



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 №4
Тема: «Вступ до паттернів проектування»

Виконала
студентка групи ІА-32:
Іванова Анастасія
Юріївна

Перевірив:
Мягкий Михайло
Юрійович

Зміст

.....	1
-------	---

Тема проекту.....	3
Теоретичні відомості.....	3
Хід роботи.....	4
Створення діаграми класів патерну.....	4
Фрагменти коду, які реалізують патерн.....	7
Висновки.....	9
Контрольні питання.....	10

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 браузері (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.

Iterator (Ітератор) — це шаблон проєктування, який дає змогу послідовно отримувати доступ до елементів колекції без розкриття її внутрішньої структури.

Основна ідея полягає в тому, щоб відокремити логіку зберігання даних від логіки їх обходу. Колекція відповідає за зберігання елементів, а ітератор — за навігацію між ними.

Ітератор зберігає поточну позицію під час обходу та надає універсальний інтерфейс для послідовного доступу.

Завдяки цьому один і той самий набір даних можна обходити різними способами — у прямому чи зворотному порядку, по парних або непарних елементах тощо — без зміни самої колекції.

Основні методи ітератора:

- First() — встановлює покажчик на перший елемент колекції.
- Next() — переходить до наступного елемента.
- IsDone() — перевіряє, чи досягнуто кінець колекції.
- CurrentItem() — повертає поточний елемент.

Переваги:

- Уніфікує доступ до елементів різних типів колекцій.
- Спрощує класи зберігання даних.
- Дозволяє реалізовувати кілька способів обходу без зміни коду колекції.

Недоліки:

- Може бути надлишковим для простих структур, які можна обійти звичайним циклом.

