Final Report, TDA593
Group 11

[redacted] Felix Kirchmann [redacted]

January 2018

Contents

1 Introduction

1.1 Social Contract

To work towards the goal of including all group members and achieving a roughly equal amount of contribution by every member, we created a social contract between us. This was also done in order to reduce disagreements as well as misunderstandings. The following agreements were made:

- Attend weekly meetings
- Do not be late
- Always keep open discussion so we don't run into any problems
- Everyone should do the task s/he is assigned to do
- Be helpful
- Be friendly with each other! If a problem arises: discuss!
- Goal and objective: We will work towards getting a 5
- Short breaks (few minutes) every half hour
- Longer breaks (10-15 minutes) every hour

1.2 Goals and objectives

During our first session together, we started discussing about our individual goals and objectives with regards to the course. We agreed rather quickly that we would strive for a 5/VG grade-wise, but more importantly that everyone wanted to learn as much as possible and would always work hard during the course.

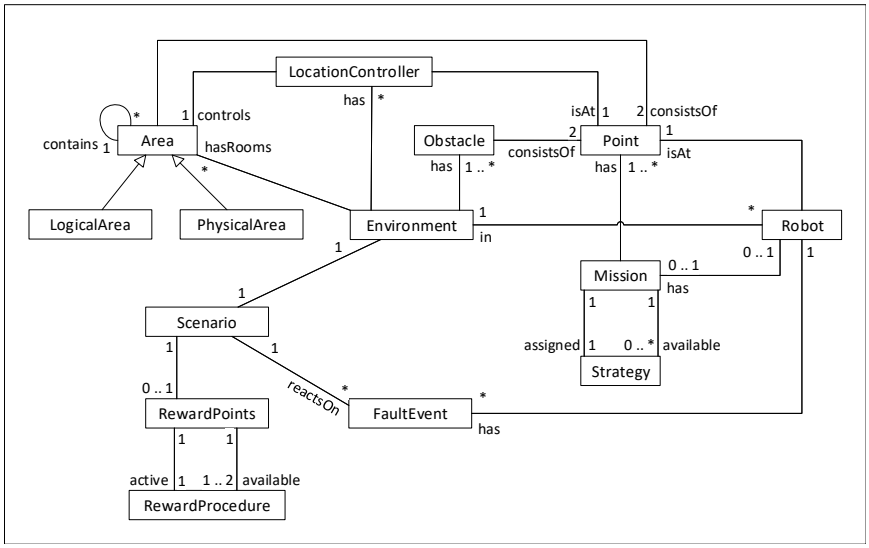
1.3 Assumptions

We made some assumptions in cases where details of the application's behaviour were not fully specified.

- An area can be nested within any type of area (e.g. a physical area can be in a logical one)
- All of the mission points have to be visited by the same robot
- All walls are strictly vertical, i.e. at a 90° angle to the floor

2 Assignment 1

2.1 Domain Model



2.2 Explanatory text

Our problem domain is centered around scenarios. A scenario describes a specific instance of the robotic domain, solving one (or more) defined problems. An example for such a scenario would be task 1.3 from assignment 3, where multiple robots work in a defined four-room environment to achieve the mission of visiting certain rooms.

Thus, each scenario is associated with one environment. The environment has any number of robots, which all have one current position within the environment.

Each robot can have up to 1 mission, which has one or multiple points, and every mission can have multiple strategies available to them, but only one of these strategies is active at any given time. A strategy is simply a specific ordering of the points from a mission.

The environment also contains any number of areas, that can be either logical areas or physical areas. Areas can be nested inside other areas. Since we assume areas to have a rectangular shape, their geometric boundaries consist of two points - the upper left and lower right coordinates of this rectangle.

To define their physical boundaries, environments have at least one obstacle, which is a wall or a boundary that goes from one point to another, vertically or horizontally. Robots cannot move through these static obstacles.

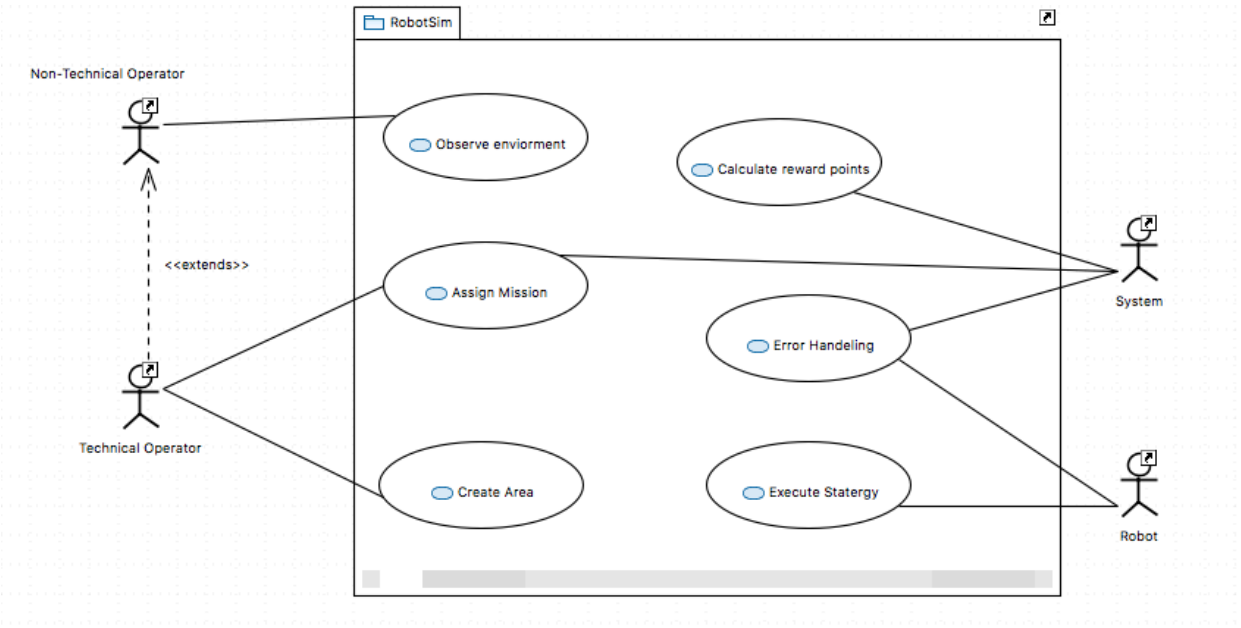
Furthermore, environments can include any number of location controllers which are based at a fixed position. Each location controller manages one area, and coordinates that no two robots are in the area simultaneously.

As an optional component, a scenario can also contain a subsystem to cal-

culate reward points based on a reward procedure. Either one or two reward procedures can be available. If two are available, only one of them is active at any given time.

Additionally, the scenario can react to faults, of which a robot can have any number.

