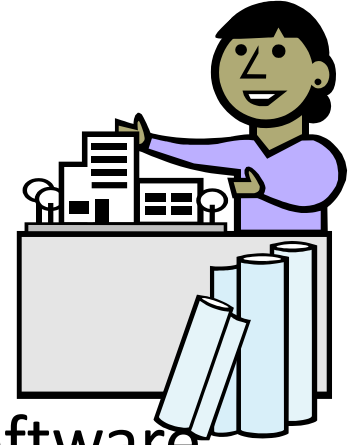
Software Architecture

Working session

Groups of Three students

For each Design pattern, propose a current application or a type of application or module of an application that could be architect with the use of the pattern.

Software Architecture



- Software architecture refers to the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures.
- *An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.*

Wikipedia

Architectural Patterns

- The fundamental problem to be solved with a **large system** is how to break it into chunks manageable for human programmers to understand, implement, and maintain.
- Large-scale patterns for this purpose are called *architectural patterns*.

Common Architecture Patterns

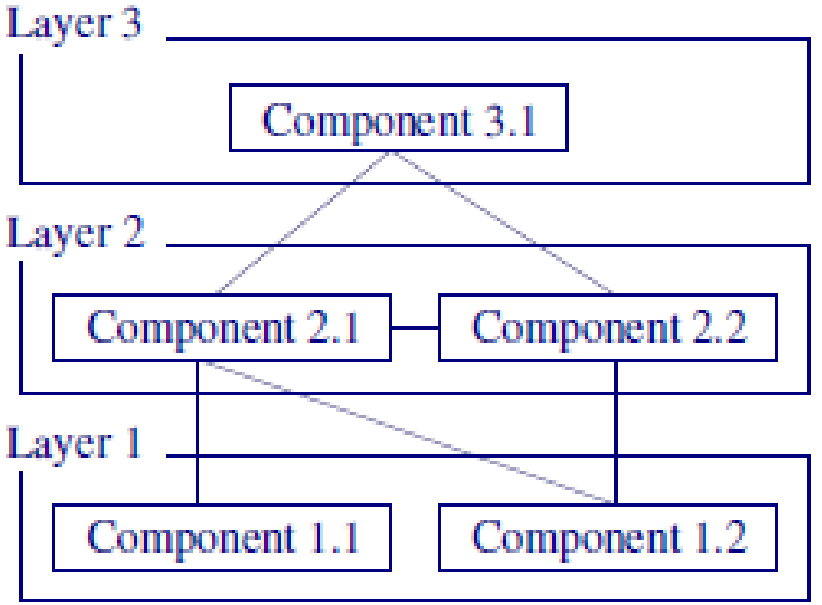
1. **Layered pattern**
2. **Client-server pattern**
3. **Master-slave pattern**
4. **Pipe-filter pattern**
5. **Broker pattern**
6. **Peer-to-peer pattern**
7. **Event-bus pattern**
8. **Model-view-controller pattern**
9. **Blackboard pattern**
10. **Interpreter pattern**

1. Layered pattern

- This pattern is also known as **n-tier architecture pattern**.
- It can be used to structure programs that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction.
- Each layer provides services to the next higher layer.

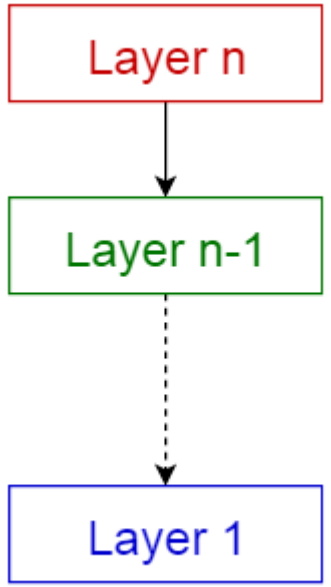
Usage

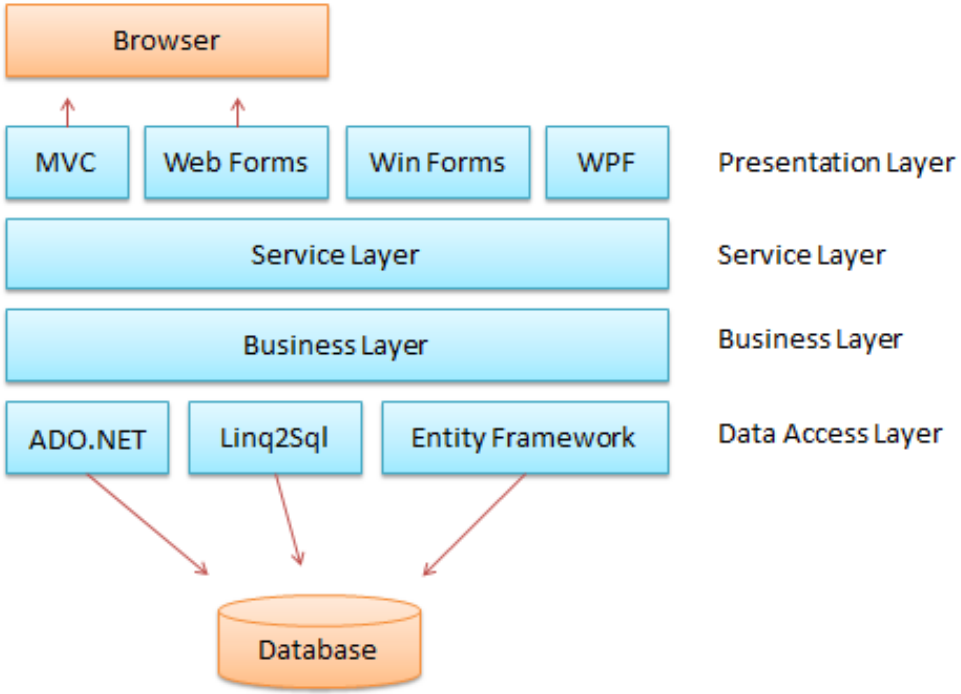
- General desktop applications.
- E-commerce web applications.

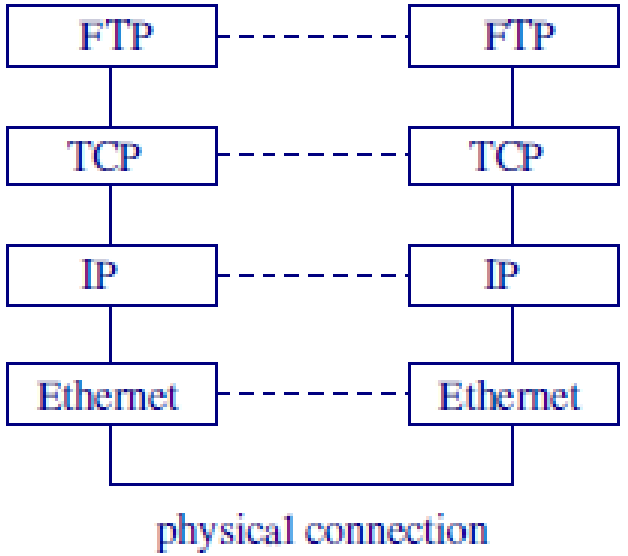


Common layers of a general information system

- **Presentation layer** (also known as **UI layer**)
- **Application layer** (also known as **service layer**)
- **Business logic layer** (also known as **domain layer**)
- **Data access layer** (also known as **persistence layer**)

