

# LES ASCENCEURS DE TOLBIAC

# Que doit faire notre système d'ascenseur ?



Simuler les utilisateurs



Simuler leurs demandes



Attribuer les demandes dans les différents ascenseurs



Faire rentrer utilisateurs dans l'ascenseur correspondant



Déplacer les ascenseurs



Faire sortir les utilisateurs à leurs étages

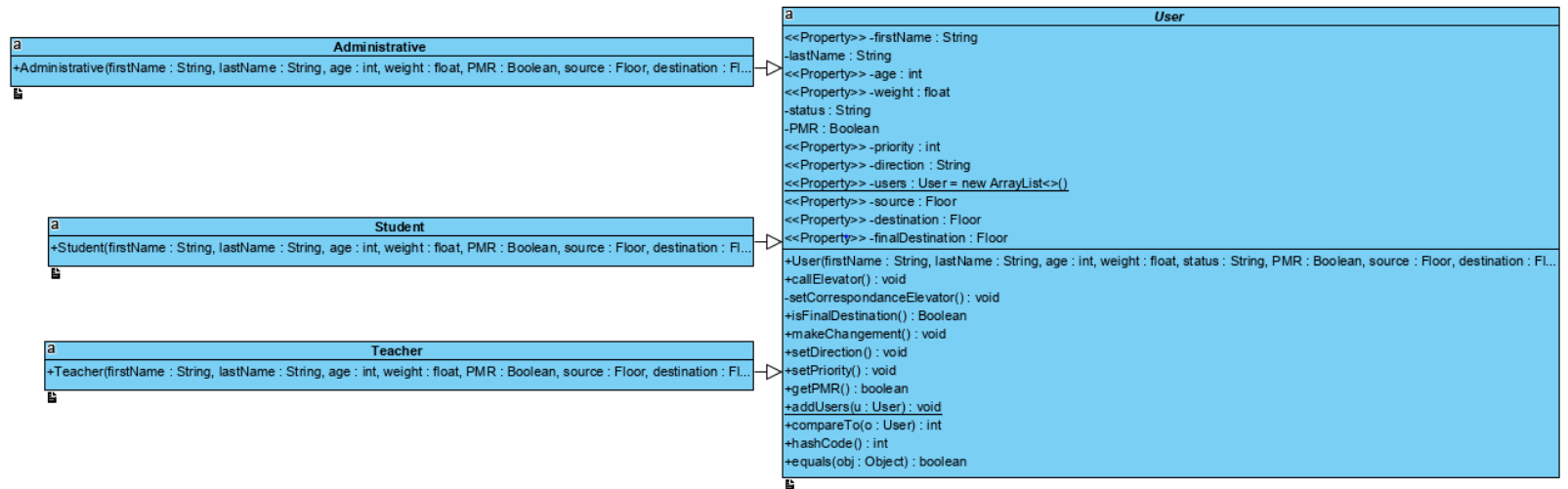


Gérer les utilisateurs devant faire un changement



Gérer la priorité des entrées et des sorties

# Simuler les utilisateurs



Nous avons une classe abstraite User, définie par : un nom, un prénom, un âge, un statut, une situation ou non de handicap, un numéro correspondant à sa priorité (plus ce numéro est grand plus l'User est prioritaire) calculé par la méthode « setPriority() », un étage de départ, un étage d'arrivé, un étage final (dont nous détaillerons l'utilité) ainsi qu'une direction calculée à partir de l'étage de départ et celui d'arrivé dans la méthode « setDirection() ».

Nous avons créé plusieurs méthodes :

« callElevator() » qui crée une nouvelle demande,

« setCorrespondance() », « isFinalDestination » et « makeChangement() » qui servent à traiter les cas d'User devant changer d'ascenseurs pour atteindre leur destination finale,

« compareTo() » pour comparer les Users en fonction de leur priorité (notre classe User implémente l'interface Comparable),

Les méthodes « equals() » et « hashCode() » ont été surchargées pour utiliser des HashMap d'User.

Dans les classes Administrative, Student et Teacher (qui héritent d'User), nous redéfinissons seulement le statut de l'User.

Ce statut sert à l'instanciation pour déterminer l'attribut de priorité.

# Gérer les utilisateurs devant faire un changement



```
a
User
<<Property>> -firstName : String
-lastName : String
><<Property>> -age : int
<<Property>> -weight : float
-status : String
-PMR : Boolean
<<Property>> -priority : int
<<Property>> -direction : String
<<Property>> -users : User = new ArrayList<>()
<<Property>> -source : Floor
<<Property>> -destination : Floor
><<Property>> -finalDestination : Floor
+User(firstName : String, lastName : String, age : int, weight : float, status : String, PMR : Boolean, source : Floor, destination : FL...
+callElevator() : void
-setCorrespondanceElevator() : void
+isFinalDestination() : Boolean
+makeChangement() : void
+setDirection() : void
>+setPriority() : void
+getPMR() : boolean
+addUsers(u : User) : void
+compareTo(o : User) : int
+hashCode() : int
+equals(obj : Object) : boolean
```

Dans le cas où la couleur de départ est différente de la couleur d'arrivée :

- A l'instanciation d'un User nous calculons l'étage pivot (à l'aide de la méthode « setCorrespondance() »). Cet étage devient la destination et la destination finale est stockée dans l'attribut « finalDestination ».
- Quand l'utilisateur arrive à sa destination pivot, le test « isFinalDestination() » est faux et permet d'appeler la méthode « makeChangement() » (qui met l'User dans la queue du Floor correspondant à sa prochaine demande).

