COMP 6471
Software Design Methodologies

Fall 2011

Dr Greg Butler

http://www.cs.concordia.ca/~gregb/home/comp6471-fall2011.html

Architectural Styles and Patterns

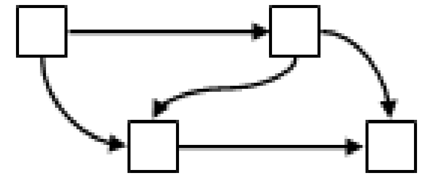
- An *architectural style* defines a family of architectures constrained by
  - **Component/connector vocabulary, e.g.,**
    - **layers and calls between them**
  - **Topology, e.g.,**
    - **stack of layers**
  - **Semantic constraints, e.g.,**
    - **a layer may only talk to its adjacent layers**
- For each architectural style, an *architectural pattern* can be defined
  - **It's basically the architectural style cast into the pattern form**
  - **The pattern form focuses on identifying a problem, context of a problem with its forces, and a solution with its consequences and tradeoffs; it also explicitly highlights the composition of patterns**

Catalogues of Architectural Styles and Patterns

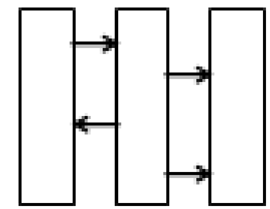
- Architectural styles
  - [Garlan&Shaw] M. Shaw and D. Garlan. *Software Architecture: Perspectives on a Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996
- Architectural Patterns
  - [POSA] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., Chichester, UK, 1996

A Classification of Software Architectures

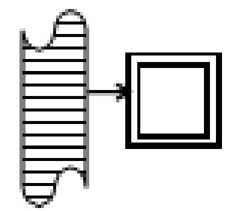
- ***Data Flow***
  - o **Data flowing between functional elements**



- ***Independent Components***
  - o **-- executing in parallel, occasionally communicating**

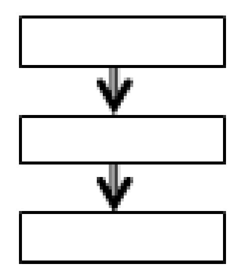


- ***Virtual Machines***
  - o **Interpreter + program in special-purpose language**



- ***Repositories***
  - o **Primarily built around large data collection**

- ***Layered***
  - o **Subsystems, each depending one-way on another subsystem**



“Pure” Form of Styles

- When we introduce a new style, we will typically first examine its “pure” form.
  - **Pure data flow styles (or any other architectural style) are rarely found in practice**
  - **Systems in practice**
    - **Regularly deviate from the academic definitions of these systems**
    - **Typically feature many architectural styles simultaneously**
  - **As an architect you must understand the “pure” styles to understand the strength and weaknesses of the style as well as the consequences of deviating from the style**

Data Flow

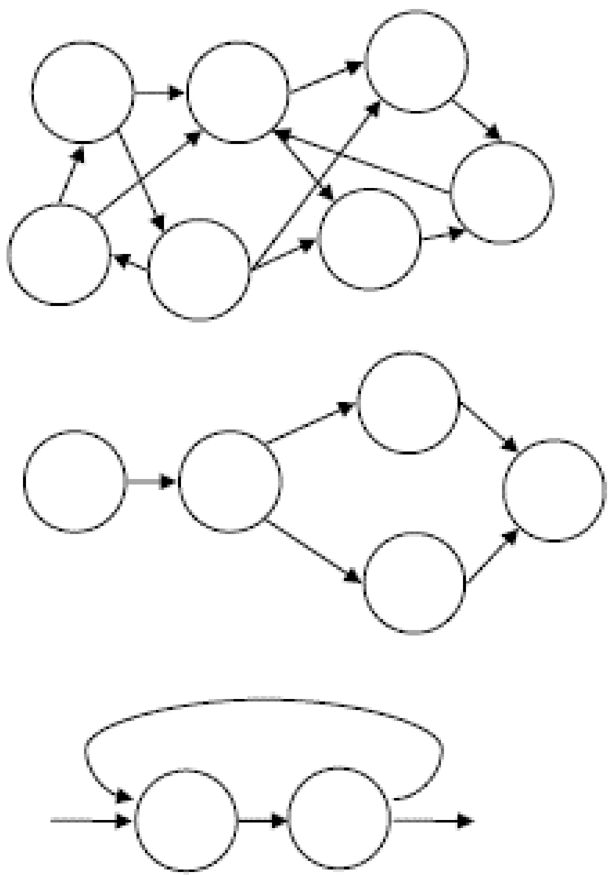
- A data flow system is one in which:
  - **The availability of data controls the computation**
  - **The structure of the design is determined by the orderly motion of data from component to component**
  - **The pattern of data flow is explicit**
  - **This is the only form of communication between components**

- There are variety of variations on this general theme:
  - **How control is exerted (e.g., push versus pull)**
  - **Degree of concurrency between processes**
  - **Topology**

Data Flow

- Components: Data Flow Components
  - **Interfaces are input ports and output ports**
  - **Input ports read data; output ports write data**
  - **Computational model: read data from input ports, compute, write data to output ports**
- Connectors: Data Streams
  - **Uni-directional**
    - **Usually asynchronous, buffered**
  - **Interfaces are reader and writer roles**
  - **Computational model: transport data from writer roles to reader roles**
- Systems
  - **Arbitrary graphs**
  - **Computational model: functional composition**