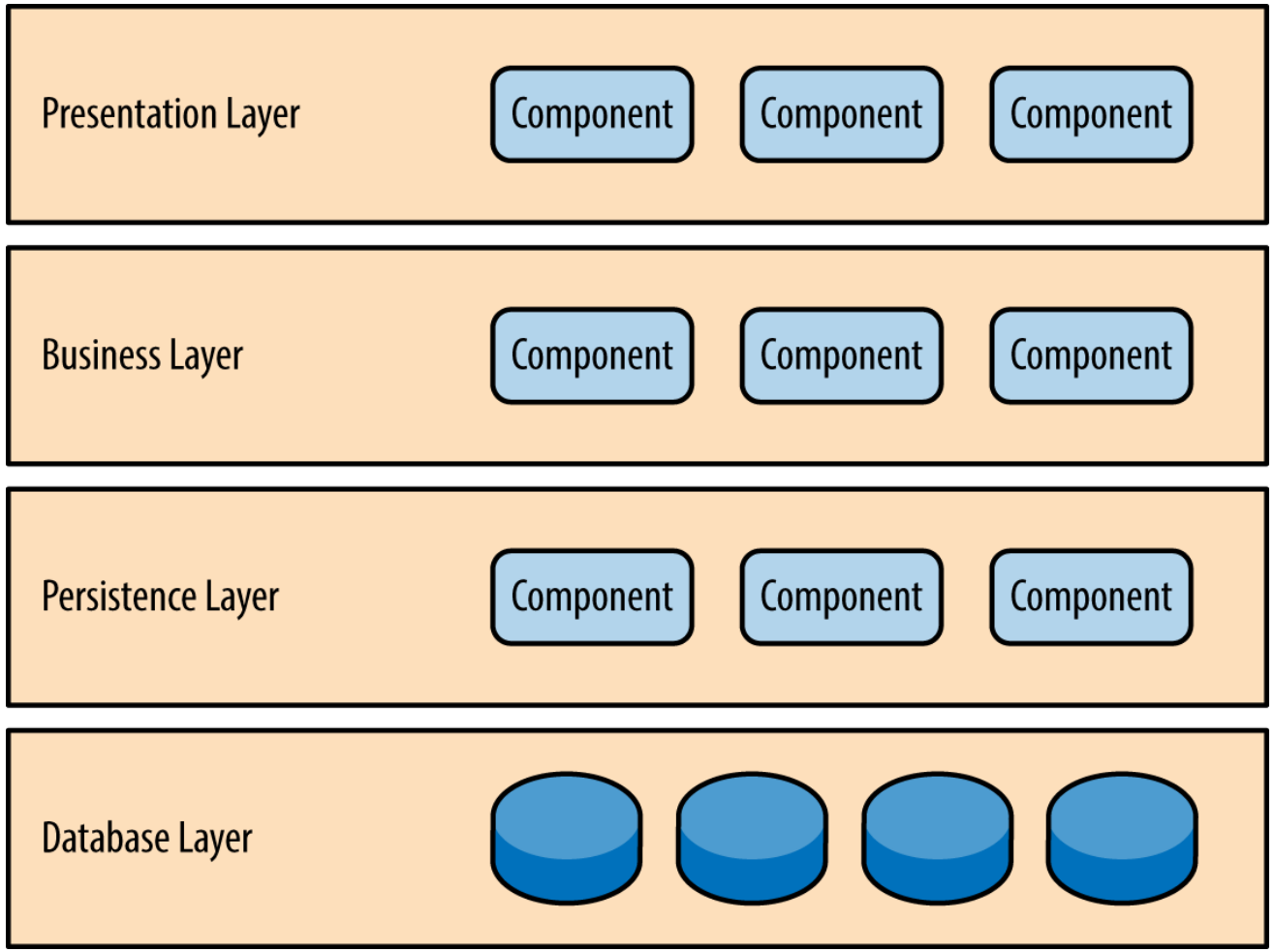




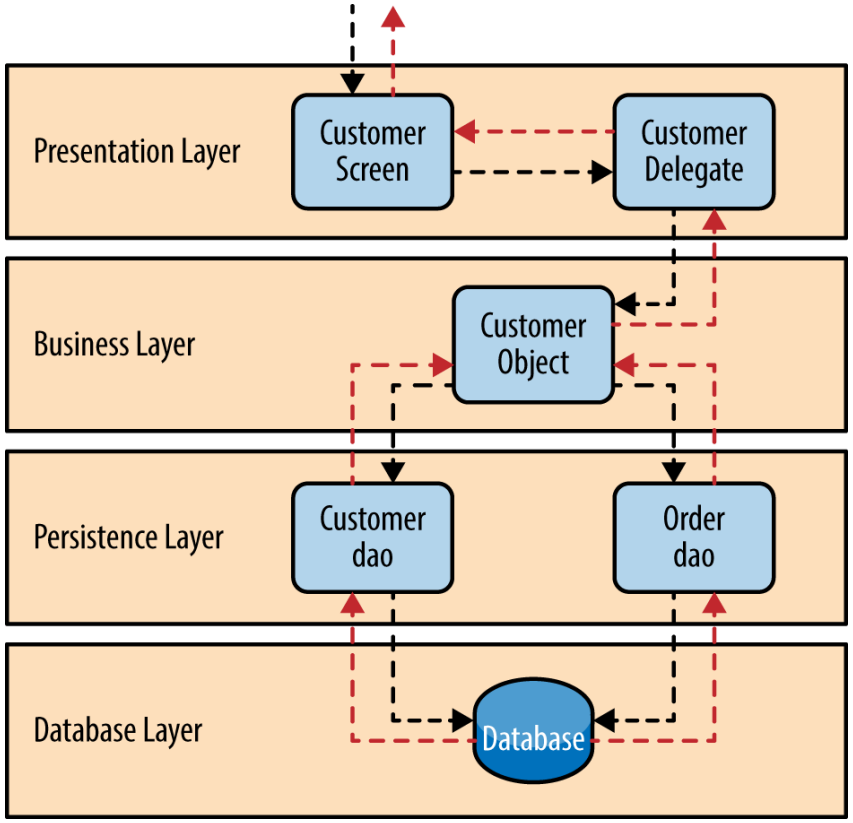


Layered pattern

- Different levels of abstraction
- Requests go down, notifictions go back up
- Possible to define stable interaces between layers
- May add or change layers over time



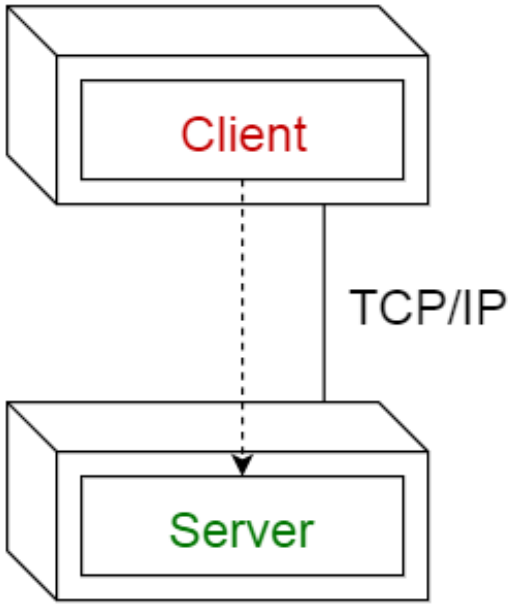
Layered pattern



Solution

- Start with the lowest later
- Build layers on top
- Use same level of abstraction within each layer
- Componenets cannot spread over two layers
- Keey lower layers lean
- Specify interfaces for each layer
- Handle errors at the lowest level possible

