**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

Гудимова Д.С.

(прізвище та ініціали)

Керівник Емел'яненко Т.Г.

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](#_Toc128415769)

[2 ТЕОРЕТИЧНА ЧАСТИНА 4](#_Toc128415770)

[2.1 Патерн Композит (Composite) 4](#_Toc128415771)

[2.2 Патерн Прототип (Prototype) 6](#_Toc128415772)

[2.3 Патерн Одинак (Singleton) 7](#_Toc128415773)

[2.4 Патерн Знімок (Memento) 8](#_Toc128415774)

[3 ПРАКТИЧНА РЕАЛІЗАЦІЯ 10](#_Toc128415775)

[3.1 Структура програми 10](#_Toc128415776)

[3.2 Інтерфейс для користувача 11](#_Toc128415777)

[3.3 Рефакторинг 12](#_Toc128415778)

[3.4 Реалізація патернів 13](#_Toc128415779)

[4 ТЕСТУВАННЯ 18](#_Toc128415780)

[5 СПИСОК ЛІТЕРАТУРИ 23](#_Toc128415781)

1 ПОСТАНОВКА ЗАДАЧІ

Провести рефакторинг роботи з непарного семестру з  
використанням патернів проектування Композит (Composite), Прототип (Prototype), Одинак (Singleton) і Знімок (Memento), таким чином, щоб:

1. Модель роботи з агрегатом описував 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 проектування є додатковою перевагою.

Програма повинна бути розроблена згідно вимогам об’єктноорієнтованого 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

Для реалізації завдання поставленого в дослідницькій роботі необхідно розглянути теоретичну частину патернів зазначених у пункті “Постановка задачі”.

2.1 Патерн Композит (Composite)

Композит — це структурний патерн проектування, що дає змогу згрупувати декілька об’єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одиничний об’єкт.

Цей патерн корисний для вирішення задач, коли потрібно працювати зі складними ієрархіями об'єктів та їх комбінаціями. Наприклад для графічних примітивів, таких як лінії, круги, прямокутники та інші фігури, цей патерн може бути корисним, коли потрібно створити та працювати зі складними об’єктами, які складаються з багатьох простих примітивів або інших композитів. Наприклад, якщо ми створюємо графічний редактор, який дозволяє користувачам створювати складні фігури, ми можемо використовувати патерн Composite, що дозволить легко додавати нові примітиви до існуючих складних фігур та забезпечує єдиний інтерфейс для доступу до всіх примітивів у дереві, що значно полегшує роботу з графічними об'єктами.

Наведемо реалізацію патерну та його учасників:

1. Компонент (Component):
   * описує загальний інтерфейс для простих і складових компонентів дерева.
2. Лист (Leaf):
   * простий компонент дерева, який не має відгалуження;
   * класи листя міститимуть більшу частину корисного коду, тому що їм нікому передавати його виконання.
3. Композит (Composite):
   * складовий компонент дерева;
   * містить набір дочірніх компонентів, але нічого не знає про їхні типи.
   * усі дочірні компоненти дотримуються єдиного інтерфейсу;
   * методи композиту переадресовують основну роботу своїм дочірнім компонентам, хоча можуть додавати щось своє до результату.
4. Клієнт (Client):
   * працює з деревом через загальний інтерфейс компонент (Component).

Приклад де Композит (Composite) допомагає реалізувати вкладені геометричні фігури [5] (дивіться рисунок 2.1).

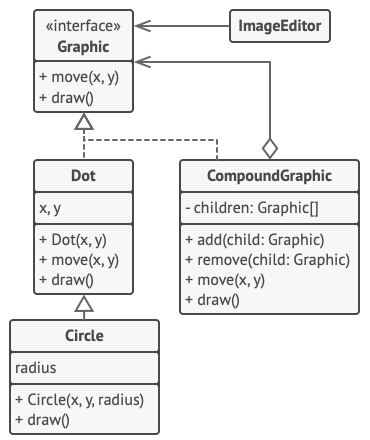


Рисунок 2.1 – Структура патерну Композит

У цьому прикладі використовується «інтерфейс» Graphic, який реалізує клас Dot в свою чергу його реалізує клас Circle, отже вони можуть наслідувати та викликати методи батьківського класу такі як «move()» і «draw()». Клас CompoundGraphic представляє з себе контейнер який може містити будь-яку кількість вкладених фігур, які можуть бути і простими фігурами, і контейнерами, які можуть містити інші фігури. Контейнер реалізує ті ж самі методи, що і прості фігури, але замість того, щоб виконувати дії безпосередньо, він передає виклики всім вкладеним фігурам, використовуючи рекурсію. Після цього він «підсумовує» результати всіх вкладених фігур. Клієнтський код (ImageEditor) взаємодіє з усіма фігурами через загальний інтерфейс фігур, і не знає, які фігури конкретно містяться в контейнері. Це дозволяє клієнту працювати з будь-яким деревом об’єктів будь-якої складності, не залежно від конкретних класів об’єктів, які формують дерево.

