Software Design
&
Architecture

Dr. Cahit Karakuş

First degree in university

- A first-place degree in university represents the pinnacle of intellectual qualifications.
- Someone which has intellectual knowledge is a person who thinks mentally and rationally.
- Someone which has intellectual knowledge describes things based on reason, knowledge, and thought rather than emotion.
- Someone which has intellectual knowledge discusses based on knowledge and logic.
- Someone which has intellectual knowledge is a cultured, educated, and thinking person.
- Someone which has intellectual knowledge is a person who interested in fields such as art, literature, science, and philosophy; a conscious and questioning on social issues.
- A first-place degree in university represents the pinnacle of intellectual qualifications. Students earning a first-place degree must be able to see beyond the accepted concepts of the day. They must be able to discuss topics with highly intelligent and knowledgeable individuals without hesitation.

How important talent is!

- A talent is a skill, either innate or developed later in life, that allows a person to perform a specific task more easily, quickly, and successfully.
- You may not be particularly talented; the important thing is to focus on your work and act with discipline.

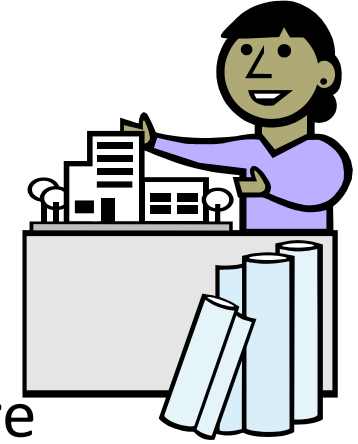
Course Introduction

- This course provides students with a solid foundation in software design and architecture. It covers fundamental design principles, architectural styles, design patterns, quality attributes, and modern practices in software engineering. Students will learn how to design, document, and evaluate software architectures through theory, case studies, and hands-on exercises.

Learning Outcomes

- - Understand the difference between software design and software architecture
  - Apply design principles such as modularity, abstraction, and SOLID
  - Model systems using UML diagrams
  - Recognize and apply common architectural styles and design patterns
  - Evaluate architectures with respect to quality attributes
  - Document and communicate architectural decisions
  - Apply modern architectural practices (Agile, DevOps, Cloud-Native)
  - Work on a project to design and present a complete software architecture

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. These structures are needed to reason about the software system.

- *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

Architecture - Design

"*Architecture* is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design."

(vs. design…)

"*Design* is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements."

Architecture Structures

- Architecture Structures (For Physical Structures):**The art of designing and constructing** buildings and other structures for people to live in, work in, or use for various purposes. Example: The architecture of Istanbul combines Byzantine and Ottoman styles.
- In Computer Science and Software: ***Architecture refers to the overall design of a system, software, or hardware, how its components are organized, and how they interact with each other.***
- Computer Science (Bilgisayar Bilimi), bilgisayarların nasıl çalıştığını, bilgiyi nasıl işlediğini ve problemleri çözmek için neleri kullanılabileceğini inceleyen bilim dalıdır. Yani yalnızca bilgisayar kullanmakla değil, bilgisayarların mantığını, dilini ve sınırlarını anlamakla ilgilenir.
- Example:
  - Computer architecture is the hardware structure (Physcial elements) of a computer.
  - Software architecture (the modular structure of software).
- Generally speaking: The order, structure, and design of something. Example: **The architecture of the organization ensures efficiency.** So, "architecture" essentially means "structure, design, order," but depending on the context, it takes on specific meanings in different fields, such as architecture, computer architecture, and software architecture.

Analogies with Civil Architecture

Civil Engineering and Civil Architecture are concerned with the engineering and design of civic structures (roads, buildings, bridges, etc.)

- Architecture vs. Construction
- Multiple views
  - Civil: Artist renderings, elevations, floor plans, blueprints
  - Software: Code, object design, boxes-and-arrows, GUI
- Architectural styles
  - Civil: Classical, Romanesque, Gothic, Renaissance, Baroque, Art Deco
  - Software: Pipe-and-filter, client/server, layered system
- Influence of style on engineering principle
- Influence of style on choice of materials

Code and Object Design

- In software engineering, code (often called source code) refers to the set of written instructions that tell a computer what to do. It is the human-readable text created using a programming language such as Python, Java, C++, or JavaScript.
- Object Design: In software engineering, object design is the process of transforming analysis models (like class diagrams or use cases) into a detailed design that describes how the system will actually be implemented using objects. It's part of the object-oriented design (OOD) phase, which bridges what the system should do (analysis) and how it will do it (implementation).🎯 In Simple TermsObject design determines how objects (classes, attributes, methods, and relationships) will be defined, interact, and collaborate to make the system work.So:In analysis, we identify what the system needs (e.g., "We have Customers and Orders").In object design, we decide how to represent those objects in code (e.g., what data and methods they have, how they communicate, what classes they belong to).

Boxes and Arrows”

- “Boxes and Arrows”In software engineering, the term “boxes-and-arrows” informally refers to diagrammatic representations of systems where:Boxes represent components or entities (e.g., modules, objects, classes, processes), andArrows represent relationships or interactions (e.g., data flow, control flow, communication, dependency).So, “boxes-and-arrows” is a generic visual modeling style — not a specific formal method, but a way to illustrate structure or behavior in a simple, intuitive way. PurposeThe goal of boxes-and-arrows diagrams is to help people see the system, not just read about it.They make abstract ideas concrete by showing:What parts exist (boxes)How they connect (arrows)What flows or dependencies exist between themIn short:Boxes-and-arrows = a visual shorthand for system architecture and logic.

GUI stands for Graphical User Interface.

- GUI is the visual part of a computer program that allows users to interact with the software using graphics, such as:
  - WindowsIcons
  - Buttons
  - MenusText boxes
- Instead of typing complex commands (like in a command-line interface), users can click, drag, or tap to perform actions.
- In Simple Terms A GUI is how the user sees and controls the software.
- So, while programmers deal with code and logic, users deal with the GUI — the “face” of the software.

Software Architecture

"The *architecture of a* ***software system*** defines that system in terms of computational components and interactions among those components. … In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the requirements and elements of the constructed system, thereby providing some rationale for the design decisions."

Software Architecture

"*Software architecture* is a level of design that goes beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem.

Structural issues include,

- gross organization and global control structure;
- protocols for communication, synchronization, and data access;
- assignment of functionality to design elements;
- physical distribution;
- composition of design elements;
- scaling and performance;
- and selection among design alternatives."

Differences Between Civil and Software Architecture

- Physical vs. conceptual
- Static vs. dynamic
- Little evolution vs. frequent evolution
- Different mathematical and scientific bases
- Different notions of “reuse”

Software Designing

- In the software development process, it determines not how a program will function, but how it will be organized.
- **Software design is the planning of a software's components, modules, interfaces, and interactions. In other words, the software's "architecture" is created to solve a problem.**
- Key Features:
  - Abstraction: Making the system more understandable by hiding unnecessary details.
  - Modularity: Dividing the system into small, independent, and reusable components.
  - Low Coupling: Reducing inter-module dependencies.
  - High Cohesion: Each module focuses on a single purpose.
  - Reusability: The ability to reuse previously developed components.

What is the software system

- A software system represents an integrated structure of programs, modules, services, and their relationships that work together to achieve a specific goal or solve a specific problem. In other words, it's not a single application, but an ecosystem of applications, databases, services, libraries, and user interfaces.

Features of the Software System

- It consists of components:
  - Application (e.g., mobile app, web app)
  - Database
  - Services (APIs, microservices)
  - User interface
  - Infrastructure (server, cloud, network)
- Works for a common purpose. For example, a banking software system maintains customer accounts, processes money transfers, and ensures security.
- It has an organized structure (architecture). Software systems are often designed with layered, microservice-based, or monolithic architectures.
- It has a lifecycle:
  - Design → Development → Testing → Deployment → Maintenance → Update.

Architecture of a software system

- Software architecture defines how a software system is organized at a high level, what its components are, how these components communicate with each other, and the principles by which it is designed.

Basic Features of Software Architecture

- Components: The fundamental building blocks of software (modules, services, classes).
- Connectors: The mechanisms that determine how these components communicate (API, messaging, data flow).
- Patterns: Design principles that determine the organization of software (e.g., layered architecture, MVC, microservices).
- Architectural Decisions: Sets of decisions that specify why a particular technology, method, or structure was chosen.
- Constraints: Limitations affecting the architecture, such as performance, security, and scalability.

Goals of Software Architecture

- Making the system understandable and manageable.
- Simplifying maintenance and development.
- Meeting quality requirements such as performance, security, and scalability.
- Ensuring that different teams (analysts, developers, testers, DevOps) work within a common framework.

Computing Components in the Software System

- Processor Logic: Some of the actual programming that meets user needs has been created. For example: A banking system runs an "interest scheduling" function. It is the most basic of calculation components.
- Algorithms: Step-by-step processes defined to solve a problem. For example: Sorting applications, machine learning models, search applications.
- Data Processing Components: Receive data, process it, and produce results. For example: Storing incoming data files, generating reports.
- Mathematical / Numerical Computing Modules: These are specialized programming libraries used in scientific software or engineering applications. For example: Programming engines such as MATLAB, NumPy, TensorFlow.
- Services: In large systems, each service performs specific programming. For example: "payment service," "inventory control service."
- Runtime (Runtime Environment): Runs the infrastructure of subroutines. For example: JVM (Java Virtual Machine), .NET CLR, Python interpreter.
- Computational Resources (Hardware Supported): Computational power provided by capabilities.
  - CPU: Central Proccessing Unit or General-purpose programming.
  - GPU: Parallel programming, artificial intelligence/image processing.
  - FPGA/ASIC: Scheduling circuits.

Elements of Software Architecture

- Perry & Wolf
  - Structural Elements
    - Processing
    - Data
    - Connecting (“glue”)
  - Form: Weighted Properties and Relationships
  - Rationale

- Shaw & Garlan:
  - Components
  - Interconnections
  - Rules of Composition
  - Rules of Behavior

∗ ∗ ∗ *This Class* ∗ ∗ ∗

- *Components*
- *Connectors*
- *Interfaces*
- *Configurations*
- → *(implies) Links*

Components

- A *component* is a building block that is …
  - A unit of **computation** or a **data store**, with an **interface** specifying the services it provides
  - A unit of **deployment**
  - A unit of **reuse**

The Difference Between Components and Objects

- Lifecycle
  - Objects are created and destroyed constantly
  - Components are created and destroyed infrequently
- Purpose of use
  - Graphics toolkit (component) vs. graphics widget (object)
  - Data store (component) vs. data structure (object)
- Type system
  - Objects are instances of a class, with classes arranged in hierarchies according to inheritance relationships
  - Components may have their own type system (may be trivial), often very few components of the same type
- Size
  - Objects tend to be small
  - Components can be small (one object) or large (a library of objects or a complete application)

Connectors

- A *connector* is a building block that enables interaction among components
  - Shared variables
  - Procedure calls (local or remote)
  - Messages and message buses
  - Events
  - Pipes
  - Client/server middleware
- Connectors may be *implicit* or *explicit*
  - Implicit: procedure calls
  - Explicit: First-class message buses

The Difference Between Components and Connectors

- Task Performed
  - Components focus on computational tasks
  - Connectors focus on communication tasks
- Application Semantics
  - Components generally implement most of the application semantics
  - Connectors do not (they may change the form of the message, but do not generally change its meaning)
- “Awareness”
  - Components (should be) unaware of who is using them and for what purpose
  - Connectors are more aware of components connected to them so they can better facilitate communication

Not everybody agrees on this!

