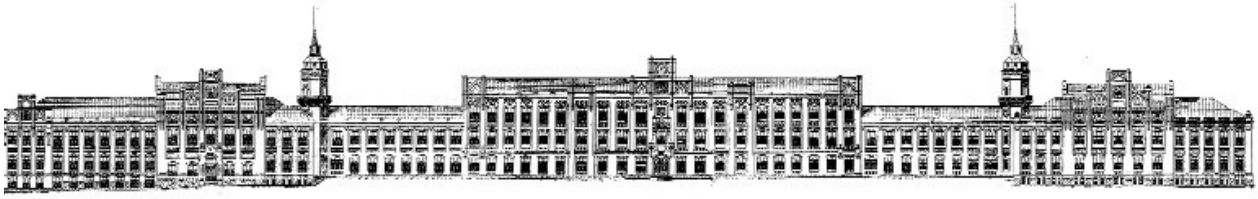


Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6
Тема: «Патерни проектування»

Виконала
студентка групи ІА-32:
Іванова Анастасія
Юріївна

Перевірив:
Мягкий Михайло
Юрійович

Зміст

Тема проекту

Варіант: 26
Опис теми: Download manager (iterator, command, observer, template method, composite, p2p) Інструмент для скачування файлів з інтернету по протоколах http або https з можливістю продовження завантаження в зупиненому місці, розподілу швидкостей активним завантаженням, ведення статистики завантажень, інтеграції в основні браузери (firefox, opera, internet explorer, chrome).

Теоретичні відомості

Патерни проєктування (Design Patterns) — це типові, перевірені на практиці способи розв'язання поширених проблем, які виникають під час проєктування програмного забезпечення.
Вони описують загальні принципи побудови структури класів і взаємодії об'єктів, не прив'язуючись до конкретної мови програмування. Патерни не є готовим кодом — це шаблони, за якими можна побудувати гнучке, масштабоване й зрозуміле рішення.
Використання патернів дозволяє:

- підвищити гнучкість і розширюваність системи;
- зменшити дублювання коду й спростити супровід;
- покращити зрозумілість архітектури для інших розробників;
- забезпечити повторне використання перевірених рішень у різних проєктах.

Observer (Спостерігач) — це поведінковий шаблон проєктування, що визначає залежність типу «один-до-багатьох» між об'єктами. Його суть у тому, що коли один об'єкт (називається Subject) змінює свій стан, усі об'єкти, які на нього підписані (Observers), отримують сповіщення і можуть реагувати на зміну стану.
Такий шаблон дозволяє реалізувати механізм підписки/розсилки повідомлень (publish–subscribe), де спостерігачі можуть у будь-який момент підписатися або відписатися від джерела подій.

Основні елементи

- **Subject** — об'єкт, за яким спостерігають; містить список підписників і надсилає їм сповіщення.
- **Observer** — інтерфейс для всіх спостерігачів, які реагують на оновлення.
- **ConcreteSubject** — конкретна реалізація Subject, що зберігає стан.
- **ConcreteObserver** — спостерігачі, які змінюють свій стан після отримання сповіщення.

Переваги

- Підтримує асинхронність та паралельність.
- Дає змогу легко додавати і видаляти спостерігачів.
- Послаблює зв'язність між компонентами.

- Підходить для UI, реактивних систем, обробки подій.

Недоліки

- Немає гарантії порядку оповіщення спостерігачів.
- Велика кількість підписників може створити навантаження на систему.
- Складно відслідковувати ланцюгові реакції в складних системах.

