Міністерство освіти і науки України
Національний технічний університет
України
“Київський політехнічний інститут імені Ігоря
Сікорського” Факультет інформатики та обчислювальної
техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконав:
студент групи ІА-з31
Курбатов Кіріл Андрійович

Перевірив:
Амонс Олександр Анатолійович

Київ 2025

Мета: Вивчити структуру шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» та навчитися застосовувати їх в реалізації
програмної системи.

Завдання

• Ознайомитись з короткими теоретичними відомостями.
• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
• Реалізувати один з розглянутих шаблонів за обраною темою.
• Реалізувати не менше 3-х класів відповідно до обраної теми.
• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону

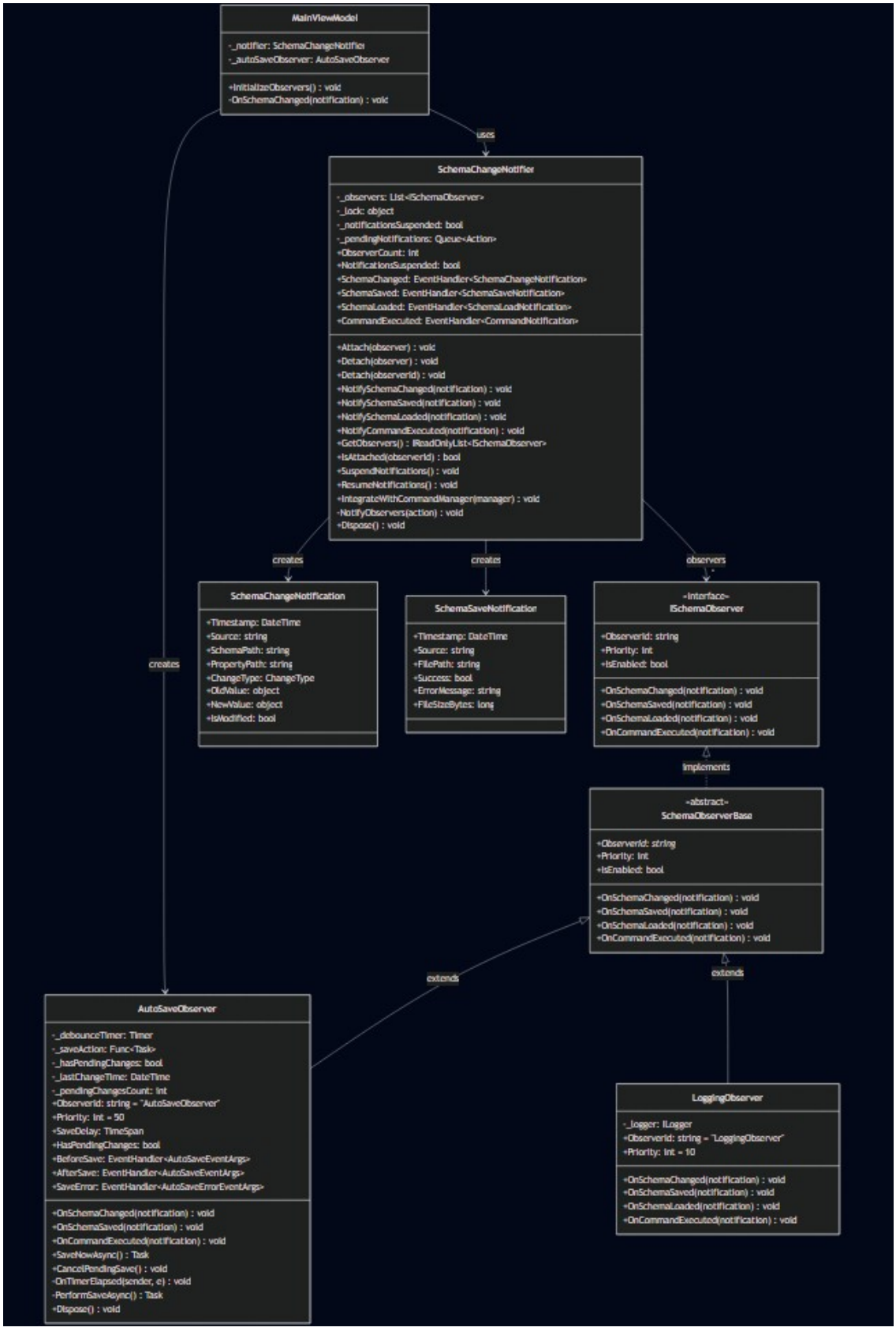

Рис. 1 — Діаграма класів патерну Observer для системи сповіщень

Реалізація патерну Observer

```
using JsonTool.Core.Command;

namespace JsonTool.Core.Observer;

public interface ISchemaObserver
{
    string ObserverId { get; }
    int Priority { get; }
    bool IsEnabled { get; set; }

    void OnSchemaChanged(SchemaChangeNotification notification);
    void OnSchemaSaved(SchemaSaveNotification notification);
    void OnSchemaLoaded(SchemaLoadNotification notification);
    void OnCommandExecuted(CommandNotification notification);
}

public abstract class NotificationBase
{
    public DateTime Timestamp { get; } = DateTime.Now;
    public string? Source { get; set; }
}

public class SchemaChangeNotification : NotificationBase
{
    public string SchemaPath { get; set; } = string.Empty;
    public string? PropertyPath { get; set; }
    public ChangeType ChangeType { get; set; }
    public object? OldValue { get; set; }
    public object? NewValue { get; set; }
    public bool IsModified { get; set; }
}

public class SchemaSaveNotification : NotificationBase
{
    public string FilePath { get; set; } = string.Empty;
    public bool Success { get; set; }
    public string? ErrorMessage { get; set; }
    public long FileSizeBytes { get; set; }
}

public class SchemaLoadNotification : NotificationBase
{
    public string FilePath { get; set; } = string.Empty;
    public bool Success { get; set; }
    public string? ErrorMessage { get; set; }
    public int PropertiesCount { get; set; }
}

public class CommandNotification : NotificationBase
{
    public ISchemaCommand Command { get; set; } = null!;
    public CommandActionType ActionType { get; set; }
    public bool Success { get; set; }
    public string? ErrorMessage { get; set; }
}
```

```
public enum ChangeType
{
    PropertyAdded,
    PropertyUpdated,
    PropertyDeleted,
    MetadataUpdated,
    SchemaReloaded,
    ContentChanged
}

public enum CommandActionType
{
    Execute,
    Undo,
    Redo
}

public abstract class SchemaObserverBase : ISchemaObserver
{
    public abstract string ObserverId { get; }
    public virtual int Priority => 100;
    public bool IsEnabled { get; set; } = true;

    public virtual void OnSchemaChanged(SchemaChangeNotification notification) { }
    public virtual void OnSchemaSaved(SchemaSaveNotification notification) { }
