

RentalAgreement SOLID Review

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 possible

```
```package com.toolstore.demo.model;
```

```
import com.fasterxml.jackson.annotation.JsonFormat;
import lombok.Builder;
import lombok.Getter;
```

```
import java.math.BigDecimal;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.text.NumberFormat;
import java.util.Locale;
```

```
@Getter
@Builder
public class RentalAgreement {
```

```
 private String toolCode;
 private String toolType;
 private String toolBrand;
 private int rentalDays;
```

```
 @JsonFormat(pattern = "MM/dd/yy")
 private LocalDate checkoutDate;
```

```
 @JsonFormat(pattern = "MM/dd/yy")
 private LocalDate dueDate;
```

```
 private BigDecimal dailyRentalCharge;
 private int chargeDays;
 private BigDecimal preDiscountCharge;
 private int discountPercent;
 private BigDecimal discountAmount;
 private BigDecimal finalCharge;
```

```
 // Output format specification
 // Move to formatter
 @Override
 public String toString() {
 DateTimeFormatter dateFormatter =
```

```

DateTimeFormatter.ofPattern("MM/dd/yy");
 NumberFormat currencyFormatter =
NumberFormat.getCurrencyInstance(Locale.US);

 return "Tool code: " + toolCode + "\n" +
 "Tool type: " + toolType + "\n" +
 "Tool brand: " + toolBrand + "\n" +
 "Rental days: " + rentalDays + "\n" +
 "Check out date: " + checkoutDate.format(dateFormatter)
+ "\n" +
 "Due date: " + dueDate.format(dateFormatter) + "\n" +
 "Daily rental charge: " +
currencyFormatter.format(dailyRentalCharge) + "\n" +
 "Charge days: " + chargeDays + "\n" +
 "Pre-discount charge: " +
currencyFormatter.format(preDiscountCharge) + "\n" +
 "Discount percent: " + discountPercent + "%" + "\n" +
 "Discount amount: " +
currencyFormatter.format(discountAmount) + "\n" +
 "Final charge: " + currencyFormatter.format(finalCharge);
 }
}
```

```

RouteLLM

Inferred Responsibility of RentalAgreement

From the code, this class appears to represent:

A domain model that encapsulates the calculated results of a tool rental transaction.

It stores:

- Tool metadata (toolCode, toolType, toolBrand)
- Rental period info (rentalDays, checkoutDate, dueDate)
- Pricing breakdown (dailyRentalCharge, chargeDays, preDiscountCharge, discountPercent, discountAmount, finalCharge)

So its **core responsibility** is to act as a **data holder for a finalized rental agreement**.

SOLID Principle Review

Let's examine it through SOLID.

1 Single Responsibility Principle (SRP)

A class should have only one reason to change.

Current Violation

RentalAgreement has **two responsibilities**:

1.  Holding rental agreement data (domain responsibility)
2.  Formatting output for presentation (`toString()` formatting logic)

Formatting involves:

- Date formatting
- Currency formatting
- String layout specification

These are **presentation concerns**, not domain concerns.

If formatting rules change (e.g., switch to EU locale, JSON output, different print layout), this class must change — violating SRP.

Recommended Improvement

Move formatting logic to a separate class:

```

public class RentalAgreementFormatter {

    private static final DateTimeFormatter DATE_FORMAT =
        DateTimeFormatter.ofPattern("MM/dd/yy");

    private static final NumberFormat CURRENCY_FORMAT =
        NumberFormat.getCurrencyInstance(Locale.US);

    public static String format(RentalAgreement agreement) {
        return "Tool code: " + agreement.getToolCode() + "\n" +
            "Tool type: " + agreement.getToolType() + "\n" +
            "Tool brand: " + agreement.getToolBrand() + "\n" +
            "Rental days: " + agreement.getRentalDays() + "\n" +
            "Check out date: " + agreement.getCheckoutDate().format(DATE_FORMAT) + "\n" +
+
            "Due date: " + agreement.getDueDate().format(DATE_FORMAT) + "\n" +
            "Daily rental charge: " +
            CURRENCY_FORMAT.format(agreement.getDailyRentalCharge()) + "\n" +
            "Charge days: " + agreement.getChargeDays() + "\n" +
            "Pre-discount charge: " +
            CURRENCY_FORMAT.format(agreement.getPreDiscountCharge()) + "\n" +
            "Discount percent: " + agreement.getDiscountPercent() + "%\n" +
            "Discount amount: " +
            CURRENCY_FORMAT.format(agreement.getDiscountAmount()) + "\n" +
            "Final charge: " + CURRENCY_FORMAT.format(agreement.getFinalCharge());
    }
}

```

Then remove the overridden `toString()` entirely (or make it simple).

