A Fuzzy ELECTRE Approach towards Selection of Software Architecture Styles

266-FBAS/MSSE/F09

Shamsul Arif

**Supervisor**

Mr. Shahbaz Ahmad

Assistant Professor Dept. of CS & SE

**Co-Supervisor**

Mr. Qaisar Javaid

Assistant Professor Dept. of CS & SE

IIU Islamabad

Department of Computer Science & Software Engineering

Faculty of Basic and Applied Sciences

INTERNATIONAL ISLAMIC UNIVERSITY,

ISLAMABAD
ABSTRACT
Selection of Software Architecture style for any system is a difficult task as many different stakeholders are involved in the selection process. Stakeholders' view on quality requirements is different and at times they may also be conflicting in nature. Also selecting appropriate styles for the software architecture is important as styles impact characteristics of software (e.g. reliability, performance). Moreover, styles influence how software is built as they determine architectural elements (e.g. components, connectors) and rules on how to integrate these elements in the architecture. Selecting the best style is difficult because there are multiple factors such as project risk, corporate goals, limited availability of resources, etc.
Contents

1. Introduction: .......................................................................................................................... 4

1.1 Software Architecture: ........................................................................................................ 4

1.2 Software Architectural Style: ............................................................................................ 4

1.3 Common Architectural Styles .............................................................................................. 5

1.3.1 Pipes and Filters ............................................................................................................. 5

1.3.2 Data Abstraction and Object-Oriented Organization ......................................................... 7

1.3.3 Event-based, Implicit Invocation ..................................................................................... 7

1.3.4 Layered Systems ............................................................................................................. 9

1.3.5 Repositories .................................................................................................................. 10

1.3.6 Table Driven Interpreters ............................................................................................... 12

1.4 Quality Attribute of Architectural Style: .......................................................................... 17

1.5 Architectural Style Selection: ............................................................................................ 13

1.6 MCDM Approach: ............................................................................................................. 13

1.6.1 Introduction: .................................................................................................................. 13

1.6.2 ELECTRE Method ......................................................................................................... 14

1.7 Fuzzy Logic ....................................................................................................................... 15

1.8 Why Use Fuzzy Logic? ....................................................................................................... 15

2. Literature Survey: ............................................................................................................... 16

3. Quality Attributes of Architectural Styles .......................................................................... 17

4. The Problem Statement ...................................................................................................... 21

5. Research Question: ............................................................................................................. 21

6. Proposed Solution ............................................................................................................... 22

7. Expected Outcome .............................................................................................................. 22

8. Research Methodology ...................................................................................................... 22

8.1. Action Research: ............................................................................................................ 23

8.2. Survey: ............................................................................................................................ 23

8.3. Aims and Objectives of Research: .................................................................................. 23

8.4. Sample Population: ........................................................................................................ 24

8.5. Data Collection Sources and Method: ............................................................................ 24

8.5.1. Primary Data: ............................................................................................................ 24

8.5.2. Secondary Data: ........................................................................................................ 24

9. References: ........................................................................................................................ 25
1. Introduction:

1.1 Software Architecture
Software architecture is being widely used today to describe a very high level design methodology of large software systems. Software architecture represents the overall structure of a system in an abstract, structured manner. The architecture deals with the structure of the components of a system, their interrelationships and guidelines governing their design and evolution over time. So, a description of architecture is composed of identifiable components of various distinct types [1]. The components comprising architecture could include objects, clients and servers, databases, filters and layers. These components could interact using several mechanisms such as procedure calls, message sending, shared variable access, clients/server protocols. The general purpose software architecture can be defined in terms of its components, component interactions, constraints, and control structure. Some commonly occurring patterns of the structural organization of components and connectors have been identified, and these patterns are referred to as architecture styles. An architecture style defines a family of software systems in terms of a pattern of software organization. The software architecture conforms to some style. Therefore, since every software system has the architecture, every software system has a style, and styles must have existed since the first software system was developed.

Software architectures significantly impact software project success [2]. However, creating architectures is one of the most complex activities during software development [3]. When creating architectures, architecture styles narrow the solution space: First, styles define what elements can exist in architecture (e.g. components, connectors). Second, they define rules on how to integrate these elements in the architecture. Moreover, styles address non-functional issues (e.g. performance) [4].

