**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*\*\*



**BÁO CÁO BÀI TẬP LỚN MÔN**

**LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG**

### **ĐỀ TÀI: Interactive Simulation of Ecosystem Food Chains**

Mã lớp: 151965

Giaó viên hướng dẫn : TS. Nguyễn Thị Thu Trang

### NHÓM 5

|  |  |
| --- | --- |
| **Họ và tên thành viên** | **Mã số sinh viên** |
| Phạm Đức Ngự Bình | 20225696 |
| Nguyễn Công Bình | 20225695 |
| Nguyễn Minh Chiến | 20215000 |
| Trần Bá Công | 20215002 |
| Khổng Minh Cường | 20225603 |

Hà Nội, 12/2024

**PHÂN CÔNG CÔNG VIỆC**

|  |  |  |  |
| --- | --- | --- | --- |
| **Họ và tên thành viên** | **Mã số sinh viên** | **Công việc** | **Tỉ lệ tham gia** |
| Phạm Đức Ngự Bình | 20225696 | -Nhóm trưởng  -Viết báo cáo  -Hoàn thiện class diagram và use cases diagram  -Quản lý repo nhóm  -Clean và thống nhất code | 100% |
| Nguyễn Công Bình | 20225695 | Code chính các lớp Organism, Animal, Plant, Herbivore, Carnivore | 100% |
| Nguyễn Minh Chiến | 20215000 | -Phụ GUI | 100% |
| Trần Bá Công | 20215002 | -Code các lớp World, Simulator  - Phụ các lớp Organism, Animal, Plant, Herbivore, Carnivore  - Fix lỗi | 100% |
| Khổng Minh Cường | 20225603 | -Code chính GUI  -Làm slide  -Test chương trình | 100% |

**CHƯƠNG I. Phân tích đề tài**

1. **Yêu cầu của đề tài**
   1. **Mục đích**: Mô phỏng hệ sinh thái gồm thực vật, động vật ăn cỏ và động vật ăn thịt, giúp người dùng hiểu cách năng lượng được truyền qua các cấp độ dinh dưỡng và ảnh hưởng của sự thay đổi kích thước quần thể đối với sự ổn định của hệ sinh thái.
   2. **Yêu cầu thiết kế**: Xây dựng hệ thống mô phỏng hệ sinh thái, bao gồm:

* **Giao diện người dùng (GUI)**:
* **Menu** gồm lựa chọn :

+Tạo hệ sinh thái theo kịch bản cân bằng, vượt ngưỡng hoặc tuyệt chủng

+Nút Help để xem hướng dẫn mô phỏng và giải thích các kiến thức cơ bản

+Nút Quit để thoát chương trình với sự xác nhận của người dùng

* **Giao diện** điều chỉnh các thông số của mô phỏng như tỉ lệ sinh, tử và hiệu quả năng lượng
* **Mô phỏng**: Bản đồ ô lưới hình chữ nhật với thực vật, động vật ăn cỏ và động vật ăn thịt được đặt trong từng ô thể hiện vị trí. Các yếu tố được mô phỏng bao gồm:
* **Hiển thị**:

+**Thực vật**: Biểu tượng cây cối, không di chuyển nhưng tự động sinh sản sau một thời gian nhất định.

+**Động vật ăn cỏ**: Biểu tượng động vật ăn cỏ như cừu, dê …, di chuyển để tìm thực vật.

+**Động vật ăn thịt**: Biểu tượng động vật ăn thịt như sư tử, sói …, di chuyển để săn bắt động vật ăn cỏ.

* **Chuyển động**:

+Động vật ăn cỏ và động vật ăn thịt di chuyển ngẫu nhiên hoặc hướng về nguồn thức ăn.

+Động vật ăn cỏ di chuyển đến thực vật gần nhất, động vật ăn thịt di chuyển để săn động vật ăn cỏ.

* **Sinh sản**:

Các sinh vật sinh sản khi đạt đủ ngưỡng năng lượng. Sinh vật mới xuất hiện ở các ô trống liền kề.

* **Tương tác**:

+Thực vật sinh ra năng lượng qua quang hợp và bị động vật ăn cỏ tiêu thụ.

+Động vật ăn cỏ tiêu thụ thực vật để có năng lượng. Nếu không có thực vật để ăn, chúng mất năng lượng theo thời gian và có thể chết.

+Động vật ăn thịt săn động vật ăn cỏ để có năng lượng. Nếu không có con mồi, chúng mất năng lượng theo thời gian và có thể chết.

* **Chuyển giao năng lượng**:

Thực vật chuyển 10% năng lượng cho động vật ăn cỏ khi bị ăn, động vật ăn cỏ chuyển 10% năng lượng cho động vật ăn thịt.

* **Dân số và năng lượng**:

Cập nhật dân số của mỗi loài và năng lượng truyền giữa các cấp sinh vật theo thời gian.

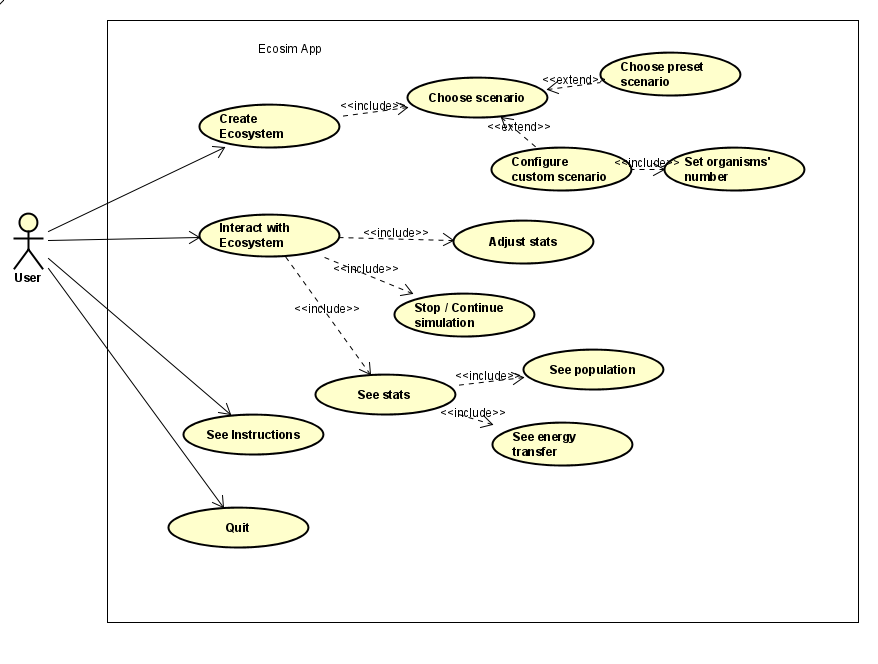
* **Tùy chỉnh**:

Điều chỉnh các thông số quần thể như tỉ lệ sinh, tỉ lệ tử, hiệu quả năng lượng.

1. **Hướng thực hiện đề tài**
   1. **Xây dựng thiét kế đề tài**
   2. **Cài đặt**
   3. **Kiểm thử**
   4. **Hoàn thiện, sửa lỗi**