 Now:

- `RentalAgreement` = domain model
- `RentalAgreementFormatter` = presentation logic

Clean separation of concerns.

2 Open/Closed Principle (OCP)

Currently:

- Changing format → modify the class
- Supporting multiple output formats → modify the class

After refactoring:

- Add new formatters (Console, JSON, PDF, etc.)

- No changes to RentalAgreement
-  Improved OCP compliance.
-

3 Liskov Substitution Principle (LSP)

No inheritance here.

-  No issues.
-

4 Interface Segregation Principle (ISP)

Not applicable — no interfaces involved.

-  No issue.
-

5 Dependency Inversion Principle (DIP)

Currently:

- RentalAgreement depends on concrete formatting implementations (NumberFormat, DateTimeFormatter).

After extracting formatter:

- Domain layer no longer depends on formatting concerns.

-  Better separation of layers.
-

Additional Improvements (Optional but Valuable)

1. Make It More Domain-Driven

You could consider:

- Extracting a Money value object instead of using BigDecimal
- Extracting a Tool value object instead of repeating tool fields

Example:

```
public class Tool {  
    private String code;  
    private String type;  
    private String brand;  
}
```

Then:

```
private Tool tool;
```

This reduces primitive obsession.

2. Validation

Currently:

- discountPercent could be negative or > 100
- rentalDays could be negative
- chargeDays could be inconsistent

You might enforce invariants in:

- A factory method
 - A service layer
 - Or a validated builder
-

3. Immutability (Already Good)

Using:

@Getter
@Builder

And no setters → good immutability practice.

You could optionally make the class final.

Final Recommendation Summary

Required Improvement (SOLID Fix)

- Move `toString()` formatting to a separate `RentalAgreementFormatter`

Optional Improvements

- Extract `Tool` value object
 - Consider Money abstraction
 - Add validation/invariants
 - Make class final
-

Final Verdict

The class is **mostly clean and well-structured**, but:

The only clear SOLID violation is mixing domain data with formatting logic.

After extracting formatting, the class becomes a very clean domain model aligned with SOLID principles.

If you'd like, I can also:

- Refactor it to a clean DDD-style version
 - Show a layered architecture version
 - Or show how this would look in a clean architecture setup
-

Excellent Now lets do the same with this class.

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

import com.toolstore.demo.model.RentalAgreement;
import org.springframework.stereotype.Component;

import java.io.PrintStream;

@Component
public class RentalAgreementFormatter {

 private final PrintStream output;

 // Default constructor uses System.out
 public RentalAgreementFormatter() {
 this(System.out);
 }

 // Constructor for testing with custom PrintStream
 public RentalAgreementFormatter(PrintStream output) {
 this.output = output;
 }

 // Prints the rental agreement output stream (console by default)
 public void print(RentalAgreement agreement) {
 output.println(agreement.toString());
 }

 // Returns the formatted string representation without printing.
 // Useful for testing or capturing output.
 public String format(RentalAgreement agreement) {
 return agreement.toString();
 }
}
```

```

Directly related to the previous class. Perform the same analysis as done on the previous class. However a problem I have is that throughout the code I wish to easily print rental agreement in the specified format like so

```
```
RentalAgreement agreement = checkoutService.checkout...
...
System.out.println(agreement);
```

```

Is there a way to fix these classes without adjusting the other code too much?

RouteLLM

Let me first check the project specification for context.

Inferred Responsibility of RentalAgreementFormatter

A service that formats and prints a RentalAgreement to an output stream.

It handles:

- Delegating formatting to RentalAgreement.toString()
 - Printing to a PrintStream (console by default, injectable for testing)
-

SOLID Principle Review

1 Single Responsibility Principle (SRP)

This class has **two responsibilities**:

1. **Formatting** — format() returns a string representation
2. **Printing** — print() sends it to an output stream

These are separate concerns. However, given the class is already named RentalAgreementFormatter, the formatting responsibility is appropriate. The print() method could arguably belong to a RentalAgreementPrinter, but this is a **minor concern** — the split is thin enough that it's acceptable here.

