Saarvive & Thrive

Contents

1. Organizational Matters

Since you have already completed the exercise phase, the practical part of the Software Engineering Lab now begins. The practical part is divided into a group phase (∼4 weeks) and a subsequent individual phase (∼2 weeks).

In the group phase, you will design, implement, and test a simulator together with your fellow students in a group of 5–7 people. Group assignment will be done by the chair on the first day of the group phase. You will be assigned a group in the CMS, your group will have access to a workroom at the university for the duration of the group phase. A tutor will be available to assist you and will be in regular contact with your group. Your tutor will let you know which room is available for you.

You will spend the first one and a half weeks (approximately) of the group phase for the design, the following two and a half weeks (approximately) for the implementation (incl. testing). You will present your designs to a member of the Chair of Software Engineering and receive feedback. The implementation includes the parsing of the simulation scenario as well as the simulation logic. In addition, you will need to develop an extensive test suite that tests all special cases that may arise, if possible.

Like customers in real life, the chair is very volatile in its requirements for the simulator to be implemented. Therefore, there will be a change of the simulation setting after the design review. Your design must be kept flexible enough to accommodate the changes in the task without much effort. The specific changes will then be announced in a separate document after the design reviews.

After the group phase, you will receive a new task in the individual phase, which must be worked on by all participants independently. You will receive more information about the individual phase after the end of the group phase. The prerequisite for participating in the individual phase is the successful completion of the group phase.

Please note:

For the schedule of the group phase, please additionally note the information from the document on Organizational Matters, which has already been provided to you in the CMS at the beginning of the SE Lab.

1.1. Design

Regarding the design, your group must complete the following tasks:

1. You must create a UML class diagram of your system. The class diagram must contain all relevant classes of your implementation, as well as their most important properties and methods. This should show which data and which functionality you encapsulate in each class. The relations of the classes among themselves must be modeled correctly. In the class diagram, all classes as well as all non-trivial methods together with their parameters must be specified. Trivial methods include, in particular, `hashCode`, `equals`, and `toString`.
2. You need to model the simulation in a state diagram to better understand the simulation flow and to ensure that you also fully consider this simulation flow in your design. Create a state diagram that is beginning at the start of the simulation (i.e., the specification files are not yet loaded) and models the simulation itself up to the end of the simulation. Try to make the diagram as fine-granular and detailed as possible.
3. You need to demonstrate the interaction of your classes using UML sequence diagrams for the following 2 scenarios:
   - Initialization of the simulation until the simulation starts.
   - Handling a single simulation tick including allocating assets and handling events. During this tick, there will be one emergency of type *FIRE* with severity 1 and one *ROAD CLOSURE* event ends. The base you assign for the emergency has enough assets to handle it.
   - You have one hospital with two ambulances and one emergency doctor car. There is enough staff for all vehicles. One ambulance is currently on route to a *MEDICAL* severity 1. In this tick, a *MEDICAL* severity 2 comes in and has to be handled.

   When the requirements change after the design review, we will announce a third scenario for which you will need to demonstrate the interaction of your classes using another UML sequence diagram.
4. You must draw up a detailed time and work plan for the implementation including testing. This plan specifies who will be responsible for which parts of the implementation and who will be responsible for testing which components. In this plan, pay particular attention to the dependencies between different components and the order of the next milestones. Do not forget to include writing of the integration and system tests in the plan as well.

Design Review & Design Defense

In the design review, you will receive feedback on your design from a member of the chair. In the design defense, your design has to be approved by a member of the chair. Both

appointments are mandatory review appointments on-site at the campus.

After the design reviews, we will introduce changes of the specification that have to be incorporated to your design before the design defense. Try to keep your design modular enough so that any changes can be easily incorporated into your design. However, you should not try to incorporate all possible changes into the design in advance.

For both appointments, design review and design defense, note that the current state of your draft that you will present to the chair member must already be uploaded in our GitLab repository 1 hour before your assigned review or defense appointment (see also Submissions).

1.2. Implementation

In this phase, you will complete the actual implementation of the simulator in your group. You will also write unit tests, integration tests, and system tests. Your simulator implementation must meet the following requirements:

1. Your implementation must pass the system tests we create. We run our system tests on your implementation regularly during the implementation phase, and the test results will be provided to you at regular intervals in due course.

2. Your implementation must pass the unit tests, integration tests, and system tests that you create. The unit tests are there to help you debug and correct errors. If you have unit tests that your implementation does not pass, you either made a mistake when creating the tests or your implementation is still buggy. Your tests are intended to achieve the highest possible code coverage while testing reasonable scenarios, which is ensured by finding mutants[1].

3. We will examine your implementation with the code analysis tool detekt. This happens automatically; your project needs to *build* and pass the code analysis tool on your `main` branch before our tests are run on your code. We expect to find no problems in this process. (If there are some rules of the code analysis tool which appear to you to be absurd in a specific situation, please contact your tutor. We expect you to be able to provide sufficient justification why you think that the rule is absurd in this situation.)

   With the build script we have created for you, you can run this tool yourself from the beginning of the implementation phase to check if your code still has problems.

Simply passing the tests is not enough, your code must also be well structured and follow the concepts presented in the lecture. In particular, each individual group member must contribute a significant amount to your group's code. (Be sure here to have git properly configured on your computer so that your commits are also assigned to you in our GitLab).

[1] https://en.wikipedia.org/wiki/Mutation_testing

Code Review

In the code review, a member of the chair will look at your code together with you, point out weak points, and give concrete suggestions for improvement. The code review is also a mandatory review appointment.

1.3. Tools Needed

To participate in the SE Lab, you need to bring your own laptop. You need to install the following tools on your device:

- Java 17 (our reference platform uses *OpenJDK 17.0.7*)[2]
- Version-control system *git*[3]
- IDE of your choice (we recommend IntelliJ[4])

The build system Gradle, which we use, does not need to be explicitly installed at your end, because we provide you with a project in which a *Gradle wrapper* is already included, which takes care of the installation by itself. Kotlin also does not have to be installed manually but will be installed via gradle. All other tools you need do not need to be installed, but are already pre-configured in the project by us.

All libraries included in the gradle configuration file (`build.gradle.kts`) may be used. Other libraries must not be used.

