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

«Патерни проектування»

Виконав
студент групи ІА-34:
Сльота Максим

Перевірив:
Мягкий М. Ю.

Київ - 2025

Тема: Патерни проектування.

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

Вихідні дані:

17. **System activity monitor** (iterator, command, abstract factory, bridge, visitor, SOA)

Монітор активності системи повинен зберігати і запам'ятовувати статистику використовуваних компонентів системи, включаючи навантаження на процесор, обсяг займаної оперативної пам'яті, натискання клавіш на клавіатурі, дії миші (переміщення, натискання), відкриття вікон і зміна вікон; будувати звіти про використання комп'ютера за різними критеріями (% часу перебування у веб-браузері, середнє навантаження на процесор по годинах, середній час роботи комп'ютера по днях і т.д.); правильно поводитися з «простоюванням» системи – відсутністю користувача.

Теоретичні відомості:

Шаблон «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. 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.

Шаблон «Factory Method» - 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».

Шаблон «Memento» - шаблон 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.

Шаблон «Observer» - Шаблон 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.

Шаблон «Decorator» - Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми [6]. 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.

Хід роботи:

1. Діаграма класів, яка представляє використання шаблону Abstract Factory

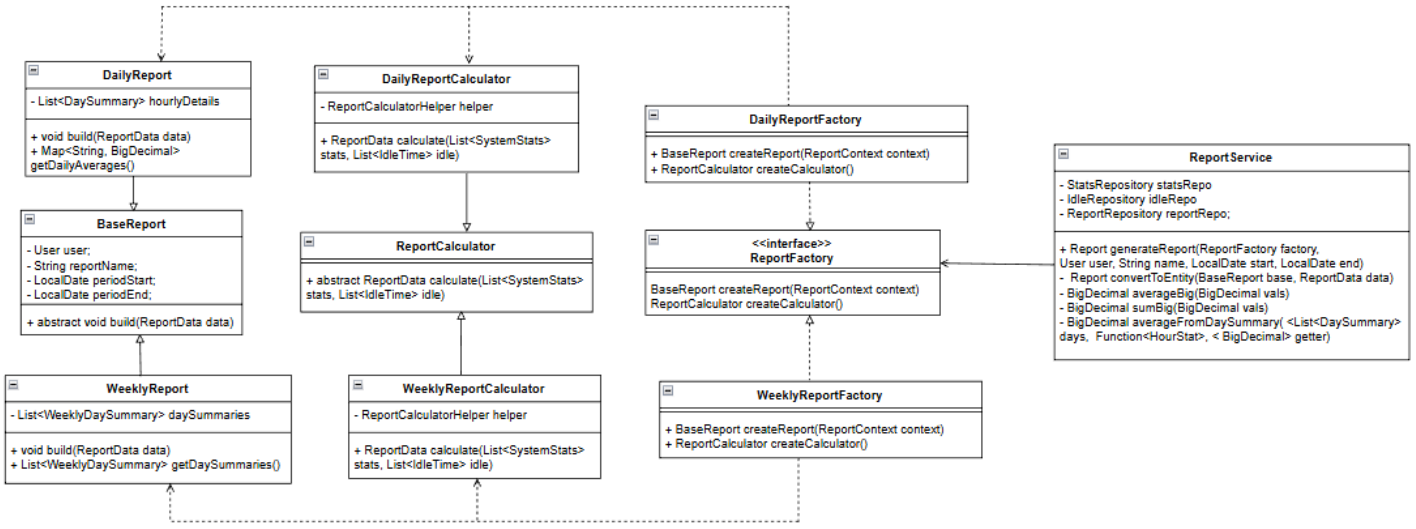


Рис 1. Діаграма класів

У проєкті System Activity Monitor шаблон «Абстрактна фабрика» виконує роль гнучкого механізму для створення різних типів звітів про активність системи. Основна ідея полягає в тому, що система звітів може формувати дані для різних проміжків часу денних і тижневих звітів і кожен з них має власну структуру, логіку розрахунків та

алгоритми побудови. Денний звіт містить погодинну деталізацію показників, тоді як тижневий узагальнені середні значення за кожен день. Система повинна створювати цілком різні за природою об’єкти, але організовано та узгоджено, без використання умовних операторів чи прямої прив’язки до конкретного типу звіту.

