**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-21-3

Спеціальності 121 Інженерія програмного забезпечення

Кравченко Є. Д.

(прізвище та ініціали)

Керівник

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

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

Члени комісії : \_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

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

(підпис) (прізвище та ініціали)

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

(підпис) (прізвище та ініціали)

м. Дніпро, 2021 р.

**ЗМІСТ**

[1 ПОСТАНОВКА ЗАДАЧІ 3](#_Toc128996726)

[2 ТЕОРЕТИЧНА ЧАСТИНА 4](#_Toc128996727)

[2.1 Патерн Composite 4](#_Toc128996728)

[2.2 Патерн Prototype 4](#_Toc128996729)

[2.3 Патерн Singleton 5](#_Toc128996730)

[2.4 Патерн Memento 5](#_Toc128996731)

[3 ПРАКТИЧНА РЕАЛІЗАЦІЯ 6](#_Toc128996732)

[3.1 Реалізація патернів 6](#_Toc128996733)

[3.1.1 Реалізація Composite 6](#_Toc128996734)

[3.1.2 Реалізація Prototype 9](#_Toc128996735)

[3.1.3 Реалізація Singleton 9](#_Toc128996736)

[3.1.4 Реалізація Memento 10](#_Toc128996737)

[3.2 Інтерфейс користувача 12](#_Toc128996738)

[4 ОПИС ТЕСТОВИХ ПРИКЛАДІВ 13](#_Toc128996739)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 21](#_Toc128996740)

# 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).

# ТЕОРЕТИЧНА ЧАСТИНА

* 1. Патерн Composite

Composite – це труктурний шаблон проектування, що поєднує об'єкти в деревоподібну структуру для подання ієрархії від часткового до цілого. Компонувальник дозволяє клієнтам звертатися до окремих об'єктів та груп об'єктів однаково.

Ключова ідея патерна Компонувальник полягає у його рекурсивній природі. В основі ієрархії класів, що реалізують цей патерн лежить абстрактний клас.

Коли ви виконуєте операцію над деревоподібною структурою, операцію можна виконати над листовим вузлом або складовим вузлом. Операція виконується безпосередньо, якщо це кінцевий вузол. Операція делегується всім дочірнім компонентам, якщо це складовий вузол. Складовий вузол має список дочірніх елементів та функцій-членів для їхнього додавання або видалення. Отже, кожен компонент може обробляти операцію відповідним чином.

* 1. Патерн Prototype

Прототип — це патерн проектування, що дозволяє копіювати об'єкти, не вдаючись у подробиці їх реалізації.

Цей патерн забезпечує гнучкість створення складних об'єктів. Ідея полягає в тому, що тепер не потрібно створювати новий екземпляр з нуля.

У цьому випадку існуючий об'єкт виступає як прототип, а скопійований об'єкт може змінювати ті ж властивості тільки в разі необхідності. Такий підхід економить ресурси та час, особливо коли створення об'єктів – трудомісткий процес.

Таким чином, прототип це просто частково або повністю ініціалізований об'єкт, копію якого ви робите. А потім ви, згодом, використовуєте їх у своїх інтересах.

* 1. Патерн Singleton

Одинак ​​– це породжуючий патерн проектування, який гарантує, що клас має лише один екземпляр, і надає до нього глобальну точку доступу.

Шаблон Singleton накладає на клас такі обов'язки:

* контроль кількості копій за умовою завдання;

- за вимогою повернути необхідну кількість екземплярів;

- ведення обліку отриманих екземплярів.

Це реалізовано з використанням основних концепцій об'єктно-орієнтованого програмування, а саме модифікаторів доступу, конструкторів та статичних методів.

* 1. Патерн Memento

Знімок – це поведінковий патерн проектування, який дозволяє зберігати та відновлювати минулі стани об'єктів, не розкриваючи подробиць їх реалізації.

По мірі виконання вашої програми ви можете зберігати контрольні точки у своїй програмі та відновлювати їх пізніше. Мета шаблону Memento полягає в тому, щоб, не порушуючи інкапсуляцію, захопити та перетворити внутрішній стан об'єкта, щоб об'єкт можна було відновити в цей стан пізніше.

Зберігати стан об’єкту можна у різних текстових форматах на фізичному диску: json, xml і т.д, та потім використовувати за потреби.