1.2 Software Architectural Style
A family of systems in terms of a pattern of structural organization, a vocabulary of components and connectors, with constraints on how they can be combined, Architectural styles are reusable 'packages' of design decisions and constraints that are applied to architecture to induce chosen desirable qualities [5].
1.3 Common Architectural Styles

We now examine some of these representative, broadly-used architectural styles. Our purpose is to illustrate the rich space of architectural choices, and indicate, what are some of the tradeoffs in choosing one style over another? To make sense of the differences between styles, it helps to have a common framework from which to view them. The framework we will adopt is to treat architecture of a specific system as a collection of computational components—or simply components—together with a description of the interactions between these components—the connectors. Graphically speaking, this leads to a view of an abstract architectural description as a graph in which the nodes represent the components and the arcs represent the connectors. As we will see, connectors can represent interactions as varied as procedure call, event broadcast, database queries, and pipes. An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition. Given this framework, we can understand what a style is by answering the following questions: What is the structural pattern—the components, connectors, and constraints? What is the underlying computational model? What are the essential invariants of the style? What are some common examples of its use? What are the advantages and disadvantages of using that style? What are some of the common specializations? [5]

1.3.1 Pipes and Filters

In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed. Hence components are termed “filters”. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed “pipes”. Among the important invariants of the style, filters must be independent entities: in particular, they should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes. Furthermore, the correctness of the output
of a pipe and filter network should not depend on the order in which the filters perform their incremental processing—although fair scheduling can be assumed. Figure 1 illustrates this style. Common specializations of this style include pipelines, which restrict the topologies to linear sequences of filters; bounded pipes, which restrict the amount of data that can reside on a pipe; and typed pipes, which require that the data passed between two filters, have a well-defined type. A degenerate case of pipeline architecture occurs when each filter processes all of its input data as a single entity. In this case the architecture becomes a “batch sequential” system. In these systems pipes no longer serve the function of providing a stream of data, and therefore are largely vestigial. Hence such systems are best treated as instances of a separate architectural style. The best known examples of pipe and filter architectures are programs written in the Unix shell. Unix supports this style by providing a notation for connecting components (represented as Unix processes) and by providing run time mechanisms for implementing pipes. As another well-known example, traditionally compilers have been viewed as pipeline systems (though the phases are often not incremental). The stages in the pipeline include lexical analysis, parsing, semantic analysis, and code generation. (We return to this example in the case studies.) Other examples of pipes and filters occur in signal processing domains, functional programming, and distributed systems. Pipe and filter systems have a number of nice properties. First, they allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters. Second, they support reuse: any two filters can be hooked together, provided they agree on the data that is being transmitted between them. Third, systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be replaced by improved ones. Fourth, they permit certain kinds of specialized analysis, such as throughput and deadlock analysis. Finally, they naturally support concurrent execution. Each filter can be implemented as a separate task and potentially executed in parallel with other filters. But these systems also have their disadvantages. First, pipe and filter systems often lead to a batch organization of processing. Although filters can process data incrementally, since filters are inherently independent, the designer is forced to think of each filter as providing a complete transformation of input data to output data. In particular, because of their transformational character, pipe and filter systems are typically not good at handling interactive applications. This problem is most severe when incremental display updates are required, because the output pattern for incremental updates is radically different from the pattern for filter output. Second, they may be hampered by having to maintain correspondences between two separate, but related streams. Third, depending on the implementation, they may force a lowest common denominator on data transmission.
resulting in added work for each filter to parse and unparsed its data. This, in turn, can lead both to loss of performance and to increased complexity in writing the filters themselves [5].

1.3.2 Data Abstraction and Object-Oriented Organization

In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects—or, if you will, instances of the abstract data types. Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation). Objects interact through function and procedure invocations. Two important aspects of this style are (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and (b) that the representation is hidden from other objects. Figure 2 illustrates this style. The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread. There are many variations. For example, some systems allow “objects” to be concurrent tasks; others allow objects to have multiple interfaces. Object-oriented systems have many nice properties, most of which are well known. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients. Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents. But object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other object. This is in contrast, for example, to pipe and filter systems, where filters do not need to know what other filters are in the system in order to interact with them. The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. In a module oriented language this manifests itself as the need to change the “import” list of every module that uses the changed module. Further there can be side effect problems: if A uses object B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa [5].

