Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____

Кафедра____Програмної інженерії_____

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

рівень вищої освіти – другий (магістерський)

Дослідження методів візуалізації алгоритмів

Виконав: студент 2 курсу, групи ІПЗм-18-1

Цуварєв В.О.

спеціальності 121 – Інженерія програмного забезпечення

Освітньо-наукової програми

Інженерія програмного забезпечення

Керівник к.т.н. Каук В.І.

Допускається до захисту _____ З.В. Дудар

Зав. кафедри, проф.

2020 р.

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Факультет _Комп'ютерних наук_____
Кафедра _Програмної інженерії_____
Рівень вищої освіти – другий (магістерський)

Спеціальність _121 – Інженерія програмного забезпечення_____
Тип програми освітньо-професійна програма_____

Освітня програма _Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові _Цуварєву Вячеславу Олександровичу_____

1. Тема роботи _Дослідження методів візуалізації алгоритмів_____
затверджена наказом університету від "27" _03__2020 р. № _473__
2. Термін подання студентом роботи до екзаменаційної комісії 05 травня 2020 р.
3. Вихідні дані до роботи методи візуалізації алгоритмів, засоби навчання програмуванню, Використовувалась ОС Windows, середовище розробки програмного забезпечення Microsoft Visual Studio._____
_____
_____
_____
4. Перелік питань, що потрібно опрацювати в роботі мета роботи, аналіз проблемної галузі і постановка задачі, огляд методів візуалізації алгоритмів, візуальних засобів навчання програмуванню та розробці алгоритмів, розгляд рішень візуалізації різних об'єктів_____
_____
_____
_____
_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка *
1.	Аналіз предметної галузі	10 березня 2020 р.	
2.	Огляд існуючих методів	18 березня 2020 р.	
3.	Методи візуалізації алгоритмів	20 березня 2020 р.	
4.	Підготовка пояснювальної записки	25 квітня 2020 р.	
5.	Підготовка презентації та доповіді	27 квітня 2020 р.	
6.	Попередній захист	29 квітня 2020 р.	
7.	Нормоконтроль, рецензування	09 травня 2020 р.	
8.	Занесення диплома в електронний архів	10 травня 2020 р.	
9.	Допуск до захисту у зав. кафедри	11 травня 2020 р.	
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання ___ _____ 2020 р.

Студент _____

Керівник роботи _____ _____к.т.н. Каук В.І.____

РЕФЕРАТ / ABSTRACT

Атестаційна робота магістра містить: 50 с., 10 рис., 3 табл., 11 джерел, 1 додаток.

NET, НАВЧАННЯ, МОДЕЛЬ, ВІЗУАЛІЗАЦІЯ, АЛГОРИТМИ, КЛІТКОВІ АВТОМАТИ, MYSQL, WPF, MVVM, ООП.

Метою роботи є дослідження методів візуалізації алгоритмів та програмного коду для навчання та розвитку навичок програмування після здобуття базових знань.

Методи розробки базуються на інструментах розробки настільних додатків на платформі .NET, засобах роботи з фреймворком візуального відображення WPF та СУБД MySql.

В результаті роботи були проаналізовані різні засоби візуалізації алгоритмів і програмного коду. У якості альтернативі існуючому програмному забезпеченню було запропоновано розширену модель кліткових автоматів. Було розроблено програмний засіб та фреймворк з програмним інтерфейсом для полегшення візуалізації загальних алгоритмів.

NET, LEARNING, MODEL, VISUALIZATION, ALGORITHMS, CELLULAR AUTOMATON, MYSQL, WPF, MVVM, OOP.

The object of the study is a research of methods of algorithms and program code visualization for studying and developing programming skills after acquiring basic skills.

Development methods are based on development tools for desktop applications on .NET platform, visual framework WPF software tools and DBMS MySql.

As a result of work different methods of methods of algorithms and program code visualization were analyzed. As an alternative to an existing software an extended model of cellular automaton was introduced. There was developed a software and a framework with programmer interface for making visualization of general algorithms easier.

ЗМІСТ

ВСТУП

Із самої появи програмування люди стикалися з алгоритмами, бо вони є основою будь-якої, навіть найпростішої програми. Перші алгоритми були низькорівневі та зводились до оперування пам’яттю для простіших арифметичних операцій. Із розвитком мов програмування та інших засобів, алгоритми становились складнішими. Складні алгоритми із рекурсивними викликами та заплутаними ітераціями, які переслідували мету як можна ефективніше (по часу або по пам’яті) виконати задачу, були складні для розуміння навіть для досвідчених спеціалістів.

На сьогодні існує багато засобів побачити візуалізації поширених алгоритмів, які використовуються для навчання програмуванню та теорії алгоритмів, таких як алгоритми сортування, алгоритми обробки графів, рекурсивних алгоритмів тощо. Вони можуть допомогти студентові чи навіть спеціалісту зрозуміти алгоритм та етапи його реалізації.

Проте візуалізація алгоритмів не обмежена лише традиційними алгоритмами та задачами. Методи візуалізації можна застосувати також до більш складних алгоритмів та систем та використовувати не для тільки для покращення розуміння конкретного алгоритму, а й для розвитку вміння вирішення загальних задач програмування. Одним із перших цю ідею виразив Сеймур Пейперт, розробивши у 1967 році мову програмування LOGO [1]. Згодом, у 2004 було зроблено наступний крупний крок у напрямку розвиваючих серед розробки – світ побачила перша версія мови Scratch, яка наслідує ідею наглядного та цікавого навчання програмуванню вже на новому рівні.

Основною ідеєю, як пише Сеймур Пейперт [2], є навчання не через пасивне роз’яснення зі сторони, а активне втручання у сам процес, яке надає можливість самостійно пізнавати галузь, у даному випадку програмування та математику. Це особливо важливо при навчанні дітей, на якому зосереджені роботи Сеймура

Пейперта, але таким же само чином ця концепція може бути використана і для навчання людей старшого віку.

Основною проблемою можна виділити те, що подібні мови програмування націлені саме на навчання базових понять, а програмування складається також з більш складних та абстрактних концепцій, навчання яким відбувається вже у більш складному та менш інтерактивному вигляді.

Метою даної роботи є розробка програмного середовища із високим рівнем візуалізації результатів. Візуалізація відомих алгоритмів надає можливість зрозуміти їх, а візуалізація коду, що розробляється, допомагає краще зрозуміти сам процес мислення та дизайну алгоритмів. Цей фреймворк можна буде використовувати для навчання програмуванню після того, як базові поняття вже засвоєні та присутні навички у розробці простих програмних систем. Це допоможе розвинути отримані навички та разом із цим здобути самостійності у прийнятті рішень замість того, щоб розробляти вже відомі алгоритми.

Об’єктом дослідження є навчання розробці програмного забезпечення.

У цій роботі будуть розглядатися у якості предмета дослідження засоби навчання розробці програмного забезпечення за допомогою візуалізації алгоритмів та програмного коду.

У цій роботі виконано порівняння вже існуючих засобів візуалізації алгоритмів, їхніх галузей використання, застосування їх у навчальних процесах та структур даних, які приймають участь у візуалізації. У якості альтернативного засобу для візуалізації пропонується модель, яка розширює традиційну модель кліткових автоматів.

Розширена модель передбачає значні можливості для створення програмного інтерфейсу для розробки алгоритмів із наглядною візуалізацією. Для створення програмного інтерфейсу була розроблена математична модель, яка відповідає вимогам гнучкої реалізації.

Для демонстрації можливостей використання моделі наведений приклад, який демонструє використання моделі із деталями можливої програмної реалізації.

Цю модель можна використовувати для створення різноманітних завдань, які б ставили перед користувачем задачу розширення програмного фреймворку, а якщо точніше – задавання своїх правил роботи алгоритму на основі кліткового автомату із урахуванням розширень математичної моделі. Ці завдання мають великий потенціал для навчання принципам програмування через наглядну візуалізацію та легку перевірку правил.

Програмна модель, яка пропонується у цій роботі, надає можливість створити безліч завдань, які б допомогли людині із певними навичками програмування розібратись у більш складних та абстрактних поняттях. Через програмні інтерфейси можна задавати частини алгоритму, а потім дивитися на результат його роботи.

Програмна реалізація передбачає велику кількість можливостей для використання шаблонів програмування та демонстрацій гарних практик, які будуть запам’ятовуватися разом з їх використанням.

1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

Візуалізація алгоритмів це, в першу чергу, – пояснення, навчання через покращене розуміння. Питанням навчання програмуванню займаються вже давно. На сьогодні було зроблено багато різних програмних засобів, присвячених навчанню різним мовам програмування. Одним із перших програмних засобів для навчання програмному способу міркування була мова LOGO, перша версія якої з'явилась у 1967 році [1], автором цієї ідеї був Сеймур Пейперт (Seymour Papert). Ця мова була створена для навчання програмуванню (у першу чергу дітей) найпростішим для людини способом: через гру. Гра полягала в тому, що можна було програмувати робота-черепаху і вона малювала узори на папері там, де проходила (малювання можна було вимикати щоб малювати не зв'язані контури). Це й було візуалізацією – кожен шаг алгоритму, кожна дія програміста відбивалась на поведінці черепашки та малювала. Не потрібно було виводити дані в консолі чи записувати в файли, щоб побачити, що робить код – для цього було окреме полотно, яке візуалізувало кожен крок програміста.

Це було інновацією: дати знання о програмуванні навіть дитині, створити середу програмування з ясним відкликом. Але найважливішим була сама ідея навчання програмуванню через наглядний и простий результат. Саме через це за допомогою LOGO могли навчатися навіть діти (див. рис. 1.1). Як стверджує сам автор у книзі «Mindstorms: Children, Computers, and Powerful Ideas» [2], для того, щоб зрозуміти якусь концепцію, треба спробувати щось зробити з її допомогою: «Make something new with it, play with it, build with it». Головною ідеєю є створення чогось нового засобами нової для людини навички.

Ідея навчання через візуалізацію таким чином еволюціонувала у різні більш сучасні програмні засоби для навчання програмуванню. Зазвичай вони концентруються на тому, щоб навчити користувача певній мові або однієї із мов на вибір або узагалі програмному чи абстрактному мисленню.

