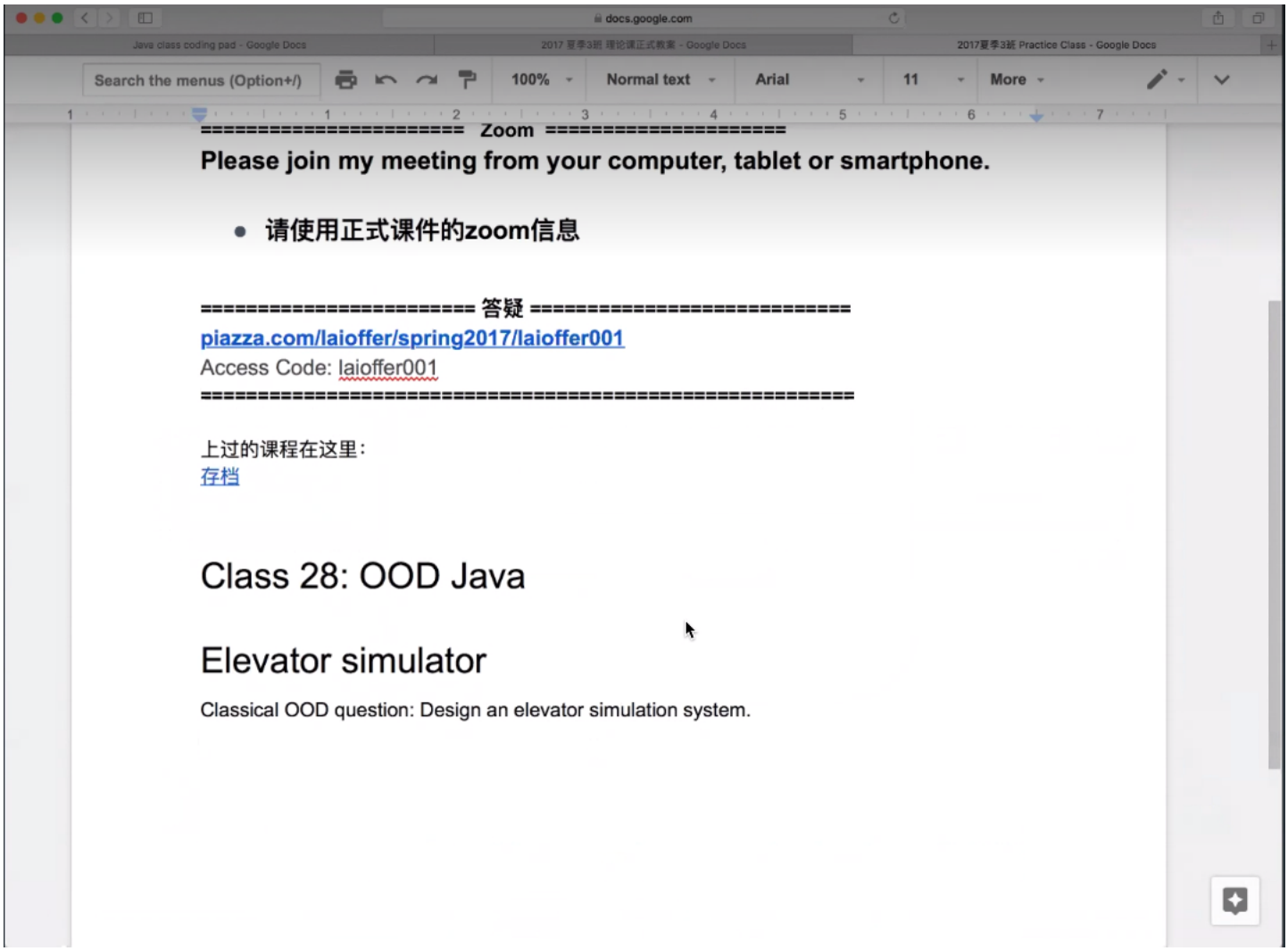
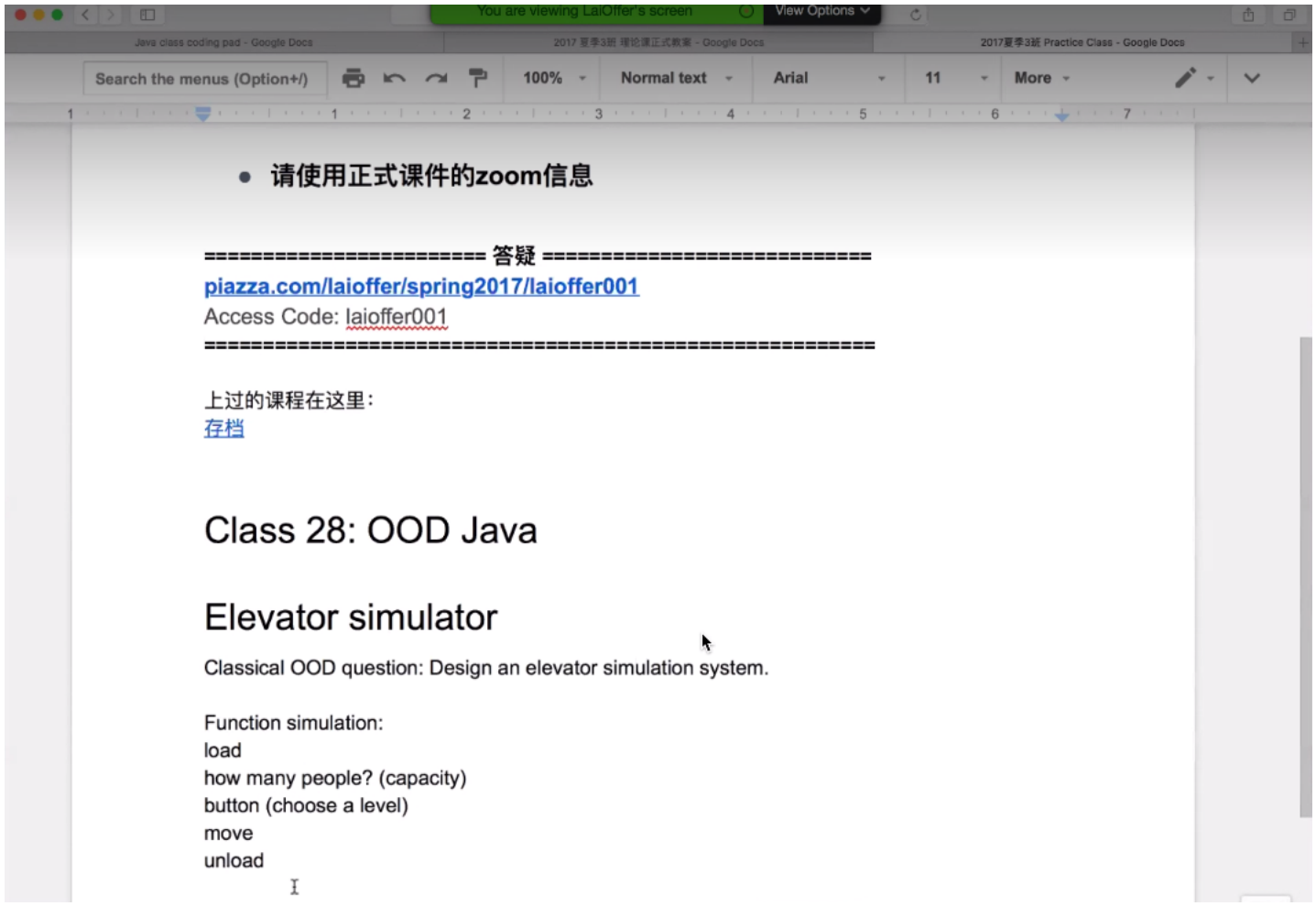
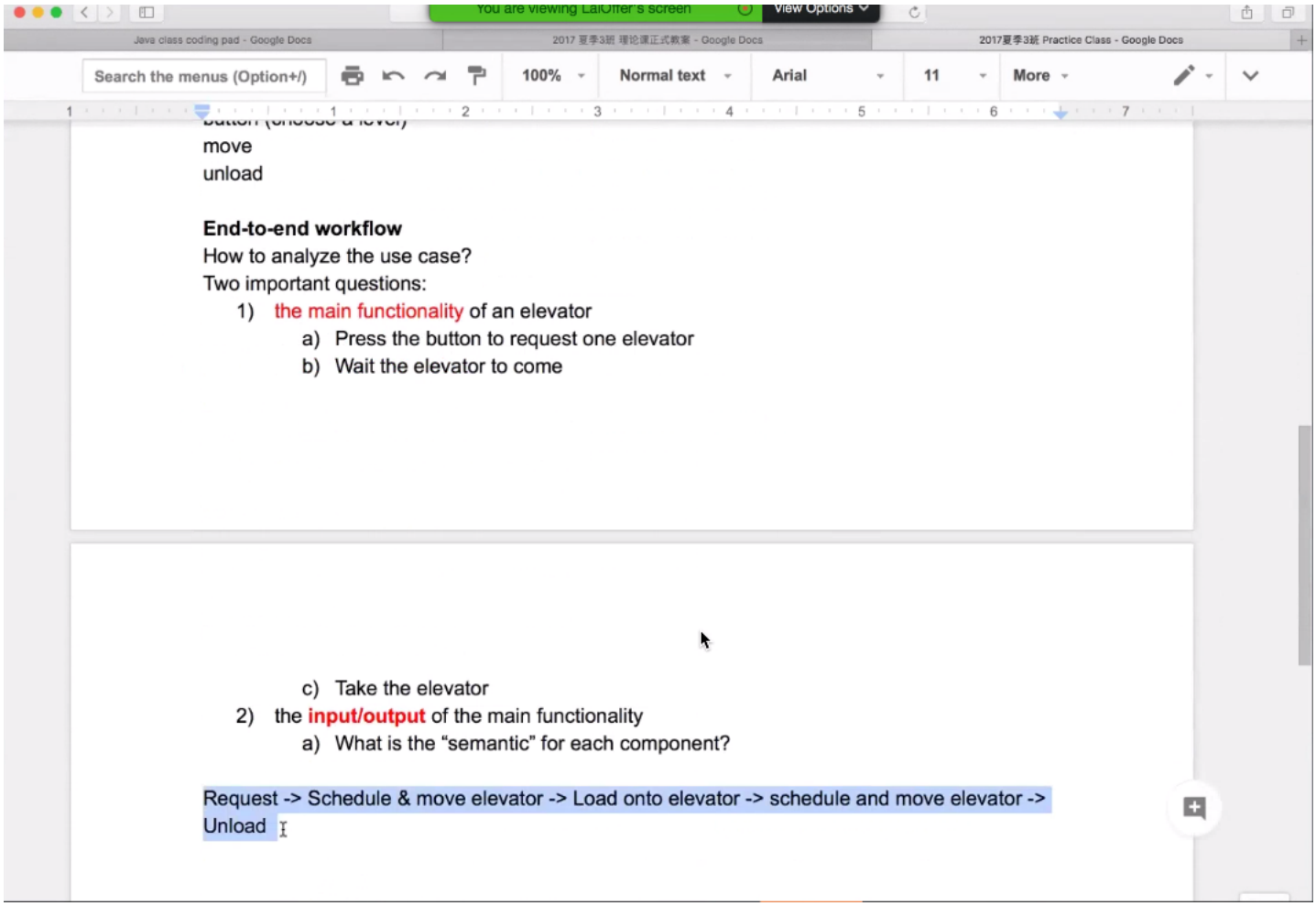
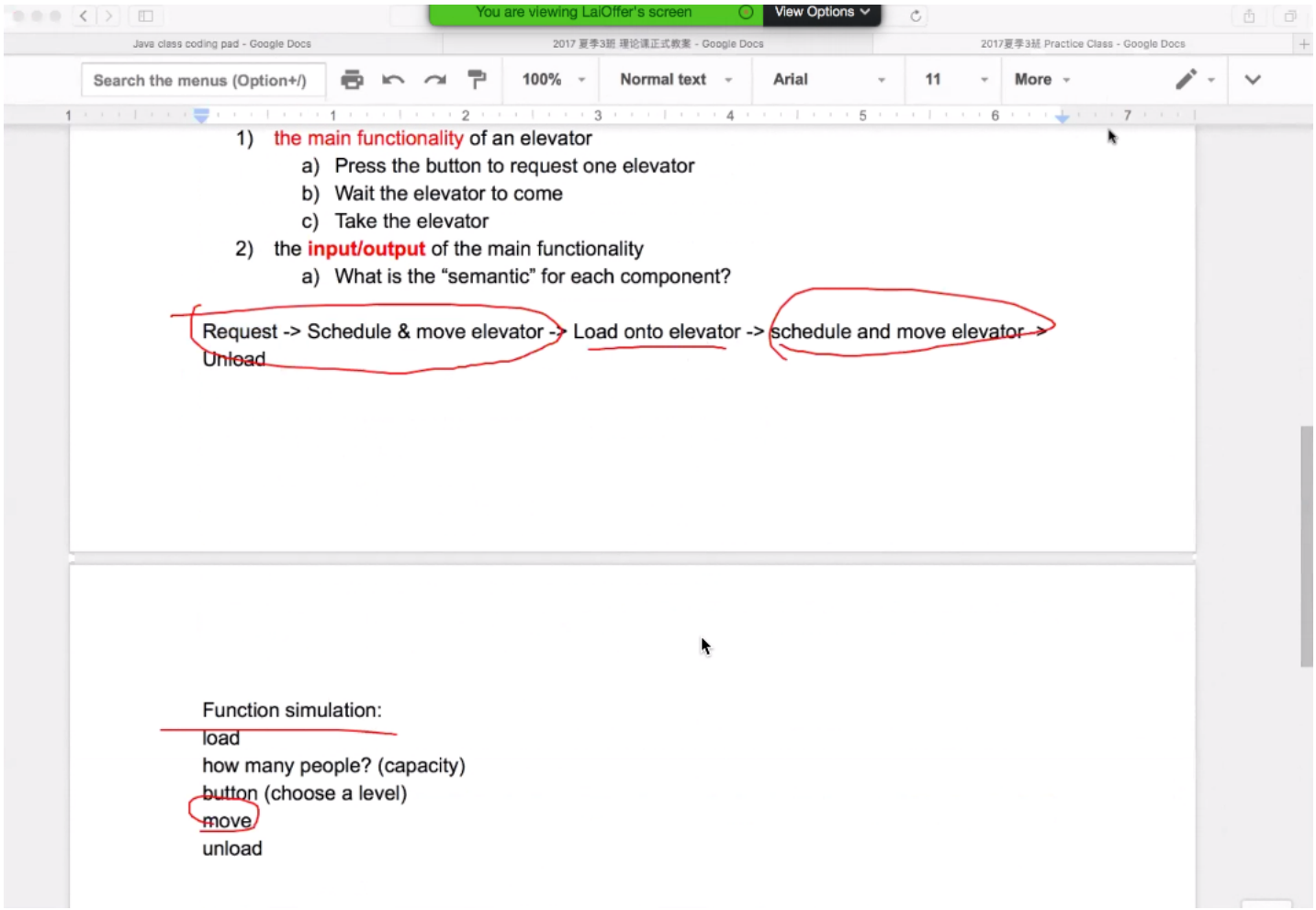

