Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Магістерська робота

освітній ступінь – магістр

на тему: **«ВДОСКОНАЛЕННЯ РЕАЛІЗАЦІЙ ПАТЕРНІВ ПРОЕКТУВАННЯ НА ПІДСТАВІ ФОРМАЛЬНИХ МОДЕЛЕЙ»**

Виконала: студентка 2-го року навчання,

Спеціальності

121 Інженерія програмного забезпечення

Семенюк Христина Романівна

Керівник Бублик В.В.,

кандидат наук, доцент

Рецензент___________________________

(прізвище та ініціали)

Магістерська робота захищена

з оцінкою__________________________

Секретар ЕК________________________

«____» _______________________ 2022 р.

Київ – 2022

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Освітній ступінь: магістр

Спеціальність: 121 Інженерія програмного забезпечення

Освітньо-наукова програма: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри

мультимедійних систем,

доцент, к.н.

О. П. Жежерун___________

«____»_______________2022 року

ЗАВДАННЯ

ДЛЯ МАГІСТЕРСЬКОЇ РОБОТИ СТУДЕНТУ

Семенюк Христині Романівні

1. Тема роботи: Вдосконалення реалізацій патернів проектування на підставі формальних моделей

керівник: роботи Бублик Володимир Васильович, кандидат наук, доцент

затверджені наказом вищого навчального закладу від «____»_______________20__ року №___ __

2. Строк подання студентом роботи______________________________

3. План роботи:

   Анотація

   Вступ

   1 Реалізація мультиметодів

   2 Засоби формалізації патернів проектування

   3 Патерн Adapter

   4 Патерн Abstract Factory

   5 Вдосконалена реалізація мультиметодів

   Висновки

   Список літератури

Тема: Вдосконалення реалізацій патернів проектування на підставі формальних моделей

Календарний план виконання роботи:

№ п/п	Назва етапу проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	вересень – жовтень 2021	
2.	Огляд технічної літератури за темою роботи	жовтень – грудень 2021	
3.	Створення практичної частини роботи	січень – травень 2022	
4.	Написання текстової частини роботи	травень – червень 2022	
5.	Надання роботи керівнику для перевірки	червень 2022	
6.	Коригування роботи за результатами перевірки	червень – липень 2022	
7.	Подання роботи на кафедру для перевірки на плагіат	03.07.2022	
8.	Здача курсової роботи	07.07.2022	

Студент Семенюк Х.Р.

Керівник Бублик В.В.

“______” _______________

Зміст

Анотація 4
Вступ... 5
1 Реалізація мультиметодів 7
2 Засоби формалізації патернів проектування 11
3 Патерн Adapter... 12
3.1 Класифікація 12
3.2 Призначення.. 13
3.3 Мотивація.. 13
3.4 Застосування 14
3.5 Структура 14
3.6 Учасники 16
3.7 Відносини.. 16
3.8 Результати 17
3.9 Реалізація... 17
4 Патерн Abstract Factory... 18
4.1 Класифікація 18
4.2 Призначення.. 18
4.3 Мотивація.. 18
4.4 Застосування 19
4.5 Структура 19
4.6 Учасники 20
4.7 Відносини.. 21
4.8 Результати 21
4.9 Реалізація... 22
5 Вдосконалена реалізація мультиметодів 22
5.1 Застосування патерна проектування Adapter у реалізації мультиметодів 22
5.2 Застосування патерна проектування Abstract Factory у реалізації мультиметодів... 25

Висновки 29
Список літератури 31

Анотація

Роботу присвячено вдосконаленню реалізації мультиметодів за допомогою застосування патернів проектування. Вибрано і застосовано патерни Adapter і Abstract Factory для спрощення публічного інтерфейсу статичного та динамічного диспетчерів, реалізовано приклади застосування вдосконалених мультиметодів. Описано застосовані патерни проектування Adapter і Abstract Factory за допомогою засобів формалізації, описано їхнє застосування в реалізації мультиметодів і переваги отриманих результатів.

Вступ

У книзі Андрія Александреску «Сучасне проектування на С++» розглянуто реалізації мультиметодів або подвійної диспетчеризації – механізму, що надає можливість виклику різних реалізацій функції залежно від динамічних типів двох аргументів функції. Подвійну диспетчеризацію застосовують, коли певна операція маніпулює двома поліморфними об'єктами за допомогою вказівників чи посилань на їхні базові класи, і її потрібно модифікувати залежно від динамічних типів цих об’єктів.

У попередній курсовій роботі було досліджено і описано подвійну диспетчеризацію і її реалізації з книги Александреску: статичний і динамічний диспетчер. Було вдосконалено ці реалізації застосуванням можливостей нових версій С++, зокрема варіативних шаблонів.

Отримана реалізація диспетчерів надає користувачеві багато можливостей: застосовувати диспетчеризацію для різних ієрархій класів і функцій-виконавців, що здійснюють основну роботу над поліморфними об’єктами, створювати симетричні і несиметричні диспетчери, обирати між статичним та динамічним диспетчером, а також між стратегіями, які використовує динамічний диспетчер. Таким чином, користувач може застосовувати певний диспетчер залежно від конкретних завдань і потреб. Проте публічний інтерфейс статичного і динамічного диспетчерів досить важкий для використання і розуміння, а також різний в обох диспетчерів. Було б корисно виправити цей недолік, застосувавши певні патерни проектування.

За мету цієї роботи було посталено вдосконалення реалізації мультиметодів, розглянутої у книзі Александреску і у попередній курсовій роботі, за допомогою застосування патернів проектування.

Щоб досягти поставленої мети, потрібно розв’язати такі завдання:

а) проаналізувати патерни проектування, вибрати ті, які можуть вдосконалити реалізацію мультиметодів;

б) вдосконалити реалізацію мультиметодів, застосувавши вибрані патерни проектування;

в) описати вибрані патерни проектування, скориставшись засобами формалізації, описаними у книгах Gang of Four (GoF) «Шаблони проектування: Елементи повторно використовуваного об'єктно-орієнтованого програмного забезпечення» та Джейсона Мак-Колм Сміта «Елементарні шаблони проектування»;

г) описати застосування вибраних патернів проектування у реалізації мультиметодів і переваги отриманих результатів.

Робота складається з п'яти розділів.

Перший розділ присвячено загальному огляду реалізації мультиметодів, розглянутої у книзі Александреску і у попередній курсовій роботі, та опису її недоліків.

Другий розділ присвячено короткому загальному опису засобів формалізації, описаних у книгах Gang of Four (GoF) «Шаблони проектування: Елементи повторно використовуваного об'єктно-орієнтованого програмного забезпечення» та Джейсона Мак-Колм Сміта «Елементарні шаблони проектування».

У третьому розділі наведено специфікацію патерна Adapter.

У четвертому розділі наведено специфікацію патерна Abstract Factory.

П'ятий розділ присвячено опису застосування вибраних патернів проектування у реалізації мультиметодів і переваг отриманих результатів.

Створено програмний продукт: вдосконалена реалізація статичної і динамічної подвійної диспетчеризації із застосуванням патернів проектування Adapter і Abstract Factory; приклад застосування – диспетчеризація викликів функцій для ієрархії класів з базовим класом Shape і трьома похідними.

1 Реалізація мультиметодів

У попередній курсовій роботі було розглянуто мультиметоди з книги Андрія Александреску «Сучасне проектування на C++», а також реалізовано їх з використанням можливостей нових версій C++, зокрема варіативних шаблонів.

Мультиметоди або множинна диспетчеризація – це механізм, який надає можливість виклику різних реалізацій функції залежно від динамічних типів кількох об’єктів, наприклад аргументів функції. Динамічний тип об’єкта ідентифікується у процесі виконання програми. Наприклад, якщо створити вказівник базового класу, який вказує на об’єкт похідного класу, то динамічним типом вказівника буде похідний клас, а не базовий. Такий об’єкт називають поліморфним.

Механізм множинної диспетчеризації не є властивістю мови C++, але він буває потрібним у деяких ситуаціях, наприклад, коли у програмі є операція, що маніпулює кількома поліморфними об’єктами за допомогою вказівників чи посилань на їхні базові класи, і яку потрібно модифікувати залежно від динамічних типів цих об’єктів.

