Access Control System

Software design, course 2023-24

1 Introduction

An access control system (ACS) is a form of physical security that manages who has access to an area at any given time. Access control systems restrict access to authorized users and provide a means to keep track of who enters and leaves secured areas.

It includes the implementation of electrified doors, turnstiles, guards and gates to keep an area secure. In an access controlled-building, authorized persons use credentials (physical, digital via a mobile device, or biometric) to make unlock requests at readers, which send information to an Access Control Unit (ACU). The ACU then triggers the electrified door hardware to unlock if authorized[1].

Many sectors benefit from access control: multi-tenant houses, retail shops, hospitals and healthcare institutions, government facilities, airports, companies with secured areas like oil and gas facilities and data centers etc. Examples of commercial cloud-based access control systems are Aviglion Alta, Kisi, SALTO Systems and ProDataKey.

Modern ACS have the two following key characteristics: 1) they can work with credentials embedded in smartphones and 2) the ACU is in the cloud.

Mobile credentials. Doors giving access to secured areas have reader devices that get the user's credential from an app installed in his/her mobile phone (figure 1). Mobiles are better devices to keep a user credential than cards in that they are not easily shared and have their own additional security measures.

Mobile credentials let you use your smartphone to unlock entries. In the access control administrative software, a user is assigned a mobile credential. The user installs the access control mobile app on their smartphone, logs in, and approaches a reader. The user then makes an unlock request using their smartphone — either by tapping a button in the app, holding up the phone to the reader, or by simply touching the reader with their hand while their phone is in their pocket or bag. This request can be sent to the ACU through the reader via Bluetooth Low Energy, via WiFi, or cellular data. Once the mobile credential is authenticated and authorized, the entry unlocks[1].

In addition, the app is a software with a graphical user interface and this opens the possibility to much more functions than simply request to

[1] *Physical access control guide* by Aviglion

Figure 1: Readers by SALTO Systems and Kisi

unlock a door, like lock it, and set it to other states, as we will do. And importantly, through the app the users granted with the right permissions and system administrators can perform these and other actions (like adding users, changing their permissions etc.) *remotely*.

Cloud-based ACU. The ACU is a software in the cloud. It stores the knowledge of who can enter where and when, and not only enter but also perform other actions like remotely lock, unlock a door, or all the doors giving access to a space etc. Users communicate with the ACU on-site through the readers attached to a door, or directly through their computer or mobile, remotely. The ACU exposes an API that, for instance, follows the REST protocol so that users send requests (GET, POST etc., more on this later) that may change the state of one or more doors.

The whole practicum consists of a first iteration of a project to build an ACS system. Of course this is too large and ambitious for a practicum. Instead, our goal is to build a prototype intended to show to users/clients and to gain knowledge about the problem and how to solve it before building the real application, from scratch or reusing part of what we have done.

In doing so, we are going to put into practice the principles of object oriented design, apply some necessary design patterns, learn good programming practices like adopting a coding style and debug with logging, and apply some user experience research and design techniques. This is the main goal.

We will just develop two key parts of an ACS, namely the design and implementation of

1. the "cloud" ACU as a web server, which will actually run locally in your desktop computer but communicating through internet with a client simulator (a web page) and later a mobile app

2. a small part of the user interface for end-users wanting to remotely manage the access to rooms in a building through their mobiles (for instance, the managers or the security staff in a company)

2 Development tools

We will need several plugins to be installed into the development environment (IntelliJ IDEA) and rely on some libraries. We recommend you to install them at the moment they are needed, not all of them from the very beginning.

2.1 First milestone

IntelliJ IDEA will be our IDE (integrated development environment). Install the "community edition" which is free and doesn't ask you to register. It is better to configure now an aspect of the code it generates automatically, in order to be compatible with another tool we will be using later, Checkstyle. Go to `File` → `Settings` → `Editor` → `Code style` → `Java` and change `Indent`, `Tab size` and `Continuation` to 2, 2 and 4, respectively.

PlantUML plugin for IntelliJ. This is to make UML class and state diagrams documenting our design. Open IntelliJ and in the launch screen go to `Configure` → `Settings`. Or create/open a project and then `File` → `Settings`. From there, select `Plugins`, look for `PlantUML integration` and install it. Once installed, we can create a class diagram `File` → `New` → `PlantUML File` → `Class`.

org.json We will need this library to have JSON objects in Java and convert them into strings, and the other way around. In the first milestone this will be our way for the ACU to build the answer to requests, to be sent to the simulator. In the third milestone the Java webserver and the Flutter app will exchange JSON strings. To add this library to your IntelliJ Java project follow these steps: `File` → `Project structure` → `Libraries` → `+` → `From Maven...`, write `org.json:json` and click on search. Install the most recent stable version (not alpha or beta), for instance `org.json:json:20230618`.

2.2 Second milestone

Checkstyle-IDEA , a plugin to helps to adopt a coding style. Once installed go to `File` → `Settings` → `Other` → `Checkstyle` and check `Treat errors as warnings` and select the style `Google checks`, less demanding than Sun checks. To see where the style rules are broken, `Analyze` → `Inspect code...` and two new tabs `Inspection results` and `Checkstyle` will appear at the bottom.

SLF4J and Logback are two libraries we will need to perform logging. Like with `org.json`, now search `ch.qos.logback:logback-classic` and select last stable version. This will install also `org.slf4j:slf4j-api` and `ch.qos.logback:logback-core`.

2.3 Third milestone

Flutter is "*Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase*". Follow installation instruction of the *Get started* section. This includes installing Android SDK (maybe by installing Android Studio, an IDE that we won't be using). Also, you'll need to install an android virtual machine (AVD) within IntelliJ that emulates some specific mobile phone on which will run our Flutter apps. But don't rush, wait until you start the third milestone. We will provide not a Flutter source code but a tutorial to build the starting point of the app.

3 Starting point

3.1 The simulator

In order to test the ACU server, and later the mobile app, we are not going to have real credential readers and doors, of course, but we will simulate them by software. We are providing a simulator in the form of web page (Html + JavaScript) that is able to display the blueprint of a test building, its doors, readers and rooms, and interact with all of them. Figure 2 shows the simulator interface.

