IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-6, Issue-9, Sept. 2018 E-ISSN: 2347-2693

Role of Software Composition in Aspect Oriented Programming

P.R. Sarode[1*], R. N. Jugele[2]

[1]Inter Institutional Computer Centre, Nagpur University Campus, RTM Nagpur University, Nagpur, India
[2]Department of Computer Science, Shri Shivaji Science College, Congress Nagar, Nagpur, India

[*]*Corresponding Author: priya.s1011@gmail.com, Tel.: +91-9503869986*

Available online at: www.ijcseonline.org

Accepted: 17/Sept/2018, Published: 30/Sept/2018

***Abstract*—** Software composition is becoming more and more vital as innovation in software engineering shifts from the development of individual components to their reuse and recombination in innovative ways. It is a key topic in computer science and particularly programming language analysis. Software composition is the process of constructing software systems from a set of software components. It aims at improving the reusability, customizability, and maintainability of large software systems. The primary motivation for software composition is reuse. Generally, the composition can be defined as any promising and expressive interaction between the complex software concept and a composition mechanism defines such an interface. The more recently proposed programming approach known as Aspect-Oriented Programming illustrate the concept of modularization i.e. managing software complexity and improving its reusability, understandability, extensibility. It provides an alternative mechanism to solve the code tangling and scattering problems in the implementation of crosscutting concerns using abstraction and composition mechanisms. This work considers different views of software composition and various existing definitions of composition units with the corresponding composition mechanisms. Also, deliberated how software composition is more efficiently reusable in aspect-oriented programming and mentioned the fundamental facts of software composition implementation based on Aspect-Oriented programming paradigm.

***Keywords*—** Software composition, composition mechanisms, object-oriented programming, aspect-oriented programming, extension,paradigm.

I. INTRODUCTION

Software engineering and programming languages exist in a shared relationship support. The most used design processes break a system down into a set of small units. To implement these units, programming languages provide mechanisms to explain the abstraction and composition mechanisms in order to implement the desired behavior [1]. A programming language coordinates well with a software design when the provided abstraction and composition mechanisms enable the developer to express the design units. In the most general terms, the composition can be defined as any possible and meaningful interaction between the software constructs involved. A composition mechanism defines such an interaction. There are many different possible kinds of software constructs, with corresponding composition mechanisms [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. Simple type definitions can be composed into compound types by type composition [14], random pieces of code can be joined together with glue and scripts [15], typed constructs can be linked by message passing, e.g. direct method calls between objects or port connections between architectural units [4], [16] and so on. In this paper, section-II remark the different views of software composition. In section-III, the existing definitions of composition units and the corresponding composition mechanisms are discussed. Section-IV of the paper presents how software composition is more efficiently reusable in Aspect-oriented programming as compared to object-oriented programming paradigm and signifies the key points of software composition implementation based on Aspect-Oriented programming (AOP) paradigm.

II. DIFFERENT VIEWS OF SOFTWARE COMPOSITION

In the inclusive as possible, considered all, the views of software composition found in the literature that is, the various perceptions (and definitions) of what composition means in all the relevant software communities. In any view of composition, the composition is performed on software entities that are observed as meaningful units of composition. Here, the focus is on units of composition that define behavior, rather than constructs that define primitive types or pure data structures. Composition mechanisms compose units of the composition into larger pieces of

software, i.e. they compose pieces of behavior into larger pieces of behavior. The outline of this section is the different views of composition and briefly discuss the generic nature of the associated units of composition and composition mechanisms.

Programming View

One view of software composition is that it is simply what a programmer does when putting bits of code together into a program or an application. In this view, any legitimate programming language construct is a unit of composition; and composition is simply joining the constructs together using some other construct (e.g. sequencing) defined in the programming language. This is called the 'programming view' of the composition. Meaningful units of composition in the programming view include functions in functional languages, procedures in imperative languages, classes [7], aspects [6] in object-oriented and aspect-oriented languages. Clearly, the 'programming view' represents programming in the small. To equate composition with this view, however, is to overlook many issues that are significant for software engineering, such as reuse and systematic or automated construction.

