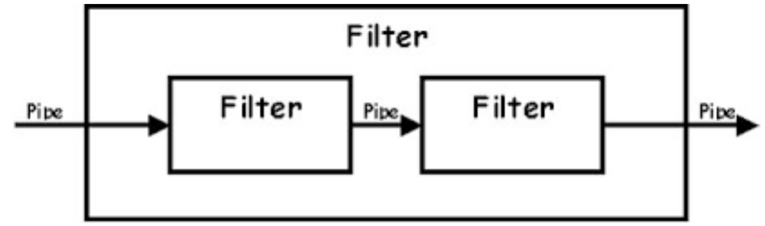
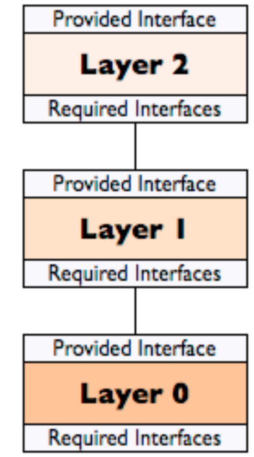
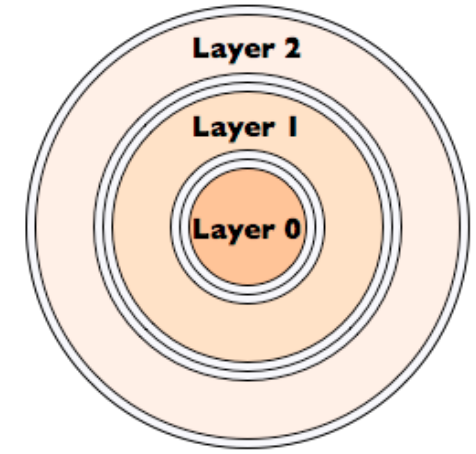




# COMP 3004

# Architectural Styles

**Winter 2020**

**Instructor: Dr. Olga Baysal**

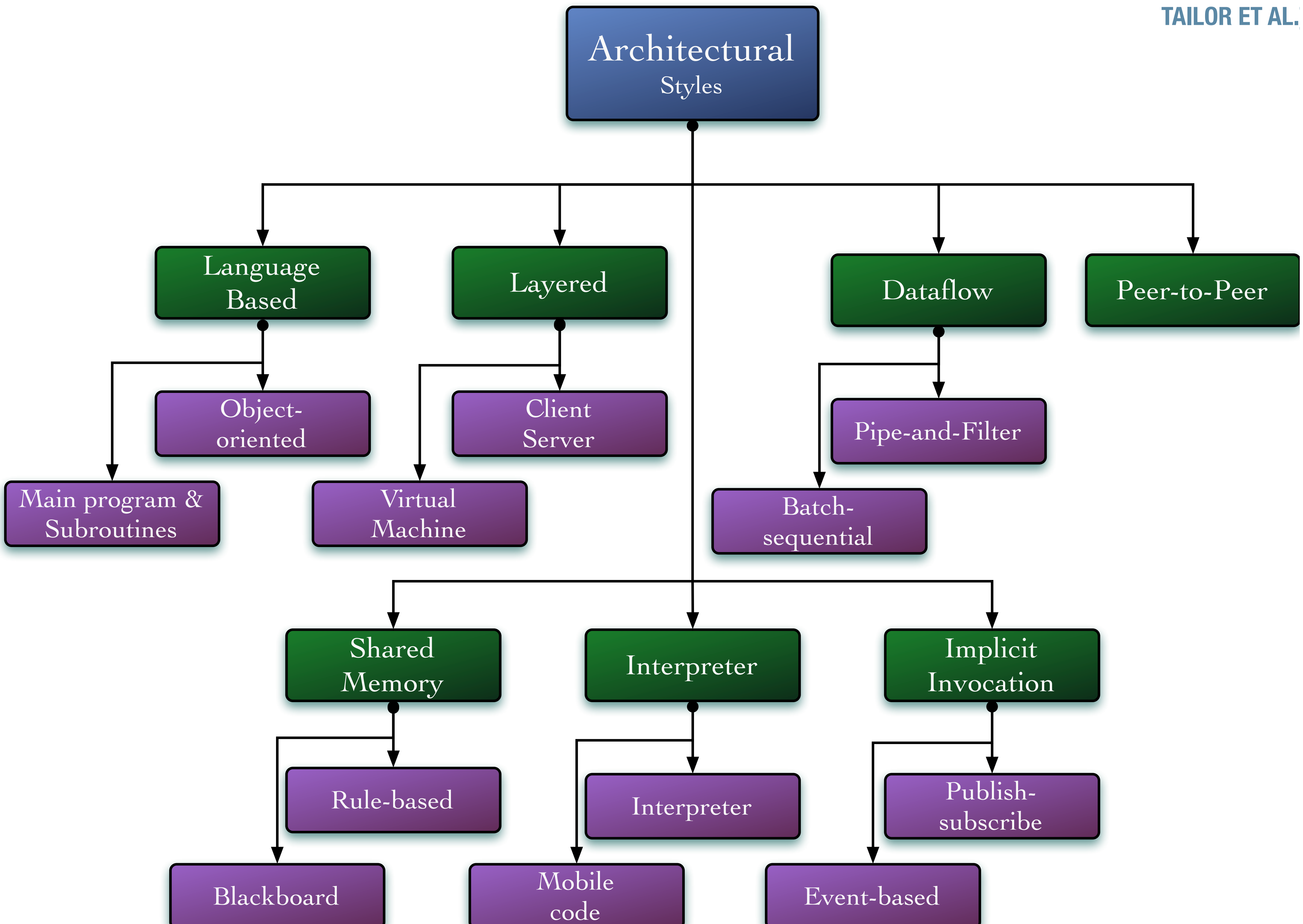
Material and slides from:
Taylor et al.,
Reid Holmes
Alexander Serebrenik

# Objectives

- Identify key architectural style categories
- What are the advantages / disadvantages of different architectural styles?
- What are typical uses of architectural styles?

# Architectural Styles

- TAILOR ET AL.