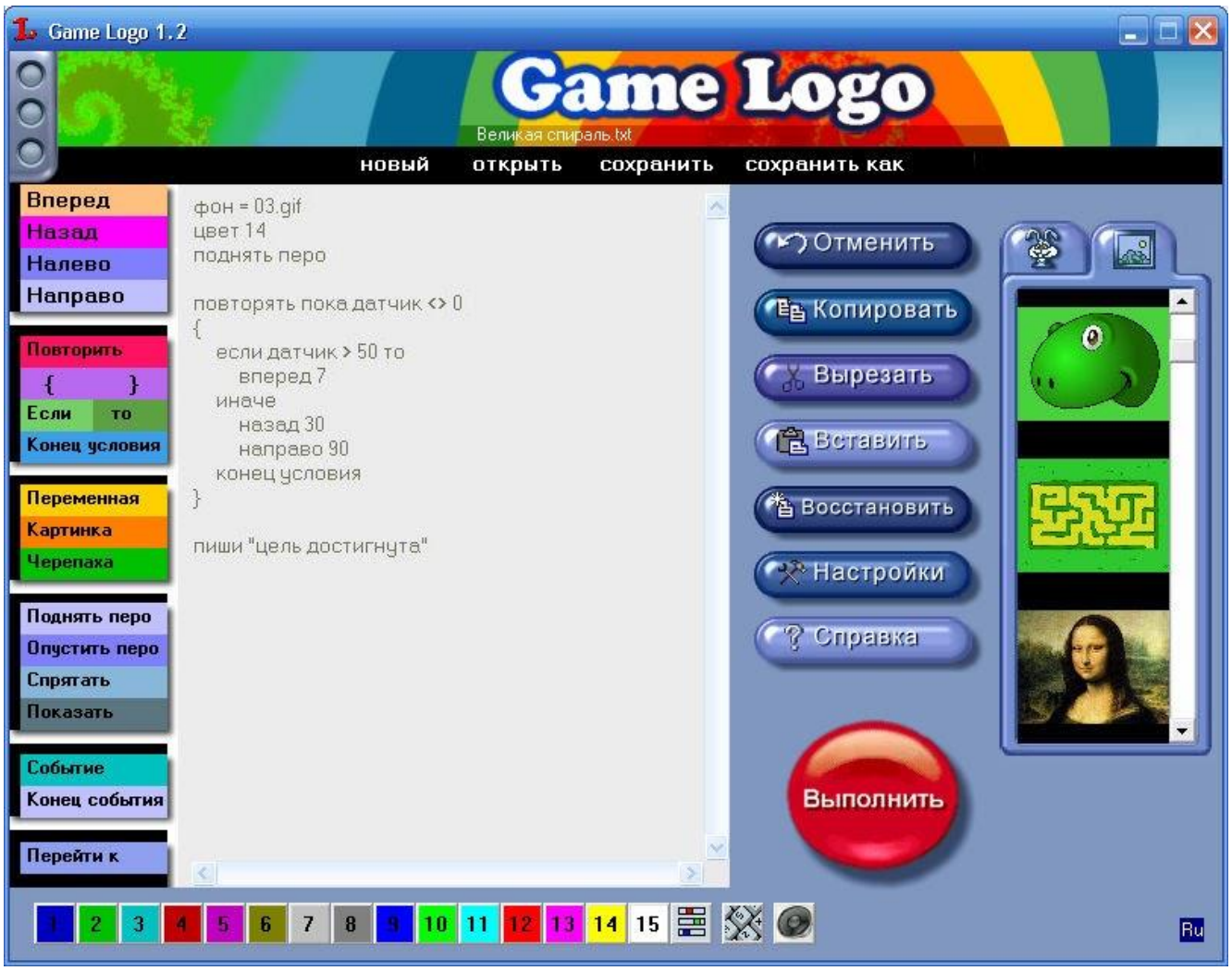

Рисунок 1.1 – Програмне середовище для мови LOGO

Прикладами програмних засобів для навчання саме мовам програмування можна назвати: checkio.org (Python/JavaScript), Codeacademy (Інтерактивні курси. Багато мов програмування на вибір та більш конкретні теми як бази даних або Test Driven Development.), Ruby Warrior (Ruby).

Вони навчають мовам програмування, даючи прості задачі, які призвані закріпити розуміння базових принципів, конструкцій мови та іншого. Час від часу деякі з них дають теоретичну інформацію, але існують варіанти і з тим, що користувач сам повинен шукати інформацію про те, як йому вирішувати завдання.

Прикладами програмних засобів, які націлені на надання узагальненого розуміння підходу до написання програм можна назвати саму мову LOGO та програмні засоби для її візуалізації та його ідейні нащадки, наприклад додаток на

Android «Coding Planets» (рис. 1.2) або мова «Scratch», розроблена Массачусетським технологічним інститутом.

Рисунок 1.2 – блочне програмування у додатку «Coding Planets»

На сьогодні, багато з них відходять від використання реальних або спеціально створених мов програмування і надають перевагу програмуванню із блоків, кубиків або іншого візуального представлення команд. Такий підхід абстрагує від синтаксису і надає можливість працювати з самим поняттями, які використовуються: командами, циклами, функціями, переходами. Це набагато краще для пояснення базових принципів програмування і найчастіше націлено насамперед на дітей. Але у блочному підході є свої мінуси: це не дає можливості звикнути до синтаксичної складової програмування. Також, це підходить для імперативного підходу до програмування, але зовсім не годиться до, наприклад, функціонального чи більш поширеного на сьогоднішній день об'єктно-орієнтованого підходу. Більш складні парадигми програмування складніше представити у візуальній формі. Через це мова «Scratch», наприклад, використовує

гібридний підхід – у блоках можна задавати константні значення та використовувати змінні (див. рис. 1.3).

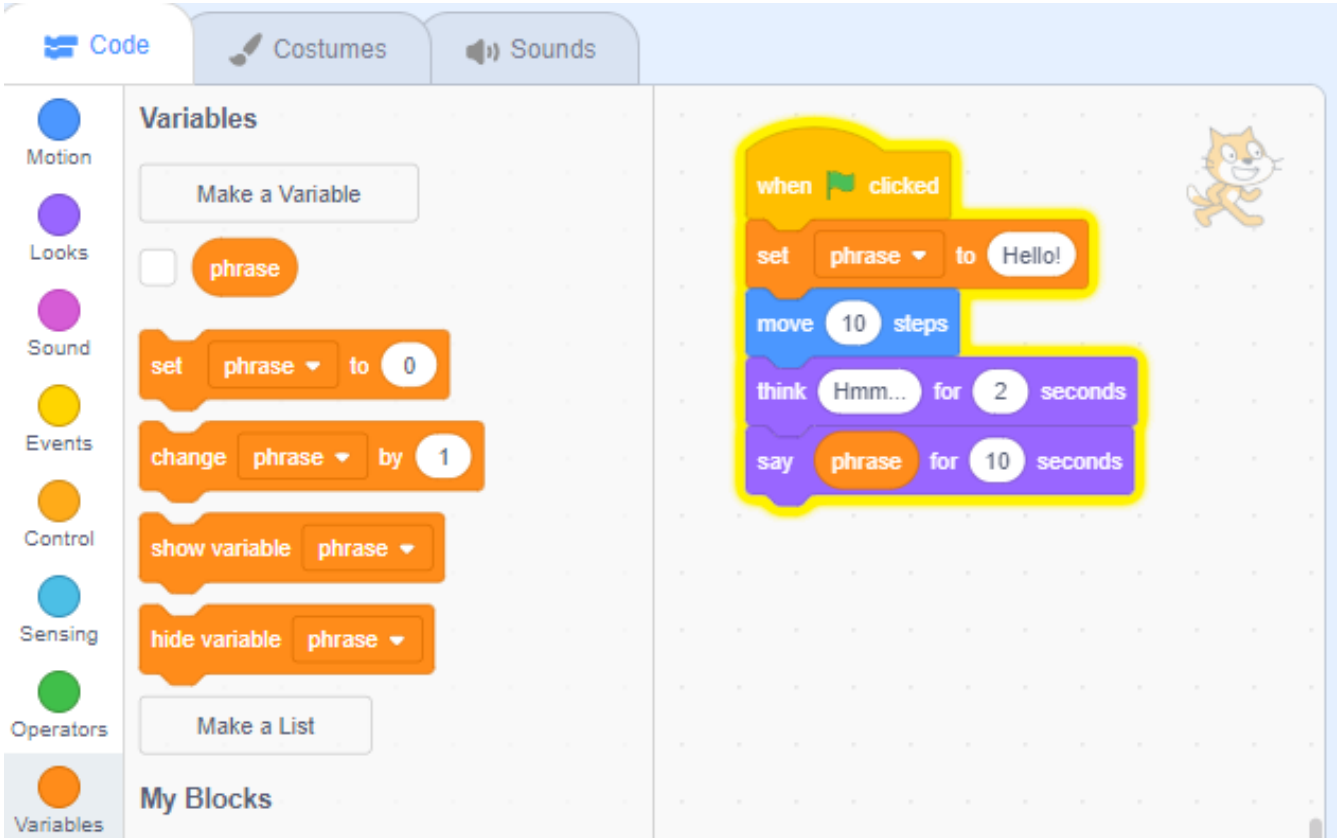

Рисунок 1.3 – гібридний підхід до програмування у мові «Scratch»

Візуалізація того, що робить програма чи окремі частини коду може переслідувати різні цілі. У статті Майка Бостока (Mike Bostock) [3] виділяється чотири основні причини візуалізації алгоритмів:

– навчання – навіть якщо ви навчаєтесь для себе, візуалізація може бути хорошим шляхом для глибокого розуміння. Реалізація візуалізації – це як навчання самого себе. Набагато простіше запам’ятати алгоритм інтуїтивно, візуально, ніж запам’ятовувати код та обов’язково забувати малі, але істотні деталі.
– налагодження – мати можливість побачити, що саме робить ваш код може покращити продуктивність. Візуалізація не витісняє необхідність тестування, але тести використовуються для знаходження факту помилки

та не пояснюють її. Візуалізація ж, навпаки, допомагає визначити несподівану поведінку в реалізації, навіть якщо результат роботи програми виглядає правильно.

- розуміння – анімації та інші способи візуалізації краще ніж формальний опис чи код, допомагаючи зробити розуміння простішим.
- розвага – набагато простіше змусити людину робити те, що здається їй цікавим. Навіть якщо людина не розуміє всі деталі алгоритму, візуальна демонстрація може продемонструвати важливість та складність алгоритму людині, яка не розбирається в темі.

Усі вони важливі для цієї роботи: навчання та розуміння йдуть рука об руку і є головною метою для програмної системи, що розробляється; налагодження є однією із основних занять розробника програмного забезпечення, а елемент розваги допомагає привернути та утримати увагу.

Існують різні способи візуалізації алгоритмів, такі як: анімація, статичне зображення, текстові візуалізації тощо. Також можна візуалізувати різні шаги чи етапи алгоритму. Вибір стилю та способу візуалізації залежить як від задачі, так і від самого алгоритму. Наприклад, для візуалізації алгоритмів сортування часто використовуються анімації. Але за анімаціями буває складно стежити, коли ще немає розуміння алгоритму. У цьому допомагають статичні зображення із усіма етапами алгоритму допомагають демонструвати логіку за цими кроками так, що людина може простежити, наприклад, пересування невеликої групи елементів масиву, у зручному для неї темпі.

У цій статті [3] також проводиться зручне розподілення візуалізацій алгоритмів за їх глибиною:

- рівень 0 / чорна коробка – найпростіший випадок, тільки показує результат роботи алгоритму. Він не пояснює алгоритм, але може перевірити результат. Також завдяки чорному ящику можливо порівняти роботу різних алгоритмів. Цей спосіб використання алгоритму може комбінуватися із більш глибоким аналізом виводу. Таким, як наприклад,

матриця зміщення для алгоритму перетасовки (див. рис. 1.4), яка демонструє то, як часто алгоритм перемішує кожні 2 індекси масиву.