    public virtual void OnSchemaLoaded(SchemaLoadNotification notification) { }
    public virtual void OnCommandExecuted(CommandNotification notification) { }
}
```

```
using JsonTool.Core.Command;

namespace JsonTool.Core.Observer;

public class SchemaChangeNotifier : IDisposable
{
    private readonly List<ISchemaObserver> _observers = new();
    private readonly object _lock = new();
    private bool _notificationsSuspended;
    private readonly Queue<Action> _pendingNotifications = new();
    private bool _disposed;

    public event EventHandler<SchemaChangeNotification>? SchemaChanged;
    public event EventHandler<SchemaSaveNotification>? SchemaSaved;
    public event EventHandler<SchemaLoadNotification>? SchemaLoaded;
    public event EventHandler<CommandNotification>? CommandExecuted;

    public int ObserverCount
    {
        get
        {
            lock (_lock) { return _observers.Count; }
        }
    }

    public bool NotificationsSuspended => _notificationsSuspended;

    public void Attach(ISchemaObserver observer)
    {
        if (observer == null) throw new ArgumentNullException(nameof(observer));

        lock (_lock)
        {
            if (_observers.Any(o => o.ObserverId == observer.ObserverId))
            {
                throw new InvalidOperationException(
                    $"Observer '{observer.ObserverId}' is already attached");
            }

            _observers.Add(observer);
            _observers.Sort((a, b) => a.Priority.CompareTo(b.Priority));
        }
    }

    public void Detach(ISchemaObserver observer)
    {
        if (observer == null) throw new ArgumentNullException(nameof(observer));
        Detach(observer.ObserverId);
    }

    public void Detach(string observerId)
    {
        lock (_lock)
        {
            var observer = _observers.FirstOrDefault(o => o.ObserverId == observerId);
            if (observer != null)
            {
                _observers.Remove(observer);
            }
        }
    }
```

```
public void ResumeNotifications()
{
    _notificationsSuspended = false;
    Queue<Action> pending;
    lock (_lock)
    {
        pending = new Queue<Action>(_pendingNotifications);
        _pendingNotifications.Clear();
    }

    while (pending.Count > 0)
    {
        var action = pending.Dequeue();
        action();
    }
}

public void IntegrateWithCommandManager(SchemaCommandManager commandManager)
{
    commandManager.CommandExecuted += (s, e) =>
    {
        NotifyCommandExecuted(new CommandNotification
        {
            Command = e.Command,
            ActionType = CommandActionType.Execute,
            Success = true,
            Source = "CommandManager"
        });
    };

    commandManager.CommandUndone += (s, e) =>
    {
        NotifyCommandExecuted(new CommandNotification
        {
            Command = e.Command,
            ActionType = CommandActionType.Undo,
            Success = true,
            Source = "CommandManager"
        });
    };

    commandManager.CommandRedone += (s, e) =>
    {
        NotifyCommandExecuted(new CommandNotification
        {
            Command = e.Command,
            ActionType = CommandActionType.Redo,
            Success = true,
