International Journal of MC Square Scientific Research Vol.1, No.1,2009

Review of Software Architectural styles for Artificial Intelligence systems

B.VINAYAGA SUNDARAM[1]

[1]Computer Center, MIT Campus Anna University Chromepet Chennai-600044, Tamilnadu India

Abstract:

Artificial Intelligence is the ability to process information properly in a complex environment. The criteria of properness are not predefined and hence not available beforehand. They are acquired as a result of information processing. The last decade, however, has seen an unprecedented interest in this area, both within the research community and among software practitioners in the industry. In this research, a new methodology is proposed to manage and structure the complexity of these systems, viz. architecting the system in a proper way. An article presents the various software architectural styles and its applications. The major contribution of paper is how to manage the increased complexity of software intensive Artificial Intelligence systems. In particular, concerned with the management of complexity of system whose structure exhibits some form of flexibility due to either changes or failures.

Keywords: Artificial Intelligence, information processing, complexity, software architecture, Layered systems

1. Introduction

Many human mental activities such as writing programs, solving mathematical problems, engaging in common sense reasoning, understanding a language and even driving a car are said to be intelligent activities. Over the past, many systems that have been built can perform these tasks to a reasonable level. More specifically, there are systems that can diagnose diseases, prove mathematical theorems, solve differential equations and even understand a limited amount of human speech also. Such systems are said to possess some degree of intelligence. In the context of biology, intelligence connects perception of the environment to actions that are necessary for the goals of life. Intelligence is the computation in the service of life, just as metabolism is chemistry of life. Intelligence does not imply perfect understanding and every intelligent being has a limited perception, memory and computational capability. AI seeks to understand the computations required for intelligent behavior and to build computer systems that exhibit some degree of intelligence. In practical terms, AI means ability to automatically perform activities that require human operators. An AI system should also exhibit the following features:

(i) It should have the flexibility in dealing with variability in the environment in an appropriate manner.

(ii) It should have more autonomy and less human intervention or monitoring.

(iii) It should understand what the user wants from limited instructions.

(iv) It should improve its performance by learning from experience.

The definition according to Hideyuki Nakashima (Nakashima 1999), for Intelligence is as follows:“Intelligence is the ability to process information properly in a complex environment. The criteria of properness are not predefined and hence not available beforehand. They are acquired as a result of information processing”.The key concept is that the properness of the information is not predetermined. The essence of intelligence lies in symbolic processing. Even though human intelligence is unlikely to consist solely of symbolic processing, still it relies on symbols. AI can be included as a sub-field of information processing, but with a major crucial difference. In information processing complete processing is presupposed whereas in AI it is neither presupposed nor even possible. The essence of intelligence lies in the method of processing complex information as little as necessitated by the environment. That is a small portion of the complex information is sufficient to process it adequately. Stuart Russell (Russell 1995) defines AI as the study of bounded optimality, or the ability of the system to generate maximally successful behavior given the available information and computational resources. One of the important approaches in complex information processing in AI is heuristics. The term heuristics refers to a method that succeeds in normal cases but is not guaranteed to do so. This term can be synonymously used with ad-hoc method. Heuristics is one of the central and essential aspects of AI.

2. SYSTEMS AND AI

In the previous section, it is stated that AI is the study of complex information processing. Simple systems are not intelligent, but conversely, being complex is by no means a sufficient condition for intelligence. A complex system is usually composed of many elements, which interact with one another. The complexity of the system is proportional to various factors such as the number of elements, the number of interactions in the system, and the complexities of the elements and their interactions. In naturally complex systems, every element is also complex in nature. The global behavior of the system arises from the interactions of these elements of the system. In this sense, we can say that a complex system is more than the sum of its parts. A complex system has properties not present in its parts. These properties are called emergent. They emerge from the interactions of the components of the system. These are specialized components together with logic-based languages that can express propositions and speech acts about these propositions (Sowa 2002). There are no efficient methodologies for designing and maintaining complex systems. It is claimed that evolution is one of the proven methodologies to build a complex system. This may work if the aim is to exhibit complex behavior. However, the primary goal is to design and implement an intelligent system. The question of building an intelligent system by evolution can be answered in two ways. Firstly, it is possible to build an intelligent system provided sufficient time is given. Secondly, it takes to long to be practical. Hence, we need a design methodology for building AI systems, which are large and complex in a quicker way. In this research, a new methodology is proposed to manage and structure the complexity of these systems, viz. architecting the system in a proper way. In the proposed method, hybrid-layered software architecture for AI system is designed and implemented to manage and structure the complexity.

