1. **Singleton Pattern**

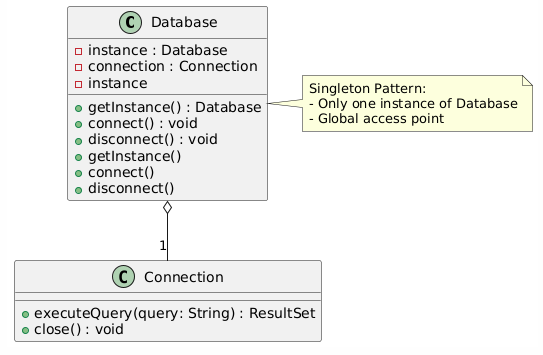
**Where It’s Used:**

In this project, the **Singleton Pattern** is perfect for managing the **Database Connection**.

* You only want **one instance** of the database to be active at any time.
* It controls access, optimizes resource usage, and prevents multiple connections that can mess with data integrity.

**Justification:**

* Ensures **global access** to the database from multiple classes (Account, Transaction, Loan, etc.).
* Prevents **duplicate connections** that might cause data inconsistency.
* Guarantees **thread safety** when multiple users interact with the system simultaneously.



**Explanation:**

1. Database class is Singleton → Only one instance can exist.
2. getInstance() is a static method to access that instance.
3. Methods like connect() and disconnect() manage the database connection.
4. Other classes interact with this single instance when they need to query or update the database.
5. **Factory Method Pattern**

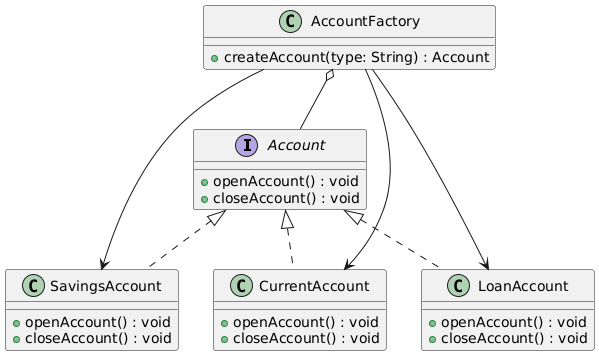
**Where It’s Used:**

In our project, the **Factory Method Pattern** is perfect for **Account Creation**.

* You have different types of accounts like **SavingsAccount**, **CurrentAccount**, and **LoanAccount**.
* The **Factory Method** allows you to dynamically create the correct account type based on user choice, without modifying existing code.

**Justification:**

* Avoids **tight coupling** between code and specific account types.
* Simplifies **object creation** and maintenance.
* Makes it easy to **add new account types** in the future.



**Explanation:**

1. Account is an interface for different account types.
2. SavingsAccount, CurrentAccount, and LoanAccount implement Account.
3. AccountFactory has the method createAccount() that determines which account to create based on the type parameter.
4. **Observer Pattern**

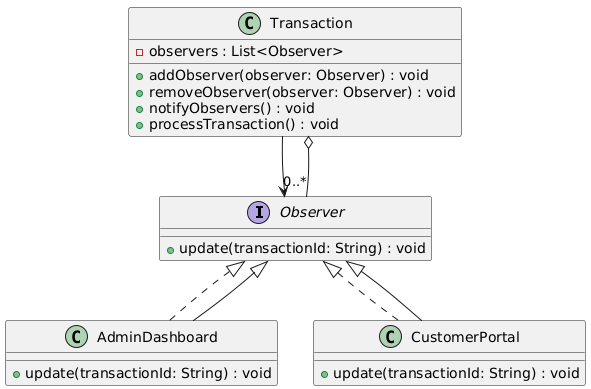
**Where It’s Used:**

In our project, the **Observer Pattern** is perfect for **Transaction Monitoring**.

* When a transaction is created or updated, multiple observers like **Admin Dashboard** or **Customer Portal** should be **automatically notified**.
* This allows real-time updates to display changes instantly.

**Justification:**

* Achieves **real-time updates** without manual refresh.
* Keeps **Admin** and **Customer** in sync with the latest transaction status.
* Decouples the **subject (Transaction)** from its **observers**, making it scalable.



### ****Explanation:****

1. Transaction is the **subject** — it tracks all observers.
2. AdminDashboard and CustomerPortal are **observers.**
3. When a transaction is processed, it **notifies** all registered observers.
4. Both portals **update their view** instantly.

## ****Facade Pattern****

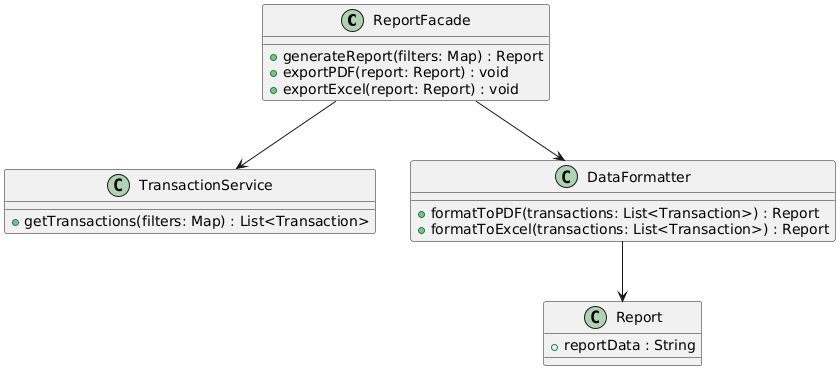
### ****Where It’s Used:****

The **Facade Pattern** is perfect for **Report Generation** in our project.

* Report generation can involve **complex processes** like querying transactions, formatting data, and exporting to PDF/Excel.
* The **Facade** provides a **simple interface** to handle all of that complexity with just one call.

### ****Justification:****

* Simplifies access to complex reporting logic.
* Hides the inner workings of data fetching, formatting, and exporting.
* Reduces the learning curve for other developers who need to generate reports.



### ****Explanation:****

1. ReportFacade provides a simple API for generating and exporting reports.
2. It interacts with:
   * TransactionService to **fetch data**
   * DataFormatter to **format the report**
3. It hides all complex steps and makes it **easy to call** from one place.

## ****Decorator Pattern****

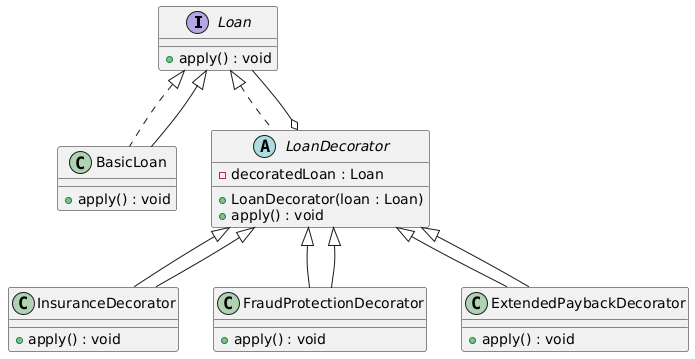
### ****Where It’s Used:****

The **Decorator Pattern** is perfect for **Loan Applications** in our project.

* When customers apply for a loan, they might want **extra features** like:
  + Insurance Coverage
  + Fraud Protection
  + Extended Payback Options
* Instead of creating multiple subclasses for every combination, we can **dynamically add features** to a basic loan object using decorators.

### ****Justification:****

* Allows you to **extend the functionality** of a loan application without modifying its core logic.
* Supports **dynamic combinations** of add-ons (e.g., a loan with both insurance and fraud protection).
* Keeps the design **flexible** and **scalable.**

****

### ****Explanation:****

1. Loan is the base interface.
2. BasicLoan is the default implementation.
3. LoanDecorator is an **abstract class** that extends the behavior of Loan.
4. Specific decorators (InsuranceDecorator, FraudProtectionDecorator, ExtendedPaybackDecorator) add functionality.