System Architecture

🧠 Architectural Patterns — Short Note

🔧 Definition

Architectural patterns are **reusable solutions** to common system-level design problems.

They capture **experience from past systems**, abstracted from implementation details.

Bengali Note: Architectural pattern মানে পূর্ববর্তী সিস্টেমের অভিজ্ঞতা থেকে পাওয়া গঠনমূলক সমাধান — যা পুনরায় ব্যবহারযোগ্য।

🎯 Purpose

- Guide system architecture before development begins
- Help achieve desired **quality attributes** (e.g., scalability, maintainability)
- Allow early discussion of **design trade-offs**

🏗️ Examples

- **Client-Server**: Clients send requests, server responds
- **Layered Architecture**: Presentation → Business Logic → Data
- **Microservices**: Independent services communicating over APIs
- **Event-Driven**: Components react to events asynchronously

🔍 Pattern vs Style

Concept	Description
Style	Broad structural approach (e.g., layered, pipe-filter)

Concept	Description
Pattern	Specific solution within a style (e.g., MVC in layered style)

Bengali Note: Style হলো সামগ্রিক গঠন, আর Pattern হলো সেই গঠনের ভিতরে নির্দিষ্ট সমাধান।

📚 Learning Goals

After studying architectural patterns, you should be able to:

- Describe structure and function of key patterns
- Compare their pros and cons
- Distinguish between style and pattern
- Identify real-world applications
- Classify patterns by style

🧠 Strategic Insight

Architectural patterns help you:

- Learn from others' experience
- Choose the right structure for your system
- Avoid reinventing the wheel

Bengali Reframe: Pattern শেখা মানে অন্যদের অভিজ্ঞতা থেকে শেখা — যাতে নিজের ডিজাইন আরও শক্তিশালী হয়।

🧠 Architectural Pattern — Short Note

🔧 Definition

An **architectural pattern** is a **proven structural organization schema** for software systems.

It defines a set of **predefined subsystems**, their **responsibilities**, and **rules for interaction**.

Bengali Note: Architectural pattern মানে সফটওয়্যার সিস্টেমের গঠনের জন্য পরীক্ষিত ও পুনরায় ব্যবহারযোগ্য কাঠামো।

🏗️ Example: Client-Server Pattern

- **Client**: Multiple instances, handles user interface
- **Server**: Single instance, processes requests, manages data
- **Relationship**: Client sends requests → Server responds

Bengali Note: Client-Server pattern এ client UI দেখায়, server প্রশ্নের উত্তর দেয় এবং ডেটা রক্ষা করে।

📚 Purpose of Patterns

- Capture **experience** from many developers
- Promote **good design practices**
- Help others **learn from proven solutions**
- Enable **early architectural decisions** before detailed design

Bengali Note: Pattern মানে অভিজ্ঞতা থেকে শেখা — যাতে ডিজাইন আরও ভালো হয় এবং ভুল কম হয়।

🧩 Architectural Style vs Pattern

Term	Meaning
Architectural Style	Broad structural concept (e.g., layered, pipe-filter)
Architectural Pattern	Specific solution within a style (e.g., MVC, Client-Server)

Bengali Note: Style হলো সামগ্রিক গঠন, আর Pattern হলো সেই গঠনের ভিতরে নির্দিষ্ট সমাধান।

🧠 Why Patterns Are Helpful — Short Note

🔧 Definition

Patterns solve **recurring design problems** using **proven solutions** accepted by experienced developers.

Bengali Note: Pattern মানে বারবার দেখা ডিজাইন সমস্যা সমাধানের জন্য পরীক্ষিত সমাধান।

✅ Key Benefits

- **Reusability**: Makes it easier to apply known solutions again
- **Shared Vocabulary**: Pattern names (e.g., MVC, Client-Server) become part of a common design language
- **Documentation**: Captures architectural decisions clearly
- **Vision Preservation**: Helps maintain original design intent during updates
- **Non-functional Support**: Addresses scalability, changeability, performance, etc.

Bengali Note:

- Pattern বারবার ব্যবহারযোগ্য
- ডিজাইনের ভাষা তৈরি করে
- কোড পরিবর্তনের সময় মূল ভিশন ধরে রাখতে সাহায্য করে
- UI পরিবর্তনের সুবিধা দেয় (যেমন MVC pattern)

🏗️ Example

- **Client-Server**: Server should not initiate communication — pattern enforces correct responsibility
- **MVC**: Supports flexible UI changes without breaking logic

---

🧠 Strategic Insight

Patterns are **building blocks** for complex systems.

They let you **learn from others' experience**, avoid reinventing the wheel, and build software with **defined properties**.

Bengali Reframe: Pattern শেখা মানে অভিজ্ঞদের পথ অনুসরণ করে নিজের ডিজাইন আরও শক্তিশালী করা।

🧠 Pattern Schema / Template — Short Note

🔧 What Is It?

A **pattern template** (or schema) is a structured format used to describe a design pattern clearly and consistently.

Bengali Note: Pattern template হলো একটি কাঠামো — যা দিয়ে pattern এর context, problem, এবং solution সুন্দরভাবে ব্যাখ্যা করা হয়।

📦 Core Components

Component	Description
Context	The situation that gives rise to the problem
Problem	A recurring issue in that context, shaped by constraints and goals (called **forces**)
Forces	Conflicting or negotiable conditions (e.g., performance vs extensibility)
Solution	A proven structural solution with components, relationships, and runtime behavior

Bengali Note:

- Context = সমস্যা কোথায় দেখা দেয়
- Problem = বারবার দেখা দেওয়া ডিজাইন সমস্যা
- Forces = সীমাবদ্ধতা ও চাহিদা (যেগুলো একে অপরের সাথে সংঘর্ষ করতে পারে)
- Solution = পরীক্ষিত সমাধান, গঠন ও আচরণ সহ

🧠 Strategic Insight

Pattern templates help:

- Standardize pattern documentation
- Make reuse easier
- Clarify trade-offs and design decisions

Bengali Reframe: Pattern template ডিজাইন সমাধানকে সহজে বোঝার ও পুনরায় ব্যবহারের উপযোগী করে তোলে।

🧠 Design vs Architectural Patterns — Short Note

🔧 Design Pattern

- Provides a **common solution** for a recurring **class-level problem**
- Involves **classes and objects** working together
- Affects **one subsystem**, not the whole system
- Helps **implement architectural patterns**

Example:

- **Observer Pattern** helps implement **MVC** by linking View to Model updates

Bengali Note: Design pattern class বা অবজেক্ট লেভেলের সমস্যা সমাধান করে — যেমন Observer pattern দিয়ে MVC pattern তৈরি করা যায়।

🏗️ Architectural Pattern

- Defines the **overall structure** of a software system
- Involves **subsystems**, not just classes
- Influences the **fundamental architecture**
- Examples: **MVC**, **Client-Server**, **Microservices**

Bengali Note: Architectural pattern পুরো সিস্টেমের গঠন নির্ধারণ করে — subsystem গুলো কিভাবে কাজ করবে তা বলে দেয়।

📚 Origin & Evolution

- Concept introduced by **Christopher Alexander** in building architecture
- Popularized in software by the **Gang of Four (GoF)** book
- Expanded into:
  - **Analysis patterns**

- **UI patterns**
- **Programming idioms**
- **Functional design patterns**

🧠 Summary Tab

Feature	Design Pattern	Architectural Pattern
Scope	Class/Object level	System/Subsystem level
Impact	One subsystem	Whole system
Example	Observer, Factory	MVC, Client-Server
Role	Implementation aid	Structural blueprint

🧠 Examples of Architectural Patterns — Short Note

Each architectural pattern defines:

- **Components**: Subsystems involved
- **Connections**: How they interact
- **Usage Examples**: Where it fits
- **Pros & Cons**: Trade-offs and design impact

🏗️ 1. **Client-Server Pattern**

- **Components**: Client(s), Server
- **Connection**: Client sends requests → Server responds
- **Usage**: Web apps, database systems
- **Advantages**: Centralized control, scalable
- **Disadvantages**: Server bottleneck, single point of failure

Bengali Note: Client-Server pattern এ client প্রশ্ন করে, server উত্তর দেয় — সহজ কিন্তু server overload হতে পারে।

🏗️ 2. **Model-View-Controller (MVC)**

- **Components**: Model (data), View (UI), Controller (logic)
- **Connection**: Controller updates Model → View reflects changes
- **Usage**: GUI apps, web frameworks
- **Advantages**: Separation of concerns, flexible UI
- **Disadvantages**: Complexity in small apps

Bengali Note: MVC pattern এ UI, data, এবং logic আলাদা থাকে — পরিবর্তন সহজ হয়।

🏗️ 3. **Layered Architecture**

- **Components**: Presentation → Business Logic → Data Access
- **Connection**: Each layer interacts only with adjacent layers
- **Usage**: Enterprise apps, banking systems
- **Advantages**: Modularity, maintainability
- **Disadvantages**: Performance overhead

Bengali Note: Layered pattern এ প্রতিটি স্তর নির্দিষ্ট কাজ করে — কিন্তু পারফরমেন্স কম হতে পারে।

🏗️ 4. **Microservices**

- **Components**: Independent services with APIs
- **Connection**: Communicate via HTTP, messaging
- **Usage**: Scalable web platforms, cloud-native apps
- **Advantages**: Scalability, independent deployment
- **Disadvantages**: Complex communication, testing challenges

Bengali Note: Microservices pattern এ প্রতিটি সার্ভিস আলাদা — স্কেল করা সহজ, কিন্তু debug কঠিন।

🧠 Layers Pattern — Short Note

🔧 Definition

The **Layers Pattern** structures an application into **levels of abstraction**, where each layer provides services to the layer above and uses services from the layer below.

Bengali Note: Layers pattern এ প্রতিটি স্তর তার নিচের স্তরের সার্ভিস ব্যবহার করে এবং উপরের স্তরকে সার্ভিস দেয়।

📦 Key Concepts

Concept	Description
Layer n	Higher abstraction (e.g., UI, application logic)
Layer n-1	Lower abstraction (e.g., OS, transport layer)
Service Calls	Typically synchronous procedure calls
Direction	Top layer → uses bottom layer; not the reverse

✅ Benefits

- **Reusability**: Lower layers can be reused by multiple upper layers (e.g., TCP used by FTP, Telnet)
- **Standardization**: Clear abstraction levels enable standardized interfaces
- **Isolation**: Changes in one layer don't affect others if interfaces are respected
- **Team Development**: Layers can be developed and tested independently

Bengali Note: প্রতিটি স্তর আলাদাভাবে তৈরি ও পরীক্ষা করা যায় — কোড পরিবর্তনের সময় অন্য স্তরে প্রভাব পড়ে না।

🧪 Examples

- **ISO/OSI Model**: 7-layer network protocol (Application → Physical)
- **Java Virtual Machine**: JVM uses OS services

- **Web Apps**: Presentation → Application Logic → Domain Logic → Data
- **Microkernel Systems**: Windows NT, QNX, JBoss

⚠️ Issues

- Lower layers are **more stable** — changes here affect upper layers
- Abstract interfaces are **hard to define**
- **Performance overhead** due to repeated data transformations
- Lower layers may do **unnecessary work** for some upper layers

---

🔄 Variants

Variant	Description	Trade-off
Relaxed Layered System	Upper layer can use any lower layer	More efficient, less maintainable
Callback	Lower layer notifies upper layer via registered function	Enables bottom-up communication

Bengali Note: Relaxed system efficiency বাড়ায়, কিন্তু maintain করা কঠিন হয়। Callback দিয়ে নিচের স্তর উপরের স্তরকে ইভেন্ট জানাতে পারে

🧠 Client-Server Pattern — Short Note

🔧 Definition

A **server** provides services to multiple **clients**.

Clients send requests; server listens and responds — often across **process/machine boundaries**.

Bengali Note: Client-Server pattern এ client অনুরোধ পাঠায়, server সেই অনুরোধের উত্তর দেয় — আলাদা মেশিনেও হতে পারে।

📦 Structure

Component	Role
Client	Requests services
Server	Provides services, always active
Communication	Uses TCP/IP or other inter-process protocols

🧪 Examples

- Remote database access (e.g., MySQL)
- Remote file systems (e.g., NFS)
- Web apps (e.g., browser ↔ web server)

✅ Benefits

- Centralized control
- Scalable with multiple clients
- Intermediate layers possible (e.g., caching, security, load balancing)

⚠️ Issues

- **Overhead**: Inter-process communication and data marshalling
- **Transparency**: Clients shouldn’t need to know server details (location/platform)
- **Threading**: Server handles requests in separate threads

🔄 State Management

Type	Description	Trade-offs
Stateless Server	Client manages session state (e.g., cookies, URL params)	REST-friendly, scalable, but less secure
Stateful Server	Server stores session state per client	Easier logic, but memory-heavy and less scalable

Bengali Note: Stateless server এ client নিজেই state পাঠায় — REST এর জন্য দরকার। Stateful server এ server সব মনে রাখে — কিন্তু স্কেল করা কঠিন।

🧠 Strategic Insight

- Client-Server is a **variant of Layered Architecture**, crossing machine boundaries.
- Can evolve into **Peer-to-Peer** if callbacks and mutual communication are added.
- Must balance **performance, fault tolerance, and scalability**.

