Checkout Service Refactoring

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.RentalAgreement;
import com.toolstore.demo.model.Tool;
import com.toolstore.demo.repository.ToolRepository;
import org.springframework.stereotype.Service;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.time.LocalDate;

@Service
public class CheckoutService {

    private final ToolRepository toolRepository;
    private final ChargeService chargeService;

    public CheckoutService(ToolRepository toolRepository,
ChargeService chargeService) {
        this.toolRepository = toolRepository;
        this.chargeService = chargeService;
    }

    public RentalAgreement checkout(String toolCode, int
rentalDayCount, int discountPercent, LocalDate checkoutDate) {
        // Validation
        validateRentalDayCount(rentalDayCount);
        validateDiscountPercent(discountPercent);

        // Get tool
        Tool tool = toolRepository.findByCode(toolCode)
                .orElseThrow(() -> new IllegalArgumentException("Tool
not found: " + toolCode));

        // Calculate due date
        LocalDate dueDate = checkoutDate.plusDays(rentalDayCount);

        // Calculate charge days
        int chargeDays = chargeService.calculateChargeDays(
~~~

```
                tool.toolType(),
                checkoutDate.plusDays(1), // Start from day after checkout
                dueDate
        );

        // Calculate charges
        BigDecimal dailyCharge = tool.toolType().getDailyCharge();
        BigDecimal preDiscountCharge = dailyCharge
                .multiply(BigDecimal.valueOf(chargeDays))
                .setScale(2, RoundingMode.HALF_UP);

        BigDecimal discountAmount = preDiscountCharge
                .multiply(BigDecimal.valueOf(discountPercent))
                .divide(BigDecimal.valueOf(100), 2,
RoundingMode.HALF_UP);

        BigDecimal finalCharge =
preDiscountCharge.subtract(discountAmount);

        // Build rental agreement
        return RentalAgreement.builder()
                .toolCode(tool.code())
                .toolType(tool.toolType().name())
                .toolBrand(tool.brand())
                .rentalDays(rentalDayCount)
                .checkoutDate(checkoutDate)
                .dueDate(dueDate)
                .dailyRentalCharge(dailyCharge)
                .chargeDays(chargeDays)
                .preDiscountCharge(preDiscountCharge)
                .discountPercent(discountPercent)
                .discountAmount(discountAmount)
                .finalCharge(finalCharge)
                .build();
    }

    private void validateRentalDayCount(int rentalDayCount) {
        if (rentalDayCount < 1) {
            throw new IllegalArgumentException("Rental day count
must be 1 or greater");
        }
    }

    private void validateDiscountPercent(int discountPercent) {
        if (discountPercent < 0 || discountPercent > 100) {
            throw new IllegalArgumentException("Discount percent
```

~~~
must be between 0 and 100");
        }
    }
}```
~~~

RouteLLM

✅ Inferred Responsibility

CheckoutService is responsible for **orchestrating the checkout process** and producing a RentalAgreement.

It currently:

1. Validates input
2. Fetches a Tool
3. Calculates due date
4. Delegates charge-day calculation
5. Calculates monetary amounts
6. Builds a RentalAgreement

So its true responsibility is:

> **Coordinate the checkout use case and produce a rental agreement.**

That’s good — orchestration is an appropriate responsibility for an application service.

🔍 SOLID Evaluation

✅ Single Responsibility Principle (SRP)

Issue: It’s doing too much.

It currently handles:

- Validation

- Date logic
- Money calculation
- Discount calculation
- Agreement construction

These are multiple reasons to change:

- Change in validation rules
- Change in discount policy
- Change in charge calculation
- Change in rounding rules
- Change in agreement construction

That violates SRP.

✅ Suggested Improvement

Move business logic into dedicated domain services:

- RentalValidationService
- PricingService
- RentalAgreementFactory

Then CheckoutService becomes a pure orchestrator.

Example:

```
public RentalAgreement checkout(...) {

    rentalValidator.validate(rentalDayCount, discountPercent);