1.3.3 Event-based, Implicit Invocation

Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines. However, recently there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast. This style has historical roots in systems based on actors, constraint satisfaction, daemons, and packet-switched networks. 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. For example, in the Field system, tools such as editors and variable monitors register for a debugger’s breakpoint events. When a debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke methods in those registered tools. These methods might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools (if any) are concerned with that event, or what they will do when that event is announced. Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events. Procedures may be called in the usual way. But in addition, a component can register some of its procedures with events of the system. This will cause these procedures to be invoked when those events are announced at run time. Thus the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls. The main invariant of this style is that announcers of events do not know which components will be affected by those events. Thus components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events. For this reason, most implicit invocation systems also include explicit invocation (i.e., normal procedure call) as a complementary form of interaction. Examples of systems with implicit invocation mechanisms abound. They are used in programming environments to integrate tools, in database management systems to ensure consistency constraints, in user interfaces to separate presentation of data from applications that manage the data, and by syntax-directed editors to support incremental semantic checking. One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system. A second benefit is that implicit invocation eases system evolution. Components may be replaced by other components without affecting the interfaces of other components in the system. In contrast, in a system based on explicit invocation, whenever the identity of a provides some system function is changed, all other modules that import that module must also be changed. The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system. When a component announces an event, it has no idea what other components will respond to it. Worse, even if it
does know what other components are interested in the events it announces, it cannot rely on the order in which they are invoked. Nor can it know when they are finished. Another problem concerns exchange of data. Sometimes data can be passed with the event. But in other situations event systems must rely on a shared repository for interaction. In these cases global performance and resource management can become a serious issue. Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked. This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure’s pre- and post-conditions when reasoning about an invocation of it [5].

1.3.4 Layered Systems
A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers. Figure 3 illustrates this style [5].

![Layered Systems Diagram](image)

Fig.1 Layered Systems

The most widely known examples of this kind of architectural style are layered communication protocols. In this application area each layer provides a substrate for communication at some level of abstraction. Lower levels define lower levels of interaction,
the lowest typically being defined by hardware connections. Other application areas for this style include database systems and operating systems. Layered systems have several desirable properties. First, they support design based on increasing levels of abstraction. This allows implementers to partition a complex problem into a sequence of incremental steps. Second, they support enhancement. Like pipelines, because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers. Third, they support reuse. Like abstract data types, different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers. This leads to the possibility of defining standard layer interfaces to which different implementers can build. (A good example is the OSI ISO model and some of the X Window System protocols.) But layered systems also have disadvantages. Not all systems are easily structured in a layered fashion. (We will see an example of this later in the case studies.) And even if a system can logically be structured as layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations. Additionally, it can be quite difficult to find the right levels of abstraction. This is particularly true for standardized layered models. One notes that the communications community has had some difficulty mapping existing protocols into the ISO framework: many of those protocols bridge several layers. In one sense this is similar to the benefits of implementation hiding found in abstract data types. However, here there are multiple levels of abstraction and implementation. They are also similar to pipelines, in that components communicate at most with one other component on either side. But instead of simple pipe read/write protocol of pipes, layered systems can provide much richer forms of interaction. This makes it difficult to define system independent layers (as with filters)—since a layer must support the specific protocols at its upper and lower boundaries. But it also allows much closer interaction between layers, and permits two-way transmission of information [5].

1.3.5 Repositories
In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems. The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database. If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard. Figure 4 illustrates a simple view of blackboard architecture. (We will examine
more detailed models in the case studies.) The blackboard model is usually presented with three major parts [5]:

**The knowledge sources:** separate, independent parcels of application dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard [5].

**The blackboard data structure:** problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem [5].

**Control:** driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable [5].

In the diagram there is no explicit representation of the control component. Invocation of a knowledge source is triggered by the state of the blackboard. The actual locus of control, and hence its implementation, can be in the knowledge sources, the blackboard, a separate module, or some combination of these. Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition. They have also appeared in other kinds of systems that involve shared access to data with loosely coupled agents [5].

![Diagram of the Blackboard](image)

*Fig. 2 The Blackboard*

There are, of course, many other examples of repository systems. Batch sequential systems with global databases are a special case. Programming environments are often organized as a collection of tools together with a shared repository of programs and program fragments.
Even applications that have been traditionally viewed as pipeline architectures may be more accurately interpreted as repository systems. For example, as we will see later, while compiler architecture has traditionally been presented as a pipeline, the “phases” of most modern compilers operate on a base of shared information (symbol tables, abstract syntax tree, etc.) [5].

