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

Лабораторна робота №6

із дисципліни «*Технології розробки програмного забезпечення*»

Тема: «Патерни проектування»

Виконав:
Студент групи ІА-31
Клим’юк В.Л.

Перевірив:
Мягкий М.Ю.

Київ-2025

Тема: Система запису на прийом до лікаря (strategy, adapter, observer, facade, visitor, client-server).
Система дозволяє пацієнтам шукати та записуватись на прийоми до лікарів, де їм виноситься висновок за результатами прийому. Також наявні функції відміни прийому та додавання відгуків про лікарів.

Репозиторій: https://github.com/StaticReadonly/kpi-trpz-labs-klim

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

Теоретичні відомості:

«Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Для цього виноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів. Хорошим прикладом використання абстрактної фабрики є ADO.NET: існує загальний клас DbProviderFactory, здатний створювати об'єкти типів DbConnection, DbDataReader, DbAdapter та ін.; існують реалізації цих фабрик і об'єктів – SqlProviderFactory, SqlConnection, SqlDataReader, SqlAdapter і так далі. Відповідно, якщо додатку необхідно працювати з різними базами даних (чи потрібна така можливість), то досить використати базові реалізації (Db.) і підставити відповідну фабрику у момент ініціалізації фабрики (Factory = new SqlProviderFactory()).

Цей шаблон передусім структурує знання про схожі об'єкти (що називаються сімействами, як класи для доступу до БД) і створює можливість взаємозаміни різних сімейств (робота з Oracle ведеться також, як і робота з SQL Server). Проте, при використанні такої схеми украй незручно розширювати фабрику – для додавання нового методу у фабрику необхідно додати його в усіх фабриках і створити відповідні класи, що створюються цим методом.

Переваги та недоліки:

+ Спрощує створення об’єктів і код стає легшим для розуміння.

+ Об’єкти створені однією фабрикою добре узгоджуються один з одним і зменшується кількість помилок взаємодії між ними.

+ Відокремлення створення об’єктів від їх використання, за рахунок чого, код стає біль структурованим.

+ Додавання нових сімейств продуктів виконується без зміни існуючого коду.

- Збільшується складність коду, особливо для простих проєктів.

- Додавання нового типу продукту є складним і вимагає змін коду в багатьох місцях.

«Фабричний метод» визначає інтерфейс для створення об'єктів певного базового типу. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон «Фабричний метод» носить ще назву «Віртуальний конструктор».

Переваги та недоліки:

+ Позбавляє клас від прив’язки до конкретних класів продуктів.

+ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.

+ Спрощує додавання нових продуктів до програми.

- Може призвести до створення великих паралельних ієрархій класів.

«Memento» використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. Об'єкт «Memento» служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту «Memento» для власних цілей, цей об'єкт є «порожнім» для кого-небудь ще. Об'єкт «Caretaker» використовується для передачі і зберігання ментo об'єктів в системі. Таким чином вдається досягти наступних цілей:

зберігання стану повністю відділяється від початкових об'єктів, що полегшує їх реалізацію;

передача об'єктів «Memento» лягає на плечі Caretaker об'єктів, що дозволяє гнучкіше управляти станами об'єктів і спростити дизайн класів початкових об'єктів;

збереження і відновлення стану реалізовані у вигляді двох простих методів і є закритими для кого-небудь ще окрім початкових об'єктів, таким чином не порушуючи інкапсуляцію.

Переваги та недоліки:

+ Не порушує інкапсуляцію вихідного об’єкта.

+ Спрощує структуру вихідного об’єкта. Не потрібно зберігати історію версій свого стану.

- Вимагає багато пам’яті, якщо клієнти дуже часто створюють знімки.

- Може спричинити додаткові витрати пам’яті, якщо об’єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.

«Observer» визначає залежність «один-до-багатьох» таким чином, що коли один об'єкт змінює власний стан, усі інші об'єкти отримують про це сповіщення і мають можливість змінити власний стан також.

Цей шаблон дуже широко поширений в шаблоні MVVM і механізмі «прив'язок» (bindings) в WPF і частково в WinForms. Інша назва шаблону – підписка/розсилка. Кожен з оглядачів власноручно підписується на зміни конкретного об'єкту, а об'єкти зобов'язані сповіщати своїх передплатників про усі свої зміни (на даний момент конкретних механізмів автоматичного сповіщення про зміну стану в .NET мовах не існує).

Переваги та недоліки:

+ Можливість паралельної та асинхронної обробки повідомлень про оновлення.

+ Спостерігачів можна добавляти та видаляти в будь-який момент часу.

