**Software Architect and Design**

Ashok Bezawada(L30063938)

Tharun Siddala(L30063834)

**TOPIC:- ARCHITECTURAL PATTERNS**

# ABSTRACT

Architectural patterns are mostly similar to software design pattern, but they provide the scope in broader way, these are mainly used for commonly occurring problems. These patterns are heart core in programming design because they offer the flexible and reliable solutions to construction-related problems, with the aim of preparing the compositional plan decisions which are useful for interaction with the understandable language for both the parties and provide the powerful quality product features . Still using and choosing acceptable compositional examples is still mostly an difficult and unplanned process in practice.

Architectural patterns are a fundamental idea in programming design because they provide trustworthy solutions to construction-related issues, assist in the documentation of compositional plan decisions, serve as a common language for partners to communicate in, and symbolize the capabilities of a product's quality features. Sadly, selecting and employing appropriate compositional examples is still mostly an impromptu and unplanned process in practice.

This is due to the lack of consensus among architects regarding the "philosophy" and level of detail of architectural patterns, as well as the absence of a standardized pattern language. To address these issues, we propose a pattern language that acts as a superset of existing collections and categorizations of architectural patterns. This terminology emphasizes the relationships between building patterns and how to categorize them using "architectural viewpoints."

**Introduction:**

Numerous talks on the topic of software architecture have focused on the ideal Languages, design and evaluation procedures, and points of view are described in architecture. Architectural patterns are regarded for their significance and play a crucial part in architectural descriptions, making them one of the few areas of software architecture where there was uniformity. Unfortunately, creating, detecting, and applying architectural patterns are still mostly arbitrary and unstructured in reality.

A multitude of unresolved issues are the cause of this: There is a semantic gap between what architectural patterns actually represent and the philosophy behind them, there is substantial disagreement regarding the level of detail in architectural patterns, and there is no widely recognized classification scheme or library for patterns used by architects.

# PATTERN LANGUAGE:

In order to find areas of agreement among those who study architectural patterns, we suggest a pattern language in this research article that serves as a superset of the existing collections and classifications of architectural patterns. This language stresses connecting the patterns to represent the "full picture," making it more than the sum of its individual parts. For the aforementioned issues, this pattern language offers the following solutions.

We refer to a pattern as architectural when it solves a problem at the architectural level of abstraction, or when it tackles the complete system architecture as opposed to simply a few specific discrete subsystems. With the exception of one, we haven't incorporated any traditional design patterns in our pattern language (INTERPRETER). On the other hand, these design patterns have the potential to be employed as architectural patterns when applied at the scale and level of a system's architecture. In the section that follows, this classification system is examined in more detail. As was previously said, this pattern language incorporates patterns from collections of existing architectural patterns. As a result, since these patterns have already been well explored, expressing them is not the primary goal of this language.

