МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

„КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. Сікорського”

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**КУРСОВА РОБОТА**

з дисципліни «Об’єктно-орієнтоване програмування»

на тему

**Шаблони проектування в ООП. Гра “П’ятнашки”**

Виконав студент

ІІ курсу групи КП-61

Свинарчук Максим Владиславович

залікова книжка КП-6126

Керівник роботи

доцент, к.т.н. Заболотня Т.М.

Оцінка

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

(дата, підпис)

КИЇВ 2018

**ЗМІСТ**

**ВСТУП**.........................................................................................................................4

1. **ОПИС СТРУКТУРНО-ЛОГІЧНОЇ ОРГАНІЗАЦІЇ ПРОГРАМИ**.........6
   1. Модульна організація програми............................................................6
   2. Функціональні характеристики.............................................................9
   3. Опис реалізованих класів.......................................................................10
   4. Основні алгоритми, реалізовані у програмі ....................................... 30
2. **ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ**.................................................................33
   1. Обґрунтування вибору та опис шаблонів проектування для реалізації програмного забезпечення гри “П’ятнашки”.............33
   2. Діаграма класів......................................................................................44
   3. Опис результатів роботи програми......................................................45

**ВИСНОВКИ**..............................................................................................................49

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**......................................................50

**ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ**

**КР -** курсова робота

**ООП -** об’єктно-орієнтоване програмування

**ПЗ -** програмне забезпечення

**CSS -** Cascading Style Sheets

**XML** **-** Extensible Markup Language (розширювана мова розмітки).

**FXML -** це мова розмітки користувальницького інтерфейсу на основі XML, створена корпорацією Oracle для конструювання інтерфейсу користувача платформою JavaFX.

**JavaFX** - платформа на основі Java для створення додатків з насиченим графічним інтерфейсом. Може використовуватися як для створення десктопних додатків, що запускаються безпосередньо з-під операційних систем, так і для інтернет-додатків, що працюють в браузерах, і для додатків на мобільних пристроях.

**ВСТУП**

Дана курсова робота присвячена розробці гри “П’ятнашки” за допомогою використання шаблонів проектування. “П’ятнашки” – популярна [головоломка](https://uk.wikipedia.org/wiki/%D0%93%D0%BE%D0%BB%D0%BE%D0%B2%D0%BE%D0%BB%D0%BE%D0%BC%D0%BA%D0%B0), придумана у [1878 році](https://uk.wikipedia.org/wiki/1878) Ноєм Чепменом. Складається з 15 однакових [квадратних](https://uk.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D0%B4%D1%80%D0%B0%D1%82) пластинок з нанесеними числами від 1 до 15 або частинами одного малюнка. Пластинки поміщаються в ігрове поле (квадрат), довжина сторони якого в чотири рази більша довжини сторони пластинок, відповідно залишається незаповненим одне квадратне поле. Мета гри – переміщаючи пластинки по коробці добитися впорядковування їх по номерах, бажано зробивши якомога менше переміщень.

Дана тематика обрана для виконання курсової роботи тому, що результати абстрагування об’єктів у цій предметній галузі дозволяють застосувати всі вивчені принципи та методи об’єктно-орієнтованого програмування для створення ПЗ, зокрема шаблони проектування.

**Об’єктом курсової роботи** є гра “П’ятнашки”, розроблена як десктопний додаток, який ні в чому не поступається класичному варіанту гри і навіть надає більший функціонал гравцю.

**Метою роботи** є розроблення програмного забезпечення длягри “П’ятнашки” з використанням шаблонів проектування.

Для досягнення описаної мети необхідно виконати такі **пункти завдань**:

* абстрагувати об’єкти предметної галузі;
* розробити структурну організацію програмного забезпечення за допомогою використання основних принципів ООП та шаблонів проектування;
* визначити та описати функціональні характеристики програми;
* обґрунтувати вибір шаблонів проектування;
* розробити графічний інтерфейс користувача;
* виконати реалізацію програмного забезпечення відповідно до технічного завдання;
* виконати тестування розробленої програми;
* оформити документацію з курсової роботи.

Розроблене ПЗ для гри “П’ятнашки” складається з таких логічних частин (модулів): основний модуль гри, який у свою чергу складається з модуля роботи з ігровим полем, модуля керування налаштуваннями гри та модуля роботи з таблицею результатів; модуль інтерфейса, модуль зв’язку інтерфейсу та програми та модуль допоміжних утиліт.

Реалізовані шаблони проектування: **Factory Method, Template Method, Prototype, Proxy (as Authenticate Proxy), Command, State, Observer, Facade, Singleton**.

Розроблене ПЗ може бути використане з розважальною метою та для розвитку розумових здібностей. Також як і будь-яка гра, яка змушує гравця міркувати логічно, шукати алгоритм для перемоги, дана гра може бути використана для навчання нейронних мереж.

До функціональних можливостей програми належать: вибір складності гри: 3х3 чи 4х4, відтворення звукових ефектів, вибір типу ігрового поля, можливість зупинити гру в будь-який час, можливість перезапуску гри в будь-який час, різні способи переміщення кнопок, підрахунок часу та кількості перемикань під час гри, збереження результату гри в таблицю рейтингу.

Для функціонування розробленої програми необхідно забезпечити наявність на комп’ютері 20 Мб вільного дискового простору та встановленого JDK 8u171.

Пояснювальна записка складається зі вступу, двох розділів, загальних висновків, списку використаних джерел (5-ти найменувань). Робота містить 62 рисунка. Загальний обсяг роботи – 50 друкованих сторінок.

1. **ОПИС СТРУКТУРНО-ЛОГІЧНОЇ ОРГАНІЗАЦІЇ ПРОГРАМИ**
   1. **Модульна організація програми**



Рис. 1.1.1. Модульна організація програми

**Модуль інтерфейсу користувача** – модуль відповідає за інтерфейс гри. Відображає ігрове поле, різні роду меню та кнопки з якими може взаємодіяти користувач. Модуль посилає запити до **модуля зв’язку інтерфейсу та програми**, необхідно оновити стан гри, змінити налаштування, тощо.

Для реалізації використовується мова розмітки FXML, таблиці стилів CSS та платформа JavaFX.

**Модуль зв’язку інтерфейсу та програми –** модуль відповідає за взаємодію між модулем інтерфейсу користувача та іншими модулями, які містять в собі основну логіку гри: основний модуль гри, модуль роботи з таблицею результатів та модуль керування налаштуваннями гри. Він відповідальний за своєчасне оновлення інтерфейсу та об’єктів гри при взаємодії користувача з програмою.

**Основний модуль гри –** модуль Game Model (Модель гри), який містить у собі основні сутності та реалізує 4 стани, в яких може перебувати гра (свого роду скінченний автомат). Встановлює взаємодію з модулем роботи з ігровим полем та модулем допоміжних утиліт.

**Модуль роботи з ігровим полем –** модуль, який напряму взаємодіє з основним модулем гри, контролює ігрове поле, відповідальний за правильну зміну положень кнопок, спосіб їх переміщення та інший функціонал системи. На основі стану ігрового поля може повідомити основний модуль про завершення гри. Виконує свого роду роль “розуму” в основному модулі гри.

**Модуль допоміжних утиліт –** модуль, який загалом ніяк не впливає на стан гри. Надає іншим модулям системи об’єкти та функціонал для виконання певних операцій, щоб спростити роботу іншим модулям. Наприклад, дає можливість завантаження та обробки зображень, відтворення звукових ефектів, роботи з таймером.

**Модуль керування налаштуваннями гри –** модуль, відповідальний за те, яким чином буде відпрацьовувати основний модуль гри. Гравець при зміні налаштувань гри працює саме з даним модулем (звичайно через модуль зв’язку інтерфейсу та програми) і він, в свою чергу, впливає (видозмінює поведінку) на модуль гри. Модуль підтримує зберігання та відтворення свого стану при вимкненні – старту гри.

**Модуль роботи з таблицею результатів –** модуль, відповідальний за збереження результатів гри та формування рейтингу. Модуль підтримує зберігання та відтворення свого стану при вимкненні – старту гри.

* 1. **Функціональні характеристики**

Взаємодія з грою розпочинається зігрового поля (відтвореного за попередніми налаштуваннями). В даний момент гра очікує на початок. Як тільки гравець почне взаємодіяти з кнопками на ігровому полі, відразу запускається відлік часу. Кожне переміщення кнопки рахується, а лічильник зображується на екрані гри.

Гравець в будь-який момент гри може поставити паузу – лічильник часу зупиниться, а ігрове поле буде сховане. Після вимкнення паузи – гра продовжиться з того ж моменту. Коли гра активна, а гравець відчиняє меню налаштувань чи таблицю рекордів – гра автоматично вмикає паузу.

У будь-який момент гравець може натиснути клавішу “Shuffle”, що призведе до наступних змін:

– якщо гра була почата, то всі результати будуть скинуті, ігрове поле перемішане, гра перейде в стан очікування;

* якщо гра в стані очікування, то поле просто буде перемішане. Причому якщо ігрове поле містить зображення – воно буде змінене на інше.

Під час гри гравець може відмінити всі зроблені ним переміщення, чи повернутися в останнє положення (якщо гравець вже зробив відміну). Звичайно ці дії будуть розцінюватися як звичайні переміщення і збільшувати лічильник ходів.

Гравець може переглянути таблицю рекордів, а також занести в неї власний результат, якщо закінчить гру. Також гравець може змінити налаштування гри:

* тип ігрового поля;
* складність гри;
* спосіб перемикання кнопок;
* чи потрібно підсвічувати кнопки на правильних позиціях;
* чи потрібно відтворювати звукові ефекти;
* режим “Hardmode”.

Якщо налаштування були змінені – гра перейде в стан очікування та змінить свій функціонал згідно з новими налаштуваннями.

* 1. **Опис реалізованих класів**

1. **Клас ButtonPrototype**

Абстрактний клас, є батьківським класом для класів **ImageButton** та **TextButton**, є класом «Prototype» для шаблону Прототип, є класом «Context» для шаблону Стан. Даний клас є мінімальною одиницею в моделі гри.

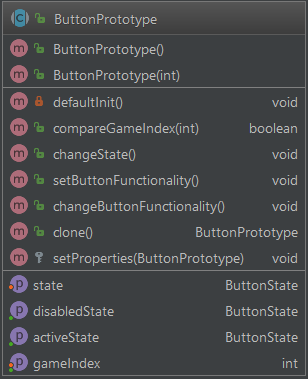


Рис. 1.3.1. Клас ButtonPrototype

*Методи:*

* clone – виконує клонування самого себе, та використовуючи метод setProperies, надає клону ті ж самі властивості, що і у оригінала. Метод setProperies необхідний, оскільки об’єкти платформи JavaFX не підтримують клонування і втрачають при цьому всі властивості;
* changeButtonFunctionality – змінює стан кнопки (викликається метод changeState) та надає їй властивості нового стану (викликається метод

setButtonFunctionality)

* compareGameIndex – порівнює ігровий індекс, який зберігає кнопка з числом, що прийшло на вхід.

1. **Клас ImageButton**

Клас, що наслідується від **ButtonPrototype**, представляє собою кнопку, яка містить у собі зображення. Є класом «ConcretePrototype» для шаблону Прототип.

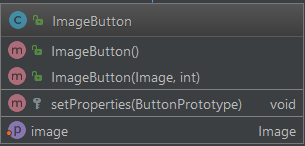


Рис. 1.3.2. Клас ImageButton

1. **Клас TextButton**

Клас, що наслідується від ButtonPrototype, представляє собою кнопку, яка містить у собі текст. Є класом «ConcretePrototype» для шаблону Прототип.

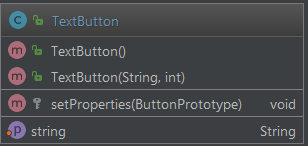


Рис. 1.3.3. Клас TextButton

1. **Клас ButtonState**

Абстрактний клас, що оголошує стандартну структуру стану кнопки, є класом «State» для шаблону Стан.