2.3 Use Case Diagram



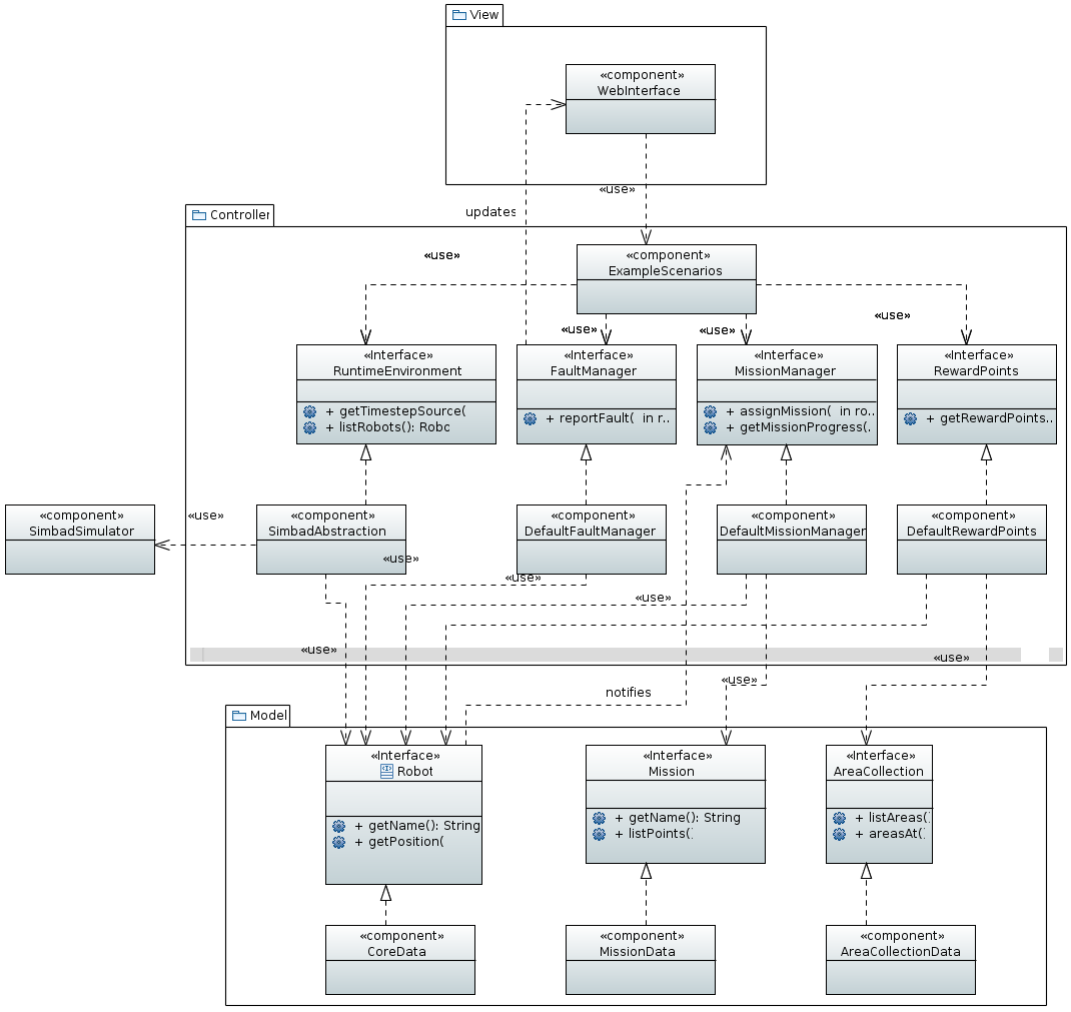
2.4 Explanatory text

- The Non-Technical operator can only observe the environment through a graphical interface.
- The Technical operator is the programmer, which can create areas and assign mission through code.
- The System handles all the logic.
- The Robot is just a robot inside the simulation, which can report errors and execute strategies(move to different points).
- Observe Environment: The non-technical operators are supposed to observe changes in the environment through a user interface. This should show where the robots are located as well as the reward points.
- Assign Mission: The technical operator can assign missions to the robots. We have to have a mission for the robot to execute otherwise the system is useless.

- Calculate Reward Points: The system calculates the reward points for the robots. It will use different strategies to calculate the reward points depending on the environment the robots are in.
- Execute Strategy: The robot executes a certain strategy over the environment. The strategy is a part of the execution of the mission.
- Error Handling: In case of an error with the robots they are to turn themselves off, and the system will react to this event.
- Create Area: This use case will be responsible for constructing the different environments the robots will be located in.

3 Assignment 2

3.1 Component Diagram



3.2 Explanatory text

- **WebInterface(c):** The web interface is the graphical interface that the users can use to observe scenarios, including missions, reward points, faults and positions of the robots within the Simbad environment.
- **ExampleScenarios(c):** ExampleScenarios is a module containing the various different scenarios (including environments, robots and missions) that can be used to run the simulations, e.g the mission from assignment 3.

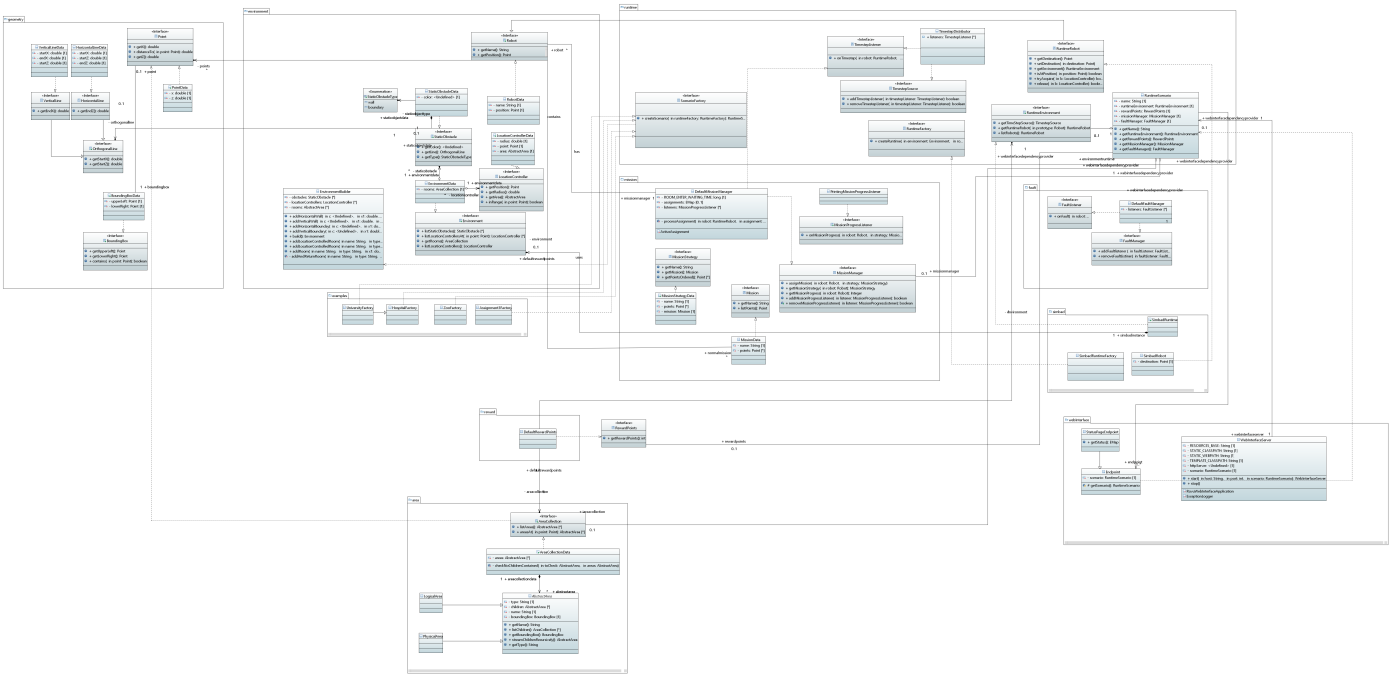
- **RuntimeEnvironment(i):** Represents all the objects being simulated in the simulator, which includes static obstacles, location controllers, rooms, robots and the TimestepSource.
- **SimbadAbstraction(c):** An abstraction layer on top of Simbad. On instantiation, it is provided with an immutable environment and a list of robots with their positions in that environment. It then uses the Simbad API to simulate the environment and exposes the results through the RuntimeEnvironment.
- **SimbadSimulator(c):** This represents the API of the Simbad simulator, as provided by the course's GitHub / Maven repository.
- **FaultManager(i):** Receives faults events and distributes them to a set of listeners.
- **DefaultFaultManager(c):** The default implementation of the FaultManager.
- **MissionManager(i):** The MissionManager contains the business logic defining robots' behaviour. It provides public methods to assign a mission (with a strategy) to a robot and to get the progress of that mission. It also uses the observer pattern to enable other classes to react to a robot reaching the next point or the end of its mission.
- **DefaultMissionManager(c):** Our default implementation of the MissionManager interface observes a TimestepSource. On each robot's timestep, the manager first checks if it has an active mission assignment for that robot. If that is the case, it sets the robot's destination to the next point in the mission. When the robot has reached this point, the manager advances its internal mission progress counter and notifies its listeners that the robot has progressed in (or completed) the mission.

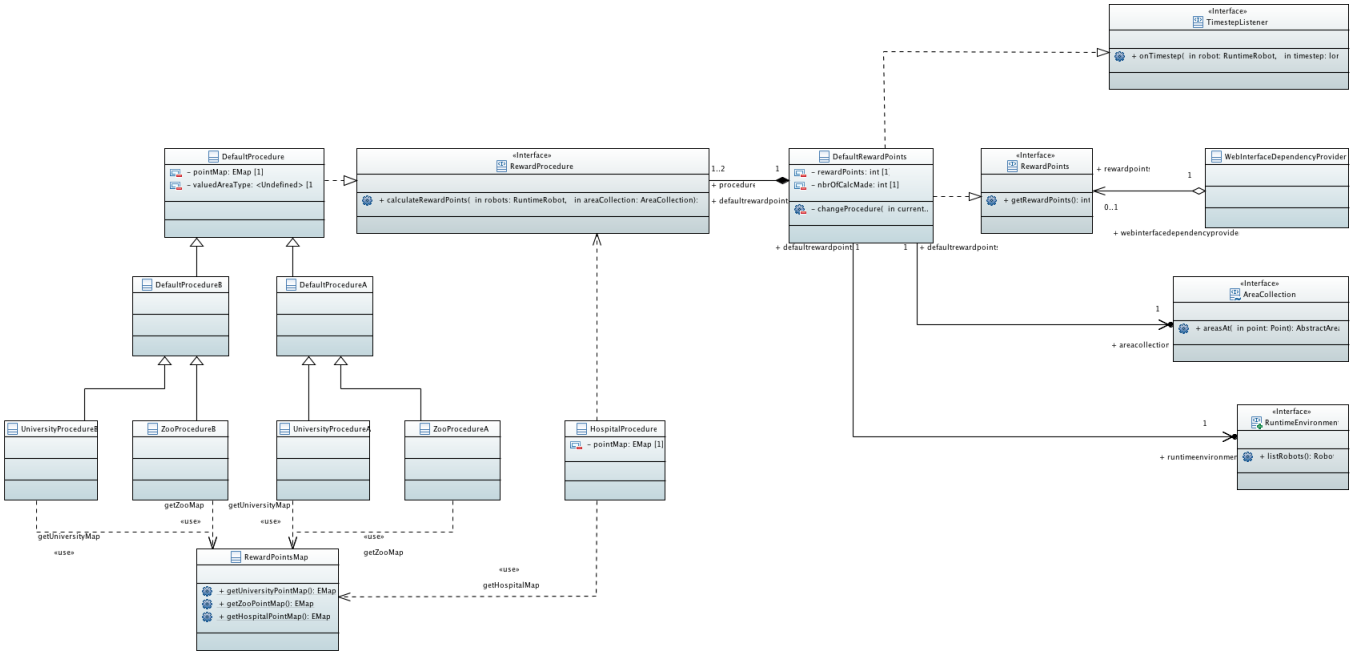
  While controlling a robot, the DefaultMissionManager also observes the rooms and location controllers at the robot's position. When the robot enters a room, it is stopped for two seconds (by setting its destination to its current position). When it enters a location-controlled area instead, the manager makes the robot acquire the location controller responsible for the area. If this does not succeed on the first try, the robot is stopped until it has successfully acquired the location controller.
- **RewardPoints(i):** Calculates the reward points for the robots using different procedures based on the areas the robots are in.
- **DefaultRewardPoints(c):** Implementation of the RewardPoints interface.
- **Robot(i):** Represents a robot with a name and a position, includes getter methods for its attributes.