# Simuler les demandes



a	Demand
<<Property>>	-direction : String
<<Property>>	-floor : Floor
+	Demand(f : Floor, dir : String)
+	hashCode() : int
+	equals(obj : Object) : boolean

UP

DOWN

La classe Demand représente l'appel d'un utilisateur à un étage. Cette demande contient deux attributs correspondants aux deux informations envoyées par un bouton « Appuyé » dans le système d'ascenseur :

- Un étage d'appel
- Une direction pouvant être égale à « up » ou « down »

Les méthode « equals() » et « hashCode() » ont été surchargées pour utiliser des Sets de Demand.

# Attribuer les demandes dans les différents ascenseurs



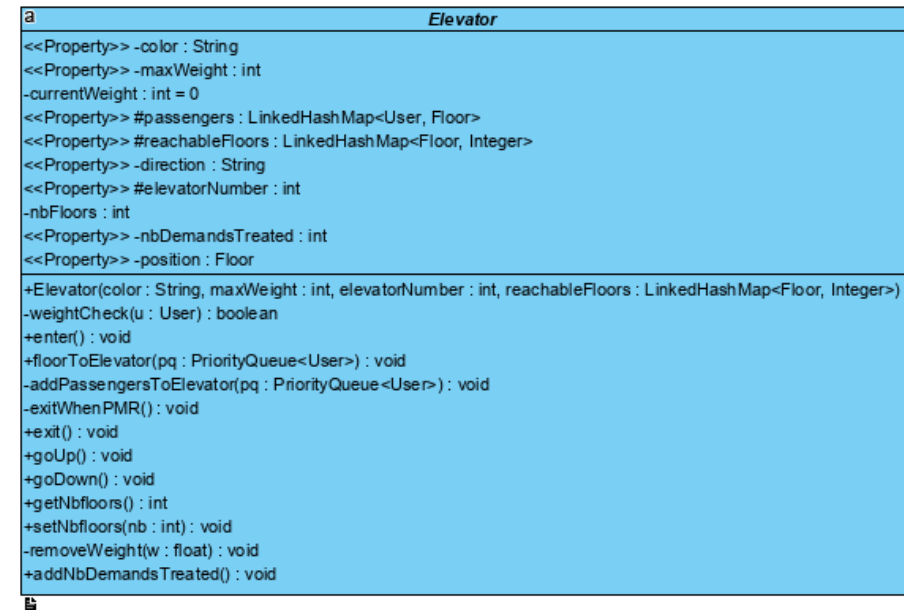
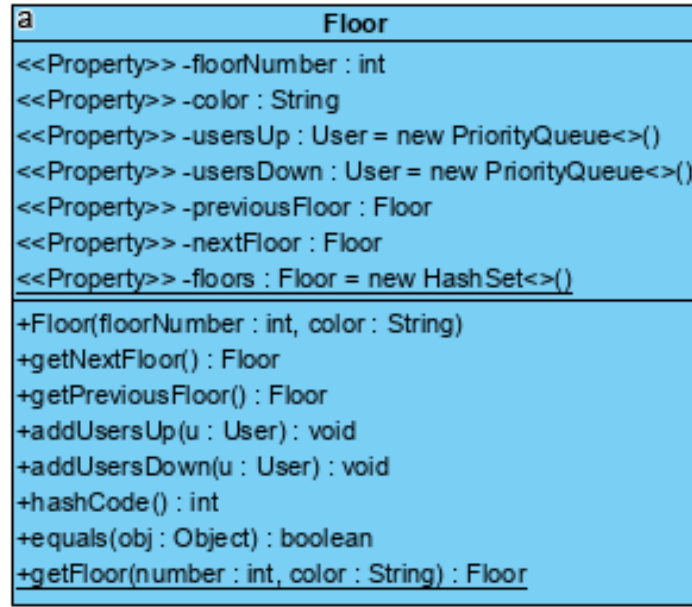

```
a Dispatcher
<<Property>> -listElevator : Map<String, List<Elevator>> = new HashMap<>()
<<Property>> -demands : Demand = new LinkedHashSet<Demand>()

+dispatch() : void
-chooseNearestElevator(d : Demand) : Elevator
-addDemandOnChooosen(choosen : Elevator, d : Demand) : void
-getNbFloorToReachDemand(choosen : Elevator, d : Demand) : int
-grabNullElevator(elevatorColor : String) : Elevator
+addDemand(d : Demand) : void
```

La classe Dispatcher contient un Set de toutes les demandes rempli par la méthode « addDemand(demand) » et a pour rôle de les attribuer aux Elevators à travers la méthode « dispatch() ». Pour chaque demande en cours :

- Le dispatcher la donne en priorité à un Elevator à l'arrêt de la bonne couleur (qu'il récupère à l'aide de la méthode « grabNullElevator(ElevatorColor) »).
- Si cetElevator est en dessous d'une demande et en monté ou au-dessus d'une demande et en descente on calcule le nombre d'itération (« getNbFloorToReachDemand(choosen, demand) ») sur les Floors à effectuer par l'Elevator avant de prendre des Users et de changer de direction. On ajoute la demande via « addDemandOnChooosen(choosen, demand) »
- Si aucun Elevator n'est à l'arrêt, il va calculer l'Elevator le plus proche dont le chemin passe par la demande

Faire rentrer les  
utilisateurs dans  
l'ascenseur  
correspondant



Si aucun Elevator ne répond à cette demande il laisse la demande pour le prochain appel à « dispatch() ». Chaque Floor possède deux PriorityQueue représentant respectivement les Users attendant pour monter (userUp) et ceux pour descendre (userDown). Les Users sont rangés par ordre de priorité.

La méthode « enter() » permet de faire rentrer les utilisateurs se trouvant à l'étage de la position de l'Elevator et étant dans la queue correspondant à la direction de l'Elevator. Elle appelle la méthode « floorToElevator(PriorityQueue) » tant que le poids (« weightCheck() ») ou la surface (« surfaceCheck() ») le permet.