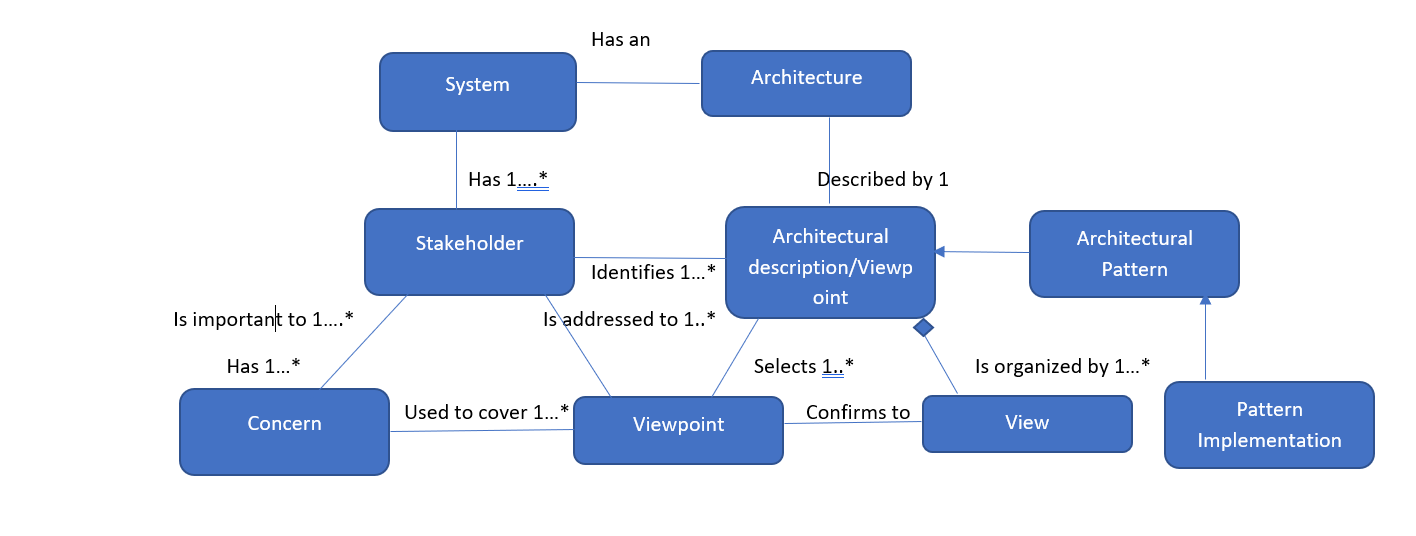
We won't go over these details again because of space considerations. Most of the focus is on the connected pattern sections that analytically describe the connections between the patterns. As a result, architects who are knowledgeable about these patterns and have an understanding of how they are connected make up the target market for this pattern language.

**CLASSIFICATION OF ARCHITECTURAL PATTERNS ACCORDING TO VIEWS:**

Architectural views are the foundation of our system for classifying architectural patterns. In an

architectural view, a system is presented from the viewpoint of a linked set of concerns (for example, how software components are allocated to network nodes in a distributed system). This model is composed of several system parts and the connections that link them.

In order to accurately document and represent viewpoints, the Viewpoint provides the various types of elements and relationships in addition to other meta-data about the views. A perspective is an example of a viewpoint for a specific system because the components and connections in it are instances of the right kinds of elements and connections in the viewpoint.



**Figure 1: Architectural views, viewpoints, and patterns**

The usage of the Unified Modeling Language to explore the relationships between views, viewpoints, and patterns is shown in Figure 1. (UML). UML semantics dictates that a perspective and a view must be realized simultaneously; as a result, the viewpoint acts as the specification and the view as the implementation of the latter. Our system of categorizing architectural patterns is based on some of the practices' most popular points of view. In light of this, an architectural pattern is considered to belong to a viewpoint if it serves to define it, or, to put it another way, if the pattern is a specialization of the perspective that defines it. The patterns' various and intricate relationships are highlighted.

Each pattern is given a principal perspective—the most appropriate—as part of our categorization strategy. A single design might be used in a second or third view, though this is quite uncommon. This is implied by the observation that both perspectives are discernible when two patterns from various points of view are combined on the same system. Consider a shared repository with a CLIENT-SERVER structure to demonstrate how the repository serves as a server to satisfy the needs of clients-accessors.

Both the data-centered and component interaction perspectives can be used to understand each of the two patterns in this instance. We have picked two categories of elements for our classification method: connectors, which act as connecting mechanisms between connectors, and components, which are units of runtime computation or data storage. Other views with other element types exist, but our pattern language does not yet apply to them. The "allocation" views, for instance, concentrate on how software components are split up among the natural parts, whereas the "module" views are concerned with implementation modules.

# OVERVIEW OF THE PATTERN LANGUAGES:

The component and connector views, as well as the patterns categorized in each view, make up the following pattern language. In order to draw attention to important connections or gaps in the realm of "traditional" architectural patterns, we have also added a few designs from other sources. In the pattern descriptions, we also discuss a few connections to different pattern languages.

*View of Adaptation*: In the view of adaptation, a system develops as a result of adjusting to its environment.

The *Language Extension View* looks at the abstraction layer that systems add to the computational architecture.

*User Interaction View:* The User Interaction View displays the user interface components' runtime arrangement.