Хід роботи

Створення діаграми класів патерну

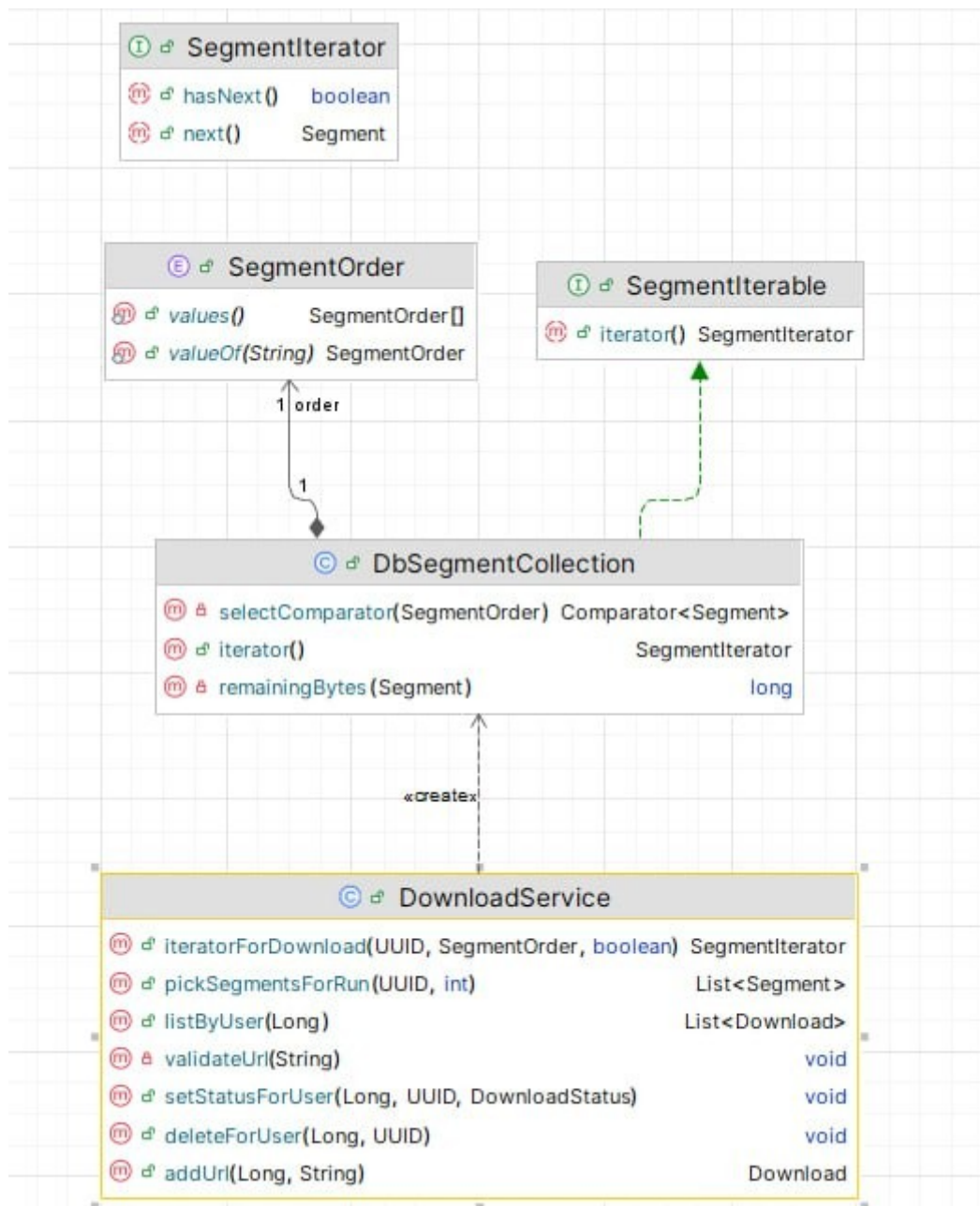


Рис. 1. Діаграма класів патерну Iterator

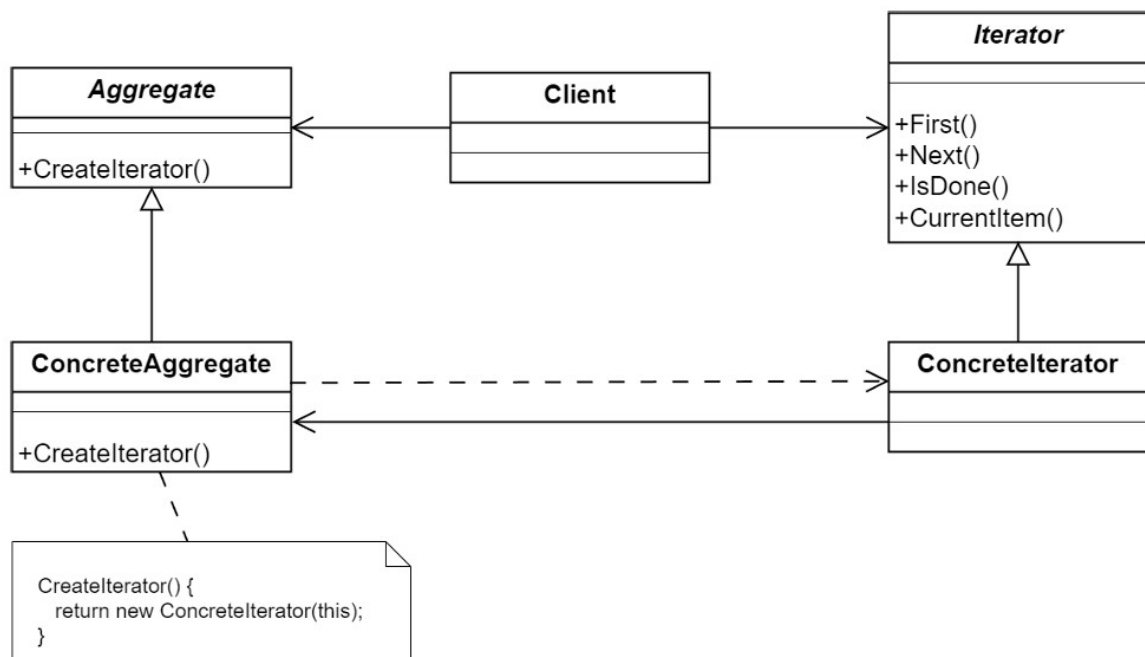


Рис. 2. Шаблон Iterator

1. **Aggregate** → **SegmentIterable**

Це абстрактний інтерфейс, який має метод `CreateIterator()`.

