Java 디자인 패턴 - Strategy & Flyweight

발표자: 한지원

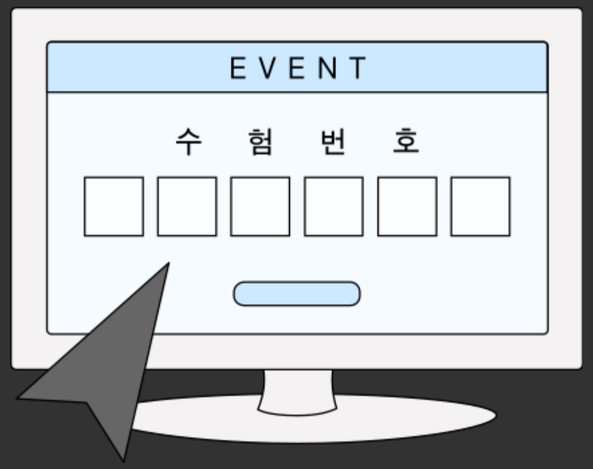
전략 패턴이란?

전략(또는 정책)

: 특정한 목표를 수행하기 위한 행동 계획

전략 패턴이란?

조건: 목표 점수, 공부 속도

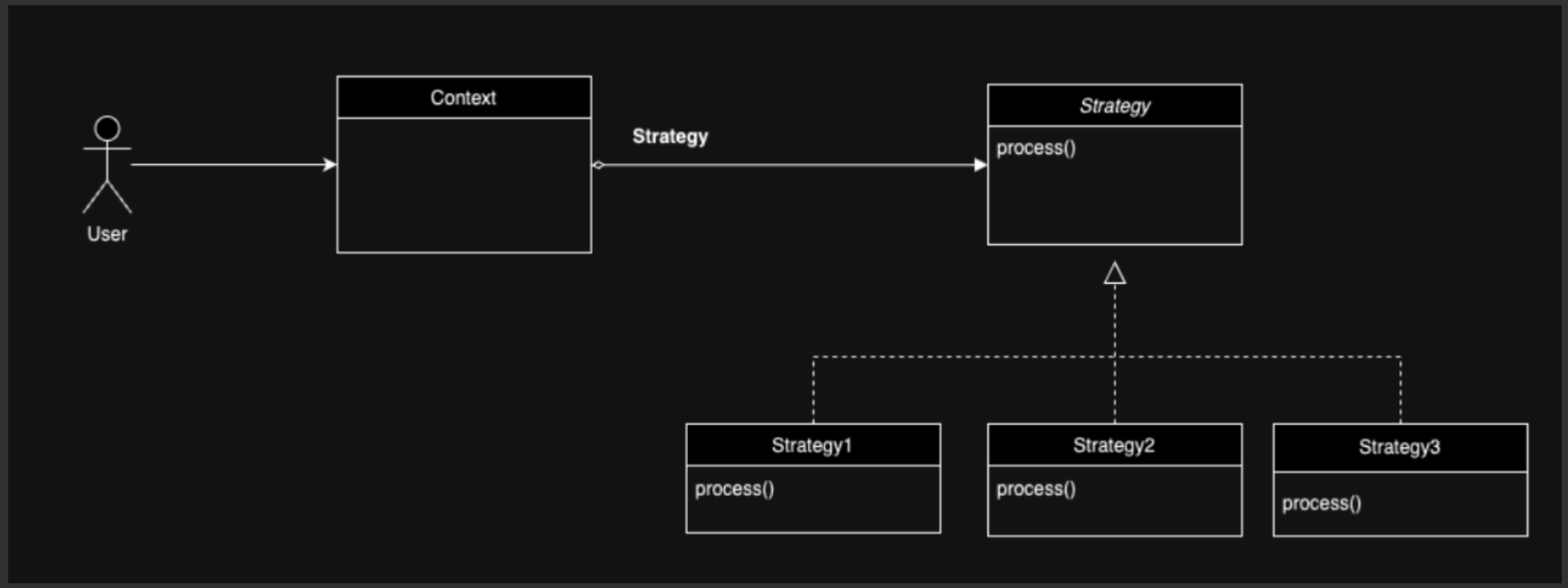


벼락치기 *VS* 오랫동안 꼼꼼히 공부하기

전략 패턴이란?

내부적으로 조금 다를 뿐 **개념적으로 관련**된 **1개 이상**의 로직이 존재할 때,
각 로직을 캡슐화하고 이들이 **상호 교환**할 수 있도록 하는 디자인 패턴

전략 패턴의 구성



- **Strategy**(전략): 공통의 연산을 인터페이스로 정의
- **ConcreateStrategy**(구체전략): Strategy 인터페이스의 구현체
- **Context(문맥)**: Strategy 객체의 참조값 관리, 사용자에서 온 요청을 각 전략 객체로 전달, Context를 통해 필요한 데이터에 접근

언제 전략 패턴을 사용하면 좋을까?