**Chương II: Phân tích thiết kế đề tài**

1. **Biểu đồ use cases**

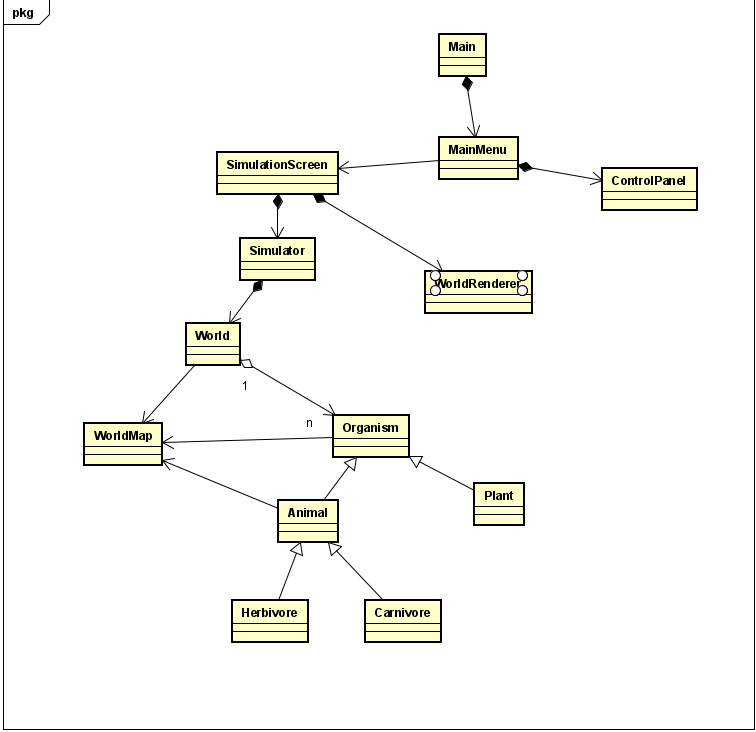


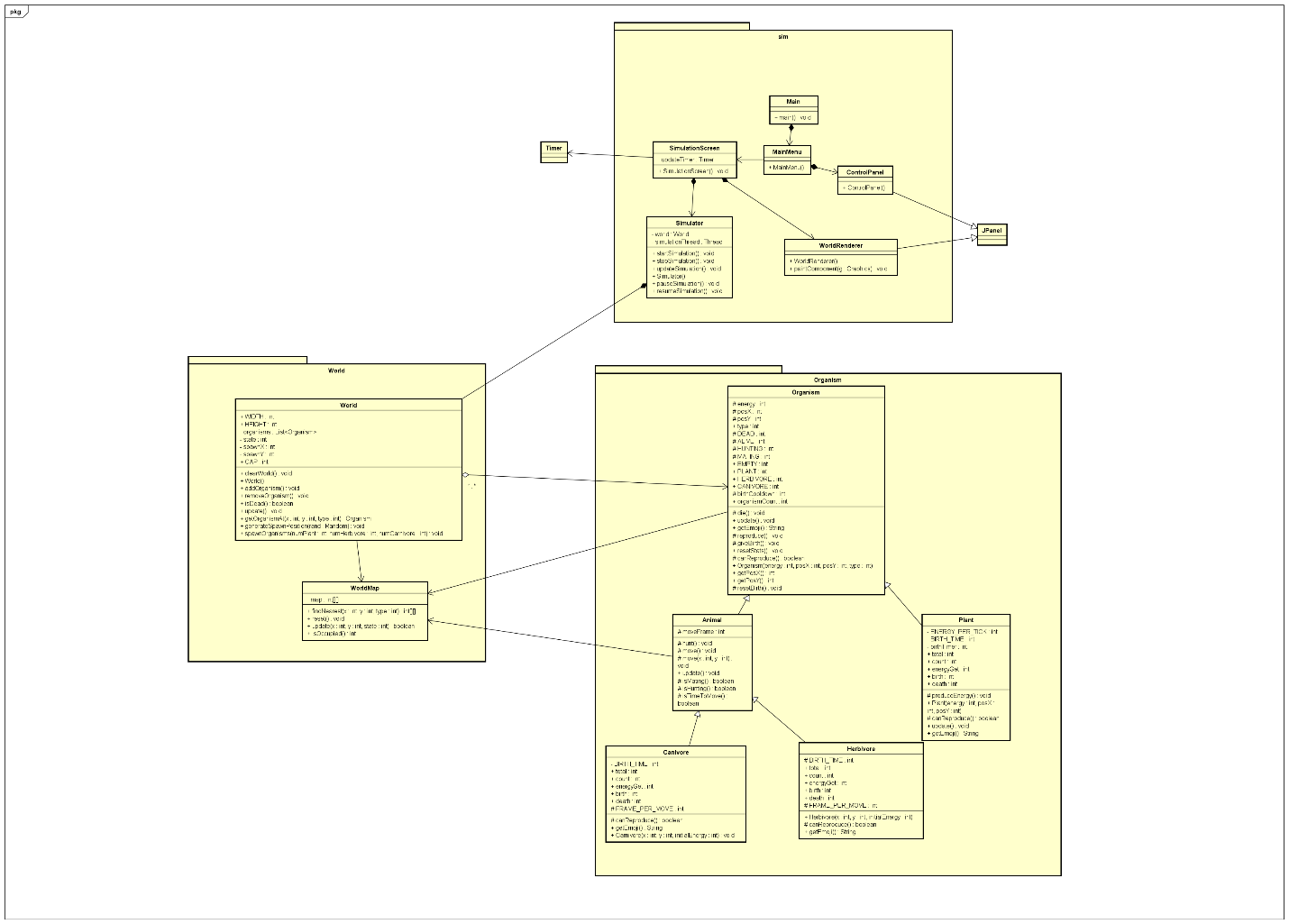
Theo yêu cầu đề tài, nhóm xây dựng **biểu đồ use case** mô tả sự tương tác giữa **Người dùng (User)** với phần mềm mô phỏng như sau :

* **Tạo hệ sinh thái (Create Ecosystem)**:
  + Người dùng có thể tạo một hệ sinh thái, bao gồm việc **Chọn kịch bản (Choose scenario)** bằng 1 trong 2 cách sau:
    - **Chọn kịch bản có sẵn (Choose preset scenario)**
    - **Tùy chỉnh kịch bản (Configure custom scenario** bằng cách **thiết lập số lượng từng loại sinh vật (Set organisms' number)**.
* **Tương tác với hệ sinh thái (Interact with Ecosystem)**:
  + Người dùng có thể tương tác với hệ sinh thái, bao gồm các chức năng:
    - **Điều chỉnh chỉ số (Adjust stats)**.
    - **Dừng / Tiếp tục mô phỏng (Stop / Continue simulation)**.
    - **Xem thống kê (See stats)**:
      * **Xem số lượng sinh vật (See population**
      * **Xem luồng năng lượng (See energy transfer)**
* **Xem hướng dẫn (See Instructions)**:
  + Người dùng có thể truy cập phần hướng dẫn sử dụng ứng dụng.
* **Thoát (Quit)**:
  + Người dùng có thể thoát khỏi ứng dụng

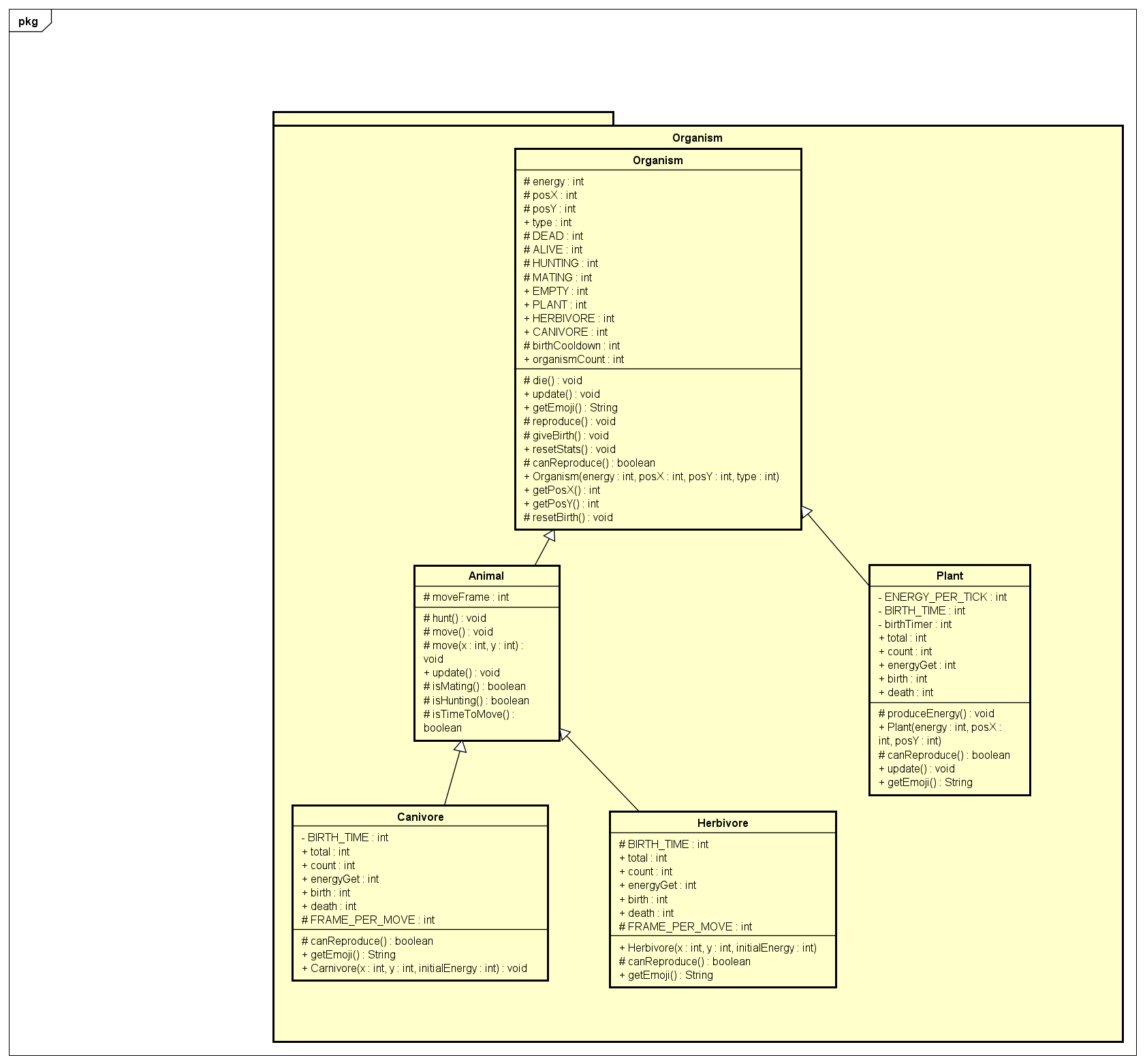
.

1. **Biểu đồ lớp**





1. Giaỉ thích thiết kế
   1. Package Organism



- Package Organism chứa các lớp biểu diễn thực thể sống với lớp trừu tượng Organism được kế thừa bởi hai lớp con là Animal biểu diễn động vật và Plant biểu diễn cây cối.

- Động vật được chia thành 2 loại là ăn cỏ và ăn thịt nên lớp Animal cũng là lớp trừu tượng, được kế thừa bởi lớp Carnivore (động vật ăn thịt) và Herbivore(động vật ăn cỏ).

-Các thực thể sống (Organism) có các thuộc tính cơ bản là vị trí trong không gian 2D(posX và posY), năng lượng(energy) , trạng thái(state), thời gian sinh sản(birthCooldown). Ngoài ra nhóm cũng thêm vào thuộc tính static organismCount để đếm số lượng sinh vật hiện tại, type để nhận diện loại sinh vật thay cho instanceof .

