *Consultation.* A la suite de cet événement, il donne un avis médical (le patient doit passer un examen médical ou alors peut être libéré). Lorsqu'un patient finit son examen, le *Physician* est alerté et une nouvelle *Consultation* a lieu. Je pense que cette classe aurait pu hériter de la classe *HealthService* dans le sens où elle possède un attribut *waitingQueue*. Cependant, j'ai jugé plus approprié de laisser l'architecture comme telle dans le cas où d'autres paramètres relatifs à l'humain seraient à implémenter dans *SimErgy*. Cela a été utile lors de l'implémentation de la *CLUI* et de sa *commandFactory*. En effet, la commande de création d'un *Physician* est très semblable à celle de *Nurse* ou *Transporter.*

Room : Ce sont les endroits où les patients sont consultés par un Physician.

BoxRoom : Utilisée pour les patients dans un état peu sévère.

ShockRoom : Utilisée pour les patients dans un état grave.

WaitingRoom : J'ai initialement créé cette classe pour représenter l'endroit ou un patient attend sa *Registration* mais comme cette salle a une capacité infinie, je ne l'ai pas employée par la suite. En effet, cette *Resource* n'est pas « bloquante ».

2.1.2 Patients

Le package *simergy.core.patients* ne contient qu'une unique classe *Patient* car il est principalement utilisé pour représenter les patients. Les autres éléments de ce package sont des *enum* utilisées pour décrire plusieurs attributs du patient. Chaque patient possède un attribut *workflow* qui stocke toutes les informations utiles durant son séjour dans le service

(historique des événements, *Physician*, *charges* etc…). En résumé, Patient possède les principaux attributs :

- **severityLevel** : L1, L2, L3, L4 ou L5 correspondant à la sévérité de l'état du patient.
- **healthInsurance** : GOLD, SILVER ou NONE correspondant à l'assurance du patient.
- **patientState** : W(waiting), V(visited), E(examinated) ou R(released) correspondant à son statut dans l'*ED*.
- **charges** : le prix dont il devra s'acquitter à la sortie de l'*ED*.
- **physician** : le *Physician* étant en charge de ce patient durant son séjour dans l'*ED*.
- **workflow** : l'objet *Workflow* qui stocke les informations relatives au événements

2.1.3 Distributions

Le package *simergy.core.distributions* contient les outils utilisés lors de la génération aléatoire de durées d'événements ou d'arrivées de nouveaux patients dans le service. Le package se compose :

- D'une interface ***ProbabilityDistribution*** possédant une méthode *generateSample()* utilisée pour générer un timestamp suivant une loi de probabilité donnée.
- De classes implémentant l'interface et représentant les différentes probabilités.

2.1.4 Système

Le package *simergy.core.system* est utilisé pour la gestion du système *SimErgy*.

La classe ***EmergencyDept*** représente le service dans sa totalité. Elle contient toutes les informations « représentatives » de l'ED :

- ***workflows*** : *ArrayList* des workflows s'exécutant dans l'ED, permettant d'accéder notamment aux événements en cours.
- ***time*** : timestamp actuel.
- ***patientGenerator*** : Classe utilisée pour générer les instances des nouveaux patients dans l'ED. A chaque arrivée d'un patient de niveau de sévérité donné, la classe génère le timestamp d'arrivée du prochain patient en fonction de la distribution de probabilité associée. Cette fonctionnalité peut être désactivée.
- ***resources*** : stocke les instances des ressources de l'ED. La structure employée est renvoyée par la fonction *generateResources()* ci-dessous :

```java
/**
    * Generates the HashMap containing all the resources.
    * The index is made of Strings representing a canonical type of the
resource.
    *
    * @return the hash map of the ED's resources.
    */
   public static HashMap<String,ArrayList<Resource>> generateResources(){
         HashMap<String,ArrayList<Resource>> resources = new
HashMap<String,ArrayList<Resource>>();
         resources.put("PHYSICIAN",new ArrayList<Resource>());
         resources.put("NURSE",new ArrayList<Resource>());
         resources.put("TRANSPORTER",new ArrayList<Resource>());
         resources.put("BOXROOM",new ArrayList<Resource>());
         resources.put("SHOCKROOM",new ArrayList<Resource>());
         resources.put("WAITINGROOM",new ArrayList<Resource>());
         resources.put("STRETCHER",new ArrayList<Resource>());
         resources.put("MRI",new ArrayList<Resource>());
         resources.put("BLOODTEST",new ArrayList<Resource>());
         resources.put("RADIOGRAPHY",new ArrayList<Resource>());
         return resources;
   }
```

Cette méthode statique créer une structure de type *HashMap* qui est initialisée avec un *String* contenant le type générique de la ressource et un *ArrayList* qui contiendra après ajout les ressources correspondantes. J'ai choisi de laisser la généricité `ArrayList<Resource>` plutôt que d'écrire – par exemple – `ArrayList<Physician>` afin de faciliter par la suite la gestion des ressources de l'*ED*.