У попередній роботі було реалізовано подвійну диспетчеризацію із книги Александреску – це частковий випадок множинної диспетчеризації, при якому функція приймає два аргументи, тобто диспетчеризація відбувається за типами двох об’єктів. Було реалізовано статичний і динамічний диспетчер. Статичний диспетчер у деяких ситуаціях є кращим варіантом, ніж динамічний, наприклад, він швидший для невеликої кількості класів. З іншого боку, динамічний диспетчер дає можливість реєструвати в одному об’єкті диспетчера симетричні і несиметричні функції і можливість реєструвати функції в одному об’єкті диспетчера у різний час і в різних місцях програми. Окрім того, динамічний диспетчер використовує патерн проектування стратегія, завдяки якому він може застосовувати різні оператори для зведення типів, наприклад static_cast і dynamic_cast, а також різні контейнери для зберігання об’єктів зворотного виклику, тобто вказівників на клієнтські функції-виконавців. Отже, у

клієнтському коді є можливість обирати між статичним і динамічним диспетчером, а також застосовувати різні стратегії у динамічному диспетчері залежно від конкретних ситуацій і потреб. Це робить реалізацію диспетчерів досить гнучкою. [1]

Проте публічний інтерфейс обох диспетчерів досить громіздкий, важкий для використання і розуміння. Потрібно передавати класу диспетчера багато шаблонних параметрів. Наприклад, для статичного диспетчера:

а) клас, який містить перевантажені методи, що виконують роботу над об’єктами;

б) шаблонний булевий аргумент, що вказує, чи мультиметод повинен бути симетричним (симетричний мультиметод: якщо список типів для обох аргументів однаковий і для клієнтських методів не важливий порядок слідування аргументів, диспетчер може сам за потреби міняти аргументи місцями перед викликом метода);

в) базові типи обох аргументів;

г) списки типів, що складаються з можливих похідних класів для обох аргументів;

д) тип результату подвійної диспетчеризації.

Приклад оголошення статичного диспетчера:

```
using FirstStaticDispatcher = StaticDispatcher<HatchingExecutor, true,
Shape, TypeList<Rectangle, Ellipse, Poly, NullType>, Shape,
TypeList<Rectangle, Ellipse, Poly, NullType>, int>;
```

У цьому прикладі HatchingExecutor – це клас, з клієнтськими методами, int – тип результату подвійної диспетчеризації, тобто тип повернення клієнтських методів, Shape – базовий клас і Rectangle, Ellipse, Poly – похідні класи ієрархії, для якої здійснюється диспетчеризація. Шаблонний булевий аргумент – true, це означає, що цей диспетчер симетричний. Тобто цьому диспетчеру потрібно передавати два аргументи, у яких динамічний тип – один з похідних класів ієрархії, а диспетчер буде викликати один з перевантажених клієнтських методів з класу HatchingExecutor і за потреби міняти порядок аргументів.

Деякі шаблонні параметри мають значення за замовчуванням, це трохи покращує ситуацію і робить оголошення диспетчера легшим. За замовчуванням ієрархія класів для обох аргументів однакова, тому можна задати лише один базовий клас і один список типів, тип результату подвійної диспетчеризації – void. Оголошення статичного диспетчера з використанням значень за замовчуванням деяких шаблонних параметрів:

```
using SecondStaticDispatcher = StaticDispatcher<HatchingExecutor, true,
Shape, TypeList<Rectangle, Ellipse, Poly, NullType>>;
```

Для динамічного диспетчера при оголошенні потрібно передати такі шаблонні параметри:

а) базові типи обох аргументів. За замовчуванням однакові, тому можна задати один;

б) тип результату подвійної диспетчеризації. За замовчуванням void;

в) клас-стратегія для зведення типів. За замовчуванням та, яка застосовує dynamic_cast;

г) клас-стратегія внутрішнього диспетчера, який використовує певний контейнер для зберігання об’єктів зворотного виклику, тобто вказівників на клієнтські функції-виконавців. Контейнер може бути асоціативним масивом (за замовчуванням) або матрицею.

Приклад оголошення динамічного диспетчера:

```
using FirstFnDispatcher = FnDispatcher<Shape, Shape, string,
StaticCaster, BasicFastDispatcher>;
```

У цьому прикладі Shape – базовий клас ієрархії, для якої здійснюється диспетчеризація, string – тип результату подвійної диспетчеризації, тобто тип повернення клієнтських методів, StaticCaster – клас-стратегія для зведення типів, який застосовує static_cast, BasicFastDispatcher – клас-стратегія внутрішнього диспетчера, який використовує асоціативний масив для зберігання об’єктів зворотного виклику. Цьому диспетчеру потрібно передавати два аргументи, які є поліморфними об’єктами, а диспетчер буде викликати функцію з контейнера зворотних викликів і за потреби міняти порядок аргументів, якщо функція симетрична.

Оголошення динамічного диспетчера з використанням значень за замовчуванням деяких шаблонних параметрів:

```
using SecondFnDispatcher = FnDispatcher<Shape>;
```

Також потрібно створити об’єкт динамічного диспетчера, бо його функції нестатичні:

```
FirstFnDispatcher disp;
```

Динамічний диспетчер має у публічному інтерфейсі функцію Add, що додає в контейнер об’єкти зворотного виклику, тобто клієнтські функції, які виконують роботу над поліморфними об’єктами і які диспетчер має викликати залежно від типу аргументів. Тобто, щоб скористатися динамічним диспетчером, потрібно у клієнтському коді оголосити і визначити функції у безіменному просторі імен, потім вказівники на них передати функції Add як шаблонний параметр без типу (non-type), таким чином зареєструвавши функції в об’єкті диспетчера. Приклад оголошення клієнтської функції у безіменному просторі імен:

```
namespace
{
	string HatchRectanglePoly(Rectangle& r, Poly& p)
	{
		return "HatchRectanglePoly(Rectangle& r, Poly& p)";
	}
}
```

Приклад виклику функції Add для реєстрації клієнтських функцій в об’єкті диспетчера:

```
disp.Add<Rectangle, Poly, HatchRectanglePoly, true>();
```

Ця функція також потребує шаблонні параметри: конкретні похідні типи аргументів для клієнтських функцій, вказівник на клієнтську функцію, шаблонний булевий аргумент, що вказує, чи метод повинен бути симетричним. Це дає реалізації динамічного диспетчера гнучкість: можливість реєструвати в одному об’єкті диспетчера симетричні і несиметричні функції і можливість

реєструвати функції в одному об’єкті диспетчера у різний час і в різних місцях програми – але ускладнює публічний інтерфейс.

Окрім того, у статичного і динамічного диспетчера різний публічний інтерфейс. У статичного диспетчера головна функція, яка здійснює диспетчеризацію, статична, у динамічного – ні. У обох диспетчерів ці функції приймають різну кількість аргументів: у динамічного – два аргументи – поліморфні об’єкти, для яких потрібно здійснити диспетчеризацію, у статичного – ці два аргументи і ще один – об’єкт класу, у якому реалізовано клієнтські функції. Приклад виклику функції статичного диспетчера:

```
FirstStaticDispatcher::Go(*rectangle, *poly, exec);
```

Приклад виклику функції динамічного диспетчера:

```
disp.Go(*rectangle, *poly);
```

Як наслідок, при застосуванні цих реалізацій мультиметодів може бути легко припуститися помилки, важко здійснювати перевірку коду і пошук помилок. Також якщо у клієнтському коді виникне потреба замінити використання статичного диспетчера на використання динамічного або навпаки, доведеться вносити багато змін у код, у тому числі там, де викликаються функції диспетчера, адже у статичного і динамічного диспетчера різний публічний інтерфейс. З цієї ж причини не можна помістити об’єкти різних диспетчерів у одну структуру даних, наприклад масив, і у циклі викликати функцію для цих об’єктів. Також не можна використати диспетчер як стратегію, адже клієнтський код залежить від того, яка саме реалізація диспетчера використовується, з яким публічним інтерфейсом.

Ці недоліки можна виправити, застосувавши патерни проектування Abstract Factory і Adapter.

