

Compte rendu TP : Architecture logicielle

Hamze Al-Rasheed - Nicolas Commandeur - Benjamin Verdant - Robin Wagner

29/05/21

Contents

| | | |
|----------|----------------------------------|-----------|
| 1 | Sujet | 1 |
| 2 | Prémice | 2 |
| 3 | Architecture utilisée | 4 |
| 4 | Choix technologiques | 6 |
| 5 | Petit pas pour l'homme | 7 |
| 6 | Grand pas pour l'humanité | 8 |
| 7 | Difficultés rencontrées | 9 |
| 8 | Ce que nous avons appris | 10 |

1 Sujet

Le principe du tp est de concevoir l'architecture d'un système de contrôle d'accès à un ensemble de bâtiments.

Un bâtiment possède un nom ainsi que des informations. Les informations sont :

- la liste des portes du bâtiment
 - Le nom de la porte
ex : Porte sud, 8A-44, etc
 - L'id de la badgeuse d'entrée et de sortie de la porte
Impair pour entrer et pair pour sortir
ex : 11 pour entrer dans le bâtiment 1 et 12 pour en sortir
 - La liste des cartes autorisées ou non
ex : Badgeuse 11 : carte 1 autorisée, carte 2 non autorisée, etc
 - L'état de la porte
ex : Ouvert ou fermé

Les utilisateurs ont chacun une **carte** qui a un **id** unique et le nom du détenteur de celle-ci.

ex : carte 1 ⇒ Livai, carte 2 ⇒ Eren

Pour accéder à un bâtiment ou à une salle, un utilisateur doit poser sa carte sur la badgeuse.

Si la personne est autorisée à entrer dans le bâtiment alors, la porte s'ouvre et une lumière verte s'affiche pendant **15 secondes** et la porte reste ouverte pendant le **même temps**.

Si la personne n'est pas autorisée à entrer dans le bâtiment, une lumière rouge s'allume sur celle-ci.

Quand une personne ouvre une porte, un laser se situe juste après celle-ci, pour compter le nombre de personne qui passe. Plusieurs cas possibles :

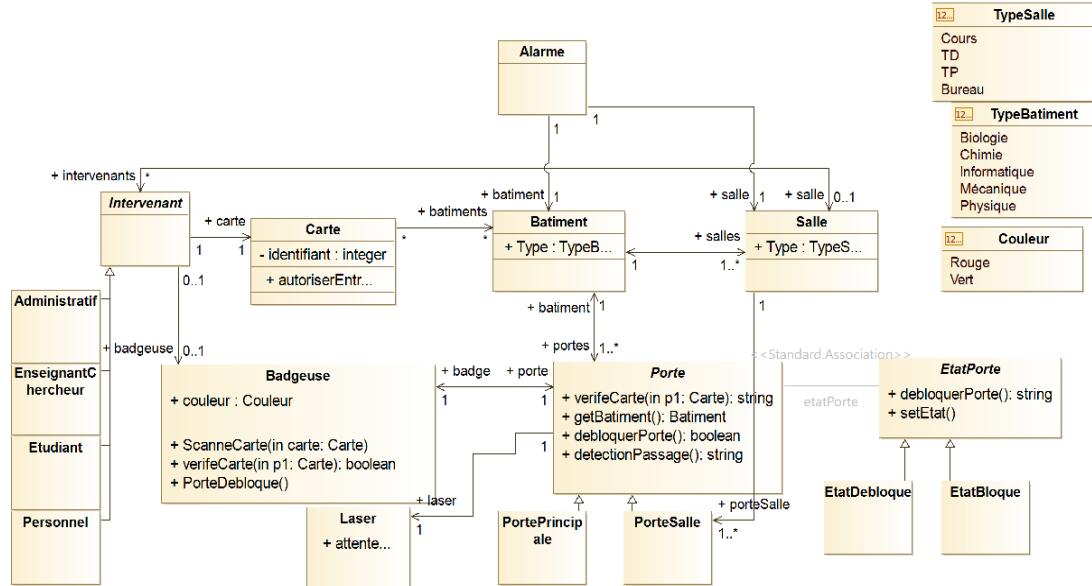
- Une seule personne passe ⇒ la personne qui a ouvert la porte est enregistrée dans le bâtiment et une trace de son passage est inscrit dans le log des passages.
- Plusieurs personnes passent ⇒ une alarme retentit
- Personne ne passe ⇒ rien ne se passe

