Facade Pattern - Descriere Generală: Facade Pattern este un design pattern structural care oferă o interfață simplificată către un subsistem complex de clase, biblioteci sau frameworks. Acesta acționează ca un "punct unic de intrare" către funcționalitățile sistemului, ascunzând complexitatea implementării și reducând dependențele între client și subsistem.

Principalele caracteristici ale Facade Pattern sunt:

1. Simplificare - Oferă o interfață mai simplă și mai ușor de utilizat
2. Decuplare - Reduce dependențele dintre client și subsistem
3. Abstractizare - Ascunde complexitatea implementării
4. Unificare - Oferă un punct unic de acces către funcționalități diverse

```
@RestController
@RequestMapping("/api/accountant")
public class AccountantController {

    @Autowired
    private ExpenseService expenseService;

    @Autowired
    private ExpenseReportService reportService;

    @PostMapping("/reports")
    @PreAuthorize("hasRole('ACCOUNTANT')")
    public ResponseEntity<Map<String, Object>> generateReport(@RequestBody Map<String, String> reportParams) {
        // ...
    }
}
```

1.Interfață Simplificată:

- Controllerul oferă endpoint-uri REST simple precum /api/accountant/reports
- Clientul nu trebuie să știe nimic despre logica complexă din spate
- Parametrii sunt primiți într-un format simplu (Map<String, String>)

2.Ascunderea Complexității:

```
@PostMapping("/reports")
public ResponseEntity<Map<String, Object>> generateReport(...) {
    // Controllerul ascunde complexitatea:
    // - Parsarea datelor
    // - Calculul statisticilor
    // - Generarea rapoartelor
    // - Manipularea bazei de date
}
```

3.Orchestrarea Subsistemelor: Controllerul coordonează mai multe servicii:

```
@Autowired
private ExpenseService expenseService;

@Autowired
private ExpenseReportService reportService;
```

4. Gestionarea Logicii de Business:

```
// În metoda generateReport:
List<Expense> expenses = expenseService.getExpensesByDateRange(startDate, endDate);
Map<String, Object> reportData = new HashMap<>();
```

```
// Calcule complexe și procesare de date
BigDecimal totalAmount = expenses.stream()
        .map(Expense::getAmount)
        .reduce(BigDecimal.ZERO, BigDecimal::add);
```

5.Mapare și Transformare Date:

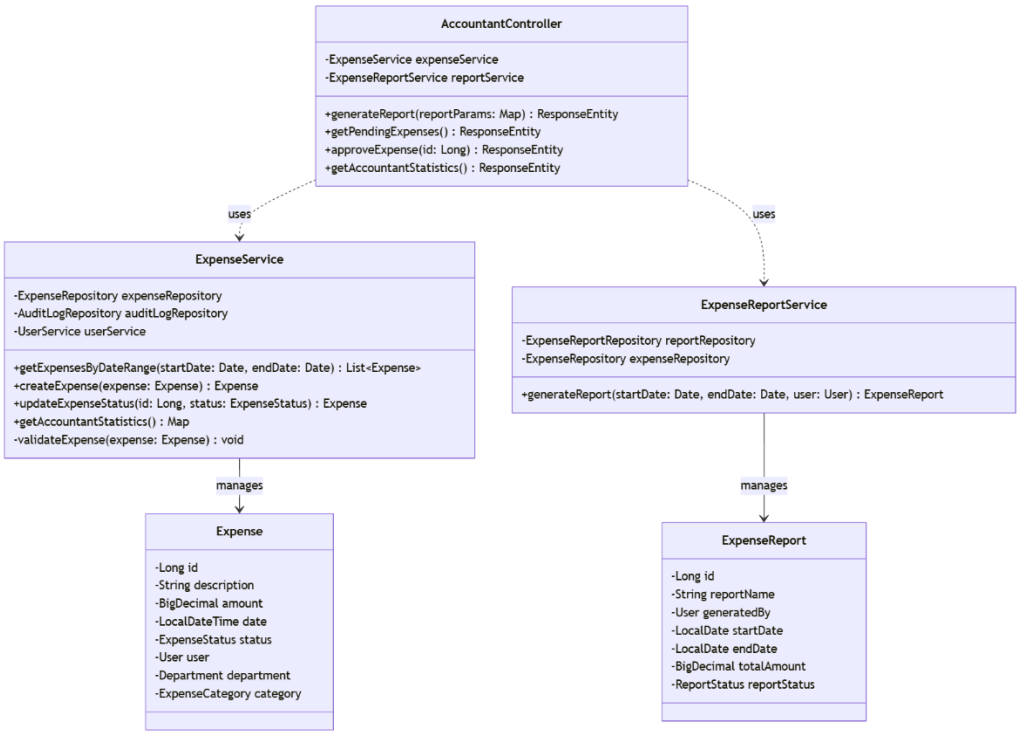
```
switch (reportType) {
    case "byCategory":
        Map<String, BigDecimal> categoryTotals = new HashMap<>();
        for (Expense expense : expenses) {
            String categoryName = expense.getCategory() != null ?
                    expense.getCategory().getName() : "Uncategorized";
            categoryTotals.merge(categoryName, expense.getAmount(), BigDecimal::add);
        }
        // ... more processing
}
```