1. **ПРАКТИЧНА РЕАЛІЗАЦІЯ**
   1. Реалізація патернів

3.1.1 Реалізація Composite

Для реалізації патерну Composite було створено ієрархію класів: абстрактний клас з віртуальними методами Figure; класи, які собою представляють прості фігури; клас Composite, який являє собою об’єднання декількох простих фігур.

Клас Figure є батьківським класом для класів Line, Circle, Square, Rectangle, Triangle, Star. Усі класи-нащадки мають перевизначені методи класу Figure, з їх допомогою у класі Composite можливість працювати з декількома фігурами як з одним об’єктом.

Наприклад, метод для малювання фігури на екрані *void draw(sf::RenderWindow&)*. Оскільки він оголошений у батьківському класі, усі його класи-спадкоємці можуть використовувати цей метод, а також сам Composite:

void Line::draw(sf::RenderWindow& window) {

if (is\_visible())

window.draw(\*line);

}

…

void Composite::draw(sf::RenderWindow& window) {

if (!is\_visible())

return;

for (int i = 0; i < figures\_count; i++) {

figures[i]->show();

figures[i]->draw(window);

}

}

Клас Composite:

class Composite : public Figure {

private:

int figures\_count; //Кількість фігур у композиті

Figure\*\* figures; // Масив фігур

Composite(const Composite& another); // Конструктор копіювання іншого композиту

public:

Composite(); // Конструктор за замовчуванням

Composite(Figure\* figure); // Конструктор для додавання фігури

~Composite(); // Деструктор

void add(Figure\* figure); // Метод додавання фігури у композит

void remove\_by\_index(int index); // Видалення фігури з композиту за індексом

void hide(); // Сховати композит

void show(); // Показати композит

bool check\_intersection\_by\_figure(Figure\* another); // Перевірити на перетин з ішною фігурою

bool check\_intersection\_by\_cords(float x, float y); // Перевірити на перетин з точними координатами

bool check\_collision(sf::RenderWindow& window); // Перевірити колізію вікна

sf::FloatRect get\_global\_bounds(); // Отримати глобальні границі

void set\_scale(float x\_scale, float y\_scale); // Назначити скейл

sf::Vector2f get\_scale(); // Отримати значення скейлу

void draw(sf::RenderWindow& window); // Намалювати фігуру

void move(float x, float y); // Перемістити фігуру на задані координати

void set\_position(float x, float y); // Назначити нову позіцію фігурі

sf::Vector2f get\_position(); // Отримати координати позиції

void set\_color(sf::Color color); // Назначити колір

sf::Color get\_color(); // Отримати колір фігури

void reset(); // Скинути фігуру до дефолтної

string get\_string(); // Отримати строковий вид фігури

Figure\* clone() const; // Метод клонування фігури

json serialize(); // Метод для серіалізації

void deserialize(json data); // Метод для десеріалізації

};

Один з класів простих фігур(Rectangle):

class Rectangle : public Figure {

private:

sf::Color default\_color;

sf::RectangleShape\* rectangle;

Rectangle(const Rectangle& another);

public:

Rectangle(float length, float width, sf::Color color);

~Rectangle();

sf::FloatRect get\_global\_bounds();

void set\_scale(float x\_scale, float y\_scale);

sf::Vector2f get\_scale();

bool check\_intersection\_by\_figure(Figure\* another\_figure);

bool check\_intersection\_by\_cords(float x, float y);

bool check\_collision(sf::RenderWindow& window);

void draw(sf::RenderWindow& window);

void move(float x, float y);

void set\_position(float x, float y);

void set\_color(sf::Color color);

sf::Color get\_color();

sf::Vector2f get\_position();

string get\_string();

Figure\* clone() const;

};

### **3.1.2 Реалізація Prototype**

Для реалізації патерну Prototype у класі Figure було оголошено віртуальний метод: *virtual Figure\* clone() const = 0.* Оскільки метод віртуальний, він буде перевизначений у всіх класах-нащадках. Це дозволяє створювати нові об’єкти шляхом копіювання вже існуючого об’єкту. Наприклад, перевизначений метод клонування у класі Star:

Figure\* Star::clone() const {

return new Star(\*this);

}

Це дозволяє створювати повну копію існуючого об’єкту. Створення нової фігури забезпечується викликом методу клонування у деякого існуючого об’єкта, наприклад:

Star\* star = new Star();

figures[i] = star->clone();