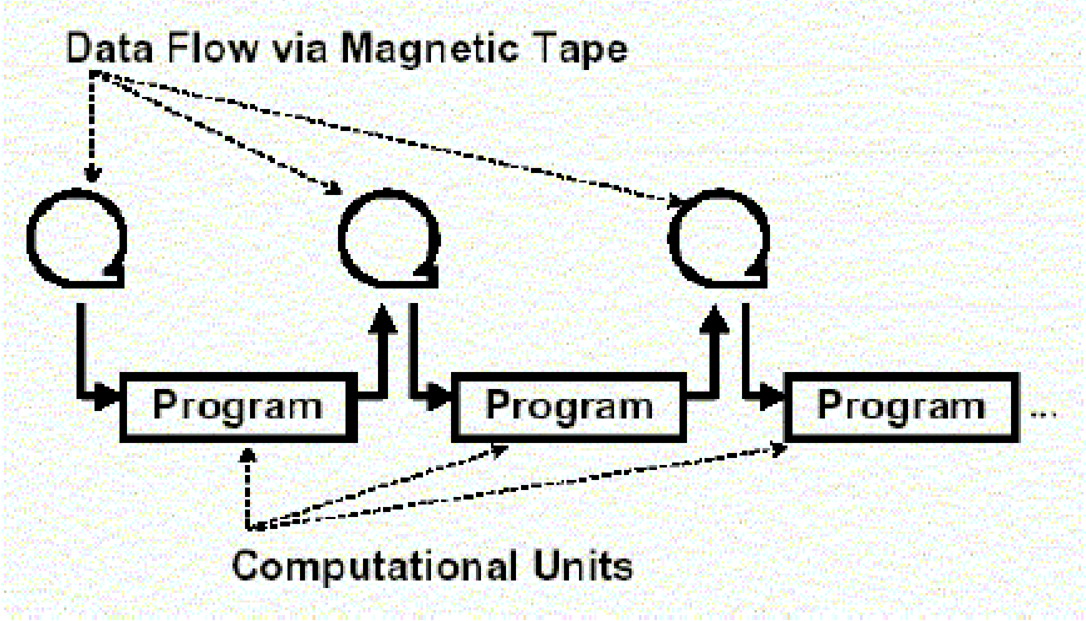
Patterns of Data Flow in Systems



- Data can flow in arbitrary patterns
- Primarily we are interested in linear data flow patterns
- ...or in simple, constrained cyclical patterns...

Characteristics of Batch Sequential Systems

- Components (processing steps) are independent programs
- Connectors are some type of media - traditionally magnetic tape
- Each step runs to completion before the next step begins

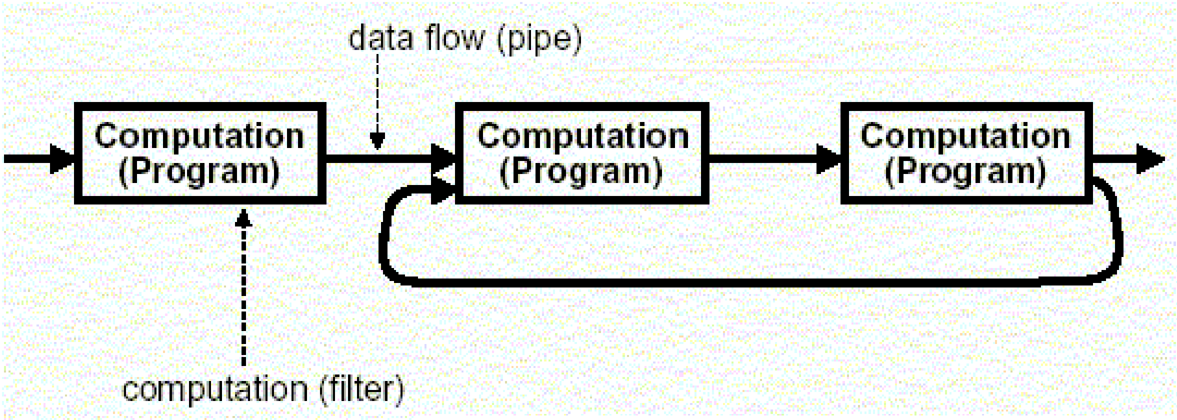


Characteristics of Batch Sequential Systems

- History
  - **Mainframes and magnetic tape**
  - **Limited disk space**
  - **Block scheduling of CPU processing time**
- Business data processing
  - **Discrete transactions of predetermined type and occurring at periodic intervals**
  - **Creation of periodic reports based on data periodic data updates**

Pipes and Filters

- The tape of the batch sequential system, morphed into a language and operating system construct
- Compared to the batch-sequential style, data in the pipe&filter style is processed *incrementally*



- “The Pipes and Filters architectural pattern [style] provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.” [POSA p53]

- Components (Filters)
  - **Read streams of data on input producing streams of data on output**
  - **Local incremental transformation to input stream (e.g., filter, enrich, change representation, etc.)**
  - **Data is processed as it arrives, not gathered then processed**
  - **Output usually begins before input is consumed**
- Connectors (Pipes)
  - **Conduits for streams, e.g., first-in-first-out buffer**
  - **Transmit outputs from one filter to input of other**

- Invariants
  - **Filters must be independent, no shared state**
  - **filters don't know upstream or downstream filter identity**
  - **Correctness of output from network must not depend on order in which individual filters provide their incremental processing**

- Common specializations
  - **Pipelines: linear sequence of filters**
  - **Bounded and typed pipes …**

Example Pipe-and-Filter Systems

- lex/yacc-based compiler (scan, parse, generate code, ..)
- Unix pipes
- Image processing
- Signal processing
- Voice and video streaming
- …