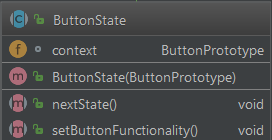


Рис. 1.3.4. Клас ButtonState

Поле context – зберігає посилання на кнопку, яка містить цей стан.

*Методи:*

* nextState – перемикає стан context, в залежності від його стану к даний момент часу.
* setButtonFunctionality – змінює властивості кнопки, характерні для даного стану.

1. **Клас ActiveButton**

Клас, що наслідується від **ButtonState**, представляє собою активний стан кнопки, з якою може взаємодіяти користувач. Є класом «ConcreteState» для шаблону Стан.

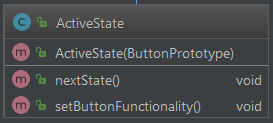


Рис. 1.3.5. Клас ActiveButton

1. **Клас DisabledButton**

Клас, що наслідується від **ButtonState**, представляє собою неактивний стан кнопки. Користувач не може взаємодіяти з кнопкою, що знаходиться в цьому стані. Є класом «ConcreteState» для шаблону Стан.

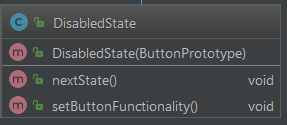


Рис. 1.3.6. Клас DisabledButton

1. **Клас ButtonCache**

Клас, який відповідає за створення, а точніше за клонування кнопок, що зберігаються в кеші. Є частиною шаблона Прототип.

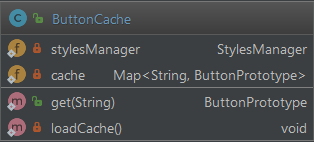


Рис. 1.3.7. Клас ButtonCache

1. **Клас GameBoard**

Абстрактний клас, є батьківським класом для класів **ImageBoard** та **TextBoard**, є класом «AbstractClass» для шаблону Шаблонний метод, є класом «Product» для шаблону Фабричний метод, є класом «Receiver» для шаблону Команда.



Рис. 1.3.8. Клас GameBoard

*Поля:*

* buttons – контейнер, що містить всі кнопки ігрового поля;
* difficulty – визначає, якого розміру дошку буде згенеровано при виклику методу init.

*Методи:*

* init – власне і є шаблонний метод, визначає алгоритм наповнення дошки кнопками;
* createBtn – абстрактний метод, що використовується в шаблонному методі init. Класи наслідники, визначають яка саме кнопка буде створюватися, надають їй певні властивості та повертають її як результат виконання цього методу;
* swapButtons – міняє місцями дві кнопки на дошці.
* shuffleGameBoard – випадковим чином перемішує ігрову дошку (детальніше див. пункт 1.4);
* isSolvable – метод, що визначає, чи існує можливість перемогти при даному розташування кнопок на дошці (детальніше див. пункт 1.4).

1. **Клас ImageBoard**

Клас, що наслідується від **GameBoard**, представляє собою ігрову дошку, кнопки якої містить у собі частини цільного зображення. Є класом «ConcreteProduct» для шаблону Фабричний метод.



Рис. 1.3.9. Клас ImageButton

*Поля:*

* boardImages – масив, що містить у собі частини цільного зображення. Використовується для створення кнопок з зображенням.

*Методи:*

* createBtn – метод, який є частиною алгоритму в шаблонному методі в класі-предку. У даному класі метод повертає кнопку з зображенням.
* changeImage – замінює картинку ігрового поля на нову, передану як параметр у методі.

1. **Клас TextBoard**

Клас, що наслідується від **GameBoard**, представляє собою ігрову дошку, кнопки якої містить у собі текст. Є класом «ConcreteProduct» для шаблону Фабричний метод.

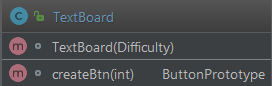


Рис. 1.3.10. Клас TextBoard

*Методи:*

* createBtn – метод, який є частиною алгоритму в шаблонному методі в класі-предку. У даному класі метод повертає кнопку з текстом.

1. **Клас GameBoardCreator**

Клас, який відповідає за створення ігрового поля. Є класом «ConcreteCreator» для шаблону Фабричний метод (а саме – Параметризований Фабричний метод).



Рис. 1.3.11. Клас GameBoardCreator

*Методи:*

* createGameBoard – повертає нову дошку, в залежності від параметрів, що були передані в метод, а саме тип ігрового поля та складність.

1. **Інтерфейс IObserver**

Інтерфейс, що визначає метод update. Є інтерфейсом «Observer» для шаблону Observer.

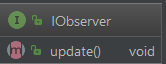


Рис. 1.3.12. Інтерфейс IObserver

1. **Клас Subject**

Абстрактний клас, є батьківським класом для класів **GameModel** та **SettingsModel**, є класом «Subject» для шаблону Спостерігач.

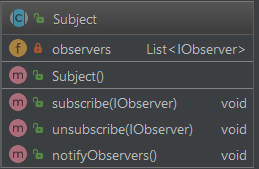


Рис. 1.3.13. Клас Subject

*Поля:*

* observers – список спостерігачів, які реалізують інтерфейс **IObserver**, та бажають знати про зміни об’єкта класу **Subject.**

*Методи:*

* subscribe – додати спостерігача;
* unsubscribe – видалити спостерігача;
* notifyObservers – повідомити всіх спостерігачів, про зміну об’єкта спостереження.

1. **Клас SettingsModel**

Клас, що наслідується від **Subject**, представляє собою модель налаштувань гри. Є класом «ConcreteSubject» для шаблону Спостерігач та класом «Singleton» для шаблону Одинак.

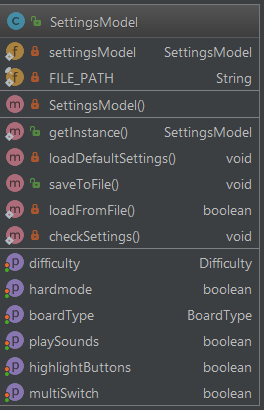


Рис. 1.3.14. Клас SettingsModel

Таким чином під час виконання програми, може існувати лише один об’єкт даного класу. На нього підписується об’єкт класу **GameModel**, щоб слідкувати за змінами налаштувань гри і, в разі чого, оновлювати свій стан.

*Методи:*

* getInstance – повертає об’єкт класу;
* saveToFile – зберігає стан налаштувань у файл у форматі XML;
* loadFromFile – відновлює стан налаштувань з файла у форматі XML;
* checkSettings – перевіряє налаштування після завантаженні їх з файлу, на випадок, якщо файл будо пошкоджено і стан об’єкту відновився некоректно.

1. **Клас GameModel**

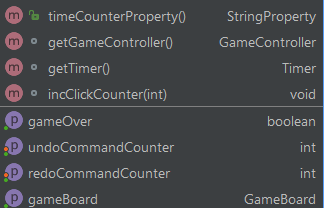
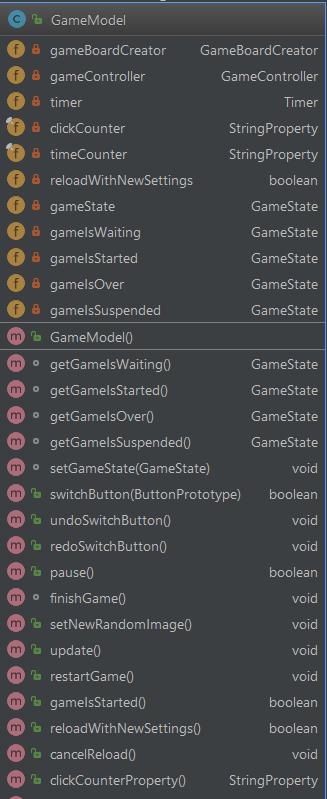


Рис. 1.3.15. Клас GameModel

Головний клас програми, який власне й визначає логіку ігри. Містить у собі всі інші класи, які хоч якимось чином задіяні в процесі гри (мається на увазі ті класи, що представляють собою якусь частинку моделі гри, а не її представлення для користувача у вигляді інтерфейсу). Є класом «Facade» для шаблону Фасад, класом «ConcreteSubject» для шаблону Спостерігач (на модель підписується клас **MainController**), та водночас класом «ConcreteObserver» для шаблону Спостерігач (підписується на оновлення класу **SettingsModel**), класом «Context» для шаблону Стан. Саме у конкретних станах гри описана основна логіка.

*Методи:*

* update – метод визивається в разі зміни об’єкту налаштувань гри. Тоді модель викликає метод restartGame.
* restartGame – метод скидає стан гри в даний момент часу, та перезавантажує всі поля об’єкту. Зазвичай викликається при зміні налаштувань гри в методі update. Також може викликатися, якщо гравець бажає почати гру знову при тих же налаштуваннях (в такому випадку не створюється нове ігрове поле і контроллер гри).
* setNewRandomImage – якщо ігрове поле з картинкою – замінює цю картинку на іншу.

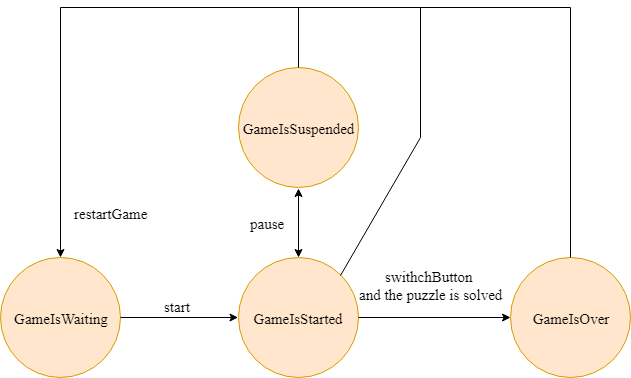


Рис. 1.3.16. Діаграма переходів GameModel в різні стани

1. **Клас GameState**

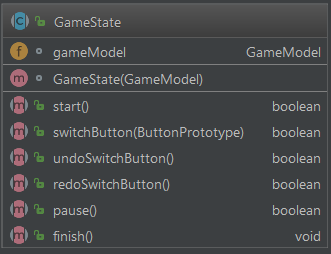


Рис. 1.3.17. Клас GameState

Абстрактний клас, що оголошує стандартну структуру стану гри, є класом «State» для шаблону Стан.

Поле gameModel – зберігає посилання на модель гри

*Методи:*

* start – абстрактний метод, який реалізується у класах, що наслідуються від даного;
* switchButton – абстрактний метод, який реалізується у класах, що наслідуються від даного;
* undoSwitchButton – абстрактний метод, який реалізується у класах, що наслідуються від даного;
* redoSwitchButton – абстрактний метод, який реалізується у класах, що наслідуються від даного;
* pause – абстрактний метод, який реалізується у класах, що наслідуються від даного;
* finish – абстрактний метод, який реалізується у класах, що наслідуються від даного.

1. **Клас GameIsWaiting**

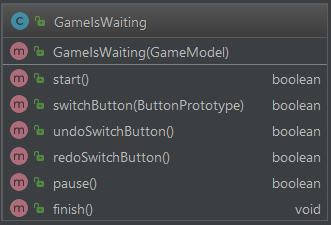


Рис. 1.3.18. Клас GameIsWaiting

Клас, що наслідується від **GameState**, представляє собою стан гри, яка очікує на початок. Є класом «ConcreteState» для шаблону Стан.

Містить логіку лише в одному методі start – запускає таймер гри, та переводить модель гри в стан **GameIsStarted.**

1. **Клас GameIsStarted**

Клас, що наслідується від **GameState**, представляє собою стан гри, яка триває в даний момент часу. Є класом «ConcreteState» для шаблону Стан.