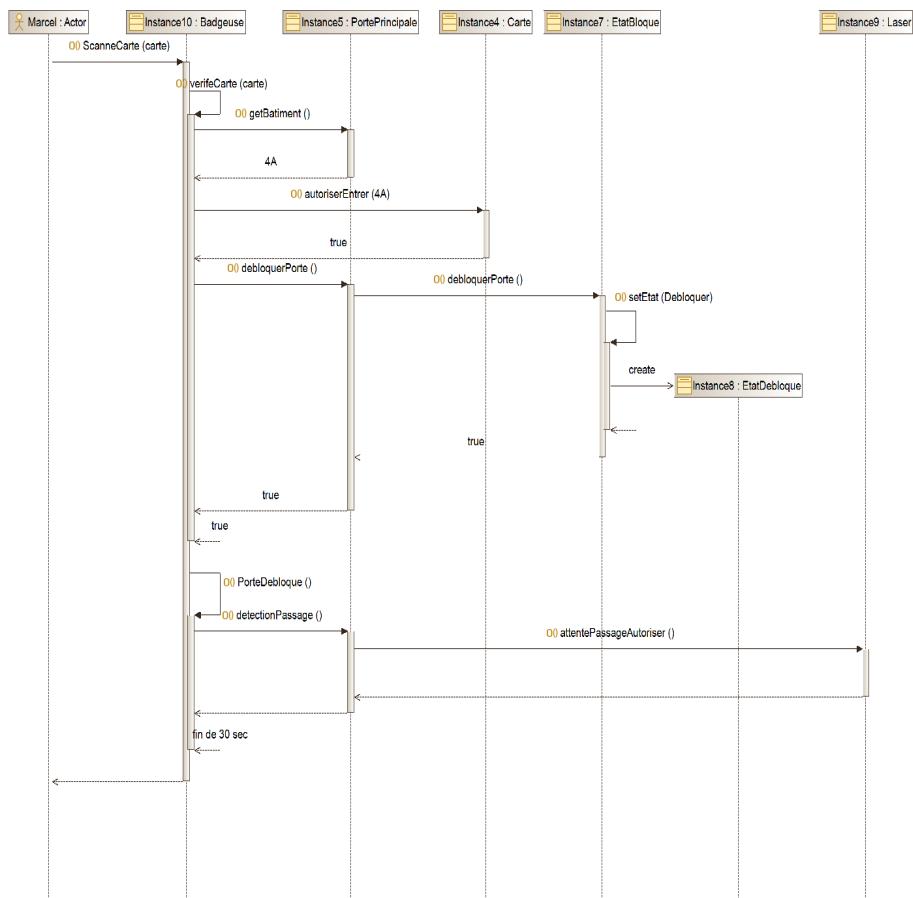
Dans tous les cas, la porte se ferme au bout de **15 secondes**

En cas d'un incendie, toutes les portes sont débloquées et on inscrit dans un fichier toutes les personnes dans les bâtiments.

2 Prémice

Tout d'abord, nous avons analysé le sujet puis nos avons fait un diagramme de classe et un prototype d'un diagramme de séquence. Ceux-ci ont servi de base et ont été modifiés au fur et à mesure.