docs.google.com
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
========== Zoom ==========
Please join my meeting from your computer, tablet or smartphone.
请使用正式课件的zoom信息
========== 答疑 ==========
piazza.com/laioffer/spring2017/laioffer001
Access Code: laioffer001
==========
上过的课程在这里：
存档
Class 28: OOD Java
Elevator simulator
Classical OOD question: Design an elevator simulation system.


You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
请使用正式课件的zoom信息
========== 答疑 ==========
piazza.com/laioffer/spring2017/laioffer001
Access Code: laioffer001
==========
上过的课程在这里:
存档
Class 28: OOD Java
Elevator simulator
Classical OOD question: Design an elevator simulation system.
Function simulation:
load
how many people? (capacity)
button (choose a level)
move
unload

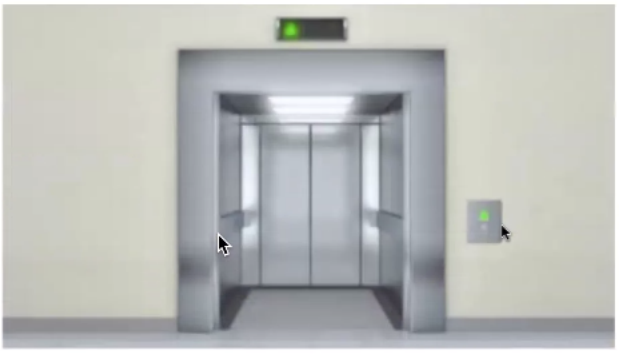

You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
move
unload
End-to-end workflow
How to analyze the use case?
Two important questions:
1) the main functionality of an elevator
a) Press the button to request one elevator
b) Wait the elevator to come
c) Take the elevator
2) the input/output of the main functionality
a) What is the "semantic" for each component?
Request -> Schedule & move elevator -> Load onto elevator -> schedule and move elevator -> Unload


You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
1) the main functionality of an elevator
a) Press the button to request one elevator
b) Wait the elevator to come
c) Take the elevator
2) the input/output of the main functionality
a) What is the "semantic" for each component?
Request -> Schedule & move elevator -> Load onto elevator -> schedule and move elevator -> Unload
Function simulation:
load
how many people? (capacity)
button (choose a level)
move
unload