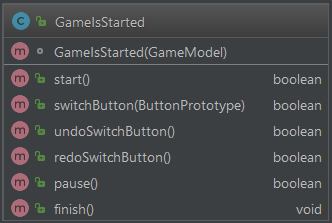


Рис. 1.3.19. Клас GameIsStarted

Містить логіку лише в наступних методах:

* switchButton – використовуючи **GameController**, перемикає кнопки. У разі вдалого перемикання збільшує лічильник перемикань, повідомляє **MainController,** що відбулися зміни. Якщо дошка була складна – переводить стан модель гри в стан **GameIsOver**;
* undoSwitchButton – відміняє останнє перемикання кнопок;
* redoSwitchButton – якщо було відмінене останнє перемикання кнопок, повертає їх в положення перед відміною;
* pause – зупиняє таймер, переводить модель гри в стан **GameIsSuspended**.

1. **Клас GameIsSuspended**

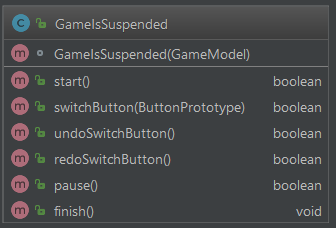


Рис. 1.3.20. Клас GameIsSuspended

Клас, що наслідується від GameState, представляє собою стан гри, яка очікує на початок. Є класом «ConcreteState» для шаблону Стан.

Містить логіку лише в одному методі pause – відновлює відлік часу, переводить модель гри в стан **GameIsStarted**.

1. **Клас GameIsOver**

Клас, що наслідується від GameState, представляє собою стан гри, яка очікує на початок. Є класом «ConcreteState» для шаблону Стан.

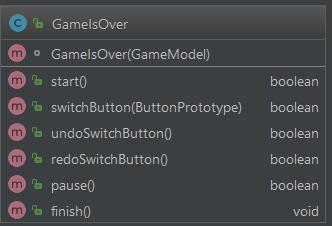


Рис. 1.3.21. Клас GameIsOver

Містить логіку лише в одному методі finish – зупиняє таймер гри.

1. **Клас HighScoreUnit**

Клас, який представляє собою одиничний результат гри гравця, використовується в **HighScoresModel**.

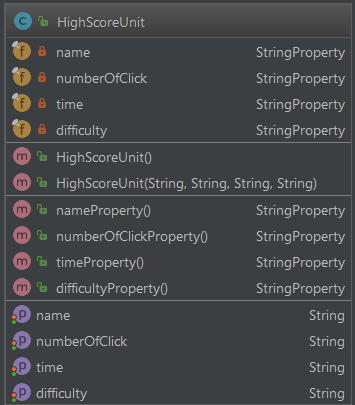


Рис. 1.3.22. Клас HighScoreUnit

1. **Клас HighScoresModel**

Клас, що представляє собою модель результатів гри. Містить у собі список об’єктів **HighScoreUnit**, які власне і зберігають одиничні результати. Є «Singleton» для шаблону Одинак. Таким чином під час виконання програми, може існувати лише один об’єкт даного класу.

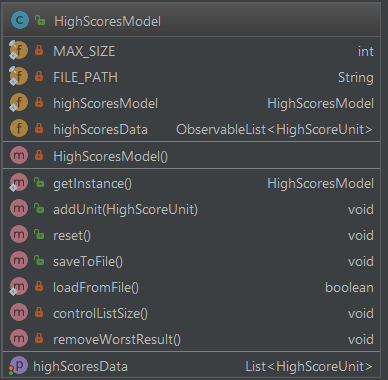


Рис. 1.3.23. Клас HighScoresModel

*Методи:*

* getInstance – повертає об’єкт класу;
* saveToFile – зберігає стан налаштувань у файл у форматі XML;
* loadFromFile – відновлює стан налаштувань з файла у форматі XML;
* controlListSize – перевіряє чи не переповнений список результатів, якщо так – викликає метод removeWorstResult, що видаляє найгірший результат за певними критеріями.

1. **Інтерфейс IHighlight**

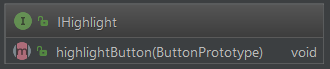


Рис. 1.3.24. Інтерфейс IHighlight

Інтерфейс, який визначає метод highlightButton. Є інтерфейсом «Subject» для шаблону Заступник.

1. **Клас StylesManager**

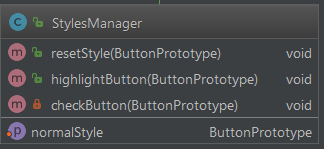


Рис. 1.3.25. Клас StylesManager

Клас, який реалізує інтерфейс **IHighlight**. Загалом призначений для зміни стилю кнопок. Є класом «RealSubject» для шаблону Заступник.

1. **Клас GameController**

Клас відповідальний, за перемикання кнопок в ігровій моделі, спосіб перемикання та підсвічення кнопок. Реалізує інтерфейс **IHighlight** і є класом «Proxy» для шаблону Заступник – виступає як заступник-захисник для **StylesManager**.

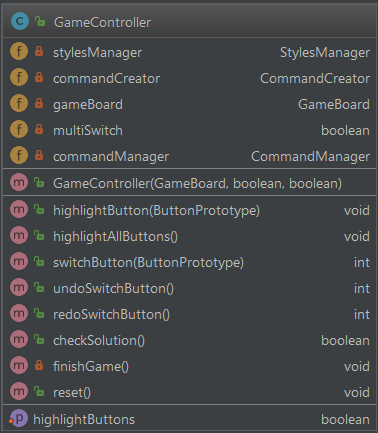


Рис. 1.3.26. Клас GameController

Містить у собі **CommandManager** (шаблон Команда), який зберігає команди перемикання кнопок та за необхідності може відмінити їх.

*Методи:*

* highlightButton – перевіряє умови доступу до об’єкта **StylesManager**;
* switchButton – створює команду перемикання для переданої кнопки, виконує цю команду та заносить її в **CommandManager**;
* checkSolution – перевіряє чи розташування кнопок у даний момент часу є вирішення головоломки;

1. **Клас Command**

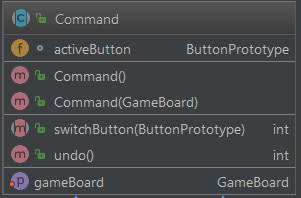


Рис. 1.3.27. Клас Command

Абстрактний клас, що оголошує стандартну структуру команди перемикання кнопок, є класом «Command» для шаблону Команда, класом «Product» для шаблону Фабричний метод.

Поле activeButton – зберігає посилання на кнопку, яку необхідно передати в метод switchButton, щоб відмінити перемикання зроблені цією командою.

*Методи:*

* switchButton – абстрактний метод, який реалізується у класах, що наслідуються від даного;
* undo – відміняє перемикання кнопок, зроблені цією командою.

1. **Клас MultiSwitch**

Клас, що наслідується від **Command**, представляє собою команду, яка підтримує перемикання множини кнопок, тобто якщо гравець, наприклад натискає на верхню кнопку колонки, а пуста клітинка міститься внизу цієї колонки, то всі кнопки колонки будуть зміщені вниз, а пуста клітинка опиниться вгорі. Є класом «ConcreteCommand» для шаблону Команда, класом «ConcreteProduct» для шаблону Фабричний метод. Реалізацію даного алгоритму можно переглянути в пункті 1.4.3

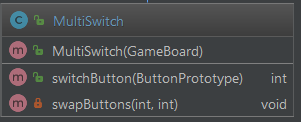


Рис. 1.3.28. Клас MultiSwitch

1. **Клас SingleSwitch**

Клас, що наслідується від **Command**, представляє собою команду, яка підтримує тільки одиничне перемикання кнопок, тобто перемкнутися може лише та кнопка, що стоїть поруч в рядку або стовпчику з пустою клітинкою. Є класом «ConcreteCommand» для шаблону Команда, класом «ConcreteProduct» для шаблону Фабричний метод. Реалізацію даного алгоритму можно переглянути в пункті 1.4.2

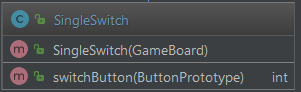


Рис. 1.3.29. Клас SingleSwitch

1. **Клас CommandCreator**

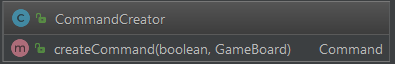


Рис. 1.3.30. Клас CommandCreator

Клас, який відповідає за створення команди. Є класом «ConcreteCreator» для шаблону Фабричний метод (а саме – Параметризований Фабричний метод).

*Методи:*

* createCommand – повертає нову команду, в залежності від параметрів, що були передані в метод.

1. **Клас CommandManager**

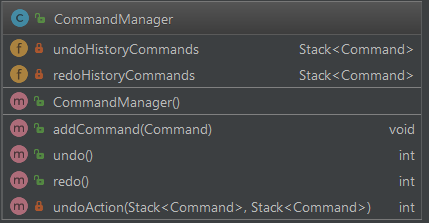


Рис. 1.3.31. Клас CommandManager

Клас, що відповідальний за зберігання команд при перемиканні кнопок в **GameController**, виконанні операцій відмани цих команд. Є класом «Invoker» для шаблону Команда.

Містить у собі два стека: undoHistoryCommand – стек з командами для відміни, redoHistoryCommand – стек з командами для повернення кнопок в стан перед відміной. Ці стеки взаємозв’язані – команди з одного стеку при виконанні операції відміни переміщуються в інший стек і навпаки.

*Методи:*

* addCommand – додає команду в стек undoHistoryCommand;
* undo – виконує операцію відміни для першої команди зі стеку undoHistoryCommand та додає цю команду на вершину стеку redoHistoryCommand;
* redo – виконує операцію відміни для першої команди зі стеку redoHistoryCommand та додає цю команду на вершину стеку undoHistoryCommand.

1. **Клас MainController**

Клас, який відповідальний за взаємозв’язок моделі гри (**GameModel**) та її представлення – головного вікна програми. Є класом «ConcreteObserver» для шаблону Спостерігач (підписується на оновлення класу **GameModel**).

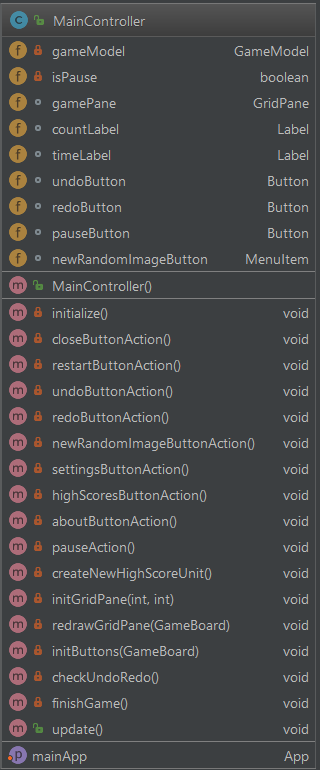


Рис. 1.3.32. Клас **MainController**

1. **Клас SettingsDialogController**

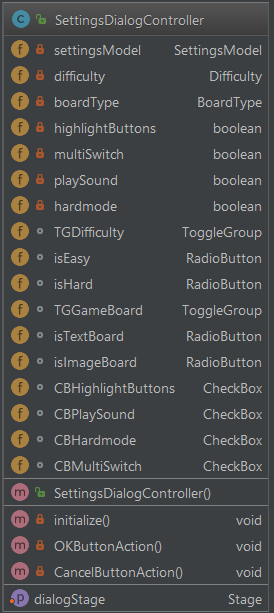


Рис. 1.3.33. Клас SettingsDialogController

Клас, який відповідальний за взаємозв’язок моделі налаштувань (**SettingsModel**) та її представлення – вікна налаштувань гри.

1. **Клас HighScoresController**

Клас, який відповідальний за взаємозв’язок моделі таблиці рекордів (**HighScoresModel**) та її представлення – вікна таблиці рекордів.

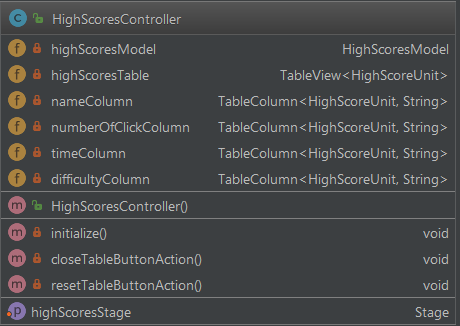


Рис. 1.3.34. Клас HighScoresController

1. **Клас ImageProcessing**

Клас із пакету «helpers», реалізовує додатковий функціонал, щоб полегшити розробку основного ПЗ.