Construction View

A higher-level view of the composition is the view that software composition is "the process of constructing applications by interconnecting software components through their plugs" [18]. The primary motivation here is systematic construction. This view is called the 'construction view' of the composition. It is at a higher level of abstraction than the 'programming view': it typically uses scripting languages [19] to connect pre-existing program units together. The 'construction view' thus represents programming-in-the-large [20], as opposed to programming-in-the-small. In the 'construction view', the units of the composition are referred to like components, but these are only loosely defined as software units with plugs, which are interaction or connection points. Consequently, components may be any software units that can be scripted together by glue. For example, components may be modules glued by module interaction languages [21], or Java Beans composed by [22], and so on.

CBD View

For Component-based Software Development (CBD) [23], [24] composition is of the essence, since components, by definition, are units of composition [45], [46]. For CBD, software reuse is, of course, a fundamental objective, in order to reduce production cost; however, in addition, CBD also seeks to automate composition as much as possible, to reduce time-to-market as well. To characterize components accurately, [17] characterize them about a component model [23], [24]. A component model defines what components are (their syntax and semantics) and what composition operators can be used to compose them. Thus in [23], a software component is defined as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard". The advent of CBD [23], [24], [8] brought about a sharper focus on not only component models (different kinds of components and composition mechanisms), but also repositories of (pre-existing) components and component reuse from such repositories. Thus, CBD is motivated by systematic construction as well as reuse of (pre-existing) third-party components. This view extends the 'construction view', by the additional emphasis on component models as well as reuse of third-party components. Software architectures also subscribe to the 'CBD view', in addition to the 'construction view', in the sense that an Architecture Description Language (ADL) [26], [27] could be considered to be a component model, with architectural units as components, and port connection as a composition mechanism for such components. However, in contrast to the 'CBD view', software architectures do not always assume or make use of third-party components or repositories of such components, as mentioned earlier. As components are independent identities, every component has its own required and provided services. When software systems are designed by assembling the independent components the role of composition becomes very crucial in delivering the required system [49].

III. EXISTING DEFINITIONS OF SOFTWARE COMPOSITION

In [17] the survey of composition mechanisms that are defined in all three views, since it does not make much sense to consider composition mechanisms that are only unary in arity, our normal assumption is that composition mechanisms are (at least) binary in arity. Composition mechanisms in all three views fall into four general categories:

i) Containment
ii) Extension
iii) Connection
iv) Coordination

In the following section definitions and explanation of each category presents, using generic units of composition for clarification and design [17] compare and contrast the category with corresponding UML mechanisms.

Containment

Refers to putting units of behavior inside the definition of a larger unit. This is illustrated in Fig. 1(a), where U3 contains U1 and U2. Containment is thus nested definition. The behavior of the container unit is defined in terms of that of the contained units, but the precise nature

of the containment differs from mechanism to mechanism. Examples of containment are nested definitions of functions, procedures, modules, and classes, as well as object composition and object aggregation. Compared to (standard) UML, our notion of containment covers more composition mechanisms.

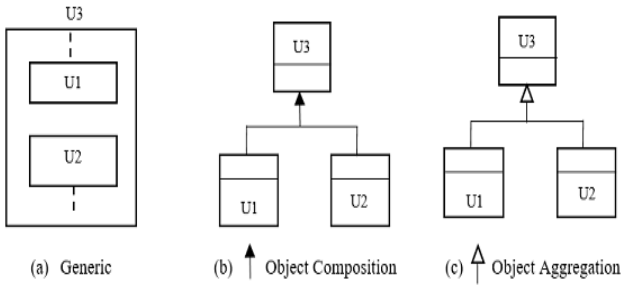

Figure. 1. Containment

In UML, containment is defined for classes only, there is no notation for nested class definition, and the only forms of containment are object aggregation and object composition Fig. 1(b).

Extension

Refers to defining the behavior of a unit by extending that of at least two other units of composition. This is extension U1 U2 U3 class extension illustrated in Fig. 2(a).

