German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher

Computer Programming Lab, *Spring 2019*
Rescue Simulation: **Milestone 2**

Deadline: 01.04.2019 @ 23:59

This milestone is a further *exercise* on the concepts of **object oriented programming (OOP)**. By the end of this milestone you should have a working game engine with all its logic, that can be played on the console if needed. The following sections describe the requirements of the milestone. Refer to the **Game Description Document** for more details about the rules.

- Please note that you are not allowed to change the hierarchy provided in milestone 1 except as explicitly stated in this milestone description. You should also conform to the method signatures provided in this milestone. However, you are free to add more helper methods and instance variables. You should implement helper methods to include repetitive pieces of code.
- All methods mentioned in the document should be `public`. All class attributes should be `private` with the appropriate access modifiers and naming conventions for the getters and setters as needed.
- You should always adhere to the OOP features when possible. For example, always use the `super` method or constructor in subclasses when possible.
- The model answer for M1 is available on the MET website. It is recommended to use this version. A full grade in M1 doesn't guarantee a 100 percent correct code, it just indicates that you completed all the requirements successfully :)
- **You need to carefully read the entire document to get an overview of the game-flow as well as the milestone deliverables, before starting the implementation.**
- The orientation of the grid we will be using can be found in Fig. 1.

Game Logic

The game is cycle-based. Before moving on to the detailed descriptions of the methods, where they should be implemented and what they should do, we give a brief overview of the flow of the simulation from the point of view of one simulation cycle. The main method responsible for proceeding the cycles of the game and thus proceeding the simulation, is the `nextCycle()` method in the `Simulator` class. The `nextCycle()` is responsible for handling the tasks performed by each of the `Simulatable` objects by calling their `cycleStep()` methods.The `cycleStep` method of each `Simulatable` object is responsible for performing the actions done by the object within each cycle. The units either move closer to their target or start treating their target if they are not idle. The disasters strike and continue inflicting damage on the buildings or citizens. Buildings and citizens continue to update their primary attributes according to their state.

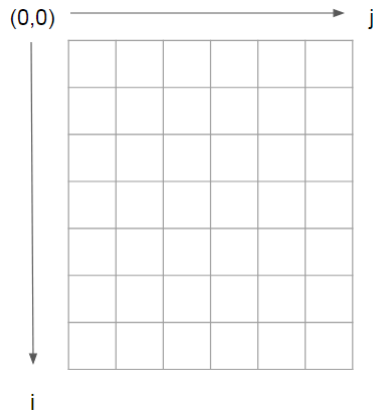

Figure 1: Map grid orientation with the indices.

1 Listerners and Interfaces

1.1 `Rescuable` Interface

You should add the following methods to the built interface:

1. `void struckBy(Disaster d)`: This method is responsible for applying a disaster to a building or a citizen. It should update the `state` where applicable and inform the appropriate `SOSListener`.
2. `Address getLocation()`: This method returns the location of the `Rescuable` object.
3. `Disaster getDisaster()`: This method returns the disaster currently affecting the `Rescuable` object.

1.2 `Simulatable` Interface

You should add the following method to the previously built interface:
`void cycleStep()`: This method is responsible for performing the actions of each `Simulatable` object, i.e. unit, disaster, citizen and building, that are done within each cycle.

Carefully consider where the method will be implemented, as well as where and how it will be overridden given the classes of the various disaster, unit, building and citizen types. **Carefully consider all the special cases when implementing and overriding this method.**

1.3 `SOSListener` Interface

You should add the `SOSListener` interface in the `model.events` package. The `CommandCenter` should be able to listen to SOS calls.

You should add the following method to the built interface:

- `void receiveSOSCall(Rescuable r)`: This method handles receiving SOS calls from any `Rescuable` reporting that they are affected by a disaster. The method should handle the call by adding said `Rescuable` to the appropriate list of visible rescuables.

1.4 `SOSResponder` Interface

You should add the `SOSResponder` interface in the `model.events` package. All units should be able to respond to SOS calls.

You should add the following method to the built interface:

- `void respond(Rescuable r)`: This method is used by the player during gameplay to decide which unit to dispatch to which rescuable. It is responsible for assigning the unit to respond to the given target `r` while updating the unit's `state`. If the unit already had a target `s` (other than `r`), then the disaster of `s` be should reactivated before the unit switches its target to `r` (as this unit will now respond to the disaster of another target instead). The only exception to this is, if a medical unit is already in the healing phase, then the disaster will not be reactivated. The method should also set the `distanceToTarget` using the Manhattan distance ($\triangle x + \triangle y$).