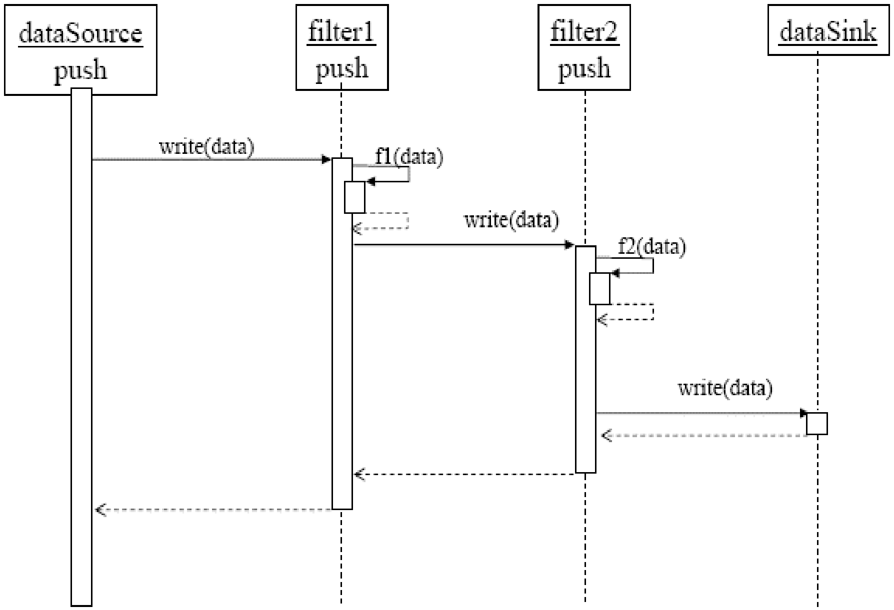
Data Pulling and Data Pushing

- What is the force that makes the data flow?

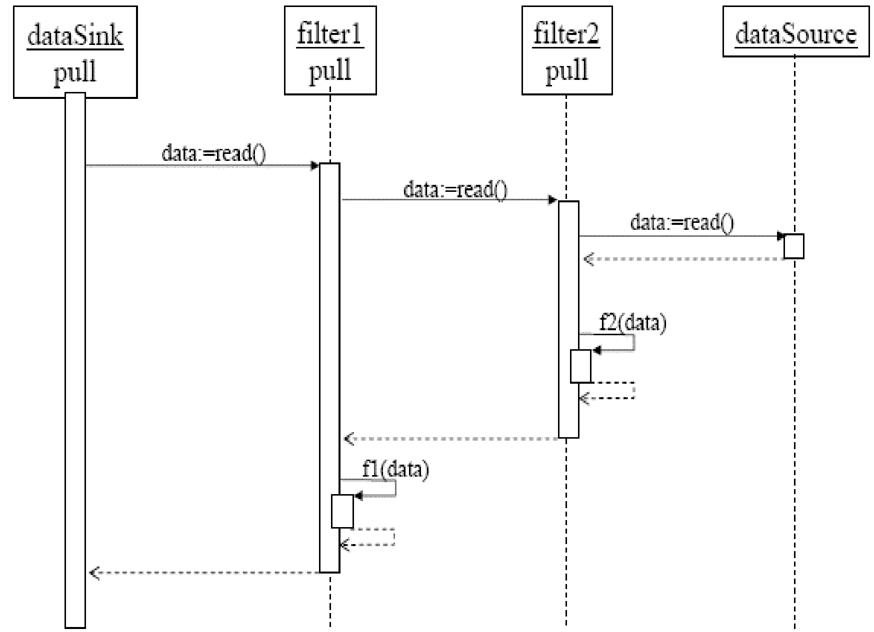
- Four choices:
  - **Push: data source pushes data in a downstream direction**
  - **Pull: data sink pulls data from an upstream direction**
  - **Push/pull: a filter is actively pulling data from a stream, performing computations, and pushing the data downstream**
  - **Passive: don't do either, act as a sink or source for data**

- Combinations may be complex and may make the "plumber's" job more difficult
  - **if more than one filter is pushing/pulling, synchronization is needed**

A Push Pipeline With an Active Source



A Pull Pipeline With an Active Sink



Pipe and Filter: Strengths

- Overall behaviour is a simple composition of behaviour of individual filters.
- Reuse - any two filters can be connected if they agree on that data format that is transmitted.
- Ease of maintenance - filters can be added or replaced.
- Prototyping e.g. Unix shell scripts are famously powerful and flexible, using filters such as sed and awk.
- Architecture supports formal analysis - throughput and deadlock detection.
- Potential for parallelism - filters implemented as separate tasks, consuming and producing data incrementally.

Pipe and Filter: Weaknesses

- Can degenerate to 'batch processing' - filter processes all of its data before passing on (rather than incrementally).
- Sharing global data is expensive or limiting.
- Can be difficult to design incremental filters.
- Not appropriate for interactive applications - doesn't split into sequential stages. POSA book has specific styles for interactive systems, one of which is Model-View-Controller.
- Synchronisation of streams will constrain architecture.
- Error handling is Achilles heel e.g. filter has consumed three quarters of its input and produced half its output and some intermediate filter crashes! Generally restart pipeline. (POSA)
- Implementation may force lowest common denominator on data transmission e.g. Unix scripts everything is ASCII.

Pipe-and-Filter vs. Batch Sequential

- Both decompose the task into a fixed sequence of computations (components) interacting only through data passed from one to another

Batch Sequential	**Pipe-and-Filter**
- course grained - high latency - external access to input - no concurrency - non-interactive	- fine grained - results starts processing - localized input - concurrency possible - interactive awkward but possible