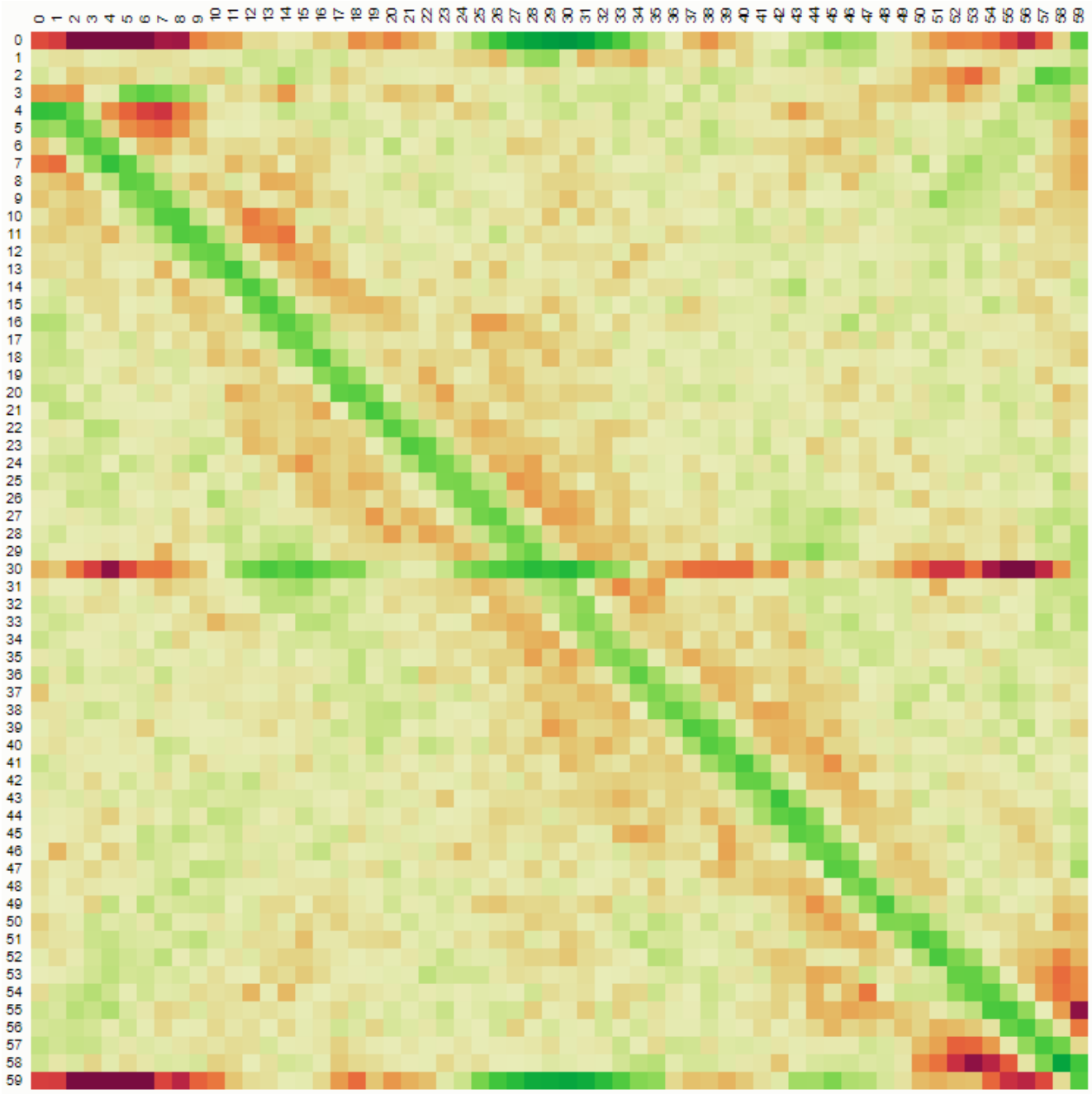

Рисунок 1.4 – приклад візуалізації рівня 0 із статті

– рівень 1 / сіра коробка – візуалізація шагів алгоритму. Деякі алгоритми, хоча і не усі, будують результат поступово. Візуалізація проміжного результату, демонструє роботу безпосередньо алгоритму. Це дозволяє

пояснити більше, не вводячи допоміжних абстракцій, бо проміжні та фінальні результати мають однакову структуру. Не зважаючи на це, така візуалізація алгоритму може породити більше питань ніж відповідей, через те, що вона не дає ніякого пояснення, чому саме алгоритм робить те, що він робить.

- рівень 2 / біла коробка – для відповіді на питання «чому?», біла коробка показує внутрішній стан алгоритму у додаток до проміжних результатів. Цей тип надає найкращі можливості для роз'яснення, але також найбільш тяжкий для читача через те, що значення та призначення внутрішнього стану повинні бути чітко описані. Існує ризик, що додаткова складність заплутує читача. Розподіл інформації на рівні може зробити зображення більш зрозумілим. Також, через те, що внутрішній стан занадто сильно залежить від типу алгоритму, цей вид візуалізації найчастіше непридатний для порівняння алгоритмів.

Для задачі розробки програмного середовища для навчання алгоритмічному мисленню, найбільш за все підходить саме другий рівень – біла коробка, через те, що основною метою візуалізації у даному випадку є саме роз'яснення роботи алгоритму (написаного самотужки або вже готового). Програмне середовище повинно демонструвати не тільки кінцевий результат, а також проміжні результати та загальний стан алгоритму. Це необхідно для максимальної демонстрації кожної зміни, яку користувач буде вносити у код.

Тобто для візуалізації, яка б ефективно надавала розуміння користувачу о виконанні певних процесів і ходу алгоритмів, потрібно обрати таке середовище, яке буде підходити за наступними ознаками:

- користувач повинен мати можливість бачити проміжні результати роботи алгоритму;
- користувач повинен мати можливість бачити стан алгоритму на кожному із кроків;
- користувач може задати свій алгоритм та побачити його кроки.

2. МЕТОДИ ВІЗУАЛІЗАЦІЇ АЛГОРИТМІВ

2.1 Вибір об’єкта візуалізації

Алгоритми бувають різні, як і засоби їх візуалізації. Найчастіше певна візуалізація підходить тільки для обмеженого сімейства алгоритмів. Наприклад, можна використовувати ту ж саму візуалізацію із відображенням елементів масиву для алгоритмів сортування, з деякими невеличкими відмінностями (для алгоритму сортування злиттям потрібно візуалізувати 2 масиви, а не 1). Але ця візуалізація не підходить для того, щоб продемонструвати алгоритм пошуку найкоротшого шляху у графі, бо самі дані і необхідна для візуалізації інформація різниться. Це призводить до того, що для фреймворку потрібно вибрати якусь базову структуру для візуалізації, яка буде підтримуватись, щоб охопити максимальну кількість можливих задач.

Для розглядання у якості можливих варіантів. були узяті наступні структури: масив, граф, клітинний автомат. Саме ці структури були обрани саме через те, що кожна з них, не зважаючи на свою простоту, дає дуже великий простір для реалізації різноманітних алгоритмів. У перелік не були включені такі структури, як дерево та зв’язаний список, бо вони є частковим випадком графа. Після пошуку в мережі інтернет, було знайдено декілька сервісів та робіт, які дозволяють візуалізувати як класичні алгоритми для масивів та графів, так і користувацькі модифікації.

Подібна задача для роботи з масивами вже вирішувалась у роботі «Дидактичний потенціал використання систем візуалізації алгоритмів у процесі навчання програмуванню» [4] де була розроблена система для демонстрації роботи алгоритмів для роботи з масивами, таких як пошук мінімального елемента та сортування. У цій роботі також проводилось опитування студентів-користувачів, у якому було виявлено, що 92% студентів вважають, що «динамічна візуалізація роботи алгоритму допомогла зв’язати команди алгоритму з їх призначенням» і 82% студентів робота з візуалізатором допомогла вирішити задачі, у яких

застосовуються алгоритми по обробці статичних структур даних (масивів). Ця робота доводить доцільність візуалізації алгоритмів на практиці та показує можливості візуалізації алгоритмів на прикладі алгоритмів, які працюють із масивами.

Один з сервісів для візуалізації алгоритмів – це algorithm-visualizer.org [5]. Він надає можливість користувачу побачити візуалізації вже готових алгоритмів, демонструючи структуру масиву та/або графа у окремому віконці та відображаючи зміни на кожному кроку алгоритму. Також там можна написати свій алгоритм та приєднати до якихось із структур даних у ньому візуалізацію. Тобто таким чином можна відобразити алгоритми, які включають у себе масиви, графи та їх комбінації.

Також дуже наглядним є visualgo.net [6]. Цей ресурс включає у собі більше візуалізацію принципу роботи тих чи інших структур даних. Окрім масивів та різних видів дерев та графів, тут присутні: зв'язаний список, хеш таблиця та масив суфіксів. Тут немає можливості написати свій алгоритм та візуалізувати бажану структуру даних, але є велика кількість візуалізацій роботи різних методів зміни та доступу до різноманітних структур даних.

Для клітинних автоматів також існує багато «пісочниць», але в них неможливо задати свої правила. Наприклад, у додатку “WBS Cellular Automata” [7] можна задавати такі дані як колір та задавати прости чисельні значення, які будуть задавати поведінку. Але не можливо вийти за рамки цих обмежень – неможливо відтворити більш складний алгоритм поведінки: додати випадкові змінні, враховувати загальну кількість клітин певного стану тощо.

2.2 Розширення математичної моделі

Базову модель клітинних автоматів згідно з Фон Нейманом [8] можна описати як множину скінченних автоматів (2.1). Тобто кожна клітинка є елементом

множини із скінченним автоматом, який описує його стани та пересування між ними.

$$\sigma_{i_1,i_2,\dots,i_n} \in \{0,1,2,\dots,k-1,k\} \quad (2.1)$$

де n – вимірність клітинного автомату,
k – кількість станів

Найчастіше за все використовується саме двовимірна модель (n дорівнює 2), а кількість станів теж найчастіше обмежується двома (клітина жива чи мертва). Складнішими є правила зміни цих станів. Правила переходу описується наступним чином (2.2).

$$\sigma_{i_1,i_2,\dots,i_n}(t+1) = \varphi(\sigma_{j_1,j_2,\dots,j_n} | \, j_m \in N(i_1,\ i_2,\dots,i_n)) \quad (2.2)$$

де n – вимірність клітинного автомату,
t – момент часу,
φ – функція правила переходу,
$N(i_1,\ i_2,\dots,i_n)$ – деяка околиця точки $(i_1,\ i_2,\dots,i_n)$

Тобто стан певної клітки на наступний крок системи залежить від станів деякої множини оточуючих кліток на даний момент. Цю модель дуже зручно використовувати для завдання простих правил моделі, кількість яких обмежена. Таким чином навіть із простими правилами поведінка може бути дуже складною та різноманітною.

Якщо використовувати клітинні автомати для навчання алгоритмам та для візуалізації певної логіки, то таке просте обмеження можливостей користувача лише заважає. Людині, яка пробує зробити візуалізацію деякого складного алгоритму, не підходить обмеження простих правил. Тому, я пропоную розширити визначення функції правила переходу, яка дозволить складати більш складні правила. Якщо у класичному варіанті функція правила переходу φ приймає тільки один аргумент, деяку околицю точки $N(i_1,\ i_2,\dots,i_n)$, то розширений варіант функції прийматиме 3 параметра. Розширену модель можна задати формулою 2.3.

$$\sigma_{i_1,i_2,\dots,i_n}(t+1) = \varphi(\sigma_{j_1,j_2,\dots,j_n} \mid j_m \in N(i_1, \quad i_2, \dots, i_n), \sigma_{i_1,i_2,\dots,i_n}, s(f)) \qquad (2.3)$$

де n – вимірність клітинного автомату,

t – момент часу,

φ – функція правила переходу,

$N(i_1,\ i_2, \dots, i_n)$ – деяка околиця точки $(i_1,\ i_2, \dots, i_n)$,

$\sigma_{i_1,i_2,\dots,i_n}$ – стан самої точки $(i_1,\ i_2, \dots, i_n)$,

$s(f)$ – функція від загального стану поля

До параметра станів деякої множини в околиці точки $j_m \in N(i_1,\ i_2, \dots, i_n)$ додається ще два. Перший із них – це стан самої точки на момент обчислення нового стану, другий – функція s від загального поля f. У якості функції від загального стану поля можуть виступати, наприклад, функція, яка рахує кількість клітин, які мають певний стан, функція від розміру поля, функція, яка рахує середню кількість сусідів для у певному стані для усіх клітин тощо. Таке розширення дозволяє створювати більш складні правила, які дозволять робити складні візуалізації та бачити результат.

Це розширення легко перевести на розповсюдженні мови програмування, бо у більшості з сучасних мов використовується об’єктно-орієнтований підхід. Тобто стан самої клітинки і стан поля можна задати через об’єкти, які буде легко використовувати для обчислення функції переходу.

Тобто, потрібно розробити такий фреймворк, який буде оперувати поняттями кліткового автомату та дозволить користувачу задавати функцію правила переходу із допомогою мови програмування. Працюючи із програмою, користувач буде отримувати задачу і писати код, який буде взаємодіяти із кодом самого фреймворку, після чого мати можливість побачити візуальне відображення введеного алгоритму функції правила переходу у клітковому автоматі, працюючим за його правилом. Це дозволить візуально побачити зміни, які користувач буде проводити у коді на поведінці кліткового автомату. На підставі побачених результатів користувач зможе краще зрозуміти чи діє його алгоритм саме так, як і задумано.