2 Засоби формалізації патернів проектування

У книзі Gang of Four (GoF) «Шаблони проектування: Елементи повторно використовуваного об’єктно-орієнтованого програмного забезпечення» та у

книзі Джейсона Мак-Колм Сміта «Елементарні шаблони проектування» визначено специфікації патернів проектування у певному форматі:

а) ім’я;

б) класифікація;

в) призначення;

г) мотивація (проблема, яку вирішує шаблон);

д) застосування (як шаблон варто чи не варто використовувати);

е) структура (у вигляді UML, PINbox);

є) учасники (сутності, які беруть участь у патерні; ролі);

ж) відносини;

з) результати;

и) реалізація.

У книзі Сміта патерни зображені за допомогою графічної системи позначень екземплярів шаблонів – Pattern Instance Notation (PIN), яка дозволяє описувати шаблони та способи їхньої взаємодії. PIN може використовуватись з іншими системами позначень, наприклад UML. Компонент PINbox зображує окремий екземпляр шаблона і дозволяє вибирати рівень деталізації при його поданні.

Система PIN та специфікації є засобами формалізації патернів проектування, формальними моделями. Ці засоби було детально розглянуто і використано у попередній курсовій роботі, у цій роботі їх також використано у наступних розділах для описання деяких патернів проектування. [2]

3 Патерн Adapter

Також відомий під назвою Wrapper (обгортка).

3.1 Класифікація

Структурний патерн – структурує класи та об’єкти.

3.2 Призначення

Перетворює інтерфейс одного класу на інтерфейс іншого, який потрібен клієнту. Робить можливою спільну роботу класів із несумісними інтерфейсами.

3.3 Мотивація

Іноді клас із бібліотеки або інший незалежно розроблений клас не вдається використати лише тому, що його публічний інтерфейс не відповідає тому, який потрібен конкретній клієнтській програмі. Навіть якщо вихідний код цього класу доступний, далеко не завжди можна вносити у нього зміни. Це нормальна ситуація, адже бібліотека не повинна пристосовуватись до кожної конкретної клієнтської програми.

Щоб вирішити цю проблему, можна створити клас, що адаптуватиме публічний інтерфейс класу з потрібними даними та поведінкою до інтерфейсу, який потрібен клієнту. Створений клас можна назвати адаптером, а потрібний клас із несумісним інтерфейсом – адаптованим класом. У адаптера має бути публічний інтерфейс такий, якого потребує клієнт, і він має використовувати дані і поведінку адаптованого класу. Реалізувати клас адаптер можна різними способами:

а) успадкувати реалізацію від адаптованого класу;

б) додати екземпляр адаптованого класу як поле у адаптер.

Ці два підходи відповідають класовому та об’єктному варіантам патерна Adapter: адаптер класу та адаптер об’єкта.

Таким чином, клієнтський код може звертатися до адаптера через такий публічний інтерфейс, який йому підходить, а адаптер буде звертатися до адаптованого класу за необхідними даними і поведінкою. Іноді адаптер може додатково містити потрібний функціонал, якого немає в адаптованому класі.

3.4 Застосування

Патерн Adapter варто застосовувати у таких ситуаціях:

а) потрібно використати наявний клас, але його публічний інтерфейс не відповідає потребам;

б) потрібно створити повторно використовуваний клас, який повинен взаємодіяти із заздалегідь невідомими або не пов'язаними з ним класами, що мають несумісні інтерфейси;

в) потрібно використати кілька наявних похідних класів з несумісним інтерфейсом. Проте у такому випадку створювати адаптери для кожного з похідних класів непрактично, краще використати адаптер об'єктів, який пристосує інтерфейс базового класу.

3.5 Структура

Адаптер класу використовує множинне наслідування для адаптації одного інтерфейсу до іншого:

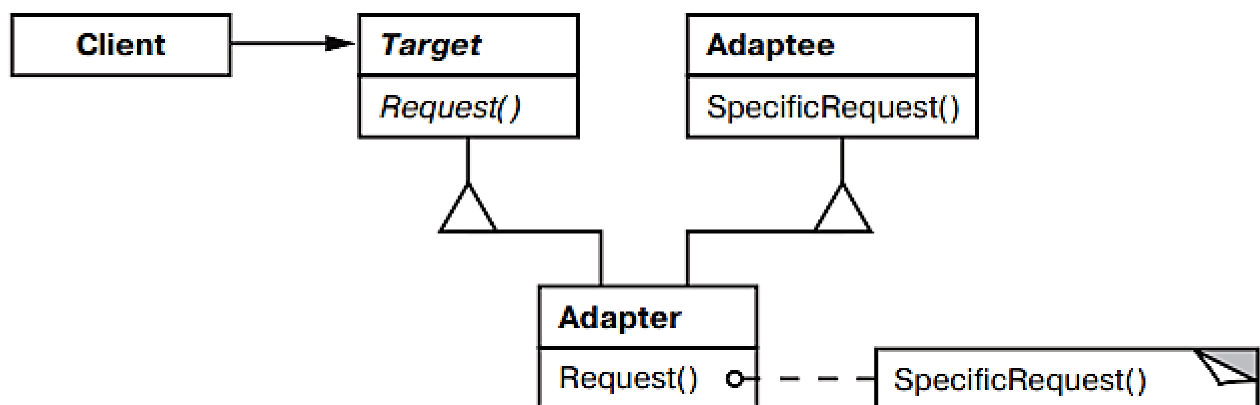

Рисунок 3.1 – Патерн Adapter, варіант адаптер класу, у вигляді діаграми UML

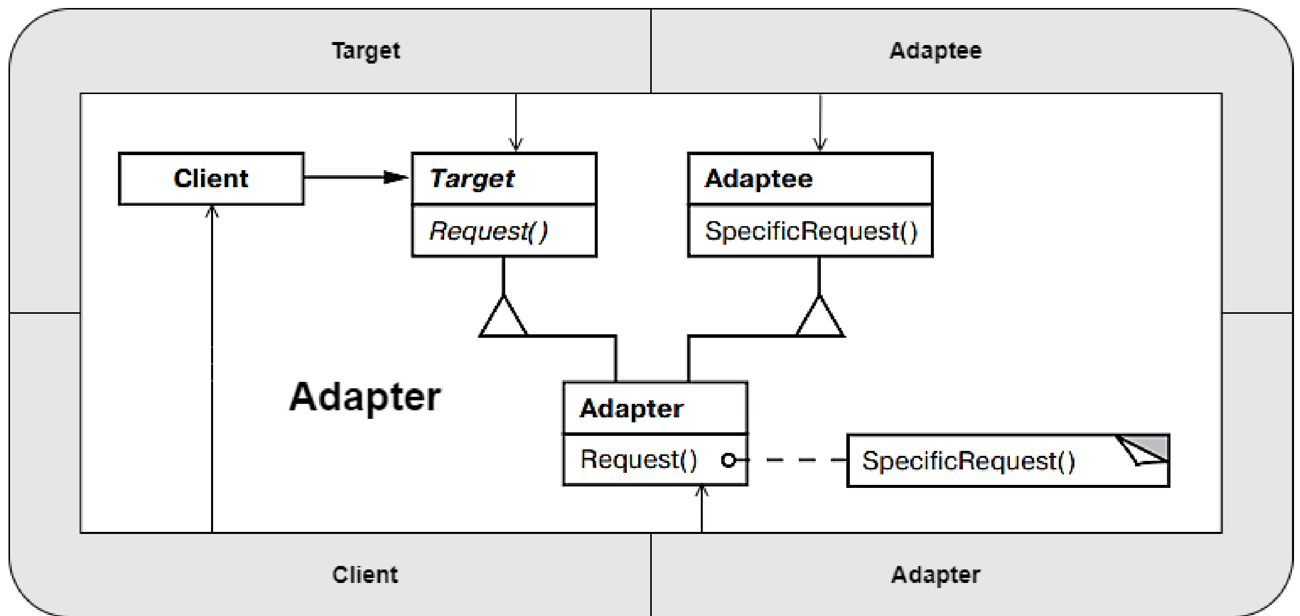

Рисунок 3.2 – Патерн Adapter, варіант адаптер класу, у вигляді розкритого компонента PINbox та діаграми UML