**public** **abstract** **class** Organism {

**public** **static** **int** *organismCount* = 0; // Current count of all organisms in the simulation

// Constants to represent cell types

**public** **static** **final** **int** ***EMPTY*** = 0, ***PLANT*** = 1, ***HERBIVORE*** = 2, ***CARNIVORE*** = 3;

// Constants to represent organism states

**protected** **static** **final** **int** ***DEAD*** = 1, ***ALIVE*** = 0, ***HUNTING*** = 3, ***MATING*** = 4;

**protected** **int** energy; // Current energy level of the organism

**protected** **int** posX; // X-coordinate of the organism's position

**protected** **int** posY; // Y-coordinate of the organism's position

**protected** **int** state;

// Current state of the organism (e.g., alive, hunting, mating)

**protected** **int** birthCooldown; // Cooldown time since the last reproduction

**public** **int** type; // Type of the organism (e.g., plant, herbivore, carnivore)

…

}

-Thuộc tính type sẽ nhận giá trị bằng một trong các hằng số ***EMPTY***, ***PLANT***, ***HERBIVORE***, ***CARNIVORE***. -Thuộc tính state sẽ nhận một trong các giá trị ***DEAD***, ***ALIVE***, ***HUNTING***, ***MATING*** biểu thị cho chết, sống, đang đi săn và đang sinh sản.

-Mọi sinh vật đều có thể sinh sản và chết đi. Do quy mô đơn giản, giả sử rằng mọi sinh vật khi có đủ điều kiện sinh sản đều sẽ sản sinh ra con, không phân biệt giới tính và không yêu cầu tìm bạn tình:

**public** **abstract** **class** Organism {

…

// Mark the organism as dead and update relevant statistics

**protected** **void** die() {

**this**.state = ***DEAD***; // Change state to DEAD

WorldMap.*update*(posX, posY, ***EMPTY***); // Clear the organism's position on the world map

}

// Handle reproduction if conditions are met

**protected** **void** reproduce() {

**if** (canReproduce()) { // Check if the organism has enough energy to reproduce

**int**[] reproducePos = chooseDirection(); // Choose a direction for the offspring

**if** (!WorldMap.*isOccupied*(reproducePos[0], reproducePos[1])) {

**this**.giveBirth(reproducePos[0], reproducePos[1]); // Create the offspring

**this**.resetBirth(); // Reset the birth cooldown

}

} **else** {

birthCooldown++; // Increment cooldown if reproduction is not possible

state = ***ALIVE***; // Maintain alive state

}

}

// Choose a random adjacent direction for reproduction or movement

**protected** **int**[] chooseDirection() {

**int**[] ans = **new** **int**[2];

ans[0] = posX;

ans[1] = posY;

**int**[] dx = { -1, 1, 0, 0 }; // Possible x-direction (left, right)

**int**[] dy = { 0, 0, -1, 1 }; // Possible y-direction (up, down)

Random rand = **new** Random();

List<Integer> choices = **new** ArrayList<>();

choices.add(0);

choices.add(1);

choices.add(2);

choices.add(3);

**int** choice;

**do** {

choice = rand.nextInt(choices.size()); // Randomly select a direction

Integer c = choices.get(choice);

ans[0] = **this**.posX + dx[c];

ans[1] = **this**.posY + dy[c];

choices.remove(c); // Remove the tested direction

} **while** (!choices.isEmpty() && WorldMap.*isOccupied*(ans[0], ans[1]));

**return** ans;

}

…

}

-Trong phương thức reproduce() có lời gọi tới phương thức trừu tượng:

**public** **abstract** **class** Organism {

…

// Abstract method to check if the organism can reproduce (implemented by subclasses)

**protected** **abstract** **boolean** canReproduce();

// Abstract method to define how an organism gives birth (implemented by subclasses)

**protected** **abstract** **void** giveBirth(**int** x, **int** y);

…

}

Do điều kiện sinh sản và cách sinh của từng sinh vật là khác nhau, nhóm vận dụng tính đa hình của lập trình hướng đối tượng và override các phương thức trên trong các lớp con là Plant, Herbivore và Carnivore:

**public** **class** Carnivore **extends** Animal {

…

**protected** **static** **final** **int** ***BIRTH\_TIME*** = 35;

// Method to give birth to a new carnivore at a specified location

@Override

**protected** **void** giveBirth(**int** x, **int** y) {

World.*addOrganism*(**new** Carnivore(x, y, 100)); // Add a new carnivore at the specified location

WorldMap.*update*(x, y, Organism.***CARNIVORE***); // Update the world map with the new carnivore's position

**this**.energy -= 40; // The parent carnivore loses 40 energy to give birth

*birth*++; // Increment the birth count

}

// Method to check if the carnivore is ready to reproduce

@Override

**protected** **boolean** canReproduce() {

// Carnivores can reproduce if they are not hunting, their birth cooldown has passed, and they have enough energy (100 or more)

**return** !isHunting() && birthCooldown >= ***BIRTH\_TIME*** && energy >= 100;

}

…

}

**public** **class** Herbivore **extends** Animal {

…

**protected** **static** **final** **int** ***BIRTH\_TIME*** = 20;

// Method to give birth to a new herbivore at a specified location

@Override

**protected** **void** giveBirth(**int** x, **int** y) {

World.*addOrganism*(**new** Herbivore(x, y, 100)); // Add a new herbivore at the specified location

WorldMap.*update*(x, y, Organism.***HERBIVORE***); // Update the world map with the new herbivore's position

**this**.energy -= 40; // The parent herbivore loses 40 energy to give birth

*birth*++; // Increment the birth count

}

// Method to check if the herbivore is ready to reproduce

@Override

**protected** **boolean** canReproduce() {

// Herbivores can reproduce if they are not hunting, their birth cooldown has passed, and they have enough energy (100 or more)

**return** !isHunting() && birthCooldown >= ***BIRTH\_TIME*** && energy >= 100;

}

…

}

**public** **class** Plant **extends** Organism {

…

**private** **final** **int** BIRTH\_TIME = 10;

@Override

**protected** **boolean** canReproduce() {

// Plants grow fast so need to check if World is full or not

**return** birthCooldown >= BIRTH\_TIME && !World.*isFull*();

}

…

}

-Ngoài sự khác biệt về hành vi sinh sản, các sinh vật còn có các phương thức khác nhau để kiếm năng lượng. Cây cối quang hợp còn động vật thì di chuyển tìm kiếm con mồi:

**public** **class** Plant **extends** Organism {

…

**private** **final** **int** ENERGY\_PER\_TICK = 10; // Năng lượng sản xuất mỗi lần cập nhật (tick)

// Phương thức sản xuất năng lượng qua quang hợp

**protected** **void** produceEnergy() {

**this**.energy += ENERGY\_PER\_TICK; // Tăng năng lượng mỗi tick

*energyGet* += ENERGY\_PER\_TICK;

}

… }

**public** **abstract** **class** Animal **extends** Organism {

// Method to move the animal to a specific destination

**protected** **void** move(**int** destX, **int** destY) {

**if** (!isTimeToMove()) **return**; // Check if the animal is allowed to move

moveFrame = 0; // Reset the move frame counter after moving

**int** x = posX, y = posY, newX = x, newY = y;

**if** (x == destX && y == destY) **return**; // If already at destination, no movement needed

// Calculate movement in the x and y directions

**int** dx = destX - x, dy = destY - y;

**if** (dx > 0) newX++;

**else** **if** (dx < 0) newX--;

**if** (dy > 0) newY++;

**else** **if** (dy < 0) newY--;

// Ensure the new position is not occupied

**if** (WorldMap.*isOccupied*(newX, newY)) **return**;

WorldMap.*update*(x, y, ***EMPTY***); // Clear the current position on the map

**this**.posX = newX; // Update the animal's x-coordinate

**this**.posY = newY; // Update the animal's y-coordinate

WorldMap.*update*(newX, newY, type); // Mark the new position on the map

}

// Method to hunt for prey

**public** **void** hunt() {

**if** (isMating()) **return**; // Do not hunt if the animal is mating

// Find the nearest prey of the appropriate type

**int**[] prey = WorldMap.*findNearest*(**this**.posX, **this**.posY, type - 1);

Organism o = **null**;

**if** (prey[0] != -1 && prey[1] != -1) o = World.*getOrganismAt*(prey[0], prey[1], type - 1);

**if** (o != **null** && !o.isDead()) {

// Check if prey is within eating distance

**int** dx = prey[0] - **this**.posX, dy = prey[1] - **this**.posY;

**if** (Math.*abs*(dx) <= 1 && Math.*abs*(dy) <= 1) {

**int** en = o.energy / 10; // Calculate energy gained

**this**.energy += en; // Increase the predator's energy

**if** (**this**.type == Organism.***CARNIVORE***) Carnivore.*energyGet* += en; // Update carnivore stats

**else** **if** (**this**.type == Organism.***HERBIVORE***) Herbivore.*energyGet* += en; // Update herbivore stats

o.die(); // Kill the prey

state = ***ALIVE***; // Set the predator's state to active

}

move(prey[0], prey[1]); // Move closer to the prey

} **else** {

move(); // Move randomly if no prey is found

state = ***ALIVE***; // Set the predator's state to active

}

}  
  
// Method to move the animal in a random direction

**protected** **void** move() {

**if** (!isTimeToMove()) **return**; // Check if the animal is allowed to move

**int** [] direction = **this**.chooseDirection();

// Ensure the new position is within bounds of the world

**if** (direction[0] >= 0 && direction[0] < World.***WIDTH*** && direction[1] >= 0 && direction[1] < World.***HEIGHT***) {

move(direction[0], direction[1]); // Move to the new position

}

moveFrame = 0; // Reset the move frame counter

}

// Check if the animal is currently hunting

**protected** **boolean** isHunting() {

**return** **this**.state == ***HUNTING***;

}

// Check if the animal is currently mating

**protected** **boolean** isMating() {

**return** **this**.state == ***MATING***;

}

// Abstract method to check if the animal is allowed to move (based on move frames)

**protected** **abstract** **boolean** isTimeToMove();

}

Nhóm sử dụng các thuật toán đơn giản để cài đặt các phương thức kiếm năng lượng:

- Với Plant thì là tăng năng lượng trực tiếp.