  Carefully consider where the method will be implemented, as well as where and how it will be overridden given the classes of the various units. **Carefully consider all the special cases when implementing and overriding this method.**

1.5 `WorldListener` Interface

You should add the `WorldListener` interface in the `model.events` package. The `Simulator` should implement this interface, as it is responsible for updating the location of any unit or citizen. You should update the constructors of the classes that will require updating locations within the world, to include and initialize the listener.

You should add the following method to the previously built interface:

`void assignAddress(Simulatable sim, int x , int y)`: This method updates the location of `sim` to the target `x` and `y`. You need to make sure to get the targeted location from the `world` array.

2 Disasters

2.1 `Disaster` Class

You should add the following method to the previously built class:

1. `void strike()`: This method is responsible for applying the current disaster on its target, activating the disaster, and inflicting the initial damage by calling the setter method(s) to update the appropriate secondary attributes of the target. Each disaster has a specific initial damage as follows:
   - `Collapse` increases `foundationDamage` by 10.
   - `Fire` increases `fireDamage` by 10.
   - `GasLeak` increases `gasLevel` by 10.
   - `Infection` increases `toxicity` by 25.
   - `Injury` increases `bloodLoss` by 30.
2. `void cycleStep()`: This method is responsible for applying the disaster's continuous effect on the secondary attributes of its target. The secondary attributes of the different targets and how they are affected are as follows:
   - `Collapse` increases `foundationDamage` by 10 per cycle.
   - `Fire` increases `fireDamage` by 10 per cycle.

- `GasLeak` increases `gasLevel` by 15 per cycle.
- `Infection` increases `toxicity` by 15 per cycle.
- `Injury` increases `bloodLoss` by 10 per cycle.

3 Rescuables

3.1 `Citizen` Class

You should add the following attributes to the class:

1. `SOSListener emergencyService`: The SOS listener assigned to the citizen in case any disaster strikes him/her. This attribute is WRITE ONLY.
2. `WorldListener worldListener`: The world listener responsible for updating the position of the citizen in the grid. This attribute is READ and WRITE.

You should update the following in the built class:

1. `void cycleStep()`: Each cycle the Citizen should update their primary attribute (`hp`) according to the values of their secondary attributes (`bloodLoss` and `toxicity`).
   - secondary attribute strictly between 0 and 30, decreases the hp by 5.
   - secondary attribute greater than or equals 30 and strictly less than 70, decreases the hp by 10.
   - secondary attribute greater than or equals 70 decreases the hp by 15.
2. `void setHp(int hp):` the hp can never exceed 100 or be less than 0. In case the given input is outside of this range, make sure that the hp value should be set correctly. The citizen in deceased once their hp reaches 0.
3. `void setBloodLoss(int bloodLoss):` the blood loss can never exceed 100 or be less than 0. The hp becomes 0 once the blood loss reaches 100.
4. `void setToxicity(int toxicity):` the toxicity can never exceed 100 or be less than 0. The hp becomes 0 once the toxicity reaches 100.

3.2 `ResidentialBuilding` Class

You should add the following attribute to the class:

1. `SOSListener emergencyService`: The SOS listener assigned to the building in case any disaster strikes it. This attribute is WRITE ONLY.

You should update the following in the built class:

1. `void cycleStep()`: Each cycle the building should update their primary attribute(`structuralIntegrity`) according to the values of their secondary attributes as follows:
   - `foundationDamage` greater than 0 decreases the `structuralIntegrity` by a random value between 5 and 10.
   - `fireDamage` strictly between 0 and 30, decreases the `structuralIntegrity` by 3
   - `fireDamage` greater than or equals 30 and strictly less than 70, decreases the `structuralIntegrity` by 5
   - `fireDamage` greater than or equals 70 decreases the `structuralIntegrity` by 7.

- the `gasLevel` does not directly affect the `structuralIntegrity`.

2. `setStructuralIntegrity(int structuralIntegrity)`: it can never be less than 0. The hp of all the citizen occupying the building should be set to 0, if the structural integrity reaches 0.
3. `setFireDamage(int fireDamage)`: it can never be less than 0 or exceed 100.
4. `setGasLevel(int gasLevel)`: it can never exceed 100 or be less than 0. If the gas level reaches 100, the hp of all the citizen occupying the building should be set to 0.
5. `setFoundationDamage(int foundationDamage)`: if it reaches 100 or more, the structural integrity of the building should be set to 0.

4 Units

4.1 `PoliceUnit` Class

You should update the following attribute:

1. `ArrayList<Citizen> passengers`: This attribute is READ ONLY.

4.2 `MedicalUnit` Class

You should update the following attribute:

1. `int treatmentAmount`: This attribute is READ ONLY.

4.3 `Unit` Class

You should add/update the following attribute to the class:

1. `WorldListener worldListener`: The world listener responsible for updating the position of the unit in the grid. This attribute is READ and WRITE.
2. `int distanceToTarget`: This attribute should be WRITE ONLY.