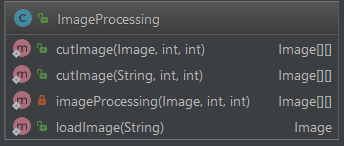


Рис. 1.3.35. Клас ImageProcessing

*Методи:*

* cutImage – розрізає цільне зображення на частини та повертає масив цих частинок зображення;
* loadImage – завантажує зображення за переданим в мметод шляхом.

1. **Клас RandomImage**

Клас із пакету «helpers», реалізовує додатковий функціонал, щоб полегшити основного ПЗ.

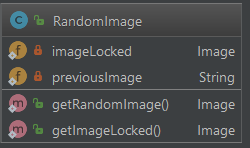


Рис. 1.3.36. Клас RandomImage

*Методи:*

* getRandomImage – завантажує та повертає рандомну картинку з директорії images в проекті;
* getImageLocked – завантажує та повертає картинку для порожнього квадрата з директорії images в проекті.

1. **Клас Timer**

Клас із пакету «helpers», реалізовує додатковий функціонал, щоб полегшити основного ПЗ. Використовується в моделі гри, для збільшення лічильника часу кожну секунду.

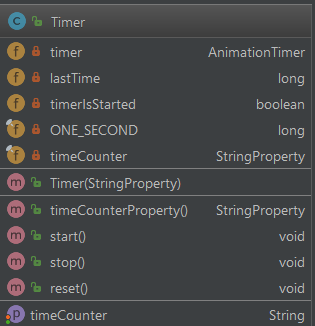


Рис. 1.3.37. Клас Timer

*Методи:*

* start – починає відлік часу;
* stop – зупиняє відлік часу;
* reset – скидує таймер.

1. **Клас Player**

Клас із пакету «helpers», реалізовує додатковий функціонал, щоб полегшити основного ПЗ.

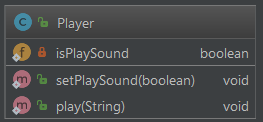


Рис. 1.3.38. Клас Player

*Методи:*

* play – відтворює звуковий ефект, шлях до якого передано в метод.

1. **Клас App**

Клас програми, який є “точкою входу” при завантаженні гри. Ініціалюзує головне вікно гри.

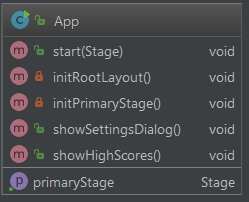


Рис. 1.3.39. Клас App

*Методи:*

* start *–* завантажує головне вікно гри, створює об’єкт класу **MainController**;
* showSettingsDialog – завантажує вікно програми та його контролер **SettingsDialogController**;
* showHighScores – завантажує вікно таблиці рекордів та його контролер **HighScoresController**.

1. **Enum BoardType**



Рис. 1.3.40 Enum BoardType

1. **Enum Difficulty**

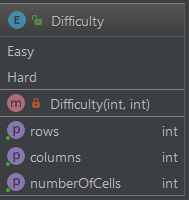


Рис. 1.3.41. Enum Difficulty

* 1. **Основні алгоритми, реалізовані у програмі**

1. **Алгоритм перемішування кнопок ігрового поля (реалізований у класі GameBoard).**

Випадкове розміщення кнопок на ігровому полі не є правильним, адже існує такі варіанти розміщення, коли зібрати “П’ятнашки” не є можливим.

Розглянемо математичне підґрунтя цієї проблеми в її класичному варіанті 4х4:

Можна показати, що рівно половину з усіх можливих 20 922 789 888 000 (=16!) початкових положень п’ятнашок неможливо привести до зібраного стану: нехай квадратик з числом *i* розташований до (якщо вважати зліва направо і зверху вниз) *k* квадратиків з числами меншими *i*. Будемо вважати *ni = k*, тобто якщо після квадрату з *i*-м числом немає чисел, менших *i*, то *k = 0*. Число *ni* показує **кількість інверсій** для клітинки з номером *і*. Також введемо число *e* - номер ряду порожньої клітинки (рахуючи з 1).

Якщо сума є непарною, то рішення головоломки не існує.

Отже, необхідно написати алгоритм, який рахує цю суму N для конкретного ігрового поля. Він виглядає наступним чином:

|  |
| --- |
| private boolean isSolvable() {  int inv = 0;  for (int i = 0; i < difficulty.getNumberOfCells(); i++)  if (buttons.get(i) != buttonLocked) {  for (int j = 0; j < i; j++)  if (buttons.get(j).getGameIndex() > buttons.get(i).getGameIndex() &&  buttons.get(j).getGameIndex() != difficulty.getNumberOfCells() - 1) {  ++inv;  }  }  if (difficulty == Difficulty.*Hard*)  inv += 1 + this.getIndexOfRow(this.buttonLocked);  return inv % 2 == 0; } |

Це універсальний варіант алгоритму, який враховує той факт, що для парного значення клітинок необхідно додавати номер строки пустої клітинки, а для непарного – це не потрібно.

Отже, тепер можна перевірити, чи існує рішення для конкретного стану головоломки. Але що робити у випадку, якщо його не існує. Для того, щоб постійно не перемішувати дошку, доки рішення не знайдеться пропонується наступний алгоритм:

|  |
| --- |
| public void shuffleGameBoard() {  Collections.shuffle(this.buttons);  if (!isSolvable()) {  if (getIndexOf(buttonLocked) <= 1) {  swapButtons(difficulty.getNumberOfCells() - 2, difficulty.getNumberOfCells() - 1);  } else {  swapButtons(0, 1);  }  } } |

Пояснення: спочатку просто перемішуємо ігрове поле, використовуючи стандартний метод в мові програмування java для колекцій. Далі перевіряємо чи існує рішення для головоломки:

* якщо рішення існує – мета досягнута;
* якщо рішення не існує – потрібно поміняти місцями два квадрати поруч у одному рядку, але при цьому обов’язково врахувати той факт, що обидва з цих квадратів не є пустою клітинкою. При такому обміні сума N стане праною і головоломку буде мати рішення.

1. **Алгоритм перемикання кнопок, в якому при одному натисканні відбувається лише одне перемикання (реалізований у класі SingleSwitch).**

|  |
| --- |
| public int switchButton(ButtonPrototype pressedButton) {  int numOfColumns = gameBoard.getColumns();  int indexOfPressedBtn = gameBoard.getIndexOf(pressedButton);  int indexOfLockedBtn = gameBoard.getIndexOf(gameBoard.getButtonLocked());  int indexDifference = indexOfPressedBtn - indexOfLockedBtn;  boolean pressedBtnInFirstCol = gameBoard.getIndexOfCol(pressedButton) == 0;  boolean pressedBtnInLastCol = gameBoard.getIndexOfCol(pressedButton) == numOfColumns - 1;  boolean lockedBtnInFirstCol = gameBoard.getIndexOfCol(gameBoard.getButtonLocked()) == 0;  boolean lockedBtnInLastCol=gameBoard.getIndexOfCol(gameBoard.getButtonLocked())==numOfColumns -1;  if (Math.abs(indexDifference) != 1 && Math.abs(indexDifference) != numOfColumns ||  Math.abs(indexDifference) == 1 &&(pressedBtnInFirstCol && lockedBtnInLastCol || pressedBtnInLastCol && lockedBtnInFirstCol)) {  Player.play("/sounds/clickMouseBang.wav");  return 0; }  if ((Math.abs(indexDifference) == 1 ||  Math.abs(indexDifference) == numOfColumns) &&  (Math.abs(indexDifference) != 1 ||  (!pressedBtnInLastCol || !lockedBtnInFirstCol) &&  (!pressedBtnInFirstCol || !lockedBtnInLastCol))) {  Player.play("/sounds/clickMouse.wav");  gameBoard.swapButtons(indexOfPressedBtn, indexOfLockedBtn);}  activeButton = pressedButton; return 1;} |

1. **Алгоритм перемикання кнопок, в якому при одному натисканні відбувається здвиг рядка чи стовпчика – багато перемикань (реалізований у класі MultiSwitch).**

|  |
| --- |
| public int switchButton(ButtonPrototype pressedButton) {  int clickCounter = 0;   int indexOfPressedBtn = gameBoard.getIndexOf(pressedButton);  int indexOfLockedBtn = gameBoard.getIndexOf(gameBoard.getButtonLocked());   int colOfPressedBtn = gameBoard.getIndexOfCol(pressedButton);  int rowOfPressedBtn = gameBoard.getIndexOfRow(pressedButton);  int colOfLockedBtn = gameBoard.getIndexOfCol(gameBoard.getButtonLocked());  int rowOfLockedBtn = gameBoard.getIndexOfRow(gameBoard.getButtonLocked());  if (colOfPressedBtn != colOfLockedBtn && rowOfPressedBtn != rowOfLockedBtn) {  Player.play("/sounds/clickMouseBang.wav");  System.out.println("Push blocked = " + indexOfPressedBtn + " cell");  } else {  if (colOfPressedBtn == colOfLockedBtn) {  if (rowOfLockedBtn < rowOfPressedBtn) {  for (int i = rowOfLockedBtn + 1; i <= rowOfPressedBtn; i++) {  clickCounter++;  int btnIndexForSwitch = indexOfLockedBtn + gameBoard.getColumns() \* (i - rowOfLockedBtn);  swapButtons(btnIndexForSwitch, gameBoard.getIndexOf(gameBoard.getButtonLocked()));  }  } else {  for (int i = rowOfLockedBtn - 1; i >= rowOfPressedBtn; i--) {  clickCounter++;  int btnIndexForSwitch = indexOfLockedBtn - gameBoard.getColumns() \* (rowOfLockedBtn - i);  swapButtons(btnIndexForSwitch, gameBoard.getIndexOf(gameBoard.getButtonLocked()));  }  }  } else {  if (colOfLockedBtn < colOfPressedBtn) {  for (int i = colOfLockedBtn + 1; i <= colOfPressedBtn; i++) {  clickCounter++;  int btnIndexForSwitch = indexOfLockedBtn + (i - colOfLockedBtn);  swapButtons(btnIndexForSwitch, gameBoard.getIndexOf(gameBoard.getButtonLocked()));  }  } else {  for (int i = colOfLockedBtn - 1; i >= colOfPressedBtn; i--) {  clickCounter++;  int btnIndexForSwitch = indexOfLockedBtn - (colOfLockedBtn - i);  swapButtons(btnIndexForSwitch, gameBoard.getIndexOf(gameBoard.getButtonLocked()));  }  }  }  Player.play("/sounds/clickMouse.wav");  activeButton = gameBoard.getButton(indexOfLockedBtn);  }  return clickCounter; }  private void swapButtons(int firstBtnIndex, int secondBtnIndex ) {  System.out.println("Change place = " + firstBtnIndex + " cell to " + secondBtnIndex + " cell");  gameBoard.swapButtons(firstBtnIndex, secondBtnIndex); } |

1. **ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ**
   1. **Обґрунтування вибору та опис шаблонів проектування для реалізації програмного забезпечення гри “П’ятнашки”**
2. **Factory Method**