- Với Animal, do động vật phải kiếm mồi để ăn nên động vật sẽ phải di chuyển đến vị trí con mồi trước nên nhóm cài đặt phương thức :

**+ protected** **void** move(**int** destX, **int** destY) để di chuyển tới vị trí xác định. Do mỗi lần di chuyển chỉ được đi tới các ô liền kề nên động vật sẽ lựa chọn ô trống liền kề gần nhất theo hướng di chuyển để di chuyển .

**+ protected** **void** move() để di chuyển tới ô trống liền kề ngẫu nhiên.

+ Các phương thức kiểm tra trạng thái isHunting() và isMating() để làm điều kiện thực thi các hàm hunt() và reproduce() để tránh nếu con vật đang kiếm mồi thì tự nhiên sinh sản và ngược lại.

+ 2 hàm move đều cùng **phạm vi truy cập, tên** nhưng khác nhau về **tham số** nên khi gọi hàm move, Java dựa vào tham số truyền vào để quyết định sẽ thực hiện hàm nào trong 2 hàm. Nếu không có tham số thì hàm move() sẽ được thực hiện. Còn nếu tham số là 2 số nguyên thì hàm còn lại được thực thi. Điều này khả thi do Java hỗ trợ **nạp chồng phương thức**.

+ Hàm isTimeToMove() để kiểm tra đã đến lúc di chuyển hay chưa.  
  
- Mọi sinh vật đều phải cập nhật trạng thái theo thời gian nên lớp Organism có phương thức trừu tượng update() và các lớp con Plant và Animal sẽ override như sau:

**public** **class** Plant **extends** Organism {

…  
  
 **public** **static** **int** *birthDeath* = 0; //Can be changed in controls

@Override

**public** **synchronized** **void** update() {

**if** (isDead()) {

// Ensure dead organisms don't continue with actions like hunting or reproducing

**return**;

}

**this**.produceEnergy();

**this**.reproduce();

}

…}

**public** **abstract** **class** Animal **extends** Organism {

…

// Reduces the animal's energy over time (simulating energy decay)

**public** **void** decayEnergy() {

**this**.energy -= 1; // Decrease energy by 1 per tick

}

// Method called each simulation tick to update the animal's state

@Override

**public** **void** update() {

**this**.decayEnergy(); // Reduce energy first

**if** (**this**.isDead()) {

**this**.die(); // Remove dead animals from the simulation

**return**;

}

moveFrame++; // Increment the frame counter for movement

// Determine the animal's state (hunting or mating)

**if** (state == ***ALIVE***) {

Random rand = **new** Random();

**if** (canReproduce()) {

**int** chance = rand.nextInt(100);

// Generate a random number between 0 and 99

**if** (chance < 50 + *birthDeath*) { //

state = ***MATING***;

} **else** {

state = ***HUNTING***;

}

}

**else** state = ***HUNTING***;

}

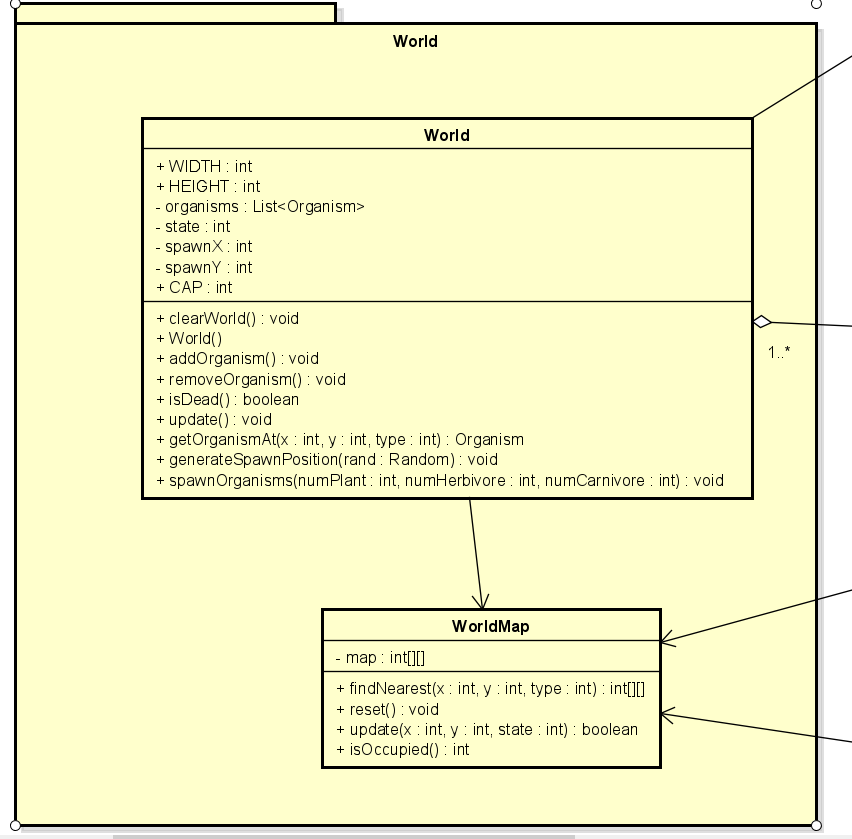
hunt(); // Attempt to hunt for food

reproduce(); // Attempt to reproduce if conditions are met

}

…  
  
}

* 1. Package World



Hệ sinh thái được biểu diễn bằng lưới hình chữ nhật. Nhóm quyết định tạo 2 lớp World và WorldMap để thuận tiện cho việc truy vấn và cập nhật.

* Lớp World lưu trữ các thông tin về chiều dài, chiều rộng (WIDTH, HEIGHT) được để cố định là 30x30, số lượng sinh vật tối đa (CAP ) cố định là 700, danh sách các sinh vật (organisms), trạng thái (state) và 2 thuộc tính spawnX, spawnY để code được gọn hơn. Lớp này thiên về việc cập nhật trạng thái nên các phương thức quan trọng sẽ được cài đặt như sau:

**public** **class** World {

**public** **final** **static** **int** ***DEAD*** = 0, ***HEALTHY*** = 1, ***CLEARED*** = 2;

**public** **final** **static** **int** ***WIDTH*** = 30, ***HEIGHT*** = 30;

**public** **final** **static** **int** ***CAP*** = 700;

**private** **static** List<Organism> *organisms*;

**private** **int** state;

**public** World() {

*organisms* = **new** ArrayList<>();

**this**.state = ***HEALTHY***;

}

**public** **static** Organism getOrganismAt(**int** x, **int** y, **int** type) {

**for** (Organism organism : *organisms*) {

**if**(organism.getPosX() == x && organism.getPosY() == y && organism.type == type && !organism.isDead())

**return** organism;

}

**return** **null**;

}

**public** **static** List<Organism> getOrganisms() {

**return** *organisms*;

}

**public** **boolean** isDead() {

**return** state == ***DEAD***;

}

**public** **void** update() {

**if** (*organisms*.isEmpty()) {

**this**.state = ***DEAD***;

**return**;

}

System.***out***.println("World IS ALIVE with " + *organisms*.size() + " organisms");

// Use an iterator to safely remove organisms while iterating

Iterator<Organism> iterator = *organisms*.iterator();

**while** (iterator.hasNext()) {

Organism o = iterator.next();

**if** (o.isDead()) {

System.***out***.println(o + " is dead");

Organism.*organismCount* --;

**if** (o.type == Organism.***PLANT***) {

Plant.*count*--;

Plant.*death*++;

} **else** **if** (o.type == Organism.***CARNIVORE***) {

Carnivore.*count*--;

Carnivore.*death*++;

} **else** **if** (o.type == Organism.***HERBIVORE***) {

Herbivore.*count*--;

Herbivore.*death*++;

}

iterator.remove(); // Safe removal during iteration

}

}

// Update remaining organisms

**for** (Organism o : *organisms*.toArray(**new** Organism[0])) { // Clone list to avoid concurrent modification exception

o.update();

}

}

**public** **synchronized** **static** **void** addOrganism(Organism organism) {

**if**(World.*isFull*()) **return**;

*organisms*.add(organism);

System.***out***.println(organism + " added");

}

**public** **synchronized** **void** removeOrganism(Organism organism) {

System.***out***.println(organism + " removed");

*organisms*.remove(organism);

}

**private** **int** spawnX, spawnY;

**public** **synchronized** **void** spawnOrganisms(**int** numPlant, **int** numHerbivore, **int** numCarnivore) {

clearWorld();

Random rand = **new** Random();

WorldMap.*reset*();

// Spawn plants

**for** (**int** i = 0; i < numPlant; i++) {

generateSpawnPosition(rand);

Plant newPlant = **new** Plant(200, spawnX, spawnY); // Initial energy of 100 for the plant

*addOrganism*(newPlant);

WorldMap.*update*(spawnX, spawnY, Organism.***PLANT***);

}

// Spawn herbivores

**for** (**int** i = 0; i < numHerbivore; i++) {

generateSpawnPosition(rand);

Herbivore newHerbivore = **new** Herbivore(spawnX, spawnY, 200); // Initial energy of 100 for

// herbivores

*addOrganism*(newHerbivore);

WorldMap.*update*(spawnX, spawnY, Organism.***HERBIVORE***);

}

// Spawn carnivores

**for** (**int** i = 0; i < numCarnivore; i++) {

generateSpawnPosition(rand);

Carnivore newCarnivore = **new** Carnivore(spawnX, spawnY, 200); // Initial energy of 100 for

// carnivores

*addOrganism*(newCarnivore);

WorldMap.*update*(spawnX, spawnY, Organism.***CARNIVORE***);

}

}

**private** **void** generateSpawnPosition(Random rand) {

**do** {

spawnX = rand.nextInt(***WIDTH***);

spawnY = rand.nextInt(***HEIGHT***);

} **while** (WorldMap.*isOccupied*(spawnX, spawnY));

}

**public** **synchronized** **void** clearWorld() {

*organisms* = **new** ArrayList<>(); // Turns out the clear() method keeps the list in memory, causing memory leaks

state = ***CLEARED***;

Organism.*resetStats*();

}

**public** **static** **boolean** isFull() {

**return** *organisms*.size() >= ***CAP***;

}

}

Ngoài các phương thức để thêm bớt sinh vật trong danh sách, lớp có một phương thức chủ đạo là update() đảm nhận công việc cập nhật trạng thái thế giới với các bước sau:  
+ Kiểm tra thế giới đã “chết” hay chưa (số sinh vật <= 0), nếu chưa thì tiếp tục

+ Kiểm tra danh sách các sinh vật, nếu sinh vật đã chết thì xóa khỏi danh sách. Nhóm sử dụng iterator để duyệt, xóa để tránh lỗi ConcurrentModification, gây mất toàn vẹn dữ liệu khi nhiều luồng cùng sử dụng danh sách

+ Duyệt qua danh sách sinh vật và gọi phương thức update() của từng phần tử. Do không thêm bớt phần tử nên danh sách được chuyển qua mảng để thuận tiện.