2.2 Патерн Прототип (Prototype)

Прототип - це патерн створення, що дозволяє створювати нові об'єкти на основі копіювання вже існуючих об'єктів, замість того, щоб створювати їх з нуля. Клонування здійснюється самим об’єктом-прототипу, що дозволяє йому скопіювати значення всіх полів, навіть приватних.

Використання патерну Прототип може зменшити кількість коду, який необхідно написати для створення нового об'єкта графічного примітиву, та сприяти більш ефективному використанню пам'яті.

Основні компоненти патерну:

1. Прототип (Prototype) визначає інтерфейс для клонування себе самого, щоб створювати нові об'єкти на його основі.
2. Конкретний прототип (ConcretePrototype) реалізує клонування себе самого, яке може використовуватися для створення нових об'єктів.
3. Клієнт (Client) використовує клонування для створення нових об'єктів на основі прототипів.

Приклад де Прототип дозволяє робити точні копії об’єктів геометричних фігур без прив’язки до їхніх класів [7] (дивитись рисунок 2.2).

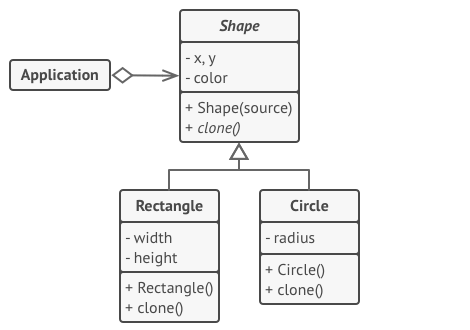


Рисунок 2.2 – Структура патерну Прототип

У прикладі клас Shape реалізує інтерфейс клонування і надає метод для відтворення самої себе. В свою чергу підкласи Rectangle і Circle використовують батьківський метод клонування, а потім копіюють власні поля до створеного об’єкта. І це дозволяє створювати нові фігури клонуючи вже існуючі. А клієнтський код (Application) звертається до Shape щоб той створив свою копію.

2.3 Патерн Одинак (Singleton)

Патерн програмування Одинак (Singleton) - це патерн створення, який гарантує, що в системі існує тільки один екземпляр певного класу, і забезпечує глобальну точку доступу до цього екземпляра.

Основна ідея патерну полягає в тому, що визначений клас має метод, який перевіряє наявність екземпляру (instance) класу та повертає посилання на нього, якщо він вже був створений, або створює новий екземпляр, якщо цього ще не було зроблено.

Основні компоненти патерну:

1. Одинак: клас, який містить єдиний статичний екземпляру, та метод для доступу до цього екземпляру.
2. Клієнт: об'єкт, який використовує екземпляр класу Одинак.

Приклад де патерн Одинак повертає користувачу зображення [3] (дивитись рисунок 2.3).

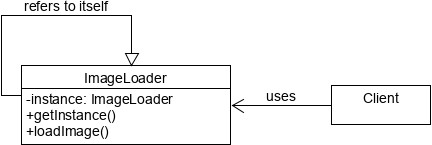


Рисунок 2.3 – Структура патерну Одинак

Уявімо, що ми маємо клас, який застосовується генерації фото і ми не хочемо створювати нове фото кожного разу, коли потрібно виконати запит. Ми можемо використовувати патерн Одинак, щоб гарантувати, що в нашому класі буде тільки один єдиний екземпляр, який будемо використовувати і в подальшому у додатку.

Отже клас Одинак буде мати статичний метод, який перевірятиме наявність екземпляр класу та повертатиме посилання на нього, якщо екземпляр вже було створено, або створюватиме новий, якщо цього ще не було зроблено. Клієнтський код буде використовувати цей статичний метод для отримання посилання на інстанцію класу.

2.4 Патерн Знімок (Memento)

Патерн Знімок (Memento) -  це поведінковий патерн проектування, що дозволяє зберігати та відновлювати попередні стани об'єктів.