+ Спостерігач і суб’єкт можуть працювати в різних потоках.

+ Реалізує принцип слабкого зв’язку між об’єктами.

- Послідовність розсилки повідомлень підписникам не підтримується.

«Decorator» призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Декоратор деяким чином «обертає» (за рахунок агрегації) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому.

Простим прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в «прокручуваному» елементі, і при необхідності з'являється полоса прокрутки. Початкові функції елементу (наприклад, рядки статусу) залишаються незмінними.

Переваги та недоліки:

+ Дозволяє мати кілька дрібних об’єктів, замість одного об’єкта «на всі випадки життя».

+ Дозволяє додавати обов’язки «на льоту».

+ Більша гнучкість, ніж у спадкування.

- Велика кількість крихітних класів.

- Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.

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.

Хід роботи:

Діаграма класів:

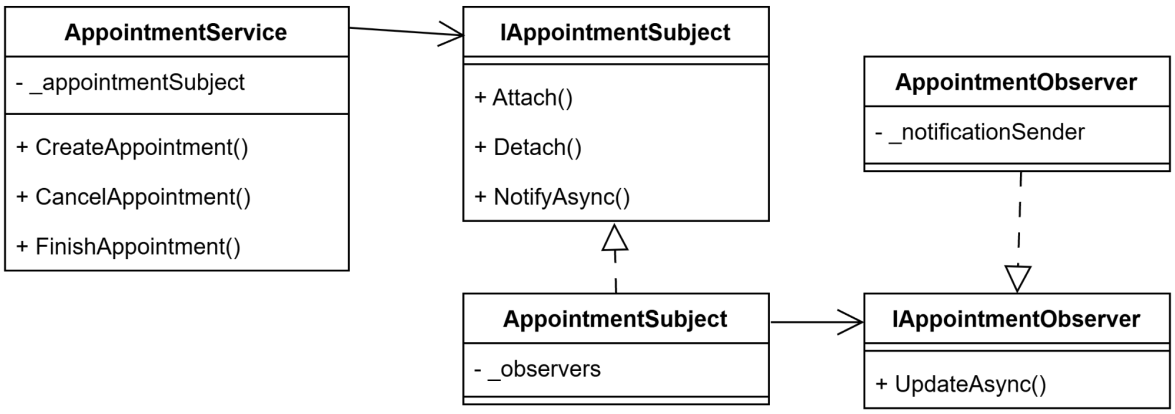


Рис.1 - Діаграма класів патерну “Observer”

На цій діаграмі:
AppointmentService - клас, який відповідає за роботу з прийомами. Використовує IAppointmentSubject для сповіщення про зміну статусу прийому.
IAppointmentSubject - інтерфейс класа суб’єкта. Містить методи Attach(), Detach() для прикріплення спостерігачів та метод NotifyAsync() для сповіщення спостерігачів.
AppointmentSubject - конкретна реалізація IAppointmentSubject. Містить список усіх підписаних спостерігачів IAppointmentObserver. Викликає у всіх метод UpdateAsync() для сповіщення про зміни статусу.
IAppointmentObserver - інтерфейс для спостерігача. Містить метод UpdateAsync() для отримання оновлень статусу від суб’єкта.
AppointmentObserver - конкретний клас-спостерігач. Містить посилання на клас для відправки повідомлень через пошту. Відправляє повідомлення по пошті, коли отримує оновлення статусу від суб’єкта.

Вихідний код класів:

```
using BookingClinic.Services.AppointmentObserver.Observer;

namespace BookingClinic.Services.AppointmentObserver.Subject
{
    Ссылок: 4
    public interface IAppointmentSubject
    {
        Ссылок: 2
        void Attach(IAppointmentObserver observer);
        Ссылок: 1
        void Detach(IAppointmentObserver observer);
        Ссылок: 5
        Task NotifyAsync(BookingClinic.Data.Entities.Appointment appointment, string email);
    }
}
```

Рис.2 - Код IAppointmentSubject

```
namespace BookingClinic.Services.AppointmentObserver.Observer
{
    public interface IAppointmentObserver
    {
        Task UpdateAsync(BookingClinic.Data.Entities.Appointment appointment, string email);
    }
}
```