 **Verdict: Acceptable as-is**, but see the bigger issue below.

2 The Real Problem — Delegation Without Value

```
public String format(RentalAgreement agreement) {  
    return agreement.toString(); // ← just delegating back to the model  
}
```

From the **previous analysis**, we established that `RentalAgreement.toString()` **should not contain formatting logic** — that belongs here. But currently:

- `RentalAgreementFormatter.format()` just calls `agreement.toString()`
- All the real formatting logic **still lives in `RentalAgreement`**

So the formatter is a **hollow wrapper** — it doesn't actually format anything.

3 Addressing Your Specific Problem

You want this to work cleanly:

```
System.out.println(agreement); // prints in the specified format
```

The good news: **you can have both** — a clean `toString()` on the model **and** a dedicated formatter — without contradiction.

The key insight from the spec is:

"The Rental Agreement must include a method to print values to console"

This means `toString()` **is** the specified output format. It's not a violation — it's a **requirement**. The previous `toString()` comment // Move to formatter was overzealous.



Recommended Solution

Keep `toString()` in `RentalAgreement` — it IS the format

Move the formatting constants out of the method body (minor cleanup):

```
@Getter  
@Builder  
public final class RentalAgreement {  
  
    private static final DateTimeFormatter DATE_FORMAT =  
        DateTimeFormatter.ofPattern("MM/dd/yy");  
    private static final NumberFormat CURRENCY_FORMAT =  
        NumberFormat.getCurrencyInstance(Locale.US);  
  
    // ... fields ...  
  
    @Override  
    public String toString() {  
        return "Tool code: " + toolCode + "\n" +  
            "Tool type: " + toolType + "\n" +  
            "Tool brand: " + toolBrand + "\n" +  
            "Rental days: " + rentalDays + "\n" +  
            "Check out date: " + checkoutDate.format(DATE_FORMAT) + "\n" +  
            "Due date: " + dueDate.format(DATE_FORMAT) + "\n" +  
            "Daily rental charge: " + CURRENCY_FORMAT.format(dailyRentalCharge) + "\n" +  
            "Charge days: " + chargeDays + "\n" +  
            "Pre-discount charge: " + CURRENCY_FORMAT.format(preDiscountCharge) + "\n"  
        +  
        "Discount percent: " + discountPercent + "%\n" +  
        "Discount amount: " + CURRENCY_FORMAT.format(discountAmount) + "\n" +  
        "Final charge: " + CURRENCY_FORMAT.format(finalCharge);  
    }  
}
```

Simplify `RentalAgreementFormatter` to just a Printer

Since `toString()` owns the format, the formatter's only real job is **printing**. Rename or slim it down:

```
@Component  
public class RentalAgreementPrinter {  
  
    private final PrintStream output;  
  
    public RentalAgreementPrinter() {
```

```

        this(System.out);
    }

    public RentalAgreementPrinter(PrintStream output) {
        this.output = output;
    }

    public void print(RentalAgreement agreement) {
        output.println(agreement); // toString() called implicitly
    }
}

```

- ✓ format() is removed — it was just toString() anyway
 - ✓ print() is the single responsibility
 - ✓ System.out.println(agreement) works perfectly everywhere
 - ✓ Testable via injected PrintStream
 - ✓ No other code needs to change
-

Summary

| Concern | Where it lives | Why |
|---------------------------------|--|---|
| Formatted string representation | RentalAgreement.toString() | Spec requires it; enables System.out.println(agreement) |
| Printing to output stream | RentalAgreementPrinter | Single responsibility, injectable, testable |
| Formatting constants | static final fields in RentalAgreement | Avoids re-instantiation on every call |

The format() method in the original RentalAgreementFormatter was redundant — removing it and renaming the class to RentalAgreementPrinter gives you a clean, focused, SOLID-compliant design with **zero changes** needed to calling code.

this implementation makes the formatter pointless for the rest of the code. Also the domain object is responsible for formatting. The specification declares a method that formats and prints the rental agreement to the console. This could be through a chain of methods or classes, it does not refer to the rental agreement class / object.

RouteLLM

You are absolutely right. If we want to strictly follow **SOLID**, the domain object (`RentalAgreement`) should only hold data, and the **Formatter** should own the logic of how that data is presented.

The challenge is satisfying your requirement:

```
System.out.println(agreement);
```

In Java, `System.out.println(obj)` **always** calls `obj.toString()`. To make this work without putting formatting logic inside the domain class, we can use a **Decorator pattern** or a **Wrapper**.