Основні компоненти патерну:

1. Творець (Originator): клас, який зберігає свій поточний стан та може створювати знімки свого стану.
2. Знімок (Memento): клас, який зберігає стан Творця у певний момент часу.
3. Опікун (Caretaker): клас, який зберігає знімки та дозволяє відновлювати стан Творця до певного збереженого стану.

Приклад де реалізовано збереження даних у файл та відновлення попереднього стану об’єкту звідти завдяки патерну Знімок [4] (дивитись рисунок 2.4)

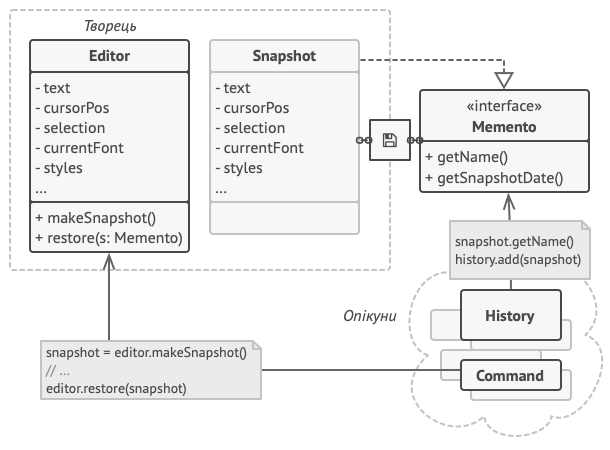


Рисунок 2.4 – Структура патерну Знімок

Уявімо, що у нас є текстовий редактор, який має можливість зберігати та відновлювати попередні стани документів. Ми можемо застосувати патерн Знімок, створивши класи «Редактор тексту» (Originator), «Знімок документа» (Memento) та «Опікун» (Caretaker).

Коли користувач зберігає документ, Творець створює новий «знімок документа» та зберігає його в файл. Коли користувач хоче відновити попередній стан документа, Опікун повертає відповідний «знімок документа» Творцю, який відновлює свій стан до попереднього, який зберігся в знімку.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

* 1. Структура програми

Для вирішення поставленого завдання у пункті «Постановка задачі» було розроблено наступні класи:

1. Figure – абстрактний клас який реалізовує спільні методи та поля для фігур.
2. Circle, Line, Square, Star та Triangle – класи які наслідують абстрактний клас Figure та реалізують поведінку примітивних фігур.
3. Composite – цей клас також наслідує Figure, але створений для можливості працювати із складними фігурами (композиції із примітивних фігур) реалізуючи патерн програмування Композит.
4. Controller – клас який слідкує за тим що відбувається на сцені та керує об’єктами на екрані додатково реалізуючи патерн Одинак.
5. Memento та FigureHistory – ці класи реалізують патерн Знімок, що дозволяє зберігати об’єкти на сцені у файл на диску та відновлювати сцену з файлу.

Реалізація програми представлена у вигляді діаграми завдяки функціоналу Visual Studio (дивитись рисунок 3.1)

