International Journal of Computer and Organization Trends
ISSN: 2249 – 2593 / https://doi.org/10.14445/22492593/IJCOT-V12I1P302
Volume 12 Issue 1, 3-10, Jan-Apr 2022

Research Article

On Software Modular Architecture: Concepts, Metrics and Trends

Mbugua Samuel Thaiya[1], Korongo Julia[2], Samuel Mbugua[3]

Department of Information Technology, Kibabii University, Kenya.

Received Date: 10 February 2022
Revised Date: 22 March 2022
Accepted Date: 26 March 2022

Abstract *- The emergence of digital computers at the tail end of the last century allowed for the evolution of computer languages from low-level languages of the 1940s to the object-oriented, scripting high-level languages of today. This evolution has, in effect, seen the size and complexity of computer programs increase by a large factor. The software industry has, in response, developed different styles for designing and developing these sophisticated computer programs. While the different styles have advantages and disadvantages and different application domains, modular architecture has stood out as an overarching architecture for designing complicated and enormous software systems of today's world. In this paper, we examine how modularity applies to software architecture design, the concepts of modularity, the metrics of modularity, and current trends in software modularization. We advance the position that modularity will keep influencing software design for the foreseeable future due to the flexibility and the several advantages to the discipline of software design.*

Keywords *- Software, Software Architecture, Modular Architecture, Modularity Metrics.*

I. INTRODUCTION

Today we live in a highly computerized world. Computers and related technologies control most aspects of today's lives. One of the major components of these computers is software which refers to the instructions that tell a computer what to do. The software comprises the entire set of programs, procedures, and routines associated with the operation of a computer system. The term is used to differentiate these instructions from the physical components of a computer system – the hardware [1].

Since the emergence of digital computers in the 1950s, writing software has evolved from using machine language through low-level assembly languages to today's high-level languages. While high-level languages allow for the writing of sophisticated computer programs, they also complicate the design of these programs. As the size and complexity of software systems increased, the design problem went beyond the algorithms and data structures of the computation: designing and specifying the overall system structure has emerged as a new kind of problem. Structural issues include overall organization and control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives [2] – this is the architecture. Software Architecture can thus be understood to mean the high-level structure of a software system. As such, software architecture can be viewed as consisting of three main components; the structure of the system, the process of creating such a structure and the documentation of the structure.

Some of the common styles to represent software architecture are Pipe and Filters, Layered, Repositories, Service-Oriented Architecture (SOA), Distributed, and Modular. To represent a complex interplay of components, there is a need to adopt a definite style for the process. In this paper, we focus on the modular architecture of software design. To understand the modular design, we first look at how layered and SOA – two of the most popular styles - define software architecture structures.

II. LAYERED ARCHITECTURE

A popular software architectural style, layered architecture focuses on the grouping of related functionality within a software application into distinct layers stacked on top of each other. Each layer provides functionality grouped by a common responsibility or role, with explicit and loosely coupled interactions between the layers [3]. This style, therefore, helps to support a strong separation of responsibilities that, in turn, supports the flexibility and maintainability of a software system. Similarly, Rengaiah notes that a layered architecture style distributes the roles and responsibilities around a broader technical function and depicts an inverted pyramid with the preceding layer accessing more focused lower-level layers [4]. In this manner, a layered style highlights the physical and often the logical layout of a software application. (Fig. 1)

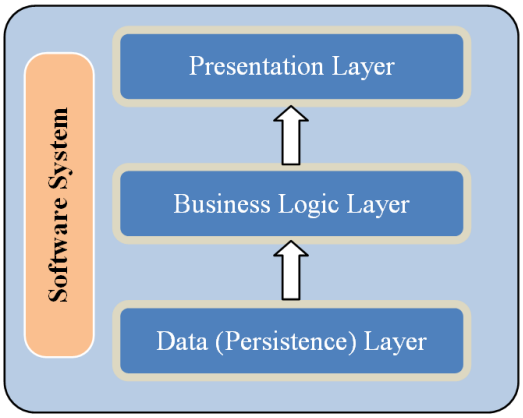

Fig. 1 Layered Architecture

This layered layout is evident, for example, in IBM's accelerator software system architecture, as shown in fig. 2