The simulator lets you to choose a certain user, action, date and time, and either a reader (same as an individual door) or an area. An area is a group of one or more doors, identified by the labels in the blueprint (like *building*, *basement*, *parking* ...). Then, builds and sends the corresponding request (either a reader or an area request) to the ACU server, receives and prints its answer and updates the blueprint accordingly, that is, paints the states of the affected doors and open or closes a door.

Note that it is prepared to test the ACU server that you will extend, that includes more functionality than the one we are providing. For instance, each user belongs to a different group, each group should have different access permissions.

A reader request concerns only one door, and you can choose any of the five actions: open, close, lock, unlock and unlock shortly. Action *unlock shortly* is intended to briefly unlock a locked door (not an area), just for a few seconds so as to be able to cross it, in the event of the selected user has permission. See the UML states diagram of figure 7.

An area request is intended to affect one or more doors, all those under a certain space (room) or partition (group of spaces and/or other partitions). We have decided that in this case only the actions lock and unlock make sense. So, the simulator prevents you from selecting a different action, and the ACU will check than an area request has one of the two valid actions.

3.2 Requests

Requests are messages that the simulator or the mobile app send to the server. The server parses such messages and instantiates a request object

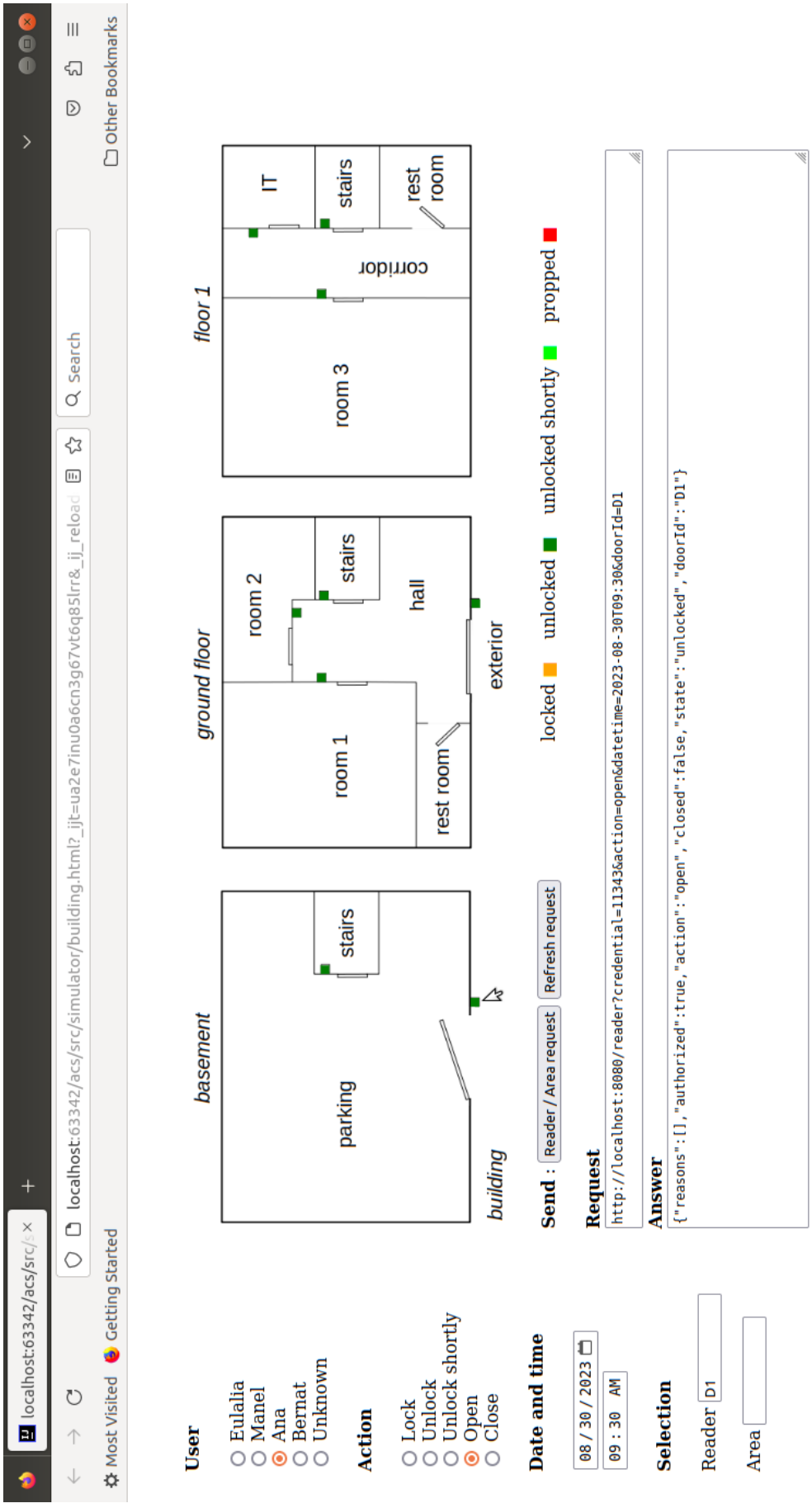

Figure 2: Simulator.

of the corresponding class, that is to be processed. The simulator and later the mobile app will issue such requests. There are four types of requests the ACU server can receive and process :

1. `refresh`, to get the state of each door, in order to display it in the simulator, so this is sent only by the simulator
2. `reader` about an user requesting a change in a door : open or close it for the moment being, later lock or unlock, and finally unlock shortly also
3. `area`, lock and unlock all the doors belonging to an area
4. `get_children` to get the state of the doors directly under an area, this has to do with the concept of areas and the mobile app to be developed at the end of the practicum

Only the processing of `refresh` requests is fully implemented in the code we provide. For reader requests, only the open and close actions. An important hint: to process and area request it suffices to transform it into the equivalent reader requests and process them.

