# Module 6
# Contact Session 01

## Patterns – Part 1

Harvinder S Jabbal
SSZG653 Software Architectures

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

# Patterns

# Outline

What is a Pattern?

Pattern Catalogue

- Module patterns
  - Layered Pattern
- Component and Connector Patterns
  - Broker Pattern
  - MVC Pattern
  - Pipe-and-Filter Pattern
  - Client Server Pattern
  - Peer-to-Peer Pattern
  - SOA Pattern
  - Publish Subscribe
  - Shared Data Pattern
- Allocation Patterns
  - Map-Reduce Pattern
  - Multi-tier Pattern

Relation Between Tactics and Patterns

Using tactics together

Summary

# What is a Pattern?



An architectural pattern establishes a relationship between:

*A context*. A recurring, common situation in the world that gives rise to a problem.

*A problem.* The problem, appropriately generalized, that arises in the given context.

*A solution.* A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:

- A set of element types (for example, data repositories, processes, and objects)
- A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
- A topological layout of the components
- A set of semantic constraints covering topology, element behavior, and interaction mechanisms

# Layer Pattern

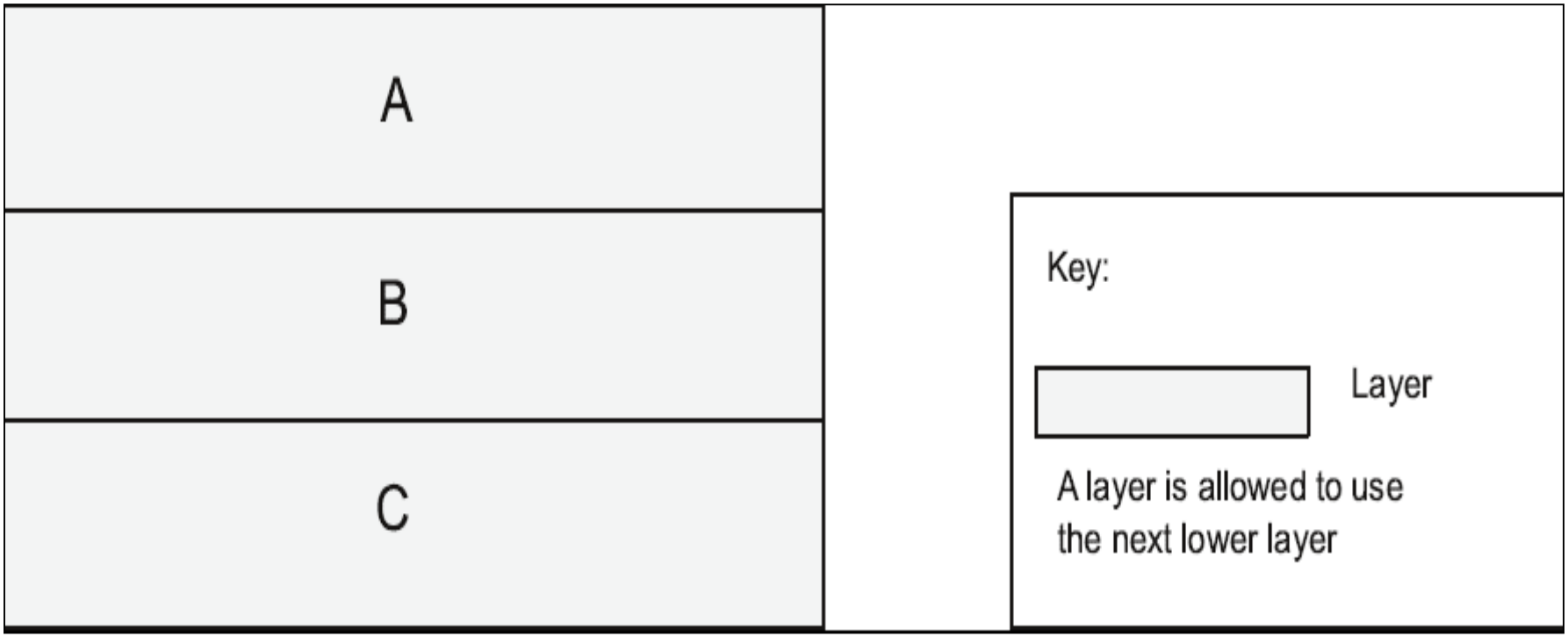
innovate achieve lead

**Context:** All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

**Problem:** The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

**Solution:** To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.

# Layer Pattern Example



# Layer Pattern Solution

Overview: The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.

Elements: *Layer*, a kind of module. The description of a layer should define what modules the layer contains.

Relations: *Allowed to use.* The design should define what the layer usage rules are and any allowable exceptions.

Constraints:

- Every piece of software is allocated to exactly one layer.
- There are at least two layers (but usually there are three or more).
- The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).