2.3 Обґрунтування та уточнення розширеної моделі

Розширена модель надає широкі можливості для створення нових завдань для фреймворку візуалізації. Як вже було сказано у розділі 2.2, для задання вхідних даних до функції правила переходу: деякої околиці точки, стану самої точки та функції від загального стану поля – у об'єктно-орієнтованих мовах програмування можна використати об'єкти із складним станом та власними методами для обчислення функцій, які характеризують цей стан.

Слід зазначити, що у широкому сенсі усі 3 параметри можна звести до 1 у канонічній моделі (2.2). Для цього треба, виконати дві умови:

- щоб множина точок у складі параметру $N(i_1,\ i_2, \dots, i_n)$ не виключала саму точку $(i_1,\ i_2, \dots, i_n)$;
- щоб множина точок включала у себе усі точки поля (у разі його скінченності), що примушує виконання першої умови.

Ця нотація надає можливість користуватись класичним визначенням, але не надає можливості розрізняти різні множини клітинок. Для простішого програмування та більш детального розуміння краще на етапі теорії відокремити ці параметри та задати їм певні обмеження, щоб уникнути ситуацій де, наприклад, буде використано стан окремої клітинки із певними координатами для визначення стану клітин у певній зоні незалежно від їх відносного положення, що не є природним правилом.

Через це було вирішено використати три окремих параметра для функції правила переходу:

- множину клітинок $N(i_1,\ i_2, \dots, i_n)$ у деякій околиці точки $(i_1,\ i_2, \dots, i_n)$, від стану яких може напряму залежати стан даної клітинки, бо сусіди мають фіксований вплив на клітинку, не такий як сама клітинка чи загальний стан поля;

- стан самої клітинки $\sigma_{i_1,i_2,\dots,i_n}$, який задає можливі переходи самої клітинки, а не впливає на рішення, як ще одна клітинка у множині сусідніх чи усіх клітинок на полі;
- функція від загального стану поля $s(f)$, а не множина усіх клітин поля, це важливо, бо результат від цієї функції не надає функції правила переходу цілісної інформації про поле або його конкретну частину, а лише надає результат деякої рівнозначної агрегації станів усіх клітинок поля.

Спочатку розглянемо окремо кожний параметр та його можливі засоби використання та обмеження. Перший параметр є класичним у теорії кліткових автоматів – це деяка околиця заданої точки. У грі «Життя» [9] та більшості її розширень, наприклад, функція рахує кількість клітинок у певному стані у цій множині і на підставі цієї кількості видає вердикт о новому стані обраної клітинки. Розширена модель передбачає більш складні стани елементів, тобто дискретних станів кожної клітинки зазвичай стає більше. Але усе можна звести до (2.4):

$$f\left(N(i_1, i_2, \dots, i_n)\right) = c(a\left(l\left(N(i_1, i_2, \dots, i_n)\right)\right)) \tag{2.4}$$

де $f\left(N(i_1, i_2, \dots, i_n)\right)$ – функція лише від сусідів клітинки,
l – функція, яка обмежує кількість клітинок у множині N,
a – функція, яка проводить агрегацію серед залишених клітинок,
c – функція, яка визначає новий стан клітинки на підставі результату агрегації,
$N(i_1,\ i_2, \dots, i_n)$ – деяка околиця точки $(i_1,\ i_2, \dots, i_n)$

Наприклад, у класичних правилах гри «Життя» функція, яка обмежує кількість клітинок у множині N – це функція, яка обирає тільки «живі» клітинки; функція, яка проводить агрегацію серед залишених клітинок – це функція, яка підраховує кількість клітинок у множині та функція, яка визначає новий стан клітинки на підставі результату агрегації, – це функція, яка визначає новий стан клітинки, як «живий», коли повернута кількість знаходиться у певних межах.

Таким чином задається звичайна функція правила переходу. Роль цієї функції у розширеній моделі – це одна із можливих функцій для обчислення переходу. У

залежності від стану самої клітинки та значення функції від стану усього поля, ця функція може перетерпіти змін.

Перед обговоренням впливу стану самої клітинки, краще визначити значення функції від загального стану поля, бо саме її значення рішуче у різниці між класичною моделлю та розширеною. Надавати доступ до усіх даних поля недоречно із точки зору інкапсуляції. Саме тому у якості параметру функції правила переходу передається вже обчислене агрегаційне значення з результату функції від загального стану поля.

Існує декілька ситуацій у яких доступ до усього стану поля може призвести до неадекватних моделей. Наприклад, можуть бути використано лише певна кількість клітинок за їх знаходженням, щоб впливати на усі клітинки на полі, або клітинки можуть кілька разів піддаватися впливу сусідніх клітинок: один раз через параметр сусідів, а другий – через параметр усього поля. Походячи із цих зображень, доцільно не тільки передавати у функцію правила переходу готовий результат обчислення, а також обмежити доступ до деякої інформації о клітинках – таких як їхнє розташування, щоб у обчислені усі вони були абсолютно рівні та із рівною вагою впливали на результат. Проте треба зазначити, що різні клітинки можуть мати різні ваги, якщо це засновано саме на їхньому стані.

За моделлю функції, вона така ж сама, як і для сусідніх клітинок (2.4), лише з тим обмеженням, що вхідні дані обмежені лише тим, що стосується безпосередньо стану клітинок, а не їх розташуванням. У разі сусідніх клітинок може бути допустимо засновуватись на їхньому розташуванні. Наприклад, щоб ті клітинки, які розташовані ближче, мали більший вплив ніж ті, які розташовані на відстані.

Що стосується стану самої клітинки, то її вплив дещо різниться. Вплив околиці клітинки та стану усього поля задаються функціями, які приймають вхідні дані і на їхній підставі дають деяку – часткову у випадку розширеної моделі – оцінку нового стану клітинки. Вплив стану самої клітинки, у свою чергу, задає можливі стани переходу та різні передумови для цих переходів.

Наприклад, у нас є 3 стани: A, B та C. Кожна клітинка, по суті, задається через скінченний автомат яка задає переходи між ними. Із стану A можна перейти тільки у стани A та B, із стану B – у C та A, із стану C – у A та B. Тобто залежно від стану клітинки відрізняються можливі стани та умови переходу до тих чи інших станів. Умови переходу є функціями від сусідів клітинки та від результату функції усього поля. Від стану самої клітинки залежить то, яку саму функцію буде обрано для обчислення переходу. У прикладі вище ми маємо три загальних функції, які характеризують алгоритм прийняття рішення у який стан перейде клітинка із даного стану за умови наданих сусідів та деякого результату функції від загального стану поля. Загальна формулу, яка описує то, як клітинка переходять у інші стани наведена нижче (2.5).

$$\sigma_{i_1,i_2,\dots,i_n}(t+1) =$$

$$\begin{cases} \omega_1\left(\sigma_{j_1,j_2,\dots,j_n} \middle| j_m \in N(i_1,i_2,\dots,i_n), s(f)\right), \sigma_{i_1,i_2,\dots,i_n}(t) = 0 \\ \omega_2\left(\sigma_{j_1,j_2,\dots,j_n} \middle| j_m \in N(i_1,i_2,\dots,i_n), s(f)\right), \sigma_{i_1,i_2,\dots,i_n}(t) = 1 \\ \dots \\ \omega_k\left(\sigma_{j_1,j_2,\dots,j_n} \middle| j_m \in N(i_1,i_2,\dots,i_n), s(f)\right), \sigma_{i_1,i_2,\dots,i_n}(t) = k \end{cases} \quad (2.5)$$

де n – вимірність клітинного автомату,
t – момент часу,
ω_n – функція правила переходу для відповідного стану самої клітинки,
$N(i_1,\ i_2,\dots,i_n)$ – деяка околиця точки $(i_1,\ i_2,\dots,i_n)$,
$\sigma_{i_1,i_2,\dots,i_n}$ – стан самої точки $(i_1,\ i_2,\dots,i_n)$,
$s(f)$ – функція від загального стану поля

Тобто на кожний стан існує своя стратегія прийняття рішень. Можливо, що у деяких станах ці стратегії однакові. Тоді ми вже говоримо не про відповідність окремих стратегій окремим станам, а про відповідність стратегій деяким множинам станів. Це особливо важливо, якщо йде мова не про дискретні стани, а про неперервні стани. Наприклад, кожна клітинка має свою температуру у якості

внутрішнього стану і її поведінка вже залежить від певних діапазонів температурних значень.

2.4 Приклад використання розширеної моделі

Розширену модель можна використовувати для створення різних завдань для розвитку навичок програмування із наглядною візуалізацією процесів, що відбуваються. У цьому пункті будуть вказані деякі приклади того, які саме завдання можна відтворити у рамках даної моделі.

У якості найбільш наглядного та простого прикладу можна взяти модифікацію гри «Життя», із додаванням впливу функції від загального стану поля. Додамо до гри правило, яке буде, залежно від загальної кількості живих клітинок спрощувати або ускладнювати появу нових клітинок для того, щоб утримувати загальну кількість живих клітинок ближче до усередненого значення.

Для початку, оформимо правила основної гри у розширеній моделі (2.6):

$$\sigma_{i_1,i_2}(t+1) = \begin{cases} \omega_1\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2), s(f)=0\right), \sigma_{i_1,i_2}(t)=0 \\ \omega_2\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2), s(f)=0\right), \sigma_{i_1,i_2}(t)=1 \end{cases};$$

$$\omega_1\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2), \sigma_{i_1,i_2}, s(f)=0\right) = \begin{cases} 1,\ count\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2)\right) = 3 \\ 0,\ count\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2)\right) \neq 3 \end{cases};$$

$$\omega_2\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2), \sigma_{i_1,i_2}, s(f)=0\right) = \begin{cases} 1,\ 2 \leq count\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2)\right) \leq 3 \\ 0,\ count\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2)\right) < 2 \\ 0,\ count\left(\sigma_{j_1,j_2} \middle| j_m \in N(i_1,i_2)\right) > 3 \end{cases}; \quad (2.6)$$

де t – момент часу,

ω_n – функція правила переходу для відповідного стану самої клітинки,

$N(i_1, i_2)$ – деяка околиця точки $(i_1,\ i_2)$,

σ_{i_1,i_2} – стан самої точки $(i_1,\ i_2)$,

$s(f)$ – функція від загального стану поля, у даному випадку повертає 0

В цій моделі функція від загального стану поля повертає константу – нуль та не використовується при обчисленні нового значення стану клітинки, у якому стані вона б не знаходилась.

Допустимо, що ми бажаємо відтворити більш сталу систему, у якій якщо загальна кількість живих клітин надмірно мала, то вікно значень, у яких створюється нова або продовжує жити стара клітинка, розширюється, а якщо загальна кількість живих клітин надмірно велика, то створення нових клітинок зупиняється, а можливість життя вже живучих зменшується. Покладемо, що при загальної кількості живих клітинок менше ніж або рівно 30% від загального ми вважаємо, що їх замало і підтримуємо життя клітинок, при кількості живих клітинок між 30% і 70% ми залишаємо правила незмінними, а при 70% та більше – ускладнюємо життя клітинкам. Функція від загального стану поля набуває наступного вигляду (2.7):

$$s(f) = bias\left(count\left(alive(f)\right)\right);$$

$$bias(n) = \begin{cases} 1, & \dfrac{n}{count(f)} \leq 0.3 \\ 0, & 0.3 < \dfrac{n}{count(f)} < 0.7 \\ -1, & \dfrac{n}{count(f)} \geq 0.7 \end{cases} \qquad (2.7)$$

де $s(f)$ – функція від значення усього поля,

$alive$ – функція, яка обмежує кількість клітинок у множині N, повертаючи лише живі клітинки,

$count$ – функція, яка проводить рахує клітинки,

$bias$ – функція упередження, яка визначає певну вагу нового стану клітинки

Із цими змінами, функція правила переходу для стану 1 із 2.6 набуде вигляду 2.8:

$$\omega_2\left(\sigma_{j_1,j_2}\middle|\, j_m \in N(i_1,i_2)\,, \sigma_{i_1,i_2}, s(f)\right) =$$
$$\begin{cases} 1,\ 2 \le count\left(\sigma_{j_1,j_2}\middle|\, j_m \in N(i_1,i_2)\right) \le 3 + s(f) \\ 0,\ count\left(\sigma_{j_1,j_2}\middle|\, j_m \in N(i_1,i_2)\right) < 2 \\ 0,\ count\left(\sigma_{j_1,j_2}\middle|\, j_m \in N(i_1,i_2)\right) > 3 + s(f) \end{cases}; \quad (2.8)$$

де ω_n – функція правила переходу для відповідного стану самої клітинки,
$N(i_1,i_2)$ – деяка околиця точки $(i_1,\ i_2)$,
σ_{i_1,i_2} – стан самої точки $(i_1,\ i_2)$,
$s(f)$ – функція від загального стану поля, задана у 2.7

Таким чином, за формулою вище можна описати систему із розширеною моделлю на мові програмування. У парадигмі об’єктно-орієнтованого програмування для більш простого розширення доцільно створити інтерфейси та базові класи, які користувач може реалізовувати та розширювати для того, щоб прописувати власні правила роботи алгоритму і бачити візуалізацію результату.

