МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ ім. Ігоря Сікорського»

Кафедра

автоматизованих систем обробки інформації та управління

**КУРСОВА РОБОТА**

з дисципліни

“ Розробка ігрових застосувань ”

на тему

" Розробка ігрового застосування Diploma Survive"

|  |  |  |
| --- | --- | --- |
| Керівник:  ст.вик. Коротенко А.А. |  | Виконали:  Кухарець Л.С.,  Ліпська В.О.,  Брідня В.І. |
| Допущений до захисту |  | студенти гр. ІП-61, ФІОТ |
| І\_\_\_І \_\_\_\_\_\_\_\_ 2020 \_\_\_\_\_\_  підпис |  | 4 курс |
| Захистив з оцінкою  \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_  оцінка підпис  І\_\_\_І \_\_\_\_\_\_\_\_\_ 2020 |  | Підпис: |

Київ 2020

**Загальні дані**

**Опис гри:**

Гра клікер-рандомайзер, що симулює навчання в університеті з великою ймовірністю бути виключеним.

Звичайне життя студента, для якого гравець приймає рішення. Герой має своє особисте життя, захоплення та бажання закінчити навчання. Прийняті рішення стосуються кожної сфери, але є фактори, які не залежать від нього, але впливають на його навчання. Таким чином, приділяючи більше зусиль для навчання, приватне життя може постраждати, і навпаки.

Навчання відбувається у вигляді імітації виконання лабораторних і набору очок досвіду в залежності від їх результатів. При наборі необхідної кількості очок, відкривається доступ до здачі іспитів, які призводять до підвищення рівня (переходу на новий семестр) або програшу (відрахування). Результати робіт формуються на основі як випадкових так і закономірних подій (причин).

**Цільова аудиторія:**

Студенти та школяри.

**Ціль гри:**

Успішно закінчити всі семестри університету, зайняти найвищу позицію на лідерборді (шляхом набрання балів) та/або відкрити всі причини відрахування.

