

Java Class 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 applicable.

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

import lombok.Getter;
import lombok.Setter;

@Setter
@Getter
public class Tool {

    private String code;
    private ToolType toolType;
    private String brand;

    public Tool(String code, ToolType toolType, String brand) {
        this.code = code;
        this.toolType = toolType;
        this.brand = brand;
    }
}
```

RouteLLM

Responsibility Inference

The Tool class is a simple **data model** whose responsibility is to represent a tool with:

- a code
- a ToolType
- a brand

This follows the **Single Responsibility Principle (SRP)** well: it does *only one thing* —model data.

SOLID Review

Here's a quick evaluation:

- **SRP (Single Responsibility):** ✓ Good. It stores data only.
- **OCP (Open/Closed):** ✓ Fine. Adding new fields is straightforward. No logic to extend.
- **LSP (Liskov Substitution):** ✓ Nothing to violate.
- **ISP (Interface Segregation):** ✓ Not relevant; no interfaces.
- **DIP (Dependency Inversion):** ✓ No dependencies to invert.

Potential Improvements (Optional)

These are *not required*, but you may consider them if they align with your design goals:

1. Make the class immutable

If tools should not change after creation:

```
@Getter
public class Tool {
    private final String code;
    private final ToolType toolType;
    private final String brand;

    public Tool(String code, ToolType toolType, String brand) {
        this.code = code;
        this.toolType = toolType;
        this.brand = brand;
    }
}
```

Immutability improves thread-safety and reduces side effects.

2. Add basic validation (optional)

Only if your domain requires it:

```
public Tool(String code, ToolType toolType, String brand) {
    if (code == null || code.isBlank()) throw new IllegalArgumentException("Tool code
required");
    this.code = code;
    this.toolType = toolType;
    this.brand = brand;
}
```

Final Assessment

Your class already complies with SOLID as a clean POJO. Improvements are optional and depend on your project's design needs.

What about this as an improvement?

```
```public record Tool(String code, ToolType toolType, String brand) {
}```
```

---

## RouteLLM

Great suggestion! Using a **Java Record** is an excellent improvement for this class. Here's why it aligns well:

### Why a Record Works Here

- **Immutability by default** — all fields are final, no setters generated
- **Removes boilerplate** — no need for Lombok `@Getter/@Setter`, constructor, `equals()`, `hashCode()`, or `toString()`
- **Clearer intent** — a record signals to other developers *"this is pure data"*, which perfectly matches the class's responsibility
- **SRP reinforced** — the record syntax makes it obvious this class has one job: hold tool data

### Things to Consider

- **No setters** — if any part of your codebase mutates a Tool after creation (e.g., `tool.setBrand(...)`), you'd need to refactor those call sites
- **JPA/Hibernate** — if Tool is a JPA `@Entity`, records are **not compatible** since JPA requires a no-arg constructor and mutable fields

- **Frameworks like Jackson** — generally work fine with records in modern versions (Jackson 2.12+)

## Verdict

If Tool is **not a JPA entity** and mutability isn't needed, the record version is a clean, modern improvement. Let me know if you'd like to check the spec file to confirm how Tool is used!

---

directly relevant is this enum

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

```
import lombok.Getter;
```

```
import java.math.BigDecimal;
```

```
@Getter
```

```
public enum ToolType {
```

```
    LADDER(new BigDecimal("1.99"), true, true, false),
```

```
    CHAINSAW(new BigDecimal("1.49"), true, false, true),
```

```
    JACKHAMMER(new BigDecimal("2.99"), true, false, false);
```

```
    private final BigDecimal dailyCharge;
```

```
    private final boolean weekdayCharge;
```

```
    private final boolean weekendCharge;
```

```
    private final boolean holidayCharge;
```

```
    ToolType(BigDecimal dailyCharge, boolean weekdayCharge, boolean weekendCharge, boolean holidayCharge) {
```

```
        this.dailyCharge = dailyCharge;
```

```
        this.weekdayCharge = weekdayCharge;
```

```
        this.weekendCharge = weekendCharge;
```

```
        this.holidayCharge = holidayCharge;
```

```
    }
```

```
}```
```




RouteLLM

Let me first check the spec file for context before reviewing.

Review of ToolType Enum

The enum is well-structured and its responsibility is clear: **define tool types with their pricing rules**. It aligns with the spec's pricing table perfectly.

SOLID Assessment

- **SRP:**  Holds pricing/charge rules per tool type — one responsibility
- **OCP:**  Adding a new tool type (e.g., DRILL) requires no changes to existing code
- **Others:**  Not applicable here

Potential Improvements

1. Rename booleans for clarity (minor)

The boolean field names are already readable, but you could make them more expressive:

```
private final boolean chargedOnWeekdays;  
private final boolean chargedOnWeekends;  
private final boolean chargedOnHolidays;
```

2. Add a `isChargeableOn(DayType day)` method (*worth considering*)

Instead of exposing raw booleans and letting callers decide the logic, encapsulate it:

```
public boolean isChargeableOn(boolean isWeekday, boolean isWeekend, boolean isHoliday)  
{  
    if (isHoliday) return holidayCharge;  
    if (isWeekend) return weekendCharge;  
    return weekdayCharge;  
}
```

This keeps charge-day logic **inside the enum** where it belongs, rather than scattered in a service — a stronger SRP/OCP application.

3. Drop Lombok @Getter

Since the fields are already final and the enum is simple, you could write explicit getters or use the `isChargeableOn` approach above and remove the Lombok dependency here entirely.

Key Takeaway

The most impactful improvement is **#2** — the spec requires calculating *chargeable days* based on weekday/weekend/holiday rules, and that logic naturally belongs in `ToolType` rather than a service class.