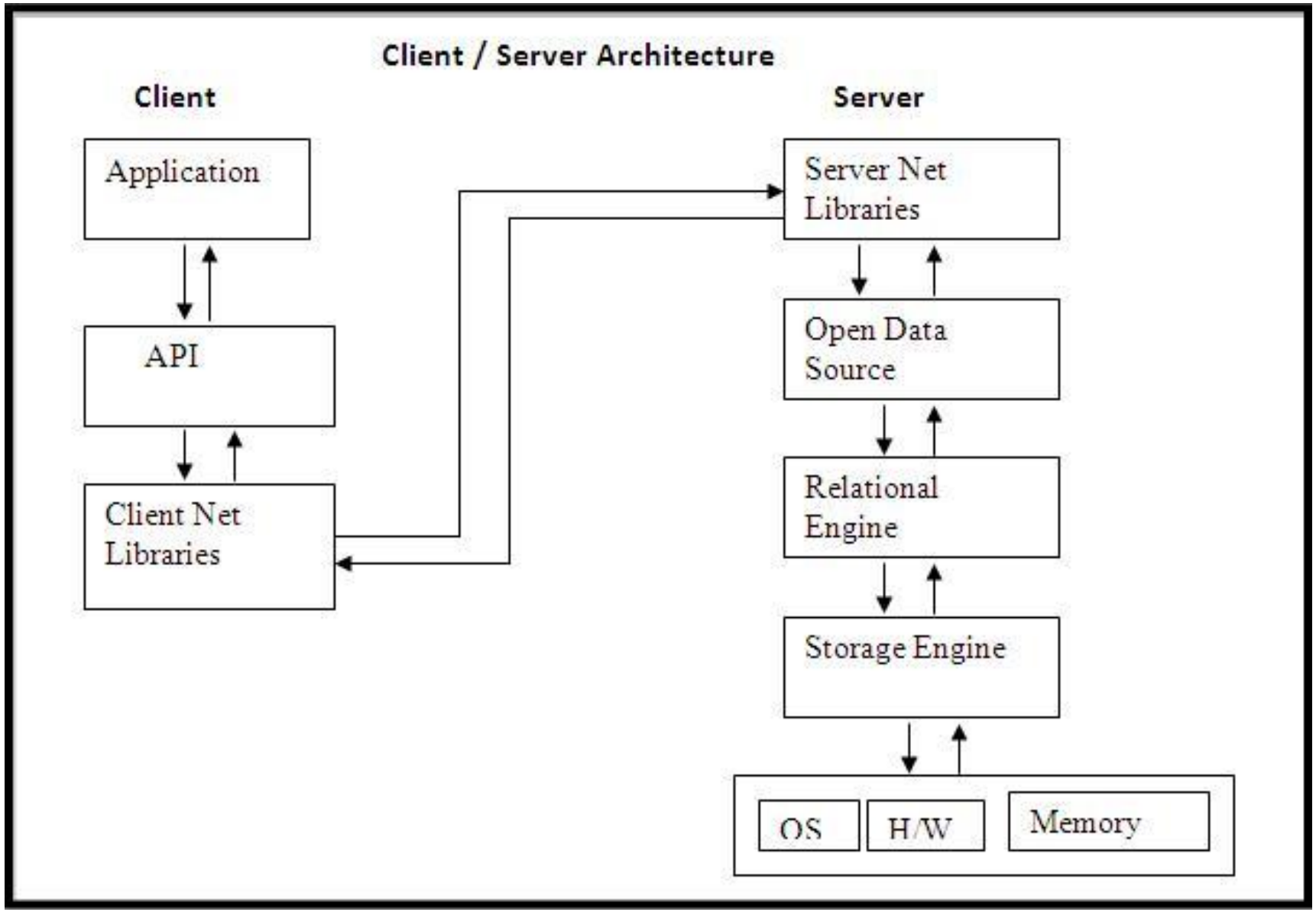
2. Client-server pattern



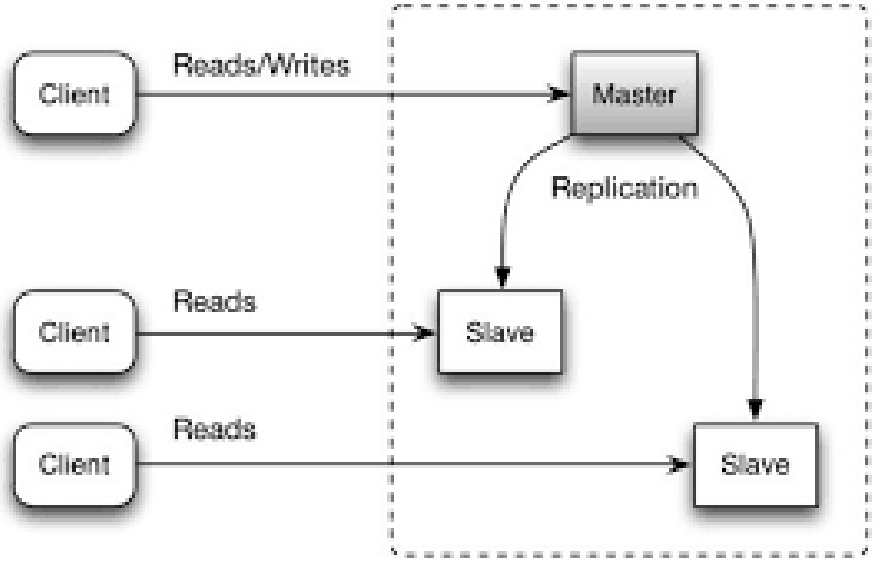
- This pattern consists of two parties;
  - a **server** and
  - multiple **clients**.
- The server component will provide services to multiple client components.
- Clients request services from the server and the server provides relevant services to those clients.
- The server continues to listen to client requests.

Usage

- Online applications such as email, *document sharing* and ***banking***.


Client / Server Architecture
Client
Server
Application
API
Client Net Libraries
Server Net Libraries
Open Data Source
Relational Engine
Storage Engine
OS
H/W
Memory

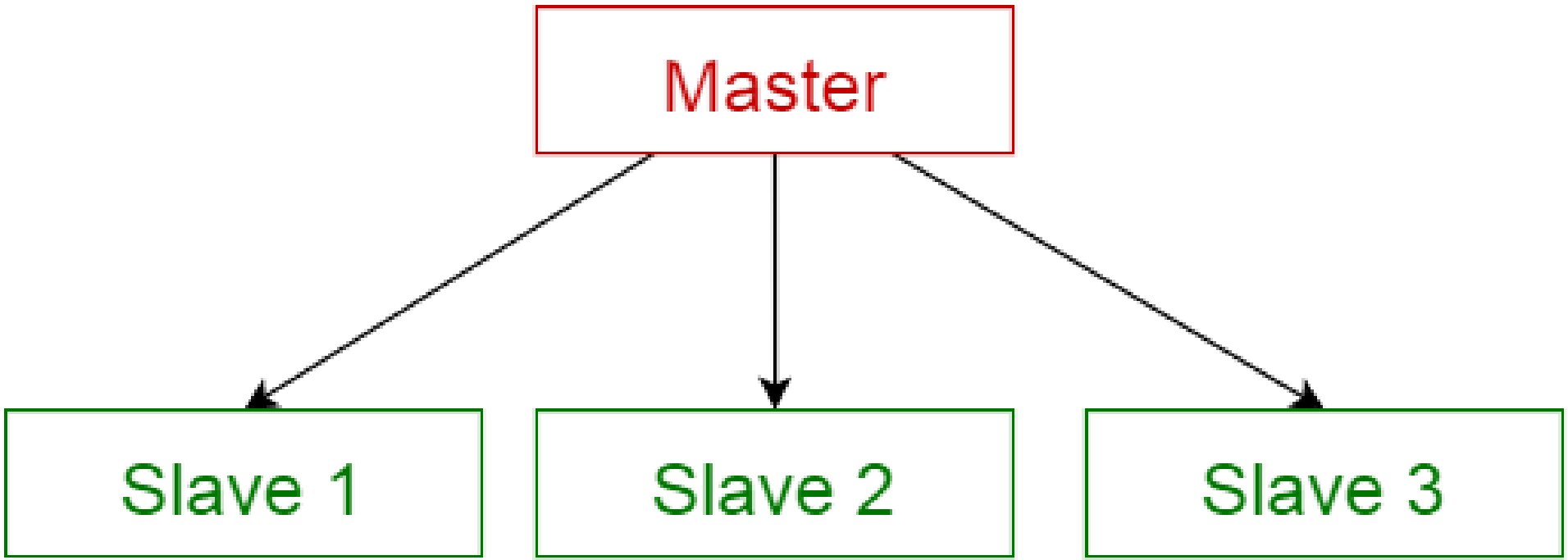
3. Master-slave pattern



- This pattern consists of two parties;
  - **master** and
  - **slaves**.
- The master component distributes the work among identical slave components, and computes a final result from the results which the slaves return.

Usage

- In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
- Peripherals connected to a bus in a computer system (master and slave drives).


Master
Slave 1
Slave 2
Slave 3

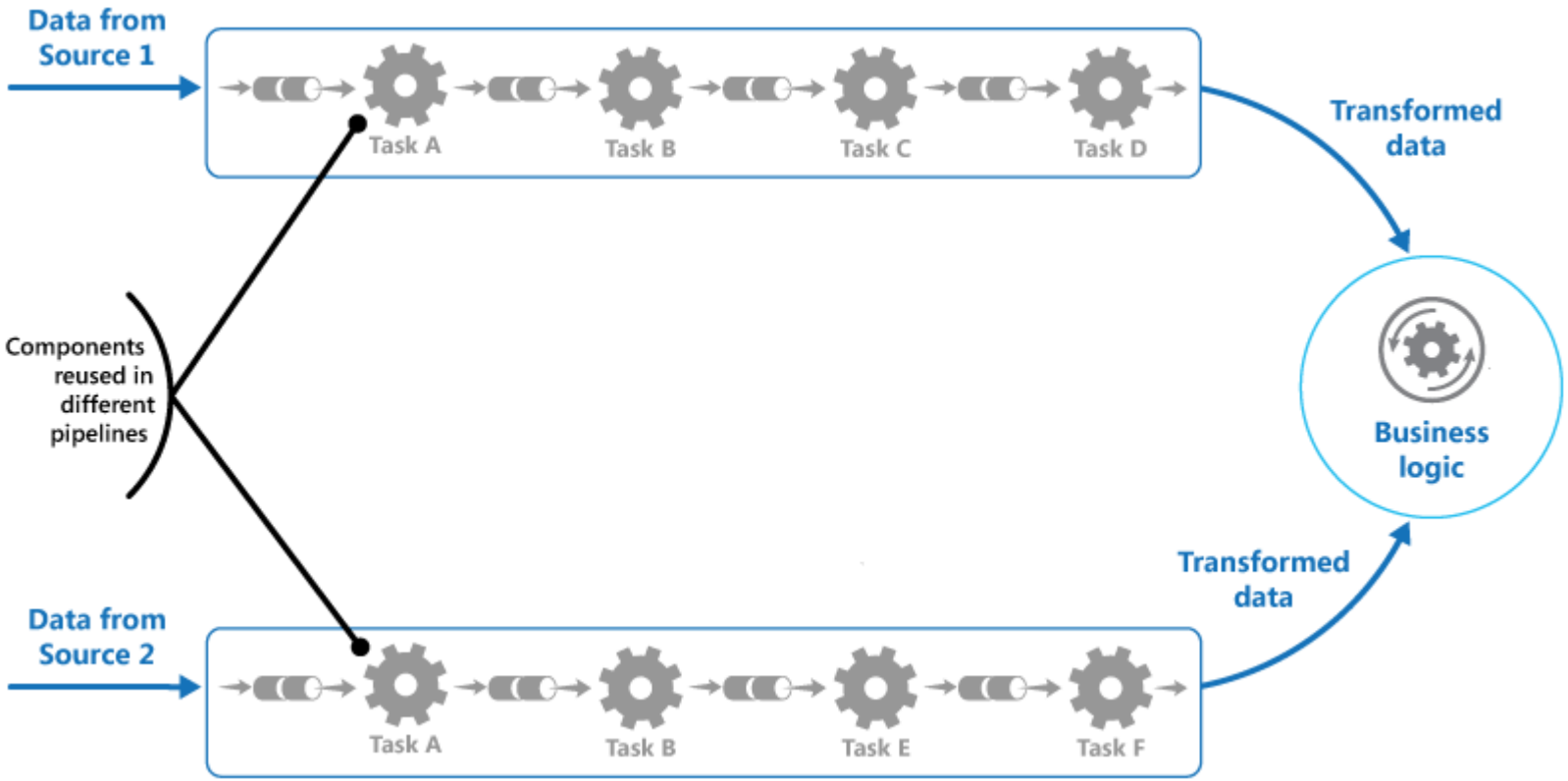
4. Pipe-filter pattern

- This pattern can be used to structure systems which produce and process a stream of data.
- Each processing step is enclosed within a **filter** component.
- Data to be processed is passed through **pipes**.
- These pipes can be used for buffering or for synchronization purposes.