1.4. Submissions

We will provide you with a *git* repository for the group phase in our GitLab[5]. To be able to clone or push to the provided repository, you need an SSH key. You have already learned how to generate a SSH key and which other technical requirements are necessary for using the repository in the practical tutorial. In the group phase, the entire group will work on one repository.

A separate branch (not `main`) is provided for submitting your design. All design documents and associated diagrams must be pushed to the repository. You can upload photos of your diagrams on paper etc. for this purpose, as long as the resolution is high enough and details are visible. The final version of your design must be pushed and marked with the tag[6] *design_defense*, which must be attached to the submitted revision. This also holds already for the preliminary version of your design, which you have to submit prior to the design

[2] https://jdk.java.net/17/
[3] https://git-scm.com/
[4] https://www.jetbrains.com/idea/
[5] https://sopra.se.cs.uni-saarland.de/
[6] https://git-scm.com/book/en/v2/Git-Basics-Tagging

review: Use the tag *design_review* in this case. To create a tag for the current commit, use the GIT command `git tag <tagname>`. Note that the submission of your design must be pushed to the repository, at latest, 1 hour before your assigned appointment for the design defense.

For your implementation, you must use the `main` branch after the design defense. There, you will be provided with a project with a minimal code framework at the start of your implementation phase. We will not consider other branches than `main` for the code submission.

For the final submission of your implementation, the last commit on the `main` branch made to the repository no later than 23:59 CEST on the day of code submission counts. It is your responsibility to verify that the push has arrived in the repository and that everything is up to date in a timely manner. Your submissions must be executable on our reference platform (*Debian 11* with *OpenJDK 17*).

1.5. Interaction with Tutors regarding Task Specification

Inaccuracies in the task specification, which follows in the next chapters, are unavoidable. As in real software projects, there may be situations which are not completely specified in the specification text. Often this holds, for instance, for specific corner cases. In such a case, interaction with the customer is necessary. That is, you need to interact with the tutors (e.g., in the forum) to clarify unspecified details.

2. Emergency Management Simulation: Saarvive and Thrive

In this years Software Engineering Lab (SE Lab), you will develop a simulation for emergency response scenarios. This simulation will be a high-level abstraction of the real world, with the goal of being able to simulate the behavior of emergency response units in a county. In the following sections, we will describe the simulation in more detail, and the requirements that you will have to fulfill. In addition, we will describe the behavior of the different resources in the simulation, and how they are represented in the configuration files.

2.1. Introduction

The simulation will take place in a county, which is divided into a number of smaller villages. You will be provided with a configuration file that describes the county, the roads within and between the villages, and the crossings between the roads. In another configuration file, you will be provided with the locations of the emergency response units, such as fire stations, police stations, and hospitals. In addition, this file contains the configuration of all assets in the county, for example, the number of fire trucks, police cars, or police dogs in an emergency central. In order to be able to simulate the behavior of the emergency response units, you will also be provided with a number of emergency calls and events that will occur during the simulation. Before you can simulate the behavior of the emergency response units, you will have to parse the configuration files and create an internal representation of the county, the emergency response units, the emergency calls, and the events. Then, you will have to simulate the behavior of the emergency response units in response to the emergency calls. The game will be played in ticks (rounds), where each tick represents a time step in the simulation. The emergency calls and events will occur at specific time steps, and you have to make sure that you handle the emergency calls and events in the correct order. The ultimate goal of the simulation is to handle as many emergencies as possible with the resources provided to you.

2.1.1. Change Introduction

In the change to the specification you will have to react to changes in the environment of your vehicles. Where you were only calculating the time a vehicle needs to arrive at the

target (emergency site), you will now have to simulate the vehicles actually driving there and reacting to changes to the environment (e.g., via certain events). Furthermore, there will be changes that affect how you allocate resources to emergencies. One example is that you can reassign vehicles that are on the way to an emergency, to a different emergency that has a higher severity.

For more information please see the following sections.

2.2. Simulation Resources

In this section, we describe the different resources that this simulation has and needs including the emergency services, the map of the world, and the emergencies to be handled.

Please note:

In this section, we define multiple different properties of resources that will be provided to your simulation via configuration files. Please note that all constraints that we state for these properties need to be validated by your program either via one of the provided `JSON schemata` or by yourself.

2.2.1. Emergency Management Control Center (EMCC)

The Emergency Management Control Center (EMCC) is the central unit where all of the emergency calls are being received and handled. This center gets the calls for each new emergency. The EMCC then hands the emergency calls to the responsible emergency service.

2.2.2. Emergency Services

In our simulation there are three emergency services: Police, Fire Department, and Ambulance Services. These services are responsible for responding to emergencies calls and handling them. In the following sections, we describe the services and their responsibilities. Please note that this is an approximation of the real-world and as such might not be coherent with the responsibilities of these services in the real world.

Police In our simulation the police force has two responsibilities: responding to crime and supporting other services (e.g., securing the site of a fire).

Fire Department The fire department has three responsibilities: responding to fires, responding to technical emergencies (e.g., car crashes), and supporting other emergency services (e.g., opening a door for the ambulance).

Ambulance The ambulance service has two responsibilities: responding to medical emergencies and supporting other emergency services (e.g., be on stand by in case of a injury during the handling of an emergency).

2.2.3. Base of Operation

Each emergency service has multiple bases of operation on the map. Each map has a number of vertices that are designated as bases of operation for the emergency services. For a valid asset configuration, there needs to be at least one base of operation for each emergency service on the map. In addition, each base of operation needs to have at least one vehicle of the corresponding emergency service assigned to it. For the fire department these are fire stations, for the police these are police stations, and for ambulances these are hospitals. The bases also determine how many staff members are available at the location. The properties of these bases of operation are specified in the configuration files.

Base of operation attributes that are specified in the config file:

- `id` ($\texttt{id} \geq 0$) The unique identifier of the base.
- `baseType` The type of the base, which is one of `FIRE_STATION`, `POLICE_STATION`, or `HOSPITAL`.
- `staff` ($\texttt{staff} > 0$) The number of staff members that are stationed at the base.
- `doctors` ($\texttt{doctors} \geq 0$) The number of emergency doctors that are stationed at the base. This property can only be present if the type of the base is `HOSPITAL`.
- `dogs` ($\texttt{dogs} \geq 0$) The number of K9 police dogs that are stationed at the base. This property can only be present if the type of the base is `POLICE_STATION`.
- `location` The vertex the base is located on. At each location (vertex) there can only be one base and the vertex has to exist in the given county.