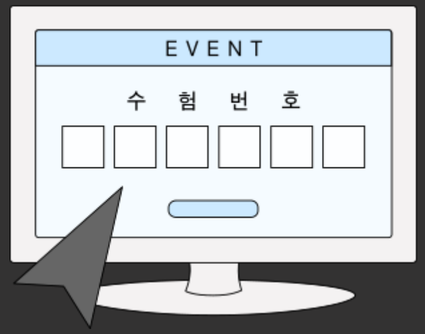
1.**동일 계열 알고리즘**이 여러 개 존재할 때

2. 사용자에 **노출하지 말아야하는 데이터**를 사용할 때, 정보의 은닉화

3. **다중 분기문**이 복잡하게 들어간 코드가 있을 때, 코드의 가독성 및 유지보수성

전략 패턴을 사용해 보자!

시험 전략



조건: 목표 점수, 이전 점수, 공부 속도

Fast 전략

하루에 많은 시간을 투자!

Master 전략

오랜 기간 천천히

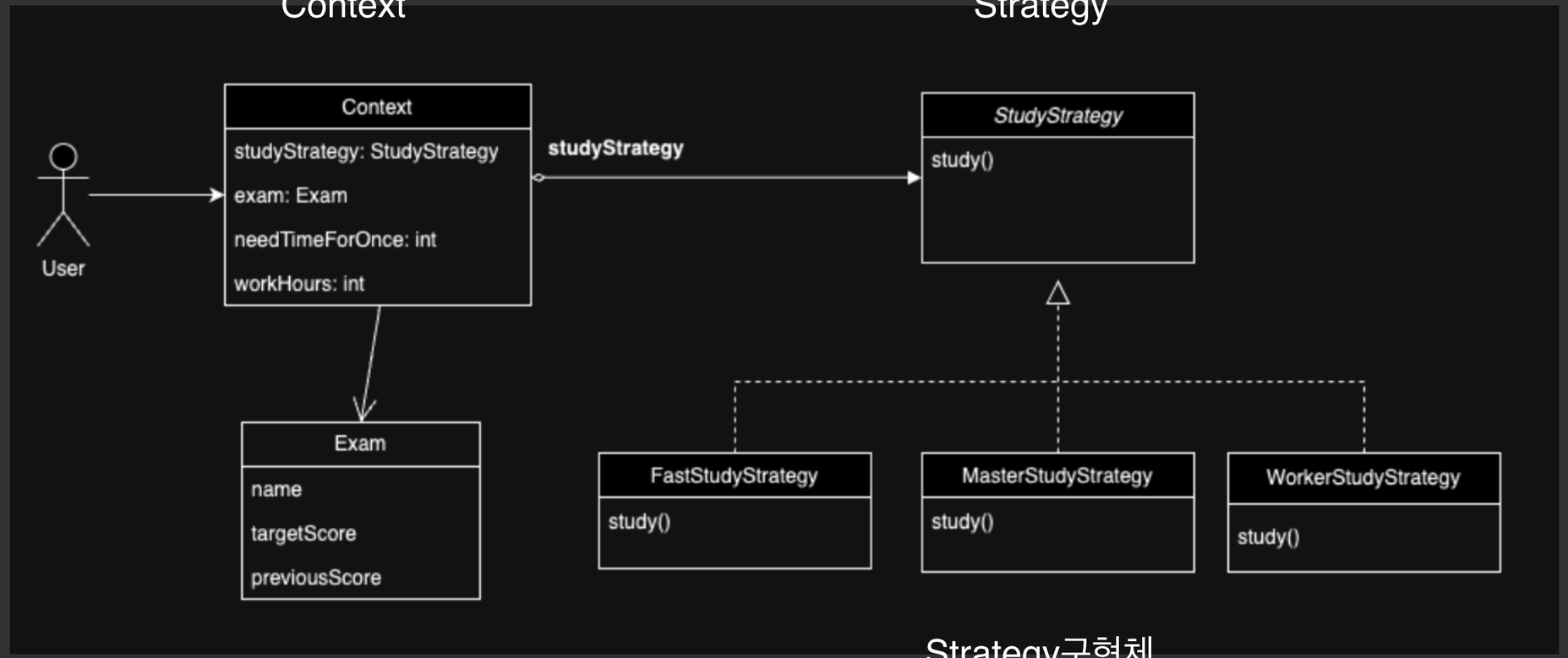
직장인 전략

퇴근 후 조금씩

전략 패턴을 사용해 보자!