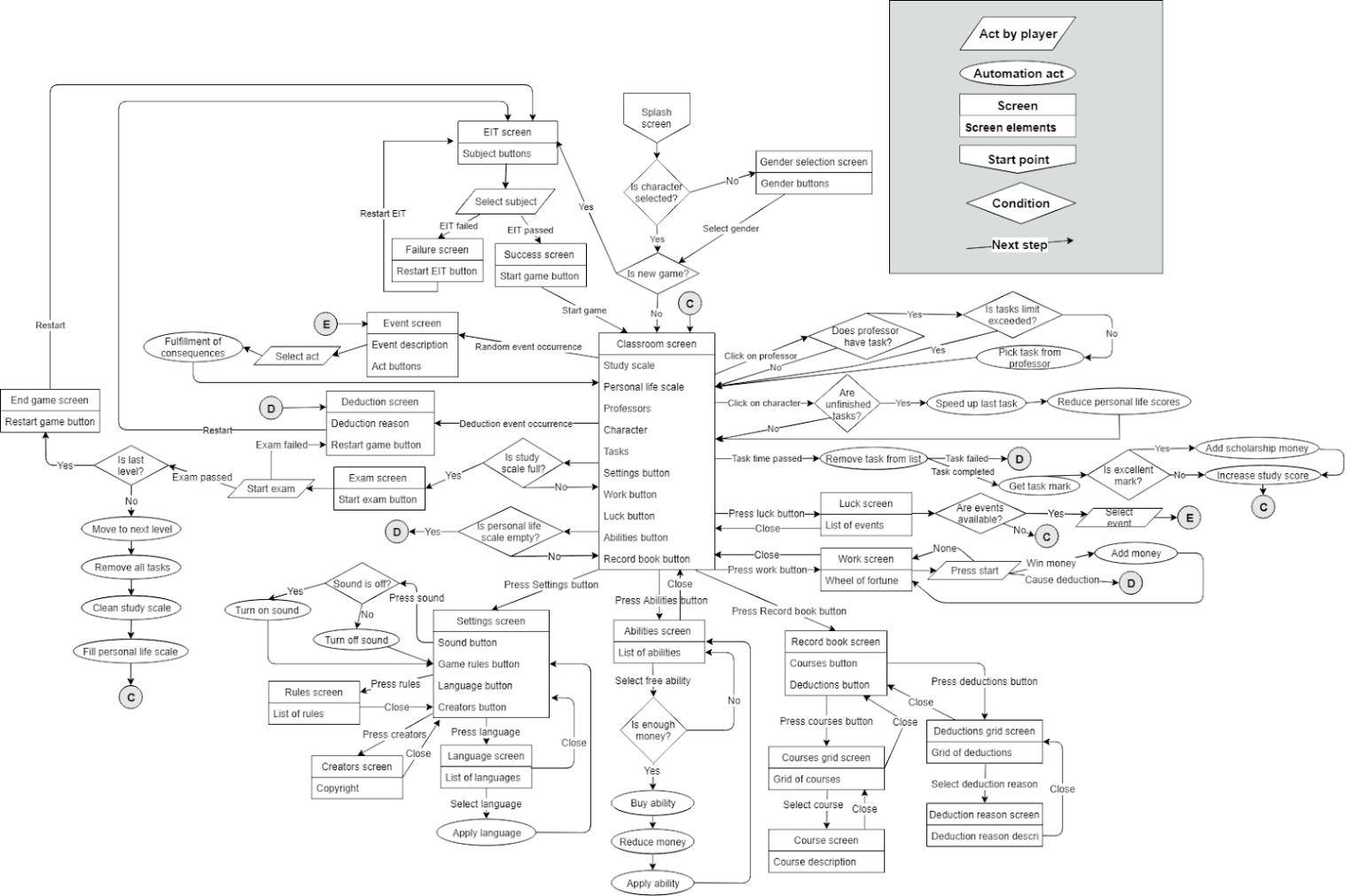
**Історія:**

Ви щойно вступили до університету і сповнені енергії, перші дні в кампусі вже заворожують, у вас є можливість познайомитися зі своїми однодумцями, приєднатися до команди та відкрити свої таланти, знайти нових друзів, а також знайти роботу та неповний день. Щодня ви робите різні варіанти, які впливають на ваше життя: деякі сфери вашого життя покращуються, а деякі погіршуються, іноді трапляються події, на які ви не можете вплинути, але вони впливають на вас. Незабаром до вас починають надходити різні завдання: виконати лабораторні роботи, відвідати лекцію, зробити доповідь, закінчити макет до півночі тощо. Завдання гравця - правильно розставити пріоритети та збалансувати всі сфери життя.

**Середовище розробки:**

Front-end: платформа Unity 2D, використовуючи мову програмування С#.

Back-end: сервер на .NET Core.

[](https://drive.google.com/file/d/1h1GU1qmrQEgOFeKzyFq8mEN-GnhXWtvP/view?usp=sharing*)

*Діаграма ходу гри*

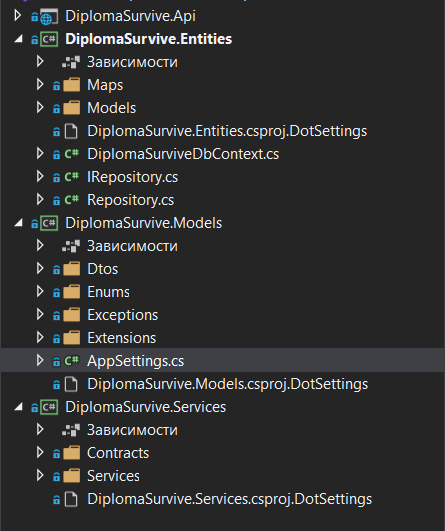
**Серверна частина**

Була необхідність реалізувати серверну частину для даної гри для створення «лідербордів», синхронізації набраних балів та керування змаганнями. Для описаної задачі був вибраний такий стек технологій: PostgreSQL і .NET Core 3.0. Завдяки кроссплатформеності сервер був розгорнутий на убунтовій машині разом з базою даних.