Адаптер об'єкта використовує композицію об'єктів:

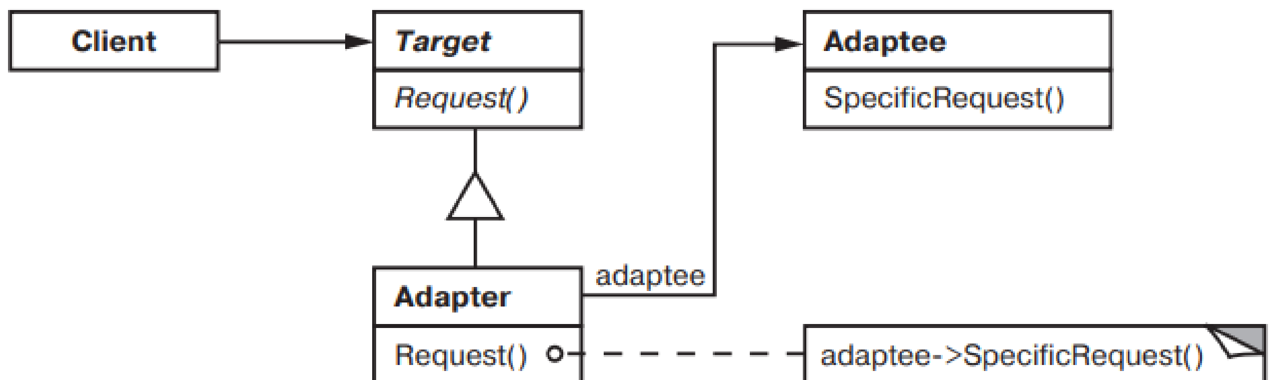

Рисунок 3.3 – Патерн Adapter, варіант адаптер об'єкта, у вигляді діаграми UML

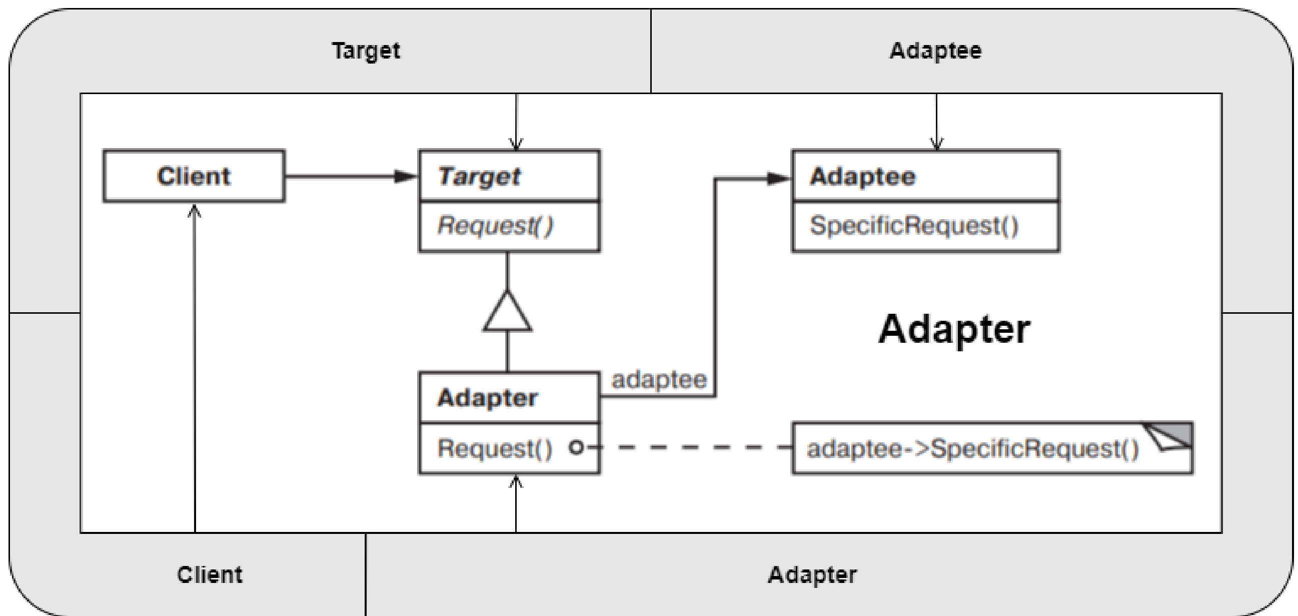

Рисунок 3.4 – Патерн Adapter, варіант адаптер об'єкта, у вигляді розкритого компонента PINbox та діаграми UML

3.6 Учасники

Target – цільовий клас – визначає публічний інтерфейс, яким користується Client та який залежить від предметної області.

Client – клієнт – вступає у відносини з об’єктами, що задовольняють інтерфейс Target.

Adaptee – адаптований клас – має інтерфейс, який потребує адаптації, і потрібні Client дані та поведінку.

Adapter – клас адаптер – адаптує інтерфейс Adaptee до Target: має дані та поведінку Adaptee і публічний інтерфейс Target.

3.7 Відносини

Client викликає методи Adapter. Adapter викликає методи об’єкта або класу Adaptee, який виконує головну роботу, потрібну Client.

3.8 Результати

Результати застосування адаптера об’єктів та адаптера класів різні. Адаптер класу:

а) адаптує інтерфейс Adaptee до інтерфейсу Target, передоручаючи дії конкретному класу Adaptee. Цей варіант патерна Adapter не працюватиме, якщо потрібно одночасно адаптувати клас та його підкласи;

б) дозволяє класу Adapter перевизначити деякі методи класу Adaptee, тому що Adapter є підкласом Adaptee;

в) вводить лише один новий об’єкт. Щоб дістатися до Adaptee, не потрібно ніякого додаткового звернення за вказівником.

Адаптер об’єкта:

а) дозволяє одному адаптеру Adapter працювати з об’єктами адаптованого класу Adaptee і його похідними класами, якщо вони є. Адаптер може додати нову функціональність відразу всім адаптованим об’єктам;

б) ускладнює перевизначення методів класу Adaptee. Для цього потрібно створити клас, похідний від Adaptee, і змусити Adapter посилатися на цей клас, а не сам Adaptee.

3.9 Реалізація

При реалізації патерна Adapter на C++ клас Adapter відкрито, публічно успадковує клас Target і закрито, приватно клас Adaptee:

```
class Adapter : public Target, Adaptee { }
```

При закритому спадкуванні похідний клас користується базовим класом виключно сам і не передає цю можливість своїм клієнтам і похідним класам, а при відкритому спадкуванні передає, отже публічний інтерфейс відкрито успадкованого класу стає інтерфейсом похідного класу. Закрите успадкування ще називають успадкуванням реалізації, а відкрите – успадкуванням реалізації і поведінки. Це якраз те, що потрібно для реалізації патерна Adapter. [3]

4 Патерн Abstract Factory

Також відомий під назвою Kit (інструментарій).

4.1 Класифікація

Патерн, що породжує об’єкти.

4.2 Призначення

Надає інтерфейс для створення сімейств взаємопов’язаних чи взаємозалежних об’єктів, не специфікуючи їх конкретних класів.

4.3 Мотивація

Наприклад, є програма для створення користувацького інтерфейсу, що складається з певної кількості різних елементів і підтримує режими з різним зовнішнім виглядом та поведінкою елементів. Важливо, щоб режим застосунку було легко змінити. Якщо об’єкти для конкретного режиму створюються в різних місцях програми, то змінити режим буде нелегко.

Можна вирішити цю проблему, визначивши абстрактний клас Factory з інтерфейсом для створення всіх основних видів елементів користувацького інтерфейсу. Також потрібні абстрактні класи для кожного виду елементів і конкретні підкласи, що реалізують елементи для кожного режиму. В інтерфейсі Factory потрібна операція, що повертає новий об’єкт кожного абстрактного класу елементів. Для кожного режиму потрібно створити клас, похідний від Factory, який буде реалізовувати операції, що створюють елементи користувацького інтерфейсу відповідного режиму. Клієнти створюватимуть елементи, користуючись лише інтерфейсом абстрактного класу Factory, не знаючи, які підкласи реалізують цей інтерфейс і які саме підкласи елементів вони використовують. Таким чином, клієнти незалежні від вибраного режиму користувацького інтерфейсу.