3. Software Architecture

The structure and organization of software systems have been discussed, to a certain extent, since the late 1960s. A well-known example from the early literature on this topic is an influential and popular article by Parnas (1972). The last decade, however, has seen an unprecedented interest in this area, both within the research community and among software practitioners in the industry. In one of the papers in the literature of software architecture (Perry et al 1992), it has been claimed that software design, while receiving much attention in the 1970s, was largely overlooked during the 1980s. This paper uses the term software architecture instead of design to evoke notions of a professional discipline and to make analogies with other fields, such as construction engineering and computer architecture. Software architecture of a system describes the structure, organization of components/ modules and their interactions not only to satisfy the systems‟ functional and non-functional requirements but also to provide conceptual integrity to the overall system structure.

Software architecture is concerned with the structure of large software intensive systems (David Garlan 2000). The architectural view is an abstract view that separates the details of implementation, algorithm and data representation and concentrates on behavioral aspects and interaction among the various components. In other words, the software architecture is a high-level design specification of the system, which provides an abstract description of the system by exposing certain properties and hiding others (Rikard Land 2002).

Hence, the software architecture plays an important function with respect to following aspects in the development of large software intensive systems:

(i) Understandability: It helps to understand a large system by the appropriate level of abstraction. It also exposes the high-level design constraints, thereby providing a way for making architectural decisions.

(ii) Reusability: Architectural designs support the reuse of large components and provide framework into which components can be integrated.

(iii) Construction: An architectural description provides a blue print for the development of a system indicating the major components and the relationships amongst them.

(iv) Evolution: The architectural description of a system separates the functionality from implementation, thereby permitting us to manage the concerns regarding performance, reusability and prototyping in an easy way.

(v) Analysis: The architectural description provides a new attribute for analyzing the system with respect to quality, performance, dependency etc,. Moreover, analysis of architectures built with different styles can also be made to arrive at good architectural design decisions (Hasan Reza et al 2005).

(vi) Management: Successful development of software addressing specific application depends on critical selection, analysis and evaluation of software architecture.

4. Definitions Of Software Architecture

qThe recent interest in the field has resulted in variety of definitions for software architecture. This section presents and discusses some of the most influential of these definitions. Perry and Wolf (1992) presents the following model of software architecture:

Software Architecture = {Elements, Form, Rationale}.

The elements of architecture can be processing elements, data elements, or connecting elements (which may themselves be processing elements or data elements or both). The form specifies constraints on elements and their interactions among each other. The rationale provides motivations on the choice of elements and the form. Although, nobody seems to question the value of documenting the rationale for software architecture, more recent definitions tend to view rationale as not being part of the architecture itself. In the first book on the topic (Shaw et al 1996), the software architecture of system is defined, as a collection of computational components–or simply components–together with a description of the interactions among these components–the connectors. This definition inspired the practitioners and tends to represent software architectures informally in the form of box and line diagrams. For such diagrams to be useful for others than their creators, it is important that the meanings of both the boxes (components) and the lines (connectors) are described.

The terminology and definitions of Shaw and Garlan (1995) have become widely adopted within the field. It has also been somewhat criticized, however, for instance in a book by staff members from the Software Engineering Institute (SEI) (Bass et al 2003) The authors argue that the term connector is inappropriate since it indicates a run-time mechanism, while software architecture also covers structures that are not observable at run-time. In the second edition of the book, the term component is also avoided since it has become so closely associated with the topic of component-based software engineering, wherein components are usually viewed as run-time entities. The latest edition of the SEI book uses the following working definition:

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the inter relationships among them”.This definition has some interesting aspects. The notion that a system may have multiple structures is closely related to the concept of architectural views, which is now widely accepted by the research community and in industry.