(a) Diagramme en séquence

3 Architecture utilisée

Nous sommes partis sur un architecture à base de tuple comme celle vue en cours, et pour cela, nous avons décidé de réaliser une classe dédié : *espaceDeTuples*. Voici sa définition :

```
class espaceDeTuples():
    def OUT(self, element):
        self.listeTuples.append(element)

    def IN(self, element, tab):
        resTemp = self.existe(element, tab)
        res = list()
        for index in tab:
            res.append(resTemp[index])
        self.listeTuples.remove(resTemp)
        return res

    def RD(self, element, tab):
        resTemp = self.existe(element, tab)
        res = list()
        for index in tab:
            res.append(resTemp[index])
        return res

    def ADD(self, element, tab):
        resTemp = self.existe(element, tab)
        index = self.listeTuples.index(resTemp)
        listTuple = list(self.listeTuples[index])
        for i in tab:
            listTuple[i] = element[i]
        self.listeTuples[index] = tuple(listTuple)

    def INUNBLOCKED(self, element, tab):
        resTemp = self.existeNonBloquant(element, tab)
        if resTemp is None:
            return None
        res = list()
        for index in tab:
            res.append(resTemp[index])
        self.listeTuples.remove(resTemp)
        return res

    def existe(self, template, tab):
        tuplePossible = []
        while (True):
            for tupleI in self.listeTuples:
                flag = True
                if len(tupleI) == len(template):
                    for i in range(len(template)):
                        if
```

```

        if type(template[i]) != type(tupleI[i]):
            flag = False
        if flag:
            tuplePossible.append(tupleI)
    for tupleI in tuplePossible:
        allIndex = list(range(len(tupleI)))
        for i in tab:
            for j in allIndex:
                if i == j:
                    allIndex.remove(j)
    flag = True
    for i in allIndex:
        if template[i] != tupleI[i]:
            flag = False
    if flag:
        return tupleI

def existeNonBloquant(self, template, tab):
    tuplePossible = []
    for tupleI in self.listeTuples:
        flag = True
        if len(tupleI) == len(template):
            for i in range(len(template)):
                if type(template[i]) != type(tupleI[i]):
                    flag = False
        if flag:
            tuplePossible.append(tupleI)
    for tupleI in tuplePossible:
        allIndex = list(range(len(tupleI)))
        for i in tab:
            for j in allIndex:
                if i == j:
                    allIndex.remove(j)
    flag = True
    for i in allIndex:
        if template[i] != tupleI[i]:
            flag = False
    if flag:
        return tupleI

def __init__(self):
    self.listeTuples = list()

```

Cette classe implémente les différentes interactions dont notre programme avait besoin.

Ps : Problème dans la fonction *existe* qui a une boucle `while True` et qui est la cause principale des problèmes rencontrés. Voir Difficultés rencontrées.

Nous avons choisi seulement cette implémentation car elle remplit à elle seule les fonctionnements nécessaires au TP.

4 Choix technologiques

Nous avons décidé d'utiliser python car celui-ci :

- Possède déjà une structure de tuples que nous avons utilisé dans la partie précédente
- Threading simple à implémenter
- Interface graphique simple à implémenter (Kivy)
- Pas trop lourd et rapide d'exécution

Dans notre TP, les espaces de tuples sont représentés par une **variable globale**. Dans le système final, cet espace serait représenté par une base de données. Chaque **agent** est représenté par un **Thread** dont la mémoire est partagée avec tous les autres Threads qui représentent les autres agents.

La classe badgeuse est représenté par :

