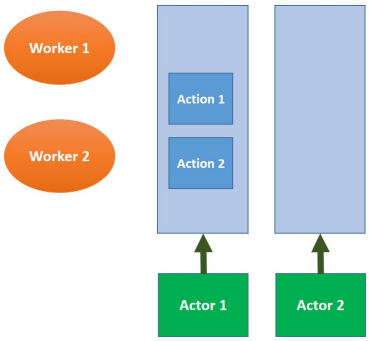
In the following assignment you are required to implement an Actor Model, and to use it for implementing a University Management System.

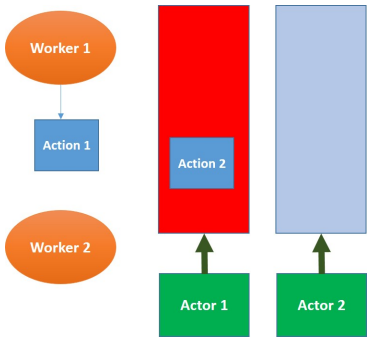
1 Part 1: Actor Thread Pool

1.1 Detailed Description

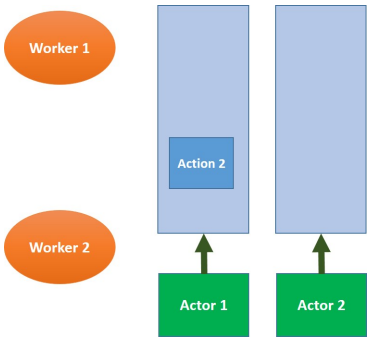
In actor thread pool, each actor has a queue of actions. One can submit a new action to the actor. The threads in the actor thread pool are assigned dynamically to the actors (the amount of actors can be significantly greater than the amount of threads), in the following way. Each thread searches for an action to execute in all actors' queues. Once the thread found such an action, it will prevent any other thread from fetching actions from that queue. Once it finished executing the action, it will allow other threads to fetch actions from this queue. Note, although a thread prevents other threads from processing actions from the queue which it is executing an action from it, the other threads are not blocked. The threads have to search for an action from other queues. Only if all queues are empty or not available (threads work on them), the threads will go sleep and should wake up once an action from an available queue is ready to be fetched.
See figure 1.



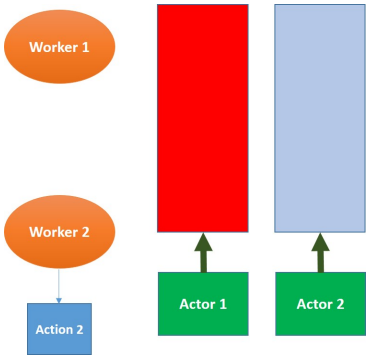
(a) Two Actors and Two Threads (Threads), actor 1 has two actions in his queue



(b) Thread 1 fetches action 1 from actor 1 and executes it. Thread 2 can not fetch actions from actor 1 queue



(c) Action 1 has been completed. Therefore, the queue of actor 1 is available for both threads



(d) Thread 2 fetches action 2 from actor 1 and executes it

Figure 1: Actor Thread Pool Description

The threads in the Thread Pool apply *event loop*. In each iteration of the loop, each thread tries to fetch an action and execute it. An important point in such design pattern is not to block the thread for a long time, unless there is no work for it. Blocking the thread for a long time while there is a work for it will have a bad effect in your implementation, since threads are idle although there is work for them. See figure 2.

