**Кросплатформенне програмування**

## **Лабораторна робота №4**

«Чистий код. Шаблони проектування»

*Виконав Заяць Ілля, група ІПЗ-41*

**Мета:**

Розширити програму, яка була створена у лабораторній роботі №1. Імплементувати довільним чином по одному шаблону з основних груп (обирати 1 з породжуючих, 1 із структурних, 1 із поведінкових).

Хід роботи

1.)- Імплементувати у програму породжуючий шаблон Builder.

2.)- Імплементувати у програму структурний шаблон Composite.

3.)- Імплементувати у програму поведінковий шаблон Observer.

4.)- Продемонструвати роботу всієї оновленої програми.

## *Завдання 1*

1.)- Імплементувати у програму породжуючий шаблон Builder. Шаблон Builder дозволяє створювати об'єкти з великою кількістю параметрів через поетапне налаштування, що робить код більш читабельним. У нашій системі, можна використати цей шаблон для зручного створення книг (наприклад, книг з різними параметрами).

2.)- Код

class BookBuilder {

private var title: String = ""

private var author: String = ""

private var isbn: String = ""

private var available: Boolean = true

private var borrower: Member? = null

fun setTitle(title: String): BookBuilder {

this.title = title

return this

}

fun setAuthor(author: String): BookBuilder {

this.author = author

return this

}

fun setIsbn(isbn: String): BookBuilder {

this.isbn = isbn

return this

}

fun setAvailable(available: Boolean): BookBuilder {

this.available = available

return this

}

fun setBorrower(borrower: Member?): BookBuilder {

this.borrower = borrower

return this

}

fun build(): Book {

return Book(title, author, isbn, available, borrower)

}

}

## *Завдання 2*

1.)- Імплементувати у програму структурний шаблон Composite. Шаблон Composite дозволяє об'єднувати об'єкти в структуру дерева, де окремі об'єкти та їхні групи можуть бути оброблені однаково. Ми можемо використати цей шаблон для управління бібліотекою, де кожен елемент (книга, розділ бібліотеки) може бути частиною більш складної структури. Варто зазначити, що в самій програмі немає безпосередньо класу з такої назвою, а сам клас називається LibraryComponent.

2.)- Код

interface LibraryComponent {

fun getBooksCount(): Int

fun getAvailableBooksCount(): Int

}

class Library : LibraryComponent {

val books: MutableList<Book> = mutableListOf()

val members: MutableList<Member> = mutableListOf()

fun add\_book(book: Book) {

books.add(book)

}

fun remove\_book(book: Book) {

books.remove(book)

}

fun find\_book\_by\_title(title: String): Book? {

return books.find { it.title == title }

}

fun register\_member(member: Member) {

members.add(member)

}

fun unregister\_member(member: Member) {

members.remove(member)

}

fun borrow\_book(member: Member, title: String) {

val book = find\_book\_by\_title(title)

if (book != null && book.isAvailable() && member.can\_borrow()) {

book.borrow(member)

} else {

println("Cannot borrow book '$title'.")

}

}

fun return\_book(member: Member, title: String) {

val book = find\_book\_by\_title(title)

if (book != null) {

book.returnBook()

} else {

println("Book '$title' not found.")

}

}

override fun getBooksCount(): Int = books.size

override fun getAvailableBooksCount(): Int = books.count { it.isAvailable() }

fun get\_books\_count\_by\_title(title: String): Int {

return books.count { it.title == title }

}

fun get\_books\_count\_by\_author(author: String): Int {

return books.count { it.author == author }

}

}

## *Завдання 3*

1.)- Імплементувати у програму поведінковий шаблон Observer. Шаблон Observer дозволяє об'єктам спостерігати за іншими об'єктами і реагувати на зміни стану. В нашому випадку, можна використовувати цей шаблон для інформування користувачів (читачів) про те, що книга стала доступною, якщо її повернув інший користувач.

2.)- Код

interface Observer {

fun update(book: Book)

}

class Member(

val name: String,

val memberId: String,

val maxBooks: Int = 3

) : Observer {

val borrowedBooks: MutableList<Book> = mutableListOf()

fun borrow\_book(book: Book) {

if (can\_borrow()) {

borrowedBooks.add(book)

} else {

println("$name can't borrow more books.")

}

}

fun return\_book(book: Book) {

borrowedBooks.remove(book)

}

fun can\_borrow(): Boolean = borrowedBooks.size < maxBooks

override fun update(book: Book) {

if (book.isAvailable()) {

println("The book '${book.title}' is now available.")