Він визначає, що будь-яка колекція може створити ітератор для обходу своїх елементів.

Фрагмент коду:

```
public interface SegmentIterable {
    SegmentIterator iterator();
}
```

Роль:

Він задає контракт: будь-яка колекція сегментів має вміти створити свій ітератор (`iterator()`), який повертає об'єкт типу `SegmentIterator`.

2. **ConcreteAggregate** → **DbSegmentCollection**

`ConcreteAggregate` реалізує інтерфейс `Aggregate` і повертає об'єкт конкретного ітератора через метод `CreateIterator()`.

Фрагмент коду:

```
public class DbSegmentCollection implements SegmentIterable {
    private final List<Segment> segments;

    @Override
    public SegmentIterator iterator() {
        ...
    }
}
```

Роль:

Зберігає внутрішню колекцію сегментів (segments), що представляє частини одного завантаження (downloadId).

У методі iterator() створюється і повертається об'єкт ітератора, який обходить сегменти відповідно до вибраної політики (наприклад, за індексом або за залишком байтів).

4. Iterator → SegmentIterator

Iterator — інтерфейс, що визначає основні методи:

- First() — перейти до початку колекції,
- Next() — перейти до наступного елемента,
- IsDone() — перевірка завершення обходу,
- CurrentItem() — отримати поточний елемент.

Фрагмент коду:

```
public interface SegmentIterator {  
    boolean hasNext();  
    Segment next();  
}
```

Роль:

Методи:

hasNext() - повертає true, поки не дійшли кінця,

next() – переходить далі і повертає елемент, на який перейшли.

5. ConcreteIterator → анонімний клас в DbSegmentCollection

ConcreteIterator — клас, який реалізує Iterator і містить:

посилання на колекцію,
поточну позицію,
логіку руху вперед.

Фрагмент коду:

```
@Override  
public SegmentIterator iterator() {  
    var it = list.iterator();  
    return new SegmentIterator() {  
        public boolean hasNext() { return it.hasNext(); }  
        public Segment next() { return it.next(); }  
    };  
}
```

Роль:

Тримає внутрішній покажчик і повертає наступний сегмент доти, доки вони не закінчаться.

6. Client → DownloadService

Client використовує ітератор, щоб послідовно обробляти елементи колекції, не знаючи її внутрішньої структури.

Фрагмент коду:

```
public List<Segment> pickSegmentsForRun(UUID downloadId, int limit) {
    var it = iteratorForDownload(downloadId, SegmentOrder.BY_LEFTMOST_GAP,
true);
    List<Segment> picked = new ArrayList<>();
    while (it.hasNext() && picked.size() < limit) {
        picked.add(it.next());
    }
    return picked;
}
```

Роль:

Отримує ітератор від DbSegmentCollection і обходить сегменти, не знаючи, як вони зберігаються чи сортуються.

7. SegmentOrder

Він задає політику, у якому порядку обходити сегменти.