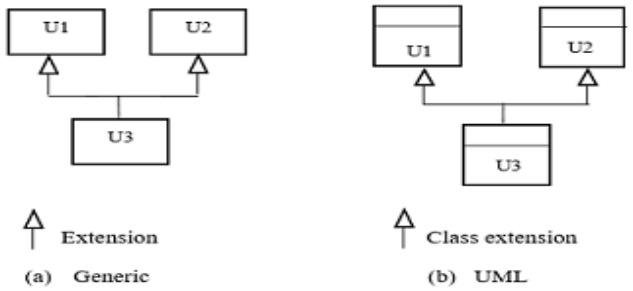

Figure.2. Extension

Examples of extension include multiple inheritance in object-oriented programming, aspect weaving [38] in aspect-oriented programming. Multiple inheritance can be defined as a composition mechanism that extends multiple classes (e.g. U1 and U2 in Fig. 2(a)) into another class (U3) that inherits from these classes. Aspect weaving can be defined as a (binary) composition mechanism that extends a class (say U1 in Fig. 2(a)) and an aspect (U2) into another class (U3) that is the result of weaving U2 into U1. (Certainly, U3 is just the new version of U1.) Other extension mechanisms, namely aspect weaving, can only be represented in UML as multiple inheritance if it is acceptable to represent an aspect as a class. However, if aspects are to be distinguished from classes, which are intended to be, in aspect-oriented then it cannot define aspect weaving as composition mechanisms in UML.

Connection

Refers to defining a behavior that is an interaction between the behavior of multiple units. This is illustrated in Fig. 3. The units either directly or indirectly invoking each other's behavior affect this interaction. Connection is thus message passing, and as such, it induces tight coupling between units that send messages to each other.

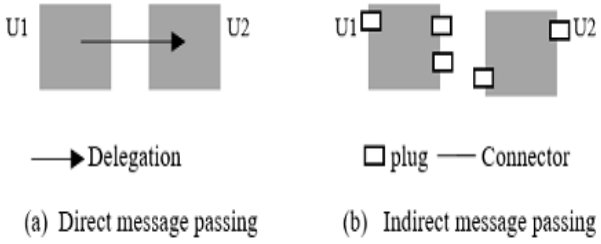

Figure. 3. Connection

Coordination

Refers to defining a behavior that results from coordinating the behavior of multiple units. This is illustrated in Fig. 4. The coordination is performed by a communication channel coordinator, which communicates with the units via a control and/or a data channel. The units themselves do not communicate directly with one another. Coordination thus removes all coupling between the units, in contrast to connection, which induces tight coupling through message passing. Examples of coordination are data coordination using tuple spaces [29], data coordination using data connectors [30] for parallel processes or active components, control coordination using orchestration [31] for (web) services, and control coordination using exogenous composition for encapsulated components.

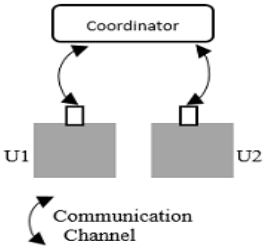

Figure.4. Coordination

Tuple spaces are used in coordination languages to coordinate parallel processes, by storing and sharing typed data objects (tuples) between the processes. In contrast to connection mechanisms, these processes communicate only with the tuple space, but not directly or indirectly with each other.

IV. SOFTWARE COMPOSITION IMPLEMENTATION BASED ON AOP

Software engineering is a field of engineering that came into existence owing to the various problems that