The server is a simple web server built in Java with "server sockets" so we adopt a standard protocol for the format of a request known as Rest or Restful API[2]. In practice this means that the request is an http string like

```
http://localhost:8080/request_reader?credential=11343
↪  &action=lock&datetime=2023-09-13T09:30&doorId=D1
```

The simulator receives the response as a JSON[3] string, that can decode to a JSON JavaScript object and access to its named fields in order to update (repaint) the door reader state. The JSON string in response to the former lock request can be for example :

```
{"authorized":true,
 "action":"lock",
 "state":"locked",
 "doorId":"D1",
 "closed":"true"
}
```

The list of the requests types that the simulator plus the app client may send to the web server ACU is in the appendix A. The way http requests are sent and then received and the JSON strings decoded is already programmed, we provide the source code. You only need to understand how it works by reading the code in order to complete the ACU.

[2]Read https://pusher.com/tutorials/backend-developer-part-1/ and https://happycoding.io/tutorials/java-server/rest-api.

[3]See here for a quick intro to JSON in JavaScript.

3.3 The ACU

The ACU is a simple web server implemented in Java with sockets so that it doesn't need external libraries. The server listens to the the address `localhost:8080` for requests in the Rest API protocol of appendix A to be sent by the simulator or the mobile app.

The web server is just a part of the ACU, its façade. It decodes the requests received as strings, builds a `Request` Java object and asks it to be processed. Please read the code of class `Webserver` and follow the execution thread in IntelliJ IDEA[4]. Figure 3 shows the classes of the starting point source code.

Two important classes in the server side are `DirectoryDoors` and `DirectoryUsers`, that instantiate and keep the list of `Door` and `User` objects appearing in the simulator, respectively. All of their methods are public static, for the moment, so they can be invoked from any part of the code without instantiating an object of these classes and passing it as parameter that could be rather cumbersome. In `Main.java` we invoke the static methods that create the doors and users appearing in the simulator. Both classes contain a method to retrieve a certain door and user, respectively, given its unique identifier, a string.

3.4 Limitations

In the starting point the ACU is pretty limited because any door can be open or closed at any time by anyone. There is no way to change the behavior of a door like granting permissions to lock and unlock it only to a certain group of users. Furthermore, the ACU works at door level, meaning if there was a way to change the behavior, it should be applied door by door, not to groups of doors (an area) that would make sense, for instance all the doors of a floor plant.

To check it, in IntelliJ IDEA load the ACU project and run first `Main.java`. Make a configuration with `File` → `Run` → `Edit configurations...` like that of figure 4 with enabled assertions (the `-ea` flag). Then open `building.html` in IntelliJ and click on one of the navigator buttons (we have used Firefox).

Things you can do are :

- see the state of each door through the color of its reader, always unlocked
- open and close doors
- send a refresh request, actually it is sent when loading the page
- unknown users (invalid credential) are recognized and any request they may issue is not authorized

[4]move the mouse on a method name and press Ctrl + click first mouse button, to go its implementation, and Shift+Alt+← and → to navigate back and forward in the traveled code path.

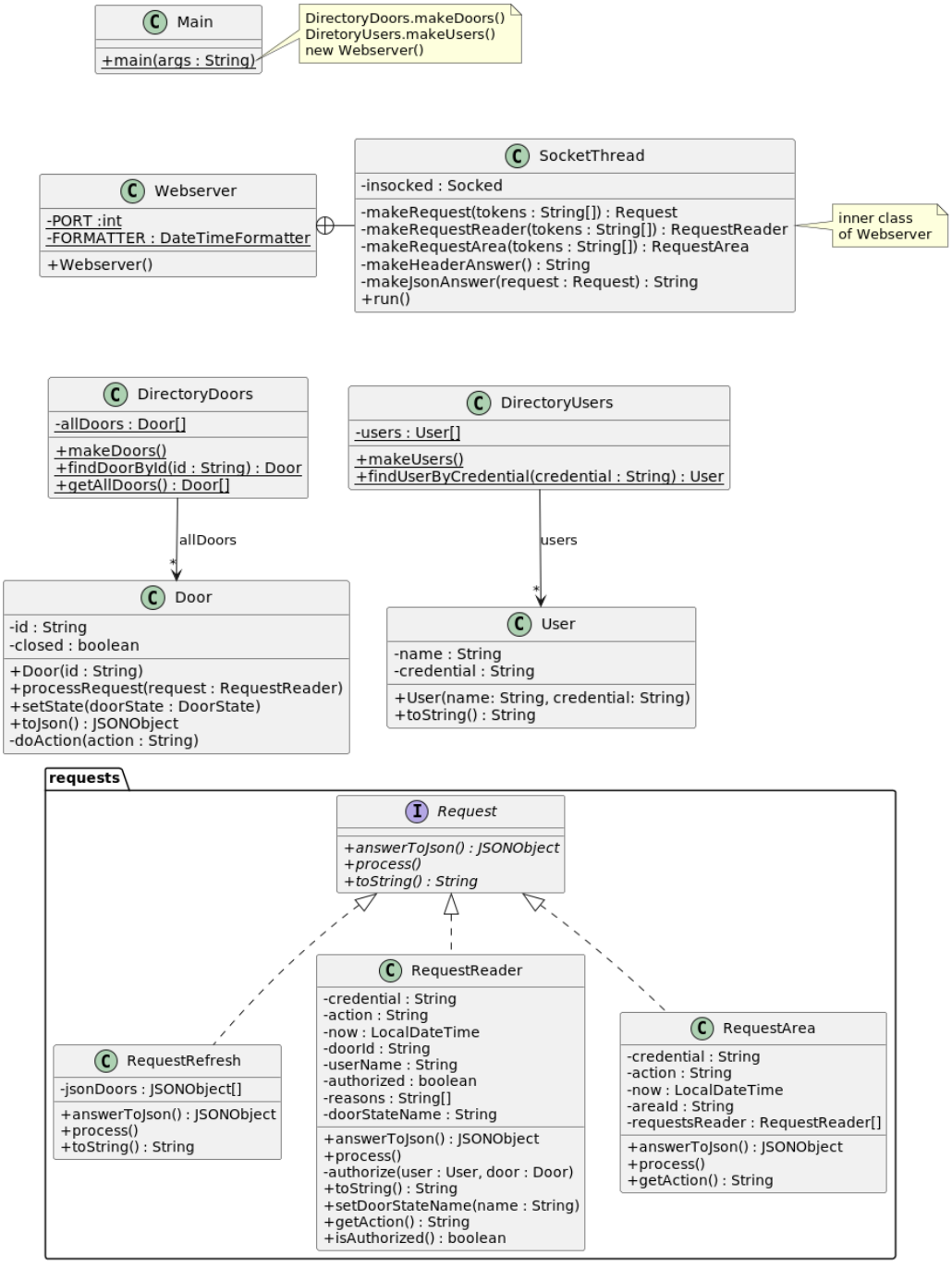

Figure 3: Class diagram of the ACU provided as starting point.

