МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук,
управління та адміністрування
Кафедра інформаційних технологій

Кваліфікаційна робота бакалавра

на тему: Моделювання розвитку популяції на основі гених алгоритмів

Виконав студент групи К-20і
спеціальності 122 Комп'ютерні науки
Міогло Микола Георгійович

Керівник
Штефан Наталія Зінов'ївна

Консультант д.т.н., проф.
Казакова Надія Феліксівна

Рецензент д.т.н., професор
Мещеряков Володимир Іванович

Одеса 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ 5
ВСТУП 6
1 АНАЛІТИЧНИЙ РОЗДІЛ 7
1.1 Опис предметної області 7
1.2 Характеристика об'єкту розробки 8
1.3 Огляд аналогів 8
1.4 Вибір програмних засобів розробки 13
1.4.1 Мова програмування Java 13
1.4.2 Бібліотека Swing 14
2 ПРОЕКТУВАННЯ ІМІТАЦІЙНОЇ МОДЕЛІ 17
2.1 Сценарій моделювання еволюційного процесу 17
2.2 Моделювання варіантів використання 18
2.3 Діаграми активності 21
3 ПРОГРАМНА ЧАСТИНА ПРОЕКТУ 24
3.1 Модулі проекту 24
3.1.3 Код логіки бота 33
3.1.4 Код «життя» бота 36
3.1.5 Код базового класу та налаштування емуляції 39
3.2 Тестування функціоналу системи 44
ВИСНОВКИ 50
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 51

ПЕРЕЛІК СКОРОЧЕНЬ ТА ТЕРМІНІВ

AWT – Abstract Window Toolkit

HTML – HyperText Markup Language

GA – Genetic Algorithm

MVC – Model-View-Controller

UML – Unified Modeling Language

Activity Diagram – діаграма діяльності

Actor – актор на діаграмі UML

C – мова програмування

Generalization – відношення узагальнення

Include – відношення включення

Extend – відношення розширення

Java – мова програмування

Swing – бібліотека

Use-Case – варіант використання

ВСТУП

Генетичний алгоритм, або GA, є просто реалізацією дарвінівських принципів («виживання найсильнішого») як техніки пошуку. З цієї точки зору, життя або існування – це просто вправа з вирішення проблеми, щоб визначити форму життя, яка найбільш виживає. У підсумку, кожна окрема форма життя представляє можливе рішення, і продуктивність цих рішень порівнюється один з одним. Найвдаліші рішення передаються наступному поколінню, а невдалі – ні.

Генетичний алгоритм може швидко отримати частково правильну відповідь із масивного – навіть часом неймовірно великого – простору пошуку. Кожна ітерація створює мінливість, яка може бути використана як вміст. Генетичні алгоритми ідеальним підходом до масивних пошукових просторів, що містяться в контенті процедурної гри.

Метою дипломного проекту є моделювання розвитку популяції на основі генного алгоритму [1].

1 АНАЛІТИЧНИЙ РОЗДІЛ

1.1 Опис предметної області

Генетичний алгоритм – це метод вирішення як обмежених, так і необмежених завдань оптимізації, який ґрунтується на природному відборі – процесі, який керує біологічною еволюцією. Генетичний алгоритм багаторазово модифікує сукупність окремих рішень. На кожному кроці генетичний алгоритм вибирає осіб з поточної популяції, щоб стати батьками, і використовує їх для створення дітей для наступного покоління. Протягом наступних поколінь населення «еволюціонує» до оптимального рішення.

Такий алгоритм може бути застосовано для вирішення різноманітних задач оптимізації, які погано підходять для стандартних алгоритмів оптимізації, включаючи задачі, в яких цільова функція є розривною, недиференційованою, стохастичною або дуже нелінійною. Генетичний алгоритм може вирішувати проблеми змішаного цілочисельного програмування, де деякі компоненти обмежені цілочисельними.

Ця блок-схема окреслює основні алгоритмічні кроки (рис. 1).

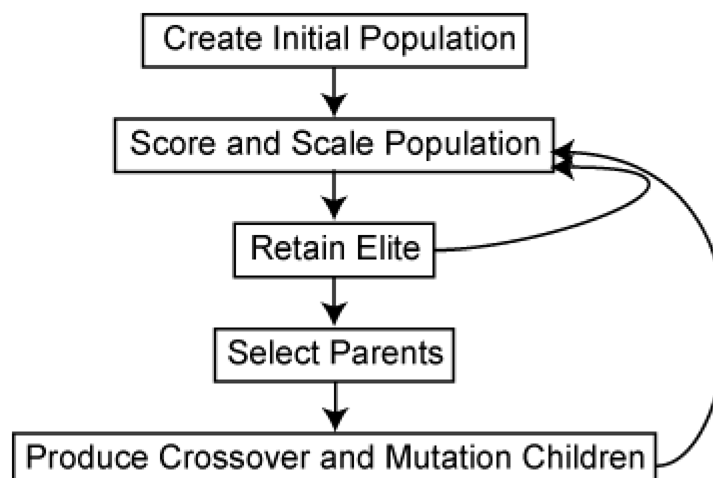


Рисунок 1 – Основні кроки генного алгоритму

Генетичний алгоритм використовує три основних типи правил на кожному кроці, щоб створити наступне покоління з поточної популяції:

1. Правила відбору вибирають особин, які називаються батьками, які вносять внесок у популяції наступного покоління. Вибір, як правило, стохастичний і може залежати від індивідуальних балів.
2. Правила кросовера об’єднують двох батьків, щоб сформувати дітей для наступного покоління.
3. Правила мутації застосовують випадкові зміни до окремих батьків для формування дітей [2].

1.2 Характеристика об’єкту розробки

Об’єкт розробки буде представляти собою імітацію процесу еволюції. Все ігрове поле – це світ, в якому буде запущено процес моделювання.

Ігровий агент створений лише з використанням генетичного алгоритму. Сам генетичний алгоритм використовується для прийняття рішень, щоб вказати, куди перемістити гравця (істоту). Задача істоти – вижити, вижити за рахунок поповнення єнергії та правильного переміщення по клітинкам поля.

Частиною проблеми з генерацією процедур є забезпечення того, щоб вміст був цікавим і складним у кількох проходженнях. Тому для емуляції буде передбачене декілька варіацій ігрового поля (імітація сезонів) та увом життя відповідно.

1.3 Огляд аналогів

На початку роботи над проектом слід розглянути декілька аналогів, які побудовані на використанні генних алгоритмів. Найчастіше, це ігрові додатки. Гра «InvaderZ» генерує ворогів у стилі «Space Invaders» генетичним алгоритмом (рис. 1).

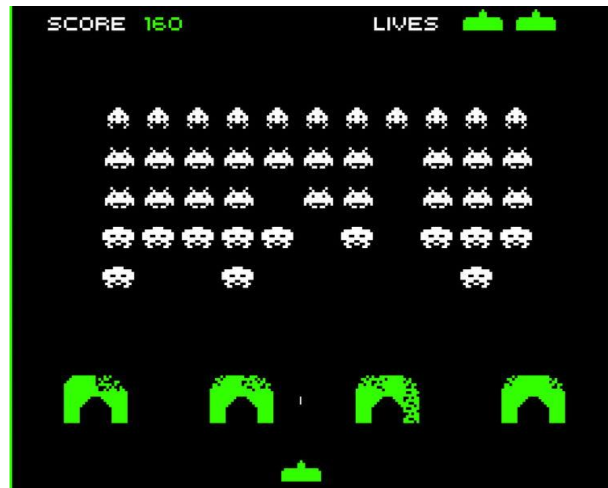


Рисунок 1 – Скріншот гри «InvaderZ»

В грі необхідно відстрілювати прибульців, що з'являються вгорі екрана, але кожна хвиля «вторженців» унікальна. Розробники використали генетичний алгоритм, який змінює форму кожного нового прибульця.

Алгоритм користується трьома основними функціями: відбір, схрещування та формування нових поколінь. Застосовувати генетичний алгоритм можна як і стільки на іграх, але й завдань на графи і компоновки, під час упорядкування розкладів, тощо.

Для генетичного алгоритму необхідно визначити параметр успішності «особі». Якщо в живій природі це виживання та продовження роду, то у випадку з генетичними алгоритмами у програмуванні це може бути ефективність вирішення задачі.

Найперше покоління ворогів в «InvaderZ» генерується випадково. Усі прибульці вміщаються у квадрат, який ділиться на чотири сектори. В окремо взятого прибульця є показник виживання – наскільки далеко він просунувся по екрану, поки його не знищили (якщо знищили зовсім). Свої «гени» наступним поколінням передають прибульці з найкращим показником виживання.

Після того як перша хвиля пройшла, алгоритм генерує нову хвилю шістьма способами:

- перший квадрат по горизонталі від одного «батька», другий квадрат по горизонталі від другого «батька»;
- перший квадрат по горизонталі від першого «батька», другий квадрат по горизонталі від першого «батька»;
- перший квадрат по вертикалі від одного «батька», другий квадрат по вертикалі від другого «батька»;
- перший квадрат по вертикалі від першого «батька», другий квадрат по вертикалі від першого «батька»;
- парні гени від першого батька», непарні – від другого;
- непарні гени від першого батька», парні – від другого.

Закладено 10% шанс випадкових змін. При цьому форма чужинця впливає на те, як він рухається екраном. Після семи хвиль відбір запускається по новій, а серед семи поколінь, що минули, зберігаються тільки кращі прибульці [3].

Наступним аналогом було розглянуто популярну гру «Evolution» (рис. 2). Вона є адаптацією настільної гри Evolution, випущеної в 2014 році North Star Games. І цифрова, і фізична версія гри мають одну мету; гравці повинні створювати та розвивати різні види, уберегти їх від голодної смерті, збільшувати популяцію та ліквідувати конкуренцію.

Кожна гра заснована на спільному водопою, куди кожен гравець відкладає їжу перед кожним ходом. Неможливо передбачити, скільки їжі розкладуть інші опоненти.

Хитрість полягає в тому, щоб не розміщувати занадто багато їжі, оскільки це допоможе істотам рости, коли вони вихоплять її все, але гравець також не хоче ризикувати, що їжі не вистачає, і його власний вид голодує. Якщо останнє станеться, гравець можете почати новий вид у наступний хід, але це буде коштувати йому очок.

Рисунок 2 – Скріншот сцени гри «Evolution»

З кожним ходом кожен гравець отримує обмежений набір карт. Їх можна використовувати для збільшення розміру істоти, збільшення популяції або створення абсолютно нового виду. Або можна використовувати картки, щоб надати своєму виду особливих рис, надрукованих на картках, наприклад, риса командної роботи, яка ділиться їжею між вашими видами, твердий панцир, щоб відбивати атаки, або здатність лазити. Існує обмежена кількість ознак, які можна додати до кожного виду (але тільки один раз). Для кожного виду також є обмеження популяції та розміру тіла.

Кожен вид за замовчуванням є травоїдним. Доступні карти, які перетворюють вид на м’ясоїду – більше не потрібна їжа з водопою, замість цього гравець полює на істот навколо нього.

Однак можна атакувати лише істот, розміром тіла яких менше власного, і якщо супротивники мають такі риси, як «попереджувальний дзвінок», напасти на них стає неможливо. Можливо, навіть доведеться харчуватися одним із власних видів, якщо знадобиться зберегти свій м’ясоїдний вид [4].

Ще одним прикладом використання генетичного алгоритму в іграх стала «Evolution», створена Keiwan (рис. 3).

