Survey of Architectural Styles

Alexander Bird, Bianca Esguerra, Jack Li Liu, Vergil Marana,
Jack Kha Nguyen, Neil Oluwagbeminiyi Okikiolu, Navid Pourmantaz
Department of Software Engineering
University of Calgary
Calgary, Canada

Abstract— **In software engineering, an architectural style is a highest-level description of an accepted solution to a common software problem. This paper describes and compares a selection of nine accepted architectural styles selected by the authors and applicable in various domains. Each pattern is presented in a general sense and with a domain example, and then evaluated in terms of benefits and drawbacks. Then, the styles are compared in a condensed table of advantages and disadvantages which is useful both as a summary of the architectural styles as well as a tool for selecting a style to apply to a particular project. The paper is written to accomplish the following purposes: (1) to give readers a general sense of several architectural styles and when and when not to apply them, (2) to facilitate software system design by by providing a reference tool for comparing the architectural styles, and (3) to be a starting point reference for implementing architectural styles by providing a description and references for further reading. After using this paper to select one or more architectural styles as candidates for a software system, readers are recommended to follow the references used in the style overviews to perform a more in-depth analysis of the style's applicability to the problem at hand. It is offered here with the hope of equipping software designers for innovation building on the foundation of architectural best practices.**

Keywords—Software Architecture; Software Architectural Styles; Patterns; Model-View-Controller; MVC; Presentation-Abstraction-Control; PAC; Pipes & Filters; Broker; Layers; Blackboard; Reactor; Interceptor; Microkernel;

I. Introduction

An architectural style, sometimes called an architectural pattern, is a set of principles that provide an abstract framework for a family of systems. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. They are what shape applications and an understanding of architectural styles provides many expected and unexpected benefits. The most important being that they provide a common language for designing software, allowing design teams to focus on the particulars of the problem at hand. They also facilitate having design conversations that are technology agnostic. This makes way for a higher level of conversation that is inclusive of patterns and principles, without getting into specifics **[30]**.

The architecture of a software system is almost never limited to a single architectural style, but is often an adaptation of one or more architectural styles that make up the complete system. Specializations must be made for the specific problem being solved. Many factors will influence the architectural styles chosen such as the capacity of an organization for design and implementation; the capabilities and experience of developers; and the infrastructure and organizational constraints **[30]**. These styles are not presented as out-of-the-box solutions, but rather a framework within which a specific software design may be made. If one were to say "that cannot be called a layered architecture because the such and such a layer must communicate with an object other than the layer above and below it" then one would have missed the point of architectural styles and design patterns. They are not intended to be definitive and final, but rather suggestive and a starting point, not to be used as a rule book but rather a source of inspiration.

The architectural styles to be discussed in this paper are listed below in the same order as they occur in the paper. The descriptions here are intended to give a rough sense of the purpose and structure of each style:

Layered Architecture: for many software systems where reusability and abstraction are important factors, the Layered architectural style is beneficial by dividing software functionality into distinct layers that interact vertically.

Pipe & Filter Architecture: a data flow system for simple data processing systems in which developers prefer little effort during the implementation phase but require many different functionalities. Perfect for systems that require many different types of data input and output in which the designers can easily analyze the system and parse data into separate packets.

Broker Architecture: another data flow system similar to Pipe & Filter architecture but with more complexity in the implementation. For developers that want to separate the server side and hide their server locations from users and local components.

Blackboard Architecture: for systems that need to integrate large and diverse specialized modules or implement non deterministic control strategies, the blackboard system is useful by dividing the core functionalities of a system into three main components: The knowledge sources, the blackboard and the control component.

Interceptor Architecture: allows functionality to be transparently added or removed from an application.

Microkernel Architecture: for software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific components.

Reactor Architecture: allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

Model-View-Controller Architecture: an interactive system that separates the user interface and functionality by simply dividing the whole system into three large modules: Model, View, and Controller.

Presentation-Abstraction-Control Architecture: another interactive system that focuses on separating the user interface and functionality by dividing the core functionalities of a system into many small modules that each consists of three components: Presentation, Abstraction, and Control.

II. Architectural Styles / Patterns

A. Layers

In software architecture, Layers is an architectural pattern in which the software being developed is organized into a hierarchical design. The layered architectural pattern is not a new concept in software development, and it describes the most widespread principle of architectural subdivision.

1) Introduction & Description

In this pattern, a software application is decomposed into groups of subtasks and each group of subtasks is at a particular level of abstraction. Each layer acts as either a server to the layer above or a client to the layer below. Moreover, each layer can only communicate with the ones immediately above it and below it.

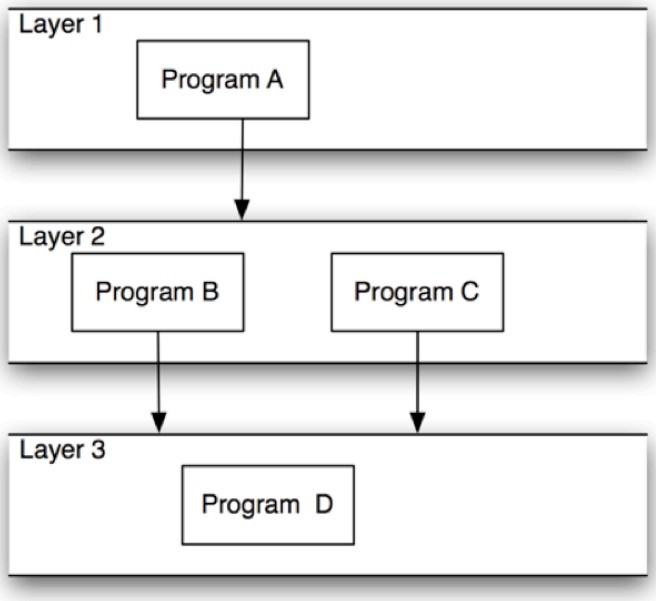
There are three main layers or components of the Layer pattern: the Data Access Layer, the Business Logic Layer, and the Graphical User Interface or Presentation Layer. As the development of a software application becomes more complex, additional layers may be added, however these three layers often serve as the core layers of this pattern.

The Data Access Layer is the layer in which data is processed. At this layer, there are operations such as Create, Retrieve, Update and Delete. The main purpose of the Data Access Layer is to provide a uniform input/output mechanism for the layer above it, despite wherever the data is coming from.

The second core layer, the Business Logic Layer, is where business logic occurs. This layer controls an application's functionality by performing detailed processing such as handling information exchange between a database and a user interface.

The final core layer of a layered architecture is the Graphical User Interface Layer or GUI. Sometimes called the Presentation Layer, this layer serves as the only visible part of a software application. It is the topmost part of the application and displays information to the user. To do this, the GUI must communicate and collect data from the other tiers, such as the Business Logic Layer. It must then wait for a success or failure message to return, at that which point it can display a visual feedback that the action has been processed **[1]**.

Figure 1. General Representation of Layered Architecture **[2]**



2) Advantages

- Division into layers promotes a separation of concerns and modularity
- The exposed workflow suits it well for incremental development
- Since each layer is easy to exchange with another layer, a layered system helps with maintainability

3) Disadvantages

- Since all upper layers rely on the lower layers, the pattern reduces efficiency
    - This is because all relevant data must pass through a number of intermediated layers, sometimes being transformed at each layer
- Dividing responsibility between layers can be difficult, especially restricting communication to adjacent layers and reducing coupling between layers**[3]**

4) Usage

The layered pattern is a popular architecture and can be applied to most systems. The layered architectural style is useful if you have existing layers that are suitable for reuse in other systems, or if the application being developed is complex and the high-level design demands separation so that development teams can focus on different areas of functionality.

An example of where a layered architecture might be used is in networking protocols. In a networking protocol system,

there are multiple layers/scenarios such as the physical layer, data link layer, and the network layer. Each layer is connected to one another, and all are contributing to how a computer communicates across machine boundaries. In the figure 3 below it is shown how the format, contents, and meaning of all messages are defined. The network protocol specifies agreements at a variety of abstraction levels, ranging from bit transmission to high-level application logic.

The layers pattern is still widely used in several applications such as Virtual Machines like JVM **[3]**, APIs, and communication protocols such as TCP/IP0**[3].**

It is easier to visualize a layered architecture by composing a sequence chart, in which each scenario is described in detail.