Хід роботи

Створення діаграми класів патерну

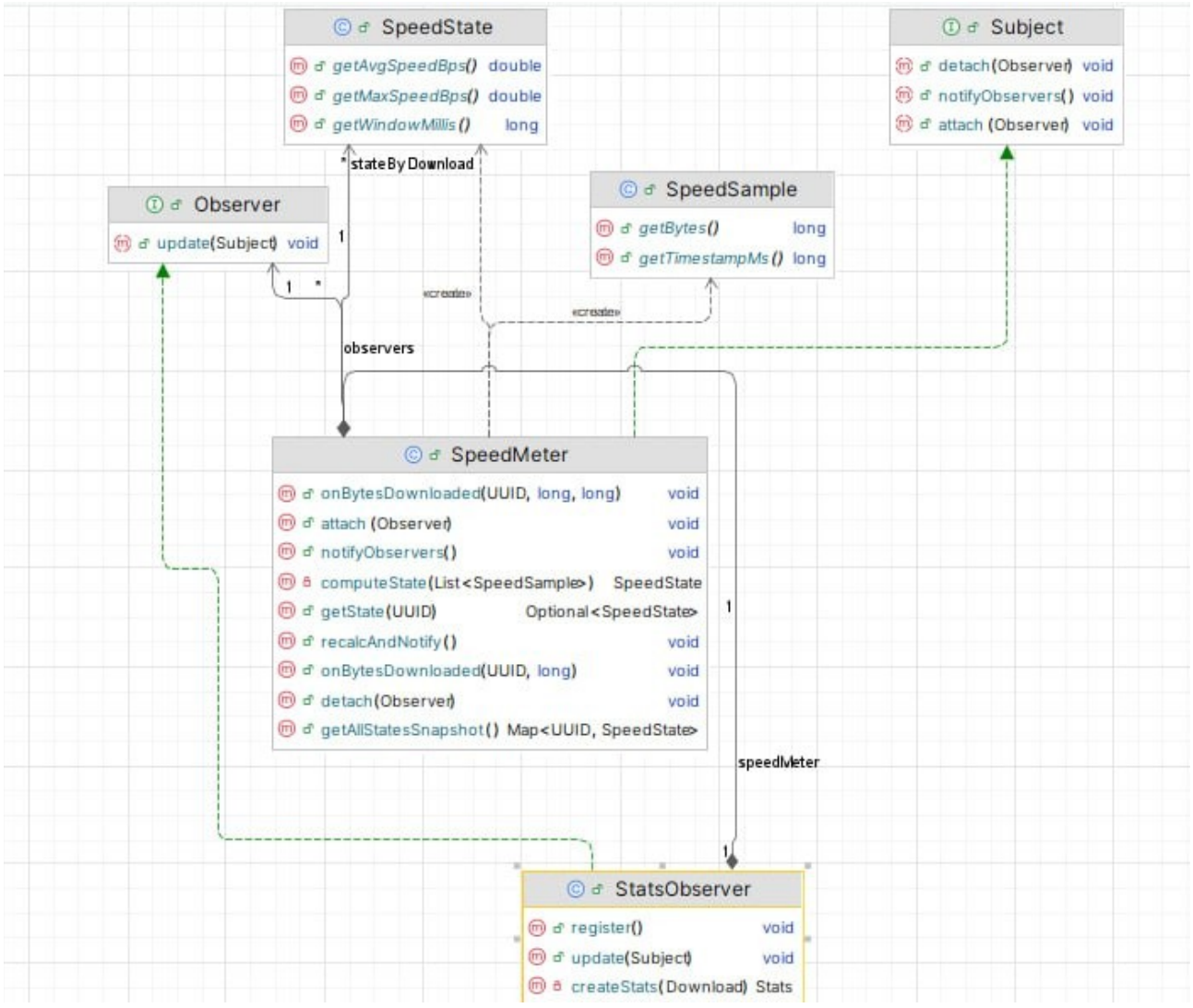


Рис. 1. Діаграма класів патерну Observer

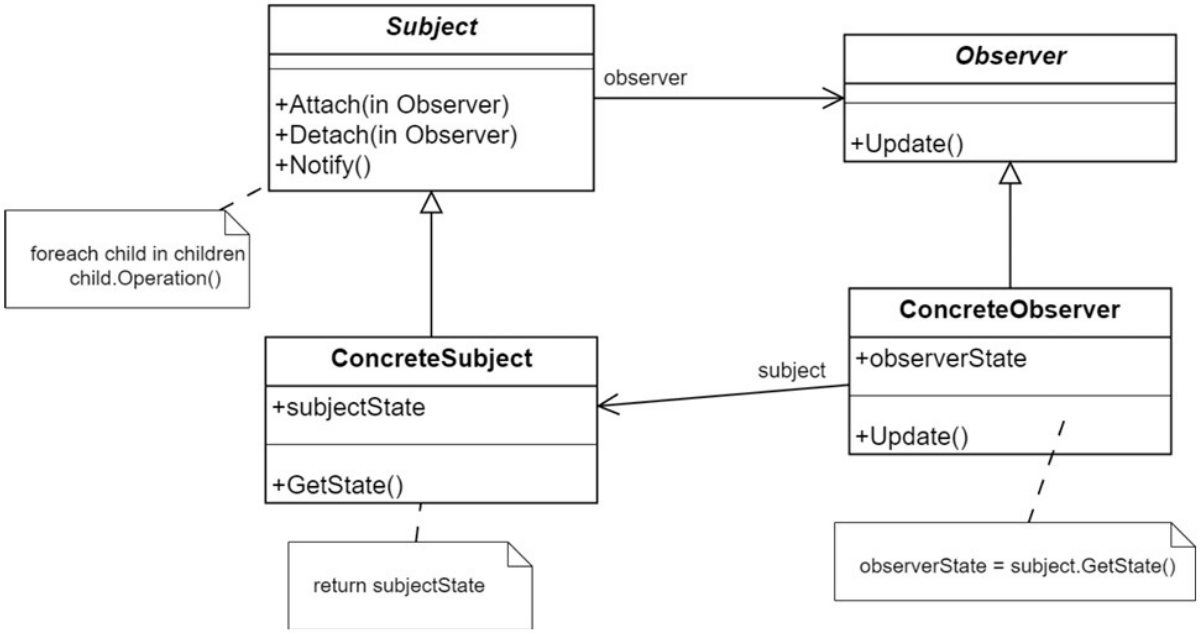


Рис. 2. Шаблон Observer

1. Subject → Subject (інтерфейс)
Задає контракт для всіх класів, які можуть виступати в ролі об’єкта-джерела подій у патерні Observer.
Фрагмент коду:

```
public interface Subject {
   void attach(Observer observer);
   void detach(Observer observer);
   void notifyObservers();
}
```

Роль:
Subject визначає базові методи - **Attach(), Detach() та Notify()**, які дозволяють додавати/видаляти спостерігачів та повідомляти їх про зміни стану.
У моєму коді цей інтерфейс є основою спостереження: будь-який клас, що хоче бути "спостережуваним", повинен його реалізувати.
Його реалізує SpeedMeter, тому саме SpeedMeter виступає ConcreteSubject, а інтерфейс Subject — це узагальнена модель поведінки з діаграми патерну.

2. Observer → Observer (інтерфейс підписника)
Описує загальну поведінку всіх об’єктів, що мають реагувати на зміни у Subject.
Фрагмент коду:

```
public interface Observer {
   void update(Subject subject);
}
```

Роль:
Задає вимогу: будь-який "спостерігач" повинен реалізувати метод update() і вміти реагувати на зміну стану у підписаного Subject.

3. ConcreteSubject → SpeedMeter (конкретна реалізація Subject)
Зберігає стан (швидкість завантажень) і надає доступ до нього спостерігачам.
Фрагмент коду:

```
@Component
public class SpeedMeter implements Subject {
   private final List<Observer> observers = new CopyOnWriteArrayList<>();

   private final Map<UUID, List<SpeedSample>> samplesByDownload = new
ConcurrentHashMap<>();
   private final Map<UUID, SpeedState> stateByDownload = new
ConcurrentHashMap<>();

   @Override
   public void attach(Observer observer) { ... }
   @Override
   public void detach(Observer observer) { ... }
   @Override
   public void notifyObservers() { ... }

   // "GetState" у конкретного сабджекта:
   public Optional<SpeedState> getState(UUID downloadId) { ... }
   public Map<UUID, SpeedState> getAllStatesSnapshot() { ... }

   @Scheduled(fixedDelay = 1000)
   public void recalcAndNotify() {
      // 1) перерахунок avg/max за вікно
      // 2) виклик notifyObservers();
   }
}
```

Роль:
Зберігає та оновлює дані про швидкість. Раз на секунду перераховує статистику та повідомляє всіх Observer про зміни.