# Architectural Style Analysis

- Summary
- Design elements (components, connectors, data elements)
- Topology
- Examples of use
- Advantages/disadvantages (aka qualities/cautions)
- Relation to programming languages/environments

# Language-Based

- Influenced by the languages that implement them
- Lower-level, very flexible
- Often combined with other styles for scalability
- **Examples**: object-oriented, main and subroutines

# Style: Main Program and Subroutines

- Decomposition of functional elements
- **Components**:
  - Main program and subroutines
- **Connections:**
  - Function / procedure calls.
- **Data elements:**
  - Values passed in / out of subroutines
- **Topology:**
  - Directed graph between subroutines and main program

# Style: Main Program and Subroutines

- **Qualities:**
  - Modularity, subroutines may be replaced with different implementations as long as interfaces are unaffected
- **Typical uses:**
  - Small programs, educational purposes
- **Cautions:**
  - Poor scalability. Data structures are ill-defined.
- **Relations to languages and environments:**
  - BASIC, Pascal, or C

# Style: Main Program and Subroutines

# Style: Object-oriented

- Encapsulation of state and actions. Objects must be instantiated before the objects’ methods can be called.
- **Components:**
  - Objects (aka instance of a class)
- **Connections:**
  - Method calls
- **Data elements:**
  - Method arguments
- **Topology:**
  - Varies. Data shared through calls and inheritance

# Style: Object-oriented

- **Qualities:**
  - Data integrity. Abstraction: implementation details hidden. Change implementations without affecting clients. Can break problems into interacting parts.
- **Typical uses:**
  - With complex, dynamic data. Correlation to real-world entities.
- **Cautions:**
  - Distributed applications hard. Often inefficient for scientific computing, data science. Potential for high coupling via constructors. Understanding can be difficult.
- **Relations to languages and environments:**
  - C++, Java

# Layered

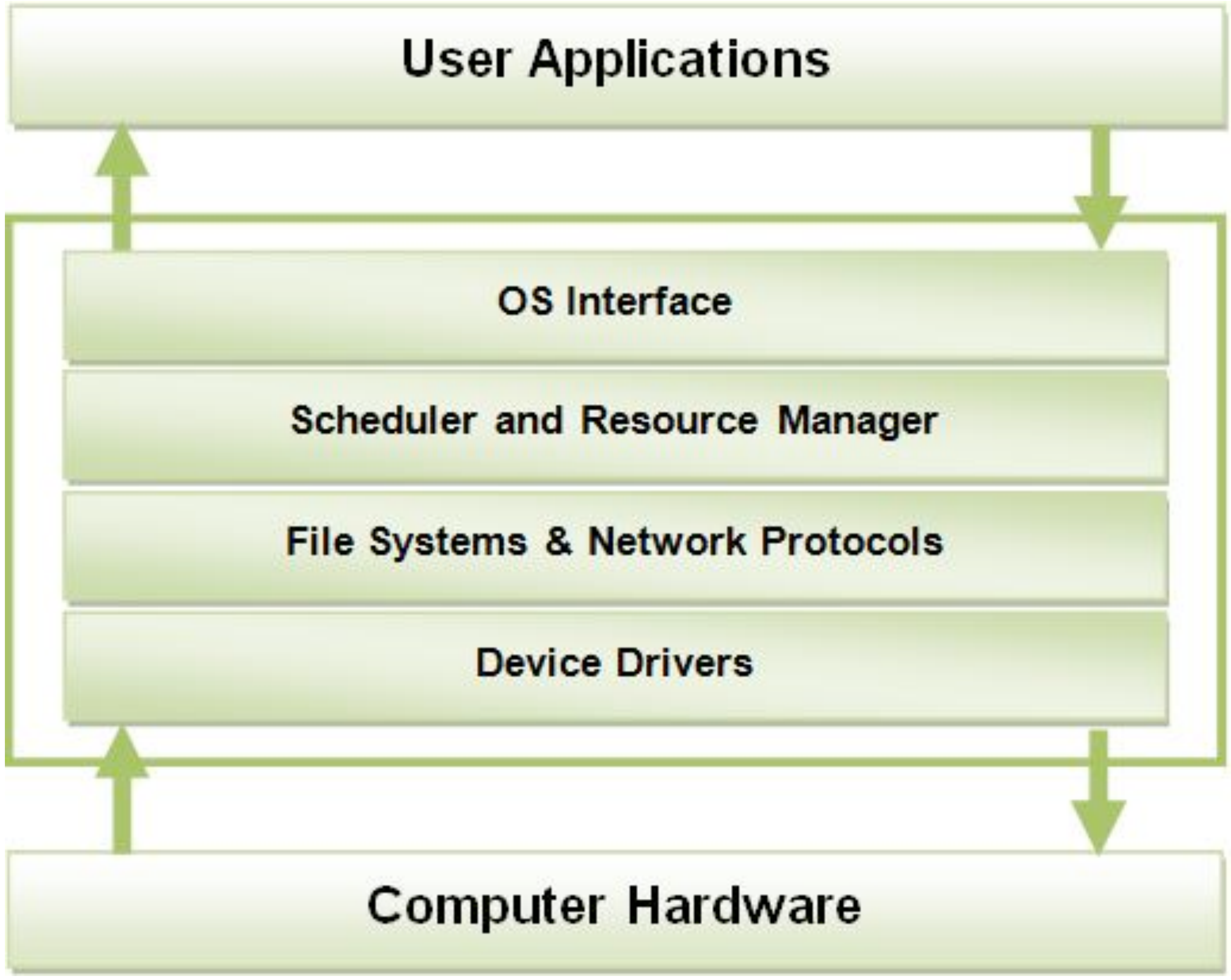
- Layered systems are hierarchically organized providing services to upper layers and acting as clients for lower layers
  - “Multi-level client-server”
  - Each layer exposes an interface (API) to be used by above layers
- Lower levels provide more general functionality to more specific upper layers
- In strict layered systems, layers can only communicate with adjacent layers
- **Examples**: virtual machine, client-server