developers of software faced while developing software project [48]. Software engineering and programming languages exist in a mutual relationship support. The most used design processes break a system down into a set of small units. To implement these units, programming languages provide mechanisms to define abstractions and composition mechanisms in order to implement the desired behavior [32]. A programming language coordinates well with a software design when the provided abstraction and composition mechanisms enable the developer to express the design units. The most used abstraction mechanisms of languages (such as procedures, functions, objects, classes) are derived from the system functional decomposition and can be grouped into a generalized procedure model [33]. However, many properties do not fit well into generalized procedures such as exception handling, real-time constraints, distribution, and concurrency control. They are usually spread over into several system modules, affecting performance and/or semantics systematically. When these properties are implemented using an object-oriented or a procedural language, their code is tangled with the basic system functionality. It is hard to separate one concern from another, see or analyze them as single units of abstraction. This code tangling is responsible by part of the complexity found in computer systems today. It increases the dependencies among the functional modules, deviating them from their original purposes, making them less reusable and error-prone. This separation of concerns is a fundamental issue in software engineering and it is used in analysis, design, and implementation of computer systems. However, the most used programming techniques do not always present themselves in a satisfactory way regarding this separation. Aspect-oriented programming allows separation of these crosscutting concerns, in a natural and clean way, using abstraction and composition mechanisms to produce executable code. The aspect-oriented programming main goal is to help the developer in the task of clearly separate crosscutting concerns, using mechanisms to abstract and compose them to produce the desired system. The aspect-oriented programming extends other programming techniques (object-oriented, structured, functional etc.) that do not offer suitable abstractions to deal with crosscutting [33]. AOP allows programmers to have the advantage of modularization for cross cutting concerns that are present in almost every part of software. In OOPs like C++ or Java, class is considered as modular unit. Similarly in AOP aspects provide the same functionality to the cross cutting concerns which provides functionality to more than one class [47]. An implementation based on the aspect-oriented programming paradigm is usually composed of:

a) A component language to program components (i.e. classes);
b) One or more aspect languages to program aspects;
c) An aspect weaver to compose the programs written in these languages;
d) Programs are written in the component language;
e) One or more programs written in the aspect language.

Components

Components (in AOP) are abstractions provided by a language to implement systems basic functionality. Procedures, function, classes, and objects are components in aspect-oriented programming. They are originated from functional decomposition. The language used to express components could be an object-oriented, an imperative or a functional one [34].

a) Aspects: Properties affecting several classes could not be well-expressed using current notations and languages. Aspects are expressed through code fragments that spread over the system classes [35]. Some concerns that are frequently aspects: concurrent objects synchronization [36], distribution [37], exception handling [38], coordination of multiple objects [39], persistence, serialization, replication, security, visualization, logging, tracing, load balance and fault tolerance amongst others.

b) Component Language: The component language should provide developers with mechanisms to write programs implementing the basic requirements and do not predict what is implemented in the aspects, this property is called obliviousness [40]. Aspect-oriented programming is not limited to object orientation, although, the most used component languages are object-oriented ones, such as Java, Smalltalk or C#.

c) Aspect language: The aspect language defines mechanisms to implement crosscutting in a clear way, providing constructions describe the aspect semantics and behavior [33]. Some guidelines observed in the specification of an AO language syntax, which must relate to the component language syntax, the language should be designed to specify the aspect in a concise and compact way and the grammar should have elements to allow composition of classes and aspects [41].

d) Aspect Weaver: The aspect weaver main responsibility is to process aspect and component languages, in order to produce the desired operation. To do that, it is essential the join-point concept. A join-point is a well-defined point in the execution or structure of a program. For instance, in object-oriented programs join-points could be method-calling, constructor calling, field read/write operations etc. The representation of those points could be generated in runtime using a reflective environment. In this case, the aspect language is implemented through meta-objects, activated at method invocations, using join-

points and aspects information to weave the arguments [42]. An aspect-oriented system design requires knowledge about what should be in classes and in aspects, as well as characteristics shared in both. Although aspect-oriented and object-oriented languages have different abstraction and composition mechanisms, they should use some common terms, allowing the weaver to compose the different programs. The weaver parses aspect programs and collects information about the (join) points referenced by the program. Afterward, it locates coordination points between the languages, weaving the code to implement what is specified in them [43]. An example of a weaver implementation is a pre-processor that traverse the classes parsing tree, looking for joint-points and inserting sentences declared in the aspects. This weaving process could be static i.e. compile time or dynamic i.e. load and runtime.