У цьому контексті шаблон «Абстрактна фабрика» дає можливість інкапсулювати створення всіх пов’язаних об’єктів у спеціальних фабриках. Фабрика денних звітів створює відповідну модель звіту DailyReport і калькулятор, який виконує денні обчислення та формує погодинну статистику. Натомість фабрика тижневих звітів формує WeeklyReport і забезпечує використання іншого калькулятора, що агрегує інформацію вже за днями. Обидві фабрики реалізують один інтерфейс, тому з точки зору сервісного коду вони виглядають однаково, хоча фактично створюють зовсім різні об’єкти та ініціюють різну логіку розрахунків. Сам сервіс, який формує звіт, працює лише з абстракціями: він не знає, який саме звіт буде створено, і не містить розгалужень для визначення його типу. Уся відповідальність за вибір структури та алгоритму переноситься у фабрики.

Доданий вихідний код системи:

ReportFactory.java - визначає спільний інтерфейс для всіх конкретних фабрик.

```
public interface ReportFactory {
   BaseReport createReport(ReportContext context);
   ReportCalculator createCalculator();
}
```

DailyReportFactory.java - створює набір об’єктів, характерних для денного звіту.

```
public class DailyReportFactory implements ReportFactory {
   @Override
   public BaseReport createReport(ReportContext context) {
      return new DailyReport(
            context.getUser(),
            context.getReportName(),
            context.getStart(),
            context.getEnd()
      );
   }
   @Override
   public ReportCalculator createCalculator() {
      return new DailyReportCalculator();
```

```
    }
```

WeeklyReportFactory.java - конкретна фабрика для тижневого звіту.

```
public class WeeklyReportFactory implements ReportFactory {
    @Override
    public BaseReport createReport(ReportContext context) {
        return new WeeklyReport(
                context.getUser(),
                context.getReportName(),
                context.getStart(),
                context.getEnd()
        );
    }
    @Override
    public ReportCalculator createCalculator() {
        return new WeeklyReportCalculator();
    }
}
```

BaseReport.java - базова модель звіту, що зберігає спільні властивості.

```
public abstract class BaseReport {
    protected User user;
    protected String reportName;
    protected LocalDate periodStart;
    protected LocalDate periodEnd;
    public BaseReport(User user, String reportName, LocalDate periodStart, LocalDate periodEnd) {
        this.user = user;
        this.reportName = reportName;
        this.periodStart = periodStart;
        this.periodEnd = periodEnd;
    }
    public User getUser() { return user; }
    public String getReportName() { return reportName; }
    public LocalDate getPeriodStart() { return periodStart; }
    public LocalDate getPeriodEnd() { return periodEnd; }

    public abstract void build(ReportData data);
}
```

DailyReport.java - модель денного звіту.

```
public class DailyReport extends BaseReport {
```

```
    private List<DaySummary> hourlyDetails;
    public DailyReport(User user, String name, LocalDate start, LocalDate end) {
        super(user, name, start, end);
    }
    @Override
    public void build(ReportData data) {
        this.hourlyDetails = data.getDaySummaries();
    }
    public List<DaySummary> getHourlyDetails() {
        return hourlyDetails;
    }
}
```

WeeklyReport.java - модель тижневого звіту.

```
public class WeeklyReport extends BaseReport {
    private List<WeeklyDaySummary> daySummaries;
    public WeeklyReport(User user, String name, LocalDate start, LocalDate end) {
        super(user, name, start, end);
    }
    @Override
    public void build(ReportData data) {
        this.daySummaries = data.getWeeklySummaries();
    }
    public List<WeeklyDaySummary> getDaySummaries() {
        return daySummaries;
    }
}
```

ReportCalculator.java - базовий абстрактний клас для всіх калькуляторів звітів.

```
public abstract class ReportCalculator {
    public abstract ReportData calculate(List<SystemStats> stats, List<IdleTime> idle);
}
```

DailyReportCalculator.java - калькулятор, який формує погодинні дані.

```
public class DailyReportCalculator extends ReportCalculator {
    private final ReportCalculatorHelper helper = new ReportCalculatorHelper();
    @Override
    public ReportData calculate(List<SystemStats> stats, List<IdleTime> idle) {
        ReportData data = new ReportData();
        data.setDaySummaries(helper.buildDaySummary(stats));
```

```
        return data;
    }
}
```