Figure 1: Configurable Attributes of Bases of Operation

2.2.4. Vehicles and Staff

To handle their previously described responsibilities, the emergency services need staff and equipment (e.g., vehicles). To be able to leave their base of operation, the vehicles need to be staffed with the required number of staff members and special staff members. We describe the arsenal of each service in the following sections:

Police

The police force in our simulation has two different types of officers: regular police officers and K9 (police dogs) as configured in the police station. In addition to these the police department has different types of vehicles: regular police cars, motorcycles, and K9 police cars (which can carry exactly one K9). The number of each vehicle present in the simulation is specified in the configuration files. Police cars now have a criminal capacity. Each emergency where criminals are to be arrested (see Table 1) now specifies how many criminals will be arrested during the handling of the emergency. In case a police car has no more capacity, it needs to go back to the base it is assigned to and stay there for 2 ticks, which resets the capacity. During these two ticks, the police car cannot be assigned. In case a police car returns to its base and has at least one criminal, it also waits for 2 ticks and resets the capacity.

Fire Department

The fire department has no special additional staff, only firefighters (i.e., we do not simulate hierarchies and special roles) as configured in the fire station. In addition to that there are different types of vehicles that the fire department has at its disposal: a fire truck carrying water for fire emergencies, a fire truck with technical equipment for technical emergencies (e.g., car crashes), a fire truck with a ladder, and a staff transport vehicle. These vehicles are specified in the configuration files. Fire trucks that carry water now have a maximum water-carry capacity. Each emergency, where water is needed (see Table 1) does now also specify how much water is needed for the handling of the emergency. This amount of water is also used up. If a fire truck with water has no more water, it needs to return to its specified base. Once there the vehicle refills its water with a rate of 300/tick. This also happens every time, such a fire truck is at its base and the water it is currently carrying is not at its specified capacity. So an empty truck with a carry capacity of 1200 would need 4 ticks to be ready for its next emergency. During the refilling, the truck cannot be assigned.

Ambulance

The ambulance services have two different types of staff: Emergency Medical Technicians (EMTs) and Emergency Doctors as configured in the hospital. In addition to them, there are two different types of vehicles: an ambulance and an emergency doctor car (which can carry only one doctor at a time). The ambulance responds to almost all calls (see Table 1)

Police vehicle attributes that are specified in the config file:

- `id` ($\texttt{id} \geq 0$) The unique identifier of the vehicle.
- `baseID` ($\texttt{baseID} \geq 0$) The identifier of the base of operation for this vehicle which has to exist and be a police station. Also the base needs to have enough staff to fully staff any vehicle stationed at this base.
- `vehicleType` The type of the vehicle. In case of the police these types can be: `POLICE_CAR`, `K9_POLICE_CAR`, or `POLICE_MOTORCYCLE`.
- `staffCapacity` ($0 < \texttt{staffCapacity} \leq 12$) The number of staff members that can be seated in the vehicle. Please note that a police dog can only be present in a vehicle of type `K9_POLICE_CAR` and does not count towards the capacity.
- `criminalCapacity` ($0 < \texttt{criminalCapacity} \leq 4$) The number of arrested criminals that can be transported in a vehicle. This capacity can only be present for vehicles of type `POLICE_CAR`.
- `vehicleHeight` ($1 \leq \texttt{vehicleHeight} \leq 5$) The height of the vehicle. This is needed for some height restricted roads on the map.

Figure 2: Configurable Attributes of Police Vehicles

Fire Department vehicle attributes that are specified in the config file:

- `id` ($\texttt{id} \geq 0$) The unique identifier of the vehicle.
- `baseID` ($\texttt{baseID} \geq 0$) The identifier of the base of operation for this vehicle which has to exist and be a fire station. Also the base needs to have enough staff to fully staff any vehicle.
- `vehicleType` The type of the vehicle. In case of the fire department these types can be: `FIRE_TRUCK_WATER`, `FIRE_TRUCK_TECHNICAL`, `FIRE_TRUCK_LADDER`, or `FIREFIGHTER_TRANSPORTER`.
- `staffCapacity` ($0 < \texttt{staffCapacity} \leq 12$) The number of staff members that can be seated in the vehicle.
- `waterCapacity` The amount of water a vehicle of type `FIRE_TRUCK_WATER` can carry. The vehicles can either carry 600l, 1200l, or 2400l. This property cannot be present for vehicles of any other type.
- `ladderLength` ($30 \leq \texttt{ladderLength} \leq 70$) The maximum length of the ladder on a vehicle of type `FIRE_TRUCK_LADDER`. This property must not be present for any other vehicle type.
- `vehicleHeight` ($1 \leq \texttt{vehicleHeight} \leq 5$) The height of the vehicle. This is needed for some height restricted roads on the map.

Figure 3: Configurable Attributes of Fire Department Vehicles

and the emergency doctor is only needed for severe medical emergencies. The vehicles are specified in the configuration files. Each ambulance now has room for one patient. If an emergency specifies that a patient has to be handled (see Table 1), the ambulance has to pick up the patient and, after the emergency handling is done, bring them to their base, which is a hospital. Once there, if the ambulance is carrying a patient, the ambulance has to wait for one tick before it can be used again.

Ambulance Service vehicle attributes that are specified in the config file:

- `id` $(\texttt{id} \geq 0)$ The unique identifier of the vehicle.
- `baseID` $(\texttt{baseID} \geq 0)$ The identifier of the base of operation for this vehicle which has to exist and be a hospital. Also the base needs to have enough staff to fully staff any vehicle.
- `vehicleType` The type of the vehicle. In case of the ambulance service these types can be: `AMBULANCE` or `EMERGENCY_DOCTOR_CAR`.
- `staffCapacity` $(0 < \texttt{staffCapacity} \leq 12)$ The number of staff members that can be seated in the vehicle. Please note that an emergency doctor can only be present in a vehicle of type `EMERGENCY_DOCTOR_CAR` and does not count towards the capacity.
- `vehicleHeight` $(1 \leq \texttt{vehicleHeight} \leq 5)$ The height of the vehicle. This is needed for some height restricted roads on the map.