Логіка лідербордів працює таким чином: налаштовується подія, що приурочена до змагання. До прикладу, Різдво – приурочена подія, визначається дата початку та дата кінця – проміжок часу, що відповідає цьому турніру, а також налаштовуються подарунки за відповідні місця. Коли наступає дата фінішу, то відбувається фіналізація: гравці сортуються за їхніми балами і переможці отримують свої подарунки.

Також, при старті сервера запускається «шедулер», який дістає з бази даних ці налаштування і таким чином отримує інформацію, коли саме потрібно виконати фіналізацію балів (нагородження), до того ж, завдяки шедулеру, усі лідерборди, що старші за 1 місяць очищуються з бази даних. Для шедулера була використана стороння бібліотека – FluentScheduler. Окрім цього, запрограмовано, що з певною періодичністю налаштування перевіряються на нові, якщо такі знайдені, то старі перезаписуються знайденими, усі заплановані джоби (а саме на фіналізацію та очистку) відміняються та створюються нові.

Архітектура побудована таким чином, що у проекті знаходиться декілька бібліотек та вхідна точка:



*Структура проекту*

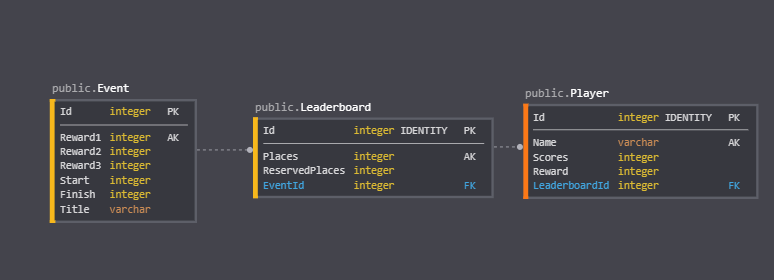
**DiplomaSurvival.Api** – це точка входу, тут усі контролери та «зв’язучі зі світом» для додатку, через цей проект приймаються запити та надається на них відповідь.

**DiplomaSurvival.Services** – цей проект зроблений для сервісів – прошарок між базою даних та контролерами.

**DiplomaSurvival.Models** – використовується як бібліотека моделей - різних data transfer object – ів, перечислень та констант проекту.