La classe ***SimErgy*** est utilisée pour la représentation d'un état de SimErgy (cette classe correspond au Model d'un *MVC Pattern*). Pour cela, il possède l'attribut ***EDs*** qui est un *Array* des *ED* crées par l'utilisateur. Lors de la sauvegarde d'une configuration, c'est un objet *SimErgy* qui est sérializé avec ses différents *ED*. C'est pourquoi l'on retrouve dans mes interfaces utilisateur la distinction entre un système (objet SimErgy) et un ED (objet EmergencyDept).

2.1.5 Evènements

Le dernier package du noyau *SimErgy* est celui décrivant les événements. Un événement est représenté par un objet *Event.* Chaque événement instancie une classe fille (***PatientArrival***, ***Registration***, ***Transportation***, ***Consultation***, ***TestTransportation***, ***Examination***, ***Outcome***). Les principaux attributs de ces classes sont :

- ***startTime*** : timestamp du début de l'événement

- ***endTime*** : timestamp de la fin de l'événement
- ***type*** : String représentant le type de l'événement (*PatientArrival*, *Examination* etc…)
- ***resources*** : *HashMap<resourceType,resource>* qui décrit les ressources nécessaires pour que l'évènement soit activé

Chaque événement possède deux occurrences :

- Occurrence 1 :
  - *state* : NS (Not Started)
  - *occurenceTime* :
    - ✓ startTime si time < startTime
    - ✓ time si startTime < time
- Occurrence 2 :
  - state : IP (In Progress)
  - occurenceTime : endTime

```
>>> executeEvent myED
Time : 2.0
Executed event : Event [name=Consultation of patient n° 0, type=CONSULTATION, startTime=2.0, endTime=10.763208534576018, patient=0, state=IP]

# Key performance indicators for ED : myED
### Patients Released : 0.0/1.0
### DTDT = NaN
### LOS = NaN

>>> executeEvent myED
Time : 10.763208534576018
Executed event : Event [name=Consultation of patient n° 0, type=CONSULTATION, startTime=2.0, endTime=10.763208534576018, patient=0, state=AF]

# Key performance indicators for ED : myED
### Patients Released : 0.0/1.0
### DTDT = NaN
### LOS = NaN
```

FIGURE 2 : DEUX OCCURRENCES D'UN EVENEMENT

Sur la Figure 2 ci-dessus, on observe la première occurrence de l'événement « Consultation of patient n°0 » au temps *time=2.0*, l'événement a alors un statut *state=IP* (In Progress). La seconde occurrence a lieu au temps *time=10.763…*, l'événement passe alors au statut *state=AF* (Already Finished).

La classe ***Workflow*** comme évoqué précédemment, sert de « suivi du parcours du patient ». Autrement dit, cette classe sert à stocker les informations relatives à un patient (notamment l'historique des occurrences d'événements pour ce patient, les différentes informations utiles pour évaluer les critères de performances etc…). C'est grâce à son attribut *currentEvent* que l'événement en cours pour un patient est stocké. Le workflow n'intervient pas dans la demande d'attribution de ressource par un événement ni dans l'attribution de celles-ci par la classe *EmergencyDept* (se référer à la partie 3.2 traitant de la simulation afin de comprendre la séquence d'événements d'un workflow).