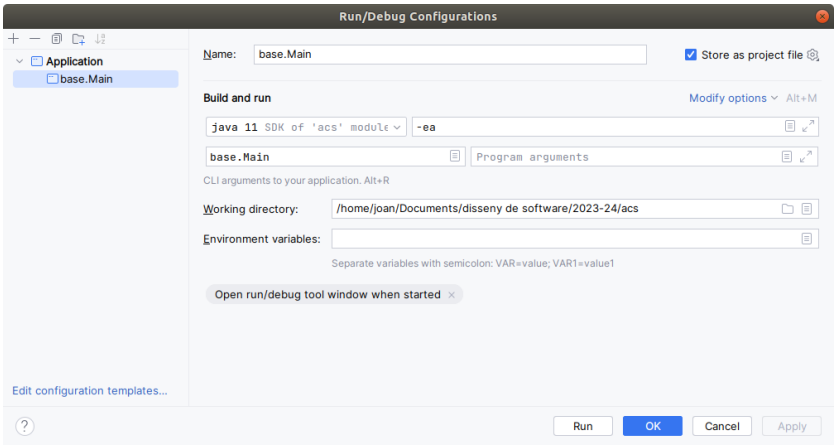

Figure 4: ACU run configuration in IntelliJ IDEA.

However, a lot of features are missing :

- all users can do the same, because there are not user groups nor permissions/privileges associated to a group
- no support to lock, unlock and unlock shortly actions (see below)
- it is not possible to lock/unlock all the doors of an area (the names in the floor plan)

In the following we are going to introduce new features of the ACU to be designed and implemented.

4 Deliverables and authorship

The project is divided in three parts or milestones, each one with its own requirements, deliverables and a grading scheme. Grades depends on how many features have you implemented and their correctness. To deliver a milestone create a zip file with name `milestoneX.zip`, $\texttt{X} = 1, 2, 3$ containing :

- The project folder in another zip file (with the run configuration saved)
- PlantUML files plus PNG images of the class diagrams
- A text file `autors.txt` with name and NIU of students that made this deliverable.

By adding your NIU into the authors file you state your authorship and that your contribution is comparable to that of the other members of the team. Moreover, in `autors.txt` explain who did what, in detail. Saying "we all did all" is not valid.

In case we have asked you during the evaluation to make some minor corrections, add a PDF file `correccions.pdf` explaining what did we asked

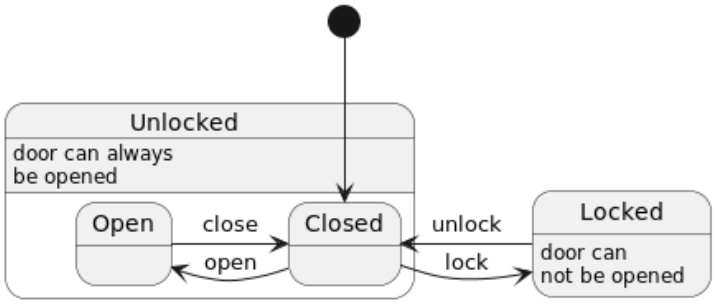

Figure 5: First two states of a door to be implemented.

you, what have you done then, and what has been the result. If necessary insert screen captures to show / prove it.

Just one of the members of the team uploads the zip file to Campus Virtual.

5 First milestone

5.1 First two door states

In the simplest version of ACU that we provide a door can only be opened and closed by anyone at anytime, and remains always in the unlocked state. The first step in increasing the behavior of a door is to introduce the states **Locked** and **Unlocked**. An unlocked door that is closed can be locked, and then no one can open the door unless it is returned to the unlocked state. Still, any system user can lock and unlock a door (but not all the doors in an area). Figure 5 shows the UML state diagram, being the labels of the links between states the names of actions.

5.2 Additional door states

When a door is locked (and necessarily closed) and you want to open it to enter some room, there are two ways. One is to unlock it if you are authorized, wait some seconds and lock it again. But this is a procedure prone to error, what if you forget to lock the door ? So a second, better way is to get it unlocked just for a few seconds so you can open the door and then *automatically* lock it. The reader will send a request to the server saying who you are (your credential, a.k.a password), which door do you want to open, what time and date is it now and the action you want to perform, that is, to shortly unlock a door. These are the four Ws (who, where, when and what) that the server needs to decide if authorize or not your request. If authorized, the door enters into state **Unlocked shortly**, which means the bolt in the door is switched off and you may open the door and enter (or not). After some seconds, the ACU *automatically tries* to return the door to the locked state.

Figure 6: A propped door.

Suppose a door has entered into the unlocked shortly state. Then two situations may happen :

1. The door is opened and then closed before say 10 seconds (i.e, an open and a close reader requests are sent to the ACU), which is the maximum duration in Unlocked shortly state. Or remains closed for 10 seconds. In the second, nothing happens. In both cases after 10 seconds the ACU realizes the door is closed and consequently returns it automatically (i.e. by itself) to the Locked state.

2. The door is opened and remains so for more than 10 seconds. It can not return automatically to Locked like before because this would be inconsistent, since the door is still open. Hence, the ACU sets the door state to **Propped** (figure 6). This way the ACU could raise an alarm: a door that should be closed is open. Once in this state, if the reader at some point sends a close request, the ACU returns it to Locked.

Figure 7 shows the state diagram corresponding to this description, which is an extension of that in figure 5.