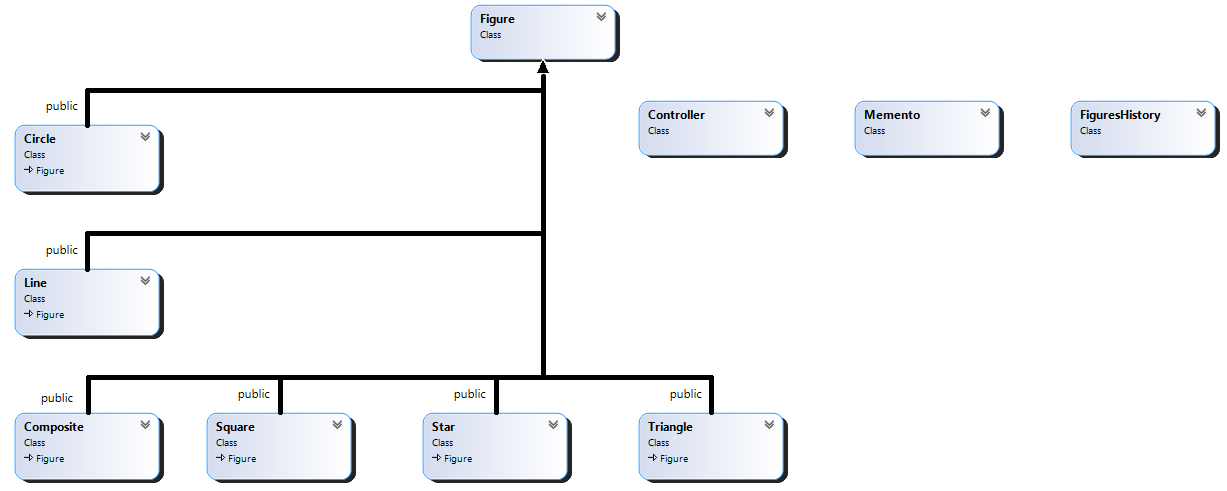


Рисунок 3.1 – Діаграма реалізації класів для завдання

* 1. Інтерфейс для користувача

Для того щоб використовувати функціонал програми користувачу потрібно взаємодіяти зі сценою завдяки клавіатурі та мишці (дивитись рисунок 3.2)

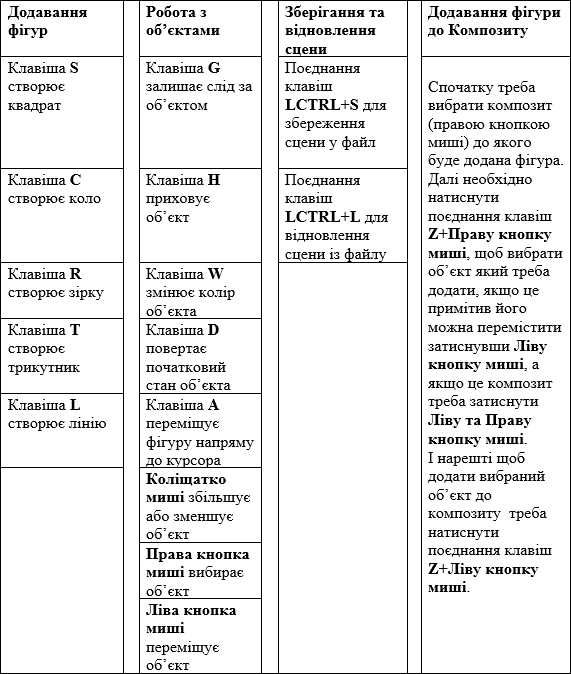


Рисунок 3.2 – Інструкція програми для користувача

* 1. Рефакторинг

Згідно постановки завдання був проведений рефакторинг проекту [8]   для полегшення розуміння коду та внесення подальших правок без зміни зовнішньої поведінки самої системи. Для рефакторингу були використані наступні принципи ООП дизайну [6]:

1. DRY (Don’t Repeat Yourself, Не Повторюйтесь):
   * повторювані блоки коду були винесені у окремий метод   
     (дивитись рисунок 3.3).

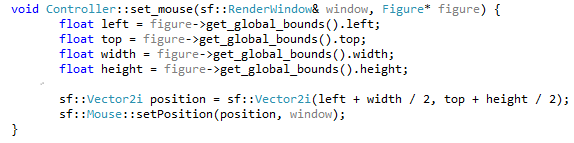


Рисунок 3.3 – Приклад принципу DRY

1. Open Closed Principle (OCP, Принцип відкритості/закритості):
   * класи розробляються так, щоб для підстроювання класу до конкретних умов застосування було достатньо розширити його (зробити нащадка) та перевизначити деякі функції (наприклад при необхідності нової фігури прямокутник буде створений клас Rectangle який наслідує клас Square).
2. Liskov’s Substitution Principle (LSP, Принцип підстановки Барбари Лісков):
   * при успадкуванні від базового класу очікувана поведінка перевизначених методів не змінюється (наприклад при розширені абстрактного класу Figure клас Square не змінює базової поведінки і може бути використаний як Фігура).
3. YAGNI (You Aren't Gonna Need It, Вам Це Не Знадобиться):
   * була видалена можливість змінювати режим відображення фігур, бо для вирішення завдання вона більше не потрібна (дивитись рисунок 3.4).

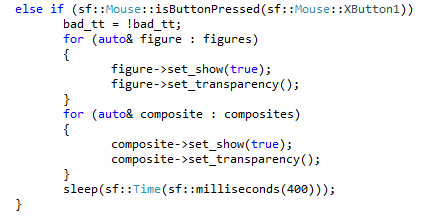


Рисунок 3.4 – Видалення непотрібної можливості

1. А також угоди написання коду (Code Convention) для мови с++ [2]:
   * змінені імена методів та типів на стиль snake\_case (наприклад «transparent\_color» «get\_global\_bounds()»);
   * додані префікс для булевих (логічних), статичних змінних та покажчиків (наприклад «is\_composite», «s\_instance» та «p\_draggable\_composite» ).
   1. Реалізація патернів

Для реалізації проекту були використано 4 патерни програмування: «Композит», «Одинак», «Прототип» та «Знімок». Далі буде описано як вони реалізовані у програмі та представлені шматки коду для кращого розуміння написаного.