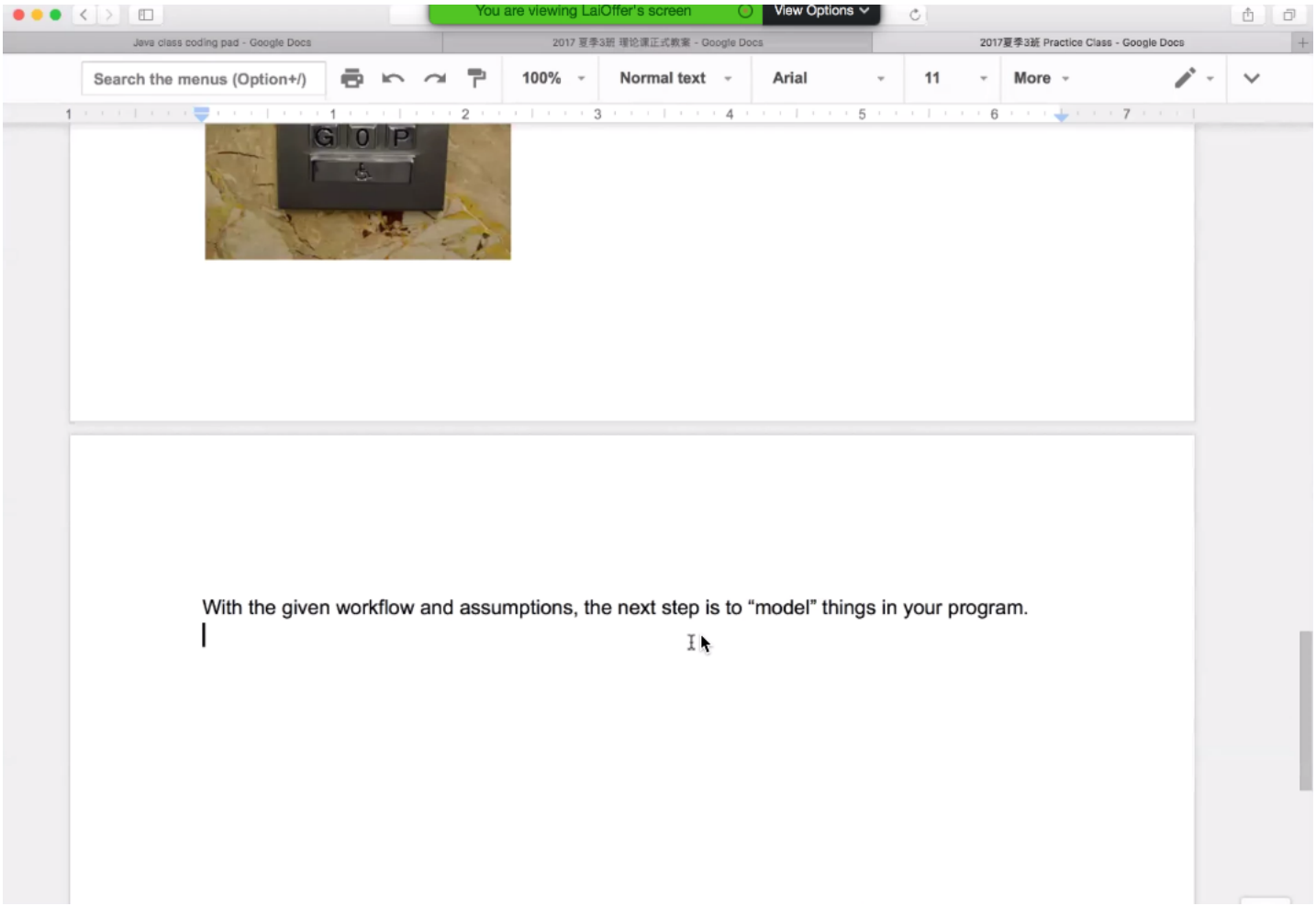
Request -> Schedule & move elevator -> Load onto elevator -> schedule and move elevator -> Unload

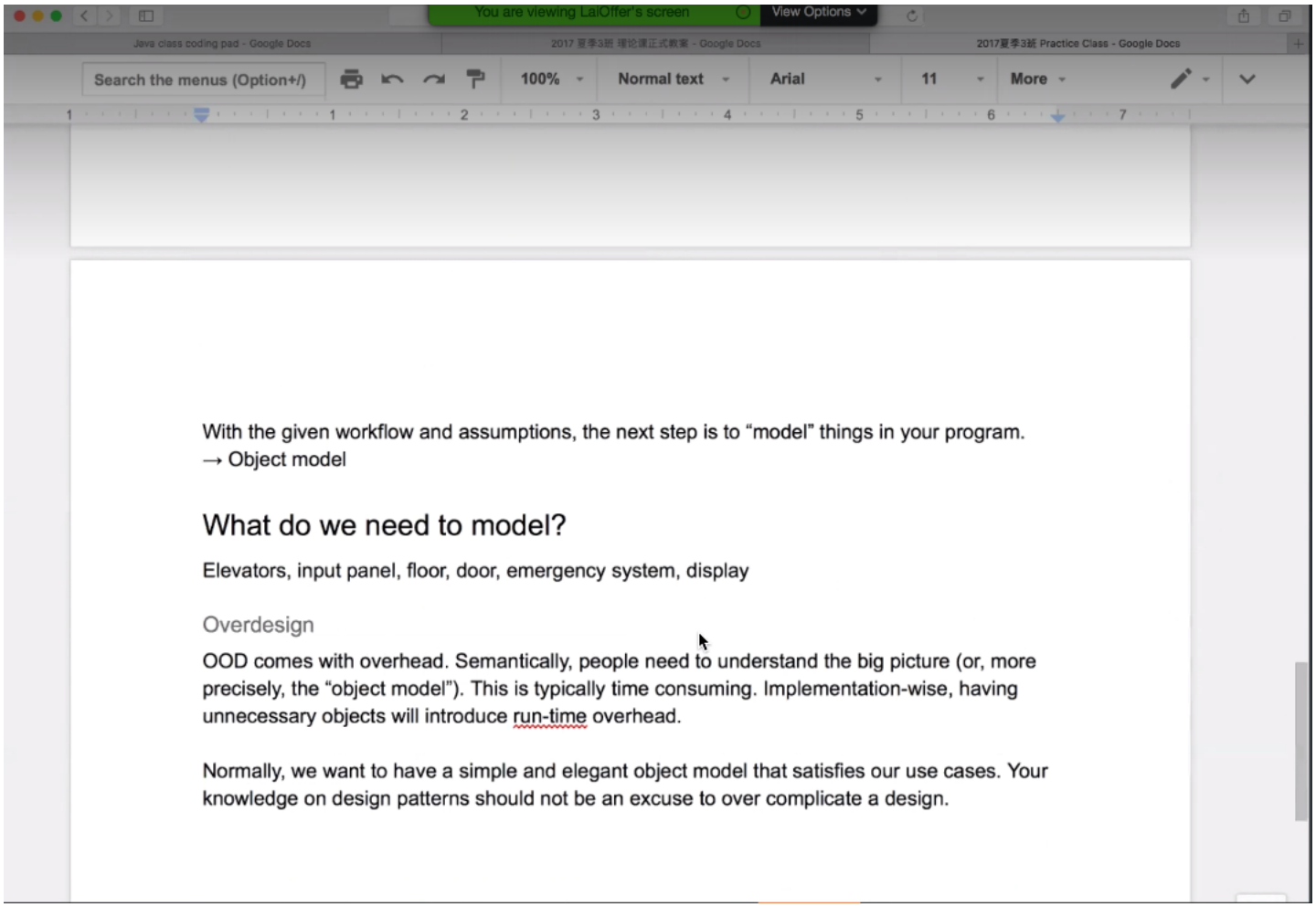
There are different types of elevators in real life. These elevators may have different input patterns (i.e., buttons) or different schedulers. How to deal with this situation?

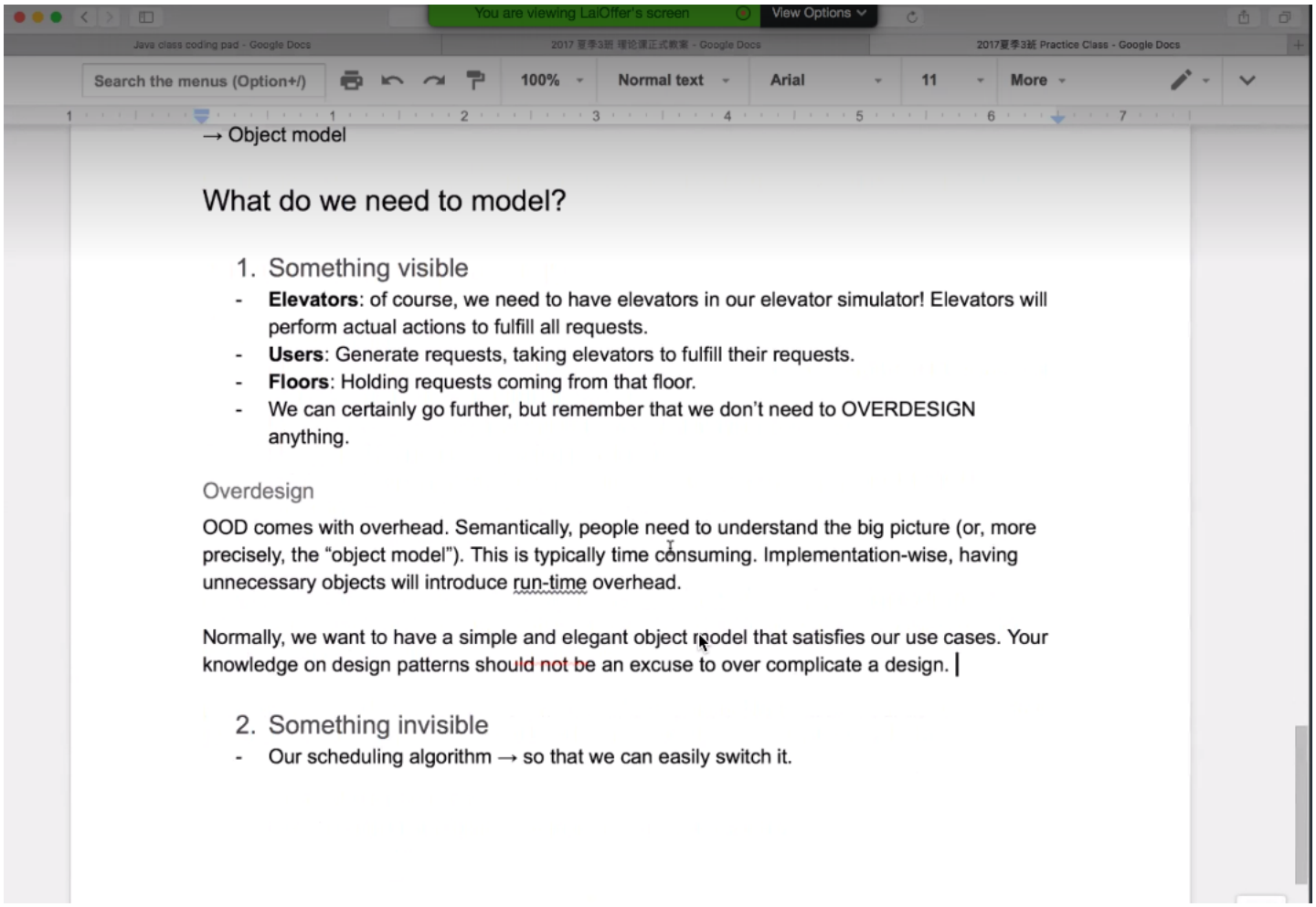
- Discuss use cases and make (reasonable) assumptions
- Provide flexibility for different implementations