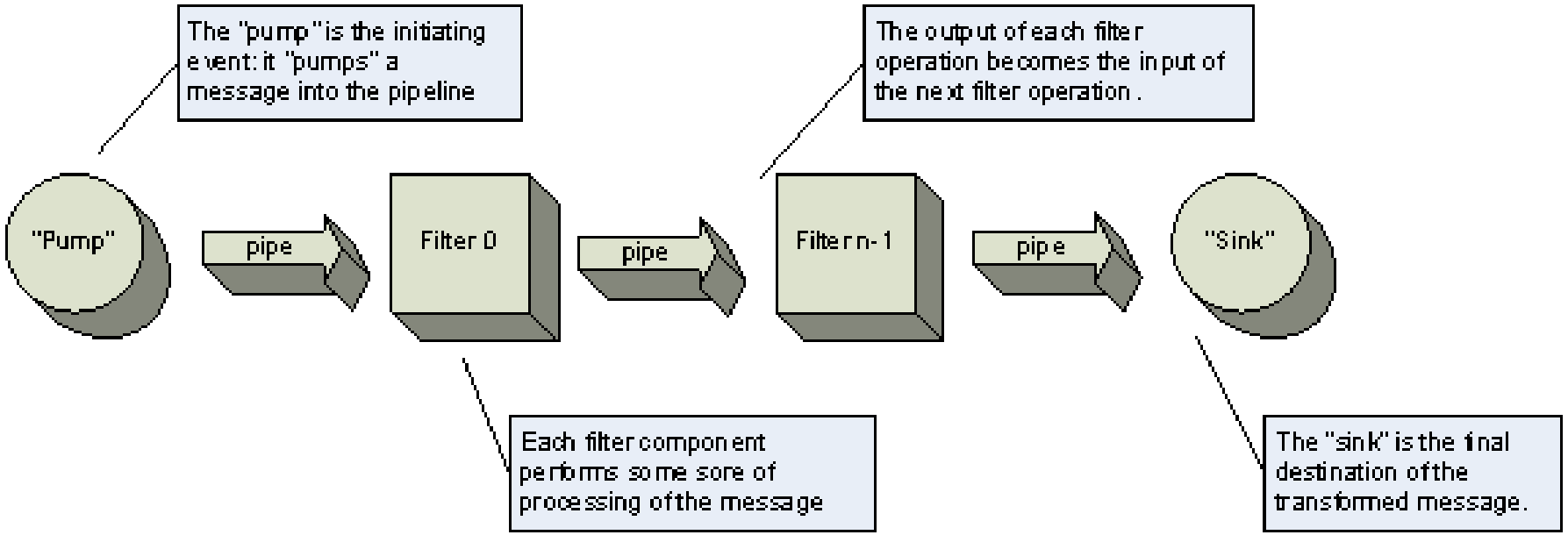
Usage

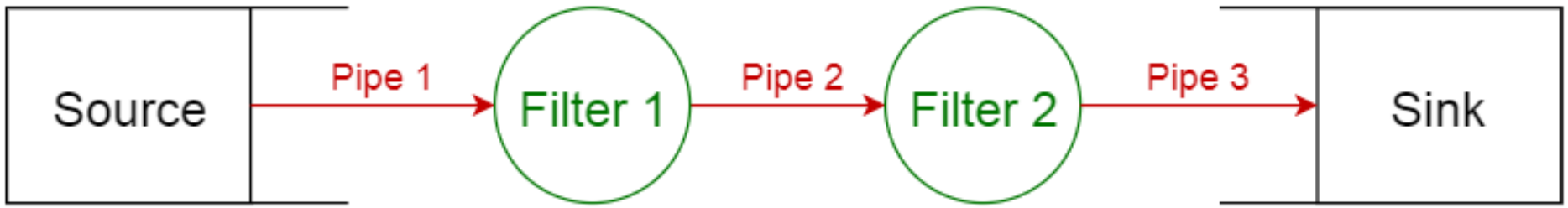
- **Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.**

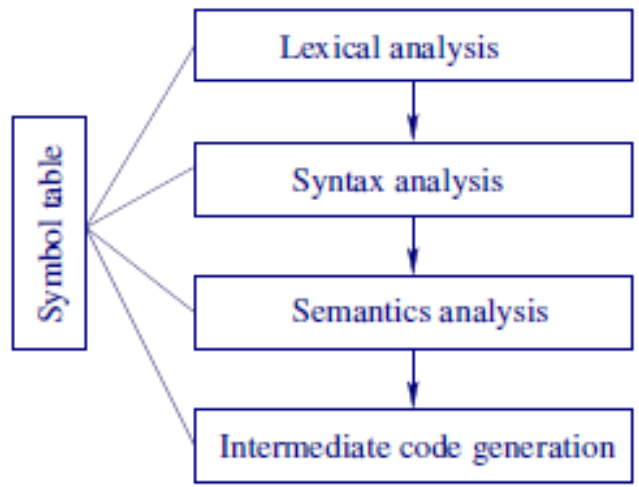
- Pipes eliminate need for intermediate files
- Can replace filters easily
- Can achive different effects through recombination
- İf data stream format is standard, filters are developed independently
- Parallelization possible



- Data Stream processing may be subdivided into stages
- May recombine stages
- Non adjacent stages do not share information
- May desire different stages to be on different processors
- Standardized data structure between stages




Source
Pipe 1
Filter 1
Pipe 2
Filter 2
Pipe 3
Sink

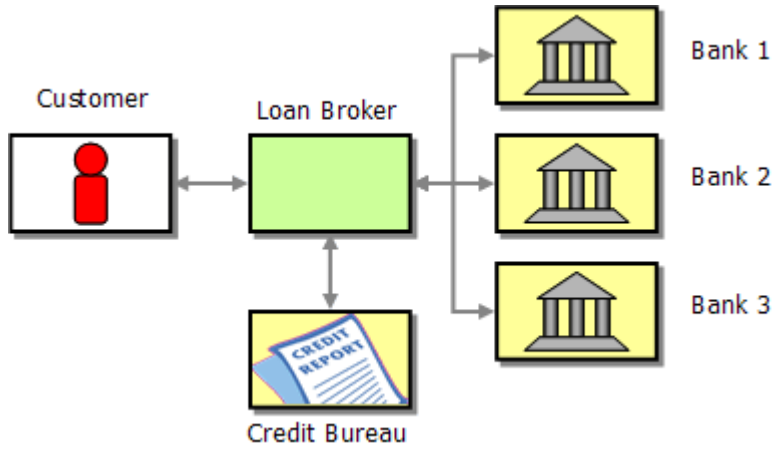

Lexical analysis
Syntax analysis
Semantics analysis
Intermediate code generation
Symbol table

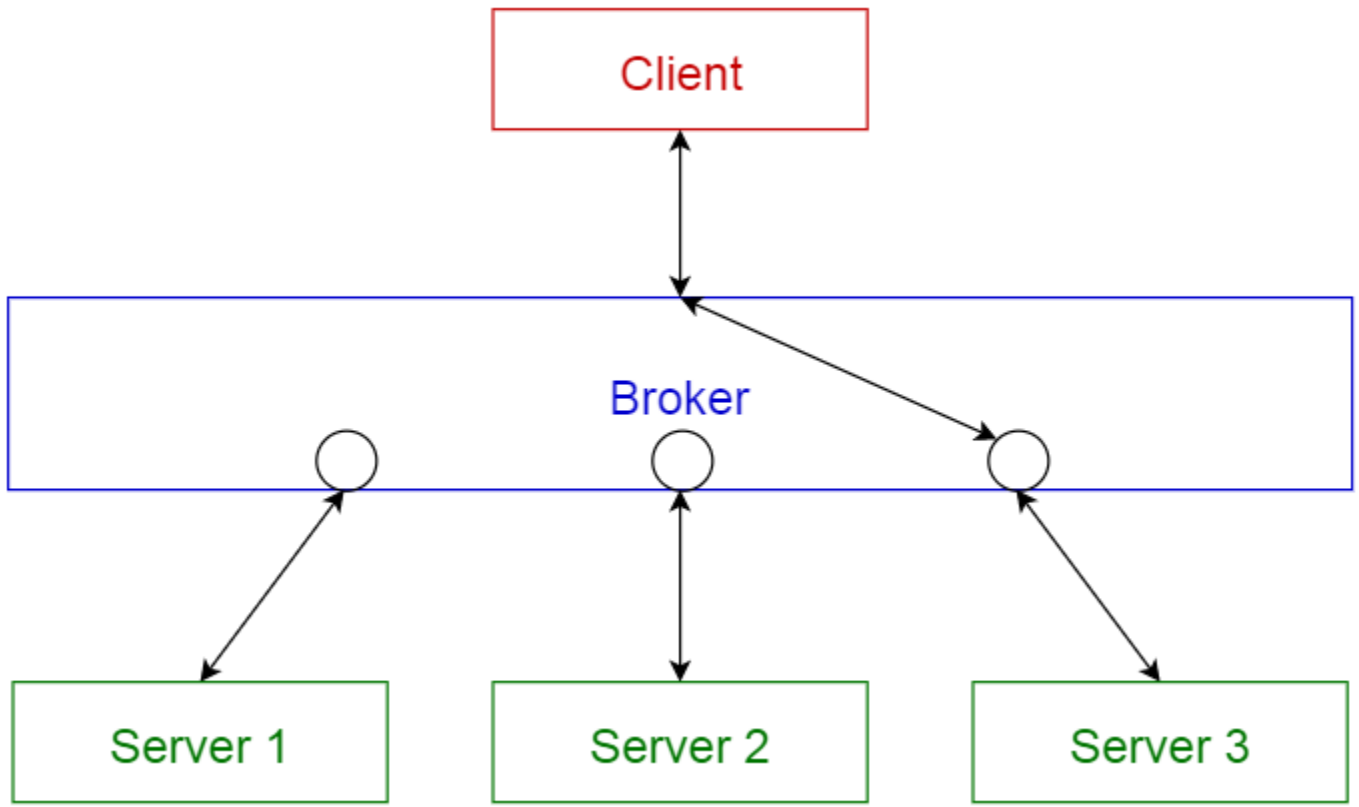
5. Broker pattern

- This pattern is used to structure distributed systems with *decoupled* components.
- These components can interact with each other by remote service invocations.
- A **broker** component is responsible for the coordination of communication among **components**.
- Servers publish their capabilities (services and characteristics) to a broker.
- Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

Usage

- Message broker software such as
- Apache ActiveMQ, Apache Kafka, RabbitMQ and JBoss Messaging.