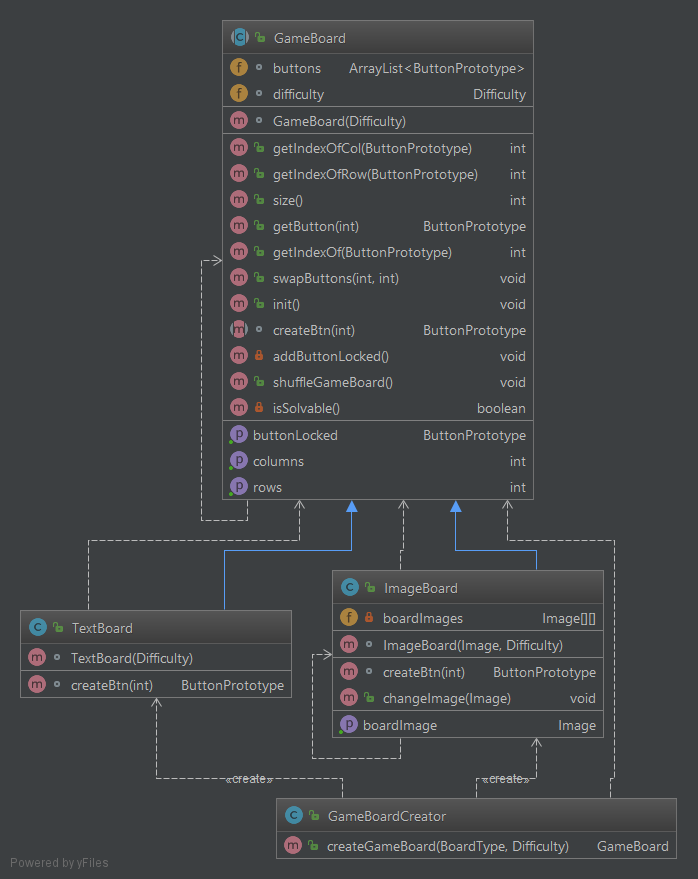


Рис. 2.1.1. UML-діаграма шаблону “**Factory Method**” для ігрового поля

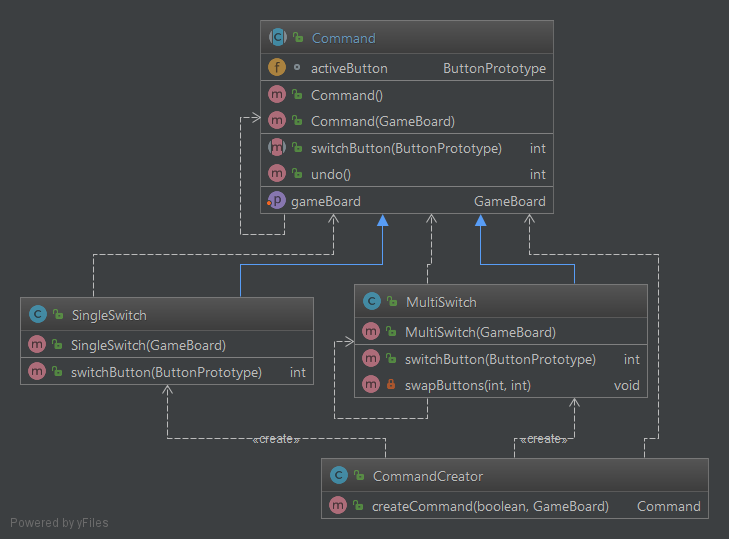


Рис. 2.1.2. UML-діаграма шаблону “**Factory Method**” для команди перемикання

*Породжуючий шаблон*. Визначає інтерфейс для створення об'єктів, але рішення про те, який саме об’єкт створювати, залишає за підкласами.

*Структура*. Реалізований у двох варіантах:

* Для створення різних типі ігрового поля (Рис. 2.1.1): клас **GameBoard** виступає в ролі «Product», а класи, що від нього наслідуються **ImageBoard** та **TextBoard** в ролі «ConcreteProduct». Клас **GameBoardCreator** -«ConcreteCreator», він містить у собі один метод, який приймає тип ігрового поля та складність і на основі цих параметрів повертає створене ігрове поле.
* Для створення команд перемикання кнопок (Рис. 2.1.2): клас **Command** виступає в ролі «Product», а класи, що від нього наслідуються **SingleSwitch** та **MultiSwitch** в ролі «ConcreteProduct». Клас **CommandCreator** -«ConcreteCreator», він містить у собі один метод, який повертає створену команду, клас якої залежить від параметрів, переданих у метод..

Обидва варінти є реалізаціями параметризованого Фабричного метода.

*Обґрунтування використання даного шаблону*.

Після створення системи класів, яка представляє різновид наявних у програмі команд, та ігрових полів треба було зручним чином організувати створення цих об’єктів за допомогою якихось сутностей. Так як кожна з команд має абстрактний клас **Command** у якості базового класу (фундаменту) (а ігрове поле у якості базового класу має абстрактний клас **GameBoard**), то було прийняте рішення про використання структури шаблону параметризований “**Factory Method**” за основу у проектуванні цих частин системи.

1. **Template Method**

*Поведінковий шаблон*. визначає функціональність конкретних методів в рамках лише абстрактних сутностей. Визначає основу алгоритму та дозволяє підкласам перевизначити деякі кроки алгоритму, не змінюючи структуру в цілому.

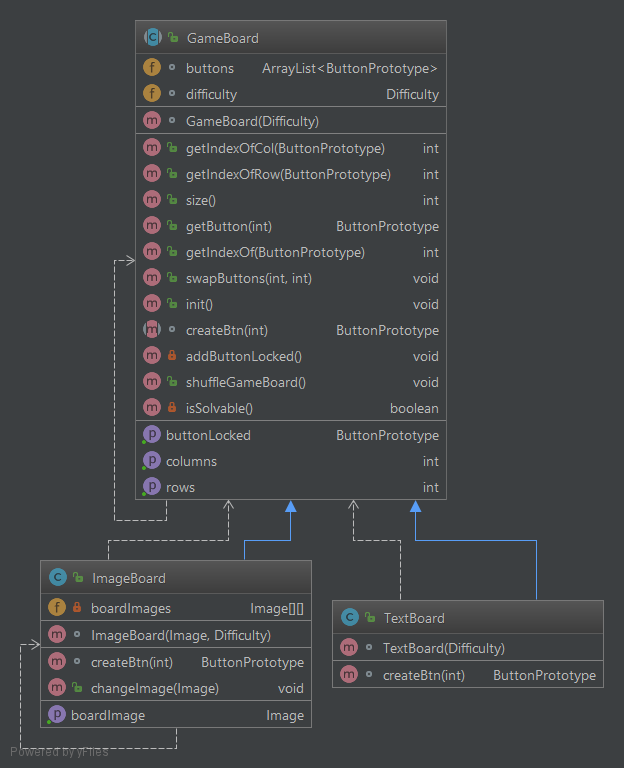


Рис. 2.1.3. UML-діаграма шаблону “**Template Method**”

*Структура*. Клас **GameBoard** виступає в ролі абстрактного класу, який в шаблонному методі init локалізує поведінку, що є загально для всіх підкласів (**ImageBoard** та **TextBoard**) і виносить в абстрактний метод createBtn поведінку, яка повинна бути реалізована для кожного класу самостійно.

*Обґрунтування використання даного шаблону.* При написанні ПЗ завжди слід уникати дублювання коду розділення. Використання даного шаблону надає хороший приклад техніки “винесення за лапки з метою узагальнення”. При чому результат виконання шаблонного метода залежить від класу нащадку, в той час як основний алгоритм залишається незмінним.

1. **Prototype**

*Породжуючий шаблон*. Задає види створюваних об’єктів за допомогою екземпляра-прототипу і створює нові об’єкти шляхом копіювання цього прототипу. Це єдиний шаблон з серії Породжуючих, котрий для створення нових об’єктів використовує не явне інстанціювання (ключове слово new), а клонування.

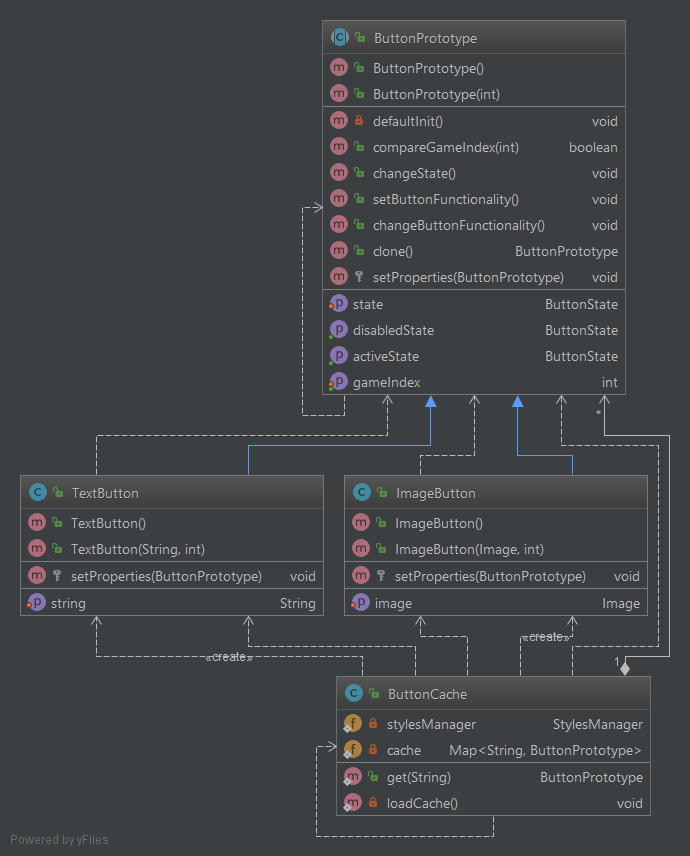


Рис. 2.1.4. UML-діаграма шаблону “**Prototype**”

*Структура*.Реалізований для зручного отримання нових об’єктів кнопок, без потреби постійно надавати їм одні й ті ж самі властивості. Клас **ButtonPrototype** виступає в ролі «Prototype», а класи-нащадки **ImageButton** та **TextButton** в ролі «ConcretePrototype». **ButtonCache** ініціалізує нові об’єкти кнопок, надає їм певні властивості, загальні для різних видів кнопок та зберігає їх в кеш. Надалі, витягаючи з кешу, та клонуючи ці об’єкти можна отримувати кнопки з потрібними властивостями.

*Обґрунтування використання даного шаблону*. Даний шаблон дає змогу абстрагуватися від способу створення та реалізації об’єктів, які входять в конкретну систему. А зберігання об’єктів в певному стані, та подальше їх копіювання для отримання нових з тими ж властивостями дозволяє зручно отримати необхідні об’єкти в будь-який частині системи.

1. **Proxy (as Authenticate Proxy)**

*Структурний шаблон.* Забезпечує створення заступника об’єкта для контролю доступу до останнього через перехоплення всіх викликів. Надає об’єкт-заступник (surrogate) або об’єкт-замінник (placeholder). Обгортаючи доступ до реального компонента, «Заступник» зменшує складність роботи з ним.

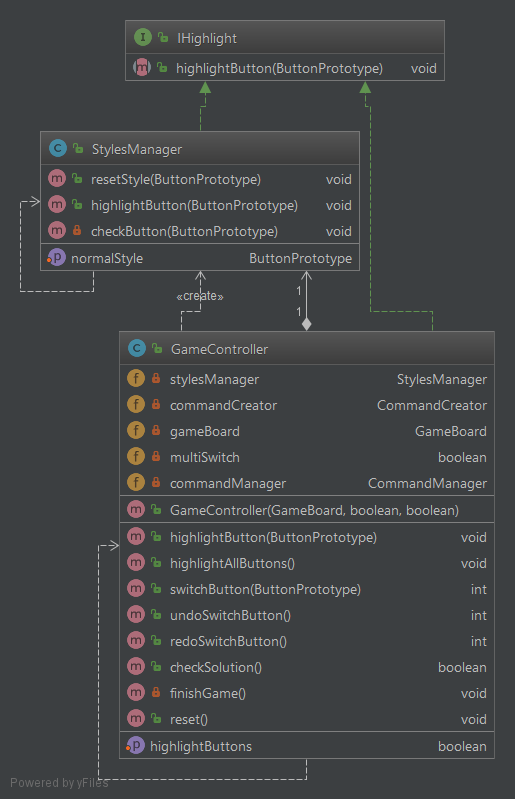


Рис. 2.1.5. UML-діаграма шаблону “**Proxy**”

*Структура*.Інтерфейс **IHiglight** виступає в ролі **«**Subject», класи **StylesManager** та **GameController**, що реалізують даний інтерфейс відповідно відіграють ролі «RealSubject» та «Proxy». **GameController** вміщає у собі об’єкт класу **StylesManager**. Коли клієнт хоче, щоб об’єкт класу **StylesManager**, виконав операцію, визначену інтерфейсом **IHiglight**,запит на виконання методу спочатку переадресовується до **GameController**. Він виконує додаткову перевірку умов доступу до об’єкту основного класу та в разі їх виконання – переадресовує запит до **StylesManager.**

*Обґрунтування використання даного шаблону*. Для того, щоб реалізувати підсвічення кнопки, яка стала в правильну позицію, необхідно перед її підсвіченням перевіряти позицію кнопки. Шаблон Заступник, а саме його варіант – Заступник – захисник, надає необхідну структуру для реалізації такого функціоналу. В такому випадку **StylesManager** не цікавиться за яких умов було підсвічено кнопку, а просто видозмінює її, а **GameController** виконує свою основнуфункцію – контролює підсвічення кнопок.