# Gérer la priorité des entrées et des sorties



a	Elevator
<<Property>> -color : String	
<<Property>> -maxWeight : int	
<<Property>> -currentWeight : int = 0	
<<Property>> #passengers : LinkedHashMap<User, Floor>	
<<Property>> #reachableFloors : LinkedHashMap<Floor, Integer>	
<<Property>> -direction : String	
<<Property>> #elevatorNumber : int	
<<Property>> -nbFloors : int	
<<Property>> -nbDemandsTreated : int	
<<Property>> -position : Floor	
+Elevator(color : String, maxWeight : int, elevatorNumber : int, reachableFloors : LinkedHashMap<Floor, Integer>)	
<-weightCheck(u : User) : boolean	
+enter() : void	
+floorToElevator(pq : PriorityQueue<User>) : void	
<-addPassengersToElevator(pq : PriorityQueue<User>) : void	
<-exitWhenPMR() : void	
+exit() : void	
+goUp() : void	
+goDown() : void	
+getNbFloors() : int	
+setNbFloors(nb : int) : void	
<-removeWeight(w : float) : void	
+addNbDemandsTreated() : void	

Les Users sont ajoutés par ordre de priorité (« addPassengerToElevator(PriorityQueue) »). S'il reste un PMR dans la queue qui n'a pas pu rentrer, la fonction « exitWhenPMR() » est appelée. Elle retire les utilisateurs par ordre décroissant de priorité jusqu'à que le PMR puisse rentrer.



# Déplacer les ascenseurs



```
a Floor
<<Property>> -floorNumber : int
<<Property>> -color : String
<<Property>> -usersUp : User = new PriorityQueue<>()
<<Property>> -usersDown : User = new PriorityQueue<>()
<<Property>> -previousFloor : Floor
<<Property>> -nextFloor : Floor
<<Property>> -floors : Floor = new HashSet<>()

+Floor(floorNumber : int, color : String)
+getNextFloor() : Floor
+getPreviousFloor() : Floor
+addUsersUp(u : User) : void
+addUsersDown(u : User) : void
+hashCode() : int
+equals(obj : Object) : boolean
+getFloor(number : int, color : String) : Floor
```

```
a Elevator
<<Property>> -color : String
<<Property>> -maxWeight : int
<<Property>> -currentWeight : int = 0
<<Property>> #passengers : LinkedHashMap<User, Floor>
<<Property>> #reachableFloors : LinkedHashMap<Floor, Integer>
<<Property>> -direction : String
<<Property>> #elevatorNumber : int
-nbFloors : int
<<Property>> -nbDemandsTreated : int
<<Property>> -position : Floor

+Elevator(color : String, maxWeight : int, elevatorNumber : int, reachableFloors : LinkedHashMap<Floor, Integer>)
-weightCheck(u : User) : boolean
+enter() : void
+floorToElevator(pq : PriorityQueue<User>) : void
+addPassengersToElevator(pq : PriorityQueue<User>) : void
-exitWhenPMR() : void
+exit() : void
+goUp() : void
+goDown() : void
+getNbFloors() : int
+setNbFloors(nb : int) : void
-removeWeight(w : float) : void
+addNbDemandsTreated() : void
```

Les étages sont définis par une couleur et un numéro. A la création du système (SystemInit), les étages sont créés à travers notre propre liste chaînée. Ainsi nous pouvons simplement accéder à l'étage précédent ou suivant par les méthodes « getNextFloor() » et « getPreviousFloor() ».

Les méthodes « goUp() » et « goDown() » utilisent ces méthodes pour changer la position d'un Elevator.

Les exceptions LastFloorException et FirstFloorException sont lancées si on essaye d'accéder respectivement au suivant du dernier Floor et au précédent du premier Floor.

# Faire sortir les utilisateurs à leurs étages



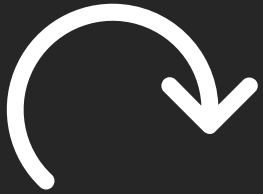
```
Elevator
<<Property>> -color : String
<<Property>> -maxWeight : int
<<Property>> -currentWeight : int = 0
<<Property>> #passengers : LinkedHashMap<User, Floor>
<<Property>> #reachableFloors : LinkedHashMap<Floor, Integer>
<<Property>> -direction : String
<<Property>> #elevatorNumber : int
-nbFloors : int
<<Property>> -nbDemandsTreated : int
<<Property>> -position : Floor

+Elevator(color : String, maxWeight : int, elevatorNumber : int, reachableFloors : LinkedHashMap<Floor, Integer>)
-weightCheck(u : User) : boolean
+enter() : void
+floorToElevator(pq : PriorityQueue<User>) : void
-addPassengersToElevator(pq : PriorityQueue<User>) : void
-exitWhenPMR() : void
+exit() : void
+goUp() : void
+goDown() : void
+getNbFloors() : int
+setNbFloors(nb : int) : void
-removeWeight(w : float) : void
+addNbDemandsTreated() : void
```

La méthode « exit() » permet de faire sortir les Users ayant pour destination la position de l'Elevator.

La méthode « removeWeight() » retire leur poids du poids courant (currentWeight) à l'intérieur l'Elevator.

# Séquence



Puisque nous sommes en programmation séquentielle, nous avons créé une classe ElevatorSequence pour gérer les séquences de notre système. La méthode « SystemEmpty() » renvoie « true » quand le système n'a plus d'User en attente dans les Floors ni dans les Elevators. « makeSequence() » fonctionne comme suit :

1. Appelle la fonction dispatch() pour attribuer les demandes, Pour chaque Elevator :
2. Gère les sorties,
3. Gère les entrées,
4. Gère les mouvements.