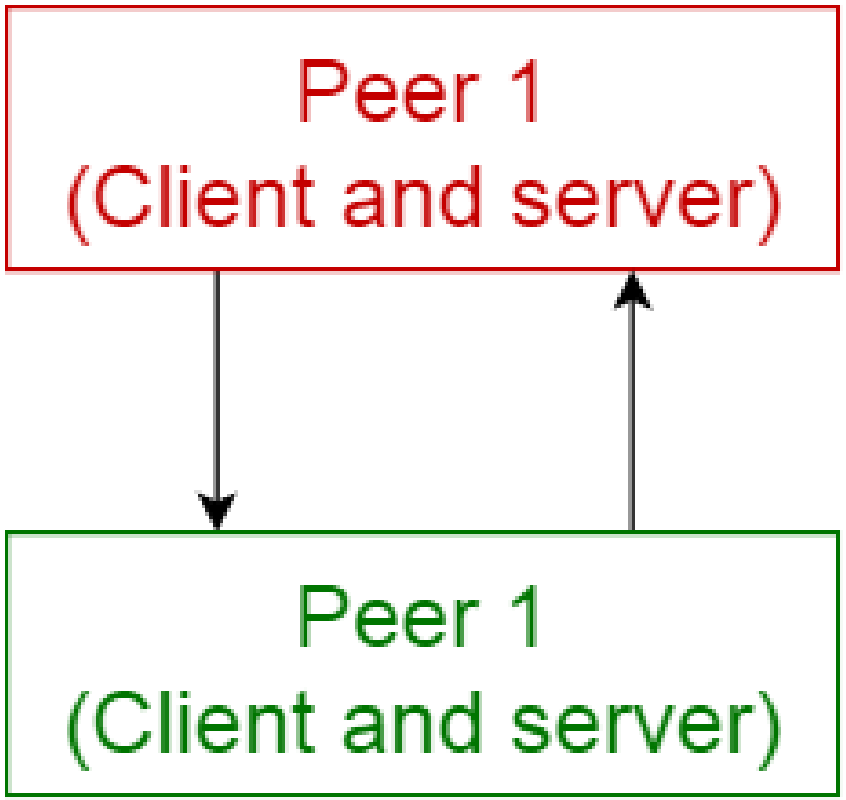

Client
Broker
Server 1
Server 2
Server 3

6. Peer-to-peer pattern

- In this pattern, individual components are known as **peers**.
- Peers may function both as a **client**, requesting services from other peers, and as a **server**, providing services to other peers.
- A peer may act as a client or as a server or as both, and it can change its role dynamically with time.

Usage

- File-sharing networks such as **Gnutella** and **G2**)
- Multimedia protocols such as **P2PTV** and **PDTP**.
- Proprietary multimedia applications such as **Spotify**.
-

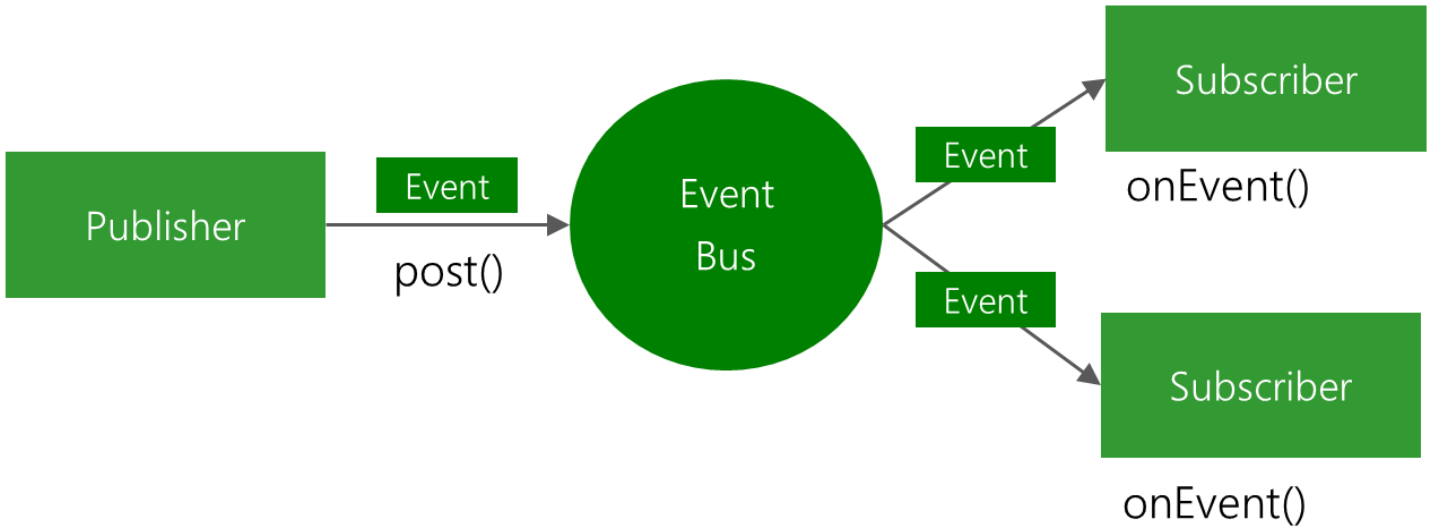

Peer 1
(Client and server)
Peer 1
(Client and server)

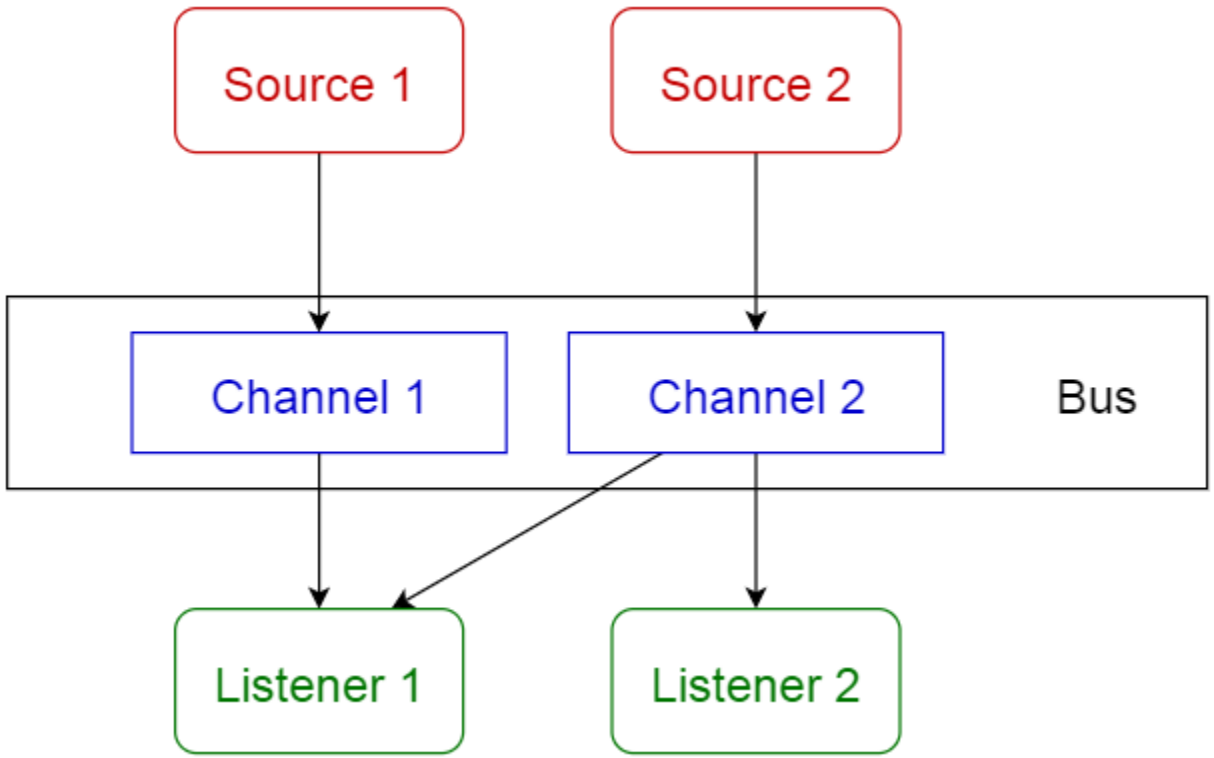
7. Event-bus pattern

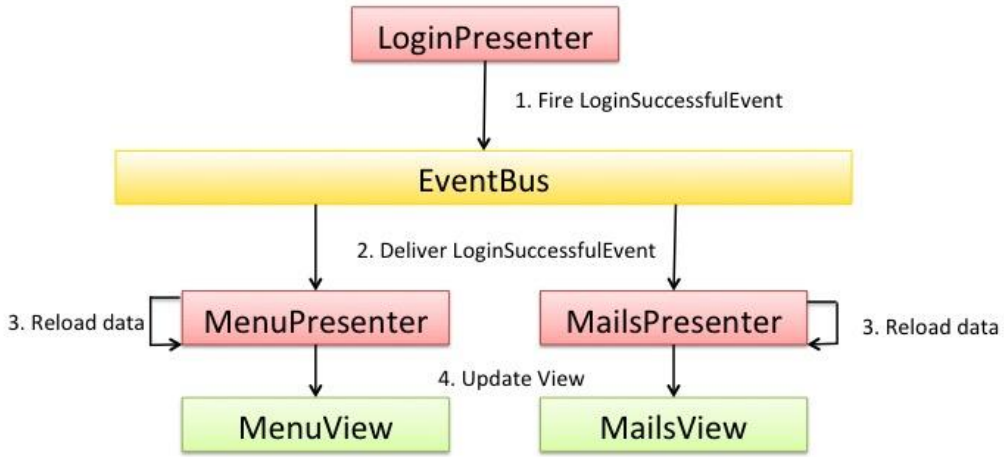
- This pattern primarily deals with events and has 4 major components;
  - **event source**,
  - **event listener**,
  - **channel** and
  - **event bus**.
- Sources publish messages to particular channels on an event bus.
- Listeners subscribe to particular channels.
- Listeners are notified of messages that are published to a channel to which they have subscribed before.