Call-and-return

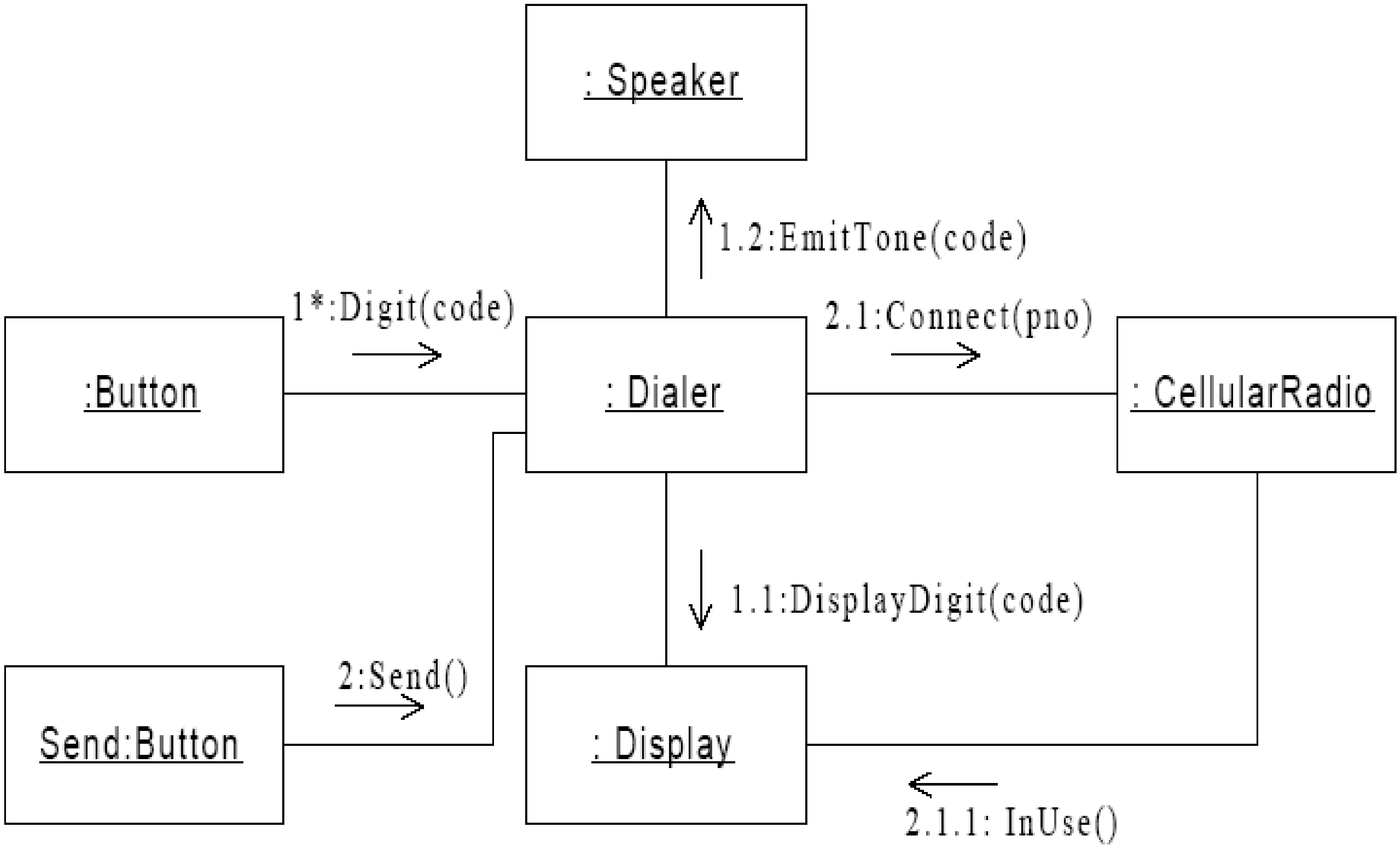
- Main program/subroutines
- Information hiding
  - **ADT, object, naive client/server**

Main Program + Subroutine Architecture

- Classic style since 60s - pre-OO.
- Hierarchical decomposition into subroutines (Components) each solving a well defined task/function.
- Data passed around as parameters.
- Main driver provides a control loop for sequencing through subroutines.

Data Abstraction / Object Oriented

- Widely used architectural style
- Components:
  - **Objects or abstract data types**
- Connections:
  - **Messages or function/procedure invocations**
- Key aspects:
  - **Object preserves integrity of representation - no direct access**
  - **Representation is hidden from objects**
- Variations:
  - **Objects as concurrent tasks**
  - **Multiple interfaces for objects (Java !)**
- Note that Data Abstraction is different from Object-Oriented - no inheritance.



Components: Classes & Objects
Connectors: Method calls

Data Abstraction & OO: Strengths

- Naturally supports information hiding, which shields implementation changes from clients
- Encapsulation and information hiding reduce coupling

=> Enhances maintainability

- Allows systems to be modeled as collection of collaborating objects

=> can be an effective means of managing system complexity

Data Abstraction & OO: Weaknesses

- Object identity must be known for method invocation

=> Identity change of an object affects all calling objects

  - Contrast this to pipe-and-filter …

- Concurrency problems through concurrent access

Object-Oriented Strengths/Weaknesses

- Strengths:
  - **Change implementation without affecting clients (assuming interface doesn’t change)**
  - **Can break problems into interacting agents (distributed across multiple machine / networks).**
- Weaknesses:
  - **To interact objects must know each other’s identity (in contrast to Pipe and Filter).**
  - **When identity changes, objects that explicitly invoke it must change (Java interfaces help though).**
  - **Side effect problems: if A uses B and C uses B, then C effects on B can be unexpected to A (and vice-versa).**
  - **Complex dynamic interactions – distributed functionality.**

Implicit Invocation

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more Events.

Other components in the system can register an interest in an event by associating a procedure with the event.

When the event is announced the system itself invokes all of the procedures that have been registered for the event.

Thus an event announcement ``implicitly'' causes the invocation of procedures in other modules.

