**Дніпровський національний університет імені ОЛеся Гончара**

**Факультет прикладної математики**

**Кафедра математичного забезпечення ЕОМ**

**КУРСОВА РОБОТА**

з дисципліни «Об’єктно-орієнтоване програмування»

на тему: «Об’єктно-орієнтований дизайн. Рефакторинг коду»

Студента 2 курсу групи \_\_\_\_ПЗ-21-2\_\_\_\_

Спеціальності 121 Інженерія програмного забезпечення

Міхайличенка Н.А.

(прізвище та ініціали)

Керівник Полонська А.Є.

Кількість балів \_\_\_\_\_\_\_\_\_\_\_\_\_

Національна шкала \_\_\_\_\_\_\_\_\_\_\_\_\_

Члени комісії : \_\_\_\_\_\_\_\_\_\_\_ Антоненко С.В.

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_ Полонська А.Є.

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_ Божуха Л.М.

(підпис) (прізвище та ініціали)

м. Дніпро, 2023 р.

ЗМІСТ

[1 ПОСТАНОВКА ЗАДАЧІ 3](#_Toc130315773)

[2 ТЕОРЕТИЧНА ЧАСТИНА 4](#_Toc130315774)

[2.1 Патерн Компонувальник (Composite) 4](#_Toc130315775)

[2.2 Патерн Прототип (Prototype) 6](#_Toc130315776)

[2.3 Патерн Одинак (Singleton) 7](#_Toc130315777)

[2.4 Патерн Знімок (Memento) 8](#_Toc130315778)

[3 ПРАКТИЧНА РЕАЛІЗАЦІЯ 10](#_Toc130315779)

[3.1 Структурна схема програми 10](#_Toc130315780)

[3.2 Опис інтерфейсу програми 14](#_Toc130315781)

[3.3 Реалізовані патерни та рефакторинг коду 15](#_Toc130315782)

[4 ТЕСТУВАННЯ 23](#_Toc130315783)

[5 СПИСОК ЛІТЕРАТУРИ 26](#_Toc130315784)

# 1 ПОСТАНОВКА ЗАДАЧІ

Провести рефакторинг лабораторної роботи 4 (геометричні фігури) з використанням патернів проектування Композит (Composite), Прототип (Prototype), Одинак (Singleton) і Знімок (Memento), таким чином, щоб:

1) агрегат представляв собою патерн Композит (Composite);

2) для створення фігур та агрегатів використовувався патерн Прототип (Prototype), передбачити реєстрацією доступних прототипів у сховищі;

3) патерн Одинак (Singleton) забезпечував існування лише одного контролера сцени;

4) патерн Знімок (Memento) використовувався для зберігання стану наявних фігур на сцені у файл на диску та відновлення сцени з файлу;

5) додавання інших патернів є додатковою перевагою.

Програма повинна бути розроблена згідно вимогам об’єктно-орієнтованого дизайну та угодам з написання коду. Важливо, щоб програма не містила «магічних» констант, коду, що дублюється, витоків пам'яті (тобто кожному new повинно відповідати свій delete у потрібній формі)

# 2 ТЕОРЕТИЧНА ЧАСТИНА

У даному розділі будуть описані всі вищенаведені патерни проектування, їх структурна реалізація у виконаній роботі.

## 2.1 Патерн Компонувальник (Composite)

Компонувальник – це структурний шаблон проектування, який об’єднує групи об’єктів в одну деревовидну структуру і дозволяє однаковим чином взаємодіяти як з групами, так і з окремими об’єктами. Це досягається за допомогою реалізації єдиного інтерфейса-прикладу всіма об’єктами «дерева», незалежно від того складові вони, чи прості. Інтерфейс забезпечує існування деяких базових методів/змінних, тобто це своєрідна гарантія підтримки всіма об’єктами потрібного списку функцій.

Як приклад можна навести процес розпакування складної посилки, яка складається із багатьох менших, упакованих в однакові обгортки. Як людина яка її розпаковує(клієнт) ми ніколи заздалегідь не знаємо, чи має конкретна частина посилки у собі один об’єкт, а чи має дві менші частини. Але ми завжди можемо розпакувати частину, незалежно від рівня вкладеності та її складності, про яку ми дізнаємося пізніше. Тобто як людина, яка це робить, ми маємо метод «розпакувати» і застосовуємо його по черзі до всіх об’єктів. В свою чергу людина є «контролером», яка може використовувати функції методи реалізованого інтерфейсу на об’єктах.

На рисунку нижче (рис. 2.1) можна побачити патерн Компонувальник зображений у вигляді UML діаграми [1]:

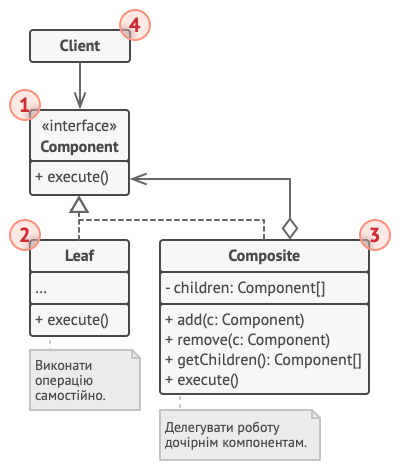


Рисунок 2.1 - UML Composite diagram

Бачимо, що структура складається з таких компонентів:

1. Component – це інтерфейс з потрібними нам методами/змінними (у даному випадку execute()), від якого наслідуються прості і складні об’єкти.
2. Leaf – це лист, тобто простий об’єкт який не має відгалуджень та має реалізовану функцію execute().
3. Composite – складний об’єкт. Окрім загальної функції execute(), він має інші методи, які використовуються при створенні або видалені об’єкта, для розміщення в ньому інших об’єктів. При цьому він не знає типи тих об’єктів які зберігає. Це можуть бути як «Листя», так і інші контейнери Композити.
4. Client – клієнт який працює з деревом через загальний інтерфейс компонентів.

Таким чином, якщо маніпулювати більш складними компонентами, програма буде рекурсивно виконувати дії для кожної складової контейнеру.

## 2.2 Патерн Прототип (Prototype)

Прототип - дозволяє створювати нові екземпляри класів через копіювання уже існуючих об’єктів. По суті цей патерн пропонує техніку клонування об’єктів. Але ключовий момент цього патерна в тому, що клієнтський код може створювати нові екземпляри, не знаючи об’єкт конкретно якого класу створюється.

Переваги використання патерну Прототип [2, c. 643]:

* складність свторення нових екземплярів залишаються прихованими і невідомими для клієнту;
* у клієнта з’являється можливість генерувати об’єкти, тип котрих йому невідомий;
* при деяких обставинах копіювання об’єкту ефективніше та менш витратно по часу ніж створення нового об’єкту.

Зазвичай Прототип використовується у системах, які створюють нові об’єкти різних типів у складній ієрархії класів.