4. ConcreteObserver → StatsObserver
Отримує повідомлення від SpeedMeter та оновлює статистику в базі.
Фрагмент коду:

```
@Component
public class StatsObserver implements Observer {

   @PostConstruct
   public void register() {
      speedMeter.attach(this);
   }

   @Override
   public void update(Subject subject) {
      SpeedMeter meter = (SpeedMeter) subject;
```

```
        meter.getAllStatesSnapshot().forEach((id, state) -> {
            Stats s = statsRepo.findById(id).orElseGet(() ->
createStats(downloadRepo.getReferenceById(id)));
            s.setAvgSpeedBps(state.getAvgSpeedBps());
            s.setMaxSpeedBps(Math.max(s.getMaxSpeedBps(),
state.getMaxSpeedBps()));
            statsRepo.save(s);
        });
    }
}
```

Роль:

Підписаний на SpeedMeter. Кожного разу, коли швидкість змінюється, отримує оновлення, зчитує новий стан і записує його в БД (avgSpeed, maxSpeed). Реагує на зміни, але не впливає на логіку вимірювання швидкості.

Робота коду

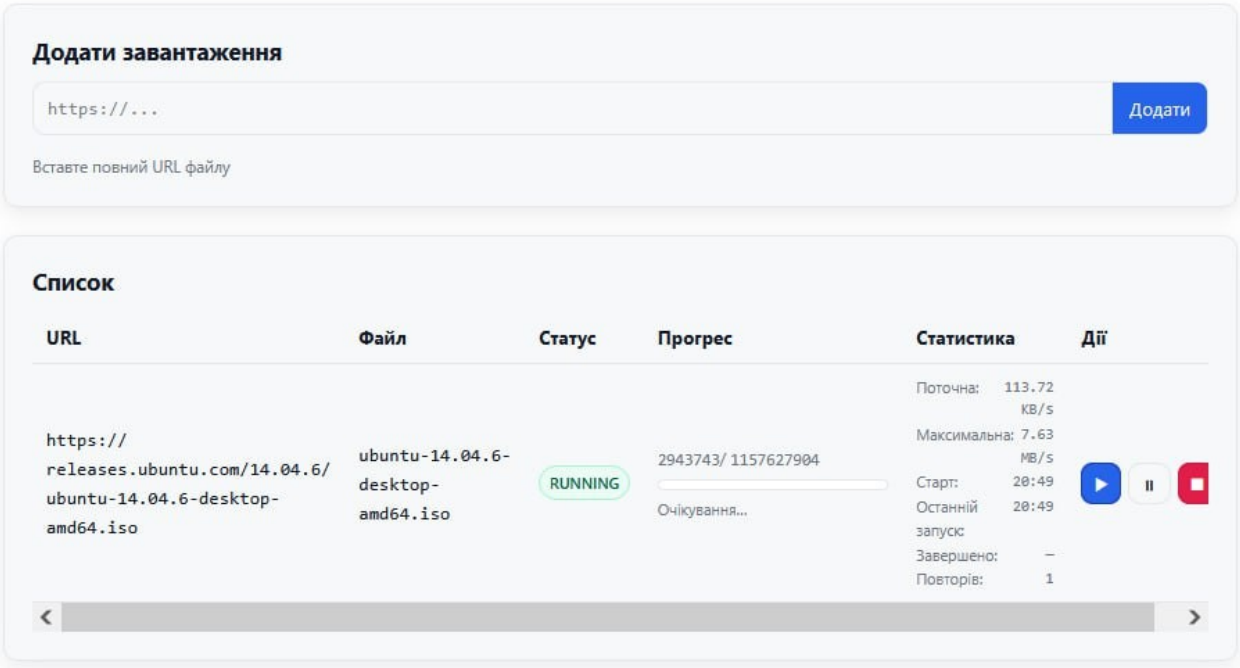


Рис. 3. Результат роботи патерну Observer (на прикладі активного завантаження на скріншоті)