}

}

}

## *Завдання 4*

1.)- Вся оновлена програма. Використовуються можливості всіх нових класів. Трохи змінений основний код, щоб продемонструвати роботу нових класів.

2.)- Код

// Book.kt

class Book(

val title: String,

val author: String,

val isbn: String,

var available: Boolean = true,

var borrower: Member? = null

) {

private val observers: MutableList<Observer> = mutableListOf()

fun isAvailable(): Boolean = available

fun borrow(member: Member) {

if (available) {

available = false

borrower = member

member.borrow\_book(this)

} else {

println("Book is not available.")

}

}

fun returnBook() {

available = true

borrower?.return\_book(this)

borrower = null

notifyObservers()

}

fun addObserver(observer: Observer) {

observers.add(observer)

}

fun notifyObservers() {

for (observer in observers) {

observer.update(this)

}

}

}

// Member.kt

interface Observer {

fun update(book: Book)

}

class Member(

val name: String,

val memberId: String,

val maxBooks: Int = 3

) : Observer {

val borrowedBooks: MutableList<Book> = mutableListOf()

fun borrow\_book(book: Book) {

if (can\_borrow()) {

borrowedBooks.add(book)

} else {

println("$name can't borrow more books.")

}

}

fun return\_book(book: Book) {

borrowedBooks.remove(book)

}

fun can\_borrow(): Boolean = borrowedBooks.size < maxBooks

override fun update(book: Book) {

if (book.isAvailable()) {

println("The book '${book.title}' is now available.")

}

}

}

// Library.kt

interface LibraryComponent {

fun getBooksCount(): Int

fun getAvailableBooksCount(): Int

}

class Library : LibraryComponent {

val books: MutableList<Book> = mutableListOf()

val members: MutableList<Member> = mutableListOf()

fun add\_book(book: Book) {

books.add(book)

}

fun remove\_book(book: Book) {

books.remove(book)

}

fun find\_book\_by\_title(title: String): Book? {

return books.find { it.title == title }

}

fun register\_member(member: Member) {

members.add(member)

}

fun unregister\_member(member: Member) {

members.remove(member)

}

fun borrow\_book(member: Member, title: String) {

val book = find\_book\_by\_title(title)

if (book != null && book.isAvailable() && member.can\_borrow()) {

book.borrow(member)

} else {

println("Cannot borrow book '$title'.")

}

}

fun return\_book(member: Member, title: String) {

val book = find\_book\_by\_title(title)

if (book != null) {

book.returnBook()

} else {

println("Book '$title' not found.")

}

}

override fun getBooksCount(): Int = books.size

override fun getAvailableBooksCount(): Int = books.count { it.isAvailable() }

fun get\_books\_count\_by\_title(title: String): Int {

return books.count { it.title == title }

}

fun get\_books\_count\_by\_author(author: String): Int {

return books.count { it.author == author }

}

}

// Librarian.kt

class Librarian(

val name: String,

val employeeId: String

) {

fun add\_book\_to\_library(book: Book, library: Library) {

library.add\_book(book)

}

fun remove\_book\_from\_library(book: Book, library: Library) {

library.remove\_book(book)

}

fun register\_new\_member(member: Member, library: Library) {

library.register\_member(member)

}

}

// BookBuilder.kt

class BookBuilder {

private var title: String = ""

private var author: String = ""

private var isbn: String = ""

private var available: Boolean = true

private var borrower: Member? = null

fun setTitle(title: String): BookBuilder {

this.title = title

return this

}

fun setAuthor(author: String): BookBuilder {

this.author = author

return this

}

fun setIsbn(isbn: String): BookBuilder {

this.isbn = isbn

return this

}

fun setAvailable(available: Boolean): BookBuilder {

this.available = available

return this

}

fun setBorrower(borrower: Member?): BookBuilder {

this.borrower = borrower

return this

}

fun build(): Book {

return Book(title, author, isbn, available, borrower)

}

}

// Main.kt

fun main() {

val library = Library()

val librarian = Librarian("John Doe", "L123")

val member = Member("Alice", "M001")

librarian.register\_new\_member(member, library)

// Використовуємо Builder для створення книг

val book1 = BookBuilder()

.setTitle("Kotlin Programming")

.setAuthor("JetBrains")

.setIsbn("123-456-789")

.build()

val book2 = BookBuilder()

.setTitle("Advanced Kotlin")

.setAuthor("JetBrains")

.setIsbn("987-654-321")

.build()

librarian.add\_book\_to\_library(book1, library)

librarian.add\_book\_to\_library(book2, library)

// Підписуємо користувача на спостереження за книгами

book1.addObserver(member)

book2.addObserver(member)