Ainsi, tout simplement notre système fonctionne avec le code ci-dessous :

a	ElevatorSequence
	<u>+makeSequence() : void</u>
	<u>-elevatorStopper(el : Elevator) : void</u>
	<u>+SystemEmpty() : boolean</u>

```
SystemInit sys = new SystemInit();  
do {  
    ElevatorSequence.makeSequence();  
} while (!ElevatorSequence.SystemEmpty());
```

# Resultats

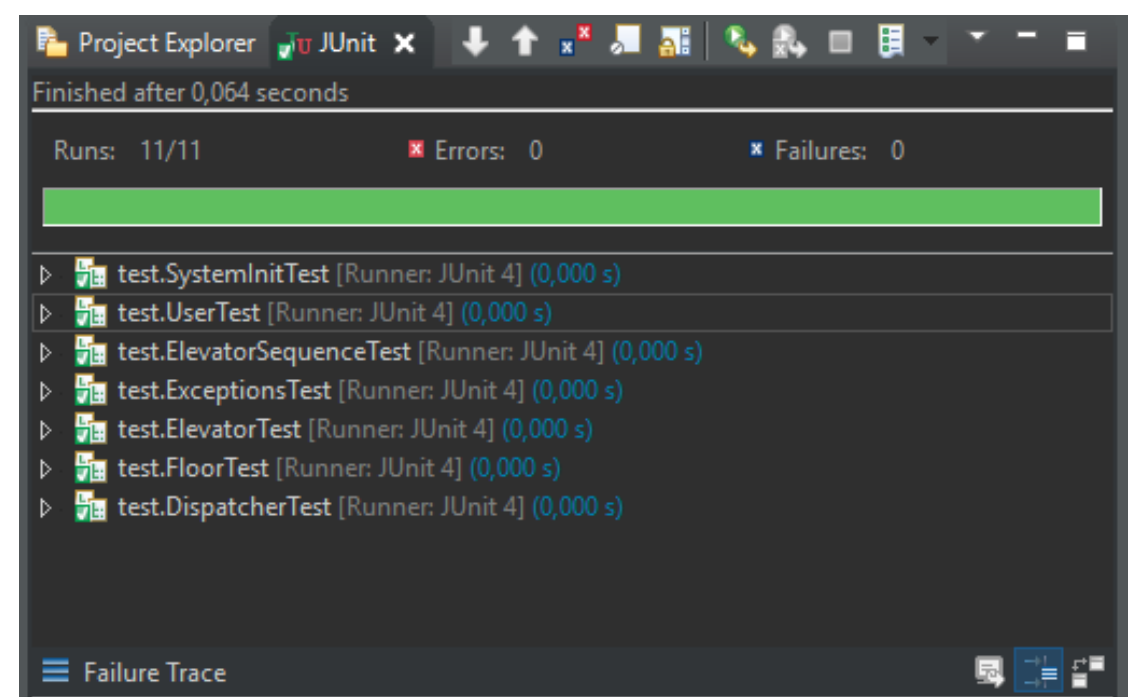
```
INFORMATIONS USERS
*****
TOTAL USERS : 200.0
Number of PMR : 11.0
Proportion of PMR : 5.5%
Number of Students : 71.0
Proportion of Students : 35.5%
Number of Teacher : 59.0
Proportion of Teacher : 29.499998%
Number of Administrative : 59.0
Proportion of Administrative : 29.499998%
Medium Weight : 89.58015
Medium Age : 54.615

INFORMATIONS SYSTEM
*****
System Duration : 21ms
System Iterations : 24
System Success : 200 users
System failures : 0 users




INFORMATIONS ELEVATORS
*****
red
-----
1 - Number of demands treated : 2
2 - Number of demands treated : 3
3 - Number of demands treated : 2
4 - Number of demands treated : 2
5 - Number of demands treated : 3
6 - Number of demands treated : 2

Most demanded : Elevator n°2
green
-----
1 - Number of demands treated : 2
2 - Number of demands treated : 4
3 - Number of demands treated : 2
4 - Number of demands treated : 4
5 - Number of demands treated : 3
6 - Number of demands treated : 2
```

# Tests Unitaires








Nous avons utilisé JUnit pour effectuer nos tests unitaires. La couverture du code (coverage) total du projet est de 69,8%. Nous laissons une grande partie non-testée de par les affichages et les fonctions servant à réaliser les statistiques de notre programme (qui représentent un grand nombre d'instructions). Seule la partie fonctionnelle est testée ici par des tests unitaires.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
ascenseur-app	 69,8 %	3 217	1 390	4 607
src/main/java	 60,9 %	2 000	1 286	3 286
src/test/java	 92,1 %	1 217	104	1 321