На рисунку зображене активне завантаження файлу у менеджері. Дані, що відображаються у полі «Статистика» (поточна швидкість, максимальна швидкість, кількість повторних запусків та часові мітки), генеруються та оновлюються автоматично завдяки роботі патерну Observer.

Під час завантаження сегменти файлу передають у SpeedMeter інформацію про отримані байти. SpeedMeter виступає джерелом подій - він періодично обчислює швидкість завантаження та викликає notifyObservers(). На це сповіщення реагує StatsObserver (ConcreteObserver), який отримує оновлені дані

та записує їх у базу. Після цього оновлена статистика відображається на інтерфейсі користувача.

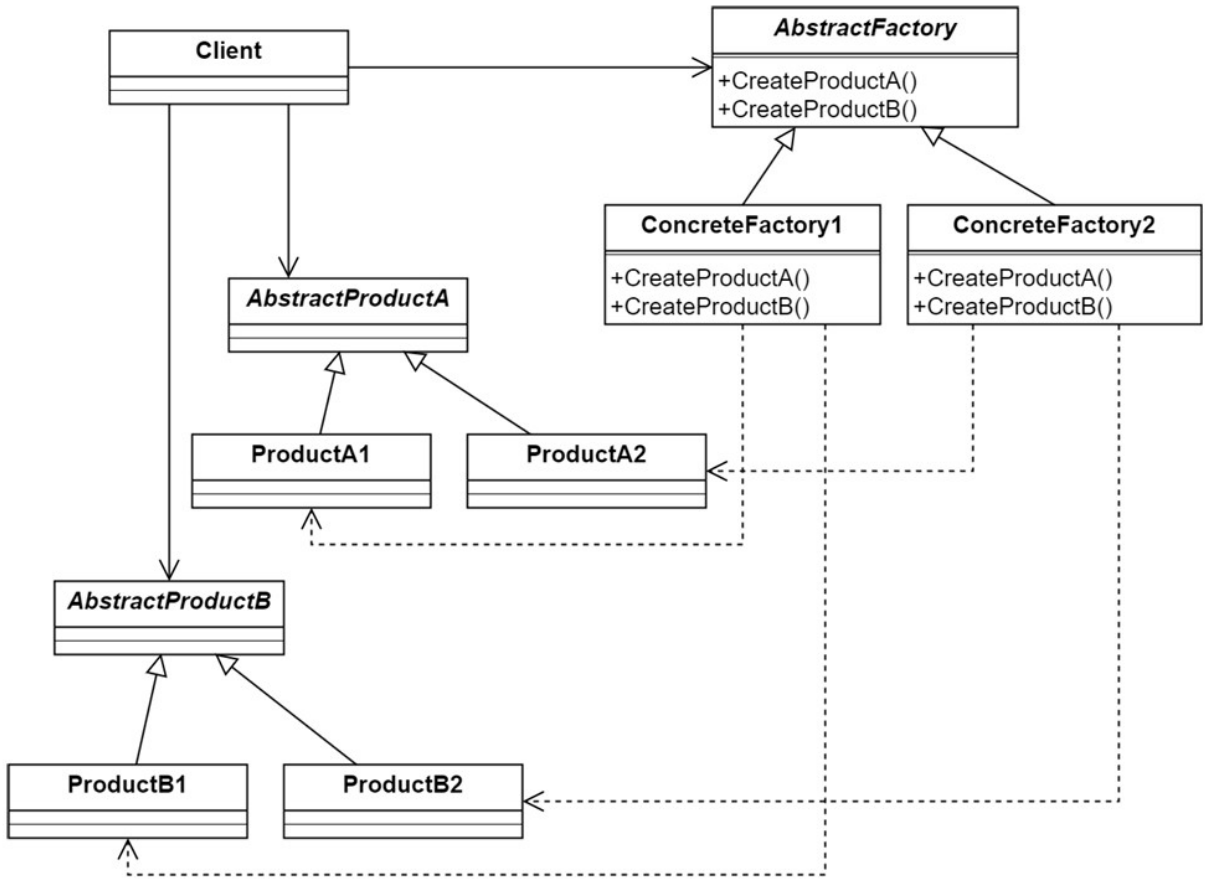
Висновок: патерн Observer у проєкті застосовано для організації реактивної взаємодії між компонентами системи – процес вимірювання швидкості завантаження відокремлений від способу обробки та збереження цих даних. Завдяки цьому логіка відстеження зміни стану не залежить від логіки реагування на ці зміни, що суттєво знижує зв’язність коду та підвищує гнучкість архітектури.
У моєму застосунку SpeedMeter є джерелом подій, який вимірює швидкість і надсилає повідомлення всім підписаним об’єктам.
StatsObserver є "спостерігачем", який отримує ці сповіщення, забирає поточні значення швидкості та оновлює статистику в БД.