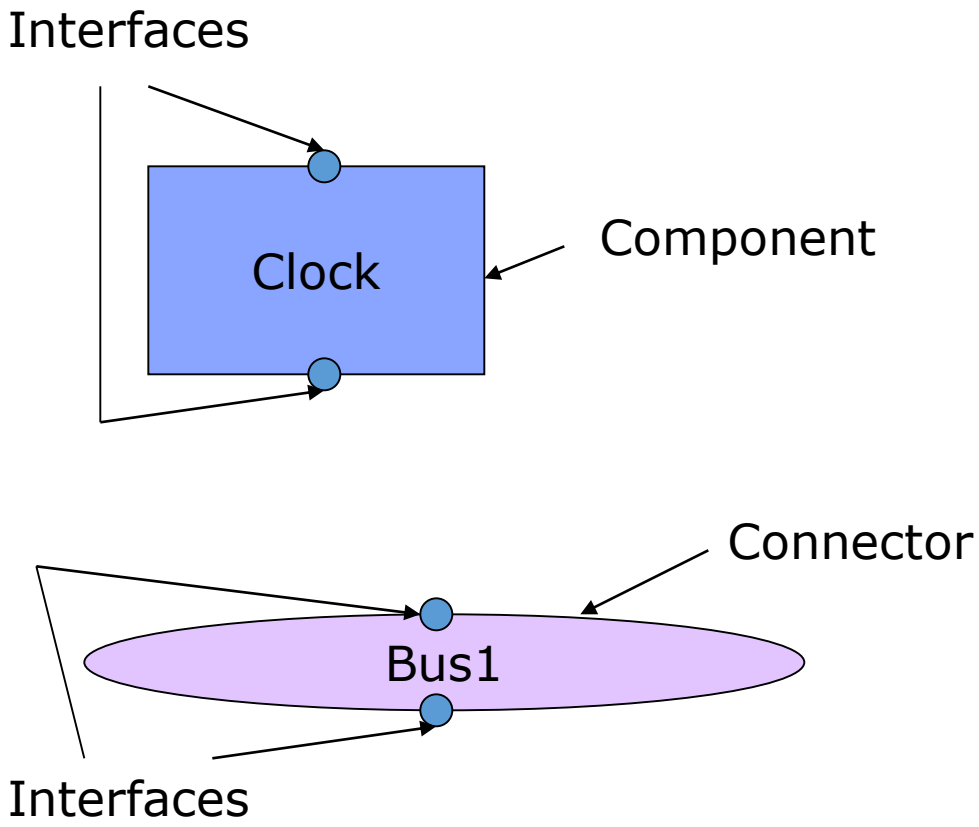
Interfaces

- An *interface* is the external “connection point” on a component or connector that describes how other components/connectors interact with it
- Provided *and* required interfaces are important
- Spectrum of interface specification
  - Loosely specified (events go in, events go out)
  - API style (list of functions)
  - Very highly specified (event protocols across the interface in CSP)
- Interfaces are *the* key to component interoperability (or lack thereof)

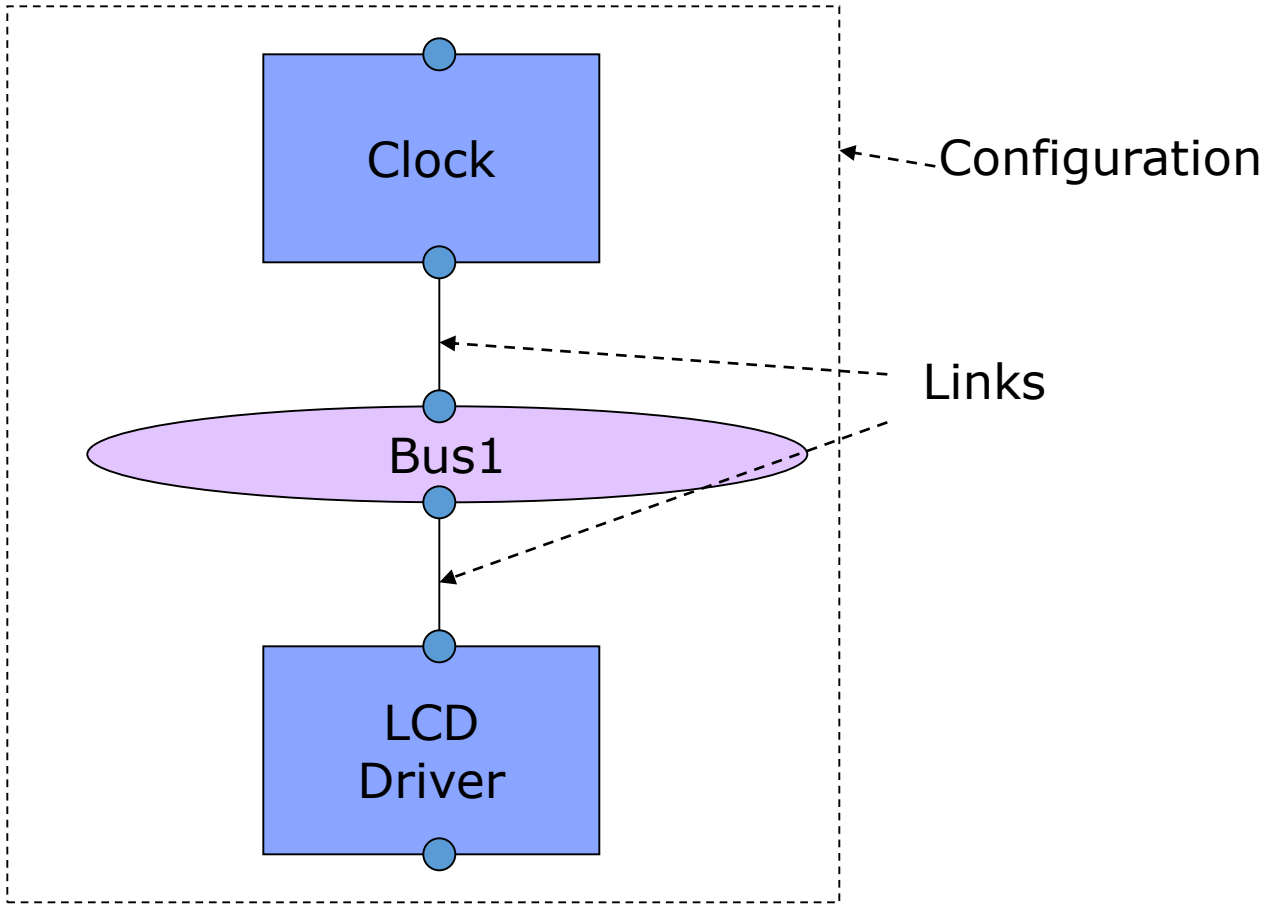
Configurations

- A *configuration* is …
  - The overall structure of a software architecture
  - The topological arrangement of components and connectors
    - Implies the existence of links among components/connectors
  - A framework for checking for compatibility between interfaces, communication protocols, semantics, …
- “If links had semantics, they’d be connectors.”
- Usually constructed according to an *architectural style*

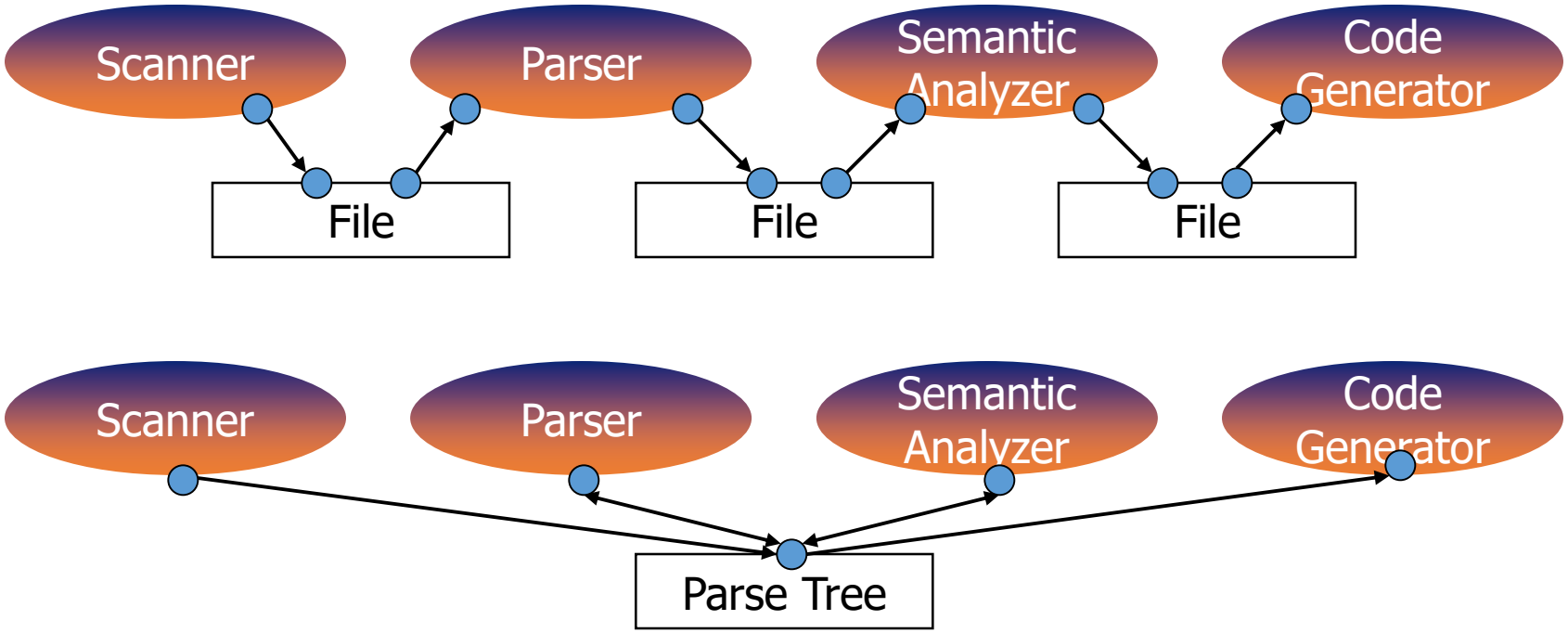
Graphically…



Graphically (cont).



Example:
Architectures for a Compiler



Legend:

What does architecture buy you?

- On its face, nothing!
- A bad architecture can imply a spaghetti code system
  - See “Big Ball of Mud,” http://www.devcentre.org/mud/mudmain.htm
- How can we use architecture to improve the qualities (-ilities) of our software systems?
- **Answer:** Architectural Styles

Architectural Styles

- An *architectural style* is …
  - A set of *constraints* you put on your development to elicit desirable properties from your software architecture.
  - These constraints may be:
    - Topological
    - Behavioral
    - Communication-oriented
    - etc. etc.
  - Working within an architectural style makes development *harder*
  - BUT architectural styles help you get beneficial system properties that would be *really* hard to get otherwise

Common Software Architectural Styles

- Dataflow Systems
  - Batch sequential
  - Pipes and filters
- Call-and-Return Systems
  - Main program and subroutines
  - Object-oriented systems
  - Hierarchical layers (onion layers)
- Independent Components
  - Communicating processes (client/server and peer-to-peer)
  - Event systems
  - Implicit invocation
- Virtual Machines
  - Interpreters
  - Rule-based systems
- Data-Centered Systems (Repositories)
  - Databases
  - Hypertext systems
  - Blackboards

The Vision: Architecture-Based Composition & Reuse

- A framework for design and implementation of large-scale software systems
- A basis for early analysis of software system properties
- A framework for selection and composition of reusable off-the-shelf components
- A basis for controlled evolution of software
- A basis for runtime evolution of software

The Reality: Architectural Mismatch

- *Architectural mismatch* refers to a mismatch between assumptions made by different components about the structure of the system and the nature of the environment in which they operate
- Assumptions about the nature of the components
  - substrate on which a component is built
  - control model
  - data model
- Assumptions about the nature of the connectors
  - protocols
  - data model

Architectural Mismatch (Cont.)