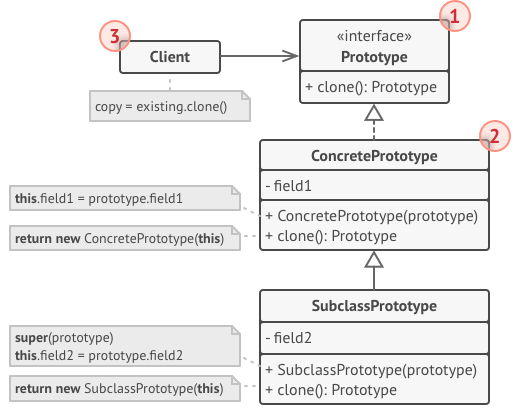


Рисунок 2.2 – UML Prototype diagram

Складові частини патерну [3] у загальному розумінні можна розділити на такі частини (рис. 2.2):

1. Prototype – інтерфейс, який описує операцію клонування, зазвичай це метод clone().
2. ConcretePrototype – клас, в якому реалізовані методи Прототипу, в даному випадку clone(). Оскільки метод реалізований в класі, то від користувача прихований весь процес створення нового об’єкту, включаючи різноманітні складнощі, про які користувачу не треба знати.
3. Client – користувач, який може створювати копії, звертаючись до реалізованих методів інтерфейсу Прототип.

## 2.3 Патерн Одинак (Singleton)

Одинак - це породжувальний патерн проектування, який надає гарантію, що має лише один екземпляр, та глобальну точку доступу до нього. Це потрібно для випадків створення контролеру, або як приклад обробнику подій, який повинен бути єдиний у програмі. Це досягається за допомогою делегації створення класу – створення приватного конструктору(обмеження доступу ззовні) та статичної функції GetInstance(), яка створює об’єкт якщо такого ще нема та повертає існуючий в іншому випадку.

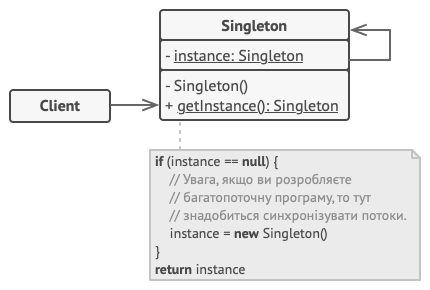


Рисунок 2.3 – UML Singleton diagram

Клас в якому реалізований шаблон Singleton (рис. 2.3) має статичну змінну instance, яка зберігає в собі єдиний екземпляр класу [4]. Якщо instance – уже ініціалізоване поле, програма повертатиме посилання саме на цей об’єкт. В іншому випадку створюватиметься новий екземпляр класу Singleton та призначатиметься instance, отже ініціалізація об’єкта-одинака буде відкладено до першого використання. Користуючись цим зовні, в інших класах, клієнти матимуть глобальну точку доступу до instance. Утім, такий патерн має проблеми багатопоточності та порушує принцип єдиного обов’язку класу.

## 2.4 Патерн Знімок (Memento)

Знімок – шаблон проектування, який забезпечує зберігання поточного стану задля можливості його подальшого відновлення. Зазвичай він використовується для опції відміни попередньої дії. Завдяки його структурі користувач має змогу зчитувати дані об’єктів, не втручаючись до їх реалізації.

Переваги використання патерну Знімок [2, с. 640]:

* зберігання стану окремо від ключового об’єкта покращує зв’язність системи;
* інкапсуляція даних ключового об’єкту;
* легкий в реалізації повернення попереднього стану.

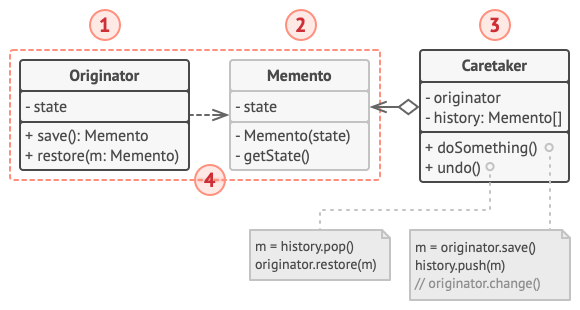


Рисунок 2.4 – UML Memento diagram

Для точнішого розуміння принципу роботи розглянемо діаграму (рис. 2.4), де наведено узагальнену схему реалізації [5]:

1. Originator – творець, який реалізовує поміж своїх, два методи для шаблону Memento – GetMemento() та SetMemento(). Задача GetMemento() перемістити необхідні дані з об’єкту до класу-знімку Memento, після чого Memento буде переданий далі, для обробки і збереження. В той час як SetMemento() робить зворотню дію, тобто встановлює завантажені дані у об’єкт-творець.
2. Memento – клас, який має тільки поля з даними для передачі в обидві сторони. Варто відмітити, що для різних мов програмування буде різна реалізація, оскільки принцип інкапсуляції описується різним чином. Ціль – це зробити дані класу Memento приватними для сторонніх користувачів, але відкрити для класів Originator та Caretaker. Як приклад, у мові С++ треба об’явити два вищенаведених класи як дружні до Memento. У той час як у Java можна вкласти Memento в Originator.
3. Caretaker – клас, який відповідальний за безпосередній запис або читання даних на диск чи інші пристрої. При записі він приймає об’єкт Memento на вхід та зберігає його стан у файл. При читанні, передає Memento в Originator.

Варто відмітити, що весь ланцюг збереження або зчитування запускається саме методами Caretaker.

# 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

## 3.1 Структурна схема програми

Рисунок 3.1 – Повна схема програми

На рисунку вище (рис. 3.1) можна побачити всю схему програми, яка складається з трьох головних модулів та трьох допоміжних класів.