// Alice borrows a book

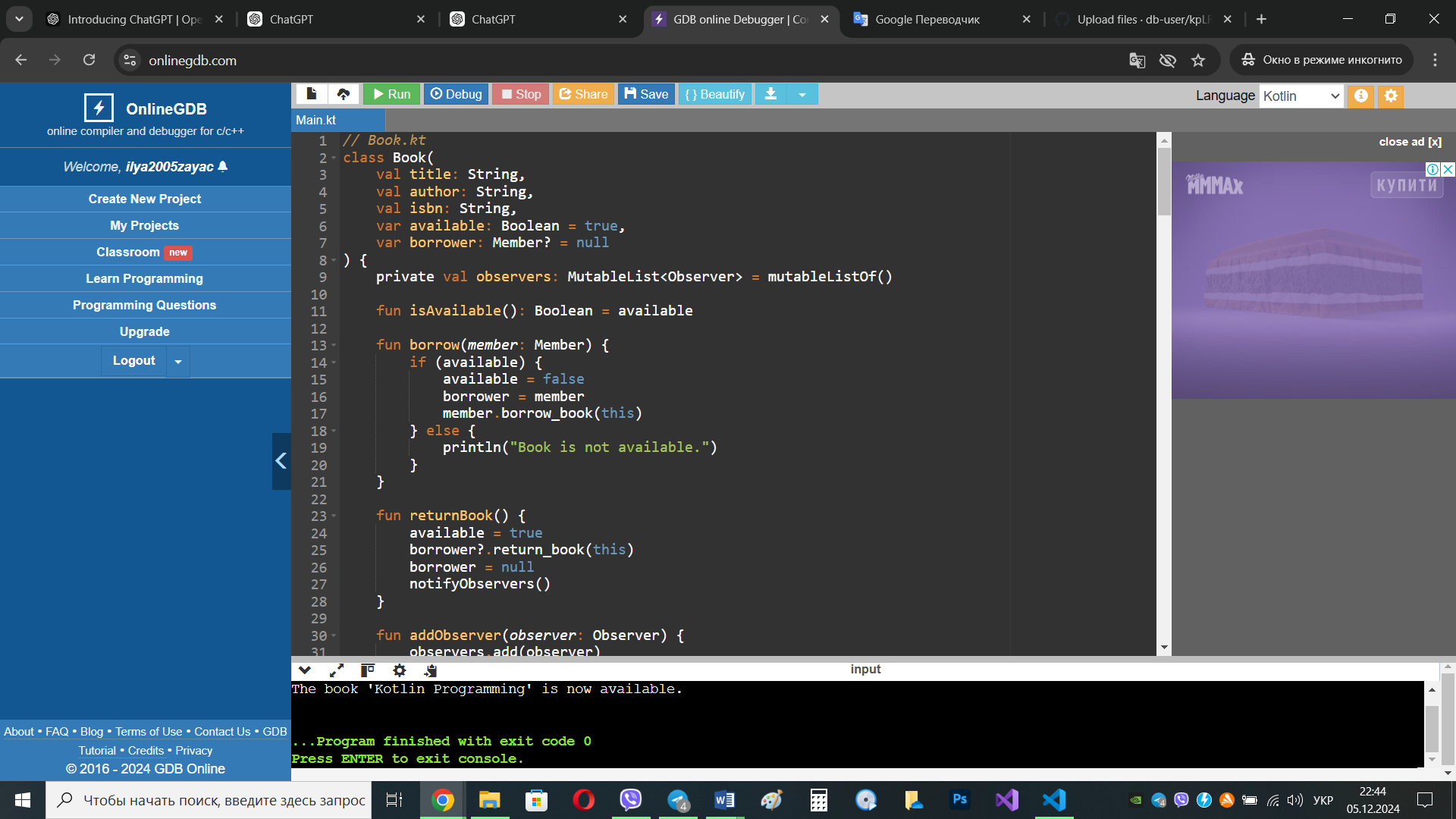
library.borrow\_book(member, "Kotlin Programming")

// Alice returns the book, спостерігачі отримують сповіщення

library.return\_book(member, "Kotlin Programming")

}

3.)- Приклад виконання (з онлайн-компілятора onlinegdb):



## **Висновок:**

Builder використовується для зручного створення книг з різними властивостями.

Composite дозволяє групувати книги за категоріями (розділами), що полегшує управління бібліотекою.

Observer дозволяє членам бібліотеки отримувати сповіщення, коли книга, яку вони хочуть позичити, стає доступною.

Ці зміни покращують гнучкість програми і дозволяють легше додавати нові функціональності в майбутньому.