WeeklyReportCalculator.java - калькулятор тижневого звіту.

```
public class WeeklyReportCalculator extends ReportCalculator {
    private final ReportCalculatorHelper helper = new ReportCalculatorHelper();
    @Override
    public ReportData calculate(List<SystemStats> stats, List<IdleTime> idle) {
        ReportData data = new ReportData();
        if (stats == null || stats.isEmpty()) {
            data.setWeeklySummaries(List.of());
            return data;
        }
        Map<java.time.LocalDate, List<SystemStats>> grouped =
                stats.stream()
                        .filter(s -> s.getRecordedAt() != null)
                        .collect(Collectors.groupingBy(
                                s -> s.getRecordedAt().toLocalDate(),
                                LinkedHashMap::new,
                                Collectors.toList()
                        ));
        List<WeeklyDaySummary> list = new ArrayList<>();
        for (var entry : grouped.entrySet()) {
            var date = entry.getKey();
            var dayStats = entry.getValue();
            BigDecimal cpu = helper.average(dayStats, SystemStats::getCpuLoad);
            BigDecimal ram = helper.average(dayStats, SystemStats::getRamUsedMb);
            BigDecimal uptime = helper.averageUptime(dayStats);

            BigDecimal idleSeconds = BigDecimal.valueOf(
                    idle.stream()
                            .filter(i -> i.getStartTime().toLocalDate().equals(date))
                            .mapToInt(IdleTime::getDurationSeconds)
                            .sum()
            );
            list.add(new WeeklyDaySummary(date, cpu, ram, uptime, idleSeconds));
        }
        data.setWeeklySummaries(list);
```

```
        return data;
    }
}
```

