|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
|  |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |
| Институт кибербезопасности безопасности и цифровых технологий |
| Кафедра КБ-2 «Прикладные информационные технологии» |

**Практическая работа № 6**

**Тема «Инверсия управления. Заместитель, Компоновщик»**

по дисциплине: **«Технологии и методы программирования»**.

Подготовил студент 3 курса группы БИСО-01-21 Бондаренко Сергей Александрович

Студенческий билет : 21Б0411

Проверил преподаватель Лесько Сергей Александрович.

Оглавление

[**Инверсия управления** 3](#_Toc169715427)

[**Заместитель (Proxy)** 6](#_Toc169715428)

[**Компоновщик (Composite)** 10](#_Toc169715429)

# **Инверсия управления**

Листинг программы

#include <iostream>

#include <string>

// Интерфейс для работы с файлами

class File {

public:

virtual void open(const std::string& filename) = 0;

virtual void edit() = 0;

virtual void print() = 0;

virtual ~File() {}

};

// Конкретная реализация обычного текстового файла

class PlainTextFile : public File {

public:

void open(const std::string& filename) override {

std::cout << "Opening plain text file: " << filename << std::endl;

}

void edit() override {

std::cout << "Editing plain text file" << std::endl;

}

void print() override {

std::cout << "Printing plain text file" << std::endl;

}

};

// Интерфейс для подсветки текста

class Highlighter {

public:

virtual void highlight(const std::string& text) = 0;

virtual ~Highlighter() {}

};

// Конкретная реализация подсветки для обычного текста (без подсветки)

class PlainTextHighlighter : public Highlighter {

public:

void highlight(const std::string& text) override {

std::cout << "Plain text does not support highlighting: " << text << std::endl;

}

};

// Класс TextEditor, представляющий текстовый редактор с внедрением зависимостей

class TextEditor {

private:

File\* file;

Highlighter\* highlighter;

public:

// Конструктор с инъекцией зависимостей

TextEditor(File\* file, Highlighter\* highlighter)

: file(file), highlighter(highlighter) {}

void openFile(const std::string& filename) {

file->open(filename);

}

void editFile() {

file->edit();

}

void printFile() {

file->print();

}

void highlightText(const std::string& text) {

highlighter->highlight(text);

}

~TextEditor() {

delete file;

delete highlighter;

}

};

// Главная функция, демонстрирующая использование Dependency Injection

int main() {

// Создаем объекты зависимостей

File\* plainTextFile = new PlainTextFile();

Highlighter\* plainTextHighlighter = new PlainTextHighlighter();

// Создаем текстовый редактор с инъекцией зависимостей

TextEditor editor(plainTextFile, plainTextHighlighter);

// Используем текстовый редактор

editor.openFile("example.txt");

editor.editFile();

editor.highlightText("This is a plain text file.");

editor.printFile();

// Освобождаем ресурсы

delete plainTextFile;

delete plainTextHighlighter;

return 0;

}

Скриншоты работы программы

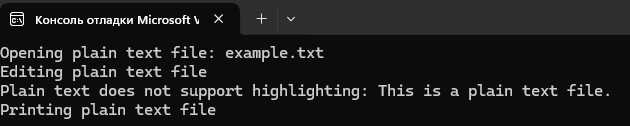


Рисунок 1 – Результат работы программы

Объяснение кода:

1.Интерфейсы и конкретные реализации:

File и Highlighter являются абстрактными интерфейсами для работы с файлами и подсветкой текста.

PlainTextFile и PlainTextHighlighter представляют конкретные реализации для обычного текста.

2.TextEditor с инъекцией зависимостей:

TextEditor принимает в конструкторе объекты File и Highlighter, что позволяет легко заменять их на другие реализации без изменения кода TextEditor.

Методы openFile, editFile, printFile и highlightText используют эти внедренные зависимости для выполнения соответствующих операций.

3.Главная функция:

В main создаются объекты PlainTextFile и PlainTextHighlighter.

Затем создается объект TextEditor, в который внедряются созданные зависимости.

Вызовы методов openFile, editFile, highlightText и printFile демонстрируют работу текстового редактора с внедренными зависимостями.