The definition further, states that architecture includes the externally visible properties of components, implying that other component properties are not part of the architecture. Finally, a recommended practice for architectural documentation from the Institute of Electrical and Electronics Engineers (IEEE 2003) defines architecture as:

"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution".

The main focus of this definition is its mention of the system‟s environment. This is also an example of a process-oriented definition that includes design and evolution principles. As is the case with rationale, the majority of the literature seems to consider such principles to be important but distinct from the architecture itself.

5. Architectural Design

It was described earlier how Perry and Wolf (1992) selected to use the term software architecture instead of the more traditional term software design. The question still arises, however, as to the precise relationship between architecture and design. A more general view which is expressed in the literature by the most popular work by Clements et al 2002 as 'architecture is design, but not all design is architecture'. In other words, a system‟s software architecture comprises some, but not all, the decisions made in the design of the system. The definitions presented in the previous section do, to varying degrees, specify which types of design decisions architecture should include. It can generally be said that software architecture is concerned with high-level design decisions that are made at an early stage of the design process. The term architectural design is often used to characterize structural issues concerned during the process such as: global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives from design solution space (Ince et al 1998). The SEI book (Bass 2003) presents guidelines for making architectural decisions that help to ensure a system‟s quality properties. Decisions that target particular properties are called architectural tactics. For example, fault-tolerance is an availability tactic and information hiding is a modifiability tactic. A set of related tactics is called an architectural strategy. Bosch (2000) suggests a method of architectural design wherein an initial architecture is designed based on the system‟s functional requirements. The architecture is then evaluated against the non-functional requirements for the system and transformed if necessary(LawrenceChung et al 1999). Various architecture analysis methods are proposed (Liliana Dobrica e al 2002). This process of evaluation and transformation is applied iteratively until the architecture is believed to meet all functional and non-functional requirements. An approach developed by Siemens Corporate Research (Hofmeister et al 2000) focuses on identifying factors that influence architectural issues, which are classified into technical, organizational, and product factors. Based on analyses of these factors, strategies are determined to resolve the issues. The early design of a system‟s architecture is also a central concept in the Rational Unified Process (Kruchten 2000). In this influential process model, a stable architecture is the main milestone of the elaboration phase, which precedes the labor-intensive construction phase. In all engineering disciplines, successful solutions to past problems are often used as models when new problems are to be solved. This is also true for software architecture, where architects have primarily drawn on their own experiences or that of their development organizations. The research community has realized the benefit of having a collection of well-documented prototype solutions.

The term architectural style is used to denote such a prototype solution. This term have also been used by Shaw and Garlan (1996). Drawing on their definitions of software architecture, the following definition of architectural style is given below:

- **Pipes and filters:** The components in this style are called filters and each has a set of inputs and a set of outputs. The outputs of a filter can be attached to inputs of other filters via simple connectors called pipes as shown in Figure 1.1. Typically, the filters transform streams of input data to streams of output data in an incremental fashion. An important constraint is that filters should be independent in the sense that they do not share state and each filter is unaware of the identities of the other filters it is connected to.

