Implementation of Decorator and Strategy Design Patterns- Ester Shumeli

Structural Design Pattern – Decorator

Inside the `Report` module, the **Decorator Design Pattern** is applied to enhance the functionality of the ReportExporter interface without modifying its existing implementation.

- The `Report` class provides the base content of the report. To support flexible and extensible export formats, additional export features are layered dynamically using decorators.
- An abstract class ReportDecorator implements ReportExporter and serves as the base class for all decorators.
- Multiple concrete decorators such as PDFExportReport, TimestampedReport, EncryptedReport, and WatermarkedReport extend ReportDecorator, each modifying or enriching the export behavior of the wrapped report object.
- This pattern allows export features to be **composed at runtime** without changing the core logic of the `Report` class.
- **Open/Closed Principle** — new export formats or enhancements can be introduced simply by creating new decorator classes, without modifying existing code.

Behavioral Design Pattern – Template Method

In the **Furniture Management System**, the report generation process initially varied across different report types, leading to duplicated code and inconsistent report structures. For example, FinanceReport, HRReport, and InventoryReport each had their own way of collecting data and formatting output, which made the system harder to maintain and scale.

To address this, I implemented the **Template Method Pattern** by introducing an abstract class called `Report`. This class defines a fixed method called generateReport`()` that outlines the overall report workflow: fetchData(), formatReport(), addTimestamp(), and storeReport`()`. The first two methods are abstract, allowing each subclass to provide its own specific logic, while the remaining steps are handled by the base class to ensure consistency and eliminate redundancy.

This design guarantees that all reports follow the same structure while still allowing flexibility in how data is gathered and displayed. For instance:

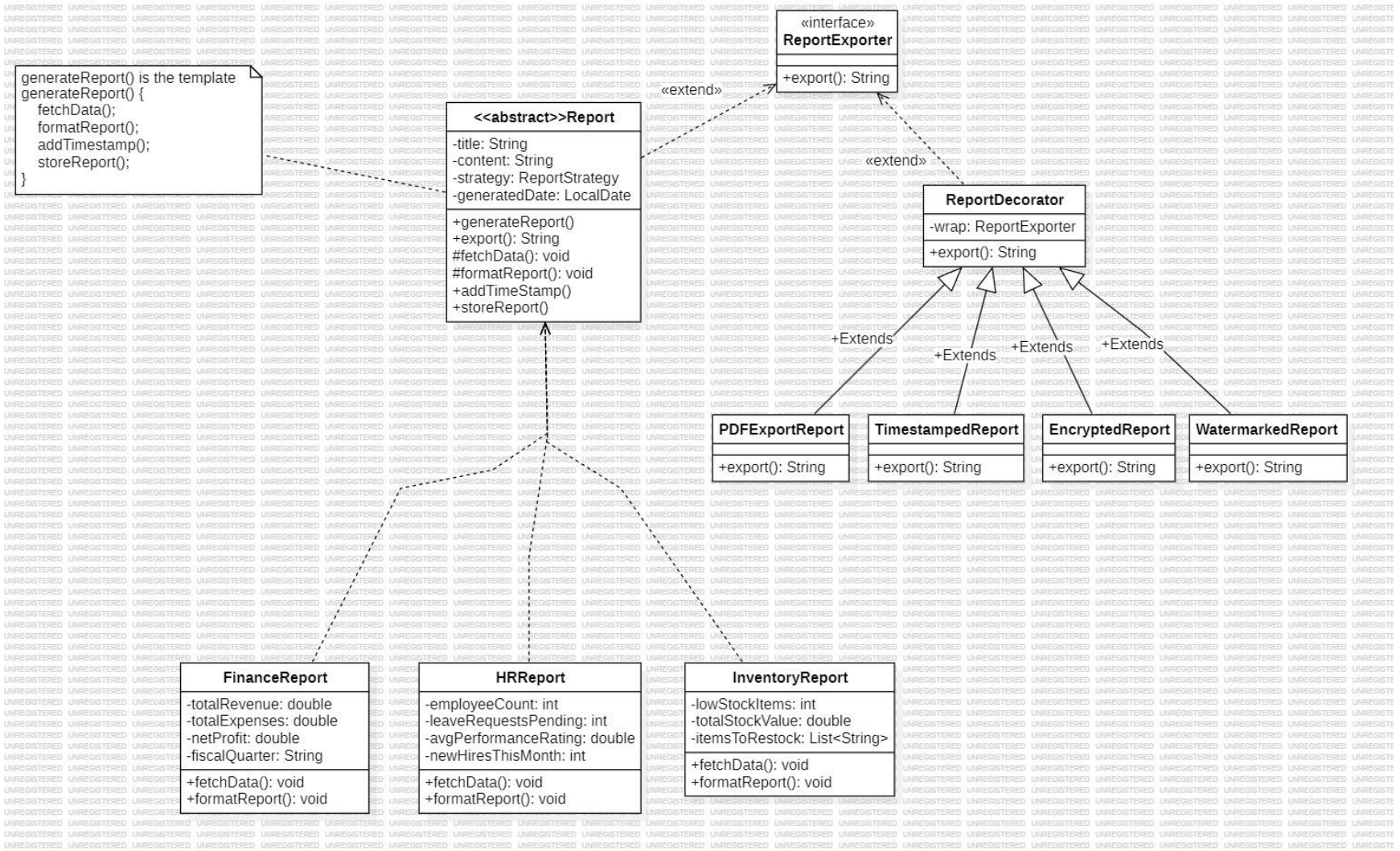
- FinanceReport fetches budget figures and calculates net profit.
- HRReport collects employee and hiring data.

- InventoryReport retrieves stock levels and restocking needs.

Despite their differences, all of these reports share the same report-generation template. The shared steps like addTimestamp() and storeReport() are implemented once in the base class, reducing duplication and improving maintainability.

By applying this pattern, the system becomes ***more scalable, modular,*** and ***easier to extend.***

Open/Closed Principle, as new report types can be added without modifying the existing report generation logic.


generateReport() is the template
generateReport() {
fetchData();
formatReport();
addTimestamp();
storeReport();
}
«interface»
ReportExporter
+export(): String
«extend»
<<abstract>>Report
-title: String
-content: String
-strategy: ReportStrategy
-generatedDate: LocalDate
+generateReport()
+export(): String
#fetchData(): void
#formatReport(): void
+addTimeStamp()
+storeReport()
«extend»
ReportDecorator
-wrap: ReportExporter
+export(): String
+Extends
+Extends
+Extends
+Extends
PDFExportReport
+export(): String
TimestampedReport
+export(): String
EncryptedReport
+export(): String
WatermarkedReport
+export(): String
FinanceReport
-totalRevenue: double
-totalExpenses: double
-netProfit: double
-fiscalQuarter: String
+fetchData(): void
+formatReport(): void
HRReport
-employeeCount: int
-leaveRequestsPending: int
-avgPerformanceRating: double
-newHiresThisMonth: int
+fetchData(): void
+formatReport(): void
InventoryReport
-lowStockItems: int
-totalStockValue: double
-itemsToRestock: List<String>
+fetchData(): void
+formatReport(): void