Implicit Invocation Example

- Components register interest in an event by associating a procedure with the event.
- When the event is announced the system implicitly invokes all procedures that have been registered for the event.
- Common style for integrating tools in a shared environment, e.g.,
  - **Tools communicate by broadcasting interesting events**
  - **Other tools register patterns that indicate which events should be routed to them and which method/procedure should be invoked when an event matches that pattern.**
  - **Pattern matcher responsible for invoking appropriate methods when each event is announced.**
- Examples:
  - **Editor announces it has finished editing a module, compiler registers for such announcements and automatically re-compiles module.**
  - **Debugger announces it has reached a breakpoint, editor registers interest in such announcements and automatically scrolls to relevant source line.**

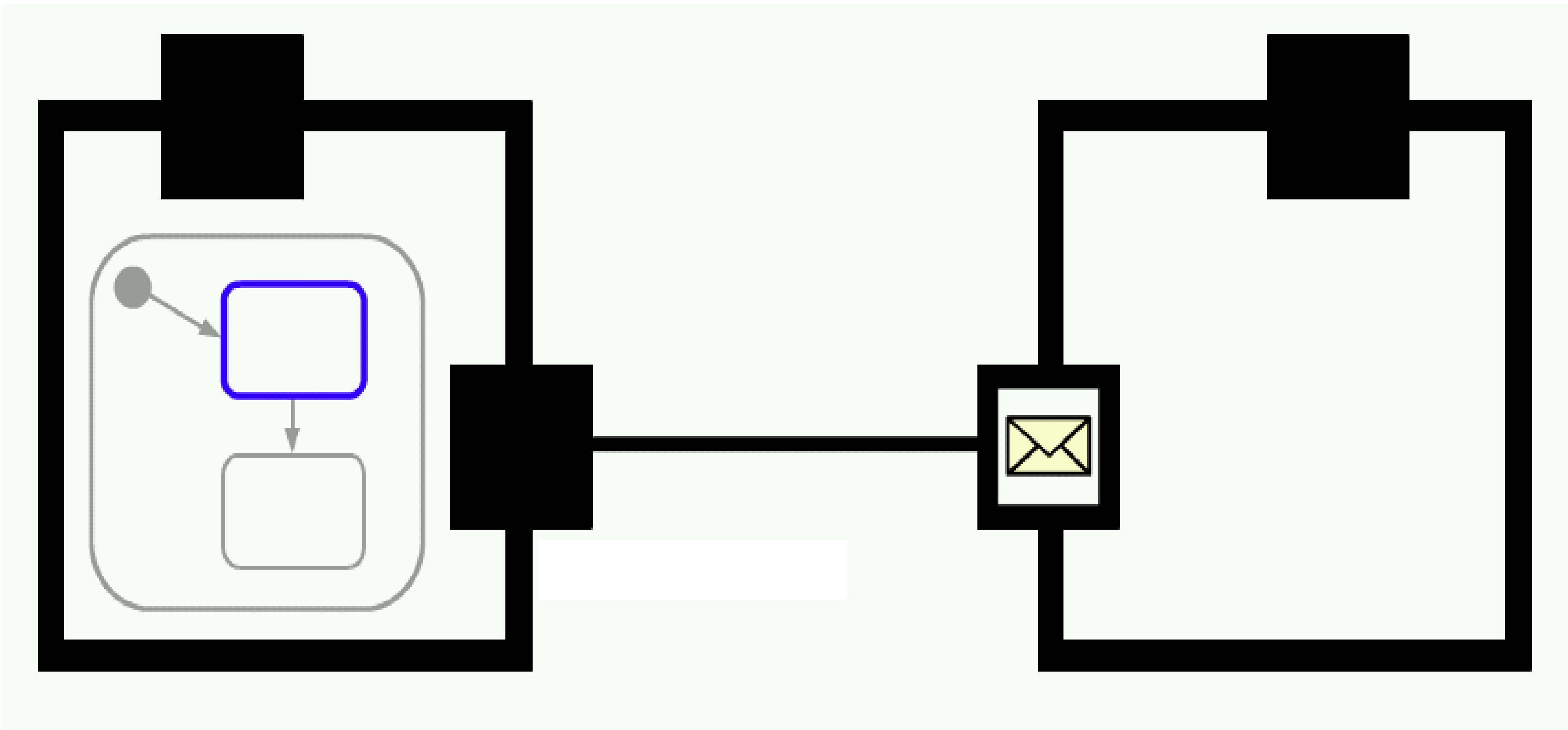
Implicit Invocation

- Strengths
  - **Strong support for reuse - plug in new components by registering it for events**
  - **Maintenance - add and replace components with minimum affect on other components in the system.**

- Weaknesses
  - **Loss of control**
    - **when a component announces an event, it has no idea what components will respond to it**
    - **cannot rely on order that these components will be invoked**
    - **cannot tell when they are finished**
  - **Ensuring correctness is difficult because it depends on context in which invoked. Unpredictable interactions.**
  - **Sharing data - see the Observer Design Pattern**

- Hence explicit invocation is usually provided as well as implicit invocation. In practice architectural styles are combined.

Event-Driven Architecture Style



Event-Driven: Communication protocols

- Synchronous communication is direct,
 time synchronized. This means that all parties involved in the communication are present at the same time.
  - Examples are: A telephone conversation (not texting), a company board meeting, a chat room event and instant messaging.
  -
- Asynchronous communication does not require that all parties involved in the communication to be present at the same time.
  - Examples are: e-mail messages, discussion boards, blogging, and text messaging over cell phones.

Event-Driven Architecture

- Component: (active or passive) object, capsule, module
  - Can be an instance of a class, an active class, or simply a module.
  - Interface provides methods and ports.
  - Publisher: individual components announce data that they wish to share with their subscribers.
  - Subscriber: individual components register their interest for published data.
- Connector: “connector”, channel, binding, callback.
  - Offers one-to-one, one-to-many, many-to-one connections;
  - Asynchronous event broadcast.
  - Synchronous event broadcast & await reply (call-and-return)

- Components do not explicitly invoke each other.
- Components generate *signals*, also called *events*.
- To receive events, objects can
  - Receive events at ports (statically or dynamically bound).
  - Register for event notification (e.g. via callback).
- Announcers do not know which components will be affected by thrown events
- System framework implements signal propagation

Event-Driven Architecture

Strengths

- Supports reuse
  - Only little coupling
- Easy system evolution
  - Introduction of new component simply by registering
- Well suited for asynchronous communication
-

Weaknesses

- Components don't have control over computation since they can only generate events; the run-time system handles event dispatching. Thus Respondents to events are not ordered.
- Exchange of data can require use of global variables or shared repository
  => resource management can become a challenge.
- Global system analysis is more challenging.
- Asynchronous event handling
- Contrast to explicit call & use of pre-, post-conditions. E.g. how to ensure that at least one object has processed an event.