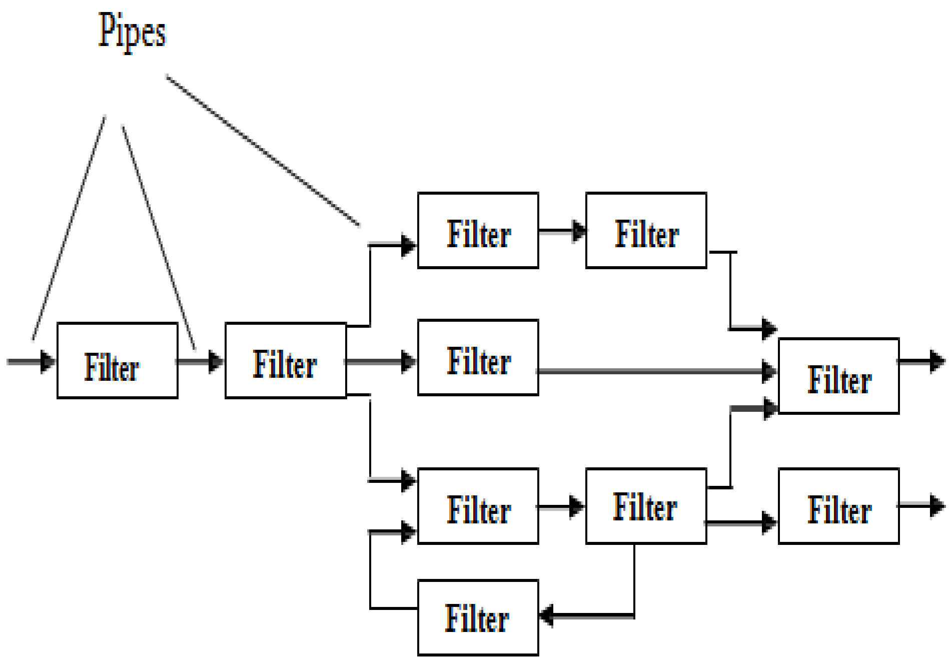

Figure 1.1 Pipe and filter architecture

• **Layered systems:** The components in this style are called layers and are commonly thought of as being stacked on top of each other as illustrated in Figure 1.2. Each layer provides services to the layer above it and is a client of the layer below it. The connectors are defined by the protocols used between the layers. A variation of the style is systems where a layer may use the services provided by all lower layers.

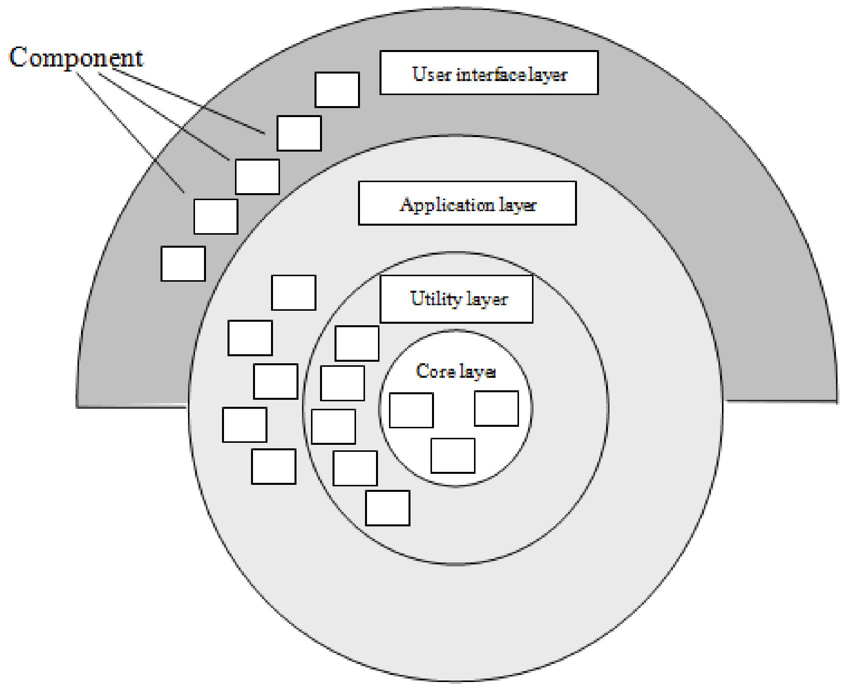

Figure 1.2 Layered architecture

- **Data centered architectures:** In this style, there are two distinct types of components: a central data store that represents the state of the system and a set of independent components that operate on the data store as depicted in Figure 1.3. An interesting sub-style is systems where computation is entirely controlled by the state of the data store and the independent components react to changes to this state in an opportunistic fashion