You should add the following methods to the previously built class:

1. `void cycleStep()`: This method is responsible for handling the different cases a unit can be in if it is not idle and is currently assigned to a target:
   - if the unit is en-route, then the `distanceToTarget` should be updated according to the Unit's stepsPerCycle and all the other necessary updates should be preformed. The unit's location should only be updated on arrival.
   - if the unit already arrived to the target's location in the previous cycle, then the `treat` method should be called as it is responsible for handling the disaster onsite. The functionality of the `treat` method for each unit type will be explained in more detail below.
   - you also need to consider the special case of the `Evacuator`, as it has a third case which is already being fully loaded with evacuees and thus needing to go back to the base and unload. You should try to utilize the `distanceToBase` attribute as well as the default behaviour of the `cycleStep()` when the unit is en-route. For example, re-calculating the `distanceToTarget` will cause the evacuator unit to go back to the building and continue handling the disaster.
2. `void treat()`: This method is responsible for treating the affected rescuable from a disaster by deactivating the disaster and fixing the secondary attributes. Depending on the unit type, this method should decrease the corresponding attribute by:

- **Buildings**: 10.
- **Citizens**: the method should first treat by the `treatmentAmount` then once the secondary attributes are restored to 0, it should call the `void heal()` method (that **should be implemented** in the relevant unit classes) to start restoring the hp of citizens by the `healingAmount`. Once the secondary attributes are restored, the `state` of the Citizens should be marked as `RESCUED` regardless of their `hp` value.

You should consider the special case of the `Evacuator` as its role is to load occupants from the building as long as it has not reached its maximum capacity. You should update the state of the evacuated citizens that reach the base to be `RESCUED`.

3. `void jobsDone()`: This method should change the unit's `state` once its job is done. This can be achieved by either successfully tending to the disaster or if the target of the unit has either died or was destroyed.

5 Simulation

5.1 `Simulator` Class

You should add/update the following attributes to the class:

1. `ArrayList<Unit> emergencyUnits`: This attribute is READ ONLY.
2. `SOSListener emergencyService`: The SOS listener. This attribute is WRITE ONLY.

You should modify the already implemented constructor such that it takes an `SOSListener` as input. The updated signature should be `Simulator(SOSListener emergencyService)`. You need to think about how and when you will set the `SOSListener` as well as the `WorldListener` attributes of the appropriate `Simulatable` objects.

You should add the following methods to the previously built class:

1. `boolean checkGameOver()`: This method checks whether the simulation is over. Please note that the simulation keeps running as long as:
   (a) There are still planned disasters that haven't been executed yet.
   (b) There are still active disasters targeting either 1) still alive citizens or 2) not destroyed buildings.
   (c) There are units that are not idle.

   Otherwise, the simulation is over.
2. `int calculateCasualties()`: This method calculates the casualties by counting the deceased citizens using their `state`.
3. `void nextCycle()`: This method is responsible for proceeding the game simulation for one cycle by performing the actions needed within each cycle. The simulation starts with the `currentCycle` set to 0. This cycle counter should always be updated at the beginning of this method. This method can be considered the core operating method of the whole simulation. All the different `Simulatable` objects i.e. units, disastesr, citizens and buildings perform different actions. The actions performed every cycle by each of the `Simulatable` classes are specified by the `cycleStep()` method in said class. The different events and checks that can happen within each cycle have a priority. The `nextCycle()` method checks each of them **sequentially according to their priority**, based on to the following:
   (a) **Planned Disasters**: The method starts by checking for the planned disasters that should be executed in the current cycle. Every one of these disasters should be removed from the list of planned disasters and the following special conditions need to be considered:

- if a building already suffers from a gas leak and it should now be struck by a `Fire` then,
  - if the gas level is 0, the fire should strike normally.
  - if the gas level is strictly between 0 and 70, then a new `Collapse` disaster should strike the building instead of the planned `Fire`.
  - if the gas level is greater than or equal 70, then the structural integrity of the building should be set to 0, resulting in the death of all its occupants.
- if a building already suffers from a fire and should now be struck by a `gasLeak`, a new `Collapse` disaster should strike the building instead of the planned `gasLeak`.
- You do not need to consider the case where a citizen can be hit by two disasters.

This method should also check all the buildings and make sure that whenever a building's `fireDamage` reach 100, a new `Collapse` disaster should be executed on said building.

After considering the above conditions, all the appropriate disasters should be executed by invoking their `strike` method and adding them to the list of executed disasters.

Finally, you should also make sure that whenever a `Collapse` disaster is executed on a building, any previous disasters on said building should be removed and the building's `fireDamage` should be set back to 0.