Bengali Reframe: Client-Server pattern হলো distributed world এর ভিত্তি — কিন্তু state, security, এবং scalability নিয়ে সাবধান থাকতে হয়।

🧠 REST Architecture — Short Note

🔧 Definition

REST stands for **Representational State Transfer**.

It's a **client-server architecture** with a **uniform interface**, **stateless communication**, and **addressable resources**.

Bengali Note: REST হলো একটি client-server আর্কিটেকচার — যেখানে প্রতিটি অনুরোধে state পাঠানো হয় এবং সার্ভার আলাদা থাকে।

📦 Key Principles

Principle	Description
Client-Server	Clients and servers are separated; clients request, servers respond
Stateless	Each request contains all necessary info; server doesn't store session state
Addressable Resources	Every resource (or server state) is accessible via a unique URL
Uniform Interface	Standardized way to interact (e.g., HTTP methods: GET, POST, PUT, DELETE)

Principle	Description
Layered System	Clients may connect through intermediaries (e.g., proxies, load balancers)

Bengali Note:

- Stateless মানে server কোনো session মনে রাখে না
- প্রতিটি resource এর জন্য আলাদা URL থাকে
- HTTP method দিয়ে কাজ হয় (GET, POST ইত্যাদি)

🧪 Example Use Case

- **Web applications** using stateless communication (e.g., RESTful APIs)
- **Mobile apps** calling backend services via REST
- **Microservices** communicating over REST endpoints

✅ Benefits

- Scalability
- Simplicity
- Loose coupling between client and server
- Easy caching and load balancing

🧠 Master-Slave Pattern — Short Note

🔧 Definition

The **Master-Slave pattern** supports **fault tolerance** and **parallel computation**.

The **master** distributes tasks to multiple **slaves**, collects their results, and computes the final output.

Bengali Note: Master-Slave pattern এ master কাজ ভাগ করে দেয়, slaves সেই কাজ করে — তারপর master ফলাফল একত্র করে।

📦 Structure

Component	Role
Master	Splits work, coordinates slaves, combines results
Slaves	Perform subtasks independently
Client	Requests service from master

🧪 Application Areas

1. **Fault Tolerance**
    - Master sends same task to multiple slaves
    - Strategies:
        - First response wins
        - Majority result wins
    - Master detects slave failure via timeouts
2. **Parallel Computation**
    - Master divides complex task into subtasks
    - Example: Matrix multiplication — each row computed by a slave
3. **Computational Accuracy**
    - Slaves run different implementations
    - Master chooses best result (e.g., average or majority)

Bengali Note:

- Fault tolerance: একাধিক slave কে একই কাজ দিয়ে ফলাফল যাচাই
- Parallel computing: বড় কাজ ভাগ করে দ্রুত সমাধান
- Accuracy: বিভিন্ন implementation থেকে সেরা ফলাফল নির্বাচন

✅ Benefits

- Improves reliability and fault tolerance

- Enables parallelism and scalability
- Supports accuracy through redundancy

⚠️ Issues

- **Master is a single point of failure**
- **Slaves must be identical or compatible**
- **Strategy must handle conflicting results**

Bengali Note: Master fail করলে পুরো সিস্টেম fail — তাই master কে শক্তিশালী রাখতে হয়।

🧠 Pipe-Filter Pattern — Short Note

🔧 Definition

The **Pipe-Filter pattern** structures systems that process **streams of data**.

Each **filter** performs a transformation, and **pipes** connect filters to pass data.

Bengali Note: Pipe-Filter pattern এ প্রতিটি ধাপ একটি filter হিসেবে কাজ করে — data sequentially flow করে pipe দিয়ে।

📦 Structure

```
Source → Filter → Filter → Sink
```

Component	Role
Filter	Encapsulates a processing step
Pipe	Transfers data between filters (may buffer/synchronize)
Source/Sink	Input/output endpoints

🧪 Examples

1. **Unix Shell Commands**

   `cat file | grep xyz | sort | uniq > out`

   Each command is a filter; data flows through pipes.

2. **Compiler Pipeline**

   Lexical Analysis → Parsing → Semantic Analysis → Code Generation → Optimization

   Each stage transforms the data stream (e.g., tokens → syntax tree → bytecode)

Bengali Note: Compiler এর প্রতিটি ধাপ একটি filter — source code থেকে machine code পর্যন্ত sequential transformation হয়।

✅ Benefits

- **Modular**: Easy to add/remove filters
- **Reusable**: Filters can be recombined
- **Concurrent**: Filters process data as it arrives (lazy evaluation)
- **Composable**: Behavior can be described as function composition: `f(g(x))`

⚠️ Issues

- **Overhead**: Data transformation between filters (e.g., string ↔ number)
- **Deadlocks**: If filters wait for full input (e.g., `sort` in Unix)
- **Not ideal for interactive apps**
- **Global state sharing breaks purity** (e.g., symbol table in compilers)

Bengali Note: Deadlock হতে পারে যদি কোনো filter সব data না পেলে output না দেয় — buffer ছোট হলে সমস্যা বাড়ে।

🧠 Strategic Insight

Pipe-Filter is ideal for:

- **Stream-based processing**
- **Batch transformations**
- **Concurrent pipelines**

But less suited for:

- **Interactive systems**
- **Stateful coordination**

Bengali Reframe: Pipe-Filter pattern stream-based কাজের জন্য দুর্দান্ত — কিন্তু interactive বা state-heavy অ্যাপের জন্য নয়।

🧠 Broker Pattern — Short Note

🔧 Definition

The **Broker Pattern** structures **distributed systems** with **decoupled components**.

A central **broker** coordinates communication between clients and servers via **remote service invocation**.