The SOLID Refactor

1. The Domain Model (Pure Data)

Remove all formatting logic, `NumberFormat`, and `DateTimeFormatter` from here. It is now a simple POJO/Data object.

```
@Getter  
@Builder  
public class RentalAgreement {  
    private String toolCode;  
    private String toolType;  
    private String toolBrand;  
    private int rentalDays;  
    private LocalDate checkoutDate;  
    private LocalDate dueDate;  
    private BigDecimal dailyRentalCharge;  
    private int chargeDays;  
    private BigDecimal preDiscountCharge;  
    private int discountPercent;  
    private BigDecimal discountAmount;  
    private BigDecimal finalCharge;  
  
    // Default toString for debugging only (e.g., IDE generated)  
    @Override  
    public String toString() {  
        return "RentalAgreement(toolCode=" + toolCode + ", ...)";  
    }  
}
```

2. The Formatter (Pure Logic)

This class now owns the "How" of the specification. It is a stateless utility or service.

```
@Component
public class RentalAgreementFormatter {
    private static final DateTimeFormatter DATE_FMT = DateTimeFormatter.ofPattern("MM/
dd/yy");
    private static final NumberFormat CURRENCY_FMT =
NumberFormat.getCurrencyInstance(Locale.US);

    public String format(RentalAgreement a) {
        return String.join("\n",
            "Tool code: " + a.getToolCode(),
            "Tool type: " + a.getToolType(),
            "Tool brand: " + a.getToolBrand(),
            "Rental days: " + a.getRentalDays(),
            "Check out date: " + a.getCheckoutDate().format(DATE_FMT),
            "Due date: " + a.getDueDate().format(DATE_FMT),
            "Daily rental charge: " + CURRENCY_FMT.format(a.getDailyRentalCharge()),
            "Charge days: " + a.getChargeDays(),
            "Pre-discount charge: " + CURRENCY_FMT.format(a.getPreDiscountCharge()),
            "Discount percent: " + a.getDiscountPercent() + "%",
            "Discount amount: " + CURRENCY_FMT.format(a.getDiscountAmount()),
            "Final charge: " + CURRENCY_FMT.format(a.getFinalCharge())
        );
    }
}
```

3. The "Bridge" (The Secret Sauce)

To allow System.out.println(agreement) to work **without** putting logic in the model, we create a lightweight **Display Wrapper**.

```
public class FormattedRentalAgreement {
    private final RentalAgreement agreement;
    private final RentalAgreementFormatter formatter;

    public FormattedRentalAgreement(RentalAgreement agreement,
RentalAgreementFormatter formatter) {
        this.agreement = agreement;
    }
```

```
        this.formatter = formatter;
    }

    @Override
    public String toString() {
        return formatter.format(agreement);
    }
}
```



How to use it in your code

In your CheckoutService (or wherever the agreement is created), you simply wrap the result before returning it or printing it.

```
// Inside your service
public FormattedRentalAgreement checkout(...) {
    RentalAgreement data = // ... calculation logic ...

    // Wrap the data with the formatter
    return new FormattedRentalAgreement(data, formatter);
}

// In your main/controller code
FormattedRentalAgreement agreement = checkoutService.checkout(...);

// THIS NOW WORKS and follows the spec perfectly!
System.out.println(agreement);
```

Why this is better SOLID:

1. SRP (Single Responsibility):

- RentalAgreement only knows about rental data.
- RentalAgreementFormatter only knows about strings, dates, and currency symbols.
- FormattedRentalAgreement only knows how to bridge the two for the console.

2. OCP (Open/Closed):

If you need to output to a Web UI or a PDF later, you just create a RentalAgreementPdfFormatter. You don't touch the RentalAgreement class.

3. **DIP (Dependency Inversion)**: The printing mechanism is no longer hardcoded into the data object.
4. **The "Print" Requirement**: The specification's requirement to "print values to console" is satisfied by the FormattedRentalAgreement.toString() being passed to System.out.println.

This approach keeps your "easy printing" syntax while completely decoupling the formatting logic from your business logic.

How might the package structure look?

Currently I have:

/model/RentalAgreement

/Service/RentalAgreementFormatter

The formatter does not feel like it should be a service.

RouteLLM

You are correct—a **Formatter** is typically a utility or a specialized component rather than a "Service" (which usually implies business logic, database transactions, or orchestrating multiple steps).