V. CONCLUSION

The main conclusion of the study and the aim of this paper is to contribute to the understanding of software composition, and eventually the existence of software composition in aspect-oriented programming in a more general context. To this extent, we have proposed and illustrated a systematic approach to analyzing software composition in a detailed and concrete manner. We have mentioned the key points of software composition implementation based on aspect-oriented programming.

VI. ABBREVIATIONS AND ACRONYMS

Following abbreviations and acronyms used in the given paper listed below.

[1] AOP- Aspect Oriented Programming
[2] CBD- Component-based Software Development
[3] ADL- Architecture Description Language
[4] UML-Unified Modeling Language

ACKNOWLEDGMENT

We thank the Babasaheb Ambedkar Research and Training Institute (BARTI), Pune for funding our research.

REFERENCES

[1] Christian Becker and Kurt Geihs. Quality of service - aspects of distributed programs. In Int'l Workshop on Aspect-Oriented Programming (ICSE 1998), April 1998.
[2] G. Bracha and W. Cook. Mixin-based inheritance. In Proc. OOPSLA/ ECOOP 90, pages 303–311. ACM Press, 1990.
[3] O. Nierstrasz and D. Tsichritzis, editors. Object-Oriented Software Composition. Prentice-Hall International, 1995.
[4] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
[5] J. Sametinger. Software Engineering with Reusable Components. Springer-Verlag, 1997.
[6] G. Kiczales et al. Aspect-oriented programming. In Proc. ECOOP 97, pages 220–242. Springer-Verlag, 1997.
[7] C. Szyperski. Universe of composition. Software Development, August 2002.
[8] C. Szyperski, D. Gruntz, and S. Murer. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, second edition, 2002.
[9] G. Alonso et al. Web Services: Concepts, Architectures and Applications. Springer-Verlag, 2004.
[10] S. Ducasse et al. Traits: A mechanism for fine-grained reuse. ACM Trans. Prog. Lang. Syst., 28(2):331–388, 2006.
[11] U. Assman. Invasive Software Composition. Springer Verlag, 2003.
[12] C. Prehofer. Feature-oriented programming: A fresh look at objects. In Proc. ECOOP'97, pages 419–443. Springer-Verlag, 2002.
[13] H. Ossher et al. Specifying subject-oriented composition. Theory. Pract. Object Syst., 2(3):179–202, 1996.
[14] M. Buchi and W. Weck. Compound types for Java. In Proc. OOPSLA 98, pages 362–373. ACM Press, 1998.
[15] J.-G. Schneider and O. Nierstrasz. Components, scripts and glue. In Software Architectures – Advances and Applications, pages 13–25. Springer-Verlag, 1999.
[16] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison-Wesley, 2nd edition, 2003.
[17] Kung-Kiu Lau and Tauseef Rana School of Computer Science, The University of Manchester, Manchester M13 9PL, UK, A Taxonomy of Software Composition Mechanisms.
[18] O. Nierstrasz and L. Dami. Component-oriented software technology. In [35], pages 3–28. Prentice-Hall, 1995.
[19] J.K. Ousterhout. Scripting: Higher-level programming for the 21st century. Computer, 31(3):23–30, 1998.
[20] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. IEEE Trans. Soft. Eng., 2(2):80–86, 1976.
[21] R. Prieto-Diaz and J.M. Neighbors. Module interconnection languages. Journal of System and Software, 6(4):307–334, 1987.
[22] F. Achermann et al. Piccola – a small composition language. In Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches, pages 403–426. Cambridge University Press, 2001
[23] G.T. Heineman and W.T. Councill, editors. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.
[24] K.-K. Lau and Z. Wang. Software component models. IEEE Trans. On Soft. Eng., 33(10):709–724, 2007.
[25] M. Broy et al. What characterizes a software component? Software – Concepts and Tools, 19(1):49–56, 1998.
[26] P.C. Clements. A survey of architecture description languages. In 8th Int. Workshop on Soft. Spec.and Design, pages 16–25. ACM, 1996.
[27] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. IEEE Trans on Soft. Eng., 26(1):70–93, 2000.
[28] G. Kiczales et al. An overview of AspectJ. In Proc. ECOOP 01, pages 327–353. Springer-Verlag, 2001.
[29] N. Carriero and D. Gelernter. Linda in context. Comm. ACM, 32(4):444–458, 1989.
[30] F. Arbab. Reo: a channel-based coordination model for component composition. Math. Struct. in Comp. Sci., 14(3):329–366, 2004.
[31] T. Erl. Service-Oriented Architecture: Concepts, Technology, and Design.Prentice Hall, 2005.