Weaknesses:

- The addition of layers adds up-front cost and complexity to a system.
- Layers contribute a performance penalty.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

# Broker Pattern

# Broker Pattern

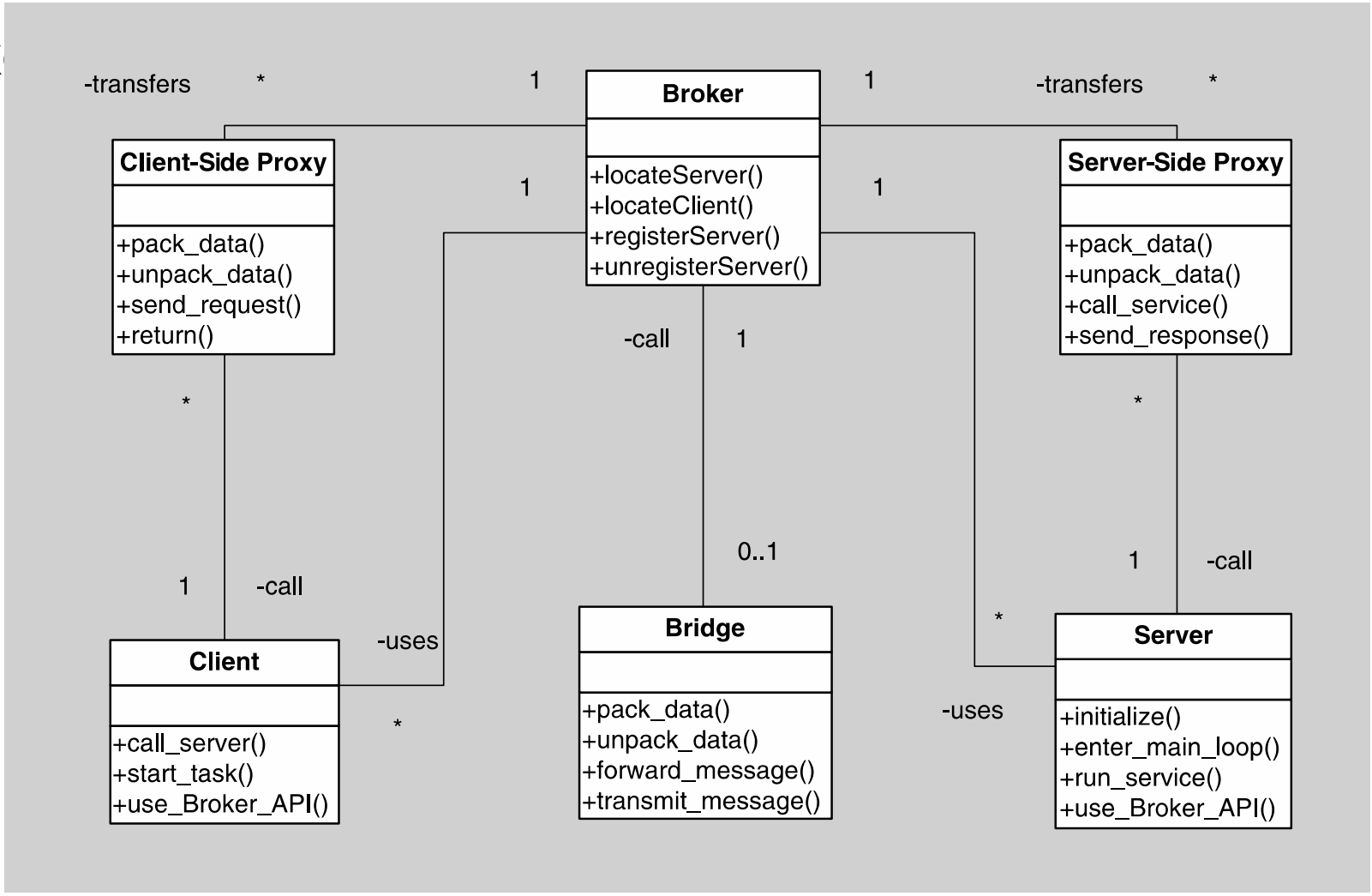


**Context**: Many systems are constructed from a collection of services distributed across multiple servers. Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

**Problem**: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

**Solution**: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker. When a client needs a service, it queries a broker via a service interface. The broker then forwards the client's service request to a server, which processes the request.

# Brok


-transfers
*
1
Broker
1
-transfers
*
Client-Side Proxy
+pack_data()
+unpack_data()
+send_request()
+return()
1
+locateServer()
+locateClient()
+registerServer()
+unregisterServer()
1
Server-Side Proxy
+pack_data()
+unpack_data()
+call_service()
+send_response()
*
-call
1
*
0..1
1
-call
1
-call
Client
+call_server()
+start_task()
+use_Broker_API()
-uses
*
Bridge
+pack_data()
+unpack_data()
+forward_message()
+transmit_message()
*
-uses
Server
+initialize()
+enter_main_loop()
+run_service()
+use_Broker_API()

# Broker Solution – 1

Overview: The broker pattern defines a runtime component, called a broker, that mediates the communication between a number of clients and servers.

