# Milestone 2

1. Coding style
2. Comments and names
3. Singletons
4. Refactoring

> Shorter milestone, only 2 (Wed) / 3 (Thu, Fri) sessions

## 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.

> Mechanical task but you can learn more of Java by looking at the *lint* warnings.
>
> Add comments justifying why do you need to break Checkstyle rules other than javadoc

## 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.

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.

> Not mechanical at all ! Do your best here, we'll assess the quality of comments.
>
> Have them written by next session if you want our feedback.

## 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

> Which classes ?
>
> Do you remember how to make a class singleton ?

## 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.

Now we have:

```
public abstract class Area {
  public abstract ArrayList<Door> getDoorsGivingAccess();
  public abstract Area findAreaById(String id);
  ...
}
```

```
public class Space extends Area {
  @Override
  public Area findAreaById(String id) {
    if (this.id.equals(id)) ...
  }

  @Override
  public ArrayList<Door> getDoorsGivingAccess() {
    return this.doorsGivingAccess;
  }
}
```

```
public class Partition extends Area {
  @Override
  public Area findAreaById(String id) {
    if (this.id.equals(id)) ...
  }

  @Override
  public ArrayList<Door> getDoorsGivingAccess() {
    ...
  }
```

In the future →

Want to move these 4 methods out of the tree classes because they make them

- complex
- non-cohesive
- prone to be changed

Also, each type of process is split into two classes.

Which design pattern ?

How to traverse the tree (order) ? Who knows how to do it ?