### **3.1.3 Реалізація Singleton**

Для реалізації патерну Singleton створено статичні поле і метод у класі Controller. Він спроектований так, щоб міг бути створений єдиний екземпляр класу. Для цього визначено приватний конструктор класу та створено статичний метод *Controller\* get\_instance()*, у якому, під час першого виклику, створюється об’єкт класу та повертається посилання на нього. Під час повторного виклику методу *get\_instance()* повертається посилання на вже існуючий об’єкт. Реалізація методу та взаємодія з ним:

Controller\* Controller::get\_instance() {

if (controller\_instance == nullptr) {

controller\_instance = new Controller();

}

return controller\_instance;

}

…

Controller\* controller = Controller::get\_instance();

### **3.1.4 Реалізація Memento**

Для реалізації патерну Memento створено клас Memento та оновлюються класи Controller, EventHandler та Figure.

Клас Memento відповідає за зберігання інформації про стан сцени у текстовому json-файлі на диску. В ньому оголошено конструктор класу та метод для доступу до стану сцени.

Для переведення сцени у послідовність байтів(серіалізації) та навпаки(десеріалізації) використовується додаткова бібліотека «nlohmann/json»

У класі Figure додано методи: *json serialize()* та *void deserialize()* для серіалізації та десеріалізації фігур відповідно.

У класі Controller оголошено методи: *Memento\* save(bool is\_auto\_move)* та void *restore\_scene(Memento\* snapshot)* для збереження та відновлення сцени в цілому в/з json формат(у).

Також до класу EventHandler додано методи: *void save\_scene(string filename)*, *void load\_scene\_form\_save(string filename)* та *void delete\_save\_file(string filename)* для запису та читання стану сцени та для видалення файлу з диску за заданим ім’ям.

class Memento {

private:

json state; // Поле для зберігання стану сцени у форматі json

public:

Memento(json origin\_state); // Конструктор з параметром

json get\_state(); // Геттер для отримання стану сцени

};

class Figure {

…

virtual json serialize(); // Метод для серіалізації фігури

virtual void deserialize(json data); // Метод для десерілалізації фігури

};

class EventHandler {

private:

…

void delete\_save\_file(string file\_name); // Видалити файл зі збереженою сценою

void save\_scene(string file\_name); // Метод для збереження стану сцену у файлі на диску

void load\_scene\_from\_save(string file\_name); // Метод для загрузки сцени з файлу };

* 1. Інтерфейс користувача

Після запуску програми з’являється вікно з пустою сценою, де користувач може виконувати усі дії, які передбачені функціоналом програми.

Керування фігурами здійснюється з використанням клавіш.

Клавіши для керування фігурами:

1. H – сховати фігуру.
2. Повторити H – показати фігуру.
3. D – скинути фігуру до дефолтної
4. Left arrow – показати попередню фігуру
5. Right arrow – показати наступну фігуру
6. G – рух фігури зі шляхом
7. F – змінити колір фігури
8. H – увімкнути/вимкнути рух по синусоїді

Клавіши для створення фігур:

1. C – створити круг
2. R – створити прямокутник
3. S – створити квадрат
4. L – створити лінію
5. P – створити зірку
6. T – створити трикутник

Клавіши для композит моду:

1. V – композит мод
2. Знову V – для додавання фігури у композит

Клавіши для керування станом сцени:

1. LCTRL + S – збереження стану сцену у файл
2. LCTRL + G – загрузити сцену з файлу
3. LCTRL + D – видалити файл зі збереженою сценою

# ОПИС ТЕСТОВИХ ПРИКЛАДІВ

Пуста сцена після запуску програми представлена на рисунку 1.