Figure 2. Abstraction levels of Layered Architecture **[20]**

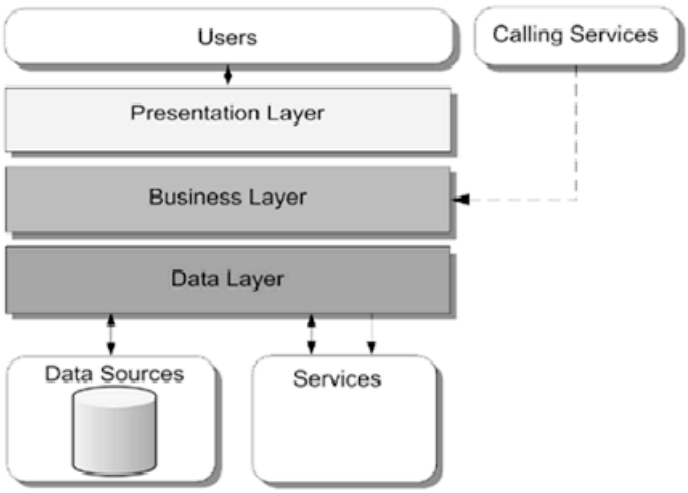


Figure 3. Layered Architecture Represented in a Sequence **[3]**

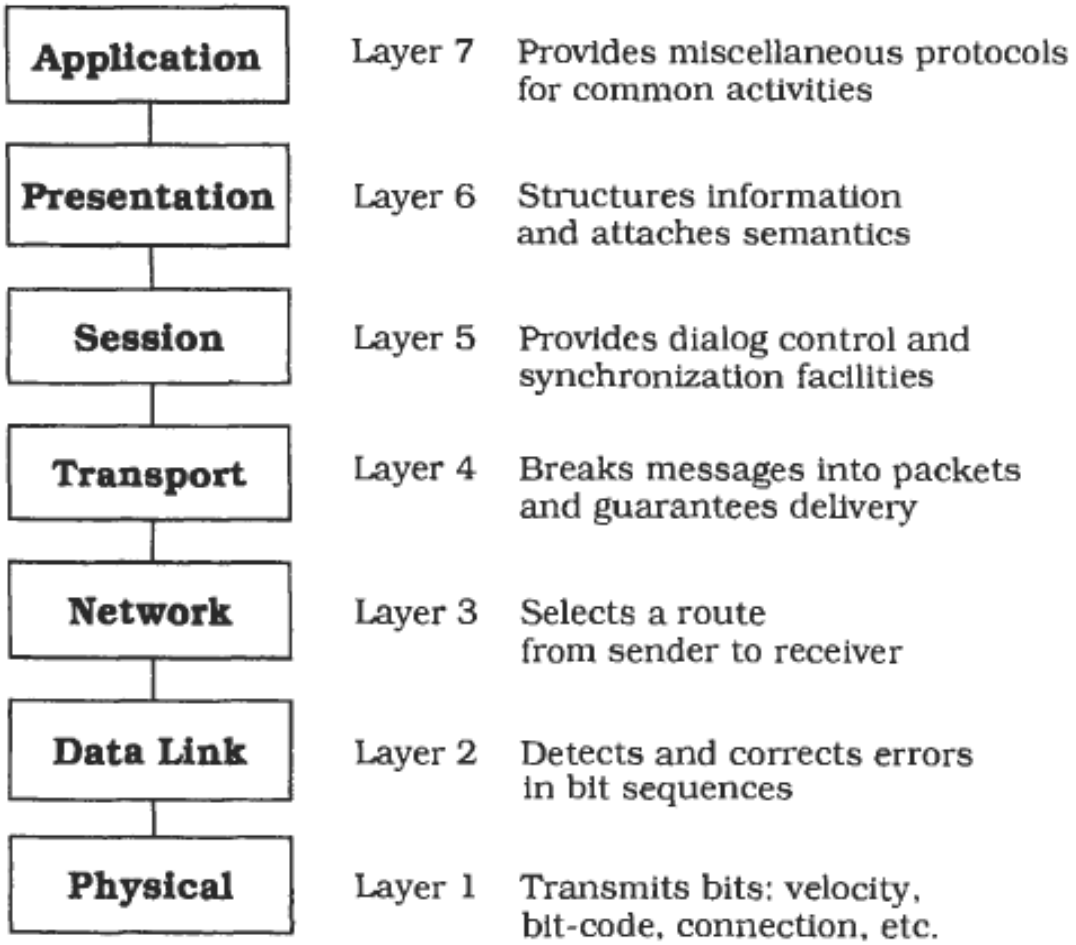


Figure 4. Pipe and Filter Architecture (Contributors' Diagram)

B. Pipes & Filter

The Pipe & Filter architecture is named for the layout of its components and is a simple architecture to apply to software systems. The pipes, from the name of the style, are connectors that transfer data and information from one component to another. The pipes hold the data and pass it through to the next filter when it is ready to process the information. The information processing at each node occurs by local transformation of input streams with an incremental computation of output, meaning output begins before the input is fully consumed **[8]**, which is the reason that the system components are called 'filters'. The pipes may be bounded or unbounded and together with the filters, they may be connected to form a generic topology **[9]**.

1) Introduction & Description

This software architecture is easy to understand and implement, and it is powerful if used correctly. The essential characteristic of each filter component is that it is completely independent of the rest of the system. It may specify the input format, and guarantees the output given certain inputs, but the correctness of the output must be unaffected by the order that the system of filters is placed together **[9]**.

The Pipe & Filter architecture is a type of Data Flow architecture, which can be characterized by the statements: instructions are executed independently (and thus can be executed concurrently or in any order) and data is consumed by an actor (in this case, the filter), which in turn yields data to the next actor**[10]**. The filters can have any number of input pipes and any number of output pipes. As mentioned earlier, the filters are independent of each other so the state of each filter is not shared with any other filter. The pipeline architecture that is commonly used in software systems is a specialization of the pipe and filter architecture. A description of the filter's function is given in the article: *An Introduction to Software Architecture* **[8]**, as components which read streams of data from inputs, producing streams of data to outputs and delivering a complete instance of the result in a standard order**[8]**. The general structure of the architecture is displayed in the figure 4., below with just a basic direct downstream flow behavior.

The architecture is not limited to the type of flow as in the diagram above, but can also include additional pipes that flow from one pipe in a loop backwards into a previous pipe for conditional processing. Common specializations of this style include *Pipelines*, restricting the topologies to a linear sequences of filters, *Bounded Pipes*, which restrict the amount of data that can reside on a pipe, and *Typed Pipes*, that require the data passed between two filters have a well-defined type **[8]**.

2) Advantages

- The style is conceptually simple, making the design task straightforward **[8]**
- Filters are well-suited to be re-used in different parts of the system as long as the two filters being connected agree on the type of data that is being transmitted between them
- It leads to a system that is easy maintain and update
- Architecture is easy to analyze for deadlock and throughput
- Filters can be implemented and tested independently
- Filters can execute in parallel

3) Disadvantages

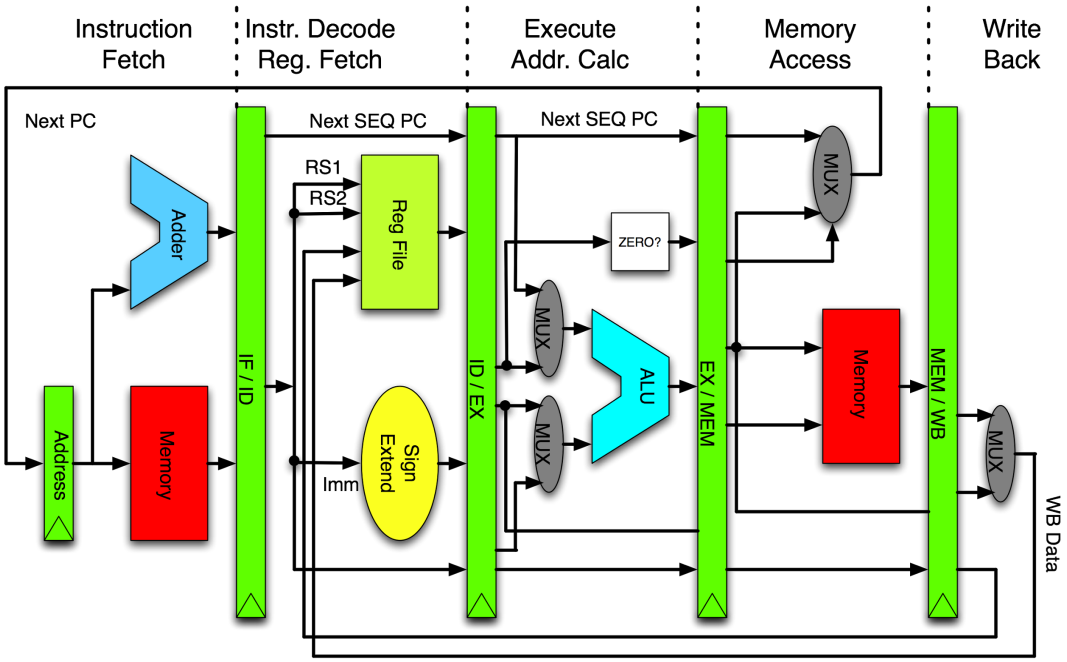
- Filters need to be planned with integration in mind
- Not suited for interactive systems
- To debug, many components must be considered together
- Efficiency is reduced because of the computation required to compose and parse data between different components

4) Usage

Compilers are implemented based on the style of the pipe and filter architecture, and viewed as pipeline systems as the data goes through stages of lexical analysis, parsing, semantic analysis, code generation **[8]**. The compiler splits source files and header files into components and the data and information is processed in parts to produce an executable program. Other examples of pipes and filters occur in signal processing domains, functional programming, and distributed systems **[8]**.

The MIPS pipeline **[14]**, portrayed in figure 5., is an example of the architecture, where the FETCH, DECODE, EXECUTE, MEMORY, and WRITE-BACK registers are the components and the data moves through the pipes to each step of the instruction processing. Using this architecture allows developers to break down the implementation of a system into parts. The developers can deal with smaller components

Figure 5. MIPS Pipeline Architecture **[14]**



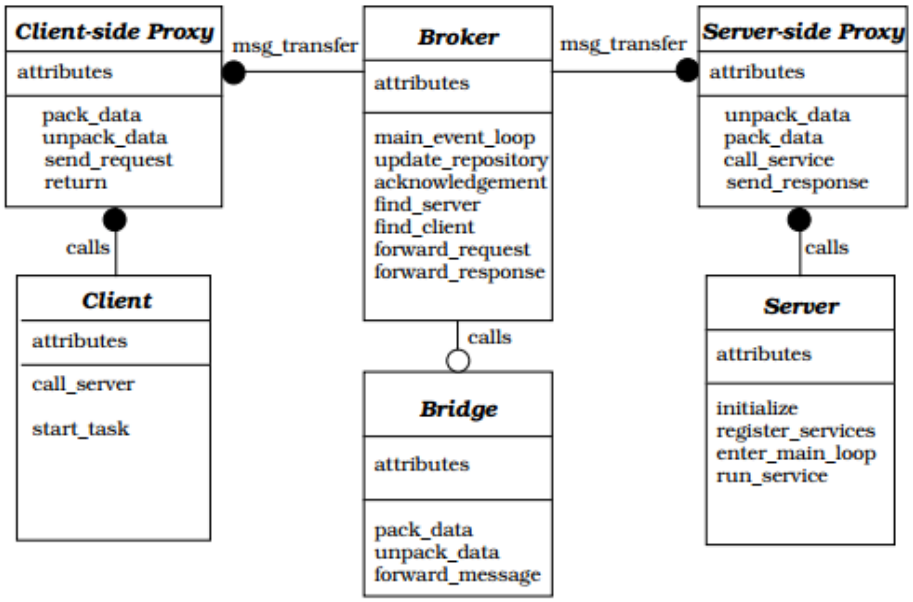
instead of one large system. Programs written in the Unix shell are some of the best-known examples of pipes & filters architecture **[8]**.

CMS Pipelines is an extension to the operating system of IBM mainframes to support Pipes and Filters architectures. The implementation of CMS pipelines follows the conventions of CMS, and defines a *record* as the basic data type that can be passed along pipes, instead of a byte or ASCII character **[13]**. LASSPTools is a toolset to support numerical analysis and graphics. The toolset consists mainly of filter programs that can be combined using UNIX pipes **[13]**.

C. Broker

The broker architectural pattern is similar to the pipe and filter pattern because of the information processing at certain components and the reusability of its components. The Common Object Request Broker Architecture (CORBA) **[15]**, standard defined by the Object Management Group is "designed to facilitate the communication of systems that are deployed on diverse platforms." **[15]** This type of pattern would be optimal in distributed systems **[16]**.