Usage

- Android development
- Notification services




Source 1
Source 2
Channel 1
Channel 2
Bus
Listener 1
Listener 2

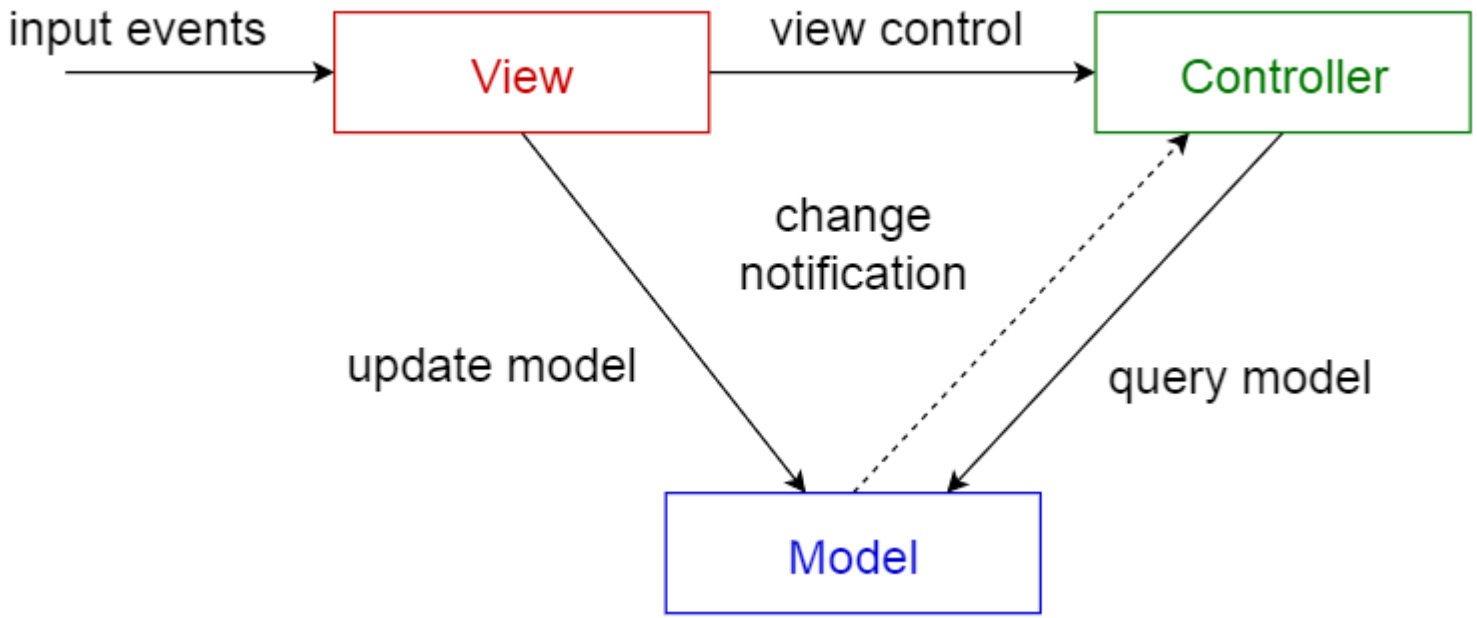

LoginPresenter
1. Fire LoginSuccessfulEvent
EventBus
2. Deliver LoginSuccessfulEvent
3. Reload data
MenuPresenter
MailsPresenter
3. Reload data
4. Update View
MenuView
MailsView

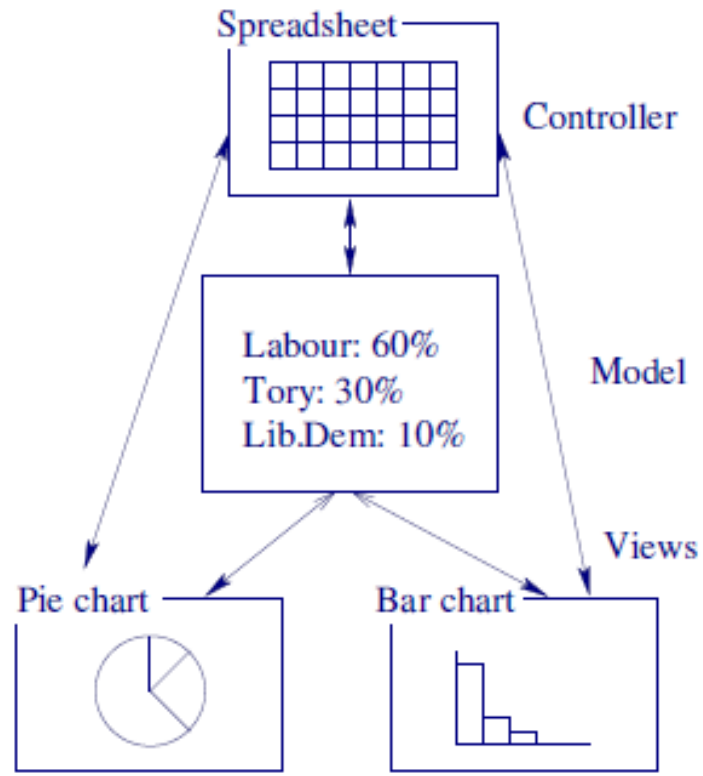
8. Model-view-controller pattern

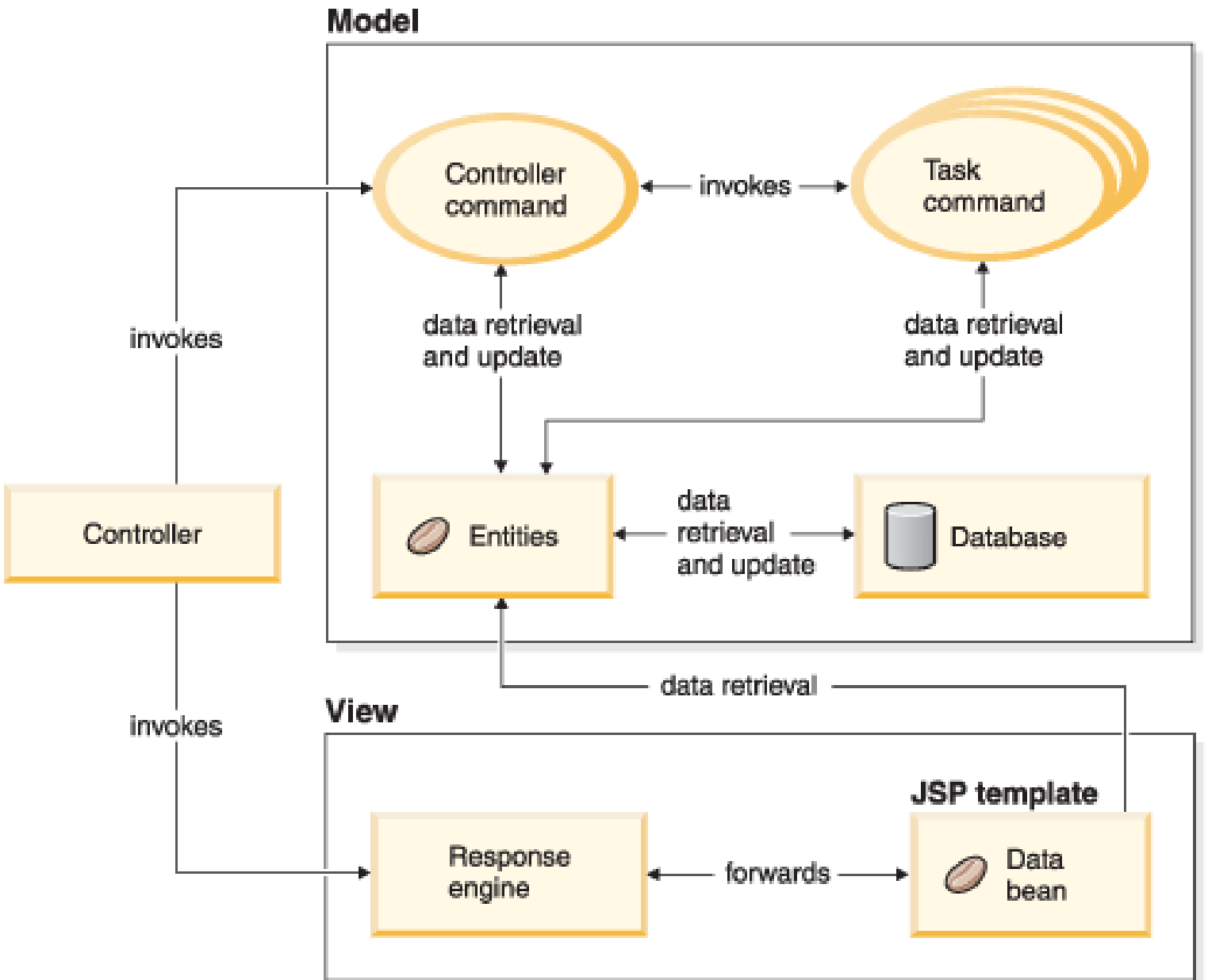
- This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,
- **model** — contains the core functionality and data
- **view** — displays the information to the user (more than one view may be defined)
- **controller** — handles the input from the user
- This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as **Django** and **Rails**.