- Assumptions about the global configuration
  - topology
  - presence of certain components or connectors
  - absence of certain components or connectors
- Assumptions about the system construction process
  - order in which elements are instantiated
  - order in which elements are combined
- Assumptions about the operating environment
  - Threading concerns
  - Availability or characteristics of lower-level functionality (OS-level, network)

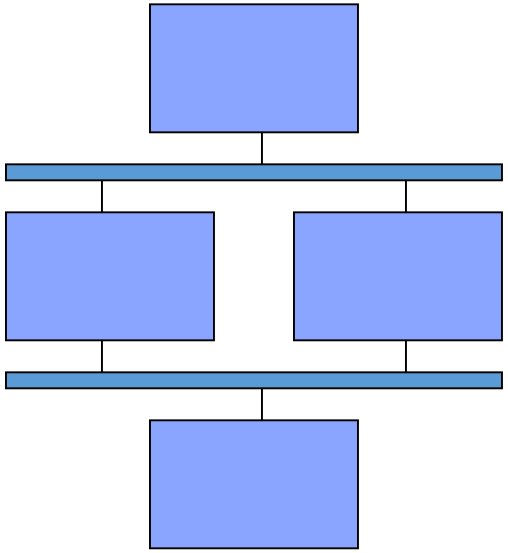
Standards: *The Solution?*

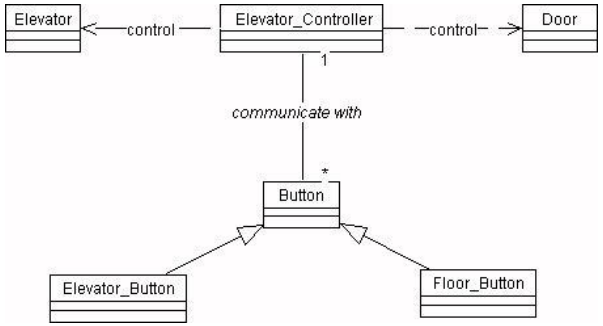
- Standards define a set of “assumptions” that all components must adhere to
  - Component interface standards (e.g., JavaBeans, ActiveX, Netscape Plug-in API)
  - Component interoperability standards (e.g., CORBA, DCOM, Java RMI)
  - Standard component frameworks (e.g., Microsoft Foundation Classes)

- But standards also reduce design flexibility

More Reality:
"Architectural Drift"

- How are architectures implemented now?






```
Python 1.5.2 (#1, Apr 18 999,
16:03:16) Copyright 1991-1995
Stichting Mathematisch Centrum,
Amsterdam >>> import asynchat #
create an instance >>> channel =
asynchat.async_chat() # access to
attributes from an instance >>>
channel.ac_in_buffer '' >>>
channel.ac_in_buffer_size 4096 # now
let's look at the name spas >>>
channel.__dict__['ac_in_buffer'] ''
>>>
channel.__dict__['ac_in_buffer_size'
] Traceback (innermost last): File
"<pyshell#11>", line 1, in ?
channel.__dict__['ac_in_buffer_size'
] KeyError: ac_in_buffer_size # not
found in the instance's namespace! #
check the class namespace >>>
asynchat.async_chat.ac_in_buffer_siz
e 4096
```

Brilliant Software Architecture → *Semi-Cohesive Object Design in UML* → *Incomprehensible Python Code*

Why does drift happen?

- Eventually, software must become code
  - Programming languages and operating systems provide poor support for entities at the architecture level of abstraction
- Induces “architectural drift” (Garlan & Allen)
  - As the system gets built and (especially) maintained, the system drifts further away from its original intended design/architecture
  - Increases software costs and failures dramatically

Solutions for Drift

- Maintain a persistent view of architecture at implementation time
- Co-evolve requirements and architecture
- Use architecture as the basis for:
  - System Design
  - Maintenance
  - System evolution
  - Deployment
- Much of this is still “research!”

The Vision

- Architecture-driven Software Development
  - Architecture as the **primary** abstraction for design
  - The architecture is defined *very* early in the development cycle (possibly in parallel with the requirements)
  - The architecture persists into the implementation and maintenance phases of development
- Gap analysis: how do we get there from here?
  - Ways to represent, manipulate, and visualize architectures
  - Tool support for designing, implementing and evolving architectures
  - Development of *Architecture-centric Development Environments*
    - Think “Visual Studio” but for architectures, rather than code

Contrast: IDE vs. Architecture-based DE

Code-based IDE Tools:

- Text editor
- GUI Builder
- Compiler
- Debugger
- Static Analysis Tools
- Project Management
- Source file configuration management

Architecture-based Tools:

- Visual Editor
- Composition assistants
- Code generator
- Instantiation & Evolution Management
- Various analysis tools
- Style-specific constraint management
- Component-based configuration management

- An extensible architecture-centric development environment
  - Tools within the environment have special support for the C2 architectural style
  - The environment itself is built in the C2 style
- Tools (integrated in various versions):
  - Editors (visual, syntax-directed, and command-line)
  - Analysis tools (static & dynamic analysis, design critics)
  - Off-the-shelf integrations
    - Code-based IDEs (Eclipse, Metamata, etc.)
    - OOAD tools (Rational Rose)
    - Hypermedia systems (Browsers, Chimera)
  - And more…

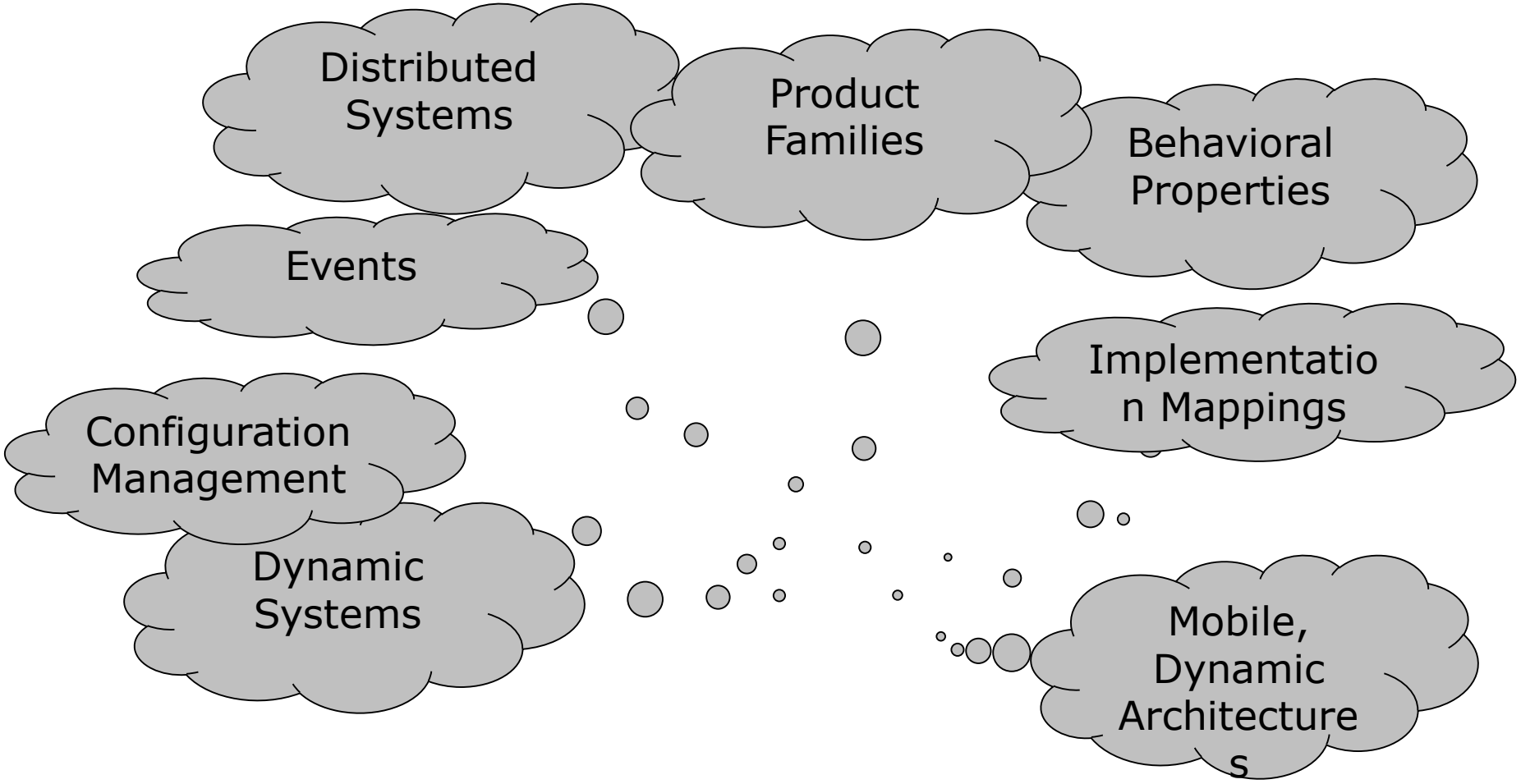
Architecture Description Languages

- How do we “write down” a software architecture?
- An *architecture description language* (or *architecture definition language*, or *ADL*) is a formal notation for describing the structure and behavior of a software architecture
- ADLs provide
  - a concrete syntax
  - a formal semantics
  - a conceptual framework
  - for explicitly modeling the *conceptual architecture* of a system
- Contrast with programming languages, which define the implementation of a system

What Goes in a Software Architecture Description?

- “An ADL must explicitly model ***components***, ***connectors***, and their ***configurations***; furthermore, to be truly usable and useful, it must provide ***tool support*** for architecture-based development and evolution. These four elements of an ADL are further broken down into constituent parts.”
- “In order to infer any kind of information about an architecture, at a minimum, ***interfaces*** of constituent components must also be modeled. Without this information, an architectural description becomes but a collection of (interconnected) identifiers, similar to a “boxes and lines” diagram with no explicit underlying semantics.”

What people have put in their descriptions



Architecture Patterns

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*.

Architecture Patterns

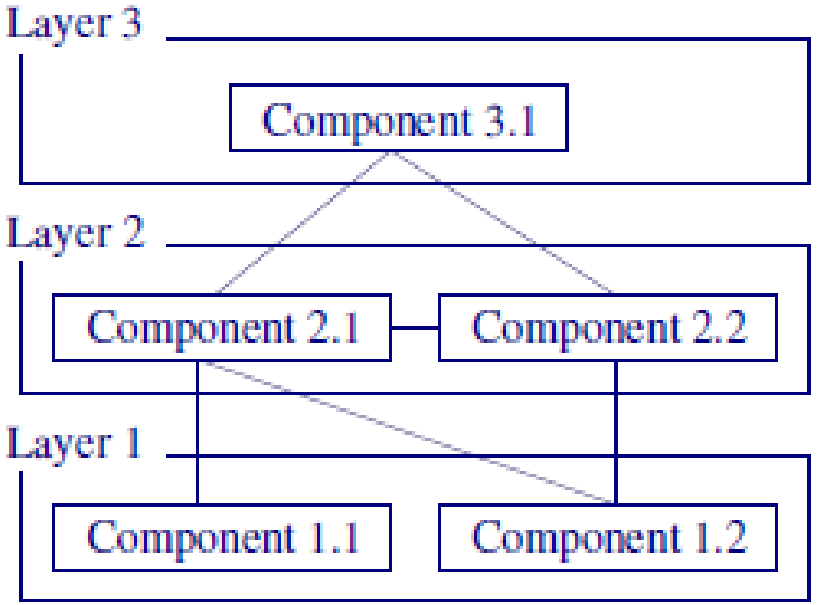
1. **Layered pattern**
2. **Client-server pattern**
3. **Master-slave pattern**
4. **Pipe-filter pattern**
5. **Broker pattern**
6. **Peer-to-peer pattern**
7. **Event-bus pattern**
8. **Model-view-controller pattern**
9. **Blackboard pattern**
10. **Interpreter pattern**

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.

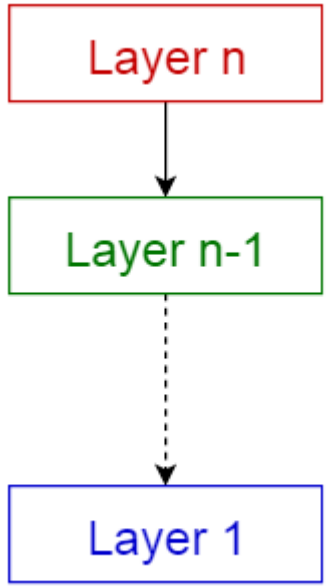
Usage

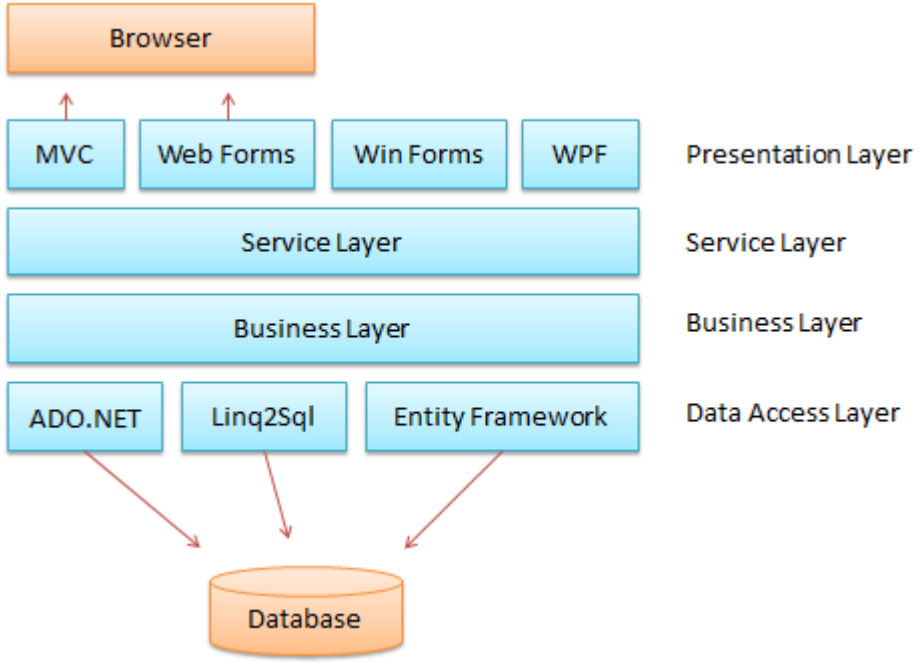
- General desktop applications.
- E commerce web applications.