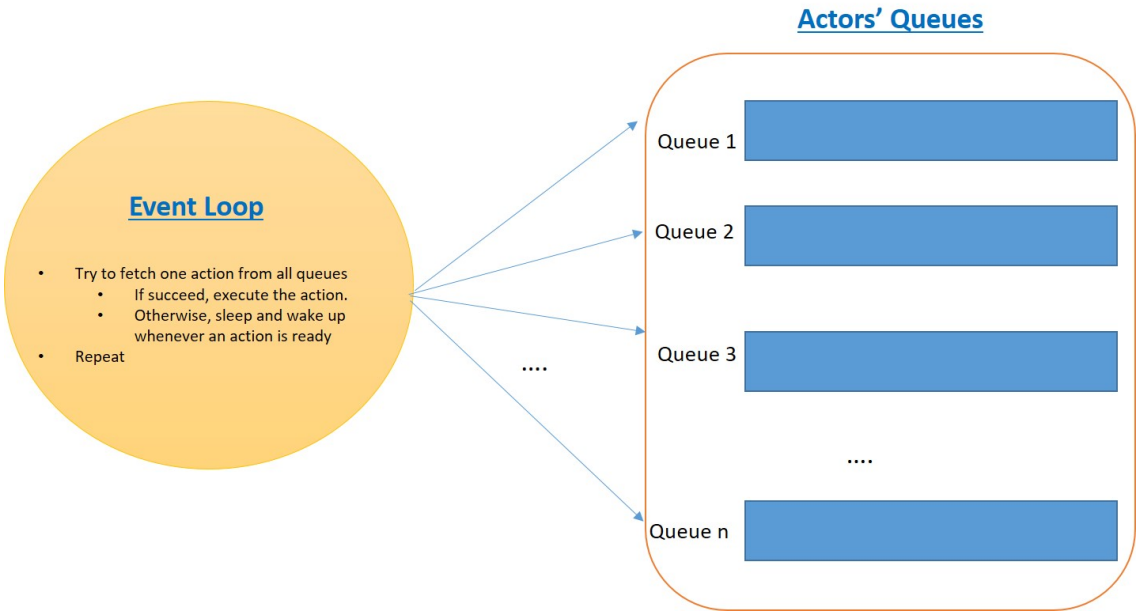


Figure 2: Event Loop

1.2 Actors and Actions

An *action* is a computational task. An *actor* is a computational entity which responses to actions it receives. An actor can: make local decisions, create more actors, send message to other actors.
Note: Only one thread can access the private state of an actor at the same time, so you shouldn't synchronize the access to the private state of the actor.

1.3 Dependency Between Actions

In some applications there might be dependencies between actions. The thread pool should fulfill these constraints, *i.e.,* the execution of an action should be suspended until the actions it depends on are completed. Suspending an action should not suspend the thread or the actor. When an action is suspended, the thread should continue with another action, and the actor's queue should be available for fetching actions by any thread. The approach to handle this is to store the continuation of the suspended action on the actor's queue, and to resume it whenever its dependencies are fulfilled by the thread which will fetch it.

1.3.1 Promise Design Pattern

In order to enable the mentioned above. We use the promise design pattern. A Promise is used for deferred computations, it represents the result of an operation that has not been completed yet. Each promise contains a set of callback functions to be applied after its value is resolved.

1.4 Example of Actor Thread Pool

In order to explain how Actor Thread Pool works, assume that we want to write a program to manage banks. Each bank has a list of customers, where customers can transfer money to customers in different accounts.
In actor thread pool, we consider each bank as an actor. We submit actions for the banks. The threads in the thread pool will search for an available action in one of the banks' queues and execute it while preventing other threads from executing an action of the same bank (which ensures that actions of the same bank are executed in the order the were received).
Let us consider the action of transferring money from a customer A in bank 1 to customer B in bank 2. This action involves the authorization of both banks, updating the customers records and a receipt for both customers. The transformation can be applied according to the following protocol:

- Bank 1 fetch a money-transformation action (from bank 1 to bank 2) from its queue.
- Bank 1 sends a confirmation-action to bank 2.
- Bank 1 indicates that the transformation-action can be completed only after the confirmation-action is resolved.
- Bank 1 continue his event loop, handling other actions in its queue.
- Bank 2 fetch the confirmation action sent from bank 1, handle and complete it by resolving the promise of this action with a value. As a result, the callbacks defined for this promise are called one by one.
- Since the dependencies of the money-transformation action (= the confirmation action) are now resolved, Bank 2 re-insert the money-tranformation action to the queue of bank 1.
- Bank 1 fetch the money-transformation action from its queue, and complete it according to the resulted value of the confirmation-action.

1.5 Implementation Of Actor Thread Pool

You are supplied with 4 classes you should implement in order to construct the actor thread pool. These classes are defined in bgu.atd.a1 package. Read the javadoc of the supplied interfaces as a complementary material to this section. The rest of this section describes the given classes and the way they should be implemented.

1.5.1 The Actor Thread Pool

An Action contains a Promise representing its current value, and a Callback function representing the continuation which should be applied on the resulted value.