Figure 4: Configurable Attributes of Ambulance Vehicles

2.2.5. The Map

The simulation takes place on a map of a county. This county consists of multiple villages with roads connecting them. This is represented by a weighted graph structure. The weight of an edge determines how long it takes to travel from one end of the road to the other end of the road. In our simulation you can travel edges with a total weight of 10 within one game tick. If you need to travel a route with a total weight of 20, that takes 2 game ticks. If the number of game ticks you need for your route is not a multiple of 10, we round up to the next multiple of 10 (i.e., a route with total length of 12 still takes 2 game ticks). The vertices of the graph are either crossroads, where two roads meet, or the end point of a road. Additionally, a road has multiple properties:

Primary Type: the primary type of a road. The possible types are one of:

- **mainStreet:** a main traveling street in a village.
- **sideStreet:** a less important street within a village.
- **countyRoad:** a road connecting two villages.

Secondary Type: the secondary type of a road. The possible types are one of:

- **oneWayStreet** a street that can only be driven on in one direction (from source to target).
- **tunnel:** a street that has a tunnel on it. This affects the maximum height of vehicles that can travel on this street.
- **none:** if the street does not have a secondary type.

Village: the name of the village a road is in. If the road is a county road, this is the name of the county.

Name: the name of the road. This is given for county roads as well as village roads.

Weight: the weight of the street (i.e., the weight of the edge).

Height: the maximum height of vehicles traveling on this road (county and village).

You have to validate the following properties of the map:

1. All IDs of vertices are unique and the smallest possible ID is 0.
2. Each vertex is connected to at least one other vertex.
3. The road name is unique within a village.
4. There are no edges from one vertex to itself.
5. There is at most one edge between two vertices.
6. All edges connect two existing vertices.
7. All edges connected to the same vertex belong to the same village or are a *countyRoad*.

8. Each village has at least one road with type *mainStreet*.
9. There is at least one *sideStreet* on the map.
10. The weight of a road must be greater than 0.
11. The height of a road is at least 1.
12. The height of a tunnel is at most 3.
13. No village name is equal to a county name.

Graph Property You can expect (i.e., we make sure) that we only provide scenarios for which the graph is strongly connected (and stays strongly connected regardless of the events we provide in our simulation scenarios) in case it is valid regarding the given constraints. More specifically, this means that there is a (shortest) path between any two vertices (in any direction) in the graph that can be traveled by any vehicle in this simulation scenario (e.g., we do not introduce a tunnel with height limit 3 that makes a vertex unreachable for a vehicle with height 4). Note that the behavior of the simulation is not defined, if it is, at any point during a scenario, the case that there is no path between two vertices that can be traveled by any vehicle in this simulation scenario.

2.2.6. Emergencies

There are four base categories of emergencies: *fire* (fire department), *accident* (fire department), *crime* (police), and *medical* (ambulance services). Each of these emergencies is mainly associated with one emergency service but depending on the severity of the emergency, other services might be required. Once an emergency call is being handled, you need to give the responsibility to the closest (regarding to the shortest path on the map considering the weights on the edges) base to the emergency of the emergency service that is responsible for this type of emergency (i.e., a fire emergency will be handled by the closest fire station). If there is a tie in distance between bases, you choose the base with the lowest Id. The emergencies have three different severity levels where an emergency level of *one* is the least severe and an emergency level of *three* is the highest. To determine how many and what types of forces (i.e., vehicles and staff) to send to an emergency, you have to use the information in Table 1. You have to make sure that all asset requirements are strictly fulfilled. This means that you must not send more assets than specified in the table but you also must not send less assets than specified in the table. Note that the number of vehicles you send have to be able to carry at least the amount of water and number of criminals that is specified for the emergencies, but you can send assets, according to the rule on which assets to send, that have more capacity than needed. During the handling of the emergency, the assets are used in the order of their id starting with the lowest id. More specifically, this means that if you send two fire trucks with water, the first one is used until it is empty and then the second one is used and if you send two police cars, the first one is used until it is full and then the second one is used.

Emergency attributes that are specified in the config file:

- `id` ($\texttt{id} \geq 0$) The unique identifier of the emergency.
- `tick` ($\texttt{tick} > 0$) The game tick in which the emergency call reaches the EMCC.
- `village` The village the emergency is happening in. If the emergency happens on a county road, the village name is the name of the county. This village/county needs to exist on the map
- `roadName` The name of the road the emergency happens on. This road has to exist in the specified village or in the specified county if the road is a county road.
- `emergencyType` The type of emergency which is either `FIRE`, `ACCIDENT`, `CRIME`, or `MEDICAL`.
- `severity` ($1 \leq \texttt{severity} \leq 3$) The severity level of the emergency.
- `handleTime` ($\texttt{handleTime} > 0$) The time in game ticks it takes to handle the emergency once all required assets are at the emergency.
- `maxDuration` ($\texttt{maxDuration} > \texttt{handleTime}$) The maximum time in game ticks the emergency can take to be handled starting when the call reaches the EMCC. If the approach to and the handle time of the emergency are larger than this, then the emergency call is considered failed.

Figure 5: Configurable Attributes of Emergencies

Table 1: Mapping of Emergency Type, Severity, and Assets needed

	Severity 1	Severity 2	Severity 3
Fire	2 Firetrucks with water 1200l water	4 Firetrucks with water 3000l water 1 Firetruck 30m ladder 1 Firefighter Transporter 1 Ambulance 1 Patient	6 Firetrucks with water 5400l water 2 Firetrucks 40m ladder 2 Firefighter Transporters 2 Ambulances 2 Patients 1 Emergency Doctor
Accident	1 technical Firetruck	2 technical Firetrucks 1 Police Motorcycle 1 Police Car 1 Ambulance 1 Patient	4 technical Firetrucks 2 Police Motorcycles 4 Police Cars 3 Ambulances 2 Patients 1 Emergency Doctor
Crime	1 Police Car 1 Criminal	4 Police Cars 4 Criminals 1 K9 1 Ambulance	6 Police Cars 8 Criminals 2 Police Motorcycles 2 K9 2 Ambulances 1 Patient 1 Firefighter Transporter
Medical	1 Ambulance	2 Ambulances 2 Patients 1 Emergency Doctor	5 Ambulances 5 Patients 2 Emergency Doctors 2 technical Firetrucks

2.2.7. Events

In addition to the emergency calls, there are certain events that can happen during the simulation. These events are:

Rush Hour: A rush hour can affect certain street types (e.g., county roads). For as long as this event is active, the weight of all roads of the specified types is changed by the factor. For example you have a road with weight 2 and the factor is 3 then the weight of the road changes to 6 for the duration of the event and back to 2 once the event is over.

