Programming Methodologies and Software Architecture

A Rama Mohan Reddy ***Dr. M M Naidu*** ***Dr. P Govindarajulu***

Sri Venkateswara University College of Engineering
Tirupati – 517 502., Andhra Pradesh., India.

Abstract
Software quality is the major issues in software engineering discipline. The complexity of a program forces for better software design methodologies for enhancing the quality of software system. Researchers and practitioners proposed many program design methodologies. In the recent years, the software architecture is evolved as a way of software development that mainly focuses on computational units and overall structure of system rather than lines-code, called components. One of the characteristics of Software architecture is that it provides a higher level of abstraction. At higher level of abstraction, evaluation of quality attributes like reusability, substitutability and reliability of the software systems become easy. Software architecture supports many modeling techniques. Designers use these models to understand the underlying design issues, to evaluate functional and non-functional requirements and to communicate design decision to its stakeholders. For the better understanding of various aspects of Software Architecture such as evolution, description language, styles, evaluation and applicability, are discussed. This survey starts from various software development methodologies and goes up to software architecture.

Key words*: Methodology, Software Architecture, Reusability, design, Implementation.*

Introduction

Software Systems are being evolved and crossed successfully so many hurdles. Complexity of the system increases with size. Quality is plays important role in software development as in section 2. In this, evolution process we noticed from its evolution the importance of Software development design methodologies. While the demand for software systems from different applications, the size and complexity of systems have been increased and opened a door for new methodologies to deal with the size and complexity. Quality, reusability, substitutability, and modifiability are became very important factor in software engineering. In section 3, we considered our ideas of deriving the program from the problem domain.

Though concepts are not new but we put our ideas in a different way and shown diagrammatically for better understanding the conversion process of requirements. In section 4, the nature of the software has been considered its properties were discussed. Section 5 covered all the conventional and unconventional design methods and in section, 6 and 7 the component-based software engineering and design patterns are discussed. From section 8, onwards architectural evolution and subsequent development in architecture-based concepts are presented. In conclusion, we compared all the design methods based on the degree of abstraction they support.

2. Evolution Process

Engineering disciplines such as Civil, Electrical and Mechanical Engineering's have reliable methods for analysis, design, fabrication, and Testing and are currently being in use. Many changes have been taken place in the above disciplines up to 1980. Most of the effort of people had been expended on hardware and the engineers struggled for improving reliability, quality, efficiency, and usability, as those were major hurdles. Hardware had been improved its stability and people started to develop large and complex systems and they were proved to highly fault tolerant and reliable. Hardware was the leader and a little amount of software was used and embedded in it. However, software engineering was a new discipline at the beginning of the computers era and its design methodologies were still in its infancy and biding state. These applications were small in size and had a less complexity. The developers had an idea that hardware had more functionality and greater role in controlling and coordinating activities than Software. They considered programming activity as an art and it is not an engineering activity.

3. The Program Development Concepts

In 1980's programmers were faced lots of problems to develop large and complex programs because the reliability and quality were major issues. This was due to the lack of reliable and established methodologies for developing programs. There was a great demand from the society for large, complex, quality and reliable software systems. Only a few methods were available for developing applications, finding its quality, and testing of programs. Research shows that these systems lacked an architecture and plan. There were many specifications, and structures, but Integrity was lost. Despite all the specifications and structures, systems were essentially using Code and Fix. After some time, the fundamental design problems were addressed [2].

The primary use of a computer in an organization is not exactly to substitute a person but to assist him in many aspects. As shown in Figure (1), an employee, who is working in an organization, has the responsibility of performing a set of activities. As shown Figure (2), a question naturally arises that what activities are to be assigned to the computer. Here, there is a need to consider two different domains for better understanding the problem. As in Figure (3), first one is problem ***domain,*** and second, ***computer domain.***

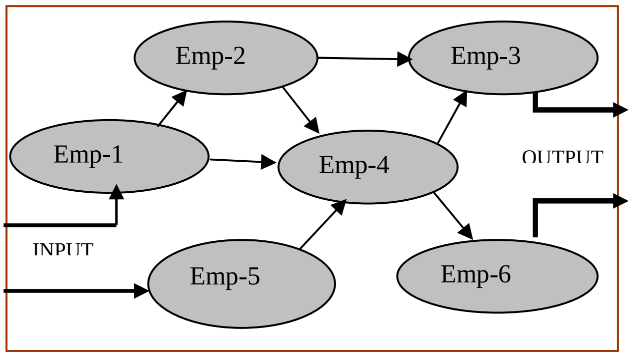

Fig. 1 A typical organization without computers.