**DiplomaSurvival.Entities** - тут знаходиться опис і маппинг таблиць. Для маппингу обраний – fluent mapping.

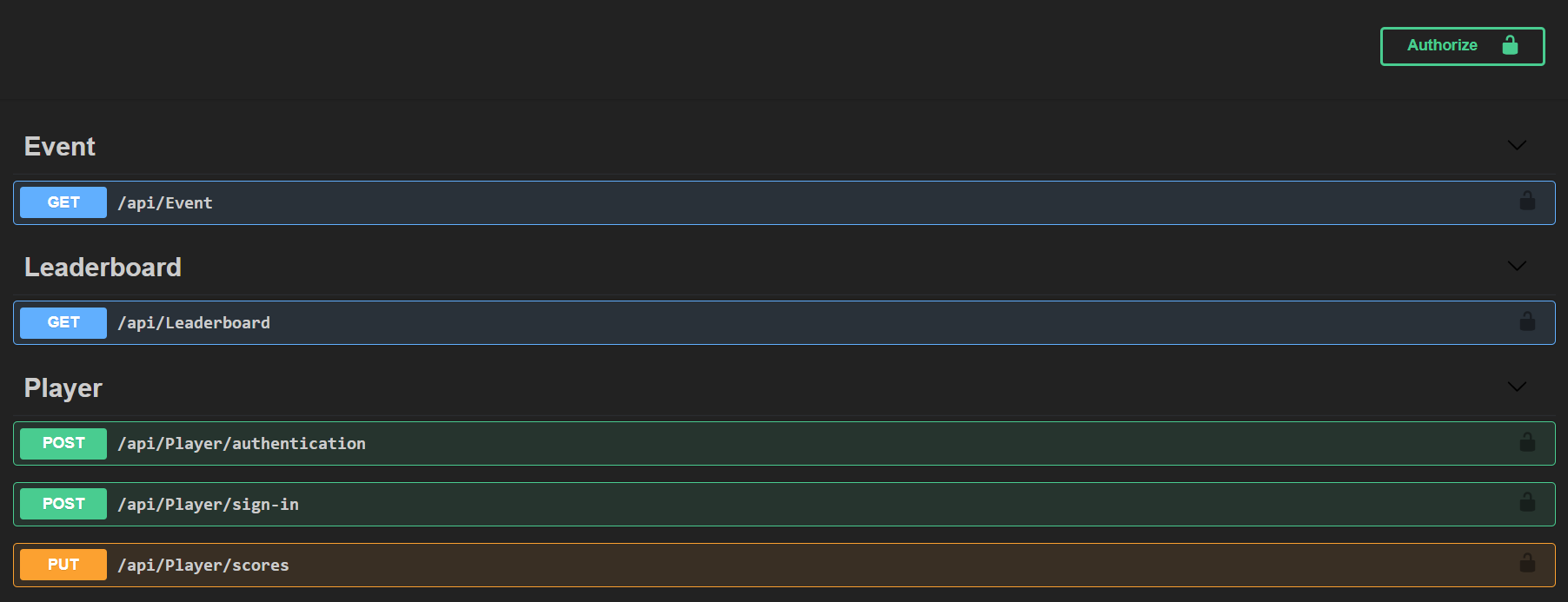
Структура бази даних досить нескладна: усього 3 таблиці, що зберігають у собі інформацію про події (нагорода за перше місце, нагорода за друге місце, нагорода за третє місце, дата початку, дата кінця та заголовок), про лідерборди (кількість виділений місць, кількість зайнятих місць та посилання на приурочену подію через зовнішній ключ) та про гравців (ім’я, бали, нагорода та посилання на лідерборд через зовнішній ключ).



*Структура бази даних*

Для швидкодії код написаний з урахуванням асинхронності та паралелізму.

Для взаємодією з API за допомогою swagger продемонстровано такі методи:



*Методи у API*

**Event:**

1. GET: api/event – запит для отримання усіх подій на даний момент.

**Leaderboard:**

1. GET: api/leaderboard – запит для отримання лідерборду гравця (розпізнавання гравця відбувається за токеном).

**Player:**

1. POST: api/player/authentication – запит для аутенфікації гравця (отримання токену).
2. POST: api/player/sign-in – запит для реєстрації гравця.
3. PUT: api/player/scores – запит для синхронізації балів гравця з сервером.