# Layered

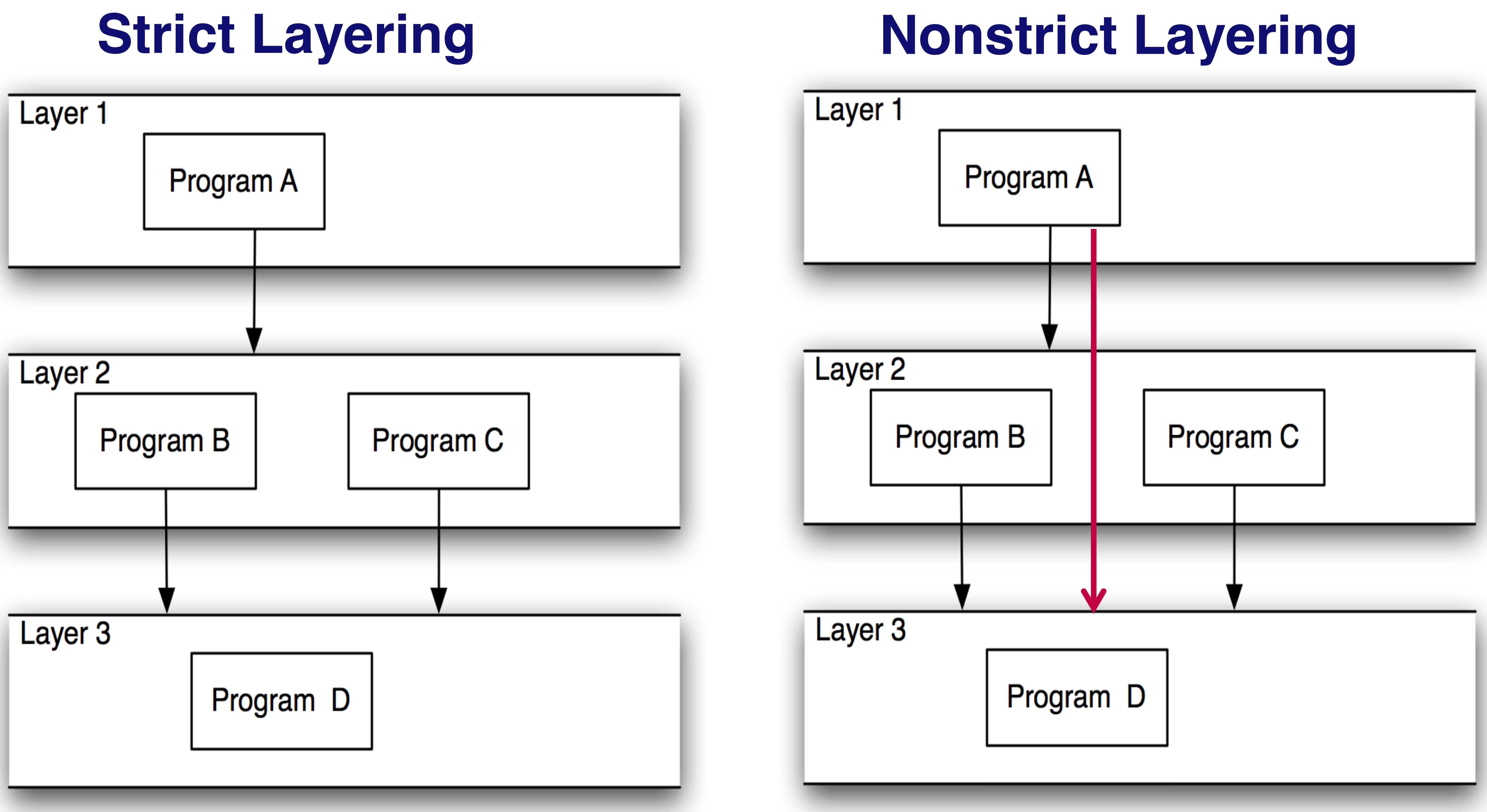
- **Advantages:**
  - Increasing abstraction levels
  - Evolvability (upper layers can evolve independently from the lower layers as long as the interface semantics is unchanged)
  - Changes in a layer affect at most the adjacent two layers
    - Reuse
  - Different implementations of layer are allowed as long as interface is preserved
  - Standardized layer interfaces for libraries and frameworks
- **Disadvantages:**
  - Not universally applicable
  - Performance (mostly for strict layering and many layers)

# Layered

- **Examples:**
  - Operating systems
  - Network protocols



# Layered


Strict Layering
Layer 1
Program A
Layer 2
Program B
Program C
Layer 3
Program D
Nonstrict Layering
Layer 1
Program A
Layer 2
Program B
Program C
Layer 3
Program D

# Style: Client-server

- Clients communicate with server which performs actions and returns data. Client initiates communication.
- **Components:**
  - Clients and server.
- **Connections:**
  - Protocols, RPC.
- **Data elements:**
  - Parameters and return values sent / received by connectors.
- **Topology:**
  - Two level. Typically many clients. No client-client communication.

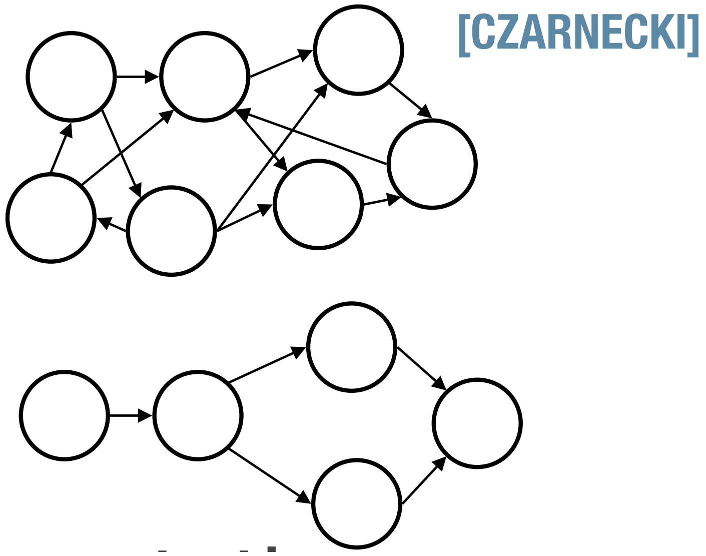
# Style: Client-server

# Style: Client-server

- **Additional constraints:**
  - Clients cannot communicate with each other.
- **Qualities:**
  - Centralization of computation. Server can handle many clients.
- **Typical uses:**
  - Applications where: centralization of computation and data on server; client performs simple UI tasks; server: high-capacity machine (processing power), many business applications.
- **Cautions:**
  - Network bandwidth / amount of requests.