Рис.3 - Код IAppointmentObserver

```
using BookingClinic.Services.NotificationService;

namespace BookingClinic.Services.AppointmentObserver.Observer
{
    Ссылок: 2
    public class AppointmentObserver : IAppointmentObserver
    {
        private readonly INotificationSender _notificationSender;

        Ссылок: 1
        public AppointmentObserver(INotificationSender notificationSender)
        {
            _notificationSender = notificationSender;
        }

        Ссылок: 2
        public async Task UpdateAsync(BookingClinic.Data.Entities.Appointment appointment, string email)
        {
            string message = string.Empty;

            if (appointment.IsCanceled)
            {
                message = $"Hello! Your appointment scheduled for {appointment.DateTime} has been canceled! " +
                    $"You can check status in app.";
            }
            else if (appointment.IsFinished)
            {
                message = $"Hello! Your appointment scheduled for {appointment.DateTime} has been finished! " +
                    $"You can check results in app";
            }
            else
            {
                message = $"Hello! Your appointment scheduled for {appointment.DateTime} has been created! " +
                    $"You can check status in app";
            }

            await _notificationSender.Send(email, "Appointment status update", message);
        }
    }
}
```

Рис.4 - Код AppointmentObserver

```
using BookingClinic.Services.AppointmentObserver.Observer;

namespace BookingClinic.Services.AppointmentObserver.Subject
{
    Ссылок: 3
    public class AppointmentSubject : IAppointmentSubject
    {
        private readonly object _sync = new();
        private readonly List<IAppointmentObserver> _observers;

        Ссылок: 1
        public AppointmentSubject()
        {
            _observers = new ();
        }

        Ссылок: 2
        public void Attach(IAppointmentObserver observer)
        {
            lock (_sync)
            {
                _observers.Add(observer);
            }
        }

        Ссылок: 1
        public void Detach(IAppointmentObserver observer)
        {
            lock (_sync)
            {
                _observers.Remove(observer);
            }
        }

        Ссылок: 5
        public async Task NotifyAsync(BookingClinic.Data.Entities.Appointment appointment, string email)
        {
            IAppointmentObserver[]? obs = null;
            lock (_sync)
            {
                obs = _observers.ToArray();
            }

            foreach (var o in obs)
            {
                await o.UpdateAsync(appointment, email);
            }
        }
    }
```

Рис.5 - Код AppointmentSubject

```
using BookingClinic.Data.Repositories.AppointmentRepository;
using BookingClinic.Data.Repositories.UserRepository;
using BookingClinic.Services.AppointmentObserver.Subject;
using BookingClinic.Services.Data.Appointment;
using BookingClinic.Services.Data.Doctor;
using Mapster;
using System.Globalization;
using System.Security.Claims;

namespace BookingClinic.Services.Appointment
{
    Ссылок: 2
    public class AppointmentService : IAppointmentService
    {
        private readonly IAppointmentSubject _appointmentSubject;
        private readonly IAppointmentRepository _appointmentRepository;
        private readonly IUserRepository _userRepository;

        Ссылок: 0
        public AppointmentService(
            IAppointmentRepository appointmentRepository,
            IUserRepository userRepository,
            IAppointmentSubject appointmentSubject)
        {
            _appointmentRepository = appointmentRepository;
            _userRepository = userRepository;
            _appointmentSubject = appointmentSubject;
        }

        Ссылок: 2
        public async Task<ServiceResult<object>> CreateAppointment(MakeAppointmentDto dto, ClaimsPrincipal principal)
        {
            var idClaim = principal.FindFirst(c => c.Type == ClaimTypes.NameIdentifier);
            var id = Guid.Parse(idClaim.Value);

            var doctor = _userRepository.GetDoctorById(dto.DoctorId);

            if (doctor == null)
            {
                return ServiceResult<object>.Failure(
                    new List<ServiceError>() { ServiceError.DoctorNotFound() });
            }
```

Рис.6 - Код AppointmentService

```
41              var clinic = doctor.Clinic;
42
43              var times = dto.AppointmentTime.Split('-');
44              var hoursMinutes = times[0].Split(':');
45              var hours = int.Parse(hoursMinutes[0]);
46              var minutes = int.Parse(hoursMinutes[1]);
47              DateTime dateTime = DateTime.ParseExact(dto.AppointmentDay.Split(',')[1].Trim(), "dd.MM.yyyy", CultureInfo.Invariar
48              dateTime = DateTime.SpecifyKind(dateTime.AddHours(hours).AddMinutes(minutes), DateTimeKind.Utc);
49
50              var app = _appointmentRepository.GetByDateTime(dateTime);
51
52              if (app != null)
53              {
54                  return ServiceResult<object>.Failure(
55                      new List<ServiceError>() { ServiceError.AppointmentAlreadyExists() });
56              }
57
58              var user = _userRepository.GetById(id)!;
59              var appointment = new BookingClinic.Data.Entities.Appointment()
60              {
61                  Id = Guid.NewGuid(),
62                  PatientId = id,
63                  DoctorId = dto.DoctorId,
64                  CreatedAt = DateTime.UtcNow,
65                  DateTime = dateTime,
66                  Address = $"{clinic.Name}, {clinic.City} {clinic.Street} {clinic.Building}"
67              };
68
69              _appointmentRepository.AddEntity(appointment);
70
71              try
72              {
73                  await _appointmentRepository.SaveChangesAsync();
74                  await _appointmentSubject.NotifyAsync(appointment, user.Email);
75
76                  return ServiceResult<object>.Success(null);
77              }
78              catch (Exception)
79              {
80                  return ServiceResult<object>.Failure(
81                      new List<ServiceError>() { ServiceError.UnexpectedError() });
82              }
83          }
```