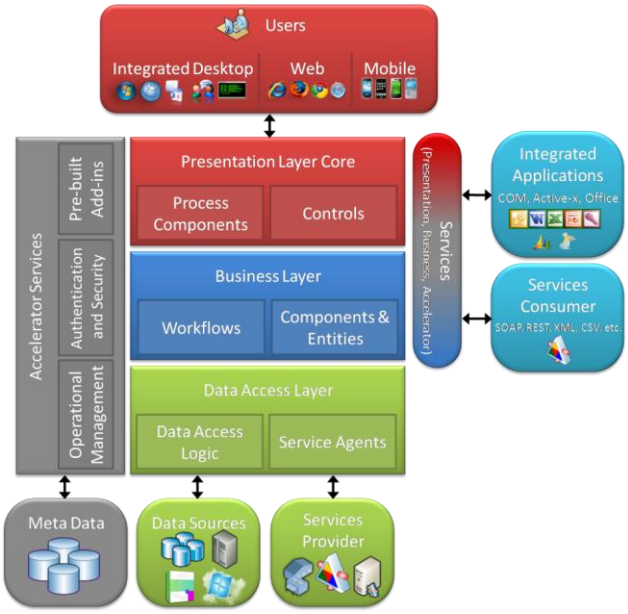

Fig. 2 IBM's Accelerator Software Layered Architecture

Source: [5]

II. SERVICE-ORIENTED ARCHITECTURE

In today's complex and technologically connected world, a key component in designing a software artefact is having it communicate with other artefacts. Inter-artefact communication allows an organization to quickly realign and adapt its business processes in response to both internal and external. The Service-oriented architecture (SOA) style is concerned with how different system business functions work with each other. These business functions are defined as a set of services.

At its core, SOA implies that you have a set of services that can perform some business function, and your clients can consume these services to get their work done [6]. Mehta and Shah observe that SOA codifies how we can publish, utilize and identify services across various technical and functional boundaries.

An SOA architecture style is thus largely concerned with the communication aspects of a software application.

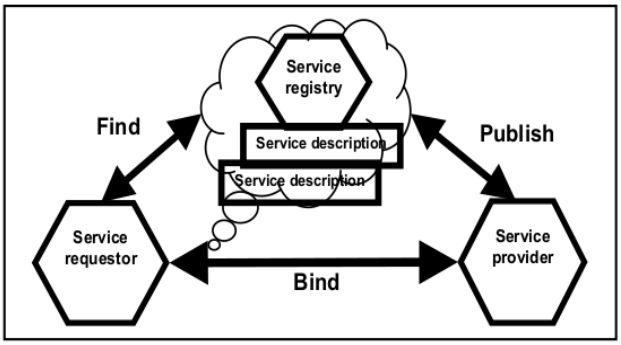

Fig. 3 Service-Oriented Architecture

Source: [6].

While layered architecture and SOA are formidable styles for software systems, they are faced with certain deficiencies, especially in the evolving world of software. A layered system is largely conceived as three tiers - data access layer, business, presentation – that work together to clarify the relationship between the different elements of a software system. However, in many modern software projects, layers have become very large themselves, as they contain several components and those depend on each other. Sometimes this dependency matrix is so complex that it naturally calls for splitting layers into more granular sub-layers [7]. This then raises the question of how many layers a software can have and how to manage the complex inter-relationships between the various components. An SOA style has similar inherent deficiencies in how much functionality the will service will be responsible for.

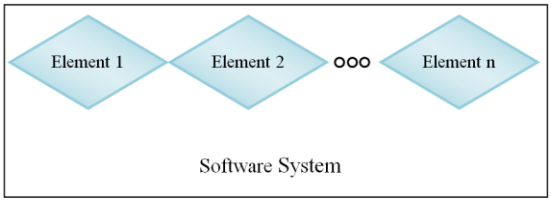

Fig. 4 SOA elements

For a layered or SOA style to work, specific components need to know and understand the other components in the layer or service. They will further have connected dependencies that may break an entire software application should one component fail. Further, Narduzzo and Rossi pose that with the advent of free and open-source software (FOSS) projects being developed by several developers located worldwide, how can we have all these developers work on the same layer or service [8]. Layered and SOA structure a monolithic software system where the "only" application offers all use-case and services. These architectural approaches are not only non-flexible but curtail skills. Newer software architects, designers and developers with different ways of doing things cannot implement their skills without decomposing the entire system. The approach limits the extent to which a software system can adapt and employ new technologies and tools.