Bengali Note: Broker pattern এ client ও server একে অপরকে সরাসরি চেনে না — broker মধ্যস্থতা করে।

📦 Structure

Component	Role
Broker	Registers services, forwards requests, returns results/errors
Server	Publishes capabilities to broker
Client	Requests service from broker

🧪 Examples

1. **Web Services**
   - **Broker**: UDDI registry
   - **IDL**: WSDL
   - **Protocol**: SOAP (XML-based)
2. **CORBA (Common Object Request Broker Architecture)**
   - Enables communication between heterogeneous object-oriented systems
   - Uses **OMG-IDL** for interface definition

3. **Microsoft OLE / UNO (OpenOffice)**
    - Uses binary interface tables for method invocation

Bengali Note: Web service এ UDDI হলো broker — client এখানে service খুঁজে পায়, SOAP দিয়ে invoke করে।

✅ Benefits

- **Loose coupling**: Components don't need to know each other's location or details
- **Central coordination**: Broker handles communication logic
- **Scalability**: Easy to add/remove services
- **Flexibility**: Supports heterogeneous systems

⚠️ Issues

- **Broker is a single point of failure**
- **Performance bottleneck** if broker is overloaded
- **Complexity** in broker implementation and service registry

---

🔤 Interface Definition Language (IDL)

IDL	Used In
OMG-IDL	CORBA
WSDL	Web Services
.NET CIL	Microsoft
Binary Tables	OLE, UNO

Bengali Note: IDL হলো server এর interface এর টেক্সট/বাইনারি বর্ণনা — যাতে client বুঝতে পারে কী কী service আছে।

🧠 Peer-to-Peer Pattern — Short Note

🔧 Definition

The **Peer-to-Peer (P2P) pattern** is a **symmetric client-server model** where each peer can act as both **client** and **server**, dynamically switching roles.

Bengali Note: P2P pattern এ প্রতিটি peer client ও server হিসেবে কাজ করতে পারে — role পরিবর্তনযোগ্য।

📦 Structure

```
Peer A ⇄ Peer B ⇄ Peer C
```

Role	Description
Client	Requests services from other peers
Server	Provides services to other peers
Multithreaded	Peers often run multiple threads for concurrency
Event Notification	Peers can notify others via event-bus or stream

---

🧪 Examples

- **DNS** (Domain Name System)
- **Sciencenet** (distributed search engine)
- **BitTorrent**, **Gnutella** (file sharing)
- **Collaborative apps** (e.g., shared drawing boards)

✅ Benefits

- **Low cost of ownership**: Each node contributes its own capacity
- **Self-organizing**: Minimal admin overhead
- **Scalable**: Performance improves with more peers
- **Resilient**: Failure of one peer doesn’t crash the system
- **Dynamic topology**: Peers can join/leave anytime

Bengali Note: P2P system নিজে নিজে সংগঠিত হয় — নতুন peer যোগ বা বাদ দিতে সমস্যা হয় না।

⚠️ Issues

Issue	Description
Quality of Service	No guarantees — peers cooperate voluntarily
Security	Hard to enforce — decentralized control
Performance	Depends on number of active peers — few peers = low performance

Bengali Note: P2P system এ security নিশ্চিত করা কঠিন — কারণ সবাই স্বাধীনভাবে কাজ করে।

🧠 Strategic Insight

P2P is ideal for:

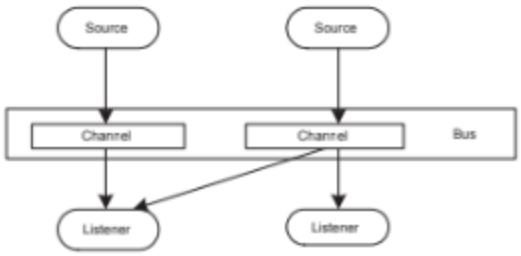
- **Decentralized systems**
- **Collaborative platforms**
- **File sharing and distributed search**

But less suited for:

- **Secure enterprise systems**
- **Guaranteed service quality**

Bengali Reframe: Peer-to-Peer pattern স্বাধীনতা ও স্কেল দেয় — কিন্তু security ও reliability নিয়ে সতর্ক থাকতে হয়।

2.7 EVENT-BUS PATTERN



🧠 2.7 Event-Bus Pattern — Short Note

🔧 Definition

The **Event-Bus pattern** handles **asynchronous event communication**.

Sources publish messages to **channels**, and **listeners** subscribe to those channels.

Bengali Note: Event-Bus pattern এ source message পাঠায়, listener সেই channel সাবস্ক্রাইব করে notification পায়।

📦 Structure

> Source → Channel → Bus → Listener

Component	Role
Source	Generates events
Channel	Logical stream for event types
Bus	Routes events to listeners
Listener	Reacts to subscribed events

🧪 Examples

Examples

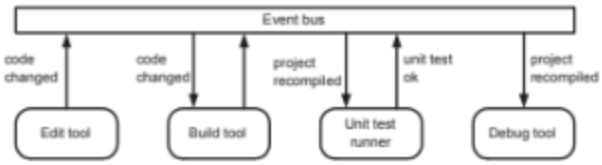


FIGURE 3.12 Software development environment

- **Java Event Model**
- **Software Dev Tools**: Edit tool → Build tool → Unit test → Debug tool
- **Real-time Middleware**: OpenSplice

- **Trading Systems**, **Process Monitoring**

✅ Benefits

- Easy to add/remove publishers and subscribers
- Loose coupling between components
- Supports concurrent processing

⚠️ Issues

- **Delivery assumptions** (order, timing) are hard to guarantee
- **Scalability bottleneck** if bus is overloaded
- **Event transformation/coordination** adds complexity

Bengali Note: Scalability সমস্যা হতে পারে যদি অনেক message একসাথে bus দিয়ে যায়।

:

🧠 Model-View-Controller (MVC) Pattern — Short Note

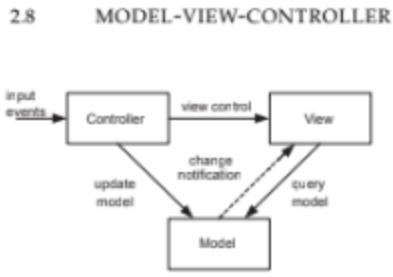


FIGURE 3.13 Model-View-Controller

🔧 Definition

MVC divides an interactive application into three components:

- **Model**: Core logic and data
- **View**: Displays data to the user

- **Controller**: Handles user input and updates the model

Bengali Note: Model data রাখে, View দেখায়, Controller input নিয়ে Model আপডেট করে।