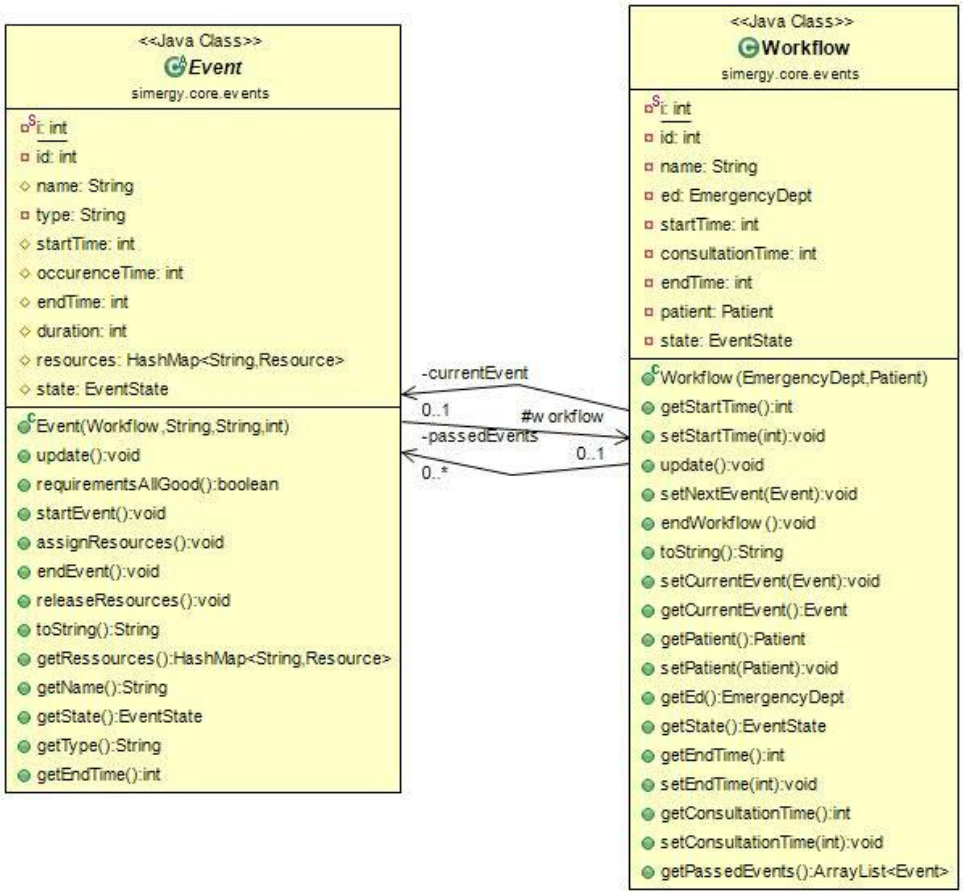

FIGURE 3 : RELATIONS ENTRE LES CLASSES WORKFLOW ET EVENT

2.2 Interface utilisateur et gestion de la sauvegarde

Afin de pouvoir exploiter le *core* de SimErgy, deux interfaces utilisateur sont disponibles. La première est l'interface en ligne de commande (CLUI) qui s'emploie comme un *shell* Unix classique. La seconde est une interface graphique (GUI) composée d'un menu permettant d'interagir avec SimErgy, d'un espace réservé à l'insertion de données et d'un dernier utilisé pour l'affichage.

J'ai essayé de créer ces deux interfaces en respectant le ***pattern Model View Controller***. En effet, j'ai voulu créer un *controller* indépendant du type d'interface employée. Pour cela il m'a d'abord fallu définir une classe abstraite ***userInterface*** afin de pouvoir utiliser la généricité de celle-ci pour créer une ***CommandFactory***. Je détaillerai plus cet aspect dans la partie 3.1.3.

Pour gérer la sauvegarde et le chargement de systèmes SimErgy, il a fallu implémenter la classe ***LoadSave.*** Lors de la sauvegarde, c'est l'objet SimErgy qui est sauvegardé (il contient toutes les informations du *Model*) avec l'extension *.ser* .

3 Analyse et Design

3.1 Design

3.1.1 Factory Pattern

3.1.1.1 Resource

La gestion des ressources se fait en employant un ***design factory pattern***. Ainsi, le package *simergy.core.resources* se compose d'une ***ResourceFactory*** chargée d'instancier les classes non abstraites. J'ai choisi de ne pas employer de *design abstract factory pattern* – bien que cela s'y prête plus en raison de la structure des *Resource* – afin de conserver au mieux l'*open-close principle* déjà mis en place. En effet, en cas d'ajout d'une nouvelle classe représentant une ressource de l'ED, un *factory pattern* est tout adapté dans le sens où cet ajout n'entraine qu'une modification légère de la *ResourceFactory*.