III. MODULAR ARCHITECTURE

A modular architecture style helps us view a system not just as a hierarchy of layers or in terms of services rendered but as a level of depth as a composition of smaller "modules" [4]. Kirkk defines a module as a "deployable, manageable, natively reusable, composable, stateless unit of software that provides a concise interface to consumers" [9].

The modular architecture thus decomposes a software program into smaller programs (modules) with standardized interfaces to allow for communication between the modules and the core system.

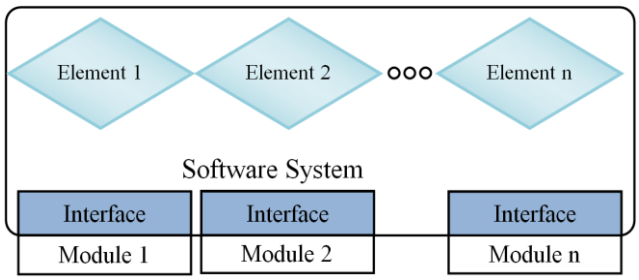

Fig. 5 Modular Architecture

The modular system architecture groups related functional requirements into a module designed as a separate structure from the core application but consumes and expose communication interfaces [8]. The communications between the modules may be implemented as I/O stream, I/O buffers, piped or other types of connections. Each module of the system should have one specific responsibility, which helps the user understand the system clearly. It should also help integrate the module with other components [10].

Rengaiah notes that a module has a clear business context, is confined to the enclosing physical layer, works within the context provided and expresses its scope through a public interface [4]. Consequently, a module helps us understand, extend, and manage the system during the design and during run time, that is, design time modularity and run-time modularity.

For instance, a modular system architecture for Unmanned Aerial Vehicles (UAV) and Unmanned Ground Vehicles (UGV), as advanced by Giakoumidis, structures distinct modules for various UAV/UGV functionality as well as path planning modules [11]. The modularity of this architecture makes a rather complicated system feasible for both development and deployment. Further, it allows for the implementation of UAVs functionalities without the need for UGVs and vice versa.

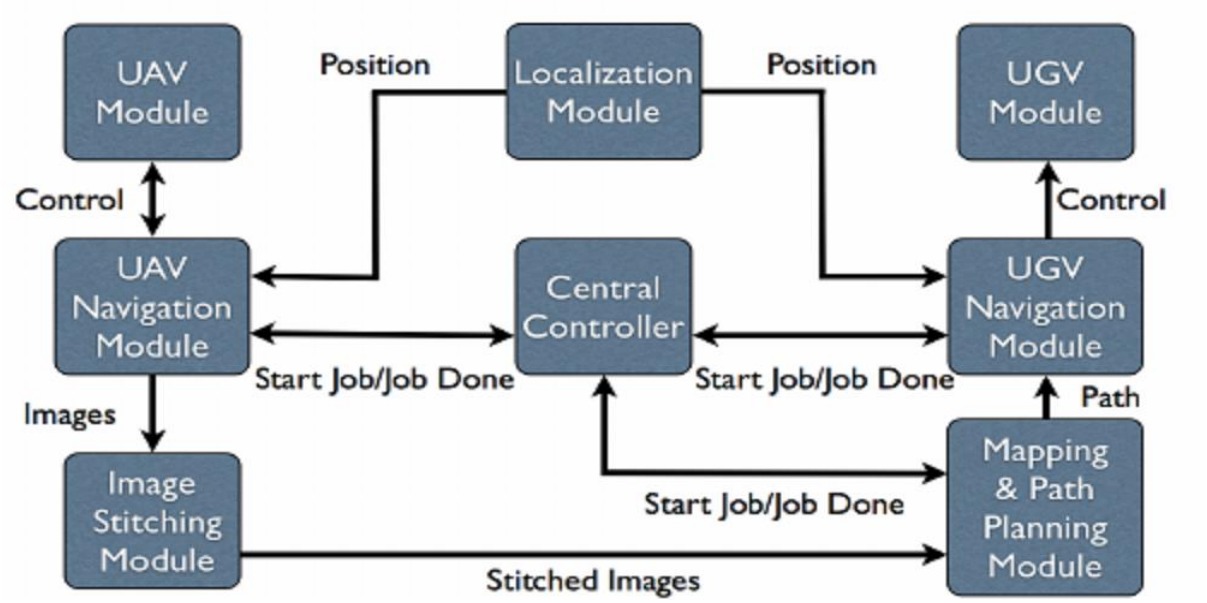

Fig. 6 Modular system architecture

Source [11]