The *Layered View* is interested in how a system, or a complicated heterogeneous object, could be broken up into working parts.

*Data Flow View:* The Data Flow View examines how data streams are sequentially processed or transformed by components.

*Data-centered Perspective:* When there are issues with how numerous components access a single central source of data, the data-centered view is helpful.

The next sections go over the various viewpoints and the corresponding patterns. The challenges each viewpoint addresses as well as the kind of components and interactions that are there are officially described. Before a more extensive investigation of how each pattern interacts with other patterns, each is briefly discussed.

# LAYERED VIEW

According to the Layered View, the system can be divided into interacting sections and is thought to be a complex heterogeneous entity. The pieces of the system are placed as far apart from one another as is practical. The interaction mechanisms between the components are implemented by connectors with the appropriate interfaces, states, and interaction protocols.

In order to preserve the effectiveness of the overall organizational structure, the different sections are often coordinated by an overall control mechanism. Figure 2 displays patterns and their connections from the Layered, Data Flow, and Data centered Views.

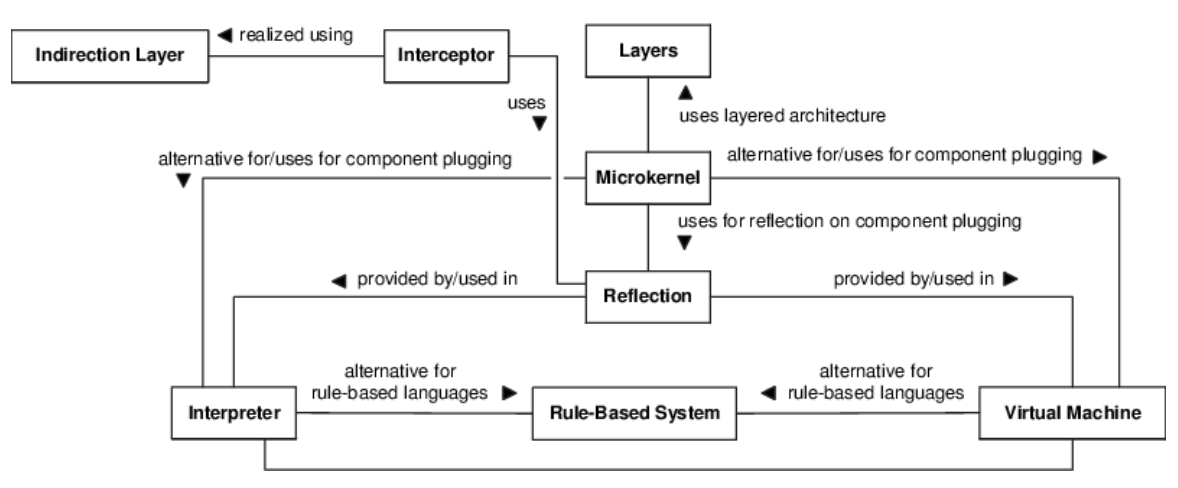


Figure 2: Overview: Layered, Data Flow, and Data-Centered Views Patterns.

# Layers as pattern

In order to preserve the effectiveness of the overall organizational structure, the various components are often coordinated by an overarching control mechanism. Figure 2 displays patterns and the connections between them from the Layered, Data Flow, and Data centered Views.

The system is structured up into LAYERS in order to achieve these aims, and each layer receives services from the one above it while simultaneously using the services provided by the layer below it. The component pieces of each LAYER all perform at the same abstraction level and communicate with one another using connectors. The result is the creation of a well-defined interface between two adjacent layers. Layers shouldn't be skipped in the original version of the pattern since higher-level layers can only access lower-level layers through the layer that is behind them.

# DATA FLOW VIEW:

In the Data Flow View, the system is represented as a set of transformations that are applied to streams of input data. The components with separate input and output ports are the pieces that perform the modifications. The components known as connectors carry data streams and have the same data-in and data-out purposes. Attachments that link input ports of components to data-out roles of connectors and output ports of components to data-in roles of connectors provide the interactions between these elements.