In the future, we will need to implement even more states, like *Do not disturb*, that would be more restrictive than Locked, and *Idle* which would be like Unlocked (anyone can open the door) but that can not be locked. Or custom states defined by clients. Hence, we have to make a design extensible in this direction of change.

5.3 Spaces and partitions

Consider a building with several floors, like the top of figure 8. A door communicates two spaces, like a corridor and a room. The door of the main entrance gives access to the entrance hall when you come from the exterior,

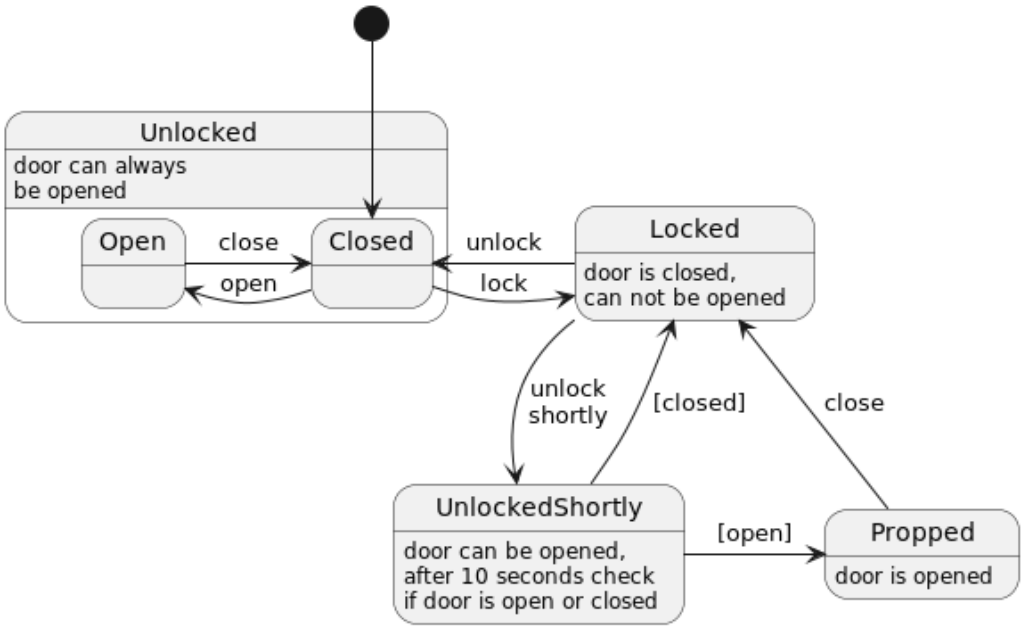

Figure 7: All the states of a door. Transitions are due to actions, some of them under certain conditions enclosed in brackets. Each group of users is authorized to perform a different subset of actions, from none (blank group) to all (admin).

an special space. A space has at least one door, but can have more, like the parking.

A door may have associated a reader of credentials in one of the two spaces it communicates (the small dark square). The reader is in the "from" space, and when the door opens you can enter the "to" space with respect to the reader. When we grant a group of users with access permissions to a space they affect the doors having it as "to" space. Consider if we will need each door to know its "to" and "from spaces" in order to implement user groups permissions, and if each space has to know the doors giving access to it, i.e. that have it as "to" space.

Our access control system not only has spaces which have doors, but also partitions. A partition is a group of one or more spaces and/or other partitions. This is because we want the administrator of the ACS to grant access privileges to an individual space or to a set of spaces, *but not directly to doors*. Working only with individual spaces would be impractical in the case of large buildings, with many spaces in several floors, or even several buildings. It is better to let the administrator to group hierarchically the spaces, for instance by floors, and then grant access floor by floor. Or to all the spaces belonging to a department plus shared rooms. Or to the whole building. In any case, to the grouping that makes more sense to him/her. Figure 8 shows several possible hierarchies. We will call such groups *partitions* because group is maybe a too generic name and partition has been employed for this purpose by one of the cited commercial systems.

Each door, space and partition has a unique identifier which is a name, like “parking”, “basement” and like “D1”, “D2” for doors. Identifiers will be used to search these elements and are displayed in the simulator and the user interface of the app.

5.4 Users, groups and schedules

Users are individual persons recognized by the ACS as a user because he/she owns a valid credential (a.k.a unique password) in the app installed in his/her mobile phone. Naturally, not all users will have equal access rights. “Rights” mean who can do what, where and when. Most physical access control systems follow a standard approach known as Role-based access control (RBAC):

> Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members or staff (or other system users) are assigned particular roles, and through those role assignments acquire the permissions needed to perform particular system functions. Since users are not assigned permissions directly, but only acquire them through their role, management of individual user rights becomes a matter of simply assigning the appropriate role to each user account[5].

So, a role is a set of access rights granted to a group of persons according to what they do in an organization. These rights are the set of the actions anyone of them can do on a set of places at certain times. Places are spaces (like a specific room) and/or may be partitions also. The later is very convenient for the administrator because a granting permissions for one partition may mean to do it for a lot of spaces in one go. Remember this means the actions that can be performed on the doors giving access to those spaces.

The times dimension is usually specified by a schedule: from an initial to an end date, which days of the week and times in the day (same for all days) these rights apply. For instance, a user in the ”undergraduate student” role/group has access to certain laboratories Q5/1023 and Q7/0073 from 9:00 to 18:00 Monday to Friday, starting Sept. 12 until Dec. 23 this year. This is the same operating systems do: in Linux there are user groups that have read, write (create, delete) and execute privileges on files or folders of the file system (but they apply always, there are no temporal constraints).

To begin with, we will work with the following roles or user groups, in relation to the building plan shown in figure 2:

- Administrator: can do anything, anywhere, anytime.
- Manager: Sept 1 2023 to Mar. 1 2024, week days plus Saturday, 8:00 to 20:00, all actions, all spaces.