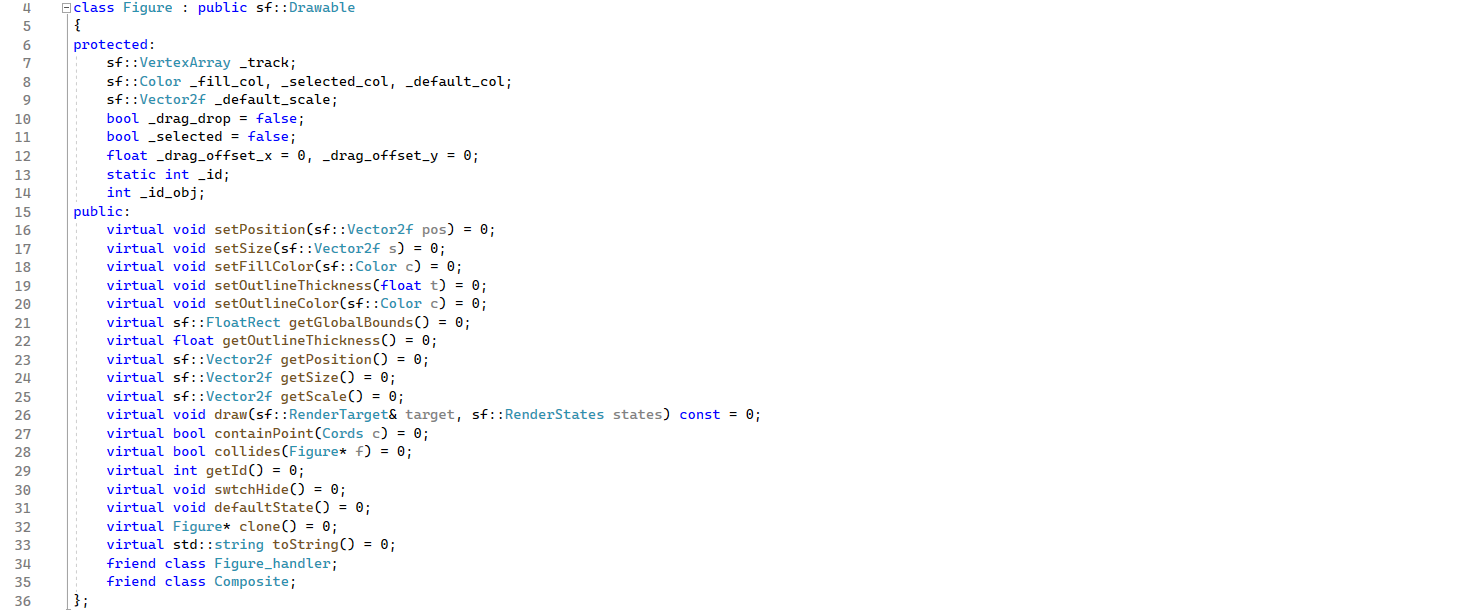


Рисунок 3.2 – Опис класу Figure

Перший з модулів – Figure\_handler та всі інші класи які пов’язані з Figure або взаємодією з фігурами. В основі всіх фігур лежить абстрактний клас Figure (рис. 3.2), який описує всі головні методи, які будуть використовуватись у контролері, а також основні приватні змінні. Було прийнято рішення описати всі потрібні змінні в класі Figure, задля слідуванню принципу ООП DRY.

Від класу Figure наслідуються 4 прості фігури: Rectangle, Circle, Triangle, Star та складений об’єкт-контейнер Composite. Після створення цих фігур в ході виконання програми, вони зберігаються у приватному полі, списку класу Figure\_handler, де і обробляються функцією update().

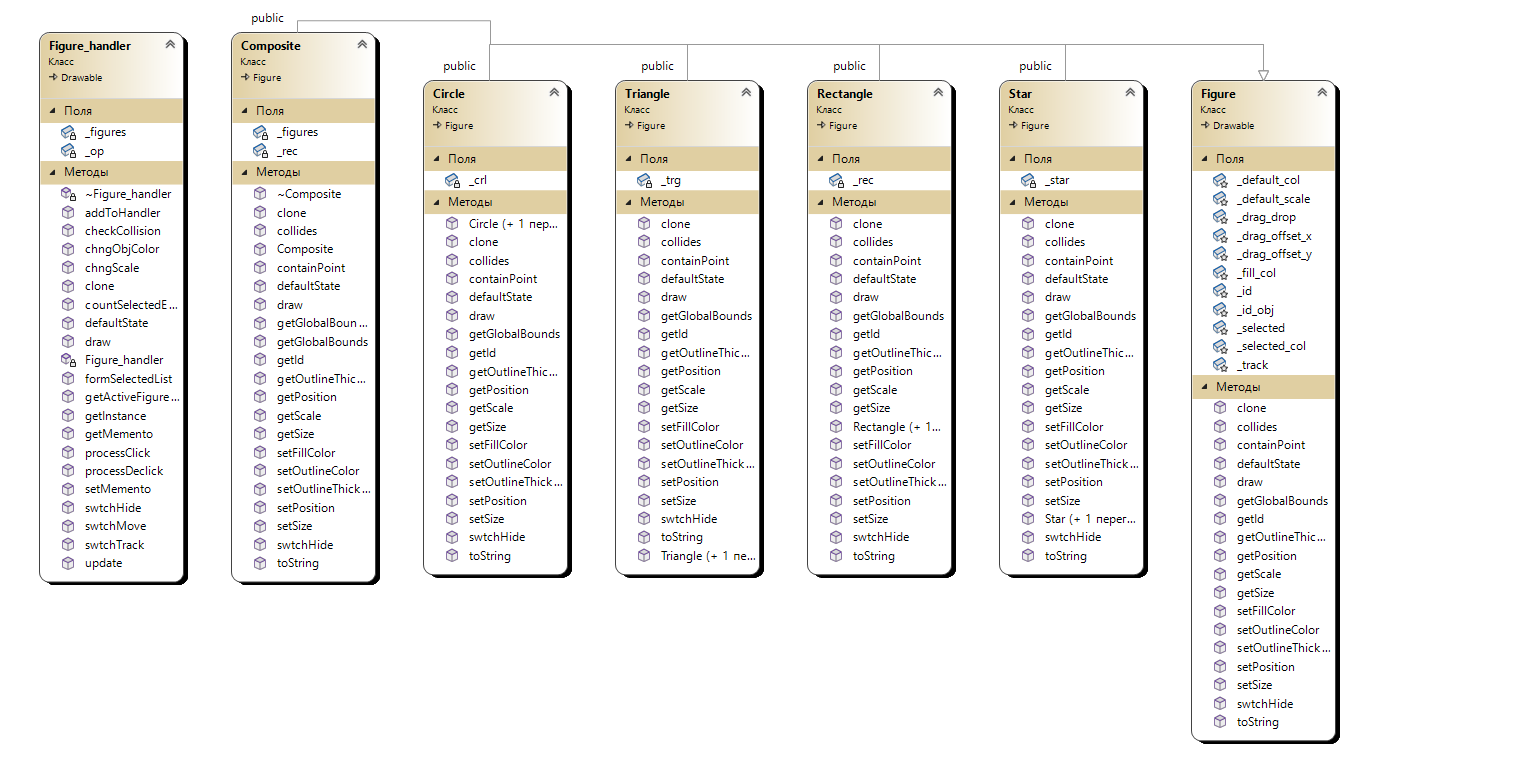


Рисунок 3.3 – Схема фігурного модуля

Як видно зі схеми (рис. 3.3) клас Composite має більше методів, за прості об’єкти, які відповідають в основному за створення композиту та видалення (очищення пам’яті, оскільки використовується динамічне виділення пам’яті). Також кожен з простих класів має приватну змінну, яка є представленням фігури згідно графічної бібліотеки SFML.

Друга частина програми присвячена візуальному інтерфейсу користувача. Задля зручного керування додатком було розроблено графічний інтерфейс користувача (GUI).