**Проект Unity**

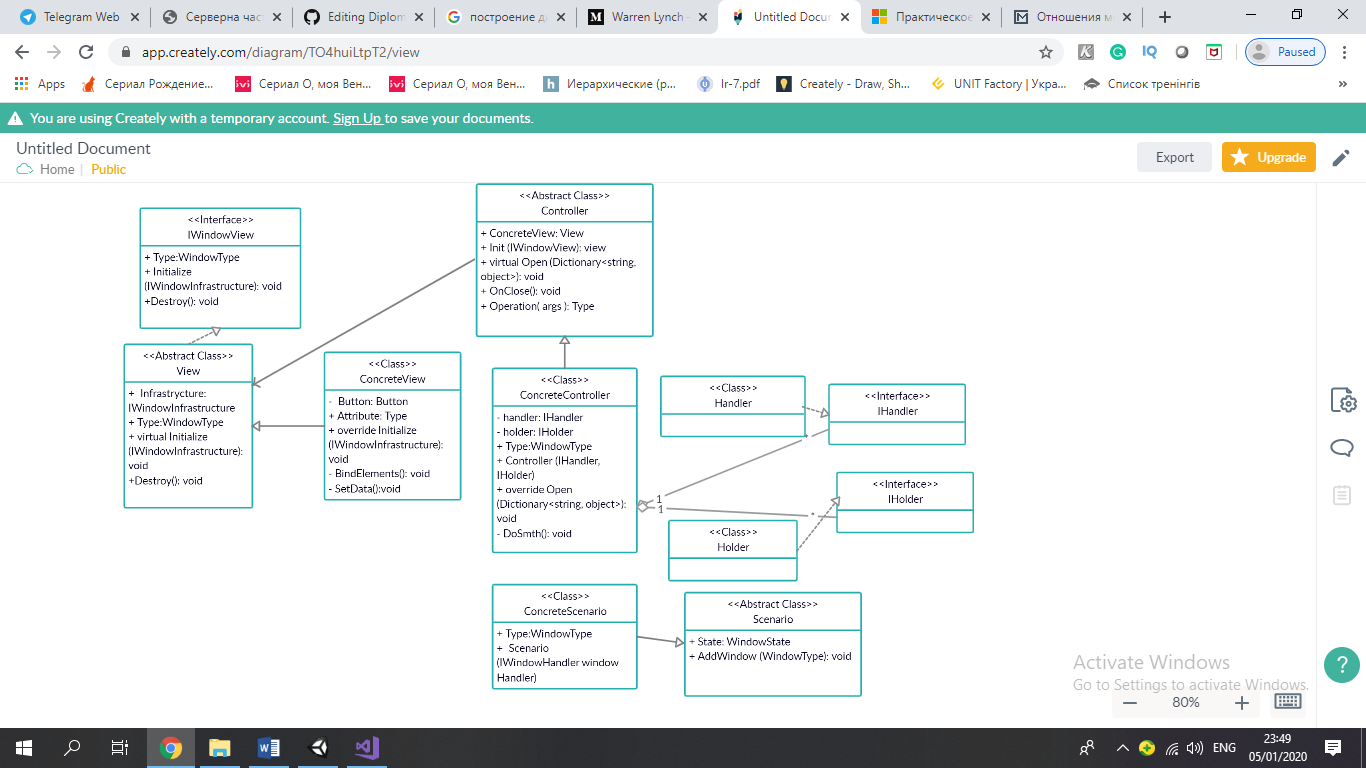
**Постановка задачі:** реалізувати фронтенд гри, що відтворює виникнення подій у системі, наприклад, виключення з університету або поява задачі від викладача, реагує на дії користувача та є легко розширюваним.

Для вирішення поставленої задачі було використано Unity та мову програмування C#.

Основна складність полягала у відділені представлення від класів обробки даних. Було вирішено зробити View нащадками MonoBehaviour, поставивши кожній у відповідність префаб, що відтворює частину меню, вікно чи попап. Обробку даних було здійчнено класами, що можуть функціонувати окремо від Unity.

Проект складається з декількох частин, таких як UI-компоненти, серіалізатор даних, класи обробки та збереження даних.

Зв’язок UI з класами обробки даних відбувається за схемою Model-View-Controller.



View відповідає за представлення конкретних елементів на сцені. Усі класи даного типу є нащадками класу View, що реалізує інтерфейс IWindowView. Кожному View відповідає префаб, що відображається на сцені. Кожна View обов’язково має тип Type, що обирається з enum WindowType. Також вью містить посилання на елементи префабу, що мають відображати дані користувача або реагувати на його дії.

View не знає про модель чи контролер. Але вона реалізує IPublisherAgg інтерфейс, що дозволяє їй публікувати події, на які підписані контролер та сценарій. При конкретних діях користувача (натисканні кнопки, виборі тогла тощо) публікується відповідна подія.

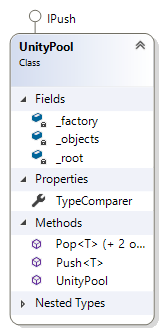
Controller відповідає за попередню обробку даних до їх потрапляння у View, викликає зміну моделі як реакцію на взаємодію з View і, навпаки, змінює View при зміні моделі. Кожен Controller обов’язково має тип Type, що обирається з enum WindowType, а також посилання на конкретну View, що відповідає йому. В конструкторі контролер отримує посилання на всі моделі та допоміжні інструменти, з якими він має працювати. Тут же відбувається початкова ініціалізація даних та підписка на події моделей. Метод Open контролера викликається під час появи префаба відповідної вью на сцені. Контролер може звернутись до відповідної йому вью чи моделей напряму.