1. Композит

Патерн Композиту реалізований завдяки створенню класу Composite який як і інші класи фігур наслідує абстрактний класу Figure і реалізує віртуальні(virtual) методи. Додатково було створено методи додавання об’єктів та видалення (дивитись рисунок 3.5).

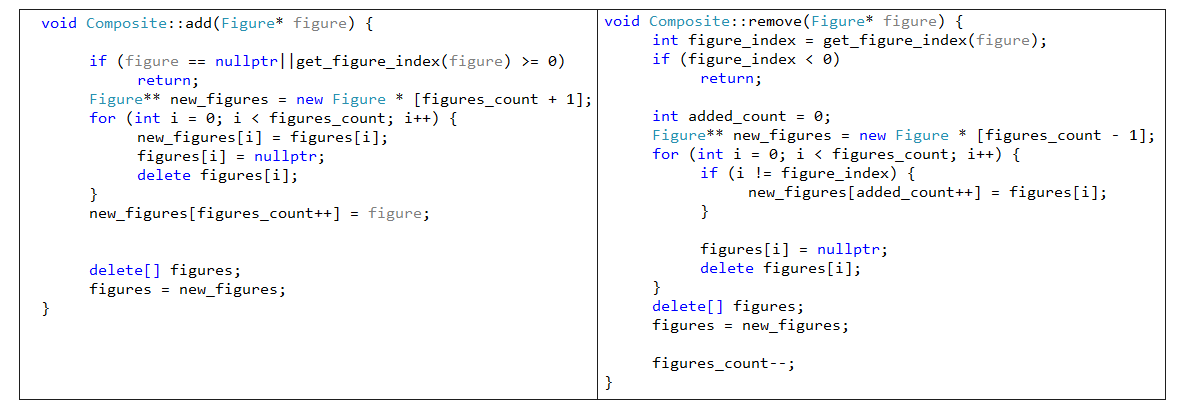


Рисунок 3.5 – Методи add та remove

Так як клас Composite реалізує абстрактний клас він також фігура хоч і складна він також може бути доданий до іншого композиту. Щоб не реалізовувати код двічі методи композиту по черзі звертаються до методів фігур які у ньому знаходяться (дивитись рисунок 3.6)

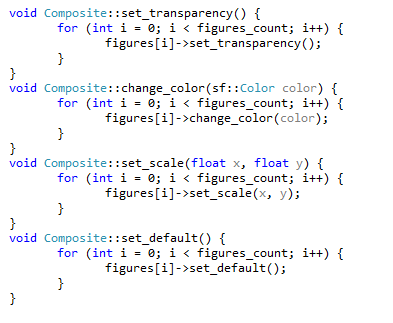


Рисунок 3.6 – Приклад перевизначених методів

1. Одинак

Патерн Одинак реалізований завдяки контролеру сцени (клас Controller), так як для виконання програми потрібна одна сцена. Для реалізації патерну в цей клас був доданий метод «get\_instance» який буде забезпечувати глобальну точку доступу до екземпляру (дивитись рисунок 3.7).

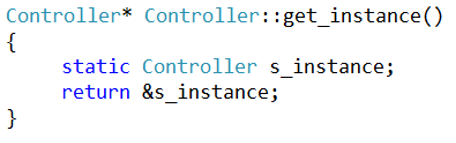


Рисунок 3.7 – Реалізація методу get\_instance

1. Знімок

Патерн Знімок реалізований завдяки створенню класів Memento та FigureHistory і використанню вже існуючого класу Controller (дивитись рисунок 3.5). На додачу для простішого запису у файл та зчитування з нього було використано бібліотеку «nlohmann.json» [1].