Фрагменти коду, які реалізують патерн

SegmentIterator:

```
package org.example.dlm.iterator;

import org.example.dlm.domain.Segment;

public interface SegmentIterator {
    boolean hasNext();
    Segment next();
}
```

SegmentIterable:

```
package org.example.dlm.iterator;

public interface SegmentIterable {
    SegmentIterator iterator();
}
```

SegmentOrder:

```
package org.example.dlm.iterator;

public enum SegmentOrder {
    BY_INDEX_ASC,
```

```

    BY_LEFTMOST_GAP,

    BY_SMALLEST_REMAINING
}
DbSegmentCollection:

package org.example.dlm.iterator;

import org.example.dlm.domain.Segment;
import org.example.dlm.domain.SegmentStatus;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class DbSegmentCollection implements SegmentIterable {

    private final List<Segment> source;
    private final SegmentOrder order;
    private final boolean onlyPending;

    public DbSegmentCollection(List<Segment> segmentsFromDb,
                               SegmentOrder order,
                               boolean onlyPending) {
        this.source = segmentsFromDb != null ? segmentsFromDb : List.of();
        this.order = order != null ? order : SegmentOrder.BY_INDEX_ASC;
        this.onlyPending = onlyPending;
    }

    @Override
    public SegmentIterator iterator() {
        List<Segment> prepared = new ArrayList<>(source);

        if (onlyPending) {
            prepared.removeIf(s -> s.getStatus() != SegmentStatus.PENDING);
        }

        prepared.sort(selectComparator(order));

        return new SegmentIterator() {
            private int idx = 0;

            @Override
            public boolean hasNext() {
                return idx < prepared.size();
            }

            @Override
            public Segment next() {
                return prepared.get(idx++);
            }
        };
    }
}

```

```

private Comparator<Segment> selectComparator(SegmentOrder order) {
    return switch (order) {
        case BY_INDEX_ASC -> Comparator.comparingInt(Segment::getIdx);

        case BY_LEFTMOST_GAP -> Comparator
            .comparingLong(Segment::getStartByte)
            .thenComparingLong(Segment::getReceivedBytes);

        case BY_SMALLEST_REMAINING -> Comparator
            .comparingLong(this::remainingBytes)
            .thenComparingInt(Segment::getIdx);
    };
}

private long remainingBytes(Segment s) {
    long total = Math.max(0, (s.getEndByte() - s.getStartByte() + 1));
    long rem = total - Math.max(0, s.getReceivedBytes());
    return Math.max(0, rem);
}
}

```

DownloadService:

```

@SuppressWarnings("unused")
@Transactional(readOnly = true)
public SegmentIterator iteratorForDownload(UUID downloadId,
                                           SegmentOrder order,
                                           boolean onlyPending) {
    List<Segment> list = segments.findByDownload_Id(downloadId);
    var collection = new DbSegmentCollection(list, order, onlyPending);
    return collection.iterator();
}

@SuppressWarnings("unused")
@Transactional(readOnly = true)
public List<Segment> pickSegmentsForRun(UUID downloadId, int limit) {
    var it = iteratorForDownload(downloadId, SegmentOrder.BY_LEFTMOST_GAP, true);
    int cap = Math.max(1, limit);

    List<Segment> picked = new ArrayList<>(cap);
    while (it.hasNext() && picked.size() < cap) {
        picked.add(it.next());
    }
    return picked;
}

```

Висновки

У ході виконання лабораторної роботи було вивчено призначення та класифікацію патернів проєктування, а також розглянуто приклад їх практичного використання під час створення програмних систем. Було реалізовано патерн Iterator (Ітератор), який забезпечує послідовний доступ до елементів колекції без розкриття її внутрішньої структури

Контрольні питання

1. Що таке шаблон проєктування?

Шаблон проєктування — це типовий спосіб розв’язання повторюваної задачі при створенні програмного забезпечення. Він описує загальну структуру рішення, яку можна застосувати в різних ситуаціях.