IV. CONCEPTS OF MODULAR ARCHITECTURE

A modular architecture scholarly is perceived as a manufacturing paradigm for designing and developing complex artefacts [12]. It is, therefore, a critical element in defining the design and development of complex software projects as it provides a comprehensive definition of the software project. As such, to achieve modularity, an architect will need to consider certain concepts:

A. Module Interaction with the Application

Every module (artefact) will need to exchange data and resources with the core system and other modules. The designer thus needs to provide interfaces/entry points for this communication. The interfaces will also define the control of how/what/who/when the exchange happens [13].

B. Module Registration

In a modular design, there is a need to create a mechanism for the system to be aware of the existence of a specific module. There are two approaches to this*:*

a) Discovery

In this approach, the main application scans for the existence of a module and, once discovered, registers the module. The main application then maintains a registry of all modules discovered and their statuses. Many modern software development frameworks have extensively applied the concept of discovery. The PHP composer utility, as used in Symphony, Laravel and several other PHP frameworks, will automatically discover modules and register them in the framework kernel [14]

b) Configuration

The designer creates specific configuration settings to allow for the module's registration. This configuration setting unwraps the module's default behaviour while allowing it to learn about other modules and the existing interfaces/entry points.

C. Module Structure (partitioning)

A module needs to have a structure to interface with the application and other modules. This structure should define an optimal and practical assignment logic [15].

D. Events

Like the Event-Driven Architecture, a module may need a structure to utilize events. It should not only be able to "*listen*" and "react" to events but also "raise" its events which will trigger reactions in other parts of the software system [13].

E. Configuration

As the module is a small customizable sub-system, there is a need to provide configuration of the module to meet user needs.

V. ADVANTAGES OF MODULAR DESIGN

The modular architectural design has been advanced to remedy the deficiencies of the popular layered and SOA architectures. By introducing granularity and separation of services, these design styles have supported the design and development of otherwise very complex software systems. Further, the modular system architecture is advantageous as it provides for:

A. Customizations

A generic standard defines systems that can change in functionalities and services offered by utilizing the modular design. By enabling or disabling some modules, an implementation can completely change how a system works and services rendered. Kibabii University in western Kenya has amplified the need for modular customizations in its Enterprise Resource Planning (ERP) system, where various modules of the ERP are customized to user desires without affecting other modules of the system [15]. Such flexibility cannot be achieved when employing a layered or SOA approach.

B. Less Inter-Dependency

Each module in the system is more independent from the core software system itself. As long as the interfaces are compatible, both modules and the core software system can evolve independently.

C. Third-Party Extensions

Rengaiah notes that modules are not part of the core system and only communicate with the core system and with each other through well-defined entry points. As such, modules can be developed by third parties [4]. The ERP system at Kibabii University encompasses modules developed by parties different from the main vendor. For instance, the Learning Management module is developed by the Moodle Open Source Community [16, 17].

D. Independent Development

Since the core system and the modules are independent in modular design, they can therefore be developed by external developers. This feature has benefited many free and open-source software (FOSA) projects [18]. Further, each module's core systems can be released with independent release cycles and developed potentially with different technologies and tools. The modular OpenMRS medical record system has different modules developed using various technologies. While the core system is developed in Java and utilizes the Spring framework, the module repository has modules developed using AngularJS, ReactJS and Vue frameworks, among other frameworks, clearly indicating how modularity allows for independent development [18]. Similarly, Narduzzo and Rossi, in their study on the design of complex software artefacts, have attributed the achievements of various Free/Open Source Software (FOSS) projects (among them: the GNU operating system, the Linux kernel, the HURD kernel) to the modular approach adopted by these FOSS projects [8].