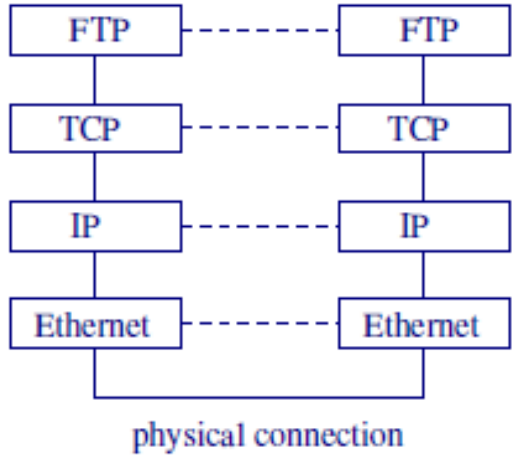


Common layers of a general information system

- **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**)

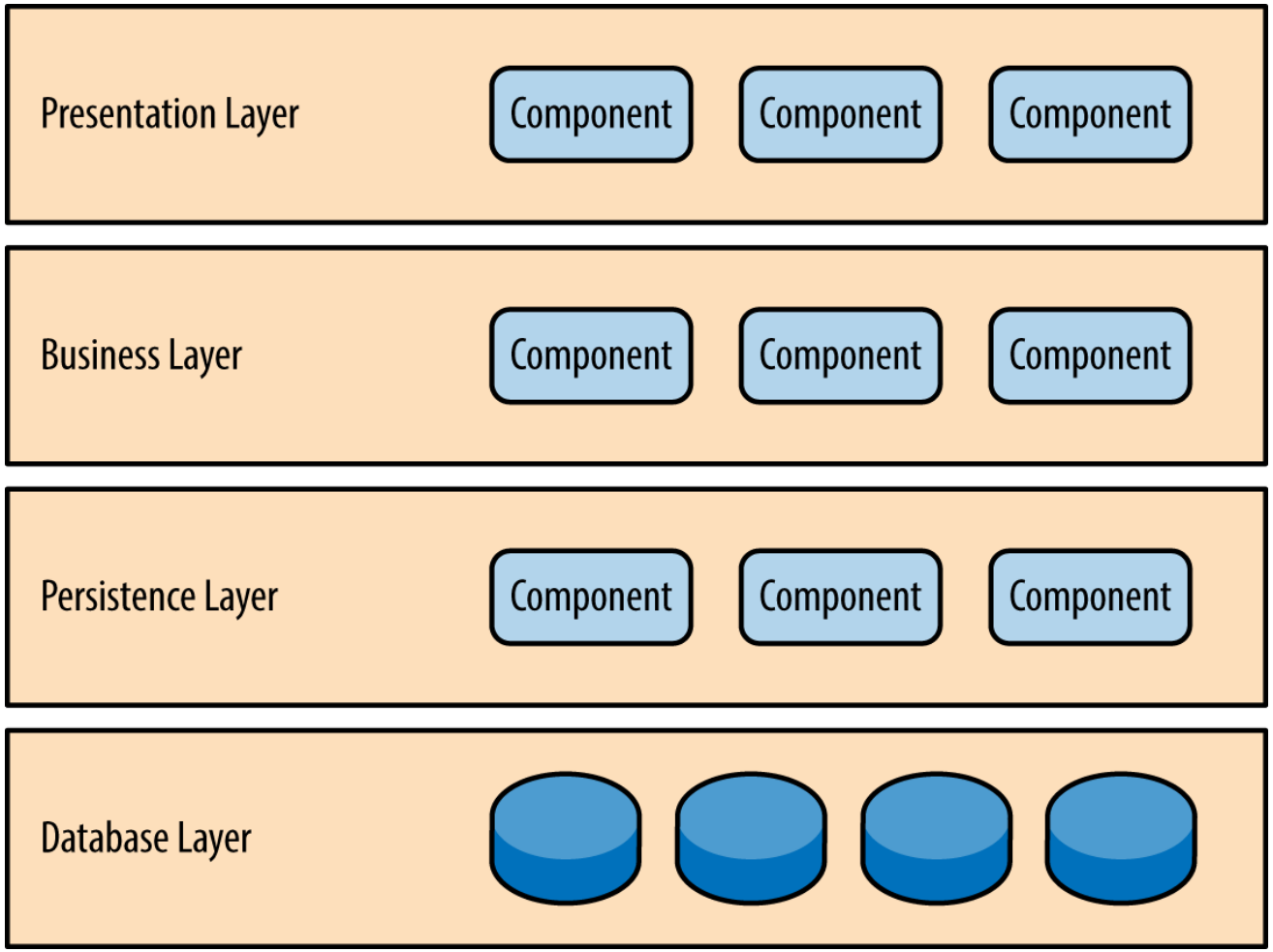




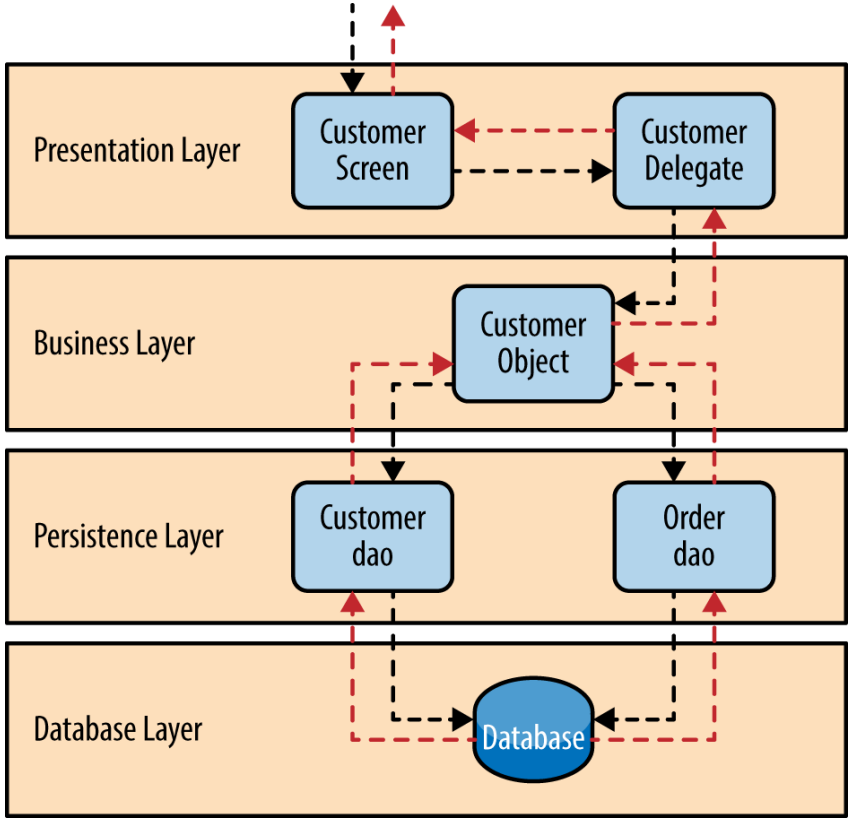


Layered pattern

- 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



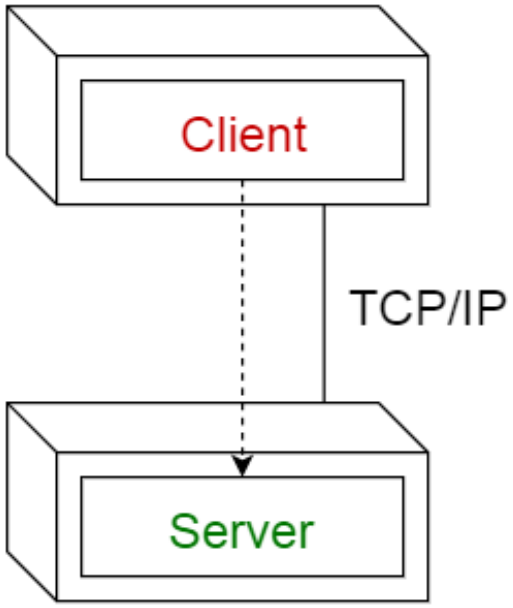
Layered pattern



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

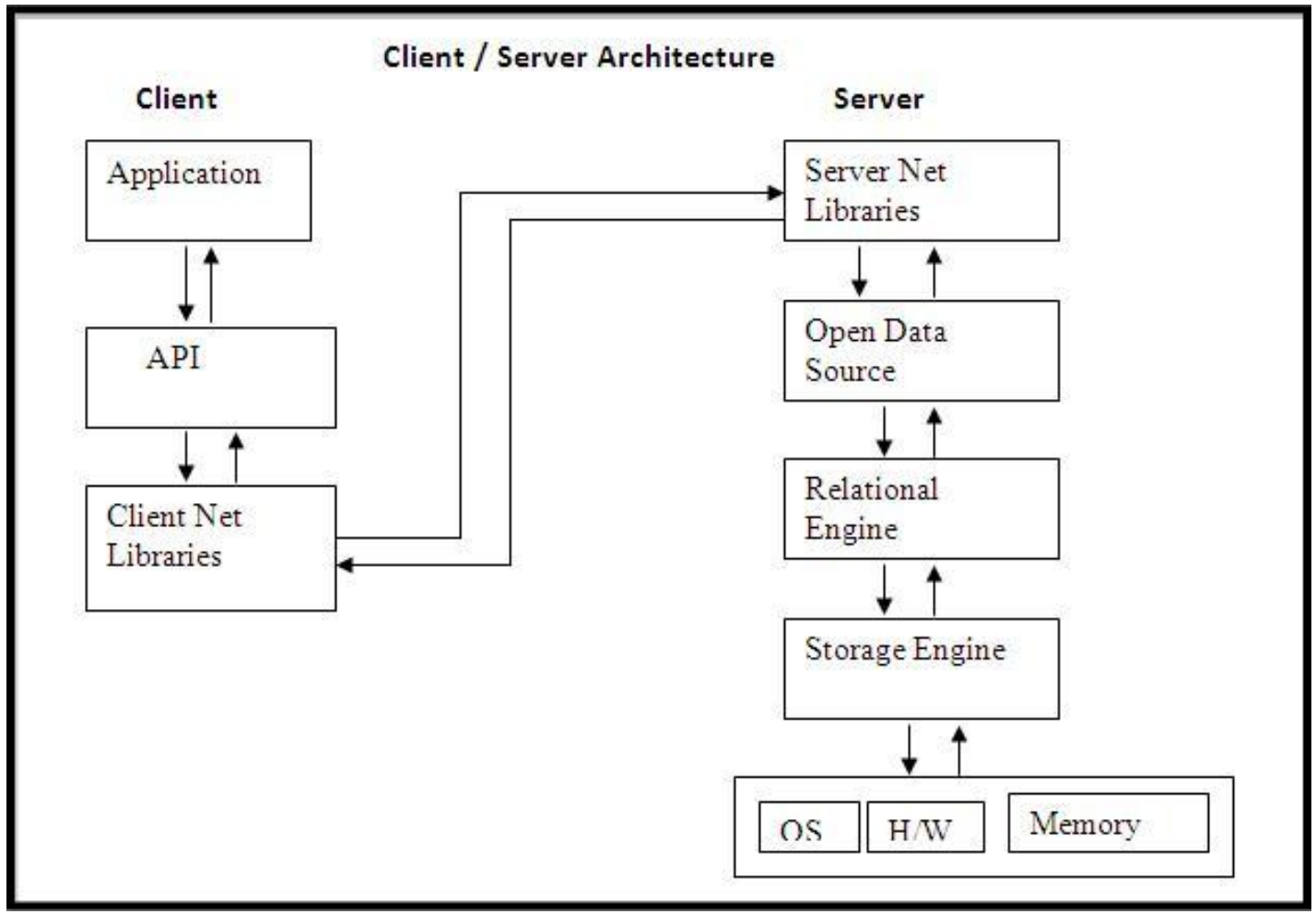
2. Client-server pattern



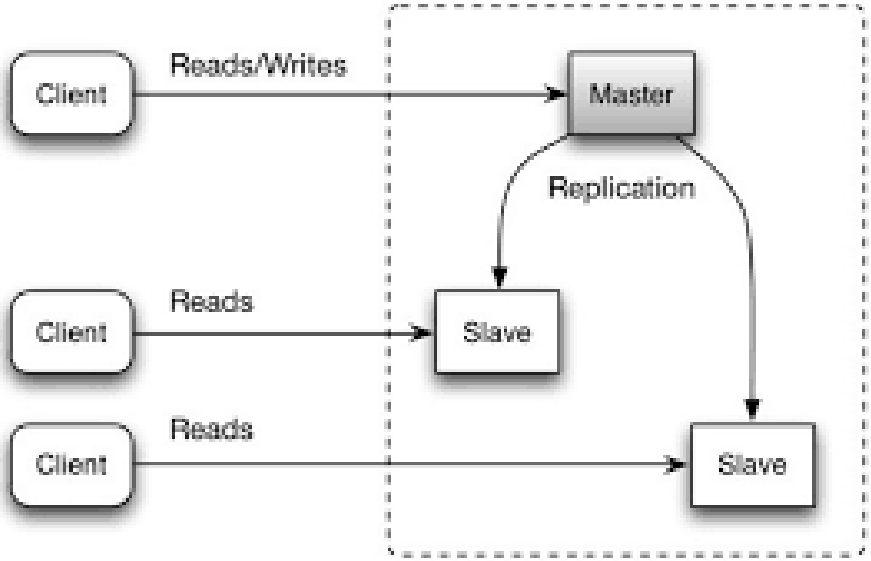
- This pattern consists of two parties;
  - a **server** and
  - multiple **clients**.
- The server component will provide services to multiple client components.
- Clients request services from the server and the server provides relevant services to those clients.
- The server continues to listen to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