📦 Structure & Flow

```
User → Controller → Model → View
```

Component	Role
Controller	Handles input events
Model	Updates data and notifies changes
View	Queries model and updates display

- Uses **Observer Pattern**: View listens for model changes
- Multiple views can be connected/disconnected at runtime

🧪 Examples

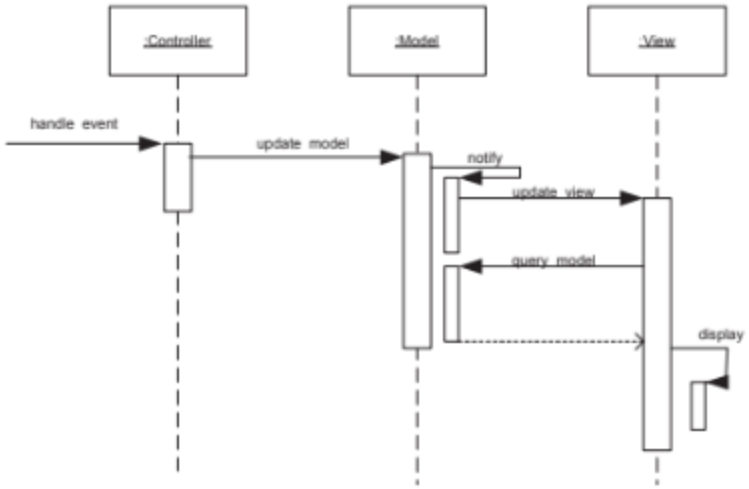


FIGURE 3.14 Sequence diagram of the MVC pattern

- **Smalltalk** (origin of MVC)

- **Web apps** (e.g., Spring MVC, Django)
- **Windows Document View** (Word, PowerPoint — print/web/overview modes)

✅ Benefits

- **Separation of concerns**: UI, logic, and input are modular
- **Multiple views**: Same model can support different UIs
- **Flexible UI updates**: Easy to change look-and-feel

⚠️ Issues

Issue	Description
Complexity	Not all UI elements (e.g., menus) fit MVC cleanly
Over-updating	One input may trigger multiple view updates
Tight coupling	View and controller often closely tied to model
Web sync	Model changes require updates in both view and controller (e.g., adding email field)

Bengali Note: MVC শক্তিশালী হলেও ছোট UI element এর জন্য অতিরিক্ত জটিলতা তৈরি করতে পারে — এবং unnecessary update হতে পারে।

🧠 Strategic Insight

MVC is ideal for:

- **GUI-heavy applications**
- **Multiple view support**
- **Framework-based development**

But less suited for:

- **Simple UIs**
- **Highly coupled logic-display scenarios**

Bengali Reframe: MVC pattern বড় অ্যাপের জন্য উপযোগী — কিন্তু ছোট বা সরল UI এর জন্য অতিরিক্ত জটিলতা আনতে পারে।

🧠 Blackboard Pattern — Short Note

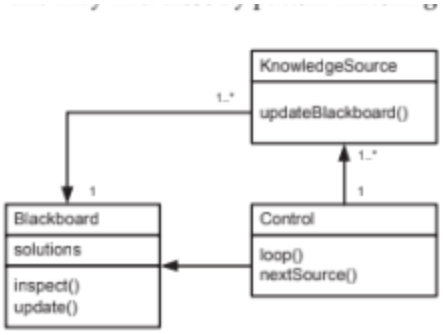


FIGURE 3.15 Blackboard pattern

🔧 Definition

The **Blackboard Pattern** is used for problems with **no deterministic solution**.

Multiple **specialized subsystems** (knowledge sources) collaborate by reading/writing to a **shared data space** called the **blackboard**.

Bengali Note: Blackboard pattern এমন সমস্যার জন্য যেখানে নির্দিষ্ট সমাধান নেই — সবাই মিলে একটি common board এ data যোগ করে ও বিশ্লেষণ করে।

📦 Structure

> Knowledge Sources ⇄ Blackboard ⇄ Control Component

Component	Role
Blackboard	Shared data space
Knowledge Sources	Add/update data based on expertise
Control	Coordinates which source acts next

- Uses **pattern matching** to find relevant data
- Solutions may be **partial or approximate**

🧪 Examples

- **Speech recognition**
- **Submarine detection**
- **3D molecular structure inference**
- **Tuple Space systems** (e.g., JavaSpaces)

✅ Benefits

- Easy to add new applications
- Easy to extend data structure
- Encourages modular collaboration

⚠️ Issues

Issue	Description
Hard to modify structure	All components depend on shared format
Agreement needed	All processes must agree on data schema
Access control	Synchronization may be required to avoid conflicts

Bengali Note: Blackboard এর structure পরিবর্তন করা কঠিন — কারণ সব component সেই structure এর উপর নির্ভর করে।

🧠 Strategic Insight

Blackboard is ideal for:

- **AI and inference systems**
- **Collaborative problem solving**
- **Non-deterministic domains**

But less suited for:

- **Simple or deterministic workflows**
- **Systems needing strict control or fast response**

Bengali Reframe: Blackboard pattern হলো collective intelligence এর ডিজাইন — যেখানে সবাই মিলে সমাধান খোঁজে।

🧠 Interpreter Pattern — Short Note

🔧 Definition

The **Interpreter Pattern** is used to design components that **interpret programs written in a dedicated language**.

It allows easy replacement or modification of the interpreted logic.

Bengali Note: Interpreter pattern এমন component তৈরি করে যা নির্দিষ্ট ভাষায় লেখা প্রোগ্রামকে ব্যাখ্যা করে — সহজে পরিবর্তনযোগ্য।

📦 Key Concepts

Concept	Description
Dedicated Language	Domain-specific or scripting language
Interpreter Component	Parses and executes instructions
Replaceability	Logic can be swapped without recompiling

🧪 Examples

- **Rule-based systems** (e.g., expert systems)
- **Web scripting**: JavaScript (client-side), PHP (server-side)
- **Postscript**: Page description language

✅ Benefits

- **Flexibility**: Easy to replace or update interpreted logic
- **Modularity**: Logic separated from core system
- **Rapid prototyping**: Ideal for evolving rules or scripts

⚠️ Issues

Issue	Description
Performance	Slower than compiled languages
Testing Gaps	Easy replacement may lead to poor testing
Security Risks	Dynamic code injection or unstable scripts