Context

Strategy



Strategy구현체

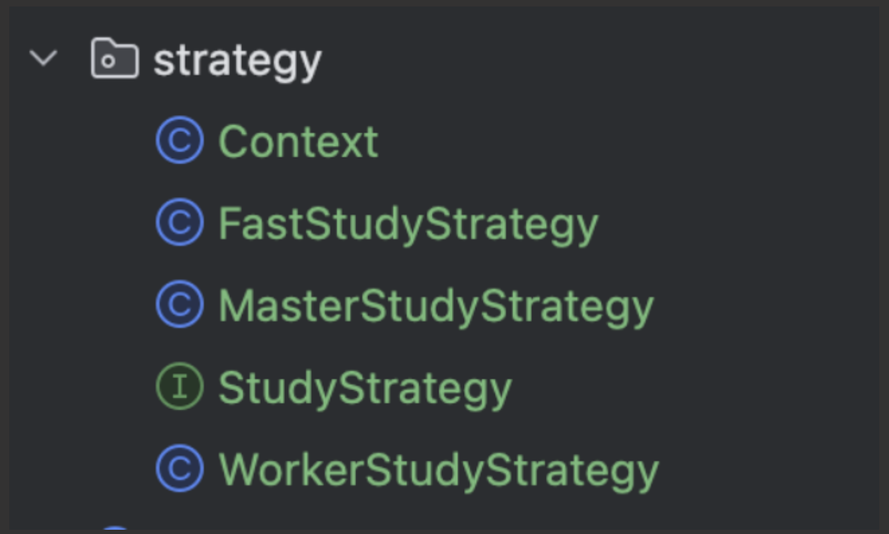
전략 패턴의 효과 

```java
@Test  new *
void testNoneStrategy() {
    Exam exam = new Exam( name: "한지원",  targetScore: 100,  previousScore: 50);

    String strategyType = "Master";
    int targetScore = exam.targetScore;
    int previousScore = exam.previousScore;
    int needTime = 0;
    int needDays = 0;

    switch (strategyType) {
        case "Master":
            System.out.println("Studying master");
            needTime = (targetScore - previousScore) * 30; // 1점 올리는 데 30시간 필요
            needDays = needTime / 6; // 하루에 6시간 공부
            System.out.println("Need time: " + needDays + " need days: " + needDays);
            break;
        case "Fast":
            System.out.println("Studying fast");
            needTime = (targetScore - previousScore) * 10; // 1점 올리는 데 10시간 필요
            needDays = needTime / 10; // 하루에 10시간 공부
            System.out.println("Need time: " + needDays + " need days: " + needDays);
        case "Worker":
            System.out.println("Studying for worker");
            needTime = (targetScore - previousScore) * 10; // 1점 올리는 데 10시간 필요
            needDays = needTime / 2; // 하루에 6시간 공부
            System.out.println("Need time: " + needDays + " need days: " + needDays);
        default:
            throw new IllegalArgumentException("Unknown strategy type: " + strategyType);
    }
}
```

```java
@Test  new *
void testStudyWhenMasterStrategy(){
    Exam exam = new Exam( name: "한지원",  targetScore: 100,  previousScore: 50);
    Context context = new Context(exam);
    context.setStudyStrategy(new MasterStudyStrategy());
    context.study();
}
```

- 다중 조건문 제거
- 유지보수성 및 가독성 향상
- 전략 세부 사항 은닉화

전략 패턴 주의사항 ⚠️



- 객체 수 증가

전략 패턴 주의사항 ⚠️

```java
public interface StudyStrategy {  5 usages  3 implementations  new *

    void study(int targetScore, int previousScore, int needTimeForOneScore, int workHours);

}
```

```java
public class WorkerStudyStrategy implements StudyStrategy {  no usages  new *

    @Override  1 usage  new *
    public void study(int targetScore, int previousScore, int needTimeForOneScore, int workHours) {
        System.out.println("Studying for worker");
        int needTime = (targetScore - previousScore) * needTimeForOneScore;
        int needDays = needTime / (12 - workHours); // 하루 공부 가능 시간: 12시간에서 근무 시간을 뺀다.
        System.out.println("Need time: " + needDays + " need days: " + needDays);
    }
}
```

```java
public class FastStudyStrategy implements StudyStrategy {  no usages  new *

    @Override  1 usage  new *
    public void study(int targetScore, int previousScore, int needTimeForOneScore, int workHours) {
        System.out.println("Studying fast");
        int needTime = (targetScore - previousScore) * needTimeForOneScore; // 1점 올리는 데 10시간 필요
        int needDays = needTime / 10; // 하루에 10시간 공부
        System.out.println("Need time: " + needDays + " need days: " + needDays);
    }
}
```

Strategy 객체와 Context 객체 사이에 의사소통 오버헤드

전략 패턴 주의사항 ⚠️

```java
public interface StudyStrategy {  5 usages  3 implementations  new *

    void study(int targetScore, int previousScore, int needTimeForOneScore, int workHours);

}
```

```java
public interface StudyStrategy {

    void study(Context context)
}
```

✅ 새로운 전략이 추가된다고 하더라도, 인터페이스가 바뀌면 안된다.

✅ Context 자체를 파라미터로 전달하는 방법도 존재하나, 클래스 간 결합도가 높아질 수 있다.