**Batch sequential is a pattern:**

Imagine a large task that can be divided into a number of smaller tasks, each of which can be expressed as a collection of unique calculations. Because it would be too difficult and impede reuse and modification, this shouldn't be done with a single monolithic component.

Using a batch sequential architecture, the entire task is broken up into discrete processing phases that are implemented as separate, independent components. Each step concludes the task at hand before executing the next step in the series and continuing until the entire assignment is accomplished. A batch of data is processed and transferred in its entirety to the next stage at each level.

**Pipes and filters in a pattern:**

Think about the possibility that a difficult job could be divided into a number of smaller tasks, each of which can be expressed as a collection of unique calculations, as in BATCH SEQUENTIAL. By processing data streams, the application also converts input data streams into output data streams. Because doing so would make the system too complex and limit its ability to be upgraded and reused, this capacity shouldn't be accomplished by a single monolithic component. Additionally, some clients require customized calculation modifications, such as the way the results are shown or the types of input data employed. To achieve this, each sub-task must be able to be dynamically built based on the requirements of the client.

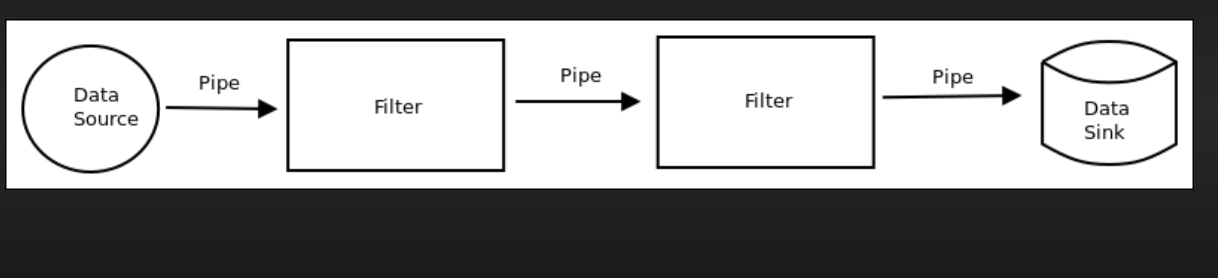


Figure 4: Pipes and filters example.

Figure 4 depicts PIPES AND FILTERS. This pattern allows forks/joins and feedback loops, however there is also a pipeline form that strictly sticks to a linear topology and prohibits both.

# DATA-CENTERED VIEW:

The system is shown in the data-centered approach as a common, permanent data store that diverse elements can access and update. Components refer to both the data storage and the areas that have access to it. There are specific components that are frequently maintained independently of the data storage. There might be several different data stores. Connectors to the data store(s) and accessors are the components that convey data written to or read from the data storage.

**Shared Repository as pattern:**

Between components, there must be data exchange. In sequential systems like LAYERS or PIPES AND FILTERS, the only way to move data between components is to send it with the invocation, which is wasteful when dealing with large amounts of data. The fact that the shared information may change from one invocation to the next and that the component interfaces must be ready to communicate a variety of data kinds may also make it inefficient. Last but not least, maintaining data persistence through time requires administration from a single location. In the SHARED REPOSITORY concept, one system component serves as a hub for all other independent components to access the system's data. This SHARED REPOSITORY permits suitable data access, for instance via a query API or language. Figure 5 shows an example of a SHARED REPOSITORY architecture.

