Rapport de projet

Software Engineering

SIMULATION D'UN DEPARTEMENT D'URGENCES

Adnane Hamid & Sylvain cizaire

Décembre 2017

TABLE DES MATIERES

INTRODUCTION – BUT DU PROJET

Ce projet a été réalisé dans le cadre du cours de Software Engineering en deuxième année à CentraleSupélec cursus centralien.

Le cadre est le suivant. Nous supposons qu'un responsable du département d'urgences d'un hôpital (ED) fasse appel à nos services. Il souhaiterait un outil lui permettant de modéliser le fonctionnement de son département sous la plupart des aspects médicaux.

Plus concrètement, il pourra voir l'évolution du système au bout d'un certain temps en jouant sur de multiples paramètres, tels que : la durée de chaque opération (une radiographie par exemple), l'affluence et le degré de sévérité des patients qui arrive à l'ED, le nombre de salles, de docteurs, d'infirmières…

Il pourra même interrompre la simulation pour apporter son grain de sel pendant celle-ci, par exemple en choisissant lui-même quel docteur devra répondre à quelle tâche.

Il souhaite donc un tel livrable, testé et approuvé. C'est ce que nous lui apportons.

MODELISATION DE L'ED

Nous examinons dans cette partie comment nous modélisons un système très complexe tel qu'un ED dans une simulation informatique. Nous avons fait énormément de simplifications, qui seront détaillées au fur et à mesure.

Parcours type

Mais avant de rentrer dans les détails, voici le parcours type qu'un patient pourra suivre dans notre simulation (Fig. 1). Le patient arrive et fait la queue comme tout le monde (*Patient Arrival*). Puis il s'enregistre auprès d'une infirmière (*Registration*), et attend dans la salle d'attente que quelqu'un l'emmène vers une salle de consultation (*Transportation/Installation*) où il sera examiné par un docteur (*Consultation*). Ce dernier décide alors de l'examen dont a besoin le patient (si nécessaire), et celui-ci est dirigé vers une salle libre d'examen correspondante (*Transportation*), où il passe son examen (*Examination*). Un docteur vient alors regarder les résultats de l'examen et décide si des examens complémentaires sont nécessaires. Si non, il est alors considéré comme soigné et peut partir de l'ED (*Outcome*).

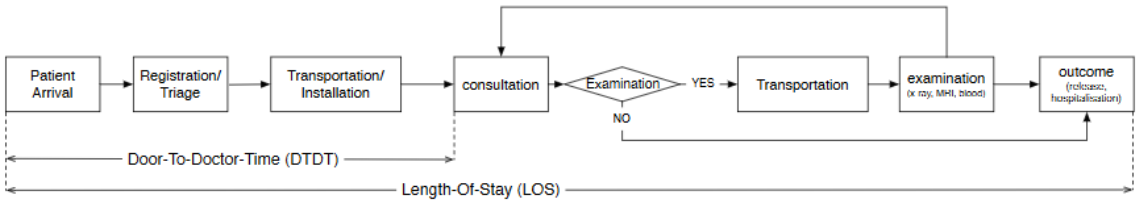


Fig. 1 – Parcours type d'un patient dans la simulation

Il est d'ores et déjà clair que des simplifications ont été faites : le patient ne peut suivre que des examens ; n'est jamais réellement soigné ; n'est pas hospitalisé ; doit toujours être examiné par un docteur alors que bien souvent l'infirmière pourrait déterminer de quoi a besoin le patient dès son enregistrement…
Pourtant, ce parcours type n'a pas été déterminé par nos soins et bien que nous aurions pu l'améliorer, ces simplifications ne sont en fait pas les nôtres mais celles du client.

Passons maintenant à une description plus concrète de notre modélisation

Etat & Ressources

- L'état du système

  Le système ED est à tout instant caractérisé par son état. Son état (classe *ED)* c'est l'ensemble de ses caractéristiques dans le cadre de la simulation. Ces caractéristiques sont toutes associées à des Ressources dans notre programme. C'est-à-dire qu'avec la sommes de toutes les ressources, on connait parfaitement l'état du système. Les ressources ont-elles-mêmes leur propre état : connaissant les états de toutes les parties prenantes du système, on connait l'état du système. Il faut distinguer les ressources humaines (classe *RessourceHR*) des ressources physiques (classe *RessourceNHR*), ainsi que le cas à part des *HealthServices*.