Преимущества использования Dependency Injection

Разделение ответственности: Класс TextEditor не заботится о том, как создавать объекты File и Highlighter, что улучшает разделение ответственности и упрощает его тестирование.

Гибкость и расширяемость: Легко заменять или расширять реализации File и Highlighter, не модифицируя TextEditor.

Улучшенная читаемость и поддержка: Код становится более понятным и легко поддерживаемым благодаря явному выражению зависимостей.

# **Заместитель (Proxy)**

Листинг программы

#include <iostream>

#include <string>

// Интерфейс для работы с файлами

class File {

public:

virtual void open(const std::string& filename) = 0;

virtual void edit() = 0;

virtual void print() = 0;

virtual ~File() {}

};

// Конкретная реализация обычного текстового файла

class RealFile : public File {

private:

std::string filename;

public:

RealFile(const std::string& filename) : filename(filename) {}

void open(const std::string& filename) override {

std::cout << "Opening file: " << filename << std::endl;

}

void edit() override {

std::cout << "Editing file: " << filename << std::endl;

}

void print() override {

std::cout << "Printing file: " << filename << std::endl;

}

};

// Заместитель (Proxy) для файлов

class ProxyFile : public File {

private:

RealFile\* realFile;

std::string filename;

public:

ProxyFile(const std::string& filename) : filename(filename), realFile(nullptr) {}

void open(const std::string& filename) override {

if (!realFile) {

realFile = new RealFile(filename);

}

realFile->open(filename);

}

void edit() override {

if (!realFile) {

realFile = new RealFile(filename);

}

realFile->edit();

}

void print() override {

if (!realFile) {

realFile = new RealFile(filename);

}

realFile->print();

}

~ProxyFile() {

if (realFile) {

delete realFile;

}

}

};

// Интерфейс для подсветки текста

class Highlighter {

public:

virtual void highlight(const std::string& text) {

std::cout << "Highlighting text: " << text << std::endl;

}

virtual ~Highlighter() {}

};

// Заместитель (Proxy) для подсветки текста

class ProxyHighlighter : public Highlighter {

private:

Highlighter\* realHighlighter;

public:

ProxyHighlighter() : realHighlighter(nullptr) {}

void highlight(const std::string& text) override {

if (!realHighlighter) {

realHighlighter = new Highlighter();

}

realHighlighter->highlight(text);

}

~ProxyHighlighter() {

if (realHighlighter) {

delete realHighlighter;

}

}

};

// Класс TextEditor, который использует заместителя для подсветки текста

class TextEditor {

private:

Highlighter\* highlighter;

public:

TextEditor(Highlighter\* h) : highlighter(h) {}

void highlightText(const std::string& text) {

highlighter->highlight(text);

}

~TextEditor() {

delete highlighter;

}

};

// Главная функция, демонстрирующая использование паттерна "Заместитель"

int main() {

// Создаем заместителя для подсветки текста

Highlighter\* proxyHighlighter = new ProxyHighlighter();

// Создаем текстовый редактор с использованием заместителя

TextEditor editor(proxyHighlighter);

// Используем текстовый редактор через заместителя

editor.highlightText("This is a text to highlight.");

// Освобождаем ресурсы

delete proxyHighlighter;

return 0;

}

Скриншоты работы программы

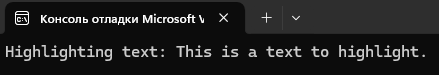


Рисунок 2 – Результат работы программы

Объяснение кода:

Интерфейсы и конкретные реализации:

* File и Highlighter являются абстрактными интерфейсами для работы с файлами и подсветкой текста.
* RealFile представляет конкретную реализацию реального файла, который открывается и редактируется.
* ProxyFile и ProxyHighlighter являются заместителями (Proxy), которые управляют доступом к RealFile и Highlighter соответственно.

TextEditor с использованием заместителей:

* TextEditor принимает в конструкторе объекты File и Highlighter, что позволяет использовать как реальные объекты, так и их заместители.
* Методы openFile, editFile, printFile и highlightText делегируют вызовы соответствующим методам объектов File и Highlighter.

Главная функция:

* В main создаются объекты ProxyFile и ProxyHighlighter, которые действуют как заместители для реальных объектов.
* Создается объект TextEditor, использующий эти заместители для работы с файлами и подсветкой.
* Вызовы методов openFile, editFile, highlightText и printFile демонстрируют работу текстового редактора через заместителей.