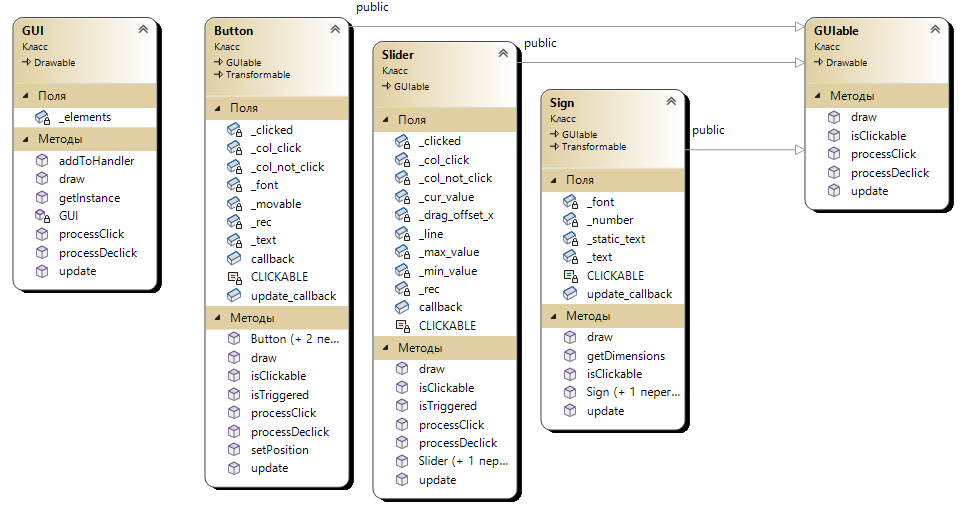


Рисунок 3.4 – Схема модуля GUI

Структура модуля GUI (рис. 3.4) дуже схожа на модуль Figure (рис. 3.3). Він також має абстрактний клас «GUIable», який описує 5 головних методів, які використовують елементи інтерфейсу. Із елементів реалізовані кнопка, оновлюваний напис та аналоговий повзунок. Напис має статичну (незмінну) частину та покажчик на функцію, яка повина повертати змінне значення.

Всі графічні елементи GUIable створюються користувачем у функції main та зберігаються у списку контролера. Кожну ітерацію головного циклу стан об’єктів оновлюється та перевіряється натиснення клавіші миші, та у випадку дії виконується цикл дій processClick/processDeclick у контролері.

Також для коректної обробки елементів кожний з них має константне поле CLICKABLE та метод, який повертає це значення. Таким чином програма може відрізнити кнопку та слайдер від напису.

Третя частина програми розроблена по особистому бажанню, для придання роботі презентабельний вид і новорічний настрій. Це оновлюваний фон з зображенням снігопаду. Будується на колекції графічних SFML елементів.

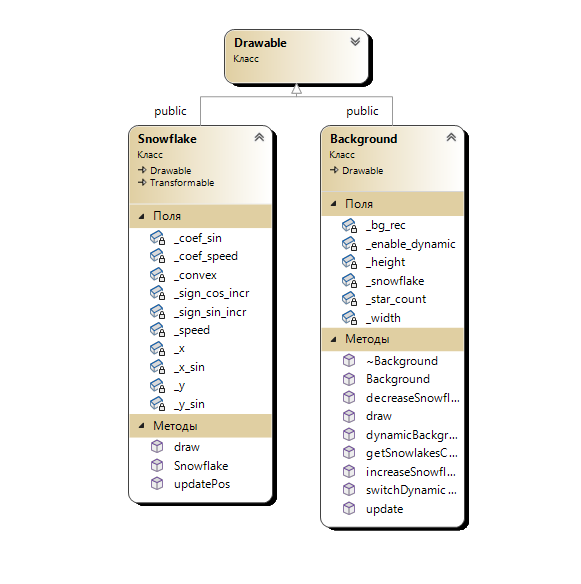


Рисунок 3.5 – динамічний фон

Є клас контейнер Background (рис. 3.5), який має колекцію об’єктів Snowflake (тобто сніжинок). У класі Snowflake описана поведінка сніжинки, тобто зміна координат з часом. У той час як клас контейнер відповідає за керування колекцією – збільшення/зменшення кількості об’єктів та увімкнення/вимкнення динамічного фону.

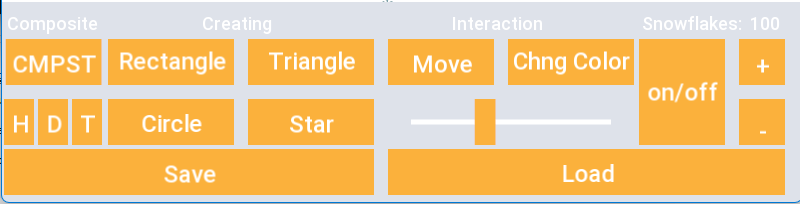


Рисунок 3.6 – Панель управління програмою

Управляти фоном можна за допомогою правої частини панелі користувацього інтерфейсу (рис. 3.6), де для цього призначені три кнопки: «+», «-» та «on/off».

## 3.2 Опис інтерфейсу програми

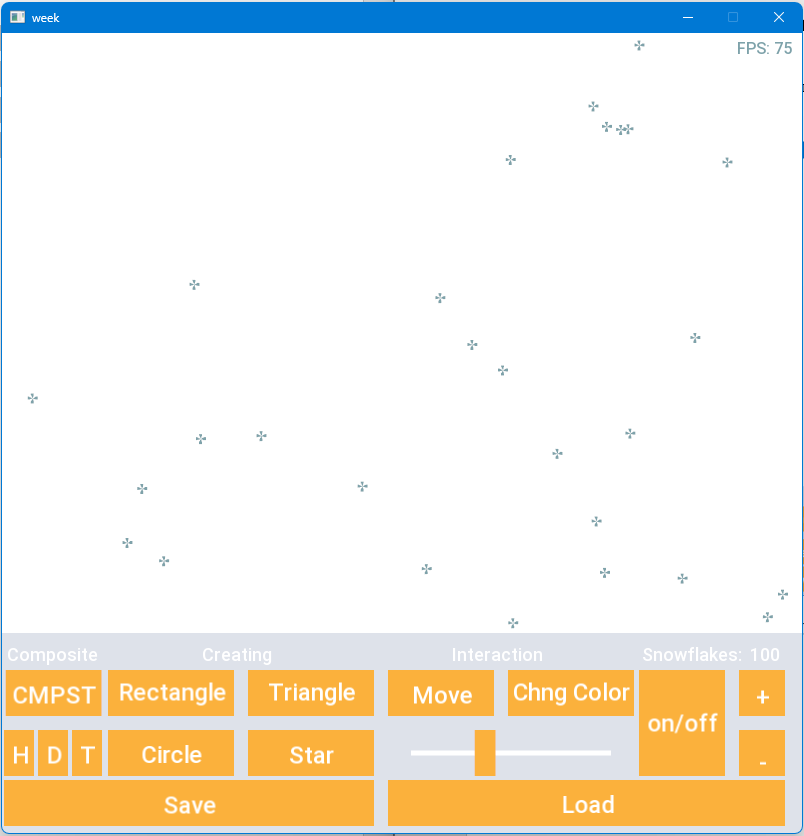


Рисунок 3.7 – Інтерфейс програми

Все керування програмою здійснюється через нижню панель з кнопками та слайдером, а також через ЛКМ та ПКМ. Навівши курсор на фігуру та затиснувши ЛКМ фігуру можна переміщувати по екрану. У той час натиснувши ПКМ по фігурі можна виділити її (візуально з’являється кольорова рамка навколо фігури) та робити різні дії з нею за допомогою панелі керування.