To follow **Clean Architecture** or **Domain-Driven Design (DDD)** principles, you should separate your pure data from your presentation logic. Here is a structure that feels more natural:

Recommended Package Structure

```
com.toolstore.demo
├── model           // Pure Domain Data (POJOs)
│   └── RentalAgreement.java
├── view            // Presentation / Formatting Logic
│   └── RentalAgreementFormatter.java
│       └── FormattedRentalAgreement.java (The "Display Wrapper")
├── service          // Business Logic / Orchestration
│   └── CheckoutService.java
└── util             // (Optional) Generic helpers
```

Why this structure?

1. com.toolstore.demo.model

This remains the home for your **Domain Entities**. These classes should be "clean"—they represent the state of your business. They don't care if they are being printed to a console, saved to a database, or sent over a REST API.

2. com.toolstore.demo.view

This is the most appropriate place for the **Formatter**. In the Model-View-Controller (MVC) pattern, the "View" is responsible for taking data and making it presentable to a human.

- **RentalAgreementFormatter**: Contains the logic for currency, dates, and labels.
- **FormattedRentalAgreement**: This is a **View Model** (or Decorator). It exists specifically to satisfy your requirement of calling `System.out.println(agreement)` while keeping the logic out of the model package.

3. com.toolstore.demo.service

The CheckoutService lives here. It uses the model to perform calculations and then uses the view components to wrap the result before handing it back to the caller.



Implementation Detail

By placing the "Display Wrapper" in the view package, you clearly signal that its only purpose is presentation.

The Wrapper (view package):

```
package com.toolstore.demo.view;

import com.toolstore.demo.model.RentalAgreement;

public class FormattedRentalAgreement {
```

```

private final RentalAgreement agreement;
private final RentalAgreementFormatter formatter;

public FormattedRentalAgreement(RentalAgreement agreement,
RentalAgreementFormatter formatter) {
    this.agreement = agreement;
    this.formatter = formatter;
}

@Override
public String toString() {
    return formatter.format(agreement);
}

// You can also expose the raw agreement if needed
public RentalAgreement getRawAgreement() {
    return agreement;
}
}

```

The Service Usage:

```

@Service
public class CheckoutService {
    @Autowired private RentalAgreementFormatter formatter;

    public FormattedRentalAgreement checkout(...) {
        // 1. Calculate data
        RentalAgreement agreement = RentalAgreement.builder()...build();

        // 2. Return the "Viewable" version
        return new FormattedRentalAgreement(agreement, formatter);
    }
}

```



Benefits of this Package Design:

- **Logical Grouping:** If you ever add a PdfRentalAgreementFormatter, it goes in the view package.
- **Dependency Flow:** view depends on model, but model depends on **nothing**. This is the gold standard for maintainable Java code.
- **Semantic Clarity:** A developer looking at the view package immediately knows that code there is about "how things look," not "how things work."

I have an idea, can I forgo the "formattedrentalagreement" class and make the method format in rental agreement formatter static and call that ?

RouteLLM

Yes, absolutely! If RentalAgreementFormatter has no instance state, making format() static is a clean and valid approach. This simplifies the design significantly.

```
package com.toolstore.demo.view;

public class RentalAgreementFormatter {
    private static final DateTimeFormatter DATE_FMT = DateTimeFormatter.ofPattern("MM/
dd/yy");
    private static final NumberFormat CURRENCY_FMT =
NumberFormat.getCurrencyInstance(Locale.US);

    // Prevent instantiation - this is a pure utility class
    private RentalAgreementFormatter() {}

    public static String format(RentalAgreement a) {
        return String.join("\n",
            "Tool code: " + a.getToolCode(),
            ...
        );
    }
}
```

Then usage becomes:

```
System.out.println(RentalAgreementFormatter.format(agreement));
```

⚠ One Trade-off to Consider

Since you are using **Spring**, making it static means:

- ✗ You **cannot** @Autowire or inject it
- ✗ It becomes harder to **mock** in unit tests (e.g. with Mockito)
- ✓ But for a **pure stateless formatter** with no dependencies, this is rarely a problem

If you are not testing the formatter itself in isolation, this is a perfectly clean solution. It also naturally fits in a util package rather than view:

```
com.toolstore.demo
└── model/RentalAgreement.java
└── service/CheckoutService.java
└── util/RentalAgreementFormatter.java ← static utility
```



Verdict

For a **stateless, dependency-free formatter**, a static utility class is idiomatic Java and arguably cleaner than injecting a Spring bean just to call one method. Go for it.