현재 코드는 과연 효율적일까?

```java
@Test  new *
void testStudyWhenMasterStrategy(){
    Exam exam = new Exam( name: "한지원",  targetScore: 100,  previousScore: 50);
    Context context = new Context(exam);
    context.setStudyStrategy(new MasterStudyStrategy());
    context.study();
}
```

Strategy + Flyweight의 필요성

```java
@Test  new *
void inefficientWhenStrategy(){

    Exam exam = new Exam( name: "한지원",  targetScore: 100,  previousScore: 50);
    Context context = new Context();
    context.setExam(exam);
    context.setNeedTimeForOneScore(30);
    context.setStudyStrategy(new MasterStudyStrategy());
    context.study();

    Exam exam2 = new Exam( name: "홍길동",  targetScore: 90,  previousScore: 50);
    Context context2 = new Context();
    context2.setExam(exam2);
    context2.setNeedTimeForOneScore(20);
    context2.setStudyStrategy(new MasterStudyStrategy());

    // 다른 주소 값을 갖는 것을 검증
    assertNotEquals(context.getStudyStrategyHashCode(), context2.getStudyStrategyHashCode());
}
```

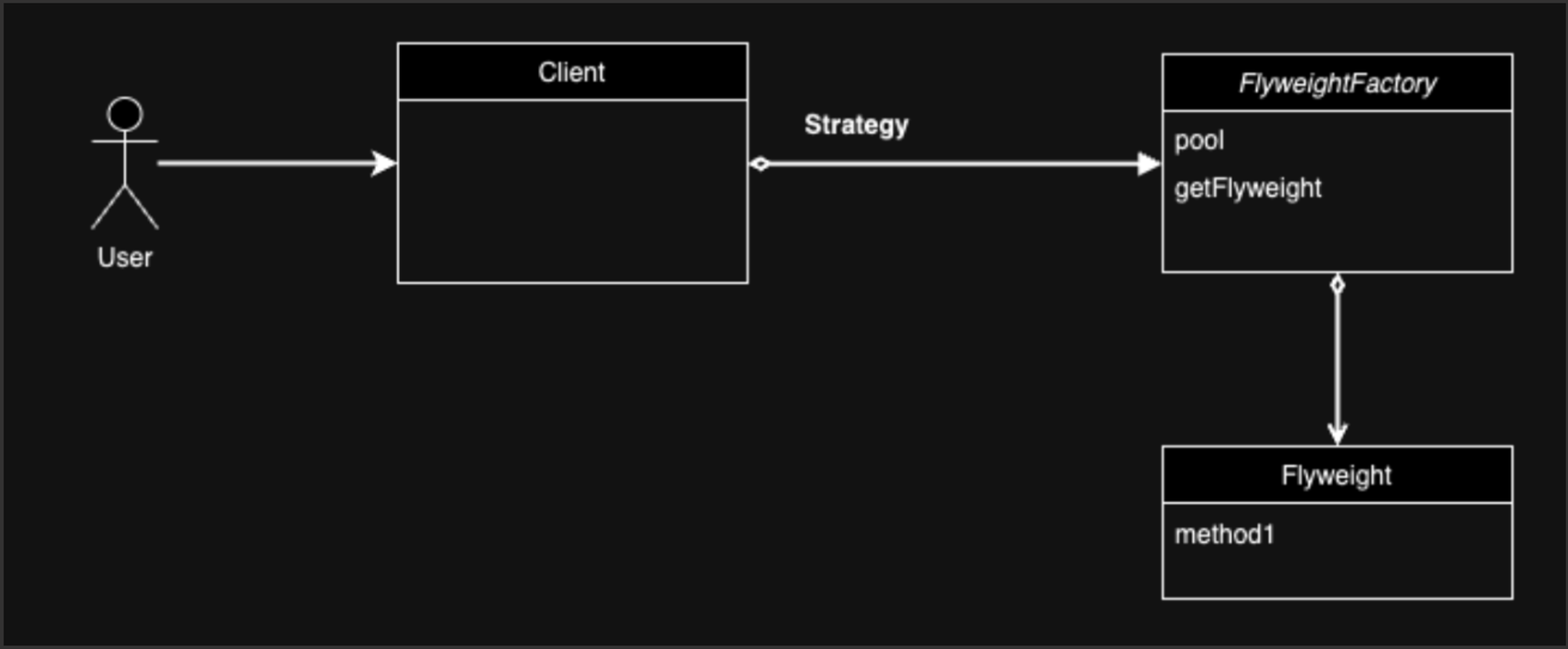
- 매번 new 연산 발생
- But, 전략 객체 반복 생성 필요 X

Flyweight 패턴이란?

객체를 가볍게!

인스턴스를 최대한 공유하고 쓸데없이 new 하지 않는다.