ReportService.java - сервіс, який працює лише з абстракціями.

```
public class ReportService {
    private final ReportRepository reportRepo;
    private final StatsRepository statsRepo;
    private final IdleRepository idleRepo;
    private ReportExporter exporter;
    public ReportService(ReportRepository rr, StatsRepository sr, IdleRepository ir) {
        this.reportRepo = rr;
        this.statsRepo = sr;
        this.idleRepo = ir;
    }
    public void setExporter(ReportExporter exporter) {
        this.exporter = exporter;
    }
    public Report generateReport(
            ReportFactory factory,
            User user,
            String name,
            LocalDate start,
            LocalDate end) {
        validateUser(user);
        validatePeriod(start, end);
        ReportContext ctx = new ReportContext(user, name, start, end);
        BaseReport baseReport = factory.createReport(ctx);
        LocalDateTime from = start.atStartOfDay();
        LocalDateTime to = end.atTime(23, 59, 59);
        List<SystemStats> stats = statsRepo.findByUserIdAndRecordedAtBetween(
                user.getId(), from, to);
        List<IdleTime> idle = idleRepo.findByUserIdAndStartTimeBetween(
                user.getId(), from, to);
        ReportCalculator calculator = factory.createCalculator();
        ReportData data = calculator.calculate(stats, idle);
        baseReport.build(data);
        Report stored = convertToEntity(baseReport, data);
        reportRepo.save(stored);
```

```
    return stored;
}
private Report convertToEntity(BaseReport base, ReportData data) {
    Report r = new Report();
    r.setUser(base.getUser());
    r.setReportName(base.getReportName());
    r.setPeriodStart(base.getPeriodStart());
    r.setPeriodEnd(base.getPeriodEnd());
    r.setCreatedAt(LocalDateTime.now());
    if (base instanceof DailyReport daily) {
        r.setDays(daily.getHourlyStats());
        r.setCpuAvg(averageFromDaySummary(daily.getHourlyStats(), HourStat::getAvgCpu));
        r.setRamAvg(averageFromDaySummary(daily.getHourlyStats(), HourStat::getAvgRam));
        List<SystemStats> stats = statsRepo.findByUserIdAndRecordedAtBetween(
                base.getUser().getId(),
                base.getPeriodStart().atStartOfDay(),
                base.getPeriodEnd().atTime(23, 59, 59)
        );
        List<IdleTime> idle = idleRepo.findByUserIdAndStartTimeBetween(
                base.getUser().getId(),
                base.getPeriodStart().atStartOfDay(),
                base.getPeriodEnd().atTime(23, 59, 59)
        );
        ReportCalculatorHelper h = new ReportCalculatorHelper();
        r.setAppUsagePercent(h.appUsagePercent(stats));
        r.setIdleTimeTotalSeconds(h.totalIdle(idle));
        r.setAvgUptimeHours(h.averageUptime(stats));
    }
 if (base instanceof WeeklyReport weekly) {
        // WeeklyDaySummary → DaySummary (порожні години)
        r.setDays(
                weekly.getDaySummaries().stream()
                        .map(w -> new DaySummary(w.getDate(), List.of()))
                        .collect(Collectors.toList())
        );
     List<WeeklyDaySummary> ws = weekly.getDaySummaries();
        r.setCpuAvg(averageBig(ws.stream().map(WeeklyDaySummary::getAvgCpu).toList()));
```

```
            r.setRamAvg(averageBig(ws.stream().map(WeeklyDaySummary::getAvgRam).toList()));
            r.setAvgUptimeHours(
                    averageBig(ws.stream().map(WeeklyDaySummary::getUptimeHours).toList())
            );
            r.setIdleTimeTotalSeconds(
                    sumBig(ws.stream().map(WeeklyDaySummary::getIdleSeconds).toList())
            );
            List<SystemStats> stats = statsRepo.findByUserIdAndRecordedAtBetween(
                    base.getUser().getId(),
                    base.getPeriodStart().atStartOfDay(),
                    base.getPeriodEnd().atTime(23, 59, 59)
            );
            ReportCalculatorHelper h = new ReportCalculatorHelper();
            r.setAppUsagePercent(h.appUsagePercent(stats));
        }
        return r;
    }
    private java.math.BigDecimal averageBig(List<java.math.BigDecimal> vals) {
        if (vals == null || vals.isEmpty()) return java.math.BigDecimal.ZERO;
        java.math.BigDecimal sum = vals.stream().reduce(java.math.BigDecimal.ZERO,
java.math.BigDecimal::add);
        return sum.divide(java.math.BigDecimal.valueOf(vals.size()), 2,
java.math.RoundingMode.HALF_UP);
    }
    private java.math.BigDecimal sumBig(List<java.math.BigDecimal> vals) {
        return vals.stream().reduce(java.math.BigDecimal.ZERO, java.math.BigDecimal::add);
    }
    private java.math.BigDecimal averageFromDaySummary(
            List<DaySummary> days,
            java.util.function.Function<HourStat, java.math.BigDecimal> getter
    ) {
        List<HourStat> hours = days.stream()
                .flatMap(d -> d.getHourlyStats().stream())
                .toList();
        if (hours.isEmpty()) return java.math.BigDecimal.ZERO;
        java.math.BigDecimal sum =
                hours.stream().map(getter).reduce(java.math.BigDecimal.ZERO, java.math.BigDecimal::add);
        return sum.divide(java.math.BigDecimal.valueOf(hours.size()), 2,
```

```
java.math.RoundingMode.HALF_UP);
    }
 public Report findById(Integer id) {
        return reportRepo.findById(id).orElse(null);
    }
    public List<Report> getReportsByUser(User user) {
        validateUser(user);
        return reportRepo.findByUserIdAndCreatedAtBetween(
                user.getId(),
                LocalDateTime.MIN,
                LocalDateTime.MAX
        );
    }
    public List<Report> getReportsInPeriod(User user, LocalDate start, LocalDate end) {
        validateUser(user);
        validatePeriod(start, end);
        return reportRepo.findByUserIdAndCreatedAtBetween(
                user.getId(),
                start.atStartOfDay(),
                end.atTime(23, 59, 59)
        );
    }
    public void deleteReport(Integer id) {
        reportRepo.deleteById(id);
    }
    public void restoreReport(Report report) {
        if (report != null)
            reportRepo.save(report);
    }
    public Path export(Report report) throws Exception {
        if (exporter == null)
            throw new IllegalStateException("ReportExporter strategy is not set.");

        return exporter.export(report);
    }
    public void deleteExportedFile(Path path) {
        try {
            Files.deleteIfExists(path);
```

```
        } catch (Exception ignored) {}
    }
    private void validateUser(User user) {
        if (user == null || user.getId() == null)
            throw new IllegalArgumentException("User is not defined.");
    }

    private void validatePeriod(LocalDate start, LocalDate end) {
        if (start == null || end == null || end.isBefore(start))
            throw new IllegalArgumentException("Invalid date range.");
    }
}
```