input events
View
view control
Controller
change
notification
update model
query model
Model


Spreadsheet
Controller
Labour: 60%
Tory: 30%
Lib.Dem: 10%
Model
Views
Pie chart
Bar chart


Model
Controller command
invokes
Task command
data retrieval and update
data retrieval and update
invokes
Controller
Entities
data retrieval and update
Database
data retrieval
invokes
View
JSP template
Response engine
forwards
Data bean

9. Blackboard pattern

- This pattern is useful for problems for which no deterministic solution strategies are known. The blackboard pattern consists of 3 main components.
  - **blackboard** — a structured global memory containing objects from the solution space
  - **knowledge source** — specialized modules with their own representation
  - **control component** — selects, configures and executes modules.
- All the components have access to the blackboard.
- Components may produce new data objects that are added to the blackboard.
- Components look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source.

Usage

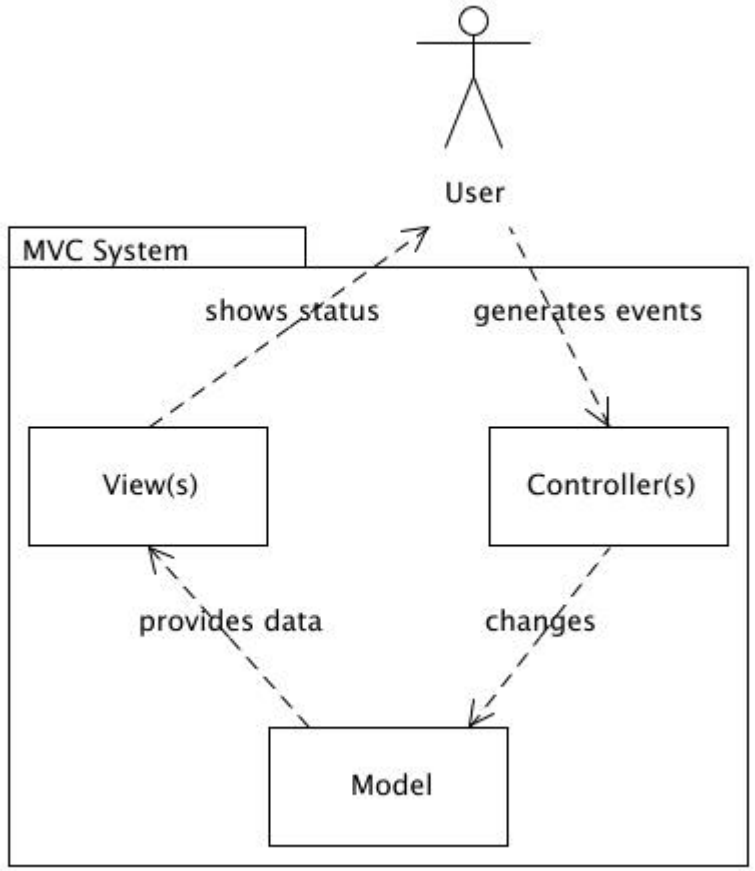
- Speech recognition
- Vehicle identification and tracking
- Protein structure identification
- Sonar signals interpretation.

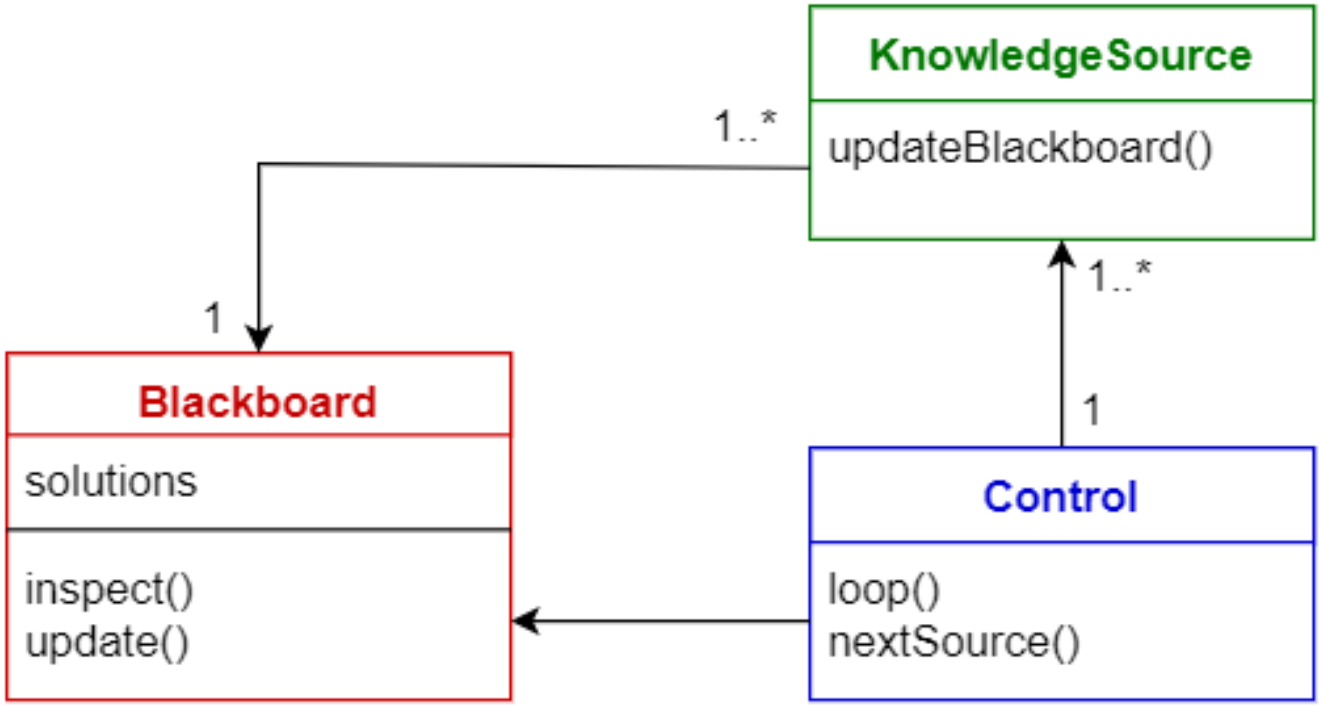
9. Blackboard pattern

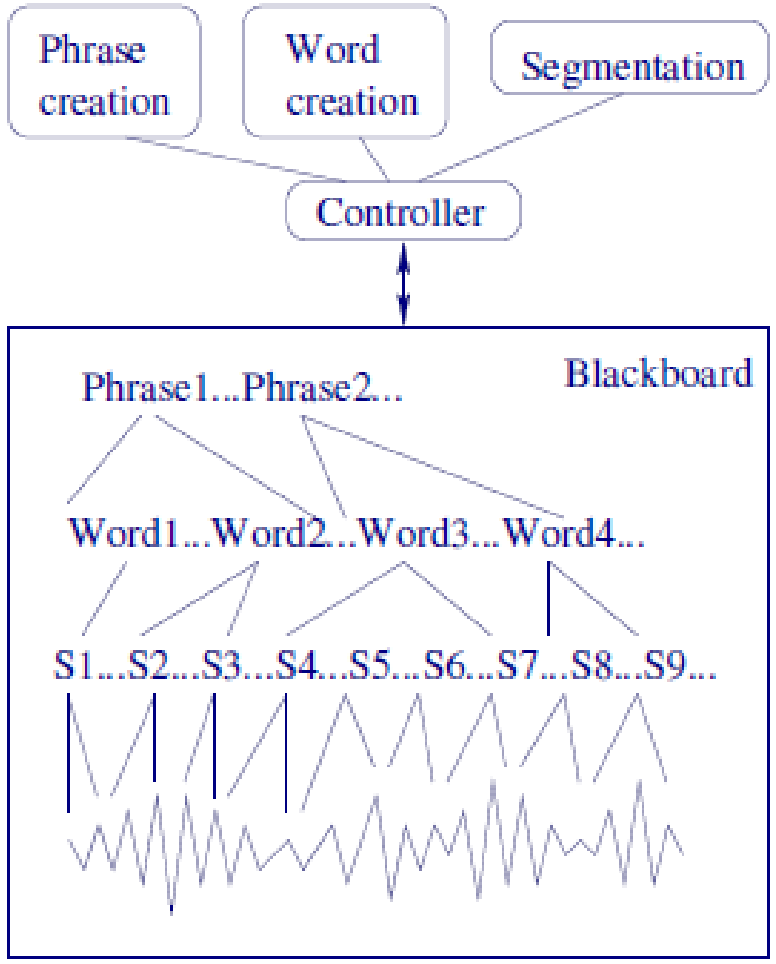
- Problem solvers work independently on part of the problem
- Share common data structure
- Central controller manages access to the blackboard
- Blackboard may be strcured in levels of abtraction to allow work at different levels
- Blackbpard contain original input and partial solutions

9. Blackboard pattern

- Difficult to test
- Difficult to guarantee an optimal solution
- Control strategy often heuristic
- May be computationally expensive
- Parallelism possible




KnowledgeSource
updateBlackboard()
1..*
1
1..*
1
Blackboard
solutions
inspect()
update()
Control
loop()
nextSource()


Phrase creation
Word creation
Segmentation
Controller
Blackboard
Phrase1...Phrase2...
Word1...Word2...Word3...Word4...
S1...S2...S3...S4...S5...S6...S7...S8...S9...