- **CoreData(c):** Acts as a collection of immutable implementations for most of the framework's data interfaces (such as robot).
- **Mission(i):** A Mission represents a list of points that a robot has to reach. It also contains a string for the name of the mission.
- **MissionData(c):** Implementation of the Mission interface.
- **AreaCollection(i):** Represents a collection of areas. It has methods to return all the areas of the collection and a method that takes a point and returns all areas of the collection that contain the point.
- **AreaCollectionData(c):** Implementation of the AreaCollection interface as an immutable data class.

4 Assignment 3

4.1 Class Diagram





4.2 Description of classes, operations and relations

- **Area**
  - **AbstractArea**: Represents an area in the environment defined by a bounding box. An area can have any number of areas Areas have a name identifying them as well as a *type*, which is used when calculating reward points. An area is either a logical area or a physical area.
  - **LogicalArea**: Extends the AbstractArea class. Represents a logical area as defined by the project description.
  - **PhysicalArea**: Extends the AbstractArea class. Represents a physical area as defined by the project description.
  - **AreaCollection**: Contains a set of areas and can also return which areas from the set contain a given point. This can be useful to determine e.g. which areas are at a robot's current position.
  - **AreaCollectionData**: Implements the interface AreaCollection.
- **Environment**
  - **Environment**: Interface to get the statical objects in the Environment. Has methods for listing the location controllers, the static obstacles (walls or boundaries) as well as all of the rooms (represented by areas).
  - **EnvironmentBuilder**: The EnvironmentBuilder implements the Builder pattern to simplify the construction of certain environments. It handles adding walls, boundaries, rooms and location controllers. Furthermore, it allows its consumer to configure a room overlap, causing the sizes of added rooms to increase by the given number. Having overlapping rooms can be useful to ensure mutual exclusion such that a robot needs to acquire the location controllers of both the room it is in and the one it wants to enter when moving between rooms.
  - **EnvironmentData**: Implements Environment. When creating an EnvironmentData object it checks that the components of the environment (static obstacles, rooms and location controllers) are present (i.e. not null).
  - **LocationController**: Supplies methods for getting the location and radius of a LocationController, as well as which area it controls. Also has a method for checking if another Point is in range of the LocationController.
  - **LocationControllerData**: Implements LocationController.
  - **Robot**: Contains methods to be able to access the name and the starting position of a robot.
  - **RobotData**: Implements Robot.

  - **StaticObstacle**: Interface which supplies methods for getting the color, line, and type of an obstacle.
  - **StaticObstacleData**: Implements StaticObstacle.
  - **StaticObstacleType**: Is an enum which is used by StaticObstacle to identify what type the obstacle is, i.e. a wall or a boundary.

- **Fault**
  - **FaultListener**: Classes implementing this interface can receive information about fault events via the onFault method.
  - **FaultManager**: Maintains a list of FaultListeners, to which it distributes fault events.
  - **DefaultFaultManager**: Implements both FaultManager and FaultListener. It will respond to calls of its onFault method by forwarding the fault information to all registered listeners.

- **Geometry**
  - **BoundingBox**: Represents a box, which has two methods, one for getting the upper left and one for getting the lower right point of the box.
  - **BoundingBoxData**: Implements the BoundingBox interface. Upon construction, it checks whether the lower right point is actually lower and to the right of the upper left point.
  - **OrthogonalLine**: Interface for representing a line. Has methods for getting the starting X and Z coordinates.
  - **HorizontalLine**: Extends OrthogonalLine. Has a method for getting the Z endpoint.
  - **HorizontalLineData**: Implements HorizontalLine.
  - **VerticalLine**: Extends OrthogonalLine. Has a method for getting the X endpoint.
  - **VerticalLineData**: Implements VerticalLine.
  - **Point**: Has two methods, getX and getZ, to be able to get the X and Z coordinates of a point.
  - **PointData**: Implementation of the Point interface. Has double precision floating point values for the X and Z coordinates.

- **Mission**
  - **Mission**: Interface which defines a mission by its name and a list of points.
  - **MissionData**: Implements Mission.
  - **MissionStrategy**: This interface defines the MissionStrategy by a name, the Mission and a list of (ordered) points.

- **MissionStrategyData**: Implements MissionStrategy. Also verifies upon construction that the list of ordered points given as the strategy contains the same points as the mission.
- **MissionProgressListener**: Interface using the observer pattern to allow classes to get notified of the mission progress of a robot.
- **PrintingMissionProgressListener**: Implements MissionProgressListener, printing a human-readable line to standard output each time a robot reaches a new point of its mission.
- **MissionManager**: The MissionManager can assign missions to robots with a strategy that is given by an instance of the MissionStrategy class, and it has a method for getting a robot's MissionStrategy. This interface also provides functionality to add and remove MissionProgressListeners, as well as getting the current mission progress.
- **DefaultMissionManager**: Implements MissionManager. The responsibility of the DefaultMissionManager is to handle the movement of the robot, making sure it reaches all points in its MissionStrategy. It accepts a mapping of RuntimeRobot to MissionStrategy and steers these robots to their destination points in the order specified by the strategy. While it does this, it observes location controllers as well as rooms and stops the robot if necessary. This can occur when the robot has entered a new room or when it needs to wait in order to acquire a location controller. Robots can also be given new missions at runtime in order to dynamically change their behaviour.

- **Reward**
  - **RewardPoints**: Has a method which returns the number of reward points the system has.
  - **DefaultRewardPoints**: Implements a counter of reward points, which it exposes by implementing the RewardPoints interface. Has one or two Procedures, which it calls to do the calculations for increasing the points. These procedures are given as instances of the RewardProcedure interface. This design uses the strategy pattern to modularize the reward calculation logic, thus allowing it to be changed at runtime. This runtime changing of reward logic happens every 20 seconds of simulated time, which is achieved by implementing the TimestepListener interface. It also uses the EnvironmentRuntime to get access to the robots, and refers to an AreaCollection to use as a list of scored areas.
  - **RewardProcedure**: Interface which supplies a method for calculating the reward points.
  - **DefaultProcedure**: Implements the RewardProcedure interface. Has a Map <String,Integer>, that given an area type (e.g. "surgery room") returns the value of that room. Also has a valued area type,

which is the class of the areas it wants to include in its calculations (e.g. PhysicalArea). Contains the logic for calculating the reward points.

- **DefaultProcedureA**: Extends DefaultProcedure. Used when computing reward points in physical areas. Gives the PhysicalArea class as an argument to the constructor of its parent.
- **DefaultProcedureB**: Extends DefaultProcedure. Used when computing reward points in logical areas. Gives the LocigalArea class as an argument to the constructor of its parent.
- **UniversityProcedureA**: Extends DefaultProcedureA. Gives the university point map as an argument to the constructor of it's parent.
- **ZooProcedureA**: Extends DefaultProcedureA. Gives the zoo point map as an argument to the constructor of its parent.
- **UniversityProcedureB**: Extends DefaultProcedureB. Gives the university point map as an argument to the constructor of its parent.
- **ZooProcedureB**: Extends DefaultProcedureB. Gives the zoo point map as an argument to the constructor of its parent
- **HospitalProcedure**: Implements Procedure. Calculates points for both physical and logical areas. Uses the hospital point map.
- **RewardPointsMaps**: Contains static methods for getting the point maps for each environment.

- **Runtime**
  - **RuntimeEnvironment**: Interface that extends the Environment interface. Whilst Environment contains all the static data of an environment, the RuntimeEnvironment interface extends this with the dynamical data provided by a simulation - such as a listing of RuntimeRobots and a TimestepSource.
  - **RuntimeFactory**: A RuntimeFactory takes an Environment and a set of Robots as input and initializes a simulation with them, which is then returned as a RuntimeEnvironment. We use the factory pattern for this so that different simulation implementations (such as Simbad) can be easily swapped out just by passing a different RuntimeFactory implementation.
  - **RuntimeRobot**: RuntimeRobot extends the Robot interface and is used during simulation runtime. It includes methods for setting a destination point, acquiring and releasing location controllers, as well as getting the current position and the RuntimeEnvironment the robot is in.
  - **RuntimeScenario**: A RuntimeScenario binds together all the different components that are used in a running simulation, such as the hospital scenario from assignment 5. It has a name as well as getters

for instances of RuntimeEnvironment, RewardPoints, MissionManager and FaultManager (the last three are optional and can be null). Specific instances of RuntimeScenario are usually created by a ScenarioFactory.

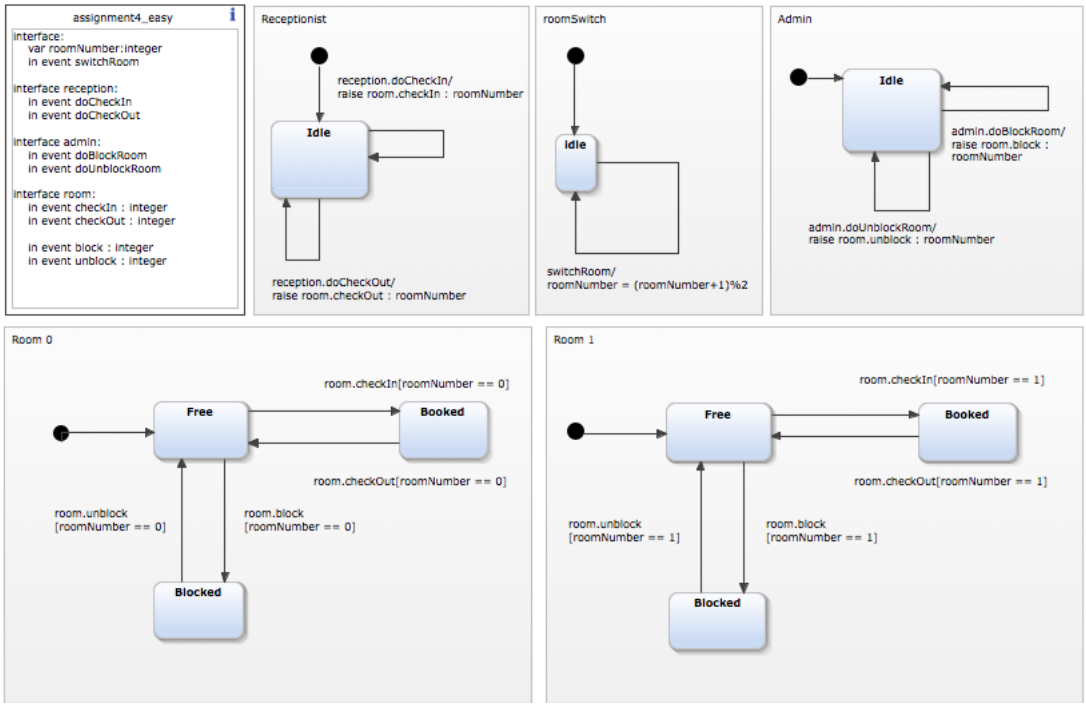
- **TimestepListener**: Following the observer pattern, the TimestepListener interface enables classes to be notified each time the simulation has progressed by implementing the onTimestep method. This can be very useful for tracking how fast time passes in the simulation, as it can be run at faster (or slower) speeds or be completely paused.
- **TimestepSource**: Has methods for adding and removing TimestepListeners.
- **TimestepDistributor**: Implements both the TimestepSource and the TimestepListener. Any time the onTimestep method is called, the information about the timestep is distributed to all added TimestepListeners.
- **ScenarioFactory**: In order to easily extend our framework to use different kinds of environments, we decided to use the factory method design pattern. This interface takes a RuntimeFactory as the input of its only method (createScenario). The interface implementation uses this RuntimeFactory to create a simulation with obstacles, robots, missions and faults. These are then collected and returned as a RuntimeScenario. This approach modularizes different scenarios (such as hospitals, zoos or the concurrency mission from assignment 3) into different implementations of the ScenarioFactory interface. Thus, new scenarios in our framework can be created easily by making a new implementation of this interface.

- **Examples**
  - **HospitalFactory**: Implements the ScenarioFactory and constructs custom environment for the Hospital.
  - **ZooFactory**: Implements the ScenarioFactory and constructs a custom environment for the Zoo.
  - **UniversityFactory**: Implements the ScenarioFactory and constructs a custom environment for the University.
  - **Assignment3Factory**: Implements the ScenarioFactory and constructs a custom environment for the concurrency mission from assignment 3.

5 Assignment 4

5.1 Task 1: Synchronization

5.1.1 State machine diagram



5.1.2 Description

There are three states which the rooms can be in, either *free*, *occupied*, or *blocked*. When a receptionist does a check in the event reception.doCheckIn is fired, and the room changes state from *free* to *occupied*, and when someone checks out the state changes from *occupied* to *free*, with the help of the event reception.doCheckOut. If a room is free it can be blocked by an admin, the event admin.doBlockRoom is fired and so the state changes from *free* to *blocked*. An admin can also change the state from *blocked* to *free* with admin.doUnblockRoom. Since you can't block a room when it's occupied and can't check in to a room that's blocked there is no transition between occupied and blocked.

To make sure that only one room is changed at a time we added the guard [roomNumber == x], where x is either 0 or 1 depending on the room ID. This way only the selected room will execute the commands.

One of the limitations with this implementation is that we can only switch between two rooms at the moment, which is not satisfactory for a normal-sized hotel. Although, if the number of rooms would increase (to say room 10), there is no functionality at the moment to directly go from room 0 to room 10. Right

now we would instead have to cycle through all of the rooms to reach room 10, which is not ideal.

5.1.3 Example scenario 1

- room.checkIn
- switchRoom
- room.checkOut
- room.block
- room.unblock
- switchRoom
- room.checkOut

We first fire room.checkIn which will change the state for room 0 to "Occupied". switchRoom is then fired which will change the room number to 1. When we now try to do a room.checkOut, nothing happens since room 1 is currently "free". However, then we run room.block and the room then changes state to "blocked" and then back to "free" again when we fire room.unblock. We then fire switchRoom again and finally fire room.checkOut that changes the state of Room 0 to "Free".

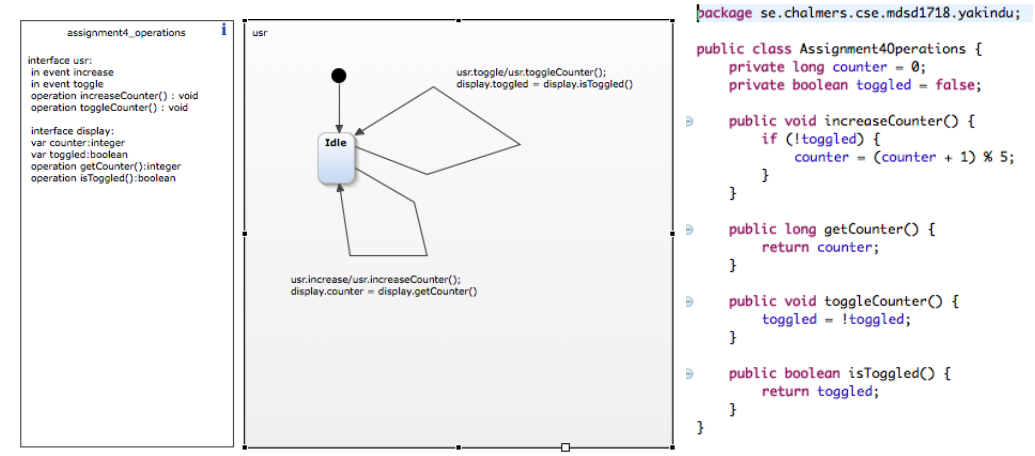
5.1.4 Example scenario 2

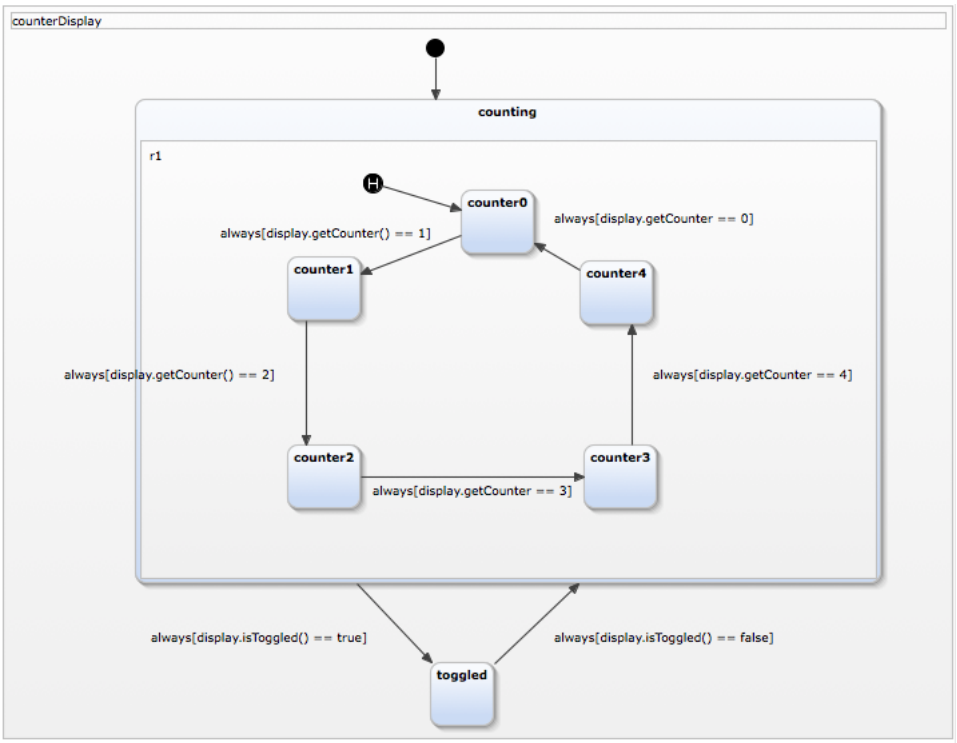
- room.checkIn
- room.checkOut

We first fire room.checkIn which changes the state of room 0 to "Occupied" and then we fire room.checkOut which changes it back to "Free".

5.2 Task 2: Operations

5.2.1 State machine diagram





5.2.2 Description

The counter consists of five states, from 0 to 4, which represents the current counter value. Each transition occurs as we increment the value of the counter.

If we are in state "counter0" and increment, the "always" statement will fire and check if the current counter value is 1 and then change state to "counter1" if this is true. The same goes for the following counter states but with different counter values as conditions for switching states. In order to change state to "toggled" the condition display.isToggled() == true must be satisfied which is true when we fire the event usr.toggle. While the current state is "toggled", incrementing the counter does nothing. The counter value is calculated using modulus so that when we reach 5, the counter will reset to 0 and change state from "counter4" to "counter0" and thus restart the counter cycle.

One limitation regarding this implementation is that it does not follow the open-closed principle. If we would like to add more counters we would both have to add another state to the state machine (and thus change around the transitions), but also change the code since it at the moment is restricted to only five counters. However, it can also be argued that this implementation is not that suitable for this type of counter, unless you know for sure that it won't have to be extended in the future.

5.2.3 Example scenario 1

- usr.increase
- usr.increase
- usr.toggle
- usr.increase
- usr.toggle
- usr.increase
- usr.increase
- usr.increase

Our starting state is "counter0". We fire the increment event and the machine will switch to state "counter1" as it satisfies the condition always[display.getCounter() == 1]. The same goes when we increment another time and it switches to state "counter2". In "counter2" we fire the toggle event which will cause a change in state to "toggled" as it satisfies the condition always[display.isToggled() == true]. Here we try to, once again, increment the counter value but since we are in the state "toggled" the if-statement in the increment method is not satisfied and thus the incrementation will not take place. Next, we fire the toggle event once again and return to state "counter2" from which we continue to fire the increment event until the counter value modulus 5 results in a reset of the counter and thus a return to state "counter0".

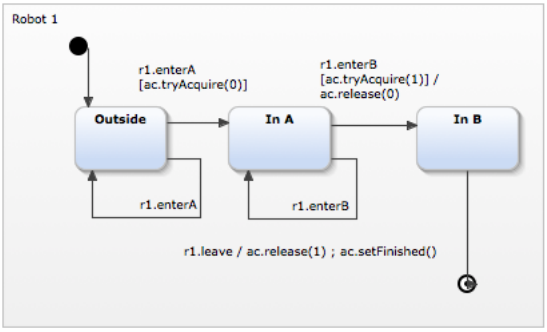
5.2.4 Example scenario 2

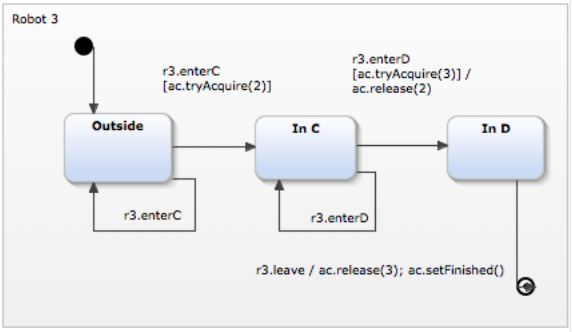
- usr.increase
- usr.increase
- usr.increase
- usr.increase
- usr.increase

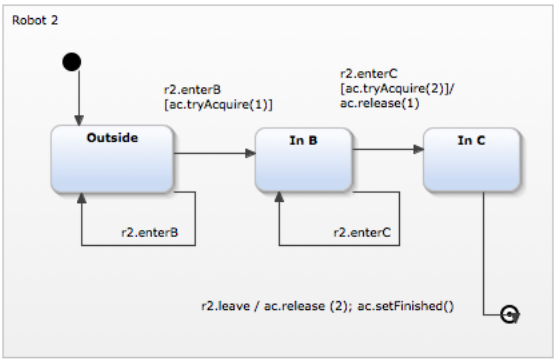
Our starting state is once again "counter0". We fire increment event and thus changes state to "counter1". We do this continuously until we arrive in the starting state "counter0" when the modulus causes the counter to reset to 0.

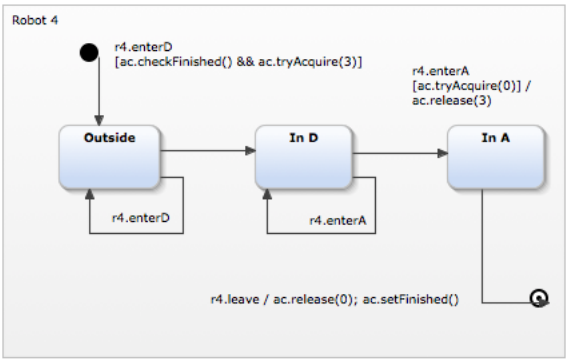
5.3 Task 3: Formalising a mission

5.3.1 State machine diagram










```
assignment4_rooms
interface:
// Define events and
// and variables here

interface r1:
  in event enterA
  in event enterB
  in event leave

interface r2:
  in event enterB
  in event enterC
  in event leave

interface r3:
  in event enterC
  in event enterD
  in event leave

interface r4:
  in event enterD
  in event enterA
  in event leave

interface ac:
  operation tryAcquire(n:integer) : boolean
  operation release(n:integer)
  operation checkFinished() : boolean
  operation setFinished() : boolean
```

```java
package se.chalmers.cse.mdsd1718.yakindu;

// This class implements an array of Area Controllers.
// Trying to acquire access to area covered by controller
// 0 is done by calling tryAcquire(0). Releasing access
// to that area is done by calling release(0), etc.
// The code performs no error checking.
public class Assignment4Rooms {

    // Increase this constant if you need more controllers.
    private boolean locked[] = new boolean[10];
    private boolean projectFinished = false;

    public boolean tryAcquire(long n) {
        int m = (int) n;
        if (locked[m]) {
            return false;
        } else {
            locked[m] = true;
            return true;
        }
    }

    public void release(long n) {
        int m = (int) n;
        locked[m] = false;
    }

    public void setFinished() {
        projectFinished = true;
    }

    public boolean checkFinished() {
        return projectFinished;
    }
}
```

5.3.2 Description

Every robot has three different states, Outside, In room 1 and In room 2. The 'In room 1/2' state is different to every robot, e.g. Robot 1 has In A and In B. It represents the mission for each robot. When a robot tries to enter the first room, the tryAcquire method is called to see if that room is occupied. If it isn't, the robot goes from state Outside to In Room 1. If the room already is occupied then the robot stays in the Outside state. When the robot tries to continue to state In room 2, it calls tryAcquire method again, and if it succeeds, then the robots releases the room it was in.

To make sure that all the robots don't enter their first room at the same time, and thus causing a deadlock, we have put restrictions on the last robot so that it can only enter its first room if another robot is done with its mission. When a robot leaves they have accomplished their mission and is therefore done executing, and end up in a final state.

We chose to only let the robot visit the rooms it needs to complete its mission, which is why there are only two rooms for each robot. In this scenario/mission it is enough for the robots to be able to move in the mission rooms. They are all still able to complete their mission, given the restriction so that all of the robots can't enter the rooms at the same time.

A limitation of this kind of solution is that if a robot with an arbitrary mission is added there is a high risk that the robots would deadlock (one robot wants to go from A to B, the other from B to A). To fix this we would need to either somehow make sure that two robots with conflicting routes are not sent into the rooms at the same time, or give the robots permission to go into rooms not included in their mission. If they can move through all of the rooms then there will always be possible movement and the robots will eventually be able to finish their mission.

If any more robots are added to the mission they would also have to wait in the beginning, and when a robot is done, only one of the waiting robots will be allowed entry.

If there were to be any more rooms added, or any removed, one would have to look at how many of the robots are permitted to enter the rooms.

5.3.3 Why this implementation is correct

Based on the description above, specially regarding avoiding deadlocks, this implementation works. Because we have restricted the path for each robot, we know that they will always enter the rooms in the specified order. Also, because the fourth robot won't enter until another one has finished we will always, in this mission, have a situation where there is at least one robot that is able to make it's move.

If we remove the movement restriction that we have and let the robots go to all of the rooms, then as long as at least one of the rooms is left empty, no matter how many rooms there are, the robots will be able to finish their missions. This is the same principle that is used with a sliding picture puzzle,

were you use the empty slot to slide squares around until you have formed the correct picture.

Finally, there are several different approaches to this problem and many different solutions. Another implementation that we thought about doing was to have a counter that keeps track on the amount of robots that enters their first room. If there are already three robots in the rooms the fourth one is not allowed to enter. This counter increments when a robot enters their first room and decrements when they leave. This way we can add more robots and be sure that no deadlock will occur, as long as they all want to visit two rooms and are going in the same direction.

Also, one could use concurrent programming in order to solve these tasks. However, since it was not a requirement in the course, as well as only half of the group knows how it works, so we chose to stick with the solution presented above.

5.3.4 Example scenario 1

- r1.enterA
- r3.enterC
- r4.enterD
- r4.enterA
- r2.enterB
- r1.enterB
- r1.leave
- r2.enterB
- r4.enterA
- r3.enterD
- r4.leave
- r2.enterC
- r2.leave
- r3.leave

Robot 1 tries to enter room A and succeeds since the room is not occupied, and the robot limit isn't reached. Then Robot 3 enters room C, Robot 4 enters room D. Robot 4 then tries to enter room A but fails, since it's occupied and thus stays in room D. Now Robot 2 tries to enter room B, but is unable to do so since the limit for the number of robots allowed inside the rooms has been reached. Robot 1 enters room B, then exits the rooms. Robot 2 now tries to

enter room B again, and succeeds. Robot 4 enters room A. Robot 3 enters room D. Robot 4 leaves the rooms. Robot 2 enters room C, then exits. And lastly Robot 3 leaves.

5.3.5 Example scenario 2

- r2.enterB
- r3.enterC
- r2.enterC
- r4.enterD
- r4.enterA
- r1.enterA
- r3.enterD
- r2.enterC
- r4.leave
- r1.enterA
- r1.enterB
- r3.leave
- r2.leave
- r1.leave

Robot2 enters Room B. Robot3 enters Room C. Robot 2 tries to enter the Room C, but fails due to Robot 3 already occupied that room. Robot 4 enters Room D. Robot 4 enters Room A. Robot 1 tries to enter Room A, but fails due to that the room is already occupied by Robot 4. Robot 3 enters Room D. Robot 2 enters Room C. Robot 4 leaves. Robot 1 enters Room A. Robot 1 enters Room B. Robot 3 leaves. Robot 2 leaves. Robot 1 leaves.

6 Assignment 5

6.1 Change of strategy

To achieve the requirement of when a robot enters a room it has to stop for two seconds, we added an AreaCollection to our Environment class that represents its rooms. The DefaultMissionManager, while controlling a robot, then checks if that robot has entered a room and stops it if necessary.

6.2 Change of environment

To simplify the creation of new environments, we added an EnvironmentBuilder that unifies the creation of environments with static obstacles, robots, rooms and location controllers using the builder pattern. On top of that, we use a factory interface to modularize the creation of specific environments (such as hospitals, zoos and universities) into factory implementations. Because of these design patterns it will be much easier to extend our program in the future

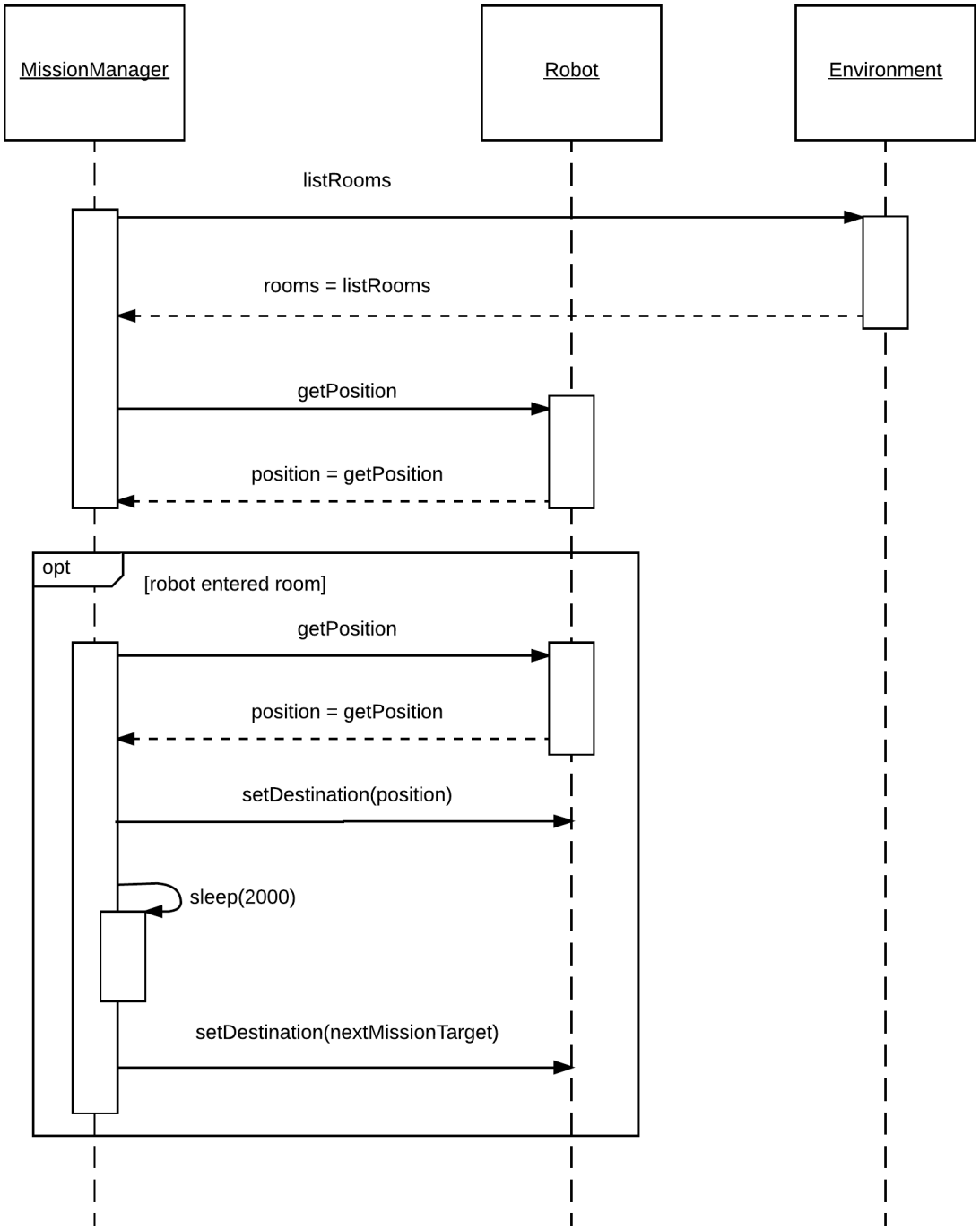
6.3 Change of logic to compute the reward

To get the required behaviour when operating in a Hospital we created a new class that implements Procedure. This class always uses all of the areas (both logical and physical, i.e. running both Procedure A and Procedure B where they apply) when calculating the points. It also has its own map that maps room type ("surgery room") to the value it's worth (20 points).

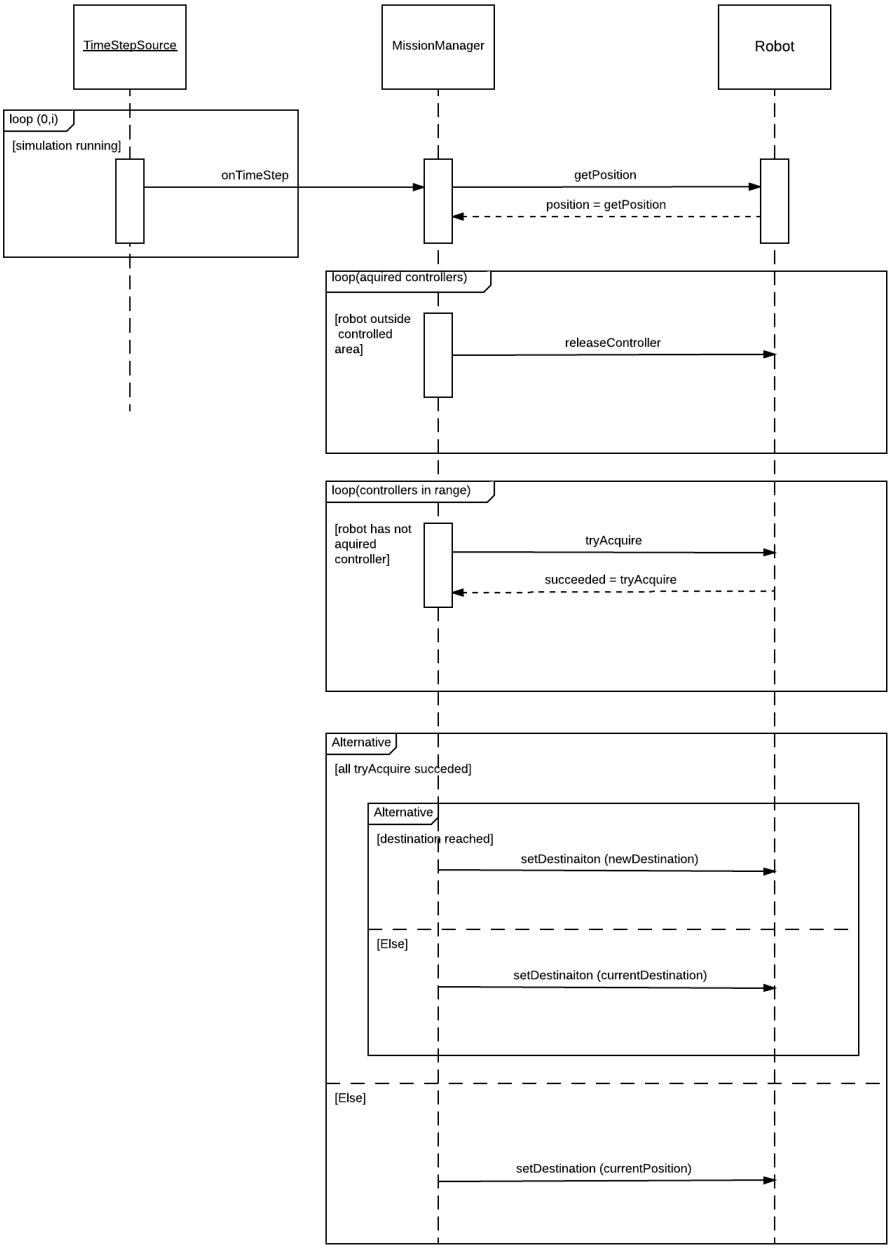
The DefaultRewardPoints is now constructed with two procedures (A and B) given individually. When a Hospital is created it should only provide the DefaultRewardPoints one procedure, leaving the second as null. When this happens the DefaultRewardPoints knows not to try switching between procedures, and simply uses the one it is given.

6.4 Sequence diagram

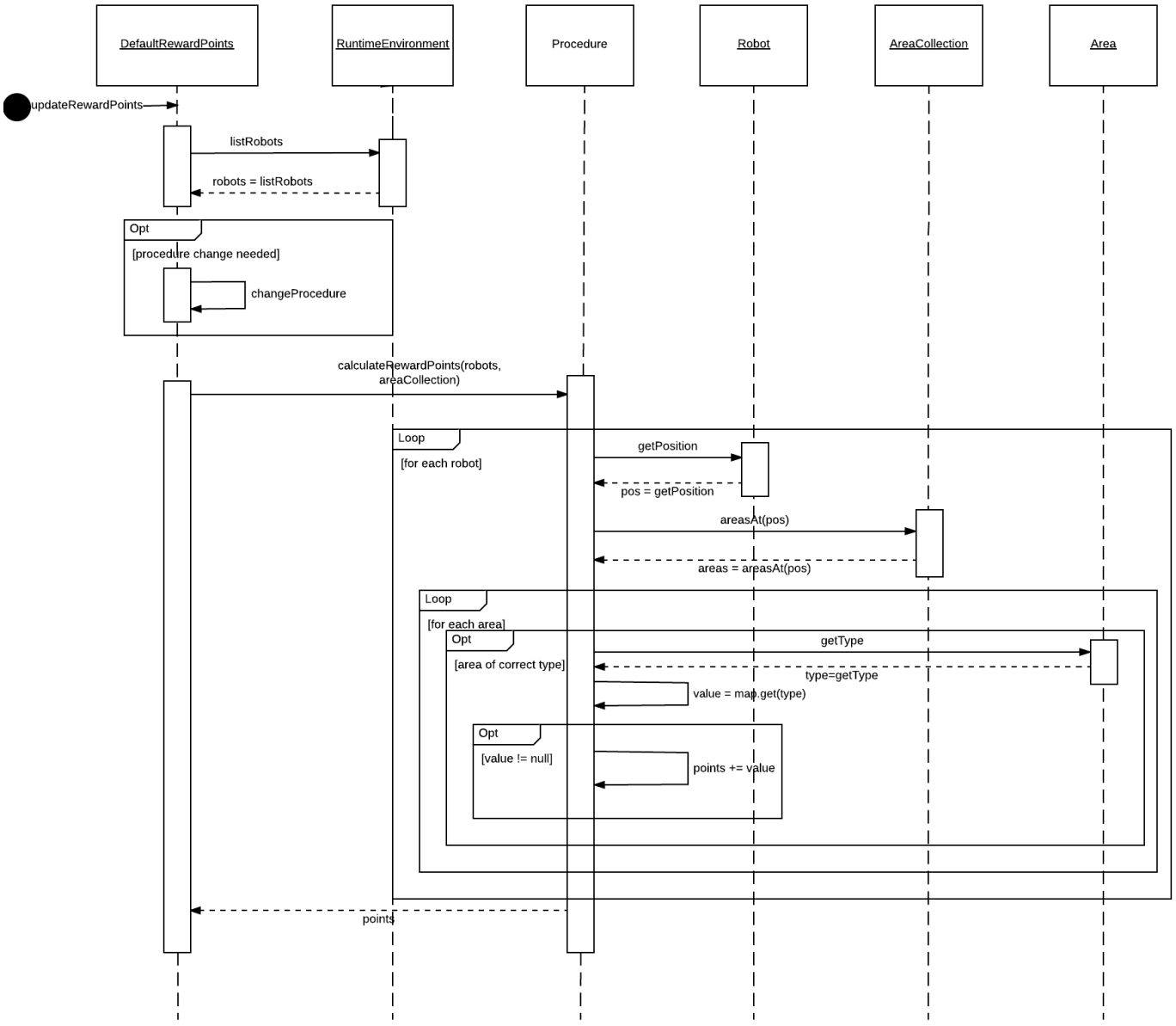
6.4.1 Scenario 1



6.4.2 Scenario 2



6.4.3 Scenario 3



7 Reflections

In this section we will specify the changes that we have made to the models and what these changes have resulted in, as well as the reasoning behind them.

7.1 Domain Model

There were no drastic changes to the domain model but there were some details we added or removed. For instance, because of robots not being able to be in the same area at the same time, we added a LocationController to the diagram. In order for the location controller to operate, it needs a location, and it needs to control a room. Because all areas are in an Environment, this means an Environment can have any number of location controllers.

Moreover, since both walls and boundaries are counted as obstacles, we changed our diagram from just having Boundary to Obstacle. Rather than having both Boundary and Wall, which would make it more difficult to extend should other obstacles arises.

The biggest change we had to make to the domain model was to remove the MissionChange component. Before this component was meant to represent the operator pushing a button to stop the mission. At the moment, changing the mission would merely be a call to our assignMission method call, where the list of points to visit would only consist of the current position the robot is currently in. However, because of lack of time we did not have time to implement this further, meaning there is currently no GUI for performing this. Hence, we chose to remove the component.

Finally, we merely changed the name from Fault to FaultEvent to better reflect what the component stands for.

7.2 Use cases

Since we got a new group member, but only for a short period of time, we had to add another use case (*Create area*) so that everyone would have one each. However, when she left we decided to keep the one we added and instead remove one of the old ones (*Terminate procedure*) which we realised was too vague and not really a use case itself, but a part of another use case (*Error handling*).

It's possible that rather than have the use case *Create area*, we should have named it *Create environment*, since that would have covered both creating the area but also all of the robots and their missions. Besides these, we have fulfilled all of the other use cases and they have been implemented into the program. *Calculate reward points* uses the different procedures as in the project description and *Observe environment* notifies the observer how the missions are going. Finally, we have a way to handle errors in the robots with the help of listeners, and *Assign mission* and *Execute strategy* together handles the logic of managing the missions of the robots. *Execute strategy* would in our program correspond to the *Mission* package.

7.3 Component diagram

The component diagram had to be made from scratch based on the feedback from the supervisor. This was due to the component diagram being extremely detailed, mostly because of us misunderstanding the assignment. We most likely thought too much into what it should look like and partly mixing the idea of a class diagram into it. Now, the updated version of the component diagram is much less detailed and gives a better overview than the one before.