Також на події моделі чи контролера підписаний Scenario. Його єдина функція – відкривати нові вікна. Кожен Scenario обов’язково має тип Type, що обирається з enum WindowType

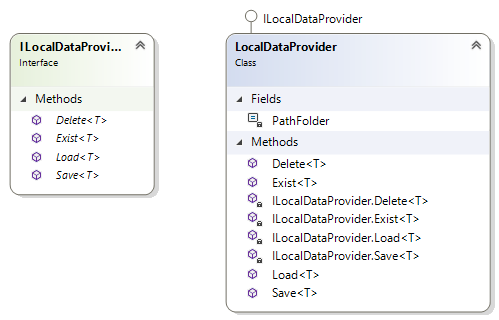
Хоча контролер моделі не є MonoBehaviour, в них можна використовувати корутини за допомогою допоміжного класу AsyncProcessor.

Моделі поділяються на handlers та holders. Хендлери обробляють дані, холдери їх лише серіалізують/десеріалізують та зберігають протягом ігрової сесії. Кожен холдер відповідає за окремий сет даних, наприклад, дані гравця, що часто змінюються під час гри в залежності від дій користувача, та дані гри, що можуть бути змінені лише з сервера. Хендлери відповідають за єдиний функціонал, як от виконання завдань, зміна прогресу тощо. Моделі не знають про контроллери, лише кикликають відповідні події. Але з моделі за необхідності можна відкрити нове вікно.

Було реалізовано патерн пул об’єктів. Він створений для ефективної маніпуляції інстанційованими об’єктами та вікнами, дозволяючи отримувати перші з них за типом їх класу, а другі – за типом enum WindowType. Основний принцип патерну – створення нового об’єкту лише тоді, коли у колекції повернених об’єктів немає підходящого. Коли об’єкт не потрібен, він не знищується, а лише вимикається і повертається у колекцію.



Для збереження поточного прогресу гравця та даних гри було написано клас LocalDataProvider, що дозволяє серіалізувати дані для їх збереження на пристрої/надсилання на сервер, та десереалізувати для використання під час ігрової сесії.



Для зручності роботи було використано фреймворк Zenject. Його застосування в проекті має низку переваг: використання класу як одинака без написання його таким (отже, і доступ до актуальних даних з будь-якої точки проекту), висока гнучкість коду (можливість змінити клас, що байндиться у єдиному місці проекту без впливу на інший код), можливість передати компоненти з редактора у класи, що не є нащадками MonoBehaviour.

**Бібліотека Data Generator**

**Постановка задачі:** побудувати систему, що відображає процесс перевірок на виключення в грі, а також виникнення екзаменів та подій, що впливають на ймовірності виключення.

Дана задача була реалізована шляхом створення бібліотеки .NET Framework мовою C#.

Основна складність полягала у створені такої програми, що не буде містить заздалегідь визначені перевірки/екзамени/події, а вони зможуть бути створенні вже під час під’єднання до UI та за необхідності додані/змінені в подальшому. Для цього вся система побудована таким чином, що зв’язки між класами задаються динамічно (через конструктори) й побудовані на інтерфейсах та абстрактних класах.

**Проблема:** необхідність обмінюватися великою кількістю параметрів з основною програмою та мати можливість реагувати на їх зміни.

**Рішення:** Для поєднання основної програми та бібліотеки був створений додатковий прошарок IPlayContext, що містить у собі декілька контекстів, через які бібліотеки має змогу отримувати, змінювати та реагувати та зміни даних під час гри. Всі контексти задані через інтерфейси та мають базові імплементації, які дозволяють оперувати даними лише у розрізі біблотеки. На кожну властивість у контексті є подія, яка реагує на зміну значення у цій властивості, а також є подія котра реагує на будь-яку зміну в контексті. Контексти: IMainContext – контекст, що містить поточний рівень та кількість грошей; IScoreContext – контекст, що відповідає за шкали приватного життя та навчання; IStudyContext – контекст, що містить додаткову інформацію про навчання; IEventContext – контекст, що містить основні події, які може викликати чи на які може реагувати програма.