Панель можна поділити всього на 6 частин (рис. 3.7):

1. Перша та крайня зліва частина – це Composite. Відповідно на кнопку «CMPST» створюється композит з виділених у даний момент фігур.
2. Під кнопкою створення композиту є 3 клавіші. З них:
   1. H – hide, приховує фігури змінюючи свій колір на колір фону.
   2. D – default, яка повертає виділені фігури в початковий стан.
   3. T – track, перемикач відображення сліду руху.
3. Розділ «Creating» містить в собі 4 кнопки, які відповідають за створення простих фігур.
4. Розділ «Interaction» відповідає за взаємодію з фігурами. Дві кнопки «Move» та «Chng Color» відповідають за увімкнення авторуху та зміну кольору відповідно, а слайдер під ними дозволяє міняти розмір об’єктів.
5. Розділ «Snowflakes», як зазначалось раніше відповідає за взаємодію з динамічним фоном, має 3 кнопки з інтуїтивно зрозумілими функціями, а також динамічно оновлюваний напис який показує кількість сніжинок які містяться в колекції у даний момент часу.
6. Останній розділ розташований внизу, під усіма іншими – це дві кнопки «Save» та «Load», які відповідають за збереження/відновлення стану сцени використовуючи патерн проектування Memento.

Також на екрані у правому верхньому кутку розташований динамічний напис «FPS», який показує кількість оновлень сцени за секунду для оцінки швидкості роботи програми.

## 3.3 Реалізовані патерни та рефакторинг коду

У програмі було реалізовано 4 патерни проектування: «Компонувальник», «Прототип», «Одинак» та знімок. Надалі роздивимось реалізацію кожного з них, а також проведений рефакторинг коду.

1) Отже, почнемо з патерна «Компонувальник». Для його реалізації був створений клас «Composite» в якому перш за все реалізовано абстрактний клас Figure.

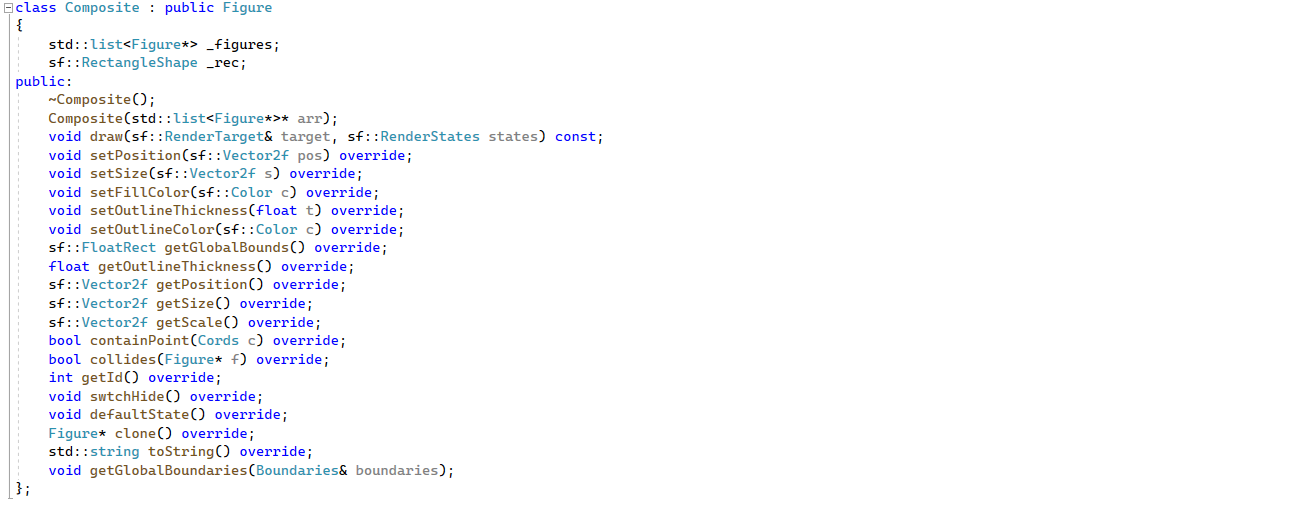


Рисунок 3.8 – клас Composite

Composite має 2 поля (рис. 3.8).

Перше – це список фігур, які містяться у контейнері. Варто зазначити, що список складається з покажчиків на Figure. Тобто всі фігури на які вказують покажчики містяться у динамічній пам’яті. По цій причині у класі був описаний деструктор, де ця пам’ять звільнюється.

Друге поле - це віртуальний прямокутник, місія якого тримати розміри та координати композиту в одному об’єкті. Розмір розраховується методом getGlobalBoundaries за крайніми положеннями фігур на сцені.

У свою чергу координати композиту – це положення крайнього лівого елементу та крайнього верхнього елементу. При створенні композиту, або встановленні нової позиції спершу встановлюються координати віртуального прямокутнику, а фігури які містяться у списку мають відносні координати. Тобто координати фігур на екрані вираховуються додаючи відносні координати та положення віртуального прямокутника.

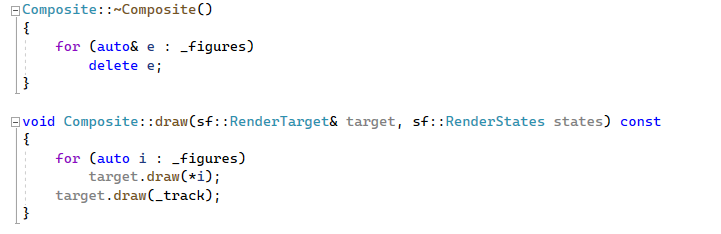


Рисунок 3.9 – Приклад методів класу «Composite»

Щодо реалізації методів класу (рис. 3.9) вони просто делегують дії до більш простих об’єктів, перебираючи кожний з них у циклі. Використання композиту у Figure\_handler (контролері) не відрізняється від використання простих фігур, оскільки походе від того ж класу Figure та має всі потрібні методи.

2) Патерн «Прототип».

Для реалізації цього шаблону в клас Figure (рис. 3.2) було додано метод Figure\* clone(), який буде виконуватись на об’єкті, клон якого нам треба зробити.