Flyweight 구성

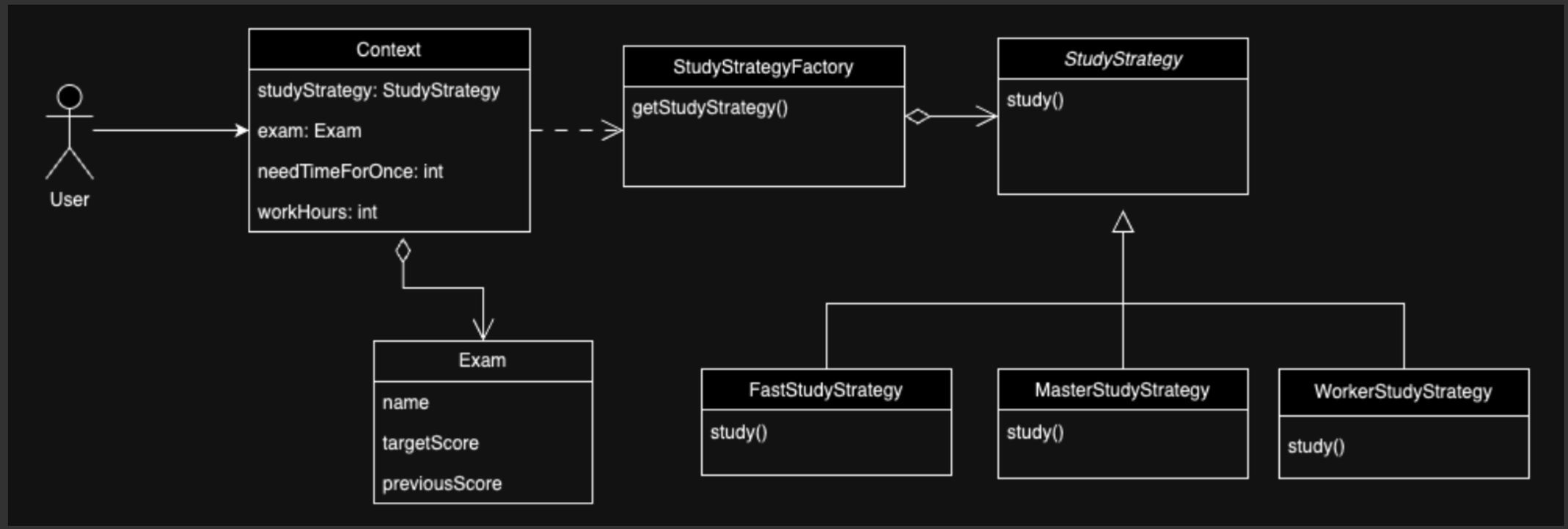


- **Flyweig**ht: 매번 객체를 생성하지 않고 공유하여 가볍게 할 대상(클래스)
- **Flyweig**htFactory: 인스턴스를 공유하게 할 공장 역할을 하는 클래스
- **Client:** FlyweightFactory를 사용하여 Flyweight를 이용하는 클래스

Strategy + Flyweight

Client

Flyweight Factory

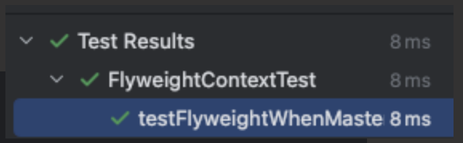


Flyweight

Flyweight 패턴의 효과 

```java
public class FlyweightContext {  7 usages  new *
    private Exam exam;  3 usages
    private StudyStrategy studyStrategy;  3 usages
    private int needTimeForOneScore;  2 usages
    private int workHours;  1 usage

    public void setStudyStrategy(String studyType) {  3 usages  new *
        this.studyStrategy = StudyStrategyFactory.getStudyStrategy(studyType);
    }
```

```java
class FlyweightContextTest {  new *

    @Test  new *
    void testFlyweightWhenMasterStrategy(){
        Exam exam = new Exam( name: "한지원",  targetScore: 100,  previousScore: 50);
        FlyweightContext context = new FlyweightContext();
        context.setExam(exam);
        context.setNeedTimeForOneScore(30);
        context.setStudyStrategy("Master");


        Exam exam2 = new Exam( name: "홍길동",  targetScore: 90,  previousScore: 50);
        FlyweightContext context2 = new FlyweightContext();
        context2.setExam(exam2);
        context2.setNeedTimeForOneScore(20);
        context2.setStudyStrategy("Master");

        // 같은 주소 값을 갖는 것을 검증
        assertEquals(context.getStudyStrategyHashCode(), context2.getStudyStrategyHashCode());
```