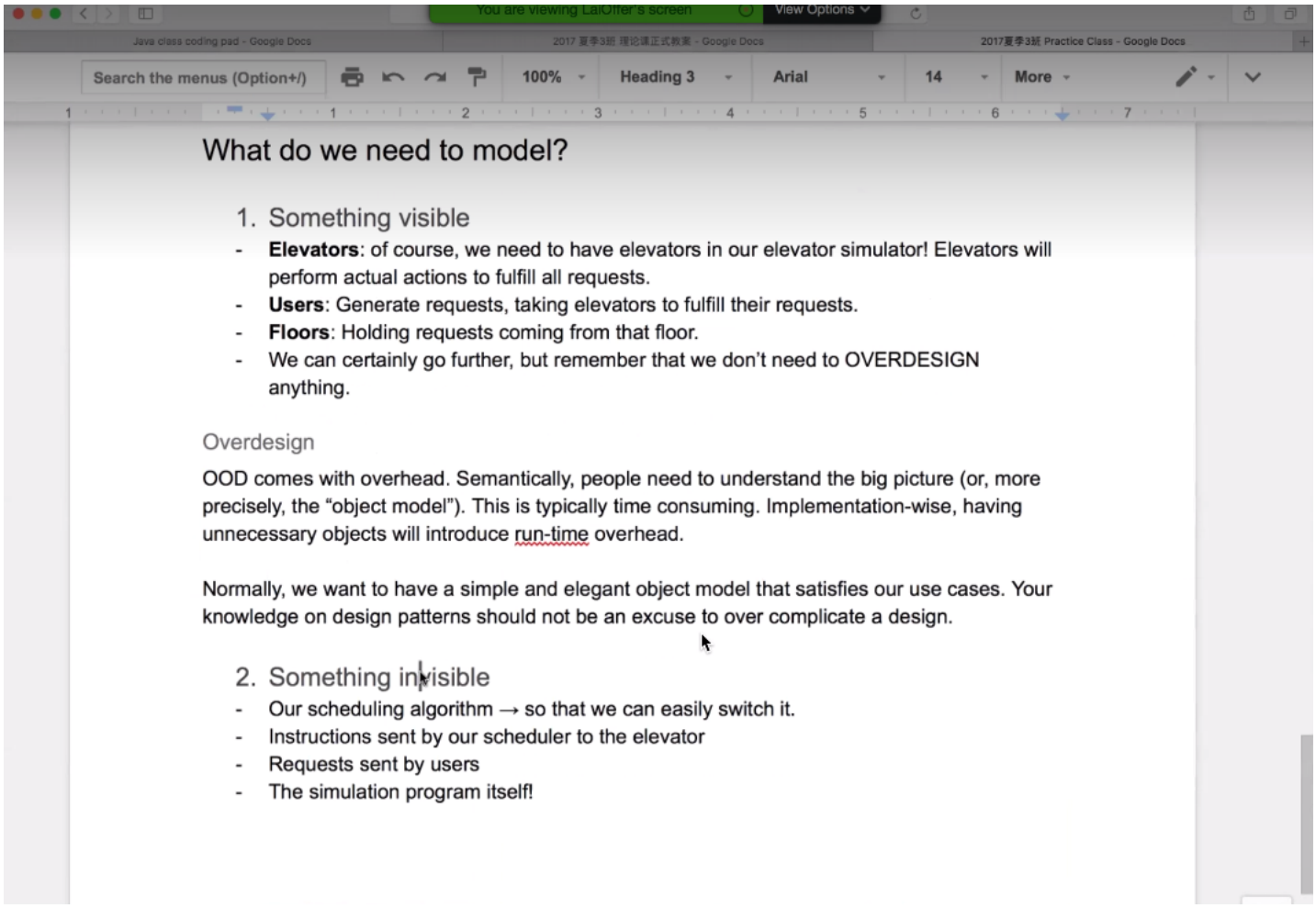


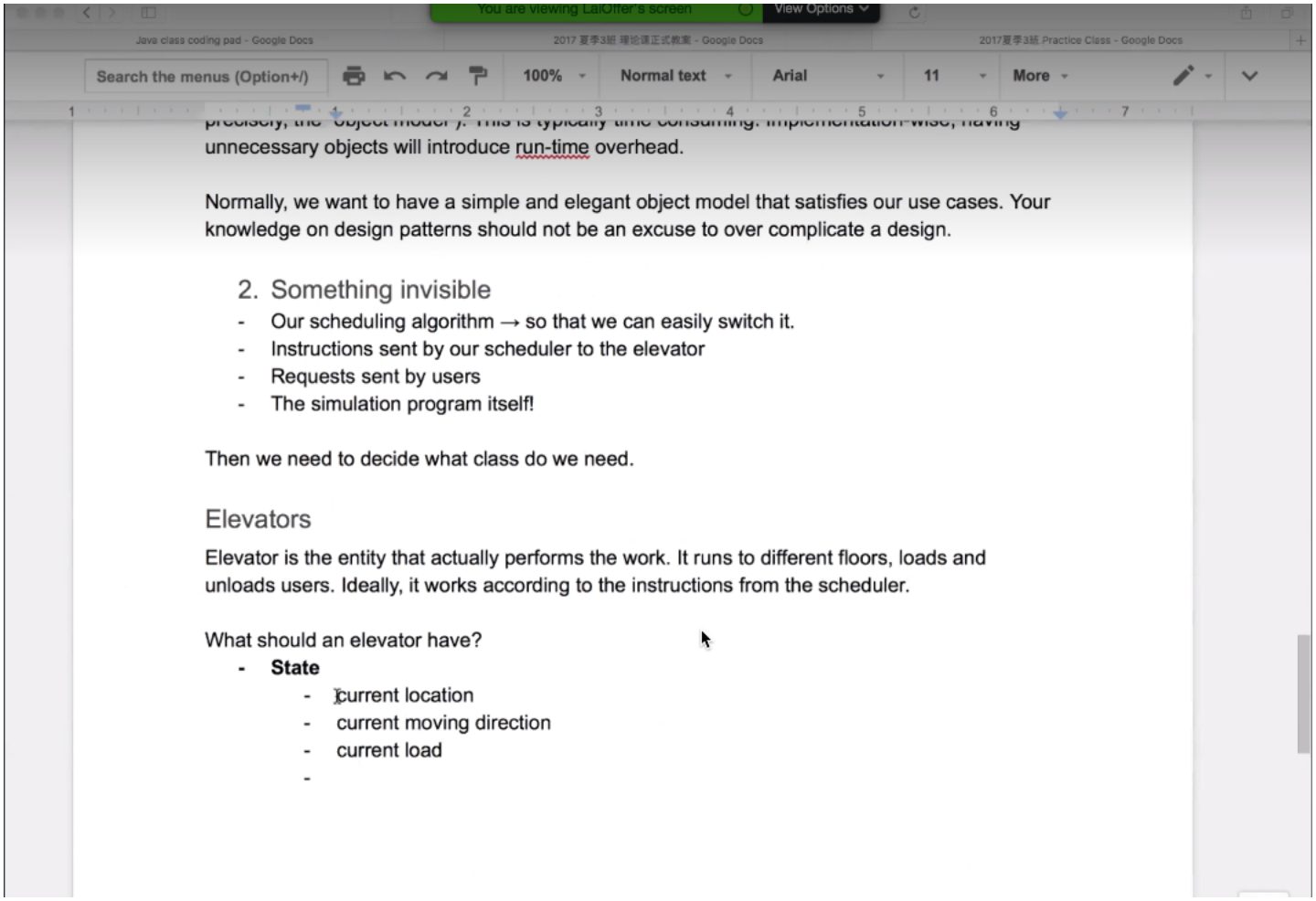
----------OR----------

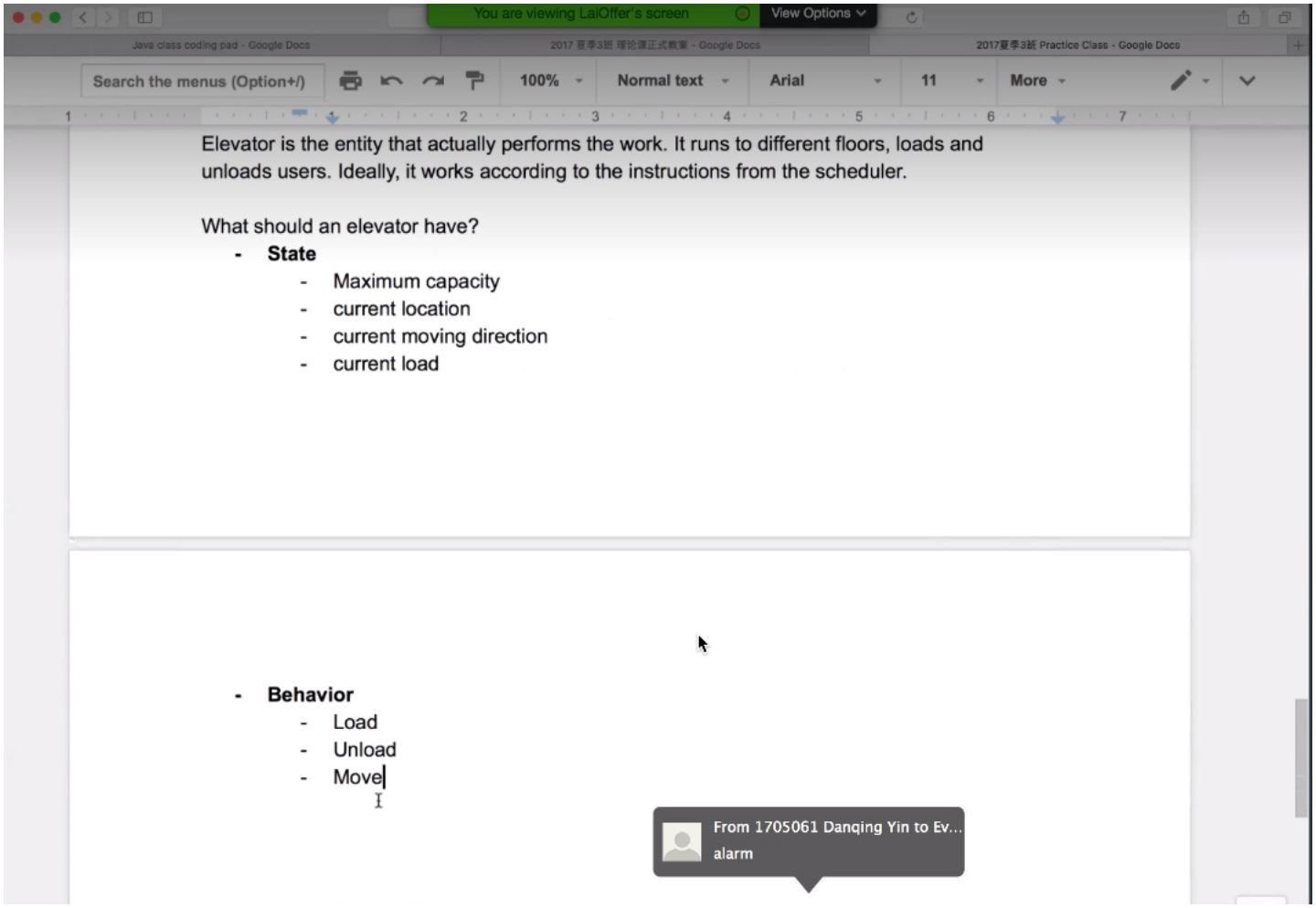

You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
G O P
With the given workflow and assumptions, the next step is to “model” things in your program.

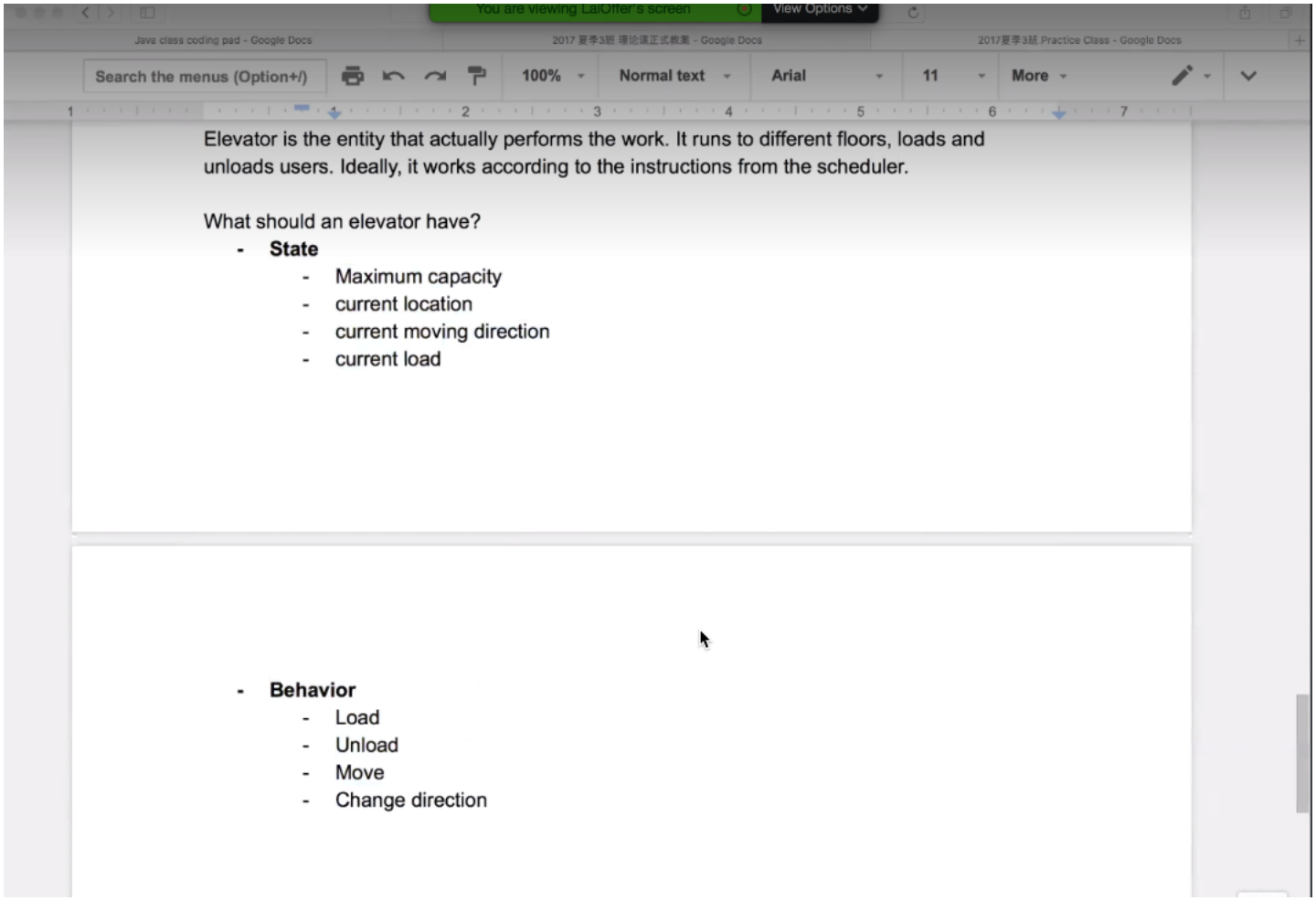

You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
With the given workflow and assumptions, the next step is to “model” things in your program.
→ Object model
What do we need to model?
Elevators, input panel, floor, door, emergency system, display
Overdesign
OOD comes with overhead. Semantically, people need to understand the big picture (or, more precisely, the “object model”). This is typically time consuming. Implementation-wise, having unnecessary objects will introduce run-time overhead.
Normally, we want to have a simple and elegant object model that satisfies our use cases. Your knowledge on design patterns should not be an excuse to over complicate a design.