Traffic Jam: A traffic jam is essentially the same as the rush hour event except that it only affects one specified road. For example, road A in village 1 gets a factor on its weight for the duration of the event. The road is specified by vertices (i.e., source and target).

Construction Site: A construction site either changes a road to a one-way street (direction source to target as specified in the event) and applies a factor on the weight or just applies a factor on the weight. A construction site can only change a road to a one-way

street if the road is not already a one-way street, however, also in this case the factor is still applied.

Road Closure: A road closure closes a given road for the duration. In case an emergency happens on a currently closed road, the road immediately opens. The remaining duration of the road closure has to be resumed once this emergency is over.

Vehicle Unavailable: This is the notification of the emergency services that certain vehicles are not available for the duration and can not be used for emergencies. If the resource is currently not at its base, it becomes unavailable for the duration once it returns to the base. The vehicle can finish the emergency it is currently assigned to. Once this emergency is over, the vehicle has to return to its base immediately. The specified vehicle cannot be redirected to another emergency. Before the vehicle becomes unavailable it has to refill its water if it is a fire truck with water or has to handle its patients/criminals if it is an ambulance or police car. In addition, an asset that is currently refilling water or handling patients/criminals at the base becomes only unavailable once it is done with its current task for the duration specified by the event.

All necessary information for the events are specified in the config files.

Event attributes that are specified in the config file:

- `id` ($\texttt{id} \geq 0$) The unique identifier of the event.
- `type` The type of event which is either `RUSH_HOUR`, `TRAFFIC_JAM`, `CONSTRUCTION_SITE`, `ROAD_CLOSURE`, or `VEHICLE_UNAVAILABLE`.
- `tick` ($\texttt{tick} \geq 0$) The game tick in which the event starts.
- `duration` ($\texttt{duration} \geq 1$) How much time in game ticks the event lasts.
- `roadTypes` The types of roads (can be multiple different types, but only primary types) that are affected by the current event. This can only be present for events of type `RUSH_HOUR`.
- `factor` ($\texttt{factor} \geq 1$) The factor by which the length of a road is changed during events of type `RUSH_HOUR`, `TRAFFIC_JAM` or `CONSTRUCTION_SITE`. The property can not be present for any other type of event.
- `oneWayStreet` Whether or not a road becomes a one-way street during an event of type `CONSTRUCTION_SITE`. The property can not be present for any other type of event.
- `source` ($\texttt{source} \geq 0$) The source vertex of a road. This is required for events of type `TRAFFIC_JAM`, `CONSTRUCTION_SITE` and `ROAD_CLOSURE` (i.e., events that are confined to a single edge/road). This vertex has to exist on the current map and the property can not be present for events of any other type.
- `target` ($\texttt{target} \geq 0$) The target vertex of a road. This is required for events of type `TRAFFIC_JAM`, `CONSTRUCTION_SITE` and `ROAD_CLOSURE` (i.e., events that are confined to a single edge/road). This vertex has to exist on the current map and the property can not be present for events of any other type. Furthermore, there needs to be an edge between `source` and `target`.
- `vehicleID` ($\texttt{vehicleID} \geq 0$) The id of a vehicle that becomes unavailable during a `VEHICLE_UNAVAILABLE` event. The vehicle must exist and the property can not be present for events of any other type.

Figure 6: Configurable Attributes of Events

2.3. Simulation Details

In this section, we will describe the different phases of the simulation, and the behavior of the emergency response units during the handling of an emergency call.

Please note:

In the following sections, we will include examples for the log outputs you have to provide during the simulation. Terms starting with a dollar sign (i.e., `$`) are placeholder variables for actual values. For example, `$tickNumber` is a placeholder variable for the number of the current tick. Every log output has to exactly match the format specified in the following sections (including the symbols, whitespace characters, and all the words in the log output).

2.3.1. Initialization

Before the simulation can start, you will have to parse the configuration files for the county, the bases, the emergency response units, and resources, as well as the emergency calls and events. You have to parse the configuration files in the following order:

1) The configuration file for the county (i.e., the map)
2) The configuration file for the emergency centrals and the emergency response units
3) The configuration file for the emergency calls and events

County Configuration The configuration file of the county will be provided in a special, simplified DOT[1] format, you can find more details in Section 2.2.5 and the grammar for it in Section A.1. In addition to parsing the configuration file, you also have to verify that the configuration file is valid according to the constraints provided in Section 2.2.5.

Asset Configuration The configuration file for the emergency centrals and the emergency response units (i.e., the *asset* configurations) will be provided in a JSON format. You can find more details for the JSON format of the asset configurations in Section 2.2. Similar to the county configuration, you also have to verify that the configuration file is valid according to the constraints provided in Section 2.2.

[1] https://graphviz.org/doc/info/lang.html

Simulation Scenario Configuration The third file that you will have to parse is the simulation scenario configuration file for the emergency calls and events. This file will also be provided in a JSON format and you can find more details for its format in Section 2.2. As for the other configuration files, you also have to verify that the configuration file is valid according to the constraints regarding emergencies and events, provided in Section 2.2.6 and 2.2.7 respectively.

After successfully validating a configuration file, you have to output to the output handle (which is provided as command line argument when starting the simulation) a log containing the type of the log (i.e., `Initialization Info`) and the name of the configuration file (where `$filename` is a placeholder variable for the name of the configuration file).

```
Initialization Info: $filename successfully parsed and validated
```

In case the configuration file is invalid, you have to output a log containing the type of the log (i.e., `Initialization Info`) and the name of the configuration file.

```
Initialization Info: $filename invalid
```

After successfully parsing and validating all configuration files, the simulation starts in tick 0. In case a configuration file is invalid, the simulation will terminate immediately after providing the specified output for the invalid configuration file.

In the following, we will describe the different phases of the simulation during a single tick.