Рис.7 - Код AppointmentService

```
85          public async Task<ServiceResult<object>> CreateAppointmentDoctor(MakeAppointmentDocDto dto, ClaimsPrincipal principal)
86          {
87              var idClaim = principal.FindFirst(c => c.Type == ClaimTypes.NameIdentifier);
88              var id = Guid.Parse(idClaim.Value);
89
90              var doctor = _userRepository.GetDoctorById(id);
91
92              if (doctor == null)
93              {
94                  return ServiceResult<object>.Failure(
95                      new List<ServiceError>() { ServiceError.DoctorNotFound() });
96              }
97
98              var clinic = doctor.Clinic;
99
100             var times = dto.AppointmentTime.Split('-');
101             var hoursMinutes = times[0].Split(':');
102             var hours = int.Parse(hoursMinutes[0]);
103             var minutes = int.Parse(hoursMinutes[1]);
104             DateTime dateTime = DateTime.ParseExact(dto.AppointmentDay.Split(',')[1].Trim(), "dd.MM.yyyy", CultureInfo.Invaria
105             dateTime = DateTime.SpecifyKind(dateTime.AddHours(hours).AddMinutes(minutes), DateTimeKind.Utc);
106
107             var app = _appointmentRepository.GetByDateTime(dateTime);
108
109             if (app != null)
110             {
111                 return ServiceResult<object>.Failure(
112                     new List<ServiceError>() { ServiceError.AppointmentAlreadyExists() });
113             }
114
115             var user = _userRepository.GetById(dto.PatientId)!;
116             var appointment = new BookingClinic.Data.Entities.Appointment()
117             {
118                 Id = Guid.NewGuid(),
119                 PatientId = dto.PatientId,
120                 DoctorId = id,
121                 CreatedAt = DateTime.UtcNow,
122                 DateTime = dateTime,
123                 Address = $"{clinic.Name}, {clinic.City} {clinic.Street} {clinic.Building}"
124             };
125
126             _appointmentRepository.AddEntity(appointment);
127
```

Рис.8 - Код AppointmentService

```
127
128             try
129             {
130                 await _appointmentRepository.SaveChangesAsync();
131                 await _appointmentSubject.NotifyAsync(appointment, user.Email);
132
133                 return ServiceResult<object>.Success(null);
134             }
135             catch (Exception)
136             {
137                 return ServiceResult<object>.Failure(
138                     new List<ServiceError>() { ServiceError.UnexpectedError() });
139             }
140         }
141
142         public ServiceResult<List<PatientAppointmentDto>> GetPatientAppointments(ClaimsPrincipal principal)
143         {
144             var idClaim = principal.FindFirst(c => c.Type == ClaimTypes.NameIdentifier);
145             Guid id = Guid.Parse(idClaim.Value);
146
147             var res = _appointmentRepository.GetPatientAppointments(id).ToList();
148
149             var appointments = res.Adapt<List<PatientAppointmentDto>>();
150             return ServiceResult<List<PatientAppointmentDto>>.Success(appointments);
151         }
152
153         public ServiceResult<List<DoctorAppointmentDto>> GetDoctorAppointments(ClaimsPrincipal principal)
154         {
155             var idClaim = principal.FindFirst(c => c.Type == ClaimTypes.NameIdentifier);
156             Guid id = Guid.Parse(idClaim.Value);
157
158             var res = _appointmentRepository.GetDoctorAppointments(id, DateTime.UtcNow.Date);
159
160             var appointments = res.Adapt<List<DoctorAppointmentDto>>();
161             return ServiceResult<List<DoctorAppointmentDto>>.Success(appointments);
162         }
163
164         public async Task<ServiceResult<object>> CancelAppointment(Guid appId, ClaimsPrincipal principal)
165         {
166             var idClaim = principal.FindFirst(c => c.Type == ClaimTypes.NameIdentifier);
167             var id = Guid.Parse(idClaim.Value);
```