[5] Wikipedia entry RBAC

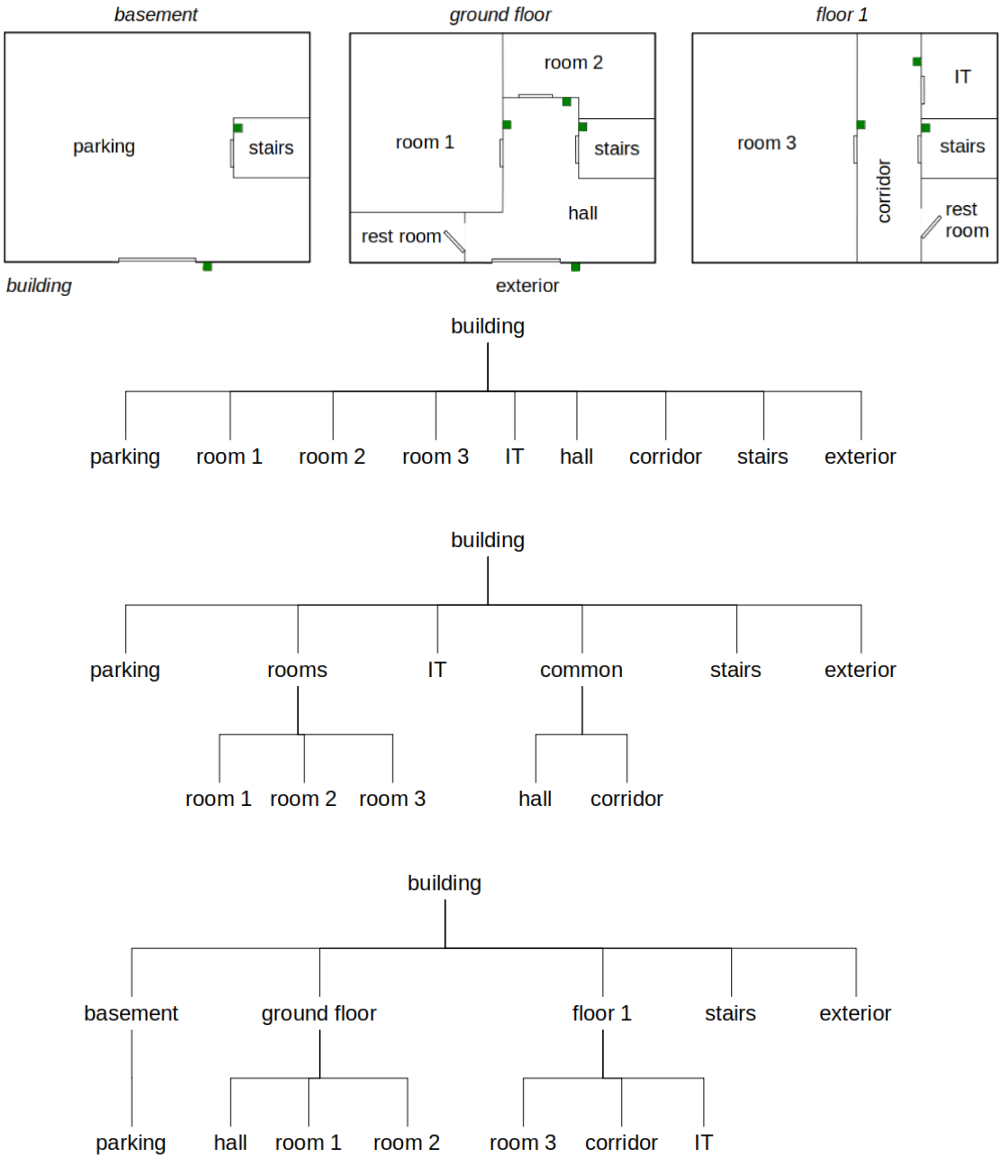

Figure 8: Top: plan of a three-story building. Bottom: three possible hierarchies of areas for the building, being the last the one chosen.

- Employee : Sept 1 2023 to Mar. 1 2024, week days 9:00 to 17:00, open, close, unlock shortly everywhere but the parking.
- Blank : group without any privilege, just to keep temporally users instead of deleting them, this is to withdraw all permissions but still to keep a user's data to give back permissions later if needed.

The design has to be flexible enough to change the schedule of a group, add, suppress or edit groups, schedules and users, though we are not going to implement these operations.

5.5 Good practices

5.5.1 Debugging with *prints*

Surely you will write a lot of sentences to print the value of variables during execution, to trace it, to make sure it works as intended, or to find errors. This is good and easier than using the debugger, for several reasons we will see. Once the code seems to works well, these messages become annoying and you want to get rid of them by deleting the print sentences. Don't ! Simply comment out them because in the next milestone you are going to transform the `System.out.println` sentences into something else called *log messages*. Yours and ours.

5.5.2 Comments

Comments are necessary. But what and how to comment ? We will study this at some point, and will apply some rules / recommendations regarding comments, see section 6.1. For now you have to write comments to explain for instance things that you have found difficult to program, or design decisions you think are important (Why a space has to know its doors ? What is a group of users and why do we want them ? etc.) Some of these things are explained in this handout, but the programmer that would maintain your code won't have it!

Same goes for names: choose carefully the names for classes, attributes, methods and variables. We mean not only to follow the (for the moment implicit) Java conventions like classes start with a uppercase letter, methods, attributes, variables with lowercase letter, compound names are like `DoorState`, `DirectoryUsers`. We mean that names should avoid the need of comments explaining the meaning of the named entities.

We are asking you to write such comments as a first step to be continued in the second milestone, where writing good comments will be mandatory.

5.5.3 Test cases

There are a lot of features in this first milestone and we have just some minutes to check they work and are well designed and programmed, view the UML class diagram, ask questions to make sure every member of the team has worked on it etc. So you have to prepare a list of test cases to be

executed in order to *convince* us in the shortest time possible that everything you claim you have done, is done and is right.

5.6 Hints

5.6.1 Useful Java classes

The `java.util.time` library has some useful classes to represent dates, time and duration: `LocalDate`, `LocalTime`, `DayOfWeek`, `Duration`. See their usage in listing 1. Note also there how `Arrays.asList()` is used to initialize an `ArrayList`.