Контрольні питання

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» використовується для створення сімейств взаємопов’язаних об’єктів без вказування їх конкретних класів. Він забезпечує узгодженість об’єктів одного стилю та дозволяє легко змінювати варіанти реалізації цілої групи продуктів.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

Шаблон містить такі ключові елементи:

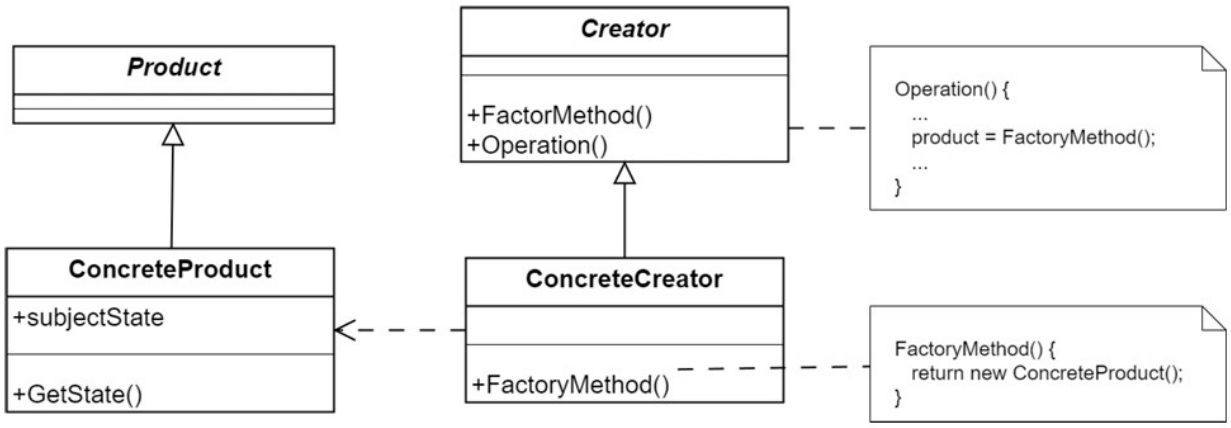
- **AbstractFactory** — оголошує інтерфейс створення продуктів.
- **ConcreteFactory** — створює конкретні об’єкти певного сімейства.
- **AbstractProduct** — інтерфейс продукту.
- **ConcreteProduct** — конкретні реалізації продуктів.
- **Client** — працює лише з інтерфейсами фабрики та продуктів.

Взаємодія: клієнт передає роботу фабриці, яка повертає потрібні об’єкти одного стилю.

4. Яке призначення шаблону «Фабричний метод»?