Elements:

- *Client,* a requester of services
- *Server,* a provider of services
- *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client
- *Client-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
- *Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages

# Broker Solution - 2

Relations: The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.

Constraints: The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a server-side proxy).

Weaknesses:

- Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck.
- The broker can be a single point of failure.
- A broker adds up-front complexity.
- A broker may be a target for security attacks.
- A broker may be difficult to test.

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

# Pipe and Filter Pattern

# Pipe and Filter Pattern

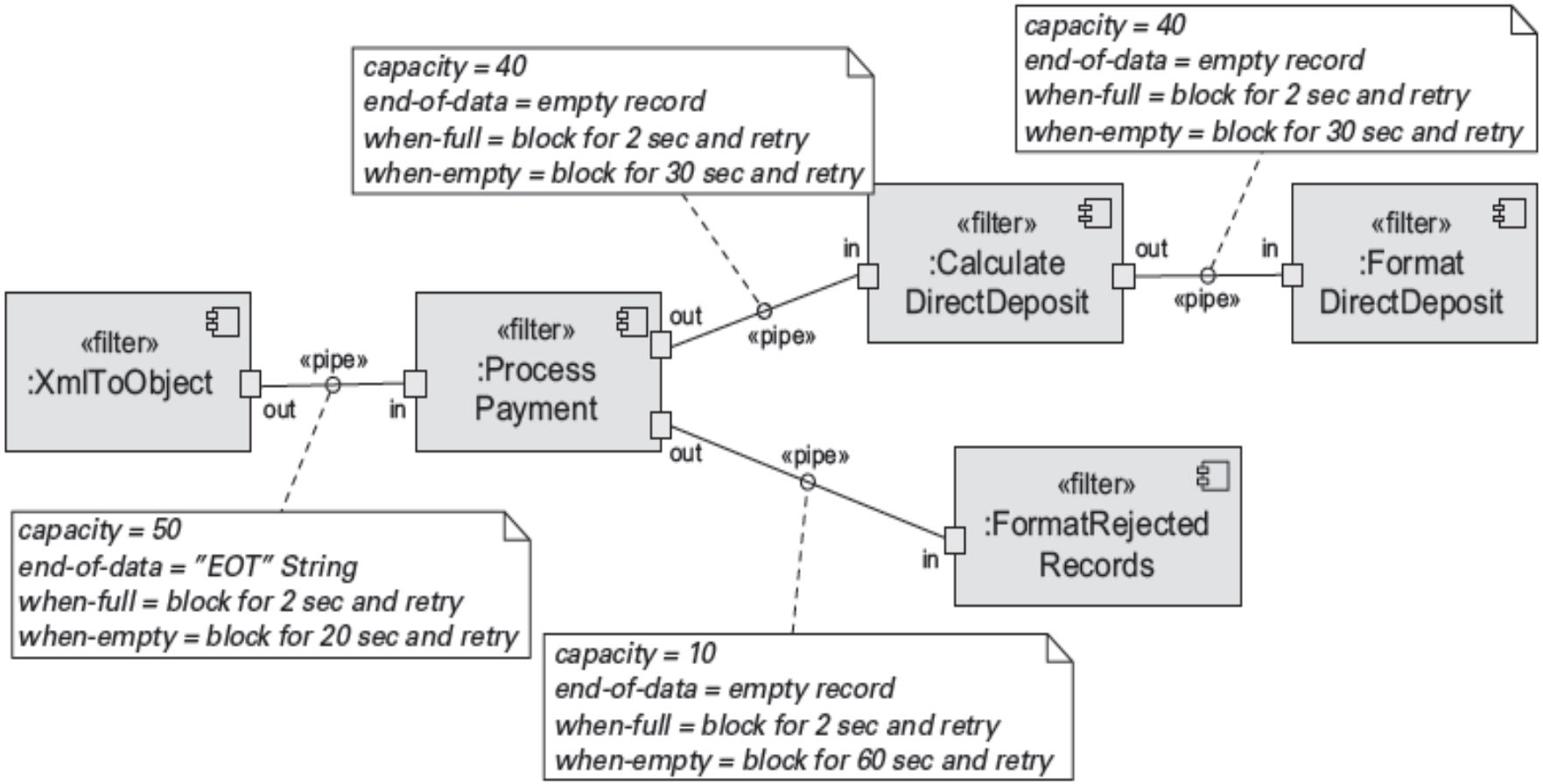


**Context:** Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

**Problem:** Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.

**Solution:** The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

# Pipe and Filter Pattern



# Pipe and Filter Solution

Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.

Elements:

- *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
- *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.

Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.

Constraints:

- Pipes connect filter output ports to filter input ports.
- Connected filters must agree on the type of data being passed along the connecting pipe.