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6
7   <groupId>com.ascenseur.demo</groupId>
8   <artifactId>ascenseur-app</artifactId>
9   <version>1.0-SNAPSHOT</version>
10
11   <name>ascenseur-app</name>
12   <!-- FIXME change it to the project's website -->
13   <url>http://www.example.com</url>
14
15   <properties>
16     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
17     <maven.compiler.source>1.7</maven.compiler.source>
18     <maven.compiler.target>1.7</maven.compiler.target>
19     <junit.jupiter.version>5.5.2</junit.jupiter.version>
20     <junit.platform.version>1.5.2</junit.platform.version>
21   </properties>
22
23   <dependencies>
24     <dependency>
25       <groupId>org.junit.jupiter</groupId>
26       <artifactId>junit-jupiter-engine</artifactId>
27       <version>5.5.2</version>
28       <scope>test</scope>
29     </dependency>
30     <dependency>
31       <groupId>org.junit.platform</groupId>
32       <artifactId>junit-platform-runner</artifactId>
33       <version>1.5.2</version>
34       <scope>test</scope>
35     </dependency>
36     <dependency>
37       <groupId>junit</groupId>
38       <artifactId>junit</artifactId>
39       <version>4.13-rc-2</version>
40       <scope>test</scope>
41     </dependency>
42   </dependencies>
43
44   <build>
45     <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be moved to parent pom) -->
46     <plugins>
47       <!-- clean lifecycle, see https://maven.apache.org/ref/current/maven-core/lifecycles.html#clean_lifecycle -->
48     </plugins>
49   </build>

```

 Build v1.0.0	16/12/2019 10:12	Dossier de fichiers	
 Elements de rendu	16/12/2019 10:46	Dossier de fichiers	
 src	16/12/2019 10:19	Dossier de fichiers	
 DOCUMENTATION PROJET.pdf	16/12/2019 09:50	Fichier PDF	1 662 Ko
 README.md	16/12/2019 10:17	Fichier MD	1 Ko

# Maven

- Nous avons choisi de faire un projet Maven ce qui nous a permis de faire un build de notre projet (Build v1.0.0).
- Le fichier POM.XML contient tous nos dépendances (ici seulement Junit5)
- Les fichiers sources se trouvent dans le répertoire « src »

# Javadoc

The screenshot shows a web browser displaying a Javadoc index page. The address bar shows the file path: `file:///C:/Users/daekc/OneDrive/Bureau/POO/POO_Ascenseur/src/doc/allclasses-index.html`. The page has a navigation bar with links: OVERVIEW, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar is a search bar labeled "SEARCH: Search". The main content area is titled "All Classes" and contains three tabs: "All Classes" (selected), "Class Summary", and "Exception Summary". Below the tabs is a table with two columns: "Class" and "Description". The table lists the following classes: Administrative, Demand, Dispatcher, Elevator, ElevatorSequence, FirstFloorException, Floor, GreenElevator, LastFloorException, Main, NoSuchFloorException, RedElevator, Student, SystemInit, Teacher, UnreachableFloor, User, Utils, and YellowElevator.

Class	Description
Administrative	
Demand	
Dispatcher	
Elevator	
ElevatorSequence	
FirstFloorException	
Floor	
GreenElevator	
LastFloorException	
Main	
NoSuchFloorException	
RedElevator	
Student	
SystemInit	
Teacher	
UnreachableFloor	
User	
Utils	
YellowElevator	

Plutôt que de commenter le code, nous nous sommes servis de la génération de Javadoc par Eclipse pour documenter notre code. Vous pouvez la retrouver dans le répertoire :  
« \POO\_Ascenseur\src\doc »

## Floors

```
0, red
users up : 0
users down : 0

9, yellow
users up : 0
users down : 0

0, green
users up : 0
users down : 0

11, yellow
users up : 0
users down : 0

12, yellow
users up : 0
users down : 0

13, yellow
users up : 0
users down : 0

14, yellow
users up : 0
users down : 0

15, yellow
users up : 0
users down : 2

9, red
users up : 0
users down : 0
```

## Users

```
Direction = down
Destination = 9, yellow
Final destination = 18, red
Source = 15, yellow
Is PMR : false

Direction = down
Destination = 5, green
Source = 8, green
Is PMR : true

Direction = down
Destination = 16, red
Source = 19, red
Is PMR : false

Direction = up
Destination = 9, green
Final destination = 20, red
Source = 0, green
Is PMR : false

Direction = up
Destination = 9, green
Final destination = 13, yellow
Source = 7, green
Is PMR : false
```

## Demands

```
down, 15, yellow
down, 8, green
down, 19, red
up, 0, green
up, 7, green
down, 18, red
down, 9, green
```

## Elevator