1.3.6 Table Driven Interpreters
In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter’s analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of its execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudo-code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated [5]. (See Figure 5.)

Interpreters are commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. We occasionally speak of a programming language as providing, say, a “virtual Pascal machine” [5].

1.4 Quality Attributes
Measurements of software quality attributes, is one of the important concepts in software architecture evaluation, and variety of techniques are used for analyzing specific quality attributes in a system. Promoting one quality attribute requirement usually has an adverse
effect on some other quality attribute requirement. Architectural decisions will promote some quality attribute requirements while inhibiting others, thus resulting in quality attribute trade-off decisions. These trade-offs are best dealt within the earliest phases of system development during the design of the architecture [6].

1.5 Architectural Style Selection

One of the important parts in software design process is the selection of software architecture style to design good software architecture. Choosing the most suitable architecture style(s) among existing ones can help us in satisfying functional and especially non-functional requirements correctly and precisely [7]. Also selecting appropriate styles for the software architecture is important as styles impact characteristics of software (e.g. reliability, performance). Moreover, styles influence how software is built as they determine architectural elements (e.g. components, connectors) and rules on how to integrate these elements in the architecture. Selecting the best style is difficult because there are multiple factors such as project risk, corporate goals, limited availability of resources, etc [7].

1.6 MCDM Approach

1.6.1 Introduction

MCDM is a multi criteria decision making method, including several techniques, which allow rating a range of criteria, and then ranking them with the opinions of industry experts. The MCDM method has high potential to reduce the cost and time and increase the accuracy of decisions and can be an appropriate framework for solving the problems. With this characteristic, decision makers have the possibility to easily examine the problem and scale it in accordance with their requirements. As we all know that the decision of the decision maker depends upon the various factors in which knowledge and experience are the most important one therefore Analytical Hierarchy process completely agrees with the behavior of the decision maker. This decision analysis methodology is multi criterion therefore; it allows subjective as well as objective factors during the process execution in order to make the precise decision. This is the reason that the managers are getting the rationales on the basis of which they can easily make the decisions [8].

MCDM methodology is a powerful tool for clarifying and documenting the importance assigned to different factors in ranking risks. At the same time, the application of such a formal analysis requires that decision makers articulate their value structure, including the prioritization factors they consider and the value placed on each prioritization factor. Some
MCDM methods are well suited to developing consensus among decision makers particularly when there are numerous stakeholders with different values [8].

There are many cases in daily life and especially in business work where decision making become more complex due to multiple and conflictive situations. In such cases one criteria favour one option while other criteria favour other option. Similarly human judgment is involved and by nature humans are subjective. There are techniques available in literature which deals such type of problems. In this section we will introduce one of the MCDM methods named ELECTRE and then by using this method and applying that to expert’s opinions, a comparison between the mentioned business models will be made [8].

1.6.2 ELECTRE method

ELECTRE is one of the MCDM techniques. This technique allows decision makers to select the best choice with maximum advantage and minimum conflict in the function of various criteria. ELECTRE creates the possibility to model a decision process by using coordination indices. These indices are concordance and discordance matrixes and they are as decision makers that can accept or reject one object against another [8].

ELECTRE Algorithm consists of five steps as mentioned in [28-8]:

**Step1:** Three sets of data A, I and W are the primary data needed for starting the algorithm.

\[ A = \{ A_i | i = 1, 2, \ldots, n \} \] Represents a set of feasible alternatives.

\[ I = \{ C_j | j = 1, 2, \ldots, m \} \] Represents a set of evaluation criteria.

\[ W = \{ W_j | j = 1, 2, \ldots, m \} \] Represents the weight of each criterion specified by experts.

**Step2:** After specifying the alternatives and criteria, a questionnaire should made from the experts in order to specify the weight of each criterion and also the performance value of each alternative in each criterion.