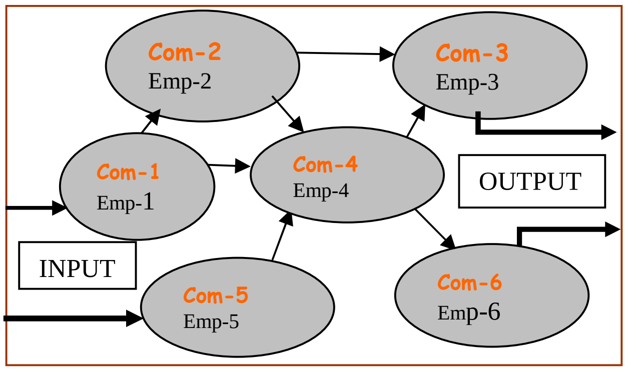

Fig. 2 The same organization with computers.

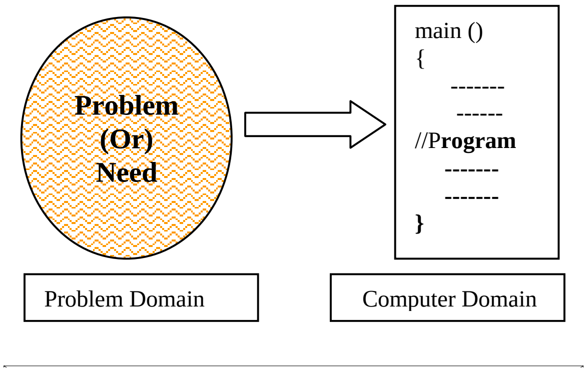

Fig. 3 A Mapping from Problem domain to Computer domain

The Problem domain has well defined infrastructure, like a language for communication, procedures to conduct business, and expertise to cater the needs of the business organization. The case with computer domain is it has also a language and methods to execute a predefined set of activities. But *these two domains are entirely different from each other.* A computer cannot understand directly the real world activities of various organizations. However, to solve this problem one possible solution is, to make the computers to read and understand real world activities directly and processes them or a developer learns some techniques that enable him to directly communicating his intensions to a computer system in machine understandable form. In this context, we are *precipitating* the issue as ***problem to program conversion***. Then, how to extract the computer based solution from problem space. A good mapping method is required to bridge the gap between Problem and Program as shown in figure (4) and (5).

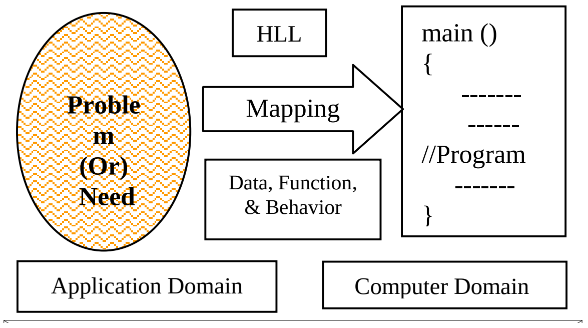

Fig. 4. A Program = Data, Function, & Behavior of Application

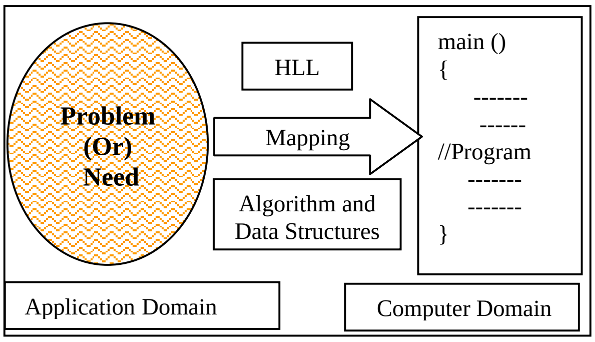

Fig. 5. Program = Algorithm + Data Structures

Primarily, the conversion methodology has two important activities to be performed; one is to understand the business and second is, take the abstraction of it. An higher-level of abstraction makes procedures easy to understand and implement. *How an activity and its associated data can take a smooth transition to become itself an algorithm and data structures*. In the real world, people and some controlling machinery are performing activities. In a typical organization, the software engineer actually assigns a subset of activities of an employee to a computer. Then the computer performs the defined tasks and assists the people.

4. Nature of the Software and demand

The basic computer executes the instructions sequentially from the top to the bottom of the list. Software is intangible artifact does not have shape and feel, so, visualization of design is hard. Therefore, it is very difficult to assess the software product for its quality attributes and the amount work involved in it. This is one of the reasons for underestimating the cost, time and effort of program frequently. The software development process is mainly labor-intensive work, requires skilled people. Normally it is easy to develop a piece of software but it is very difficult to understand and modify properly without understanding its complete functional as well as technical designs.

