



**COMSATS UNIVERSITY ISLAMABAD,
ABBOTTABAD**

Design Pattern

Assignment # 02

Submitted by:

Laiba binte tahir FA21-BSE-019

Submitted to:

Mam Faiza Hameed

Contents

Q1: Design patterns in java APIs	3
1. Creational design Patterns	3
2. Structural Design Patterns	3
3. Behavioral Design Pattern	4
Q2: Design Patterns (Intent and Explanation).....	5
1. Front Controller Pattern	6
2. Application Controller Pattern.....	7
3. Dependency Injection Pattern	8
4. Data Mapper Pattern	9
5. Domain Object Factory Pattern	10
6. Adaptive Design Pattern	11
7. Null Object Pattern	12
8. Service Locator Pattern	13

Q1: Design patterns in java APIs

1. Creational design Patterns

1. *java.lang.Runtime and java.lang.Desktop*

- **Singleton Pattern:** Above java APIs follows the singleton design pattern because they provide a single instance of object and that's globally accessible. `GetDesktop ()` and `GetRuntime()` ensure that only one instance of the runtime environment is accessible to all application throughout.

2. *com.google.common.collect.MapMaker*

- **Builder Pattern:** The above class from Google's Guava library implements the Builder pattern to create customized `ConcurrentMap` instances. It allows incremental configuration of features.

3. *java.util.Calendar, java.text.NumberFormat, java.nio.charset.Charset*

- **Factory Pattern:** These provide static factory methods to create instances. `Calendar.getInstance()` returns an appropriate calendar instance. Without exposing the exact implementation. `NumberFormat.getInstance()` returns a locale-specific instance for formatting numbers. `Charset.forName()` returns a `Charset` object for the specified charset name, hiding the instantiation details.

4. *javax.xml.parsers.DocumentBuilderFactory, javax.xml.transform.TransformerFactory, javax.xml.xpath.XPathFactory*

- **Abstract Factory pattern:** Each creates specific objects (`DocumentBuilder`, `Transformer`, `XPath`) that belong to XML processing domain also the implementations are abstracted from the client. Also, can be configured to use specific implementations via configuration files.

2. Structural Design Patterns

1. *java.lang.Integer and java.lang.Boolean*

- **Flyweight pattern:** Wrapper classes like `Integer` and `Boolean` reuse objects `Integer.valueOf()` caching small integers b/w -128 and 127. This reduces memory usage by sharing the instances.

2. *java.io.InputStreamReader, java.io.OutputStreamWriter, java.util.Arrays*

- **Adapter Pattern:** `InputStreamReader` and `OutputStreamWriter` adapt streams to character streams. And `Array.asList()` adapts array into `List` making it usable with collection framework.
- 3. ***java.io.BufferedReader, java.io.DataInputStream, java.io.BufferedOutputStream, java.util.zip.ZipOutputStream, java.util.Collections#checkedList()***
 - **Decorator Pattern:** the above classes wrap existing streams to add new functionality without changing their structure. Buffered, data, and Zip stream or type-checking enhance base functionality without altering the underlying object.
 - Decorators like Buffered, Data, and Zip streams or type-checking for Collections enhance base functionality without altering the underlying object.

3. Behavioral Design Pattern

1. ***javax.servlet.FilterChain***
 - **Chain of Responsibility Pattern:** Filterchain allows multiple filters to process requests sequentially. Each filter passes the request to the next filter in chain until final resource is reached i-e Servlet
2. ***java.lang.Runnable and java.util.concurrent.Callable***
 - **Command Pattern:** Runnable and callable do encapsulation of a task or command in object to be executed later, which promotes decoupling as tasks can be executed by any thread.
3. ***java.util.Iterator***
 - **Iterator Pattern:** The Iterator provides a way to traverse collections without exposing their underlying structure. It encapsulates traversal logic and simplifies access.
4. ***java.util.Comparator and javax.servlet.Filter***
 - **Strategy Pattern:** Comparator allows you to define different comparison strategies for sorting objects. Filter provides a strategy for processing requests or responses in servlets.
5. ***java.util.AbstractList, java.util.AbstractSet, java.util.AbstractMap***
 - **Template Method Pattern:** These abstract classes define a skeleton (template) for specific collection implementations. Subclasses must implement specific methods, but the overall flow remains defined in the abstract class.
6. ***java.io.InputStream, java.io.OutputStream, java.io.Reader, java.io.Writer***
 - **Template Method Pattern:** These I/O classes define an abstract flow for input and output operations, where concrete subclasses implement specific behavior.

7. *java.util.EventListener and java.util.Observer/java.util.Observable*

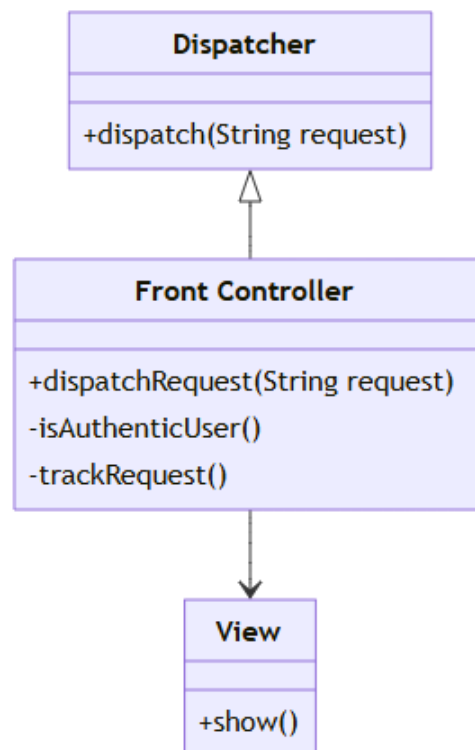
- **Observer Pattern:** EventListener is used to observe and respond to events (like UI events). Observer/Observable follow the observer pattern, where the Observable object notifies all registered observers when its state changes.

Q2: Design Patterns (Intent and Explanation)

All diagrams have been made using tool: <https://www.mermaidchart.com/>

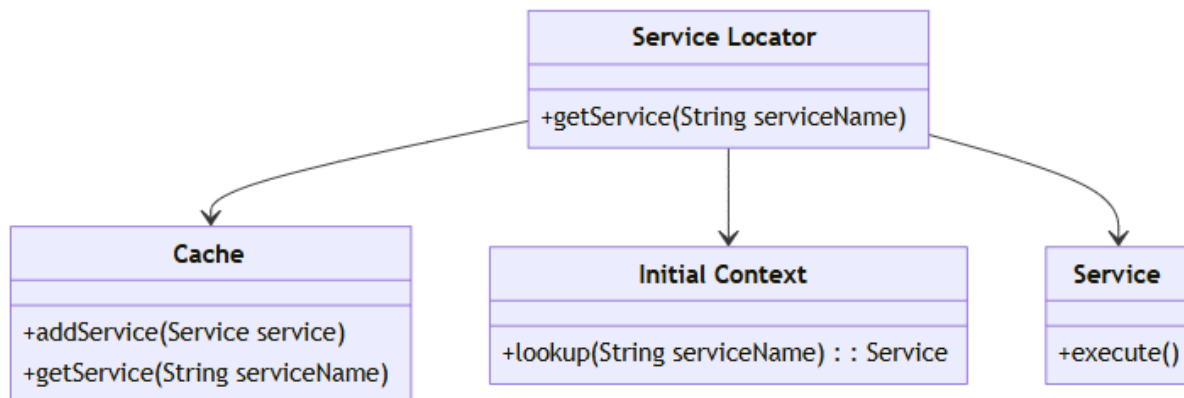
1. Front Controller Pattern

Type	Architectural Design Pattern
Intent	Provides a centralized request handling mechanism to control and dispatch requests.
Problem	Handling multiple requests individually can lead to duplicated code and lack of central control in a web application.
Solution	Centralize the control logic by using a single-entry point (controller) to manage and delegate requests to handlers.
Consequences	Pros: Simplifies request processing and improves maintainability.
	Cons: Can become a bottleneck if not designed carefully.
Structure	FrontController: Receives requests.
	Handlers (or Dispatchers): Process specific requests.
	View: Presents the data.