```
red : up : 9 : {} : 3
red : up : 9 : {} : 2
red : null : 0 : {} : 0
red : null : 0 : {} : 0
red : null : 0 : {} : 0
red : null : 0 : {} : 0
green : up : 4 : {} : 3
green : up : 4 : {user.Student@5cf87a79=floor.Floor@b6391527} : 0
green : up : 4 : {} : 0
green : up : 4 : {} : 4
green : null : 0 : {} : 0
green : null : 0 : {} : 0
yellow : up : 9 : {} : 5
yellow : null : 0 : {} : 0
yellow : null : 0 : {} : 0
yellow : null : 0 : {} : 0
yellow : null : 0 : {} : 0
yellow : null : 0 : {} : 0
```

# IHM

Les méthodes suivantes permettent d'afficher les informations relatives aux différents objets du système pour connaître leurs états :

- `Utils.displayDemandsDetails();`
- `Utils.displayElevatorDetails();`
- `Utils.displayFloorsDetails();`
- `Utils.displayUsersDetails`



# SystemStats



La création d'une classe SystemStats nous a permis d'avoir des indicateurs sur la performance du système.

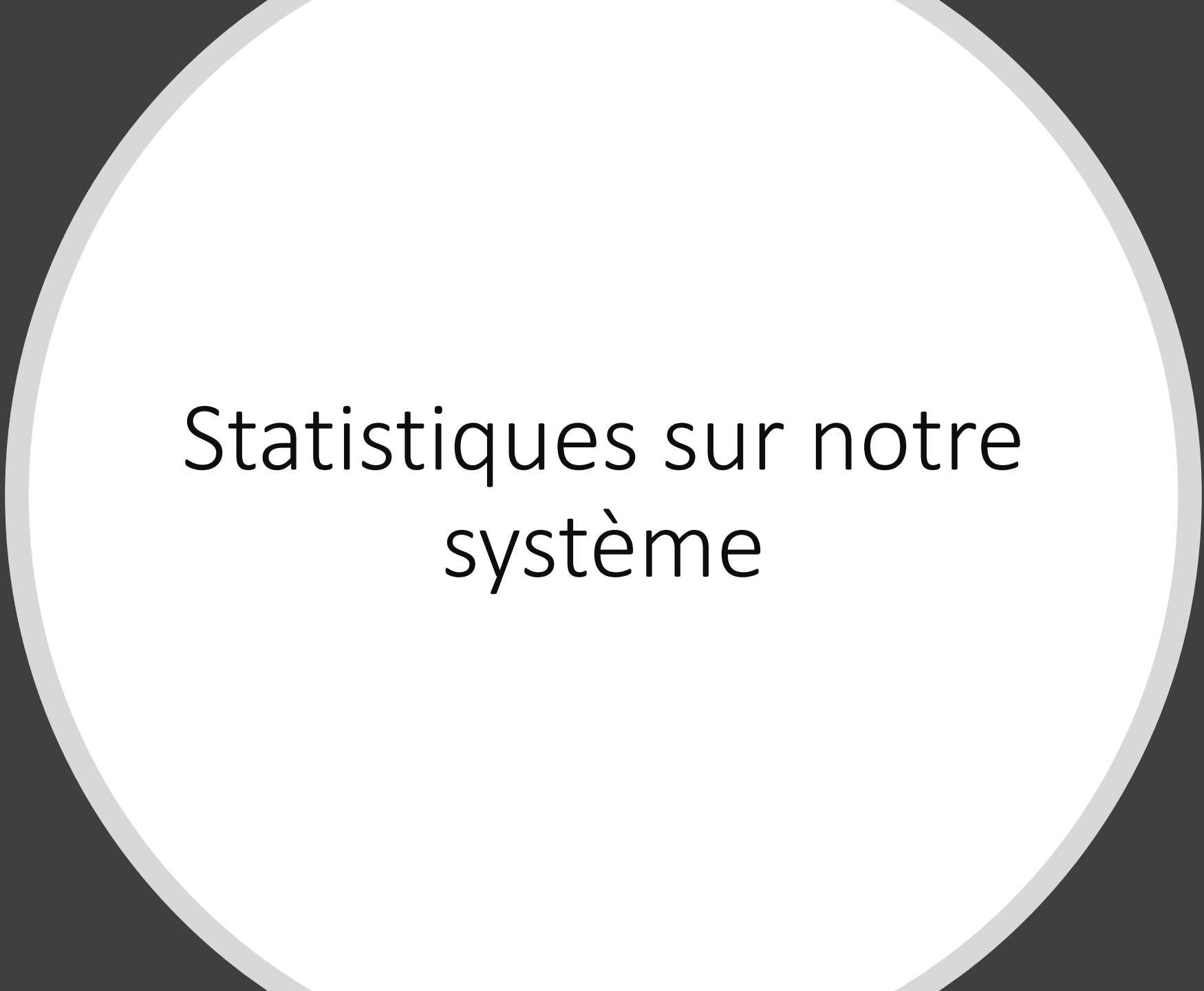
```
INFORMATIONS USERS
*****
TOTAL USERS : 200.0
Number of PMR : 11.0
Proportion of PMR : 5.5%
Number of Students : 71.0
Proportion of Students : 35.5%
Number of Teacher : 59.0
Proportion of Teacher : 29.499998%
Number of Administrative : 59.0
Proportion of Administrative : 29.499998%
Medium Weight : 89.58015
Medium Age : 54.615

INFORMATIONS SYSTEM
*****
System Duration : 21ms
System Iterations : 24
System Success : 200 users
System failures : 0 users

INFORMATIONS ELEVATORS
*****
red
-----
1 - Number of demands treated : 2
2 - Number of demands treated : 3
3 - Number of demands treated : 2
4 - Number of demands treated : 2
5 - Number of demands treated : 3
6 - Number of demands treated : 2