[32] Christian Becker and Kurt Geihs. Quality of service - aspects of distributed programs. In Int'l Workshop on Aspect-Oriented Programming (ICSE 1998), April 1998.
[33] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, 11th European Conf. Object-Oriented Programming, volume 1241 of LNCS, pages 220–242. Springer Verlag, 1997.
[34] B. Tekinerdogan and M. Aksit. Deriving design aspects from canonical models. In Workshop on Aspect-Oriented Programming (ECOOP 1998) [1].
[35] Krzystof Czarnecki and Ulrich W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Boston, 2000.
[36] John Dempsey and Vinny Cahill. Aspects of system support for distributed computing. In Workshop on Aspect-Oriented Programming (ECOOP 1997), June 1997.
[37] Cristina Videira Lopes. D: A Language Framework for Distributed Programming. PhD thesis, College of Computer Science, Northeastern University, 1997.
[38] H. Ossher and P. Tarr. Operation-level composition: A case in (join) point. In Workshop on Aspect-Oriented Programming (ECOOP 1998) [1].
[39] William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications, pages 411–428, Sep 1993.
[40] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Workshop on Advanced Separation of Concerns (OOPSLA 2000), October2000.
[41] Kai B¨ollert. On weaving aspects. In Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999), June 1999.
[42] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case study for aspect-oriented programming. Technical Report SPL-97-009, Palo Alto Research Center, 1997.
[43] K. B¨ollert. Aspect-oriented programming case study: System management application. In Workshop on Aspect-Oriented Programming (ECOOP 1998) [1].
[44] C. Szyperski. Back to universe. Software Development, September 2002.
[45] C. Szyperski. Universe of composition. Software Development, August 2002.
[46] C. Szyperski, D. Gruntz, and S. Murer. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, second edition, 2002.
[47] Jatin Arora, Jagandeep Singh Sidhu and Pavneet Kaur, "Applying Dependency Injection Through AOP Programming to Analyze the Performance of OS", International Journal of Computer Sciences and Engineering, Vol.3, Issue.2, pp.45-50, 2015.
[48] Biswajit Saha, Debaprasad Mukherjee, "Analysis of Applications of Object Orientation to Software Engineering, Data Warehousing and Teaching Methodologies", International Journal of Computer Sciences and Engineering, Vol.5, Issue.9, pp.244-248, 2017.
[49] Maushumi Lahon and Uzzal Sharma, "The Intricacies of Software Component Composition", International Journal of Computer Sciences and Engineering, Vol.03, Issue.01, pp.111-117, 2015.

Authors Profile

Priyanka Sarode awarded B.Sc. degree in 2006 and MCA degree in 2009 from Rashtrasant Tukdoji Maharaj Nagpur University, Nagpur. Currently she is pursuing Ph.D. in Computer Science and a research fellow at Inter Institutional Computer Center, Rashtrasant Tukdoji Maharaj Nagpur University, and Nagpur. Her research work is acknowledged by BARTI, Pune. Her main research work focuses on Programming languages, Aspect Oriented Programming. Mail Id: priya.s1011@gmail.com Mobile No. 950386998

Ravikant Jugele is a M.Sc. in Computer Science from Marthwada University, Aurangabad in 1993. He also completed Ph. D. in Computer Science from Rashtrasant Tukdoji Maharaj Nagpur University. He is currently working as an Associate Professor in Department of Computer Science, Shivaji Science College, Congress nagar, Nagpur. Since 1995, his research interests include Multimedia and Hypermedia, cloud computing, Programming languages, Artificial Intelligence, Deep Technology and so on. Mail Id: rn_jugele@yahoo.com.