1. **Command**

*Поведінковий шаблон*. Перетворює операцію в об'єкт. Це дозволяє передавати операції як аргументи при викликах методів, ставити операції в чергу, логувати їх, а також підтримувати можливість скасування операцій.

*Структура*.Даний шаблон використовується, щоб інкапсулювати дію перемикання кнопок у формі об’єкту та додати можливість відміняти перемикання (в обидві сторони). Абстрактний клас **Command** виступає в ролі «Command», а класи нащадки **SingleSwitch** та **MultiSwitch** в ролі «ConcreteCommand». Кожна команда при створенні запам’ятовує посилання на ігрове поле, власне з яким і виконує операцію переміщення кнопок. В такому разі клас **GameBoard** виступає в ролі «Receiver» для даного шаблону. Клас **CommandManger** при вдалому виконанні команди, запам’ятовує її, щоб в майбутньому за необхідності використати її для відміни перемикання, тоді **CommandManger** виконує роль«Invoker».

*Обґрунтування використання даного шаблону*. Головне завдання даного шаблону в даному ПЗ – підтримання відміни операцій перемикання кнопок. А винесення операцій в окремі класи дає змогу інкапсулювати логіку перемикання в окремому об’єкті, та при необхідності швидко та безболісно змінювати її.

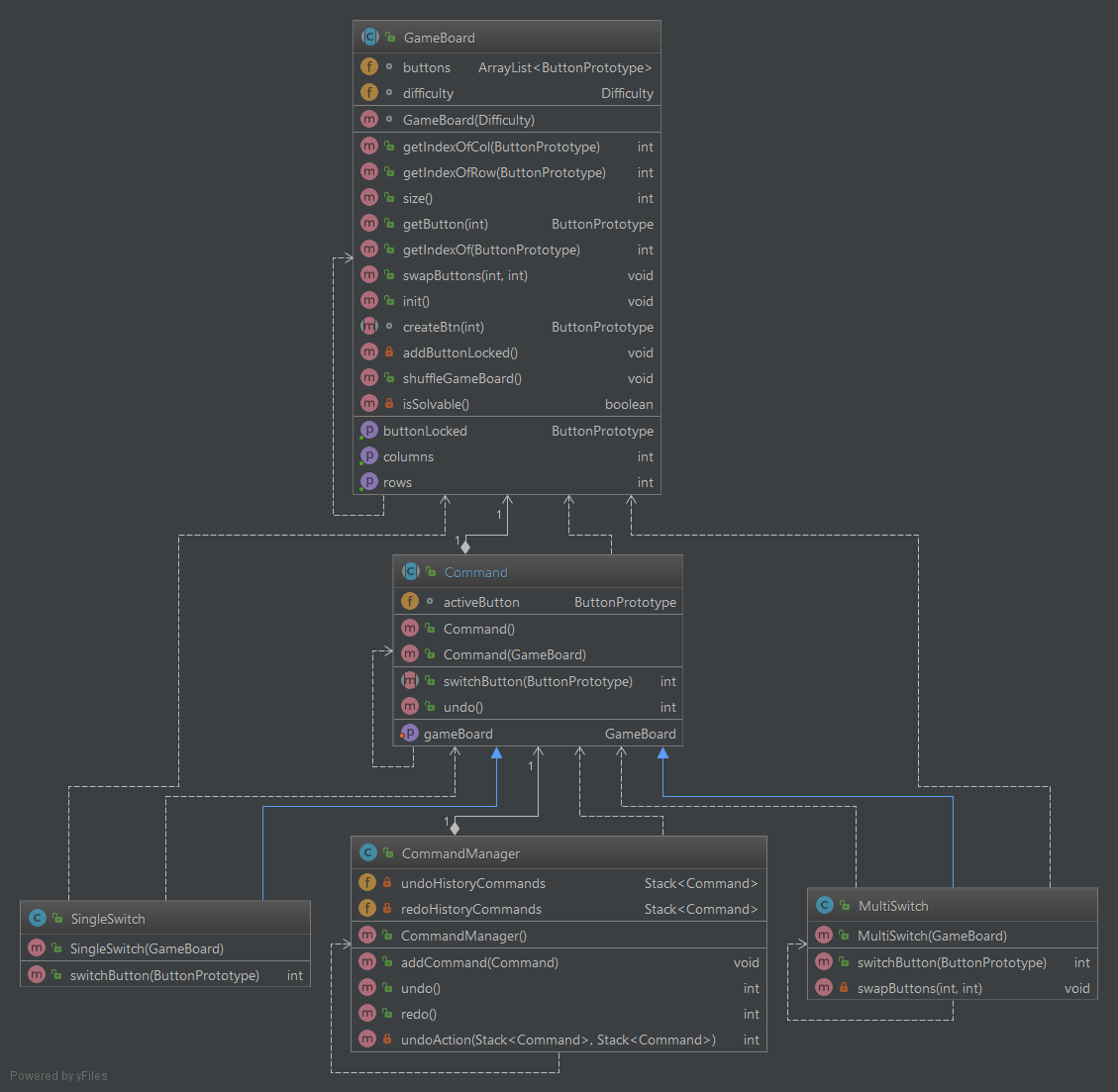


Рис. 2.1.6. UML-діаграма шаблону “**Command**”

1. **State**

*Поведінковий шаблон*. Дозволяє об'єкту варіювати свою поведінку залежно від внутрішнього стану (динамічна Strategy). Передбачає реалізацію поведінки, асоційованої з певним станом об'єкту, а також забезпечення зміни поведінки відповідно до зміни внутрішнього стану.

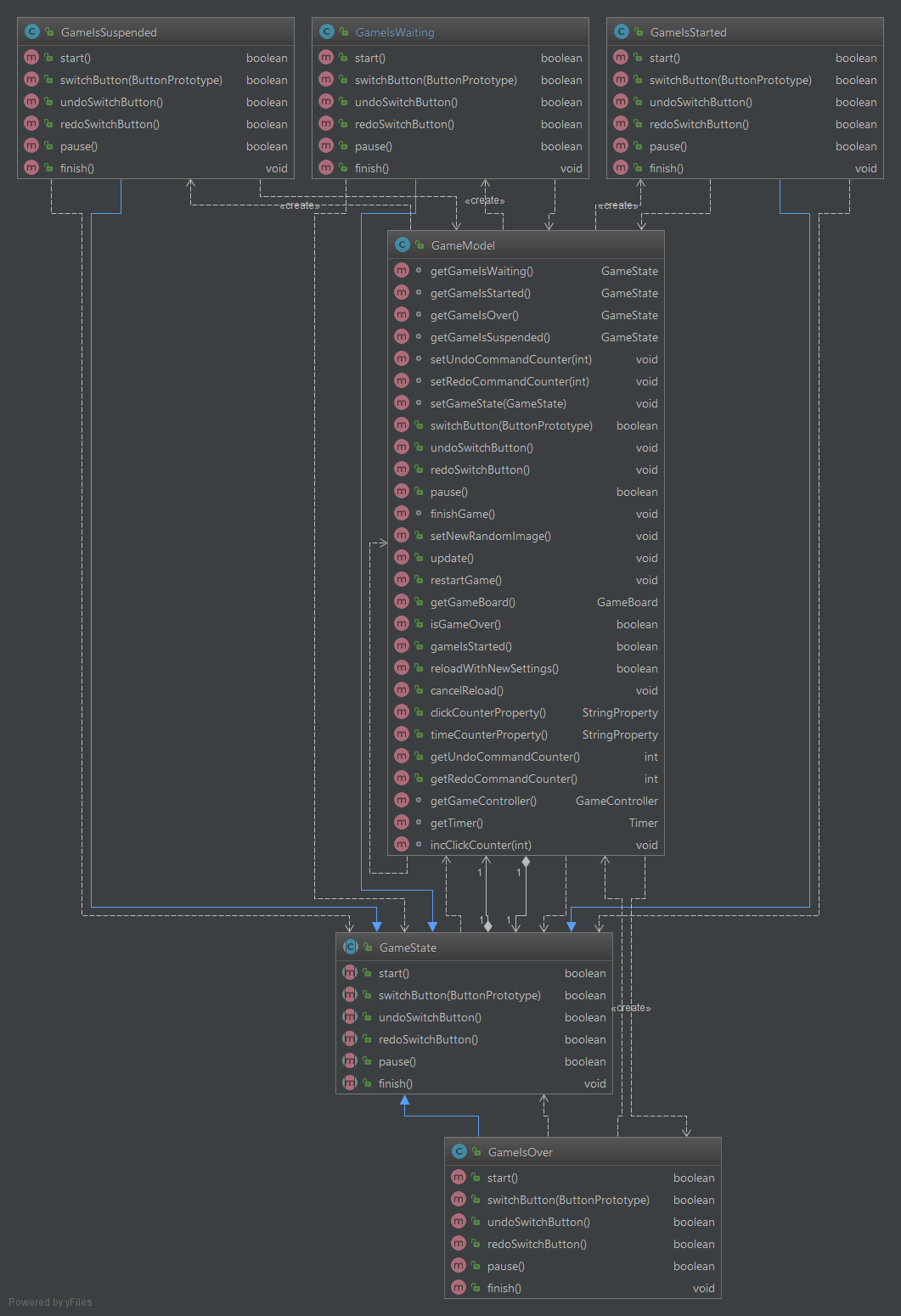


Рис. 2.1.8. UML-діаграма шаблону “**State**” для моделі гри



Рис. 2.1.7. UML-діаграма шаблону “**State**” для кнопки

*Структура*.Реалізований у двох варіантах:

* для класу **ButtonPrototype**: В даній реалізації **ButtonPrototype** виконує роль «Сontext»,абстрактний клас **ButtonState** – «State», а класи-нащадки **ActiveState** та **DisabledState –«ConcreteState»**. Оскільки наявно лише два стани кнопки, то відповідно при зміні стану – вони заміняють один одного. Кнопка в стані **ActiveState –** це кнопка, з якою може взаємодіяти гравець, в стані **DisabledState** – кнопка неактивна**.**
* для класу **GameModel**: В даній реалізації **GameModel** виконує роль «Сontext»,абстрактний клас **GameState** – «State», а класи-нащадки **GameIsWaiting, GameIsStarted, GameIsSuspend** та **GameIsOver – «ConcreteState»**. Вся логіка гри зосереджена в різних станах, які реалізують одні операції та забороняють інші. Переходи між станами відбуваються в залежності від дій користувача (див. Рис 2.1.9).