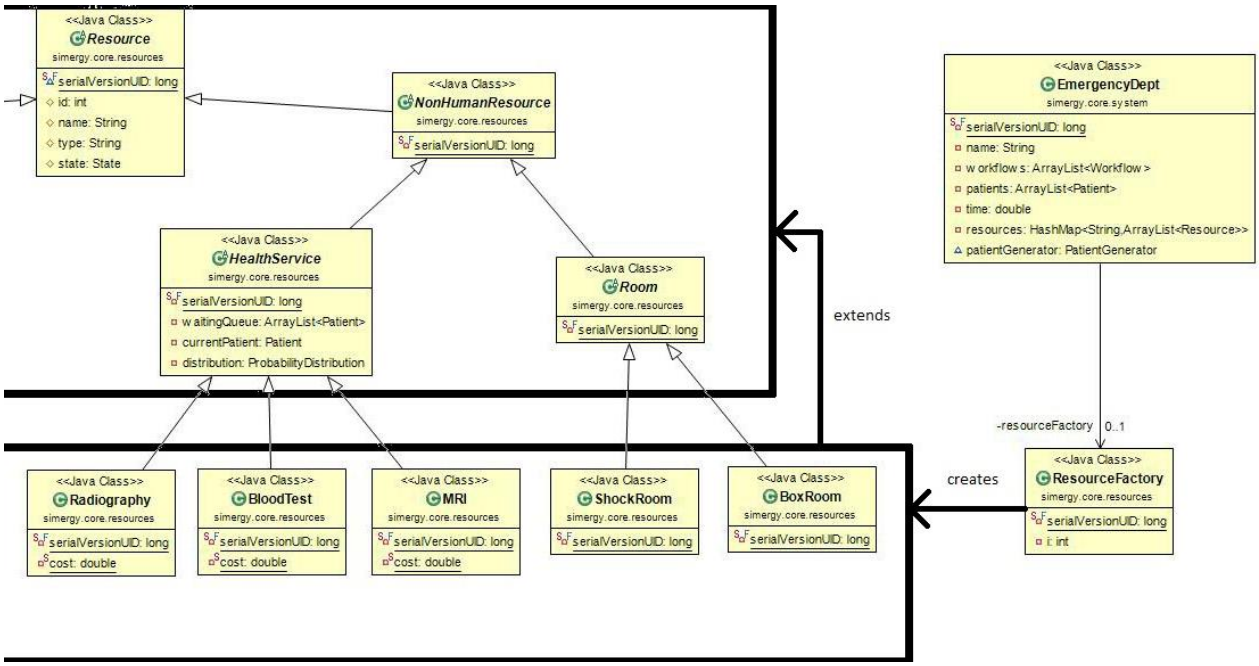

FIGURE 4 : FACTORY PATTERN EMPLOYE POUR LE DESIGN DES RESSOURCES

3.1.1.2 Command

Un *design factory pattern* est aussi utilisé pour la gestion des commandes des interfaces utilisateur. Cela m'a permis de n'avoir à créer qu'un seul *Controller* pour les deux interfaces employées (CLUI et GLUI).