At the end of the 1980's due to computer revolution and the effect of computer on the society, more and more users have had shown interest for using the computers in their areas, i.e., in business organizations, scientific and engineering, had created a lot of demand for both the software and hardware. Number people had involved for developing large, complex, and qualitative applications and caused to increase development cost. The actual cost of software had become many more times than hardware. The same hardware could be used differently in different situations to meet various needs of the user using different programs, e.g., Engineering, Scientific, Business, Communication, Multimedia and Data Base Applications. The aspect of software in a computer system was becoming many times more important and demanded careful attention. This intern demanded good and effective methodologies to develop programs. Developing more robust, complex, quality, error free and reliable programs are posing many challenges on software development methodologies and on developers.

Researchers and Industry people proposed a number of methodologies for developing large and complex systems with high quality, and low-cost. Changes in Business process, errors in the software, portability, advancements in technology are some of the reasons for modifications in software. Modifications are mandatory in the *long-lived and green software*. Program modifications with out much understanding of the business process and its development methodology other wise they create chaos in the application. The change must be managed properly.

5. Program Design Methodologies

5. 1 Functional Programming

The functional programming, in which a program is viewed as a set of mathematical functions and equations, describing a relation between input and output. Prolog [Widstrm, 1987; Leler, 1987] supports this perspective. This programming method has different levels of abstraction as shown in figure (6).

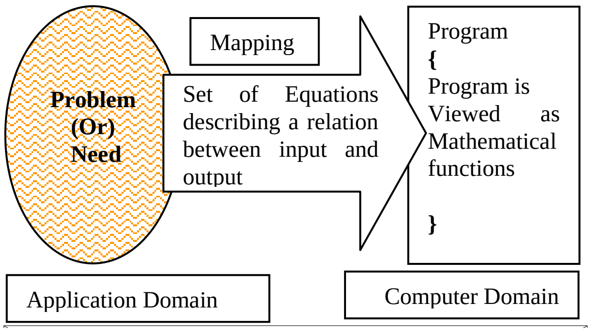
Problem
(Or)
Need
Mapping
Set of Equations describing a relation between input and output
Program
{
Program is Viewed as Mathematical functions
}
Application Domain
Computer Domain

Fig. 6. Functional Programming and its mapping

5. 2 Procedural Programming

In this methodology, "A program execution is regarded as sequence of procedure calls and manipulation of variables", shown in figure (7).

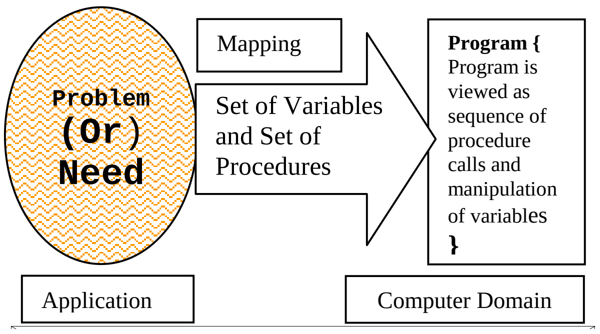
Problem (Or) Need
Mapping
Set of Variables and Set of Procedures
Program { Program is viewed as sequence of procedure calls and manipulation of variables }
Application
Computer Domain

Fig. 7 Procedural Programming transformations

5. 3 Structured Analysis & Design (SA / SD)

In the 1970s, much attention was paid to the notion of ***structured programming*.** The analysis consists of interpreting the system concept or real-world environment into data and control. Data flow diagrams are used to represent its design as shown in figure (8). This approach of designing software has limitations and not fully catering the needs of the designers. Because of the increased size and complexity of the program. One more reason is that the designing and developing programs were becoming a very large-scale activity in the software development.

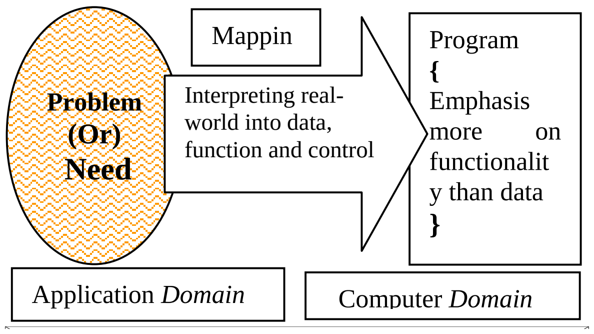
Problem (Or) Need
Mappin
Interpreting real-world into data, function and control
Program { Emphasis more on functionality than data }
Application Domain
Computer Domain

Fig. 8 Structured Analysis and Structured design

5. 4 The Jackson System Development (JSD)

In this method, the software development focuses on to construct a physical model of the real world. According to requirements, the functions can be added or changed but the physical model remains the same. As in figure (9) a program execution is regarded as a physical model, simulating the behavior of either a real or an imaginary part of the world.