Рис.9 - Код AppointmentService

```
167             var id = Guid.Parse(idClaim.Value);
168
169             var appointment = _appointmentRepository.GetById(appId);
170
171             if (appointment == null)
172             {
173                 return ServiceResult<object>.Failure(
174                     new List<ServiceError>() { ServiceError.AppointmentNotFound() });
175             }
176             if (id != appointment.PatientId)
177             {
178                 return ServiceResult<object>.Failure(
179                     new List<ServiceError>() { ServiceError.AppointmentNotFound() });
180             }
181
182             appointment.IsCanceled = true;
183             var user = _userRepository.GetById(id)!;
184             _appointmentRepository.UpdateEntity(appointment);
185
186             try
187             {
188                 await _appointmentRepository.SaveChangesAsync();
189                 await _appointmentSubject.NotifyAsync(appointment, user.Email);
190                 return ServiceResult<object>.Success(null);
191             }
192             catch (Exception)
193             {
194                 return ServiceResult<object>.Failure(
195                     new List<ServiceError>() { ServiceError.UnexpectedError() });
196             }
197         }
198
199         public async Task<ServiceResult<object>> FinishAppointment(FinishAppointmentDto dto, ClaimsPrincipal principal)
200         {
201             var idClaim = principal.FindFirst(c => c.Type == ClaimTypes.NameIdentifier);
202             var id = Guid.Parse(idClaim.Value);
203
204             var appointment = _appointmentRepository.GetById(dto.Id);
205
206             if (appointment == null)
207             {
208                 return ServiceResult<object>.Failure(
```

Рис.10 - Код AppointmentService

```
208                     return ServiceResult<object>.Failure(
209                         new List<ServiceError>() { ServiceError.AppointmentNotFound() });
210                 }
211                 if (appointment.DoctorId != id)
212                 {
213                     return ServiceResult<object>.Failure(
214                         new List<ServiceError>() { ServiceError.AppointmentNotFound() });
215                 }
216
217                 var user = _userRepository.GetById(appointment.PatientId)!;
218                 appointment.IsFinished = true;
219                 appointment.Results = dto.Results;
220                 _appointmentRepository.UpdateEntity(appointment);
221
222                 try
223                 {
224                     await _appointmentRepository.SaveChangesAsync();
225                     await _appointmentSubject.NotifyAsync(appointment, user.Email);
226                     return ServiceResult<object>.Success(null);
227                 }
228                 catch (Exception)
229                 {
230                     return ServiceResult<object>.Failure(
231                         new List<ServiceError>() { ServiceError.UnexpectedError() });
232                 }
233             }
234         }
235     }
236
```