**Step3:** In this step, concordance matrix has constituted.

Concordance index measures the preference rate of one object over another, according to the weights assigned to the decision criteria. The concordance index for two alternatives index h and k is defined as follow:

\[
C(h, k) = \frac{W^*}{W^* + W^-}, \forall h, k ; h \neq k
\]
Where:
\[ W^+ = \sum_{j=1}^{l} W_j, \quad W^- = \sum_{j=1}^{l} W_j, \quad W^0 = \sum_{j=1}^{l} W_j \]

And:
\[ I^+ = \{ C_j \mid g_j(A_h) > g_j(A_k) \} \]
\[ I^- = \{ C_j \mid g_j(A_h) = g_j(A_k) \} \]
\[ I^- = \{ C_j \mid g_j(A_h) < g_j(A_k) \} \]

Is the performance value for alternative Ah in criterion j.

**Step4:** In this step discordance matrix has constituted. Discordance index measures the worse rate of one object against one another. The discordance index for two alternatives h and k is defined as follow:
\[ D_j(h, k) = \sum_{j=1}^{l} \frac{g_j(A_h) - g_j(A_k)}{\max_{j=1}^{l} \{ g_j(A_h), \theta_j \}}, \forall h, k ; h \neq k \]

\( \theta \) refers to the R-degree parameter used by a decision maker for criteria j to represent the degree of attention devoted by the decision maker to criteria j.

**Step5:** Both index values are then compared with the respective threshold (Sc and Sa) to determine their significance. Concordance and discordance values are considered significant if:
\[ C(h, k) \geq S_c, \quad D(h, k) < S_a \]

### 1.7 Fuzzy Logic
Fuzzy logic is all about the relative importance of precision: How important is it to be exactly right when a rough answer will do? Fuzzy logic is a convenient way to map an input space to an output space. This is the starting point for everything else, and the great emphasis here is on the word “convenient.”

### 1.8 Why Use Fuzzy Logic?
- Fuzzy logic is conceptually easy to understand.
  The mathematical concepts behind fuzzy reasoning are very simple. What makes fuzzy nice is the “naturalness” of its approach and not its far-reaching complexity.
- Fuzzy logic is flexible.
  With any given system, it’s easy to massage it or layer more functionality on top of it without starting again from scratch.
- Fuzzy logic is tolerant of imprecise data.
Everything is imprecise if you look closely enough, but more than that, most things are imprecise even on careful inspection. Fuzzy reasoning builds this understanding into the process rather than tacking it onto the end.

- Fuzzy logic can model nonlinear functions of arbitrary complexity.

You can create a fuzzy system to match any set of input-output data. This process is made particularly easy by adaptive techniques like ANFIS (Adaptive Neuro-Fuzzy Inference Systems).

- Fuzzy logic can be built on top of the experience of experts.

In direct contrast to neural networks, which take training data and generate opaque, impenetrable models, fuzzy logic lets you stand on the shoulders of people who already understand your system.

- Fuzzy logic can be blended with conventional control techniques.

Fuzzy systems don’t necessarily replace conventional control methods. In many cases fuzzy systems augment them and simplify their implementation.

- Fuzzy logic is based on natural language.

The basis for fuzzy logic is the basis for human communication. This observation underpins many of the other statements about fuzzy logic.

2. Literature Survey:

K. Delhi Babu et al. [7] presented a more precise and suitable decisions in selection of architecture styles have been presented by using ANP inference to support decisions of software architects in order to exploit properties of styles in the best way to optimize the design of software architecture.

M. Galster et al. [9] conducted a study that presents a method, called SYSAS, for the systematic selection of architecture styles. In SYSAS, style selection is based on (a) Characteristics of basic architectural elements that are relevant for the developer, and (b) Characteristics of the target system that is visible to the end user. The selection procedure requires ratings about the importance of characteristics of architectural elements and results in a ranking of styles. SYSAS can be applied at system level as well as for choosing styles for individual subsystems. A case study is presented to illustrate SYSAS and its applicability and added benefit. Additional case studies are performed to compare results of SYSAS with judgments of experts.
K. Delhi Babu et al. [10] present a method, called SSAS, for the selection of software architecture styles. Moreover, this selection is a multi-criteria decision-making problem in which different goals and objectives must be taken into consideration. In this paper, the authors suggest an improved selection methodology, which reflects interdependencies among evaluation criteria and alternatives using analytic network process (ANP) within a zero-one goal programming (ZOGP) model.

S. Moaven et al. [11] designed a Decision Support System (DSS) has been designed which provides software architects with more precise and suitable decisions in architecture styles selection. The DSS uses fuzzy inference to support decisions of software architects and exploits properties of styles in the best way while making decisions.