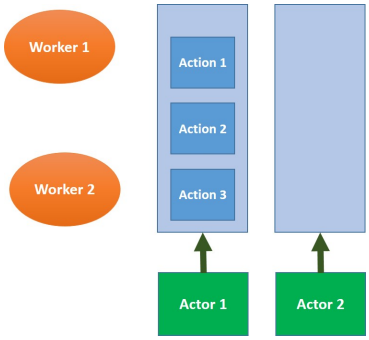
An Actor has an *action queue*, a string *id*, and a *PrivateState* object. The Actor Thread Pool holds all queues of all actors in the system. The *submit* method of the Actor Thread Pool gets an action (or a list of actions) along with corresponding actor. If the actor already defined in the thread pool, the action is added to its queue. Otherwise, the action queue for this actor is created and the action is added to this queue.
The actor thread pool maintains a set of threads. The assignment of actors to threads is done dynamically, each thread searches for an action in all actors' queues. Once the thread found an action, it prevents any other thread from executing tasks of the same actor.

The flow is as follows: a thread fetches an action from a queue. The thread calls the *handle* method of the action, which calls the *start* method of the action in case its promise value is not resolved, or apply callback continuation in case the promise value is resolved.
As mentioned, a given actor may need the help of other actors in order to continue its current action. In this case, it will define these actions as dependencies, by calling the *then* method (of the Action class) with the continuation which should be applied after these actions are completed. Then, the actions will be sent to other actors by *sendMessage* method. After the actor has sent all messages, it proceeds with other tasks. When the sent actions are completed, the suspended action will be re-inserted to the actor's queue. At some point, the actor will fetch the re-inserted suspended action and handle it by executing its continuation (since the promise value is now resolved).

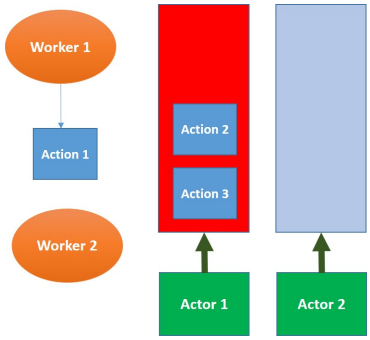
1.5.2 Summary and Additional Clarifications

The following is a summary and additional clarifications about the implementation of different parts of the framework.

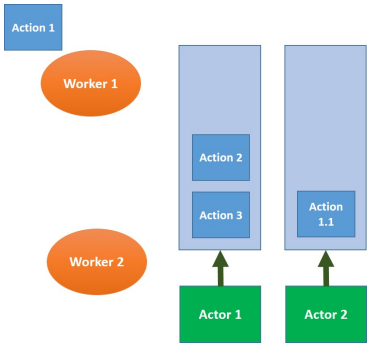
Important: It is mandatory to read the all javadocs of the interfaces carefully, as these provides necessary hints of the implementation. You can add methods and fields to the classes. But you are not allowed to remove or change any method signature that we provided you with. Otherwise you will fail our automatic tests. Notice that you still can add the "synchronized" keyword to some methods if needed. However, remember that in concurrent programming, you should try to find good ways to avoid blocking threads as much as possible.



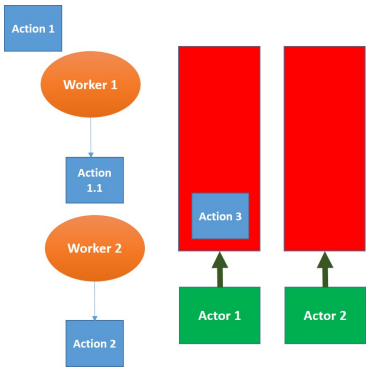
(a) Two actors, and two Threads. Actor 1 has three actions



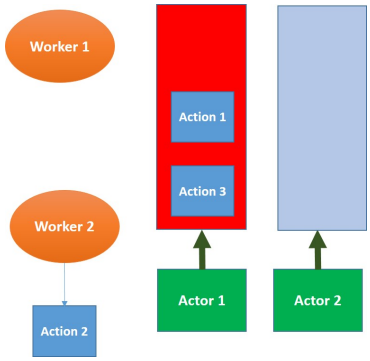
(b) Thread 1 fetches Action 1 from actor 1 and executes it. Thread 2 is prevented from fetching action from actor 1



(c) Action 1 sent a message Action 1.1 to Actor 2, and waits for its result by Action 1.1. The Threads can fetch actions from both actors.