Клас Factory також встановлює залежності між конкретними класами елементів: кожен підклас абстрактного класу Factory створює кілька елементів для одного конкретного режиму.

4.4 Застосування

Патерн Abstract Factory варто застосовувати, коли:

а) система не повинна залежати від того, як створюються, компонуються й представляються об'єкти, що входять у неї;

б) взаємопов'язані об'єкти, що входять у сімейство, повинні використовуватись разом, і потрібно забезпечити виконання цього обмеження;

в) система повинна налаштовуватись одним із сімейств об'єктів, які входять до неї;

г) потрібно надати бібліотеку об'єктів, розкриваючи лише їхні інтерфейси, але не реалізацію.

4.5 Структура

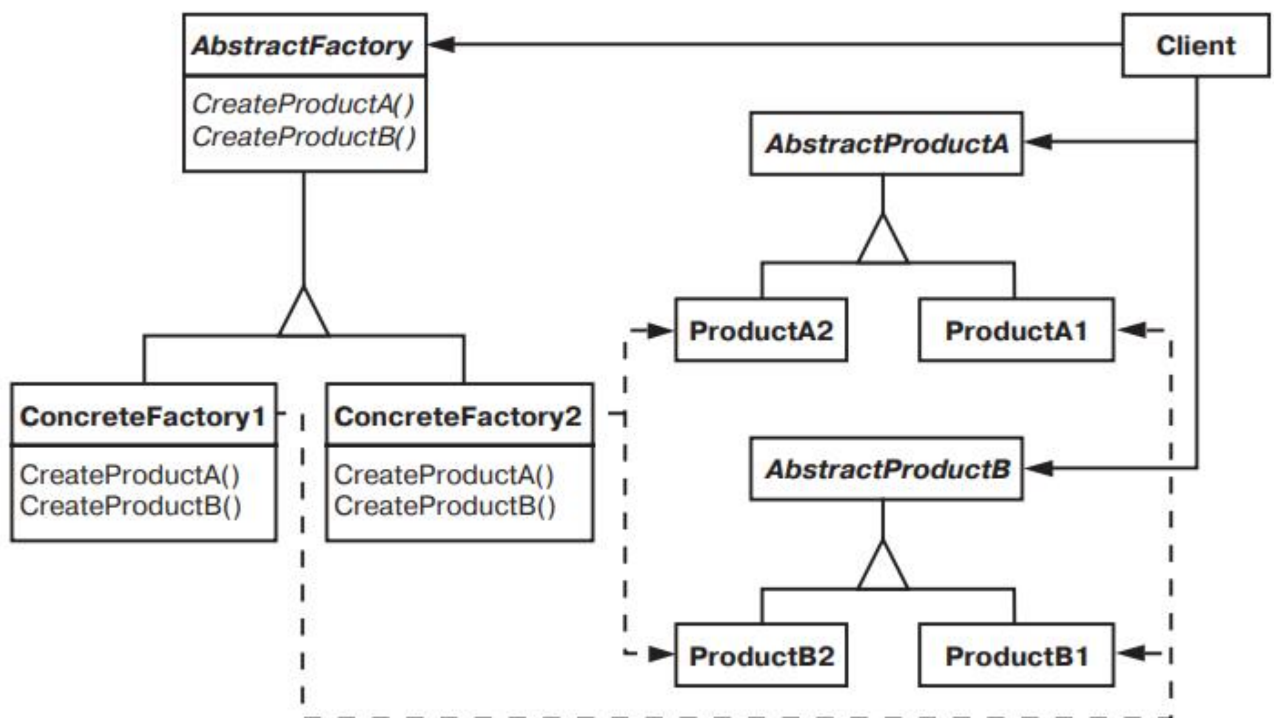

Рисунок 4.1 – Патерн Abstract Factory у вигляді діаграми UML

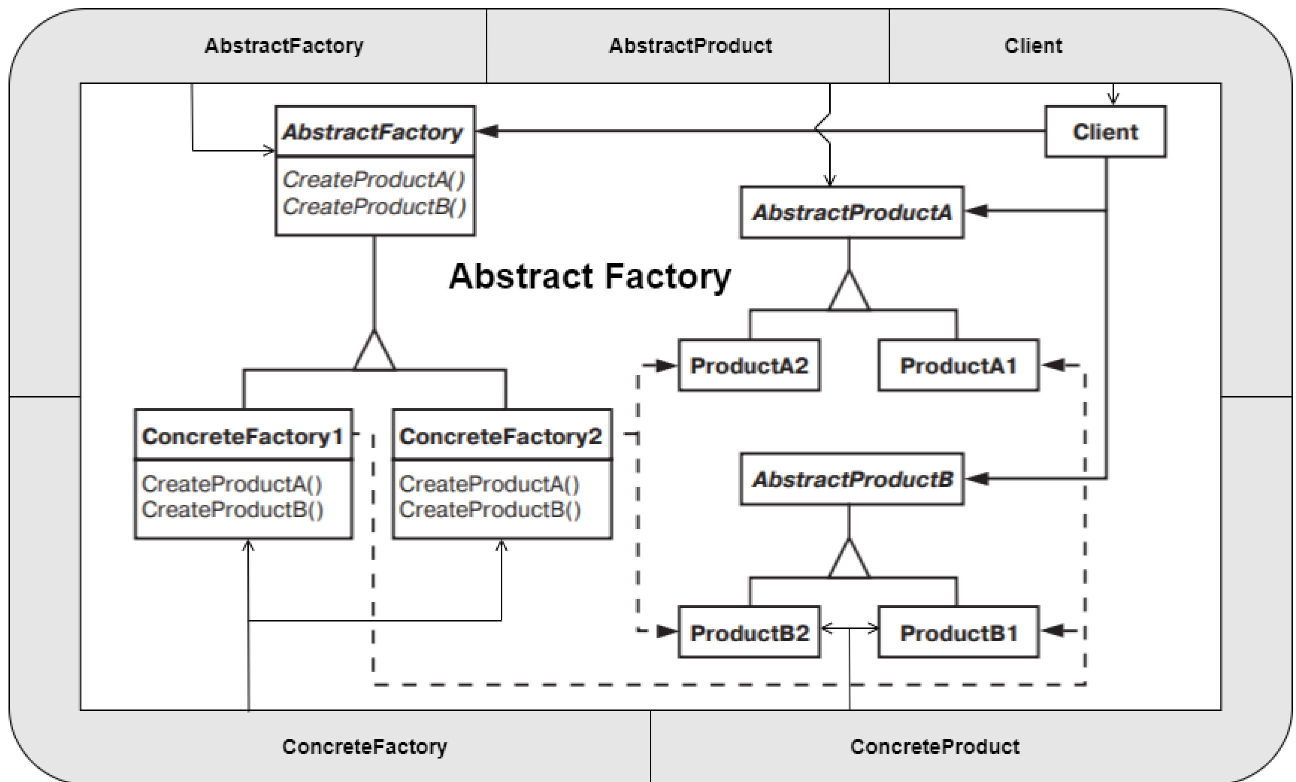

Рисунок 4.2 – Патерн Abstract Factory у вигляді розкритого компонента PINbox та діаграми UML

4.6 Учасники

AbstractFactory – абстрактна фабрика – оголошує інтерфейс для операцій, що створюють абстрактні об’єкти-продукти.

ConcreteFactory – конкретна фабрика – реалізує операції, що створюють конкретні об’єкти-продукти.

AbstractProduct – абстрактний продукт – оголошує інтерфейс підвиду об’єкта-продукта.

ConcreteProduct – конкретний продукт – визначає об’єкт-продукт, який створюється відповідною конкретною фабрикою, реалізує інтерфейс AbstractProduct.

Client – клієнт – користується виключно інтерфейсами, які оголошені у класах AbstractFactory і AbstractProduct.