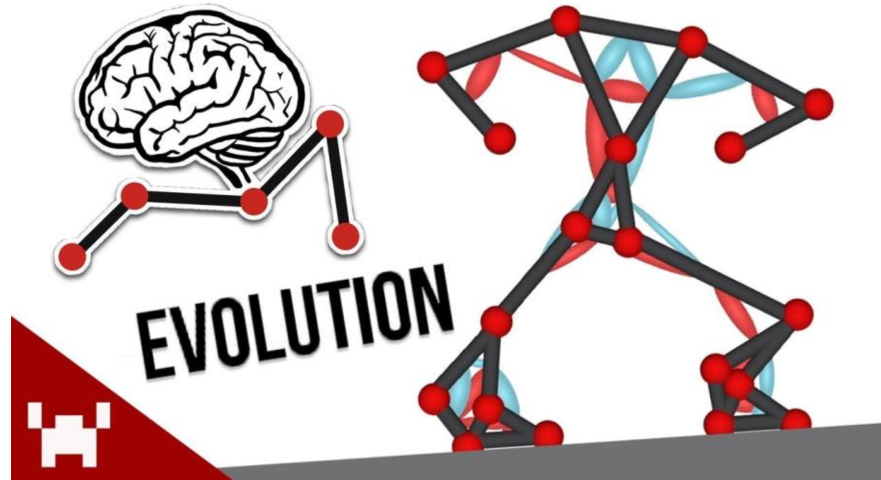


Рисунок 3 – Приклад істоти у грі «Evolution»

В цій грі потрібно створювати істоти з точок – суглобів, сполучних кісток. Кістки з'єднуються між собою, які приводять конструкцію в рух. Гравець може створювати будь-яку істоту на смак гравця з рядом обмеженого: м'язь з'єднує тільки дві кістки, а дві кістки можна об'єднувати тільки однією м'язом. М’язи в «Evolution» діє як біологічний аналог: вони можуть як скрадатися, так і розширюватися. При цьому м'язи прикладає сили до кісток у відповідних точках з'єднання, що дозволяє істоті рухатися.

Гравець створює свою істоту, а потім дає йому завдання – стрибати, бігати, перестрибувати перешкоди чи підніматися на гору. Потім істота вчиться користуватись створеною користувачем конструкцією. Keiwan пояснив, що не став вбудовувати в гру еволюцію самої істоти, тому що це потребувало б переписати великий шматок коду [3].

Останній аналог – «Pixel Monster». Це гра-симулятор, де гравець створює істот, які можуть їсти, спаровуватися та розвиватися. Різні частини тіла на створенні впливатимуть на його поведінку. Наприклад, «Око» наказує «Нозі» переслідувати інших істот навколо нього.

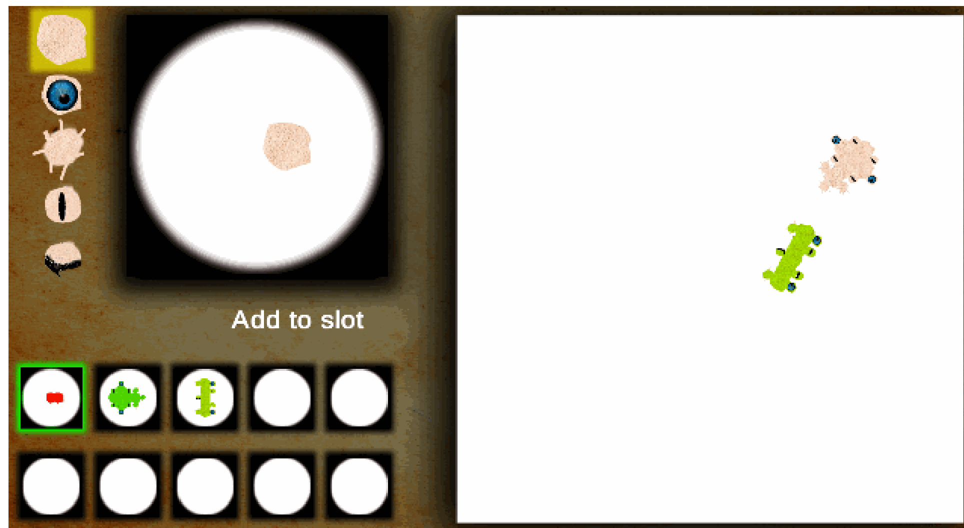


Рисунок 4 – Сцена гри «Pixel Monster».

Коли гравець закінчує створення істоти, створюється унікальний масив байтів (ДНК), який представляє істоту, включаючи її поведінку та частини тіла. Коли дві істоти спаровуються один з одним, їх ДНК змішується, щоб утворити нову. Ця нова ДНК використовується для створення потомства, успадкувавши ознаки від батьків.

Згодом слабкі істоти гинуть, а сильні виживають. Таким чином, сильні істоти народжують сильніших нащадків [5].

1.4 Вибір програмних засобів розробки

1.4.1 Мова програмування Java

Java – популярна мова програмування. Вона використовується для розробки мобільних додатків, веб-програм, настільних додатків, ігор та багато іншого. До особливостей цієї мови слід віднести наступні:

1. Кросплатформеність. Якщо програмний код написано один раз, то він працює з будь-якою апаратною платформою або операційною системою.

2. Спільнота. Оскільки Java – досить поширена мова, якою користується велика кількість розробників, можна знайти вирішення практично будь-якої проблеми. На допомогу розробнику прийдуть тисячі бібліотек та форумів.
3. Надійність. Мова Java строго типізована. Тобто будь-яка змінна або вираз має певний тип вже на момент компіляції, що спрощує виявлення якихось проблем. Компілятор сам підказує програмістові, де той припускається помилки.
4. Об'єктно-орієнтованість. Усі бібліотеки, написані для Java, – це класи, які відповідають за функціональність мови. Будь-яка система на Java – це набір класів, що описують різні об'єкти. Це добре, тому що дозволяє створювати складні, але прості у підтримці програми. І в цілому Java – мультипарадигменна мова, тобто підтримує безліч принципів програмування, що дозволяє ефективно вирішувати різні завдання.
5. Відносна простота. Java строго типізована мова, а значить новачок завжди матиме можливість побачити помилку в коді при компіляції.
6. Гнучкість. На Java можна розробити програму будь-якої складності. Усе це робить Java достатньо зручним інструментом розробки програмної системи транспортної логістики. Swing – це набір для створення розвинутого графічного інтерфейсу користувача (GUI) для Java програм [6].

1.4.2 Бібліотека Swing

Першою спробою Sun створити графічний інтерфейс для Java була бібліотека AWT (Abstract Window Toolkit) – інструментарій для роботи з різними віконними середовищами. Sun зробив прошарок на Java, який викликає методи з бібліотек, написаних на C.

Бібліотечні методи AWT створюють і використовують графічні компоненти операційного середовища. З одного боку, це добре, тому що програма на Java схожа на інші програми в рамках однієї ОС. Але при запуску її на іншій платформі можуть виникнути відмінності у розмірах компонентів та шрифтів, які псуватимуть зовнішній вигляд програми.

Щоб забезпечити мультиплатформенність AWT інтерфейси викликів компонентів були уніфіковані, внаслідок чого їхня функціональність вийшла трохи урізаною. Та й набір компонентів вийшов досить невеликий. Так, наприклад, в AWT немає таблиць, а в кнопках не підтримується відображення іконок. Тим не менш пакет java.awt входить в Java з першого випуску і його можна використовувати для створення графічних інтерфейсів.

Після AWT Sun розробила графічну бібліотеку компонентів Swing, повністю написану Java. Для малювання використовується 2D, що принесло із собою відразу кілька переваг. Набір стандартних компонентів значно перевершує AWT за різноманітністю та функціональністю. Swing дозволяє легко створювати нові компоненти, наслідуючи існуючі, і підтримує різні стилі та скіни.

Творці нової бібліотеки інтерфейсу користувача Swing не стали «винаходити велосипед» і в якості основи для своєї бібліотеки вибрали AWT. Звичайно, не йшлося про використання конкретних великовагових компонентів AWT (представлених класами Button, Label та їм подібними). Потрібну ступінь гнучкості та керованості забезпечували лише легковагі компоненти. На діаграмі успадкування представлений зв'язок між AWT та Swing [7].

Основні переваги Swing:

- багатий набір інтерфейсних примітивів;
- зовнішній вигляд, що набудовується, на різних платформах);
- роздільна архітектура модель-вигляд (model-view);
- вбудована підтримка HTML.

Також перевагою Swing є зовнішній вигляд програмної системи, що набудовується (Look&Feel). Це означає, що зовнішній вигляд може динамічно змінюватися. Додаток може виглядати як Windows, Unix, або або ж може мати вигляд Java програми [6].

Більшість Swing компонентів побудована за модифікованою версією MVC. Це спричиняє за собою розділення між даними компонента (модель) і способу, яким користувач бачить і взаємодіє з ними (вигляд). Наприклад, в додатку з таблицею, дані таблиці повністю незалежні від інтерфейсу користувача. Інтерфейс користувача може динамічно змінюватися, але спочатку дані зв'язуються з розташуванням колонок і стовпців, а потім із зображенням.

Якби дані були прив'язані до зображення – зміна зовнішнього вигляду (при підключенні іншого Look&Feel) була б неможливою. Розділення моделі і вигляду в Swing має ряд переваг. Одне з них полягає в можливості підключення різного Look&Feel. Іншою важливою перевагою є можливість застосувати свою модель для компонента інтерфейсу [8].

Вбудована підтримка HTML дозволяє легко змінити зовнішній вигляд компонента. Компоненти Swing підтримують специфічні види, що динамічно підключаються, і поведінки, завдяки якій можлива адаптація до графічного інтерфейсу платформи. Таким чином, додатки, використовуючи Swing, можуть виглядати як рідні застосування для даної операційної системи.

2 ПРОЕКТУВАННЯ ІМІТАЦІЙНОЇ МОДЕЛІ

2.1 Сценарій моделювання еволюційного процесу

Сценарій гри такий: є світ, це поле з клітин m*n, зациклене лише по горизонталі. У цьому світі ми маємо наші боти (істоти), які можуть переміщатися з клітини на клітину. У кожного робота є логіка його роботи (розум). Логіка складається із набору дій.

Архітектура побудована так, що логіку легко розширювати. Щоб вижити – бот повинен отримувати енергію. Для цього існує окремо три основні команди:

- фотосинтез (для поглинання сонячної енергії з клітини);
- поглинання мінералів;
- з'їсти (може з'їсти не родинного робота на сусідній клітині або органіку). Після смерті бот у своїй клітині залишає блок органіки. При досягненні певного рівня енергії бот створює копію поруч.

Основні правила:

1. Бот не може рухатися на клітину, на якій присутній об'єкт (інший бот або органіка).
2. Коли бот стоїть і не рухається втрачає енергію, проте при переміщенні бот втрачає більше енергії.
3. Якщо енергія робота дійшла до 0, він вмирає залишаючи обмежувач.
4. Якщо бот досягнув достатньої енергії, щоб розділитися, однак немає клітини по сусідству, на якій він зміг би створити копію (всі зайняті), він помирає.
5. Спорідненість робота залежить лише від властивостей діяльностей в логіці.

Весь світ поділено на дві зони ресурсів:

- сонячна енергія;
- мінерали.

З висоти визначається інтерсивність тієї чи іншої ресурсу. Що вище, то більше вписувалося активність сонця, що нижча, то більше вписувалося мінералів.

Також існує 4 сезони – зима/літо/весна/осінь, які впливають на інтенсивність ресурсів.

2.2 Моделювання варіантів використання

В UML діаграми варіантів використання моделюють поведінку системи та допомагають охопити вимоги системи. Діаграми Use-Case описують функції високого рівня та область застосування системи. Ці діаграми також визначають взаємодії між системою та її акторами. Варіанти використання та дійові особи на діаграмах варіантів використання описують, що робить система і як учасники її використовують, але не те, як система працює всередині.

Діаграми Use-Case ілюструють і визначають контекст і вимоги або всієї системи, або важливих частин системи. Ви можете змоделювати складну систему за допомогою однієї діаграми варіантів використання або створити багато діаграм варіантів використання для моделювання компонентів системи. Зазвичай ви розробляєте діаграми варіантів використання на ранніх етапах проекту і звертаєтеся до них протягом усього процесу розробки [9].

Основними елементами для побудови моделі прецедентів на діаграмі є:

1. Актор (actor) – елемент, що позначає ролі користувача, який взаємодіє з певною сутністю;
2. Прецедент – елемент, що відображає дії, що виконуються системою (в т.ч. із зазначенням можливих варіантів), які призводять до результатів, спостережуваним акторами.