Bengali Note: Interpreter pattern flexible হলেও performance কম এবং security risk বেশি — কারণ কোড runtime এ পরিবর্তন হয়।

🧠 Strategic Insight

Interpreter is ideal for:

- **Dynamic rule engines**
- **Configurable scripting**
- **Domain-specific languages**

But less suited for:

- **Performance-critical systems**
- **Strictly tested production environments**

Bengali Reframe: Interpreter pattern দ্রুত পরিবর্তনযোগ্য লজিকের জন্য উপযোগী — কিন্তু নিরাপত্তা ও গতি নিয়ে সতর্ক থাকতে হয়।

🧠 KWIC Problem — Short Note

🔧 Problem Definition

- **Input**: A list of lines (each line = sequence of words)
- **Output**: All **circular shifts** of each line, **sorted**

Example:

Input: `man eats dog`

Output:

```
dog man eats
eats dog man
man eats dog
```

Bengali Note: KWIC problem এ প্রতিটি লাইনের শব্দ ঘুরিয়ে নতুন লাইন তৈরি হয় — তারপর সব লাইন sort করা হয়।

🏗️ 3.1 Shared Data Pattern — KWIC Classical Solution

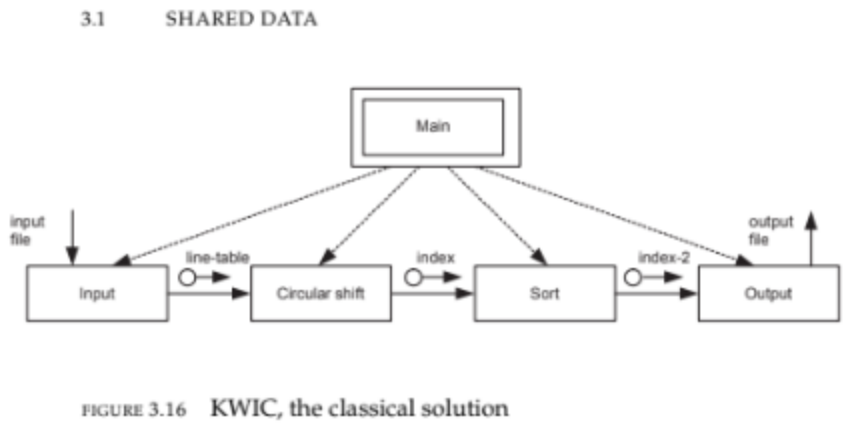


FIGURE 3.16 KWIC, the classical solution

📦 Structure (Yourdon Chart)

```
Main
 ├── Input → adds lines to shared data
 ├── Circular Shift → adds shifted indexes
 ├── Sort → adds sorted indexes
 └── Output → writes final result
```

Component	Role
Shared Data	Central repository (line-table, index, index-2)
Subroutines	Operate sequentially, modifying shared data
Main	Coordinates function calls

✅ Benefits

- **Modular decomposition**: Each function does one task
- **Shared memory**: Easy data exchange between components
- **Functional clarity**: Each step is isolated and testable

⚠️ Issues

Issue	Description
Tight coupling	All components depend on shared data structure
Global state	Hard to modify shared structure without affecting all
Synchronization	Needed if parallelized or extended

Bengali Note: Shared data structure পরিবর্তন করলে সব ফাংশনে প্রভাব পড়ে — তাই careful design দরকার।

🧠 Strategic Insight

The KWIC problem is a classic benchmark to:

- Compare **Shared Data**, **Pipe-Filter**, **Repository**, and **Layered** patterns
- Analyze **modularity**, **data flow**, and **coupling**
- Practice **functional decomposition** and **index-based manipulation**

Bengali Reframe: KWIC হলো architectural pattern শেখার জন্য আদর্শ সমস্যা — যেখানে data flow, modularity, এবং coupling বিশ্লেষণ করা যায়।

🧠 KWIC with Layers Pattern — Short Note

🔧 Problem Recap

- **KWIC** = **KeyWord In Context**
- Input: lines of words
- Output: all **circular shifts**, **sorted**

Bengali Note: KWIC problem এ প্রতিটি লাইনের শব্দ ঘুরিয়ে নতুন লাইন তৈরি হয় — তারপর সব লাইন sort করা হয়।

🏗️ 3.2 Layers Pattern Solution

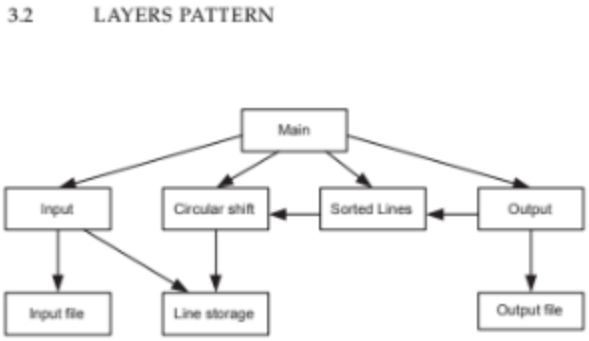


FIGURE 3.17 KWIC, Layers pattern

📦 Structure

```
Main
 ├── Input Layer → reads file, stores lines
 ├── Circular Shift Layer → generates shifted lines
 ├── Sort Layer → sorts shifted lines
 └── Output Layer → writes result to file
```

Layer	Responsibility
Input	Reads input file, stores lines
Circular Shift	Performs circular shifts
Sort	Sorts shifted lines
Output	Writes sorted lines to output file

- **Objects** encapsulate data and logic
- **No shared data** — only method calls between layers
- **Encapsulation** allows internal changes without breaking interfaces

Bengali Note: প্রতিটি layer নিজস্ব data ও method ব্যবহার করে — অন্য layer এর সাথে method call এর মাধ্যমে যোগাযোগ করে।

✅ Benefits

- **Encapsulation**: Data structures hidden inside objects
- **Modularity**: Each layer has a clear role
- **Maintainability**: Easy to change internals without affecting other layers
- **Layered abstraction**: Each layer only calls the one below

---

⚠️ Issues

Issue	Description
Strict layering	Limits flexibility — can't skip layers
Performance overhead	Due to method chaining and abstraction
Debugging complexity	Harder to trace across layers

Bengali Note: Layer strict হলে flexibility কমে — performance ও debugging এ সমস্যা হতে পারে।

🧠 Strategic Insight

- Layers pattern in KWIC promotes **clean separation** and **encapsulation**
- Ideal for **object-oriented design**
- Useful when **data hiding** and **interface stability** are priorities