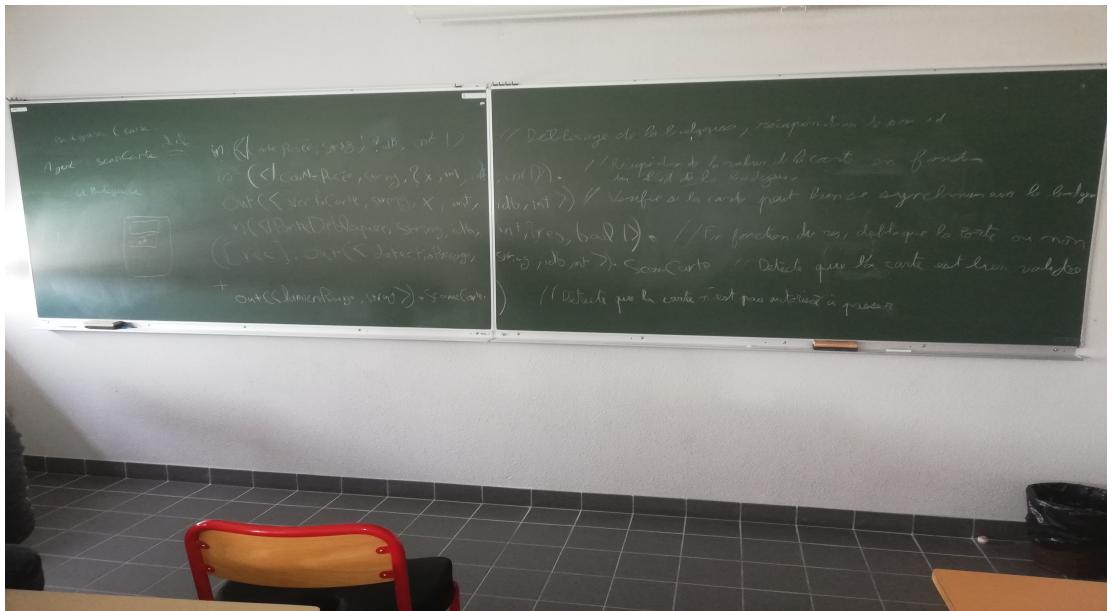
- lecteurCarte : dépose un tuple dans l'espace de tuple
- verifCarte : autorise la personne à entrer dans le bâtiment
- scanCarte : lance la détection et l'allumage des leds
- lumiereVerte : s'allume lorsque la personne est autorisée
- lumiereRouge : s'allume lorsque la personne n'est pas autorisée
- detectionPassage : le capteur dépose dans la base de tuple un tuple lorsque qu'il détecte lorsqu'une personne passe
- etatPorte : permet d'ouvrir une porte et de la fermer. En cas d'incendie, ferme toutes les portes.

Nous avons aussi des agents pour les fonctions secondaires :

- declencheAlarme : Souleve l'alerte si il y a plus d'une personne qui passe
- incendie : le capteur d'incendie dépose un tuple qui est détecté par cet agent va déclencher une alarme.

5 Petit pas pour l'homme

Nous avons décidé alors de commencer par l'agent qui permettait de scanner une carte, qui servira de *core* aux restes des agents. Tout d'abord, nous avons fait un exemple en langage algorithmique :