Далі розглянемо компоненти патерну та їх методи:

1. Controller (Творець):
   * Метод «set\_memento» який зберігає поточні дані сцени у форматі json;
   * Метод «get\_memento» який повертає дані для сцени із формату json.
2. Memento (Знімок):
   * Має приватні поля-контейнери figures та composites які зберігають поточні фігури та композити зі сцени.
3. FigureHistory (Опікун):
   * Метод «save\_state» який викликає метод класу Controller «set\_memento» і зберігає данні у файл;
   * Метод «load\_state» який викликає метод класу Controller «get\_memento» і відновлює данні із файлу.
4. Прототип

Патерн Прототип реалізований завдяки реалізації конструктора копіювання для кожної фігури та композиту і методу «clone» який повертає посилання на глибоку копію об’єкта (дивитись рисунок 3.8). Також «clone» реалізований у композиті таким чином що створюється новий контейнер такого ж розміру як і оригінальний і після чого для кожної фігуру по черзі викликається метод «clone» і повертається посилання на копійований об’єкт (дивитись рисунок 3.9)

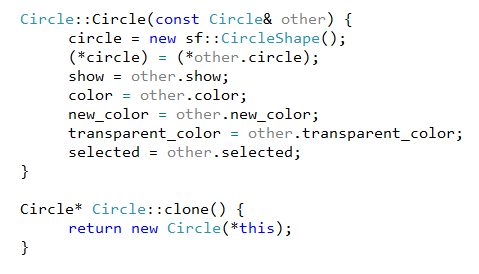


Рисунок 3.8 – Реалізація конструктора копіювання та методу clone для примітиву

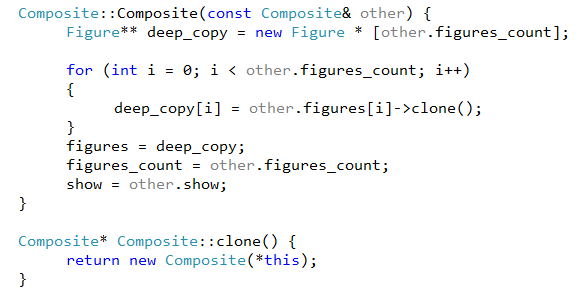


Рисунок 3.8 – Реалізація конструктора копіювання та методу clone для композиту

4 ТЕСТУВАННЯ

Для перевірки коректності реалізації програми та патернів програмування було проведено ряд тестів результат яких будуть надані нижче.

Приклад використання патерну Композит (дивитись рисунок 4.1)

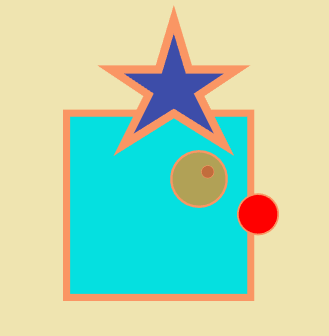
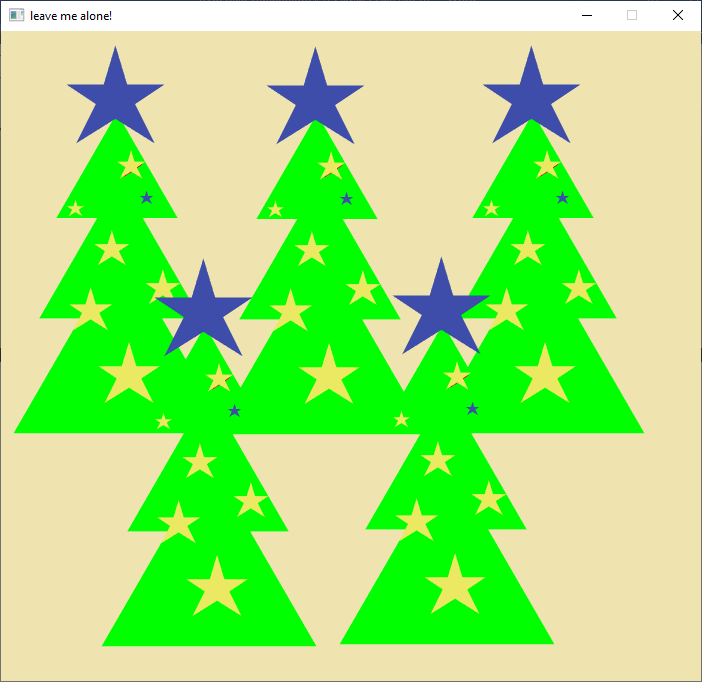


Рисунок 4.1 – Створення композиту

Приклад використання патерну Прототип (дивитись рисунок 4.2). У прикладі спочатку було створено композит ялинка з використанням прототипів трикутника та зірки , а потім при завершені композиту були створені його прототипи. Під час створення прототипів використовується метод «clone» та конструктор копіювання реалізований у примітивів та композитів. Отже можна побачити, що патерн Прототип реалізований у композиті працює за тим же самим принципом що й примітивів.