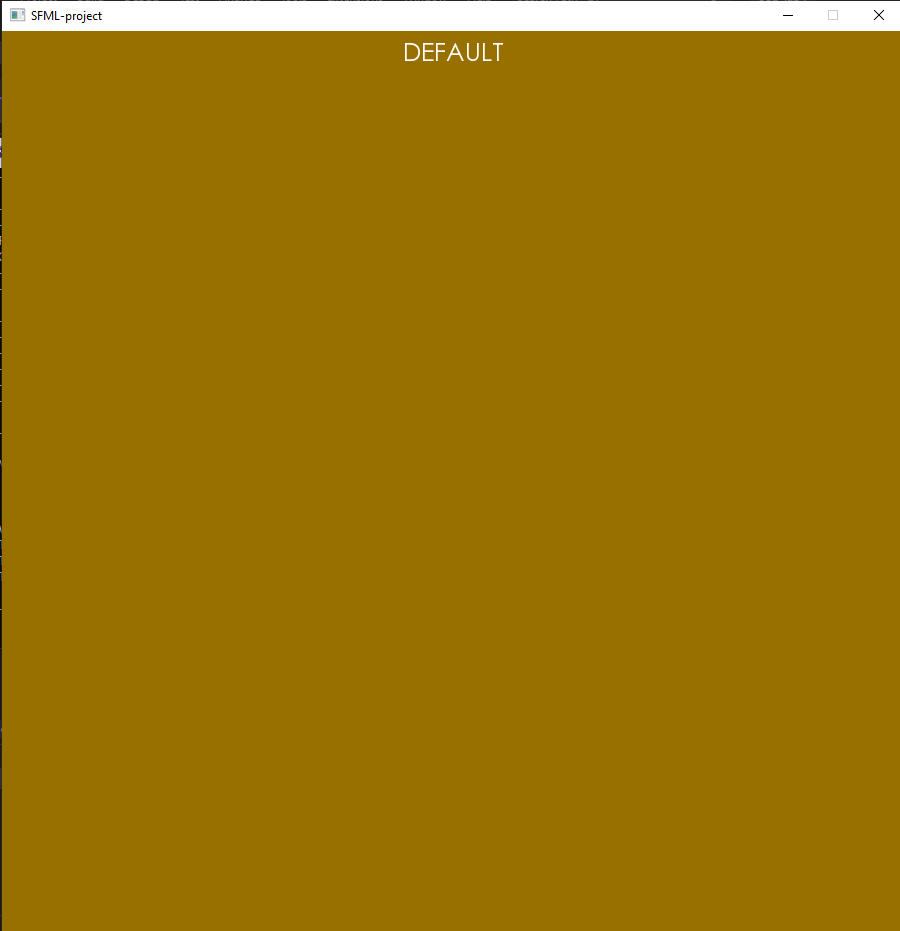


Рисунок 1 – Пуста сцена програми

Переміщення трикутника зі слідом представлене на рисунку 2.

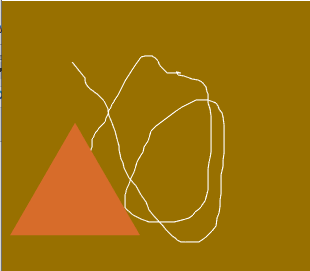
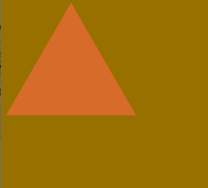


Рисунок 2 – Переміщення трикутника зі слідом

Рух простої фігури та композита в автоматичному режимі з переміщенням зі слідом представлений на рисунку 3 та 4 відповідно.