Most demanded : Elevator n°2
green
-----
1 - Number of demands treated : 2
2 - Number of demands treated : 4
3 - Number of demands treated : 2
4 - Number of demands treated : 4
5 - Number of demands treated : 3
6 - Number of demands treated : 2
```

a SystemStats	
-nbSequenceIterations	: int
-totalUserReachDestination	: int
-timeStart	: long = 0
-timeEnd	: long = 0
-totalWeight	: float
-totalAge	: float
-nbPMR	: float
-nbStudent	: float
-nbTeacher	: float
-nbAdmin	: float
+getStats()	: String
+addPMR()	: void
+addAdmin()	: void
+addStudent()	: void
+addTeacher()	: void
+addAge(a : float)	: void
+addWeight(w : float)	: void
+addSequenceIteration()	: void
+setTimeStart()	: void
+setTimeEnd()	: void
+addUserReachDestination()	: void

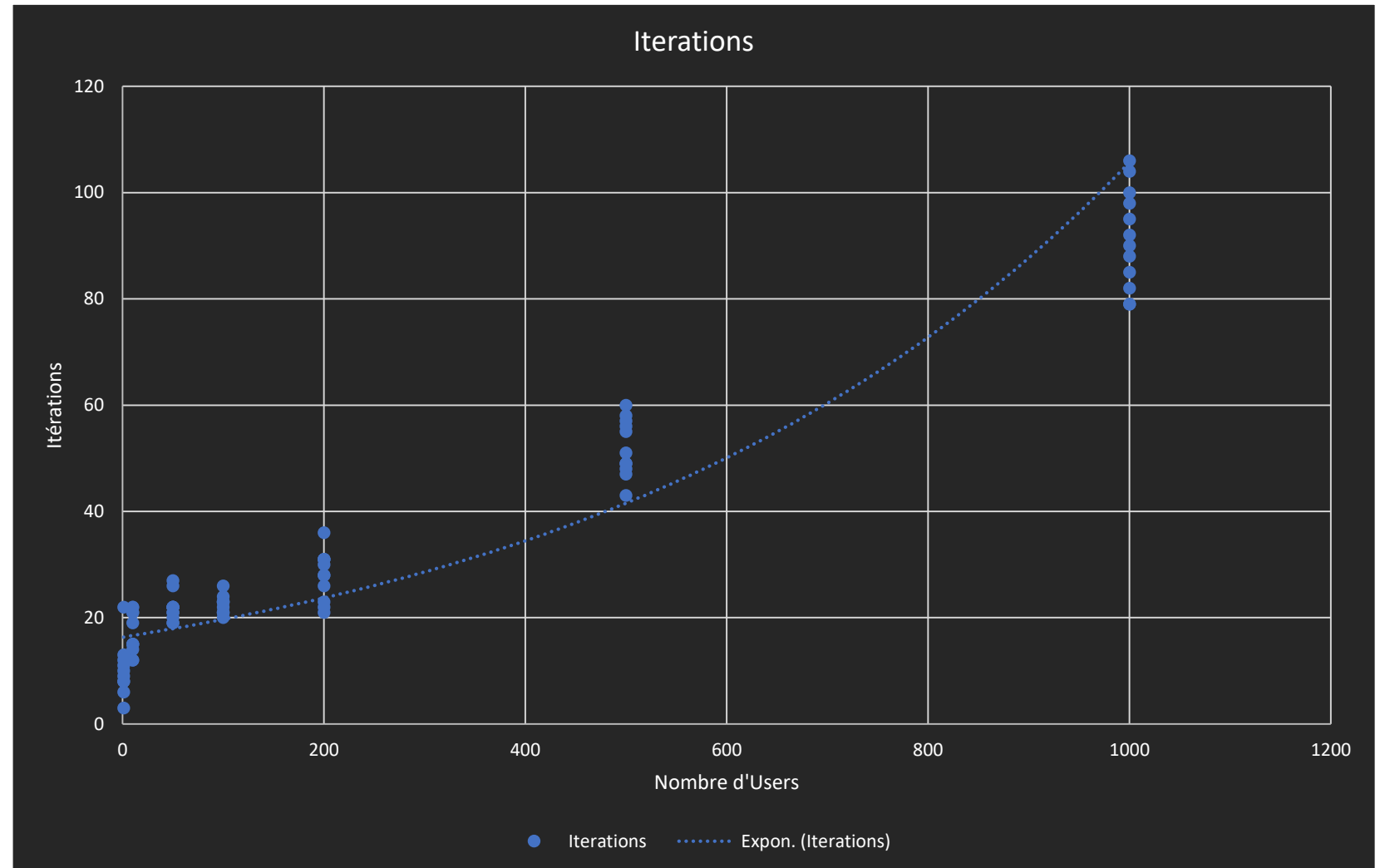


# Statistiques sur notre système

Coefficient de Pearson : 0,979242557