We have removed a lot of interfaces and components but kept the MVC pattern in the diagram. Before, we more or less had each class as its own component with corresponding interface which resulted in a cluttered diagram with excessive details about the program. For instance, we included mostly everything covering areas as well as a lot of data types, whereas now we have replaced all of that with just the component *AreaCollection*. Since AreaCollection uses the areas, which uses the bounding boxes and such, we still have all of those details covered but it doesn't show directly in the diagram, as intended.

7.4 Class diagram

When we started with the class diagram it was a helpful tool for us to reach common ground with how we wanted the program to be structured, though we tried a little too hard to make a prescriptive diagram. Thus we got a bit lost in the details, which we had problems envisioning, and it took a bit too long since we ended up changing quite a bit anyway. For us it would probably have worked better if we had first made a general class diagram, so that we could see what parts of the program would be communicating and where would could use design patterns in an effective way, and after doing so start implementing. Lastly we could have made a more detailed diagram describing our system. The class diagram is generally a good way of documenting the code but in our case we feel that our class diagram is too big, making it hard to read. It might be more useful if we had split it up in smaller parts, either by package or maybe by use case, making it easier to read and getting an overview of said parts.

7.5 Changes to reward system:

At first we were going to give the DefualtRewardPoints a map <AbstractArea, Long> that would contain mapping from an area to the amount of points the area should give. By doing this we would have had to create the map when creating the reward points and for each area assign it a value individually, then pass it down to the procedure. We changed it so that the procedures have the a reference to a map, <String,Int>, instead. By having String instead of AbstractArea (and adding a String type, e.g. "surgery room", to AbstractArea) we are able to give all of the rooms of the same type the same value. This way we can have the same map regardless of the number of areas in the environment. If you want to add new types of areas you only need to add them in the right environment in the RewardPointsMap class, which provides the procedures with

their respective maps. Adding a new environment type is also easy this way.

We also added DefaultProcedure, DefaultProcedureA, DefaultProcedureB, and specific procedures for the different environment types. The DefaultProcedure contains the logic for calculating the reward points since the only difference between Procedure A and B are what areas to take into account. DefaultProcedureA and -B tells the DefaultProcedure what Class to value (the variable valuedAreaType), and then the environment specific procedure gets the map to use from the RewardPointsMaps (the variable pointMap).

7.6 State machines

Since the way we chose to solve the problem that we modelled with this diagram was a fairly simple one, and some of the members in our group has some experience with concurrent programming, we were quite sure that we had thought of the ways that it could go wrong and that they wouldn't happen in our program. We can see how and why they would be useful when solving more complex problems. It is a good way to make sure that the program can reach all the desired states, and that it doesn't get stuck. We also think that it can be of help when trying to come up with a solution, both that you can try them out and see what happens and that it can give a different perspective, one based in states, from which to approach the problem.

7.7 Sequence Diagram

Similarly to the State machines, nothing was changed for this diagram (except for one return arrow not being dashed). However, the heavier part of this assignment was to also make certain changes to the system (which has already been discussed in that section).

7.8 Responsible and Controller of Use Cases

For each use case we chose one responsible and one controller. Even though the responsible didn't have to be responsible for implementing the use case, in some cases this is what happened anyway. However, besides this we did not keep it consciously in mind during the project who was responsible or controller for which use case but rather tried the most efficient way to distribute the work load. A lot of parts in the project are dependent of each other which is why we found it difficult to split up the work in any other way.

Furthermore, it did not seem necessary in our case, because everyone was eager to participate during all parts of this course and so getting assignments done was rarely a big issue.

7.9 Usage of Design Patterns

- Factory pattern was used for the creation of RuntimeEnvironments and Scenarios. We used the factory pattern for the creation of scenarios be-

cause it would allow us easily to add new scenarios in the future if needed. However, a scenario factory must instantiate the simulation itself with the required obstacles and robots, since these elements can not be added once the simulation is running. Having the scenario factory directly instantiate e.g. the SimbadRuntime would not be desirable though, as this would make each scenario directly dependant on a specific simulator implementation. To fix this, we introduced the RuntimeFactory interface, which is implemented by the SimbadRuntimeFactory class. This allows the ScenarioFactory implementations to accept any implementation of RuntimeFactory as their input, thus decoupling them from simulator implementations.

- The builder pattern is used in the EnvironmentBuilder class. The EnvironmentBuilder is used by the different ScenarioFactories to simplify the creation of specific environments with obstacles, rooms and location controllers. Since the instantiation of an Environment with its contents is otherwise very verbose, this realization of the builder pattern helps us to avoid the telescoping constructor anti-pattern.
- We used the Strategy pattern for reward procedures. This allows the DefaultRewardPoints class to easily switch between different methods for calculating reward points. It also simplifies the process of extending the framework with new logic for reward points calculation.
- The simbad abstraction relies on the adapter pattern to decouple the simbad simulator from our framework. For example, the SimbadRobot class adapts Simbad's API to expose robot information via our framework's RuntimeRobot interface. This means that if Simbad's API ever changes, the changes within our framework can usually be contained within the simbad abstraction module. It has also allowed us to implement a workaround for a bug in Simbad within the abstraction, thus making it completely transparent to the rest of the framework. In the future, this abstraction layer may also enable our framework to switch to different simulator implementations.
- The observer pattern is used in the interfaces MissionProgressListener, TimestepListener and FaultListener. The MissionProgressListener is used by the DefaultMissionManager to be able to broadcast how much progress the robots have done (how many mission points they have visited). The FaultListener is used by the FaultManager for it to be able to report faults. The TimestepListener is used by the TimestepDistributor, which is used for knowing when/how often the simulation updates. For example, DefaultMissionManager and DefaultRewardPoints use this for updating robot destinations and timing the calculation of reward points.

7.10 The social contract

In the beginning of this course we established a social contract in order for us to work more efficient together. We feel that we have upheld the contract by

not being late and always be up to help each other. The breaks have been very good in giving us some rest in between our work so not to exhaust ourselves in the process. We feel that a successful project or team will benefit from a social contract and it acts as a template for how a team should behave and work together and in our case it has worked out well.

7.11 Future options

Because of the given time frame we did not have the time to fix everything we intended to do.

One thing that we would like to change is to remove the unnecessary interfaces from the data classes, such as Point and BoundingBox, and change the names from PointData to Point (same for BoundingBox).

Furthermore, as our program works now, we do not use the strategies for missions with any design pattern. When we first thought about the strategies we had a different opinion of how to construct it that differed from the teachers view but had we had the time to reconstruct it we would have used the strategy design pattern. Lastly, as we mentioned earlier we never had any time to fully implement the GUI for changing the mission. Had we had more time this implementation would be completed. There would be a button to stop the mission, and when pressed it would assign a new mission to the robot. The current destination of the robot would be the only point in the list of points for it to visit, meaning it would stop moving.

8 Results and learning outcome

8.1 Results

The result of our work consists of our six different diagrams as well as the Java implementation. The finished product consists of any number of robots in an environment, with missions consisting of destination points the robots need to visit. The criteria from the description stated that any two robots cannot be in the same room at the same time, as well as when a robot enters a new room it has to stop for two seconds and both of these things are working properly.

The use cases and the domain model together represent the problem domain the simulation has been based on. The component diagram as well as the class diagram together forms the complete overview of the program's structure and was the foundation on which the program was built.

Additionally, the behavioural models, the state machines and sequence diagrams, became the assurance that the simulation reacted to certain event properly.

8.2 Modelling

Structuring a program with the help of models will give a better understanding and it will also make it simpler to extend the program later on. However, as clear as this sounds, there are several factors to take into account when using models in order to not over-complicate nor miss important aspects.

During this course we have been introduced to many new ways of modelling software where some models have been more useful in a certain context, and other models useful in another.

For documenting a software system then the domain model, component and class diagram have been more useful. In different ways these provide context and explanation for how the system should be *structured*, which is useful both for the engineer but also for someone on the outside who wants an understanding of the project. However, in our case we had the least use for the component diagram due to our misunderstanding of it, as discussed above.

Concerning the *behaviour* of the system, however, the state machines and sequence diagram provide better information. Their contribution lies in to show how the program would respond to certain actions, which can be an important insight in case it does not respond the way it was supposed to.

When models are appropriate for modelling a system very much depends on the situation and the project. For smaller projects it is unlikely that you will need big, complex diagrams to represent the system. The use case diagrams and domain model are probably always useful to some extent. They are not overly complicated to create and they give a clear and concise summary of what the system should be able to do. However, for smaller projects it is less likely that you will need both component and class diagram, unless it is for an outsider who wishes to understand the work flow. Nevertheless, there is more work required for these and should thus be more carefully chosen.

Regarding both state machines and the sequence diagram these might also seem redundant for smaller projects with less complexity. However, given a project with more interactions and logic it is probably wise to include either one of them at least since they are the only behavioural models that give insight in how the program will behave when used in different contexts.

To summarise, whether or not to use a certain model is highly individual but they should be chosen with regard to the size and complexity of the project. Documentation should always be available for a system, whether it is in the form of plain text or models. Models give a better understanding of both the structure and behaviour of the system but it should also be taken into consideration that each individual will have a vision of how a model should look like and be structured. Hence, there is not simply one correct way to construct them.