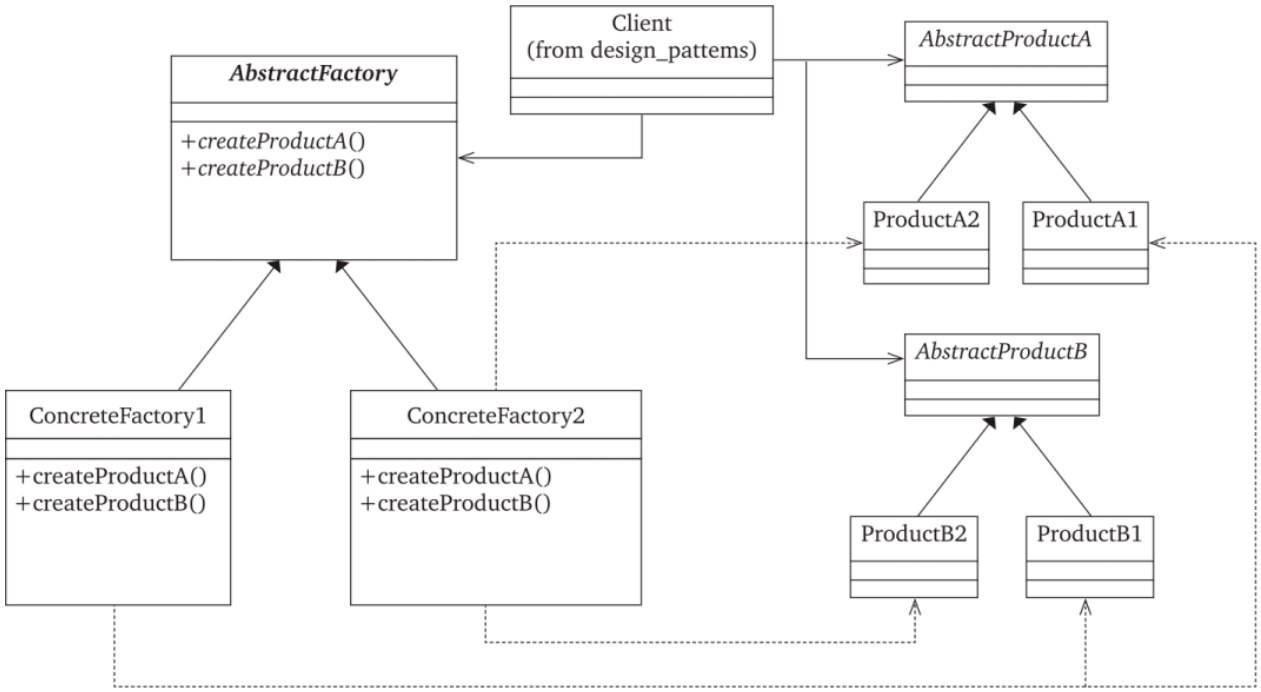
Висновок: У ході виконання лабораторної роботи було реалізовано шаблон проєктування «Абстрактна фабрика» для формування різних типів звітів у системі моніторингу активності. Такий підхід дозволив відокремити логіку створення звітів від бізнес-логіки сервісів, забезпечивши гнучкість, масштабованість і простоту розширення функціональності. Для кожного типу звіту (денного та тижневого) були створені власні фабрики, моделі та калькулятори, що дало змогу інкапсулювати відмінності у структурі та розрахунках, а сам сервіс став залежним лише від абстракцій.

Відповідь на контрольні питання:

1. Яке призначення шаблону «Абстрактна фабрика»?