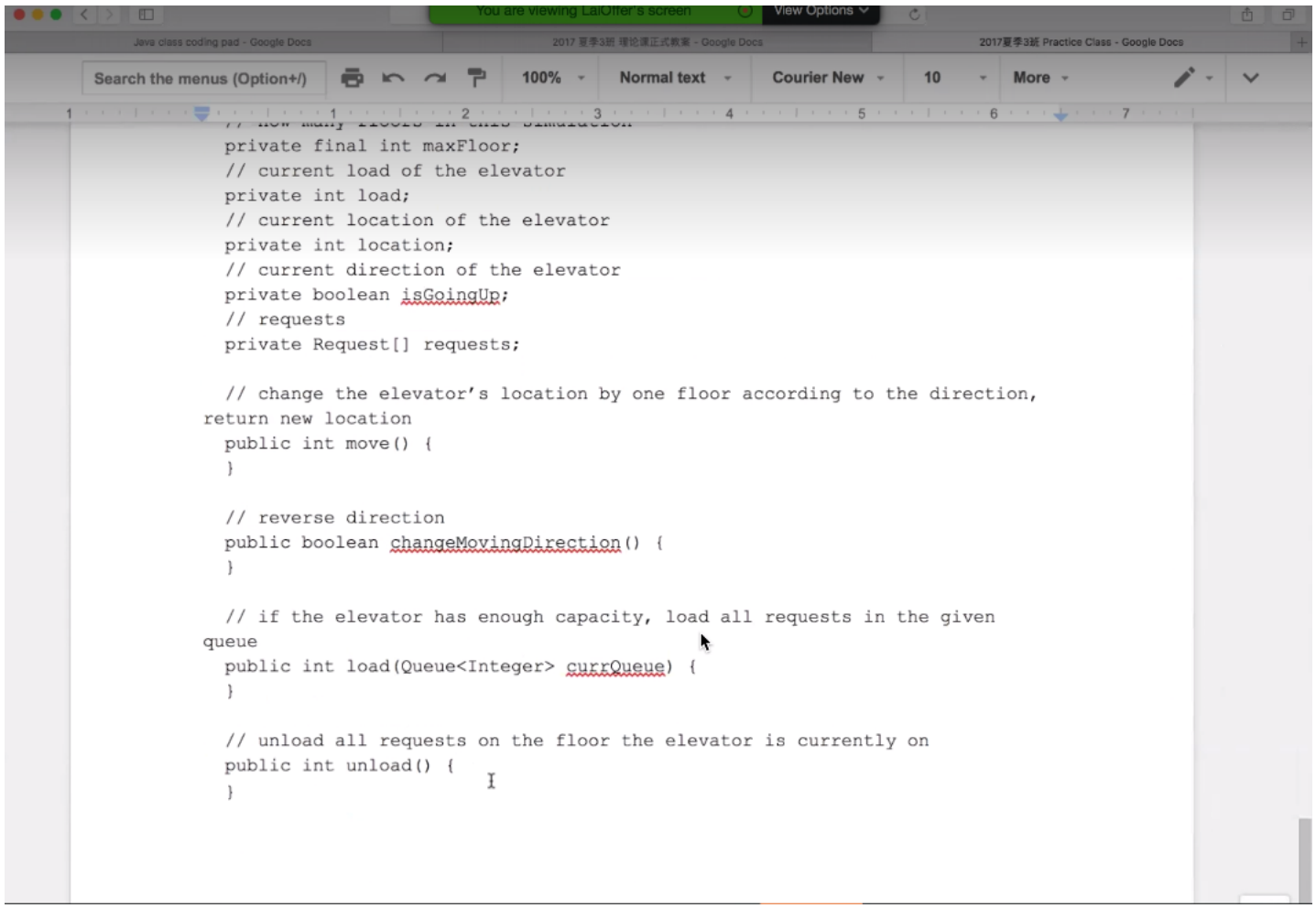

You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
→ Object model
What do we need to model?
1. Something visible
- Elevators: of course, we need to have elevators in our elevator simulator! Elevators will perform actual actions to fulfill all requests.
- Users: Generate requests, taking elevators to fulfill their requests.
- Floors: Holding requests coming from that floor.
- We can certainly go further, but remember that we don't need to OVERDESIGN anything.
Overdesign
OOD comes with overhead. Semantically, people need to understand the big picture (or, more precisely, the "object model"). This is typically time consuming. Implementation-wise, having unnecessary objects will introduce run-time overhead.
Normally, we want to have a simple and elegant object model that satisfies our use cases. Your knowledge on design patterns should not be an excuse to over complicate a design.
2. Something invisible
- Our scheduling algorithm → so that we can easily switch it.

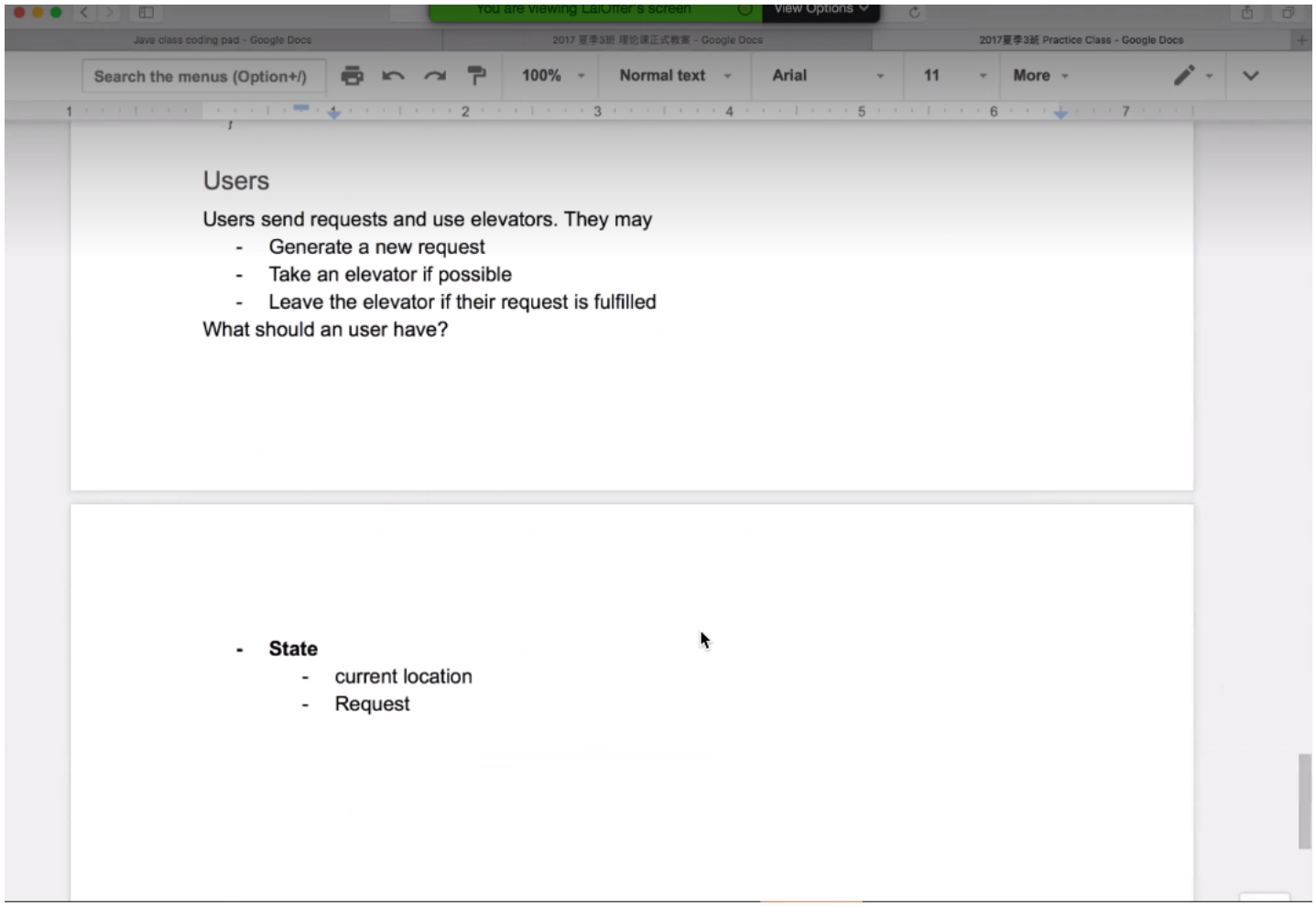

You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Heading 3
Arial
14
More
What do we need to model?
1. Something visible
- Elevators: of course, we need to have elevators in our elevator simulator! Elevators will perform actual actions to fulfill all requests.
- Users: Generate requests, taking elevators to fulfill their requests.
- Floors: Holding requests coming from that floor.
- We can certainly go further, but remember that we don't need to OVERDESIGN anything.
Overdesign
OOD comes with overhead. Semantically, people need to understand the big picture (or, more precisely, the "object model"). This is typically time consuming. Implementation-wise, having unnecessary objects will introduce run-time overhead.
Normally, we want to have a simple and elegant object model that satisfies our use cases. Your knowledge on design patterns should not be an excuse to over complicate a design.
2. Something invisible
- Our scheduling algorithm → so that we can easily switch it.
- Instructions sent by our scheduler to the elevator
- Requests sent by users
- The simulation program itself!