E. Smaller Core Application

The size of the main software system is significantly reduced since much functionality can be implemented via modules. This translates to a better understanding of the system and better maintainability.

F. Reusability

A well-conceived module is fully reusable. You can reuse the old solution whenever you have the same need again.

G. Refactorability

The fewer inter-dependencies in a project, the easier it is to make large changes across multiple modules [19].

H. Scalability

Modular design allows software applications to scale as it is almost impossible to build large applications without good modularization. Brinkman and Delamore note that the complexity of a monolithic system built without modules will destroy productivity [19].

VI. CHALLENGES OF MODULAR DESIGN

Whereas the modular architecture provides a convenient architecture for software projects and places software designers and developers at the centre of software evolution, thus underlining that business agility can be enabled through critical design, it is also laced with some challenges:

A. Architectural Mismatch

One issue is with systems that integrate orthogonal functionality into a single modular artefact which introduces artificial coupling of functionalities driven by a specific implementation requirement. While such coupling may have some locally optimal performance, this often may come at the expense of the global optimality of the software system [20].

B. Physical Variability

Physical variability refers to how different variable modules within the same software system are. For the software application to support all these variable modules, it will need to provide a generic interface/entry point. When designing a generalized interface, it is often the case that neither the union of all possible capabilities nor the intersection of such capabilities is satisfactory. The generic product interface thus supports capabilities that often lie in between [20].

C. Inaccuracy in Modularity Analysis

When choosing certain architecture abstractions, styles and mechanisms for decomposing a system, architects may leave some functionalities/services non-modularized. These functionalities will thus not be comprehensively provided for in separate modular units in the architecture description, often leaving functional traces in some modules. This architectural description may lead to some false positives in the architecture assessment process [21]. A fully modularized feature is left in the architectural description, which may lead to false negatives in the analysis process.

D. Blurred Inter-Modular Boundaries

The dependency between system requirements is a piece of pivotal information for software architects to design well-defined modules. However, as modules take distinctive paths to design change, existing coupling metrics may inaccurately identify architectural inter-module dependencies. The overall outcome is blurred inter-modular boundaries and tight interfaces coupling [21]. Similarly, the evolution of the software system may keep increasing the complexity of the design, effectively omitting finite details of modular characteristics. The phenomenon, if unchecked, will also lead to a fuzzy inter-modular boundaries description.

E. Inaccuracy in Identifying Instabilities:

The output of a modular system is based on the seamless and smooth function of every module and the communication of the modules. Where there is instability in the system, there remains a challenge in identifying the source of instability in the complex modular system. The problem is that conventional metrics cannot accurately identify the unstable element/module.

Table 1. Modular system pros and cons summary

S/N	Pros	Cons
1	Customizations	Architectural Mismatch
2	Less interdependency	Physical variability
3	Third-party extensions	Inaccuracy in modular analysis
4	Independent development	Vague inter-modular boundaries
5	Small and robust core application	Difficulty in identifying instabilities
6	Reusability	
7	Re-factorability	
8	Scalability	

VII. MODULARITY METRICS

A module is an assemblage of components that share a common characteristic and assemble according to this common characteristic to accomplish a designated objective. Within each module, components are strongly connected among themselves and relatively weakly connected to components in other modules. It is important to measure the strength of these connections between components which determines the modularity of a software system. Software architects acclaim software modularity metrics to monitor projects, discover non-conformities and point out risks like low modularity in software projects since the early stages of project development [22]. Some of the metrics to measure modularity include:

A. Cohesion

Cohesion refers to the relationship between the internal elements and how cohesive the connections are [23].