**Проблема:** необхідність створення системи, що імітує процесс екзамену (як для сесії так і для ЗНО), в якій було б зручно створювати нові питання в залежності від відповіді на попередні та змінювати при цьому ймовірність виключення з університету.

**Рішення:** Так як на кожне питання завжди є два варіанти відповіді, екзамен розроблений у вигляді інтерфейсу IExam, що при старті повертає вказівник на вершину дерева. Кожним вузлом дерева дерева є сторінка ExamPage, що містить текст питання та кнопки ExamButton (що характеризують відповіді на питання), а також метод, що встановлює нову ймовірність. Кожна кнопка має метод, що повертає наступний вузел дерева, змінюючи ймовірність виключення (шляхом множення поточної на зазначений коефіцієнт), а у випадку відсутності наступного вузла повертає сторінку результату екзамену. Листями дерева є сторінки ExamPage, що мають тип Fail чи Success (прохід чи провал екзамену відповідно). Кожен екзамен містить початкову ймовірність, тип (сесія чи ЗНО) та рівень якому він відповідає (за замовчуванням екзамен може бути викликаний на будь якому рівні). Всі екзамени містяться у сховищі, та дістаються звідти у довільному порядку за допомогою сервісу IExamService (що містить окремі методи для сесії та ЗНО).

**Проблема:** необхідність створення системи, що імітує процесс виникнення випадкових подій, реакція на яку змінює поточний стан гри.

**Рішення:** Система виникнення подій реалізується за допомогою сервісу IPlayEventsService, який містить метод, що повертає довільну подію, та властивість, що вказує на наявність події. Кожна подія задана у вигляді сторінки PlayEvent, що містить текст події, та довільну кількість кнопок (реакцій на подію), які виконують певні дії при їх обрані. Також кожна подія містить метод, що вказує на доступність події в залежності від поточного стану гри.

**Проблема:** необхідність обробки великої кількості перевірок на виключення під час гри.

**Рішення:** Був створений сервіс IExclusionCheck, що повертає пустий об’єкт якщо виключення не відбулося, та об’єкт виключення в іншому випадку, а також містить властивість IsNecessary, що вказує на необхідність запуску перевірки. Базова реалізація цього сервісу працює шляхом виконання лише доступних перевірок ICheck у сховищі доки не настане виключення чи не закінчаться перевірки. Кожна перевірка ICheck містить тип виключення, необхідність запуску перевірки, ланцюг умов ICheckStep та метод, що повертає ймовірність виключення. Умови перевіряються за ланцбгом і відбувається перехід до наступної умови тільки якщо виконана попередня. За базовою реалізацією кожна умова підписана на зміни відповідних параметрів у контексті та змінює свій стан (необхідність запуску) при спрацюванні підписки. Коли стан умови змінюється (вона потребує перевірки), вона викликає подію необхідності перевірки, на яку реагує батьківський об’єкт, та за необхідності змінює свій стан. Якщо виконання перевірки провалюється її стан змінюється на відстуність необхідності перевірки. Таким чином, перевірка стає необхідною тільки тоді, коли будь яка її умова (раніше перевірена) стає необхідною. А сервіс перевірок, що підписан на кожну перевірку, змінює свій стан на необхідний до запуску, тільки тоді, коли будь-яка перевірка стає необхідною. Крім цього, кожна перевірка може мати свій пріоритет, який визначає в якому порядку вона буде виконана.

**Проблема:** необхідність зміни розподілу випадкових величин у подальшому.

**Рішення:** Створення інтерфейсу INumberGenerator, який містить методи генерування випадкових цілих та дробових величин в залежності від різних граничних значень. Цей генератор передається у необхідні класи за допомогою конструкторів чи властивостей у вигляді інтерфейсу. Базова реалізація використовує стандартний генератор випадкових величин в С# Random за рівномірним розподілом, проте це може бути змінено шляхом реалізації іншого розподілу (наприклад, нормального) на базі цього інтерфейсу.