(d) Thread 1 fetches Action 1.1 from Actor 2, and Thread 2 fetches action 2 from Actor 1



(e) Thread 1 completes Action 1.1. The Promise object of Action 1.1 is resolved, and Action 1 re-inserted to actor's 1 queue. As a result, the continuation of action 1 will be applied later by the thread which will fetch Action 1 later

Figure 3: Task "life cycle" - a sample of class interaction within the Actor Thread Pool .

- **Action**: an abstract class that represents an action. An action is an object which holds the required information to handle an action in the system. The action also holds a Promise<R> object which will hold its result, and a callback to be applied after the dependencies (in case they were defined) are fulfilled. The main Action methods are:
  - start: This is the abstract method, that should be implemented for each action type, it implements the action behavior.
  - sendMessage: This methods submit an action (message) to other actor.
  - then: add a callback to be executed once all given actions (the dependencies) are completed.
  - complete: resolves the internal result - should be called by the action derivative once it is done.
  - getResult: returns the task promised result.
- **Promise**: Represents a promise result. Promise includes these methods:
  - get: return the resolved value if such exists.
  - resolve: called upon completing the operation, it sets the result of the operation to a new value, and trigger all subscribed callbacks.
  - subscribe: add a callback to be called when this object is resolved. If the promise is already resolved - the callback should be called immediately.

- isResolved: return true if this object has been resolved.

- **ActorThreadPool**: manages the actors and threads in the system. The constructor creates an ActorThreadPool which has n threads. The ActorThreadPool includes these methods:
  - submit: Adds an action to an actor's queue. If the actor is not present in the system, it will create it.
  - start: start the threads belongs to this thread pool.
  - shutdown: closes the thread pool - this method interrupts all threads and wait for them to stop - it returns only when there are no live threads in the queue. After calling this method, one should not use the queues anymore

Submit all your package, with all classes.
The package hierarchy as specified in section 5.1. Make sure you compile with Maven.

1.6 A Code Example

The following simple example clarifies the usage of the framework to implement a system of banks. Each bank has an actor. The Transmission action is an action submitted for the actor of the sender bank in order to transfer an amount of money from its customer to another customer in different bank. In the start method, the bank sends a message to the second banks asking for confirmation, since the confirmation involves access to the private state of the customer in the other bank. In *then* method, we define the confirmation action as a dependency for the completion of the transformation action.
In the Confirmation action we actually do some checks and decide on the answer.

In the main method we start the thread pool and submit actions. Following, an example of a simple main method which submits one action to the Thread Pool,

1.6.1 Simple Main Method

Listing 1: Submitting Transmission action to the Actor Thread Pool

```
1  ActorThreadPool pool = new ActorThreadPool (8);
2  Action<String> trans = new Transmission (100 , "A","B" , "bank2", "bank1");
3  pool.start();
4  pool.submit(trans, "bank1", new BankStates());
5  CountDownLatch l = new CountDownLatch(1);
6  trans.getResult().subscribe(() -> {
7  l.countDown();
8  });
9  l.await();
10 pool.shutdown();
```

1. In line 1 we create a thread pool with 8 threads.
2. In line 2 we create an action of type Transmission. Transferring 100 from "A" in "bank1" to "B" in "bank2".
3. In line 3 we start the thread pool.
4. In line 4 we submit the action to the thread pool. We pass the id of the actor of bank1. Also, in order for the thread pool to maintain a Private State object for "bank1" actor, we send an object of Private State (here we send BankState which extends PrivateState). If the actor is already in the Pool, this object should be ignored. That is, the thread pool should hold only one PrivateState object per actor, which is the one received when we submit an action for the actor for the first time.
5. The main thread waits for the action to complete before existing the program, by using CountDownLatch mechanism.