- new 생성자를 사용하지 않는다.
- 인스턴스의 공유로 자원 효율성 up

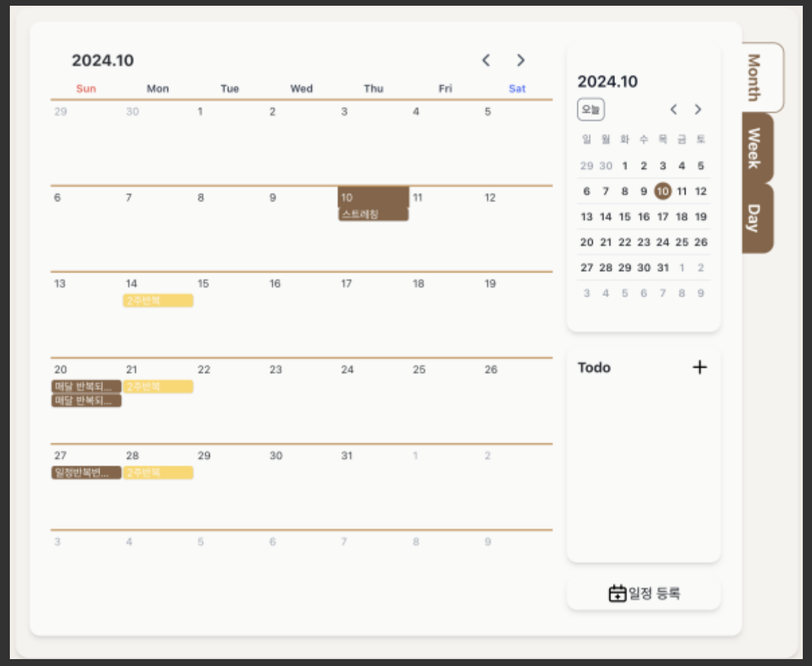
Flyweight 패턴 주의사항 ⚠️

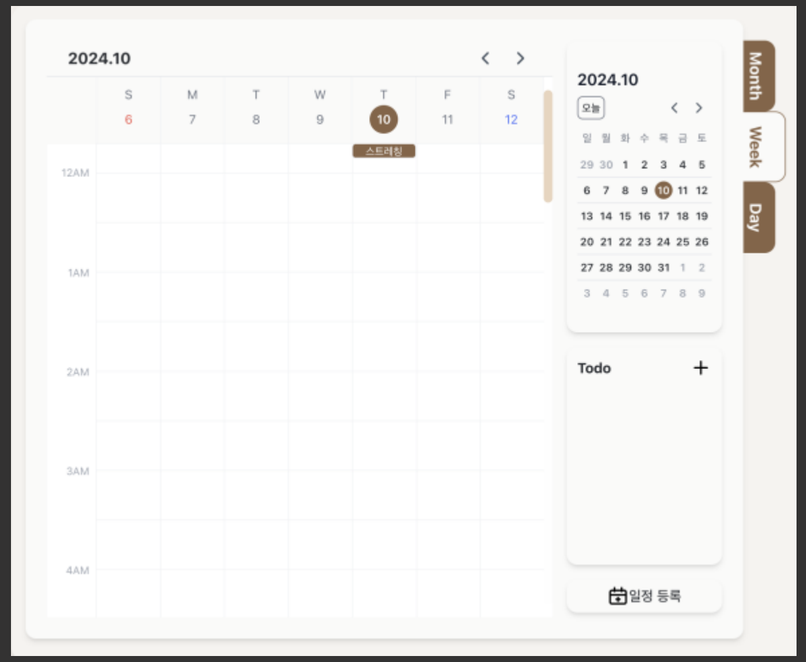
```java
public class StudyStrategyFactory {  1 usage  new *

    private static final Map<String, StudyStrategy> studyStrategyMap = new HashMap<>();

    private StudyStrategyFactory() {  no usages  new *
    }
    public synchronized static StudyStrategy getStudyStrategy(String strategyType) {  1
        if (!studyStrategyMap.containsKey(strategyType)) {
            switch (strategyType) {
                case "Fast":
                    studyStrategyMap.put(strategyType, new FastStudyStrategy());
                    break;
                case "Master":
                    studyStrategyMap.put(strategyType, new MasterStudyStrategy());
                    break;
                case "Worker":
                    studyStrategyMap.put(strategyType, new WorkerStudyStrategy());
                    break;
            }
        }
        return studyStrategyMap.get(strategyType);
    }
}
```

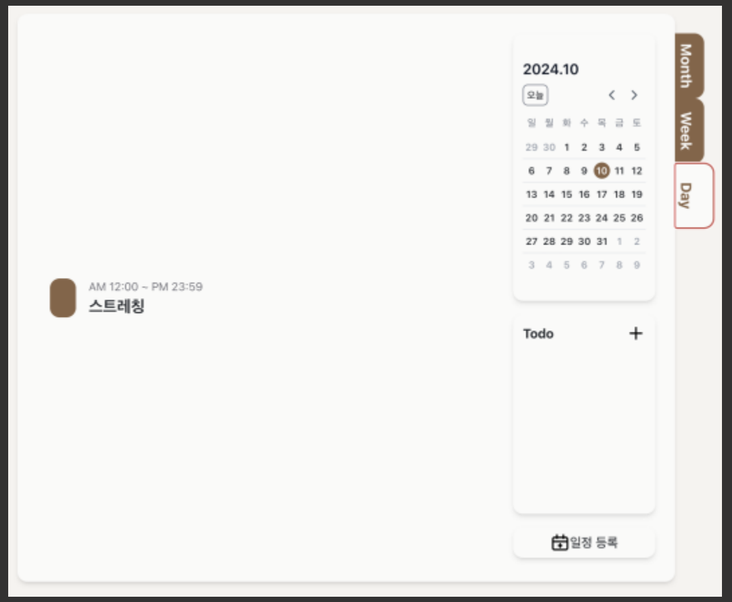
- Flyweight 객체는 가비지컬렉션되지 않는다.
- sychronized로 동기화한다.