Listing 1: Useful Java time and date classes

```
ArrayList<DayOfWeek> monToFri = new
   ArrayList<>(Arrays.asList(DayOfWeek.MONDAY,
   DayOfWeek.TUESDAY, DayOfWeek.WEDNESDAY,
   DayOfWeek.THURSDAY, DayOfWeek.FRIDAY));
ArrayList<DayOfWeek> monToSat = (ArrayList<DayOfWeek>)
   monToFri.clone().add(DayOfWeek.SATURDAY);
LocalDate d = LocalDate.of(2023, 9, 1); // Sept 1 2023
LocalTime h = LocalTime.of(9, 0); // 9:00 AM
LocalDateTime dt1 = LocalDateTime.now();
//...
LocalDateTime dt2 = LocalDateTime.now();
Duration duration = Duration.between(dt1, dt2);
```

5.6.2 Counting time

This refers to the 10 seconds we have to wait for before a door in the shortly unlocked state transitions to either locked or propped if it has or has not been closed, respectively. It seems a simple thing to implement like in listing 2.

Listing 2: How *not* to wait for 10 seconds

```
try {
    Thread.sleep(10000); // miliseconds
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

The problem of this code is that during 10 seconds the application pauses and does not execute anything else than `sleep`, so you can not interact with it to open, lock etc. another door, print something etc. In other words, it becomes *unresponsive*. What we need is to wait for 10 seconds before

checking if the door has been closed or not to change to a new state and, *in parallel*, continue executing the ACU.

The solution is to have a second thread running in parallel with the main one. In this second thread we will have a unique clock that is an Observable, while all the doors in the unlocked shortly state are its observers. The clock tells those doors what's the time every second, so they can decide to swich to state propped or unlocked, or remain in unlocked shortly.

In addition to `Observable`, `Observer` in `java.util`, study in this same package `Timer`, `TimerTask` and its method `scheduleAtFixedRate`. Their joint use allows to create a inside a clock, a thread that executes some function periodically (like every second) by means of the timer's method. As for usage examples, see for instance [6,7].

5.7 Grading

E Nothing or too few new done beyond the starting point, or does not work at all.

D No comments added, or too few or useless. Added door states locked and unlocked. The ACU still works at door level (no partitions or spaces), like in the starting point code.

C In addition to locked and unlocked states, added spaces and partitions (areas), but not yet user groups and schedules. A user has associated areas and upon a reader request or an area request, you check for each door involved that the user can stay in its "from" and "to" spaces. There is a sufficient number of good comments.

B C + introduced user groups and schedules, access permissions are granted to groups based on areas and schedules.

A B + introduced states propped and unlocked shortly which work well.

To all items add that the UML class diagram has to be updated according to the changes in the code.

6 Second milestone

6.1 Comments and names

Code must have good comments. Do not assume the programmer that will read your code has also read this handout or knows anything about the project, but imagine he/she is the first time hears about it and has to understand the code to maintain it. Follow the guidelines we have explained in the classroom on what are good and bad comments.

[6] https://www.baeldung.com/java-timer-and-timertask

[7] https://alvinalexander.com/source-code/java/java-timertask-timer-and-scheduleatfixedrate-e

Furthermore, each class should have a header comment explaining its purpose, how does it fit into the design, whether it is part of some design patter and why. Don't state the evident like "Class DoorState is abstract", ' "A DoorState is the state of a door". Instead explain what is a door state and why do we need a door to have an associated state, who and when sets the state of a door, that we are applying the state design pattern and why. Comments should explain also code (and design decisions) difficult to understand, not obvious because the way it is (uses not common features of Java or some library) or its purpose. Any tricky or important detail has to be explained. On the other hand, classes, attributes, methods, variables, constants will have sensible names that reduce the amount of comments needed. Finally, do not write Javadoc comments.

6.2 Style

Almost all the Java code (yours and mine) has to follow the style enforced by Checkstyle, except for javadoc rules. Occasionally you can deviate from the style, it is already ok if you remove 90-95% of the issues raised by Checkstyle.

Checkstyle helps us to write the code in a certain style, but complementary to it are the "lint" type tools that check for source code *possible* bugs. Do `Analize` → `Inspect Code...` and solve the issues in the category `Java`. Not all of them, only those that you think are real enhancements.

6.3 Logging

Add the file `logback.xml` to the project in the `src/` directory. Then transform the `System.out.println` sentences we and you have written in the first milestone, and in this second also, into proper logging messages. Remember that each class must declare a `Logger` object. We are asking yo to do the following:

- Write messages of Debug, Info and Warning levels and check they appear or not when you change the level or the root logger. For instance, trace how a door request is made and processed. Issue a warning message when a door enters the propped state or you unlock an already unlocked door. And debug messages to implement the refactoring with the new design pattern.
- Prepare the file logback.xml so that you can choose, by commenting out and uncomment some lines, whether to : 1) log messages from classes of the first milestones only, 2) from classes of this milestone only, 3) from all classes.
- Send the messages to the console and also a to a text or Html file.

6.4 Singletons

Some classes in our design are singleton, that is, during an execution we will make at most one instance of each of them. However, the code doesn't show