Possible implementation of the Transmission action:

1.6.2 Transmission Action - Basic Implementation

```
1  public class Transmission extends Action<String>{
2  int amount;
3  String sender;
4  String receiver;
5  String receiverBank;
6
7  public Transmission(int amount, String receiver, String sender,
8  String receiverBank, String senderBank) {
9  this.senderBank = senderBank;
10 this.sender = sender;
11 this.receiver = receiver;
12 this.amount = amount;
13 this.receiverBank = receiverBank;
14 }
15
16 @Override
17 protected void start() {
18 List<Action<Boolean>> actions = new ArrayList<>();
19 Action<Boolean> confAction = new Confirmation(sender,receiver,
20 receiverBank, new BankStates());
21 actions.add(confAction);
22 then(actions, () ->{
23 Boolean confirmationResult = actions.get(0).getResult().get();
24 if (confirmationResult == true){
25 complete("transmission succeed");
26 System.out.println("transmission succeed");
27 }
28 else{
29 complete("transmission failed");
30 System.out.println("transmission failed");
31 }
32 sendMessage(confAction , receiverBank, new BankStates());
33
34 });
35 }
36 }
```

1. line 1. By Action<String> we define the result type of this action to be String.
2. line 17. We override the abstract start method. Here we place the behavior of the action.
3. line 19. We create another action of type Confirmation. This action will be submitted for "bank2"'s actor, since it requires access to its private state.
4. line 22. We define that the completion of the transmission action depends on the completion of the confirmation action.
5. line 23-31. This is the continuation which complete the transmission action. When the result of the confirmation action will be resolved, the transmission action will be inserted again to bank1 queue, the transmission action will be fetched later by actor 1 and the action will be completed by applying the defined continuation.
6. line 32. In sendMessage we put the action (which is the message) in "bank2"'s actor queue.

2 Part 2: University Management System

In this section you will simulate a University Management System using your Actor Thread Pool framework from part 1. The university has a set of departments, each department offers a set of courses to students (of all depart-

ments) and each student can register for courses if he meets the prerequisites and there is an available space for him. The register/unregister request are considered until the end of the registration period. In this System we define Actors and their Private States. It is obligatory to comply to our definitions without any change.
In this section you are not allowed to use any type of synchronization on the Actors private state. You have to plan your implementation of the actions in such way that you will not need synchronization.

2.1 Program Flow

The program is divided into three phases. Once a phase is completed, you will proceed to the next phase.

- Phase 1: All open course actions appear in Phase 1. There might appear other actions as well.
- Phase 2: Any action can appear
- Phase 3: Any action can appear

At first you open all suggested courses of all departments. Once the first phase is completed, it is possible for the students to register for the courses.

Note: Students can be added while the registration is run.
Note: Students can register for courses in different departments.

2.2 Actors

In order to simulate the university, you should create three types of actors:

- An actor per student.
- An actor per course.
- An actor per department's secretary.

2.3 Private States Of Actors

Each actor should maintain in its private state a log of all actions it has preformed. In addition, the private states include:

- Department: A department's private state includes a *list of courses* and the *list of students in the department*.
- Course: A course's private state includes the *number of available places*, the *list of students in the course*, *number of registered students*, and *prerequisites*.
- Student: A student's private state includes *grades-sheet* and `department's signature`. The grades-sheet contains all courses taken by the student with his grades.

Important: You are not allowed to use concurrent data structures to maintain the the private states of the actors.
Important: You are not allowed to extend the private state beyond what is described above, e.g, a department cannot hold a list of students enrolled in each course, it is part of the private state of the course only. Also, the department cannot hold the grades-sheet of the students, it is part of the private state of the student only.

2.3.1 Logging Actions

In its private state, the actor maintains a list of all actions it has executed. The list holds only the description of the action as it is shown in the JSON file in section 4.9.2. *E.g.*, "Open Course", "Add Student", etc.

2.4 Actions

Following is a list of actions that you should enable.

Important: For each action you might create new actors, create new actions and submit them to other actors, etc. You must implement the action in a way which avoids synchronization on the private states of the actors.