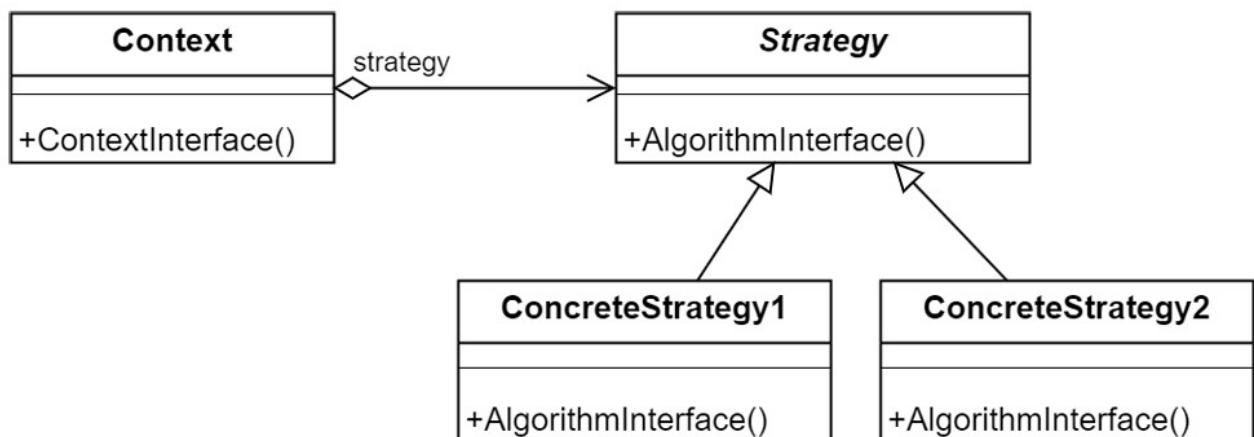
2. Навіщо використовувати шаблони проєктування?

Вони забезпечують повторне використання перевірених рішень, спрощують розробку, підвищують гнучкість, масштабованість і зрозумілість програмної системи.

3. Яке призначення шаблону «Стратегія»?

Патерн «Стратегія» дозволяє визначати різні алгоритми, інкапсулювати їх і взаємозамінно використовувати у програмі, не змінюючи код клієнта.

4. Нарисуйте структуру шаблону «Стратегія».



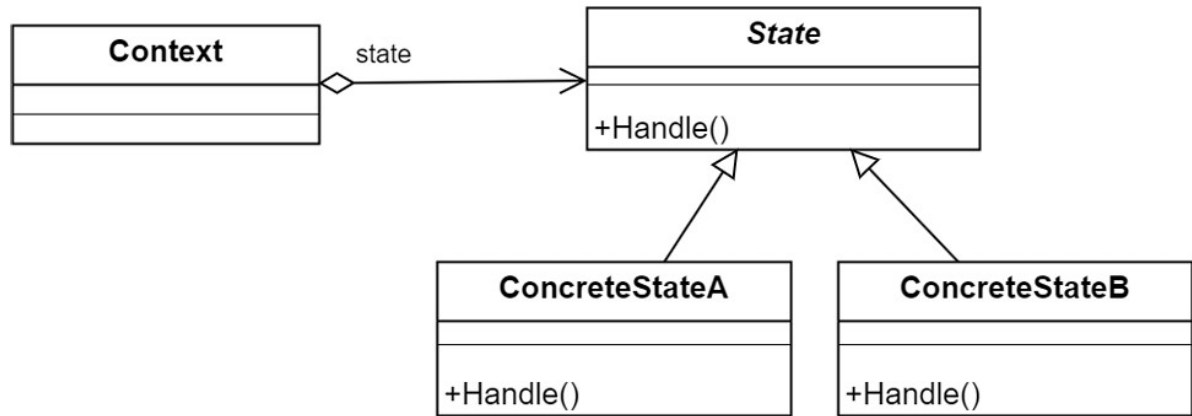
5. Які класи входять у шаблон «Стратегія» та яка між ними взаємодія?

- Context — зберігає посилання на поточну стратегію.
 - Strategy — інтерфейс усіх алгоритмів.
 - ConcreteStrategy — реалізації різних алгоритмів.
- Контекст викликає метод стратегії через інтерфейс, не знаючи її конкретного типу.

6. Яке призначення шаблону «Стан»?

Патерн «Стан» дозволяє об’єкту змінювати свою поведінку при зміні внутрішнього стану, ніби він належить до іншого класу.