Шаблон «Фабричний метод» визначає інтерфейс для створення об’єктів певного типу і дозволяє підкласам вирішувати, який саме клас створювати. Він розв’язує проблему створення об’єктів-підтипів без зміни клієнтського коду.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

- **Creator** — містить фабричний метод.
- **ConcreteCreator** — реалізує фабричний метод для конкретного класу продукту.
- **Product** — базовий інтерфейс продукту.
- **ConcreteProduct** — конкретний продукт.

Creator викликає фабричний метод, ConcreteCreator створює потрібний ConcreteProduct.

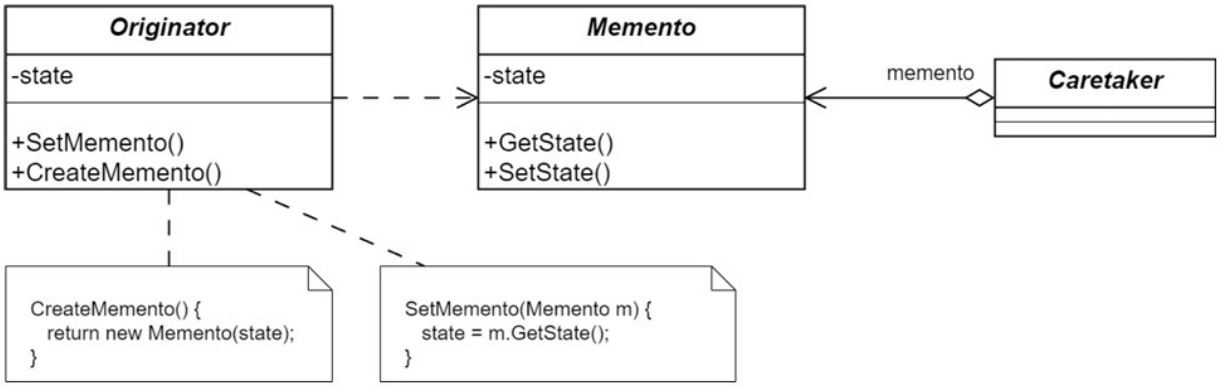
7. Чим відрізняється «Абстрактна фабрика» від «Фабричного методу»?

Абстрактна фабрика	**Фабричний метод**
Створює сімейства пов’язаних об’єктів	Створює один об’єкт певного типу
Часто містить багато методів створення	Містить один фабричний метод

Забезпечує узгодженість стилів	Забезпечує можливість підміни реалізації

8. Яке призначення шаблону «Знімок» (Memento)?
Патерн дозволяє зберігати і відновлювати стан об’єкта, не порушуючи інкапсуляції.
Стан зберігається в окремому об’єкті Memento, який доступний тільки тому об’єкту, що його створив.

9. Нарисуйте структуру шаблону «Знімок».