Figure 6. Basic Broker Architecture **[19]**



1) Introduction & Description

The broker component is the main medium for communication between the other components in the system. It is responsible for relaying requests from clients to servers, and then bring the response from the server and exceptions if any back to the client. There are unique keys and identifiers that allow for the broker to locate the origin of the requests and then determine the destination of the transmissions that it has been asked to handle. The Broker pattern is defined by six main components: clients, servers, brokers, bridges, client-side proxies, and server-side proxies.

The broker is the component that is responsible for organizing and coordinating the inter-system communication. Brokers have to register themselves with the server and then they will become available for use by the clients to service requests for the server. The broker can forward requests to the server and then transmit the results or exceptions back to the client on the user-side. The broker will be available through method interfaces and then clients can use the interface to obtain server functionality by sending requests through the broker. The broker takes care of locating the server for the client. The broker component is local and keeps track of when a request arrives at the server, the broker responsible for maintaining the server requests then directly passes transmissions to the server. If the requested server is currently asleep or doing nothing, the local broker activates the server before proceeding with forwarding requests. Depending on the required functionality of the system, other simple functions can be integrated into the broker component.

The Broker Architecture style makes it less complicated for the developer to create a distributed system, because the process and distribution is transparent. It can be used to develop a dynamic system and integration will be easy because it is a very flexible pattern that allows dynamic change, addition, deletion, and relocation of objects.

2) Advantages

- Location transparency as the location of data and information does not matter to the server
- Easy testing and debugging of the system as well as adding robustness, until extra component are added to the system
- Reusable components
- Portability as the implementation details are hidden in this pattern
- Quick development of client applications as their functionality can be modeled to mimic existing services developed previously

3) Disadvantages

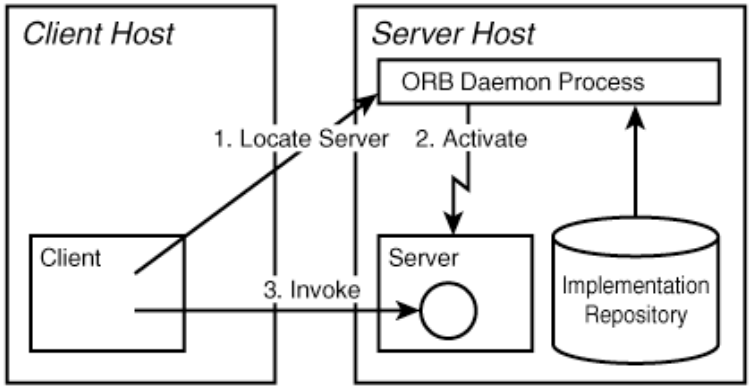
- Hard to predict behavior of components as the pattern in used in dynamic systems
- Hard to judge the outputs and performance of the system because of the its flexibility and dynamic behavior
- Slower performance compared to system where the major statistics are concrete and well-known by the developer
- Less tolerant to faults compared to non-distributed software systems (when server or broker fails, all applications depending on either of them also fail)[19]

4) Usage

This architectural framework would work well when applied to a structure distributed software system. Furthermore, if the system happens to contain decoupled components that communicate with each other through remote service invocations, the pattern would do well to organize the system. The broker architecture pattern is often used for management softwares that require interaction between independent components. It allows for the simplification of transmitting messages and data between its participating components. The diagram below shows the six main components of the broker architecture, characterized by their basic duties and relationships with other components in the system.

The World Wide Web is a relevant example with which people are familiar with and follows the Broker architecture pattern. Hypertext browsers, such as Hotjava and Netscape, are brokers and the WWW servers are the service providers that are serviced by the brokers. The CORBA technology has been used to develop many systems and applications that implement the broker architecture as a basic communication structure. The diagram below illustrates a simple CORBA client-server invocation handled by the broker (depicted by the client driver that lies within the client-side host.

Figure 7. CORBA Server Invocation on a Dormant Server **[18]**



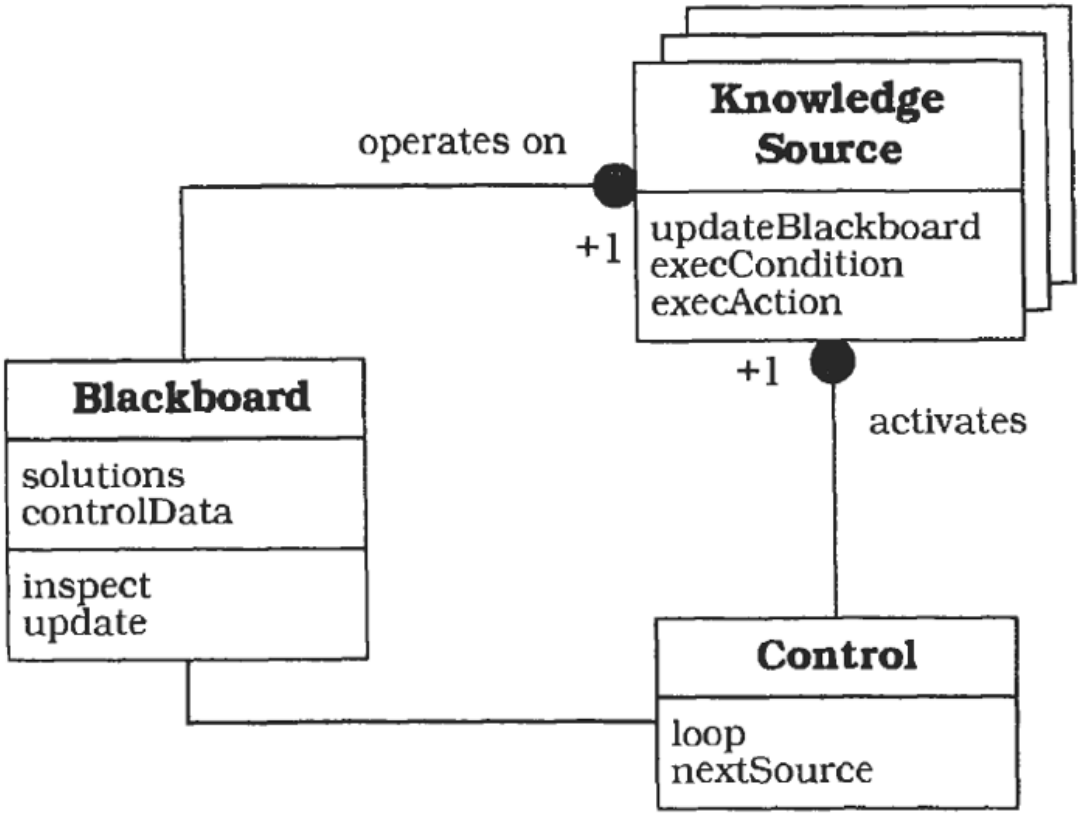
D. Blackboard

The blackboard architecture allows an interpretative use of knowledge. "It evaluates alternative actions, chooses the best for the current situation, and then applies the most promising knowledge source" **[16]**. Because of this the blackboard architecture excels in handling immature domains where experimentation is helpful.

1) Introduction & Description

This pattern provides a computational framework for design and implementation of systems that need to integrate large and diverse specialized modules by implementing non-

Figure 8. Blackboard General Representation [16]



deterministic control strategies. There is no predetermined strategy for how the 'partial problem solvers' should merge their knowledge. Moreover, a variety of alternative solutions can be generated. "In such cases it is often enough to find an optimal solution for most cases, and a suboptimal solution, or no solution for the rest" **[16]**.

There are three main components of the blackboard pattern: the blackboard, the knowledge sources and the control shell/component.

The blackboard is a structured global memory containing objects from the solution space. It serves as a dynamic library where problems, partial solutions and other contributed information are shared. The knowledge sources are highly specialized modules, with their own representation. The knowledge sources retrieve data from the blackboard. In a way similar to human experts on an actual blackboard, the knowledge sources provide their specific expertise to the application and contribute to the solving process. The third main component of the blackboard model is the control shell. This component selects, configures, and executes the knowledge source modules previously mentioned. The control shell's main purpose is to control the flow of the problem solving activity in the system.

There are several key features in blackboard architecture. Blackboard architectures have multiple independent cooperating knowledge sources. No one knowledge source can solve the task of a system alone; a solution can only be built by assimilating the results of several knowledge sources. There are multiple competing hypotheses or blackboard entries. Hypothesis rejected from the problem solving process, are removed from the blackboard.

Implementing the blackboard architectural pattern requires several steps. The first step is to define the various intermediate solutions and their representation. This is called the solution space. From this, the blackboard structure should be defined, and the corresponding knowledge sources that may provide solutions to the problem are identified. The knowledge sources may link with existing modules. When this happens, the modules have to be put in the form of knowledge sources. This means that triggering conditions must be added; input variables have to be linked to the blackboard data so they can be bound at run time, and results have to be put in the blackboard. The final component is to specify the control shell. "It often takes the form of a complex scheduler that makes use of a set of domain-specific heuristics to rate the relevance of executable knowledge sources"**[12]**.

2) Advantages

- A system well-suited for solving complex problems with no accepted approach to solving them
    - Especially when several modules are required to be dynamically combined to solve the problem
- Leads to high code reusability and modularity
- Effectively separates concerns for each module
- Modularity and encapsulation leads to high maintainability.

- As there are multiple modules working together to solve the problem, the system is robust and offers high fault-tolerance while solving problems.

3) Disadvantages

- Designing the blackboard is challenging, specifically for defining a good control strategy
    - Requires tuning, only possible through experiments
- High data coupling, especially when multiple actions depend on the same structure
    - The data structure is the fragile part of the system
- Lowered efficiency - requires extra computational overhead to reject wrong hypothesis (potential solutions)

4) Usage

In software development, the Blackboard pattern is often used for software where there are problems and no deterministic strategies are known.

The first system that was based on the blackboard architectural pattern was the HEARSAY-II speech recognition system. The task of this particular application was to answer queries about documents and to retrieve documents from a collection of abstracts of Artificial Intelligence publications. In HEARSAY-II, the control component is consisted of the following:

Focus-control-database: contains a table of primitive change types of blackboard changes that can be executed for each change type. An example of a primitive change types are 'new syllable' or 'new word created bottom-up'. This means that a new word appeared on the blackboard and it was inferred using hypotheses on lower levels.

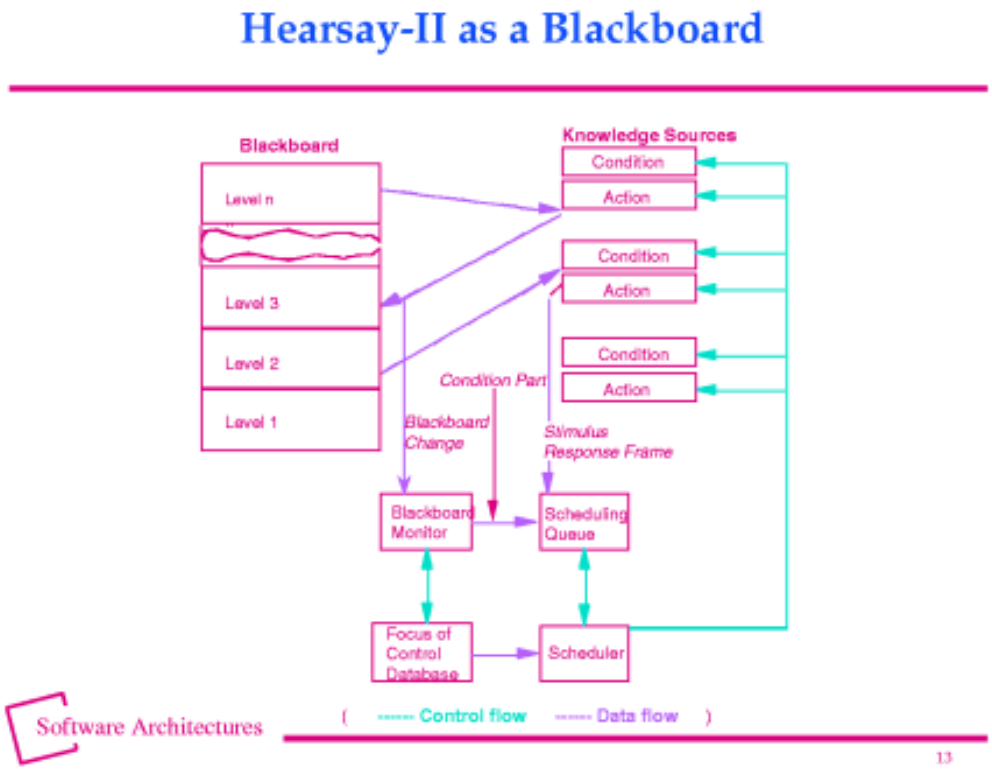
Scheduling queue: contains pointers to, and performs based on conditions or action parts of the knowledge sources. The scheduling queue is merely a container that determines which task should have highest priority from the knowledge source and passes it to the scheduler.

Monitor: keeps track of each change made to the blackboard. The monitor inserts pointers to applicable condition parts into the scheduling queue based on the corresponding primitive change types. If a condition part is ran and the calculated response frame is not empty, a pointer to the matching action part is placed in the scheduling queue.

Scheduler: uses experimentally-derived trial and error procedures to calculate priorities for the condition and action-parts waiting in the scheduling queue. In addition the scheduler takes into account overall blackboard state information such as which competing hypotheses have the highest support from hypotheses on lower levels. The scheduler selects the condition or action part with the highest priority for execution.

The scheduler communicates and activates the knowledge

Figure 9. Hearsey-II Represented as a Blackboard Architecture**[29]**



sources, and information such as whether to read the data, perform some computation or write data is passed between the control to the blackboard to solve the problem **[16].**

Some other examples of where the blackboard architectural pattern is used are: vehicle identification and tracking, identification of the structure of protein molecules, and solar signals interpretation (RADARSAT-1) **[11]**.

E. Interceptor

The Interceptor architectural pattern allows services to be added transparently to a framework and triggered automatically when certain events occur i.e enables functionality to be easily added to the system in order to dynamically change its behavior.

1) Introduction & Description

It is made up of three components:

Interceptor interface: Interceptor interface can define arbitrary number of operations that are executed by Dispatcher in specific cases at the beginning of the request processing or at the end of the request processing.

Dispatcher: Dispatcher class typically implements Interceptor interface.

Context: Context is typically simple data object used to carry data for interceptors or to keep results of intercepting.

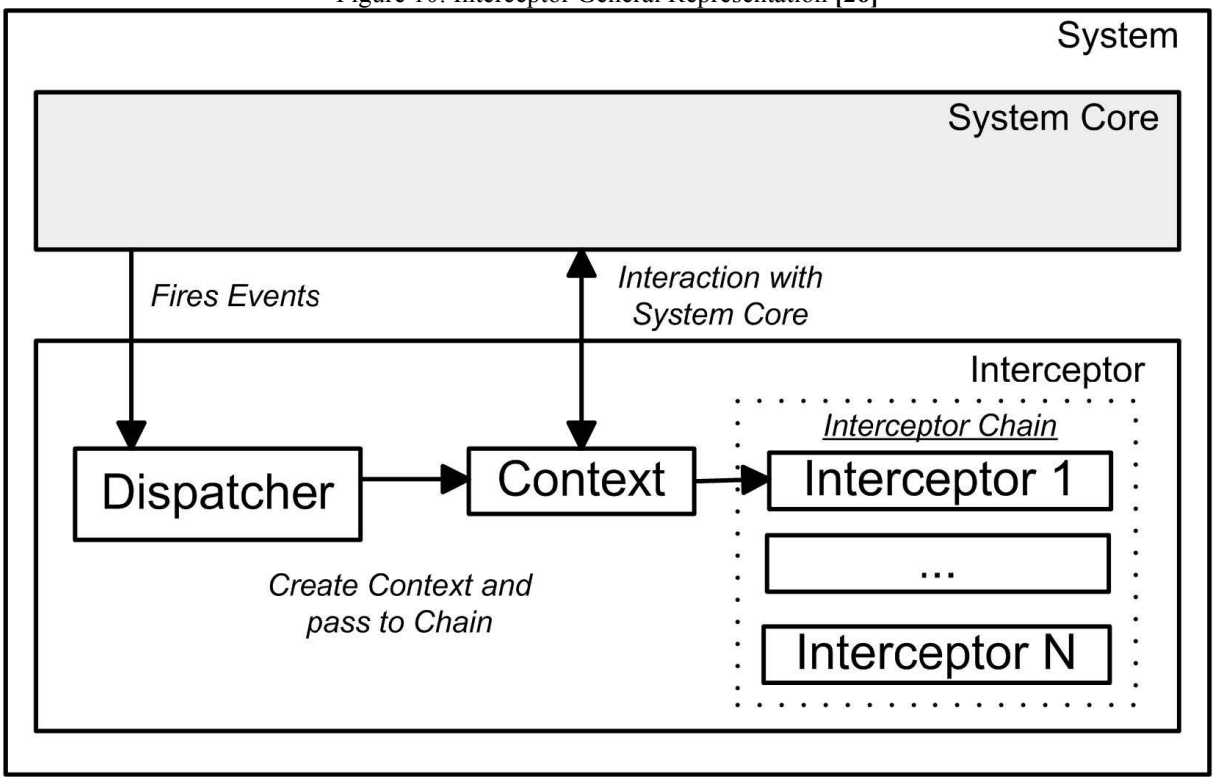
2) Advantages

- It increases reusability
- It increases flexibility
- It increases extensibility
- Easy monitoring of the applications behavior
- The rest of the system does not have to know if something has been added or changed and can keep working as before
- Decoupling of communications between a sender and receiver of an interceptor request (this permits any interceptor to fulfill the request and allows interceptors to change system functionality, even at run-time)

3) Disadvantages

- Inherent openness of pattern introduces potential vulnerability to the system
- Complex design issues, anticipating the requirements of applications that use a specific concrete framework is non-trivial, which makes it hard to decide which interceptor dispatchers to provide
- Malicious or erroneous interceptors, if a concrete framework invokes an interceptor that fails to return, the entire application may block

Figure 10: Interceptor General Representation **[26]**



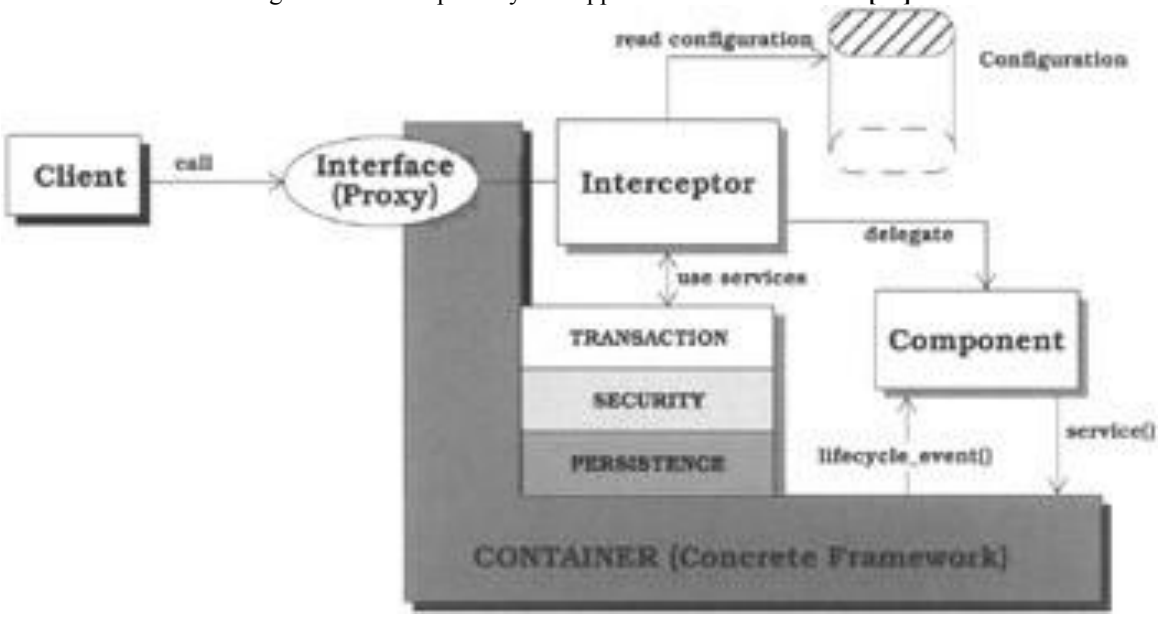
4) Usage

Variants of the Interceptor pattern have been implemented by various component-based application servers for server-side components. Examples include Enterprise JavaBeans (EJB), Common Object Request Broker Architecture (CORBA) Components, and Microsoft Component Object Model (COM+). Frameworks are introduced to shield components from the system-specific run-time environment. This helps developers focus on the application-specific business logic by taking advantage of the infrastructural services such as transactions, security, or persistence that are provided by the framework.

Below is a diagram of this application:

Interceptors are used in a broad range of domains to increase flexibility and extensibility. Web browsers, for example, implement the Interceptor pattern to help third-party vendors and users to integrate their own tools and plugins **[28]**. In another domain, web servers (e.g. Apache 2.0) implement interceptor to allow modules to tie into the core server **[26]**. In Common Object Request Broker Architecture **(**CORBA) implementations such as TAO and Orbix, the Interceptor pattern is applied so that application developers can integrate additional services to handle specific types of events. Interceptors enhance Object Request Broker (ORB) flexibility by separating request processing from the traditional ORB communication mechanisms required to send and receive requests and replies **[24]**.

Figure 11. Interceptor Style's Application in Frameworks **[24]**



F. Microkernel

Microkernel is one of the Adaptable System architectural patterns. Microkernel is designed for systems that must respond to rapid changes to requirements and environment, and that must be highly scalable **[16]**.

1) Introduction & Description

There are five components in Microkernel. They are Microkernel, Internal Server, External Server, Adapter and Client. Microkernel is the core of the system and client is the newly added applications. Since one of the characteristics of this system is adaptability, it must be easy to add more components.

In this design, the system is separated in five components. The first component is called Microkernel, and it is the main component of the design, it maintains system resources and other components will need to interactive with this component. Next is the Internal Server, it is the internal system extended functionality provider to the Microkernel component. It includes important software and hardware dependencies in order to allow the system to run. External Server provides interfaces to clients. It uses interface provided Microkernel to execute client application. It will execute the application in a separated process for the client. In order to make external application more portable, we have an Adapter component that communication with External Server to separate direct dependencies. Different systems have different External Server interfaces. Without the central adapter, it would be necessary to have tight coupling between the Client and External Server. Lastly we have a component called client, which is the external application.

Microkernel can be run with only Microkernel and Internal Server components. External Server, Adapter and Client components are only required only if you would like to add new software or hardware extensions.

Figure 12. Microkernel Components

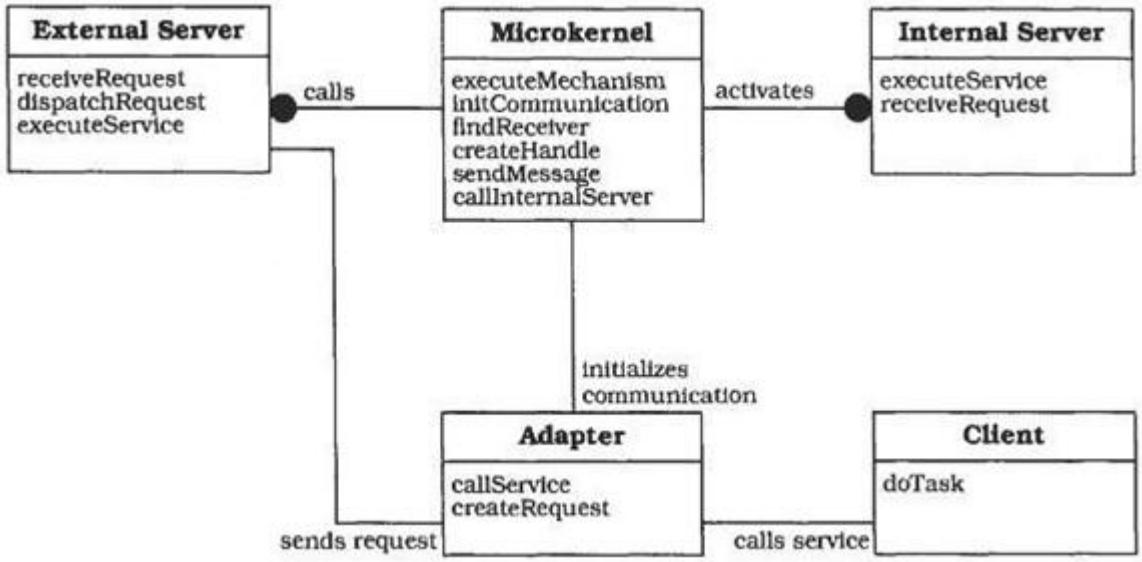
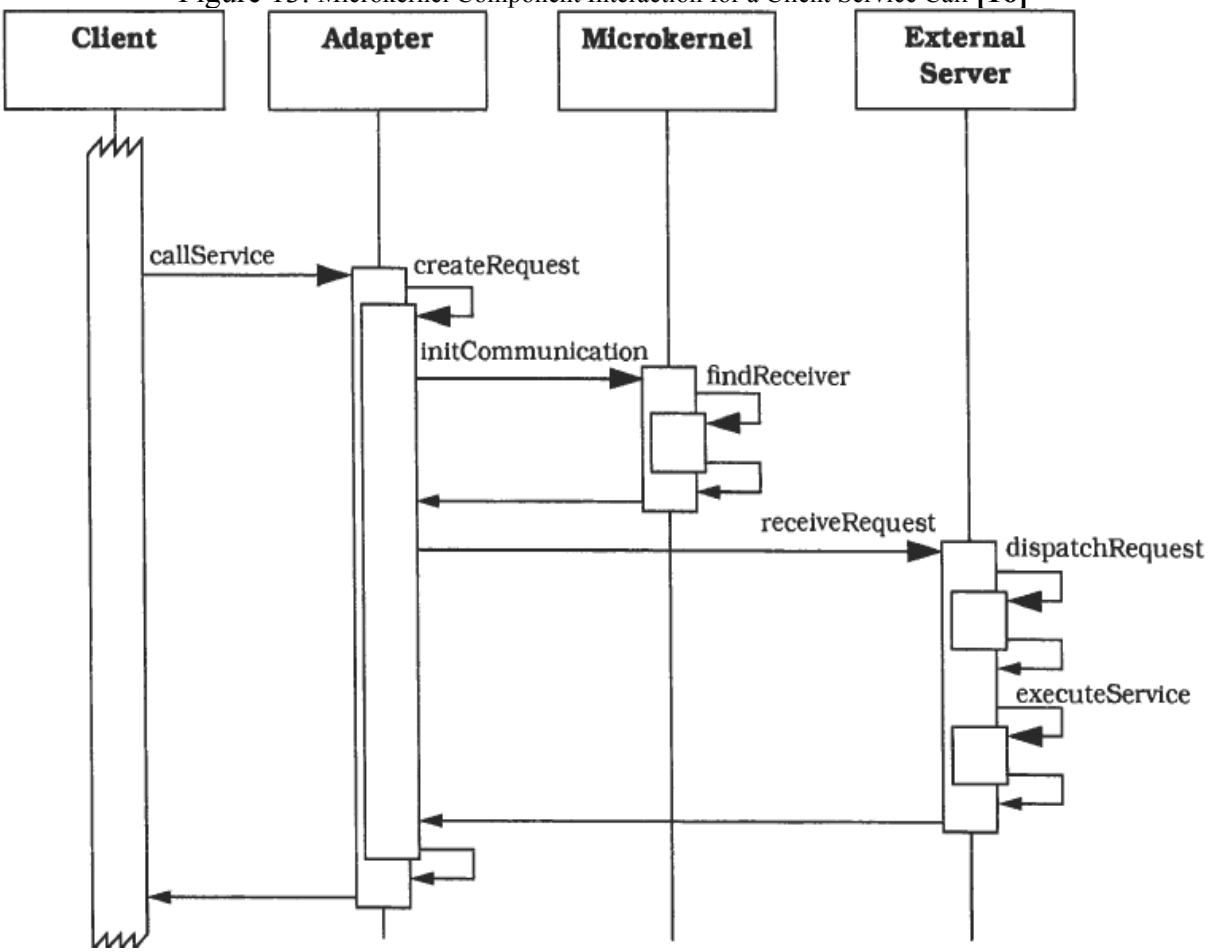


Figure 13. Microkernel Component Interaction for a Client Service Call **[16]**



2) Advantages

- Portable, simple to port to different software and hardware environments
- Adaptable, easy to add additional server components to add more functionality
- Extensible, to extend this design you can just add another external server and client
- Scalable, Microkernel can be design with the support of distributed, this will gain other advantages such as reliability and transparency [16][17]

3) Disadvantages

- Performance, making a new service call requires the client to communicate with adapter, then microkernel and finally the external server. This process is much slower than just a direct service call [16]

4) Usage

Microkernel architecture has been use in Mach, Amoeba, Chorus, Windows NT and Microkernel Datenbank Engine **[16]**.

5) Possible Modifications

Microkernel can also be distributed. A Distributed Microkernel can be built by running the same system over more than one machine. One of the reasons we need Distributed Microkernel is reliability, by running same system on different machines we will have many backup of the same application. If one of the machines fails, there are other machines. Another reason for Distributed Microkernel is to increase scalability, when one machine is not enough for demands, you can simple add another machine running the same system to separate the server load. It is a good design to prevent system failure and provide maximum up time for users **[16][17]**.

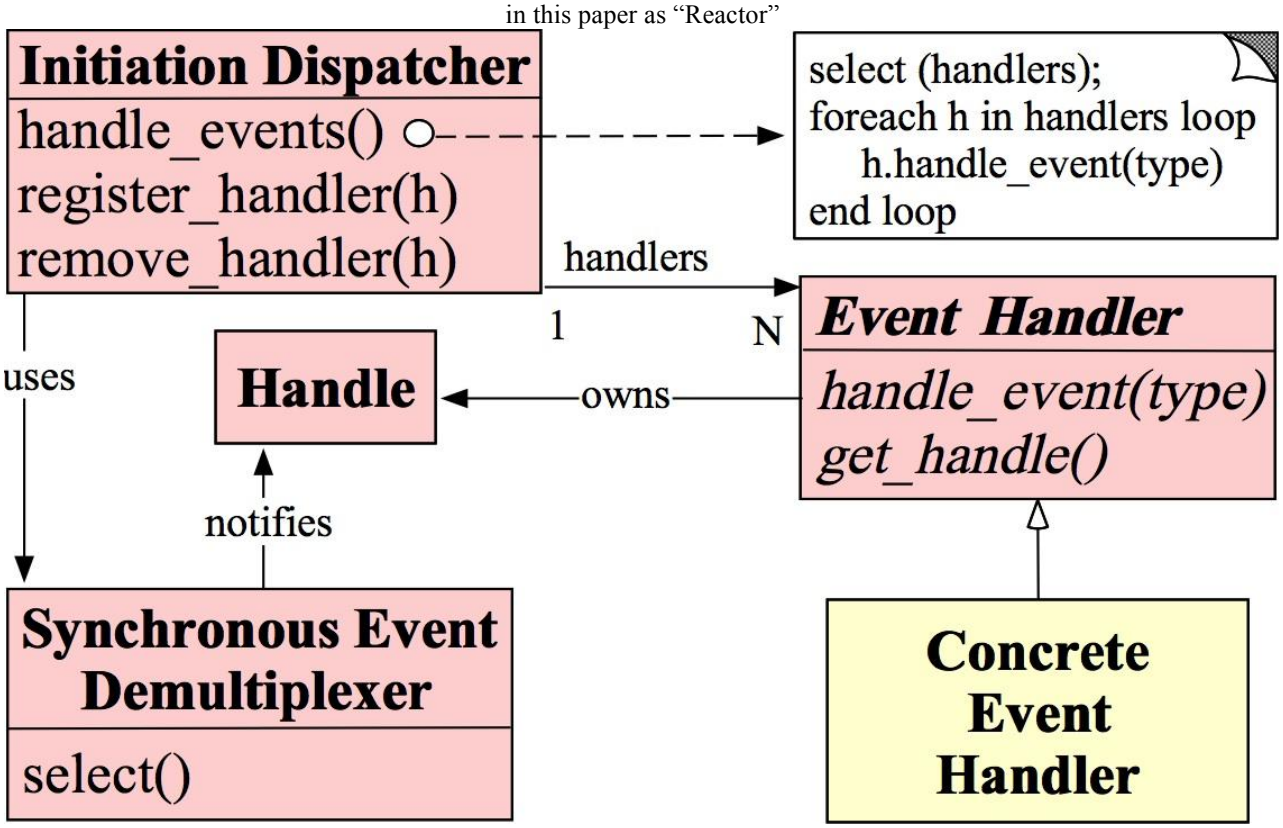
G. Reactor

The Reactor architectural pattern allows event-driven applications to demultiplex(opposite of multiplexing) and dispatch service requests that are delivered to an application from one or more clients. This type of architectural pattern is usually used in networked software applications, applications that receives multiple service requests simultaneously, but processes them synchronously and serially.

1) Introduction & Description

It explains how to register handlers for particular event types, and how to activate handlers when events occur, even when events come from multiple sources in a single-threaded environment.

Figure 14. Reactor Architecture's General Representation **[25].** Note that "Initiative Dispatcher" is the object referred to in this paper as "Reactor"



There are five parts of the reactor architecture:

Reactor: A.k.a. Initiation Dispatcher. A reactor has two main purposes: the first is to allow applications to register or de-register event handlers, and the second is to wait for and respond to the events that would trigger the registered event handlers. It stores the event handlers as a set of *Handle* objects, one for each event handle.

Handles: Provided by the operating system, the handle identifies the events that trigger the event handler. The events that trigger the event handlers are called 'indication events'. Events can originate from external and internal sources (relative to the system hardware). Examples of external indication events are connections being made, or data reads completing. An example of an internal indication event is a time-out.

Event Handler: Provides one (or more) methods for handling events. These methods are executed when indication events occur on associated Handles in order to process the event.

Concrete Event Handler: One implementation of the event handler interface. Each event handler is associated with one handle to specify which indication events it should respond to.

Synchronous Event Demultiplexer: Waits for indication events to occur on the handle set. Used by the Reactor to determine when events must be responded to. It indicates to the Reactor when an event handler may be executed without blocking. If an event handler was executed before the indication event occurred, it is likely to block. This is the essential reason for the Synchronous Event Demultiplexer.

The reactor pattern operates as follows: First, Event Handlers are registered with the Reactor. Then, the reactor uses the Synchronous Event Demultiplexer to wait for indication events for any of the registered Event Handlers. When an indication event occurs, the Reactor is informed through the Synchronous Event Demultiplexer and executes the Event Handler associated with the Handle from which the initiation event occurred. The type of indication event that occurred is passed as a parameter to the event handler method. The event handler processes the request, and returns information to the client that requested the service (through the indication event) by means of the event handle. The Reactor then continues waiting for indication events and repeats the above when one occurs.

2) Advantages

- Modularity, reusability and configurability, The pattern decouples event-driven application functionality into several components
- Handlers can be created easily
- Handler objects do not need to be aware of how events are dispatched
- Separation of concerns, this pattern decouples application-independent demultiplexing and dispatching mechanisms from application-specific hook method functionality

3) Disadvantages

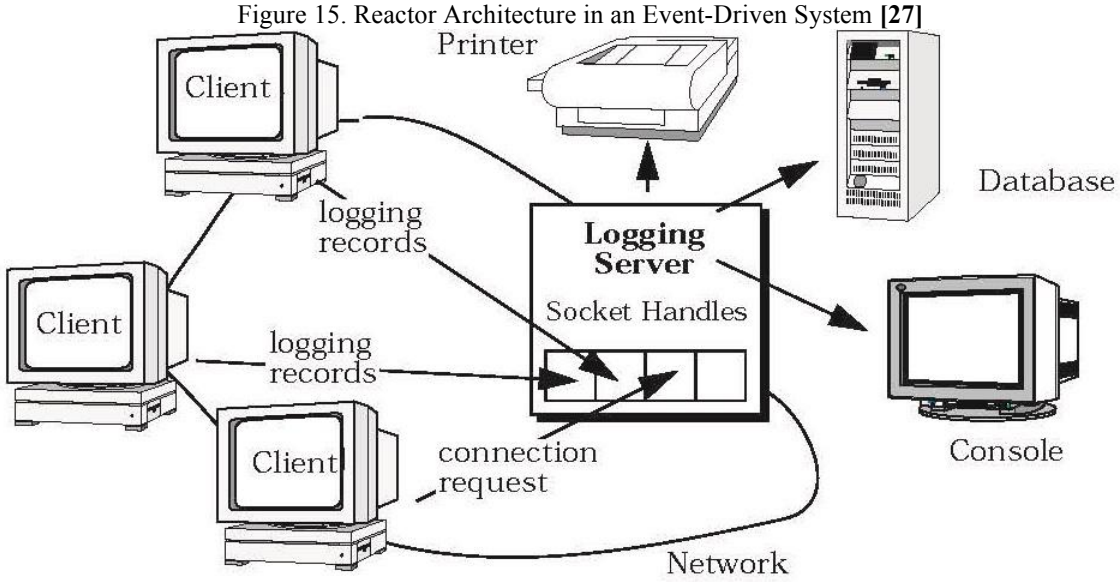
- Restricted applicability, This pattern can be applied most efficiently if the operating system supports synchronous event demultiplexing on ```handle sets
- Difficult to learn initially
- Difficult to debug, it can be hard to debug applications in this pattern due to the inverted flow of control
- It does not scale to support a large number of simultaneous clients and/or long-duration client requests well
- This architecture is not suitable for handling operations that take a long duration to complete

4) Usage

This architectural pattern is used in event driven applications in a distributed system, particularly servers, that must be prepared to handle multiple service requests simultaneously, even if those requests are processed serially within the application.

Consider the event-driven server for a logging service shown above. Client applications use the logging service to record information about their status in a distributed

Figure 15. Reactor Architecture in an Event-Driven System **[27]**



environment. This status information commonly includes error notifications, debugging traces, and performance reports. Logging records are sent to a central logging server, which can write the records to various output devices, such as a console, a printer, a file, or a network management database.

This architectural pattern is used in the InterViews windowing system where it is known as the dispatcher.

H. Model-View-Controller

Model View Controller is the most well known architectural style for interactive systems. MVC fits under the interactive systems category since the focus is on keeping the user interface and the functionality of the system separated. MVC is simple, easy to learn, and effective which is why it is a popular architectural style.

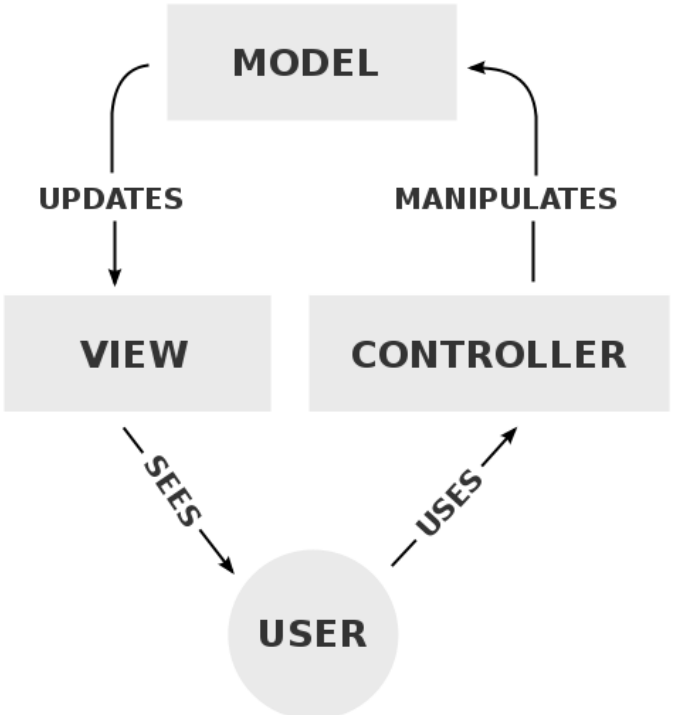
1) Introduction & Description

MVC, as the name goes, has three main components that are *Model*, *View* and *Controller*. An important characteristic MVC has is a change propagation mechanism, which we will discuss later. The purpose of this pattern is to prevent altering the functionality of the system when changes are made in the interface design. Another purpose is that it allows the system to have many different user interfaces which all use just one underlying data and logic model.

The Model component contains the data and logic. All the processing and core functionalities are in this component. The most important responsibility of the Model component is to synchronize all the components together **[4]**. It contains a change propagation mechanism and registers(detects and acknowledges) both the View component and the Controller component.

The View component is the part that is visible to the user. The main responsibility of this component is to extract data from the Model component and display this data to the user. There can be more than one view in a system since information from the Model can be displayed in many different ways **[4]**. In fact, any system may have one or more views. The number of views is only limited by the different useful ways the information in the model can be displayed as well as human inventiveness. A second responsibility of this component is to implement an update method for the change-propagation mechanism.

Figure 16. MVC General Representation **[21]**



The Controller component is the part that handles all user input and comes in between the *Model* and *View* components. The main responsibility of this component is to handle events and translate these events into requests to the Model component or the View component **[4]**. Similar to the View component, the Controller component also implements an update method for the change-propagation mechanism. Typically, the combination of *View* and *Controller* form the user interface **[4]**.

*2) Advantages **[4]***

- Reduces complexity of the system
- Improves maintainability
- Separation of concerns minimizes impacts of changes and so increases maintainability

*3) Disadvantages **[4]***

- The change propagation mechanism limits performance
- Lack of structure within each individual component leads to a risk of high coupling
- The entire system relies on the Model component, which means changes to it can break the entire system

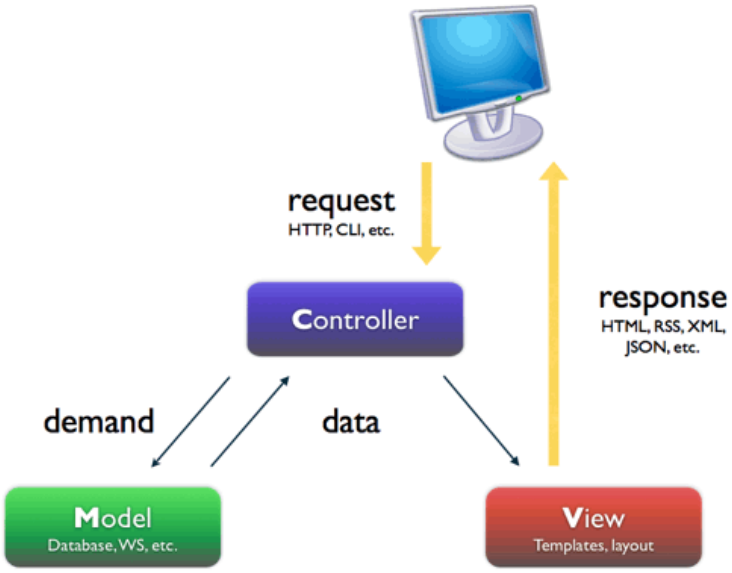
4) Usage

In general, the MVC pattern can be applied to any system. This pattern is used in systems that consist of a front end that involves user interaction through an interface and a back end that involves data processing. MVC is most ideal in systems that require multiple interfaces for different users or adaptable interfaces that change constantly.

This pattern works very well and is commonly found in web applications. *PHP Framework for Database Management* is such an example **[7].** PHP is a language used for developing web-base applications and has many benefits. This PHP framework simplifies the web development process by offering an MVC implementation including services to update and manipulate the database **[7].** Figure 17 below illustrates a web application implemented in *PHP Framework for Database Management*.

In this example, the Model contains the business logic of the web application and connects to the database. The business logic consists of functions for manipulating the database such as connecting to the database, inserting, updating, deleting, and selecting. The View is the user interface and where there are endless design possibilities since it can be designed with HTML, CSS, JavaScript, and many more. In the example, the user interacts with the View to trigger events that the Controller handles. The Controller contains all code for handling the control flow and the user's interactions with the interface. In the example below, the Controller extracts data from the database in the Model and then updates the View.

Figure 17. MVC Implementation of a Web Application **[22]**



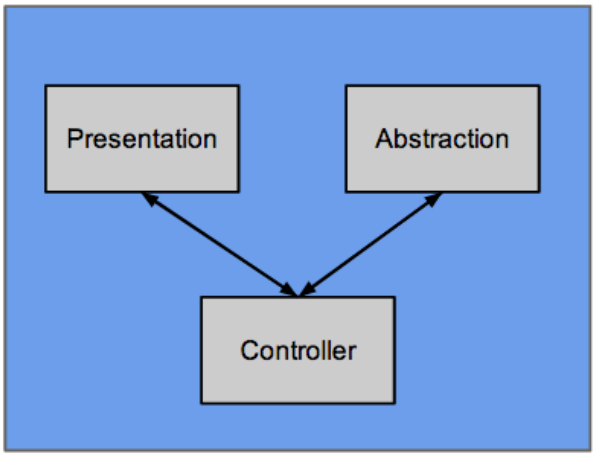
I. *Presentation-Abstraction-Control*

The Presentation-Abstraction-Control pattern is similar to the MVC pattern, the other interactive system architecture presented in this paper. It is, however, much more complex. The main difference between PAC and MVC is that functionality is subdivided into tasks. PAC consists of agents that collectively form the system's functional core whereas the functionality core in MVC is in one component.

1) Introduction & Description

The PAC pattern structures the system into a hierarchy of agents. This hierarchy consists of the following: one top-level PAC agent, several intermediate-level PAC agents, and several more bottom-level PAC agents (refer to Figure 19. below). Each of these PAC agents are then subdivided into the following three components: Presentation, Abstraction, and Control (refer to Figure 18. below).

Figure 18: A PAC agent (Contributors' Diagram)



*2) Advantages **[6]***

- Division of functionality into tasks results in very loose coupling
- Both the data model and the interface in an agent can be easily modified without breaking code of the loose coupling
- It is well suited to parallel development since functionality is subdivided into smaller tasks
- Only the Control components need to be updated and synchronized

*3) Disadvantages **[6]**:*

- The coordination of a network of all the PAC components can be very complex
- It is difficult to ensure consistency between all the Control components
- Data exchange between top and bottom levels is inefficient because of many intermediate steps along the way
- It is likely to result in duplicate agents
    - If there are two or more bottom-level agents that stem off from different intermediate-level agents and these bottom-level agents have the same Presentation component or Abstraction component, there will be duplicate code.

4) Usage

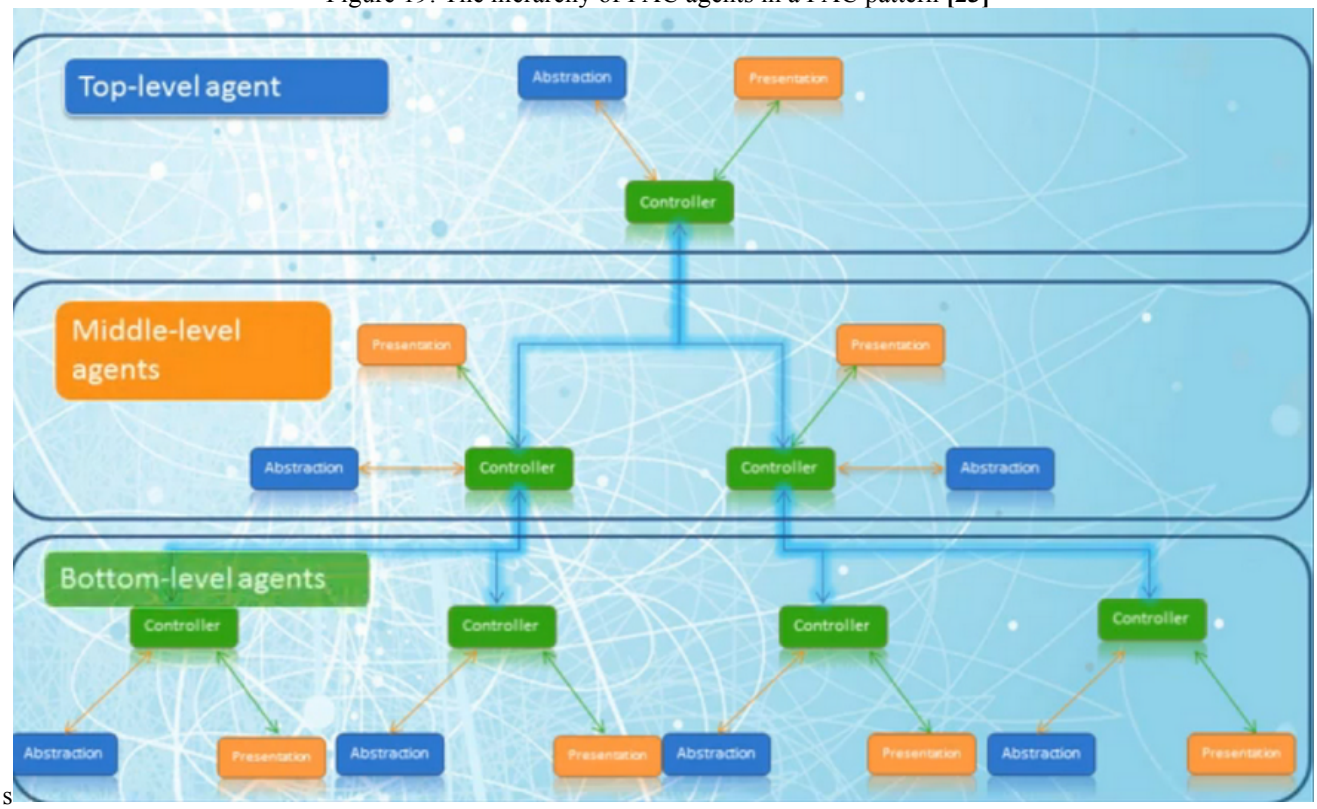
Since the PAC pattern fits under the interactive systems category and focuses mainly on separating functionality and user interaction, the PAC pattern can be applied to almost any MVC pattern. Ideally, the PAC pattern is applied in systems that require further organization and some layering within the data, user interface, and control components.

An example of the usage of the PAC pattern is an information system that consists of a graphical user interface that has a spreadsheet for data entry and displays the information in different types of charts and tables **[5]**. In this example, the user interacts with the system to view a bar chart after putting data in the spreadsheet. Figure 20. below illustrates how this information system might look when implemented in PAC pattern and Figure 21. shows a sequence diagram for the case of a user accessing a chart containing some data.

The following gives a detailed description of the process involved in this example of applying the PAC pattern (refer to figures 20. and 21. below):

Top-Level: The user interacts with the Presentation component in the top-level PAC agent that communicates to the Control component in the top-level agent to talk to the "view coordinator" agent in the intermediate-level.

Figure 19: The hierarchy of PAC agents in a PAC pattern **[23]**



s

Intermediate-Level ("view coordinator"): The user then interacts with the Presentation component of the "view coordinator" agent which communicates to the Control component in the "view coordinator" agent to talk to the "bar chart" agent in the bottom-level.

Bottom-Level ("bar chart"): The Control component of the "bar chart" agent goes back up the hierarchy to retrieve data from the top-level agent. The data is then saved in the Abstraction component of the "bar chart". The Control component of "bar chart" then communicates to the Presentation component of "bar chart" to display the bar chart.

Figure 20. Information System Implemented in PAC Architecture **[5]**

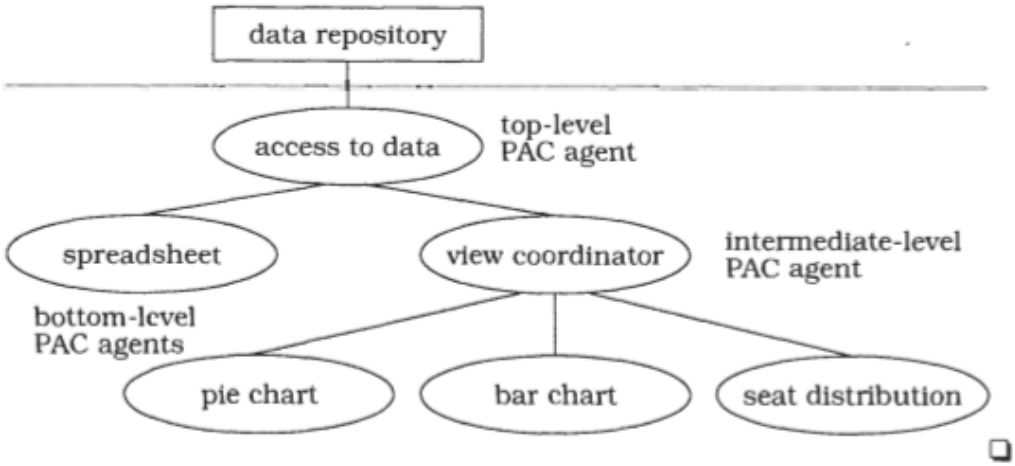
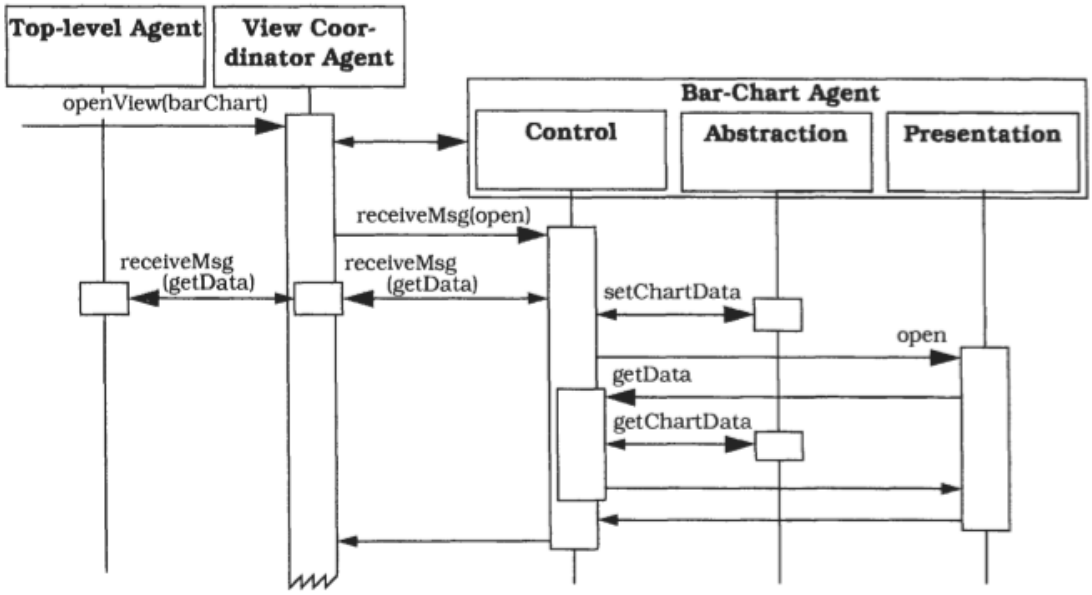


Figure 21. Sequence Diagram of an Information System Implemented in PAC **[5]**



III. ARCHITECTURAL STYLES EVALUATION

Considering the above discussions, this section compares the different architectural styles against one another. The selection of the right architectural style for a system is a multicriteria problem for any organization or team that dives into this domain. Each criteria maps to a particular concern with a set of alternatives that have a range of impacts on quality attributes. Therefore, it is important to have some guidelines on what factors should be considered, which styles are more compatible with each other and which have the negative effect on the other.

Figure 22. (below) provides a summary of each of the advantage/disadvantage categories presented above. They have been grouped into four category groups: design (which refers to the design process), system, structure (code structure) and other benefits. Each style is given a rating of either plus (+), minus(-), or no rating ( ) for every category. The plus indicates that for the style in the given row, the category in the given column is an advantage. The minus indicates that the category is not met and so the opposite of it is a disadvantage. By way of example, the minus on the Pipe and Filter row in the Low Coupling column indicates that the Pipe and Filter style has the disadvantage of high coupling. No rating should be read as "no comment" and indicates that no information is provided one way or another. The styles have been ordered in the same order as they have been presented above, for the sake of consistency.

To use the following figure to identify a suitable architectural style for a given software project, the key concerns out of the list of categories should be selected. Refer then to the columns for those categories, and select styles that meet those requirements (+). If no rating has been provided, it will be necessary to perform further investigation to determine whether or not the style in question will meet the project requirements in that category.

Figure 22. Architectural Styles Evaluation Chart

Architectural Style	design				system				structure			other benefits				
	Simplicity	Ease of Design	Incremental development	Maintainability	Good for interactive systems	Good for computationally complex systems	Good for distributed systems		Inherently secure	Separation of concerns	Modularity	Low Coupling	Efficiency	Scalability	Extendibility	Portability	Fault-tolerance
Layers	+	-	+	+					+	+		-					
Pipe and Filter	+	+	+		-				+	+	-	-					
Broker							+				+	-		+	+	-	
Blackboard		-				+				+	-	-				+	
Interceptor	-	-		+				-	+	+	+			+			
Microkernel	-			+			+		+			-	+	+	+		
Reactor	-	+		-		-			+	+			-				
MVC	+	+		-	+	-			+	+	-	-					
PAC	-		+	+					-		+						

IV. Conclusion

Throughout the previous sections, various architectural styles (patterns) were discussed with enough detail to give an understanding of the applicability of an individual style and to compare between multiple styles. For a software designer, there are a few key areas to address when selecting a style. These follow from the advantages and disadvantages of the styles presented above: What is the domain of the system at hand? How complex is the problem that it must solve? How will it change with time? What other goals do the stakeholders have? This being said, the conclusion of the matter is simpler than this. Will you choose to apply an appropriate architectural style to the system being designed? The tools in this paper provide a means of selecting a style based on a variety of criteria, and references for further evaluating one or more styles that are suspected to be applicable. What remains to be done is to ensure that one's software designing process incorporates this key step: select an appropriate architectural style.

REFERENCES

[1] T. Bart, "A practical introduction to layered architecture — Part One", blog, 12 Apr. 2009; http://fewagainstmany.com/blog/

[2] R.Taylor, et al., in Software Architecture: Foundations, Theory, and Practice, John Wiley & Sons., 2006

[3] F. Buschmann, et al., "Layers," in *Pattern-Oriented Software Architecture: A System of Patterns*, Vol. 1, England: John Wiley & Sons Ltd., 1996, pp. 31,32, 46-48, 50, 72, 79, 88 ,94

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Model-View-Controller," in *Pattern-Oriented Software Architecture: A System of Patterns*, Vol. 1, England: John Wiley & Sons Ltd., 1996, pp. 125-129, 141-143.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Presentation-Abstraction-Control," in *Pattern-Oriented Software Architecture: A System of Patterns*, Vol. 1, England: John Wiley & Sons Ltd., 1996, pp. 145, 148, 154, 167.

[6] D. Plakalović, D. Simić, "Applying MVC and PAC patterns in mobile applications," *Journal of Computing*, Vol. 2, No. 1, pp. 69-70, Jan. 2010.

[7] C. Supaartagorn, "PHP Framework for Database Management Based on MVC Pattern," *International Journal of Computer Science & Information Technology*, Vol. 3, No. 2, pp. 252-253, Apr. 2011.

[8] David Garlan and Mary Shaw, *An Introduction to Software Architecture,* CMU-CS, Pittsburgh, PA, 1994, pp. 6-8.

[9] Swapna S. Gokhale and Sherif Yacoub, Reliability Analysis of Pipe and Filter Architecture Style, Storrs, CT, 2004.

[10] A.R. Hurson and K.M. Kavi, Dataflow Computers: Their History and Future, Denton, TX, 2008.

[11] D. Corkill, "Countdown to Success: Dynamic objects, GBB, and RADARSAT-1" in *Communications of the ACM, 1997* pp. 48–58.

[12] P. Lalanda, "Two complementary patterns to build multi-expert systems," in *Patterns*

[13] *Languages of Programs*, Monticello, Illinois, 1997.

[14] Frank Buschmann, et al. "Architectural Patterns", in *Pattern-Oriented Software Architecture*, vol. 1, *A System of Patterns*, West Sussex, England: Wiley, 1996.

[15] *MIPS Architecture*, Wikipedia, The Free Encyclopedia, 2009.

[16] Jon, Object Management Group, Inc., (2014, Feb. 27) *CORBA® BASICS* [Online]. Available: http://www.omg.org/gettingstarted/corbafaq.htm, Retrieved: April 3, 2014.

[17] Frank Buschmann, et al. "Architectural Patterns", in *Pattern-Oriented Software Architecture*, vol. 1, *A System of Patterns*, West Sussex, England: Wiley, 1996.

[18] *Microkernel*, Wikipedia, The Free Encyclopedia.

[19] Fintan Bolton, "CORBA Architecture" in *Pure CORBA*, USA: Sams, 2001.

[20] *The Broker Architectural Framework,* Siemens AG, Munich, Germany, pp 3-19.

[21] http://kitchaiyong.files.wordpress.com/2010/10/figure-1_6-common-layered-application-architecture1.png

[22] http://en.wikipedia.org/wiki/File:MVC-Process.svg

[23] http://bpesquet.developpez.com/tutoriels/php/evoluer-architecture-mvc/images/3.png

[24] http://mgalalm.files.wordpress.com/2011/08/screenshot1.png

[25] Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2 by Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann Page 120

[26] Schmidt, Douglas, An Object Behavioral Pattern for

[27] *Demultiplexing and Dispatching Handles for Synchronous Events,* Department of Computer Science, Washington University Available: http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

[28] Extending Message-Oriented Middleware using Interception (2004), Edward Curry, Desmond Chambers, and Gerard Lyons. Department of Information Technology,National University of Ireland, Galway, Ireland. Venue : In Proc. of the 3rd Int. Workshop on Distributed Event-Based Systems

[29] Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2 by Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. Page 154

[30] Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2 by Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann Page 121

[31] *http://www.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/1998/Lectures/20.Blackboard/base.012.gif*

[32] Microsoft Developer Network. 2014. *Chapter 3: Architectural Patterns and Styles.* Available: *http://msdn.microsoft.com/en-us/library/ee658117.aspx*