DesignDocument

SRIRAM MADDIRALA

April 2018

1 Observer Pattern Design

One implementation of the observer pattern would be where the subject is arena and the robot is the observer. In this case the robot would (in its handle collision) check the entities it is colliding with and then update the sensors according to the status of the entity it is colliding with, by passing the other entity to the sensor, and its own profile information like behavior pattern and position and then update the motionhandler through the sensor. The information about the other entities would have to be sent to Robot and the Robots would have to check each pair for collisions or proximities and direct the sensors behavior to update motion handler. The sensors would be passed the particular motion or event that has occurred i.e. CollisionWithLight, CollisionWithRobot, CollisionWithWall and CollisionWithFood. Upon updatetimestep the position of the entity will be modified by sensor through sensor acting on motionhandler which acts on motion behavior which updates the position of the entity. Another implementation of the observer pattern is where the subject is Arena because it has all the data about all the entities, while the sensors are the observers. This would work by having the individual sensors of the entities receive communication from the arena on what type of collision has occurred which could vary like CollisionWithLight, CollisionWithRobot, CollisionWithWall and CollisionWithFood. The sensors could be stored in the robot which could be accessed by the arena and passed information to through the arena. When two entities are colliding through the communication class the Arena would have them adjust accordingly. This means that if the entity is a Robot and has fear, the reading from the right sensor impacts the velocity of the right wheel and the left sensor impacts the left wheel with respect to the lights. If the entity is a Robot and has exploratory behavior then the reading from the right sensor impacts the velocity of the left wheel, and the left sensor impacts the right wheel and the left sensor impacts the right wheel with respect to the lights. If the entity is a robot and senses food it should be aggressive i.e. the sensors have a positive correlation between the right sensor and left wheel, and between the left sensor and right wheel. If the entity is a Robot and is really hungry then it should ignore and go through the lights. If the robot isn't hungry then it would not go towards the food even when it senses it. If the Robot hits food or light it should be able to pass through it but if the robot hits the wall or another robot it should move

back in an arc which would be implemented in sensor through calling functions of the motion-handler at different timesteps i.e. a state machine. After the required action is taken and are updated to the new state the arena would call timestepupdate on all the entities which would update position depending on the motionhandler which had been updated by their sensors after the sensors reached their new state. This would require iterating through every entity for every mobile entity and check for collisions and then send the relevant communication depending on the entities involved. I prefer the pattern where the subject is the Arena and the observers are the sensors. This is because it would allow division of the information that robots would hold i.e. information about other entities would not exist in a given entity and would put the burden on the arena which already holds information about other entities in it. It would also allow the Arena to work on updating the sensors on collisions rather than the entities which makes sense as sensors are supposed to be handlers of collision instead of the Robot and this cuts the Robot out entirely. One issue I do have with the chosen implementation is that the preprocessing and decision-making of iterating through all entities and happens in arena which might be not be in line with what the observer pattern necessitates as it might be more in line if all the information about all the entities are passed to the sensors for them to parse through them and conduct all the processing. I prefer doing it in arena, however, as each sensor would then have to conduct preprocessing so it would be more efficient to do it in Arena.

2 Strategy Pattern Design

One implementation of the strategy pattern is where the context is Arena and the Strategy is ArenaEntity and the Strategies implementing it are ArenaMobileEntity and ArenaImmobileEntity and the Strategies implementing those are Obstacle, Base and Robot. This occurs when reset is called on every entity in the Arena Reset, when this done in the arena context on those entities, the reset from each entity's individual implementation is utilized as reset is virtual and not implemented in the ArenaEntity and instead implemented in the entities. This also occurs in Arena when arena advances time as the the arena then calls TimestepUpdate on each entity which is defined as a virtual method that is initialized but not implemented for mobile entities in ArenaEntity and is implemented in Obstacle and Robot. Therefore, Timestep update, when called on Obstacle and Robot, uses the implementation of TimestepUpdate in Obstacle and Robot. Another implementation of the pattern is where the context is Arena and the strategy is Sensor and the strategies are LightSensor and FoodSensor. When the Arena sends the information to the sensors about the other entities it would call a function called handle proximity in the Sensor Class which would work accordingly for LightSensor: it the LightSensor detects Light and has fear, the reading from the right sensor impacts the velocity of the right wheel and the left sensor impacts the left wheel with respect to the lights. If the entity is a Robot and has exploratory behavior then the reading from the

right sensor impacts the velocity of the left wheel, and the left sensor impacts the right wheel and the left sensor impacts the right wheel with respect to the lights. If the Sensor is FoodSensor then the sensors have a positive correlation between the right sensor and left wheel, and between the left sensor and right wheel. If the Robot is really hungry then the foodsensor should take steps to ensure that the Robot ignores and goes through the lights. This could be done by having the sensor not change the path in motionhandler causing the robot to go through the light after motionbehavior handles it. If the robot isn't hungry then it would not go towards the food even when it senses it

3 Adding a stimuli: Logic

In order to add a new stimuli to my project one would have to first add the physical stimuli with its necessary properties, draw the stimuli, keep track of the stimuli's effects on each entity and have a way to quantify it (through a sensor) and then handle the response from each entity to the stimulus and realize it.