Для програмної реалізації даного правила користувачеві потрібно буде задати об’єкт, відповідний за логіку обчислення функції від загального стану поля. Згідно з математичним визначенням, цей алгоритм складається із 3 частин: функції фільтрування, яка буде обмежувати множину усіх клітин множиною тих, які мають певний стан, у даному випадку, це живі клітинки; функції агрегації, яка повертає одне значення, яке характеризує усю множину, повернену першою функцією, у даному випадку це підрахунок кількості живих клітинок у множині; функції, яка порівнює значення, яке повернула попередня функція, із деякими умовами, у даному випадку, це – порівняння із граничними значеннями.

Також потрібно буде задати іншу стратегію для обчислення нового стану клітинки у разі находження її у живому стані. Треба задати функцію, яка б, по-перше, використовувала таку ж саму функцію для підрахунку кількості живих клітинок, як і у функції від загального стану поля, по-друге використовувала б

результат функції від загального стану поля і по-третє, порівнювала б результат першої функції із фіксованим значенням, яке б зміщувалось на результат другої.

Таким чином розширена модель надає великі можливості для створення різних завдань по програмуванню, які, незважаючи на свою різноманітність, мали б декілька важливих рис: усі завдання підпорядковуються однієї моделі, яка у програмуванні була б представлена у якості програмного інтерфейсу, який надає точки розширення, які можуть бути використані програмістом для задання своєї логіки, та наочність візуалізації, яка б допомогла програмістові побачити результати своєї роботи, побачити, як саме працює їх алгоритм, та додати потрібні зміни до коду, якщо поведінка відрізняється від бажаної. Це надає великі можливості для навчання принципам програмування у зручній середі.

2.5 Порівняння отриманої моделі із існуючими моделями

Отримана модель для візуалізації передбачає використання для систем інтерактивного навчання програмуванню. Для оцінки її відносно інших моделей та систем, які використовуються по тому ж призначенню, використаємо наступні критерії:

- глибина візуалізації;
- наочність;
- простота взаємодії;
- варіативність;
- складність шаблонів.

Глибина візуалізації, яка розглянута детальніше у розділу 1, це кількість інформації про те, що відбувається на різних етапах під час роботи алгоритму. Простота взаємодії – це відносна характеристика, яка визначається тим, як просто взаємодіяти (модифікувати) алгоритм. Наочність – це відносна міра того, як даний метод допомагає розуміти концепції програмування. Варіативність – це кількість

гіпотетичних алгоритмів, які може продемонструвати даний метод. Складність шаблонів – це рівень та кількість демонстрації можливих шаблонів програмування.

Порівняємо здобуту модель для візуалізаціями із основними існуючими моделями: візуалізацією операцій над масивами, візуалізацією операцій над графами та класичних середовищ виконання клітинних автоматів. Для цього виставимо по кожному критерію ранги, стартуючи від 1 у таблиці 1.

Таблиця 1 – рангове порівняння критеріїв

	Глибина візуалізації	Наочність	Простота взаємодії	Варіативність	Складність шаблонів
Масиви	1	2	1	4	4
Графи	1	3	1	3	3
Клітинні автомати	2	4	2	2	2
Розширені клітинні автомати	2	1	2	1	1

Для порівняння потрібні абсолютні значення, тому перенесемо рангові оцінки на бали від 0 (погано) до 1 (добре) у таблиці 2.

Таблиця 2 – абсолютне порівняння критеріїв

	Глибина візуалізації	Наочність	Простота взаємодії	Варіати-вність	Складність шаблонів	Сума
Масиви	1	0,75	1	0,25	0,25	3,25
Графи	1	0,5	1	0,5	0,5	3,5
Клітинні автомати	0,5	0,25	0,5	0,75	0,75	2,75
Розширені клітинні автомати	0,5	1	0,5	1	1	4

Як можна побачити, не зважаючи на менші можливості щодо глибини візуалізації та простоти взаємодії, саме розширені клітинні автомати є найбільш доречними для використання для навчання програмуванню ніж інші методи візуалізації.

3. ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Проектування архітектури ПЗ

Програмне забезпечення таких масштабів потребує детального обмірковування та розділення відповідальності між різними модулями. Головною частиною програмного забезпечення є настільний додаток. Програмне забезпечення передбачає наявність серверної частини, але вона не розглядається у рамках даної дипломної роботи і винесена на майбутню доробку.

Щодо вибору мови програмування для розробки настільного додатку, розглядалися 2 об’єктно-орієнтовані мови програмування: C# (WPF або WinForms) або Java (Swing). Переваги та недоліки цих платформ було проаналізовано, для кожного важливого атрибута була виставлена оцінка, а самим атрибутам була надана вага (вказана у лапках) у згідно з тим, наскільки це важливо для проекту (див. табл. 3). У результаті було обчислено сумарну оцінку.

Таблиця 3 – Порівняння мов програмування

Мова програмування та фреймворк	Адаптивний інтерфейс (5)	Підтримка користувацьких елементів інтерфейсу (4)	кросплатформність (2)	Підтримка у останніх версіях (3)	Сумарна оцінка
C# WPF	5	5	1	4	59
Java Swing	3	4	5	1	44

Отже, рамках додатку для ПК було вирішено використати платформу .NET з використанням WPF. Розробляти додаток на платформу Windows краще всього саме використовуючи продукти Microsoft. Для них є велика підтримка зі сторони документації та спільноти. WPF надає широкі можливості для розробки інтерфейсу з використанням різних ефектів та складної логіки відображення майже без написання великих об’ємів коду. Це значно спрощує процес розробки. Мінусом цього варіанту є відсутність кросплатформної підтримки – цей додаток буде тільки

для системи Windows. Але для клієнтської сторони це було розглянуто як неважливе. Також фреймворк .NET має вбудований компілятор, який можна використовувати прямо із програми на C#.

Розроблена частина програмного забезпечення була розбита на кілька модулів, ураховуючи шаблон MVVM. Згідно цьому шаблону, додаток розподіляється на 3 основні рівні: модель (Model), вигляд (View) та модель вигляду (View-Model). Модуль сервісів знаходиться декілька окремо, він інкапсулює взаємодії, для яких потрібні декілька рівнів і визивати їх напряму порушало б структуру залежностей.

На високому рівні рішення розподілено на декілька модулів, кожен з яких складається з декількох модулів низького рівня (підмодулів), які у .NET Framework називаються збірками. Кожен модуль високого рівня групує декілька пов'язаних між собою збірок, які, в свою чергу представляють собою окремий простір імен з тісно пов'язаними класами (див. рис. 3.1) для керування залежностями [10].

До моделі належать модулі "Data", "DB" та "ProgramSculptor". Модуль "ViewModel" цілковито відповідає за модель вигляду. Вигляд розбитий на кілька маленьких модулів, безпосередньо збірка "UI", яка представляє найвищій рівень інтерфейсу користувача - там зібрані тільки описи різних елементів програмного інтерфейсу на мові xaml. Там немає звичайних класів і логіки. Збірка "UI.Content" об'єднує класи для обробки даних, які будуть відображатися, а "UI.Controls.Common" об'єднує деякі елементи інтерфейсу користувача. Обі ці збірки використовуються в сервісах для відображення різних діалогових вікон, тому винесені окремо, щоб сервіси цілковито не залежали від модуля "UI".

Модуль "Data" відповідає за дані у додатку, він є основою моделі програми. В ньому прописані усі класи, які представляють собою дані, з якими працює програма. Модуль "FileManaging" надає набір методів для роботи з вкладеною файловою структурою у якій будуть зберігатися деякі дані. Модуль "DataAccessInterfaces" містить інтерфейси для доступу к даним, без конкретних реалізацій. Усі ці інтерфейси використовуються у вищих рівнях замість реалізацій.