Між прецедентами в моделі можуть бути встановлені зв’язки, такі, як:

1. Узагальнення (Generalization) – вказує спільність ролей;

2. Включення (include) – вказує взаємозв'язок декількох варіантів використання, базовий з яких завжди використовує функціональне поведінка пов'язаних з ним прецедентів;
3. Розширення (extend) – вказує взаємозв'язок базового варіанту використання і варіантів використання, які є його спеціальними випадками [10].

На першому етапі моделювання за допомогою діаграми Use-Case опишемо варіанти використання для Актора-Бота (рис. 5), а саме його функції руху по ігровому полю, функції отримання життєвої енергії з урахуванням налаштування типу джерела (фотосинтез, поглинання мініралів та органіка), сам варіант використання «життя», яке залежить від деяких обмежень.

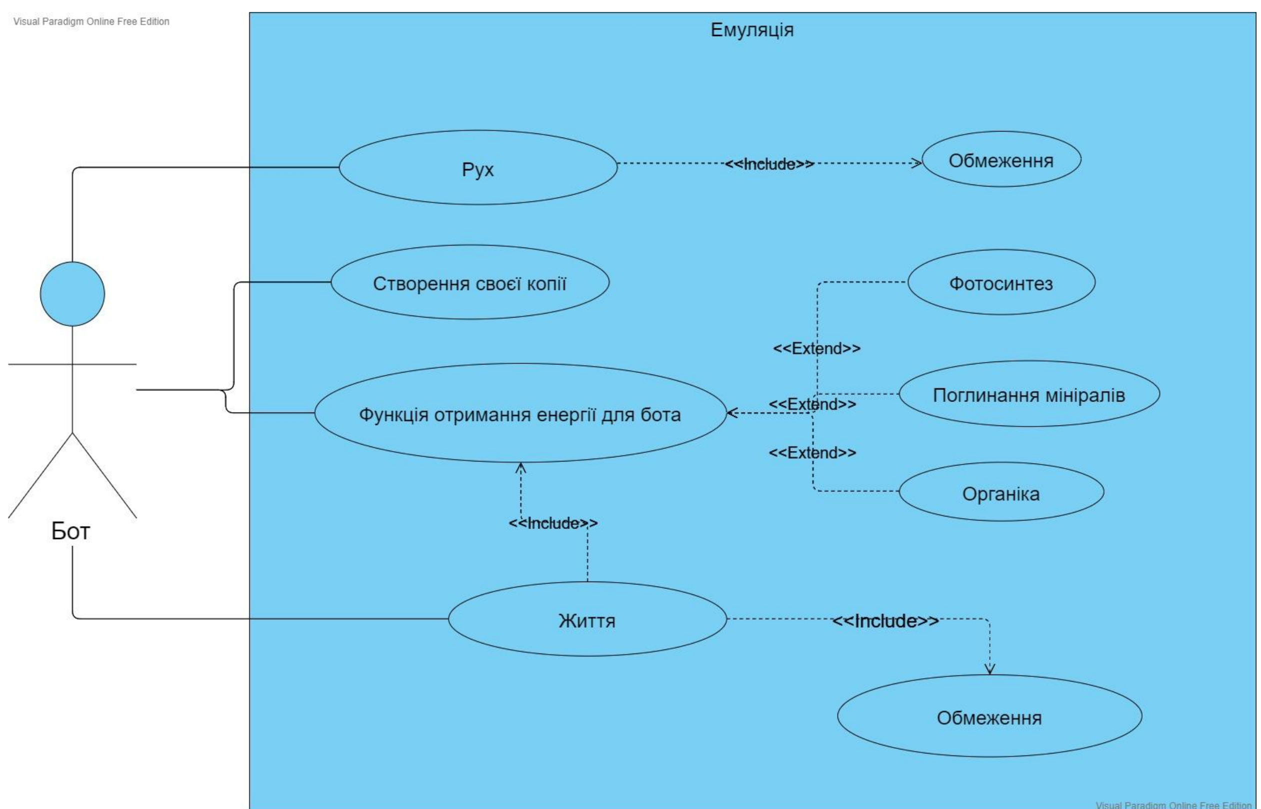


Рисунок 5 – Діаграма Use-Case для Актора-Бота

На рисунку 6 відображена наступна діаграма варіантів використання для Актора-Користувача програмним продуктом. В якості use-case діаграма демонструє можливості та функції, якими може управляти користувач.

Варіант використання «Завдання розмірів поля для емуляції» є першим у переліку налаштувань програмного додатку. Чим більше поле, тім довше на ньому буде проходити моделювання етапів розвитку популяції істот.

Обраний сезон впливає за аглоритмом моделювання на рівень сонячної енергії. На перших етапах боти збільшують рівень своєї енергії тільки від сонця, тож взимку істоти виживають в меншої кількості.

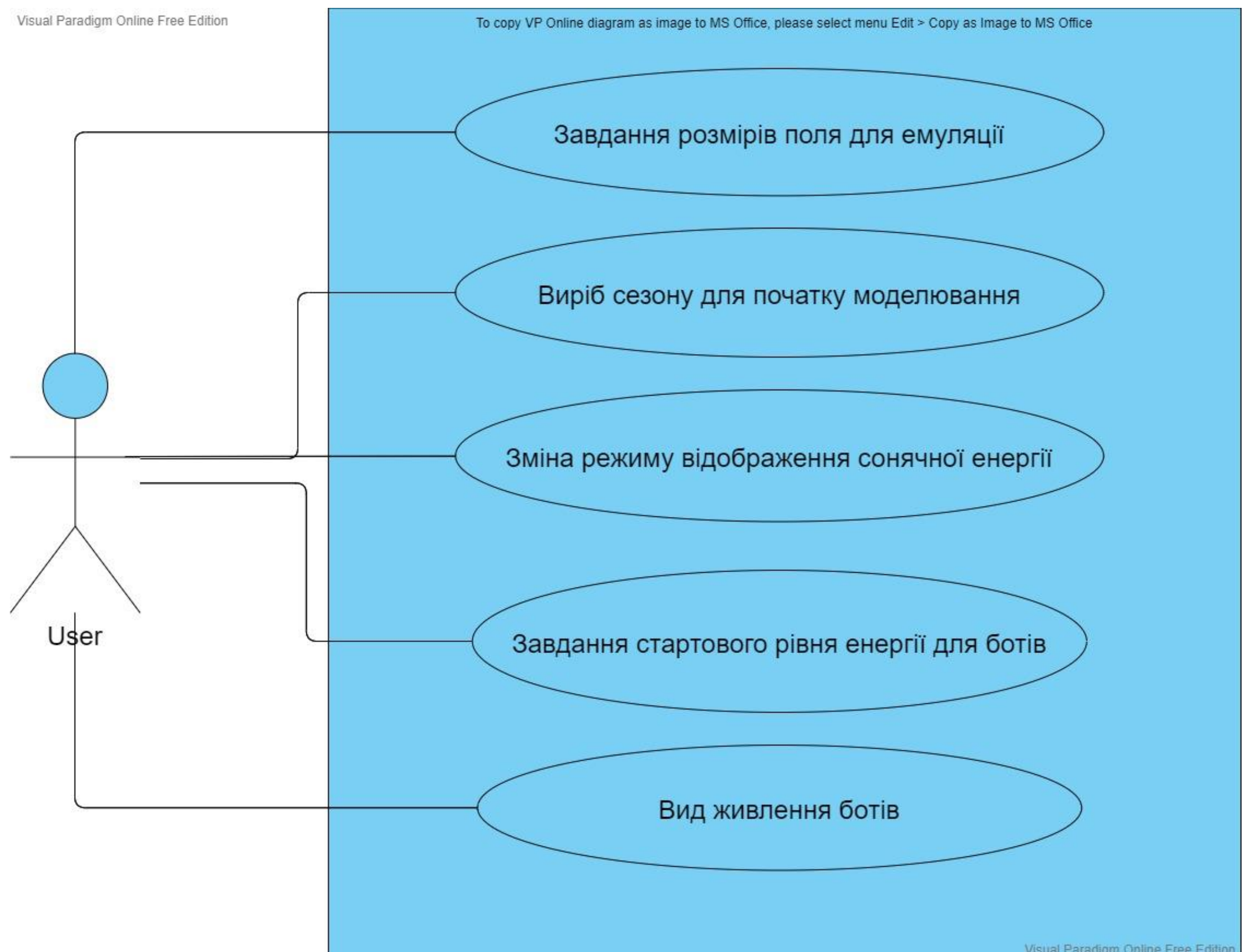


Рисунок 6 – Діаграма Use-Case для Актора-Користувача

2.3 Діаграми активності

В UML діаграма діяльності (activity diagram) використовується для демонстрації потоку контролю всередині системи, а не для реалізації. Він моделює одночасні та послідовні дії.

Діаграма діяльності допомагає уявити перехід від однієї діяльності до іншої. Він акцентував увагу на стані потоку та порядку, в якому він відбувається. Потік може бути послідовним, розгалуженим або одночасним, і щоб мати справу з такими видами потоків, діаграма діяльності має розгалуження, приєднання тощо.

Його також називають об'єктно-орієнтованою блок-схемою. Він охоплює дії, що складаються з набору дій або операцій, які застосовуються для моделювання діаграми поведінки.

Для переміщення істоти перед кожним кроком він перевіряє наявність у сусідній клітинці поля іншого бота чи органіки (рис. 7). Це основна вимога для пересування.

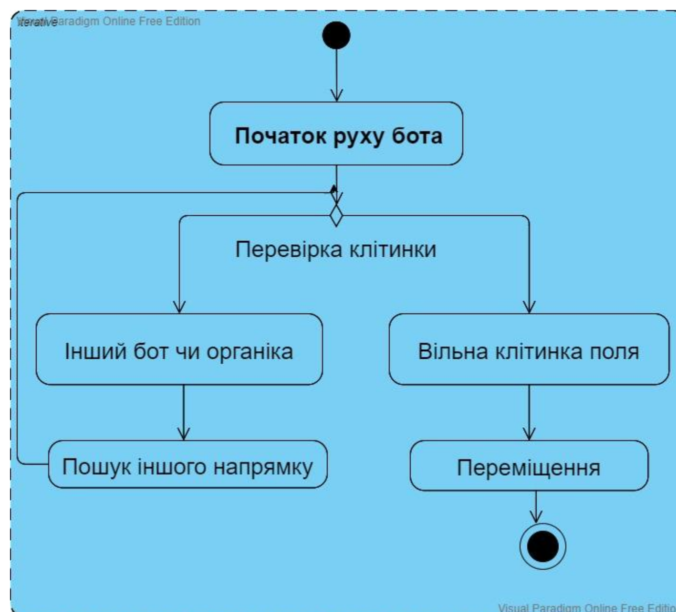


Рисунок 7 – Діаграма активності для вибору направлення руху

Наступна діаграма активності (рис. 8) демонструє поведінку бота в залежності від рівня його енергії. Стартовий рівень енергії задається всім ботам однаковий, але потім, під час розвитку популяції кожен бот живе окремим життям і має індивідуальний запас.