Beneficiile Implementării:

1. **Simplificare pentru Client:**
    - Clientul (frontend) trebuie doar să facă un apel API simplu
    - Nu trebuie să știe despre serviciile din spate sau logica de business
2. **Securitate:**
    - Controllerul gestionează autorizarea (@PreAuthorize)
    - Validează datele de intrare
    - Gestionează erorile și returnează răspunsuri corespunzătoare
3. **Maintainability:**
    - Logica complexă este împărțită între servicii
    - Controllerul doar orchestrează și coordonează
    - Modificările în servicii nu afectează interfața API
4. **Flexibilitate:**
    - Poți modifica implementarea serviciilor fără a afecta clientul
    - Poți adăuga noi funcționalități păstrând aceeași interfață

Această implementare este un exemplu excelent de Facade Pattern deoarece:

- Ascunde complexitatea sistemului backend
- Oferă o interfață REST clară și simplă
- Gestionează toate dependențele și interacțiunile complexe
- Permite clientului să utilizeze sistemul fără a cunoaște detaliile implementării



Explicație a diagramei UML:

1. **Facade (AccountantController)**:
    - Reprezintă interfața simplificată pentru client
    - Ascunde complexitatea sistemului prin expunerea doar a metodelor necesare
    - Coordonează interacțiunile între diferite subsisteme
2. **Subsisteme**:
    - ExpenseService: Gestionează operațiile legate de cheltuieli
    - ExpenseReportService: Gestionează generarea rapoartelor
    - Expense: Entitate care reprezintă o cheltuială
    - ExpenseReport: Entitate care reprezintă un raport
3. **Relații**:
    - AccountantController folosește (uses) ExpenseService și ExpenseReportService
    - ExpenseService gestionează (manages) entitățile Expense
    - ExpenseReportService gestionează (manages) entitățile ExpenseReport

4. **Design Pattern Facade ilustrat prin**:
    - AccountantController acționează ca fațadă, oferind o interfață simplificată
    - Subsistemele complexe (serviciile și entitățile) sunt ascunse în spatele fațadei
    - Clientul interacționează doar cu AccountantController, fără a cunoaște complexitatea din spate

Această diagramă UML ilustrează clar cum Facade Pattern:

- Simplifică interfața pentru client
- Ascunde complexitatea implementării
- Reduce cuplarea între client și subsisteme
- Oferă un punct unic de acces către funcționalitățile sistemului

Exemplu de Utilizare

Request pentru Generare Raport:

POST /api/accountant/reports

Content-Type: application/json


```
{
  "startDate": "2024-01-01",
  "endDate": "2024-01-31",
  "reportType": "byCategory"
}
```


Response:


```
{
  "totalAmount": 15000.00,
  "transactionCount": 25,
  "averageAmount": 600.00,
```

```
  "chartData": [
    {
      "name": "Travel",
      "amount": 5000.00
    },
    {
      "name": "Office Supplies",
      "amount": 10000.00
    }
  ]
}
```