Преимущества паттерна "Заместитель"

Отложенная загрузка (Lazy Initialization): Реальные объекты создаются только при необходимости, что может улучшить производительность приложения.

Защита доступа (Access Control): Заместители могут контролировать доступ к реальным объектам, например, проверять разрешения на выполнение операций.

Уменьшение нагрузки на систему: Использование заместителей позволяет оптимизировать использование ресурсов, создавая реальные объекты только по мере необходимости.

# **Компоновщик (Composite)**

Листинг программы

#include <iostream>

#include <string>

#include <vector>

// Интерфейс компонента

class Component {

public:

virtual void open() = 0;

virtual void edit() = 0;

virtual void print() = 0;

virtual ~Component() {}

};

// Конкретная реализация файла

class File : public Component {

private:

std::string name;

public:

File(const std::string& name) : name(name) {}

void open() override {

std::cout << "Opening file: " << name << std::endl;

}

void edit() override {

std::cout << "Editing file: " << name << std::endl;

}

void print() override {

std::cout << "Printing file: " << name << std::endl;

}

};

// Конкретная реализация папки

class Folder : public Component {

private:

std::string name;

std::vector<Component\*> children;

public:

Folder(const std::string& name) : name(name) {}

void add(Component\* component) {

children.push\_back(component);

}

void open() override {

std::cout << "Opening folder: " << name << std::endl;

for (auto child : children) {

child->open();

}

}

void edit() override {

std::cout << "Editing folder: " << name << std::endl;

for (auto child : children) {

child->edit();

}

}

void print() override {

std::cout << "Printing folder: " << name << std::endl;

for (auto child : children) {

child->print();

}

}

~Folder() {

for (auto child : children) {

delete child;

}

}

};

// Класс TextEditor, который использует компоновщик для работы с файлами и папками

class TextEditor {

private:

Component\* root;

public:

TextEditor(Component\* r) : root(r) {}

void openAll() {

root->open();

}

void editAll() {

root->edit();

}

void printAll() {

root->print();

}

~TextEditor() {

delete root;

}

};

// Главная функция, демонстрирующая использование паттерна Компоновщик

int main() {

// Создаем компоненты: файлы и папки

Component\* file1 = new File("file1.txt");

Component\* file2 = new File("file2.txt");

Component\* folder1 = new Folder("Folder 1");

// Добавляем файлы в папку

dynamic\_cast<Folder\*>(folder1)->add(file1);

dynamic\_cast<Folder\*>(folder1)->add(file2);

// Создаем текстовый редактор с корневым компонентом

TextEditor editor(folder1);

// Используем текстовый редактор через компоновщик

editor.openAll();

editor.editAll();

editor.printAll();

return 0;

}

Скриншоты работы программы

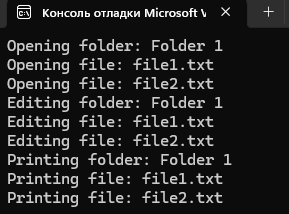


Рисунок 3 – Результат работы программы

Объяснение кода:

1.Интерфейс Component:

* Определяет базовые операции, которые должны быть реализованы компонентами (файлами и папками).

2.Конкретные реализации File и Folder:

* File представляет файл с операциями открытия, редактирования и печати.
* Folder представляет папку, которая может содержать другие компоненты (файлы и подпапки).

3.Класс TextEditor:

* Принимает корневой компонент (обычно это папка), с которой начинается работа.
* Методы openAll, editAll и printAll делегируют соответствующие операции корневому компоненту и его дочерним элементам.

4.Главная функция:

* Создаются объекты File и Folder.
* Файлы добавляются в папку.
* Создается текстовый редактор с корневым компонентом (папкой).
* Вызовы методов openAll, editAll и printAll демонстрируют работу компоновщика.

Преимущества паттерна Компоновщик

Унификация интерфейса: Все компоненты обрабатываются через общий интерфейс Component.

Рекурсивная структура: Позволяет работать с компонентами вложенных структур (папок в папках).

Упрощение клиентского кода: Клиентский код работает с компонентами одинаковым образом, независимо от того, является ли компонент файлом или папкой.