    Tool tool = toolRepository.findByCode(toolCode)
            .orElseThrow(() -> new ToolNotFoundException(toolCode));

    return rentalAgreementFactory.create(
            tool,
            rentalDayCount,
            discountPercent,
            checkoutDate
    );
}
```

Now:

- CheckoutService = use case coordinator
- Business rules live elsewhere

Much cleaner.

✅ Open/Closed Principle (OCP)

Currently:

- Discount calculation is hardcoded
- Rounding is hardcoded
- Charge calculation logic partly lives here

If discount rules change (e.g., tiered discount, seasonal discount), you must modify this class.

✅ Improvement

Introduce a DiscountPolicy interface:

```
public interface DiscountPolicy {
    BigDecimal calculateDiscount(BigDecimal preDiscountCharge);
}
```

Inject it:

```
private final DiscountPolicy discountPolicy;
```

Now you can add new discount strategies without modifying CheckoutService.

✅ Liskov Substitution Principle (LSP)

No obvious violation here. Nothing is inheriting from this class.

✅ Safe.

✅ Interface Segregation Principle (ISP)

Not violated directly.

But if ChargeService becomes too broad, consider splitting it:

- ChargeDayCalculator
- HolidayPolicy
- etc.

Right now it seems fine.

✅ Dependency Inversion Principle (DIP)

You depend on:

- ToolRepository
- ChargeService

If these are interfaces (they should be), you're following DIP correctly.

One improvement:

- Don’t depend on concrete exception types (IllegalArgumentException) for business errors.
- Use domain-specific exceptions:

```
throw new ToolNotFoundException(toolCode);
```

Cleaner, more expressive, and better layered.

💡 Additional Architectural Improvements

1️⃣ Move Monetary Calculations to a Pricing Service

Money logic should not live in orchestration.

Extract:

```
public PricingDetails calculatePricing(
    Tool tool,
    int chargeDays,
    int discountPercent
);
```

Encapsulate:

- pre-discount charge
- discount amount
- final charge
- rounding rules

Now rounding policy changes don’t affect checkout.

2️⃣ Move RentalAgreement Construction to Factory

Instead of:

```
RentalAgreement.builder()...
```

Use:

```
rentalAgreementFactory.create(...)
```

This:

- Centralizes construction rules
- Prevents duplicated agreement-building elsewhere
- Keeps CheckoutService clean

3️⃣ Use Domain-Specific Value Objects

Instead of primitive obsession:

```
int rentalDayCount
int discountPercent
```

Use:

```
RentalDays
DiscountPercentage
```

This:

- Encapsulates validation
- Prevents invalid states
- Improves domain expressiveness

Example:

```
public record DiscountPercentage(int value) {
    public DiscountPercentage {
        if (value < 0 || value > 100)
            throw new IllegalArgumentException("Invalid discount");
    }
}
```

Now validation disappears from CheckoutService entirely.

✅ What You Did Well

✔ Proper constructor injection
✔ Delegated charge-day logic
✔ Immutable return object (builder)
✔ Clear orchestration flow
✔ Proper rounding using BigDecimal

This is already good code.

🧠 What an Ideal SOLID Version Would Look Like

CheckoutService becomes:

```
public RentalAgreement checkout(CheckoutRequest request) {

    Tool tool = toolRepository.findByCode(request.toolCode())
            .orElseThrow(() -> new ToolNotFoundException(request.toolCode()));

    return rentalAgreementFactory.create(tool, request);
}
```

All business logic moves to domain layer.

Now:

- CheckoutService = Application layer
- Pricing rules = Domain layer
- Validation = Value objects
- Construction = Factory

Perfect separation.

🎯 Final Verdict

Is refactoring required?
👉 **Not strictly — your class is acceptable for small projects.**

Is improvement possible under SOLID?
👉 **Yes — especially SRP and OCP improvements.**

Your class is currently:

> A well-written but slightly overloaded orchestration service.

If this were production enterprise code, I would refactor pricing + discount + validation out.

If this is a coding assignment (Toolstore kata style), it is perfectly acceptable.

If you'd like, I can show you a **fully refactored SOLID version** of this exact class structure.