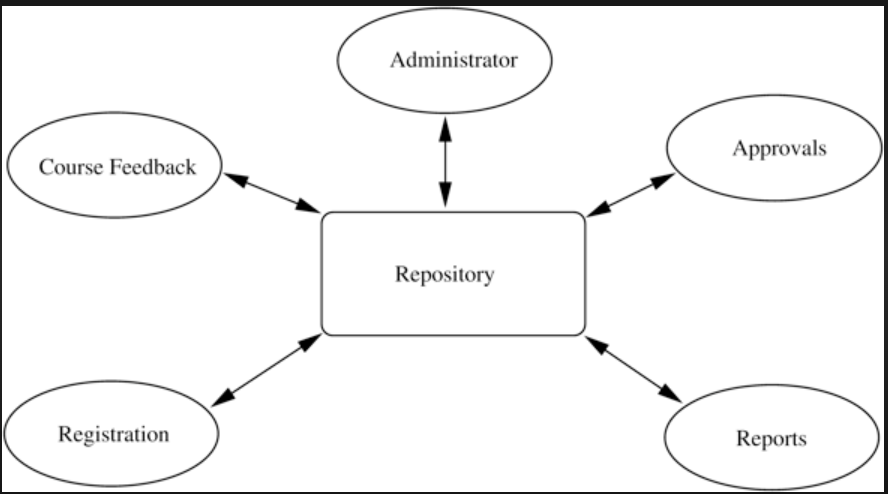


Figure 5: Shared repository example(clients are course feedback,registration……e.t.c)

**Blackboard as pattern:**

Think of a scenario in which a SHARED REPOSITORY is necessary for the computation's shared data but no deterministic solution techniques are offered. Applications like speech and picture recognition are just two examples. However, developing a solution for these kinds of applications ought to be simple. The difficult work is broken down into manageable sub-tasks in a BLACKBOARD architecture, each with deterministic answers. The BLACKBOARD is a SHARED REPOSITORY that computes heuristics and enhances step-by-step solutions using the results of its users. The BLACKBOARD is available to each client, who can use it to check for fresh inputs and to view findings after they have been processed. A control component keeps an eye on the blackboard and schedules the clients according to its condition .

A blackboard architecture is depicted in Figure 6. The control component may be a part of the clients, the blackboard, or a combination of the three, despite the fact that it was built as a separate component.

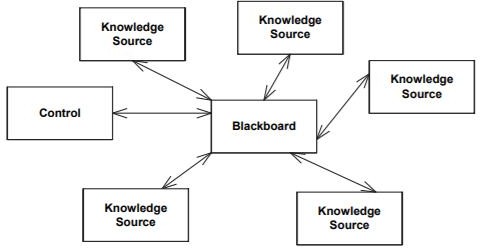


Figure 6: Blackboard example.

# ADAPTION VIEW:

The system is split into two pieces in the adaptation view: a core component that doesn't change over time and an adaptable component that does change over time or in different iterations of the system. This strategy uses two different sorts of elements: invariable components and adaptive components (these are often called variation points). Connectors with clear interfaces enable communication between these two types of components. It's important to note that a number of connectors include adjustment capabilities, making them suitable for use as variation points. Figure 7 summarizes the patterns and connections between the Language Extension View and the Adaptation View.

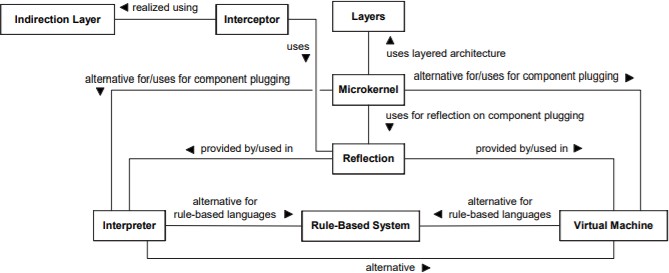


Figure 7: Overview: patterns of the Adaptation and Language Extension View.

**Microkernel as pattern:**

Consider a group of interconnected systems that must accommodate several variations of a single system. Each version allows for the development of components in a variety of methods, and other elements like the services that are available, the open APIs, and the user interfaces themselves may change. The system family should be designed using a similar design in order to facilitate software maintenance and reuse. For the services required by both individual systems and the complete family of systems, a MICROKERNEL offers a plug-and-play design.

Version-specific services are kept on private, MICROKERNEL-only internal servers that are not accessed by clients. Clients utilizing the MICROKERNEL get APIs and user interfaces from external servers. The MICROKERNEL architecture is only accessible to clients via external servers. The MICROKERNEL pattern promotes adaptable architectures that may alter in response to a system's requirements and interfaces. Figure 8 shows how a MICROKERNEL design works.