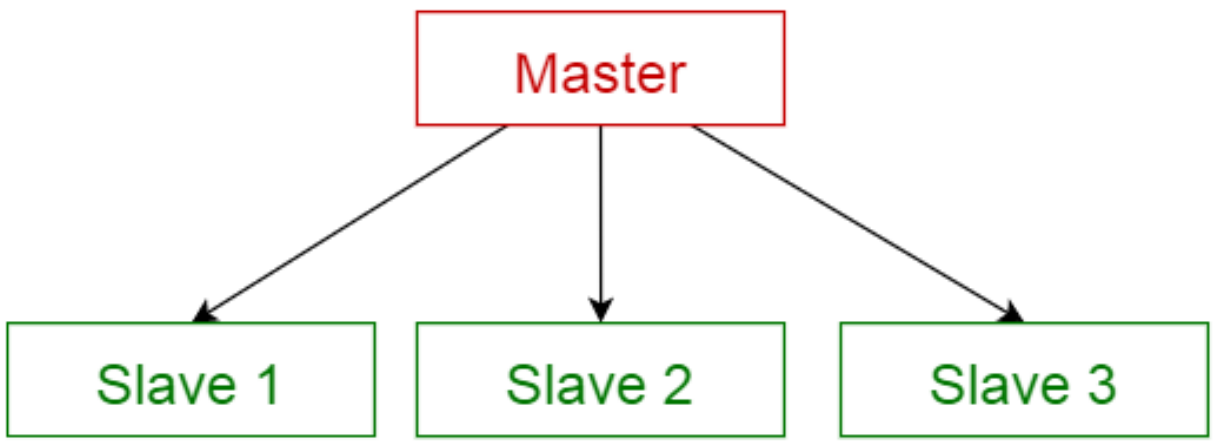
3. Master-slave pattern



- 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

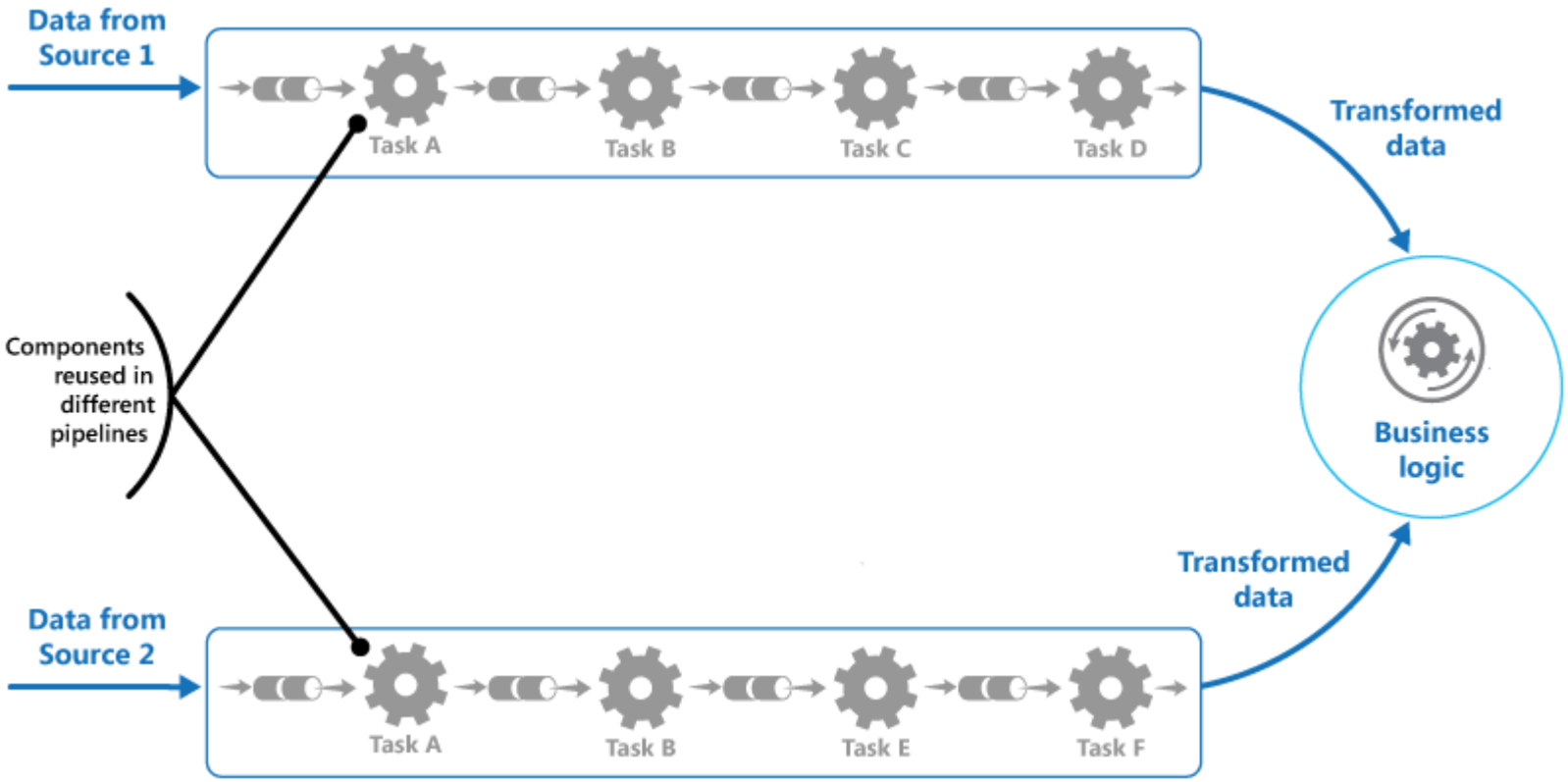
4. Pipe-filter pattern

- 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.

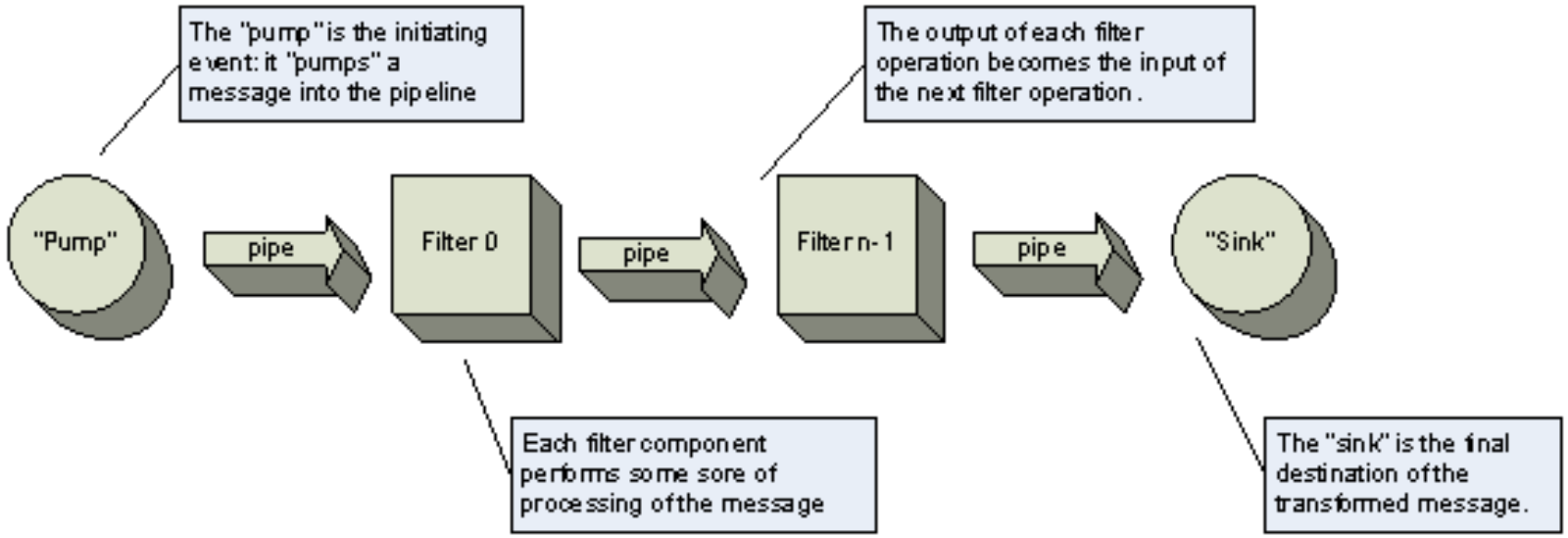
Usage

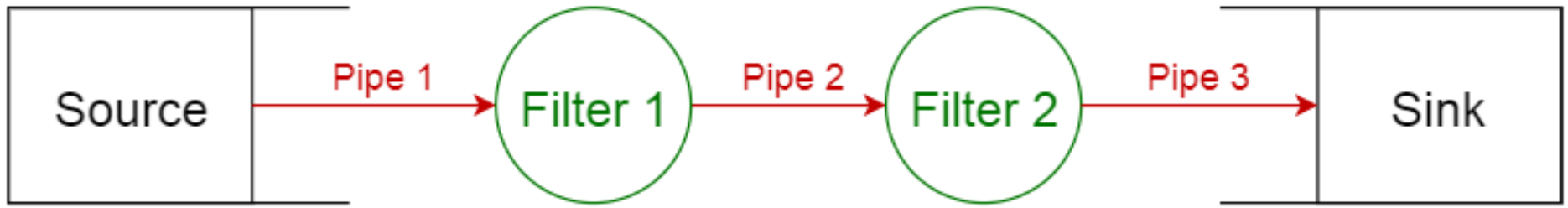
- Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
- Workflows in bioinformatics.
-

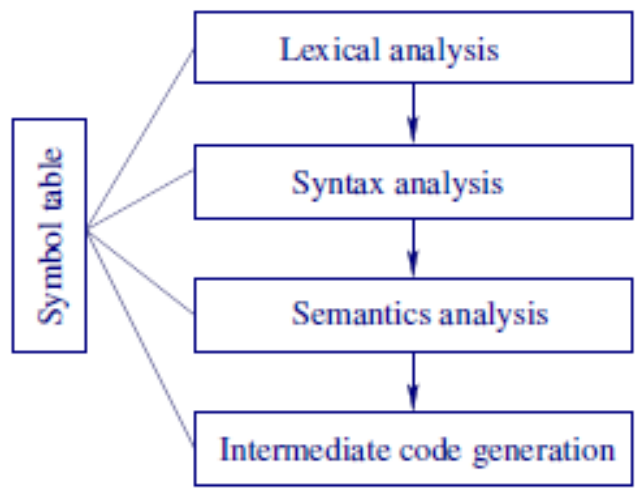
- Pipes eliminate need for intermediate files
- Can replace filters easily
- Can achive different effects through recombination
- İf data stream format is standard, filters ay bbe developed independently
- Parallelization possible