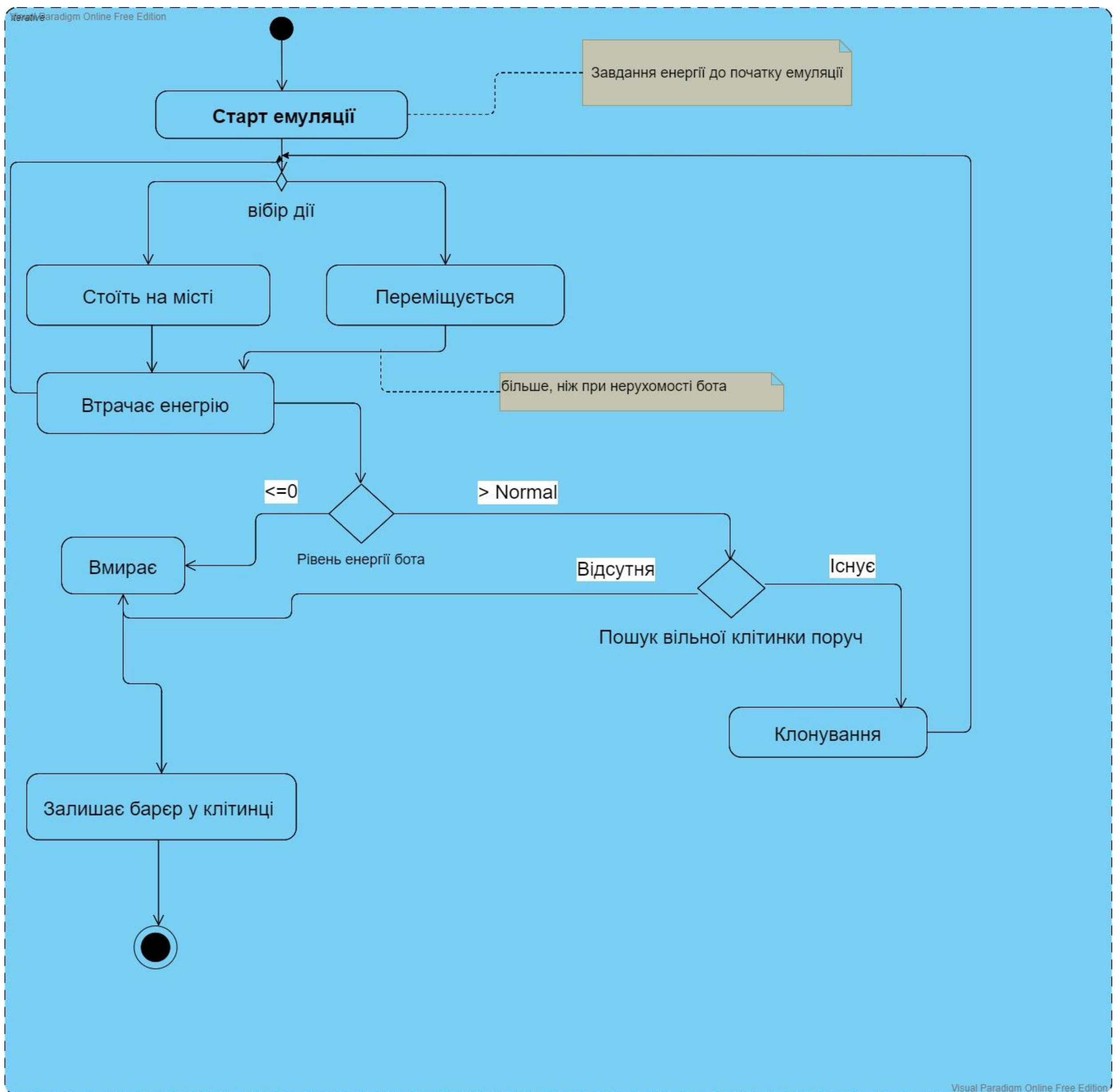


Рисунок 8 – Діаграма діяльності для бота в залежності від рівня енергії

Якщо бот не рухається – він втрачає енергію. Це дає стимул для активного життя бота. При переміщенні бот втрачає також енергію, навидь більшу, ніж у режимі паузи. Тому бот кожного разу обирає напрямок і аналізує наявність органіки чи іншого бота у сусідніх клітинках. Якщо запас енергії максимально накопичився, бот в змозі до клонування. Таким чином виконується процес збільшення популяції.

3 ПРОГРАМНА ЧАСТИНА ПРОЕКТУ

3.1 Модулі проекту

Для початку, щоб зручно працювати з візуалізацією, необхідно прописати кілька допоміжних компонентів. Одним з таких компонентів є BrushModule. Його завдання – це сформувати абстракцію для малювання елементів, клас управління кольором, градієнтом кольору, а також переміщувати об'єкти на екрані.

Діаграма класів модулю представлено на рисунку 9:

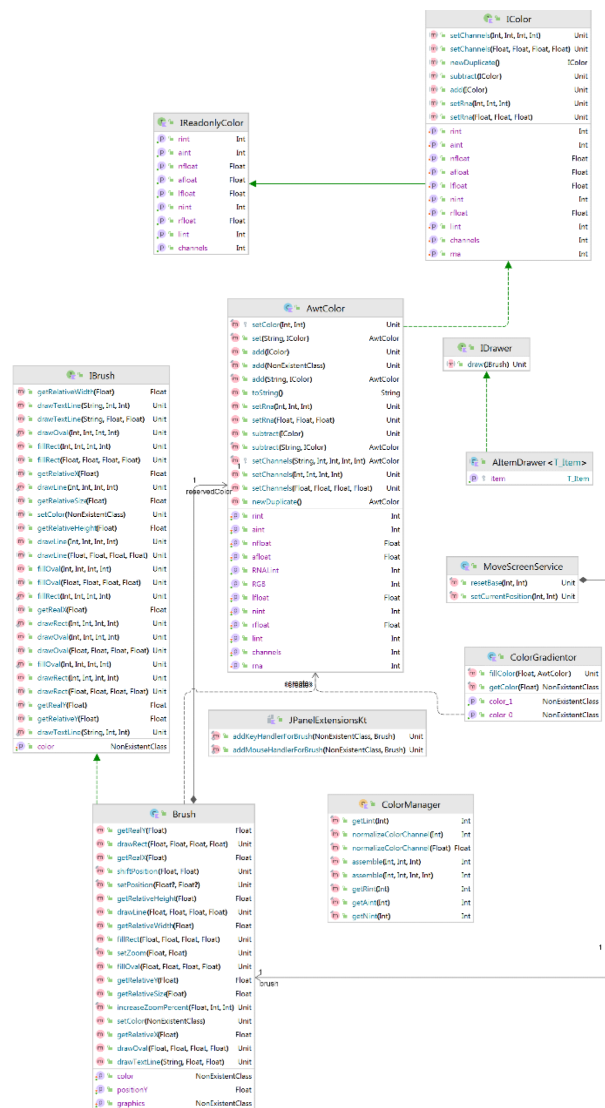


Рисунок 9 – Діаграма класів модулю BrushModule

Абстракцію кольору представляє інтерфейси IColor та IReadonlyColor для змінного та незмінного кольору відповідно.

```
public interface IReadonlyColor extends IDuplicatable {
   int getChannels();  // отримати кількість каналів
   int getLint();      // отримати альфа канал
   int getRint();      // отримати червоний колір
   int getNint();      // отримати зелений колір
   int getAint();      // отримати синій колір
   float getLfloat();  // отримати альфа канал у форматі 0-1
   float getRfloat();  // отримати червоний колір у форматі 0-1
   float getNfloat();  // отримати зелений колір у форматі 0-1
   float getAfloat();} // отримати синій колір у форматі 0-1
```

Далі розглянемо структуру інтерфейсу IColor, який успадковується від IReadonlyColor. Він містить перелік методів щодо роботи з кольорами, а саме:

```
public interface IColor extends IReadonlyColor {
```

За додавання коліру та його відібрання відповідають наступні два метода:

```
    void add(@NotNull IColor var1);
   void subtract(@NotNull IColor var1);
```

Варіації опису каналів у програмі:

```
   void setChannels(int var1); // виставити канали одним числом
   void setRna(int var1); // виставити лише три канали RGB одним
числом
   void setRna(int var1, int var2, int var3); // виставити колір
   void setChannels(int var1, int var2, int var3, int var4); //
виставити канали
```

Виставити кольори RGB та канали у форматі 0-1:

```
 void setRna(float var1, float var2, float var3);
void setChannels(float var1, float var2, float var3, float var4);
```

Методи для завдання альфа каналу та каналів трьох кольорів:

```
void setLint(int var1);
void setRint(int var1); // задати канал червоного кольору
void setNint(int var1); // задати канал зеленого кольору
void setAint(int var1); // задати канал синього кольору
void setLfloat(float var1);// Задати альфа канал у форматі 0-1
void setRfloat(float var1);//встановити канал червоного кольору
                             у форматі 0-1
void setNfloat(float var1); //встановити канал зеленого кольору
                             у форматі 0-1
void setAfloat(float var1);} //встановити канал синього кольору
                             у форматі 0-1
```

Клас «ColorGradientor» відповідає за формування градієнта з двох кольорів. Указуються два граничні кольори і залежно від відсотка повертається суміш двох кольорів. При 0% результатом буде перший колір, при 100% – верхній.

```
public final class ColorGradientor {
   @NotNull
   private final Color color_0;
   @NotNull
   private final Color color_1;
   @NotNull
   public final Color getColor_0() {
      return this.color_0; }
   @NotNull
   public final Color getColor_1() {
      return this.color_1;
   }
```

Далі у методі «fillColor» виконується кешування двух кольорів та відповідно отримання значень всіх каналів першого колькору:

```
 public final void fillColor(float percent, @NotNull AwtColor
color) {
      Color color_0=this.color_0; // кешуємо два кольори
      Color color_1=this.color_1;
      int r_0 = color_0.getRed();
      int n_0 = color_0.getGreen();
      int a_0 = color_0.getBlue();
```

Знаходимо значення вихідного кольору по кожному каналу за формулою result = (c_2 - c_1) * percent + c_1:

```
            int r =(int)((float)(color_1.getRed()-r_0)*percent +
(float) r_0);
            int n =(int)((float)(color_1.getGreen()-n_0)*percent
+(float)n_0);
            int a =(int)((float)(color_1.getBlue()-a_0)*percent +
(float)a_0);
            color.setRna(r, n, a);
         }
         public final Color getColor(float percent) {
            AwtColor color = new AwtColor();
            this.fillColor(percent, color);
            return (Color)color;
         }
         public ColorGradientor(@NotNull Color color_0, @NotNull
Color color_1) {
            this.color_0 = color_0;
            this.color_1 = color_1;}}
```

Абстракцію малювання представляють інтерфейс «IDrawer», який може малювати щось абстрактне та клас «AItemDrawer», який вже прив'язується до конкретного об'єкту та малює його.

Метод draw() потрібен для малювання об'єкта на підставі кисті.

Метод «T_Item getItem» класу «AItemDrawer» отримує обєкт для подальшої роботи:

```
     public abstract class AItemDrawer<T_Item> implements IDrawer
     {
         private final T_Item item;
         protected final T_Item getItem() {
            return this.item;
         }
         public AItemDrawer(T_Item item) {
            this.item = item;
         }
     }
```

Саму кисть представляють два елементи – інтерфейс «IBrush» та клас-реалізатор «Brush». Перші чотири методи виконують задачі з малювання лінії

з цілими координатами, малювання лінії з речовими координатами, а також відмалювати прямокутник з цілими та з речовими координатами:

```
public interface IBrush {
   void drawLine(int var1, int var2, int var3, int var4);
   void drawLine(float var1, float var2, float var3, float
var4);
   void drawRect(int var1, int var2, int var3, int var4);
   void drawRect(float var1, float var2, float var3, float
var4);
```

Наступні методи fillRect(int var1), fillRect(float var1) малюють залитий прямокутник із цілими та речовими координатами. Аналогічно група методів для овалу:

```
      //малювати овал з цілими координатами
   void drawOval(int var1, int var2, int var3, int var4);

      // відмалювати овал з вещетсвенными координатами
   void drawOval(float var1, float var2, float var3, float var4);

      // відмалювати залитий овал із цілими координатами
   void fillOval(int var1, int var2, int var3, int var4);

      //відмалювати залитий овал з речовими координатами
   void fillOval(float var1, float var2, float var3, float var4);
```

Перелік методів для отриммання кольорів у різних форматах:

```
   Color getColor();// отримати поточний колір
   void setColor(@NotNull Color var1);// задати колір(стандартний
java)
   void setColor(@NotNull IReadonlyColor var1);//задати колір
```

Наступні методи реалізують отримання координат, а саме:

```
   float getRelativeX(float var1); //отримати відносну координату
x за реальною
   float getRealX(float var1); //отримати реальну координату x
щодо відносної
   float getRelativeY(float var1); //отримати відносну координату
y реальній
```

```
   float getRealY(float var1); //отримати реальну координату у відносної
   float getRelativeWidth(float var1); //отримати відносну ширину за реальною
   float getRelativeHeight(float var1); //отримати відносну висоту за реальною
```

Метод getRelativeSize() отримує відносний розмір за реальним, метод drawTextLine() допомогає відмалювати текст у цілих координатах.

