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

Теорія Розробки Програмного Забезпечення

Шаблон «State»

Предметна область: **Особиста бухгалтерія**

Виконав
студент групи ІА-14:
Яковенко Ю.О.

Перевірив:
Мягкий М.Ю.

Київ 2023

Мета: Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

Виконання

Завданням на лобораторну було реалізувати шаблон ***state***.

Я використав його в сервісному класі FinantialArrangement. Він працює з такою ж сутністю і репозиторієм. Реалізація полягає в тому, що клас моделі FinantialArrangement є єдиним для кредитів та депозитів. Таблиця в бд також одна.

Але деякі бізнес методи у кредитів та депозитів різні. Тож в класі FinantialArrangement є змінна state в якій може бути значення кредиту або депозиту. І в залежності від цього в сервісному класі обирається імплементація інтерфейсу FinantialArrangementCalculations.

Інтерфейс який реалізують стейт класи.

```
public interface FinancialArrangementCalculations {

    void makePayment(FinancialArrangement arrangement);

    int calculateCurrentSumInitValue(FinancialArrangement arrangement);

    int calculateRefundSum(FinancialArrangement arrangement);

    int calculatePaymentSumByTimeLine(FinancialArrangement arrangement);

    boolean isOutOfDate(FinancialArrangement arrangement);

    boolean isFullyRepaid(FinancialArrangement arrangement);

    int timeLineBetweenDates(LocalDate start, LocalDate end);

    FinancialArrangementState operatedState();
}
```

Імплементації.

```
@Component
public class CreditCalculations implements FinancialArrangementCalculations {

    @Override
    public void makePayment(FinancialArrangement arrangement) {
        arrangement.setCurrentSum(arrangement.getCurrentSum() -
calculatePaymentSumByTimeLine(arrangement));
    }

    @Override
    public int calculateCurrentSumInitValue(FinancialArrangement arrangement)
{
        return calculateRefundSum(arrangement);
    }
```

```
    @Override
    public int calculateRefundSum(FinancialArrangement arrangement) {
        int overpayment = arrangement.getStartSum() *
arrangement.getPercent() / 100;
        return arrangement.getStartSum() + Math.round(overpayment);
    }

    @Override
    public int calculatePaymentSumByTimeLine(FinancialArrangement
arrangement) {
        int paymentsNum = timeLineBetweenDates(
                arrangement.getStartDate(), arrangement.getEndDate());
        return (calculateRefundSum(arrangement) - arrangement.getStartSum())
/ paymentsNum;
    }

    @Override
    public boolean isOutOfDate(FinancialArrangement arrangement) {
        int requirePayments =
timeLineBetweenDates(arrangement.getStartDate(), LocalDate.now());
        if(requirePayments == 0) return false;

        int paidOut = calculateRefundSum(arrangement) -
arrangement.getCurrentSum();
        int madePayments = paidOut /
calculatePaymentSumByTimeLine(arrangement);

        return !(madePayments >= requirePayments);
    }

    @Override
    public boolean isFullyRepaid(FinancialArrangement arrangement) {
        return arrangement.getCurrentSum() == 0;
    }

    @Override
    public int timeLineBetweenDates(LocalDate start, LocalDate end) {
        return (int) ChronoUnit.MONTHS.between(start, end);
    }

    @Override
    public FinancialArrangementState operatedState() {
        return FinancialArrangementState.CREDIT;
    }
}
```

```
@Component
public class DepositCalculations implements FinancialArrangementCalculations
{

    @Override
    public void makePayment(FinancialArrangement arrangement) {
        arrangement.setCurrentSum(arrangement.getCurrentSum() +
calculatePaymentSumByTimeLine(arrangement));
    }

    @Override
    public int calculateCurrentSumInitValue(FinancialArrangement arrangement)
{
        return arrangement.getStartSum();
    }

    @Override
```

```
    public int calculateRefundSum(FinancialArrangement arrangement) {
        int onceProfit = arrangement.getStartSum() * arrangement.getPercent()
/ 100;
        int allProfit = onceProfit * timeLineBetweenDates(
                arrangement.getStartDate(), arrangement.getEndDate());

        return arrangement.getStartSum() + allProfit;
    }

    @Override
    public int calculatePaymentSumByTimeLine(FinancialArrangement
arrangement) {
        int paymentsNum = timeLineBetweenDates(
                arrangement.getStartDate(), arrangement.getEndDate());
        return  (calculateRefundSum(arrangement) - arrangement.getStartSum())
/ paymentsNum;
    }

    @Override
    public boolean isOutOfDate(FinancialArrangement arrangement) {
        int requirePayments =
timeLineBetweenDates(arrangement.getStartDate(), LocalDate.now());
        if(requirePayments == 0) return false;

        int profit = arrangement.getCurrentSum() - arrangement.getStartSum();
        int madePayments = profit /
calculatePaymentSumByTimeLine(arrangement);

        return !(madePayments >= requirePayments);
    }

    @Override
    public boolean isFullyRepaid(FinancialArrangement arrangement) {
        return arrangement.getCurrentSum() ==
calculateRefundSum(arrangement);
    }

    @Override
    public int timeLineBetweenDates(LocalDate start, LocalDate end) {
        return (int) ChronoUnit.YEARS.between(start, end);
    }

    @Override
    public FinancialArrangementState operatedState() {
        return FinancialArrangementState.DEPOSIT;
    }
}
```

В конфігурації я створюю бін списку цих класів.

```
@Bean
@Autowired
public List<FinancialArrangementCalculations>
financialArrangementCalculationsList(CreditCalculations credit,

DepositCalculations deposit) {
   return List.of(credit, deposit);
}
```

Впроваджую його в сервісний клас.

```
@Service
public class FinancialArrangementService {


    private final List<FinancialArrangementCalculations>
financialArrangementCalculationsList;

    @Autowired
    public FinancialArrangementService(List<FinancialArrangementCalculations>
financialArrangementCalculationsList) {

        this.financialArrangementCalculationsList =
financialArrangementCalculationsList;
    }
```

Приклад одного з методів де використовується FinantialArrangementCalculations.

```
@Transactional(readOnly = true)
public FinancialArrangement getOne(int id) {
    FinancialArrangement arrangement =
financialArrangementRepository.findById(id)
            .orElseThrow(() -> new IllegalArgumentException("Invalid id: " +
id));

    //TODO: find a prettier way to choose FinancialArrangementCalculations
    FinancialArrangementCalculations calculations =
getCalculationsByState(arrangement.getState());

    arrangement.setRefundSum(calculations.calculateRefundSum(arrangement));
    if(calculations.isOutOfDate(arrangement))
arrangement.setStatus(Status.OVERDUE);

    return arrangement;
}
```

Та метод за допомогою якого обирається імплементація.

```
private FinancialArrangementCalculations
getCalculationsByState(FinancialArrangementState state)
        throws IllegalArgumentException{
    return financialArrangementCalculationsList.stream()
            .filter(c -> c.operatedState() != null)
            .filter(c -> c.operatedState().equals(state))
            .findAny()
            .orElseThrow(() -> new IllegalArgumentException("No such
state"));
}
```

Висновок: В даній лабораторній роботі я реалізував частину проекту використавши шаблон проектування “State”