# Dataflow

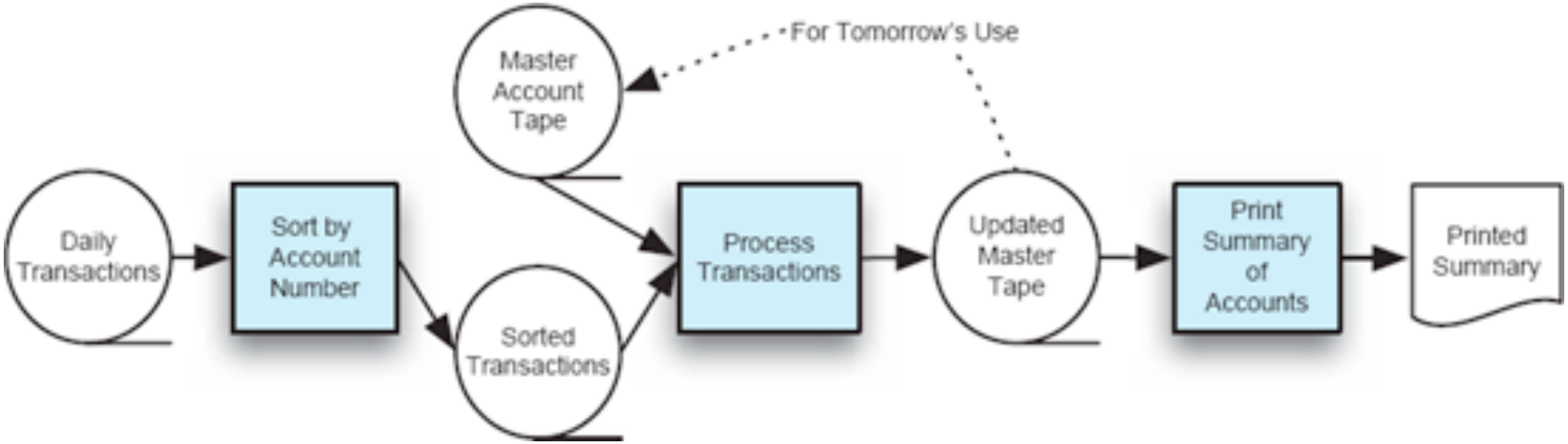
- Separate programs are executed in order; data is passed as an aggregate from one program to the next




- **Examples**: batch sequential, pipe-and-filter

# Style: Batch Sequential

- "The Granddaddy of Styles"
- Separate programs are executed in order
- Aggregated data (on magnetic tape) transferred by the user from one program to another



# Style: Batch Sequential

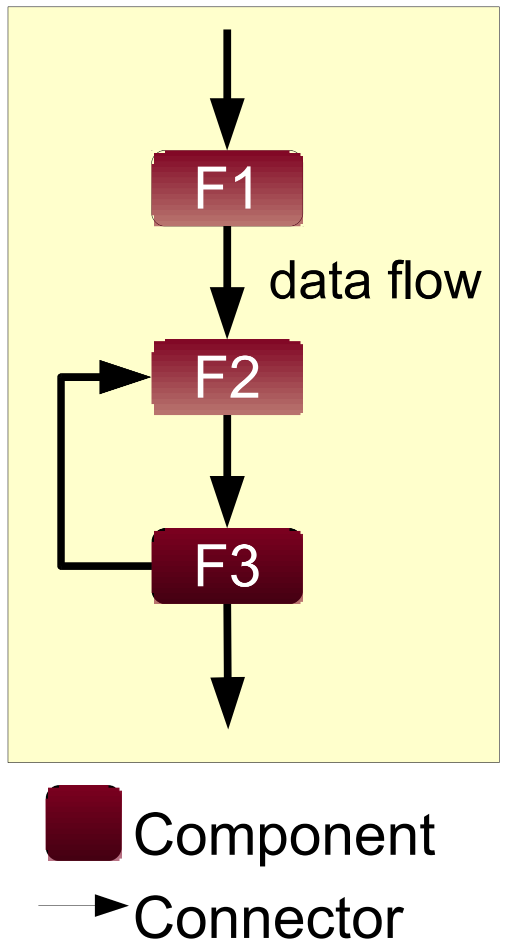
- Separate programs executed one at a time, till completion
- **Components:**
  - Independent programs
- **Connectors:**
  - "The human hand" carrying tapes between the programs, aka "sneaker-net"
- **Data elements:**
  - Aggregated on tapes
- **Topology:**
  - Linear

# Style: Batch Sequential

- **Additional constraints:**
  - One program runs at a time, to completion.
- **Qualities:**
  - Simplicity, severable executions.
- **Typical uses:**
  - Transaction processing in financial systems.
- **Cautions:**
  - No concurrency
  - No interaction between components.

# Style: Pipe and Filter

- Streams of data are passed concurrently from one program to another.
- **Components:**
  - Independent programs (called filters).
- **Connections:**
  - Explicitly routed by OS.
- **Data elements:**
  - Linear data streams, often text.
- **Topology:**
  - Typically pipeline.



# Style: Pipe and Filter

- **Example:** ls invoices | grep -e August | sort
- **Qualities:**
  - Filters are independent and can be composed in novel sequences.
- **Typical uses:**
  - Very common in OS utilities, shells.
- **Cautions:**
  - Not optimal for interactive programs or for complex data structures.