4.7 Відносини

Зазвичай під час виконання програми створюється єдиний екземпляр класу ConcreteFactory. Ця конкретна фабрика створює об’єкти-продукти з певною реалізацію. Для створення інших видів об’єктів клієнт має скористатися іншою конкретною фабрикою. AbstractFactory делегує створення об’єктів-продуктів своєму підкласу ConcreteFactory.

4.8 Результати

Патерн Abstract Factory має такі переваги і недоліки:

а) ізолює конкретні класи, допомагає контролювати класи об’єктів, створюваних програмою. Оскільки фабрика інкапсулює відповідальність за процес створення об’єктів класів, вона ізолює клієнта від деталей реалізації класів. Клієнти маніпулюють екземплярами через їхні абстрактні класи, а назви конкретних класів, об’єкти яких створює конкретна фабрика, відомі тільки їй, у коді клієнта вони не згадуються;

б) полегшує заміну сімейств продуктів. Клас конкретної фабрики з’являється у програмі лише при створенні об’єкта фабрики. Це полегшує заміну конкретної фабрики, що використовується програмою: можна змінити конфігурацію продуктів, просто підставивши іншу конкретну фабрику. Оскільки абстрактна фабрика створює все сімейство продуктів, заміняється відразу все сімейство;

в) гарантує поєднуваність продуктів. Якщо продукти певного сімейства спроектовані для спільного використання, то важливо, щоб програма в кожний момент працювала лише з продуктами одного сімейства. Патерн Abstract Factory дозволяє дотримуватися цього обмеження;

г) важко додавати новий вид продуктів, тобто розширювати фабрику. Інтерфейс абстрактної фабрики фіксує набір продуктів, які можна створити, для додавання нових продуктів необхідно розширити інтерфейс фабрики, тобто змінити клас AbstractFactory та всі його підкласи.

4.9 Реалізація

Зазвичай програмі потрібен лише один екземпляр класу ConcreteFactory на кожне сімейство продуктів, тому для реалізації найкраще застосовувати патерн Singleton. Він гарантує, що клас має лише один екземпляр, і надає до нього глобальну точку доступу. [3]

5 Вдосконалена реалізація мультиметодів

5.1 Застосування патерна проектування Adapter у реалізації мультиметодів

Патерн проектування Adapter застосовано у реалізації статичного диспетчера таким чином:

а) використано адаптер класу;

б) клас StaticDispatcher виконує у патерні роль Adaptee, адже він має необхідну поведінку, але його публічний інтерфейс варто адаптувати;

в) створено новий абстрактний клас Dispatcher з віртуальною абстрактною функцією Go. Цей абстрактний клас виконує роль Target у патерні, він має такий публічний інтерфейс, як у динамічного диспетчера. Цей публічний інтерфейс вибрано як такий, що більше підходить клієнту. Динамічний диспетчер FnDispatcher тепер наслідує клас Dispatcher;

г) створено новий клас StaticDispatcherAdapted. Він виконує роль Adapter у патерні, отже він публічно наслідує клас Dispatcher і приватно клас StaticDispatcher. Він імплементує віртуальну функцію Go, ця функція викликає статичну функцію Go з адаптованого класу StaticDispatcher;

д) клас StaticDispatcherAdapted містить поле, яке ініціалізується в конструкторі. Це об’єкт класу, у якому реалізовано клієнтські функції. Цей об’єкт потрібно передавати при виклику функції Go з класу StaticDispatcher, але тепер це буде робити клас адаптер StaticDispatcherAdapted внутрішньо, а не клієнтський код.

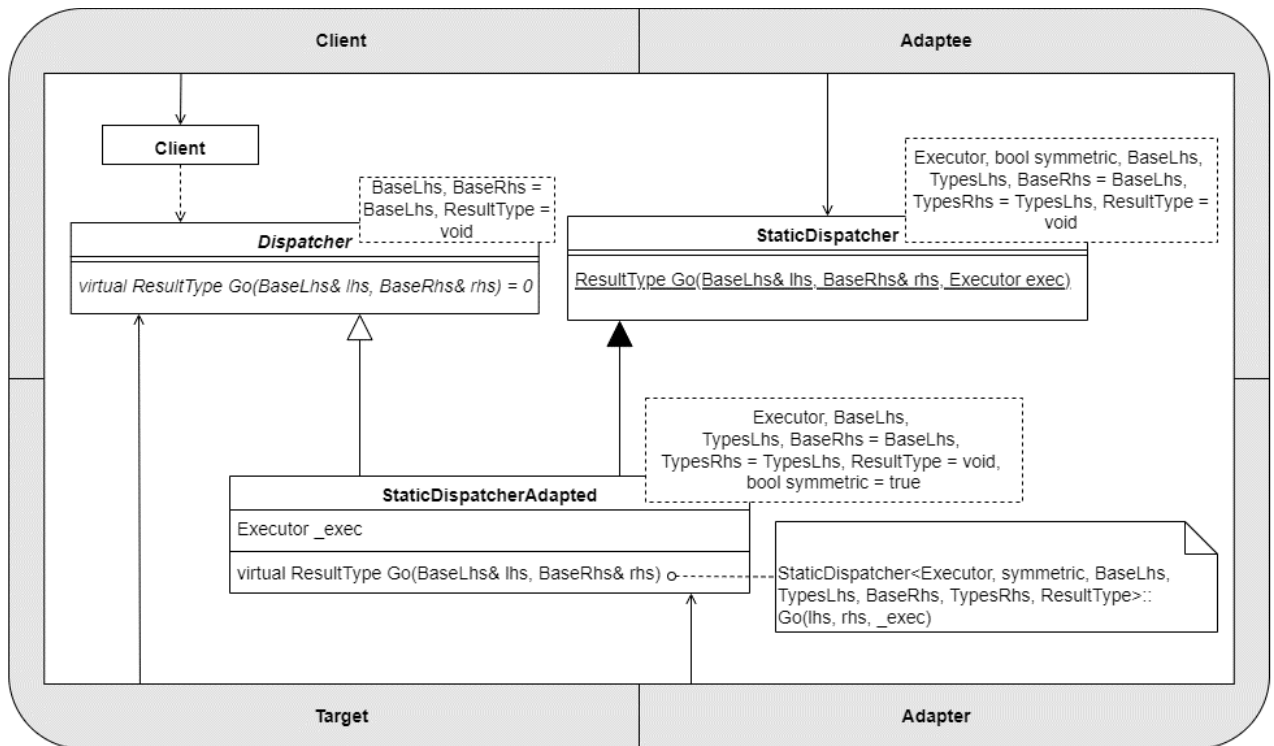

Рисунок 5.1 – Патерн Adapter та статичний диспетчер у вигляді розкритого компонента PINbox та діаграми UML

Завдяки застосуванню патерна проектування Adapter у реалізації статичного диспетчера його публічний інтерфейс не став легшим, але він став таким самим, як у динамічного диспетчера, окрім того, що в об’єкті динамічного диспетчера потрібно реєструвати клієнтські функції. Завдяки однаковому публічному інтерфейсу і спільному базовому класу це вдосконалення вже дає певні переваги:

а) до статичного і динамічного диспетчерів можна застосувати патерн проектування Abstract Factory і зробити їхній публічний інтерфейс легшим;

б) клієнтський код менше, ніж раніше, залежить від того, яка реалізація диспетчера використовується, тому можна значно легше замінити використання статичного диспетчера на використання динамічного або навпаки. Наприклад, можна передавати об’єкт статичного або динамічного диспетчера як аргумент у метод;

в) можна помістити об’єкти різних диспетчерів у одну структуру даних, наприклад масив, і у циклі викликати функцію для цих об’єктів.