- Les Ressources humaines

  Les ressources humaines sont tous les patients qui sont dans l'ED ou qui arriveront plus tard dans la simulation, les docteurs (classe *Physician*), les infirmières (classe *Nurse*), et les transporteurs qui accompagnent les patients (classe *Transporter*). Dans chaque ressource humaine est stocké l'état de la personne : le patient est-il en train d'attendre ou d'être examiné ; l'infirmière est-elle disponible ?

- Les Ressources physiques

  Les ressources physiques sont toutes les salles de l'ED : salle d'attente, salle d'un certain type d'exam, salle de consultation, etc. Incorporer ces ressources dans la simulation nous permet de prendre en compte le fait que dans un ED, il ne peut pas y avoir trop de personnes dans une salle d'attente ou dans tout autre salle. Pour ce projet, nous avons fait la simplification que chaque salle, hormis les salles d'attente, ne pouvaient contenir qu'un seul patient.

- Les *HealthServices*

  Les *HealthServices* sont un type de Ressource à part. En effet, ils représentent les différents départements de l'ED : département radiographies, département des examinations, etc. Ils permettent de donner les paramètres avec lesquelles les évènements associés s'effectuent. Ils contiennent la *waitingList* qui permet de déterminer quel prochain patient pourra utiliser le service ; le coût de l'évènement, la durée de l'évènement, etc. C'est sur les HealthServices que l'on pourra modifier comment fonctionne l'ED.

Evenements

Maintenant que nous avons nos ressources, il nous faut un moyen de les faire interagir entre elles. Cela passe par le biais des évènements (classe *Event*).

Un évènement, c'est une occurrence qui a lieu à un certain moment de la simulation et tel qu'il y a un état pré-évènement et un état post-évènement. Par exemple, un évènement pourra modéliser le fait qu'un patient a fini sa radiographie. L'état du patient passera alors de *taking-exam* à *waiting-verdict*, sa location passera de la salle d'examen à une salle d'attente, et l'état de la salle d'exam passera d'occupée à libre.

L'intégralité de l'évolution du système passe par l'exécution des évènements qui, mis bouts-à-bouts, finissent par faire passer le système de son état initial à son état final.

Maintenant que les différents objets de notre simulation sont plus clairs, attaquons plus précisément le design de notre programme et comment il a été conçu.

SIMULATION AUTOMATIQUE

Dans cette partie, nous allons aborder le cas où l'utilisateur souhaite rentrer un état initial pour l'ED, et souhaite connaître l'état à un instant *tempsMax* : il n'interagit pas avec l'algorithme.

Entrons dans le vif du sujet : Le cœur de l'algorithme

Fonctionnement de la boucle principale

```
while (this.getSimTime() < tempsMax) {                    //(1)
      this.getNewEnabledQueue(this.getED());              //(2)
      this.getEventQueue().sort() ;                       //(3)
      Event nextEvent = this.dequeue() ;                  //(4)
      This.setSimTime(nextEvent.getTimestamp()) ;         //(5)
      nextEvent.getEventType().accept(this) ;             //(6)
      this.getEventHistory.add(nextEvent);                //(7)
            }
```

Fig. 2 – Code de la boucle de la simulation automatique

En peu de lignes est résumé tout le principe de la simulation automatique que l'on va détailler. Sommairement, il s'agit d'une boucle qui à chaque itération exécute un évènement.

Chaque évènement est en fait un couple (*eventType*, *timestamp*), où l'*eventType* décri tout le fonctionnement de l'event, et le timeStamp décri le moment où l'event aura lieu (en temps de simulation).

On s'inspire alors du modèle du discrete-event stochastic simulator (DESS). On va créer deux listes qui évolueront à chaque itération :

- La *EnabledQueue* est une liste dans laquelle apparait tous les *eventType* possible à un moment donné. Remarquons que cela peut rapidement devenir une grande liste : chaque paramètre différent crée un *eventType* possible. Par exemple, si on a 3 patients qui attendent un docteur pour une consultation, et qu'il y a 3 salles de consultation et 3 docteurs, cela fait déjà $3^3 = 27$ *eventTypes*.