  
Рисунок 4.2 – Створення прототипів

Приклад використання патерну Одинак (дивитись рисунок 4.3). У прикладі спочатку було створено два екземпляри «» «» класу Controller завдяки методу «get\_instance» (дивитись рисунок 4.4) і використаний метод «test» який виводить його адресу у консоль(дивитись рисунок 4.5), а потім інший екземпляр який також визиває цей метод. Нижче наведено результат тестування.

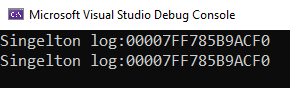


Рисунок 4.3 – Результат роботи патерну Одинак

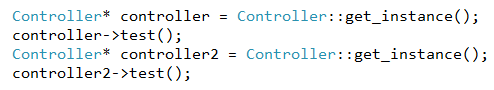


Рисунок 4.4 – Створення 2 покажчиків класу Controller та виклик методу test



Рисунок 4.5 – Метод test

Приклад використання патерну Знімок. У прикладі спочатку була збережена пуста сцена у файл (дивитись рисунок 4.6)

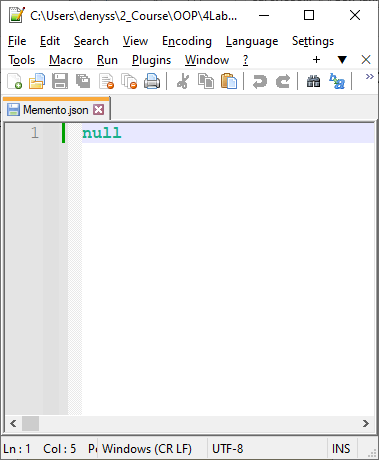
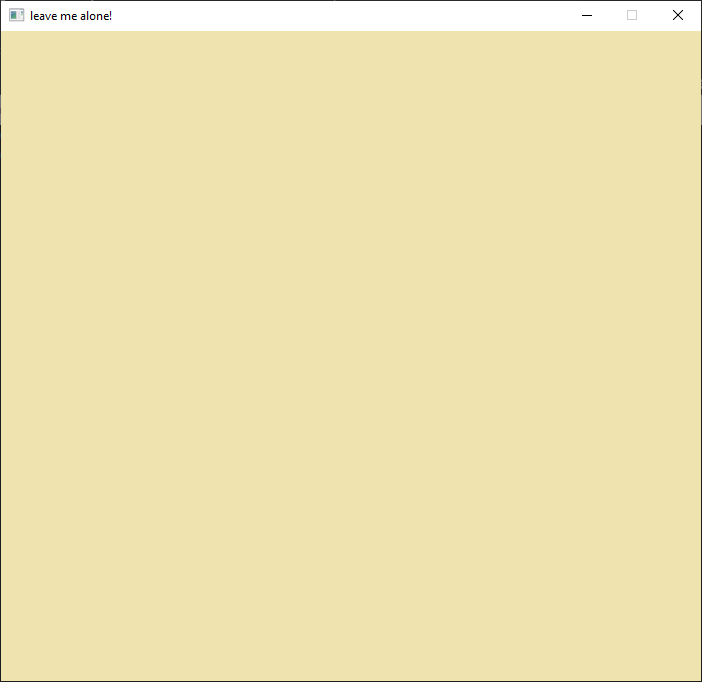


Рисунок 4.6 – Збереження пустої сцени

Потім було створено два композити (дивитись рисунок 4.7) й збережено сцену у файл (дивитись рисунок 4.8), а після збереження вони були об’єднані у один композит (дивитись рисунок 4.8), а наприкінці сцена була відновлена із файлу до збереженого стану (дивитись рисунок 4.10).