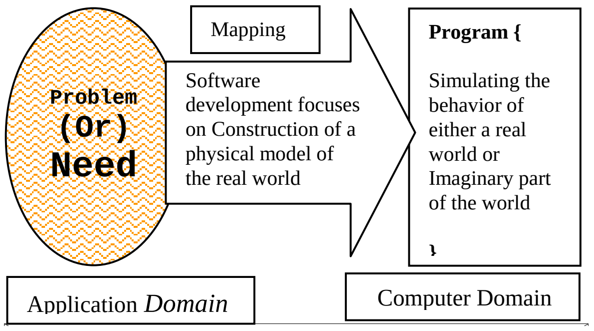
Problem (Or) Need
Mapping
Software development focuses on Construction of a physical model of the real world
Program { Simulating the behavior of either a real world or Imaginary part of the world }
Application Domain
Computer Domain

Fig. 9 The concept Jackson System Development

5. 5 Formal Methods

Formal Methods (FM) consists of a set of techniques and tools based on mathematical modeling and formal logic. Those are used to specify, verify requirements and design of a computing systems. Some developers find that it can reduce overall development life cycle cost by eliminating many costly defects prior to coding. The concept is shown in figure (10).

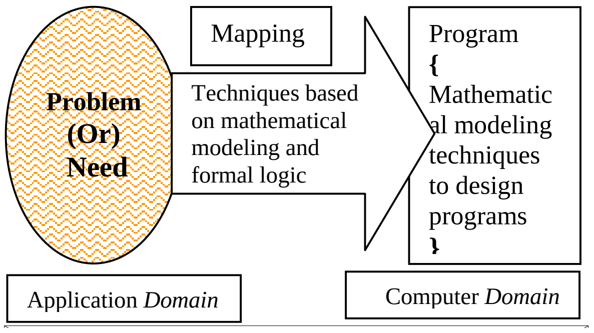
Problem (Or) Need
Mapping
Techniques based on mathematical modeling and formal logic
Program { Mathematical modeling techniques to design programs }
Application Domain
Computer Domain

Fig. 10 The principle behind the Formal Methods

5. 6 Object-Oriented Analysis and Design

Software development requires an introduction of a way of thinking that is how to solve the problem using the underlying conceptual framework. The primary advantages of OOP are real world apprehension, stability, reusability of designs and implementations. The OOP is close to the natural perception of real world [Krogdahl and Olsen, 1986]. If the programs are implemented in a natural way that they are much closer to the real world aspects, then programs are easy to write, understand and modify. This methodology has higher level of abstraction. Software activities and its types of artifacts are software architecture elements and reusable design aspects. The analysis first finds objects in the problem space, describes them with attributes, adds relationships, refines them into super, sub-types, and then defines associative objects

The Object-Oriented programming provides a natural framework for modeling the application domain. Object-Orientation is a new paradigm that is viewed by many as the best solution to most large and complex problems. Advantages of modeling are the real world into objects is thought to follow a more natural human thinking process shown in figure (11).

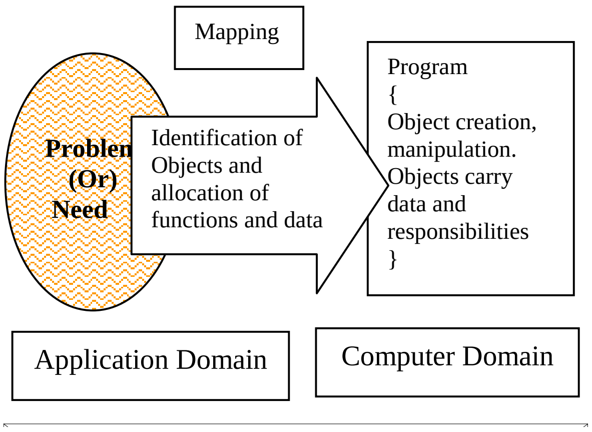
Mapping
Problem (Or) Need
Identification of Objects and allocation of functions and data
Program { Object creation, manipulation. Objects carry data and responsibilities }
Application Domain
Computer Domain

Fig. 11 Object Oriented Programming

5. 7 Aspect-Oriented Programming (AOP)

AOP is merely another patch to cover one of OOP's numerous shortcomings. First, isolate everything into a tiny package, relate and share things and make dynamic tags at runtime [7].

6. Component-based Software development

. The idea of component-based software engineering has been driven to the point of advocating construction of systems by simply assembling existing commodity components. Others develop those components for general use [8, 9, and 10]. For the same reasons there is also a growing interest in middleware solutions, either Object-Oriented ones [11, 12, 13, 14, 15, 16] or message-oriented ones [17]. The conversion process has been shown in figure (12).