# Style: Pipe and Filter

- **Advantages:**
  - Simplicity
  - Filters are independent
  - New combinations can be easily constructed
  - Concurrent execution
- **Disadvantages:**
  - Poor performance
    - each filter has to parse data
    - sharing global data is difficult
  - Not appropriate for interaction
  - Low fault tolerance threshold
    - what happens if a filter crashes?
  - Data transformation
    - increases complexity & computation

# Shared State

- Characterized by:
  - Central store that represents system state
  - Components that communicate through shared data store
- Central store is explicitly designed and structured
- **Examples**: blackboard, rule-based

# Style: Blackboard

- Two kinds of components
  - Central data structure — blackboard
  - Components operating on the blackboard
- System control is entirely driven by the blackboard state
- Shared blackboard: problem description
- Multiple experts
  - identify a (sub)problem they can solve
  - work on it
  - post the solution on the blackboard
  - enable other experts to solve their problem

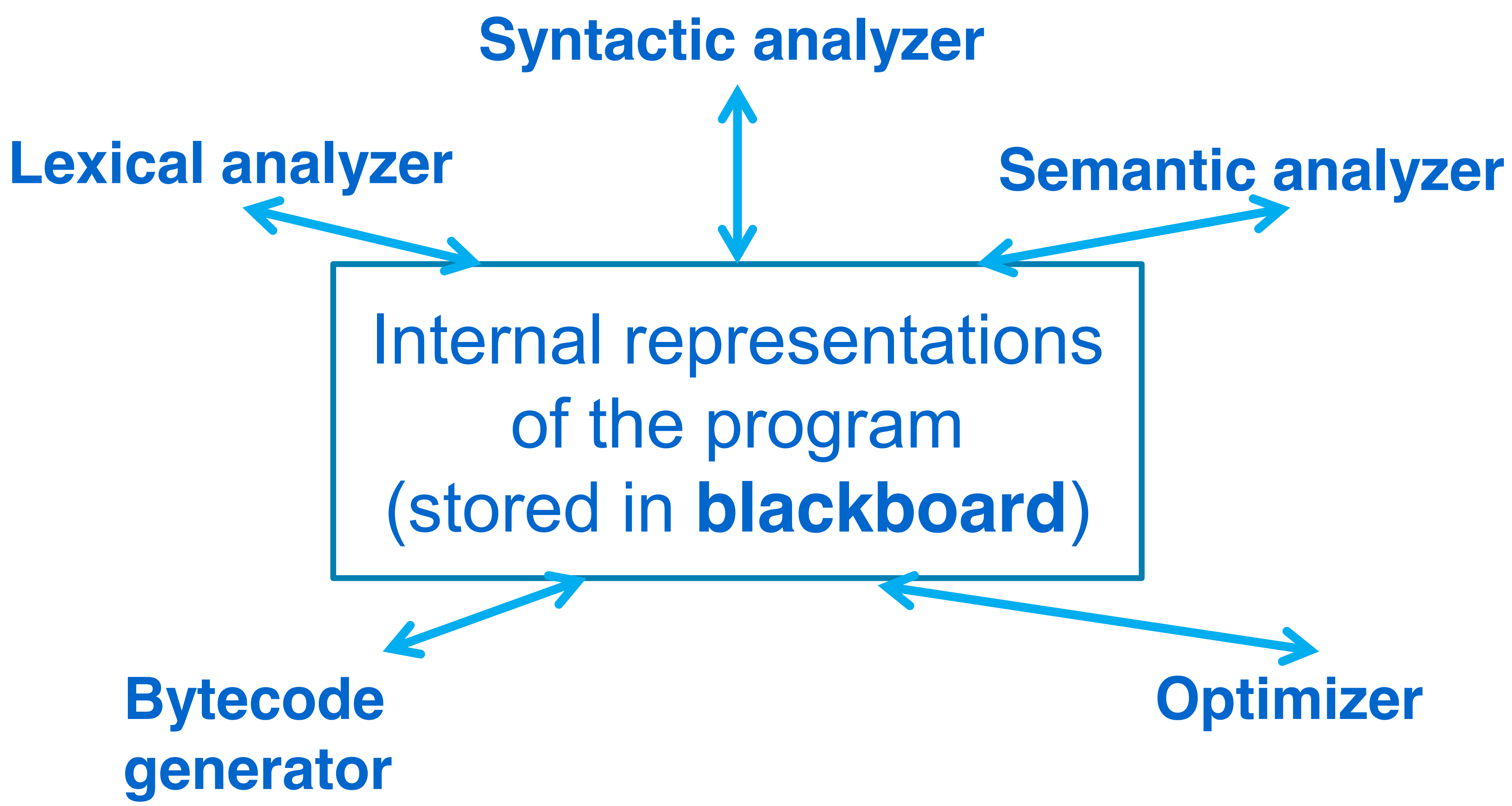
# Style: Blackboard

- Independent programs communicate exclusively through shared global data repository.
- **Components:**
  - Independent programs (knowledge sources), blackboard
- **Connections:**
  - Varies: memory reference, procedure call, DB query
- **Data elements:**
  - Data stored on blackboard
- **Topology:**
  - Star; knowledge sources surround blackboard

# Style: Blackboard

- **Variants:**
  - Pull: clients check for blackboard updates
  - Push: blackboard notifies clients of updates
- **Qualities:**
  - Efficient sharing of large amounts of data. Solution strategies should not be preplanned. Data/problem determine the solutions!
- **Typical uses:**
  - Heuristic problem solving in AI, compiler
- **Cautions:**
  - Not optimal if regulation of data is needed or the data frequently changes and must be propagated to all other components

# Style: Blackboard



# Interpreter

- **Compilers** translate the (source) code to the executable form **at once**
- **Interpreters** translate the (source) code instructions **one by one** and execute them
- **Main idea**:
  - Commands interpreted dynamically
  - Programs parse commands and act accordingly, often on some central data store
- **Examples**: basic interpreter, mobile code