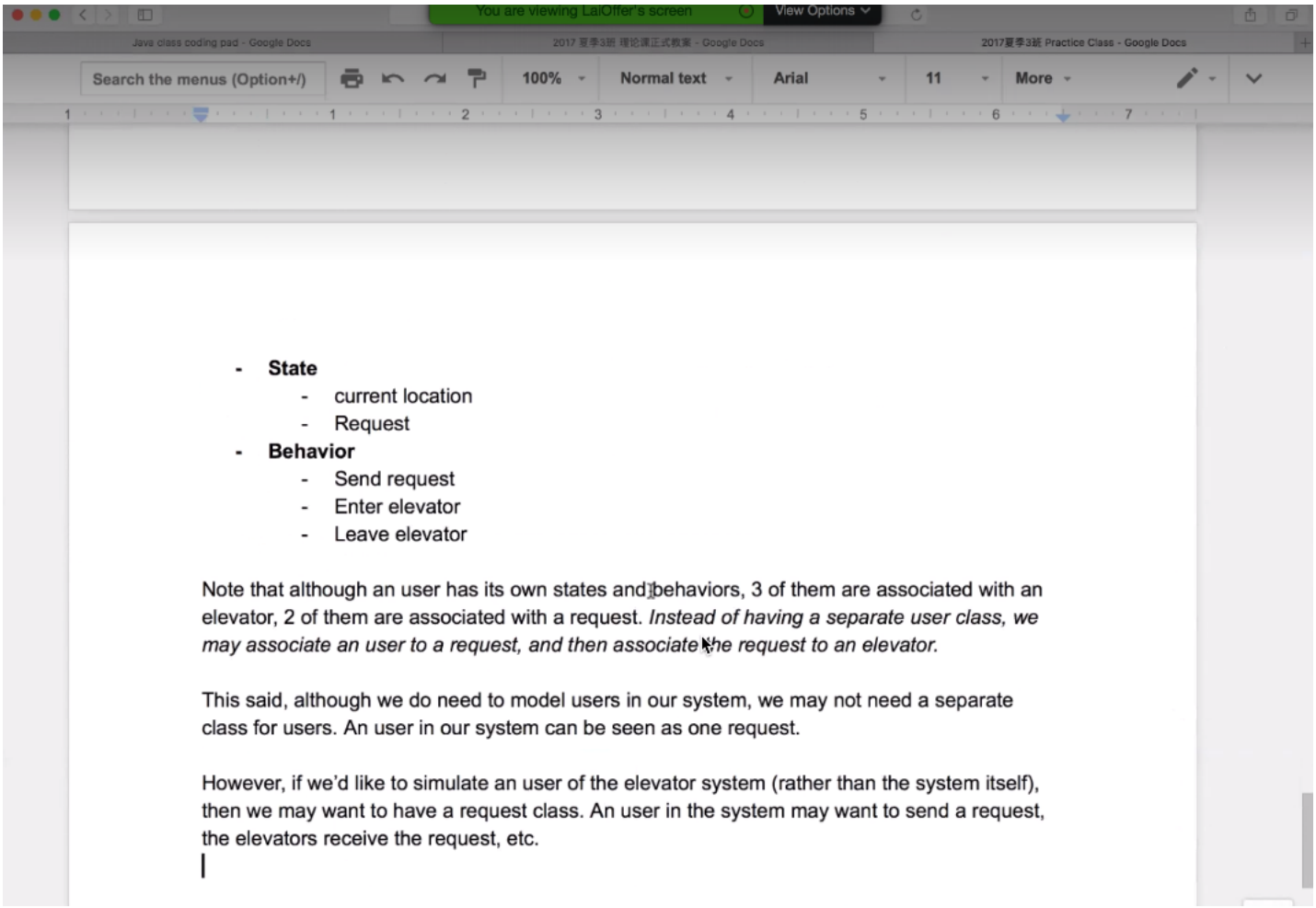

You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
unnecessary objects will introduce run-time overhead.
Normally, we want to have a simple and elegant object model that satisfies our use cases. Your knowledge on design patterns should not be an excuse to over complicate a design.
2. Something invisible
- Our scheduling algorithm → so that we can easily switch it.
- Instructions sent by our scheduler to the elevator
- Requests sent by users
- The simulation program itself!
Then we need to decide what class do we need.
Elevators
Elevator is the entity that actually performs the work. It runs to different floors, loads and unloads users. Ideally, it works according to the instructions from the scheduler.
What should an elevator have?
- State
  - current location
  - current moving direction
  - current load
  -

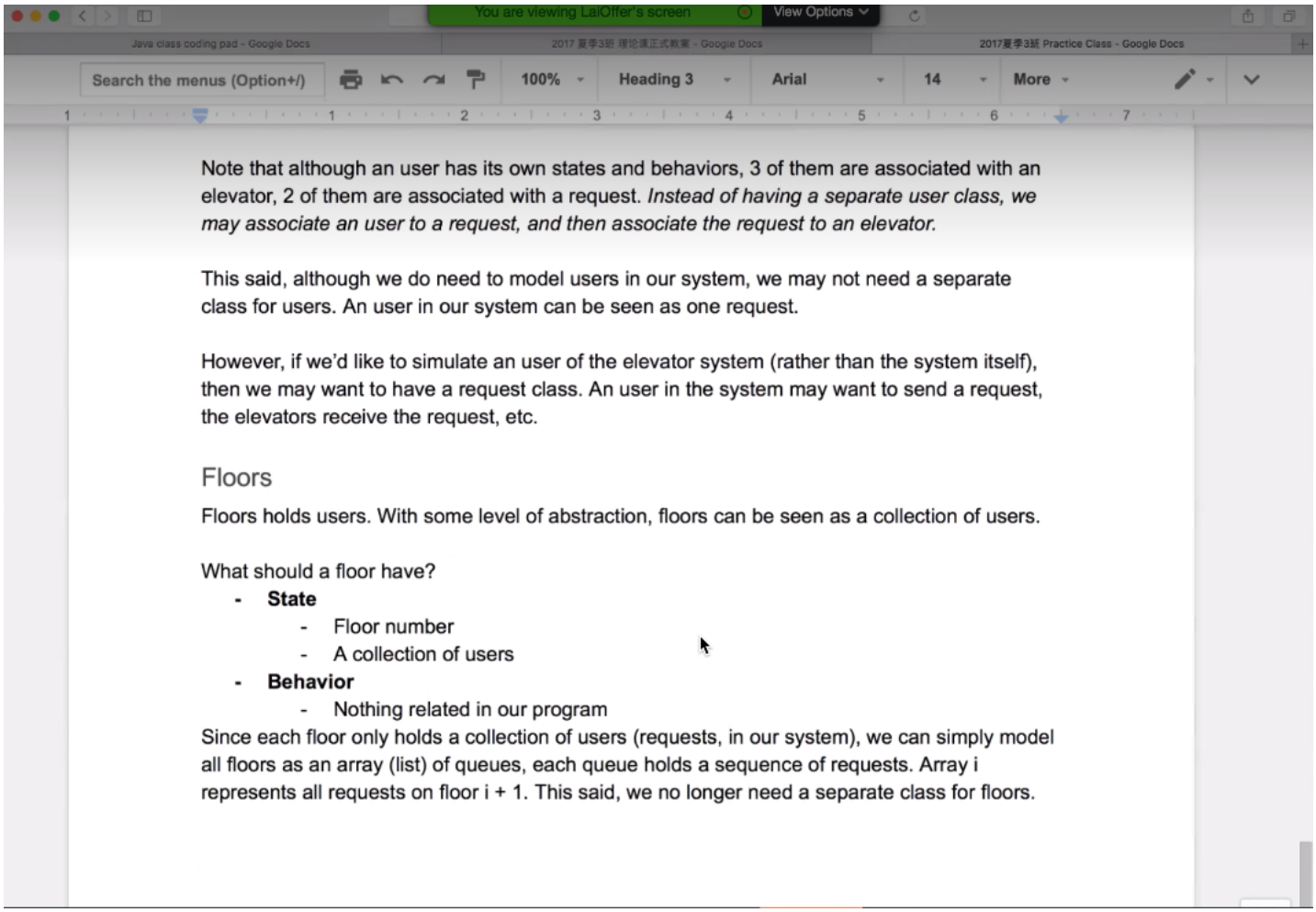

You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
Elevator is the entity that actually performs the work. It runs to different floors, loads and unloads users. Ideally, it works according to the instructions from the scheduler.
What should an elevator have?
- State
- Maximum capacity
- current location
- current moving direction
- current load
- Behavior
- Load
- Unload
- Move
From 1705061 Danqing Yin to Ev...
alarm


You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
Elevator is the entity that actually performs the work. It runs to different floors, loads and unloads users. Ideally, it works according to the instructions from the scheduler.
What should an elevator have?
- State
  - Maximum capacity
  - current location
  - current moving direction
  - current load
- Behavior
  - Load
  - Unload
  - Move
  - Change direction


You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Courier New
10
More
private final int maxFloor;
// current load of the elevator
private int load;
// current location of the elevator
private int location;
// current direction of the elevator
private boolean isGoingUp;
// requests
private Request[] requests;
// change the elevator's location by one floor according to the direction, return new location
public int move() {
}
// reverse direction
public boolean changeMovingDirection() {
}
// if the elevator has enough capacity, load all requests in the given queue
public int load(Queue<Integer> currQueue) {
}
// unload all requests on the floor the elevator is currently on
public int unload() {
}


Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
Users
Users send requests and use elevators. They may
- Generate a new request
- Take an elevator if possible
- Leave the elevator if their request is fulfilled
What should an user have?
- State
- current location
- Request


You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
- **State**
  - current location
  - Request
- **Behavior**
  - Send request
  - Enter elevator
  - Leave elevator
Note that although an user has its own states and behaviors, 3 of them are associated with an elevator, 2 of them are associated with a request. *Instead of having a separate user class, we may associate an user to a request, and then associate the request to an elevator.*
This said, although we do need to model users in our system, we may not need a separate class for users. An user in our system can be seen as one request.
However, if we'd like to simulate an user of the elevator system (rather than the system itself), then we may want to have a request class. An user in the system may want to send a request, the elevators receive the request, etc.



Note that although an user has its own states and behaviors, 3 of them are associated with an elevator, 2 of them are associated with a request. *Instead of having a separate user class, we may associate an user to a request, and then associate the request to an elevator.*

This said, although we do need to model users in our system, we may not need a separate class for users. An user in our system can be seen as one request.

However, if we'd like to simulate an user of the elevator system (rather than the system itself), then we may want to have a request class. An user in the system may want to send a request, the elevators receive the request, etc.

