Case study on the Open-Closed Principle (OCP) banking domain

case study on the Open-Closed Principle (OCP) tailored to the **banking domain**, complete with explanation, full code, and important discussion/interview questions.

✅ Case Study Title

“Extending Fund Transfer Rules Without Modifying Existing Code: Applying OCP in a Banking Domain”

🧠 Background

In a real-world **banking fund transfer system**, we often deal with various types of **transfer rules** based on:

- Account type (e.g., savings, corporate)
- Transfer mode (e.g., NEFT, RTGS, IMPS)
- Regulatory rules (e.g., daily limit, KYC, RBI compliance)

Initially, developers might put all this logic inside a **single** `FundTransferService` class using multiple `if-else` or `switch` blocks. This tightly couples the service to **changing business logic**, violating the **Open-Closed Principle**.

❌ Problem: Poor OCP Design

```
public class FundTransferService {
    public boolean validateTransfer(Account from, Account to, double amount,
String type) {
        if ("IMPS".equalsIgnoreCase(type)) {
            return amount <= 50000;
        } else if ("NEFT".equalsIgnoreCase(type)) {
            return amount <= 200000;
        } else if ("RTGS".equalsIgnoreCase(type)) {
            return amount >= 200000;
        }
        return false;
    }
}
```

- Every time a new rule is added or modified, **this class must change**, breaking OCP.
- **Unit testing** becomes harder.
- Not suitable for **pluggable rules** in a growing business environment.

✅ Refactored Design: OCP Compliant

Open for **extension** (new rule classes), Closed for **modification** (no change to service layer).

💡 Design Intent

- Introduce an **interface** `TransferRule`.
- Implement **concrete rules** (IMPSRule, NEFTRule, RTGSRule).
- Use a **factory** or **strategy** pattern to select rule dynamically.

📐 Class Diagram (Text)

```
TransferRule (interface)
    └── IMPSRule
    └── NEFTRule
    └── RTGSRule

FundTransferService
    └── validate(TransferRule rule)

TransferRuleFactory (optional)
    └── returns rule instance based on type
```

✅ Step-by-Step Refactored Code

1. TransferRule Interface

```
public interface TransferRule {
    boolean isValid(double amount);
}
```

---

2. Concrete Rule Implementations

```
public class IMPSRule implements TransferRule {
    public boolean isValid(double amount) {
        return amount <= 50000;
    }
}

public class NEFTRule implements TransferRule {
    public boolean isValid(double amount) {
        return amount <= 200000;
    }
}

public class RTGSRule implements TransferRule {
    public boolean isValid(double amount) {
        return amount >= 200000;
    }
}
```

3. Rule Factory (optional)

```
public class TransferRuleFactory {
    public static TransferRule getRule(String type) {
        switch (type.toUpperCase()) {
            case "IMPS": return new IMPSRule();
            case "NEFT": return new NEFTRule();
            case "RTGS": return new RTGSRule();
            default: throw new IllegalArgumentException("Unsupported type: " +
type);
        }
    }
}
```

4. OCP-Compliant FundTransferService

```
public class FundTransferService {
    public boolean validate(double amount, TransferRule rule) {
        return rule.isValid(amount);
    }
}
```

5. Client Code (App)

```
public class BankingApp {
    public static void main(String[] args) {
        double amount = 75000;
        String type = "NEFT";

        TransferRule rule = TransferRuleFactory.getRule(type);
        FundTransferService service = new FundTransferService();

        boolean result = service.validate(amount, rule);
        System.out.println("Is Transfer Valid? " + result);
    }
}
```

🧪 Unit Testing Advantages

You can now unit test each rule class in isolation:

```
@Test
public void testNEFTRule() {
    TransferRule rule = new NEFTRule();
    assertTrue(rule.isValid(100000));
    assertFalse(rule.isValid(300000));
}
```

💬 Discussion Questions

1. What is the Open-Closed Principle? How does this design follow it?

2. How would you extend this system for international transfers?
3. What pattern can be used if rules become too many and need configuration-driven loading?
4. Could we use Spring’s `@Component` + `@Qualifier` to inject rules dynamically?
5. Can this approach support validation chaining (e.g., KYC check + amount check)?