Приклад оголошення статичного диспетчера після застосування патерна Adapter:

```
using FirstStaticDispatcherAdapted =
StaticDispatcherAdapted<HatchingExecutor, Shape, TypeList<Rectangle,
Ellipse, Poly, NullType>, Shape, TypeList<Rectangle, Ellipse, Poly,
NullType>, int>;
```

Як і раніше, потрібно передавати класу диспетчера багато шаблонних параметрів, вони такі, як було описано вище, але змінений їхній порядок, щоб можна було задати більше значень шаблонних параметрів за замовчуванням: шаблонний булевий аргумент – true, тобто диспетчер симетричний за замовчуванням. Оголошення статичного диспетчера з використанням значень за замовчуванням деяких шаблонних параметрів:

```
using SecondStaticDispatcherAdapted =
StaticDispatcherAdapted<HatchingExecutor, Shape, TypeList<Rectangle,
Ellipse, Poly, NullType>>;
```

Також потрібно створити об’єкт статичного диспетчера, бо тепер його функції нестатичні. Конструктор приймає один аргумент – об’єкт класу, у якому реалізовано перевантажені клієнтські функції-виконавці:

```
HatchingExecutor exec;
FirstStaticDispatcherAdapted disp(exec);
```

Приклад виклику головної функції статичного диспетчера, яка здійснює диспетчеризацію, є тепер нестатичною і приймає два аргументи, як і в динамічного диспетчера:

```
disp.Go(*p2ellipse, *p3poly);
```

Завдяки застосуванню патерна проектування Adapter можна помістити об’єкти статичного і динамічного диспетчерів у одну структуру даних, наприклад масив, і у циклі викликати функцію для цих об’єктів:

```
SDAdaptedForArray* staticDispatcher = new SDAdaptedForArray(exec);
FnDispatcherForArray* fnDispatcher = new FnDispatcherForArray();
Dispatcher<Shape, Shape, int>* dispatchers[2] =
      { staticDispatcher, fnDispatcher };
for (int i = 0; i < 2; i++) {
```

```
	dispatchers[i]->Go(*rectangle, *poly);
}
```

Така можливість може знадобитись наприклад для того, щоб порівняти поведінку кількох різних типів диспетчерів.

5.2 Застосування патерна проектування Abstract Factory у реалізації мультиметодів

Патерн проектування Abstract Factory застосовано у реалізації статичного та динамічного диспетчера таким чином:

а) абстрактний клас Dispatcher виконує роль AbstractProduct;

б) класи StaticDispatcherAdapted і FnDispatcher виконують ролі ConcreteProduct;

в) створено абстрактний клас DispatcherFactory, який у патерні виконує роль AbstractFactory і містить віртуальну абстрактну функцію CreateDispatcher, реалізації якої мають створювати та повертати об’єкт типу Dispatcher;

г) створено клас StaticDispatcherFactory, похідний від DispatcherFactory, який виконує роль ConcreteFactory і реалізує функцію CreateDispatcher. Ця фабрика створює об’єкти типу StaticDispatcherAdapted – статичного диспетчера із застосуванням патерна Adapter, передає при створенні всі необхідні шаблонні параметри, створює об’єкт класу, у якому реалізовано перевантажені функції-виконавці, передає цей об’єкт конструктору класу StaticDispatcherAdapted. Таким чином, ця фабрика створює об’єкти статичного диспетчера з конкретними шаблонними параметрами для певної ієрархії класів і функцій-виконавців, і це не можна змінити, не змінюючи реалізації фабрики;

д) створено конкретну фабрику FnDispatcherFactory, яка створює об’єкти типу FnDispatcher – динамічного диспетчера, передає при створенні шаблонні параметри, а також реєструє клієнтські функції в об’єкті диспетчера. Ці функції визначені у безіменному просторі імен;

е) створено конкретну фабрику FnFastDispatcherStaticCastingFactory, яка створює об’єкти типу FnDispatcher – динамічного диспетчера, передає при створенні шаблонні параметри, реєструє клієнтські функції в об’єкті диспетчера;

є) конкретні фабрики відрізняються одна від одної тим, що вони по-різному створюють і налаштовують об’єкти диспетчерів, тобто передають їм різні шаблонні параметри, ієрархії класів, функції-виконавці. Як наслідок, диспетчери, створені різними фабриками, по-різному поводяться;

ж) всі конкретні фабрики реалізовано як Singleton;

з) можна створювати ще конкретні фабрики, які будуть породжувати об’єкти диспетчерів з іншими шаблонними параметрами, можливо, для іншої ієрархії класів, з іншими клієнтськими функціями.

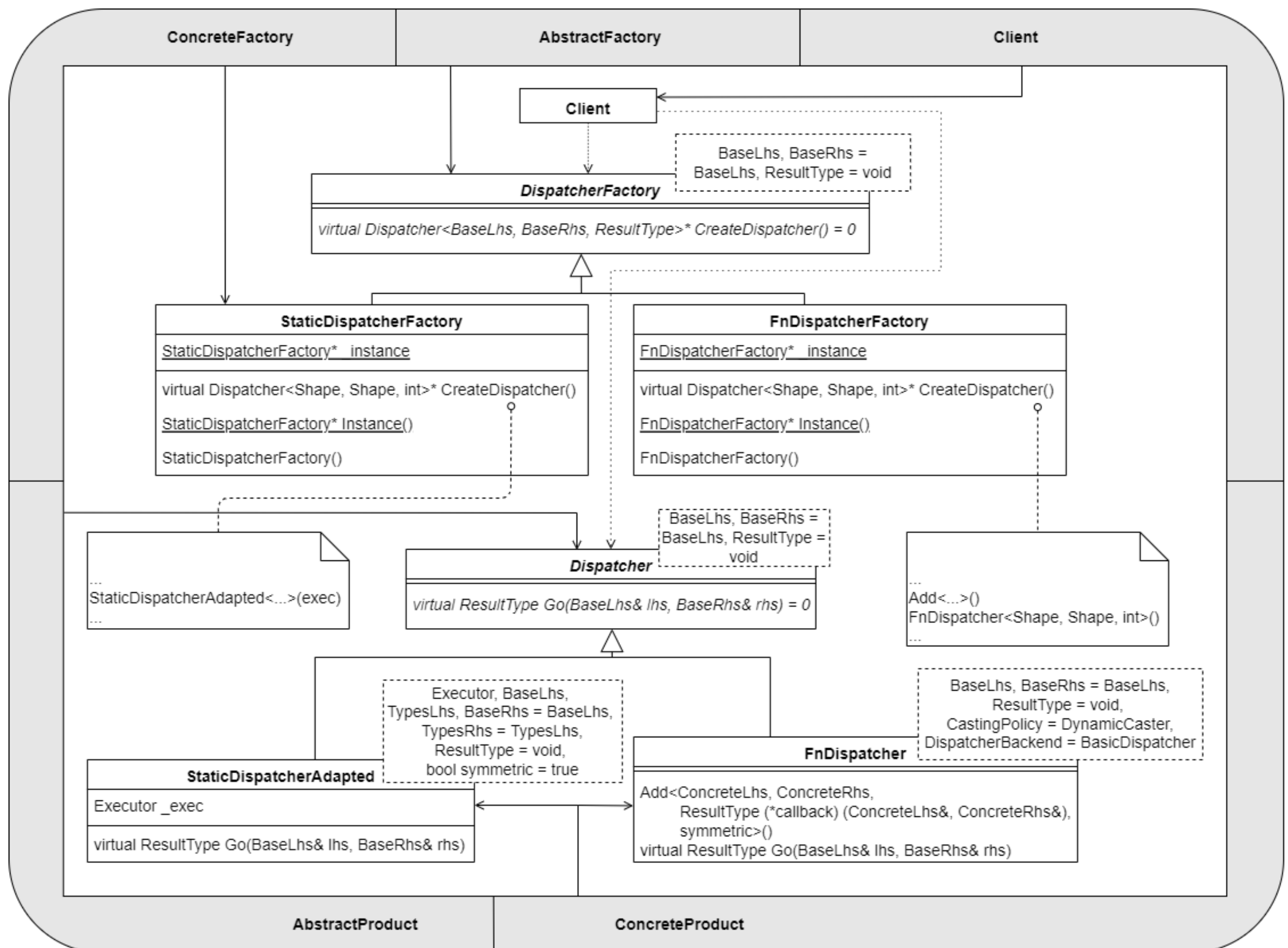

Рисунок 5.2 – Патерн Abstract Factory та статичний диспетчер у вигляді розкритого компонента PINbox та діаграми UML

У цьому випадку застосування патерна Abstract Factory кожна фабрика створює не сімейство взаємопов’язаних продуктів, а лише один вид об’єктів,

тому тут немає частини переваг патерна, описаних вище, проте він суттєво покращує реалізацію мультиметодів. Завдяки застосуванню патерна Abstract Factory у реалізації диспетчерів їхній публічний інтерфейс став значно легшим для використання, зокрема для створення об'єктів, адже всі потрібні налаштування здійснює фабрика, і вони не ускладнюють клієнтський код.