Цей шаблон потрібен для створення цілих сімейств об’єктів, які повинні працювати разом. Створюється не один об’єкт, а групу пов’язаних об’єктів, і все це робиться через одну фабрику, не вказуючи конкретні класи.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



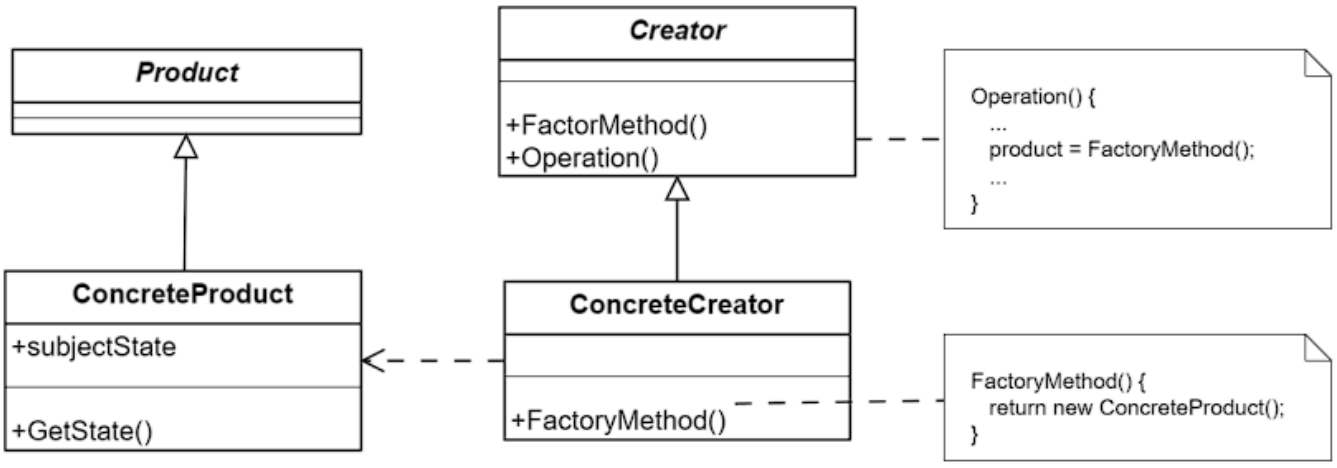
3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?

Абстрактна фабрика: описує набір методів для створення продуктів. Конкретні фабрики: реалізують методи та створюють конкретні продукти. Абстрактні продукти: описують інтерфейси продуктів. Конкретні продукти: реальна реалізація. Клієнт: працює через абстрактну фабрику й абстрактні продукти, не знаючи конкретних класів.

4. Яке призначення шаблону «Фабричний метод»?

Він дозволяє передати створення об’єкта у спадкоємців. Замість того, щоб створювати об’єкт напряму через new, ти оголошуєш метод, який кожен підклас перевизначає під себе.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

взаємодія?

Creator викликає фабричний метод. ConcreteCreator сам вирішує типовий продукт створити. Product / ConcreteProduct — результат створення. Клієнт працює тільки через Creator, не знаючи конкретних продуктів.

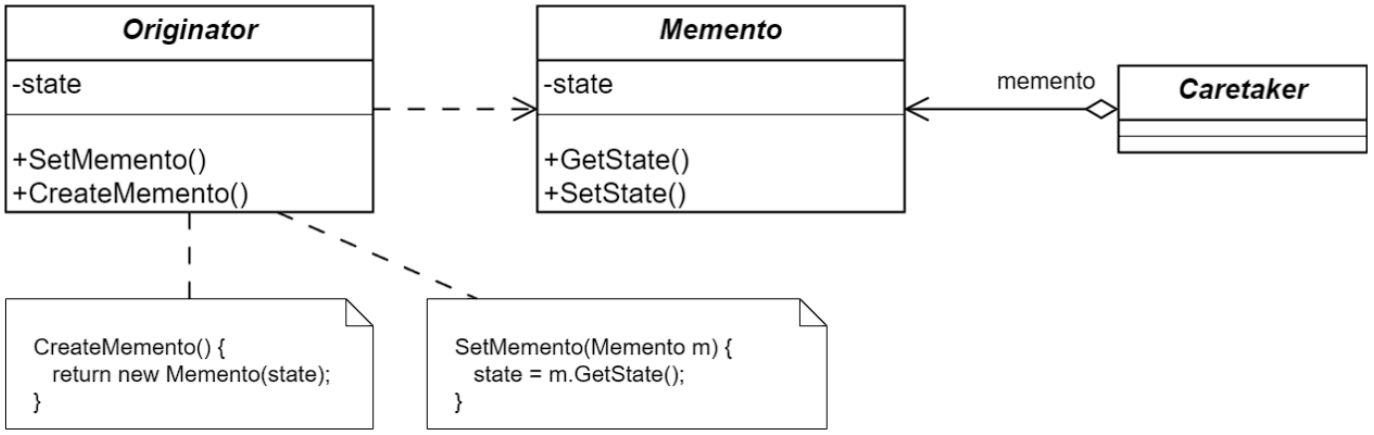
7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

по суті: Абстрактна фабрика створює цілі сімейства продуктів одразу. Фабричний метод створює один продукт, але створення переноситься у підкласи.