# Style: Basic Interpreter

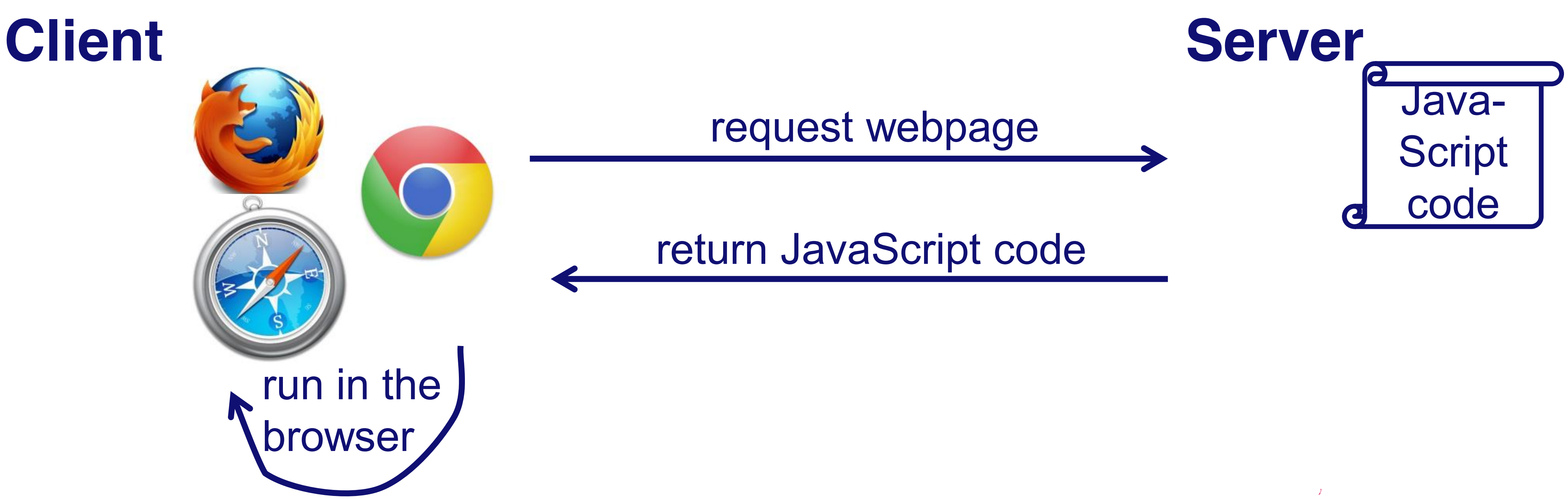
- Interpreter parses and executes input commands, updating the state maintained by the interpreter
- **Components:**
  - Command **interpreter**, program/interpreter **state**, user interface
- **Connectors:**
  - Typically very closely bound with direct procedure calls and shared state
- **Data elements:**
  - Commands.
- **Topology:**
  - Tightly coupled three-tier

# Style: Basic Interpreter

- **Qualities:**
  - Highly dynamic behaviour possible.
- **Typical uses:**
  - Great when the user should be able to program herself (e.g., Excel formulas)
- **Cautions:**
  - Performance (it takes longer to execute the interpreted code, but many optimizations might be possible);
  - Memory management may be an issue (when multiple interpreters are invoked simultaneously)

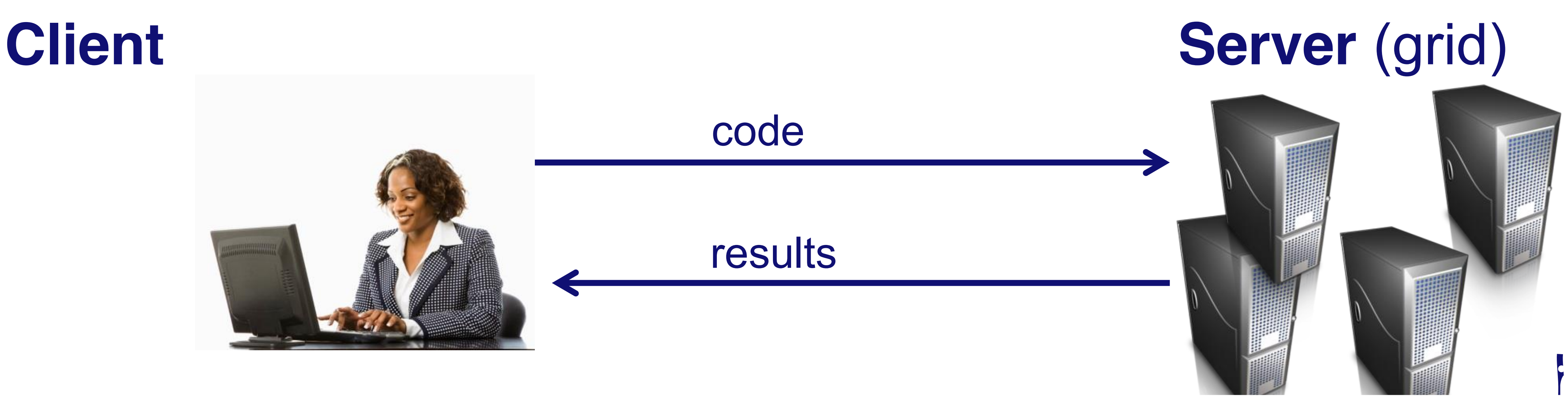
# Style: Mobile Code

- Sometimes interpretation can not be performed locally
  - **Code-on-demand:**
    - Client has resources and processing power
    - Server has code to be executed
    - Client requests the code, obtains it and runs it locally