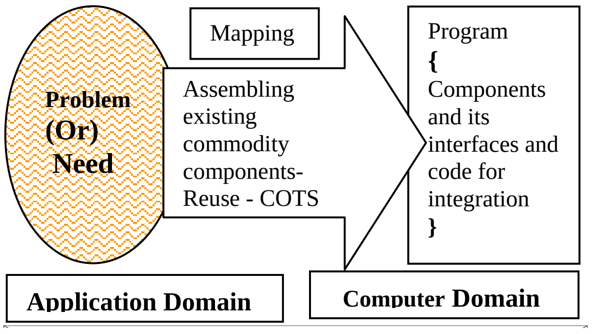
Mapping
Problem (Or) Need
Assembling existing commodity components- Reuse - COTS
Program { Components and its interfaces and code for integration }
Application Domain
Computer Domain

Fig. 12 Component-Based System Development

7. Design patterns

Object-Oriented approach is one of the best software design methodologies, are suitable for designing small scale and very large-scale programs. It has the features of extending data types and reuse of code and data structures. Object-Oriented approach has greater support for reuse and reusability. Designing frameworks promote reuse. Designing frameworks addresses a specific problem domain at code level. Further, as shown in figure (13), one-step ahead ***designing reusable design aspects*** is challenging one. Some of the design problems in very large-scale programming are solved by using the design patterns as building blocks. Design patterns allow developers for reusing successful designs and architectural designs in their new design solutions [18].

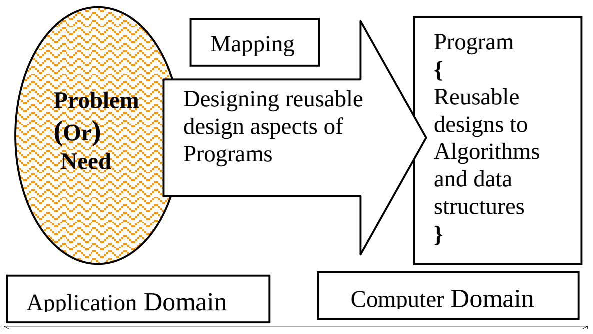
Mapping
Problem (Or) Need
Designing reusable design aspects of Programs
Program { Reusable designs to Algorithms and data structures }
Application Domain
Computer Domain

Fig. 13 Design patterns in Software Development

8. Evolution of Software Architecture

Software architecture is a composition of software structural elements, i.e., Components, connectors and Constraints and the rationale. It includes the organization of components, component interactions, the granularity of interactions and the basis for software architecture to form a system. An architectural style is characterized by type of fundamental patterns of control and data flows, functions allocated to various components, types of connectors [19], and types of constraints. The conceptual separation between what and the how applies to the software architecture. Software architecture is concerned with the what. The notion of architecture is a common description of a class of systems [6].

In 1968, Edsger Dijkstra stated, "How software is partitioned and structured as opposed to simply programming to produce a correct result" [Dijkstra, 1968]. Dijkstra introduced the idea of a layered structure. David Parnas called it as *information hiding* [20]. The principle of using an element via its interface only and some observations of the various structures to be found in software systems [Parnas, 20, 74, 76, 21]. Parnas [1976] recognized that the structure of a system influences the qualities of that system. What exactly constitutes the interface to software elements are names of the programs and parameters, they take. Architectures cannot be understood except in light of the business issues that spanned there and see the ways to analyze architectures without waiting for the system to be built. The software architecture provides a higher level of abstraction for dealing with the complexity of the systems very easily. In 1972, Parnas [36] described the use of modularization and Information hiding as a means of high-level system decomposition to improve flexibility and comprehensibility. In 1974, Stevens et al, introduced the notions of module coupling and cohesion to evaluate alternatives for program decomposition.

9. Software Architecture

Most early research on software architecture was concentrated on design methodologies. Object-Oriented Design [3] advocates a way to structure problems that leads naturally to an object-based architecture. One of the first design methodologies to emphasize design at the architectural level is the Jackson System Development [4]. There has been some initial work at investigating methodologies for the analysis and development of architectures, Kazman et al., have described design methods for eliciting the architectural trade-off analysis via ATAM[1999].

A shown in figure (14), Perry and Wolf [22] define software architecture as set of architectural elements and rationale. The rationale provides the underlying basis for the software architecture for choice of architectural style, the choice of elements and the form [6]. Rationale is an important aspect of software architecture research and of architectural description in particular. Perry, Wolf [22], Garlan, and Mary Shaw in 1993, described software architecture as system structure. Software architecture is collection of components, and connectors. Mary Shaw et al [23] defined software architecture as s system in terms of components and their interactions