Bên cạnh đó, các phương thức clearWorld() và spawnOrganism() cũng là các phương thức hỗ trợ khởi tạo giả lập. Phương thức getOrganismAt() truy vấn sinh vật ở vị trí xác định.

* Lớp WorldMap được tạo ra để truy vấn, cập nhật các ô trống trong thế giới. Nhóm quyết định để thuộc tính duy nhất của lớp là 1 mảng 2D có kích thước cố định với các phương thức truy vấn, cập nhật được cài đặt như sau:

**public** **class** WorldMap {

**private** **static** **int**[][] *map* = **new** **int**[30][30];

**public** **static** **int**[] findNearest(**int** x, **int** y, **int** type) {

**int** [] ans = **new** **int**[2];

ans[0] = ans[1] = -1;

**double** minDistance = Double.***MAX\_VALUE***;

**for** (**int** i = 0; i < World.***WIDTH*** ; ++i) {

**for**(**int** j = 0; j < World.***HEIGHT***; ++j) {

**if** (*map*[i][j] == type) {

**double** distance = Math.*sqrt*(Math.*pow*(x - i, 2) + Math.*pow*(y - j, 2));

**if** (distance < minDistance) {

minDistance = distance;

ans[0] = i;

ans[1] = j;

}

}

}

}

**return** ans;

}

**public** **static** **void** update(**int** x, **int** y, **int** state) {

**if**(x < 0 || y < 0 || x >= 30 || y >= 30) **return**;

*map*[x][y] = state;

}

**public** **static** **boolean** isOccupied(**int** x, **int** y) {

**return** x < 0 || x >= World.***WIDTH*** || y < 0 || y >= World.***HEIGHT*** || *map*[x][y] != Organism.***EMPTY***;

}

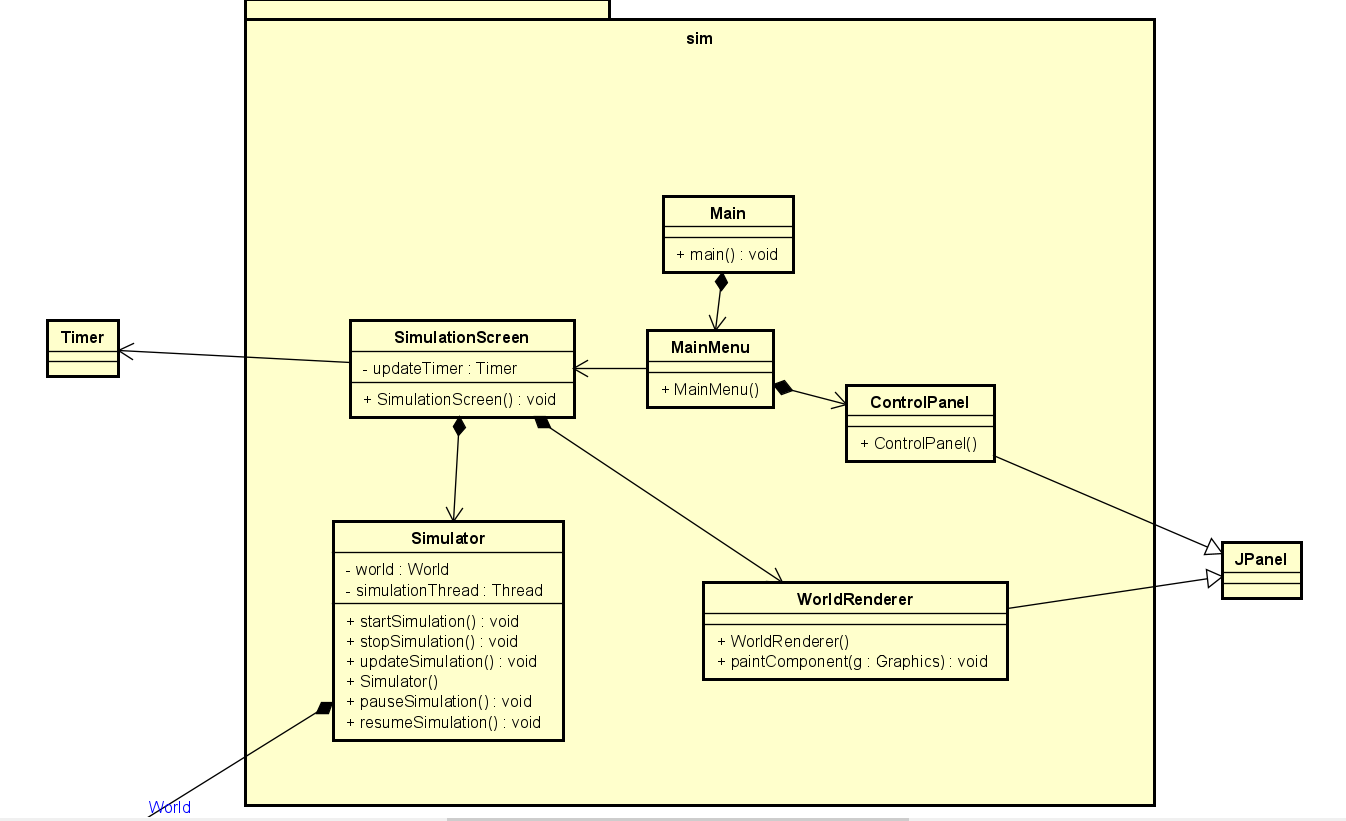
**public** **static** **void** reset() {

*map* = **new** **int**[30][30];

}

}

* 1. Package sim



Package sim chứa các lớp điều phối giả lập và giao diện người dùng:

-Lớp Simulator là lớp điều phối luồng cập nhật thế giới. Do bản chất giả lập tương tụ như game, cần update trạng thái qua từng frame, nhóm tham khảo Game Loop Pattern trên mạng và tiến hành cài đặt như sau:

**package** sim;

**import** World.World;

**public** **class** Simulator {

**public** **enum** SimulatorStatus {

***RUNNING***, ***STOPPED***, ***PAUSED***

}

**private** **volatile** SimulatorStatus status; // Simulation status (RUNNING or STOPPED)

**private** **final** World world; // The simulation world object

**public** Simulator() {

**this**.world = **new** World();

**this**.status = SimulatorStatus.***STOPPED***;

}

// Starts the simulation using a SwingWorker.

**private** Thread simulationThread;

**public** **void** startSimulation() {

**if** (simulationThread != **null** && simulationThread.isAlive()) {

System.***out***.println("Simulation is already running.");

**return**;

}

status = SimulatorStatus.***RUNNING***;

simulationThread = **new** Thread(() -> {

**try** {

**while** (status != SimulatorStatus.***STOPPED*** && !world.isDead()) {

**if** (status == SimulatorStatus.***PAUSED***) {

Thread.*sleep*(100); // Small delay to avoid busy-waiting

**continue**;

}

updateSimulation(); // Perform simulation logic (update world state)

// Delay for the simulation loop

Thread.*sleep*(500);

}

} **catch** (InterruptedException e) {

Thread.*currentThread*().interrupt();

System.***err***.println("Simulation thread interrupted.");

} **finally** {

// Ensure finalization after the thread ends

System.***out***.println("Simulation finished or paused.");

status = SimulatorStatus.***STOPPED***;

}

});

simulationThread.start(); // Start the simulation thread

System.***out***.println("Simulation started.");

}

// Stop simulation

**public** **void** stopSimulation() {

status = SimulatorStatus.***STOPPED***;

**if** (simulationThread != **null**) {

simulationThread.interrupt();; // Stop the SwingWorker task

}

System.***out***.println("Simulation stopped.");

}

// Check if running

**public** **boolean** isSimulationRunning() {

**return** status == SimulatorStatus.***RUNNING***;

}

// Update word

**protected** **void** updateSimulation() {

world.update(); // Update the world state

System.***out***.println("World updated.");

}

**public** **void** pauseSimulation() {

status = SimulatorStatus.***PAUSED***;

}

**public** **void** resumeSimulation() {

**if**(status == SimulatorStatus.***PAUSED*** ) status = SimulatorStatus.***RUNNING***;

}

}

Nhóm tạo một luồng riêng biệt để tính toán việc cập nhật thế giới để đồng bộ công việc với các lớp đồ họa

* Lớp WorldRenderer thực hiện công việc vẽ đồ họa cho thế giới là 1 lưới chữ nhật 30x30 với các sinh vật trong các ô

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.swing.border.Border;