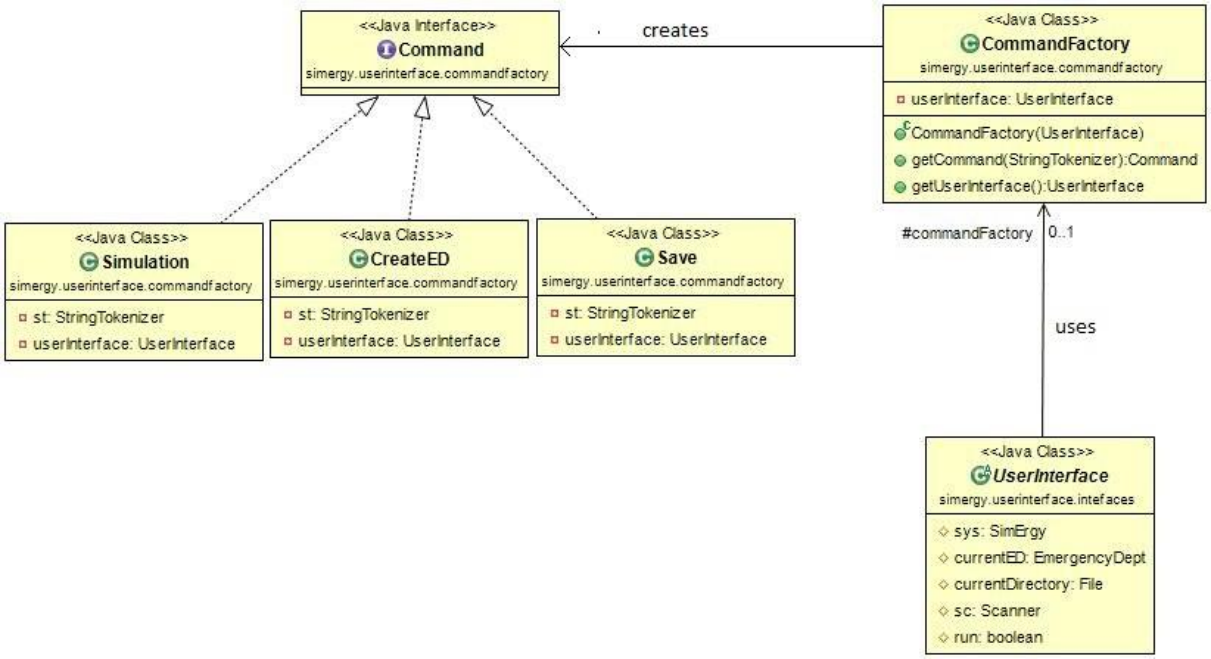

FIGURE 5 : FACTORY PATTERN EMPLOYE POUR LE DESIGN DES COMMANDES

3.1.2 Observer Pattern

Pour pouvoir informer un *Physician* que l'un de ses *Patient* a terminé un examen, j'utilise un design pattern observer. En effet, *Physician* est placé en *Observer* de l'*Observable Patient* et dispose de la méthode ***notify()*** afin d'ajouter ce patient à l'*ArrayLis*t des patients à consulter.

3.1.3 MVC Pattern

Afin d'implémenter une interface utilisateur, j'ai déployé une solution respectant le ***MVC pattern***. La Figure 6 ci-dessous décrit la structure de l'interaction CLUI-Command-SimErgy. Cette figure faite au brouillon avant l'implémentation m'a servi à bien respecter ce pattern.

L'implémentation de la GUI a été exactement la même concernant le MVC pattern puisque que cette classe **implémente la classe abstraite *UserInterface*** comme la CLUI.