1. Open A New Course:
    - Behavior: Opens a new course in a specified department.
    - Actor: Should be initially submitted to the Department's actor.
2. Add Student:
    - Behavior: Adds a new student to a given department.
    - Actor: Should be initially submitted to the Department's actor.
3. Participating In Course:
    - Behavior: Registers a student to the course, if succeeds, adds the course to the grades-sheet of the student and give him a grade if supplied. See input example.
    - Actor: Should be initially submitted to the course's actor.
4. Unregister:
    - Behavior: If the student is enrolled in the course, unregister him (update the list of students of course, remove the course from the grades sheet of the student and increases the number of available spaces).
    - Actor: Should be initially submitted to the course's actor.
5. Close A Course:
    - Behavior: Closes a course. Should unregister registered students and remove the course from the department courses' list and the grade-sheets of the students. The number of available spaces of the closed course should be updated to -1. DO NOT remove its actor. After closing the course, all requests for registration should be denied.
    - Actor: Should be initially submitted to the department's actor.
6. Open new places in a course:
    - Behavior: Increases the number of available spaces for the given course.
    - Actor: Should be initially submitted to the course's actor.
7. Check Administrative Obligations:
    - Behavior: The department's secretary allocates one of the computers available in the warehouse, and check for each student if she meets some administrative obligations. The computer generates a signature and save it in the private state of the students.
    - Actor: Should be initially submitted to the department's actor.
8. Announce the end of registration period:
    - Behavior: From this moment, reject any further changes in registration. And, close courses with number of students less than 5.
    - Actor: Should be initially submitted to the department's actor.
9. Register With Preferences: The student supply a list of courses he is interested in them, and wish to register for ONLY one course of them. The courses are ordered by preference. That is, if he succeed to register for the course with the highest preference it will not try to register for the rest, otherwise it will try to register for the second one, and so on. At the end, he will register for at most one course.

2.5 Warehouse

The warehouse class holds a finite amount of computers. When the department wants to acquire a computer, it should lock its mutex if it is free. And release it once it finished the work with the computer. If the computer is not free, the department should not be blocked. It should get a promise which will be resolved later when the computer becomes available.

2.5.1 Computer

A computer has a method of checkAndSign. This method gets a list of the grades of the students and a list of courses he needs to pass (grade above 56). The method checks if the student passed all these courses.

2.6 Simulator

The simulator class runs the simulation. We may replace your ActorThreadPool implementation with our own during testing, so be sure to implement all simulation functionality in the Simulator class, not in the ActorThreadPool! Once constructed, calling its start() function will perform the following:

- Parse the Json Files.
- Submit actions to the thread pool.
  DO NOT create an ActorThreadPool in *start*. You need to attach the ActorThreadPool in the *main* method, and then call *start*.

Calling simulator.end() will preform the following:

- shut down the simulation.
- returns a *Map* containing the private states of the actors as serialized object to the file "result.ser". You may do so using this code, or similar:

Listing 2: A Snippet Code For Writing The Output

```
1    Map<String, PrivateState> SimulationResult;
2    SimulationResult = SimulatorImpl.end();
3    FileOutputStream fout = new FileOutputStream("result.ser");
4    ObjectOutputStream oos = new ObjectOutputStream(fout);
5    oos.writeObject(SimulationResult);
```

- The output filename MUST BE result.ser

2.7 Input Format

2.7.1 The JSON Format

The input files for this assignment is given as JSON files (http://www.json.org). There are a number of different options for parsing JSON files. We recommend the Gson library: https://github.com/google/gson/blob/master/UserGuide.md.

2.7.2 Input File

Below, an example of JSON input file. The JSON file is provided to the program as a command line argument. You can assume that the **input is legal**, *e.g.*, student does not try to register to a course which is not exist, etc.

```
{
"threads": 8,
"Computers" : [
{
"Type":"A",
"Sig Success": "1234666",
"Sig  Fail": "999283"
},
{
"Type":"B",
"Sig Success": "4424232",
"Sig  Fail": "5555353"
}
],
"Phase 1" : [
```

```
{
"Action":"Open Course",
"Department": "CS",
"Course": "SPL",
"Space": "400",
"Prerequisites" : ["Data Structures", "Intro to CS"]
},
{
"Action":"Open Course",
"Department": "CS",
"Course": "Data Bases",
"Space": "30",
"Prerequisites" :  ["SPL"]
},
{
"Action": "Add Student",
"Department": "CS",
"Student": "123456789"
}
],
"Phase 2" : [
{
"Action": "Add Student",
"Department": "Math",
"Student": "132424353"
},

{
"Action": "Participate In Course",
"Student": "123456789",
"Course": "SPL",
"Grade": ["98"]
},
{
"Action": "Add Student",
"Department": "CS",
"Student": "5959595959"
},
{
"Action": "Add Spaces",
"Course": "SPL",
"Number": "100"
},
{
"Action": "Participate In Course",
"Student": "123456789",
"Course": "Data Bases",
"Grade": ["-"]
},
{
"Action": "Register With Preferences",
"Student": "5959595959",
"Preferences": ["Data Bases","SPL"],
"Grade": ["98","56"]
},
{
```

```
"Action": "Unregister",
"Student": "123456789",
"Course": "Data Bases"
},
{
"Action": "Close Course",
"Department": "CS",
"Course": "Data Bases"
},
{
"Action" : "End Registeration"
},
{
"Action" : "Administrative Check",
"Department": "CS",
"Students": ["123456789","5959595959"],
"Computer": "A",
"Conditions" : ["SPL", "Data Bases"]
}
],
"Phase 3": [
{
"Action" : "Administrative Check",
"Department": "CS",
"Students": ["123456789","5959595959"],
"Computer": "A",
"Conditions" : ["SPL", "Data Bases"]
}
]



}
```