Event-Driven Architecture: Examples

- UIs
  - Macintosh computers popularized the “main event loop” approach for UI applications.

- Other examples include
  - Constraint satisfaction systems (e.g. some database systems).
  - Daemons.
  - S/W environments that make use of multiple tools: e.g. text editor registers for events from debugger.

Model-View-Controller

- A decomposition of an interactive system into three components:
  - **A model containing the core functionality and data,**
  - **One or more views displaying information to the user, and**
  - **One or more controllers that handle user input.**
- A change-propagation mechanism (i.e., observer) ensures consistency between user interface and model, e.g.,
  - **If the user changes the model through the controller of one view, the other views will be updated automatically**
- Sometimes the need for the controller to operate in the context of a given view may mandate combining the view and the controller into one component
- The division into the MVC components improves maintainability

Data-Oriented Repository

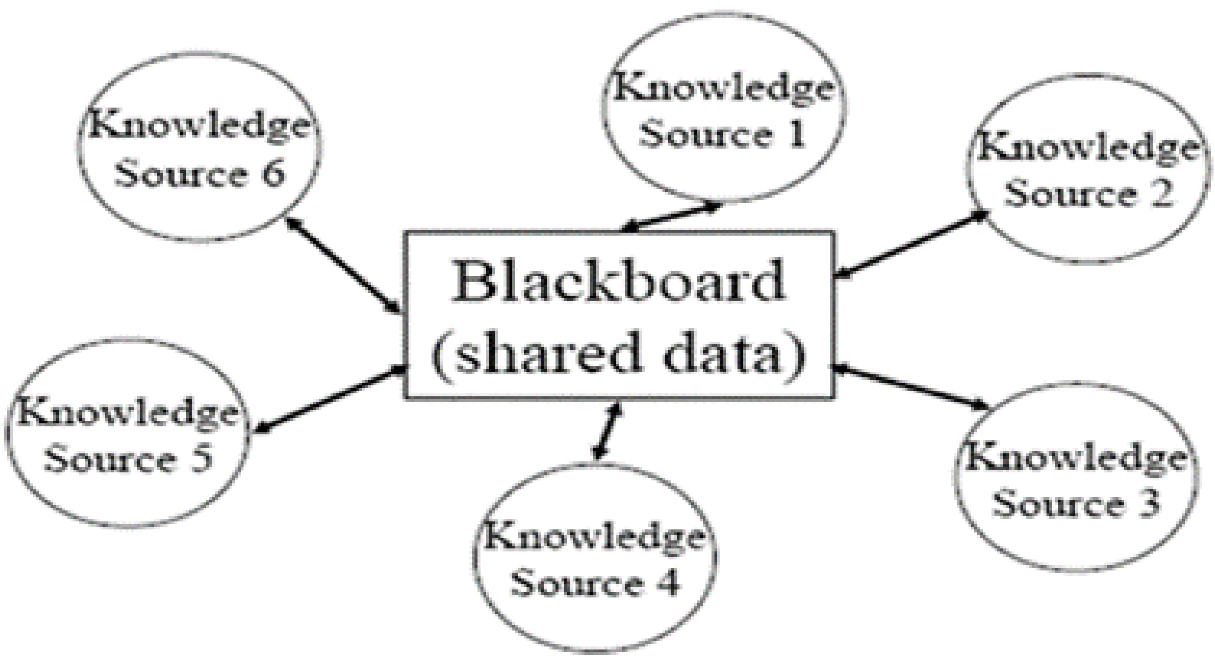
- Transactional databases
  - **True client/server**
- Blackboard
- Modern compiler

Repositories / Data Centred

- Characterised by a central data store component representing systems state and a collection of independent components that operate on the data store.
- Connections between data store and external components vary considerably in this style:
  - *Transactional databases*: **Incoming stream of transactions trigger processes to act on data store. Passive.**
  - *Blackboard architecture*: **Current state of data store triggers processes. Active.**

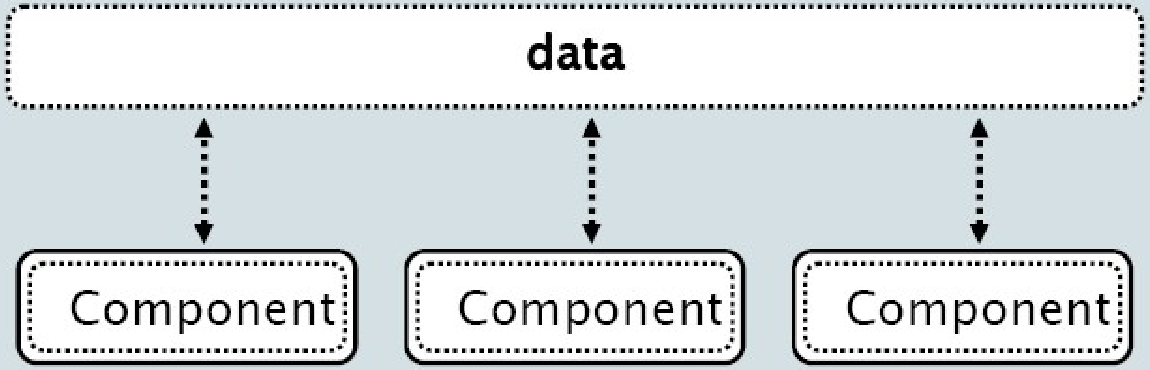
Blackboard

- Characteristics: cooperating 'partial solution solvers' ***collaborating but not following a pre-defined strategy.***
- Current state of the solution stored in the blackboard.
- Processing triggered by the state of the blackboard.