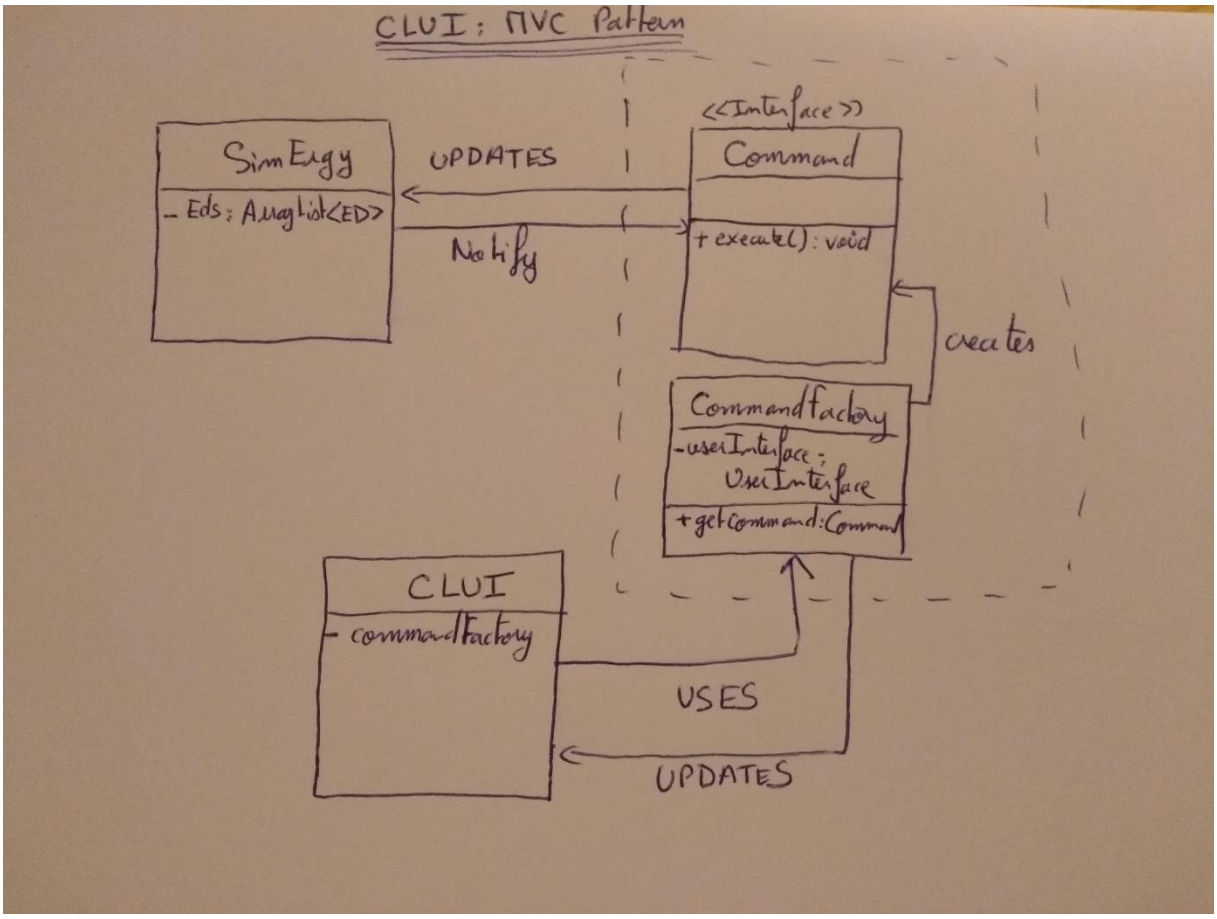

FIGURE 6 : MVC PATTERN EMPLOYE POUR LA CLUI

3.2 Simulation

3.2.1 Allocation des ressources

La classe ***EmergencyDept*** gère l'attribution des objets *Resource* à différents événements en profitant de la généricité du foctory patern. En effet, cela permet d'utiliser la généricité pour attribuer les ressources :

```
/**
 * Gives a resource and set its state to VISITING.
 *
 * @param resourceType the resource type
 * @return the resource
 * @throws ResourceNotAvailableException if the resource is not available
```

```
 */
public Resource giveResource(String resourceType) throws
ResourceNotAvailableException{
	/*
	 * Check si la ressource demandée est disponible ou non
	 * Si oui, change son état et la renvoie
	 * Sinon, leve une exception
	 */
	ArrayList<Resource> askedResource = resources.get(resourceType);
	for(Resource ressource : askedRessource){
		if(ressource.getState() == State.IDLE){
			ressource.setState(State.VISITING);
			return ressource;
		}
	}
	throw new ResourceNotAvailableException(resourceType);
}
```

Cette méthode est employée lors de l'exécution d'un événement pour attribuer les ressources nécessaires à son exécution et passer leur état de *Idle* à *Visiting*. Ici le principe d'encapsulation est utilisé afin de passer outre le type de chaque ressource. La méthode reçoit un String correspondant à la ressource demandée par un événement et renvoie cette ressource sans jamais considérer le type d'instance manipulée.

Ainsi, lors de l'occurrence 1 d'un événement, lorsque *Event.startEvent()* est appelée, la méthode *Event.assignResources()* l'est aussi et va parcourir le *HashMap* des *Resource* nécessaires à l'*Event*. Pour chacun de ces types de *Resource*, la méthode giveResource(String) dont le code est affiché plus haut est appelée.