Bengali Reframe: KWIC problem এ Layers pattern ব্যবহার করলে প্রতিটি ধাপ আলাদা থাকে — ফলে পরিবর্তন সহজ হয়, কিন্তু strict layering এর কারণে কিছু সীমাবদ্ধতা থাকে।

🧠 KWIC with Event-Bus Pattern — Short Note

3.3 EVENT-BUS

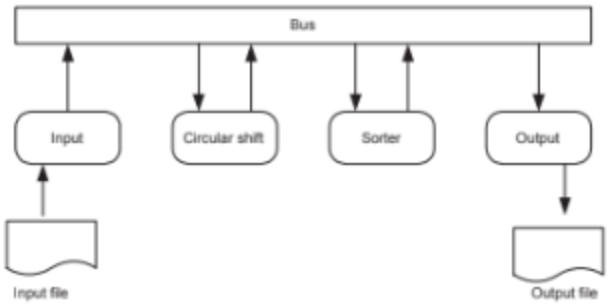


FIGURE 3.18 KWIC, Event Bus pattern

🔧 Problem Recap

- **KWIC** = KeyWord In Context
- Input: lines of words
- Output: all **circular shifts**, **sorted**

Bengali Note: KWIC problem এ প্রতিটি লাইনের শব্দ ঘুরিয়ে নতুন লাইন তৈরি হয় — তারপর সব লাইন sort করা হয়।

🏗️ 3.3 Event-Bus Pattern Solution

📦 Structure

```
Input → [Event Bus] → Circular Shift → [Event Bus] → Sorter → Output
```

Component	Role
Input	Reads file, publishes "line inserted" event
Circular Shift	Listens for input event, publishes "shifted line" event
Sorter	Listens for shift event, sorts lines
Output	Listens for sorted data, writes to file
Bus	Routes events between components

- **Implicit invocation**: Components react to events, not direct calls

- **Shared data**: Used between Input–Shift and Shift–Sort–Output

Bengali Note: Event-Bus pattern এ component গুলো একে অপরকে সরাসরি call করে না — event এর মাধ্যমে কাজ করে।

✅ Benefits

- **Parallelism**: Components can run concurrently
- **Loose coupling**: Components don't need to know each other
- **Dynamic extensibility**: Easy to add new listeners or publishers

---

⚠️ Issues

Issue	Description
No data hiding	Shared data is exposed across components
Complexity	Two shared data spaces + event bus = harder to manage
Delivery assumptions	Event order and timing may be unpredictable
Scalability bottleneck	Event bus may become overloaded

Bengali Note: Event-Bus pattern flexible হলেও data hiding নেই — এবং scalability সমস্যা হতে পারে যদি message বেশি হয়।

🧠 Strategic Insight

Event-Bus is ideal for:

- **Reactive systems**
- **Concurrent processing**
- **Dynamic workflows**

But less suited for:

- **Strict encapsulation**
- **Simple linear pipelines**

Bengali Reframe: KWIC problem এ Event-Bus pattern concurrency ও flexibility দেয় — কিন্তু structure জটিল হয় এবং data hiding কমে যায়।

🧠 KWIC with Pipe-Filter Pattern — Short Note

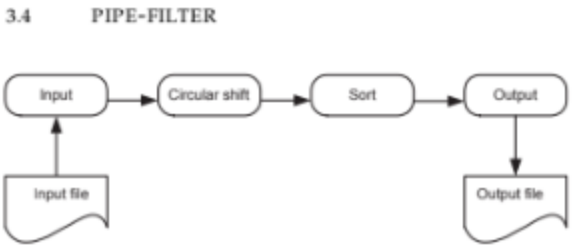


FIGURE 3.19 KWIC, Pipe-filter pattern

🔧 Problem Recap

- **KWIC** = KeyWord In Context
- Input: lines of words
- Output: all **circular shifts**, **sorted**

Bengali Note: KWIC problem এ প্রতিটি লাইনের শব্দ ঘুরিয়ে নতুন লাইন তৈরি হয় — তারপর সব লাইন sort করা হয়।

🏗️ 3.4 Pipe-Filter Pattern Solution

📦 Structure

```
Input → Circular Shift → Sort → Output
```

Component	Role
Input Filter	Reads file, emits lines
Circular Shift Filter	Transforms lines into all circular shifts
Sort Filter	Sorts shifted lines
Output Filter	Writes sorted lines to output file
Pipes	Transfer data between filters in a uniform format

- Each filter is **independent**, processes data **incrementally**
- Filters can be **reused** and **reordered**

- **Uniform data format** required for smooth pipe flow

Bengali Note: প্রতিটি filter data stream এর উপর কাজ করে — output পরবর্তী filter এ যায় pipe দিয়ে।

✅ Benefits

- **Modular & Reusable**: Filters can be recombined
- **Parallelism**: Filters can run concurrently like Event-Bus
- **Encapsulation**: Internal logic hidden like Layers pattern
- **Simple Composition**: System behavior = `f(g(x))`

⚠️ Issues

Issue	Description
Data Format Overhead	Filters may use different internal structures → translation cost
Limited Interactivity	Not ideal for interactive applications
Deadlocks	If a filter waits for full input before producing output

Bengali Note: Filters এর মধ্যে data format mismatch হলে overhead বাড়ে — এবং কিছু filter full input না পেলে deadlock হতে পারে।

🧠 Strategic Insight

Pipe-Filter is ideal for:

- **Stream-based processing**
- **Batch transformations**
- **Concurrent pipelines**

But less suited for:

- **Interactive systems**
- **Highly stateful coordination**

Bengali Reframe: KWIC problem এ Pipe-Filter pattern concurrency ও simplicity দেয় — কিন্তু data format mismatch হলে সমস্যা হতে পারে।

🧠 Architectural Styles — Short Note

🔄 Patterns vs Styles

Concept	Description
Pattern	Bottom-up: solution reused repeatedly for a specific problem
Style	Top-down: system classified by component & connector configuration

Bengali Note: Pattern হলো নির্দিষ্ট সমস্যার সমাধান — Style হলো system এর গঠনভিত্তিক শ্রেণিবিন্যাস।

🏗️ Mary Shaw & Paul Clements Classification

1. 🔁 Interacting Processes

- Each component has its own **thread of control**
- Communication via:
  - **Asynchronous messages**
  - **Events (implicit invocation)**
  - **Remote Procedure Calls (RPC)**