K. Delhi Babu et al. [12] propose a new AHP-GP-Visualization model which is integrated approach of AHP-GP and visualization. This model considers both the functional and nonfunctional requirements. AHP-GP model alone is focused on nonfunctional requirements only and visualization techniques focus on functional key areas. In this proposed framework model, overcome the limitations of the existing models for software architecture selection.

N. Eftekhar et al. [13] propose that what kind of architecture styles is suitable for which applications on the basis of their quality attributes. The expert judgment was done in order to find relative effectiveness.

3. Quality Attributes of Architectural Styles

The following attributes are generic in the reported literature for arch style selections the detail explanation of these attributes are available in [14], [15] & [6].

Software quality is defined as the degree to which software possesses a desired combination of attributes. The architecture is created based on a set of requirements that it has been to fulfill. A quality attribute can be defined as a property of a software system. A quality requirement is a requirement that is placed on a software system by a stakeholder; a quality attribute is what the system actually presents once it has been implemented. Properties of the system (such as policies and mechanisms built into the system) and its environment that have an impact on the concerns. Depending on the attribute, the attribute-specific factors are internal or external properties affecting the concerns. Factors might not be independent and
might have cause/effect relationships. Factors and their relationships would be included in the system’s architecture:

**Performance factors** — the aspects of the system that contribute to performance. These include the demands from the environment and the system responses to these demands.

**Dependability impairments** — the aspects of the system that contribute to (lack of) dependability. There is a causal chain between faults inside the system and failures observed in the environment. Faults cause errors; an error is a system state that might lead to failure if not corrected.

**Security factors** — the aspects of the system that contribute to security. These include system/environment interface features and internal features such as kernelization.

**Safety impairments** — the aspects of the system that contribute to (lack of) safety. Hazards are conditions or system states that can lead to a mishap or accident. Mishaps are unplanned events with undesirable consequences.

In this section we will describe some of system quality attributes which expected to be the most troublesome. Those attributes are selected which are more troublesome in practice.

**Maintainability:**
Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specification.

**Reliability:**
The capability of the software product to maintain a specified level of performance when used under specified conditions.

**Usability:**
Concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides

**Testability:**
The capability of the software product to enable modified software to be validated

**Portability:**
The capability of the software product to be transferred from one environment to another, the environment may include organizational, hardware or software environment, meaning that the system is not bounded to a specific platform/location to offer its service.

**Reusability:**
Stating that an element can be easily reused, it’s necessary that elements agree on the type of received and produced objects. Reusability depends on degree of dependency among components and it is better to be less.
**Modifiability:**
The modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specification.

**Performance:**
Performance is defined as the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage

**Security:** Is a measure of the system’s ability to resist unauthorized usage while still providing its services to legitimate users.

**Availability:**
Availability is measured as the limit of the probability that the system is functioning correctly at time $t$, as $t$ approaches infinity. This is the Steady-State availability of the system. It may be calculated as:

$$\alpha = \frac{MTTF}{MTTF + MTTR}$$

Where MTTF is the mean time to failure, and MTTR is the mean time to repair. During the development of the architecture it is therefore important to validate that the architecture has the required quality attributes, this is usually done using one or more architecture evaluations. Quality attributes of large systems can be highly limited by a system’s requirements and constraints. Thus, it is in our best interest to try to determine as early as possible whether the system will have the desired qualities. Quality requirements should be described concretely before architecture is developed.
Table 1: Quality attributes related to each architecture style [13].

<table>
<thead>
<tr>
<th>Style</th>
<th>Quality Attribute Mapping Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>layered</td>
<td>If the quality is addressed in a positive way the score is “+”, if using this style will penalize the system on matter of that particular quality the score is “-”, and finally if the style is neutral towards that quality the score is “+/-” [13].</td>
</tr>
<tr>
<td>Client/Server</td>
<td></td>
</tr>
<tr>
<td>Pipes &amp; Filters</td>
<td></td>
</tr>
<tr>
<td>Repository</td>
<td></td>
</tr>
<tr>
<td>Object-Oriented</td>
<td></td>
</tr>
<tr>
<td>Event Based</td>
<td></td>
</tr>
<tr>
<td>Blackboard</td>
<td></td>
</tr>
<tr>
<td>Batch Sequential</td>
<td></td>
</tr>
<tr>
<td>Virtual Machine</td>
<td></td>
</tr>
<tr>
<td>Main Program/Setup (FSC)</td>
<td></td>
</tr>
<tr>
<td>Implicit Invocation</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Quality attributes related to each architecture style [13].