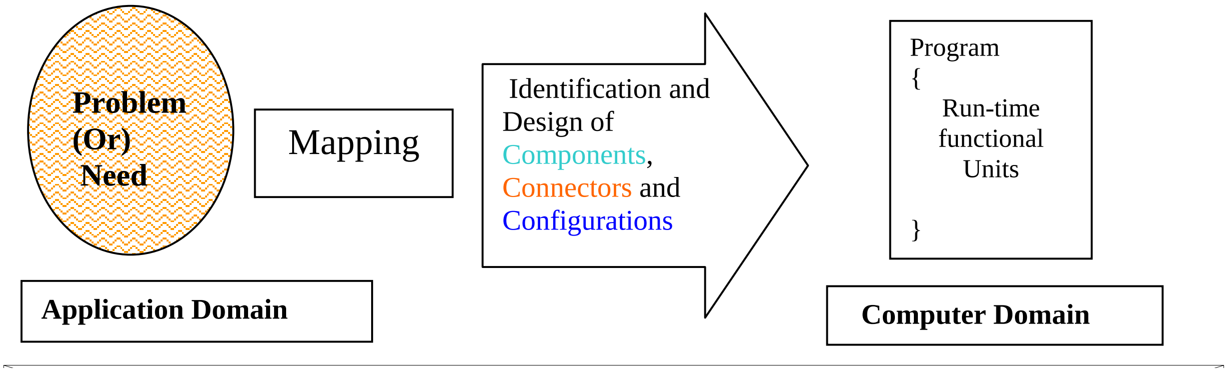
Problem
(Or)
Need
Mapping
Identification and Design of Components, Connectors and Configurations
Program
{
Run-time functional Units
}
Application Domain
Computer Domain

Fig. 14 Architecture-Based Software Development

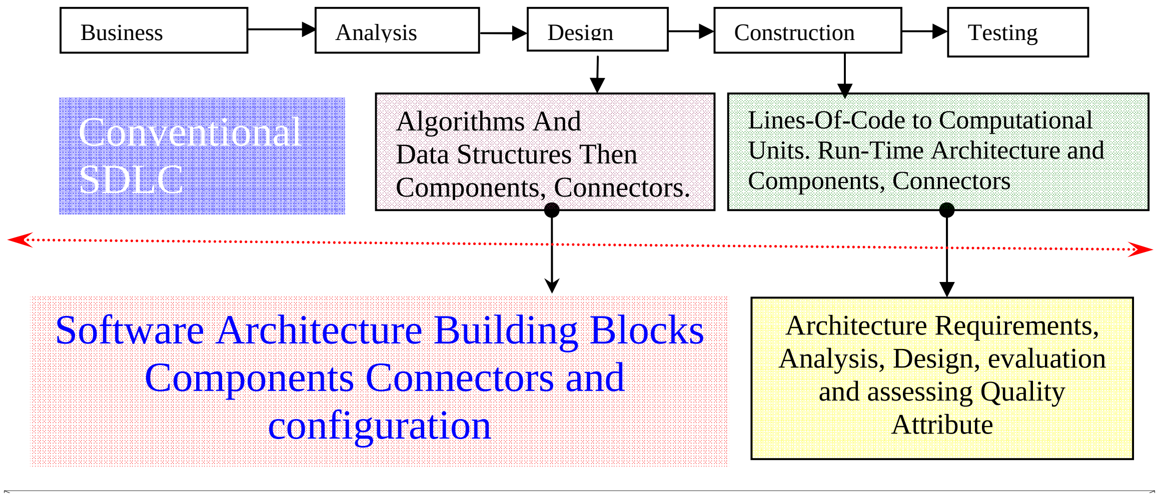
Business
Analysis
Design
Construction
Testing
Conventional SDLC
Algorithms And Data Structures Then Components, Connectors.
Lines-Of-Code to Computational Units. Run-Time Architecture and Components, Connectors
Software Architecture Building Blocks Components Connectors and configuration
Architecture Requirements, Analysis, Design, evaluation and assessing Quality Attribute

Fig. 15. Software Development Life Cycle and Architectural Systems

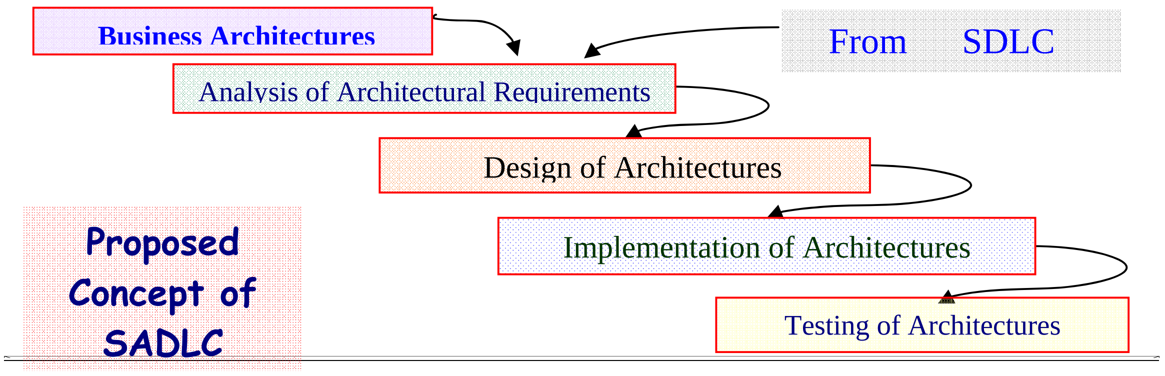
Business Architectures
From SDLC
Analysis of Architectural Requirements
Design of Architectures
Proposed Concept of SADLC
Implementation of Architectures
Testing of Architectures