2.3.2. Simulation Ticks

The simulation will be played in ticks, where each tick represents a single time step in the simulation. The simulation will start in tick 0 and will terminate after the last tick of the simulation scenario has been processed. The simulation scenario ends either after the number of ticks specified as command line argument (the maximum number of ticks to simulate), or the tick in which the last emergency call has been processed (i.e., the last emergency call failed or has been resolved successfully (which is defined later)), whichever comes first. Before the first tick of the simulation, you have to output the following log to the output handle:

```
Simulation starts
```

Each tick can be split into different phases, the first phase in each round is the *emergency phase*, in which you detect all incoming emergencies for this round and distribute them to the appropriate base. In the second phase, the *planning phase*, you will have to compute how to handle the emergencies in a way that you distribute your available assets in the best possible way, to resolve all new emergencies efficiently. In the third, and last, phase each round, the *update phase*, you will be updating the status of all ongoing emergencies in your simulation.

Before the start of each *emergency phase* (i.e., whenever the current game tick increases), you have to output a log containing the type of the log (i.e., `Simulation Tick`)the number of the current tick.

```
Simulation Tick: $tickNumber
```

In the following, we will describe the different phases of the simulation in more detail.

Emergency Phase

In the *emergency phase* you will first check which emergencies are scheduled for this particular round. All scheduled emergencies for this round have to be distributed to the responsible base. The responsible base is the base that has the shortest path (see Section 3.1) to the emergency (ignoring height restrictions) and is responsible for this type of emergency, if there are multiple possible bases you have to choose the base with the lowest id. The shortest path can be calculated using the Dijkstra algorithm[2]. For each individual emergency during this round, you have to output a log containing the type of the log (i.e., `Emergency Assignment`), the emergency id and the base id in order of the emergency id, starting from the lowest id to the highest.

```
Emergency Assignment: $emergencyId assigned to $baseId
```

Planning Phase

During the planning phase, you will have to compute the best possible way to handle all the emergencies that emerged during the emergency phase. Each base will have to compute the best possible way to handle the emergencies that emerged in its area of responsibility. The order to handle the emergencies in a tick is determined by the severity of the emergency, where the most severe emergency has to be handled first. In case of a tie, you will have to handle the tied emergencies starting from the emergency with the lowest id to the highest id.

Allocating Assets First, you will have to compute the number of assets you have to allocate from your base to the current emergency to be handled. To decide which assets you have to allocate, you always have to allocate the maximum possible number of assets from your base to handle the emergency. In case there are multiple possible sets of assets you have to choose the set of assets containing the assets with the lowest individual ids. To allocate assets to an emergency, you will have to compute the shortest path from your base to the source or target vertex of the edge the emergency is on (you have to pick the vertex that is closer to your base), for each individual asset according to its height restrictions, to know how long the assets will take to arrive there. For more information on how to get a unique shortest path in case there is more than one possible shortest path, see Section 3.1.

[2] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

```
Asset Allocation: $assetId allocated to $emergencyId; $ticksToArrive
ticks to arrive.
```

If there are not enough assets available at the base to handle the emergency (e.g., the emergency needs asset from another emergency service or you do not have enough assets available in your base), ~~you will have to request the exceeding assets from other bases in the county, starting from the base closest to your base.~~ first you have to check whether there are assets (of the types still needed for the current emergency) belonging to your base on their way to an emergency with lower severity than the current emergency or on their way back to the base. If that is the case, you need to reallocate them to the current emergency (if they are able to arrive at the emergency in time) and they will need to update their route. In case you reallocate assets that are on their way to an emergency, the base that is responsible for that emergency will have to re-handle it in the ~~next~~ current tick. Whenever you re-allocate assets (starting with the asset with the smallest id) you have to output a log containing the type of the log (i.e., Asset Reallocation), the id of the asset and the id of the emergency the asset is reallocated to.

```
Asset Reallocation: $assetId reallocated to $emergencyId.
```

Only after you have reallocated such assets, you can send out requests to other bases. To do so, you will have to compute the shortest path from your base to the bases of the emergency services that you need assets from (i.e., police cars can only be requested at a police base). In case of a tie, you will have to choose the base with the lowest id. For each request for assets (requests are sent in the order of the id of the bases you send requests to (starting from lowest to highest id)), you have to output a log containing the type of the log (i.e., `Asset Request`), the emergency id, the base id of the base you request the assets from, and a request id, which is a unique identifier (starting from 1) and increasing for each request made throughout the simulation.

```
Asset Request: $requestId sent to $baseId for $emergencyId.
```

You have to send all requests of an emergency call before starting the plan for the next emergency call. In case your assets cannot arrive in time at the emergency, you will send no requests to other bases and try to handle the emergency in the next tick again.

Request Phase

After you have addressed all emergencies of this tick, you will have to process all requests for assets starting from the request with the lowest id to the highest id. For each request, you have to compute the number of assets you have to allocate to the emergency for which the request has been made. To allocate assets to an emergency, you will have to compute the shortest path from your base to the emergency and output the log accordingly (as described in the previous section). For requests you cannot reallocate assets.

In case there are not enough assets available at the base to handle the request for the emergency or the assets can not arrive in time, you will have to request the exceeding assets from the base that is the next closest base to the base the emergency was initially assigned to and output a log accordingly (as described in the previous section). In case there is no base left to request assets from, you will have to output a log containing the type of the log (i.e., `Request Failed`), and the emergency id for which the request failed and you will have to try to handle the emergency in the next tick again (in the base the emergency was initially assigned to).

```
Request Failed: $emergencyId failed.
```

For each request for assets, you have to output a log as was described in the previous section for asset request in the order you handle the requests.

Update Phase

The last phase of each tick is the *update phase*, in which you will have to update the status of all allocated assets, staff members, ongoing emergencies, and events.