Examples:

- **Event-Bus**, **Client-Server**, **Peer-to-Peer**

Bengali Note: প্রতিটি অংশ নিজে চালিত হয় — message বা event দিয়ে যোগাযোগ করে।

---

2. 🔄 Dataflow Style

- Data flows **sequentially** between components
- Ideal for **stream processing**

Examples:

- **Pipe-Filter**, streaming **Client-Server**

Bengali Note: Data একের পর এক component এ যায় — stream এর মতো।

---

3. 🗄️ Data-Centred Style

- Centralized **shared data store**
- Components interact via **data repository**

Examples:

- **Blackboard**, database-driven **Client-Server**

Bengali Note: সব component একটি common data store ব্যবহার করে।

---

4. 🏛️ Hierarchical Style

- System split into **subsystems**
- Limited interaction between layers

Examples:

- **Interpreter**, **Layers**

Bengali Note: System স্তরে বিভক্ত — প্রতিটি স্তর নির্দিষ্ট কাজ করে।

---

5. 📞 Call and Return Style

- Caller waits for **response**
- Common in **procedural** or **non-threaded OOP**

Examples:

- **Master-Slave**, **Layers**, non-threaded **OOP**

Bengali Note: Caller function result না পাওয়া পর্যন্ত অপেক্ষা করে — sequential execution হয়।

🧠 Strategic Insight

Architectural styles help:

- Classify systems by **control flow**, **data flow**, and **component interaction**
- Map patterns to broader system **design philosophies**

Bengali Reframe: Architectural style হলো system design এর রূপরেখা — যেখানে pattern গুলো style অনুযায়ী শ্রেণিবদ্ধ হয়।

🧠 Choosing a Style or Pattern — Short Note

🔍 Decision Factors

Requirement	Description	Pattern Fit
Maintainability	Ease of adding/changing components (e.g., filters, input format)	✅ Pipe-Filter (easy to add filters), ❌ Pipe-Filter (hard to change input format)
Reusability	Can components be reused elsewhere?	✅ Pipe-Filter (uniform data format)
Performance	Fast response, efficient memory usage	✅ Pipe-Filter & Event-Bus (parallelism), ❌ Event-Bus (startup complexity, data transformation overhead)
Explicitness	Can user get feedback per stage?	❌ Pipe-Filter (no stage-wise feedback)
Fault Tolerance	Can system recover from component failure?	✅ Master-Slave (redundant slaves, strategy-based recovery)

Bengali Note:

- Maintainability: নতুন filter যোগ করা সহজ — কিন্তু input format বদলানো কঠিন
- Reusability: uniform format থাকলে component অন্য system এ ব্যবহার করা যায়
- Performance: parallelism থাকলে দ্রুত কাজ হয় — কিন্তু Event-Bus জটিল
- Explicitness: Pipe-Filter এ প্রতিটি ধাপে feedback দেওয়া যায় না
- Fault tolerance: Master-Slave pattern redundancy দিয়ে সমস্যা মোকাবেলা করে

⚙️ Implementation Matters

- **Threads vs Processes**: Same pattern may behave differently depending on concurrency model

- **Machine Communication Speed**: Affects distributed performance
- **Processor Capacity**: Influences computation vs communication balance

Bengali Note: Pattern ভালো হলেও implementation ঠিক না হলে performance কমে যেতে পারে।

🧠 Strategic Insight

There's **no one-size-fits-all** pattern.

Choose based on:

- **System priorities**
- **Runtime constraints**
- **Scalability and fault tolerance needs**

Bengali Reframe: Pattern নির্বাচন করতে হলে system এর প্রয়োজনীয়তা, performance, এবং future scalability বিবেচনা করতে হয়।Here's a concise, Bengali-annotated discussion-style response to all three questions — perfect for exam prep, peer dialogue, or architectural reflection:

🧠 Discussion Questions — KWIC & Architectural Patterns

a. Architectural Patterns as Combinations of Tactics

Yes — architectural patterns are often **built from tactics**, which are strategic decisions to improve system quality.

Pattern	Tactics Used	Bengali Note
Pipe-Filter	Modularity, Reusability, Concurrency	প্রতিটি filter আলাদা — সহজে যোগ/পরিবর্তন করা যায়
Client-Server	Separation of concerns, Scalability	Client ও Server আলাদা — স্কেল করা সহজ
Master-Slave	Fault tolerance, Parallelism	Slave fail করলে master অন্য slave ব্যবহার করতে পারে

Pattern	Tactics Used	Bengali Note
Event-Bus	Loose coupling, Asynchronous communication	Event দিয়ে component গুলো আলাদা থাকে
Layers	Encapsulation, Maintainability	প্রতিটি layer নিজস্ব কাজ করে — পরিবর্তন সহজ

Strategic Insight: Patterns = tactical bundles for solving recurring architectural problems.

---

b. Pattern Drift Over Time — Should It Be Prevented?

Yes — **pattern drift** can lead to:

- Loss of original design intent
- Increased complexity
- Reduced maintainability

How to prevent it:

- **Document rationale**: Why was the pattern chosen?
- **Code reviews**: Check for adherence to pattern structure
- **Architecture diagrams**: Keep updated visual references
- **Refactoring routines**: Periodically realign with original pattern

Bengali Note: Pattern কেন ব্যবহার করা হয়েছে তা ভুলে গেলে design ভেঙে পড়ে — তাই rationale ও diagram সংরক্ষণ জরুরি।

---

c. Crosscutting Concerns in Architectural Patterns?

Not always — but **some architectural patterns** do involve crosscutting concerns:

Concern	Design Pattern	Architectural Pattern
Logging, Security, Transactions	✅ Often handled via AOP	⚠️ Sometimes embedded in Event-Bus or Broker
Communication protocols	❌ Not typical in design patterns	✅ Central in Client-Server, Event-Bus

Concern	Design Pattern	Architectural Pattern
Fault tolerance	✅ Strategy pattern may help	✅ Master-Slave pattern directly addresses it

Conclusion:

- **Design patterns** = finer-grained, often crosscutting
- **Architectural patterns** = coarse-grained, system-wide — harder to modularize via AOP

Bengali Reframe: Design pattern ছোট concern handle করে — Architectural pattern পুরো system এর গঠন ঠিক করে।

Let me know if you want a Bengali-annotated cheat sheet mapping tactics to patterns, or a contest-style simulation showing how pattern drift affects maintainability. We can scaffold it together.