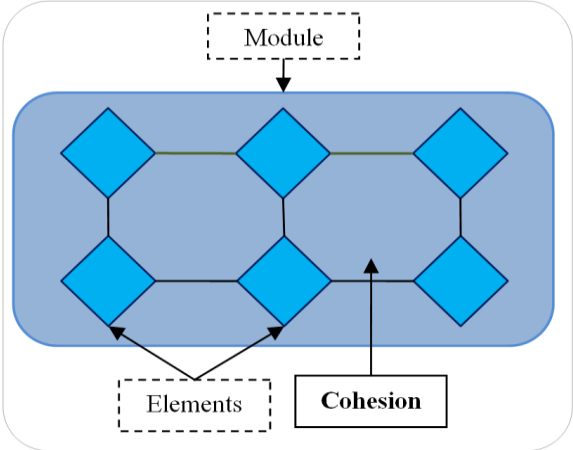

Fig. 7 Modular Cohesion

The Page-Jones theory on structured systems design focuses on different forms of cohesion for a modular architecture: 1) Functional cohesion refereeing to when all elements of a module contribute to a single well-defined task. 2) Sequential cohesion is when elements of a module are grouped because the output from one element is the input to another element (for example, a function that reads data from a file and processes the data). 3) Logical cohesion in cases where elements are grouped logically. 4) Temporal cohesion where components are related together in a time-space, e.g. an initialization module. 5) Communicational cohesion if all activities within the module act on the same input or output data. 6) Procedural cohesion in instances where activities of a module are sequentially executed together to perform a specific task, and 7) Co-incidental cohesion where elements in a module are related together in an unplanned and random manner. This relationship is deemed meaningless as it may lead to further decomposition of the module [24].

B. Coupling (Degree of Interdependence)

Coupling in software engineering refers to the degree of interdependence between software modules, measuring how closely connected two modules are [25]. The software requirement specification document defines various aspects of inter-modular dependence and independence.

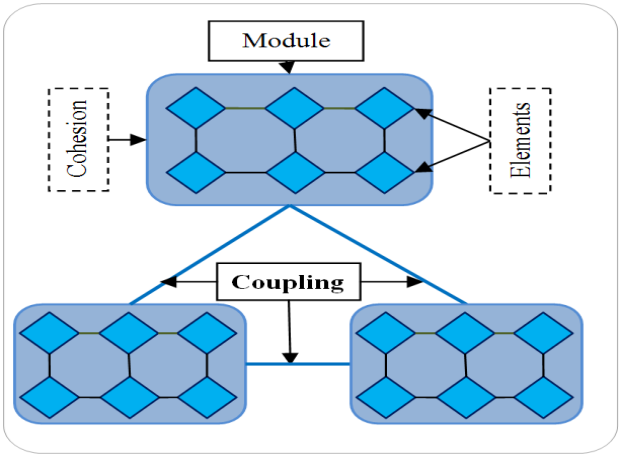

Fig. 8 Modular Coupling

The degree of interdependence between modules can be broadly categorized as afferent and efferent coupling. Afferent coupling is a metric that measures the total number of elements outside of a module that depends on elements within the module. In contrast, the efferent coupling measures the total number of elements within the module that depend on elements outside the module [22]. Further, the Page-Jones model identifies various forms of coupling in modular designs: 1) Content coupling, which refers to interdependence where one module can directly access or modify or refers to the internal content of another module. This is the highest form of interdependence. 2) Common coupling, where a number of modules have access to read and modify a shared global area. 3) Control coupling in cases where one module controls how another module functions. 4) Data Coupling occurs when one or more modules have some parameterized data communication. 5) Stamp coupling, where modules share some common data structures but work on different sections of the shared data structure [22, 24].

C. Instability

The instability metric measures the instability of components, where stability is measured by calculating the effort to change a component without impacting other components within a software application. Santos et al. support this position while analyzing Martin's instability measure. They opine that if an entity has a high value of instability, then there is a high risk of undesired changes that could affect the analyzed entity's behaviour due to changes in other system entities [22].

VII. MODULARITY TRAVELOGUE

The quality of a software artefact and its longevity is determined by its architecture to a great extent [26]. It is therefore imperative for software architecture to evolve with time to meet the evolving needs of software users. Eoin Woods takes a pragmatic look at the five ages of evolution of software systems and the accompanying five stages of software architectures [27]. The review illuminates the path modularity has taken over the architectural ages to present-day architecture. With each evolution, modularity has changed from the original monolithic modules to today's microservice modules.