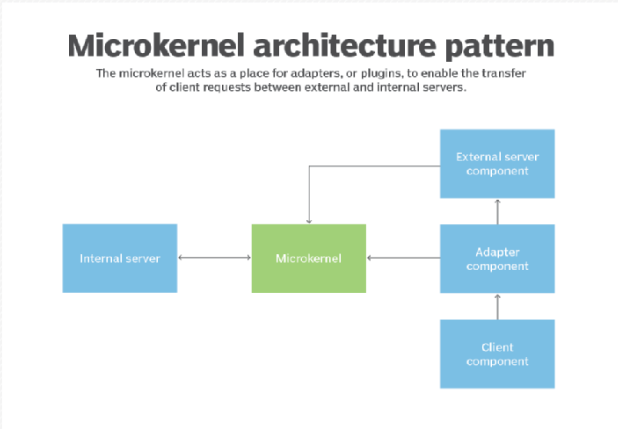
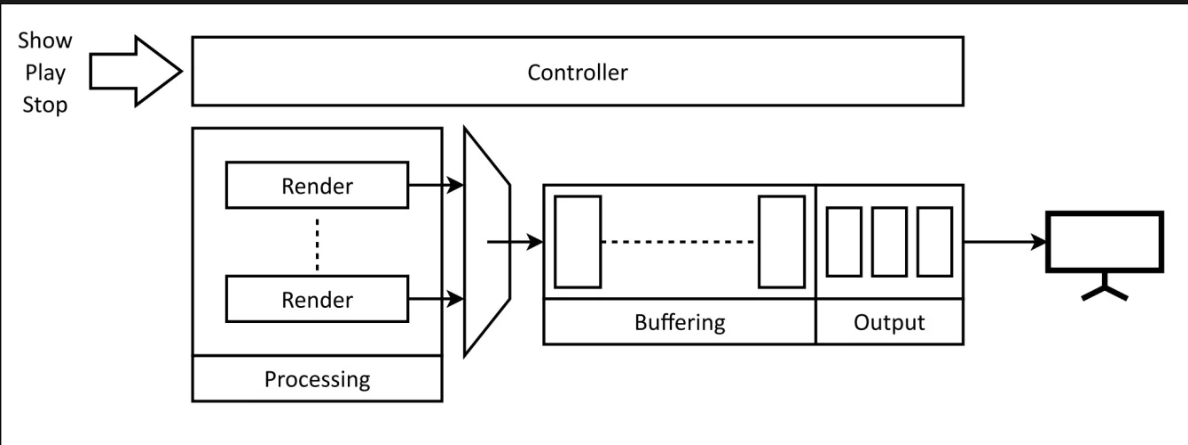


Figure 9: Microkernel example.

Software systems develop and change with time, and frequently unforeseen changes are required. It gets harder to react spontaneously when unexpected things happen. In a REFLECTION design, all of a system's structural and behavioral properties are included in meta-objects, keeping the parts responsible for application logic apart. The latter may consult the former to carry out their obligations (which may have changed at any time). As a result, REFLECTION makes it possible to define a system in a way that allows it to respond to unanticipated situations automatically.

The Reflection architecture is separated into layers, with meta-level meta-objects at the meta-level having variable structure and behavior and base-level application logic components that rely on the meta-objects. This is not a system with just layers, though, as the base layer also uses the services of the meta layer, and the meta layer may also use the base layer's services. For creating a whole reflecting system, this is helpful. Reflection can also be provided by other architectures. For instance, a lot of the present-day interpreters and virtual machines are reflective in the sense that they may offer reflection utilizing the information supplied in the pattern implementations. Figure 9 shows a reflection architecture.



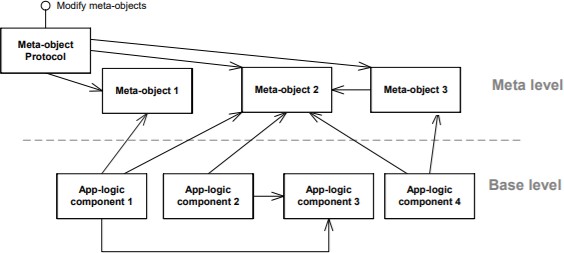


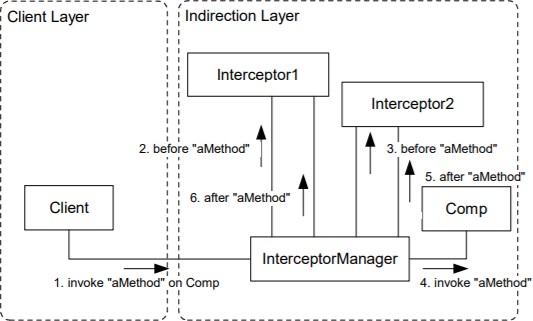
Figure 9: Reflection example.

# LANGUAGE EXTENSION VIEW

The Language Extension View divides the system into two halves, none of which are native to the software/hardware environment. Applications consist of both native and non-native parts. These components communicate indirectly by way of an interpreter component, which "translates" the latter into the former. Data containing program instructions written in the non-native language and the internal state of the non-native component serves as the link between these parts.

**Interpreter as a pattern:**

The syntax and grammar of a language must be parsed and interpreted by an application. The language has to be translated during runtime (i.e. using a compiler is not feasible). With the language comes an interpreter that can parse and run code. Scripts are used to build the interpretable program, which are then interpreted during runtime. These scripts can be executed on any platform that supports the INTERPRETER. The interpreter, for instance, may assign a class to each grammatical rule in the language. According to these rules, the interpreter's parser examines language instructions and assigns interpretation classes. There are several more sophisticated designs for Interpreters.



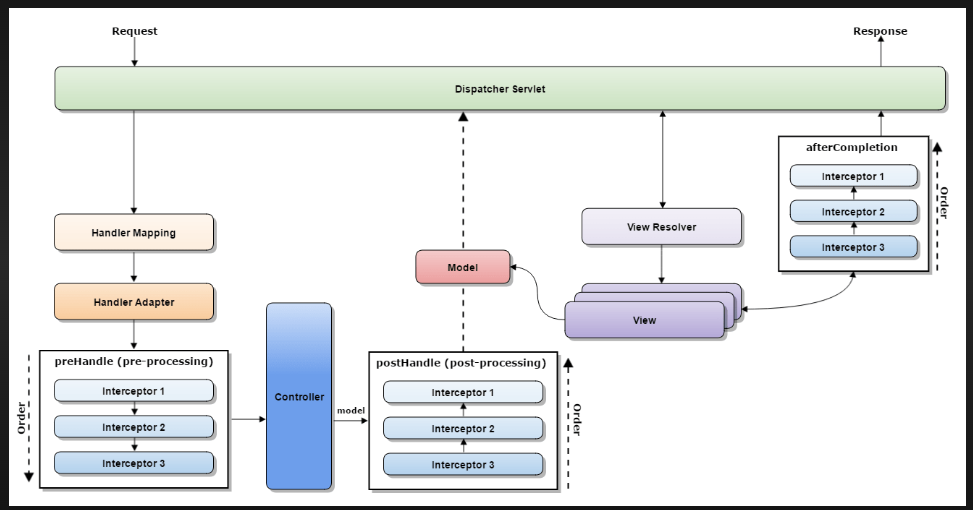


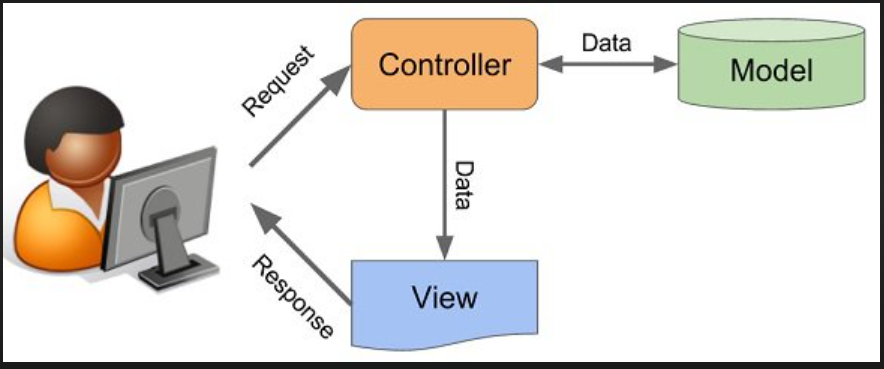
Figure 10: Interceptor example.

**USER INTERACTION VIEW:**

The system is depicted in the User Interaction View as two parts: one that displays the user interface and one that connects the application logic to the user interface.

**Model-View-Controller as a pattern:**

Various UIs could be offered in a framework. All or some of the data in the program are addressed by each user interface (UI). Every UI should promptly and correctly reflect changes to the information. Furthermore, any of the UIs should be upgradable without affecting the apps' logic that deal with information. A Model represents certain application data and the governing logic without the use of user interfaces. Additionally, it has a Controller connected to each View that gathers client data and transforms it into a request to the Model, as well as at least one View that shows the client a particular section of the information.



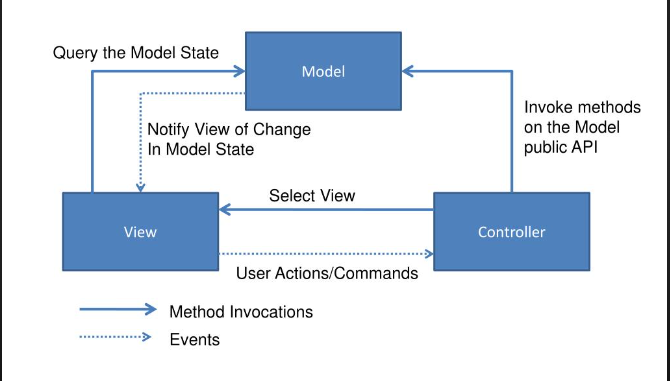


Figure 11: Model-View-Controller example.

**CONCLUSION:**

This study report suggests that in order to provide practitioners with access to a single complete source of patterns, existing architectural pattern methods should be merged into a pattern language. We concentrated on identifying the links between them in order to turn architectural patterns from a collection of unique patterns into a language. We have provided links to the designs' original sources for anyone who are interested in learning more about the complex details of each design .

Because compiling all architectural patterns into a cohesive pattern language is a substantial task, we limited this research to patterns from the early and key catalogues and categorizations that deal with components and connections. We are looking for input from other community members so that we can at least agree on these fundamental patterns.

**REFERENCES:**

<https://qph.fs.quoracdn.net/main-qimg-1421ccfea1ebd3321e9029adeba6c85a>

<https://www.researchgate.net/figure/summarizes-the-patterns-and-their-relationships-from-the-Adaptation-View-and-the-Language_fig5_221034694>

C. Alexander, S. Ishikawa, M. Silverstein, M. Jakobson, I. Fiksdahl-King, and S. Angel. A Pattern Language – Towns, Buildings, Construction. Oxford Univ. Press, 1977.

C. Alexander. The Timeless Way of Building. Oxford Univ. Press, 1979.

<https://home.cs.colorado.edu/~kena/classes/5828/s07/lectures/18/>

<https://www.techtarget.com/searchapparchitecture/tip/What-is-a-microkernel-architecture-and-is-it-right-for-you>

L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison-Wesley Longman, Reading, MA, 1998.

Frank Buschmann and Kevlin Henney. Explicit interface and object manager. In Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPlop 2003), Irsee, Germany, July 2003.

<https://charlestolman.com/2016/06/15/accu2016-talk-on-software-architecture-design-4-a-design-example/>

Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. Documenting Software Architectures: Views and Beyond. Addison- Wesley, 2002.

Coplien. Software Patterns Management Briefing. SIGS, 1996.

<https://stackoverflow.com/questions/3984296/model-view-controller-design-pattern-code-example>

J. Coutaz. PAC, an Object Oriented Model for Dialog Design. In Proceedings Interact’87, Stuttgart, pages 431–436. IEEE Computer Society, 1987.

<https://www.slideserve.com/maik/model-view-controller-mvc-architecture>

J. Garcia-Martin and M. Sutil-Martin. Virtual machines and abstract compilers- towards a compiler pattern language. In Proceedings of EuroPlop 2000, pages 375–396, Irsee, Germany, July 2000.