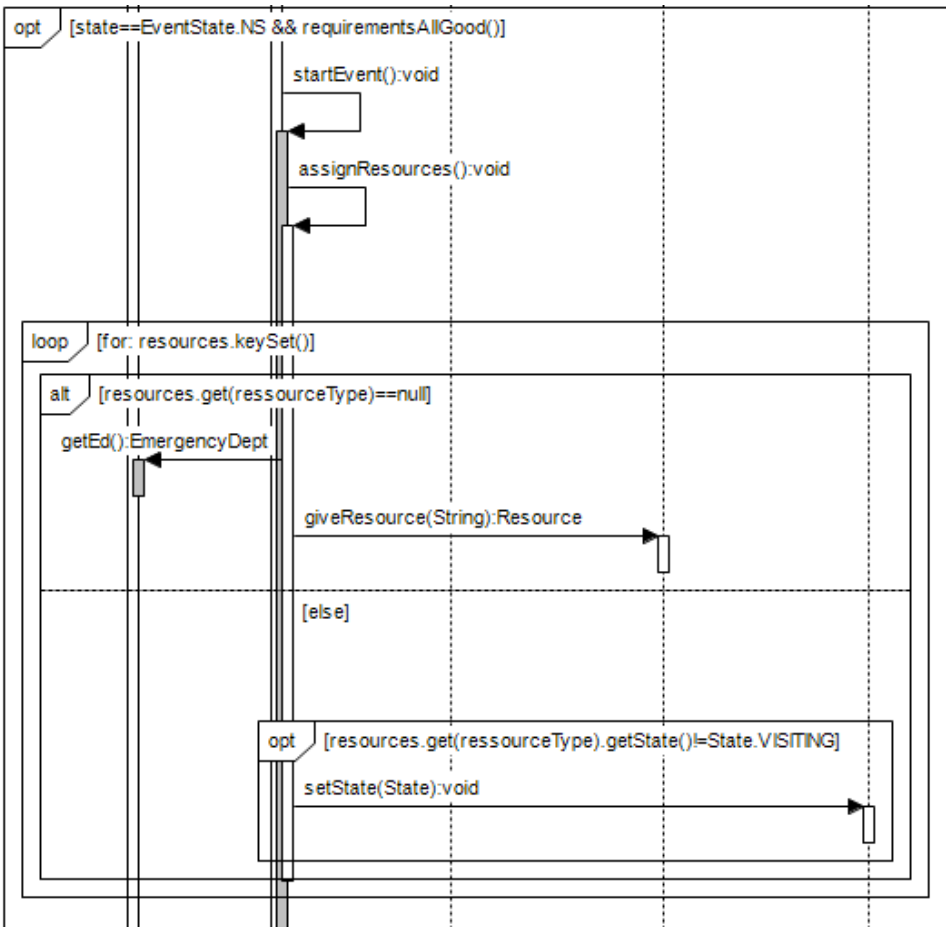

FIGURE 7 : ASSIGNATION DES RESSOURCES

3.2.2 Transition des événements

Lors de l'occurrence 2 d'un évennement, celui-ci se termine et un nouvel objet *Event* doit être instancié. Pour se faire, chaque objet *Event* possède une méthode ***createNextEvent()*** qui simplement instancie l'événement suivant en fonction de l'historique du workflow du patient. Par exemple une *Registration* instanciera ensuite une *Transportation* qui instanciera à son tour une *Consultation*.

4 Tests

4.1 Tests JUnit

Les tests sont des éléments indispensables faisant partie intégrante du processus de développement. Etant seul sur le projet *SimErgy*, je n'ai pas eu le temps de concevoir des tests pour toutes les méthodes. J'ai donc pallié à cela en testant les fonctionnalités des packages plutôt que les méthodes. C'est-à-dire que j'ai préféré tester une méthode appelant plusieurs méthodes de différentes classes du même package. Le test *StartEventTest* en est un bon exemple : Il permet de tester le bon fonctionnement de la méthode Event.*startEvent()*. Ainsi, on teste en même temps la méthode *Event.assignResources()* et la méthode *EmergencyDept.giveResource().* J'ai fait cela en adoptant le paradigme du Test Driven Development afin d'être certain des actions des différentes méthodes.

La principale difficulté pour tester les différentes méthodes vient de la forte synergie entre les classes et de leur grande interactivité. En effet, ma solution encapsule beaucoup la séquence d'événements que traverse un patient (génération automatique de l'évènement suivant, allocation automatique des ressources…). Les divers tests employés se trouvent dans le package *simergy.tests*. Ces tests m'ont permis de tester principalement le noyau *SimErgy* et l'automatisation des évènements. J'estime avoir couvert ainsi environ 75% des méthodes conçues dans *simergy.core*.

Les tests relatifs à l'interface utilisateur ont quant à eux été effectués via les différents Scénarios test présentés ci-après.

4.2 Scénarios Test

Afin de tester plus en détail *SimErgy*, j'invite le lecteur à lancer la méthode suivante :

simergy.userinterface.interfaces.CommandLineUserInterface.main(String[])

Cette méthode lance la CLUI. Plusieurs systèmes sont déjà enregistrés. J'ai pris le soin de préparer une sauvegarde my_simergy.ser permettant d'initier un ED basique si le lecteur veut effectuer des tests qui lui sont propres.

Dans le dossier test du projet se trouvent différents scénarios afin de tester SimErgy. En gras sont indiqués les commandes et les paramètres à saisir dans la CLUI.

5 Réalisation du projet

5.1 Outils utilisés

Afin de développer *SimErgy*, j'ai utilisé l'environnement Papyrus. La principale raison étant que celui-ci dispose d'une grande ergonomie qui facilite l'implémentation (auto-complétion, arborescence etc…). Bien que j'aie travaillé seul sur ce projet, j'ai utilisé des moyens employés habituellement sur des projets développés par plusieurs personnes. Afin de versionner mon code, j'ai utilisé le protocole GIT en local afin de garder une trace des différentes manipulations du code. Cela m'a aussi permis d'effectuer des push sur GitHub afin d'éviter de perdre mon travail en cas de problème de PC. Une fonctionnalité de Papyrus permet de lier un projet à son répertoire Git depuis l'interface, ce qui a été très utile pour effectuer des *pull/push* à l'ouverture et à la fermeture de l'application. Cela m'a permis d'effectuer mes *commit* sur mon GitHub (https://github.com/DonaCrio/simergy) de manière plus régulière.