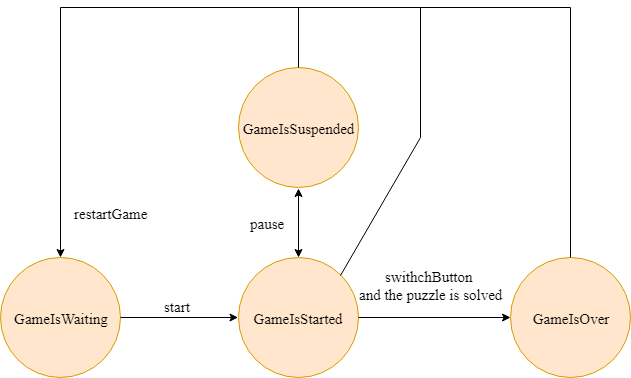


Рис. 2.1.9. Діаграма переходів GameModel в різні стани

*Обґрунтування використання даного шаблону*. Шаблон був використаний для спрощення виконання операцій в залежності від внутрішнього стану об’єкта, а поведінка моделі гри залежить від того, в якому стані вона знаходиться у даний момент. Він дозволяє позбутися багатьох умовних операторів в коді.

1. **Observer**

*Поведінковий шаблон*. Створює механізм підписки, за допомогою якого одні об'єкти можуть підписуватися на оновлення, що відбуваються в інших об'єктах.

*Структура*. Інтерфейс **IObserver** виступає в ролі «Observer», абстрактний клас **Subject** в ролі «Subject», класи-наслідники GameModel та SettingsModel в ролі «ConcreteSubject». Клас **MainController** і воодночас **GameModel** реалізують інтерфейс **IObserver** та виступають в ролі «ConcreteObserver».

**MainController** підписується на оновлення **GameModel**, а модель гри в свою чергу повідомляє спостерігачів, коли гра була закінчена, коли гра була перезавантажена чи відбулася подія відміни перемикання кнопок.

Клас **GameModel**, який виступає «Subject» для **MainController** водночас є«Observer» для **SettingsModel** та слідкує за змінами в моделі налаштувань і в разі чого – перезавантажує гру, та повідомляє **MainController** про зміни вGameModel.

*Обґрунтування використання даного шаблону*. При виконанні логіки, зв’язаної зі змінами налаштувань гри, відбуваються події, про які слід сповістити модель гри. Вона в свою чергу після перезавантаження з новими налаштуваннями повідомляє контролер ігрового поля програми, щоб той оновив інтерфейс. Даний шаблон дозволяє реалізувати таку можливість сповіщень.

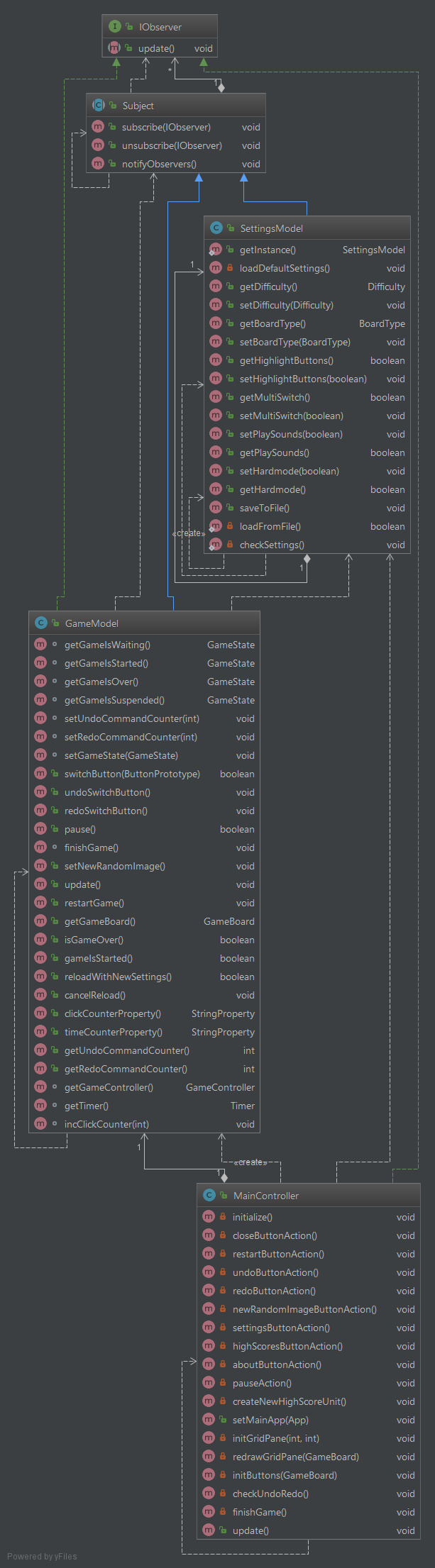


Рис. 2.1.10. UML-діаграма шаблону “ **Observer** ”

1. **Facade**

*Структурний шаблон*. Надає єдиний простий інтерфейс до безлічі інших інтерфейсів в деякій складній підсистемі.

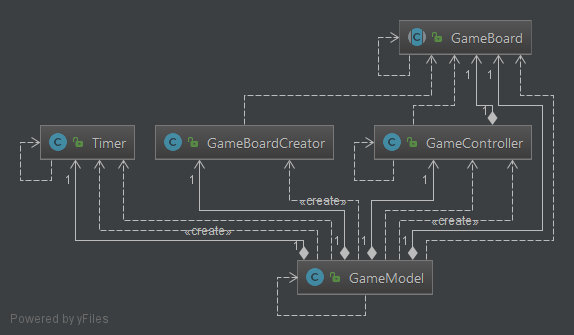


Рис. 2.1.11. UML-діаграма шаблону “ **Facade**”

*Структура*.В ролі «Facade» виступає клас **GameModel**. Він включає в себе об’єкти інших класів, які незалежно один від одного не мають можливості реалізувати певний функціонал системи. **GameModel** комутує об’єкти класів підсистем таким чином, щоб сформувати єдиний, простий інтерфейс, що представляє собою логіку гри та дозволяє керувати нею через цей інтерфейс.

*Обґрунтування використання даного шаблону*. Реалізований для того, щоб мати один об’єкт, через який можна безпечно та зручно взаємодіяти зі всією системою.

* 1. **Діаграма класів**

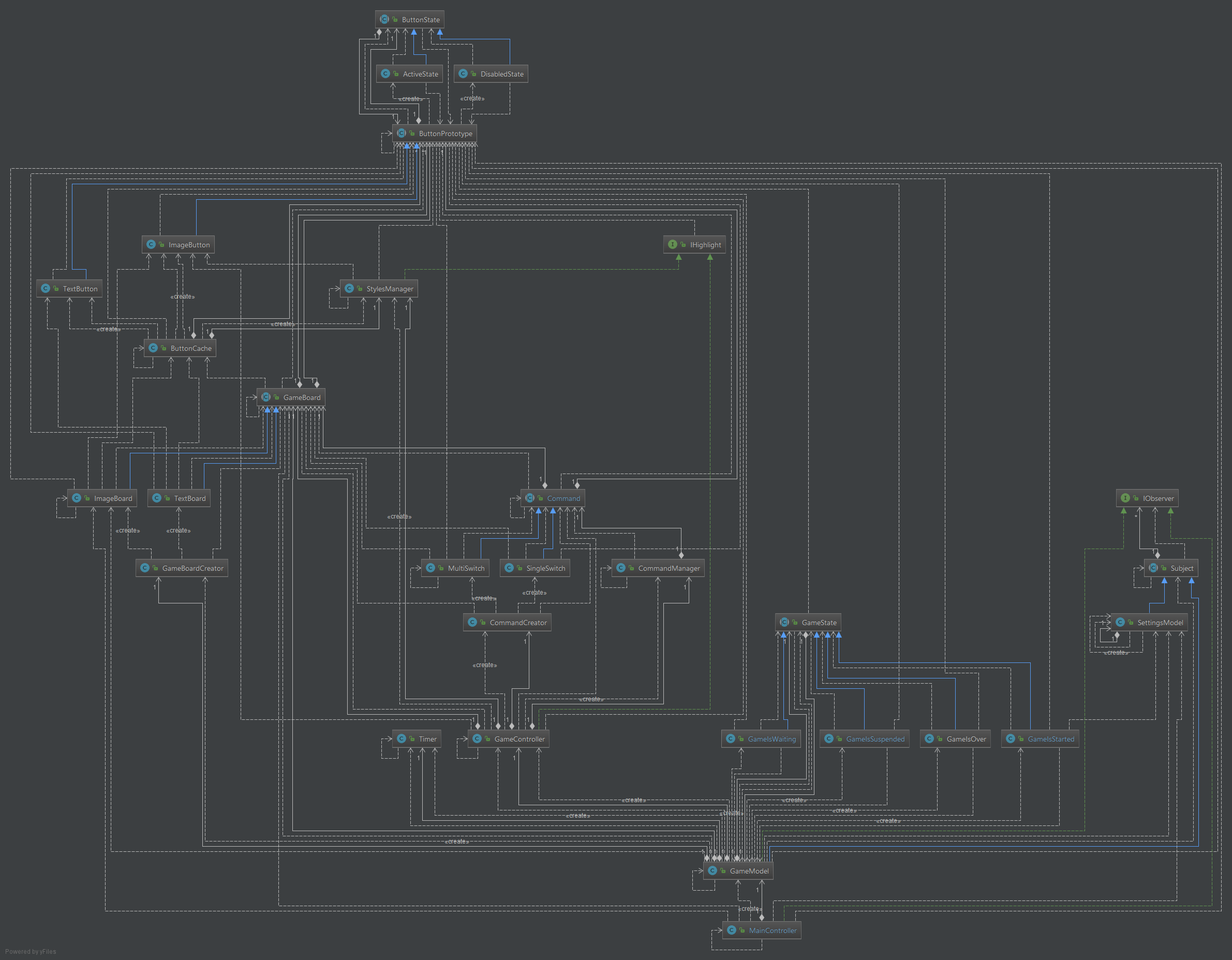


Рис. 2.2.1. UML-діаграма всіх класів

* 1. **Опис результату роботи програми**

При запуску гри відразу відображається ігрове поле, та кнопки керування. Гра знаходиться в режимі очікування.

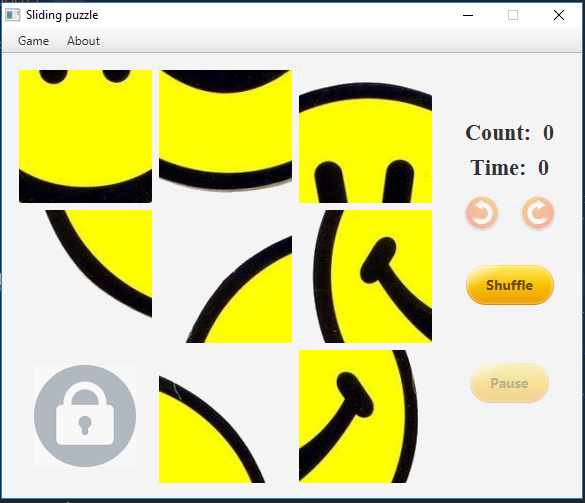


Рис. 2.3.1. Головне вікно гри

Як тільки гравець почне взаємодіяти з ігровим полем, запуститься відлік часу та підрахунок кількості переходів.

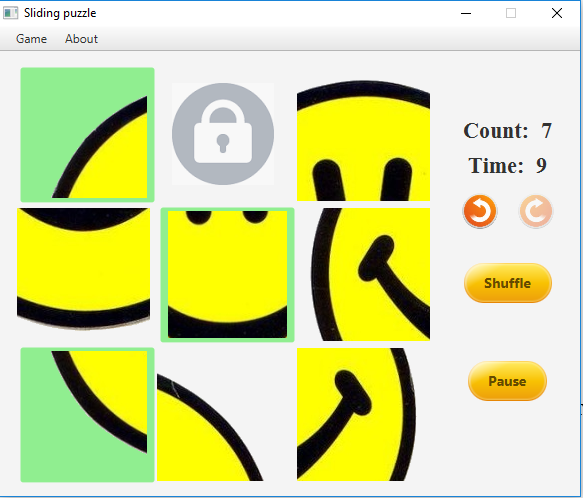


Рис. 2.3.2. Процес гри

Якщо гра в процесі, то гравець може в будь-який момент призупинити її. В такому випадку ігрове поле буде сховане, відлік часу також зупиниться. Гра ставиться на паузу також, коли гравець починає взаємодіяти з іншими вікнами програми.