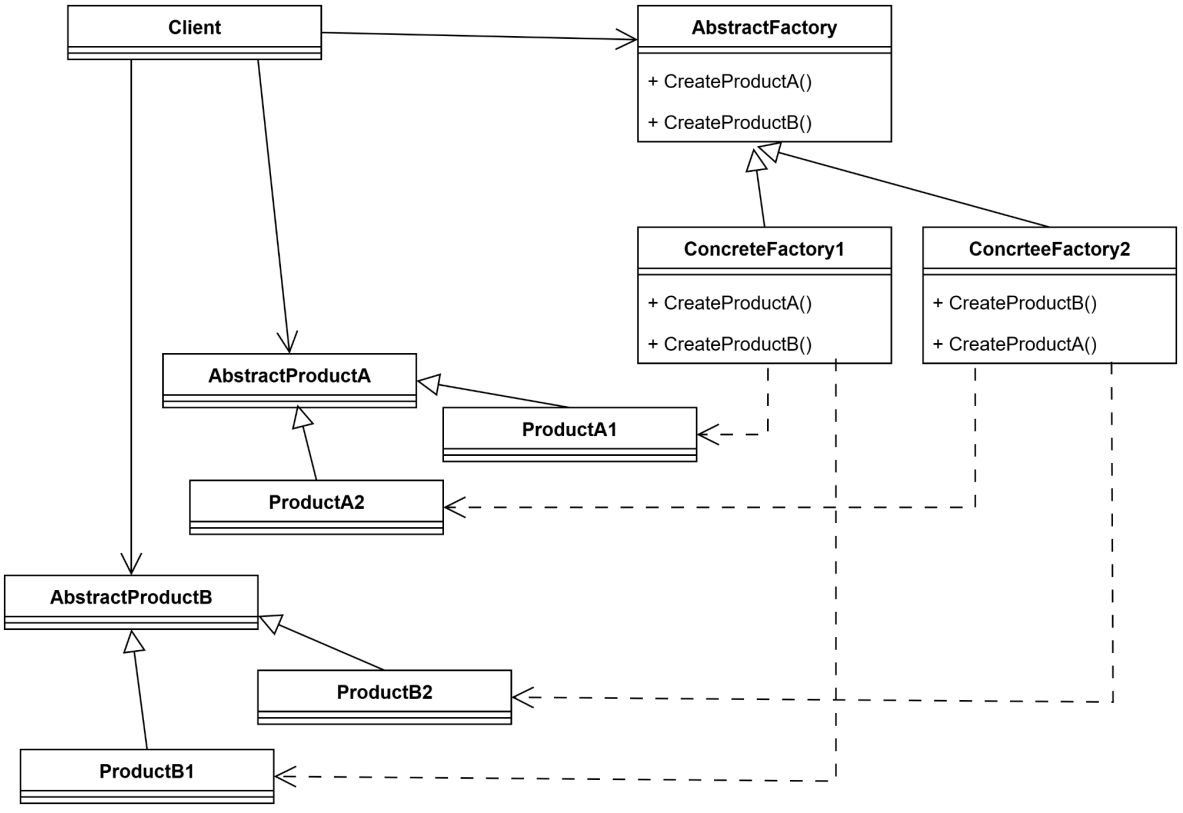
Рис.11 - Код AppointmentService

Контрольні питання:

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Виноситься загальний інтерфейс фабрики і створюються його реалізації для різних сімейств продуктів. Цей шаблон передусім структурує знання про схожі об'єкти, що називаються сімействами, і створює можливість взаємозаміни різних сімейств.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?

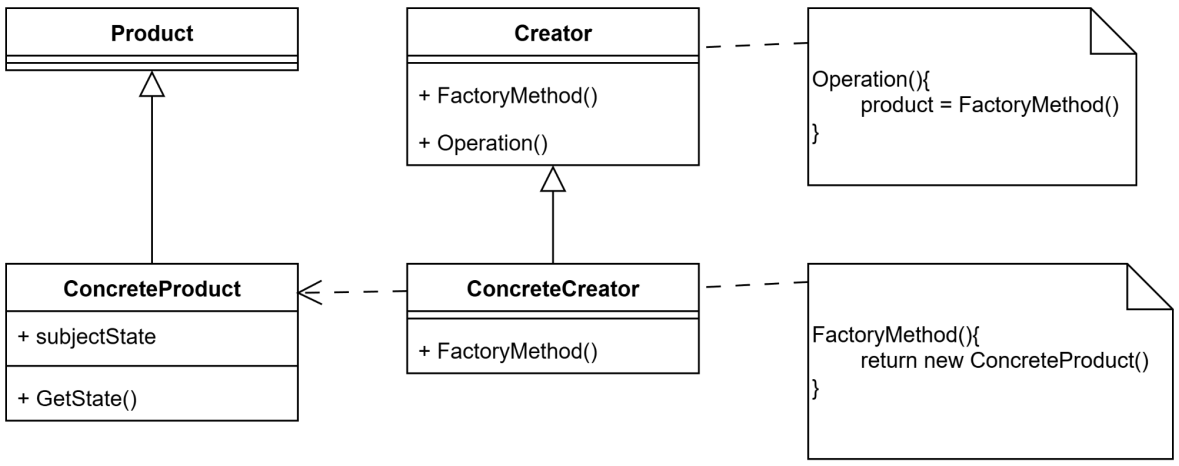
Client - клієнтський клас що використовує фабрику для створення сімейства об’єктів.

AbstractFactory - базовий клас фабрики. Має методи для створення певного сімейства об’єктів AbstractProductA, AbstractProductB.
AbstractProductA, AbstractProductB - базові класи для різних типів продуктів.
ConcreteFactory1, ConcreteFactory2 - конкретні реалізації фабрики. Одна створює сімейство об’єктів ProductA1, ProductA2, інша - ProductB1, ProductB2.
ProductA1, ProductA2, ProductB1, ProductB2 - класи, що породжуються конкретними фабриками і належать до певних сімейств об’єктів.

4. Яке призначення шаблону «Фабричний метод»?

Визначає інтерфейс для створення об'єктів певного базового типу. Він дозволяє делегувати вибір типу створюваного об’єкта підкласам, що робить код гнучкішим і зручним для розширення.
Наприклад, застосунок працює з мережевими драйверами і використовує клас Packet для зберігання даних, що передаються в мережу. Залежно від використовуваного протоколу, існує два перевантаження: TcpPacket, UdpPacket. І відповідно два створюючі об'єкти (TcpCreator, UdpCreator) з фабричним методом (який створює відповідні реалізації). Базова функціональність (передача пакету, прийом пакету, заповнення пакету даними) поміщається у базовий клас PacketCreator. Таким чином поведінка системи залишається тою ж, проте з'являється можливість підстановки власних об'єктів в процес створення і роботи з пакетами.