Le nombre d'itérations du système est fortement corrélé au nombre d'Users

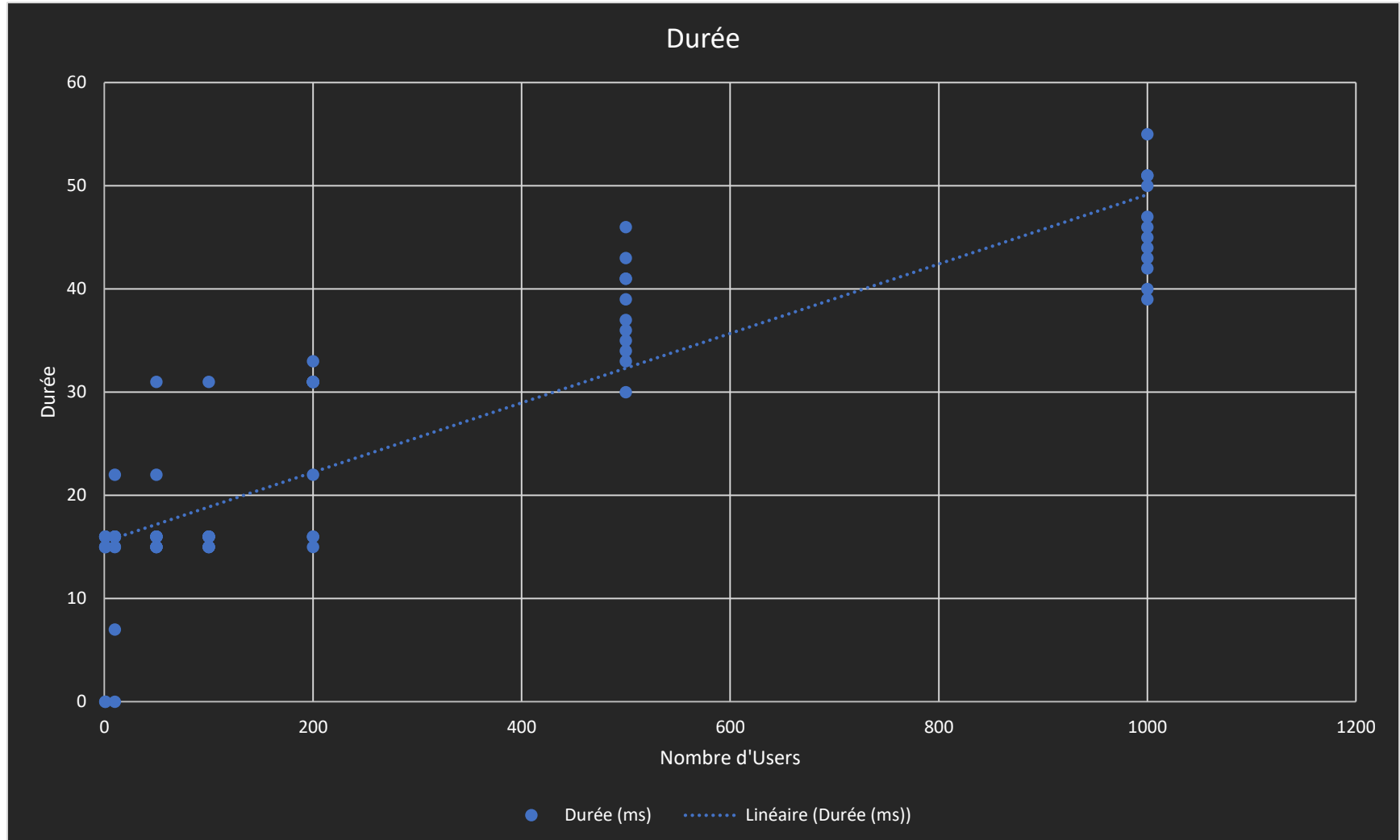
Corrélation  
entre le  
nombre  
d'utilisateur et  
le nombre  
d'itérations



Coefficient de Pearson : 0,883175475

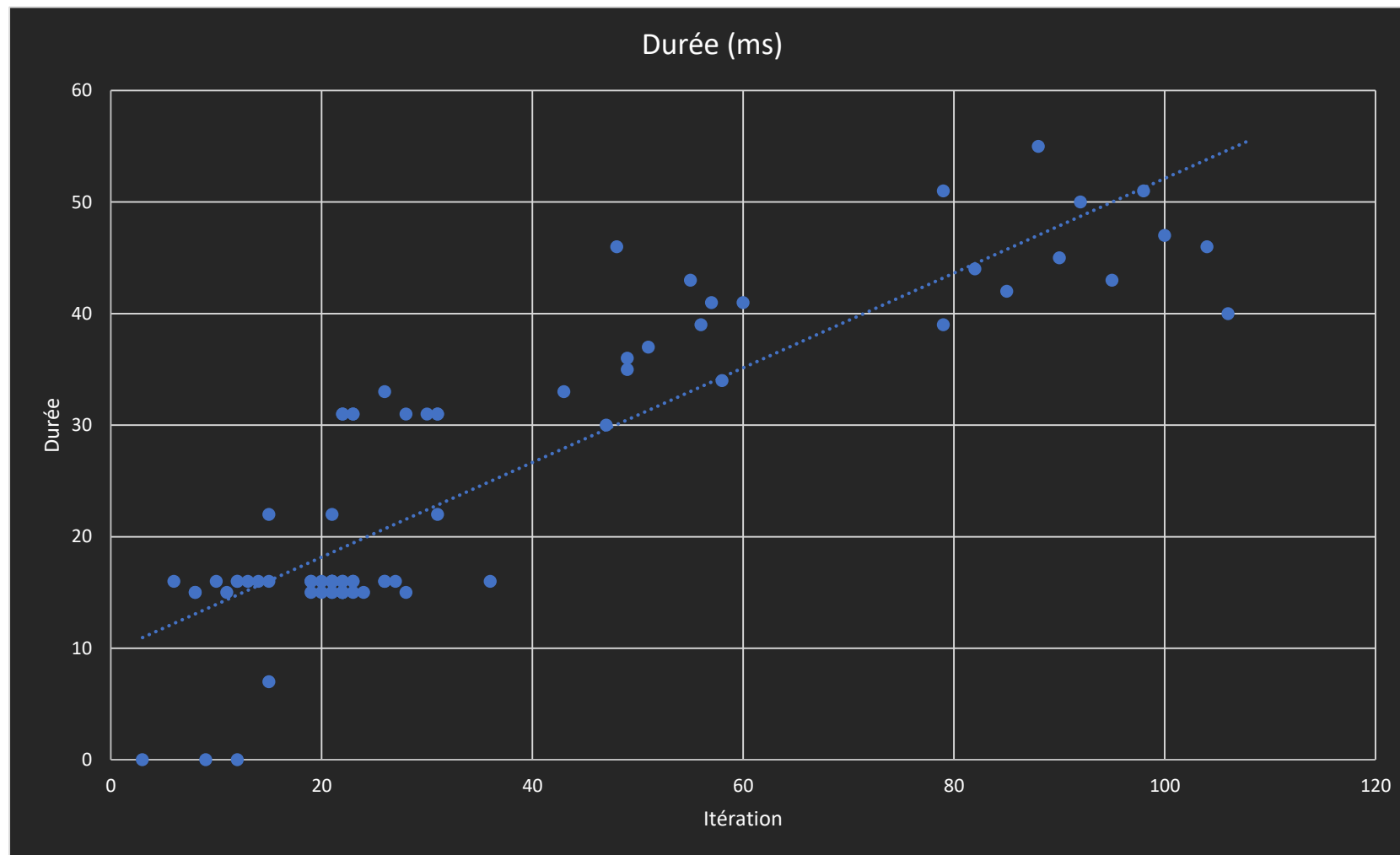
Le temps mis par le system pour effectuer toutes ses taches est corrélé au nombre d'User, Néanmoins cette corrélation est moins forte qu'avec les itération. On peut expliquer cela par le nature des User. En effet, s'il y a beaucoup d'User PMR par exemple, le système dois mettre plus de temps par itérations

Corrélation entre  
le nombre  
d'utilisateur et la  
durée  
d'exécution du  
programme



Coefficient de Pearson : 0,872814409

Corrélation  
entre la durée  
d'exécution du  
programme et  
le nombre  
d'itérations



Données Brute

