Java ChargeService Refactor

I have a Java class in my project that I want to follow Java Solid Principles. Infer the class's responsibility and look for ways to improve it, ONLY if applicable.

```
package com.toolstore.demo.service;

import com.toolstore.demo.model.ToolType;
import org.springframework.stereotype.Service;

import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.Month;
import java.time.temporal.TemporalAdjusters;

@Service
public class ChargeService {

    public int calculateChargeDays(ToolType toolType, LocalDate startDate, LocalDate endDate) {
        int chargeDays = 0;
        LocalDate currentDate = startDate;

        while (!currentDate.isAfter(endDate)) {
            if (isChargeable(toolType, currentDate)) {
                chargeDays++;
            }
            currentDate = currentDate.plusDays(1);
        }

        return chargeDays;
    }

    private boolean isChargeable(ToolType toolType, LocalDate date) {
        boolean isWeekend = isWeekend(date);
        boolean isHoliday = isHoliday(date);

        if (isWeekend && !toolType.isWeekendCharge()) {
            return false;
        }

        if (isHoliday && !toolType.isHolidayCharge()) {
            return false;
        }

        if (!isWeekend && !toolType.isWeekdayCharge()) {
            return false;
        }

        return true;
    }

    private boolean isWeekend(LocalDate date) {
        DayOfWeek dayOfWeek = date.getDayOfWeek();
        return dayOfWeek == DayOfWeek.SATURDAY || dayOfWeek == DayOfWeek.SUNDAY;
    }

    private boolean isHoliday(LocalDate date) {
        return isIndependenceDay(date) || isLaborDay(date);
    }

    private boolean isIndependenceDay(LocalDate date) {
        int year = date.getYear();
        LocalDate july4th = LocalDate.of(year, Month.JULY, 4);

        // If July 4th falls on Saturday, observed on Friday (July 3rd)
        if (july4th.getDayOfWeek() == DayOfWeek.SATURDAY) {
            return date.equals(july4th.minusDays(1));
        }

        // If July 4th falls on Sunday, observed on Monday (July 5th)
        if (july4th.getDayOfWeek() == DayOfWeek.SUNDAY) {
            return date.equals(july4th.plusDays(1));
        }

        // Otherwise, observed on July 4th itself
        return date.equals(july4th);
    }

    private boolean isLaborDay(LocalDate date) {
        // Labor Day is the first Monday in September
        int year = date.getYear();
        LocalDate firstMondayInSeptember = LocalDate.of(year, Month.SEPTEMBER, 1)
                .with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY));
```

~~~~
        return date.equals(firstMondayInSeptember);
    }
}```
~~~~

RouteLLM

✅ Inferred Responsibility of ChargeService

The class is responsible for:

> **Calculating the number of chargeable rental days for a tool between two dates based on tool charging rules (weekday/weekend/holiday).**

That’s a clear and valid domain responsibility. 👍
However, there are **some SOLID improvements that are applicable**, particularly around **Single Responsibility** and **Open/Closed**.

🔎 SOLID Analysis & Improvements

1️⃣ Single Responsibility Principle (SRP)

❌ Current Issue

ChargeService currently handles **two different responsibilities**:

1. ✅ Calculating charge days
2. ❌ Determining what qualifies as a holiday (business calendar logic)

Holiday logic (Independence Day, Labor Day) is unrelated to “charge calculation.”
It’s **calendar/business rule logic** and should be separate.

If tomorrow you add:

- Memorial Day
- Thanksgiving
- Company-specific holidays
- Different country holidays

You’ll be modifying ChargeService.

That’s a red flag.

✅ Recommended Refactor

Extract a HolidayService (or BusinessCalendarService).

Example:

```
public interface HolidayService {
    boolean isHoliday(LocalDate date);
}
```

Implementation:

```
@Service
public class UsHolidayService implements HolidayService {

    @Override
    public boolean isHoliday(LocalDate date) {
        return isIndependenceDay(date) || isLaborDay(date);
    }

    private boolean isIndependenceDay(LocalDate date) {
        ...
    }

    private boolean isLaborDay(LocalDate date) {
        ...
    }
}
```

Then inject into ChargeService:

```
@Service
public class ChargeService {

    private final HolidayService holidayService;

    public ChargeService(HolidayService holidayService) {
        this.holidayService = holidayService;
    }

    private boolean isChargeable(ToolType toolType, LocalDate date) {
        boolean isWeekend = isWeekend(date);
        boolean isHoliday = holidayService.isHoliday(date);
        ...
    }
}
```

✅ Now ChargeService only calculates charges
✅ Holiday rules are isolated
✅ Easier to test
✅ More maintainable

2️⃣ Open/Closed Principle (OCP)

Right now:

- Adding a new holiday requires modifying ChargeService
- That violates OCP (class should be open for extension, closed for modification)

By extracting HolidayService, you can:

- Add CanadaHolidayService
- Add CompanyHolidayService
- Add composite holiday strategies

Without touching ChargeService.

3️⃣ Dependency Inversion Principle (DIP)

Currently:

ChargeService depends on concrete holiday logic.

After refactor:

It depends on HolidayService interface ✅
That aligns perfectly with DIP.

4️⃣ Minor Clean Code Improvements

These aren’t SOLID violations, but worth noting:

✅ Simplify isChargeable

Instead of multiple if returns:

```
private boolean isChargeable(ToolType toolType, LocalDate date) {
    boolean isWeekend = isWeekend(date);
    boolean isHoliday = holidayService.isHoliday(date);

    if (isHoliday) {
        return toolType.isHolidayCharge();
    }

    if (isWeekend) {
        return toolType.isWeekendCharge();
    }

    return toolType.isWeekdayCharge();
}
```

This improves readability and removes overlapping checks.

5️⃣ Optional Improvement: Encapsulate Charging Rules

Right now ToolType exposes:

```
isWeekendCharge()
isHolidayCharge()
isWeekdayCharge()
```

A more object-oriented approach would be:

```
public boolean isChargeable(LocalDate date, BusinessCalendar calendar)
```

Then ToolType owns its charging behavior.

This would move toward:

- Better encapsulation
- Rich domain model
- Lower service logic

But this depends on your architecture style (DDD vs anemic model).

✅ Final Verdict

The class is mostly good.

But the following improvement **is applicable and recommended**:

✅ Extract Holiday logic into its own service

That fixes:

- SRP
- OCP
- DIP

- Testability
- Maintainability

Everything else is optional refinement.

🎯 Clean Architecture Version (Conceptually)

- ChargeService → calculates charge days
- HolidayService → determines holidays
- ToolType → contains charge configuration
- BusinessCalendar (optional) → handles weekend/holiday logic

---

If you’d like, I can refactor the full class into a SOLID-compliant version for you.