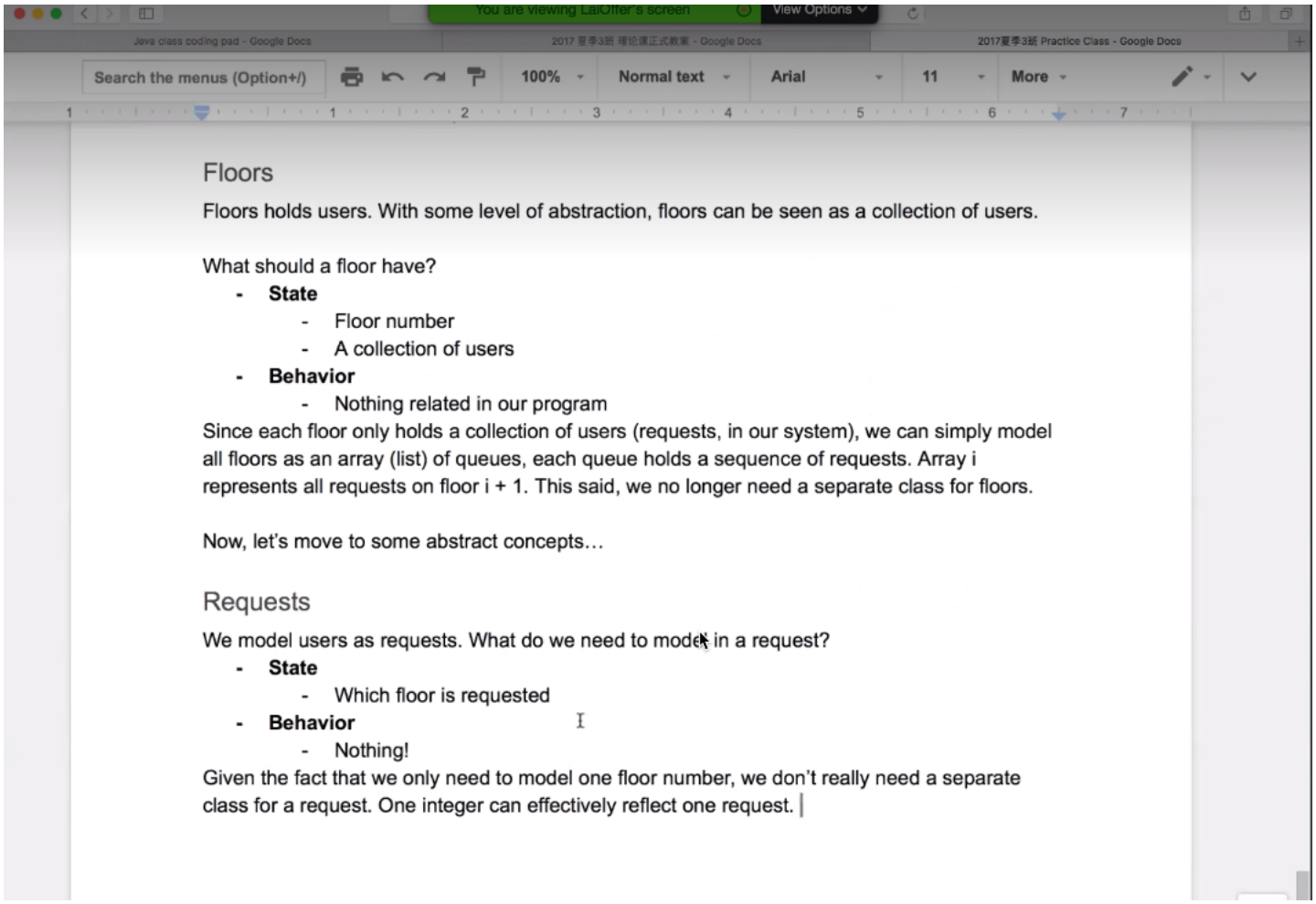
Floors

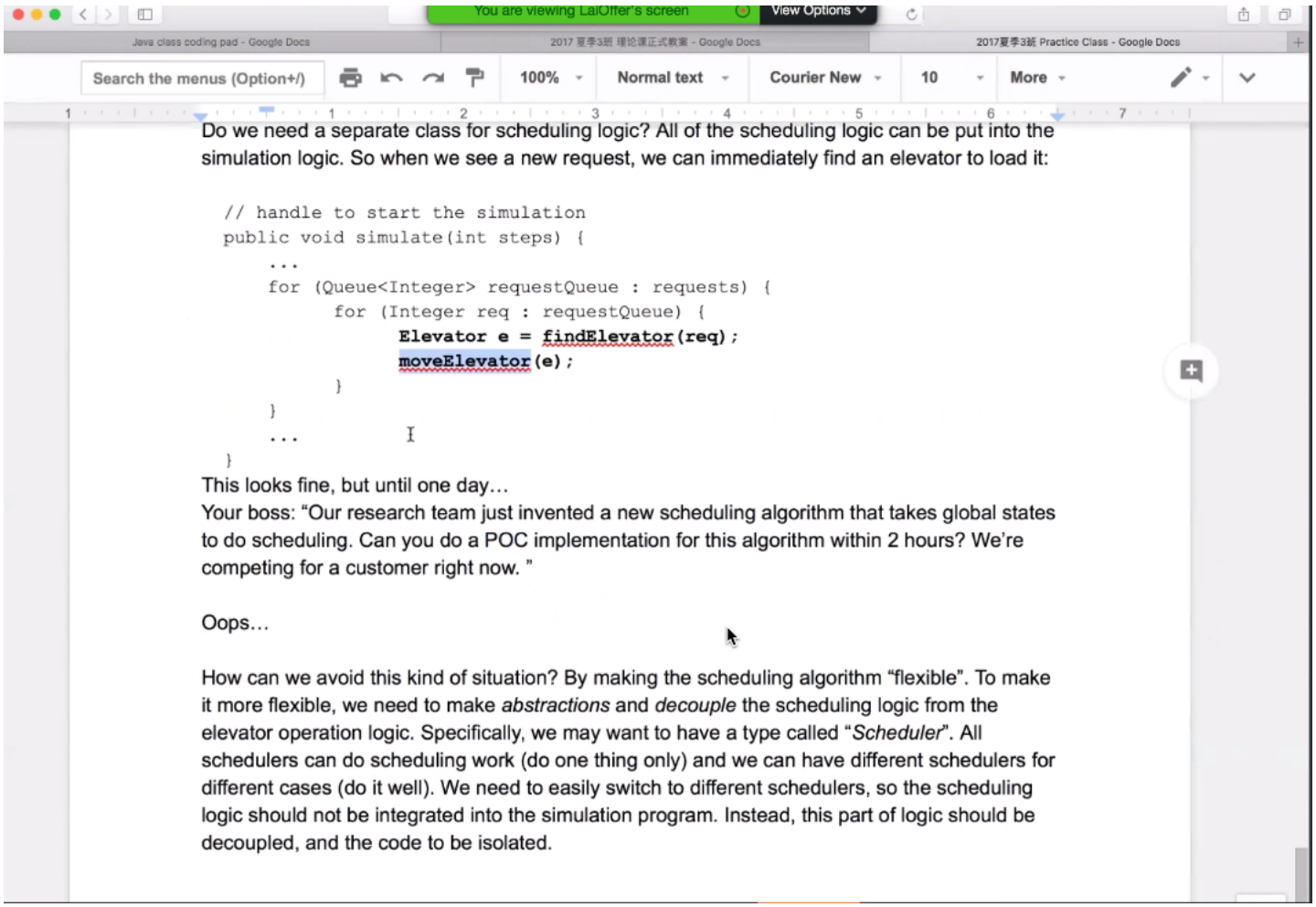
Floors holds users. With some level of abstraction, floors can be seen as a collection of users.

What should a floor have?

- **State**
  - Floor number
  - A collection of users
- **Behavior**
  - Nothing related in our program

Since each floor only holds a collection of users (requests, in our system), we can simply model all floors as an array (list) of queues, each queue holds a sequence of requests. Array i represents all requests on floor i + 1. This said, we no longer need a separate class for floors.


You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Arial
11
More
Floors
Floors holds users. With some level of abstraction, floors can be seen as a collection of users.
What should a floor have?
- State
  - Floor number
  - A collection of users
- Behavior
  - Nothing related in our program
Since each floor only holds a collection of users (requests, in our system), we can simply model all floors as an array (list) of queues, each queue holds a sequence of requests. Array i represents all requests on floor i + 1. This said, we no longer need a separate class for floors.
Now, let's move to some abstract concepts…
Requests
We model users as requests. What do we need to model in a request?
- State
  - Which floor is requested
- Behavior
  - Nothing!
Given the fact that we only need to model one floor number, we don't really need a separate class for a request. One integer can effectively reflect one request.


You are viewing LaiOffer's screen
View Options
Java class coding pad - Google Docs
2017 夏季3班 理论课正式教案 - Google Docs
2017夏季3班 Practice Class - Google Docs
Search the menus (Option+/)
100%
Normal text
Courier New
10
More
Do we need a separate class for scheduling logic? All of the scheduling logic can be put into the simulation logic. So when we see a new request, we can immediately find an elevator to load it:
// handle to start the simulation
public void simulate(int steps) {
...
for (Queue<Integer> requestQueue : requests) {
for (Integer req : requestQueue) {
Elevator e = findElevator(req);
moveElevator(e);
}
}
...
}
This looks fine, but until one day…
Your boss: "Our research team just invented a new scheduling algorithm that takes global states to do scheduling. Can you do a POC implementation for this algorithm within 2 hours? We're competing for a customer right now. "
Oops…
How can we avoid this kind of situation? By making the scheduling algorithm "flexible". To make it more flexible, we need to make *abstractions* and *decouple* the scheduling logic from the elevator operation logic. Specifically, we may want to have a type called "*Scheduler*". All schedulers can do scheduling work (do one thing only) and we can have different schedulers for different cases (do it well). We need to easily switch to different schedulers, so the scheduling logic should not be integrated into the simulation program. Instead, this part of logic should be decoupled, and the code to be isolated.