🏢 Industry Relevance

- Payment gateways and banking microservices frequently need rule-driven engines.
- This design is used in **rule engines**, **fraud detection**, **transaction validation**, and **compliance checking**.
- Helps build **modular**, **testable**, and **pluggable** services.

🔧 Real-World Extensions (Optional)

- Add rule registration via **Spring Boot Configuration + Map<String, Rule>**.
- Support **runtime rule creation via DSL or YAML**.
- Combine with **Strategy + Chain of Responsibility** for rule pipelines.

Use of `switch` in `TransferRuleFactory` does violate the Open-Closed Principle (OCP).

Let’s examine why and how to fix it **professionally** using a better OCP-compliant approach in the **banking context**.

❌ Why This Is a Violation of OCP

```
public class TransferRuleFactory {
    public static TransferRule getRule(String type) {
        switch (type.toUpperCase()) {
            case "IMPS": return new IMPSRule();
            case "NEFT": return new NEFTRule();
            case "RTGS": return new RTGSRule();
            default: throw new IllegalArgumentException("Unsupported type: " + type);
        }
    }
}
```

- The factory must **change every time** a new transfer type (like "UPI", "SWIFT", etc.) is added.
- Violates **“Closed for modification”**.

- Not truly **extensible** without modifying existing logic.

✅ OCP-Compliant Fix Using Spring Boot + `@Component` + Map Injection

This makes it **pluggable**, dynamic, and truly **Open for Extension**.

🔁 Step-by-Step Refactoring Using Spring

---

1. Create Interface

```
public interface TransferRule {
    boolean isValid(double amount);
    String getType(); // e.g., IMPS, NEFT, etc.
}
```

---

2. Concrete Rules with `@Component`

```
@Component
public class IMPSRule implements TransferRule {
    public boolean isValid(double amount) { return amount <= 50000; }
    public String getType() { return "IMPS"; }
}

@Component
public class NEFTRule implements TransferRule {
    public boolean isValid(double amount) { return amount <= 200000; }
    public String getType() { return "NEFT"; }
}

@Component
public class RTGSRule implements TransferRule {
    public boolean isValid(double amount) { return amount >= 200000; }
    public String getType() { return "RTGS"; }
}
```

---

3. Registry Class with Autowired Map

```
@Component
public class TransferRuleRegistry {

    private final Map<String, TransferRule> ruleMap = new HashMap<>();

    @Autowired
    public TransferRuleRegistry(List<TransferRule> rules) {
        for (TransferRule rule : rules) {
            ruleMap.put(rule.getType().toUpperCase(), rule);
        }
    }
    public TransferRule getRule(String type) {
        TransferRule rule = ruleMap.get(type.toUpperCase());
        if (rule == null) {
            throw new IllegalArgumentException("No rule found for type: " +
type);
        }
        return rule;
    }
}
```

4. Service Class

```
public class FundTransferService {

    private final TransferRuleRegistry registry;

    public FundTransferService(TransferRuleRegistry registry) {
        this.registry = registry;
    }

    public boolean validate(String type, double amount) {
        TransferRule rule = registry.getRule(type);
        return rule.isValid(amount);
    }
}
```

5. REST Controller (for testing)

```
@RestController
@RequestMapping("/transfer")
public class TransferController {

    private final FundTransferService service;

    public TransferController(FundTransferService service) {
        this.service = service;
    }

    @GetMapping("/validate")
    public ResponseEntity<String> validate(@RequestParam String type,
                                           @RequestParam double amount) {
        boolean result = service.validate(type, amount);
        return ResponseEntity.ok("Transfer Valid: " + result);
    }
}
```

✅ OCP Compliance Achieved

- To **add a new transfer type**, just create a new `@Component` implementing `TransferRule`.
- **No existing code** in registry, service, or controller changes.
- Rule discovery is now **automatic** and **dynamic** via Spring context.

- This also opens up possibilities for **conditional bean loading**, profiles, and plugin-based rules.

📌 Summary

Aspect	Before (switch-case)	After (OCP-compliant)
Extensibility	❌ Requires code change	✅ Add new class only
OCP Compliance	❌ Violated	✅ Maintained
Pluggability	❌ Static binding	✅ Dynamic via Spring DI
Real-world Applicability	❌ Hard to maintain	✅ Professional-grade extensibility