Fig. 16 Software Architecture Development Life Cycle (**S A D L C**)

In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and element of the constructed system [22 and 6]. Shaw and Garlan [24] further elaborated this definition. Additional rationale for distinguishing configurations within architectural description languages is presented in Medvidovic and Taylor [25]. Perry and Wolf [22] define processing elements as "transformers of data," while Shaw et al. [1995] describe components as "the locus of computation and state." This is further clarified in Shaw and Clements [1997]. A component is a unit of software that performs some function at run-time.

In figures (15) and (16), we have not considered the formal representation of process models like Waterfall model or linear sequential model approaches, but a life-cycle view of implementing different phases of software development considered generally. We are trying to show and understand software architecture and it is Life-Cycle in the Software Development Process as Software Architecture Development Life Cycle (SADLC).

10. Architectural Description Languages (ADL)

An ADL is, according to Medvidovic and Taylor [25], a language that provides features for the explicit specification and modeling of a software system's conceptual architecture, including at a minimum; components, component interfaces, connectors, and architectural configurations. Darwin's interesting qualities are that it allows the specification of distributed architectures and dynamically composed architectures [26]. Like design methodologies, ADLs often introduce specific architectural assumptions that may affect their ability to describe some architectural styles, and may conflict with the assumptions in existing middleware [27].

11. Software Architectural Styles

An Architectural Styles increase the abstraction. A software architecture description defines the structure (high-level design) of a software system in terms of components and relationships. Styles are mechanisms for categorizing architectures and for defining their common characteristics [27].

12. Conclusion

We have discussed briefly the evolution of methodologies. First extension of this work is detailed survey on software architectures. In functional programming the activities and data are considered from problem domain. In Object-Orientation the data object are considered as main computational units. Similarly to Object-Orientation in case of software architecture we are trying to understand and study how to identify and extract the architectural elements from real world problem domain so that the real world components reflect in Analysis, Design and Code. In this direction we are proposing a Software Architecture Development Life cycle (SADLC) in order to study and understand the Architectural issues in problem domain and implementation of architectures. In each methodology every author directly or indirectly tried to enhance the abstraction for making programming job easy. Parts of the material have been considered from research article, and from the World Wide Web. I express sincerely my thanks to all of the authors of the article for their guidance in understanding properly the software architecture.

Table 1. Summary of Methodologies

METHODOLOGY/ PROGRAMMING	**ABSTRACTION LEVELS**	**DEGREE OF ABSTRACTION**
Functional Programming	The level of abstraction is low..	**3**
Procedural Programming	The level of abstraction has been increased to procedures.	**4**
Structured Analysis and Design	Interpreting the real world into data and control flow.	**6**
Jackson System Development	Simulated - abstraction	**5**
Formal methods	mathematical abstractions	**3**
Object-Orientation	The programs are closer to the real world	**7**
Component-based Software Development	Assembling from existing components.	**7**
Design patters	Designing reusable design aspects.	**4**
Software Architectures		**8**
ADLs	Description of Architectures in UML	**6**
Styles	Interfaces and connections form a style.	**6**
Views	Increase the abstraction	**7**
Quality attributes	Abstraction specification aspects	**-**

*** Assuming that an abstraction rating is given on a scale from 1 to 10 and 1 indicates the lowest. This is not formal but to understand we used the above convention. This measurement is not based on any scientific method**.**

Reference:

[1] David Garlan and Mary Shaw. An Introduction to Software Architecture, School of Computer science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, January, 1994.

[2] Composition of Software Architectures. Ph.D. Dissertation, University of Rennes I, France, Feb. 2002.

[3] G. Booch. Object-Oriented development. IEEE Transactions on Software Engineering, 12(2), Feb. 1986, pp.211-221.

[4] J. R. Cameron. An *overview* of JSD. IEEE Transactions on Software Engineering, 12(2), Feb. 1986, 222-240.

[5] M. Shaw. Comparing architectural design styles. IEEE Software, 12(6), Nov. 1995, pp. 27-41.

[6] Roy T. Fielding. Software Architectural Styles for Network-Based Applications. University of California, Irvine. Phase II Survey, 1999.

[7] Object-Oriented beta Hand book. Www reference.

[8] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE, IEEE *Software*, 15(5):37-46, September / October 1998.

[9] Paul C. Clements. From Subroutines to Subsystems: Component-Based software Development. *The American programmers, 8(11),* November 1995. Also available from http://www.sei.cmu.edu/publications/articals/cb-sw-dew.html. This journal is now called "The Journal of Information Technology Management", and can be found at http://www.cutter.com/itjournal.