**import** Organism.Carnivore;

**import** Organism.Herbivore;

**import** Organism.Organism;

**import** Organism.Plant;

**import** World.World;

**public** **class** WorldRenderer **extends** JPanel {

**private** **static** **final** **long** ***serialVersionUID*** = -4155452838523149843L;

**public** WorldRenderer() {

setPreferredSize(**new** Dimension(World.***WIDTH*** \* 20, World.***HEIGHT*** \* 20)); // scale the grid by cell size

// Add a border to the panel

Border border = BorderFactory.*createLineBorder*(Color.***BLACK***, 2); // Black border with width 2

setBorder(border);

}

@Override

**protected** **synchronized** **void** paintComponent(Graphics g) {

**super**.paintComponent(g); // Clear the background

// Set the grid size and cell size

**int** cellWidth = 20;

**int** cellHeight = 20;

// Enable anti-aliasing for smoother rendering

Graphics2D g2d = (Graphics2D) g;

g2d.setRenderingHint(RenderingHints.***KEY\_ANTIALIASING***, RenderingHints.***VALUE\_ANTIALIAS\_ON***);

// Draw the grid (optional, can be commented out for better performance)

g2d.setColor(Color.***GRAY***);

**for** (**int** i = 0; i <= World.***WIDTH***; i++) {

g2d.drawLine(i \* cellWidth, 0, i \* cellWidth, World.***HEIGHT*** \* cellHeight);

}

**for** (**int** i = 0; i <= World.***HEIGHT***; i++) {

g2d.drawLine(0, i \* cellHeight, World.***WIDTH*** \* cellWidth, i \* cellHeight);

}

// Get a snapshot of organisms to avoid concurrent modification

List<Organism> organismsSnapshot;

**synchronized** (World.**class**) {

organismsSnapshot = **new** ArrayList<>(World.*getOrganisms*());

}

// Draw the organisms efficiently

**for** (Organism organism : organismsSnapshot) {

// Set the color based on the organism type

**if** (organism **instanceof** Plant) {

g2d.setColor(Color.***GREEN***);

} **else** **if** (organism **instanceof** Herbivore) {

g2d.setColor(Color.***BLUE***);

} **else** **if** (organism **instanceof** Carnivore) {

g2d.setColor(Color.***RED***);

}

// Calculate position and size

**int** x = organism.getPosX() \* cellWidth;

**int** y = organism.getPosY() \* cellHeight;

// Draw the organism (emoji or circle as fallback)

String emoji = organism.getEmoji();

**if** (emoji != **null** && !emoji.isEmpty()) {

g2d.drawString(emoji, x + cellWidth / 4, y + cellHeight \* 3 / 4); // Emoji centered

} **else** {

g2d.fillOval(x + cellWidth / 4, y + cellHeight / 4, cellWidth / 2, cellHeight / 2); // Fallback

}

}

}

}

- Lớp SimulationScreen là lớp kết nối giữa việc tính toán giả lập và vẽ đồ họa lên màn hình:

**import** java.awt.BorderLayout;

**import** java.awt.Dimension;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.BorderFactory;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JOptionPane;

**import** javax.swing.JPanel;

**import** javax.swing.JTextArea;

**import** javax.swing.SwingUtilities;

**import** javax.swing.Timer;

**import** Organism.Animal;

**import** Organism.Carnivore;

**import** Organism.Herbivore;

**import** Organism.Plant;

**import** World.World;

**public** **class** SimulationScreen {

**private** Timer updateTimer;

**public** SimulationScreen() {

// Create the world simulation

JFrame frame = **new** JFrame("Ecosystem Simulator");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.setLayout(**new** BorderLayout());

World myWorld = **new** World();

// Create the view to render the world (30x30 grid)

WorldRenderer myView = **new** WorldRenderer();

// Panel for displaying stats

JTextArea statsArea = **new** JTextArea();

statsArea.setEditable(**false**);

statsArea.setBorder(BorderFactory.*createTitledBorder*("Stats"));

statsArea.setPreferredSize(**new** Dimension(300, 0)); // Adjust width as needed (300px for example)

// Create the controller for the simulation

Simulator mySim = **new** Simulator();

JButton pauseButton = **new** JButton("Pause Simulation");

JButton resumeButton = **new** JButton("Resume Simulation");

JButton quitButton = **new** JButton("Quit");

JPanel stopResume = **new** JPanel();

stopResume.add(resumeButton);

stopResume.add(pauseButton);

stopResume.add(quitButton);

// Pause button event listener

pauseButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

mySim.pauseSimulation();

statsArea.append("Simulation paused...\n");

}

});

// Resume button event listener

resumeButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

mySim.resumeSimulation();

statsArea.append("Simulation resumed...\n");

}

});

frame.add(stopResume, BorderLayout.***WEST***);

frame.add(myView, BorderLayout.***CENTER***); // Add the view (30x30 grid) here

frame.add(statsArea, BorderLayout.***EAST***);

frame.pack();

frame.setVisible(**true**);

**if** (updateTimer != **null**) {

updateTimer.stop();

updateTimer = **null**; // Nullify to avoid potential reuse

}

**try** {

// Parse input values from text fields

**int** plants = Integer.*parseInt*( ControlPanel.*fieldP*.getText());

**int** herbivores = Integer.*parseInt*( ControlPanel.*fieldH*.getText());

**int** carnivores = Integer.*parseInt*( ControlPanel.*fieldC*.getText());

// Clear the current world and spawn organisms

myWorld.clearWorld();

myWorld.spawnOrganisms(plants, herbivores, carnivores);

// Start the simulation (e.g., by invoking a controller method)

mySim.startSimulation();

statsArea.append("Simulation started...\n");

// Initialize the timer for periodic updates

updateTimer = **new** Timer(500, **new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

// Run the update in the EDT for thread-safety

SwingUtilities.*invokeLater*(() -> {

myView.repaint(); // Safely update the UI view

statsArea.setText( // Safely update the stats in the UI

"Plants: " + Plant.*count*+ "\n" +

"Herbivores: " + Herbivore.*count* + "\n" +

"Carnivores: " + Carnivore.*count* + "\n" +

"Energy from Sun to Plants: " + Plant.*energyGet*+ "\n" +

"Energy from Plants to Herbivores: " + Herbivore.*energyGet* + "\n" +

"Energy from Herbivores Carnivores: " + Carnivore.*energyGet* + "\n" +

"Plant birth rate: " + String.*format*("%.2f",(**double**)100\*Plant.*birth*/Plant.*total*) + "%\n"+

"Herbivore birth rate: " + String.*format*("%.2f", (**double**)100 \* Herbivore.*birth* / Herbivore.*total*) + "%\n" +

"Carnivore birth rate: " + String.*format*("%.2f", (**double**)100 \* Carnivore.*birth* / Carnivore.*total*) + "%\n" +

"Plant death rate: " + String.*format*("%.2f", (**double**)100 \* Plant.*death* / Plant.*total*) + "%\n" +

"Herbivore death rate: " + String.*format*("%.2f", (**double**)100 \* Herbivore.*death* / Herbivore.*total*) + "%\n" +

"Carnivore death rate: " + String.*format*("%.2f", (**double**)100 \* Carnivore.*death* / Carnivore.*total*) + "%\n"

);

});

}

});

updateTimer.start(); // Start the timer to begin updates

} **catch** (NumberFormatException ex) {

// Handle invalid number input gracefully

JOptionPane.*showMessageDialog*(

**null**,

"Please enter valid numbers for plants, herbivores, and carnivores.",

"Input Error",

JOptionPane.***ERROR\_MESSAGE***

);

}

// Quit button event listener

quitButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

**int** result = JOptionPane.*showConfirmDialog*(frame,

"Are you sure you want to quit?",

"Confirm Exit",

JOptionPane.***YES\_NO\_OPTION***);

**if** (result == JOptionPane.***YES\_OPTION***) {

mySim.stopSimulation();

frame.dispose(); // Close the current simulation screen

**new** MainMenu(); // Open the main menu

}

}

});

}

**protected** **static** **void** setBirthDeath(**int** x) {

Animal.*birthDeath* = x;

}

**public** **static** **void** main(String[] args) {

**new** SimulationScreen();

}

**public** **class** SimulationScreen {

**private** Timer updateTimer;

**public** SimulationScreen() {

// Create the world simulation

JFrame frame = **new** JFrame("Ecosystem Simulator");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.setLayout(**new** BorderLayout());

World myWorld = **new** World();

// Create the view to render the world (30x30 grid)

WorldRenderer myView = **new** WorldRenderer();

// Panel for displaying stats

JTextArea statsArea = **new** JTextArea();

statsArea.setEditable(**false**);

statsArea.setBorder(BorderFactory.*createTitledBorder*("Stats"));

statsArea.setPreferredSize(**new** Dimension(300, 0)); // Adjust width as needed (300px for example)

// Create the controller for the simulation

Simulator mySim = **new** Simulator();

JButton pauseButton = **new** JButton("Pause Simulation");

JButton resumeButton = **new** JButton("Resume Simulation");

JButton quitButton = **new** JButton("Quit");

JPanel stopResume = **new** JPanel();

stopResume.add(resumeButton);

stopResume.add(pauseButton);

stopResume.add(quitButton);

// Pause button event listener

pauseButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

mySim.pauseSimulation();

statsArea.append("Simulation paused...\n");

}

});

// Resume button event listener

resumeButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

mySim.resumeSimulation();

statsArea.append("Simulation resumed...\n");

}

});

frame.add(stopResume, BorderLayout.***WEST***);

frame.add(myView, BorderLayout.***CENTER***); // Add the view (30x30 grid) here

frame.add(statsArea, BorderLayout.***EAST***);

frame.pack();

frame.setVisible(**true**);