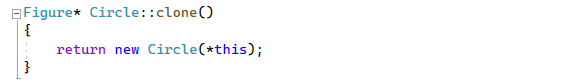


Рисунок 3.10 – Метод клонування у класі Circle

У простих об’єктах метод запускає конструктор копіювання, передаючи екземпляр самого себе (рис. 3.10). Цим можна користуватись, адже у простих фігурах немає покажчиків на поля з динамічним виділенням пам’яті. У той час як у Composite створюється спочатку копія контйнеру з об’єктами, а після цього новий контейнер передається у конструктор Composite і повертається новий об’єкт (рис. 3.11).

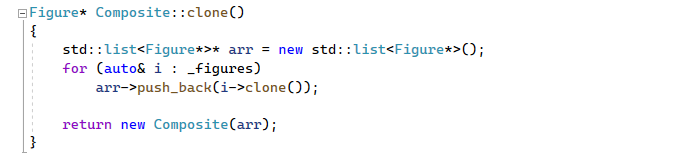


Рисунок 3.11 – Метод clone() у класі Composite

3) Патерн «Одинак».

У структурі програми наявні 2 контролери об’єктів, кожний з яких відповідає за різні типи. Тому було прийняте рішення про реалізацію патерну «Одинак» для двух класів, а саме для «Figure\_handler» та «GUI».

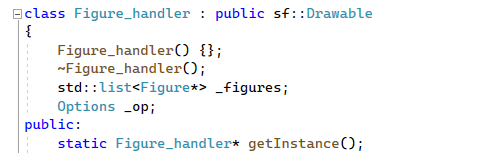


Рисунок 3.12 – «Одинак» у класі Figure\_handler

Зміни, які були додані до оригінальної версії класу: параметр доступу конструктору змінено з public на private та додано відповідно статичний метод getInstance(), який повертає екземпляр класу (рис. 3.12).

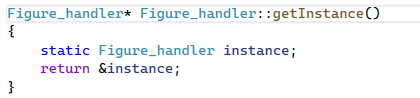


Рисунок 3.13 – Метод getInstance у класі Figure\_handler

Як видно з рисунку 3.13, статична змінна, в якій лежить об’єкт класу описана всередині статичної функції. Таким чином якщо змінна не ініціалізована – буде створений новий об’єкт. У противному випадку повернеться адреса вже існуючого.

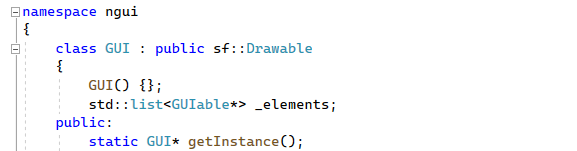


Рисунок 3.14 «Одинак» у класі GUI

У класі GUI реалізація патерну «Одинак» ідентична такому у Figure\_handler (рис. 3.14, 3.15).

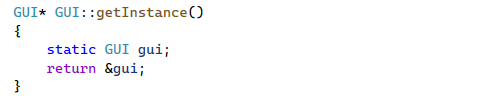


Рисунок 3.15 – Метод getInstance у класі GUI

4) Патерн «Знімок».

Для реалізації патерну «Знімок» був модифікований клас Figure\_handler та створено 2 нових класи: Memento та Caretaker.

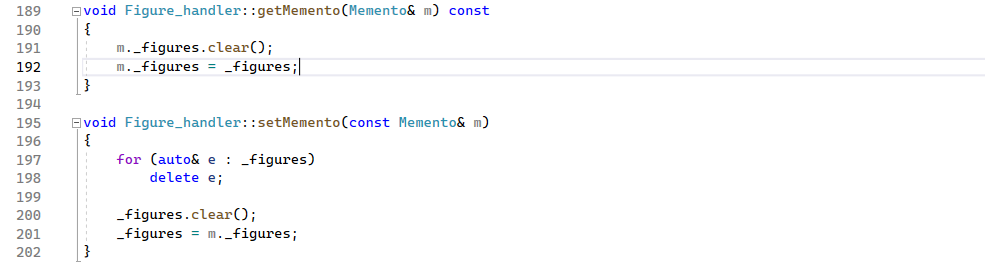


Рисунок 3.16 – Модифікація Figure\_handler

У клас Figure\_handler було додано два методи, які повертають/записують поточний стан сцени: getMemento та setMemento. Також в одному з випадків звільнюється динамічно виділена пам’ять, щоб запобігти витокам памяті.

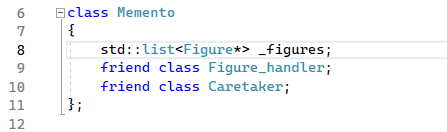


Рисунок 3.17 – Клас Memento

Вище на рисунку 3.17 подана реалізація класу Memento, який складається тільки з одного поля – списку об’єктів фігур та позначення класів Figure\_handler та Caretaker дружніми для можливості доступу до класу та одночасно збереження інкапсуляції відносно інших об’єктів.

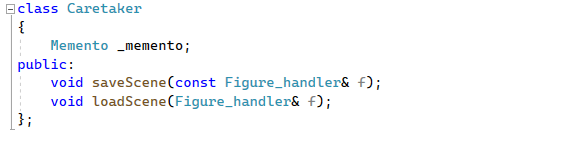


Рисунок 3.18 – Опис класу Caretaker

Клас Caretaker (рис. 3.18) містить в собі поле типу Memento та два методи які з ним взаємодіють та які працюють саме з записом у файл.

Під час запису викликається спершу метод getMemento() (рис. 3.16) на даному контролері, з якого виймається стан сцени і надалі записується у файл. Було обрано формат файлу “.txt” для збереження даних.

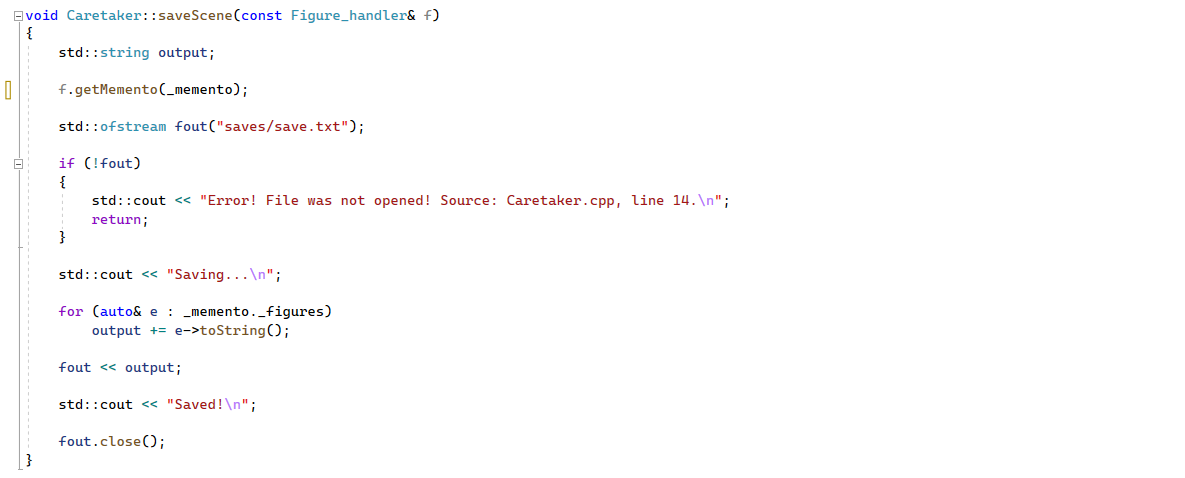


Рисунок 3.19 – Метод saveScene

При записі почергово кожного об’єкта викликається метод toString, який будує наступні структури (рис. 3.19):