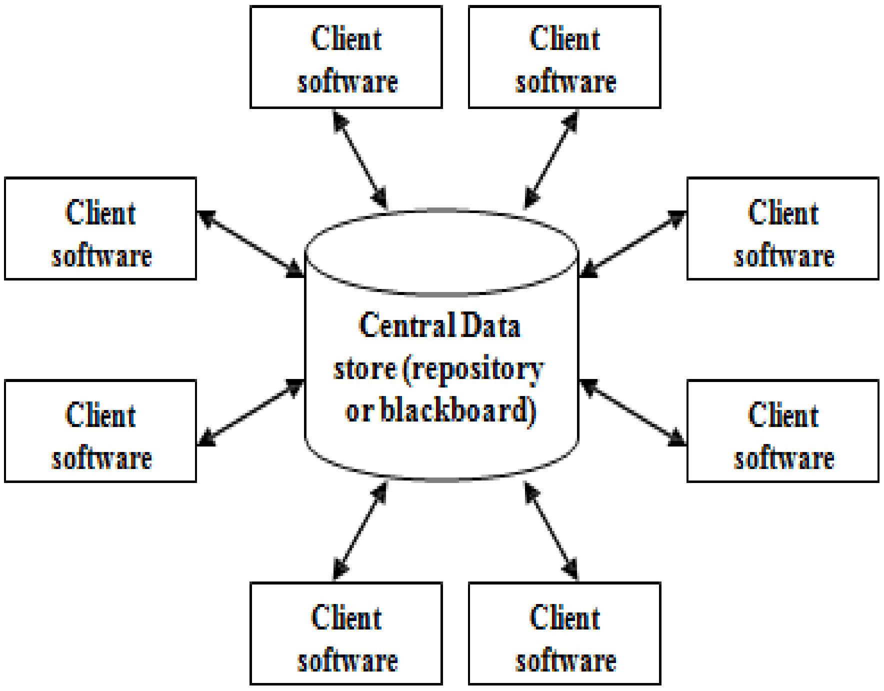

Figure 1.3 Data centered architecture

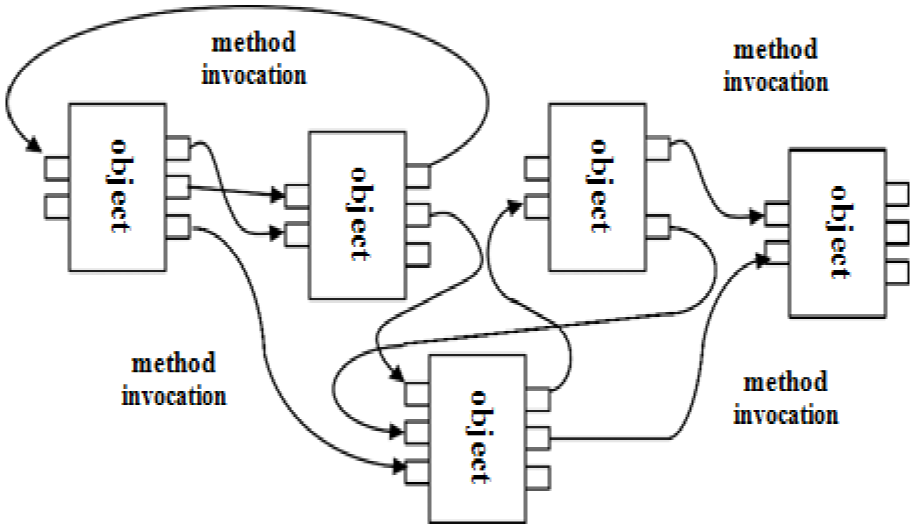

Figure 1.4 Object oriented architecture

Surjati Typically, the sets of components and connectors are dynamic, since objects can create and delete other objects and object references can be passed as parameters to operations.

