World of Balance Game

Lobby_Server - https://github.com/hunvil/Lobby_Server
Database file - https://github.com/hunvil/Lobby_Server/blob/master/sql/wob.sql
(Exported MySQL database SQL file)
Database configuration - https://github.com/hunvil/Lobby_Server/blob/master/conf/db.conf
(The database name, user name and password needs to be configured here)
Game Server configuration –
https://github.com/hunvil/Lobby_Server/blob/master/conf/gameServer.conf
(The port on which the server listens is configured here)
Request/Responses
https://github.com/hunvil/Lobby_Server/tree/master/src/net

The World of Balance Game Server now constitutes the ATN Engine within the game server. The source code for the same is available at github at the url above. This document first discuses the core concepts and important classes in the WoB Game Server, the database tables that were modified from the earlier semesters and then the wrapper methods for the following functions used in the ATN Engine which needs to be consumed for any lobby-based game client.

1) Species invasion (Add New species)

2) Species proliferation (Increase Biomass of existing species)

3) Species exploitation (Delete species)

4) Species removal/reduction (Decrease Biomass of existing species)

5) Updating the system parameters (Both the node and link parameters – Note: A provision is provided for link parameter though it has not been tested)

Core concepts and classes involved in World of Balance Server

Before diving into the details on the wrappers for the ATN Engine, we will have a brief description of some important classes of the WoB server. In the WoB game, the player has the capability to login, enter the world and join a lobby in the event the lobby

capacity has not reached its maximum. Once the player has joined the lobby, the player sends a request to start the ecosystem and the game begins.

The requirements for the Lobby based game for future clients are given below.

1. In a typical Lobby based Game the user registers and logs in to the WoB to play the game.
2. The user is able to enter the world.
3. The user is able to join the lobby if the lobby has not reached its maximum capacity.
4. The user is able to add a zone to his ecosystem. If the zone is not currently assigned and is available to occupy, it will be allocated to the player.
5. The user can manage the species in the ecosystem i.e. they can add biomass to a species or remove biomass from a species.
6. If the user is able to manage his ecosystem over a period of time without the species going extinct the environment score is updated and extra credits or points are assigned to the user.
7. The user has access to the game shop. With the credits available, the user can buy additional species at a price and add it to his ecosystem.
8. The user is able to request prediction results at any point in the game.

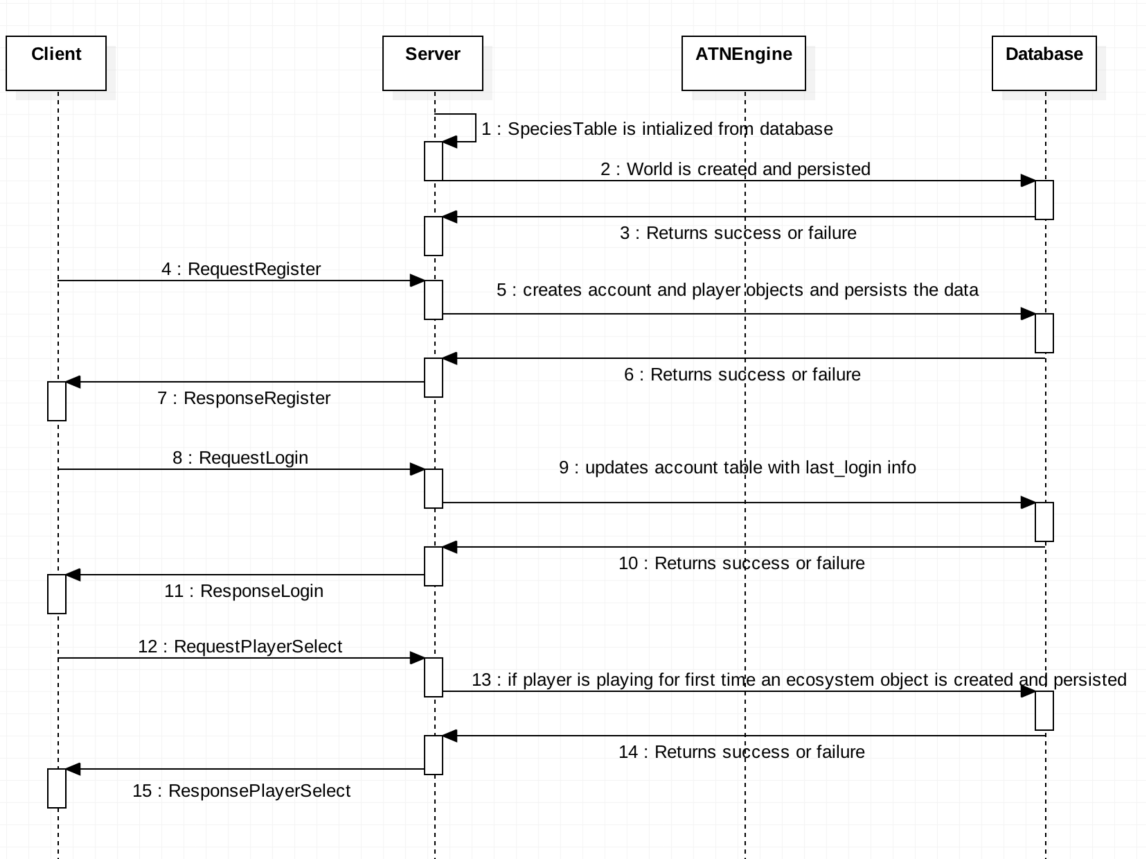

Figure 1: Sequence of events during Game Server initialization

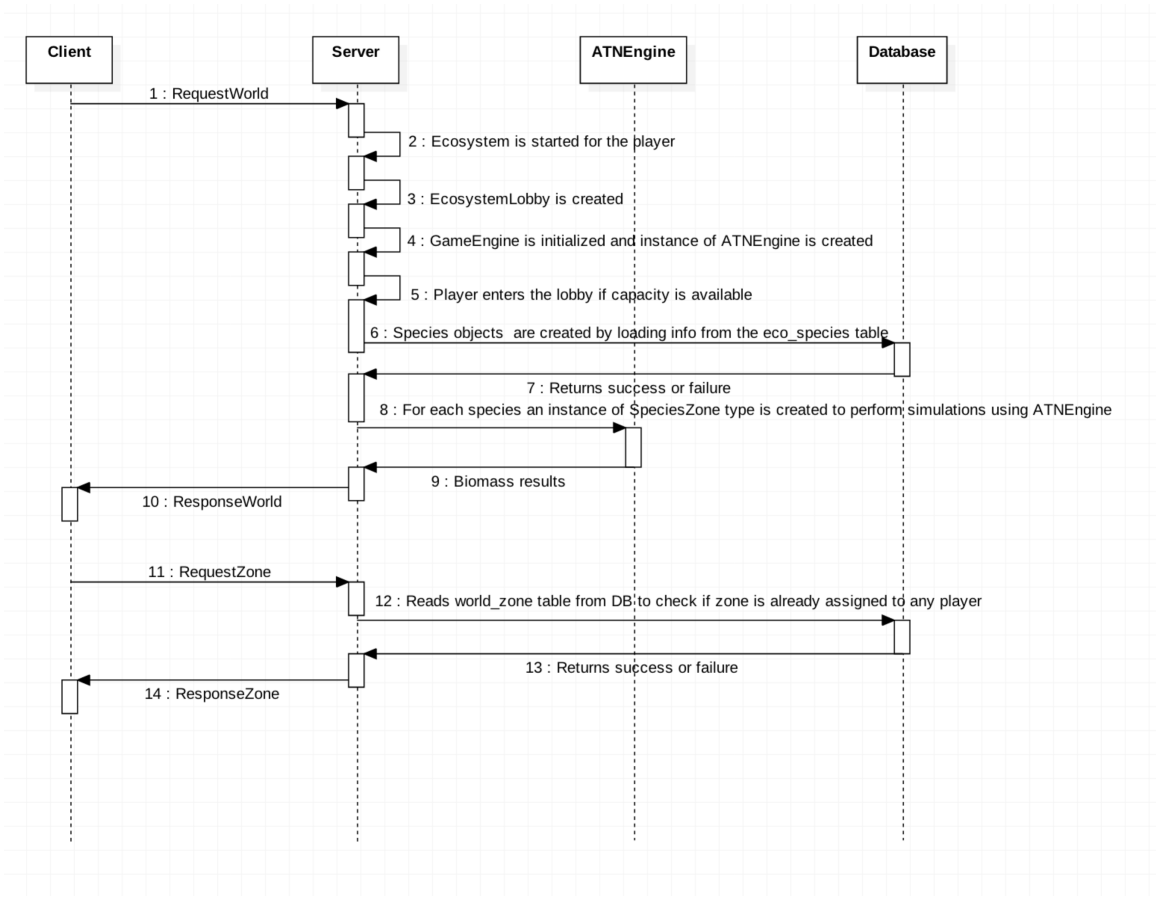

Figure 2: Sequence of events during RequestWorld

When the game server is started a sequence of events occurs as shown in Figure **1** and Figure **2**. All the worlds in the game are initialized first. There are three important singletons that manage the game, the `WorldController` class is a singleton that manages all the worlds in the game, the `EcosystemContoller` class manages all the ecosystems in the game and the `LobbyController` manages all the lobbies in the game.