Для простої фігури:

Тип\_фігури

Колір

Розмір\_фігури

Позиція\_Х Позиція\_Y

Для композиту

Composite

<Список фігур>

/Composite

При десеріалізації об’єктів з файлу методом loadScene програма зчитує типи фігур. Якщо тип Composite, то вона рекурсивно починає зчитувати вміст, поки не дійде до закриваючого тегу.

Крім реалізації наведених вище патернів була приділена увага рефакторингу та покращенню якості кода в цілому.

Був примінений принцип ООП DRY спільно з «Відокремленням методу», для знаходження однакових ділянок коду і винесення їх в окрему функцію з подальшим її визовом в цих місцях. Деякі функції були розділені на 2 або 3 частини, заради зменшення конкретної функції в розмірі. Це полегшує сприйняття коду, особливо якщо іменувати новостворені по їх призначенню.

По всьому проекту був відредагований стиль іменування змінних, класів та функцій. Відтепер всі змінні мають стиль snake\_case, функції – camelCase, а класи – PascalCase. Приватні поля мають префікс «\_», а константи Caps’ом.

Також була приділена увага очищенню динамічної пам’яті, створенню відповідних деструкторів та дослідженню на витоки пам’яті.

# 4 ТЕСТУВАННЯ

Перевіримо функціональність реалізованих патернів та коду у цілому.

Спробуємо створити композит та зберегти його у файл (рис. 4.1).

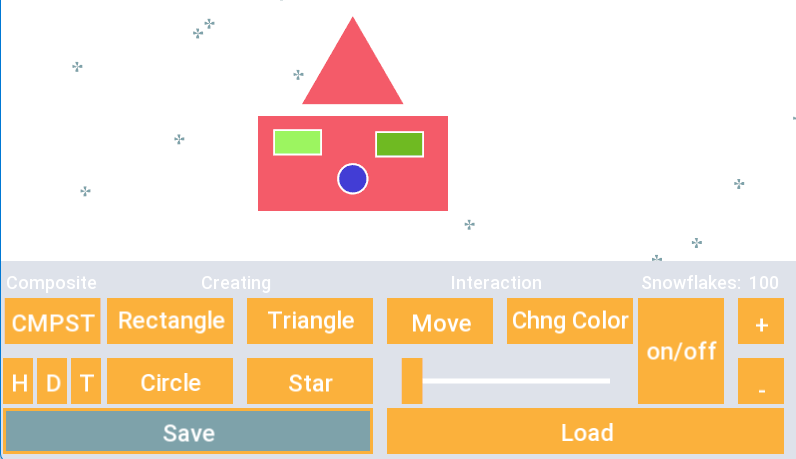


Рисунок 4.1 – Створення та збереження композитної фігури

Відкривши файл бачимо записані дані (рис. 4.2).

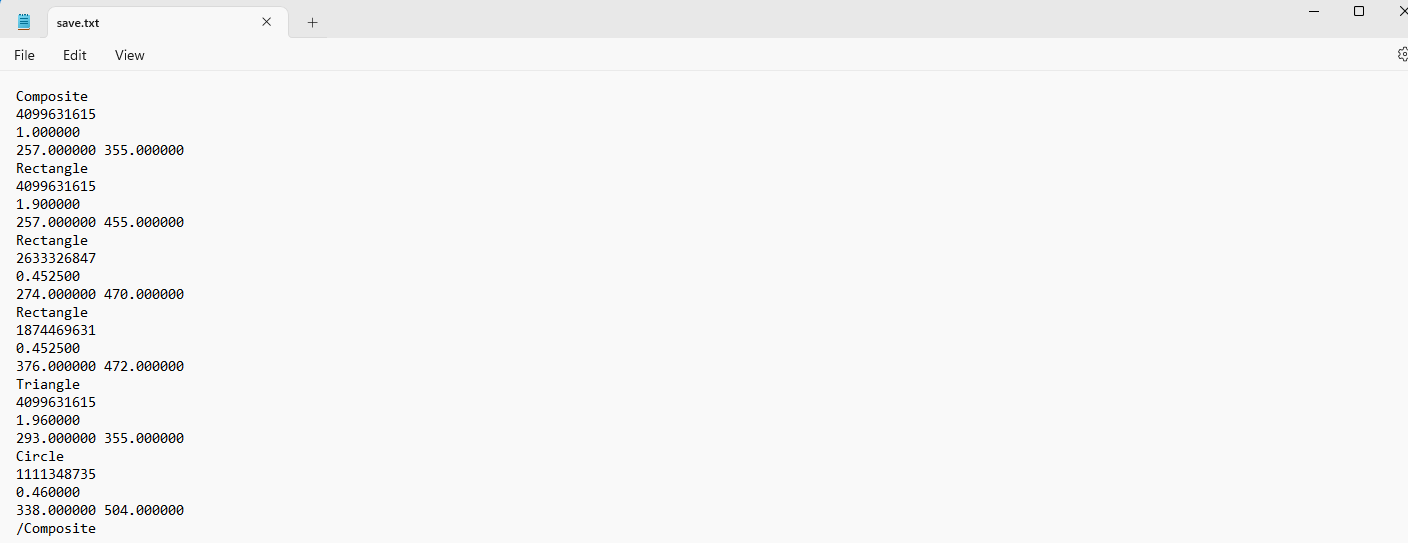


Рисунок 4.2 – Збережені дані сцени

Тепер перемістимо об’єкт, змінимо його колір та спробуємо завантажити попередній стан. Рисунок 4.3 – До, рисунок 4.3 – Після.