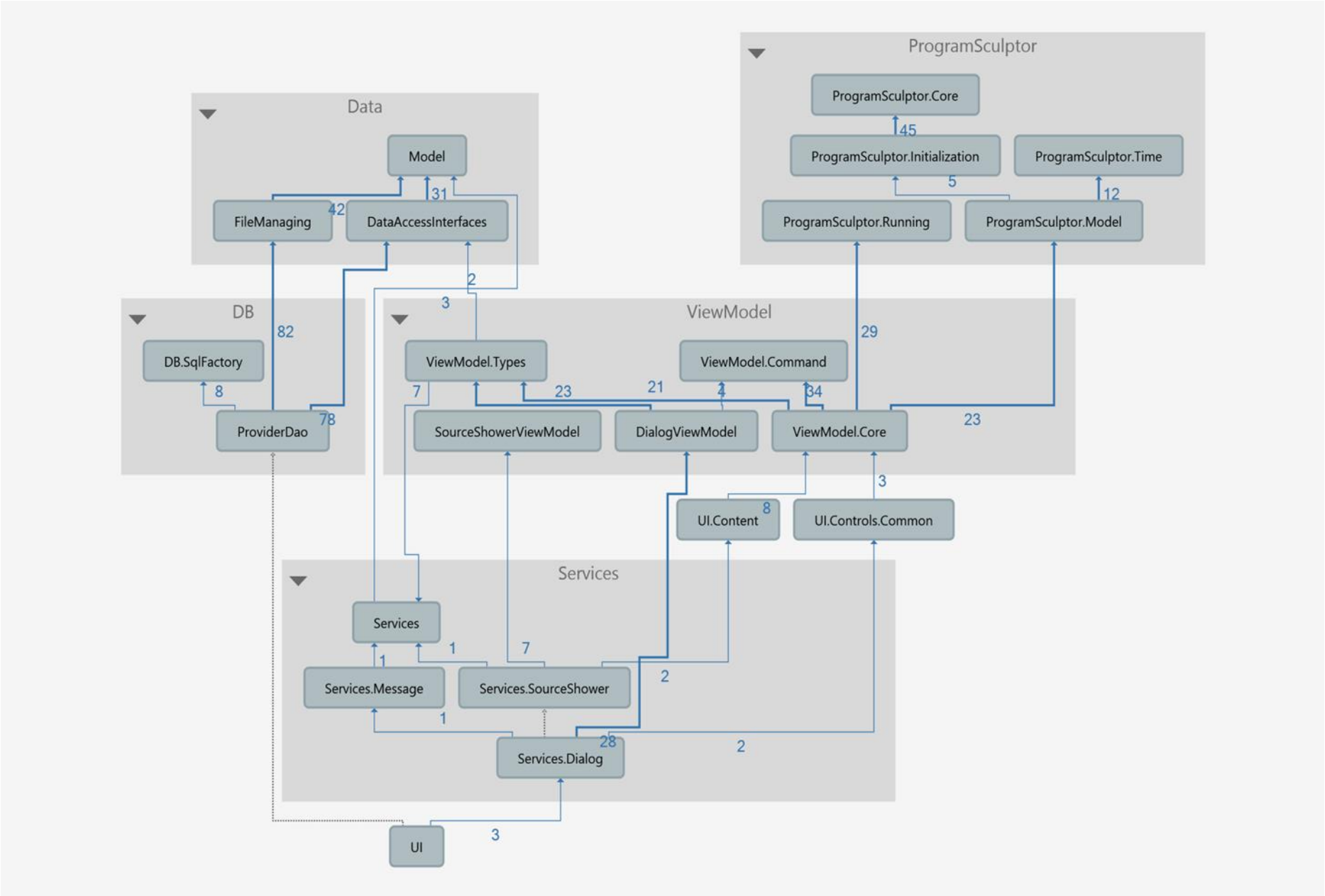

Рисунок 3.1 – Діаграма залежності проекту

Модуль “DB” містить у собі реалізацію інтерфейсів доступу до даних для баз даних. Модуль “ProviderDao” містить реалізації інтерфейсів із модуля “DataAccessInterfaces”, які засновані на доступу до бази даних через провайдер, якій підходить під будь-яку базу даних. Це значно спрощує зміну бази даних. Модуль “DB.SqlFactory” використовується для зберігання SQL-запитів та іншої статичної с точки зору коду інформації, яка може змінюватися з розвитком проекту. Для зберігання використовуються xml файли конфігурації. Для зміни якого-небудь запиту достатньо змінити ці файли без необхідності перекомпільовувати код. У сукупності модуль DB ніде явно не використовується, класи з нього підтягуються динамічно у момент роботи програми. Цьому у жодному іншому модулі не використовується посилання на нього на момент компіляції.

Модуль “ProgramSculptor” містить у собі фреймворк з яким буде працювати користувач. Назва пов’язана з роботою скульпторів, які виліплюють із безформної глини осмислені і красиві скульптури. Фреймворк на даний момент розбитий на 5 модулів, один з яких призначено для використання користувачем. Модуль з основними формами і класами, на яких буде базуватися користувач називається “ProgramSculptor.Core”. Модуль “ProgramSculptor.Time” містить у собі допоміжні класи, для управління ходом часу та реакцією на зміни в моделі. Модуль “ProgramSculptor.Initialization” відповідає за заповнення поля різними елементами та іншу ініціалізацію моделі. У подальшому планується відкрити його також для можливості розширення користувачем у процесі роботи - щоб можна було задавати свої правила для заповнення моделі. Наприклад, зробити кластери зі своїх об’єктів на початку, або навпаки розмістити усі елементи на деякій мінімальній відстані. “ProgramSculptor.Model” відповідає за різні можливі моделі. Зараз там тільки одна - “звичайна” - з полем-сіткою. Модуль “ProgramSculptor.Running” містить класи для запуску моделі та компіляції коду під час роботи програми, що потрібно для виконання коду користувача.

Модуль “ViewModel” відповідає за “модель вигляду” - частину шаблону MVVM, яка відображає дані з моделі на вигляд і задає логіку їхньої зміни. Модуль “ViewModel.Command” містить у собі команди, які використовуються у решті

модулю. Модуль “SourceShowerViewModel” містить у собі моделі відображення для сервісу відображення вихідного коду. Модуль “ViewModel.Core” об’єднує більшість класів, які створюють модель вигляду у проекті. В більшості елементів, що відображаються, демонструють стан цих об’єктів. Модуль “DialogViewModel” призначений для моделі вигляду сервісу діалогів і відображає стан діалогових вікон. Модуль “ViewModel.Types” призначений для узагальненого інтерфейсу перегляду списку типів, створених користувачем.

Модуль сервісів забезпечує легке застосування складної логіки через прості інтерфейси. Сервіси використовуються на рівні ViewModel, щоб визивати такі речі як діалогові вікна чи вікон з повідомленнями. Модуль під назвою “Services” не містить у собі безпосередньо сервісів, а тільки інтерфейси, які реалізуються в інших модулях. Модуль “Services.Message” надає функціонал для демонстрування користувачу повідомлень за допомогою вікон з повідомленнями. Уся інформація, яку потрібно відобразити користувачу, посилається у ці сервіси через інтерфейс, що надає можливість взаємодіяти з користувачем без прямого використання користувацького інтерфейсу у моделі вигляду. Модуль “Services.Dialog” відповідає за передачу управління діалоговим вікнам для виконання певних дій та отримання результатів дій користувача у цих вікнах. Діалоги використовують повноцінну модель MVVM всередені себе, зі своїм відображенням, яке засновується на спільних класах відображення та окрему модель відображення, про яку йшла річ вище. Модуль “Services.SourceShower” виконує функцію схожу на показ повідомлень, але замість декількох речень повідомлення демонструється вихідний код деякого класу, який потрібно було відобразити.

3.2 Проектування бази даних

Для виконання поставленої задачі потрібно було зберігати дані завдань та рішень користувачів, як і дані самих користувачів. Для зберігання частини даних

було вирішено використовувати базу даних. Виключення було зроблено для файлів з кодом, який пише користувач. Ураховуючи те, що окрім малих об’ємів даних (назв завдань, їхніх формулювань, логінів користувачів тощо), треба зберігати багато даних з файлів (файлів с кодом, тестами тощо), похідний код було рішено зберігати у файловій системі, а не у базі даних.

Питання обрання СУБД для виконання задачі було одним з найскладніших з усього проектування. Розробка додатків під системи корпорації Microsoft зазвичай використовуються їхні продукти, як MS SQLServer. Але так як для додатку ще планується додати сервер для синхронізації та взаємодії з іншими користувачами, який може бути розташовано на іншій платформі із іншим оточенням, то було обрано СУБД, якою буде зручно користуватись як на додатку для Windows, так і на сервері на якій завгодно платформі. Тому у якості СУБД була обрана MySql.

У базі даних присутні дані про місце знаходження певних файлів. Наприклад, файли тестів завдань. Вони добавлені для зберігання цих даних на сервері, коли файли можуть лежати не у файловій системі на тій ж самій машині, а на іншій машині і доступ до них буде відбуватися через ці шляхи. У клієнтського додатку ці дані зберігаються у файловій системі у спеціальному каталогу, путь до якого не зберігається у базі даних, а засновується на назвах сутностей, наприклад, завдань.

Отже, у базі даних зберігаються дані о наступних видах даних:

– завдання;
– рішення;
– користувачі;
– розташування файлів користувача.

На рис. 3.2 зображено діаграму сутностей, які використовуються у базі даних. З рисунку видно, що між завданнями та користувачами зв’язок багато-до-багатьох. Але рішення існують не тільки для зв’язку, а має у собі дані про рішення та набір пов’язаних з рішенням файлів. Самі вирішення, як і вирішення у MS Visual Studio, є проектами з кодом. У цьому випадку цей код – це вирішення конкретної задачі конкретним користувачем. Через це вони пов’язані із файлами коду, розташування яких зберігається у базі даних.

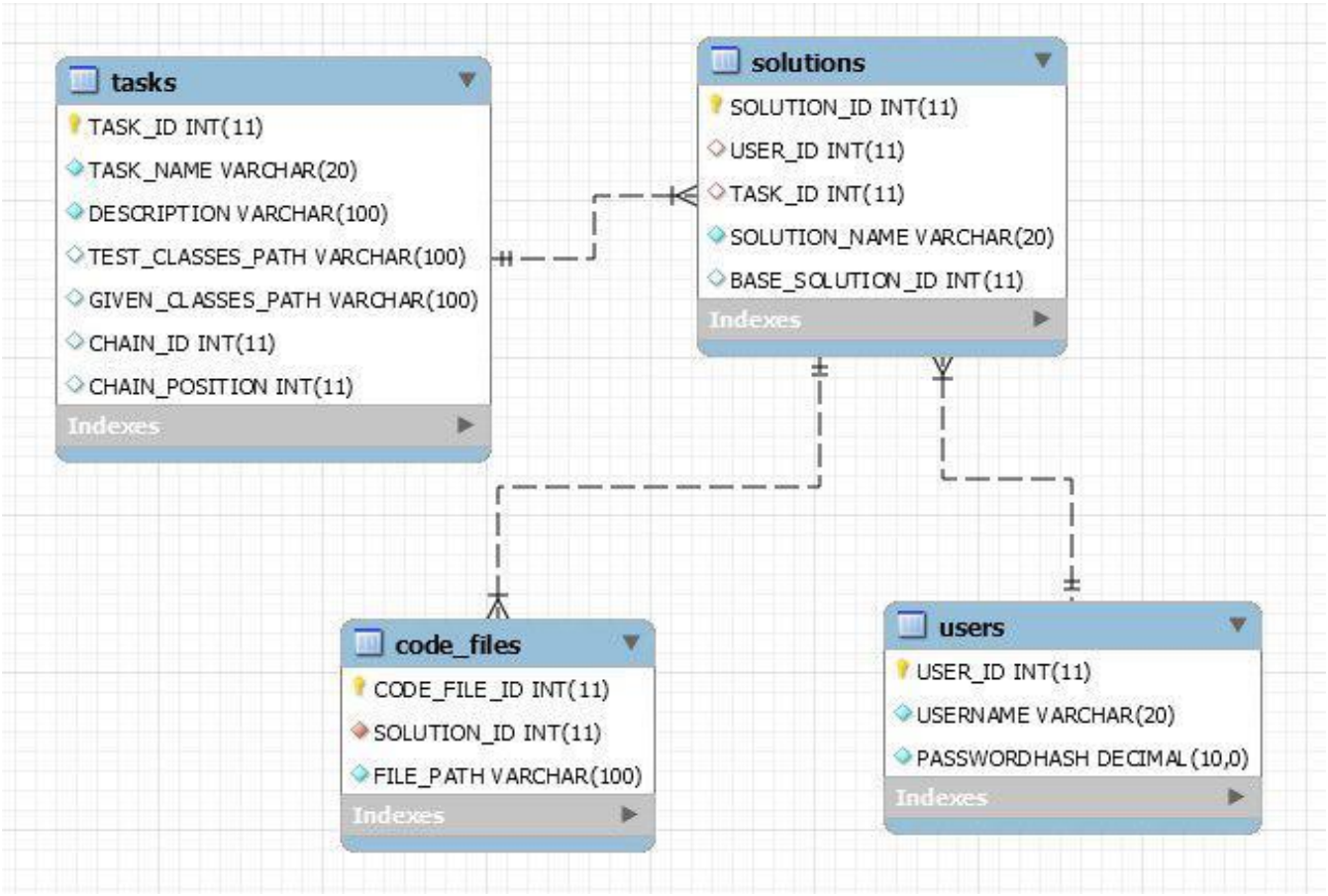

Рисунок 3.2 – Діаграма сутностей бази даних

Питання обрання СУБД для виконання задачі було одним з найскладніших з усього проектування. Розробка додатків під системи корпорації Microsoft зазвичай використовуються їхні продукти, як MS SQLServer. Але так як для додатку ще планується додати сервер для синхронізації та взаємодії з іншими користувачами, який може бути розташовано на іншій платформі із іншим оточенням, то було обрано СУБД, якою буде зручно користуватись як на додатку для Windows, так і на сервері на якій завгодно платформі. Тому у якості СУБД була обрана MySql.

У базі даних присутні дані про місце знаходження певних файлів. Наприклад, файли тестів завдань. Вони добавлені для зберігання цих даних на сервері, коли файли можуть лежати не у файловій системі на тій ж самій машині, а на іншій машині і доступ до них буде відбуватися через ці шляхи. У клієнтського додатку ці дані зберігаються у файловій системі у спеціальному каталогу, путь до якого не зберігається у базі даних, а засновується на назвах сутностей, наприклад, завдань.