7. Нарисуйте структуру шаблону «Стан».



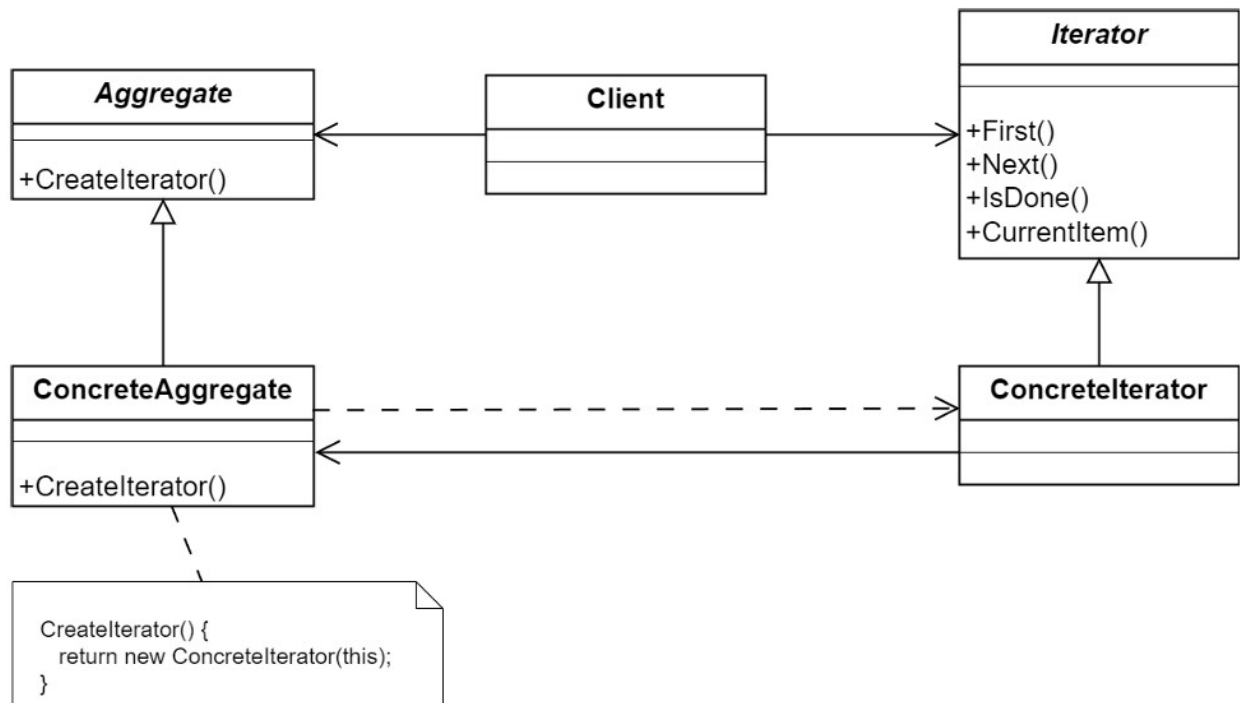
8. Які класи входять у шаблон «Стан» та яка між ними взаємодія?

- **Context** — містить поточний стан і делегує йому запити.
- **State** — інтерфейс для різних станів.
- **ConcreteState** — конкретні реалізації станів, які змінюють поведінку контексту.

9. Яке призначення шаблону «Ітератор»?

Патерн «Ітератор» забезпечує послідовний доступ до елементів колекції без розкриття її внутрішньої структури.

10. Нарисуйте структуру шаблону «Ітератор».



11. Які класи входять у шаблон «Ітератор» та яка між ними взаємодія?

- **Iterator** — інтерфейс із методами доступу (**first()**, **next()**, **isDone()**, **currentItem()**).
- **ConcreteIterator** — реалізація обходу конкретної колекції.
- **Aggregate** — інтерфейс колекції, що створює ітератор.