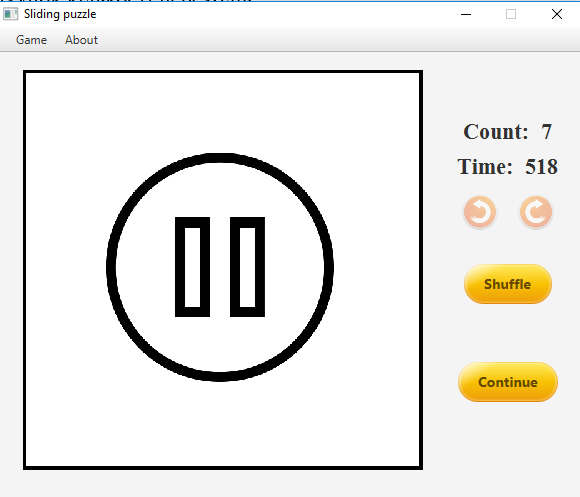


Рис. 2.3.3. Гра призупинена

У випадку, коли гравець успішно закінчує гру з’являється діалог, якій пропонує ввести ім’я гравця, та зберегти результат до таблиці рекордів.

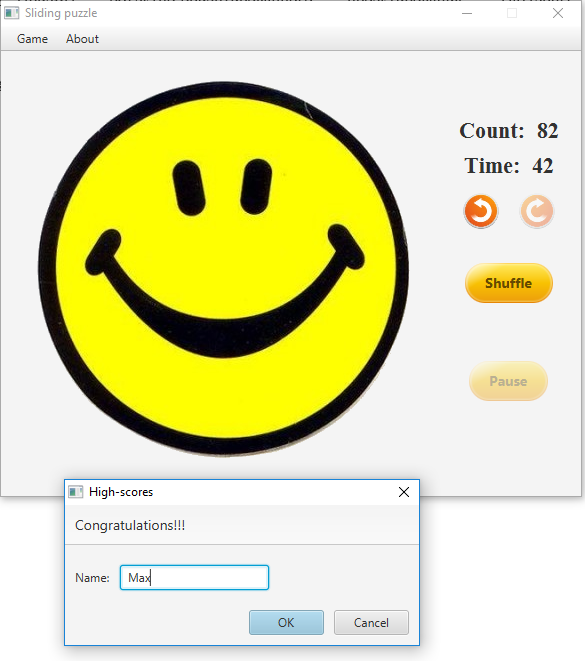


Рис. 2.3.4. Головоломка вирішена

У меню гри гравець може відкрити вікно налаштувань, талицю рекордів, оновити картинку (якщо ігрове поле – картинка) та закрити гру.

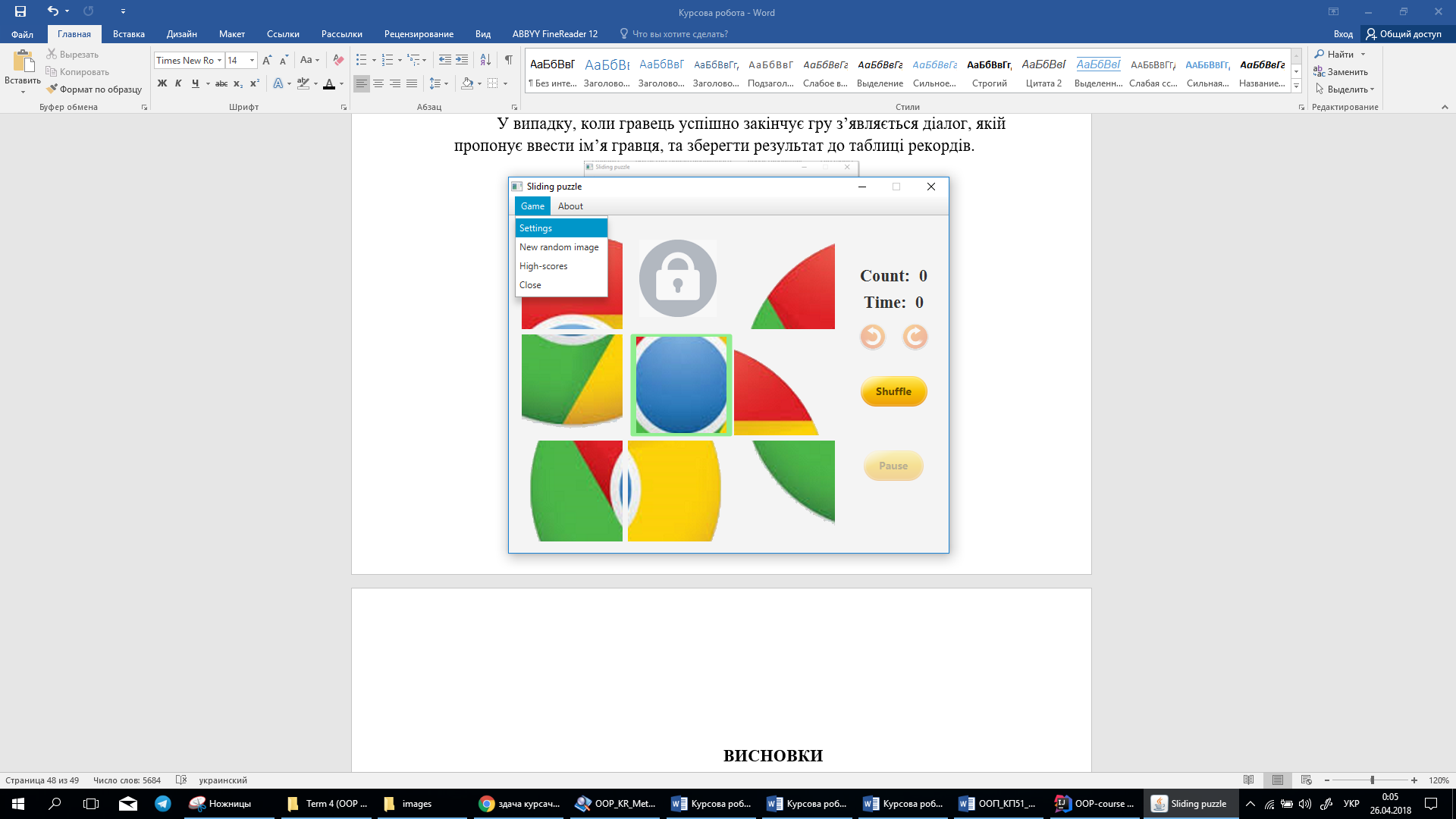


Рис. 2.3.5. Меню гри

У вікні налаштувань гравець може змінити налаштування гри. Якщо він після змін натисне кнопку «ОК» - поле гри буде оновлене.

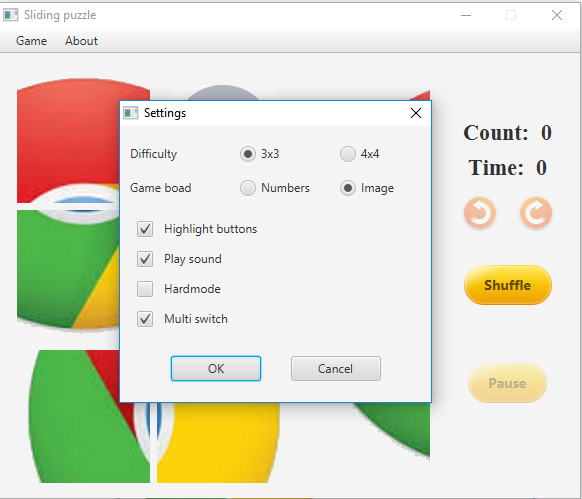


Рис. 2.3.6. Вікно налаштувань гри

У будь-який момент часу гравець може переглянути таблицю рекордів та сортувати її за 4 параметрами. Також гравець може скинути всі результати.

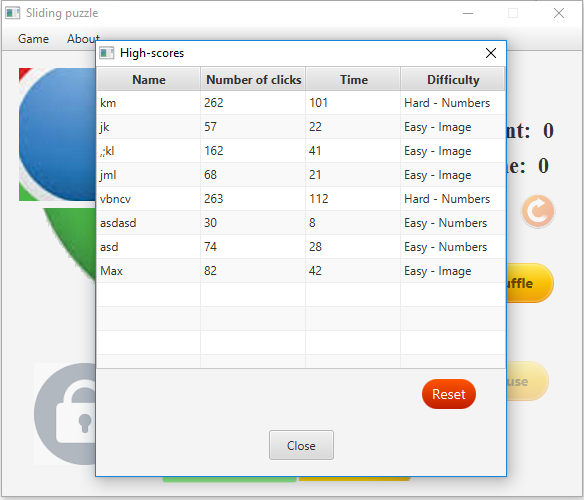


Рис. 2.3.7. Таблиця результатів

Приклад гри в режимі 4х4 та ігровим полем з числами.

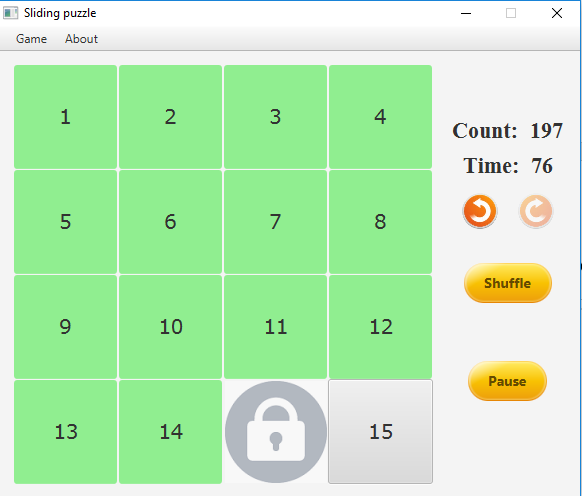


Рис. 2.3.8. Таблиця результатів

**ВИСНОВКИ**

Метою даної курсової роботи було розроблення програмного забезпечення гри “П’ятнашки” з використанням шаблонів проектування. Підставою для розроблення стало завдання на виконання курсової роботи з дисципліни «Об’єктно-орієнтоване програмування» студентами ІІ курсу кафедри ПЗКС НТУУ «КПІ ім. І.Сікорського».

Для досягнення поставленої мети у повному обсязі виконано завдання, визначені у аркуші завдання на курсову роботу; розроблено графічні матеріали; реалізовано всі вимоги до програмного продукту, програмного та апаратного забезпечення, наведені у технічному завданні; створено відповідну документацію.

Розроблене програмне забезпечення дозволяє використовувати ігровий додаток з метою приємного проведення вільного часу, розвитку моторики очей та логічного складу розуму гравця.

Програму створено на основі використання шаблонів проектування: зокрема до структури розробленого програмного забезпечення входить реалізація восьми шаблонів, які належать до різних груп шаблонів проектування (структурних, породжувальних та поведінкових).

Для розроблення програмного забезпечення використані мова програмування Java SE 8.

Перспективним напрямком подальшого дослідження даної тематики є портування гри на мобільні пристрої.

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**

1. Язык шаблонов. Города. Здания. Строительство. / Кристофер Вольфганг Александер. – 1977. – 1096 с.
2. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. – 1994. – 395 с.
3. Руководство Microsoft по проектированию архитектуры приложений. / С. Сомасегар, Скотт Гатри, Девид Хилл. – 2009. – 529 с.
4. Wikipedia електронний ресурс - Режим доступу:

[https://ru.wikipedia.org/wiki/Игра\_в\_15](https://ru.wikipedia.org/wiki/%D0%98%D0%B3%D1%80%D0%B0_%D0%B2_15) (дата останнього звернення 25.04.2018)

1. sitepoint електронний ресурс - Режим доступу: <https://www.sitepoint.com/randomizing-sliding-puzzle-tiles/> (дата останнього звернення 25.04.2018)