First, you will have to update the status of all allocated assets. Assets that were allocated to an emergency in this tick, will have to drive to the emergency for a certain amount of ticks (depending on the distance between the base and the emergency). The time it takes for an asset to drive from the base to the emergency starts in the next tick after the asset has been allocated to the emergency. For each asset arriving at an emergency or back at the base in this tick, you have to output a log containing the type of the log (i.e., `Asset Arrival`), the asset id, the vertex id of the vertex the asset arrived at. The order in which the assets arrive at the emergency is determined by the asset id (starting from the asset with the lowest id to the highest id).

```
Asset Arrival: $assetId arrived at $vertexId.
```

For each emergency for which all necessary assets have arrived at the emergency, you have to output a log containing the type of the log (i.e., `Emergency Handling Start`) and the emergency id (in the order of the emergency id starting from the lowest id to the highest id) .

```
Emergency Handling Start: $emergencyId handling started.
```

For each emergency that has been resolved in this tick, i.e., an emergency that has been resolved by the assets working on the emergency for the predefined amount of ticks, you have to output a log containing the type of the log (i.e., `Emergency Resolved`) and the emergency id(in the order of the emergency id starting from the lowest id to the highest id) . After the emergency got handled successfully the allocated assets can return to the bases they were allocated from on the shortest path starting from the emergency (i.e., the node the vehicle arrived at) going back to the respective bases. This means that the assets will

have to calculate the shortest path from the emergency to the base they were allocated from immediately after the emergency has been resolved.

```
Emergency Resolved: $emergencyId resolved.
```

For each emergency that cannot be resolved anymore (i.e., the ticks needed handling the emergency exceeds the number of ticks left to handle the emergency), you have to output a log containing the type of the log (i.e., `Emergency Failed`) and the emergency id(in the order of the emergency id starting from the lowest id to the highest id) .

```
Emergency Failed: $emergencyId failed.
```

Attention!

Events that end in the current tick or start in the current tick can have influence on the shortest path of assets that are currently on the way to or from an emergency. In this case, you need to reroute the affected assets. The only exception to this is the *Vehicle unavailable* event as this does not have an influence on the map.

After all emergencies have been updated, you will have to handle all events that are ending and scheduled for this tick. First, for each event that ends in this tick, you have to output a log containing the type of the log (i.e., `Event Ended`) and the event id in the order of the event id starting from the lowest id to the highest id.

```
Event Ended: $eventId ended.
```

For each event that is scheduled for this tick, you have to output a log containing the type of the log (i.e., `Event Triggered`) and the event id. Events can only start if they affect at least one road, which is not yet affected by any other event for the next tick. In case there are multiple roads affected, and at least one road does not have an active event at that moment, the event is triggered, and the counter for the event starts. You have to make sure, that at any time a road is not affected by an event anymore, but there is an event active that could affect this road, the road will be affected immediately by that event until the event ends. That means, that the event takes priority over any event that might be scheduled for the same tick on that road. Conflicts between scheduled events in the same tick are resolved in the order of the event id, the event with the lowest id will affect the road.

```
Event Triggered: $eventId triggered.
```

In case there is no road that can be affected in this tick, the event will be rescheduled for the next tick (and not be triggered).

Rerouting Assets In case there were any events ending or starting in the current tick, you have to recalculate shortest paths for assets, starting from the asset with the lowest id to the highest id, after all events were handled. The starting points of your calculations are the current positions of the assets. To keep track of the current positions of the assets, you need to know the path they are currently taking and how far along they are on that path. Note that the position of the assets has already been updated in this tick during the asset update phase. The position of your assets can be anywhere on an edge of the map. For example an asset traveling from vertex 1 to vertex 7 (see Figure 7) would be halfway through the edge between vertices 6 and 7 after one tick. In case you reroute assets in the current tick you need to output a log containing the type (i.e., Assets Rerouted) and the number of assets that were rerouted in this tick.

```
Assets Rerouted: $numberReroutedAssets
```

Assets that are rerouted in the current tick will have to follow the new shortest path from the next tick on and will not change their destination in case the new shortest path will take too long to arrive at their destination in time. More specifically, rerouting never changes the destination (only assigning and re-assigning does) of an asset. An asset is considered to be rerouted if the shortest path it is currently following changes with regard to the set of edges it is using to reach its destination or the total weight of the route changes.

Please note:

~~In case an asset is on an edge that is affected by an event (starting or ending) in the current tick, the asset has to ignore the effects of the event until it leaves the edge (i.e., this does not affect the shortest path of that asset). Assets that enter the edge in the following ticks have to take the new modalities of the edge into consideration.~~ An asset can never end up on a vertex unless that vertex is the home base of the asset. Upon reaching the end of an edge, the asset leaves its current edge and, unless it is arriving at its home base, immediately enters the next edge on its current route, this might be the same edge if this is the shortest path.

In case the asset is on an edge that is affected by an event (starting or ending) in the current tick, the path of that asset has to be recalculated, starting from both nodes of the edge (in case the edge is a one-way street only from the node the asset is allowed to drive to). You have to add the remaining weight of the edge the asset needs to travel to reach the respective node to the weight of the calculated paths. The asset is then considered rerouted if the criteria above apply to the new route.

2.3.3. Final Evaluation

After the last tick of the simulation has been processed, you will have to output a log containing the type of the log (i.e., `Simulation End`).

```
Simulation End
```

After the simulation has ended, you will have to output logs for the final statistics for the simulation, containing the type of the log (i.e., `Simulation Statistics`). First, you have to output a log containing the number of rerouted assets.

```
Simulation Statistics: $numberReroutedAssets assets rerouted.
```

~~First,~~ Second, you have to output a log containing the number of received emergencies.

```
Simulation Statistics: $numberReceivedEmergencies received emergencies.
```

~~Second,~~ Third, you have to output a log containing the number of still ongoing emergencies.

```
Simulation Statistics: $numberOngoingEmergencies ongoing emergencies.
```

Then, you have to output a log containing the number of failed emergencies.

```
Simulation Statistics: $numberFailedEmergencies failed emergencies.
```

you have to output a log containing the number of resolved emergencies.

```
Simulation Statistics: $numberResolvedEmergencies resolved emergencies.
```

3. Technical Details

In this chapter, we explain more technical details—from the build script, to code analysis tools, and, finally, to technical implementation details. Here, we also explain the framework that is provided by us and must be used by you.

3.1. Shortest Path Algorithm

Computing the shortest path between two vertices in a graph can be done using the *Dijkstra*[1] algorithm. In case there are multiple shortest paths, you have to choose the path that follows, starting from the source vertex, at every step of the path, the vertex with the lowest id. In Figure 7, the shortest path from vertex 2 to vertex 7 is either 2-4-3-7 or 2-4-1-6-7 (as both have a cost of 60). Following the definition above, in our simulation there is only one shortest path (i.e., 2-4-1-6-7), as the vertex id 1 is smaller than vertex id 3.