Blackboard Style (1)

Concept: Concurrent transformations on shared data



Components: processing units (typically knowledge source)

Connectors: blackboard
interaction style: asynchronous

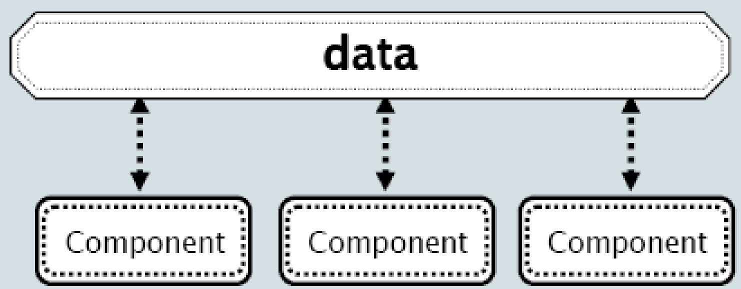
Topology: one or more transformation-components may be connected to a data-space, there are typically no connections between processing units (bus-topology)

Blackboard Style (2)

Behaviour Types:

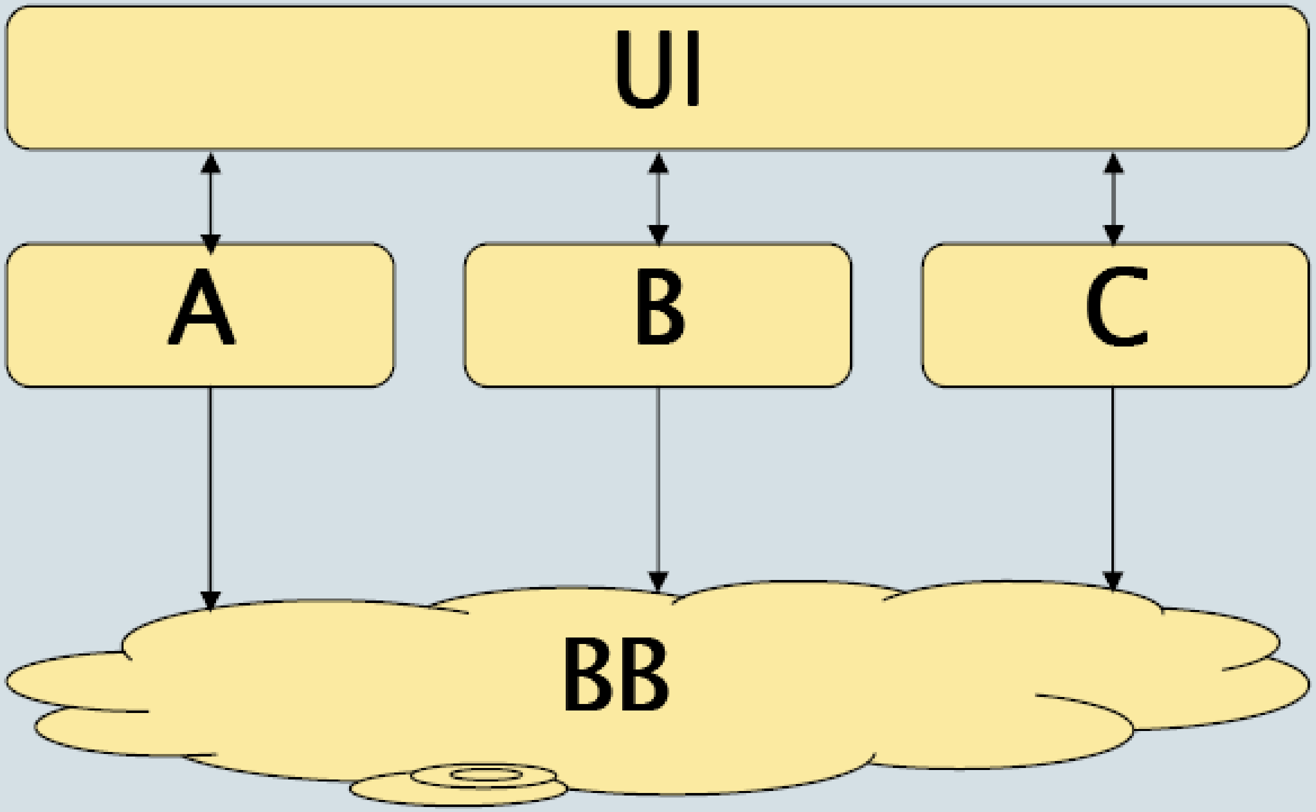
a. **Passive repository**
Accessed by a set of components; e.g. database or server

b. **Active repository**
Sends notification to components when data of interest changes; e.g. blackboard or active database



Constraints:
Consistency of repository: Various types of (transaction) consistency

Layering & Blackboard



Blackboard Style (3)

Advantages:

- Allows different control heuristics
- Reusable & heterogeneous knowledge sources
- Support for fault tolerance and robustness by adding redundant components

+/- Dataflow is not directly visible

Disadvantages

- Distributed implementation is complex
  - distribution and consistency issues

Blackboard Characteristics

- Data may be structured (DB) or unstructured
- Data may be selected based on content
- Applications may insert/retrieve different data-type per access.