- 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

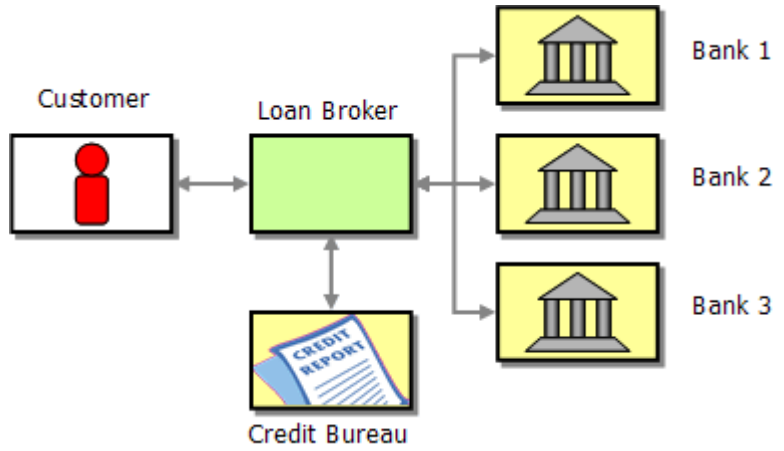

Lexical analysis
Syntax analysis
Semantics analysis
Intermediate code generation
Symbol table

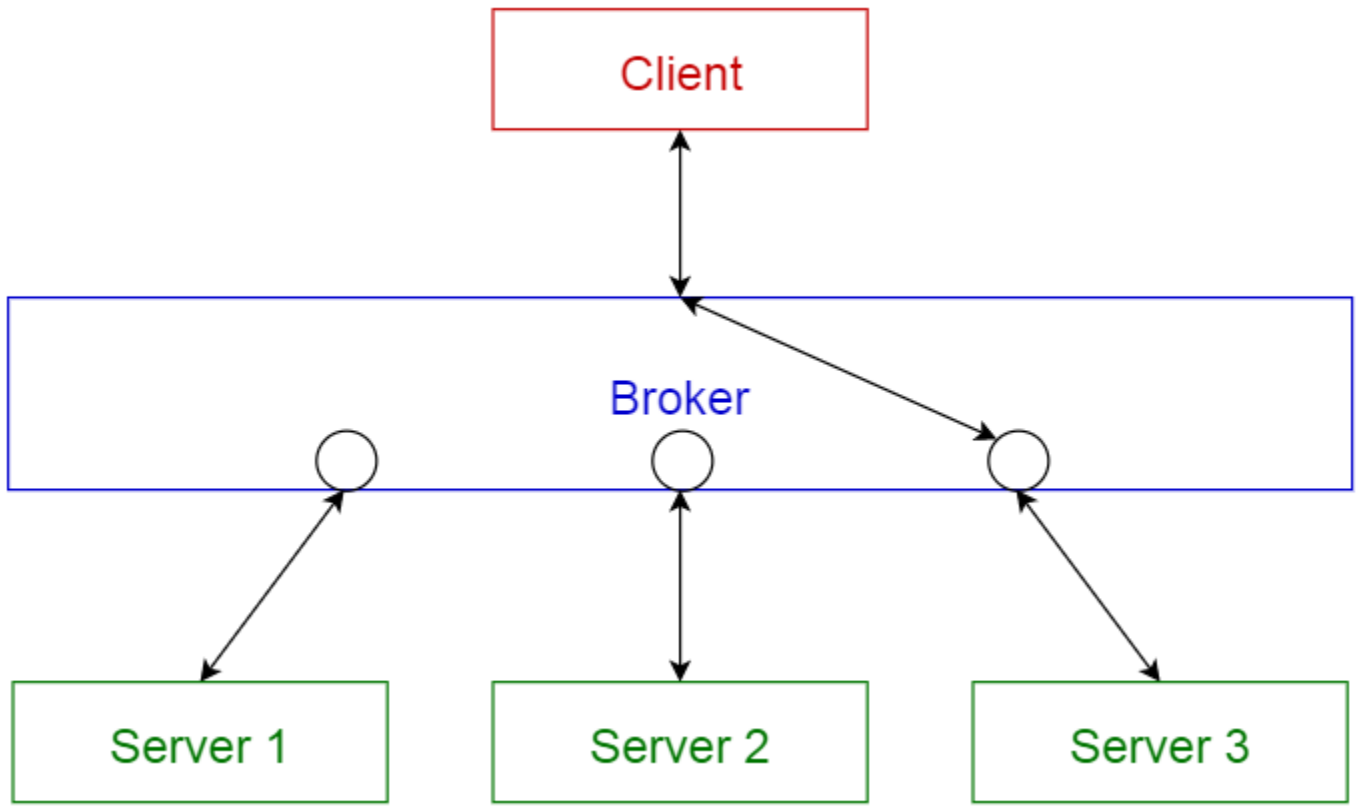
5. Broker pattern

- 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.
-



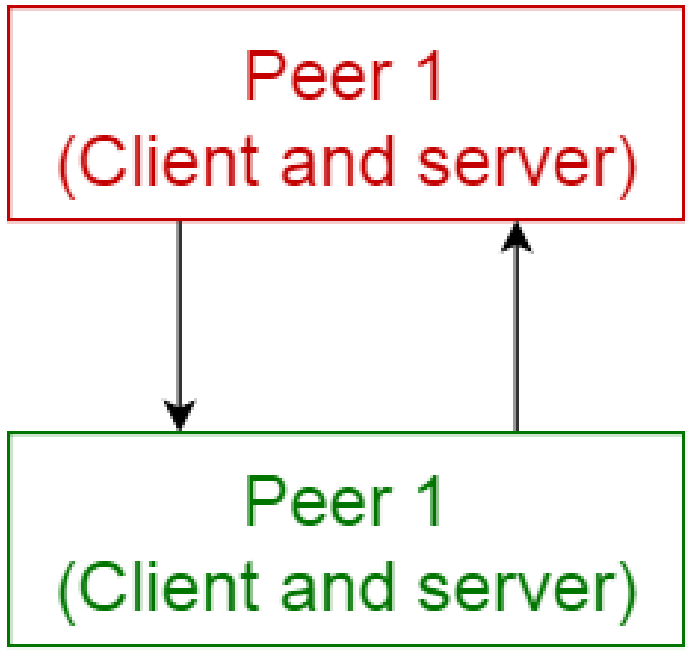

Client
Broker
Server 1
Server 2
Server 3

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**.
-

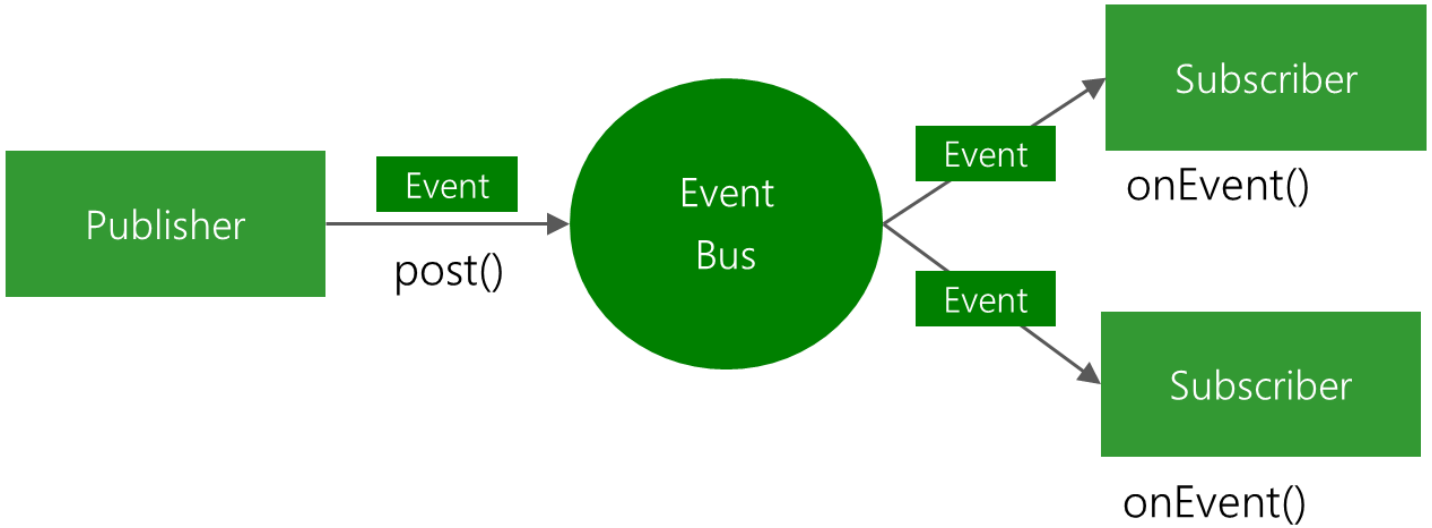

Peer 1
(Client and server)
Peer 1
(Client and server)

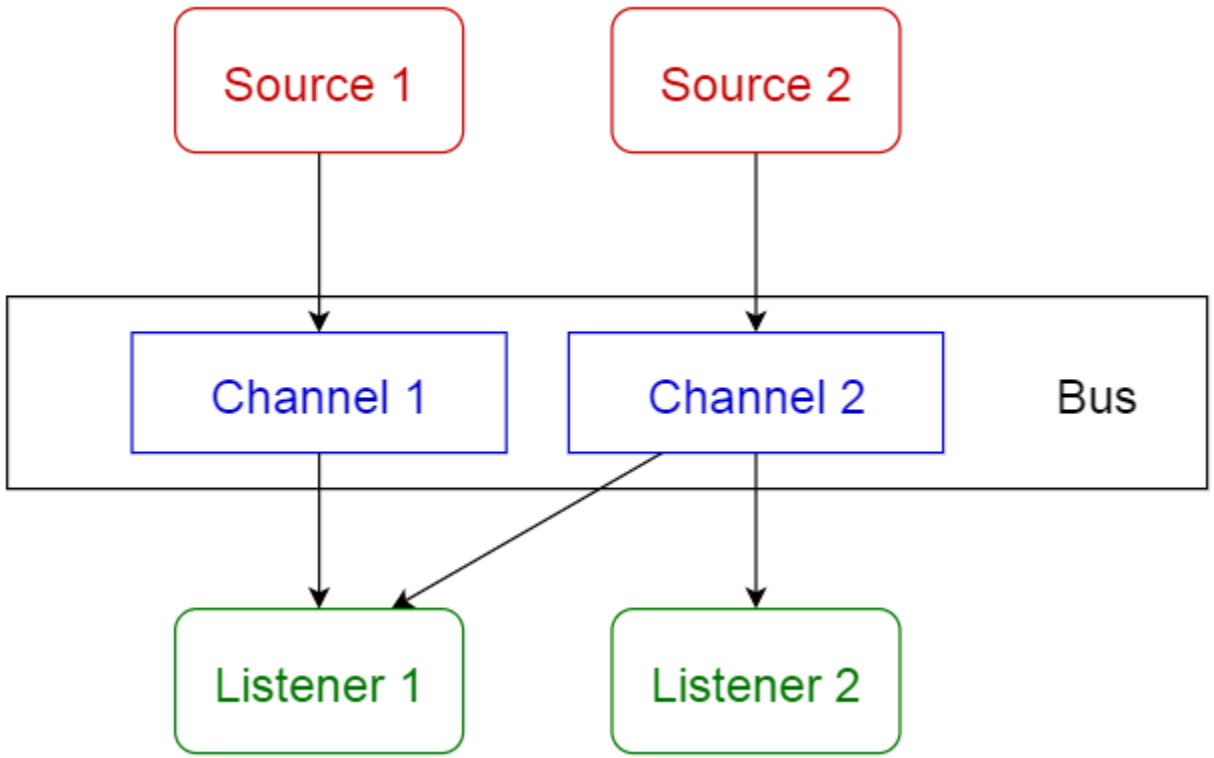
7. Event-bus pattern

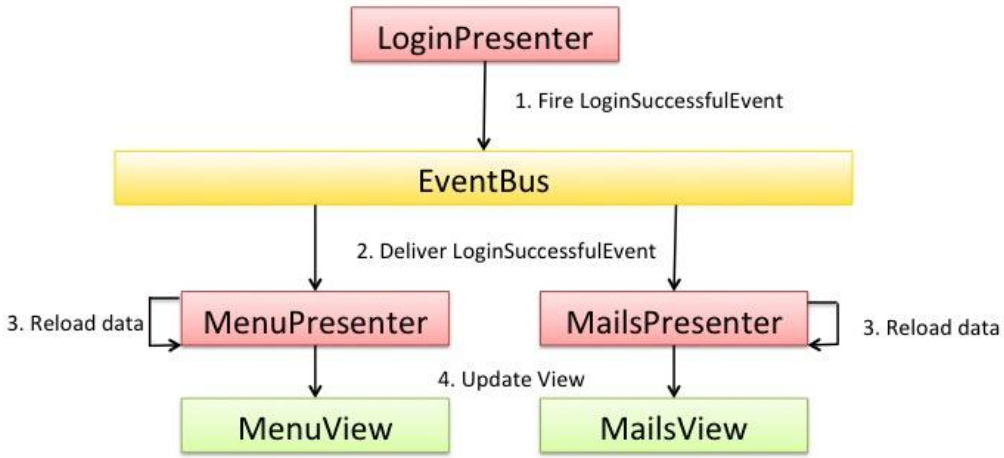
- This pattern primarily deals with events and has 4 major components;
  - **event source**,
  - **event listener**,
  - **channel** and
  - **event bus**.
- Sources publish messages to particular channels on an event bus.
- Listeners subscribe to particular channels.
- Listeners are notified of messages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