- ConcreteAggregate — конкретна колекція, яка повертає свій ітератор. Ітератор обходить елементи колекції, не змінюючи її структуру.

12. У чому полягає ідея шаблону «Одинак»?

Патерн «Одинак» гарантує, що клас матиме лише один екземпляр, і надає глобальну точку доступу до нього.

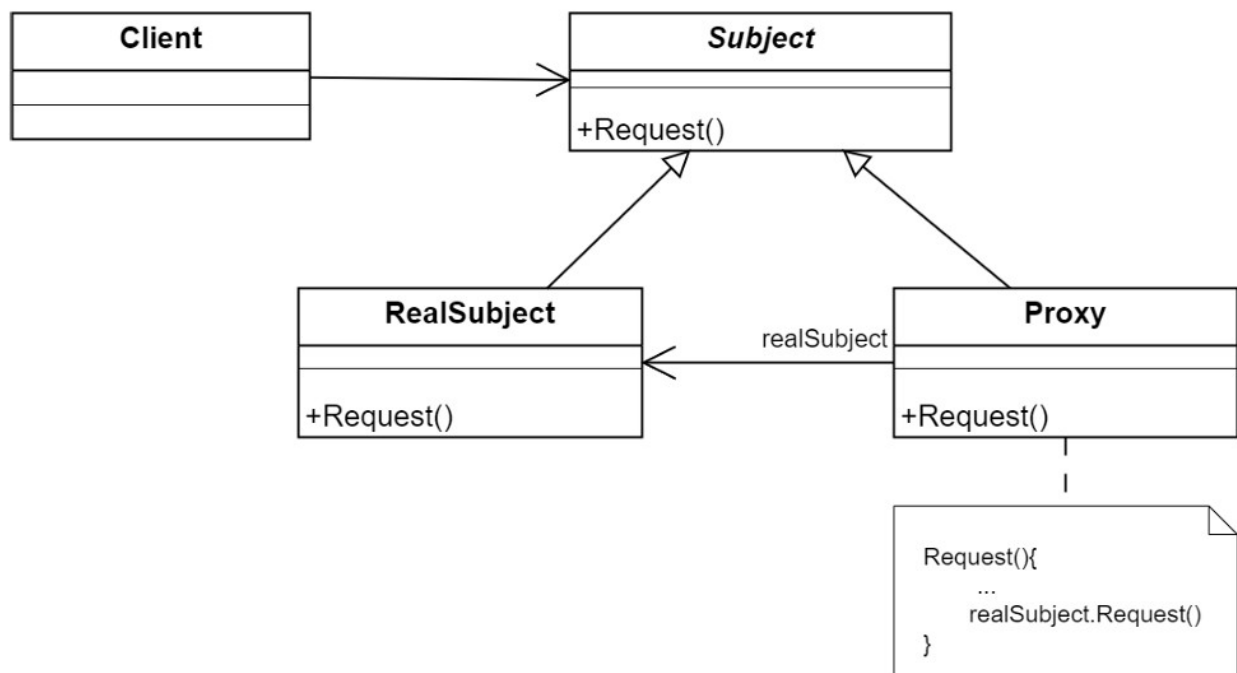
13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Тому що він порушує принципи інкапсуляції й ускладнює тестування, оскільки створює глобальний стан, який важко контролювати.

14. Яке призначення шаблону «Проксі»?

Патерн «Проксі» використовується для контролю доступу до іншого об'єкта, створюючи його замісник, який перехоплює запити перед передачею реальному об'єкту.

15. Нарисуйте структуру шаблону «Проксі».



16. Які класи входять у шаблон «Проксі» та яка між ними взаємодія?

- Subject — спільний інтерфейс для Proxy та RealSubject.
- RealSubject — реальний об'єкт, до якого здійснюється доступ.
- Proxy — об'єкт-посередник, який контролює виклики до RealSubject. Proxy приймає запит від клієнта та вирішує, коли або як передати його реальному об'єкту.