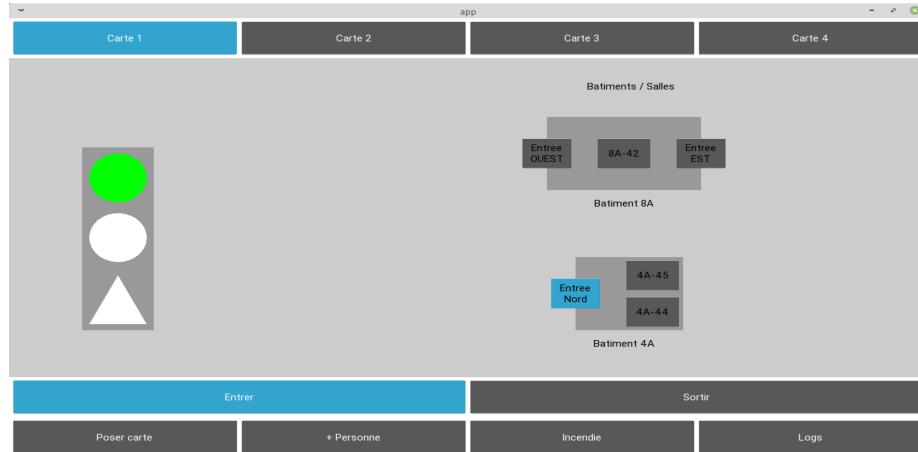


(a) 1er jet

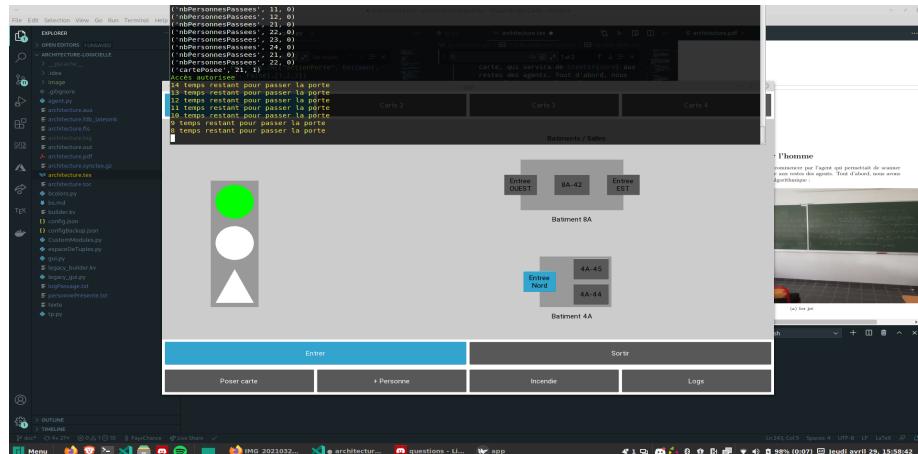
Assez difficile au début, mais petit à petit comme l'oiseau qui fait son nid, cet agent nous a bien servi pour le reste.

6 Grand pas pour l'humanité

Nous avons ensuite réalisé les autres agents. Nous les avons testé un par un, puis ensemble. Nous avons dû créer des agents spécialement pour l'interface, pour avoir un rendu direct avec l'espace de tuple.



(a) Interface graphique avec Kivy



(b) Interface avec la console qui affiche le temps

7 Difficultés rencontrées

Au début de notre travail nous avons voulu faire le TP avec JAVA car Modelio nous permettait de créer un diagramme d'objet et des diagrammes de séquences que nous pouvions directement exporter en code JAVA. Cependant nous nous sommes rendu compte que JAVA n'était pas une bonne idée puisque, lui-même, consomme déjà beaucoup de mémoire pour exécuter un programme. De plus l'architecture créée par Modelio ne convenait pas au sujet et aux objectifs du TP. Après avoir pris du recul nous avons décidé d'utiliser Python car c'est un langage adapté pour faire une architecture à base de tuples.

La seconde difficulté qui nous est apparue est lors de l'assignation de nos fonctions sur des threads. Nous avons commencé à assigner les threads fonction par fonction et à les tester un par un pour vérifier qu'ils fonctionnaient correctement indépendamment les uns des autres. Lorsque toutes les fonctions avaient été vérifiées nous avons commencé à faire tourner tous les threads en même temps. Mais, le programme avait un temps d'exécution beaucoup trop long pour pouvoir l'utiliser. Nous avions plus de **200** threads qui fonctionnaient les un en même temps que les autres. C'est pour cela, que nous avons minimisé le nombre de badgeuse, de bâtiment... lors de la démonstration.

Ensuite nous avons eu des soucis lors de la création de l'inteface et avant tout pour l'imaginer. Il nous fallait une interface qui puisse interagir avec l'espace de tuple en temps réel et de façon intuitive.

8 Ce que nous avons appris

Pour conclure, trop de thread tue les threads. Les threads ont leur limite, il aurait été peut être plus judiceux de faire des processus, mais nous aurions dû changer la représentation de notre espace de tuple avec une base de données. Lier le backend avec l'interface tout en utilisant les tuples et composant connecteur pour les boutons.