Припускається, що на комп'ютері користувача клієнтського додатку не буде достатньо багато рішень з файлами, які б займали занадто багато місця.

3.3 Приклади найцікавіших алгоритмів та методів

У проекті не було багатьох використано складних алгоритмів, бо більша частина програми працює з даними користувача і завдань. Цікавим може бути алгоритм пошуку сусідніх клітинок на полі відносно заданої. Проблема полягає у тому, що клітинки можуть бути розташовані на границі і тоді у пошук не потрібно включати деякі напрями. Наприклад, якщо клітинка розташована у лівому верхньому куті, то шукати клітинки вище чи зліва не має сенсу. Тому було використано винос метода [11] для відділення частини, яка відповідає за підготовку даних, а саме границь пошуку, від частини, де ці дані використовуються. Нижче наведено реалізацію у вигляді двох методів:

```
public IEnumerable<Cell> GetNearbyCells(Cell cell)
{
    int minX = cell.X == 0 ? 0 : cell.X - 1;
    int minY = cell.Y == 0 ? 0 : cell.Y - 1;

    int maxX = cell.X == Size - 1 ? Size - 1 : cell.X + 1;
    int maxY = cell.Y == Size - 1 ? Size - 1 : cell.Y + 1;
    return GetOthersInRange(cell, minX, minY, maxX, maxY);
}

private IEnumerable<Cell> GetOthersInRange(
    Cell cell, int minX, int minY, int maxX, int maxY)
{
    IList<Cell> cells = new List<Cell>();
    for (int x = minX; x <= maxX; x++)
    {
        for (int y = minY; y <= maxY; y++)
        {
            Cell found = this[x, y];
            if (found != cell)
            {
                cells.Add(found);
            }
        }
    }
    return cells;
}
```

Функція повертає тип IEnumerable<Cell>, що дозволяє абстрагуватися від реального представлення даних. Для обчислення границь у функції GetNearbyCells використовуються тернарні вирази, бо це найбільш лаконічно відображає їхню суть – потрібно взяти одну границю, якщо ми на краю поля та іншу, якщо ми на середині поля по цій осі. Таким чином обчислюються границі, які будуть передані на обробку у іншу функцію – GetOthersInRange.

Друга функція не залежить від першої. Насамперед, в неї можна передавати будь-які параметри і вона поверне усі клітинки, які знаходяться у цьому діапазоні та на є такою ж самою, яку передали. Це дозоляє використовувати її і в інших місцях, якщо потрібно. Наприклад, можна знайти усі клітинки у певній частині поля, що може використовуватись у об’єктах з іншою поведінкою, ніж елементи, наприклад, у об’єктах які діють статично на частину поля.

У самому методі відбувається створення тимчасового списку для зберігання результатів. Потім, проходиться по усім клітинкам у заданих межах по осям X та Y. Кожна клітинка перевіряється на спів падіння з переданою клітинкою. Цей метод повинен знаходити тільки інші клітинки у діапазоні. Тому у разі збігу клітинка до результату не додається.

Функція пошуку сусідніх клітинок використовується у елементах, щоб вони могли «бачити» сусідні клітинки і взаємодіяти з ними. Таким чином, елемент не може взаємодіяти з клітинками поміж цієї множини.

Цей алгоритм є пошуком по околиці Мура 1-ого порядку – сукупності із 8 клітин, які оточують клітину з даними координатами та стикаються із нею хоча б краями.

В деяких варіаціях цей окіл може визначатися інакше. Наприклад, існують такі клітинні автомати, які існують на нескінченному замкнутому просторі. Тобто двовимірне поле з’єднуються із самим собою по краям. Якщо клітка у верхньої границі, то вона вважається сусідньою для відповідних кліток у нижній границі. Простір поля замикається в фігуру під назвою тор. І немає такого місця, яке знаходилося би на краю.

3.4 Створення користувацького інтерфейсу

Проектування користувацького інтерфейсу почалось з розробки макетів інтерфейсу користувача за допомогою онлайн-сервісу Creately. Були розроблені макети для основного вікна додатку у двух виглядах. На рис. 3.3 зображено вигляд головного вікна з режимом пісочниці.

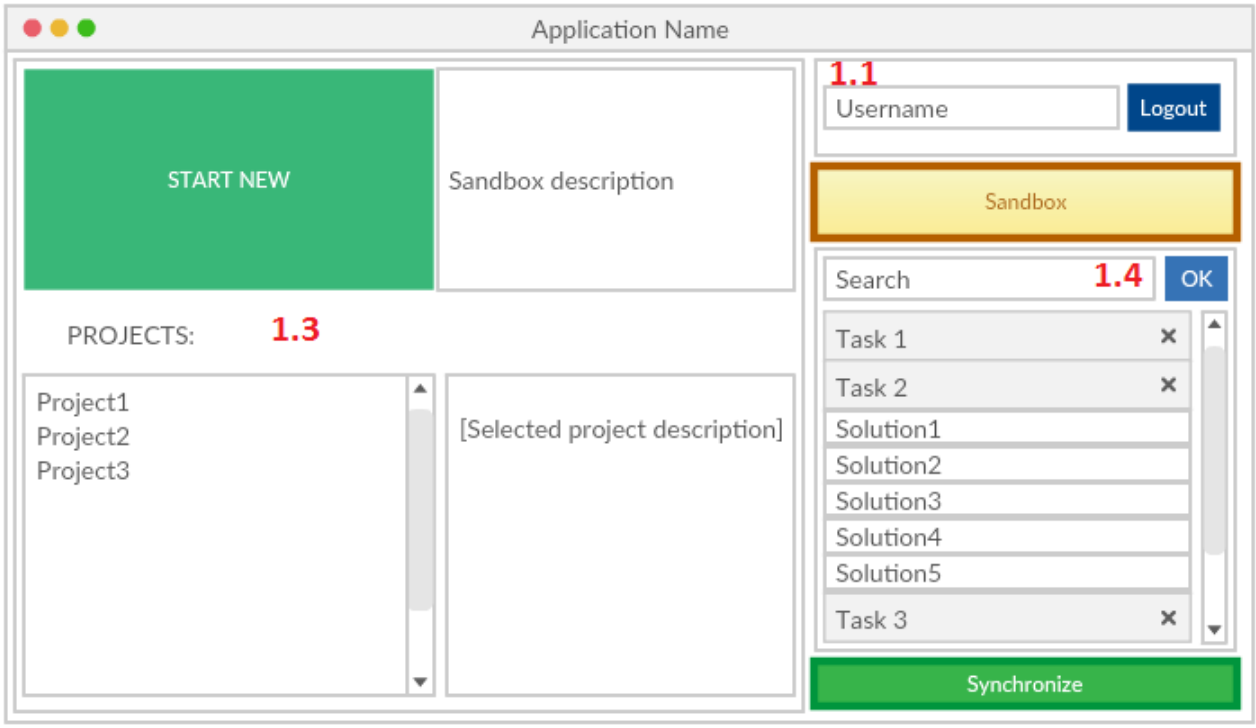

Рисунок 3.3 – Головне вікно програми

На головному вікні верху розташовано ім’я користувача з кнопкою виходу чи входу, якщо зараз користувач зайшов, як гість. Нижче розташовано список завдань, кожен елемент якого по кліку відкривається в список існуючих рішень користувача, який зараз зайшов у програму, чи користувача-гостя, а саме вікно переходить до іншого вигляду, де можна переглянути дані про обране завдання (див. рис. 3.4). Цей список можна отфільтрувати за назвою завдання, щоб було

простіше шукати потрібний. Також там розташовуються кнопка “Sandbox” для переходу на перегляд проектів пісочниці, як на лівій частині рис. 3.3.

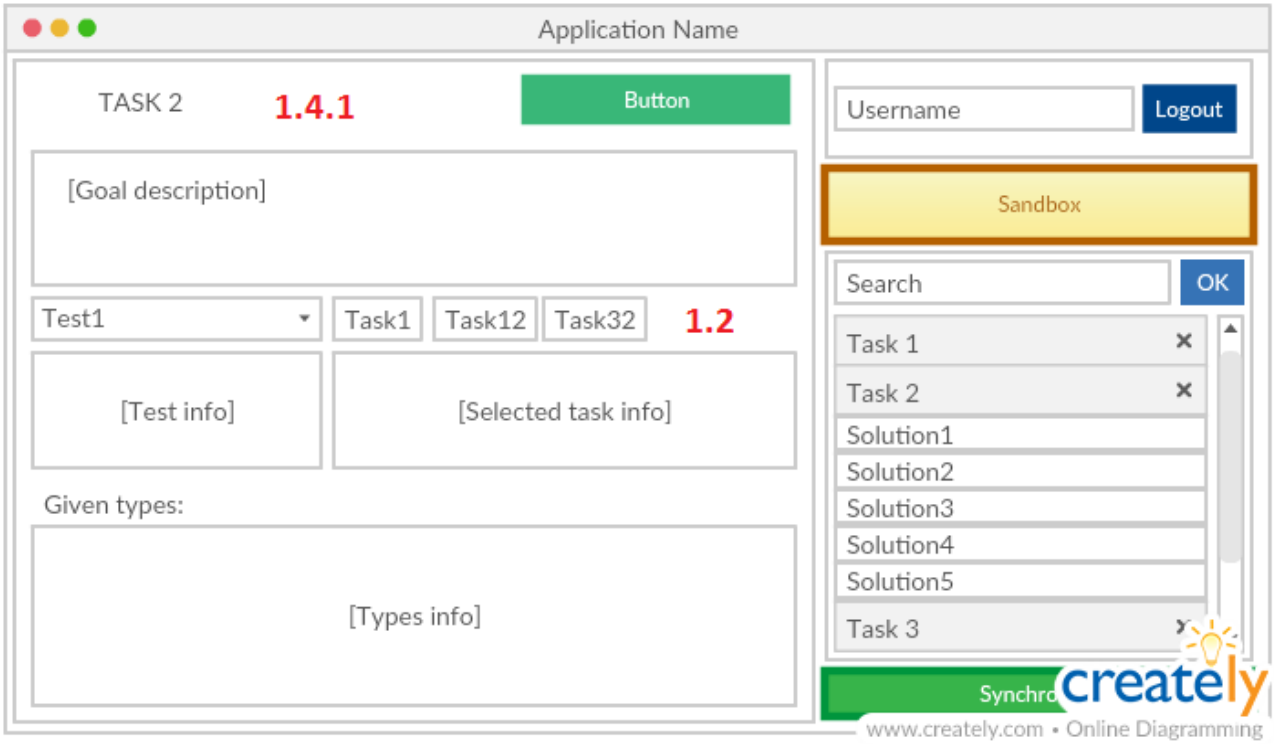

Рисунок 3.4 – Головне вікно програми з обраним завданням

Додаток спроектовано таким чином, що не має необхідності у відкритті вікон, окрім діалогових та допоміжних (вікон з похідним кодом тощо). Увесь контент знаходиться у головному вікні, тільки змінюється його наповнення. Це зроблено для того, щоб користувачу не було потрібно оперувати декількома відкритими вікнами, бо в цьому немає необхідності.

При обранні конкретного завдання показується інформація про нього: назва; опис; тести для перевірки, якщо вони є; типи, які можуть надаватися користувачу для виконання завдання; а також ланцюг з завданнями, які пов’язані з цим, як попередні, так і наступні, к яким можна перейти.

При настисканні кнопки початку нового рішення у завданні або у пісочниці, відкривається вікно з моделлю (рис. 3.5) та користувач може просуватися по

виконанню завдання на панелі знизу. Коли користувач впорався з певною частиною, він може перейти до наступної, а також повернутися до попередньої.