  This in contrast to pub-sub where data of the same type is retrieved repeatedly

Blackboard Style (4) Quality Factors

Extensibility: components can be easily added

Flexibility: functionality of components can be easily changed

Robustness: + components can be replicated,
- blackboard is single point of failure

Security: - all process share the same data
+ security measures can be centralized around blackboard

Performance: easy to execute in parallel fashion
consistency may incur synchroniz.-penalty

Blackboard Style (5) Application Conte

Rules of thumb for choosing blackboard (o.a. from Shaw):

- if representation & management of data is a central issue
- if data is long-lived
- if order of computation
  - can not be determined a-priori
  - is highly irregular
  - changes dynamically
- if units of different functionality (typically containing highly specialized knowledge) concurrently act on shared data (horizontal composition of functionality)

Example application domain: expert systems

Examples of Blackboard Architectures

- Problems for which no deterministic solution strategy is known, but many different approaches (often alternative ones) exist and are used to build a partial or approximate solution.
  - **AI: vision, speech and pattern recognition (see POSA case study)**
  - **Modern compilers act on shared data: symbol table, abstract syntax tree (see Garlan and Shaw case study)**

- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - ➔ **Data-sharing**
  - **Hierarchical**
  - **Other**

Data-sharing

- Compound documents
- Hypertext
- Fortran COMMON
- LW processes

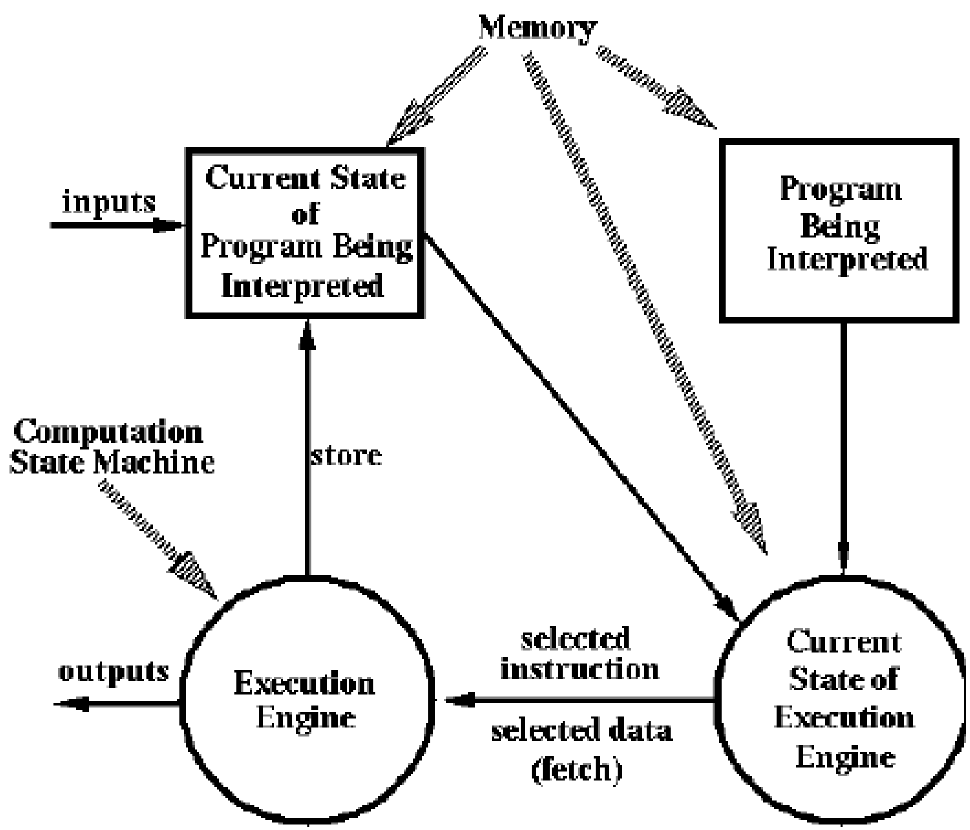
Layered Systems

- Applicability
  - **A large system that is characterised by a mix of high and low level issues, where high level issues depend on lower level ones.**
- Components
  - **Group of subtasks which implement a 'virtual machine' at some layer in the hierarchy**
- Connectors
  - **Protocols / interface that define how the layers will interact**
- Invariants
  - **Limit layer (component) interactions to adjacent layers (in practice this may be relaxed for efficiency reasons)**
- Typical variant relaxing the pure style
  - **A layer may access services of all layers below it**
- Common Examples
  - **Communication protocols: each level supports communication at a level of abstraction, lower levels provide lower levels of communication, the lowest level being hardware communications.**

Interpreter

- Architecture is based on a virtual machine produced in software.
- Special kind of a layered architecture where a layer is implemented as a true language interpreter.
- Components are ‘program’ being executed, its data, the interpretation engine and its state.
- Example: Java Virtual Machine. Java code translated to platform independent bytecodes. JVM is platform specific and interprets (or compiles - JIT) the bytecodes.

Interpreter



Interpreter – More Examples

- Programming and scripting languages
  - Awk, Perl, …
- Rule-based systems
  - Prolog, Coral, …
- Micro-coded machine
  - Implement machine code in software
- Presentation package
  - Display a graph, by operating on the graph

Distributed Peer-to-Peer Systems

- Components
  - Independently developed objects and programs offering public operations or services
- Connectors
  - Remote procedure call (RPC) over computer networks
- Configurations
  - Transient or persistent connections between cooperating components
- Underlying computational model
  - Synchronous or asynchronous invocation of operations or services
- Stylistic invariants
  - Communications are point-to-point

Heterogeneous Architectures

- In practice the architecture of large-scale system is a combination of architectural styles:
  - **('Hierarchical heterogeneous') A Component in one style may have an internal style developed in a completely different style (e.g, pipe component developed in OO style, implicit invocation module with a layered internal structure, etc.)**
  - **('Locational heterogeneous') Overall architecture at same level is a combination of different styles (e.g., repository (database) and mainprogram-subroutine, etc.)**
    Here individual components may connect using a mixture of architectural connectors - message invocation and implicit invocation.
  - **('Perspective heterogeneous') Different architecture in different perspectives (e.g., structure of the logical view, structure of the physical view, etc.)**

Example of Heterogeneous Architectures: Enterprise Architectures

- Multi tier (at the highest level), distributed (including broker pattern), transactional databases, event-based communication, implicit invocation, object-oriented, MVC (e.g., for presentation in the client), dataflow for workflow, etc.