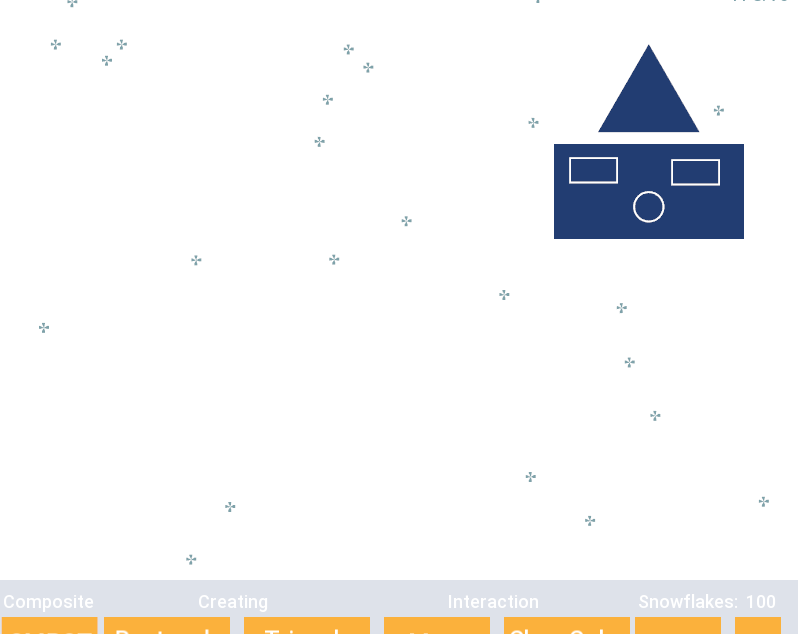


Рисунок 4.3 – Композит змінений

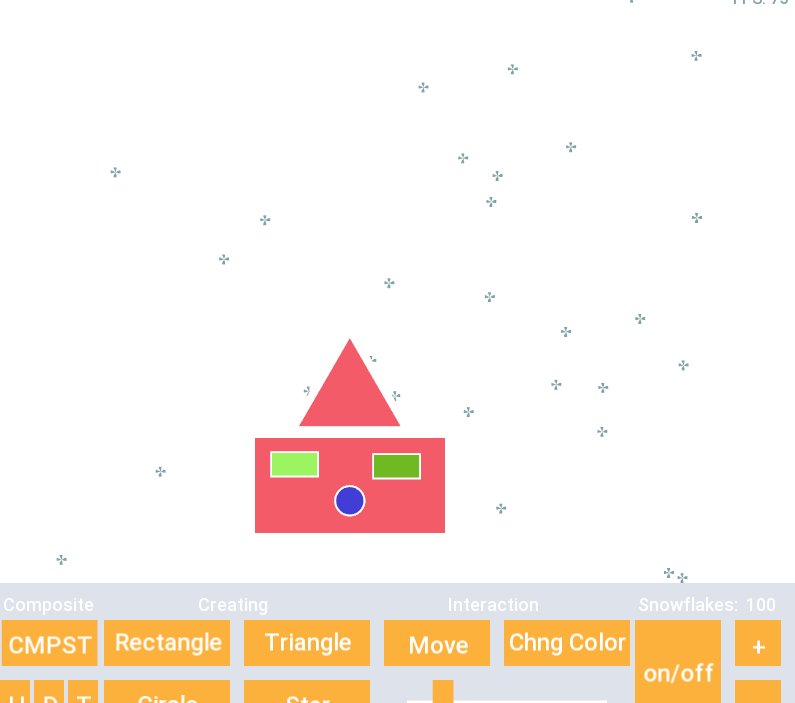


Рисунок 4.4 - Композит відновлений

У випадку якщо файл порожній, буде завантажена пуста сцена.

Тепер спробуємо клонувати цей об’єкт. Щоб зробити копію треба виділити об’єкт та натиснути по кнопці «С», яка з’явиться поруч з ним (рис. 4.5). Після створення копії маємо 2 однакових об’єкти (рис. 4.6).

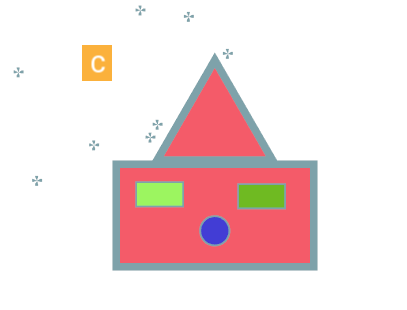


Рисунок 4.5 – Виділена фігура з кнопкою «Клонувати» поряд

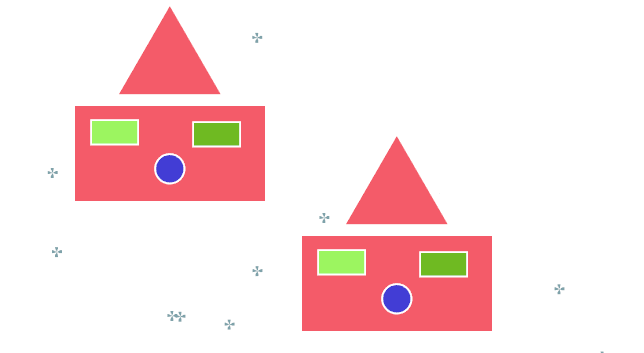


Рисунок 4.6 – Об’єкт скопійований

# 5 СПИСОК ЛІТЕРАТУРИ

1. Патерн Компонувальник. URL: <https://refactoring.guru/uk/design-patterns/composite> (дата звернення: 19.03.2023)
2. Фримен Эр., Фримен Эл., Сьерра К., Бейтс Б. — Паттерны проектирования (Head First O’Reilly) – 2011 (дата звернення: 19.03.2023)
3. Патерн Прототип. URL: <https://refactoring.guru/uk/design-patterns/prototype> (дата звернення: 19.03.2023)
4. Патерн Одинак. URL: <https://refactoring.guru/uk/design-patterns/singleton> (дата звернення: 19.03.2023)
5. Патерн Знімок. URL: <https://refactoring.guru/uk/design-patterns/memento> (дата звернення: 19.03.2023)
6. Прототип (Prototype). URL: <https://metanit.com/sharp/patterns/2.4.php> (дата звернення: 19.03.2023)
7. Короткий опис чотирьох патернів: [14. Патерни\_проєктування\_для\_курсової\_роботи.uk.pdf](https://dnufpm.sharepoint.com/:b:/r/sites/2-20-21/DocLib/2%20%D1%81%D0%B5%D0%BC%D0%B5%D1%81%D1%82%D1%80/%D0%9B%D0%B5%D0%BA%D1%86%D1%96%D1%97/14.%20%D0%9F%D0%B0%D1%82%D0%B5%D1%80%D0%BD%D0%B8_%D0%BF%D1%80%D0%BE%D1%94%D0%BA%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B4%D0%BB%D1%8F_%D0%BA%D1%83%D1%80%D1%81%D0%BE%D0%B2%D0%BE%D1%97_%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B8.uk.pdf?csf=1&web=1&e=5etOpd) (дата звернення: 20.03.2023)
8. Патерн проектування Компонувальник (Composite): <https://habr.com/ru/post/85166/> (дата звернення: 19.03.2023)
9. Техніки та прийоми рефакторингу коду. URL: <https://refactoring.guru/uk/refactoring/techniques> (дата звернення: 17.03.2023)