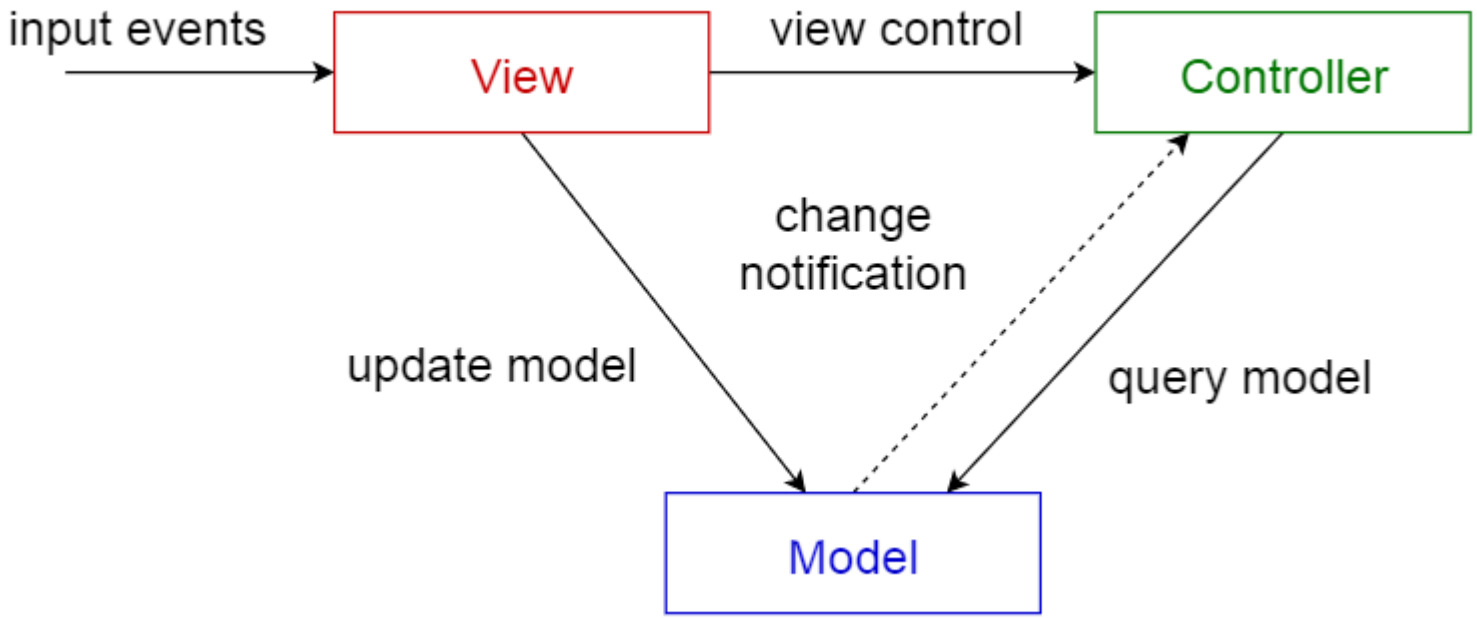

LoginPresenter
1. Fire LoginSuccessfulEvent
EventBus
2. Deliver LoginSuccessfulEvent
3. Reload data
MenuPresenter
MailsPresenter
3. Reload data
4. Update View
MenuView
MailsView

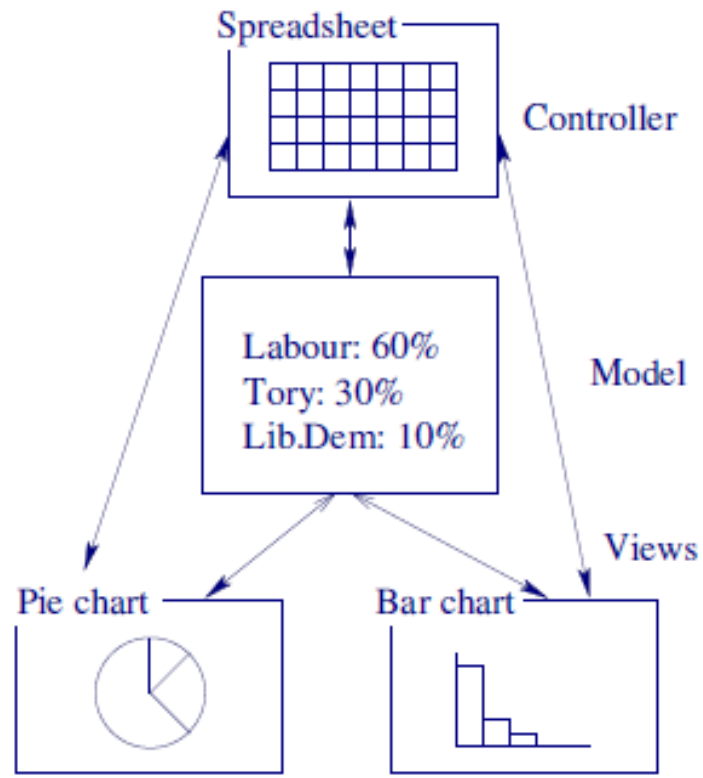
8. Model-view-controller pattern

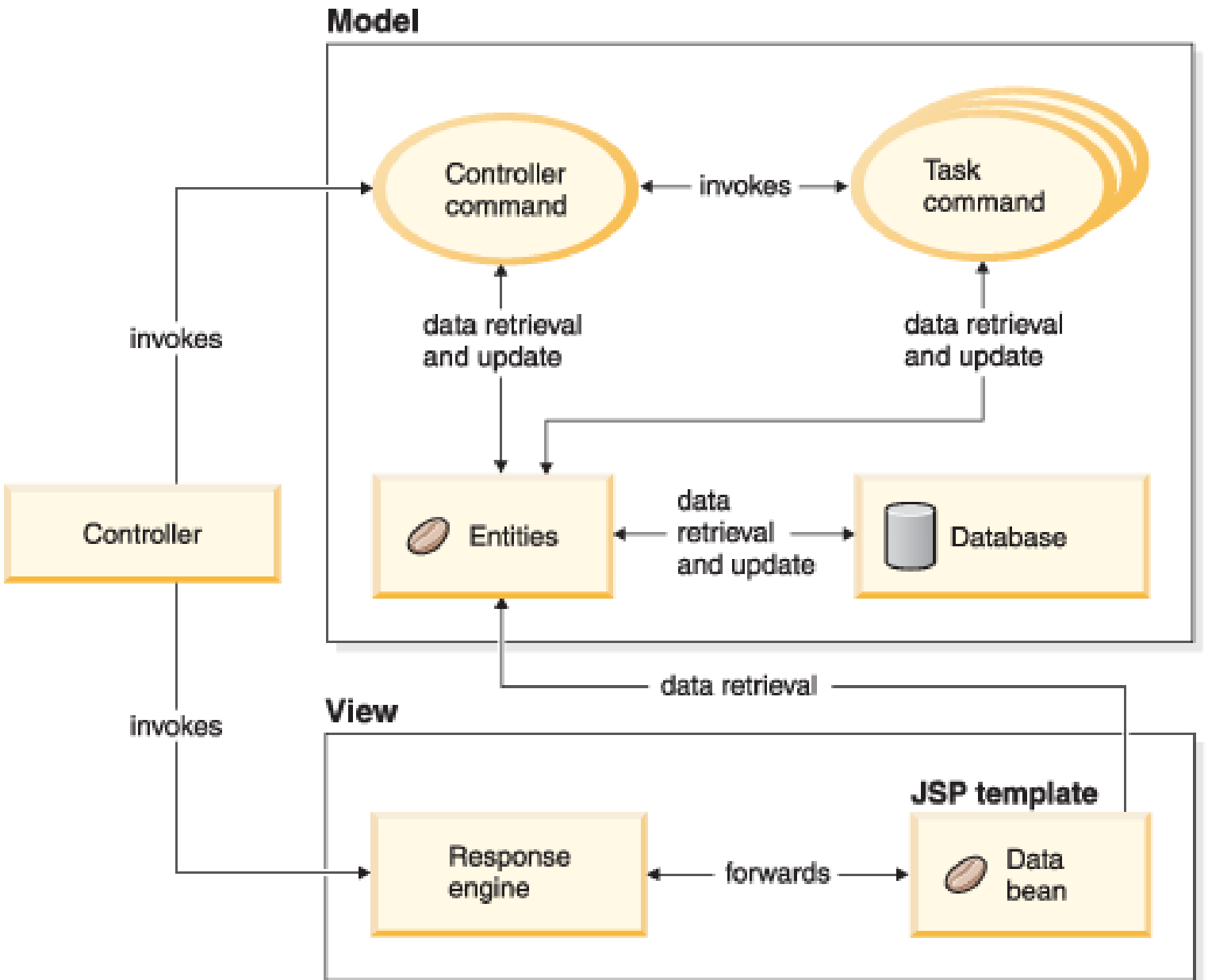
- This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,
- **model** — contains the core functionality and data
- **view** — displays the information to the user (more than one view may be defined)
- **controller** — handles the input from the user
- This is done 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

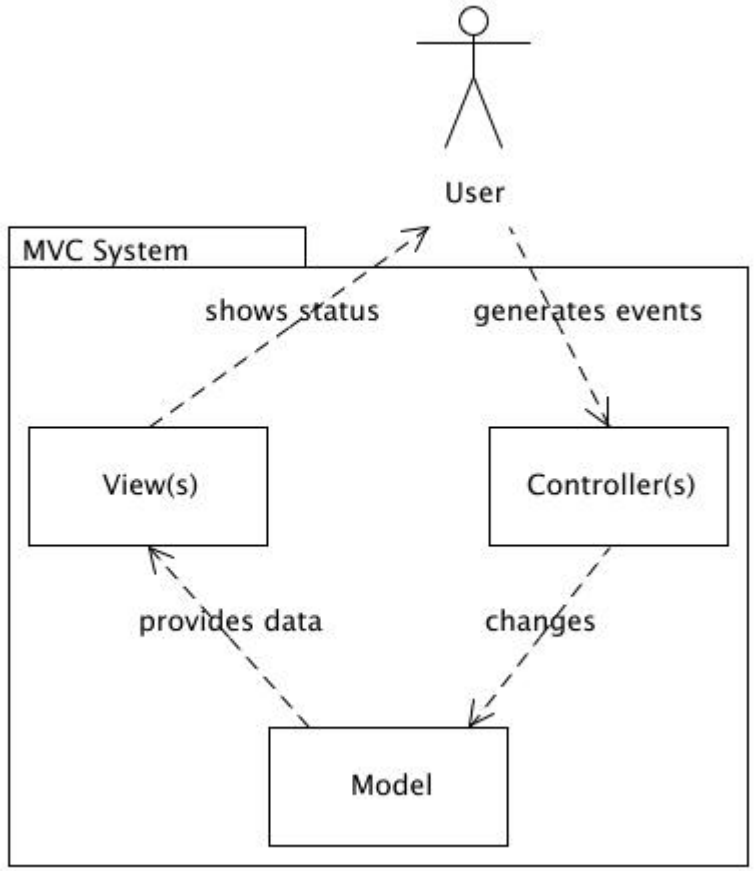
- This pattern is useful for problems for which no deterministic solution strategies are known. The blackboard pattern consists of 3 main components.
  - **blackboard** — a structured global memory containing objects from the solution space
  - **knowledge source** — specialized modules with their own 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.