(b) **Units**: after activating all new/planned disasters, the priority goes to the **Units** by having the method call the `cycleStep()` of all the available units. More details about the **Unit**'s `cycleStep()` are provided later in this document.

(c) **Executed Disasters**: The method should then check for the executed disasters, to proceed their simulation actions. Here you need to ensure that the disaster is active and has been added in a previous cycle and not newly added in the current cycle (in which case its continuous damage should not be handled this cycle). In this case the disaster should continue doing its effect by calling its `cycleStep()`.

(d) **Residential Buildings and Citizens**: Finally, the method should call the `cycleStep()` of all the buildings followed by all the citizens. This will make sure that all buildings and citizens are updated correctly based on the previous events.

An overview of the flow of the method is given in Fig. 2.

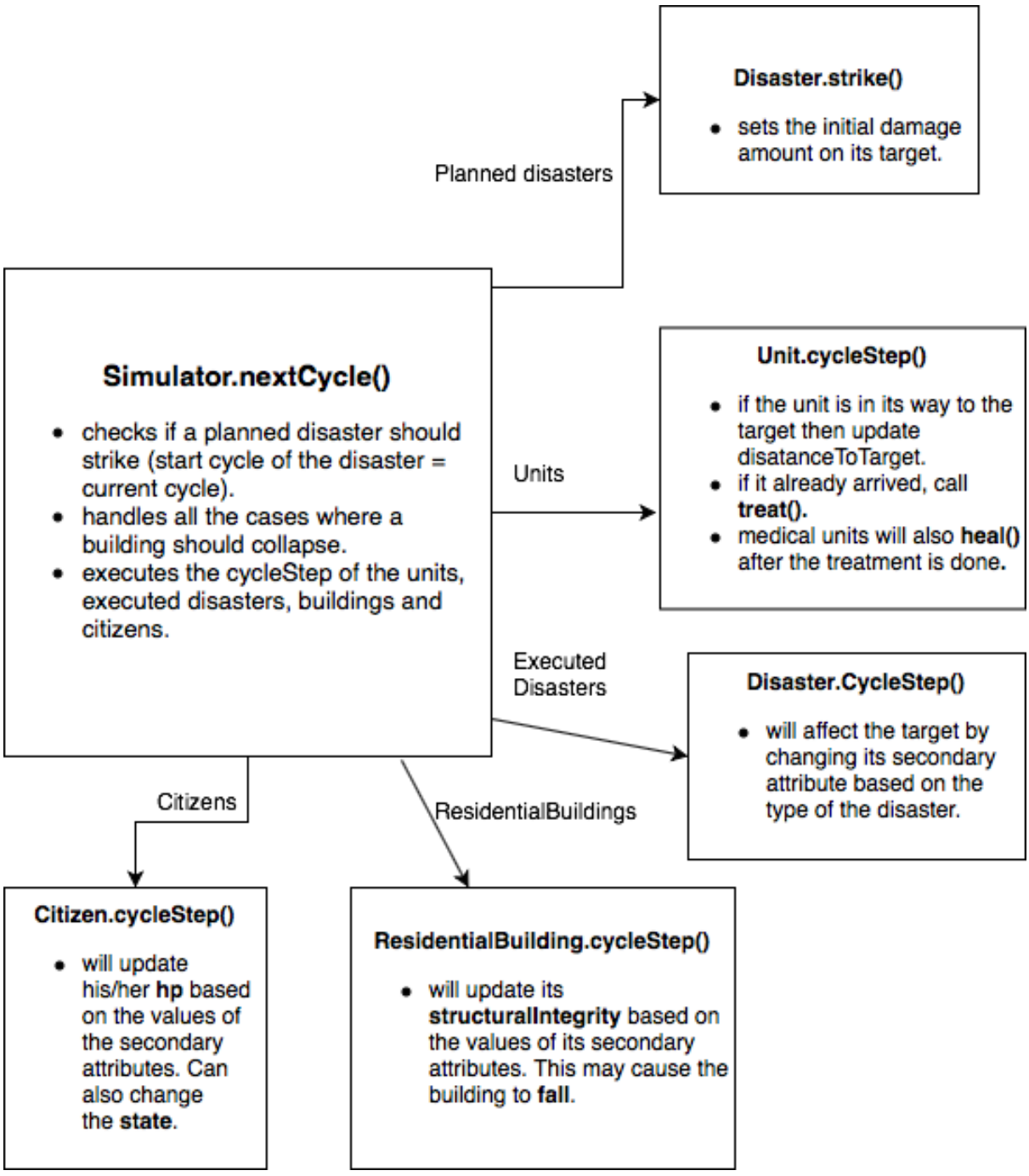

Figure 2: Overview of the flow of the `nextCycle` method.