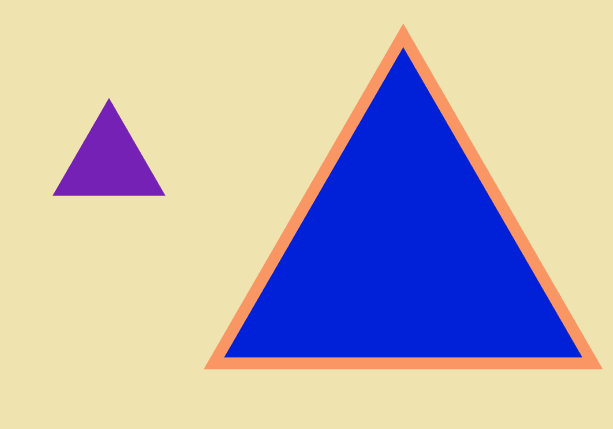


Рисунок 4.7 – Створення 2 композитів

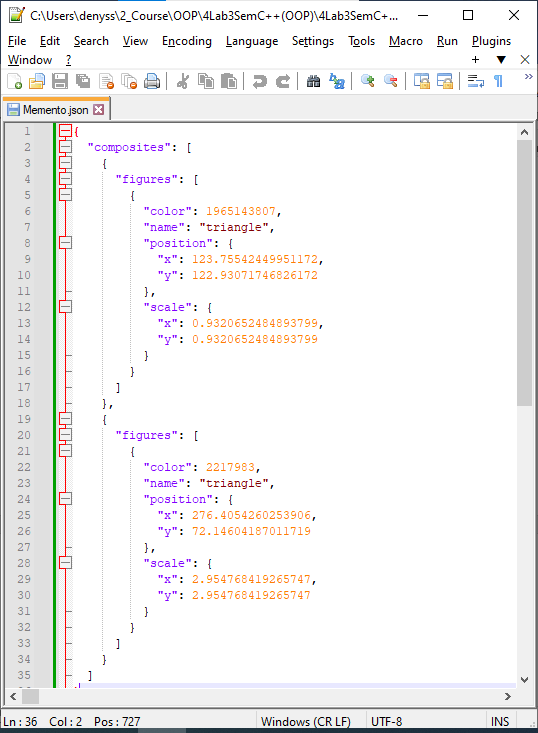


Рисунок 4.8 – Збереження сцени у файл

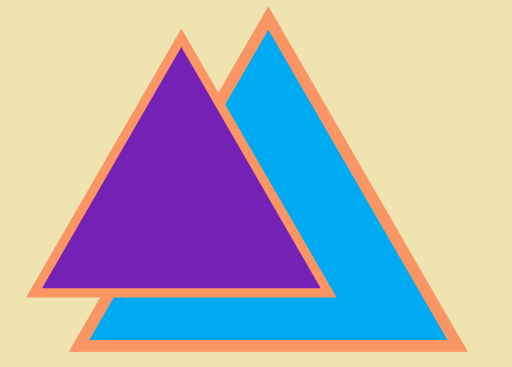


Рисунок 4.9 – зміна розміру, кольору та об’єднання в один композит

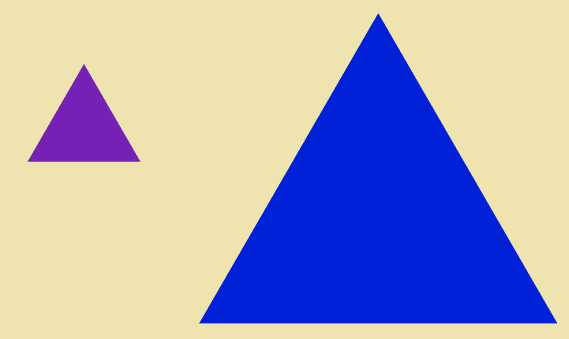


Рисунок 4.10 – Відновлення сцени із файлу до збереженого вигляду

5 СПИСОК ЛІТЕРАТУРИ

1. JSON for Modern C++ API Documentation. URL: <https://json.nlohmann.me/api/basic_json/> (дата звернення: 25.02.2023).
2. Naming Convention in C++. URL: <https://www.geeksforgeeks.org/naming-convention-in-c/> (дата звернення: 24.02.2023).
3. Singleton Pattern Explained. URL: <https://darrenfinch.com/singleton-pattern-explained-creational-design-patterns/> (дата звернення: 23.02.2023).
4. Знімок. URL: <https://refactoring.guru/uk/design-patterns/memento> (дата звернення: 25.02.2023).
5. Компонувальник. URL: <https://refactoring.guru/uk/design-patterns/composite> (дата звернення: 11.02.2023).
6. Навіщо дотримуватись принципів програмування? URL: <https://senior.ua/articles/kiss-dry-solid-yagni--navscho-dotrimuvatis-principv-programuvannya>
7. Прототип. URL: <https://refactoring.guru/uk/design-patterns/prototype> (дата звернення: 24.02.2023).
8. Рефакторинг. URL: <https://refactoring.guru/uk/refactoring> (дата звернення: 17.02.2023).