- **Event-based systems**: The components in this style have interfaces that provide both operations and events. A component‟s operations may be invoked directly by other components as in object-oriented systems as illustrated in Figure 1.5. In addition, a component may register an interest in an event that another component provides by associating one of its own operations with it. When the second component subsequently announces the event, the registered operation is invoked, along with any operations that other components have registered. Thus, there are two distinct types of connectors in this style. A valuable property of these and other common styles is that the consequences of using them as the basis for a system‟s software architecture are fairly well understood. The pipes and filters style, for instance, results in systems of highly independent components, where filters can suitably be developed and tested separately and possibly reused in different configurations. A possible disadvantage is that all filters have to comply with the data format required by the pipes, which may not be optimally suited for their computation and result in loss of performance and increased internal complexity.

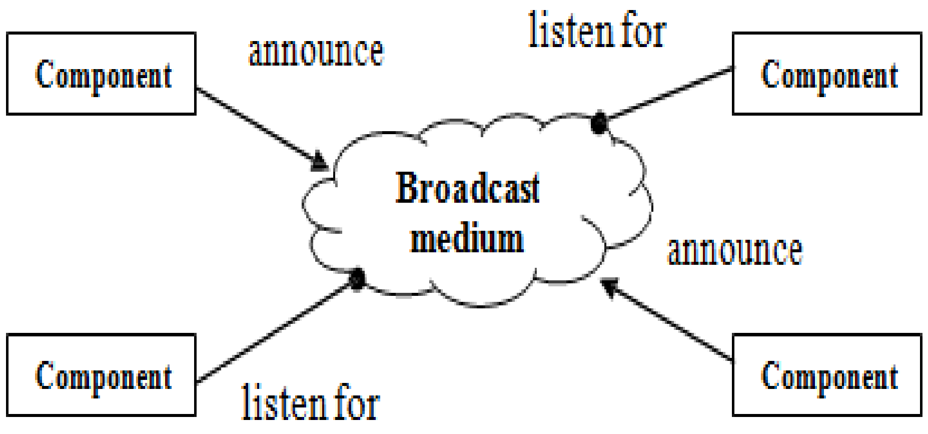

Figure 1.5 Event based architecture

An advantage of object-oriented systems is that algorithms and data representation are encapsulated such that it can be maintained locally. On the other hand, system wide modifications, such as adding new objects, can be difficult since objects need to know the identity of other objects in order to invoke their operations. Event-based systems represent a possible solution to this problem, although the components are not as independent as in the pipes and filters style. A common approach in practice is that systems can incorporate several architectural styles. For instance, a system may have components and connectors that match the types defined by several styles. An example is a layered event-based system where each layer provides both operations and events to the layer(s) above it. Another way to combine styles is to mix different components and connectors in the same system, which is sometimes called heterogeneous architectures (Pressman 2006). For instance, a part of a system could be organized as a repository wherein one or more of the independent components exchange data with another part of the system that consists of pipes and filters. Hierarchical heterogeneity occurs when a component in a system of one style is internally organized using another style. A common example is a layer containing an object structure, which may even be reflected in the layer‟s services.

A popular approach within the software engineering community is the use of object-oriented design patterns (Gamma et al 1995). Since architecture is commonly viewed as a special case of design, it is not surprising that the patterns paradigm has also been applied to architectural design. The most comprehensive work in this area has been done by at the German company Siemens, and this approach is called pattern-oriented software architecture (Bushmann et al 1996). As with other design patterns, this effort focuses on cataloging known solutions to known problems in given contexts. This approach is similar that of identifying and documenting architectural styles, and there is now a widespread view that patterns and styles are synonymous.

6. Conclusion

This paper presents the various software architectural styles and its applications. The major contribution of this paper is how to manage the increased complexity of software intensive AI systems. In particular, concerned with the management of complexity of system whose structure exhibits some form of flexibility due to either changes or failures. The main limitation in the present method of approach on software architecture is the lack of comprehensiveness in the design and implementation of layered software architecture for AI system has been addressed in the proposed method. By comprehensive approach, we mean a component oriented architectural description with:

Detailed description of the system.

(i) Clarification of desired level of flexibility in the architecture and the relation of flexibility to application semantics.

(ii) Formalism of environment requirements.

(iii) Evaluation of the software in the architecture in terms of their functional requirements and nonfunctional requirements.