**if** (updateTimer != **null**) {

updateTimer.stop();

updateTimer = **null**; // Nullify to avoid potential reuse

}

**try** {

// Parse input values from text fields

**int** plants = Integer.*parseInt*( ControlPanel.*fieldP*.getText());

**int** herbivores = Integer.*parseInt*( ControlPanel.*fieldH*.getText());

**int** carnivores = Integer.*parseInt*( ControlPanel.*fieldC*.getText());

// Clear the current world and spawn organisms

myWorld.clearWorld();

myWorld.spawnOrganisms(plants, herbivores, carnivores);

// Start the simulation (e.g., by invoking a controller method)

mySim.startSimulation();

statsArea.append("Simulation started...\n");

// Initialize the timer for periodic updates

updateTimer = **new** Timer(500, **new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

// Run the update in the EDT for thread-safety

SwingUtilities.*invokeLater*(() -> {

myView.repaint(); // Safely update the UI view

statsArea.setText( // Safely update the stats in the UI

"Plants: " + Plant.*count*+ "\n" +

"Herbivores: " + Herbivore.*count* + "\n" +

"Carnivores: " + Carnivore.*count* + "\n" +

"Energy from Sun to Plants: " + Plant.*energyGet*+ "\n" +

"Energy from Plants to Herbivores: " + Herbivore.*energyGet* + "\n" +

"Energy from Herbivores Carnivores: " + Carnivore.*energyGet* + "\n" +

"Plant birth rate: " + String.*format*("%.2f",(**double**)100\*Plant.*birth*/Plant.*total*) + "%\n"+

"Herbivore birth rate: " + String.*format*("%.2f", (**double**)100 \* Herbivore.*birth* / Herbivore.*total*) + "%\n" +

"Carnivore birth rate: " + String.*format*("%.2f", (**double**)100 \* Carnivore.*birth* / Carnivore.*total*) + "%\n" +

"Plant death rate: " + String.*format*("%.2f", (**double**)100 \* Plant.*death* / Plant.*total*) + "%\n" +

"Herbivore death rate: " + String.*format*("%.2f", (**double**)100 \* Herbivore.*death* / Herbivore.*total*) + "%\n" +

"Carnivore death rate: " + String.*format*("%.2f", (**double**)100 \* Carnivore.*death* / Carnivore.*total*) + "%\n"

);

});

}

});

updateTimer.start(); // Start the timer to begin updates

} **catch** (NumberFormatException ex) {

// Handle invalid number input gracefully

JOptionPane.*showMessageDialog*(

**null**,

"Please enter valid numbers for plants, herbivores, and carnivores.",

"Input Error",

JOptionPane.***ERROR\_MESSAGE***

);

}

// Quit button event listener

quitButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

**int** result = JOptionPane.*showConfirmDialog*(frame,

"Are you sure you want to quit?",

"Confirm Exit",

JOptionPane.***YES\_NO\_OPTION***);

**if** (result == JOptionPane.***YES\_OPTION***) {

mySim.stopSimulation();

frame.dispose(); // Close the current simulation screen

**new** MainMenu(); // Open the main menu

}

}

});

}

**protected** **static** **void** setBirthDeath(**int** x) {

Animal.*birthDeath* = x;

}

}

-Lớp ControlPanel có chức năng cho người dùng điều chỉnh thông số giả lập:

**public** **class** ControlPanel **extends** JPanel {

**private** **static** **final** **long** ***serialVersionUID*** = -4155452838523112843L;

**static** JTextField *fieldP*;

**static** JTextField *fieldH*;

**static** JTextField *fieldC*;

JLabel currentPlanLabel;

JLabel birthDeathLabel;

**static** **int** *birthDeathModifier* = 0;

**public** ControlPanel() {

setLayout(**new** GridLayout(12, 1, 5, 5));

// Labels and text fields for P, H, and C

JLabel labelP = **new** JLabel("Plant:");

*fieldP* = **new** JTextField("0");

JLabel labelH = **new** JLabel("Herbivore:");

*fieldH* = **new** JTextField("0");

JLabel labelC = **new** JLabel("Carnivore:");

*fieldC* = **new** JTextField("0");

currentPlanLabel = **new** JLabel("Current Plan: Custom");

birthDeathLabel = **new** JLabel("Birth and Death: Balanced");

labelP.setHorizontalAlignment(JTextField.***CENTER***);

labelH.setHorizontalAlignment(JTextField.***CENTER***);

labelC.setHorizontalAlignment(JTextField.***CENTER***);

// Add key listeners to the text fields

*fieldP*.addKeyListener(**new** FieldListener());

*fieldH*.addKeyListener(**new** FieldListener());

*fieldC*.addKeyListener(**new** FieldListener());

// Control buttons

JButton helpButton = **new** JButton("Help");

JButton choosePlanButton = **new** JButton("Choose plan ...");

JButton birthDeathButton = **new** JButton("Change Birth/Death Rate ...");

// Add components to the control panel

add(labelP);

add(*fieldP*);

add(labelH);

add(*fieldH*);

add(labelC);

add(*fieldC*);

add(choosePlanButton);

add(currentPlanLabel);

add(birthDeathButton);

add(birthDeathLabel); // Add it to your panel or layout

add(helpButton);

setVisible(**true**);

birthDeathButton.addActionListener(**new** ActionListener(){

**int** choose = 0;

@Override

**public** **void** actionPerformed(ActionEvent e){

String[][] plan = { { "0", "Balanced"}, { "20", "Higher birth rate"}, { "-20", "Higher death rate"} };

*birthDeathModifier* = Integer.*parseInt*(plan[choose][0]);

birthDeathLabel.setText("Birth and Death: " + plan[choose][1]);

choose = (choose + 1) % 3;

}

});

// Help button event listener

helpButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

JOptionPane.*showMessageDialog*(ControlPanel.**this**, "Help:\nStart - Start the simulation\nPause - Pause the simulation\nResume - Resume the simulation\nQuit - Quit the application\nPlants get energy from sunlight\nHerbivores consume plants\nCarnivores consume herbivores\nThe predators only get 10% energy of their preys");

}

});

// Choose plan button event listener

choosePlanButton.addActionListener(**new** ActionListener(){

**int** choose = 0;

@Override

**public** **void** actionPerformed(ActionEvent e){

String[][] plan = { { "100","10","1", "Balanced"}, { "0","100","1", "No plant"}, { "100","10","0", "No Carnivore"} };

*fieldP*.setText(plan[choose][0]);

*fieldH*.setText(plan[choose][1]);

*fieldC*.setText(plan[choose][2]);

currentPlanLabel.setText("Current plan: " + plan[choose][3]);

choose = (choose + 1) % 3;

}

});

}

**public** **class** FieldListener **extends** KeyAdapter **implements** ActionListener {

@Override

**public** **void** keyReleased(KeyEvent e) {

// Get the source of the event (the specific text field that triggered the event)

Object source = e.getSource();

// Only allow numbers in the text fields

**char** keyChar = e.getKeyChar();

// Check if the key pressed is not a digit and not backspace

// If the text field is empty after a key event, set it to "0"

**if** (source == *fieldP* || source == *fieldH* || source == *fieldC*) {

JTextField field = (JTextField) source;

String curr = field.getText();

System.***out***.println(curr);

// If the field is empty, set it to "0"

**if** (!Character.*isDigit*(keyChar) && keyChar != KeyEvent.***VK\_BACK\_SPACE***) {

**if**(field.getCaretPosition() == 1) field.setText("0");

**else** field.setText(curr.substring(0, curr.length() - 1));; // Consume the event to prevent invalid input

}

**if** (field.getText().isEmpty()) {

field.setText("0");

}

// If the user starts typing, the field should show the new number

**else** **if** (field.getText().equals("0" + keyChar) && Character.*isDigit*(keyChar) ){

field.setText(Character.*toString*(keyChar));

}

}

// Update the label if any of the fields have text

**if** (*fieldP*.getText().isEmpty() || *fieldH*.getText().isEmpty() || *fieldC*.getText().isEmpty()) {

currentPlanLabel.setText("Current plan: Custom");

}

}

@Override

**public** **void** actionPerformed(ActionEvent e) {

Object source = e.getSource();

JTextField field = (JTextField) source;

field.setCaretPosition(field.getText().length());

}

}

}

* Lớp MainMenu có chức năng hiển thị và cho người dùng lựa chọn trong Menu:

**import** java.awt.\*;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.\*;

**import** World.World;

**public** **class** MainMenu {

**public** MainMenu() {

// Create the JFrame for the main window

JFrame frame = **new** JFrame("Ecosystem Simulator");

frame.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

frame.setLayout(**new** BorderLayout(10, 10)); // Add padding between components

// Create the control panel (buttons and inputs)

ControlPanel controlPanel = **new** ControlPanel();

// Panel for the menu buttons (start and quit)

JPanel menu = **new** JPanel();

menu.setLayout(**new** BoxLayout(menu, BoxLayout.***Y\_AXIS***)); // Vertical button layout

menu.setBorder(BorderFactory.*createEmptyBorder*(10, 10, 10, 10)); // Padding around buttons

// Add buttons for starting and quitting the simulation

JButton startButton = createButton("Start Simulation");

JButton quitButton = createButton("Quit");

menu.add(startButton);

menu.add(Box.*createVerticalStrut*(15)); // Add space between buttons

menu.add(quitButton);