it, at least in an explicit way. Apply the Singleton design pattern to these classes and modify the code that uses them accordingly.

6.5 Some refactoring

In order to implement the first milestone you probably have coded methods to traverse the tree of partitions, spaces and doors for several purposes :

- to get the list of the doors under a partition (in an area request)
- to find a partition or an space (i.e an area) by its id (to create a user group)
- to get the list of spaces under an area (to authorize a request)

We foresee the need of even more traversals, like making the list of propped doors, so more methods will be added to the partition and space classes. We want to move the traversal methods out of these classes so that implementing new functionalities needing to traverse the tree doesn't affect the tree classes, but still be able to traverse the tree doing something different each time. For this you need to apply a certain design pattern. Once done check the code still works well.

6.6 Grading

E There are too few comments or most of them are bad comments. The Checkstyle rules break at too many places.

D Something in the middle of E and C.

C Good comments and naming. The only (or almost) Checkstyle rules that break are Javadoc related. Your and our `System.out.println()` sentences have been converted to logging messages at the proper level (DEBUG, INFO or WARNING, maybe even TRACE) that are output to console and a file

B C + + Singletons

A B + Refactoring

For B and A of course update the UML class diagrams.

7 Third milestone

Now that we have implemented the basic functionality and tested it by means of the simulator it's time to make the user interface for the mobile app. As shown in figures 1, the app installed in a user's mobile interacts with the reader in order to open doors. However, we obviously can not implement and test this feature in the practicum since there is no real reader is available.

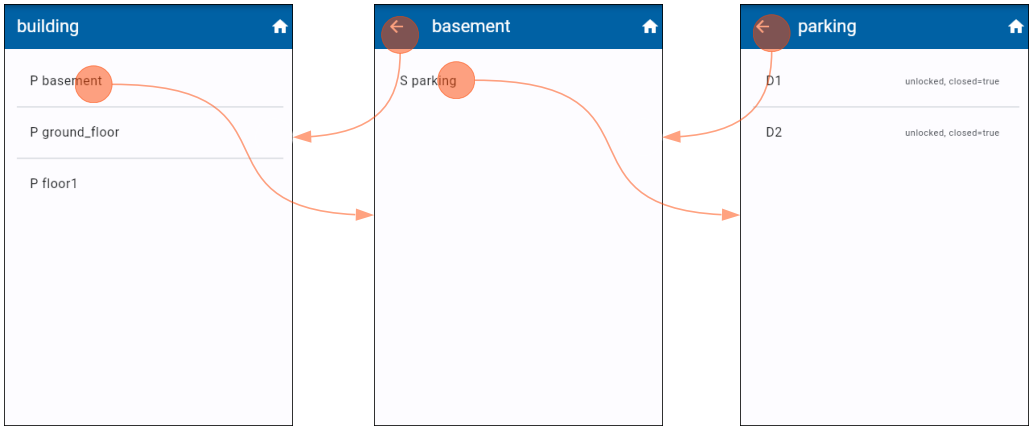

Figure 9: Client app in Flutter with navigation in the hierarchy of partitions and a drawer.

Hence, we will implement in the app only the features corresponding to its usage in remote mode : open, close, lock and unlock an individual door or the doors in a space or a partition (but not unlock shortly). Actually, what maybe makes more sense are the lock and unlock operations. In the end, however, all these actions amount to send requests to the ACU.

Therefore, the app should allow the user to, at the very least,

1. navigate through the tree of partitions, spaces and doors
2. show if doors are closed or open, and their state, in a way the user will easily understand
3. send to the ACU reader and area requests for the lock and unlock actions, and then process the answer to show the updated state of doors or area

We will provide a tutorial to build a Flutter app that only implements the navigation part. Therefore, it lacks a lot of usability (figure 9) : you can not send any request, in the interface there is not distinction between a door, an space or a partition in a way the user will understand which is a must etc. You can take build the third milestone on top of it.

7.1 Grading

E Less than D : navigation doesn't work (including the home button) or what is a partition, space and door is not crystal clear

D Items 1 (plus home button) and 2

C D + item 3

You can get **B** and **A** by increasing in diverse ways the usability of the interface beyond **C**. Here are some suggestions:

- Add the open, close, lock and unlock actions : not only allow to perform them but also show their result.
- If all the doors of a space or partition are locked, or they all are unlocked (i.e no door is in another state), denote it.
- If the user orders to lock or unlock all the doors in an area but this is not possible because there is at least a propped door, or in unlocked shortly state, show a message.
- Choose intuitive icons, colors or whatever to display the state of areas and doors, and the possible actions.
- Also, there are things the interface could do that are not a direct transposition of a functionality of the server. For instance,
  - have a shortcut to the list of the most recent doors / areas, so as to reduce the number of navigation clicks
  - another shortcut to the list of propped doors
- More interestingly, internationalize the app and localize it to 3 locales: Catalan, Spanish and English.

A Web server API

Table 1: Requests that the web server we provide accepts and corresponding answer. `get_children` will be sent only by the app. It is not yet implemented in the ACU.

`http://localhost:8080/refresh`	`{"doors":[`
	`{"originSpace":"exterior",`
	`"destinationSpace":"parking",`
	`"closed":true,`
	`"id":"D1",`
	`"state":"unlocked"},`
	`...`
	`{"originSpace":"corridor",`
	`"destinationSpace":"IT",`
	`"closed":true,`
	`"id":"D9",`
	`"state":"unlocked"}`
	`]}`

`http://localhost:8080/reader?`
`↪ credential=43295&action=lock`
`↪ &datetime=2023-09-21T09:30`
`↪ &doorId=D1`
	`{"reasons":[`
	`"Not allowed destination parking for`
	`group employees",`
	`"Not allowed action lock for group`
	`employees"],`
	`"authorized":false,`
	`"action":"lock",`
	`"state":"locked",`
	`"doorId":"D1",`
	`"closed":true}`

`http://localhost:8080/area?`
`↪ credential=95783&action=lock`
`↪ &datetime=2023-09-21T09:30`
`↪ &areaId=basement`
	`{"areaId":"basement",`
	`"action":"lock",`
	`"requestsDoors":[`
	`{"reasons":[],`
	`"authorized":true,`
	`"action":"lock",`
	`"state":"locked",`
	`"closed":"true",`
	`"doorId":"D1"},`
	`{"reasons":[],`
	`"authorized":true,`
	`"action":"lock",`
	`"state":"locked",`
	`"closed":"true",`
	`"doorId":"D2"}`
	`]}`

Table 2: Continuation.

`http://localhost:8080/get_children?`
`↪ basement`
	`{"areas":[`
	`{"access_doors":[`
	`{"originSpace":"exterior",`
	`"destinationSpace":"parking",`
	`"closed":true,`
	`"id":"D1",`
	`"state":"locked"`
	`},`
	`{"originSpace":"stairs",`
	`"destinationSpace":"parking",`
	`"closed":true,`
	`"id":"D2",`
	`"state":`
	`"locked"`
	`}`
	`],`
	`"id":"parking",`
	`"class":"space"}`
	`],`
	`"id":"basement",`
	`"class":"partition"`
	`}`