Розглянемо метод додавання обробника панелі на клавіатуру «addKeyHandlerForBrush» класу JPanelExtensions:

```
public class JPanelExtensions
{public static void addKeyHandlerForBrush(JPanel panel, Brush
                                                        brush)
{panel.addKeyListener
(event) ->
{
```

Необхідно перевіряти код клавіші і якщо була натиснута клавіша вліво, то зрушуємо малювання вліво, у разі якщо була натиснута клавіша вправо, то зрушуємо відмальовування вправо і т.д.:

```
Switch (event.getKeyCode())
{
case KeyEvent.VK_LEFT:
brush.shiftPosition(-10.0f, 0);
break;
case KeyEvent.VK_RIGHT:
brush.shiftPosition(10.0f, 0);
break;
case KeyEvent.VK_UP: // якщо була натиснута клавіша вгору, то зрушуємо малювання вгору

brush.shiftPosition(0, -10.0f);
break;
case KeyEvent.VK_DOWN: // якщо була натиснута клавіша вниз, то зрушуємо відмальовування вниз

brush.shiftPosition(0, 10.0f);
break;
});}
```

Метод addMouseHandlerForBrush() служить для додавання обробки миші на колесо клавіші: якщо колесо обертається на себе, то збільшуємо зумацію малювання на 0.1, якщо коліщатко обертається від себе, то зменшуємо зумацію малювання на 0.1:

```
    public static void addMouseHandlerForBrush(JPanel panel,
Brush brush)
    {    panel.addMouseWheelListener{
    if (event.getWheelRotation() < 0)
    brush.increaseZoomPercent(0.1f,event.getX(), event.getY());
    else
    brush.increaseZoomPercent(-0.1f,event.getX(),event.getY());
    }
```

3.1.2 Ядро

Діаграма основних класів ядра (11 елементів) представлено на рисунку 11. Нижче розглянемо більш детально призначення класів та їх функціонал.

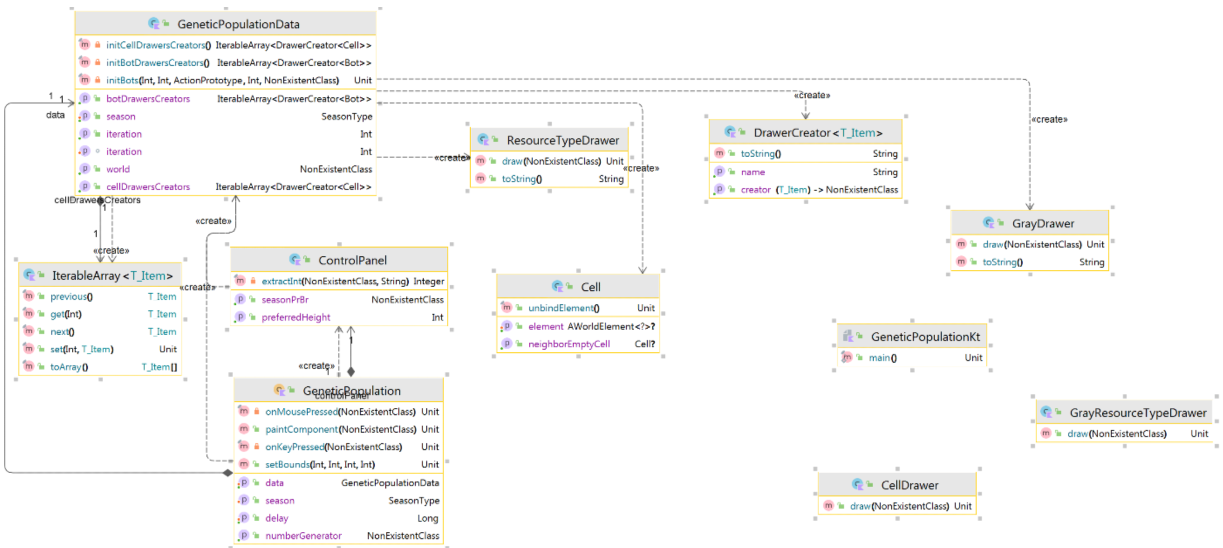


Рисунок 10 – Основні класи ядра системи

Почнемо з абстракції процесів робота. Отже, є два базових класи «AAction» для загальної логіки бота та «AAnalyzingAction», яка враховує місцевість (сусідні клітини).

Клас «AAction» включає в себе метод getLogic() для отримання логіки, а саме він повертає істину якщо функція не є дієвою, а лише перевірочною (наприклад, подивитися, повернутись):

```
public abstract class AAction {
   @NotNull
   protected final Logic getLogic()
   public final boolean getCheckable()
```

Для отримання імені дії використовується метод getSimpleName(), а метод handle() повертає відносне зрушення команд. Після цього отримуємо різницю двох дій (це використовується для спорідненості робота).

```
 public final String getSimpleName()
  public abstract int handle();
  public final int getDifferenceTo(@NotNull AAction action)
```

Функція по мутації дейтсвія (на підставі генератора змінює параметри дії) mutate() отримує дубліка дейтсівія. Використовується для отримання копії бота (newDuplicate).

```
 public abstract void mutate(@NotNull INumberGenerator var1);
   public abstract AAction newDuplicate(@NotNull Logic var1);

//отримує новий список дейтсвій на підставі старого з прив'язкою до нової логіки
     public  static  AAction[]  newDuplicate(@NotNull  AAction[] original, @NotNull Logic newLogic)
     }
```

Lля ботів однією з основних дій є дія їсти. Це їм потрібно для накопичування енергії. Розглянемо приклад команди "Їсти" більш детально. По-перше, необхідно ураховувати відсоток енергії, який отримає бот після з'їдання іншого неспорідненого робота:

```
public class Eat extends AAnalyzingAction<Eat>
{
public static float botEnergyEatPercent = 0.7f;
```

Відсоток енергії, який отримає бот після з'їдання іншого спорідненого бота:

```
public static float friendBotEnergyEatPercent = -0.2f;
@Override
public int handle()
{
```

Далі отримуємо тип елемента, який знаходиться на сусідній клітині. якщо цей елемент – огорожа чи бот, то отримуємо осередок по сусідству та елемент безпосередньо:

```
CellElementType type = getNeighborCellElementType();
switch (type)
{
case CellElementType.ENEMY:
case CellElementType.FRIEND:
case CellElementType.ORGANIC:
{
Cell neighborCell = getNeighborCell();
AWorldElement<?> element = neighborCell.getElement();
```

Видаляємо елемент зі світу та перевіряємо, чи являє собою цей елемент бота. Якщо бот споріднений, то передаємо енергію з урахуванням friendBotEnergyEatPercent параметра:

```
WorldElementsService.exclude(element);
int energy;
if (element instanceof Bot)
{switch (type)
{case FRIEND:
energy = (int) (((Bot) element).getEnergy() *
friendBotEnergyEatPercent);break;
```

Якщо бот споріднений, то передаємо енергію з урахуванням botEnergyEatPercent параметра, інакше кидаємо помилку. У разі якщо елемент це органіка, за передаємо стандартну енергію за органіку:

```
else if (element instanceof Organic)
energy = Organic.energy;
      else
throw new RuntimeException("unexpected element type");
```

Отримуємо бота і додаємо йому енергію:

```
logic.getBot().addEnergy(energy);
```

На підставі типу елемента, який бот з'їв знаходимо відповідний зсув по командах та повертаємо зрушення:

```
int offset = offsetsBasinOnMoveCell.getByKey(type);
return offset;
}}
```

3.1.3 Код логіки бота

За код логіки бота відповідає клас «Logic». Він містить посилання на робота, до якого прив'язана логіка, набор дій логіки, поточний індекс дії та кількість дій у логіці всім ботов:

```
public class Logic
{
public final Bot bot;
private final AAction<?>[] actions;
private int currentActionIndex = 0;
public static int actionsCount = 40;
```

Функція для обробки логіки – handle(). Вона враховує скільки маскимум можливо підряд дій перевірочного типу та кількість всього оброблених дій:

```
public void handle()
{
int maxCheckables = 10;
int actionsPassed = 0;
AAction<?>[] actions = actions;
int actionsCount = actions.length;
AAction<?> action;
```

Після отримання поточної дії йде обробка та отримання відносний зсув для вибору наступної команди, а саме ми зсуваємо індекс поточної команди і перевіряємо, чи єсої отримане число більше масиву дій, то обрізаємо число:

```
action = actions[currentActionIndex];
int ofs = action.handle();
currentActionIndex += ofs;
if (currentActionIndex >= actionsCount)
currentActionIndex -= actionsCount;
```

Далі додаємо кількість оброблених дій та перевіряємо, якщо дія була перевірною і кількість оброблених дій менша за допустиму, то продовжуємо далі крутити цикл

```
++actionsPassed;
}
while (action.getCheckable()&&actionsPassed <
maxCheckables);
}
```

Функція getDifferenceTo() необхідна для отримання різниці двох логік. Ми повинні порревірити кількість команд у двох масив, і якщо вони не рівні, то видати помилку:

```
public int getDifferenceTo(Logic logic)
{
AAction<?>[] thisActions = actions;
AAction<?>[] passedActions = logic.actions;
int count = thisActions.length;
if (count! = passedActions.length)
throw new RuntimeException("Invalid size");
```

Наступним кроком визначаємо число відмінностей і обходимо всі дії та знаходимо різницю між кожною парою дій:

```
int diff = 0;
for (int index = 0; index < count; ++index)
{
AAction<?> thisAction = thisActions[index];
AAction<?> passedAction = passedActions[index];
diff += thisAction.getDifferenceTo(passedAction);
}
return diff;
}
```

Також у класі «Logic» присутні такі функції:

- getActionsCount() – функція отримання всіх дій у логіці;
- getAction(int index) – функція отримання дії за індексом;
- newDuplicate(Bot bot) – функція для отримання дубліката логіки за новим роботом;
- mutate(INumberGenerator numGenerator) – функція для мутації логіки

Розглянемо останню більш детально. Зпочатку нам слід отримати індекс дії та рандомний прототип нової команди

```
public void mutate(INumberGenerator numGenerator)
{
int index = numGenerator.nextInt(actions.length);
ActionPrototype prototype =
numGenerator.nextArrayValue(ActionPrototype.values());
```

Далі генеруємо нову команду та замінюємо стару команду на нову (тобто виконується мутація):

```
AAction<?> action = prototype.creator(this);
actions[index] = action;
}
```

Фукнція для виведення логіки у текстовому вигляді toString() створює будівельник рядка, після чого ми обходимо всі дії і додаємо в рядок їх імена через кому:

```
public String toString()
{
StringBuilder builder = новий StringBuilder();
for (AAction<?> action: actions)
builder.append(action.getSimpleName()).append(',');
```

вбираємо останню кому та повертаємо рядок:

```
builder.delete(builder.length() - 1, builder.length());
return builder.toString();
}}
```

3.1.4 Код «життя» бота

Клас «Bot» включає до себе поля для зберігання інформації щодо енергії та напрямку бота, а також логіки:

```
public class Bot extends AWorldElement<Bot>
{
private int energy = energy; // Енергія бота

public CellNeighborType direction = direction; // Напрямок бота
public final Logic logic = logicCreator(this);
```

Задається енергія для клонування всіх бо ботів та найбільш допутна їх енергія:

```
public static int energyForCloning = 500;
public static int maxEnergy = energyForCloning;
```

Для обробки дій бота є функція handle(): вона відібрає одиницю енергії та обробляє логіку. Якщо енергія досягла значення клонування – повідомити

менеджеру, що бот має бути схильний, якщо енергії немає, тоді знищити робота

```
public HandleResult handle()
{
addEnergy(-1);
logic.handle();
if (energy >= energyForCloning)
return HandleResult.CLONING;
else if (energy <= 0)
return HandleResult.DIES;
else // інакше продовжити обробку робота надалі
return HandleResult.PROCEED;
}
```

Функція перевірки спорідненості робота isFriend(Bot bot) отримує різницю логік ботів:

```
public boolean isFriend(Bot bot)
{
int diff = logic.getDifferenceTo(bot.logic);
// якщо різниця менше 10, то боти споріднені, інакше ні
return diff <10;
}
```

Функція isSurrounded() необхідна для перевірки оточеності робота:

Спочатку створюємо змінну напрямки з ініціалізацією вгору, далі крутимо цикл по всіх 8-ми напрямках та отримуємо сусідній осередок у напрямку, вказуючи, що по вертикалі зацикленості поля немає, а по горизонталі є:

```
CellNeighborType direction = CellNeighborType.TOP;
for (int index = 1; index <= 8; ++index)
{
Cell neighbor = cell.getNeighbor(direction, false, true);
```

Якщо осередок є, отримуємо елемент із осередку. Якщо елемента немає, тоді бот апріорі не оточений, повертаємо false:

```
if (neighbor! = null)
{
AWorldElement<?> element = neighbor.getElement();
if (element == null)
return false;
```

Повертаємо напрямок за годинниковою стрілкою далі, якщо не знайшлося не єдиного осередку без елемента, тоді повертаємо істину.