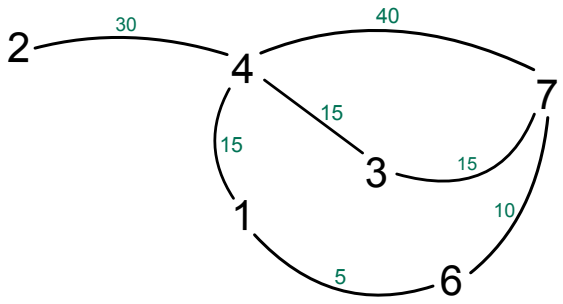

Figure 7: Example graph

3.2. Command-Line Interface

The simulator is started with these command line parameters:

`--map` Path to the map. (always required) String

`--assets` Path to the file with assets. (always required) String

[1] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

`--scenario` Path to the scenario file. (always required) String

`--ticks` Maximum allowed number of simulation ticks. Int

`--out` Path to output file. Uses 'stdout' by default. String

`--help` Usage info

3.3. Build Script

We provide you with a build script. A build script is responsible for managing the dependencies of a software project and creating a finished executable program from the project files. As a build tool, we use *gradle*[2].

The build script can be found in the `build.gradle.kts` file. Changes to your build script will be rolled back from the test server for execution. You can (and should) build the project yourself by either running the `./gradlew build` command or running the *gradle* task `build` directly in an IDE (e.g. *IntelliJ*). We use *Kotlin 1.8.21* with the *Java JDK Version 17*.

In the build script there are dependencies on various libraries. We recommend that you look at these libraries and consider whether you can use them. Other libraries that are not entered in the build script must not be used. For security reasons, we do not allow reflection, network connections, or other connections to the outside world. Also, with the exception of loading configuration files and creating an output file, you may not read, create, or modify files in the file system. For loading configuration files and writing an output file, you may only access the paths given in the command line parameters and the *resources* folder for reading the JSON schema.

3.4. Code Quality

Good code quality helps software to remain readable and it helps to avoid errors. In the SE Lab, you will learn about automated tools that help you write good and (hopefully) bug-free code. Specifically, we use the tool DETEKT[3]. You can use this tool with *gradle*.

DETEKT is a static analysis tool, which means it doesn't need to run your code to find errors. DETEKT examines the source code for certain patterns that are known to cause errors. Examples of such patterns are unreachable code (you probably forgot to call a method) or using 200 parameters in a function. You will notice that DETEKT does not allow `print|println`. For the logging functionality, you have to use a `PrintWriter` from the package `java.io`.

Furthermore, DETEKT provides style rules in order to check your code for style issues (e.g.,

[2] https://gradle.org/
[3] https://detekt.dev/docs/intro

missing documentation for public functions). This helps improve readability and maintainability of your code.

You can find a report of the analysis in the directory `build/reports/detekt`.

4. Tests

In the group phase we expect system, unit, and integration tests from you. You can find the requirements, which are demanded from your application, in the specifications from the previous chapters.

Try to achieve the highest possible statement and branch coverage in the implementation phase. This means that in your code as many statements and branches as possible should be covered by the tests. The coverage can be calculated, e.g., with *JaCoCo*[1], also *IntelliJ* has already integrated tools for this.

4.1. Unit Tests

For unit tests, use *JUnit 5*[2] and *Mockito*[3] which is integrated into the project via the `Mockito-Kotlin` wrapper. Various tutorials for using these frameworks can be found on the Internet[4]. We will provide you with a sample test at the beginning of the implementation phase.

The unit tests can be run with `./gradlew test` (or `gradlew.bat test`).

Hint for implementation

Unit tests belong in the given directory `src/test/kotlin` in the project folder.

4.2. Integration Tests

It is your job to also write integration tests to test how the modules work together. In integration tests, you access individual methods similar to unit tests. You can write and execute these tests like Unit Tests.

[1] https://www.jacoco.org/
[2] https://junit.org
[3] https://site.mockito.org
[4] https://junit.org/junit5/docs/current/user-guide/

4.3. System Tests

You also need to write system tests that sufficiently test the simulator for all functionality.

In order to test the simulator, you have to provide the configuration files for the simulation scenario you want to test. Make sure that your simulation scenario does not violate the assumption that your graph stays strongly connected as specified in Section 2.2.5 Your system tests will also run on the reference implementation, so that you can determine if your assumption about the simulation behavior is correct.

Note that your system tests are also run on faulty simulations (mutants), so your tests must find faulty implementations.

Hint for implementation

System tests belong in the given directory `src/systemtest/kotlin` in the project folder.

A. Appendix

A.1. Grammar for County Configuration

Terminals are shown in bold font (e.g., `village`, `->`, `;`, `[`, or `]`) and non-terminals are shown in italics (e.g., *map* or *attributes*). The vertical bar (i.e., |) separates alternatives.

⟨map⟩	::=	`digraph` *Id* `{` ⟨glStmtList⟩ `}`
⟨glStmtList⟩	::=	⟨vertex⟩⟨edge⟩
⟨vertex⟩	::=	*Id*`;` | *Id*`;`⟨vertex⟩
⟨edge⟩	::=	*Id* `->` *Id* `[` ⟨attributes⟩ `];` | *Id* `->` *Id* `[` ⟨attributes⟩ `];`⟨edge⟩

⟨attributes⟩	::=	`village =` *Id*`; name =` *Id*`; heightLimit =` *Id*`;`
		`weight =` *Id*`; primaryType =` ⟨primaryType⟩`;`
		`secondaryType =` ⟨secondaryType⟩`;`

⟨primaryType⟩	::=	`mainStreet` | `sideStreet` | `countyRoad`
⟨secondaryType⟩	::=	`oneWayStreet` | `tunnel` | `none`

An *Id* is either a string or a number. In case Id is a string it starts with a letter and can contain letters and underscores , allowed letters are only ASCII characters with an index that is contained in [97, 122] or [65, 90] . In case Id is a number it starts with a non-zero digit [1-9] and can only contain digits [0-9] ~~and a single dot (for decimal numbers)~~ or is the number 0 . Also, any amount of whitespace may be inserted between terminals.