5. Нарисуйте структуру шаблону «Фабричний метод».



6. Які класи входять в шаблон «Фабричний метод», та яка між ними взаємодія?

Creator - базовий клас творця. Оголошує метод для створення продукту та додаткові для роботи з ним.
ConcreteCreator - конкретний творець, визначає метод для створення продукту і створює продукт ConcreteProduct свого типу.
ConcreteProduct - клас конкретного продукту для кожного творця.
Product - базовий клас для всіх продуктів.

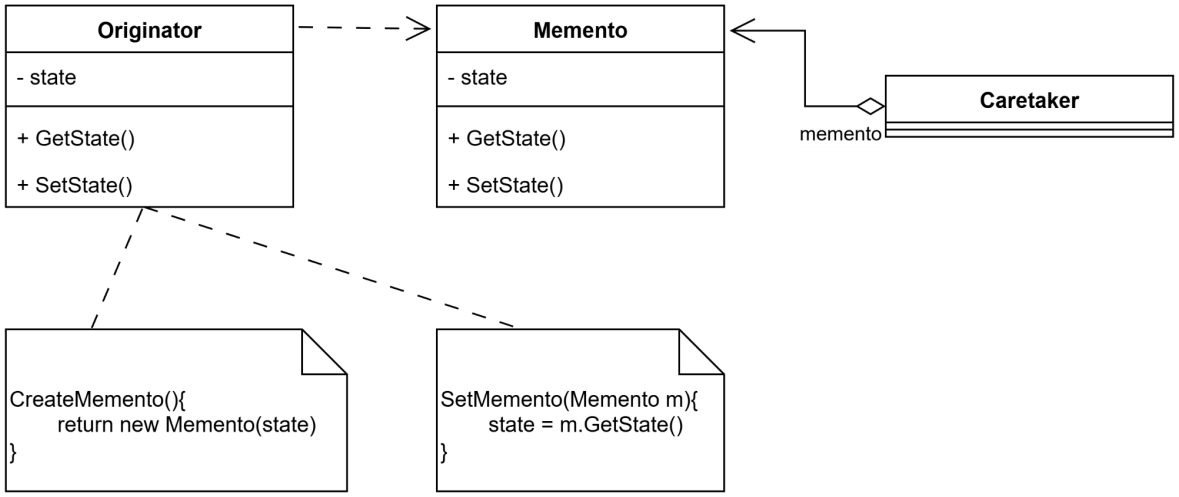
7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

Фабричний метод призначений для створення одного об’єкту. Тобто є базовий клас творця і похідні конкретні творці. Кожен такий творець створює об’єкт свого типу, що наслідується від базового. Наприклад є конкретний творець для дерев’яного стільця, а інший - для золотого. Кожен творець створює свій стілець через загальний метод.
Абстрактна фабрика визначає методи для створення декількох різних об’єктів, але які належать до якогось сімейства. Наприклад може бути фабрика для створення дерев’яних меблів (стільці, дивани, столи), а може бути для золотих.

8. Яке призначення шаблону «Знімок»?

Використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. Він дає змогу зберігати та відновлювати минулий стан об’єктів, не розкриваючи подробиць їхньої реалізації. Виділяє клас-знімок, що визначає стан, клас для зберігання знімків станів, та клас для встановлення та відновлення знімків. Можна застосовувати в парі з шаблоном “Команда” для забезпечення операцій відміни команд.

9. Нарисуйте структуру шаблону «Знімок».