[10] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Comm. ACM,* 35(9):160-165, September 1992.

[11] David Krieger and Richard M. Adler. The Emergence of Distributed component Platforms. *Computer,* 31(3):43-53, March 1998.

[12] Microsoft Corporation. COM: Technical Overview. [Online] Available at http: //www.microsoft.com/com/wpaper/[2001 March 14], March 2001.

[13] Microsoft Corporation. DCOM: Technical Overview. [Online] Available at

http://www.microsoft.com/com/wpaper/[2001, March 14], March 2001.

[14] Object Management Group. The Common Object Request Broker: Architecture and Specification – Revision 2.3.1 [Online] Available at http://cgo.omg.org/library/c2indx.html[2001, March 14], October 1999. Document formal /99-10-07.

[15] Object Management Group. CORBA Services. [Online] Available at http://www.omg.org/thchnology/documents/formal/corba_services_available_electro.htm, [2001, March 14], October 2000.

[16] Sun Microsystems. Enterprise JavaBeans Technology. [Online] Available at http:// java.sum.com /products/ejb [2001, March 14], March 2001.

[17] IBM Corporation. MQSeris Version 5. [Online] Available at http://www-4.ibm.com/software/ts/mqseries/v5/ [2000, March 4], March 2000.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. PEARSON Education, Eleventh Indian Reprint, 2003.

[19] Nenad Medvidovic, and Nikunj R. Mehta, Understanding Software Connector Compatibilities Using A Connector Taxonomy, SoDA'02 December 21-22, 2002, Bangalore, India.

[20] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), Dec. 1972, pp.1053-1058.

[21] D. L. Parnas. Designing software for ease of extension and contraction. IEEE Transactions on Software Engineering, SE-5(3), Mar. 1979.

[22] D. E. Perry and A. L Wolf. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17(4), Oct. 1992, pp.40-52.

[23] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnick. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 314-335.

[24] M. Shaw and D. Garlan. Software Architecture: Perspectives on an emerging discipline. Prentice-Hall, 1996.

[25] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description Languages. In Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Sep. 1997, pp. 60-76.

[26] J. Magee and J. Kramer, Dynamic Structure in software architectures. In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96), San Francisco, Oct. 1996, pp. 3-14.

[27] E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, May 16-22, 1999, pp. 13-22.

[28] M. Shaw. Toward higher-level abstractions for software systems. Data & Knowledge Engineering, 5, 1990, pp. 119-128.

[29] D. Garlan and M. Shaw. An intro-duction to software architecture. Ambriola & Tortola (eds.), Advances in Software Engineering & Knowledge Engineering, vol. II, World Scientific Pub Co., Singapore, 1993, pp.1-39.

[30] D. Garlan R. Allen, and J. Ockerbloom,. Exploiting an architectural design environments. In Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Enginee-ring (SIGSOFT'94), New Orleans, Dec. 1994, pp.175-188.

[31] M. Shaw and P. Clements. A field guide to box logy; Preliminary classification of architectural styles for software systems. In Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97),Washington, D. C, Aug; 1997, pp. 6-13.

[32] N. L. Kerth and W. Cunningham. Using patterns to improve out architectural vision. IEEE Software, 14(1), Jan. 1997, pp. 53-59.

[33] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison Wesley, Trading, Mass., 1998.

[34] P. B. Kruchten, The 4+1 View Model of architecture. IEEE Software, 12(6), Nov. 1995, pp. 42-50

[35] C. Alexander, S. Ishikawa, M, Silver-stein, M. Jacobson, I. Fiksdahl-King, and S. Angel. A Pattern Language. Oxford University Press, New York, 1997.

[36] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), Dec, 1972, pp. 339-344.

[37] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why it is Hard to Build Systems out of Existing Parts. In *Proceedings of the 17th International Conference on Software Engineering, pages 179-185, April 1995.* Also available from http://www.cs.cmu.edu/afs,www://cs.cmu.edu/project/ able/paper-abstracts/arh-mismatch-icse17.html

[38] http://www.csl.sri.com/neumann.html.

About Authors

- **A Rama Mohan Reddy**, Associate Professor of Computer Science and Engineering, Sri Venkateswara University College of Engineering, TIRUPATI. Completed His M.Tech computer Science from NIT Warangal and His doing his PhD in Software Architecture that is the sub field of Software Engineering, in Sri Venkateswara University, Tirupati., India

- **Dr. M M Naidu**, Professor of Computer Science and Engineering, Sri Venkateswara University college of Engineering, TIRUPATI. He completed PhD from IIT, Delhi, India.

- **Dr. P Govindarajulu**, Professor of Computer Science, Sri Venkateswara University, TIRUPATI., INDIA. He completed his PhD from IIT Mumbai., India