- La *EventQueue* est une liste dans laquelle apparait tous les évènements qui sont prévus d'être exécutés dans le futur. Elle est ordonnée par ordre croissant de *timestamp* des évènements.

Ceci étant dit, examinons le fonctionnement de cette boucle.

(1) La simulation tourne en boucle jusqu'à ce que le temps interne à la simulation *simTime* atteint celui fixé par l'utilisateur *tempsMax.*
(2) On commence par actualiser la valeur de la *enabledQueue*, pour qu'elle soit bien en phase avec l'état actuel du système. On en profite pour ajouter les évènements associés à ces nouveaux *eventType* à notre *eventQueue*. Et de même, on retire de notre *eventQueue* les éléments qui ne sont plus associés à des *eventType* de la *enabledQueue*.
(3) On s'assure que la nouvelle *eventQueue* est bien ordonnée.
(4) On enlève le premier élément de la eventQueue et on le stocke dans un nextEvent. En effet, le premier élément de la eventQueue est toujours le prochain évènement à être exécuté par la simulation.
(5) On associe alors le temps de simulation au *timestamp* de l'évènement que l'on va exécuter.
(6) On exécute alors ce *nextEvent*. C'est-à-dire que l'on applique les transformations associées à son *eventType* à l'état du système. Cela passe par la méthode *accept du visitor pattern* associé. C'est cette étape qui est cruciale car c'est elle qui change l'état du système en fonction du type d'évènement qui doit être exécuté.
(7) Enfin, on ajoute à l'historique de notre simulation l'évènement que l'on vient d'exécuter. Puis, on relance la boucle !

Beginning events & ending events

Une question se pose alors : Comment le temps passe dans la simulation automatique ? C'est-à-dire, comment lorsque l'on créé un évènement on lui met la bonne valeur de *timestamp* ? Il nous faut introduire ici notre principe de *beginningEvents* et de *endingEvents*.

En fait, le principe du DESS doit être un peu modifié dans notre cas présent. En effet, ce qu'on imagine comme une « action » dans l'ED (par exemple, passer une radiographie) ne correspond pas à la définition que nous avions donné d'un évènement, i.e. tel qu'il y a un état pré-évènement et un état post-évènement. En effet, **il y a ici non pas deux états mais bien 3 états distincts** : le patient attend de faire sa radiographie, puis est en train de faire sa radiographie et enfin a fini sa radiographie. Voici comment nous avons contourné ce problème.

Chaque (ou presque) « action » dans l'ED est associée à deux évènements : un *beginningEvent* et un *endingEvent*. Le *beginningEvent* fait passer le système de l'état pré-action à l'état action en cours, puis le *endingEvent* fait passer le système de l'état action en cours à l'état post-action.

Pour cela, on choisi de n'associer la *enabledQueue* qu'aux *beginningEvents*. Ces derniers suivent donc le principe basique du DESS. En revanche, à chaque fois qu'un *beginningEvent* est effectivement exécuté (étape (6)), il créé automatiquement un *endingEvent* qui aura le *timestamp* : *timestamp* du *beginningEvent* + durée de l'action.

Ce *endingEvent* est alors ajouté à la eventQueue. En effet, les *eventTypes* associés aux *endingEvents* sont toujours possibles, c'est-à-dire qu'ils pourraient toujours être dans la *enabledEvent*. Cela s'explique par le fait qu'un *endingEvent* n'est que de la libération de ressource, il n'y a pas besoin de ressources supplémentaires qui pourraient être utilisées par une autre action en même temps de simulation.

Ainsi, en utilisant ce processus, on revient finalement au principe du DESS.

Precisions sur les methodes

Passons maintenant un peu plus de temps sur l'implémentation des méthodes utilisées dans la boucle de simulation automatique, en s'appuyant sur l'exemple d'un patient qui doit faire une radiographie.

Première étape

La première étape pour que cette action puisse être réalisée, c'est qu'elle soit possible : le *beginningEvent* associé doit être doit la liste des *enabledEvent*. Voici le code associé :

```
@Override
public void visit(Radiography e) {

        e.getPatient().setState("taking-xray-exam");
        e.getPatient().updateHistory(new PatientMilestone("taking-xray-exam",this.simTime));
        e.getLocation().fill();
        e.getPatient().setLocation(e.getLocation());
        e.messagePhysi(this.simTime);
        this.eventQueue.getQueue().add(new Event(new EndRadiography(e.getPatient(),e.getLocation()),this.simTime + e.getDuration()));
        e.getPatient().computeCharges(e.getHs().getCost());
        ed.getRadioService().getwList().remove(e.getPatient());

}
```

On effectue une boucle par ressource clé associée à l'action : Pour chaque type de ressource nécessaire à l'action, on regarde parmi toute la base de ressources lesquelles sont en état d'effectuer l'action. Puis finalement, on crée le *eventType* associé au *beginningEvent*. Dans notre exemple, le patient doit être en attente de sa radiographie, et une salle de radiographie doit être disponible.

Ensuite un évènement associé est ajouté dans la *eventQueue*. Son *timestamp* correspond simplement au temps dans la simulation : il n'y a aucune raison qu'il ne soit pas exécuté dès qu'il peut l'être.

Deuxième étape

Ensuite, l'évènement est exécuté. Il est bon de préciser ici les aspects de l'objet *eventType*. Voici ses attributs :

```
private String type;
private double duration;
private int cost;
private Patient patient;
private Physician physician;
private Nurse nurse;
private Transporter transporter;
private Room location;
private HealthService hs;
private boolean isBeginning = true;
```

Le point important ici est le booléén *isBeginning*. En effet, on lui donnera la valeur de true si c'est un *beginningEvent* et false si c'est un *endingEvent*.

Voici donc comment l'évènement est exécuté. Pour chaque eventType, on associe la méthode *accept(Visitor visitor)* qui permet au visitor de visiter cet évènement. Le visitor ici, c'est la liste *enabledEvents*. Pour notre exemple de radiographie, cela donne :

```
@Override
public void visit(Radiography e) {
    if (!ed.getRadioService().getwList().isEmpty()) {
    Patient p5=ed.getRadioService().getwList().get(0);
    for (int j=0;j<ed.getRadiographyRoomList().size();j++) {
        if (ed.getRadiographyRoomList().get(j).isDispo()) {

            this.enabledQueue.add(new Radiography(p5,ed.getRadiographyRoomList().get(j),ed.getRadioService()));
            break;
        }
    }
    }
}
```

Précisons que c'est ainsi que le temps passe dans la simulation : le *endingEvent* ainsi crée a pour *timestamp* le *timestamp* du *beginningEvent* auquel on ajoute la durée de l'action.

AUTRES REMARQUES

possibleQueue

En plus de *l'eventQueue* et de la *enabledQueue* qui changent à chaque itération de la boucle principale, on utilise une *possibleQueue* dans la classe *EnabledEvent*. Celle-ci est instantiée dès le début et permet d'utiliser notre visitor pattern non pas sur les caractéristiques précises des eventTypes, mais sur le type *d'eventType*. Par exemple, on ne visitera pas *l'eventType* (Mr. Michon fait sa radio avec le docteur Bougard), mais on visitera *BloodTest*.

Management des cas de conflits

Il peut arriver des « cas de conflit ». Par exemple, s'il y a un patient qui a besoin d'une *Consultation*, qu'un autre a besoin d'un verdict, et qu'il n'y a qu'un docteur de disponible, que doit faire le docteur ? Grâce à l'utilisation de notre *possibleQueue*, nous faisons en sorte qu'en cas de conflits, c'est toujours l'évènement le plus avancé dans le parcours typique d'un patient qui est prioritaire. Cela permet de libérer l'hôpital et d'optimiser le door to door time.

waitingQueue pour les docteurs

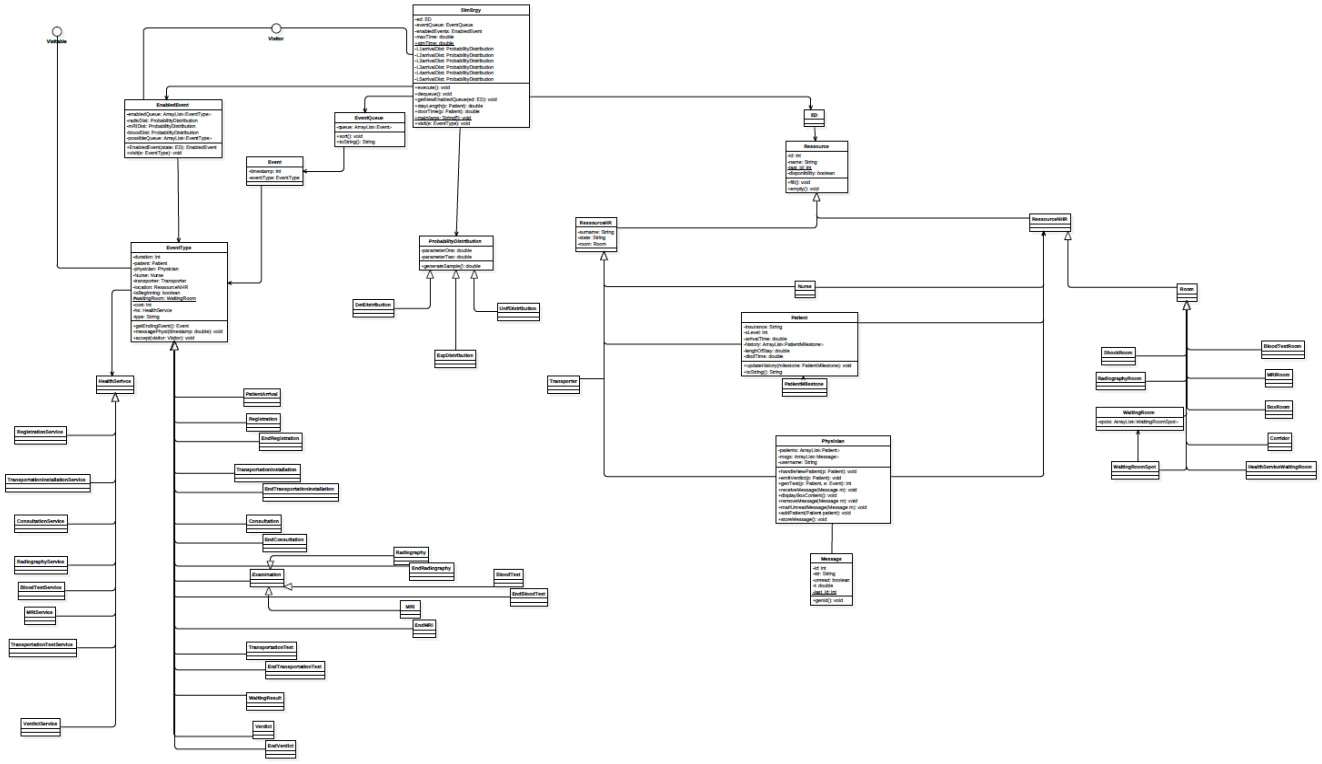
En plus d'avoir dans chaque *HealthService* une *waitingQueue* qui détermine quel sera le prochain patient à utiliser le service, il y a une *waitingQueue* pour les docteurs, pour savoir lequel doit prend en charge un nouveau patient lors d'une nouvelle *consultation.* Celle-ci est designée de la sorte qu'en cas de litiges, c'est toujours le docteur qui a le moins de patients en cours qui prendra en charge le nouveau. Cela permet de limiter les cas où un docteur est surchargé à un instant T car il doit faire le verdict de nombreux patients.

La Visitor Pattern

Nous avons effectué un *visitor pattern* pour *EnabledEvent* et pour *SimErgy*. Cela permet à notre code d'augmenter en flexibilité : Si notre client veut revoir la modélisation de son ED, par exemple en changeant le coût d'une radiographie car il a acheté une machine plus chère, il le faire plus facilement grâce à ce pattern

DIAGRAMME UML

Voici le diagramme des classes associée à notre programme. Pour pouvoir agrandir l'image, merci de trouver le fichier joint pdf nommé « Diagramme UML.pdf ».



SIMULATION GUIDEE PAR UN UTILISATEUR

Notre programme offre également la possibilité à l'utilisateur d'interagir avec lui pour lui donner plus de personnalisation. Par exemple, il peut choisir lui-même que c'est le docteur Michon qui va examiner la patiente Mme. Durand. Pour cela, l'utilisateur devrai utiliser la command-line user interface (CLUI)

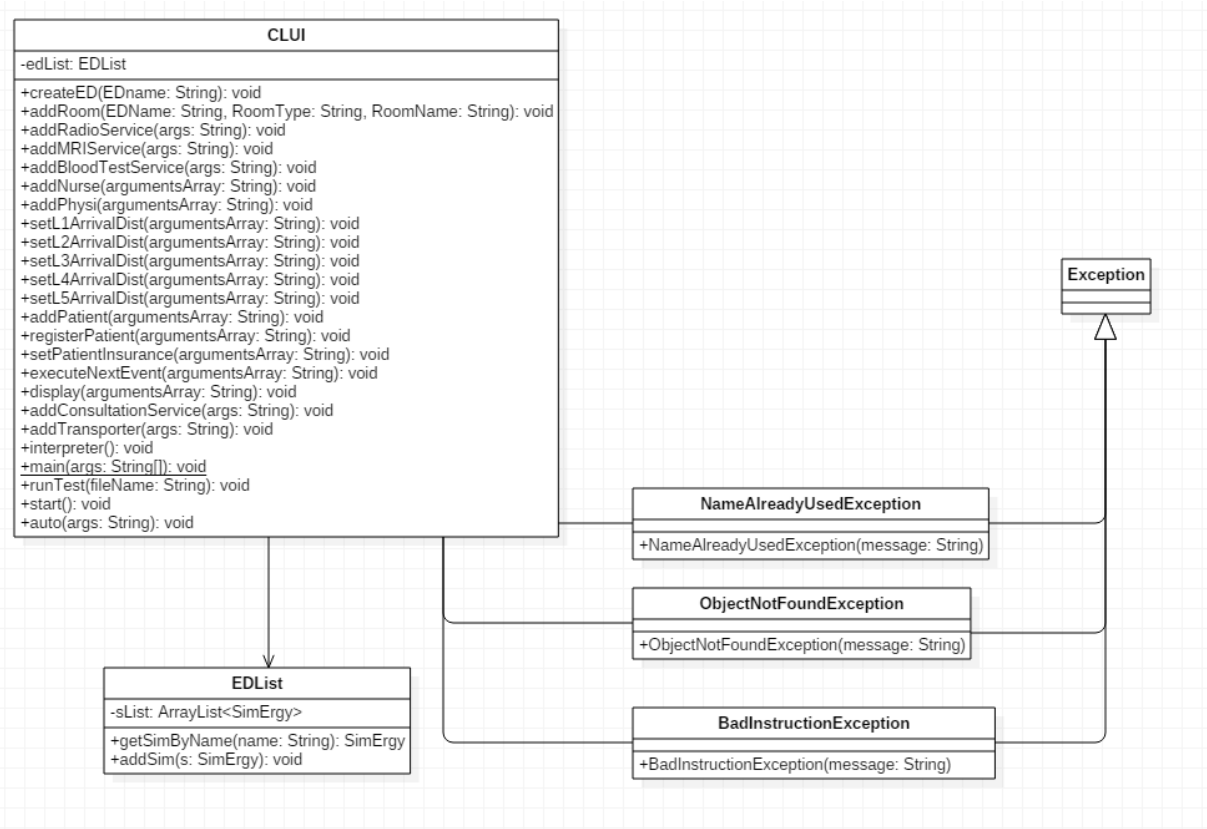
EXPLICATIONS

Ces fonctionnalités se retrouvent dans la classe CLUI. Chaque commande que peut faire l'utilisateur est associée à une méthode dans la classe CLUI.

C'est ensuite la méthode start qui orchestre le tout : pour une commande donnée, elle exécute la méthode associée. Pour cela, elle se sert d'un scanner qui lit les instructions dans la console.

DIAGRAMME UML

Voici ce que cela donne avec un diagramme UML. Pour agrandir l'image, merci de regarder le fichier joint « Diagramme CLUI.pdf »


CLUI
-edList: EDList
+createED(EDname: String): void
+addRoom(EDName: String, RoomType: String, RoomName: String): void
+addRadioService(args: String): void
+addMRIService(args: String): void
+addBloodTestService(args: String): void
+addNurse(argumentsArray: String): void
+addPhysi(argumentsArray: String): void
+setL1ArrivalDist(argumentsArray: String): void
+setL2ArrivalDist(argumentsArray: String): void
+setL3ArrivalDist(argumentsArray: String): void
+setL4ArrivalDist(argumentsArray: String): void
+setL5ArrivalDist(argumentsArray: String): void
+addPatient(argumentsArray: String): void
+registerPatient(argumentsArray: String): void
+setPatientInsurance(argumentsArray: String): void
+executeNextEvent(argumentsArray: String): void
+display(argumentsArray: String): void
+addConsultationService(args: String): void
+addTransporter(args: String): void
+interpreter(): void
+main(args: String[]): void
+runTest(fileName: String): void
+start(): void
+auto(args: String): void
Exception
NameAlreadyUsedException
+NameAlreadyUsedException(message: String)
ObjectNotFoundException
+ObjectNotFoundException(message: String)
BadInstructionException
+BadInstructionException(message: String)
EDList
-sList: ArrayList<SimErgy>
+getSimByName(name: String): SimErgy
+addSim(s: SimErgy): void

CLASSE JUNIT DE TEST

Nous n'avons effectué une classe de test JUnit que pour les classes *EnabledEvents* (classe *EnabledEventsTest*) et *Consultation* (classe *ConsultationTest*). En effet, *EnabledEvents* est la classe où le JUnit était à la fois la plus difficile et la plus utile à coder.

Nous aurions du dans l'idéal coder une classe de test pour chaque classe de notre algorithme.

Cependant, d'une part nous étions déjà très en retard et ne voulions pas déranger plus que nécessaire le correcteur. Et d'autre part nous avons pu tester nous-mêmes par d'autres moyens nos classes.

En effet, nous avons imaginé de nombreux tests scénario pour nous permettre de vérifier la viabilité de la globalité de l'algorithme, diminuant l'importance de tester chaque méthode de chaque classe.

TEST SCENARIO

Il est important de pouvoir tester notre algorithme. Si notre client ne nous fait aps confiance, il fera probablement confiance au test que nous lui proposons. Pour cela, une méthode runTest de notre classe CLUI lit les instructions que le client veut tester, et permet de renvoyer l'état du système en fonction de ces instructions.

Pour tester notre programme, voici la procédure à appliquer.

Ouvrir le fichier java « ». Aller dans le package « CLUI », puis exécuter la classe « CLUI ». Dans la console, taper des commandes telles que décrites dans le cahier des charges de ce même projet.

Pour une vue d'ensemble plus directe, il est intéressant d'appliquer un test scenario qui modélise tout le fonctionnement normal d'une journée d'un ED. Le client peut en créer autant qu'il le souhaite, mais nous en avons fait un pour lui. Pour y accéder, il suffit d'exécuter d'entrer dans la console :

runTest testScenario1.txt

Et appuyer sur la touche entrée.

Ensuite ouvrir le fichier **testScenario1output.txt** qui a été créé à l'issu de l'exécution.

Il faut faire attention à ce que eclipse puisse trouver l'emplacement **testScenario1output.txt** et **init.ini**.

REPARTITION DES TACHES

Nous nous sommes réparti les tâches naturellement et équitablement. Voici comment :

Tâches	Temps de réalisation (en heures)	Temps de réalisation par Sylvain	Temps de réalisation par Adnane
S'impregner du principe d'Emergency Department et du but du projet	8	4	4
Arriver à décrire l'état du système	2	2	0
Comprendre en profondeur le système de DESS	4	2	2
Appliquer une variante du système de DESS pour matcher au projet	8	7	1
Coder toutes les classes de Ressource et EventType	12	6	6
Codage complet de la DESS (avec classe SimErgie et EnabledEvent)	11	8	3
Rectifier les erreurs de codage	7	1	6
Rectifier les erreurs de design facilement améliorables	5	2	3
Tester plus globalement l'agorithme	8	3	5
Coder l'implémentation permettant au correcteur de tester l'algorithme	2	0	2
Coder la partie CLUI	12	0	12
Coder tous les tests Junit	4	1	3
Faire le diagramme UML	6	6	0
Rédiger le rapport	12	7	5
Total	101	49	52

REGARD CRITIQUE

Flexibilité / Design Pattern

Nous nous rendons compte que notre code n'est pas aussi flexible qu'il aurait pu être. Nous avons pu instaurer après coup un *visitor pattern* pour les *eventTypes*, mais avec le recul, nous nous apercevons que nous aurions pu instaurer notamment un *singleton pattern* pour avoir un identifiant unique pour toutes nos ressources et un *strategy pattern* pour les distributions de probabilités, ou encore un *observer* pour que chaque docteur soit notifié lorsqu'il a un message.

Le problème, c'est qu'à ce stade du projet, il est difficile de bousculer toutes les classes pour mettre en place ces *design patterns*. En fait, nous aurions dû imaginer tous *les design patterns* de notre programme avant même de commencer à coder.

Nous n'avons pas pris ce chemin car il nous était très difficile de faire ce travail d'imagination. De comprendre comment allait fonctionner l'algorithme avant même de mettre les mains dans le cambouis. Nous avons alors cédé pour la solution de facilité, nous disant que nous réfléchirons plus tard à optimiser le programme avec les bons *design patterns*.

Nous ressorti aguerri de ce projet. Car nous comprenons maintenant pourquoi il est très important de faire l'effort de concevoir tout le design du programme avant d'entamer la programmation. Dans nos futures programmations, nous ne referons plus cette erreur.

Test Driven Development ?

Nous n'avons pas suivi cette approche mais l'approche classique. En fait, lorsque nous avions achevé la première version de notre algorithme, nous nous sommes rendu compte de tous les écarts qu'il y avait par rapport aux exigences. Par exemple, les patients n'avaient pas de docteur attitré et ils pouvaient être examiné par un docteur puis avoir le verdict avec un autre. Autre point : nous n'avons pas réussi à faire en sorte que les patients les plus sévèrement atteints soient consultés avant les autres.

Avec le recul, nous pensons qu'adopter la méthode TDD nous aurait forcer encore une fois à avoir une vue plus claire avant de nous lancer dans la réalisation du code.

Retard dans les délais de livraison

Nous rendons en retard ce projet. Nous ne cherchons pas à nous justifier mais à expliquer pour quelles raisons nous n'avions pas achevé le projet en temps et en heure. D'une part, nous avons sous-estimé la durée des phases de tests : il y avait beaucoup de bugs dans notre programme, et plus encore de points de perfectibilité. D'autre part, nous avons eu des problèmes d'Eclipse, de java, et de perte d'ordinateur… Nous avons alors préféré travailler ce qu'il restait à faire pendant les vacances et rendre ce projet en retard plutôt que bâclé, et cela nous ai d'autant plus utile.

CONCLUSION

Malgré tout, nous sommes satisfaits nous. Car ce projet nous a beaucoup appris. Il nous a appris tant en compétences techniques que plus globales. Clairement, nous sommes meilleurs en java, UML, et plus largement en algorithmique et design pattern après ce projet qu'à nos débuts. Mais surtout, nous avons goûté aux difficultés du travail de groupe sur un même projet, aux difficultés d'organisation et de planification d'un projet, aux difficultés et à l'importance de designer tout son programme avant de s'y lancer. Il y a beaucoup d'erreurs que nous ne referons plus.

Nous tenons à remercier particulièrement Paolo Ballarini pour nous avoir aidés directement ou indirectement à développer ces compétences et pour sa patience et sa compréhension qui l'ont poussé à retarder par deux fois la date butoir de ce projet