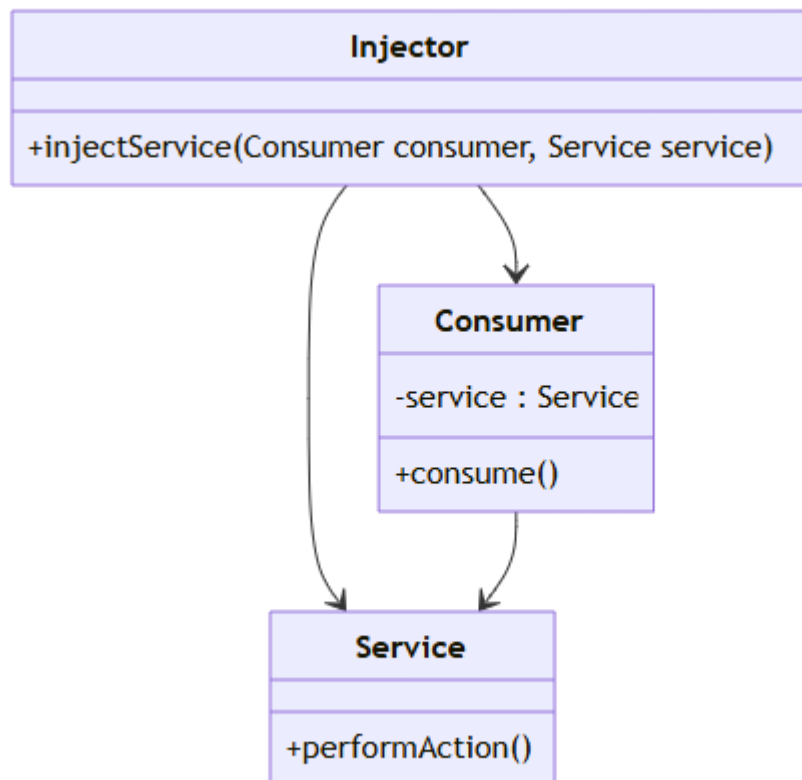
2. Service Locator Pattern

Type	Architectural Design Pattern
Intent	Provide a centralized registry to locate and retrieve services.
Problem	Managing object creation and lifecycle in a distributed system can be complex.
Solution	Use a service locator to abstract service instantiation and access.
Consequences	Pros: Simplifies dependency management and reduces code duplication.
	Cons: Hides dependencies and can lead to runtime errors.
Structure	Service Locator: Retrieves instances from a registry.
	Services: Perform the required functionality.



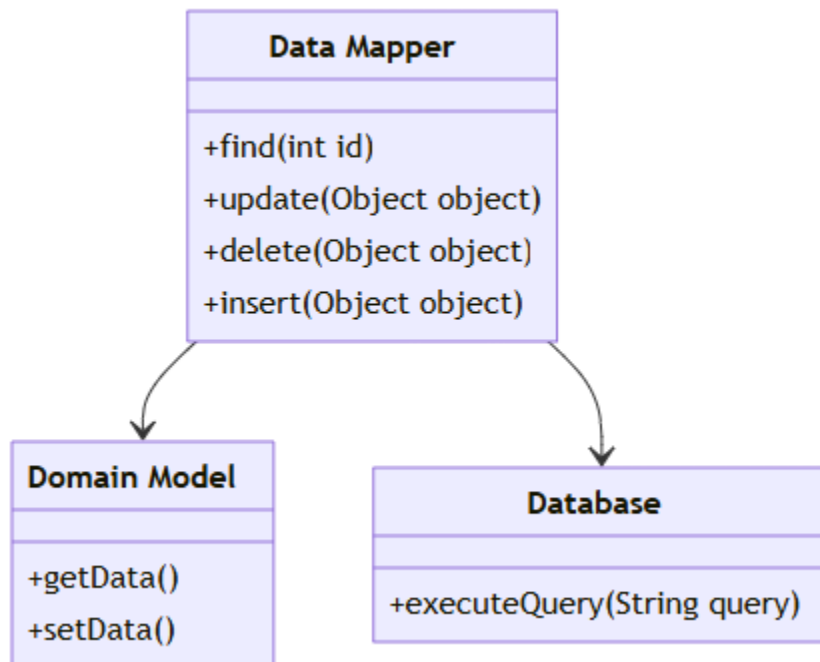
3. Dependency Injection Pattern

Type	Creational Design Pattern
Intent	Decouple object creation from its use, allowing dependencies to be injected at runtime.
Problem	Tight coupling between objects makes unit testing and future modifications difficult.
Solution	Use a container or framework to inject required dependencies into objects.
Consequences	Pros: Increases testability and flexibility.
	Cons: Can lead to overuse of frameworks and complexity in configuration.
Structure	Injectors: Provide the dependencies.
	Dependent Objects: Use those dependencies.



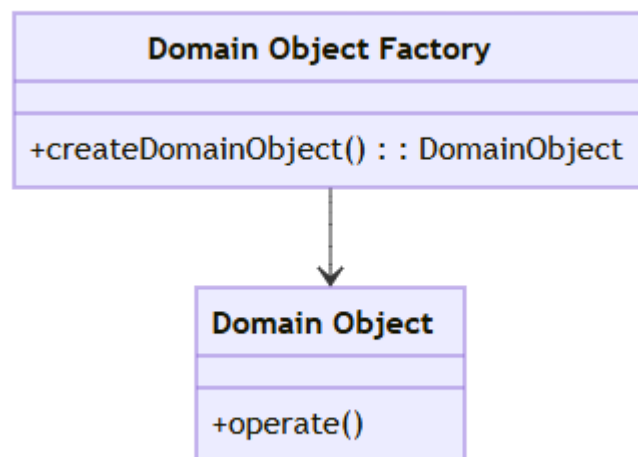
4. Data Mapper Pattern

Type	Architectural Design Pattern
Intent	Abstract the mapping of objects to database tables to decouple business logic from persistence logic.
Problem	Mixing database queries with domain logic results in low cohesion and poor scalability.
Solution	Use a mapper class to handle the object-relational mapping.
Consequences	Pros: Increases separation of concerns and testability.
	Cons: Adds complexity and overhead for mapping configurations.
Structure	Domain Objects: Represent the application state.
	Mapper: Interacts with the database and translates data to/from domain objects.



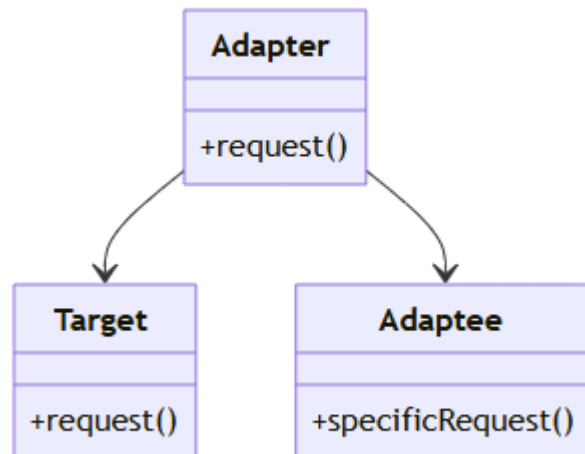
5. Domain Object Factory Pattern

Type	Creational Design Pattern
Intent	Encapsulate the creation logic of domain objects to ensure consistency and abstraction.
Problem	Creating domain objects in multiple places can lead to inconsistent behavior.
Solution	Use a factory to centralize the creation process and ensure encapsulation.
Consequences	Pros: Provides a single place to manage creation logic.
	Cons: Adds complexity if overused for simple objects.
Structure	Factory Class: Handles the creation of domain objects.



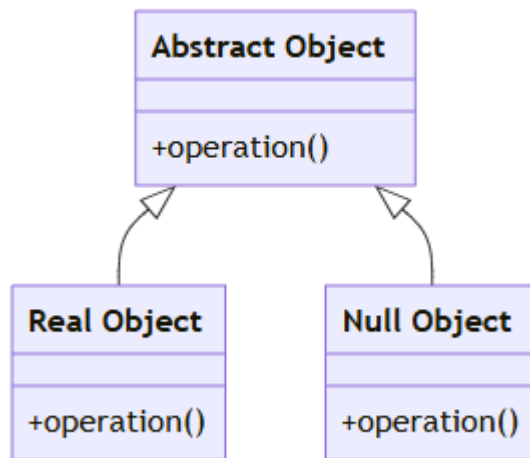
6. Adaptive Design Pattern

Type	Structural Design Pattern
Intent	Convert one interface into another that a client expects.
Problem	Components with incompatible interfaces cannot interact directly.
Solution	Introduce an adapter to bridge the incompatibility.
Consequences	Pros: Allows reusability of existing components.
	Cons: May result in additional layers and performance overhead.
Structure	Adapter: Sits between the client and the service to enable communication.



7. Null Object Pattern

Type	Behavioral Design Pattern
Intent	Provide a default object to eliminate null checks in code.
Problem	Repeated null checks clutter the code and increase complexity.
Solution	Use a default object with neutral behavior as a substitute for null.
Consequences	Pros: Simplifies code and improves readability.
	Cons: Increases the number of classes to maintain.
Structure	Abstract Class: Defines behavior.
	NullObject: Provides the default implementation.



8. Application Controller Pattern

Type	Architectural Design Pattern
Intent	Centralizes and decouples the processing logic from the input-handling logic.
Problem	When logic is dispersed, changes in business rules require multiple updates.
Solution	Use a controller that maps user requests to business logic and delegates responsibilities appropriately.
Consequences	Pros: Promotes reusability and separation of concerns.
	Cons: Adds an additional layer of complexity.
Structure	Controller: Manages requests and delegates them to appropriate services or commands.
	Model and View: Remain separate.