8. Яке призначення шаблону «Знімок»?

Зберегти стан об’єкта так, щоб можна було повернутися назад, але не розкривати внутрішню реалізацію цього об’єкта.

9. Нарисуйте структуру шаблону «Знімок».



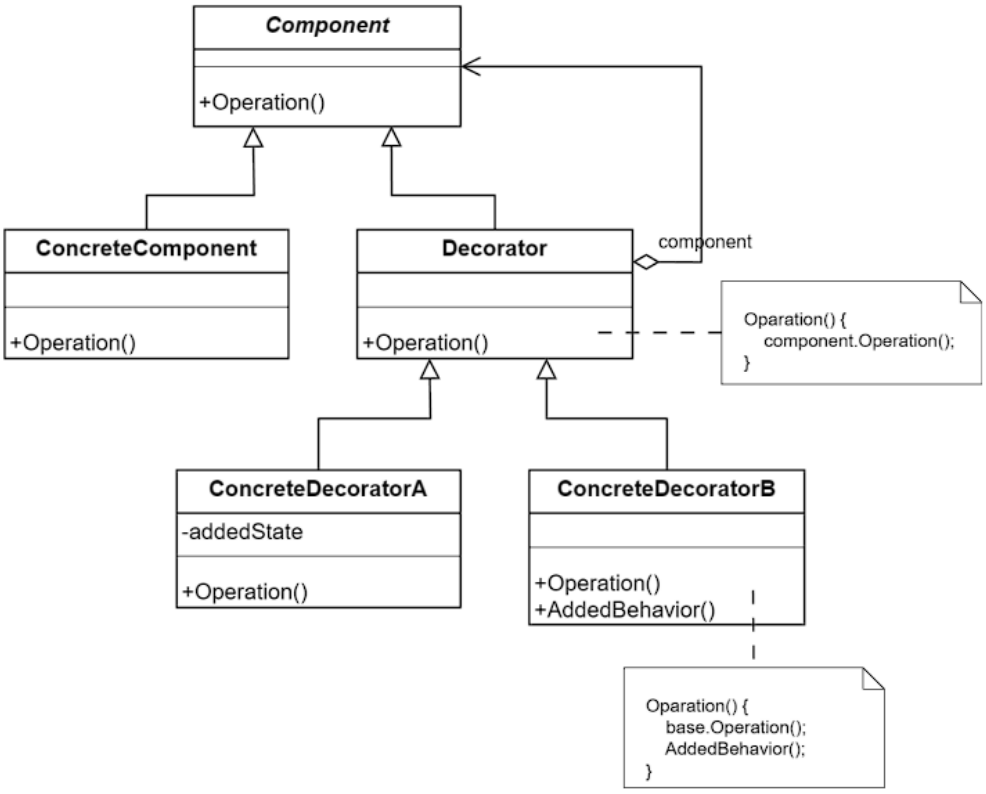
10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

Originator створює знімок і вміє себе відновлювати з нього. Memento зберігає внутрішній стан. Caretaker тримає історію, але не має доступу до вмісту Memento.

11. Яке призначення шаблону «Декоратор»?

Додає нову поведінку об’єкту без зміни його класу. Обгортає об’єкт у додаткові функції.

12. Нарисуйте структуру шаблону «Декоратор».



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

Component - загальний інтерфейс. ConcreteComponent - "основний" об’єкт. Decorator містить в собі компонент і делегує йому роботу. ConcreteDecorator додає зверху нову поведінку.

14. Які є обмеження використання шаблону «декоратор»?

Може бути занадто багато дрібних класів. Ланцюжки декораторів можуть стати заплутаними. Важко налагоджувати - не одразу видно, які декоратори застосовано. Декоратор працює лише якщо всі об’єкти мають один інтерфейс.