와플 프로젝트 - 전략 패턴 적용 사례

달력 서비스







Month, Week, Day 단위로 달력을 조회하는 기능

AS-IS

```
switch (repetition.getRepetitionCycle()) {
    case YEAR -> {...}
    case MONTH -> {...}
    case WEEK -> {
        Duration diffDateTime = Duration.between(startDateTime, endDateTime);

        int repetitionWeek = originSchedule.getRepetition().getDayOfWeek();
        DayOfWeek dayOfWeek;

        int mask = 1;
        for (int i = 0; i < 7; i++) {
            if ((repetitionWeek & mask) != 0) {
                dayOfWeek = switch (i) {
                    case 0 -> DayOfWeek.SUNDAY;
                    case 1 -> DayOfWeek.MONDAY;
                    case 2 -> DayOfWeek.TUESDAY;
                    case 3 -> DayOfWeek.WEDNESDAY;
                    case 4 -> DayOfWeek.THURSDAY;
                    case 5 -> DayOfWeek.FRIDAY;
                    case 6 -> DayOfWeek.SATURDAY;
                    default -> throw new ScheduleException(ScheduleErrorCode.NOT_APPROPRIATE_REPETITION_CYCLE);
                };

                startDateTime = originSchedule.getStartDate().with(TemporalAdjusters.next(dayOfWeek));
                endDateTime = startDateTime.plus(diffDateTime);

                while (startDateTime.toLocalDate().isEqual(repetitionEndDate) || startDateTime.toLocalDate()
                        .isBefore(repetitionEndDate)) {

                    Schedule repetitionSchedule = scheduleRepository.save(
                            originSchedule.toRepetitionSchedule(startDateTime, endDateTime));

                    memberScheduleService.createMemberSchedule(repetitionSchedule,
                            List.of(new MemberIdResponse(memberId)));

                    startDateTime = startDateTime.plusWeeks(1);
                    endDateTime = endDateTime.plusWeeks(1);
                }
            }

            mask <<= 1;
        }
    }
    default -> throw new ScheduleException(ScheduleErrorCode.NOT_APPROPRIATE_REPETITION_CYCLE);
}
```

☑️ 복잡한 switch ~ case 문으로
가독성 저하, 유지보수 어려움 ⚠️

TO-BE

```java
public interface CalendarStrategy {  7 usages  2 implementations  jiwonhan

    CalendarType getCalendarType();  1 usage  2 implementations  jiwonhan

    List<ScheduleFindResponse> getAllSchedule(long calendarId, LocalDate startDate);
}
```