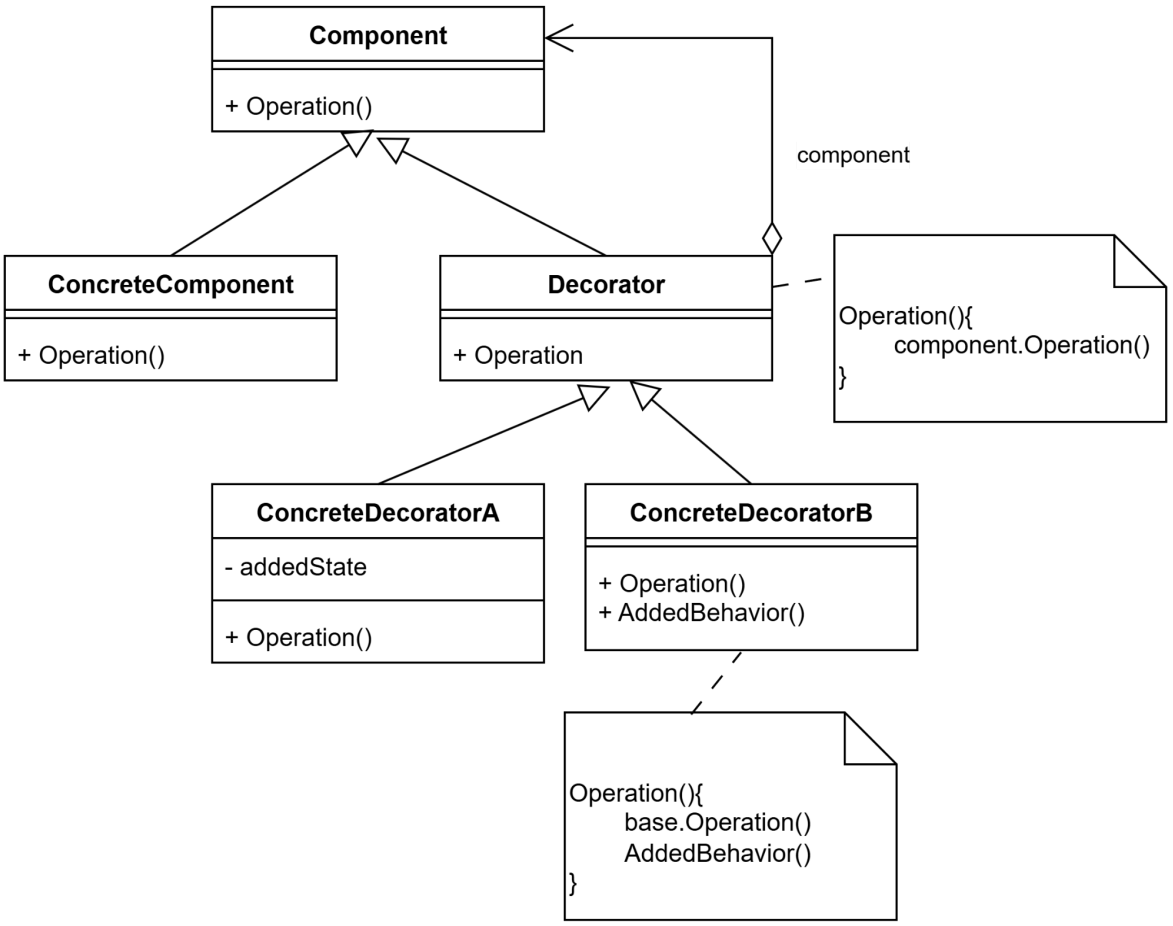
10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

Originator - клас творця, який може створювати знімки свого стану та відновлювати минулий стан.
Memento - клас-знімок, що містить стан творця.
Caretaker - передає та зберігає об’єкти Memento.

11. Яке призначення шаблону «Декоратор»?

Призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Обгортає об’єкт зі збереженням його функцій, але дозволяє додати додаткові дії. Надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі.
Прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в «прокручуваному» елементі, і при необхідності з'являється полоса прокрутки. Початкові функції елементу (наприклад, рядки статусу) залишаються незмінними.

12. Нарисуйте структуру шаблону «Декоратор».



13. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?

Component - базовий клас.
ConcreteComponent - початковий об’єкт, який можна обгорнути для додавання нового функціоналу.
Decorator - базовий клас декоратора для Component. Містить посилання на клас, який обгортає.
ConcreteDecoratorA, ConcreteDecoratorB - конкретні реалізації декораторів. Додають додатковий функціонал до об’єкта що обгортається.

14. Які є обмеження використання шаблону «декоратор»?

- Може ускладнювати структуру коду, якщо створювати багато обгорток.
- Важко конфігурувати об’єкти, які обертаються в декілька декораторів.

- Не підходить, коли потрібно змінювати поведінку всіх об’єктів класу одразу.

Висновки:
У ході лабораторної роботи я ознайомився з шаблонами «Abstract Factory» для створення сімейств об’єктів, «Factory Method» для можливості розширення створюваних об’єктів, «Memento» для відслідковування станів об’єкту, «Observer» для реалізації моделі підписка/розсилка, «Decorator» для доповнення функціоналу класів. Обрав та реалізував патерн “Observer” для реалізації механізму оповіщення користувача про зміни статусу прийому. Патерн включає в себе суб’єкта, який сповіщає підписаних спостерігачів при зміні статусу прийому, та їхні інтерфейси. Вивчене покращило моє розуміння патернів проектування та знадобиться в подальшій розробці.