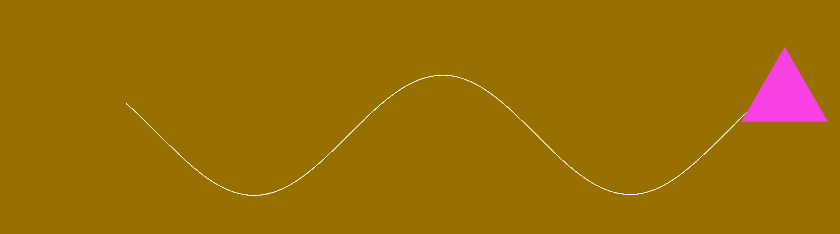


Рисунок 3 – Рух простої фігури в автоматичному режимі

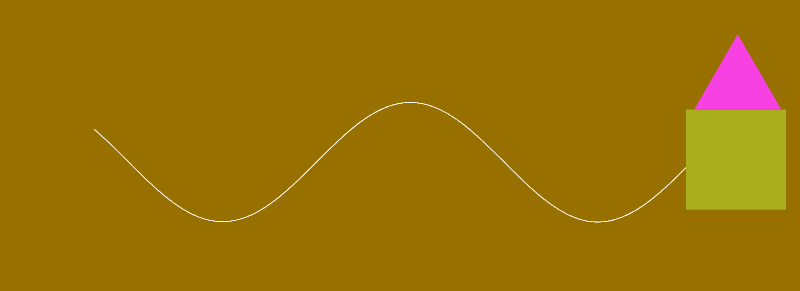
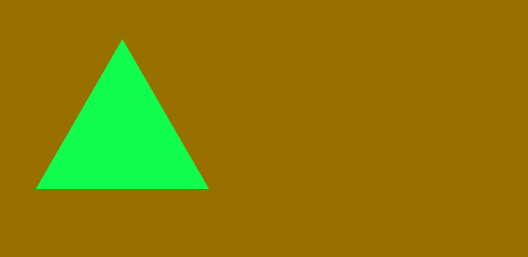
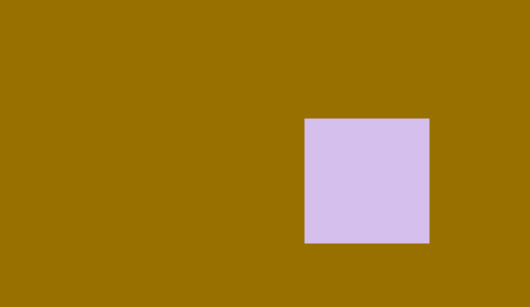


Рисунок 4 – Рух композиту в автоматичному режимі

Зміна кольору під впливом іншого об'єкта представлена на рисунку 5.





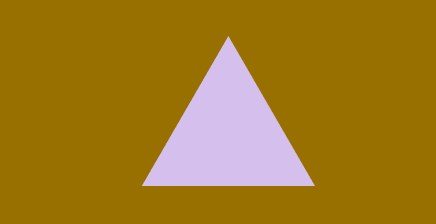
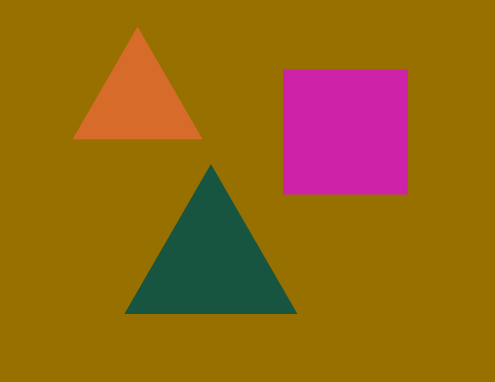
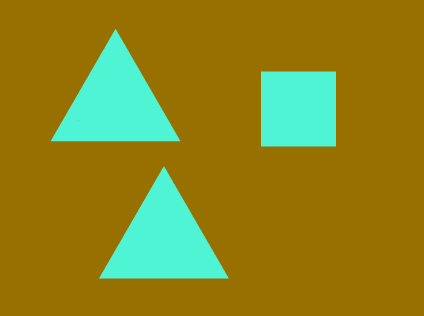


Рисунок 5 – Зміна кольору під впливом іншого об'єкта

Зміна кольору за командою та відновлення початкового стану представлені на рисунку 6.





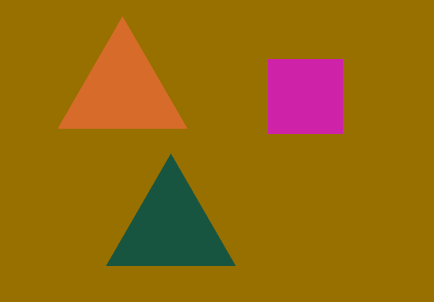
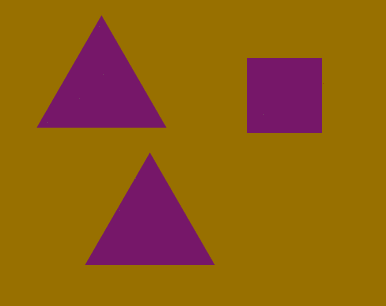
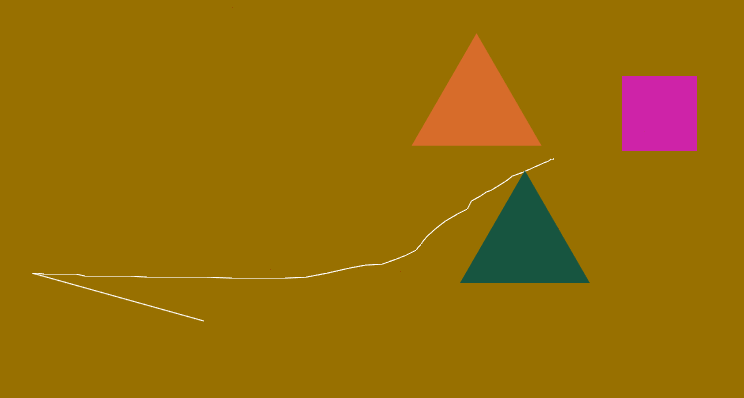


Рисунок 6 – Зміна кольору за командою та відновлення початкового стану

Рух, зміна кольору та зміна розміру агрегованої фігури (патерн Composite) показані на рисунку 7.