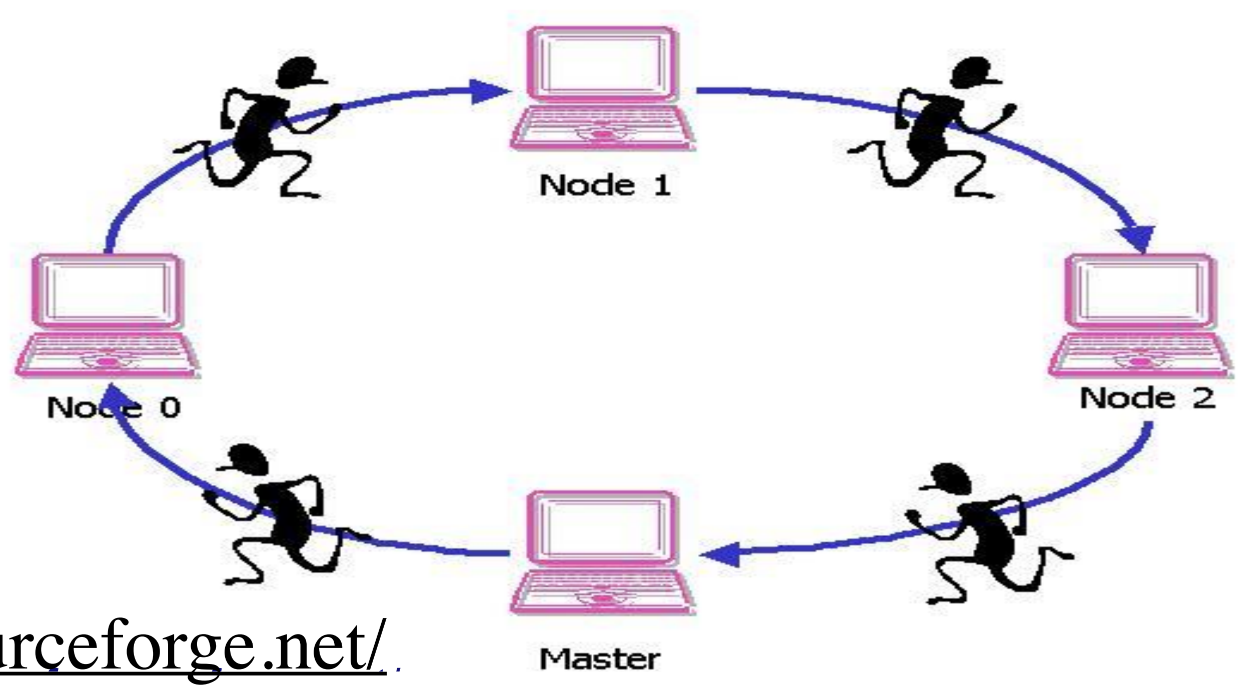
# Style: Mobile Code

- Sometimes interpretation can not be performed locally
  - Code-on-demand
  - **Remote execution/evaluation**
    - Client has code but does not have resources to execute it
      - Software resources (e.g., interpreter)
      - Or hardware resources (e.g., processing power)



# Style: Mobile Code

- Sometimes interpretation can not be performed locally
  - Code-on-demand
  - Remote execution/evaluation
  - **Mobile agent**
    - Initiator has code and some resources but not all
    - Can autonomously decide to migrate to a different node to obtain additional resources




http://maf.sourceforge.net/

# Style: Mobile Code - Security

- Code being executed might be **malicious**!
  - Privacy invasion
  - Denial of service

- **Solutions:**
  - Sandboxing
    - Mobile code runs only in a restricted environment, “sandbox”, and does not have access to vital parts of the system
  - Signing
    - Only mobile code signed by a trusted party can be executed

# Style: Mobile Code

- Code and state move to different hosts to be interpreted.
- **Components:**
  - Execution dock, compilers / interpreter.
- **Connections:**
  - Network protocols.
- **Data elements:**
  - Representations of code, program state, data.
- **Topology:**
  - Network.

# Style: Mobile Code

- **Variants:**
  - Code-on-demand, remote evaluation, and mobile agent.
- **Qualities:**
  - Dynamic adaptability. Performance (resources).
- **Typical uses:**
  - Processing large amounts of distributed data. Dynamic behaviour / customization
- **Cautions:**
  - Security challenges. Transmission/network costs.

# Implicit Invocation

- **Basic idea**
  - Event announcement instead of method invocation
  - “Listeners” register interest in and associate methods with events
  - System invokes all registered methods implicitly
- **Style invariants**
  - “Announcers” are unaware of their events’ effects
  - No assumption about processing in response to events
- **Examples**: publish-subscribe, event-based

# Implicit Invocation

- **Advantages**:
  - Component reuse
  - System evolution
  - Both at system construction-time & run-time
- **Disadvantages**:
  - Counter-intuitive system structure
  - Components relinquish computation control to the system
  - No knowledge of what components will respond to event
  - No knowledge of order of responses

# Style: Publish-Subscribe

- **Subscribers** register/deregister to receive specific messages or specific content.
- **Publishers** broadcast messages to subscribers.
- **Analogy**: newspaper subscription
  - Subscriber chooses the newspaper
  - Publisher delivers only to subscribers
  - Publisher has to maintain a list of subscribers

- Sometimes we'll need proxies to manage distribution.

# Style: Publish-Subscribe

- Subscribers register/deregister to receive specific messages or specific content. Publishers broadcast messages to subscribers either synchronously or asynchronously.
- **Components:**
  - Publishers, subscribers, proxies for managing distribution
- **Connectors:**
  - Typically a network protocol is required.  Content-based subscription requires sophisticated connectors.
- **Data Elements:**
  - Subscriptions, notifications, published information
- **Topology:**
  - Subscribers connect to publishers either directly or through intermediaries.

# Style: Publish-Subscribe

- **Qualities:**
  - Highly-efficient one-way notification with low coupling.
- **Typical uses:**
  - News, GUI programming, multi-player network-based games, social media “friending”.
- **Cautions:**
  - Scalability to large numbers of subscriber may require specialized protocols.

# Style: Event-based

- Independent components asynchronously emit and receive events.
- **Components:**
  - Event generators / consumers.
- **Connections:**
  - Event bus.
- **Data elements:**
  - Events.
- **Topology:**
  - Components communicate via bus, not directly.