5.2 Organisation du travail

Comme évoqué plus haut, j'ai utilisé la méthode du Test Driven Development afin de développer *SimErgy*. La Figure ci-dessous présente un exemple de processus pour tester le bon démarrage d'un évènement avec le test Junit *StartEventTest*, tout en implémentant la solution.

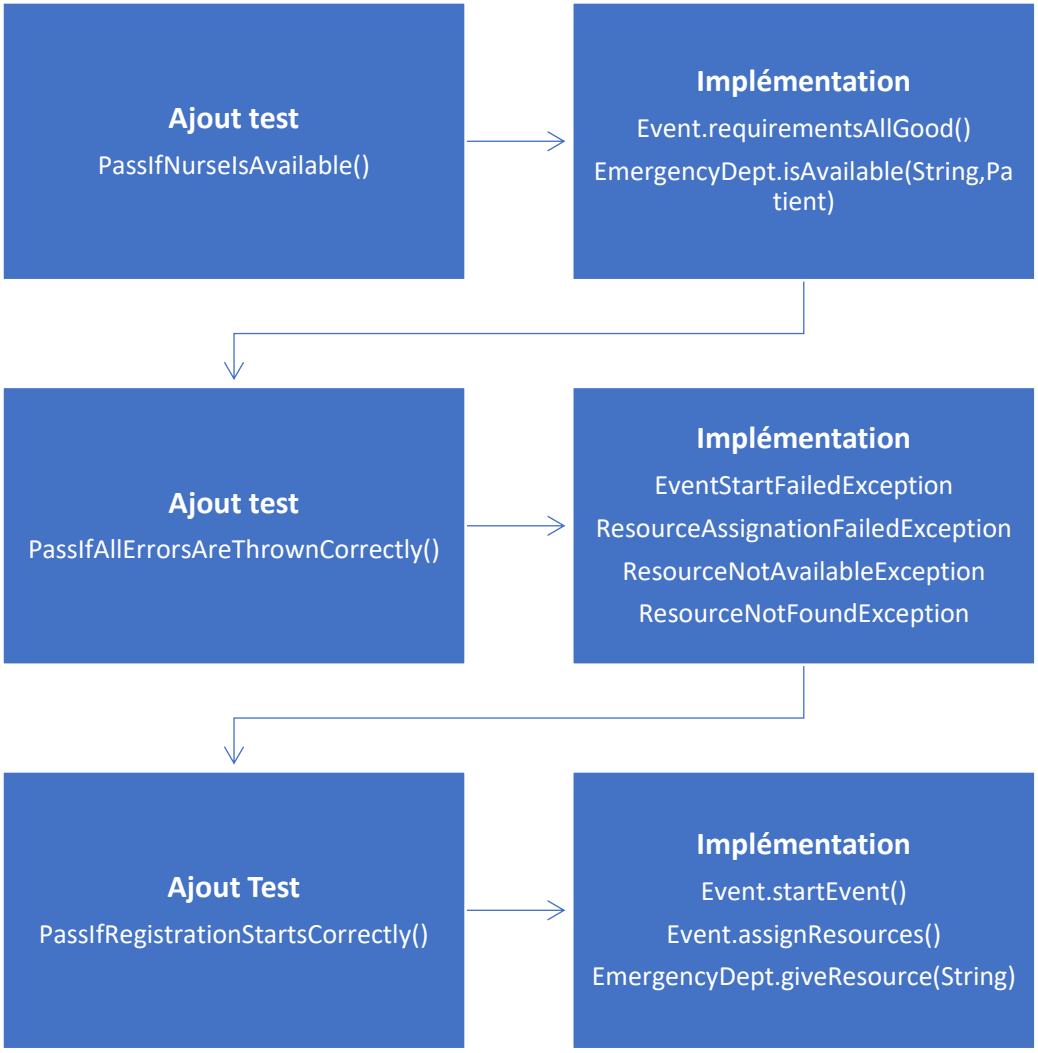

FIGURE 8 : EXEMPLE DE TEST DRIVEN DEVELOPMENT