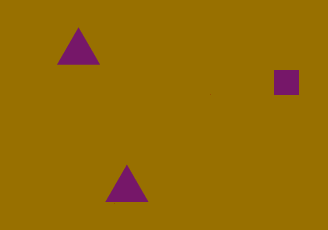


Рисунок 7 – Рух, зміна кольору та зміна розміру агрегованої фігури

Формування композиту з простих фігур та створення його клону зі зміненим кольором показане на рисунку 8.

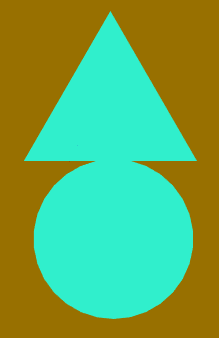
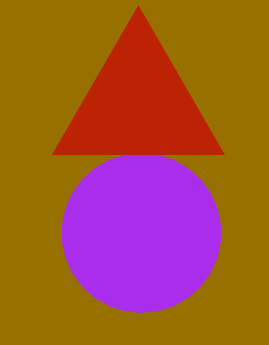


Рисунок 8 – Композит та його клон з іншим кольором

Формування нового композиту з двох композитів показане на рисунку 9.



Рисунок 9 – Формування нового композиту з двох композитів

Збережено стан композиту представленого на рисунку 6.10 у файл, змінено стан композиту на сцені, змінений композит представлено на рисунку 11.

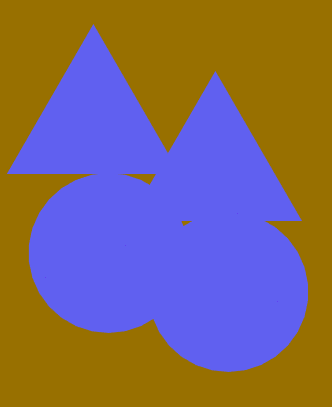


Рисунок 10 – Композит перед зберіганням



Рисунок 10 – Змінений композит

Вивантажений із файлу композит показано на рисунку 11.

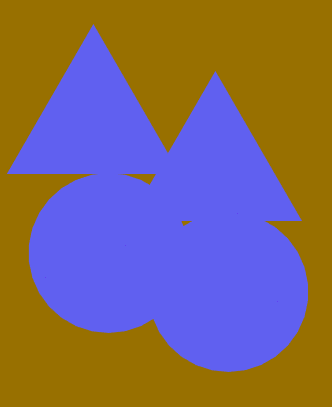


Рисунок 11 – Вивантажений із файлу композит

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

# Прототип (Refactoring Guru) URL: <https://refactoring.guru/ru/design-patterns/prototype> (дата звернення: 03.03.2023).

# Прототип (Refactoring Guru) URL: <https://martalex.gitbooks.io/gameprogrammingpatterns/content/chapter-2/2.4-prototype.html> (дата звернення: 03.03.2023).

# Компонувальник(шаблон проектування) (Wikipedia) URL: <https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)#%D0%9F%D1%80%D0%B8%D0%BC%D0%B5%D1%80_%D0%BD%D0%B0C> (дата звернення: 03.03.2023).

# Одинак (Refactoring Guru) URL: <https://refactoring.guru/ru/design-patterns/singleton> (дата зверення: 03.03.2023).

# [Infinitive1](https://habr.com/ru/users/Infinitive1/). Пояснюю Pattern Memento (Знімок) URL: <https://habr.com/ru/post/689948/> (дата звернення: 03.03.2023).

# Vishal Chovatiya. Prototype Design Pattern in Modern C++ URL: <https://dzone.com/articles/prototype-design-pattern-in-modern-c> (дата звернення: 03.03.2023).

# Паттерн Компонувальник на C++ (TECH GEEK) URL: <https://tech-geek.ru/c-composer-pattern/> (дата звернення: 03.03.2023).

# The Composite Pattern (Modernes C++) URL: <https://www.modernescpp.com/index.php/the-composite-pattern> (дата звернення: 03.03.2023).