# Style: Event-based

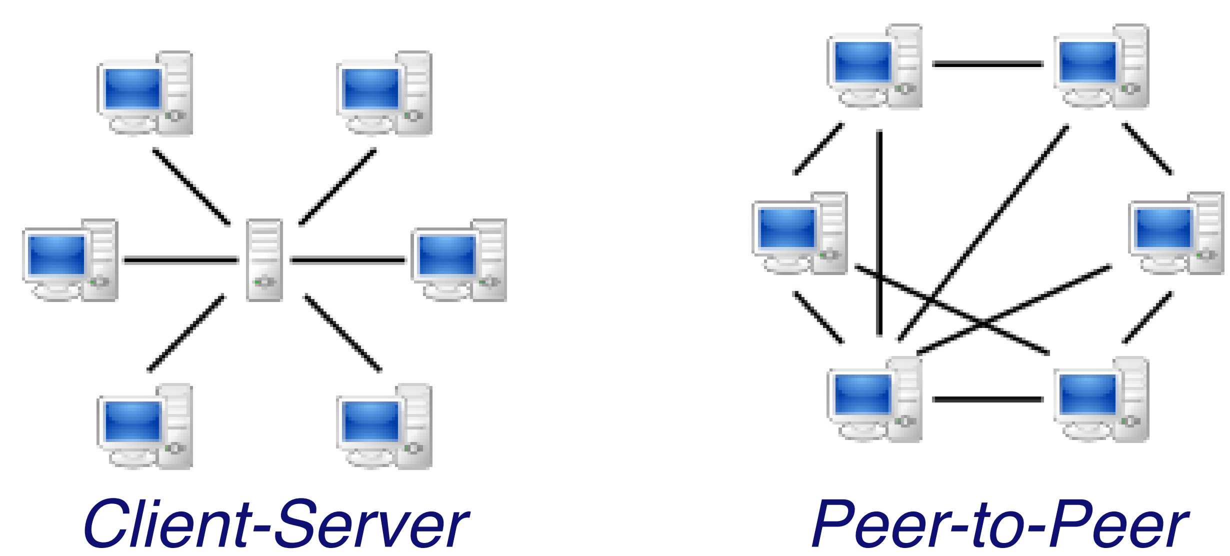
- **Qualities:**
  - Highly scalable. Easy to evolve. Effective for heterogenous applications (as long as components can communicate with the bus they can be implemented in any possible way).
- **Typical uses:**
  - User interfaces. Enterprise information systems with many independent components (e.g., financial markets, HR, production, etc.).
- **Cautions:**
  - No guarantee when the event will be processed.

# Peer to Peer

- Network of loosely-coupled peers
- Peers can act as either clients or servers
- State and logic are decentralized amongst peers
- Peers: independent components, having their own state and control thread

# Style: Peer-to-peer

- State and behaviour are distributed among peers that can act as clients or servers.
- **Components:**
  - Peers (aka independent components).
- **Connections:**
  - Network protocols.
- **Data elements:**
  - Network messages.




- **Topology:**
  - Network. Can vary arbitrarily and dynamically.

# Style: Peer-to-peer

- **Qualities:**
  - Decentralized computing. Robust to node failures. Scalable.
- **Typical uses:**
  - When informations and operations are distributed.
- **Cautions:**
  - Security (peers might be malicious or egoistic). Latency (when information retrieval time is crucial).

# Styles Summary

| Style Category & Name | Summary | Use It When | Avoid It When |
|---|---|---|---|
| ***Language-influenced styles*** | | | |
| Main Program and Subroutines | Main program controls program execution, calling multiple subroutines. | Application is small and simple. | Complex data structures needed.<br>Future modifications likely. |
| Object-oriented | Objects encapsulate state and accessing functions | Close mapping between external entities and internal objects is sensible.<br>Many complex and interrelated data structures. | Application is distributed in a heterogeneous network.<br>Strong independence between components necessary.<br>High performance required. |
| ***Layered*** | | | |
| Virtual Machines | Virtual machine, or a layer, offers services to layers above it | Many applications can be based upon a single, common layer of services.<br>Interface service specification resilient when implementation of a layer must change. | Many levels are required (causes inefficiency).<br>Data structures must be accessed from multiple layers. |
| Client-server | Clients request service from a server | Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation. | Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine capabilities rival or exceed the server's. |

# Styles Summary

| | | | |
|---|---|---|---|
| ***Data-flow styles*** | | | |
| Batch sequential | Separate programs executed sequentially, with batched input | Problem easily formulated as a set of sequential, severable steps. | Interactivity or concurrency between components necessary or desirable.<br>Random-access to data required. |
| Pipe-and-filter | Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters | [As with batch-sequential] Filters are useful in more than one application. Data structures easily serializable. | Interaction between components required. Exchange of complex data structures between components required. |
| ***Shared memory*** | | | |
| Blackboard | Independent programs, access and communicate exclusively through a global repository known as blackboard | All calculation centers on a common, changing data structure;<br>Order of processing dynamically determined and data-driven. | Programs deal with independent parts of the commondata.<br>Interface to commondata susceptible to change. When interactions between the independent programs require complex regulation. |

# Styles Summary

| | | | |
|---|---|---|---|
| ***Interpreter*** | | | |
| Interpreter | Interpreter parses and executes the input stream, updating the state maintained by the interpreter | Highly dynamic behavior required. High degree of end-user customizability. | High performance required. |
| Mobile Code | Code is mobile, that is, it is executed in a remote host | When it is more efficient to move processing to a data set than the data set to processing.<br>When it is desirous to dynamically customize a local processing node through inclusion of external code | Security of mobile code cannot be assured, or sandboxed.<br>When tight control of versions of deployed software is required. |

# Styles Summary

| | | | |
|---|---|---|---|
| ***Implicit Invocation*** | | | |
| Publish-subscribe | Publishers broadcast messages to subscribers | Components are very loosely coupled. Subscription data is small and efficiently transported. | When middleware to support high-volume data is unavailable. |
| Event-based | Independent components asynchronously emit and receive events communicated over event buses | Components are concurrent and independent.<br>Components heterogeneous and network-distributed. | Guarantees on real-time processing of events is required. |
| ***Peer-to-peer*** | Peers hold state and behavior and can act as both clients and servers | Peers are distributed in a network, can be heterogeneous, and mutually independent.<br>Robust in face of independent failures.<br>Highly scalable. | Trustworthiness of independent peers cannot be assured or managed.<br>Resource discovery inefficient without designated nodes. |

# Summary: Architectural Patterns vs. Architectural Style vs. Design Patterns

- **Architectural patterns** define the implementation strategies of those components and connectors ("how?")
  - More domain specific
- **Architectural styles** define the components and connectors ("what?")
  - Less domain specific
- Good architecture makes use of **design patterns** (on a more fine-granular level)
  - Usually domain independent

# Summary

- Different styles result in
  - Different architectures
  - Architectures with greatly differing properties
- A style does not fully determine resulting architecture
  - A single style can result in different architectures
  - Considerable room for
    - Individual judgment
    - Variations among architects
- A style defines domain of discourse
  - About problem (domain)
  - About resulting system

# In-Class Activity

- Design cuLearn in three different styles
  - What are the components, connectors, and topology?

- Decide on the right style for your app
  - Justify your choice of the style