References

1. Abhay Kothari and Ramani A.K. (2006), „Qualitative Assessments of Software Architectures of Configuration Management systems‟, Journal of Computer Science, Vol.2, No.1, pp.07-12.

2.Adnan Rawashdeh and Bassem Matalkah (2006), „A New Software Quality Model for Evaluating COTS component‟, Journal of Computer Science, Vol.2, No.4, pp.373-383.

3. Surjati, 3. Androutsopoulos I., Koutsias J., Chandrinos K.V. and Spyropoulos C.D. (2000), Experimental Comparison of Naive Bayesian and Keyword Based Anti-spam filtering with personal e-mail messages‟, Proc. of SIGIR-00, 23rd ACM International Conference on Research and Development in Information Retrieval, Athens,Greece, pp.160-167.

4. Arkin R.C. (1990), „Integrating Behavioral, Perceptual, and world knowledge in reactive navigation‟, Journal of Robotics and automation, Vol.6, pp.105-122 Elsevier Publishers.

5. Babloyantz A. (1986), „Molecules, Dynamics and Life. An Introduction to Self- Organization of Matter‟, II Edition, Wiley and Sons.

6. Bass L., Clements P. and Kazman R. (2003), „Software Architecture in Practice‟, II Edition , Addison-Wesley.

7. Belkin N.J. and Croft W.B. (1992), „Information Filtering and Information Retrieval: Two Sides of the Same Coin?‟, Communications of ACM, Vol.35, No.12, pp.29-38.

8. Surjati, 3. Androutsopoulos I., Koutsias J., Chandrinos K.V. and Spyropoulos C.D. (2000), Experimental Comparison of Naive Bayesian and Keyword Based Anti-spam filtering with personal e-mail messages‟, Proc. of SIGIR-00, 23rd ACM International Conference on Research and Development in Information Retrieval, Athens,Greece, pp.160-167.

9. Bonasso R.P., Kortenkamp D., Miller D.P. and Slack M. (1996), „Experiences with an Architecture for Intelligent, Reactive Agents, Intelligent Agents‟, Proc. of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95), Lecture Notes in Artificial Intelligence Springer-Verlag, Vol.1037, pp.187-202.

10.Booch G. (1994), „Object-Oriented Analysis and Design with Applications‟, II Edition, Addison-Wesley.

11.Bosch J. (2000), „Design and Use of Software Architectures‟, I Edition, Addison- Wesley.

12.Burmeister B. and Sundermeyer K. (1992), „Cooperative Problem-Solving Guided by Intentions and Perception‟, Decentralized A.I., Vol.3, pp.36-47.

13.Bushmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M. (1996),Pattern-Oriented Software Architecture - A System of Patterns‟, I Edition, John Wiley & Sons.

14.Castelfranchi C. (1995), „Guarantees for Autonomy in Cognitive Agent Architecture, in Intelligent Agents: Theories, Architectures, and Languages‟, Springer-Verlag: Heidelberg, Germany, Lecture notes in Artificial Intelligence, Vol.890, pp.56-70.

15.Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R. and Stafford J. (2002), „Documenting Software Architectures: Views and Beyond‟, I Edition, Addison-Wesley.

16.David Garlan (2000), „Software Architecture: a Roadmap‟, In the future of Software Engineering, pp.91-101, ACM press.

17.David Ramamonjiso and Issam A. Hamid (1999), „Design and Implementation of Multi Agent for Intelligent Software‟, IEEE Computer society, pp.268-271.

18.Dolores Del Castillo and Jose Ignacio Serrano (2004), „A Multistrategy Approach for Digital Text Categorization from Imbalanced Documents‟, Sigkdd Explorations, Vol.6, No.1, pp.70-77.

19.Drucker H., Wu D. and Vapnik V. (1999), „Support Vector Machines for Spam Categorization", IEEE Trans. on Neural Networks, Vol.10, No.5, pp.1048-1054.

20.Elaine J. Weyuker (1999), „Evaluation Techniques for Improving the Quality of Very large Software Systems in Cost Effective Way", The Journal of Systems and Software, pp.97-103.