World

A World represents a type of ecosystem. In our game the world represents the Serengeti ecosystem. The species information (eg: African Clawless Otter, African Fish Eagle, Tree Mouse etc.) that is used in the game pertains to the Serengeti grassland environment. The

prey-predator relationship derived is based upon years of studying this ecosystem by ecologists. The Serengeti grassland ecosystem is located in northern Tanzania in East Africa. In future we could add another world that maps to a different ecosystem. The '`World`' table in database is shown in Figure **3**.

world_id	name	time_rate	type	day	create_time
1	Serengeti	1	0	5	2014-03-11 19:19:26

Figure 3: Structure of 'world' table in DB

species_id	name	organism_type	cost	description	category	biomass	diet_type
1	Aardvark	0	50	The aardvark is a medium-sized, burrowing...	Small Animal	66	1
2	African Clawless Otter	0	50	The African clawless otter, also known as th...	Large Animal	13	1
3	African Fish Eagle	0	10	The African Fish Eagle is a large species of e...	Bird	2.8	1
4	Bat-Eared Fox	0	50	The bat-eared fox is a canid of the African s...	Large Animal	3.35	1
5	Black Backed Jackal	0	60	The black-backed jackal, also known as the...	Large Animal	9.6	0
6	Black Mamba	0	50	The black mamba, also called the common...	Small Animal	1.59	1
7	Buffalo	0	50	The African buffalo is a large African bovine...	Large Animal	1590	2
8	Thrips	0	10	Thrips are tiny, slender insects with fringed...	Insect	1.1	2
9	Rove and Ground Beetles	0	10	The rove beetles are a large family of beetle...	Insect	0.071	1
10	Orb-Weaver Spider	0	15	The "typical" orb-weaver spiders are the mo...	Insect	0.004	1
11	Ants	0	10	Ants are social insects of the family Formici...	Insect	0.000...	0

Figure 4: Structure of 'species' table in DB

We initialize the species information in the World by reading the entire '`species`' table of the database. A few species from the table are shown in Figure **4**. During game initialization we iterate through all the list of species in the database and create the `SpeciesTable` class and instances of `Species` type POJOs for each species. At any point of time we can ask the `SpeciesTable` class for information about any specific species. Each `SpeciesType` object is further separated into either `PlantType` or `AnimalType` for plant or animal species respectively. The important attributes used in the World class are shown below in Table **1**.

`playerList`	The `playerList` variable in the `World` class maintains the list of players who have joined the world.
`world_id`	The id of the world in which the player is playing.
`zoneList`	A two dimensional array of Zone. The World is divided into multiple Zones.

Table 1: Attributes of the World class

Zone

A Zone is an area of land in a World. A World is divided into multiple Zones. When a player requests a `Zone`, a zone is assigned to the player if it is currently not owned by anyone else. In the game, a player could own many zones if needed. As shown in Figure **5** we see that the zone_id = 1 is assigned to player_id = 166 and the other zones are not assigned to anyone currently.

zone_id	world_id	row	column	terrain_type	vegetation_capacity	player_id
1	1	0	0	3	100	166
2	1	0	1	3	800	NULL
3	1	0	2	3	600	NULL
4	1	0	3	4	200	NULL
5	1	0	4	4	200	NULL
6	1	0	5	4	200	NULL
7	1	0	6	3	200	NULL
8	1	0	7	3	800	NULL
9	1	0	8	3	800	NULL
10	1	0	9	3	100	NULL
11	1	0	10	3	200	NULL

Figure 5: Structure of ‘world_zone’ table in DB

row, column	Two dimensional array represented by row and column for a zone
terrain_type	Terrain types can be configured to either desert, jungle or grasslands in the game
vegetation_capacity	Each zone can have different level of vegetation capacity

Table 2: Attributes of Zone class

The interaction of `World`, `WorldController` and `Zone` are shown in Figure **6**.

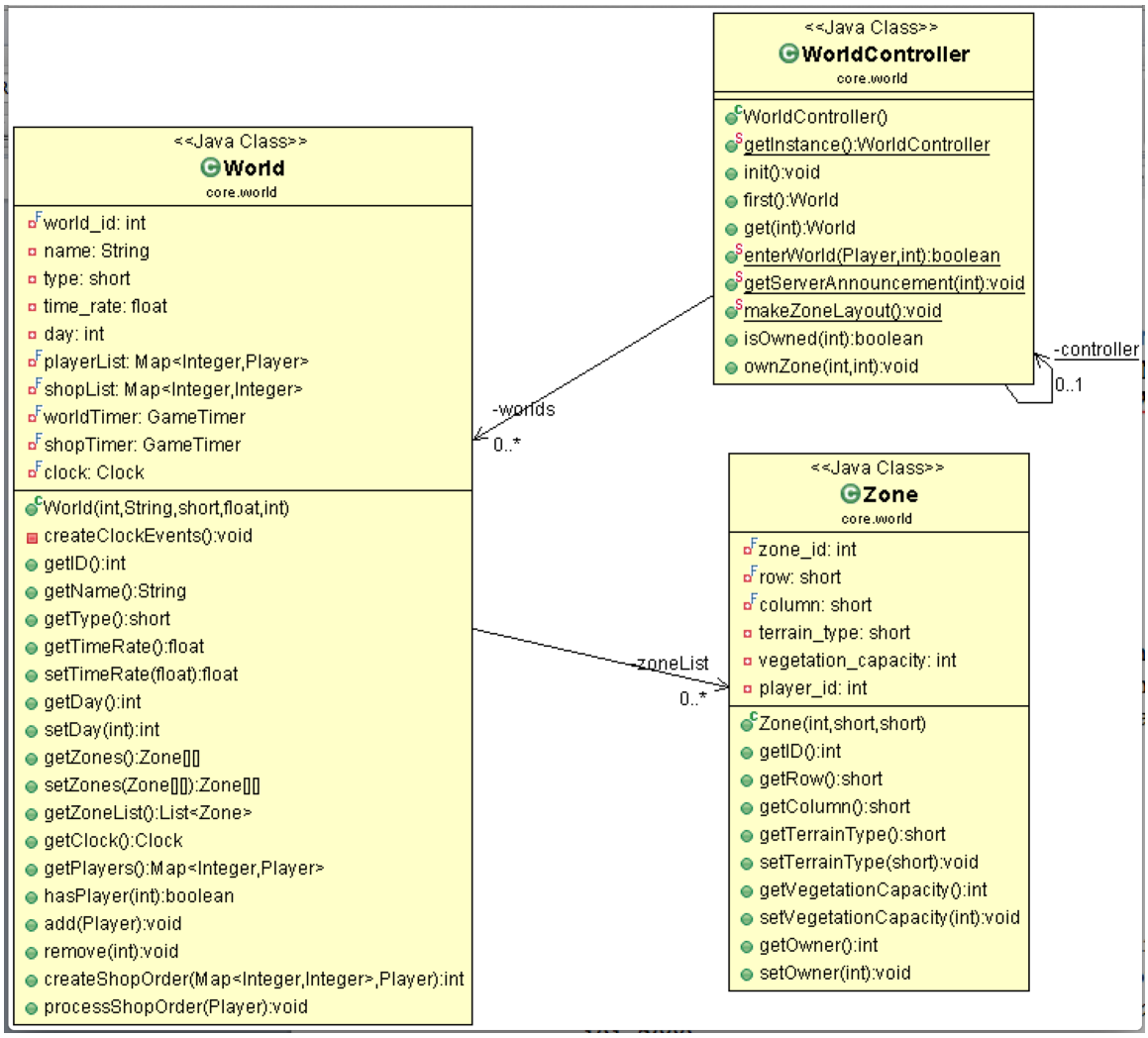

Figure 6: Class Diagram of World and its relation to Zone and WorldController

Ecosystem

The Ecosystem class is used to store information about a specific section of an Environment such as species residing in this zone. The `EcosystemController` class manages all the ecosystems in the game. The '`createEcosystem`' method is used to create a new ecosystem and '`startEcosystem`' method is used to start an existing ecosystem. When the player enters the world, an ecosystem is created if it hasn't been and a new entry is added to the database. The structure of the ecosystem in the database is shown in Figure **7**. The player is assigned a unique id for their ecosystem and is identified as '`eco_id`'. Ecosystem class manages the addition and removal of species to the ecosystem. The `zoneNodes` variable of the Ecosystem class gets all information of nodes in an ecosystem.

eco_id	world_id	player_id	name	type	manipulation_id	atn_manipulation_id	current_time_step	play_time	access_type	score	high_score
226	1	149	demo1's Ecosystem	0	NULL	NULL	1	0	1	0	0
227	1	150	j1's Ecosystem	0	NULL	NULL	1	0	1	0	0
228	1	151	newtest's Ecosystem	0	NULL	NULL	1	0	1	0	0
229	1	152	abh159's Ecosystem	0	NULL	NULL	1	0	1	0	0
230	1	153	k's Ecosystem	0	NULL	NULL	1	0	1	0	0
231	1	154	t's Ecosystem	0	NULL	NULL	1	0	1	0	0
232	1	155	c1's Ecosystem	0	NULL	NULL	1	0	1	0	0
233	1	156	c2's Ecosystem	0	NULL	NULL	1	0	1	0	0
234	1	159	c5's Ecosystem	0	NULL	NULL	1	0	1	0	0
239	1	166	hjr's Ecosystem	3	975bf1c7-8ba1-...	aa41887f-48fd-44...	20	96994701	1	27229	38061

Figure 7: Structure of 'ecosystem' table in DB

`eco_id`	A unique id for the ecosystem
`atnManipId`	A 128- bit universally unique identifier is created and assigned as the manipulation id for the ecosystem
`score, highEnvScore, accumEnvScore`	The score related attributes for the ecosystem
`speciesList`	The hashmap of all the species in the ecosystem with (K, V) as (`species_id`, `Species`)
`addNodeList`	The hashmap of all the species that are modified in the ecosystem with (K, V) as (`node_id`, `biomass`)

speciesChangeList	The prediction results are stored in the hashmap with (K, V) as (species_id, biomass)
zoneNodes	The ZoneNodes class was created to allow persistent SpeciesZoneType hashmap for each player. This class handles the management of the system parameters.

Table 3: Attributes of the Ecosystem class

Lobby

The LobbyController class is a singleton that manages the lobby list created in the game. The 'createSystemLobby' method creates a lobby of type EcosystemLobby for the player and their ecosystem, and adds it to the lobby list. A new GameEngine is initialized for each lobby that is created. If the lobby has not reached its maximum capacity, the player is allowed to enter the lobby and start managing their ecosystem. If a lobby is successfully created, the client is informed via the ResponseEcosystem protocol. The interaction between Lobby, LobbyController, EcosystemController and EcosystemLobby classes is shown in Figure **8**.

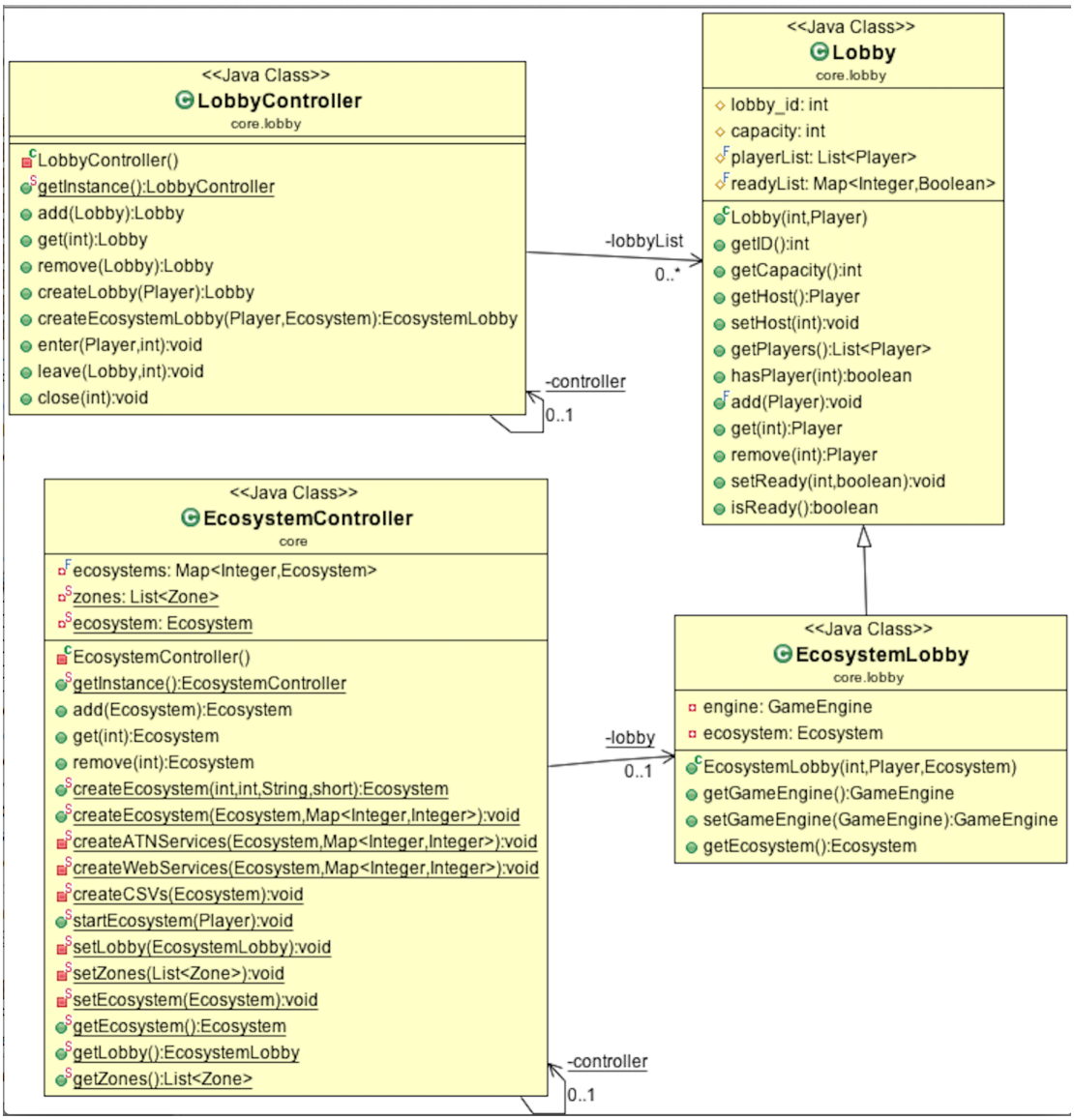

Figure 8: Class diagram of EcosystemController and its relation with LobbyController

GameEngine

The `GameEngine` class is used to control the in-game time as well as performing certain actions at specific time intervals for its assigned World. Actions such as performing predictions and species interpolation. Other methods contained in this class decide how an organism of a particular species gets created and handled. The `GameEngine` class has an ExecutorService object that executes Runnable tasks that are submitted to it. A queue of `ATNPredictionRunnable` tasks in maintained in the `GameEngine`. When a new instance of `ATNPredictionRunnable` is created it is added to the queue and when each task is dequeued for execution it is first initialized with the necessary parameters like the current list of species in the ecosystem and species that are modified and then submitted for execution. The Ecosystem and its relation to `GameEngine` and `ATNEngine` class are shown in Figure **9**.

`speciesList`	The `speciesList` refers to all the species in the Ecosystem
`newSpeciesNodeList`	The `newSpeciesNodeList` refers to the species that are added or modified when the game is in progress.

Table 4: Attributes of the GameEngine class

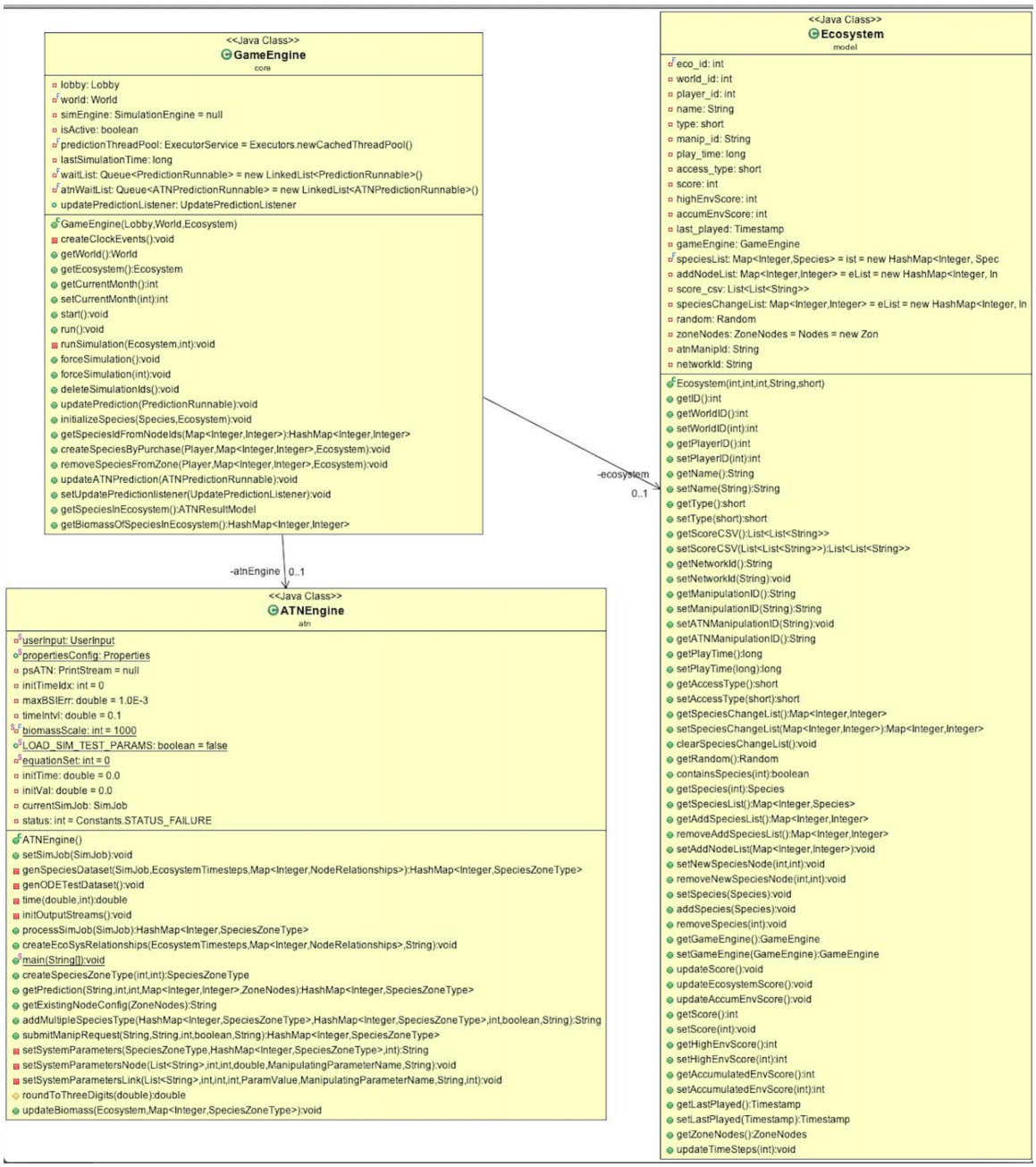

Figure 9: Class diagram for Ecosystem and its relation to GameEngine and ATNEngine

ZoneNodes

The `ZoneNodes` class was created to allow persistent `SpeciesZoneType HashMap` for each player, so that players or the system could make modifications to their species parameters, currently carrying capacity (plants), growth rate (plants) and metabolic rate (for both plants and animals). Any update to the species parameters will have to be done to the `nodes` and `plantNodes` object in the `ZoneNodes` class. For each Species object, an instance of `SpeciesZoneType` is created with node index and their biomass values. When we add each `SpeciesZoneType` to the ZoneNodes class it is separated as a `PlantNode` or `AnimalNode` for identifying the plants and animals separately since each type of species have different parameter properties.

nodes	The hashmap of all the nodes in the ecosystem with (K, V) as (`species_id`, `SpeciesZoneType`)
plantNodes	The hashmap of plant nodes in the ecosystem with (K, V) as (`species_id`, `SpeciesZoneType`)

Table 5: Attributes of the ZoneNodes class

SpeciesType

The SpeciesType class is an abstract class that is used to represent a single class of species belonging in one of the animal or plant groups. The species information as shown in Figure **4** along with prey-predator info is used to create SpeciesType objects. The important attributes are shown in the Figure **10**.

```
protected int species_id;
protected String name;
protected int cost;
protected String description;
protected float biomass;  //per-unit biomass
protected short diet_type;
protected int model_id;
protected float carrying_capacity;
protected float metabolism;
protected String category;
protected float trophic_level;
protected float growth_rate;
protected int organism_type = -1; // Animal or Plant
protected int[] preyList = new int[0];
protected int[] predatorList = new int[0];
protected Map<Integer, Float> nodeDistribution = new HashMap<Integer, Float>();
protected Map<Integer, SimTestNode> simTestNodeParams;
protected Map<Integer, Consume> simTestLinkParams;
```

Figure 10: Attributes of SpeciesType class

SpeciesZoneType

The `SpeciesZoneType` class stores the information aggregated to the same type of species in the ecosystem. In addition to all the system parameters info it also constitutes the species count, the total biomass of all species of the same `SpeciesType`. We calculate the species count by dividing the total biomass for the species by the per unit biomass. When the biomass value is updated in the `getPrediction` method, `biomassUpdated` value is set to true and when the system parameters are modified the `paramsUpdated` value is set to true. The attributes for the `SpeciesZoneType` class are shown in Figure **11**.

```
protected String name;
public int nodeIndex;
protected List<Integer> lPreyIndex;
protected List<Integer> lPredatorIndex;
protected List<SpeciesType> lPrey;
protected List<SpeciesType> lPredator;
protected int speciesCount;
protected double perSpeciesBiomass;
public double currentBiomass;
protected SpeciesTypeEnum type;
protected double trophicLevel;
public boolean biomassUpdated = false;
public boolean paramUpdated = false;
protected SpeciesType speciesType;

protected double paramK = Constants.PARAM_INITVALUE;  // carrying capacity (plants only)
protected double paramR = Constants.PARAM_INITVALUE;  // growth rate (plants only)
protected double paramX = Constants.PARAM_INITVALUE;  // metabolic rate
//link parameters are animal only
protected List<ParamValue> paramE = new ArrayList<ParamValue>();
;  // assimilation efficiency
protected List<ParamValue> paramD = new ArrayList<ParamValue>();
;  // predator interference
protected List<ParamValue> paramQ = new ArrayList<ParamValue>();
;  // functional response control parameter
protected List<ParamValue> paramA = new ArrayList<ParamValue>();
;  // functional response control parameter
protected List<ParamValue> paramB0 = new ArrayList<ParamValue>();
;  // relative half saturation density
protected List<ParamValue> paramY = new ArrayList<ParamValue>();
;  // max ingestion rate

protected double dfltK;
protected double dfltR;
protected double dfltX;
```

Figure 11: Attributes of SpeciesZoneType class

Species

The Species class refers to a single species in the ecosystem. The attributes are shown below.

species_id	The id of the species corresponding to the id in the ‘species’ table
speciesType	The reference to the SpeciesType object

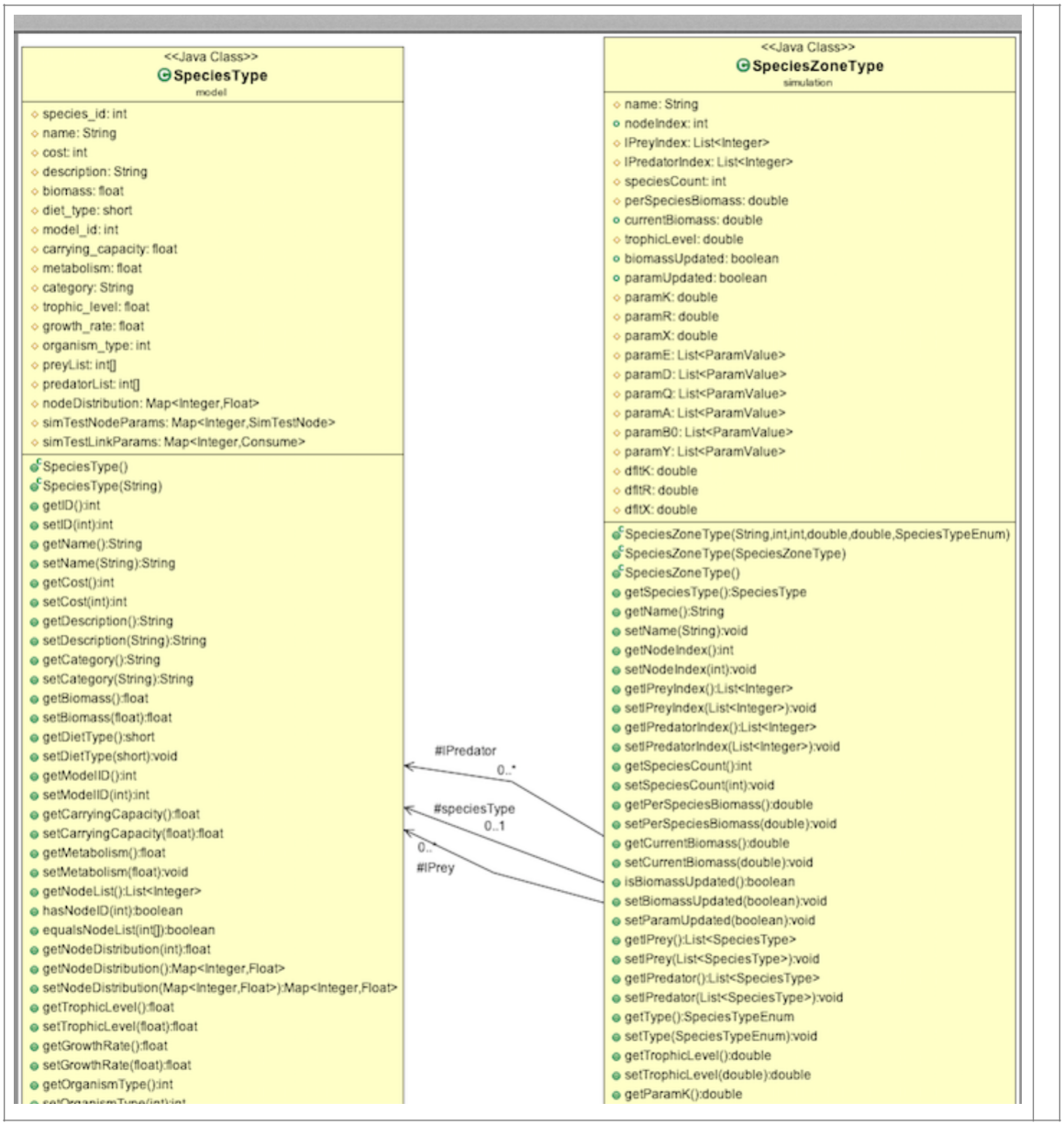

Figure 12: Relation of SpeciesType and SpeciesZoneType class (part 1)

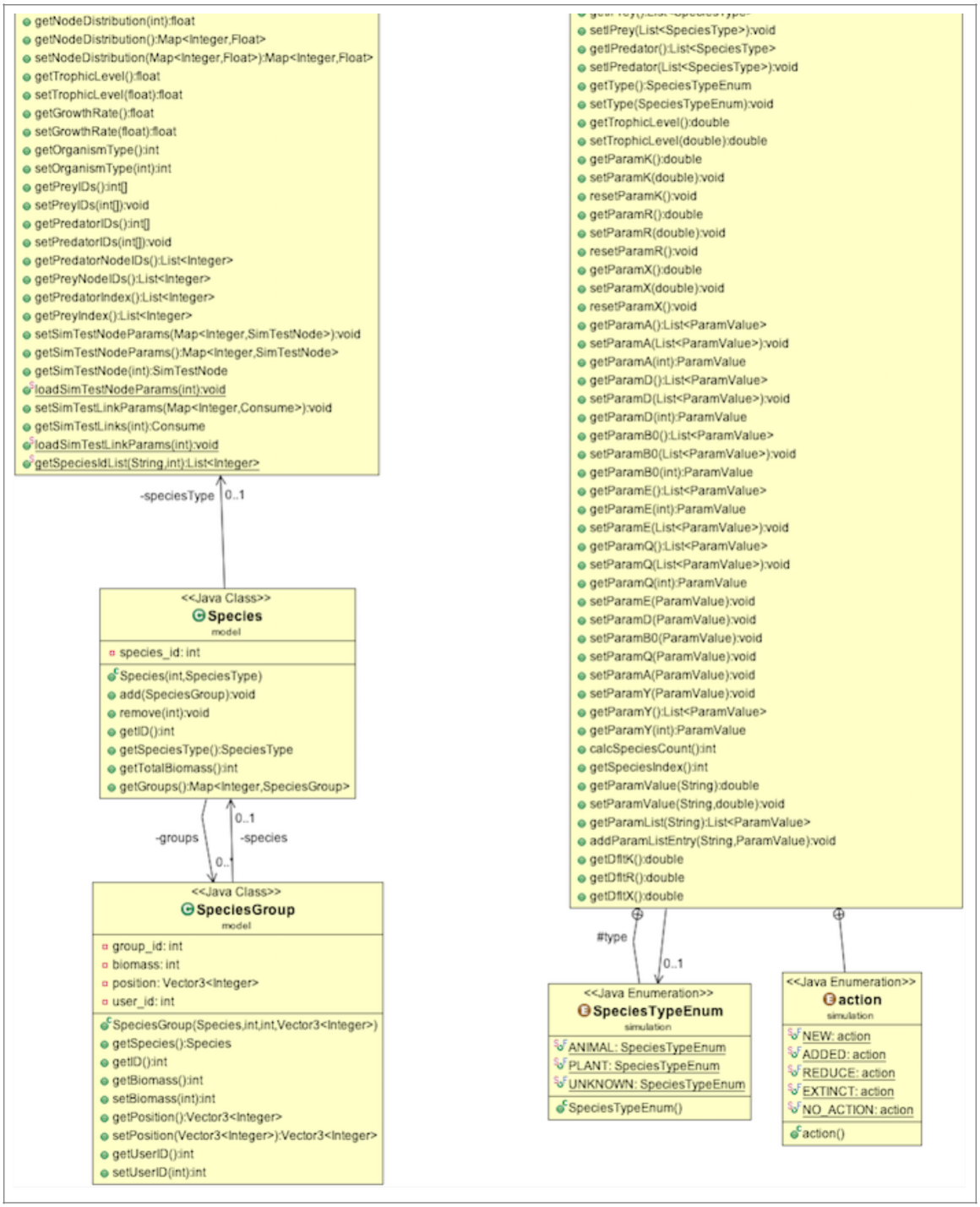

Figure 13: Relation of SpeciesType and SpeciesZoneType class (part 2)

Database

WoB Game uses MySQL database for storing data behind the scenes. While we have gone over most of the tables for the model objects, we will now discuss the tables used for the lobby-based game. `eco_species`, `eco_node_add` and `eco_species_change`. The '`eco_species`' table has the list of species in the ecosystem at any point of time in the game. The '`eco_node_add`' table is used as a placeholder until we get the prediction results for each of the species being added to the ecosystem. The results obtained from a population dynamics simulation (prediction results) are stored in the '`eco_species_change`' table. The `EcoSpeciesDAO`, `SpeciesChangeListDAO` and the `ZoneNodeAddDAO` are the data access layer objects that are used to store the '`eco_species`', '`eco_node_add`' and '`eco_species_change`' information respectively.

Name	Type	Description
eco_id	int	Primary key of the table. Id of the ecosystem
node_id	int	Node index of the species to be added or decreased or increased in the ecosystem
amount	int	Current biomass of the species to be increased, decreased in the ecosystem

Table 6:Structure of the 'eco_node_add' table

Name	Type	Description
group_id	int	Primary key of the table. All the species belonging to the same SpeciesType are added to the same group.
eco_id	int	Id of the ecosystem
species_id	int	Species Id of the species in the ecosystem
amount	int	Current biomass of the species in the ecosystem

Table 7: Structure of the ‘eco_species’ table

Name	Type	Description
eco_id	int	Primary key of the table. Id of the ecosystem
species_id	int	Species_id of the species whose prediction results are available
amount	int	Current biomass of the species either increased or decreased

Table 8: Structure of the ‘eco_species_change’ table

Wrappers for ATN Engine

Our next goal is to provide wrappers to the ATN Engine that will run simulations and get biomass predictions for the game. At the moment, since we don’t have a real game client to test this, we created a swing application GUI interface to trigger the required wrappers for different manipulation actions. The wrappers are designed keeping in mind the important manipulation actions of species in the ecosystem. We provide all the functionality for performing manipulations like Species Invasion where we add new species to the ecosystem, Species Removal where we remove species from the ecosystem, Species Proliferation where we add more species of the same type by increasing the species biomass, Species Exploitation where we decrease the number of species of same type by decreasing the species biomass, getting biomass or prediction results of various nodes in the ecosystem, modifying the manipulating node parameters such as carrying capacity, metabolism rate, growth rate and link

parameters for animal type species. The following methods are added for a lobby-based game to consume.

createATNServices	The method in the EcosystemController class used to create the unique ATN manipulation Id.
createEcosystem	The method in the EcosystemController class used to create the ecosystem if it wasn't already created
startEcosystem	The method in the EcosystemController class used to start the ecosystem by loading data from database
initializeSpecies	The method in the GameEngine class is used to create a SpeciesZoneType of object and add it to the ZoneNodes class of the ecosystem.
runSimulation	The method in the GameEngine class that is used run the simulation for the species in the ecosystem
updatePrediction	The method in the GameEngine class that is used to analyze the data from pre and post simulation.
createPurchaseBySpecies	The method in GameEngine class that is called when biomass is varied or a species is purchased or added.
removeSpeciesFromZone	The method in GameEngine class that is called when a species node is to be removed from ecosystem.
run	The method in the ATNPredictionRunnable class that is used to getPrediction results and updatePrediction results.
getPrediction	The method in the ATNEngine class to run simulations for the specified timesteps.

constructNodeConfig	The method in the ATNEngine class to assemble the node configuration to submit to the ATN Engine.
setSystemParameters	The method in the ATNEngine class to append the system parameters.
setSystemParametersNode	The method in the ATNEngine class to create an individual node parameter type.
setSystemParametersLink	The method in the ATNEngine class to create an individual link parameter type.
updateBiomass	The method in the ATNEngine class to persist the latest biomass to the database.

1. createATNServices method in EcosystemController class

We allow a provision in the game where a player could start with an empty ecosystem and build it from scratch or some default set of species could be provided for a player's ecosystem during its creation based on different levels the player is in the game. To allow this we initialize the biomass and additional parameters in the createATNServices method and persist the data in the database. The ATN Engine is initialized with a unique manipulation id for the ecosystem, which is referred to as the `atnManipId`. It then runs simulations for the default species that are assigned to the player's ecosystem.

`createATNServices` method in EcosystemController class

```
    private static void createATNServices(Ecosystem ecosystem, Map<Integer, Integer>
nodeBiomassList) {
                Log.println("Creating ATN Engine ...");
                ATNEngine atnEngine = new ATNEngine();
                if(ecosystem.getATNManipulationID() == null){
                        String atnManipId = UUID.randomUUID().toString();
                        ecosystem.setATNManipulationID(atnManipId);
                }

        // Update Zone Database
        EcosystemDAO.updateATNManipulationID(ecosystem.getID(),
ecosystem.getATNManipulationID());
        // Initialize Biomass and Additional Parameters
        List<SpeciesZoneType> mSpecies = new ArrayList<SpeciesZoneType>();
        for (Entry<Integer, Integer> entry : nodeBiomassList.entrySet()) {
            int node_id = entry.getKey(), biomass = entry.getValue();
            mSpecies.add(atnEngine.createSpeciesZoneType(node_id, biomass));
        }

        // First Month Logic
        for (SpeciesZoneType szt : mSpecies) {
            int species_id =
ServerResources.getSpeciesTable().getSpeciesTypeByNodeID(szt.getNodeIndex()).getID();
            //Will write the values into 'eco_species' table
            EcoSpeciesDAO.createSpecies(ecosystem.getID(), species_id, (int)
szt.getCurrentBiomass());
        }
    }
```

Table 9: createATNServices method in EcosystemController class

The `nodeBiomassList` parameter in Table **9** is initialized with a map of node index of the species and its biomass. For each species in the `nodeBiomassList` the ATN Engine creates a `SpeciesZoneType` object. An entry of each species assigned to the ecosystem is persisted in the '`eco_species`' table of the database. The `createEcosystem` method would be called for the first time in the event the player does not have an ecosystem created yet. For all subsequent loading, the '`startEcosystem`' method is used.

2. createEcosystem method of EcosystemController class

When a new player enters the world for the first time the `createEcosystem` method is called. The `createATNServices` method creates the `SpeciesZoneType` objects and persists the species inforamtion into the database. The environment score is then calculated based

on the biomass of the available species in the ecosystem and is written to the database. The log entry for currently available species is created and persisted in the database.

'createEcosystem' method of EcosystemController class

```
    public static void createEcosystem(Ecosystem ecosystem, Map<Integer, Integer>
speciesList) {
        // Map Species IDs to Node IDs
        Map<Integer, Integer> nodeBiomassList =
GameFunctions.convertSpeciesToNodes(speciesList);
        if(Constants.useAtnEngine){
               createATNServices(ecosystem, nodeBiomassList);
        }
        if(Constants.useSimEngine){
               // Perform Web Services
               createWebServices(ecosystem, nodeBiomassList);
        }
        // Update Environment Score
        double biomass = 0;

        for (Map.Entry<Integer, Integer> entry : speciesList.entrySet()) {
            SpeciesType speciesType =
ServerResources.getSpeciesTable().getSpecies(entry.getKey());
            biomass += speciesType.getBiomass() * Math.pow(entry.getValue() /
speciesType.getBiomass(), speciesType.getTrophicLevel());
        }

        if (biomass > 0) {
            biomass = Math.round(Math.log(biomass) / Math.log(2)) * 5;
        }

        int env_score = (int) Math.round(Math.pow(biomass, 2) +
Math.pow(speciesList.size(), 2));
        ScoreDAO.updateEnvironmentScore(ecosystem.getID(), env_score, env_score);
        // Generate CSVs from Web Services
        if(Constants.useSimEngine){
               createCSVs(ecosystem);
        }
        // Logging Purposes Only
        {
            String tempList = "";
            for (Entry<Integer, Integer> entry : speciesList.entrySet()) {
                tempList += entry.getKey() + ":" + entry.getValue() + ",";
            }
            LogDAO.createInitialSpecies(ecosystem.getPlayerID(), ecosystem.getID(),
tempList);
        }
    }
```

Table 10: createEcosystem method of EcosystemController class

3. startEcosystem method of EcosystemController class

When ‘startEcosystem’ method is called, the current species are retrieved from the ‘eco_species’ table and a Species type object is created for each species in the ecosystem. A typical Species object has all the information about the species including biomass, species id, node id etc. For performing simulations for a list of species in the ecosystem, the ATN Engine requires the species object to be of SpeciesZoneType type. For each Species object, an instance of SpeciesZoneType is created with node index and their biomass values. This SpeciesZoneType is separated as a PlantNode or AnimalNode for identifying the plants and animals separately since each type of species have different parameter properties. We use the ZoneNodes class to store this information.

‘startEcosystem’ method of EcosystemController class

```
   public static void startEcosystem(Player player) {
        // Get Player Ecosystem
        Ecosystem ecosystem = EcosystemDAO.getEcosystem(player.getWorld().getID(),
player.getID());
        if (ecosystem == null) {
            return;
        }
        // Get Ecosystem Zones
        List<Zone> zones = WorldZoneDAO.getZoneList(player.getWorld().getID(),
player.getID());
        if (zones.isEmpty()) {
            return;
        }
        // Load Ecosystem Score History

ecosystem.setScoreCSV(CSVParser.convertCSVtoArrayList(CSVDAO.getScoreCSV(ecosystem.getID
())));
        // Ecosystem Reference
        player.setEcosystem(ecosystem);
        // Create Lobby to Contain Ecosystem
        EcosystemLobby lobby =
LobbyController.getInstance().createEcosystemLobby(player, ecosystem);
        if (lobby == null) {
            return;
        }
        // Send Ecosystem to Player
        if(!Constants.DEBUG_MODE){
                ResponseEcosystem response = new ResponseEcosystem();
                response.setEcosystem(ecosystem.getID(), ecosystem.getType(),
ecosystem.getScore());
                response.setPlayer(player);
                response.setZones(zones);
                NetworkFunctions.sendToPlayer(response, player.getID());
        }
        // Load Existing Species
        for (Species species : EcoSpeciesDAO.getSpecies(ecosystem.getID())) {
            lobby.getGameEngine().initializeSpecies(species, ecosystem);
        }
        // Recalculate Ecosystem Score
        ecosystem.updateEcosystemScore();

       // Update Last Access
        EcosystemDAO.updateTime(ecosystem.getID());
    }
```

Table 11: startEcosystem method of EcosystemController class

4. initializeSpecies method of the GameEngine class

The `initializeSpecies` method is invoked when `startEcosystem` is called. For each Species object, an instance of `SpeciesZoneType` is created with node index and their biomass values. This instance is also added to the ecosystem's ZoneNodes class.

`initializeSpecies` method of the GameEngine

```
    public void initializeSpecies(Species species, Ecosystem ecosystem) {
        ecosystem.setSpecies(species);
        if(Constants.useAtnEngine){
                SpeciesType type = species.getSpeciesType();
                for (Entry<Integer, Float> entry :
type.getNodeDistribution().entrySet()) {
                    int node_id = entry.getKey(), biomass = (int)
(species.getTotalBiomass() * entry.getValue());

                    SpeciesZoneType szt = atnEngine.createSpeciesZoneType(node_id,
biomass);
                    ecosystem.getZoneNodes().addNode(node_id, szt);
                }
        }
    }
```

Table 12: initializeSpecies method of the GameEngine class

5. runSimulation method of the GameEngine class

The `runSimulation` method is used to create a runnable task and submit it to the executor service for execution. A new instance of the `ATNPredictionRunnable` runnable task is created with the current species already available in the ecosystem and the new list of species that are to be manipulated and added to the queue of runnable tasks. Each time a runnable object is dequeued from the `atnWaitList` the run method of `ATNPredictionRunnable` is called.

`runSimulation` method in GameEngine

```
    private void runSimulation(Ecosystem ecosystem, int currentTimeStep) {
        ecosystem.updateScore();

        Map<Integer, Species> speciesList = ecosystem.getSpeciesList();
        Map<Integer, Integer> newSpeciesNodeList = ecosystem.getAddSpeciesList();
        if(Constants.useAtnEngine){
                ATNPredictionRunnable atnRunnable = new ATNPredictionRunnable(this,
ecosystem, atnEngine, ecosystem.getManipulationID(), currentTimeStep,
                        speciesList, newSpeciesNodeList, ecosystem.getZoneNodes());
                atnWaitList.add(atnRunnable);

                if (atnWaitList.size() == 1) {
                    lastSimulationTime = atnRunnable.initialize();
                    predictionThreadPool.submit(atnRunnable);
                }
        }
    }
```

Table 13: runSimulation method of the GameEngine class

6. updatePrediction method of the GameEngine class

This method is used to analyze the biomass results once a simulation has been performed. The difference in biomass data pre and post simulation is calculated and we determine if the species increased or decreased in biomass over the specified time steps. The ‘eco_species_change’ table in database maintains the biomass difference of each species after the simulation happens. In this method we update the table with the latest difference in biomass values. A positive value indicates the biomass increase and a negative value indicates the biomass decrease.

`updatePrediction` method in GameEngine

```
    public void updatePrediction(PredictionRunnable runnable) {
        long milliseconds = System.currentTimeMillis();

        Ecosystem zone = runnable.getZone();
        // Remove species nodes that were just used
        for (Entry<Integer, Integer> entry :
runnable.getNewSpeciesNodeList().entrySet()) {
            int node_id = entry.getKey(), biomass = entry.getValue();
            zone.removeNewSpeciesNode(node_id, biomass);
        }
        // Execute the most recent Prediction request; drop all before it.
        waitList.remove(runnable);
        if (!waitList.isEmpty()) {
            for (int i = 0; i < waitList.size() - 1; i++) {
                PredictionRunnable r = waitList.poll();
                Log.printf("Dropped Prediction Step [%d]", r.getID());
            }

            PredictionRunnable nextRunnable = waitList.poll();
            lastSimulationTime = nextRunnable.initialize();
            predictionThreadPool.submit(nextRunnable);
        }

        Log.printf("Running Prediction Step...");

        Map<Integer, Integer> nodeDifference = new HashMap<Integer, Integer>();
        Map<Integer, SpeciesZoneType> nextSpeciesNodeList =
runnable.getNextSpeciesNodeList();

        runnable.createCSVs();

        try {
            Log.println("Interpreting Biomass Results...");
            // Determine the positive and negative change in biomass of species.
            Map<Integer, Integer> currentSpeciesNodeList =
runnable.getCurrentSpeciesNodeList();

            for (SpeciesZoneType species : nextSpeciesNodeList.values()) {
                int node_id = species.getNodeIndex();
                int nextBiomass = (int) species.getCurrentBiomass();

                if (currentSpeciesNodeList.containsKey(node_id)) {
                    int currentBiomass = currentSpeciesNodeList.get(node_id);
                    nodeDifference.put(node_id, nextBiomass - currentBiomass);
                } else {
                    nodeDifference.put(node_id, nextBiomass);
                }
            }

            Map<Integer, Integer> speciesChangeList = new HashMap<Integer, Integer>();

            // Shuffle the order at when each species get processed.
            List<Integer> speciesList = new
ArrayList<Integer>(runnable.getCurrentSpeciesList().keySet());
            Collections.shuffle(speciesList);
            // Adjust the number of species by creating or reducing the existing amount
            for (int species_id : speciesList) {
                SpeciesType speciesType =
ServerResources.getSpeciesTable().getSpecies(species_id);
```

Table 14:updatePrediction method of GameEngine class

7. createSpeciesByPurchase method of the GameEngine class

We provide API in the GameEngine class to add a new species to the ecosystem as game is progressing. The “Purchase” action is initiated by the user by buying species with the credits they have via the GameShop. We force a simulation action when a new species is added to the ecosystem. We re-use the same method to increase the biomass of the existing species in the ecosystem. The user initiates the “Buy Biomass” action by buying additional biomass for the species with the credits available. Since the game is already aware the list of species available during game play, it determines whether the species that is added is a existing species or if it is a new species.

`createSpeciesByPurchase` method in GameEngine

```
    public void createSpeciesByPurchase(Player player, Map<Integer, Integer>
speciesList, Ecosystem ecosystem) {
        for (Entry<Integer, Integer> entry : speciesList.entrySet()) {
            int species_id = entry.getKey(), biomass = entry.getValue();
            SpeciesType speciesType =
ServerResources.getSpeciesTable().getSpecies(species_id);

            for (int node_id : speciesType.getNodeList()) {
               ecosystem.setNewSpeciesNode(node_id, biomass);
            }

            Species species = null;

            if (ecosystem.containsSpecies(species_id)) {
                species = ecosystem.getSpecies(species_id);

                for (SpeciesGroup group : species.getGroups().values()) {

                    EcoSpeciesDAO.updateBiomass(group.getID(), group.getBiomass());
                    group.setBiomass(group.getBiomass() + biomass /
species.getGroups().size());
                    if(!Constants.DEBUG_MODE){
                            ResponseSpeciesCreate response = new
ResponseSpeciesCreate(Constants.CREATE_STATUS_DEFAULT, ecosystem.getID(), group);
                            NetworkFunctions.sendToLobby(response, lobby.getID());
                    }
                }

            } else {
                    int group_id = EcoSpeciesDAO.createSpecies(ecosystem.getID(),
species_id, biomass);

                    species = new Species(species_id, speciesType);
                    SpeciesGroup group = new SpeciesGroup(species, group_id, biomass,
Vector3.zero);
                    species.add(group);
                    if(!Constants.DEBUG_MODE){
                            ResponseSpeciesCreate response = new
ResponseSpeciesCreate(Constants.CREATE_STATUS_DEFAULT, ecosystem.getID(), group);
                            NetworkFunctions.sendToLobby(response, lobby.getID());
                    }
            }

            ecosystem.addSpecies(species);

            // Logging Purposes
            int player_id = player.getID(), zone_id = ecosystem.getID();

            try {
                StatsDAO.createStat(species_id, getCurrentMonth(), "Purchase", biomass,
player_id, zone_id);
            } catch (SQLException ex) {
                Log.println_e(ex.getMessage());
            }
        }
    }
```

Table 15: createSpeciesByPurchase method of the GameEngine class

8. removeSpeciesFromZone method of the GameEngine class

We provide API in the GameEngine class to remove a new species from the ecosystem as game is progressing. The user initiates the “Remove Species” action with the credits available. The reference of the species to be removed is eliminated in all the relevant classes and the `species_id` is dropped from the `eco_species` table.

`removeSpeciesFromZone` method in GameEngine

```
     public void removeSpeciesFromZone(Player player, Map<Integer, Integer>
speciesListForRemoval, Ecosystem ecosystem){
        for (Entry<Integer, Integer> entry : speciesListForRemoval.entrySet()) {
            int species_id = entry.getKey(), biomass = entry.getValue();

            SpeciesType speciesType =
ServerResources.getSpeciesTable().getSpecies(species_id);

            for (int node_id : speciesType.getNodeList()) {
               ecosystem.removeNode(node_id);
            }

            Species species = null;

            if (ecosystem.containsSpecies(species_id)) {
                species = ecosystem.getSpecies(species_id);

                for (SpeciesGroup group : species.getGroups().values()) {

                    EcoSpeciesDAO.removeSpecies(group.getID());
                    group.setBiomass(0);
                    if(!Constants.DEBUG_MODE){
                           ResponseSpeciesCreate response = new
ResponseSpeciesCreate(Constants.REMOVE_STATUS_DEFAULT, ecosystem.getID(), group);
                           NetworkFunctions.sendToLobby(response, lobby.getID());
                    }
                }

            }
            ecosystem.removeSpecies(species_id);
            ecosystem.removeEntry(species_id);

            // Logging Purposes
            int player_id = player.getID(), zone_id = ecosystem.getID();

            try {
                StatsDAO.createStat(species_id, getCurrentMonth(), "Remove", biomass,
player_id, zone_id);
            } catch (SQLException ex) {
                Log.println_e(ex.getMessage());
            }
        }
    }
```

Table 16: removeSpeciesFromZone method of the GameEngine class

9. run method of the ATNPredictionRunnable class

The `run` method first calls the `getPrediction` on the ATN Engine followed by writing the biomass results returned from the `getPrediction` to the database. The `updatePrediction` method from the GameEngine is called to analyze the prediction biomass results.

`run` method of ATNPredictionRunnable

```
@Override
 public void run() {
     if (isReady) {
         try {
             nextSpeciesNodeList = atnEngine.getPrediction(manipulation_id,
                     startTimestep, runTimestep, newSpeciesNodeList, zoneNodes);
             atnEngine.updateBiomass(zone, nextSpeciesNodeList);
             gameEngine.updateATNPrediction(this);

             Log.printf("Total Time (Simulation): %.2f seconds",
                     Math.round((System.currentTimeMillis() - executionTime) / 10.0) /
100.0);
         } catch (Exception ex) {
             Log.println_e(ex.getMessage());
         }
     }
 }
```

Table 17:run method of the ATNPredictionRunnable class

10. getPrediction method of the ATNPredictionRunnable class

The `getPrediction` method is called on the ATN Engine that runs the simulations on the food web and updates the biomass information. Subsequently the updatePrediction is called to analyze the change of biomass for all the species in the ecosystem. These change in species results are sent to the client via ResponsePrediction.

The `masterSpeciesList` gets all the species available in the ecosystem from the `ZoneNodes` object. The `addSpeciesNodeList` refers to the list of species that are newly

modified. We iterate through each `node_id` in the `addSpeciesNodeList` and determine if it already exists in the ecosystem. If the `masterSpeciesList` contains the `node_id` we update the object by adding the new biomass value, else we create a new instance of `SpeciesZoneType` object and add the new node to the `masterSpeciesList`. The `mNewSpecies` variable keeps track of all the new species that are added to the ecosystem. All the new species are added to the `ZoneNodes` object too. The `constuctNodeConfig` method is used to construct the node configuration in a format acceptable by the SimJob class. The `submitManipRequest` method submits the simulation job for processing to the ATN Engine. The `mUpdateBiomass` variable stores the prediction biomass results, which are applied to the `masterSpeciesList`.

`getPrediction` method of ATNEngine

```
  public HashMap<Integer, SpeciesZoneType> getPrediction(String networkOrManipulationId,
          int startTimestep, int runTimestep, Map<Integer, Integer> addSpeciesNodeList,
          ZoneNodes zoneNodes)
          throws SimulationException {
      long milliseconds = System.currentTimeMillis();

      Log.printf("\nPrediction at %d\n", startTimestep);

      HashMap<Integer, SpeciesZoneType> masterSpeciesList = new HashMap<Integer,
SpeciesZoneType>(zoneNodes.getNodes());

      HashMap<Integer, SpeciesZoneType> mNewSpecies = new HashMap<Integer,
SpeciesZoneType>();
      HashMap<Integer, SpeciesZoneType> mUpdateBiomass = new HashMap<Integer,
SpeciesZoneType>();

      SpeciesZoneType szt;
      String nodeConfig = null;

      for (int node_id : addSpeciesNodeList.keySet()) {
          int addedBiomass = addSpeciesNodeList.get(node_id);

          if (!masterSpeciesList.containsKey(node_id)) {
              szt = createSpeciesZoneType(node_id, addedBiomass);
              mNewSpecies.put(node_id, szt);
              masterSpeciesList.put(node_id, szt);
          } else {
              szt = masterSpeciesList.get(node_id);
              szt.setCurrentBiomass(Math.max(0, szt.getCurrentBiomass() +
addedBiomass));
              szt.setBiomassUpdated(true);
          }
      }
      // Insert new species
      if (!mNewSpecies.isEmpty()) {
          zoneNodes.addNodes(mNewSpecies);
      }
     //create nodeConfig
      try {
          nodeConfig = constuctNodeConfig (
                  mNewSpecies,
                  masterSpeciesList,
                  startTimestep,
                  false,
                  networkOrManipulationId
          );
      } catch (Exception ex) {
          Log.println_e(ex.getMessage());
      }
     // get new predicted biomass
      try {
          if(!masterSpeciesList.isEmpty() || !mNewSpecies.isEmpty()){
                 mUpdateBiomass = submitManipRequest("ATN", nodeConfig, startTimestep +
runTimestep, false, null);
          }
      } catch (Exception ex) {
          Log.println_e(ex.getMessage());
          return null;
```

Table 18: getPrediction method of the ATNPredictionRunnable class

As shown in the sequence diagram in Figure **14**, the sequence of events from RequestPrediction to ResponsePrediction is shown.

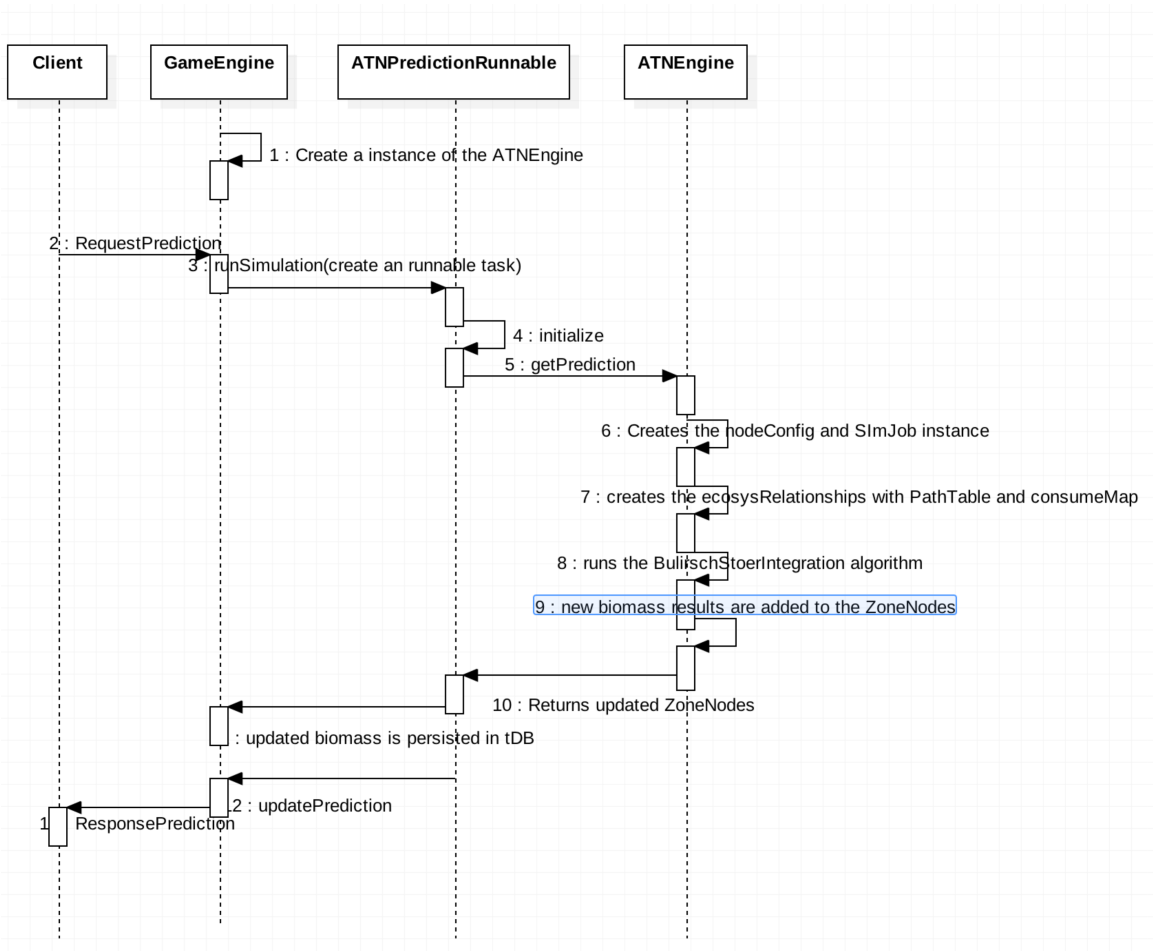

Figure 14: Sequence of events during RequestPrediction

11. constructNodeConfig method of the ATNEngine class

The `constuctNodeConfig` method assembles the node configuration as described below. A `node_config` example is shown in **Error! Reference source not found.**.

Number of Species in the ecosystem, [Node Index of Species 1], Total Biomass of Species 1, Per Species Biomass of Species 1, Number of Node Parameters for Species 1, Node parameter type = Node Parameter Value, Number of link parameters for Species 1, Link Parameter type = Link parameter value, ……, [Node Index of Species N], Total Biomass of Species N, Per Species Biomass of Species N, # of Node Parameters for Species N, Node parameter type = Node Parameter Value, # of link parameters for species N, Link Parameter type = Link parameter value

`constuctNodeConfig` method of ATNEngine

```
  public String constuctNodeConfig(
                  HashMap<Integer, SpeciesZoneType> manipSpeciesMap,
                  HashMap<Integer, SpeciesZoneType> fullSpeciesMap,
                  int timestep,
                  boolean isFirstManipulation,
                  String networkOrManipulationId){


                StringBuilder builder = new StringBuilder();
                builder.append(fullSpeciesMap.size()).append(",");
                Object[] keys = fullSpeciesMap.keySet().toArray();
                Arrays.sort(keys);
                for (Object nodeIndex : keys) {
                        SpeciesZoneType species = fullSpeciesMap.get(nodeIndex);
                        Map<Integer, SimTestNode> simTestNodeParams =
species.getSpeciesType().getSimTestNodeParams();
                        SimTestNode nodeParams =
simTestNodeParams.get(species.getNodeIndex());

builder.append("[").append(species.getNodeIndex()).append("]").append(",");
                        builder.append((int) species.getCurrentBiomass()).append(",");

builder.append(roundToThreeDigits(nodeParams.getPerUnitBiomass())).append(",");
                        String systemParam = this.setSystemParameters(
                             species,
                             fullSpeciesMap,
                             timestep);
                        builder.append(systemParam);
                }
                String node_config = builder.substring(0, builder.length()-1);
                System.out.println("NodeConfig : "+ node_config);
                //call processsim job here
                return node_config;
        }
```

Table 19: constructNodeConfig method of the ATNEngine class

12. setSystemParameters method of the ATNEngine class

The `setSystemParameters` method is called from the `constuctNodeConfig` method of ATNEngine that assembles the parameters. Since the node parameters carrying capacity (K) and growth rate (R) are only effective if species type is plant we need to check the `SpeciesTypeEnum` for the species. Higher carrying capacity indicates, the maximum biomass that the species can reach. If we set the value of carrying capacity to be X, it

means that maximum biomass of that species can go up to X. Higher the growth rate means that the species with higher growth rate will gain biomass faster. The metabolic rate node parameter is applicable to both plants and animals. Higher metabolic rate means that the biomass of species will decrease faster compared to other species. We have tested the values for K, R and X in the project. The link parameters are implemented so that it is assembled with the node configuration. The assembling of the link parameters as a `node_config` has been tested but the manipulation of link parameters has not been tested.

`setSystemParameters` method which constructs the node and link parameters

```
                private String setSystemParameters(
                          SpeciesZoneType species,
                          HashMap<Integer, SpeciesZoneType> fullSpeciesMap,
                          int timestepIdx) {

                     SpeciesTypeEnum type = species.getType();
                     int nodeIdx = species.getNodeIndex();

                     List<String> sParams = new ArrayList<String>();
                   StringBuilder builder = new StringBuilder();
                     if (type == SpeciesTypeEnum.PLANT) {
                           //YES, need to divide by Constants.BIOMASS_SCALE.

                           setSystemParametersNode(sParams, timestepIdx, nodeIdx,
                                   species.getParamK(),
                                   ManipulatingParameterName.k,
"carryingCapacityDefault");
                           if(false){ //HJR Currently I have turned off R and X
                                  setSystemParametersNode(sParams, timestepIdx,
nodeIdx, species.getParamR(),
                                          ManipulatingParameterName.r,
"growthRateDefault");
                                  setSystemParametersNode(sParams, timestepIdx,
nodeIdx, species.getParamX(),
                                          ManipulatingParameterName.x,
"metabolicRateDefault");
                           }
                           builder.append(sParams.size()).append(",");
                           for(int i = 0; i< sParams.size() ; i++){
                               builder.append(sParams.get(i)).append(",");
                           }
                           builder.append("0").append(",");
                     }
                     else if (type == SpeciesTypeEnum.ANIMAL) {
                           // Metabolic rate (x) are effective for both animals and
plants
                           // higher metabolic rate means that biomass of species will
decrease compared to other species
                           // Assimilation efficiency (e) is only available for animals.
                           // higher assimilation efficiency means that biomass of
species will increase.
                           setSystemParametersNode(sParams, timestepIdx, nodeIdx,
species.getParamX(),
                                   ManipulatingParameterName.x, "metabolicRateDefault");
                           builder.append(sParams.size()).append(",");
                           for(int i = 0; i< sParams.size() ; i++){
                              builder.append(sParams.get(i));
                              builder.append(",");
                           }
                           sParams.clear();
                           //loop through prey, adding link parameters
                           if
(Integer.valueOf(propertiesConfig.getProperty("submitLinkParameterSettings")) == 1) {
                                 int preyCnt =
species.getSpeciesType().getPreyNodeIDs().size();
                                 for (int preyIdx :
species.getSpeciesType().getPreyNodeIDs()) {
                                       if (fullSpeciesMap == null || !
```

13. setSystemParametersNode method of the ATNEngine class

In order to simplify we assemble one node parameter at a time and construct all of them together in `setSystemParameters` method.

`setSystemParametersNode` is used to create a single node parameter with type and value

```
            private void setSystemParametersNode(List<String> sParams,
                    int timestepIdx, int nodeIdx, double value,
                    ManipulatingParameterName manipParam, String dfltValProp) {
                String nodeParam = new String();
                nodeParam = manipParam.name().toUpperCase() + "=";
                /* node parameters can't have negative value. if they have negative
value, it means
                 that data is not assigned yet. */
                if (value < 0) {
                        //
sParams.append(Double.valueOf(propertiesConfig.getProperty(dfltValProp)));
                } else {
                        nodeParam +=roundToThreeDigits(value);
                }
                sParams.add(nodeParam);
            }
```

Table 20: setSystemParametersNode method of the ATNEngine class

14. setSystemParametersLink method of the ATNEngine class

In order to simplify we assemble one link parameter at a time and construct all of them together in `setSystemParameters` method.

`setSystemParametersLink` is used to create a single link parameter with type and value

```
            private void setSystemParametersLink(
                    List<String> sParams,
                    int timestepIdx,
                    int predIdx,
                    int preyIdx,
                    ParamValue pvalue,
                    ManipulatingParameterName manipParam,
                    String dfltValProp,
                    int preyCnt
            ) {
                String linkParam = new String();
                linkParam = "[" + preyIdx + "],";
                linkParam += manipParam.name().toUpperCase()+"=";
                /* node parameters can't have negative value. if they have negative
value, it means
                 that data is not assigned yet. */
                if (pvalue != null) {
                        linkParam +=pvalue.getParamValue();
                } else {
                        linkParam
+=Double.valueOf(propertiesConfig.getProperty(dfltValProp));
                }
                sParams.add(linkParam);
            }
```

Table 21: setSystemParametersLink method of the ATNEngine class

15. updateBiomass method of the ATNEngine class

The biomass needs to be persisted to the 'eco_species' table of the database for future predictions.

`updateBiomass` persists the biomass after simulation to the '`eco_species`' table

```
              public void updateBiomass(Ecosystem ecosystem, Map<Integer,
SpeciesZoneType> nextSpeciesNodeList) {
                for (Entry<Integer, SpeciesZoneType> entry :
nextSpeciesNodeList.entrySet()) {
                    int species_id = entry.getKey();
                    SpeciesZoneType szt = entry.getValue();
                              int biomassValue = (int)
entry.getValue().getCurrentBiomass();
                              Species species =
ecosystem.getSpecies(szt.getSpeciesType().getID());
                        for (SpeciesGroup group : species.getGroups().values()) {
                            group.setBiomass(biomassValue);

                            EcoSpeciesDAO.updateBiomass(group.getID(),
group.getBiomass());
                        }
                             }
                }
```

Table 22: updateBiomass method of the ATNEngine class