4 Creation of Stimuli and Drawing

First the stimuli class would have to be made, a stimuli.h and a stimuli.cc if it is neccessary. This part is fairly straightforward as it just entails fleshing out the base attributes and methods that the stimuli class requires after inheriting from a parent ArenaEntity class. If the stimuli is mobile it would inherit from ArenaMobileEntity otherwise ArenaImmobileEntity would be its parent. Then one would have to go into EntityFactory.cc/.h and create an instance of the stimuli and increment the count of entities as well as add a count for the stimuli. The initial values of its attributes would be defined in its EntityFactory::CreateStimuli method and whatever isn't initialized in that method would have to have been initialized in the initialization list of the constructor of the stimuli. Then it would be returned. If the stimuli is to have random position then a line of the sort stimuli-¿setPose(SetPoseRandomly()); would have to be called, a stimuli-¿setType(); would also have to be set therefore one would have to go into the EntityType class and define an enum kStimuli. Setting the radius and color are fairly trivial but would suffice to have them be set manually using any params defined for the stimuli in params.h for example one could define one such parameter in that class like so: define STIMULIRADIUS 10 . After this is all taken care of you would have to go into arena.cc and add a line in the constructor that would look like this AddEntity(kStimuli,STIMULINUMBER), this would create the required stimuli and add them to the entities array. Then this stimuli would also automatically be drawn as is added to the entities vector.

5 Creation of StimuliSensor

After the stimuli class is somewhat fleshed out, you would have to add a StimuliSensor class to track the stimuli. The class would inherit from the Sensor class and apart from a constructor and destructor no changes would be required unless the way that the stimuli reading is calculated needs to be altered. In that case you would have to go to my sensor class, make the CalculateReading method virtual and then go to StimuliSensor and have the method header be void CalculateReading(int x, int y) override. Then any changes that are neccesary for how reading is to be calculated (for example changing the upper bound for the reading or multiplying by a constant) could be done. Then one would have to go to the robot.h file and instantiate the StimuliSensor in the private section like: StimuliSensor examplesensor; Then the getter for the sensor needs to be: StimuliSensor getexample()return examplesensor; This is because if the ampersand is not added then when this sensor is being fed readings in arena then the readings would be fed to a copy of the sensor making the effort futile.Then you would have to initialize it in the initialization list in robot.cc constructor. Then in the handlecollision method you would have to define the protocol upon collision with the stimuili i.e. if (objecttype==Kstimuli) the protocol would have to be defined in the curly brackets. Then in the TimestepUpdate function you would have to set the wheel velocity of the motionhandler dependent on the sensor reading for example: motionhandler.setvelocity (getmotionhandler().getvelocity().left+ rightFood.getReading(),getmotionhandler(). getvelocity().right+ leftFood.getReading()); the if statement would be replaced in your version by any conditional you would have to check of before velocity is impacted by the stimulus. Then after motionbehavior.UpdatePose is called you would have to make a call to your sensor to Reset the reading. This process could be replicated for multiple sensors. Once you have your sensors setup and position where you want them if you want them to be drawn on a good template would be : nvgBeginPath(ctx); nvgCircle(ctx, xpos, ypos, staticcast¡float¿(2)); nvgFillColor(ctx, nvgRGBA(250, 250, 0, 255)); nvgFill(ctx); nvgStrokeColor(ctx, nvgRGBA(0, 0, 0, 255)); nvgStroke(ctx)

Here you would simply have to replace xpos, ypos with the position of your sensor. This would go in the DrawRobotSensors method of graphicsArenaViewer.cc and you could replicate this for each sensor you have.

6 Reading the Stimuli

The last step would be to feeding the stimuli to the sensor. This would be done in the arena.cc class as all the information is already there and it would occur in the updateentitiestimestep method in the body of the following nested for loop: for(auto ent1: robotentities) for (auto ent2 : entities) you would have to check if ent2 was a stimuli by having a line if(ent2-¿gettype()==kStimuli) in the body of this conditional you could simply feed the sensor the position of the stimuli like so: ent1-¿getleftFood().CalculateReading (ent2-¿getpose().x,ent2-¿getpose().y);

Instead of getleftFood() you would have to just put in the getter function for the sensor you defined in robot.h and that would be it. You would be done.