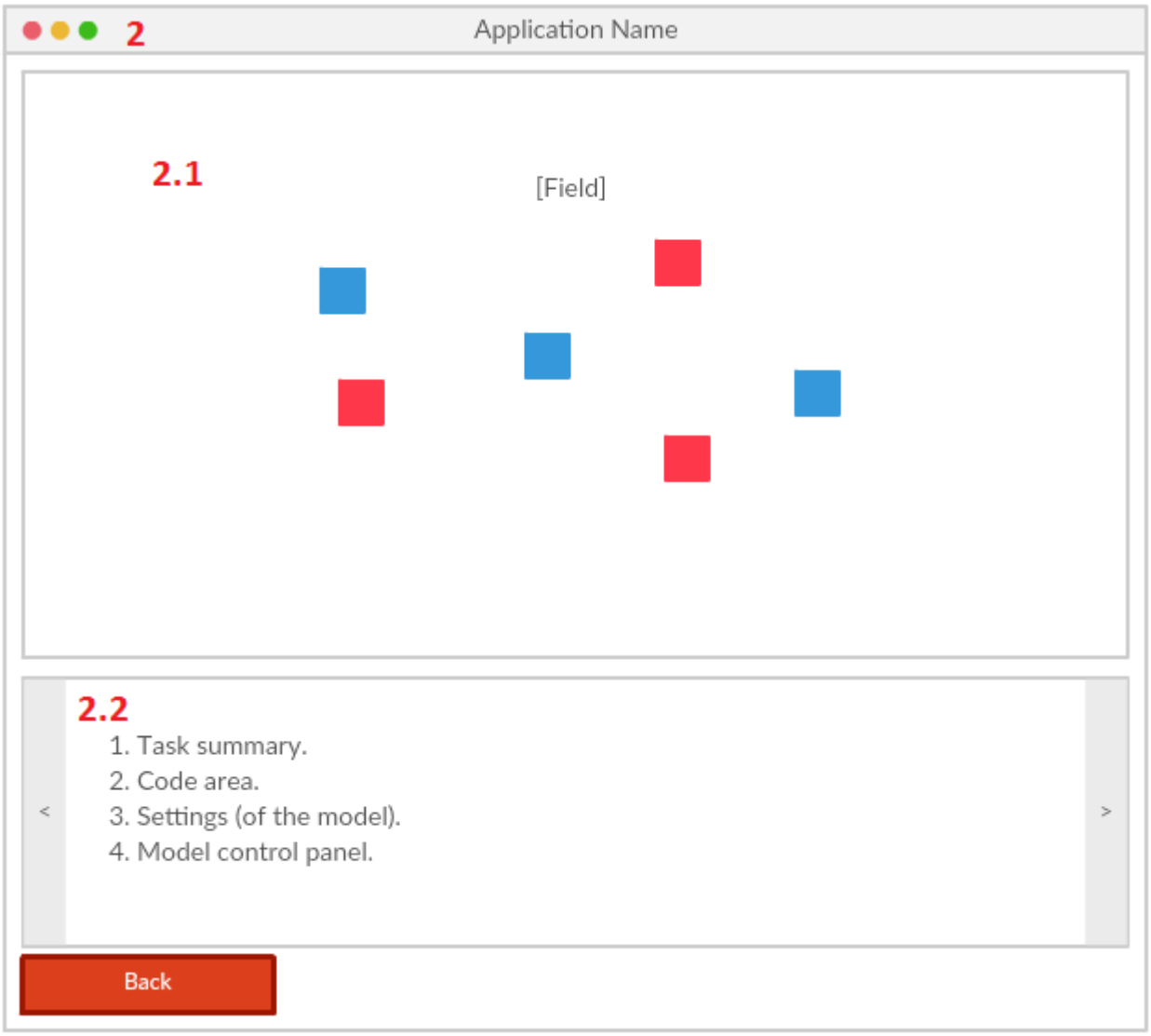

Рисунок 3.5 – Вікно рішення

На другій панелі відбувається основна робота – написання коду. Там можна створювати та редагувати файли з кодом. На панелі знизу можна працювати з поточною частиною виконання завдання. Вид кожної із панелей показано на рис. 3.6. Перша панель відображує інформацію о завданні, щоб з нею можна було у кожний момент можна було звіритися, не повертаючись до попереднього вікна.

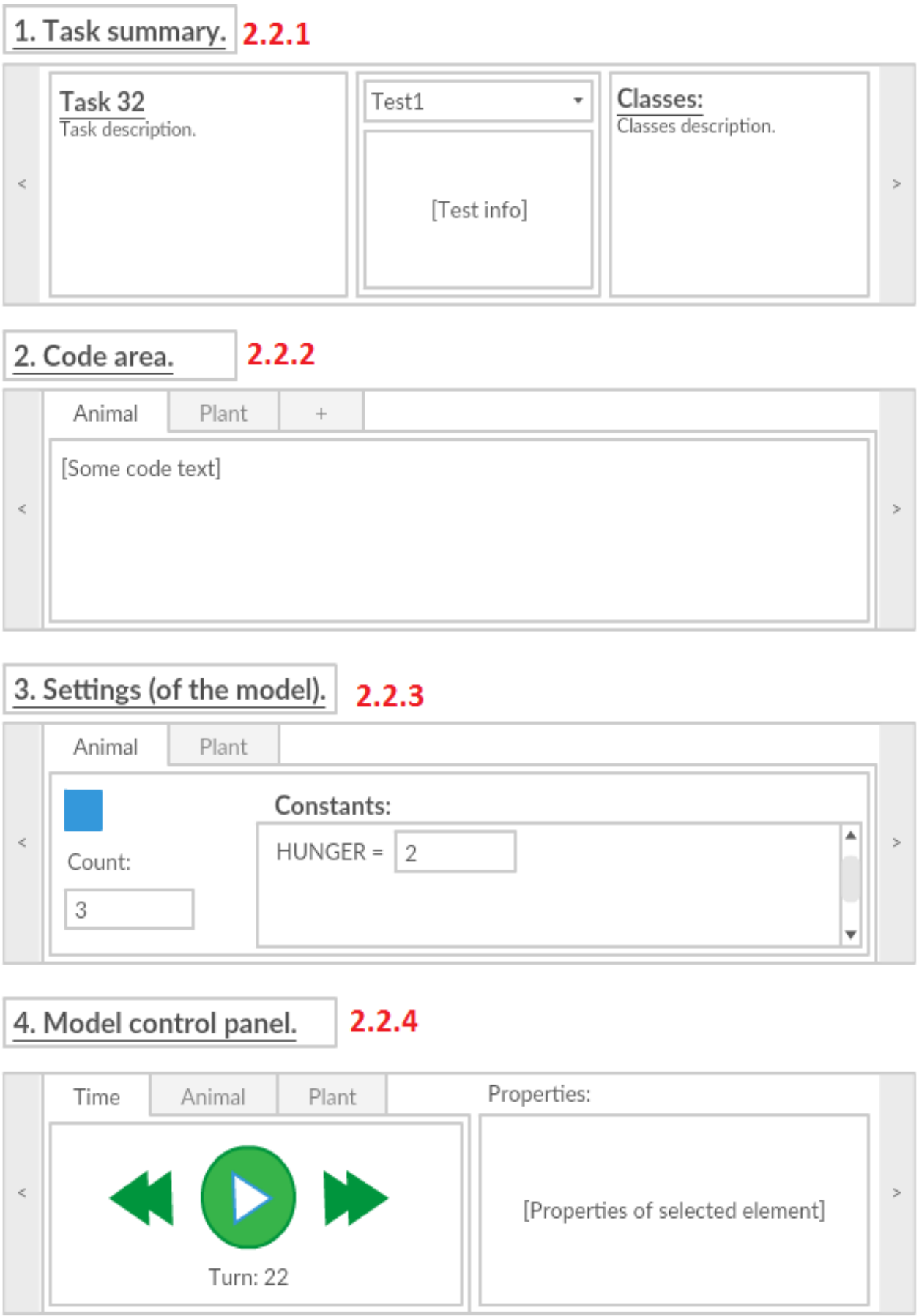

Рисунок 3.6– Панелі завдання

Третя панель відповідає за налаштування моделі. Тут можна виставити такі параметри як колір елементів на полі, їхню кількість на початку та метод заповнення поля елементами.

При переході до останньої моделі відображається модель і можна контролювати її роботу, переходячи до наступного ходу та переглядати дані об об’єктах.

У процесі розробки були створені розмітки на мові xaml для створення елементів користувацького інтерфейсу на основі цих макетів ще до того, як відбулось їх асоціювання з даними. Це допомогло відокремити логіку та дані від слою відображення. Деякі рішення щодо інтерфейсу користувача були змінені у процесі розробки. Наприклад, було вирішено дещо змінити вигляд панелі завдання для більш зручного розташування інформації. Але загальний вигляд залишився тим самим і знання того, як і які заплановано відображати дані сильно допомагає при розробці інших слоїв додатку.

ВИСНОВКИ

У цій роботі було розглянуто різні засоби візуалізації алгоритмів на різних структурах даних з точки зору навчання програмуванню та розробці алгоритмів з їхньою допомогою. Були порівняні існуючі засоби візуалізації та їхня прикладна значущість. Було виявлено, що існує багато засобів для візуалізації 3 основних структур даних: масивів, графів та клітинних автоматів.

Згідно з результатами було вирішено створити модифікацію клітинних автоматів, яка б була найбільше спрямована на розвиток навичок програмування при створенні нових правил та алгоритмів. Ця модифікація передбачає розширення математичної моделі клітинних автоматів. Розширення здобувається шляхом додавання нових вхідних параметрів, які повинні розширити діапазон можливих задач та створити модель, яка гарно переноситься на програмний код і надає максимум можливостей для розширення і модифікації.

Використовувати дану модель можна для написання фреймворку, який б містив у собі базові класи та інтерфейси для розробки коду, який б розширював модель новими правилами без необхідності писати код для візуалізації та відстеження стану моделі. Також структура моделі надає можливість використовувати різні шаблони програмування та інші гарні практики, використання яких допоможе користувачу краще їх зрозуміти та навчитися працювати із системами, розробленими іншими людьми через задані програмні інтерфейси.

Для подальшого дослідження слід підкреслити необхідність створити набір із 20 завдань у наступних групах (по 5 завдань на кожну): базове розуміння фреймворку, використання функції від загального стану поля, шаблони проектування та архітектурний підхід до створення складних додатків. На даному наборі завдань можна буде навчати програмістів більш складним навикам розробки у інтерактивному вигляді.

ПЕРЕЛІК ПОСИЛАНЬ

1. Logo History *Massachusetts Institute of Technology*: веб-сайт URL: http://el.media.mit.edu/logo-foundation/what_is_logo/history.html (дата звернення: 20.03.2020)

2. Papert S. Mindstorms: Children, Computers, and Powerful Ideas. New York. : Basic Books, 1980 – 230 с.

3. Visualizing Algorithms *Mike Bostock*: веб-сайт. URL: https://bost.ocks.org/mike/algorithms/ (дата звернення: 20.03.2020)

4. Моглан Д.В. Дидактический потенциал использования систем визуализации алгоритмов в процессе обучения программированию // Открытое образование. 2019. №2.

5. Algorithm Visualizer : веб-сайт. URL: https://algorithm-visualizer.org/ (дата звернення: 21.03.2020)

6. Visualising data structures and algorithms through animation: веб-сайт. URL : https://visualgo.net/ru (дата звернення: 21.03.2020)

7. Cellular Automata: веб-сайт. URL: https://www.microsoft.com/en-us/p/cellular-automata/9wzdncrdfwpl (дата звернення: 21.03.2020)

8. Дж. фон Нейман. Теория самовоспроизводящихся автоматов. М.: Мир, 1971. – 384 с.

9. Клумова И. Н. Игра «Жизнь» // Квант. 1974. № 9.

10. Холл, Г.М. Адаптивный Код на C#: проектирование классов и интерфейсов, шаблоны и принципы SOLID / пер. с англ. И. Берштейн. М.: Издательство «Вильямс», 2016 – 432 с.

11. Фаулер М. Рефакторинг: улучшение существующего кода. / пер. с англ. С. Маккавеева; С.-Пб.: Символ-Плюс, 2003. – 432 с.