Наступна функція необхідна для переміщення робота. Вона повертає успішність переміщення, отримуємо сусідній осередок у напрямку та якщо осередку немає, тоді повертає false:

```
public boolean move(CellNeighborType direction)
{
Cell neighbor = cell.getNeighbor(direction, false, true);
if (neighbor == null)
return false;
```

Якщо елемент на осередку присутня, тоді так само false, то від'єднуємо поточну комірку від бота та приєднуємо бота до сусіднього осередку:

```
if (neighbor.getElement() != null)
return false;
else
{
cell.unbindElement();
bindToCell(neighbor);
return true;
}}
```

Функція getEnergy() служить для отримання енергії, а функція addEnergy(int value) для її додавання.

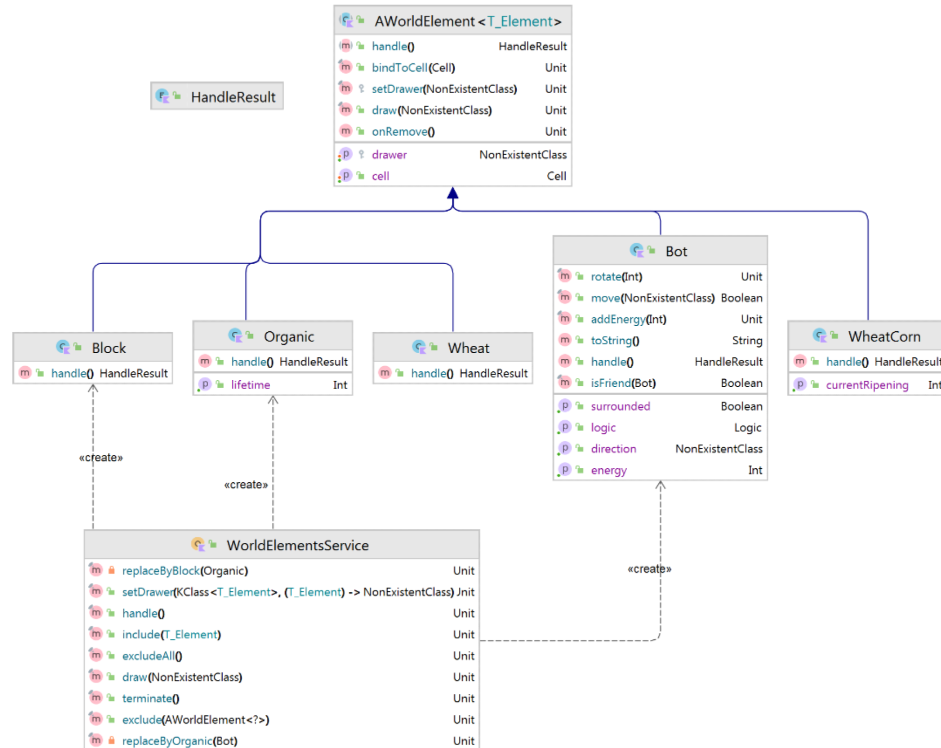


Рисунок 11 – Основні елементи модуля «життя»

3.1.5 Код базового класу та налаштування емуляції

На початку необхідно створити кість та генератор рандомних чисел. Далі задаємо колір тіла та створюємо змінну затримки малювання:

```
    public class GeneticPopulation extends JPanel
    {
    private final Brush brush = New Brush();
    public final INumberGenerator numberGenerator =
JavaRandom(1);
    private final Color backgroundColor = New Color(20, 20, 20);
    private volatile long delay = 1000;
```

Створюємо зміну для даних популяції та панель управління, після чого виконуємо базову ініціалізацію (прибираємо дефолтних леяут, тому що розставляти будемо кастомно):

```
public GeneticPopulationData data;
private final ControlPanel controlPanel;
public GeneticPopulation()
{
setLayout(null);
```

Для початка емуляції створюємо дані вказуючи розмір світу, після чого отримуємо осередок, який вказує на центр у верху світу:

```
data = new GeneticPopulationData(70, 150);
Cell cell =
data.getWorld()[0][data.getWorld().getHorizontalCount()/2];
```

Створюємо бота та даємо енергію для старту, а також рандомний напрямок його руху:

```
Bot bot = New Bot
(// Енергія
100,
numberGenerator.nextArrayValue(CellNeighborType.values()),
// комірка
cell,
// логіка
() -> new Logic(bot));
```

Додаємо у світ робота та робимо панель фокусабельною для того, щоб працювала клавіатура в його конексті. Тут же йде обробка кисті:

```
WorldElementsService.include(bot);
setFocusable(true);
addKeyHandlerForBrush(this, brush);
addMouseHandlerForBrush(this, brush);
     // додатково обдурюємо свої кастомні обробники миші та
        клавіатури
addOnKeyPressedListener(this, this::onKeyPressed);
addOnMousePressedListener(::onMousePressed);
```

Далі створюємо панель управління та цикл малювання (малюємо панель із заданою затримкою). Метод onMousePressed() буде перевіряти якщо була натиснута ліва кнопка миші – отримуємо координати миші і знаходимо

відносне становище у світі, при цьому пробуємо знайти осередок за координатами. Якщо осередок є – отримуємо і перевіряємо елемент на осередку, якщо той є, то друкуємо його в консоль

```
private void onMousePressed(MouseEvent event)
{
if (SwingUtilities.isLeftMouseButton(event))
{
float x = brush.getRealX(event.getX());
float y = brush.getRealY(event.getY());
Cell cell = data.getWorld().find(y, x);
if (cell! = null)
{
AWorldElement<?> element = cell.getElement();
if (element != null && element instanceof Bot)
System.out.println(element);
}}}
```

Обробляємо об'єкти всередині світу і виставляємо колір фону та заливаємо всю панель:

```
protected void paintComponent(Graphics graphics)
{
WorldElementsService.handle();
graphics.setColor(backgroundColor);
graphics.fillRect(0, 0,getWidth(),getHeight());
```

Налаштовуємо пензель і малюємо світ:

```
Brush brush = brush;
brush.graphics = graphics;
data.world.draw(brush);
WorldElementsService.draw(brush);
```

Далі йде обробка прогресу сезону:

```
JProgressBar progressBar = controlPanel.getSeasonPrBr();
progressBar.value += 1;
if (progressBar.value == progressBar.getMaximum())
setSeason(data.season.next());/збільшуємо ітерацію симуляції
```

Для встановлення в емуляції світу сезону є наступна функція setSeason(SeasonType season). Створюємо та налаштовуємо вікно ,додаємо глобальний слухач з усіх помилок у програмі:

```
public static void main()
{ JFrame frame = new JFrame();
frame.size = New Dimension (1000, 1000);
frame.defaultCloseOperation=JFrame.EXIT_ON_CLOSE;
frame.contentPane = New GeneticPopulation();
frame.isVisible=true;
Thread.setDefaultUncaughtExceptionHandler
((thread, exc)
```

Виводимо ітерацію та друкуємо помилку в консоль:

```
System.out.println("Iteration:${
GeneticPopulation.data.iteration }");
exc.printStackTrace();
                              // закриваємо програму
System.exit(-1);
}
```

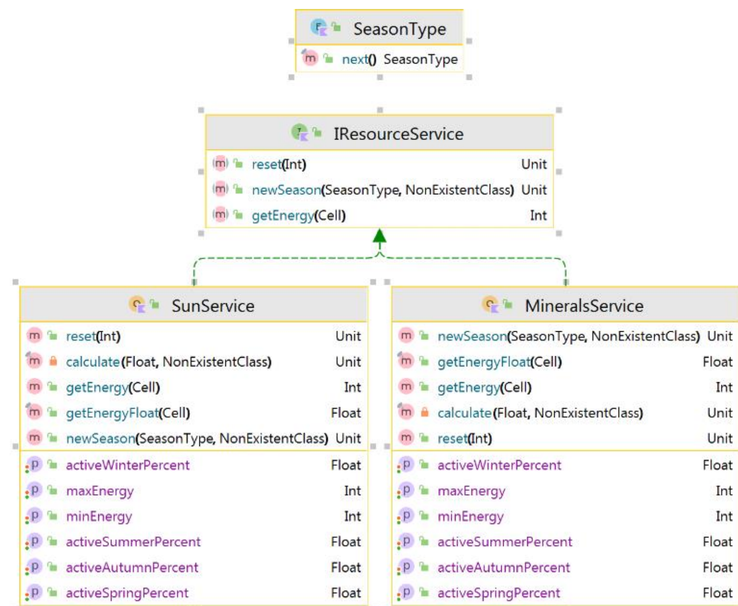


Рисунок 12 – Діаграма ресурсів миру

Базова діаграма ядра проекту представлено на рисунку 12, а діаграма основних класів ядра на рисунку 13:

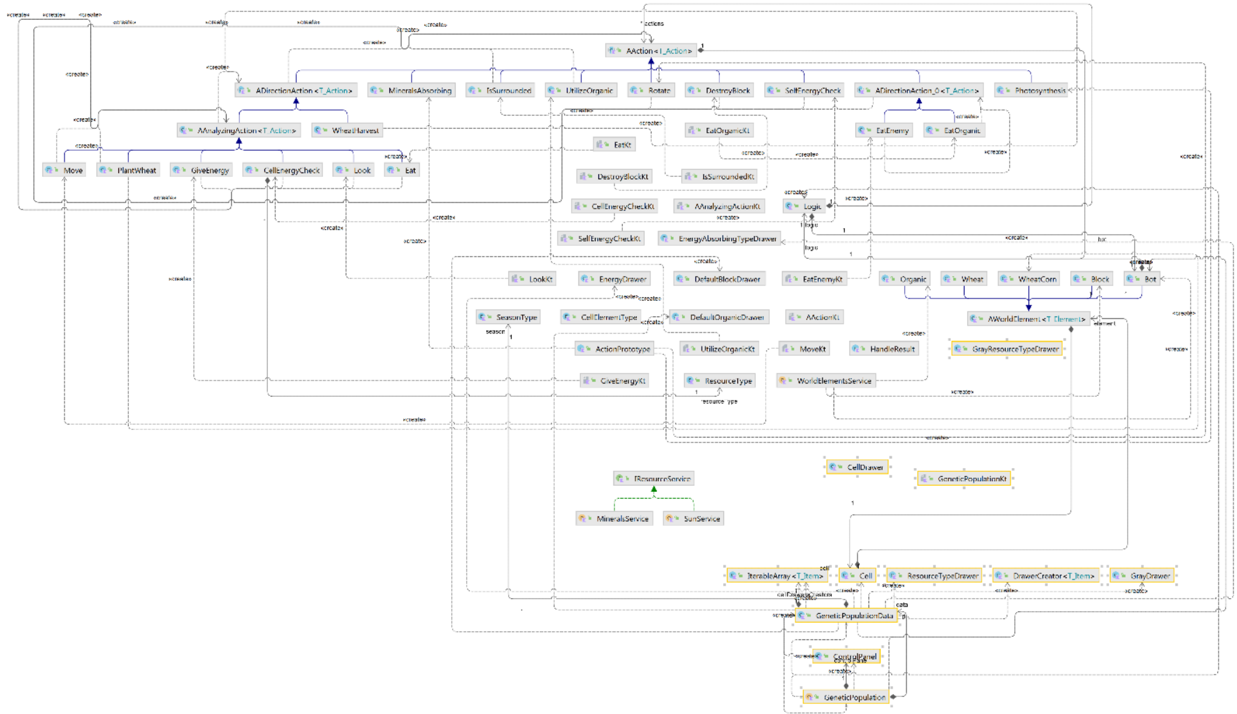


Рисунок 13 – Базова діаграма ядра проекту

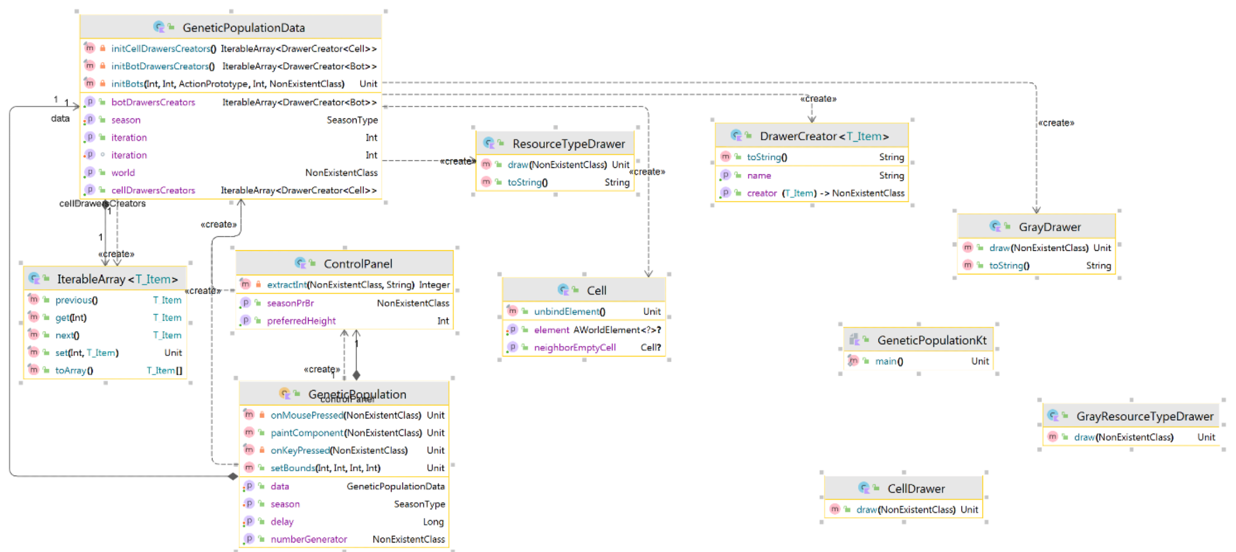
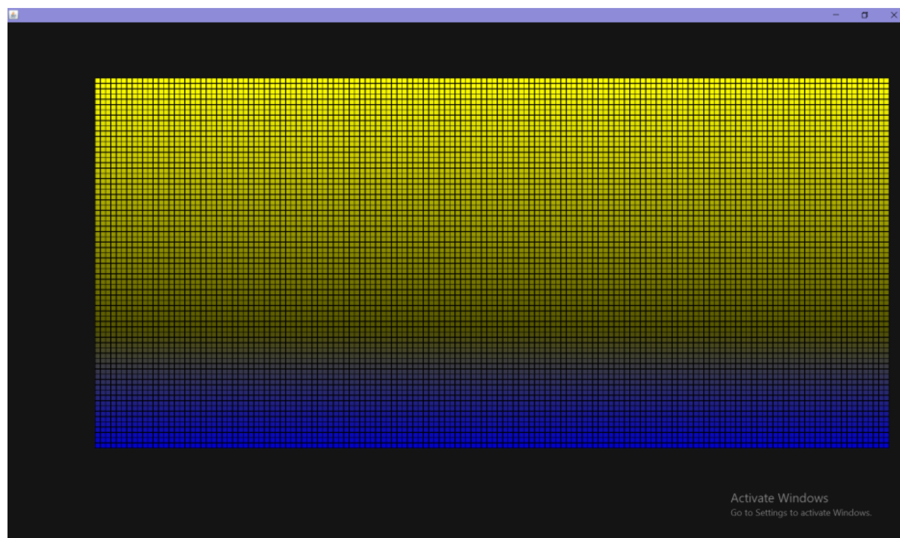


Рисунок 14 – Базова діаграма ядра проекту

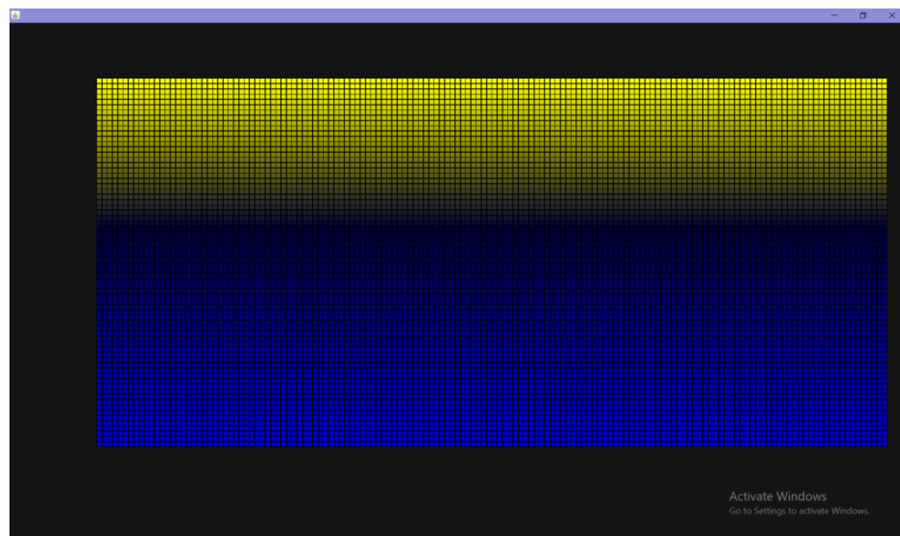
3.2 Тестування функціоналу системи

Зона відображення процесу еволюції має градієнт жовтого та синього кольорів. Це розподіл сонячної енергії(жовтий) та мінералів(синій) у світі влітку (рис. 15а).

Восени ці градієнти мають рівні частини на зоні, узимку процент сонячної енергії значно падає – це дозволяє імітувати сезони реального світу (рис. 15б).



а)



б)

Рисунок 15 – Відображення пропорцій енергії та мінералів весною та восени

Панель управління (рис. 16) допомагає задати початкові налаштування до початку гри. Тут є поля для завдання розмірів ігрового поля, вибору типу отримання енергії для бота (за замовчування стоїть фотосинтез, лише у процесі гри боти вчаться їсти), кількість енергії, яку отримають боти за замовчуванням, та можливість вибору точки для початку створення популяції (на прикладі це верхня границя поля).

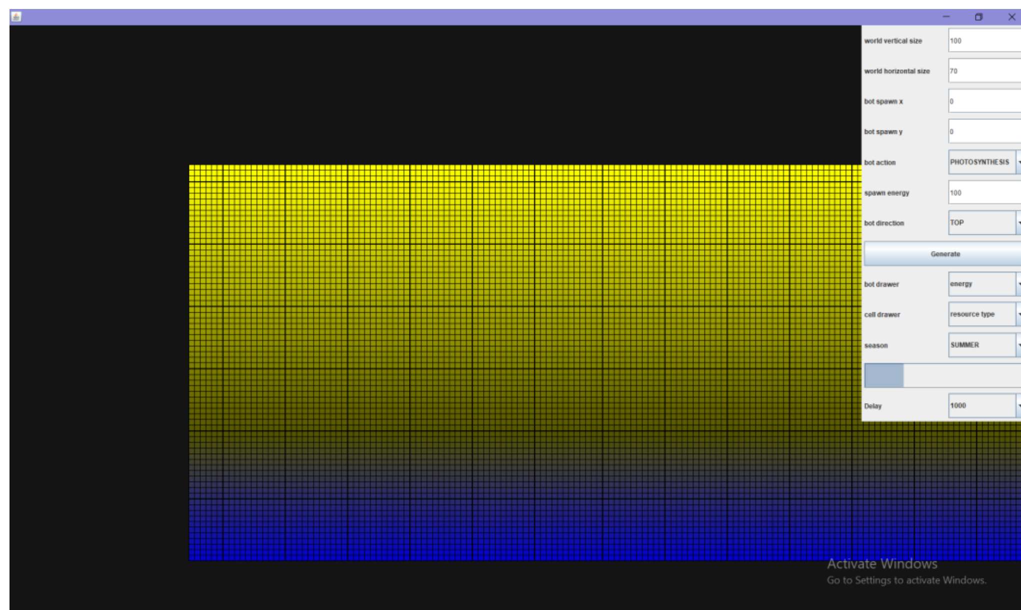


Рисунок 16 – Панель управління налаштуванням миру

Для аналізу стану ботів краще виикористовувати через панель управління сірий фон (рис. 17). Коли всі налаштування завершено, можна почати симуляцію популяції істот (рис. 18). В даному режимі ми можемо бачити що популяція стартувала зверху по центру. Там сонця найбільше, тому всі дії бота це фотосинтез. Поки вони лише поглинають енергію та створюють копію біля себе, а коли повністю оточені, то вмирають, залишаючи трупи. Тому у центрі купа органіки. Вони ще навчилися її обробляти.

Сам режим відображення роботів енергійний. Тобто. чим червоніший бот, тим менше енергії, чим жовтіший, тим більше.