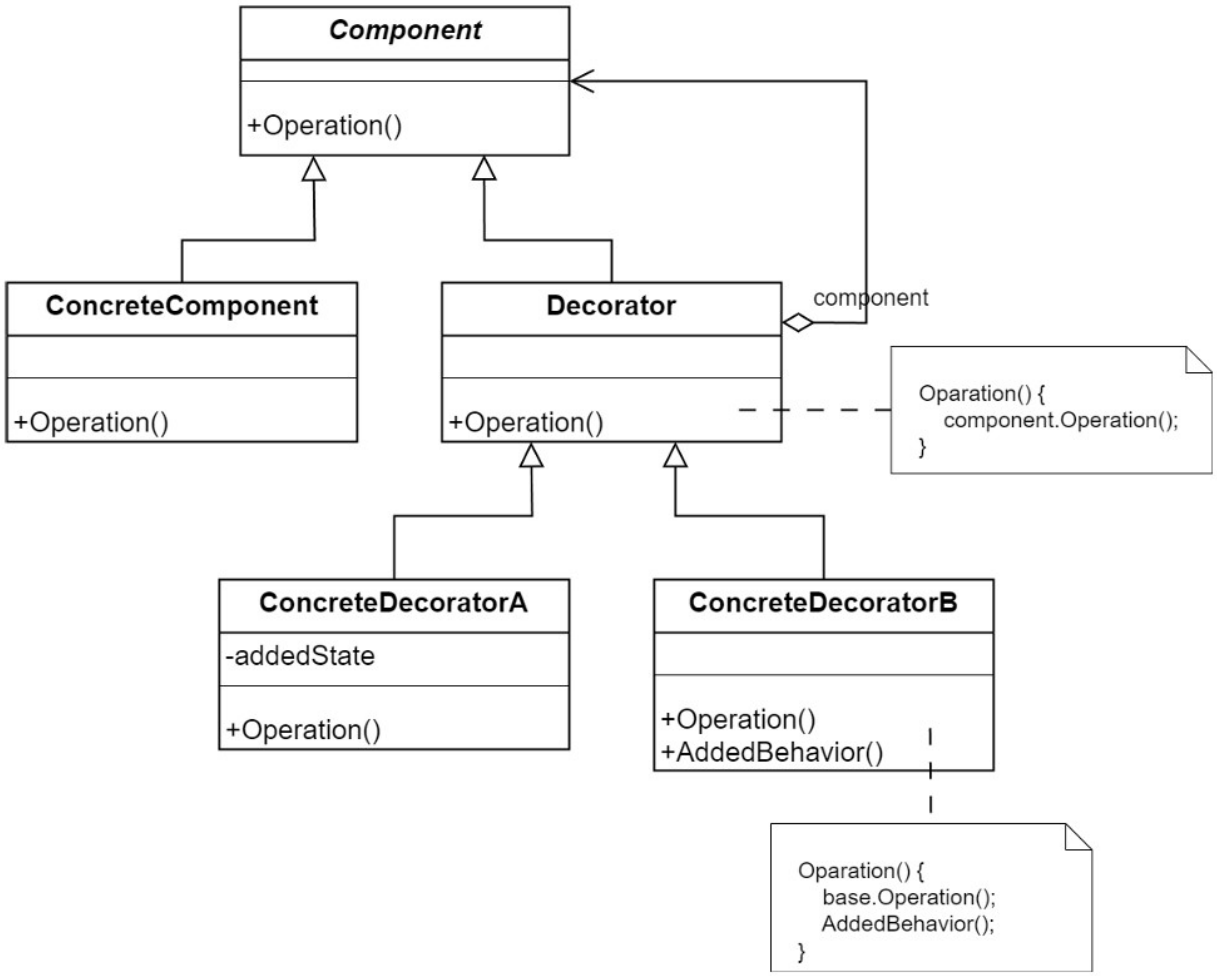


10. Які класи входять у шаблон «Знімок», та яка між ними взаємодія?
- **Originator** — створює та відновлює власний стан.
- **Memento** — зберігає стан.
- **Caretaker** — зберігає Memento, але не має доступу до його внутрішнього стану.

Originator створює Memento, Caretaker тимчасово зберігає його, пізніше Originator витягує стан назад.

11. Яке призначення шаблону «Декоратор»?
Шаблон дозволяє динамічно додавати нову поведінку або відповідальність об’єктам, «обгортаючи» їх у додаткові класи-декоратори. Це альтернатива спадкуванню, але більш гнучка.

12. Нарисуйте структуру шаблону «Декоратор».



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

- **Component** — базовий інтерфейс.
- **ConcreteComponent** — початковий клас.
- **Decorator** — містить посилання на Component.
- **ConcreteDecorator** — додає нову поведінку.

Decorator делегує виклик початковому об’єкту та додає додаткові функції.

14. Які є обмеження використання шаблону «Декоратор»?

Створюється багато дрібних класів, що ускладнює структуру.
Важко керувати об’єктами, обгорнутими в кілька декораторів одночасно.
Може ускладнити відлагодження, бо поведінка розподілена між багатьма обгортками.