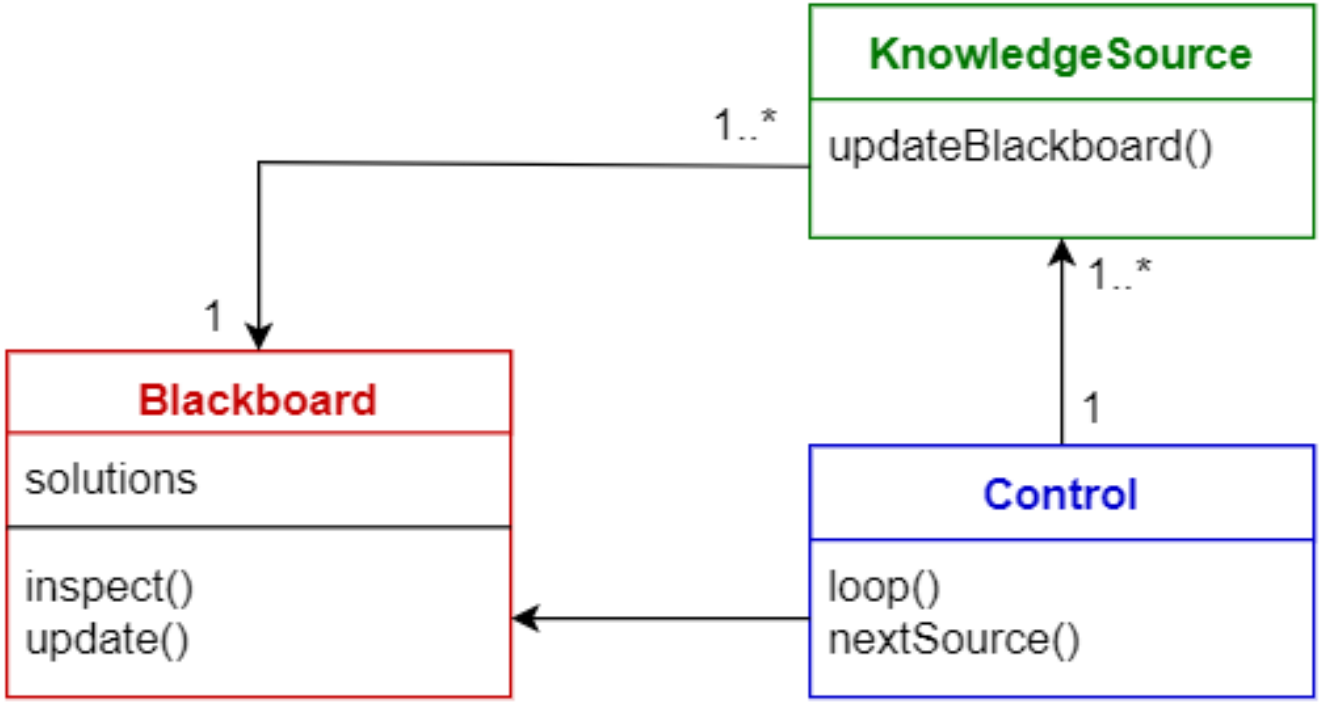
Usage

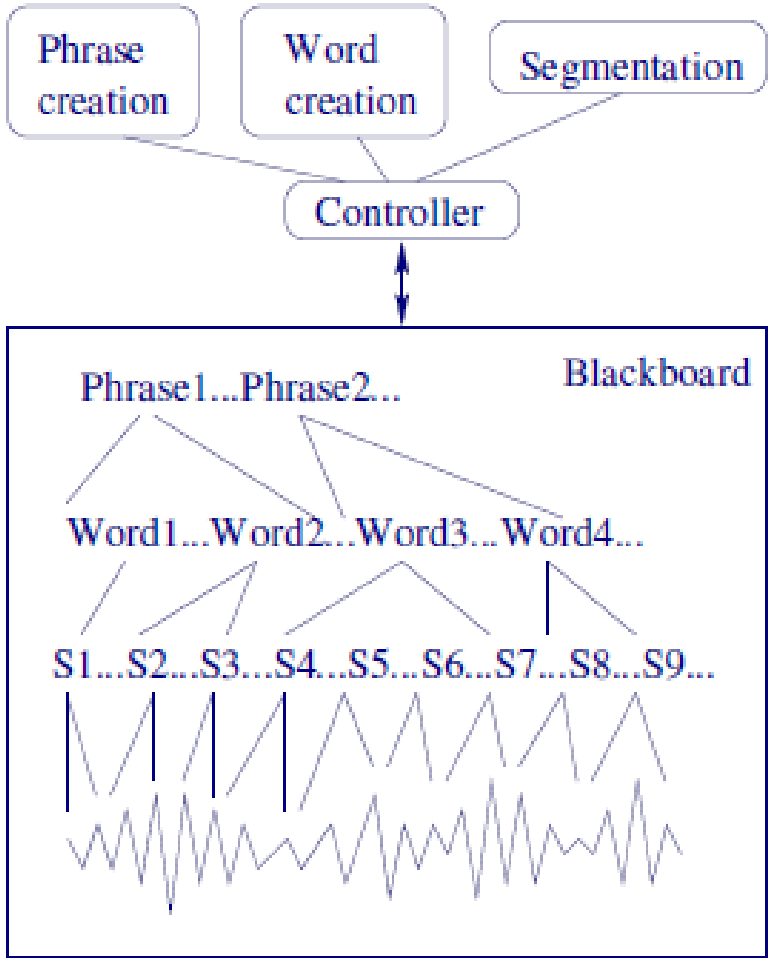
- Speech recognition
- Vehicle identification and tracking
- Protein structure identification
- Sonar signals interpretation.

- 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

- 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()

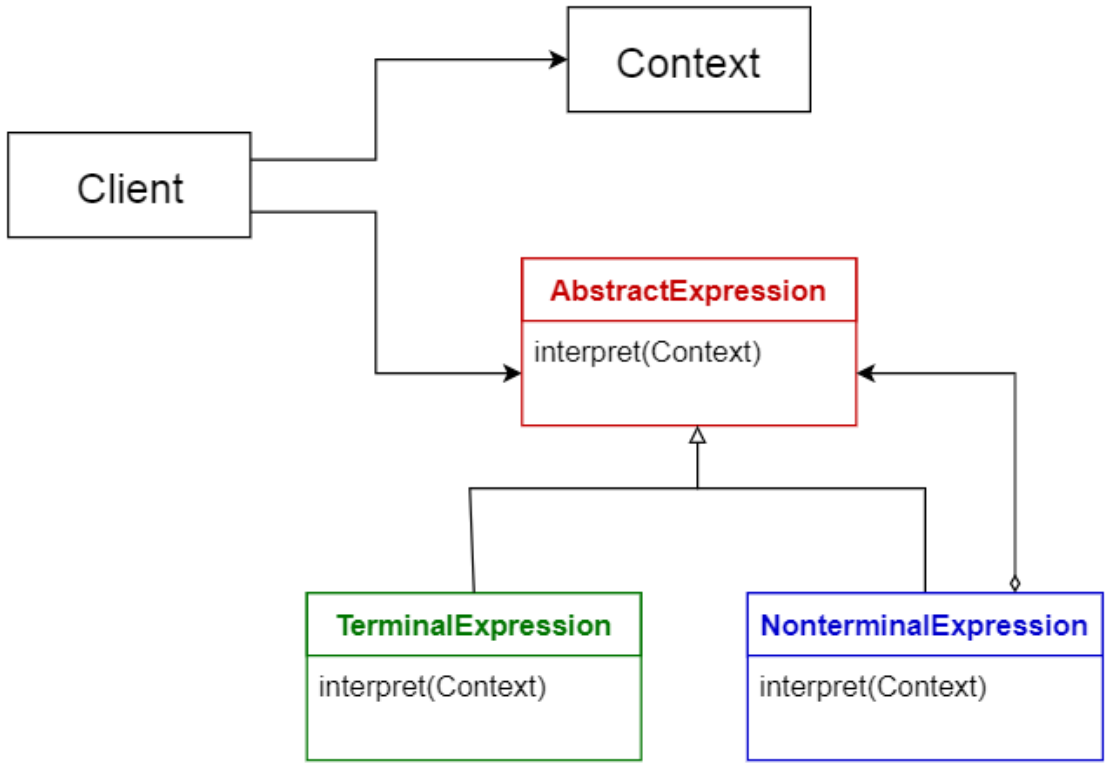

Phrase creation
Word creation
Segmentation
Controller
Blackboard
Phrase1...Phrase2...
Word1...Word2...Word3...Word4...
S1...S2...S3...S4...S5...S6...S7...S8...S9...

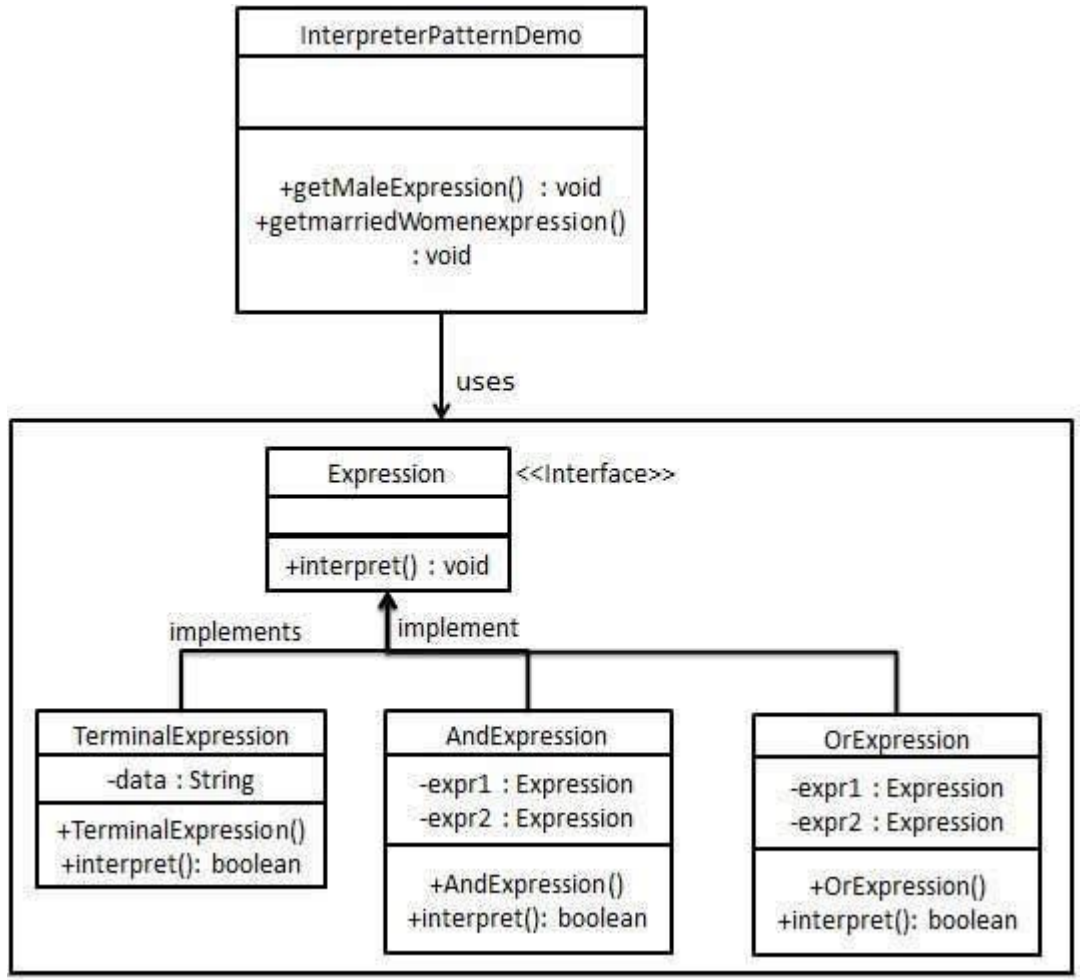
10. Interpreter pattern

- 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.


Context
Client
AbstractExpression
interpret(Context)
TerminalExpression
interpret(Context)
NonterminalExpression
interpret(Context)


InterpreterPatternDemo
+getMaleExpression() : void
+getmarriedWomenexpression()
: void
uses
Expression
<<Interface>>
+interpret() : void
implements
implement
TerminalExpression
-data : String
+TerminalExpression()
+interpret(): boolean
AndExpression
-expr1 : Expression
-expr2 : Expression
+AndExpression()
+interpret(): boolean
OrExpression
-expr1 : Expression
-expr2 : Expression
+OrExpression()
+interpret(): boolean

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.