Style – Attribute mapping description:
If the quality is addressed in a positive way the score is “+”, if using this style will penalize the system on matter of that particular quality the score is “-”, and finally if the style is neutral towards that quality the score is “+/-” [13].

It is important to consider the most positive as well as the most negative influences of imposing an architectural style. If a system has to be designed for high reusability, then the communication between different components of architecture should be loosely coupled. This causes reduction of interrelationship between components. For example, layered, pipe & filter and data-centered styles, could be good selections in this case. If a system has to be designed for high reusability, then the components should have modular features. Since each component can be tested independently, modification of a component or sub-component will not cause other components to be modified. For example, pipe & filter and object oriented styles are good selections in this case [13].
4. The Problem Statement

It is known from the literature review that the Architecture style selection is multi criterion decision making problem because of which selection of Software Architecture style for any system is a difficult task as stake holders’ conflicts arise in the selection process. In previous research different MCDM approaches were applied to evaluate different architecture styles, like AHP, AHP-GP, ANP, and ANP-GP but in these approaches the issues like vagueness, impreciseness and uncertainty were not considered.

5. Research Question:

In order to achieve the above objectives we investigated the following questions.

Q1. Which architecture style is most suitable with respect to QAs based on Multi-criteria Decision Making?

Q2. What is the relative importance of the reported criteria for selection of architecture styles?
6. Proposed Solution

In order to overcome the identified limitations, the Fuzzy ELECTRE Approach for the selection of Software Architecture style has been applied to find the relative importance among NFRs & Arch styles as shown in fig 4.

The proposed approach is comprised of the following steps:

**Step 1:**
The first step was the identification of relative importance of each arch style w.r.t Quality Attributes using Fuzzy ELECTRE Process, for the trade-off analysis.

**Step 2:**
After trade-off analysis of each arch style w.r.t Quality Attributes we obtained a prioritized form of arch style.

**Step 3:**
In this step we compared the proposed approach with existing approaches to validate the result of proposed solution.

**Step 4:**
In conclusion, we obtained the desired blue print as a reference tool to be used helpful for selecting architectural style w.r.t Quality Attributes.

7. Expected Outcome

The outcome of this research is the prioritized form of architectural style selection to provide a support for decision-makers in choosing the best architectural style having its association with the required quality attributes for a specific domain.

8. Research Methodology

8.1 Survey:
Surveys enable the researcher to obtain data about practices, situations or views at one point in time through questionnaires or interviews. Quantitative analytical techniques are then used to draw inferences from this data regarding existing relationships. The use of surveys permit a researcher to study more variables at one time than is typically possible in laboratory or field experiments, whilst data can be collected about real world environments. A key weakness is that it is very difficult to realize insights relating to the causes of or processes involved in the phenomena measured. There are, in addition, several sources of bias such as
the possibly self-selecting nature of respondents, the point in time when the survey is conducted and in the researcher him/herself through the design of the survey itself.

8.2 Aims and Objectives of Research:
The aim of this research is to do quantitative analysis of the reported software architecture styles and their correspondence with non functional attributes. This analysis will give a decision framework for the selection of software architecture styles on the basis of relative importance of non functional attributes. To achieve this aim we have set the following objectives:
- To identify relative importance of each reported attribute and style.
- To facilitate decision making and policy changing

8.3 Sample Population
An expert pool will be selected because ELECTRE is decision making process. Proper profiling will be done in order to select those people whose expertise exists in architectural level. Their demographic variables will be:
- Organization type
- Experience duration
- Roles and responsibilities
- Department
- Country
- Age
The sample size will be 80—100 responses in order to do statistical analysis, keeping guidance from previous research in this domain.

8.4 Data collection Sources and Method:
Data collection will be divided into two categories, primary and secondary data. Normally the primary data is gathered by the researcher in order to address the problem at hand for the specific purpose. Secondary data is the data which has already been collected and is available in research journals, literature, papers, etc.

8.4.1 Primary data:
The primary data will be collected from the sample population which will be selected for this study using questionnaire designed. Action research, survey and interview will be conducted
to get data from sample population. Then Geometric mean will be calculated of the group data.

8.4.2 Secondary data:
Secondary data will be collected from relevant literature in order to validate the primary data.

9. References:


