University of Exeter

Coursework Specification
ECM1410: Object-Oriented Programming
Module Leader: Dr. Diogo Pacheco
Academic Year: 2025/26

Title: CityRescue- *Emergency Dispatch Simulator in Java*

Submission deadline: 5th March

This assessment contributes 50% of the total module mark and assesses the following intended learning outcomes:

1 demonstrate an appreciation of object-oriented modelling techniques;

2 interpret and modify program fragments in an object-oriented language;

3 follow an object-oriented development method to produce a design from a specification;

4 systematically test programs developed;

5 document software to accepted standards;

6 interpret a requirements specification;

7 systematically break down a problem into its components;

8 understand and choose between programming languages, and basic techniques;

9 use technical manuals and books to interpret technical errors;

10 analyse a problem and synthesis a solution;

You are reminded of the University's regulations on collaboration and plagiarism. You must avoid plagiarism, collusion, and any academic misconduct behaviours. Further details about academic honesty and plagiarism can be found at https://ele.exeter.ac.uk/course/view.php?id=1957.

Use of AI tools in AI-Minimal Assessments

Module Code and Name: ECM1410 – Object-oriented programming.

The University of Exeter is committed to the ethical and responsible use of Generative AI (GenAI) tools in teaching and learning, in line with our academic integrity policies where the direct copying of AI-generated content is included under plagiarism, misrepresentation and contract cheating under definitions and offences in TQA Manual Chapter 12.3.

This assessment falls under the category of AI-Minimal in the University's Guidance on use of Gen AI in Assessment. Copilot should be turned off during development.

This means: You may use AI tools for checking spelling and grammar mistakes only, with no other impact on the structure or content of the assessment. This is because using GenAI tools outside of these uses prevents fair assessment of your ability to achieve module learning outcomes.

When writing your assessment, you must never use AI tools:

1. For uses other than checking your spelling and grammar.
2. To translate more than a word or short phrase into English.
3. To upload sensitive or identifying material to an AI tool
4. To present material that has been generated by AI as your own work or the work of someone else.

Instructions

This coursework specification focuses on the problem narrative, required behaviour, and method contracts. **All practical instructions for completing and submitting the coursework** (including how to obtain the starter code, what files to submit, deadlines, marking/autograding details, and any additional rules or constraints) will be provided in a **separate instruction sheet**, which will be uploaded alongside this document. You must read and follow **both** documents; where the instruction sheet gives additional operational guidance (e.g., submission format), it should be treated as authoritative for those practical details.

CityRescue

1. The story

It's 07:59 on a rainy Monday and the city's control room is already busy. A small fire is reported near the market. A cyclist is injured on a side street. A shopkeeper calls in a burglary.

Your job is to build the software behind the dispatcher's screen — the part that keeps track of the city map, the stations, the emergency units, and the incidents as they unfold over time. The front-end (and the marking tests) will call your code through an interface. If your backend makes the same choices every time, the city stays calm — and the automated tests stay happy.

1.1 What you are building

- A 2D grid city map with obstacles.
- Stations placed at fixed locations on the grid.
- Emergency units (ambulances, fire engines, police cars) based at stations.
- Incidents that are reported, dispatched, handled on-scene, and resolved as time ticks forward.
- Deterministic behaviour: the same input must always lead to the same output.

1.2 What we are *not* asking you to do

This coursework is meant to be challenging, but it must be solvable using the material taught in the module. So we deliberately avoid a few topics that are commonly used in real systems.

- No file reading/parsing is required (the tests will call your methods directly).
- No Java Collections required (use arrays).
- No shortest-path algorithms required (a movement rule is provided).
- No JUnit requirement (you may use assert and the provided driver/tests).

2. The world model (your classes)

To keep your code organised and genuinely object oriented, you must model the main ideas in the problem as classes. Think of each class as a 'role' in the simulation.

2.1 Required classes

- CityRescueImpl — your main class that implements the CityRescue interface.
- CityMap — knows the grid size, which cells are blocked, and whether a move is legal.
- Station — stores station details and the units based there (with a capacity).
- Incident — stores incident details and its lifecycle status.
- Unit (abstract) — common unit behaviour shared by all emergency vehicles.
- At least three Unit subclasses — e.g., Ambulance, FireEngine, PoliceCar.

2.2 Polymorphism (a key marking point)

Units are not all the same. A fire engine should respond to fires, not medical emergencies. An ambulance resolves incidents faster than a fire engine. We want you to express these differences using inheritance + overriding, not giant if-statements.

Your Unit abstract class must include at least one abstract method, for example:

```
boolean canHandle(IncidentType type);
int getTicksToResolve(int severity);
```

3. Enums (your state machines)

Enums are how we keep the simulation states clear and safe. They also make it harder to accidentally invent invalid states.

Enum	Values
IncidentType	MEDICAL, FIRE, CRIME
UnitType	AMBULANCE, FIRE_ENGINE, POLICE_CAR
IncidentStatus	REPORTED, DISPATCHED, IN_PROGRESS, RESOLVED, CANCELLED
UnitStatus	IDLE, EN_ROUTE, AT_SCENE, OUT_OF_SERVICE

4. The big rule: determinism

In the real world, two dispatchers might make two different 'reasonable' decisions. In automated marking, that's a nightmare.

So in this assessment whenever there is a choice, you must follow the tie-break rules exactly. That way, the expected output is unique and your program is testable.

4.1 Incident processing order

Always consider incidents in ascending incidentId order.

4.2 Choosing the 'best' unit (tie-breakers)

Among eligible units (correct type, not OUT_OF_SERVICE, and not already assigned):

1. Shortest Manhattan distance ($|x_1 - x_2| + |y_1 - y_2|$)
2. If tied: lowest unitId
3. If tied: lowest homeStationId

4.3 Movement preference

When multiple moves are possible, use fixed direction order: N, E, S, W.

5. Movement (obstacles included, no pathfinding)

Yes, there are obstacles. No, you don't need BFS or Dijkstra. Instead, units follow a simple, deterministic rule that is easy to implement with loops and conditionals.

Important: Units do **not** perform pathfinding and do **not** try to avoid loops. You must apply the movement rule exactly as written, even if an obstacle layout would cause a unit to oscillate or fail to reach its target.

Movement rule (applied once per tick for each EN_ROUTE unit):

1. List the four candidate moves in order N, E, S, W.
2. Ignore moves that go out of bounds or into a blocked cell.
3. Take the first legal move that reduces Manhattan distance to the target.
4. If none reduce distance, take the first legal move in N, E, S, W order.
5. If no legal move exists, the unit stays put this tick.

6. Incident lifecycle (from call to calm)

Incidents move through a predictable lifecycle. Your job is to enforce the legal transitions and update everything in the right order each tick.

- REPORTED — the incident exists, but no unit has been assigned yet.
- DISPATCHED — a unit has been assigned and is travelling.
- IN_PROGRESS — the unit has arrived and is working on scene.
- RESOLVED — the incident is complete and the unit becomes available again.
- CANCELLED — the incident was withdrawn before completion.

6.1 How long does an incident take to resolve?

To keep this coursework deterministic, each unit type takes a fixed number of ticks once it reaches the scene:

Unit type	Ticks at scene
AMBULANCE	2
POLICE_CAR	3
FIRE_ENGINE	4

6.2 Cancellation and escalation

Rules:

1. cancelIncident is allowed only when an incident is REPORTED or DISPATCHED.
2. If a DISPATCHED incident is cancelled, the unit is immediately released back to IDLE (it stays at its current location). Idle units remain stationary at their last location until dispatched again (they do not automatically return to their home station).
3. escalateIncident may change severity (1–5) unless the incident is RESOLVED or CANCELLED.

7. Design by contract (and exceptions)

Every public method in the interface is a promise. If callers follow the preconditions, your method must achieve the postconditions. If inputs are invalid, you should throw the documented exception.

One rule matter more than all the others:

1. If a public method throws an exception, the state of the entire system must be unchanged.

Practical hint: validate everything first, then apply changes only once you know the call is valid.

8. The public interface (22 methods)

The marking tests will call your solution through this public interface. Your implementation class must be called CityRescueImpl and must implement CityRescue.

8.1 Contract table (what each method promises)

#	Method	In plain English	Preconditions	Postconditions	Throws
1	`void initialise(int width, int height) throws InvalidGridException`	Start a fresh simulation.	width > 0, height > 0.	Clears all stations/units/incidents; tick becomes 0; obstacle grid emptied.	InvalidGridException if width/height invalid.
2	`int[] getGridSize()`	Ask the city how big it is.	—	Returns {width, height}.	—
3	`void addObstacle(int x, int y) throws InvalidLocationException`	Place a roadblock.	Location is in bounds.	Cell becomes blocked.	InvalidLocationException if out of bounds.
4	`void removeObstacle(int x, int y) throws InvalidLocationException`	Remove a roadblock.	Location is in bounds.	Cell becomes unblocked.	InvalidLocationException if out of bounds.
5	`int addStation(String name, int x, int y) throws InvalidNameException, InvalidLocationException`	Build a station.	Name is not blank; location in bounds and not blocked.	Creates station; returns stationId (starts at 1, increases).	InvalidNameException or InvalidLocationException.
6	`void removeStation(int stationId) throws IDNotRecognisedException, IllegalStateException`	Close a station (only if empty).	Station exists.	Station removed only if it owns no units.	IDNotRecognisedException if unknown; IllegalStateException if it still owns units.
7	`void setStationCapacity(int stationId, int maxUnits) throws IDNotRecognisedException, InvalidCapacityException`	Change station parking capacity.	Station exists; maxUnits > 0; not less than current unit count.	Capacity updated.	IDNotRecognisedException or InvalidCapacityException.
8	`int[] getStationIds()`	List station IDs.	—	Returns station IDs in ascending order.	—
9	`int addUnit(int stationId, UnitType type) throws IDNotRecognisedException, InvalidUnitException, IllegalStateException`	Add a vehicle to a station.	Station exists; station has free capacity; type not null.	Creates unit at station location; status IDLE; returns unitId (starts at 1, increases).	IDNotRecognisedException, InvalidUnitException, or IllegalStateException.
10	`void decommissionUnit(int unitId) throws IDNotRecognisedException, IllegalStateException`	Retire a unit (only when free).	Unit exists.	Unit removed only if not EN_ROUTE or AT_SCENE.	IDNotRecognisedException or IllegalStateException.
11	`void transferUnit(int unitId, int newStationId) throws IDNotRecognisedException, IllegalStateException`	Move a unit to a new home station.	Unit and station exist; unit is IDLE; destination has space.	Home station updated; unit location becomes the destination station location.	IDNotRecognisedException or IllegalStateException.

12	`void setUnitOutOfService(int unitId, boolean outOfService) throws IDNotRecognisedException, IllegalStateException`	Toggle OUT_OF_SERVICE.	Unit exists; if setting true then unit must be IDLE.	Unit becomes OUT_OF_SERVICE or returns to IDLE.	IDNotRecognisedException or IllegalStateException.
13	`int[] getUnitIds()`	List unit IDs.	—	Returns unit IDs in ascending order.	—
14	`String viewUnit(int unitId) throws IDNotRecognisedException`	Describe one unit (deterministic).	Unit exists.	Returns a formatted unit string.	IDNotRecognisedException if unknown.
15	`int reportIncident(IncidentType type, int severity, int x, int y) throws InvalidSeverityException, InvalidLocationException`	Log a new incident.	type not null; 1 ≤ severity ≤ 5; location in bounds and not blocked.	Creates incident as REPORTED; returns incidentId (starts at 1, increases).	InvalidSeverityException or InvalidLocationException.
16	`void cancelIncident(int incidentId) throws IDNotRecognisedException, IllegalStateException`	Cancel an incident.	Incident exists.	Allowed only if REPORTED or DISPATCHED; if DISPATCHED then release the unit; status becomes CANCELLED.	IDNotRecognisedException or IllegalStateException.
17	`void escalateIncident(int incidentId, int newSeverity) throws IDNotRecognisedException, InvalidSeverityException, IllegalStateException`	Change severity.	Incident exists; 1 ≤ newSeverity ≤ 5; not RESOLVED/CANCELLED.	Severity updated.	IDNotRecognisedException, InvalidSeverityException, or IllegalStateException.
18	`int[] getIncidentIds()`	List incident IDs.	—	Returns incident IDs in ascending order.	—
19	`String viewIncident(int incidentId) throws IDNotRecognisedException`	Describe one incident (deterministic).	Incident exists.	Returns a formatted incident string.	IDNotRecognisedException if unknown.
20	`void dispatch()`	Assign units to waiting incidents.	—	For each REPORTED incident (lowest id first), assign the best eligible unit using tie-breakers. Sets incident DISPATCHED and unit EN_ROUTE.	Never throws.
21	`void tick()`	Advance time by one tick.	—	Tick++ then update in this order: move EN_ROUTE units (ascending unitId), mark arrivals, process on-scene work, resolve completed incidents (ascending incidentId).	Never throws.
22	`String getStatus()`	Produce a full snapshot for the UI/tests.	—	Returns a multi-line deterministic status report.	Never throws.

Definitions used throughout this specification

The grid uses a **0-based coordinate system**. A location (x, y) is **in bounds** iff 0 ≤ x < width and 0 ≤ y < height.

IDs (stationId, unitId, incidentId) **start at 1** and are **monotonically increasing**. **IDs are never reused**, even if an entity is removed.

9. Output (what the tests will compare)

Your getStatus() output is designed to be compared as plain text. That means small differences matter: spelling, spacing, ordering. Keep it consistent and always sort by ID where lists appear.

9.1 getStatus() format

Use this exact layout:

```
TICK=7
STATIONS=2 UNITS=3 INCIDENTS=2 OBSTACLES=5
INCIDENTS
I#1 TYPE=FIRE SEV=4 LOC=(3,1) STATUS=IN_PROGRESS UNIT=2
I#2 TYPE=CRIME SEV=2 LOC=(0,4) STATUS=REPORTED UNIT=-
UNITS
U#1 TYPE=AMBULANCE HOME=1 LOC=(1,1) STATUS=IDLE INCIDENT=-
U#2 TYPE=FIRE_ENGINE HOME=2 LOC=(3,1) STATUS=AT_SCENE INCIDENT=1 WORK=2
U#3 TYPE=POLICE_CAR HOME=1 LOC=(1,2) STATUS=EN_ROUTE INCIDENT=2
```

9.2 viewUnit() format

```
U#2 TYPE=FIRE_ENGINE HOME=2 LOC=(3,1) STATUS=AT_SCENE INCIDENT=1 WORK=2
```

9.3 viewIncident() format

```
I#1 TYPE=FIRE SEV=4 LOC=(3,1) STATUS=IN_PROGRESS UNIT=2
```

10. Storage limits (arrays only)

Because you're using arrays, you must set sensible maximum sizes. Pick constants and enforce them. If the caller tries to exceed a limit, throw a documented exception (recommended: CapacityExceededException).

- MAX_STATIONS = 20
- MAX_UNITS = 50
- MAX_INCIDENTS = 200

Obstacles should be stored as a Boolean grid, e.g. blocked[x][y].

11. Exceptions you must provide

- InvalidGridException
- InvalidLocationException
- InvalidNameException
- InvalidCapacityException
- InvalidSeverityException
- InvalidUnitException
- IDNotRecognisedException
- CapacityExceededException (recommended)

12. Marking guide (example)

This is an example breakdown. Your module leader may adjust weightings, but the idea is the same: correctness first, then determinism, then design quality.

Component	Suggested weighting
Core correctness (dispatch, tick order, lifecycle)	40%
Determinism (tie-breakers, ordering, output)	15%
Object-oriented design (abstraction, encapsulation, polymorphism)	20%
Exceptions + contracts (validation, state unchanged on exception)	15%
Documentation (JavaDoc, clarity)	10%