The json object contains the following fields:

- *threads* - an integer defining the amount of threads in the *Actor Thread Pool*.
- Computers - an array of computers in the warehouse, each computer has two signatures.
- Phase 1 - An array of all open courses actions (some other action might appear). All actions in Phase 1 should be completed before proceeding to Phase 2.
- Phase 2 - An array of actions. All actions in Phase 2 should be completed before proceeding to Phase 3.
- Phase 3 - An array of actions.

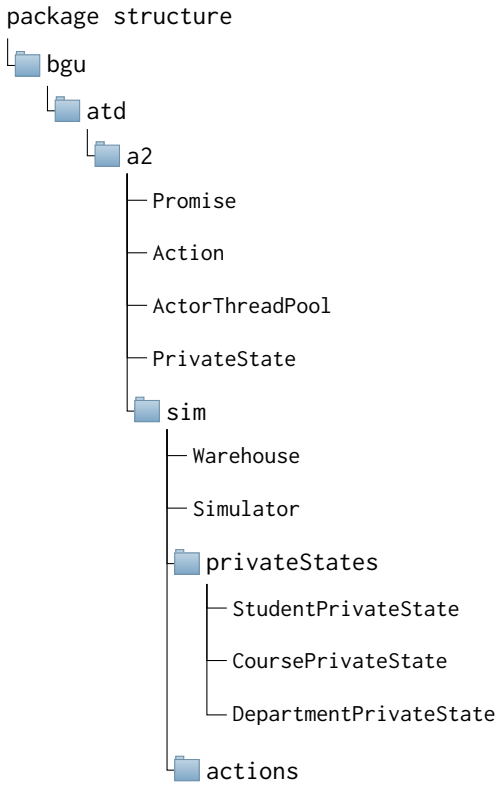
Note: the number and order of the actions might be different. You must insert the actions in the order they appear in the json.

Important: The simulation file should be given as argument in the main function. That is, you can't use a predefined name or location. The run command is:
mvn exec:java -Dexec.mainClass="bgu.atd.a1.sim.Simulator" -Dexec.args="myFile.json".
In Dexec.args we pass arguments to the main method - in our case this argument represents the simulation file (the file name is NOT necessary myFile.json). //

3 Submission Instructions

3.1 Packages

In order for your implementation to be properly testable, you must conform to the following package structure.



You may add classes as you see fit. Your application should run correctly if we replace the your implementation of the first part with our own implementation.
Important: If you do not conform to the above structure, automated tests may fail
Important: All actions which extend Action should be placed in bgu.atd.a1.sim.actions.
Use maven as your build tool.

3.2 Deadlocks, Waiting, Liveness and Completion

Deadlocks You should identify possible deadlock scenarios and prevent them from happening.
Waiting You should understand where wait cases may happen in your program, and how you have solved these issues.
Liveness Locking and Synchronization are required for your program to work properly. Make sure you don't damage the liveness of the program. Remember, the faster your program completes its tasks, the better.
Completion As in every multi-threaded design, special attention must be given to the shut down of the system. When the ActorThreadPool receives a "shutdown" command, the program needs to terminate. This needs to be done gracefully, not abruptly.

3.3 Submission Instructions

- Submit the whole package as specified in section 5.1.
- The submitted zip should contain the src folder and the pom.xml file only! no other files are needed.