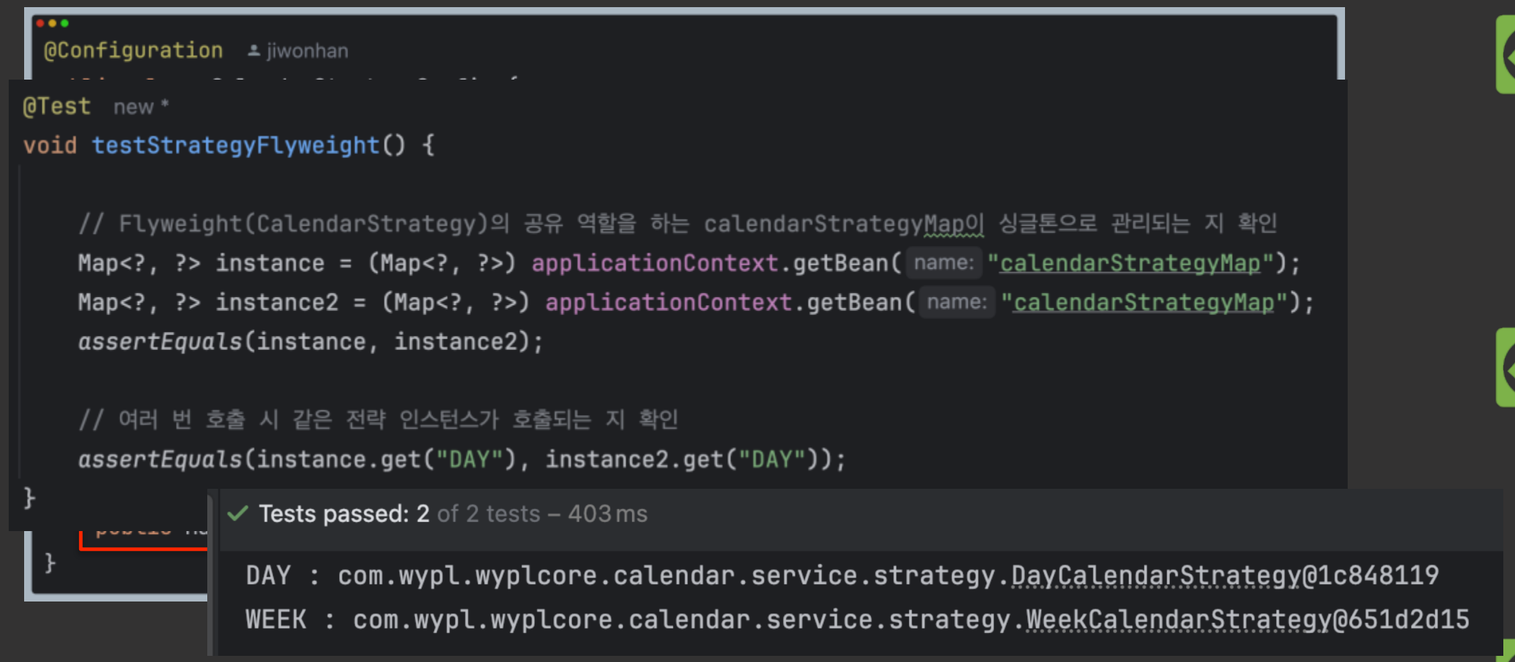
캘린더의 타입으로
WEEK, DAY, MONTH, YEAR가 존재

```java
@Component  jiwonhan *
@RequiredArgsConstructor
public class DayCalendarStrategy implements CalendarStrategy {

    private final CalendarRepository calendarRepository;

    @Override  1 usage  jiwonhan
    public CalendarType getCalendarType() { return CalendarType.DAY; }

    @Override  1 usage  jiwonhan *
    public List<ScheduleFindResponse> getAllSchedule(long calendarId, LocalDate startDate) {
        List<Schedule> schedules = calendarRepository.findSchedulesByIdBetweenStartDateAndEndDate(calendarId, startDate, startDate);
        return schedules.stream()  Stream<Schedule>
                .map(ScheduleFindResponse::from)  Stream<ScheduleFindResponse>
                .toList();
    }
}
```

각 캘린더 타입에 따라,
연산 후 Schedule 리스트를 반환

WeekCalendarStrategy,
DayCalendarStratey 등 캘린더 타입에 맞는 전략 패턴 적용

TO-BE



- 매번 CalendarStrategy를 생성하지 않고 공유하게 하는 Flyweight 패턴 적용
- 자료구조 Map으로 CalendarStrategy를 공유하는 pool 생성
- 스프링 Bean 등록으로 calendarStrategyMap이 싱글톤으로 관리

결과

```
switch (repetition.getRepetitionCycle()) {
    case YEAR -> {...}
    case MONTH -> {...}
    case WEEK -> {
        Duration diffDateTime = Duration.between(startDateTime, endDateTime);

        int repetitionWeek = originSchedule.getRepetition().getDayOfWeek();
        DayOfWeek dayOfWeek;

        int mask = 1;
        for (int i = 0; i < 7; i++) {
            if ((repetitionWeek & mask) != 0) {
                dayOfWeek = switch (i) {
                    case 0 -> DayOfWeek.SUNDAY;
                    case 1 -> DayOfWeek.MONDAY;
                    case 2 -> DayOfWeek.TUESDAY;
                    case 3 -> DayOfWeek.WEDNESDAY;
                    ...
                };

                startDateTime = originSchedule.getStartDate().with(TemporalAdjusters.next(dayOfWeek));
                endDateTime = startDateTime.plus(diffDateTime);

                while (startDateTime.toLocalDate().isEqual(repetitionEndDate) || startDateTime.toLocalDate()
                        .isBefore(repetitionEndDate)) {

                    Schedule repetitionSchedule = scheduleRepository.save(
                            originSchedule.toRepetitionSchedule(startDateTime, endDateTime));

                    memberScheduleService.createMemberSchedule(repetitionSchedule,
                            List.of(new MemberIdResponse(memberId)));

                    startDateTime = startDateTime.plusWeeks(1);
                    endDateTime = endDateTime.plusWeeks(1);
                }

                mask <<= 1;
            }
        }
        default -> throw new ScheduleException(ScheduleErrorCode.NOT_APPROPRIATE_REPETITION_CYCLE);
}
```

```
List<ScheduleFindResponse> foundScheduleFindResponses = calendarStrategyMap.get(calendarType).getAllSchedule(foundCalendar.getId(), startDate);
```

유지보수성 증대

새로운 달력(Year 단위 달력)조회 기능 추가 시, 확장에 유리

References

- GoF의 디자인 패턴
- Java 언어로 배우는 디자인 패턴 입문