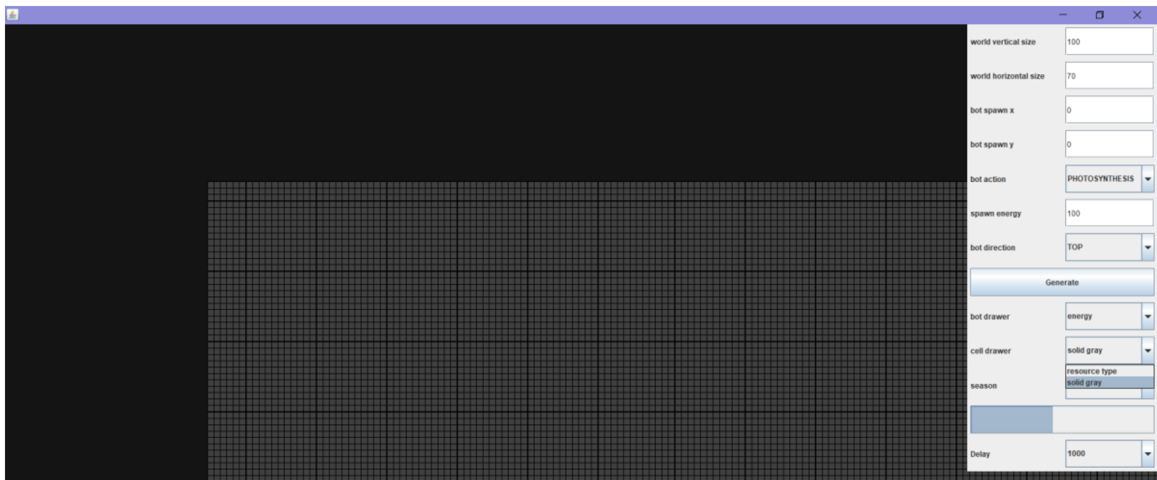


Рисунок 17 – Режим вімкнення кольорового фону

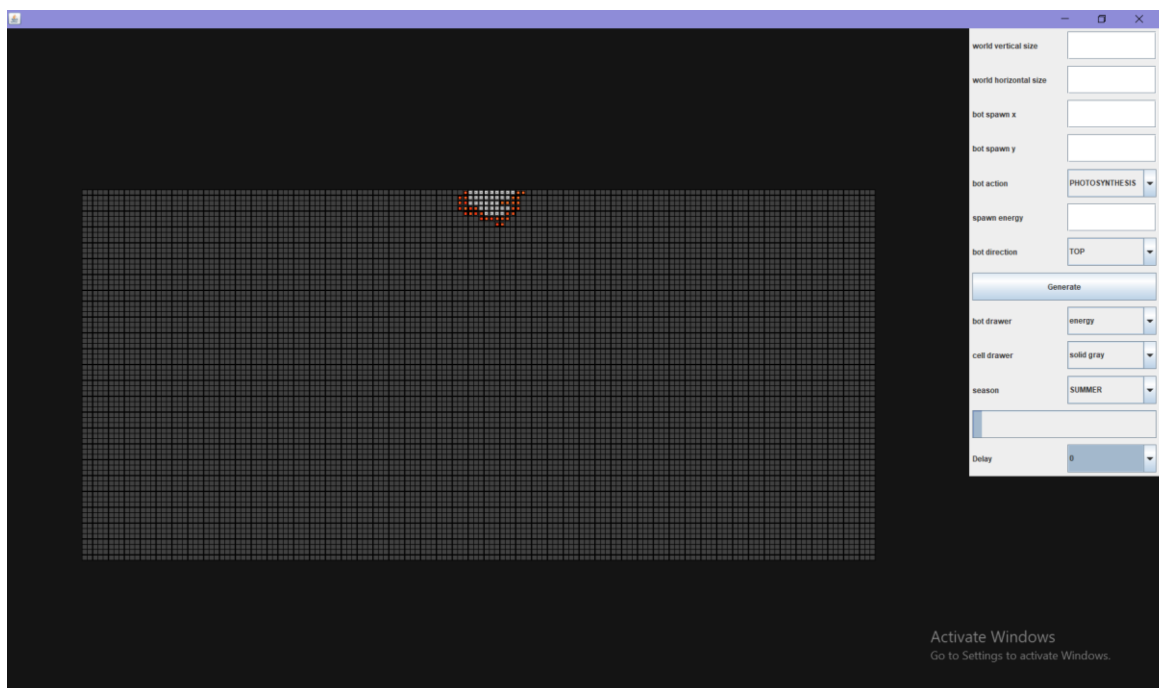


Рисунок 18 – Приклад початку симуляції

Через деякий час можна побачити, що деякі боти пішли мандрувати (рис. 19), тобто вони навчилися ходити і рандомно для старту обирають направлення руху. Вони відокремилися від загальної маси і пішли своїм шляхом.

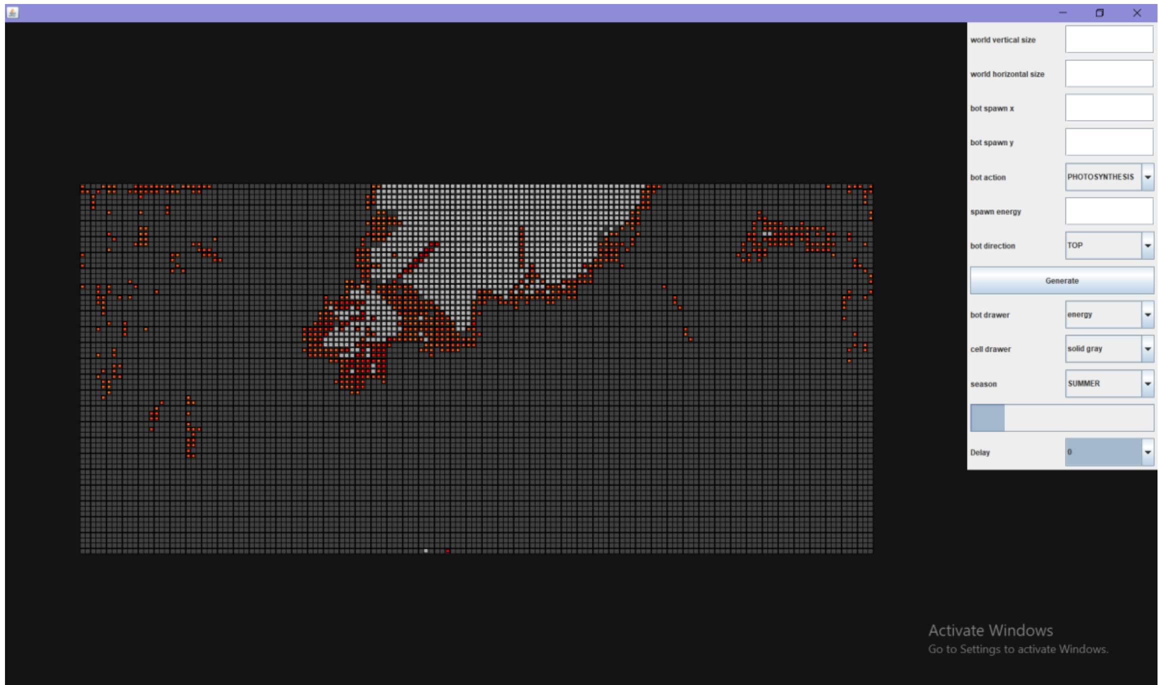


Рисунок 19 – Приклад переміщення ботів

На рисунку 20 відображено інформацію про тип робота. Зелений колір означає, що вони здебільшого переробляють сонячну енергію.

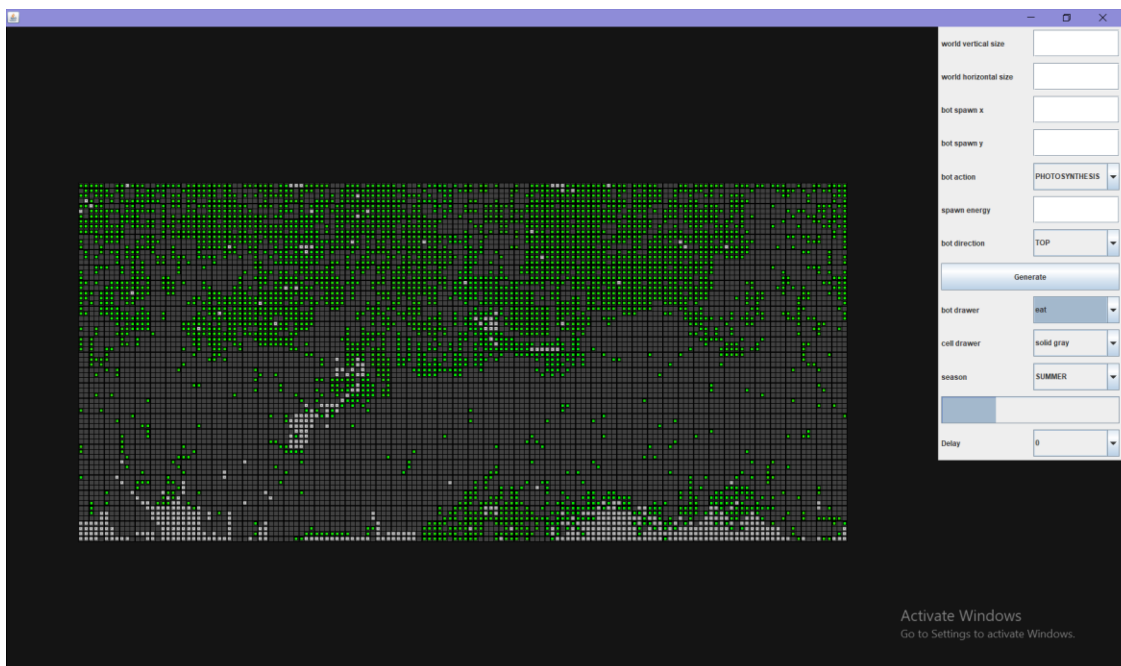


Рисунок 20 – Приклад переробки сонячної енергії ботами

Тут відображено інформацію про тип робота. Зелений означає, що вони переробляють сонячну енергію. Через ще якийсь час сформувалися окремі групи (рис. 21).

Рисунок 21 – Формування ботів у групи

За рахунок того, що дій було поставлено досить багато на робота, складно спочатку візуально розрізняти групи. Необхідно довго чекати, щоб отримати наступний результат, який показано на рисунку 22:

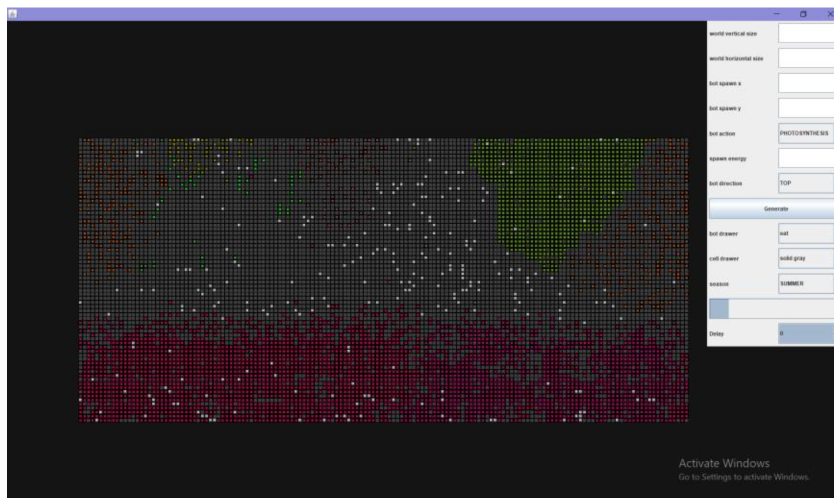


Рисунок 22 – Збільшення кількості вижитих ботів

Також, для отримання аналогічного результату є можливість поставити меншу кількість команд. В такому разі за коротчий час ми отримуємо результат, а саме одиниці ботів, яки прилаштувалися до умов існування та навчилися переробляти різні типи енергії. Це дало їм сили збільшити цикл свого існування.

В якості тестування програмного продукту були запущені симуляції на всі чотири сезони та з різними стартовими кількостями енергій. Це заняло багато часу, але такий підхід дозволив протестувати дипломний проект на всіх стадіях роботи.

ВИСНОВКИ

Генетичні алгоритми набирають обортів при розробці ігрових додатків. Це допомагає імітувати більш реальні умови у сценаг гри та створювати різноманітність кожного левелу.

Для дипломної роботи було взято ідею генетичного алгоритму для спроби імітації поведінки розвитку окремих істот (ботів) у власному світі. До функціоналу розробленої моделі були додані чотири сезона року, а також декілька видів форми ждерела енергії, ака необхідна для існування ботів.

Аналітичний огляд аналогів допоміг при проектуванні аглоритму роботи програмного додатку та моделюванню поведінки істот. В якості інструментальних засобів розробки вибрано мову програмування Java та бібліотеку SWIFT. За допомогою UML-інструментів побудовані діаграми варіантів використання та діаграма активності.

У програмному додатку розроблена панель для налаштування умов розвитку популяції та перемикання на різні режими для відображення стану колонії.

Логіка програмного ядра спроектована на подальше розширення функціоналу імітаційного світу та розуму істот в ньому.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Genetic Algorithms in Games (Part 1). URL: https://www.gamedeveloper.com/design/genetic-algorithms-in-games-part-1-. (дата звертання 25.04.2022)
2. What Is the Genetic Algorithm? URL: https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html (дата звертання 25.04.2022)
3. Игра InvaderZ генерирует врагов в стиле Space Invaders генетическим алгоритмом. URL: https://habr.com/ru/news/t/476732/ (дата звертання 25.04.2022)
4. Evolution: The Video Game. URL: https://www.gamereactor.eu/evolution-the-video-game-review/ (дата звертання 30.04.2022)
5. Pixel Monster. URL: https://dannydaisun.itch.io/pixel-monster (дата звертання 03.05.2022)
6. Oracle Java. URL: https://www.oracle.com/cis/java/ (дата звертання 03.05.2022)
7. Библиотека Swing. URL: https://java-online.ru/libs-swing.xhtml (дата звертання 08.05.2022)
8. Java. Swing. Основы. URL: https://pro-prof.com/forums/topic/java-swing-%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D1%8B (дата звертання 10.05.2022)
9. Use-case diagrams. URL: https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-use-case (дата звертання 18.05.2022)
10. Підходи до аналізу і проектування інформаційних систем. URL: https://elearning.sumdu.edu.ua/free_content/lectured:de1c9452f2a161439391120eef364dd8ce4d8e5e/20151203140326/204841/index.html (дата звертання 26.05.2022)