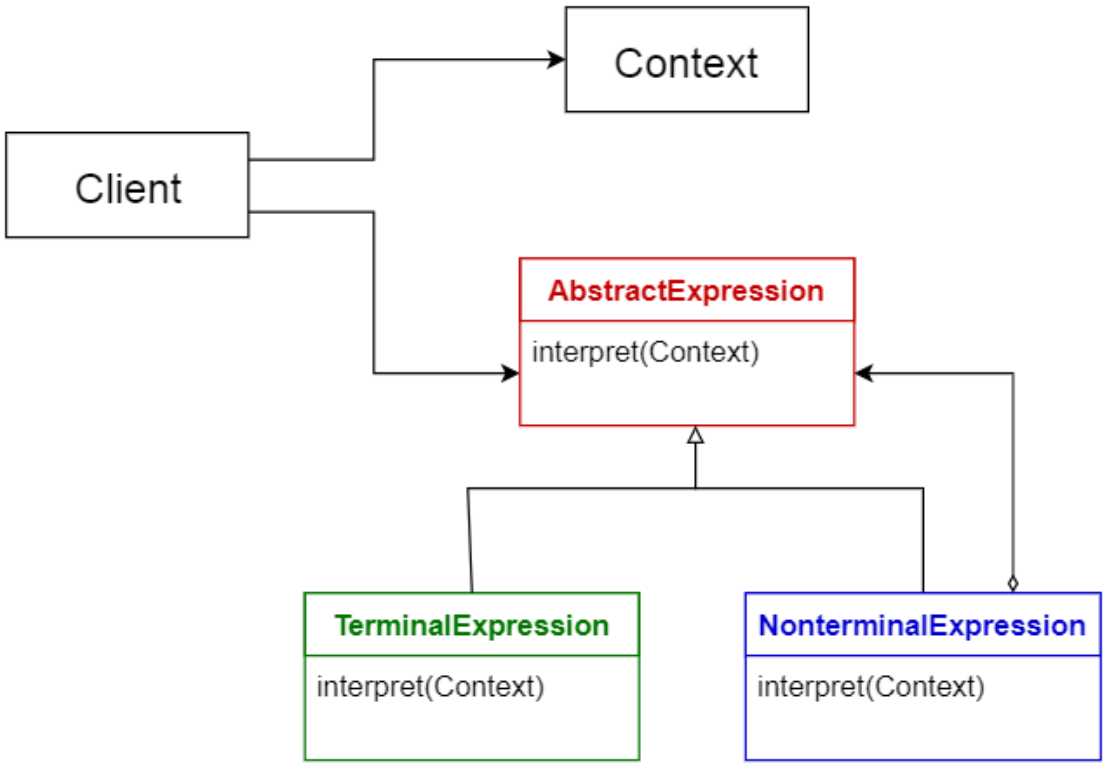
10. Interpreter pattern

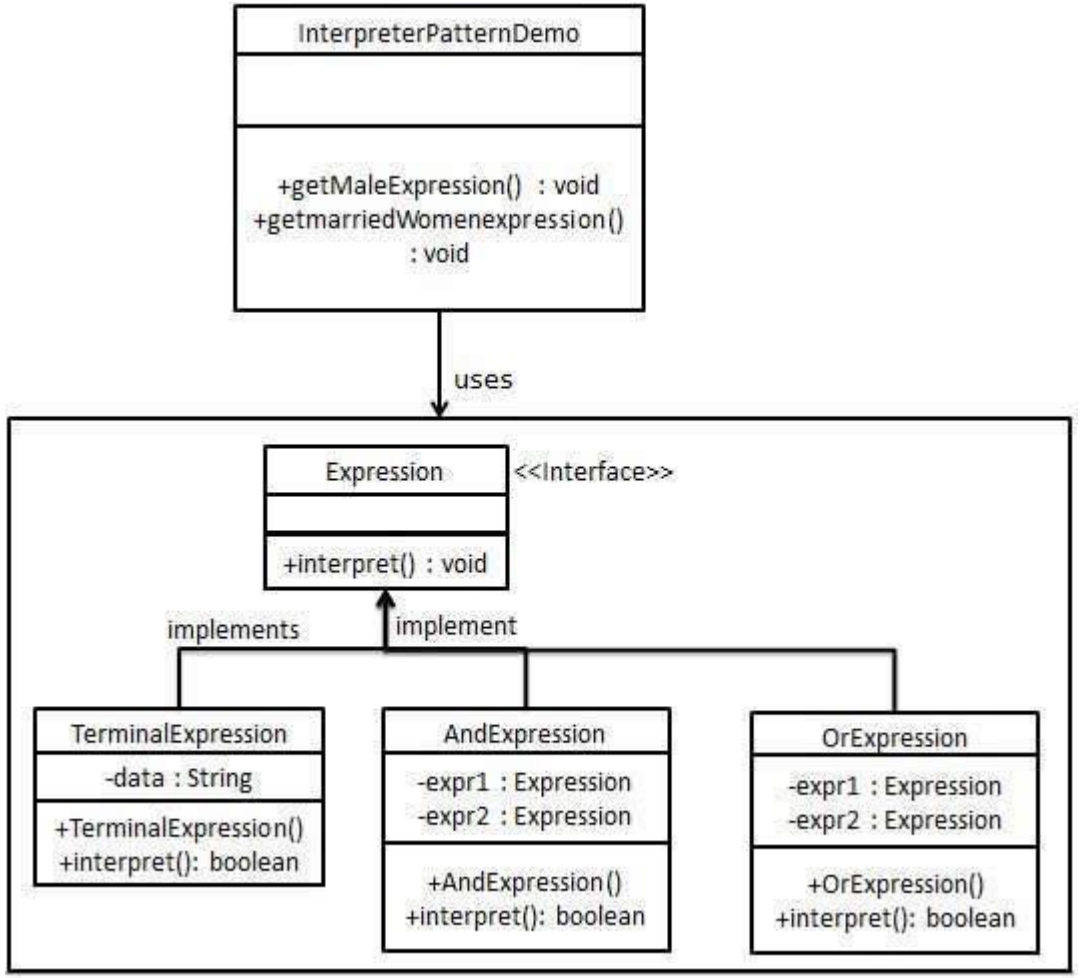
- This pattern is used for designing a component that interprets programs written in a dedicated language.
- It mainly specifies how to evaluate lines of programs, known as sentences or expressions written in a particular language.
- The basic idea is to have a class for each symbol of the language.

Usage

- Database query languages such as SQL.
- Languages used to describe communication protocols.

9. Blackboard pattern





Name	Advantages	Disadvantages
Layered	A lower layer can be used by different higher layers. Layers make standardization easier as we can clearly define levels. Changes can be made within the layer without affecting other layers.	Not universally applicable. Certain layers may have to be skipped in certain situations.
Client-server	Good to model a set of services where clients can request them.	Requests are typically handled in separate threads on the server. Inter-process communication causes overhead as different clients have different representations.
Master-slave	Accuracy - The execution of a service is delegated to different slaves, with different implementations.	The slaves are isolated: there is no shared state. The latency in the master-slave communication can be an issue, for instance in real-time systems. This pattern can only be applied to a problem that can be decomposed.
Pipe-filter	Exhibits concurrent processing. When input and output consist of streams, and filters start computing when they receive data. Easy to add filters. The system can be extended easily. Filters are reusable. Can build different pipelines by recombining a given set of filters	Efficiency is limited by the slowest filter process. Data-transformation overhead when moving from one filter to another.
Broker	Allows dynamic change, addition, deletion and relocation of objects, and it makes distribution transparent to the developer.	Requires standardization of service descriptions.
Peer-to-peer	Supports decentralized computing. Highly robust in the failure of any given node. Highly scalable in terms of resources and computing power.	There is no guarantee about quality of service, as nodes cooperate voluntarily. Security is difficult to be guaranteed. Performance depends on the number of nodes.
Event-bus	New publishers, subscribers and connections can be added easily. Effective for highly distributed applications.	Scalability may be a problem, as all messages travel through the same event bus
Model-view-controller	Makes it easy to have multiple views of the same model, which can be connected and disconnected at run-time.	Increases complexity. May lead to many unnecessary updates for user actions.
Blackboard	Easy to add new applications. Extending the structure of the data space is easy.	Modifying the structure of the data space is hard, as all applications are affected May need synchronization and access control.
Interpreter	Highly dynamic behavior is possible. Good for end user programmability. Enhances flexibility, because replacing an interpreted program is easy.	Because an interpreted language is generally slower than a compiled one, performance may be an issue.

- https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013