// Add components to the JFrame

frame.add(menu, BorderLayout.***WEST***); // Add the menu (buttons) to the left

frame.add(controlPanel, BorderLayout.***EAST***); // Add the control panel to the right

frame.pack(); // Pack to fit the components

frame.setLocationRelativeTo(**null**); // Center the window on the screen

frame.setVisible(**true**); // Make the frame visible

// Start button event listener

startButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

**int** plants = Integer.*parseInt*(ControlPanel.*fieldP*.getText());

**int** herbivores = Integer.*parseInt*(ControlPanel.*fieldH*.getText());

**int** carnivores = Integer.*parseInt*(ControlPanel.*fieldC*.getText());

**if** (plants + herbivores + carnivores > World.***CAP***) {

JOptionPane.*showMessageDialog*(**null**, "Total number of organisms exceeds the limit of " + World.***CAP***,

"ILLEGAL ORGANISM NUMBER", JOptionPane.***ERROR\_MESSAGE***);

**return**;

}

SimulationScreen.*setBirthDeath*(ControlPanel.*birthDeathModifier*);

frame.dispose(); // Close the main menu frame

**new** SimulationScreen(); // Launch the simulation screen

}

});

// Quit button event listener

quitButton.addActionListener(**new** ActionListener() {

@Override

**public** **void** actionPerformed(ActionEvent e) {

**int** result = JOptionPane.*showConfirmDialog*(frame,

"Are you sure you want to quit?",

"Confirm Exit",

JOptionPane.***YES\_NO\_OPTION***);

**if** (result == JOptionPane.***YES\_OPTION***) {

System.*exit*(0); // Exit the application

}

}

});

}

//Create Button

**private** JButton createButton(String text) {

JButton button = **new** JButton(text);

button.setAlignmentX(Component.***CENTER\_ALIGNMENT***); // Center align the button

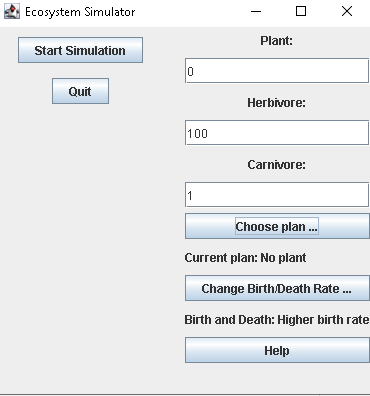
button.setPreferredSize(**new** Dimension(150, 40)); // Set preferred size

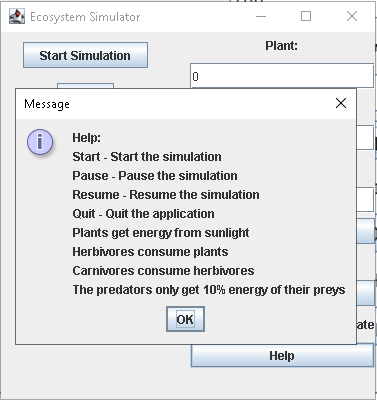
**return** button;

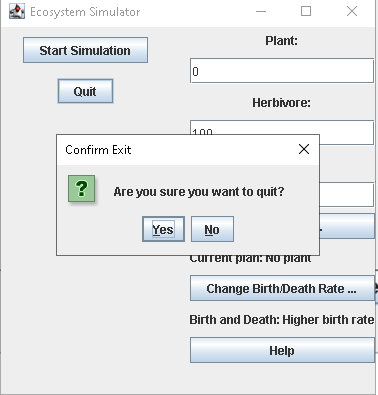
}

}  
  
  
-Nhóm cũng tạo 1 lớp Main() để chạy chương trình.

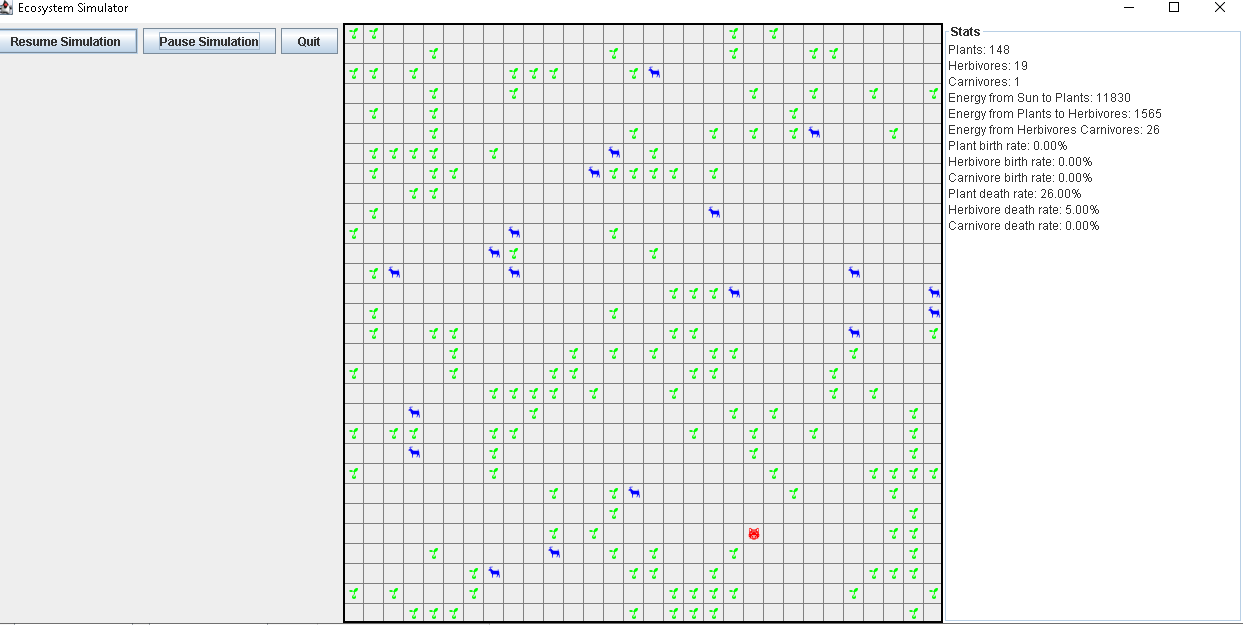
1. Demo  
   -Menu:

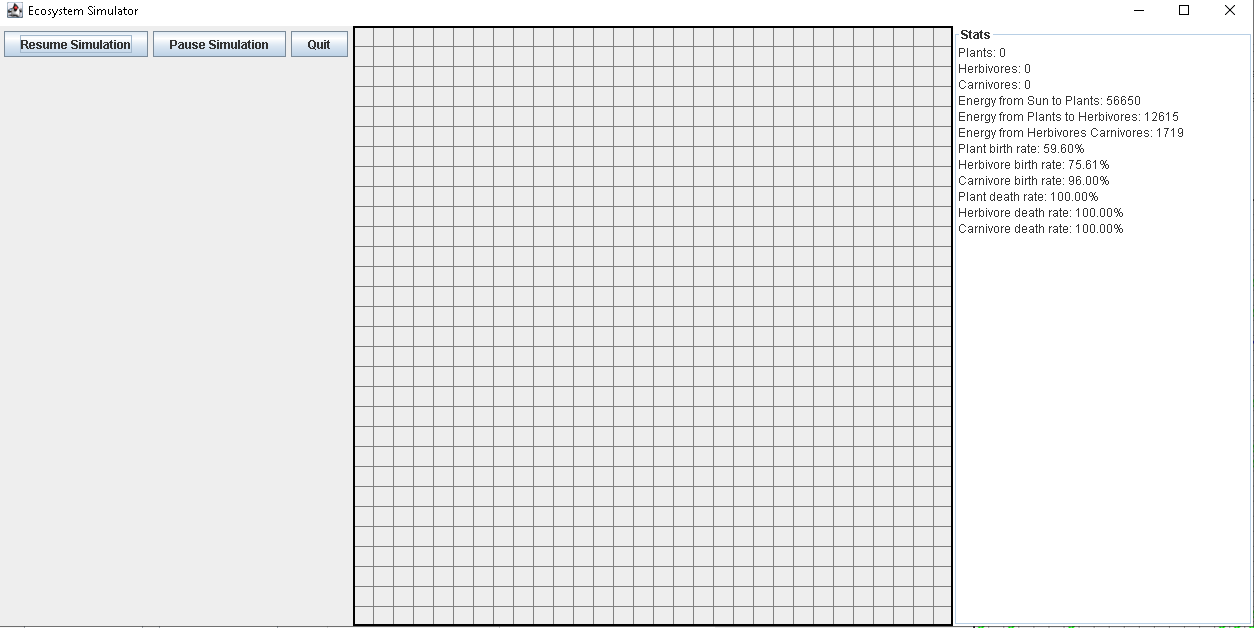






* Mô phỏng :





CHƯƠNG III: Kết luận

1. Kết quả thu được

* Áp dụng lý thuyết lập trình hướng đối tượng vào dự án
* Sử dụng thành thạo hơn ngôn ngữ lập trình Java
* Áp dụng các kiến thức về quản lý luồng, các pattern design hữu ích từ lập trình game vào dự án
* Từ quan sát mô phỏng, nhóm kết luận được rằng thế giới sẽ luôn hội tụ về 2 trạng thái, hoặc là tuyệt chủng hoàn toàn hay chỉ có sinh vật sản xuất sống sót và phát triển. Nguyên nhân là do quy luật 10% khi chỉ 10% năng lượng của sinh vật cấp thấp được chuyển qua cho sinh vật cấp cao hơn.

1. Hướng phát triển, cải thiện

* Thêm các chức năng mới như tăng, giảm tốc độ mô phỏng
* Viết mã nguồn gọn gàng, sạch sẽ và dễ bảo trì hơn

TÀI LIỆU THAM KHẢO

1.Bài giảng Lập trình hướng đối tượng và tài liệu Lab – TS. Nguyễn Thị Thu Trang

2. [Game Loop Pattern in Java: Mastering Smooth Game Mechanics](https://java-design-patterns.com/patterns/game-loop/#benefits-and-trade-offs-of-game-loop-pattern) – Java Design Pattern