Приклад оголошення статичного диспетчера після застосування патерна Abstract Factory:

```
DispatcherFactory<Shape, Shape, int>* factory =
StaticDispatcherFactory::Instance();
Dispatcher<Shape, Shape, int>* dispatcher = factory->CreateDispatcher();
```

У цьому прикладі спочатку створено або отримано наявний об'єкт конкретної фабрики, реалізованої як Singleton. Після цього з об'єкта фабрики отримано створений об'єкт диспетчера, який не потребує більше ніяких налаштувань.

Виклик головної функції статичного диспетчера, яка здійснює диспетчеризацію:

```
dispatcher->Go(*p1rectangle, *p3poly);
```

До вдосконалення реалізації статичного диспетчера, його потрібно було оголошувати і використовувати таким чином:

```
using FirstStaticDispatcher = StaticDispatcher<HatchingExecutor, true,
Shape, TypeList<Rectangle, Ellipse, Poly, NullType>, Shape,
TypeList<Rectangle, Ellipse, Poly, NullType>, int>;
HatchingExecutor exec;
FirstStaticDispatcher::Go(*rectangle, *poly, exec);
```

У цьому прикладі видно, що до вдосконалення публічний інтерфейс статичного диспетчера був важчим і відрізнявся від публічного інтерфейсу динамічного диспетчера.

Щоб використати інший диспетчер, потрібно використати іншу конкретну фабрику:

```
DispatcherFactory<Shape, Shape, int>* factory =
FnFastDispatcherStaticCastingFactory::Instance();
Dispatcher<Shape, Shape, int>* dispatcher = factory->CreateDispatcher();
```

```
dispatcher->Go(*p1rectangle, *p3poly);
```

Цей приклад відрізняється від наведеного вище прикладу оголошення статичного диспетчера лише назвою класу конкретної фабрики, яка з’являється у коді лише раз, тобто її легко замінити. Отримування об’єкта диспетчера і виклик головної функції диспетчера відбувається так само, як і з інших конкретних фабрик диспетчерів. Окрім того, у отриманому об’єкті динамічного диспетчера вже зареєстровані функції-виконавці, а до застосування патерна Abstract Factory це потрібно було робити у клієнтському коді. Це давало широкі можливості, але ускладнювало публічний інтерфейс: динамічний диспетчер потрібно було оголошувати і використовувати таким чином:

```
FnDispatcher<Shape, Shape, int>* dispatcher = new FnDispatcher<Shape,
Shape, int>();
dispatcher->Add<Rectangle, Poly, HatchRectanglePoly, true>();
dispatcher->Add<Ellipse, Poly, HatchEllipsePoly, true>();
dispatcher->Add<Poly, Poly, HatchPolyPoly, true>();
dispatcher->Go(*p1rectangle, *p3poly);
```

Кожна конкретна фабрика створює об’єкти диспетчера з однаковими налаштуваннями: з однаковими шаблонними параметрами, для однієї ієрархії класів і набору функцій-виконавців. У клієнтському коді доводиться використовувати одну з реалізованих конкретних фабрик, а якщо потрібен якийсь інший варіант диспетчера, можна реалізувати ще одну конкретну фабрику. Таким чином, після застосування патерна Abstract Factory, реалізація мультиметодів стала менш гнучкою, важче створити довільний диспетчер, для якого немає конкретної фабрики, але публічний інтерфейс створення об’єктів диспетчерів став легшим. Якщо один раз правильно реалізувати конкретну фабрику, потім її зручно використовувати для створення конкретного виду диспетчерів. Як результат, легше використовувати диспетчери, здійснювати перевірку коду і пошук помилок, заміняти об’єкти різних реалізацій диспетчерів.

Висновки

У попередній курсовій роботі було досліджено і описано подвійну диспетчеризацію і її реалізації з книги Андрія Александреску «Сучасне проектування на C++»: статичний і динамічний диспетчер. Ці реалізації диспетчерів гнучкі, надають користувачеві багато можливостей, але у них досить важкий публічний інтерфейс, а також різний у статичного і динамічного диспетчерів. У цій роботі було виправлено цей недолік шляхом застосування патернів проектування Adapter і Abstract Factory.

Патерн Adapter перетворює інтерфейс одного класу на інтерфейс іншого, який потрібен клієнту, робить можливою спільну роботу класів із несумісними інтерфейсами. У роботі Adapter було застосовано в реалізації статичного диспетчера, щоб адаптувати його публічний інтерфейс і зробити таким, як у динамічного диспетчера. Завдяки цьому стало можливо застосувати до статичного і динамічного диспетчера патерн проектування Abstract Factory, у якому диспетчери виконуватимуть роль конкретних продуктів зі спільним базовим класом, що виконуватиме роль абстрактного продукту. Також завдяки однаковому публічному інтерфейсу диспетчерів у вдосконаленій реалізації легше замінити використання статичного диспетчера на використання динамічного або навпаки.

Патерн Abstract Factory надає інтерфейс для створення сімейств взаємопов’язаних чи взаємозалежних об’єктів, не специфікуючи їх конкретних класів, інкапсулює відповідальність за процес створення об’єктів класів, ізолює клієнта від деталей реалізації класів. При застосуванні цього патерна клієнти маніпулюють екземплярами через їхні абстрактні класи, а назви конкретних класів відомі тільки конкретній фабриці.

У цій роботі патерн Abstract Factory використаний для того, щоб конкретні фабрики створювали і налаштовували об’єкти диспетчерів. Завдяки застосуванню патерна публічний інтерфейс диспетчерів став значно легшим для використання, адже всі потрібні налаштування при створенні об’єкта диспетчера

здійснює фабрика, і вони не ускладнюють клієнтський код. Як результат, легше використовувати диспетчери, здійснювати перевірку коду і пошук помилок. Також після вдосконалення реалізації мультиметодів клієнтський код менше залежить від того, яка саме реалізація диспетчера використовується, об'єкти різних реалізацій диспетчерів можна легше заміняти, передавати як аргументи, використовувати як стратегії, додавати у одну структуру даних і опрацьовувати її у циклі, наприклад, щоб порівняти поведінку кількох різних типів диспетчерів.

У роботі було описано застосовані патерни проектування Adapter і Abstract Factory за допомогою засобів формалізації, описаних у книгах Gang of Four (GoF) «Шаблони проектування: Елементи повторно використовуваного об'єктно-орієнтованого програмного забезпечення» та Джейсона Мак-Колм Сміта «Елементарні шаблони проектування»: специфікацій та системи Pattern Instance Notation (PIN). Також було описано застосування вибраних патернів проектування у реалізації мультиметодів і переваги отриманих результатів.

Було створено програмний продукт: вдосконалена реалізація статичної і динамічної подвійної диспетчеризації із застосуванням патернів проектування Adapter і Abstract Factory; приклад застосування – диспетчеризація викликів функцій для ієрархії класів з базовим класом Shape і трьома похідними.

Перспективи покращення досягнутих результатів:

а) реалізувати множинну диспетчеризацію для більшої кількості об'єктів.

Список літератури

1. Александреску А. Современное проектирование на C++. Издательский дом «Вильямс», 2002. 336 с.

2. Смит Джейсон Мак-Колм. Элементарные шаблоны проектирования. ООО “И.Д. Вильямс”, 2013. 304 с.

3. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. СПб: Питер, 2001. 368 с.

4. Nystrom Robert. Game Programming Patterns. Genever Benning, 2014. 354 с.

5. Design Patterns – Adapter Pattern. URL: https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm# (дата звернення: 08.06.2022).

6. Design Pattern – Abstract Factory Pattern. URL: https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm (дата звернення: 10.06.2022).

7. What is Unified Modeling Language (UML)? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/ (дата звернення: 20.06.2022).