            Source = "CommandManager"
        });
    };
}
```

```
using JsonTool.Core.Command;

namespace JsonTool.Core.Observer;

public class SchemaChangeNotifier : IDisposable
{
    private readonly List<ISchemaObserver> _observers = new();
    private readonly object _lock = new();
    private bool _notificationsSuspended;
    private readonly Queue<Action> _pendingNotifications = new();
    private bool _disposed;

    public event EventHandler<SchemaChangeNotification>? SchemaChanged;
    public event EventHandler<SchemaSaveNotification>? SchemaSaved;
    public event EventHandler<SchemaLoadNotification>? SchemaLoaded;
    public event EventHandler<CommandNotification>? CommandExecuted;

    public int ObserverCount
    {
        get
        {
            lock (_lock) { return _observers.Count; }
        }
    }

    public bool NotificationsSuspended => _notificationsSuspended;

    public void Attach(ISchemaObserver observer)
    {
        if (observer == null) throw new ArgumentNullException(nameof(observer));

        lock (_lock)
        {
            if (_observers.Any(o => o.ObserverId == observer.ObserverId))
            {
                throw new InvalidOperationException(
                    $"Observer '{observer.ObserverId}' is already attached");
            }

            _observers.Add(observer);
            _observers.Sort((a, b) => a.Priority.CompareTo(b.Priority));
        }
    }

    public void Detach(ISchemaObserver observer)
    {
        if (observer == null) throw new ArgumentNullException(nameof(observer));
        Detach(observer.ObserverId);
    }

    public void Detach(string observerId)
    {
        lock (_lock)
        {
            var observer = _observers.FirstOrDefault(o => o.ObserverId == observerId);
            if (observer != null)
            {
                _observers.Remove(observer);
            }
        }
    }
```

```
public void ResumeNotifications()
{
    _notificationsSuspended = false;
    Queue<Action> pending;
    lock (_lock)
    {
        pending = new Queue<Action>(_pendingNotifications);
        _pendingNotifications.Clear();
    }

    while (pending.Count > 0)
    {
        var action = pending.Dequeue();
        action();
    }
}

public void IntegrateWithCommandManager(SchemaCommandManager commandManager)
{
    commandManager.CommandExecuted += (s, e) =>
    {
        NotifyCommandExecuted(new CommandNotification
        {
            Command = e.Command,
            ActionType = CommandActionType.Execute,
            Success = true,
            Source = "CommandManager"
        });
    };

    commandManager.CommandUndone += (s, e) =>
    {
        NotifyCommandExecuted(new CommandNotification
        {
            Command = e.Command,
            ActionType = CommandActionType.Undo,
            Success = true,
            Source = "CommandManager"
        });
    };

    commandManager.CommandRedone += (s, e) =>
    {
        NotifyCommandExecuted(new CommandNotification
        {
            Command = e.Command,
            ActionType = CommandActionType.Redo,
            Success = true,
            Source = "CommandManager"
        });
    };
}
```