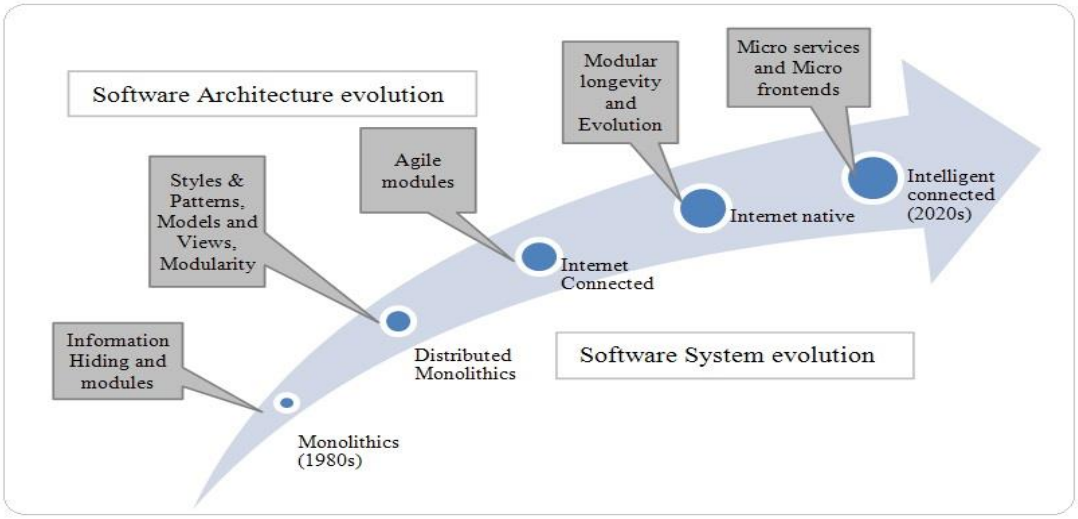

Fig. 9 Modularity travelogue: Adopted with changes from Eoin [27]

The modularity of software has largely paralleled that of the software industry, with architects' techniques and concerns changing in response to the changing challenges the industry has faced.

Today's software systems are more network-centric, and intelligent modularity has morphed to provide intelligent interfaces and entry points that no longer need to be bound to physical computing stations. David Garlan supports this position and identifies the network-centric nature of software artefacts as a driver for present and future software architectures [28]. By evolving to support code reusability through modern frameworks, the concept of modularity is embedding itself in the emerging new architectures while allowing for the development of new software paradigms [27].

The concept of modularity is embedded in new cloud-based technologies like SaaS, PaaS and IaaS. As businesses become more differentiated, their software needs become more custom necessitating customizable software to create their unique experience. SaaS vendors utilize modular SaaS systems where customers can get different experiences by turning on or off services packages in modules.

Eoin's travelogue shows that present and future intelligent, Internet-native systems will continue to be dynamic and composed of fine-grained network modules (micro-services) [27]. The modules are often built on SaaS/PaaS platforms, allowing customers to choose what modules fit their needs, economic capacity, and technical viability to serve unique requirements. Modularity is an architectural design poised to remain a dominant style of designing software systems.

VIII. CONCLUSION

Modularity occupies a pivotal position in the design of good software system architectures. Several software projects have adopted modular design by going a level beyond layered, and SOA approaches. The architecture resolves the problem of monolithic complexity and granular layered systems that are difficult to design, develop and implement. Studies have proven and documented that modular design allows for refactorbility, reusability, customizations, software collaborations, and scalability. Further, it is established that modularity leads to minimalistic core applications allowing for in-depth understanding and simplifying maintenance of the core system.

However, modular design also comes with some challenges: architectural mismatch, physical variability and blurry inter-module boundaries expose designers to barriers that need solutions. However, it is notable that considerable development of modularity measurement metrics will keep fore sighting the challenges and thus have them addressed at the inception stages of software projects. Further, current trends in software systems design and development show that modularity remains a dominant style where existing and emerging styles incorporate the concept of modularity to address the inherent limitations of non-modular systems.

REFERENCES

[1] (2021). Britannica T, Editors of Encyclopaedia, Software. Encyclopedia Britannica. [Online]. Available: https://www.britannica.com/technology/software
[2] Garlan D, & Shaw M, An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering. (1993) 1-39.
[3] (2010). Chapter 3: Architectural Patterns and Styles. [Online]. Available:https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658117(v=pandp.10)?redirectedfrom=MSDN
[4] (2014). Rengaiah P, On Modular Architectures. Medium. [Online]. Available:https://medium.com/on-software-architecture/on-modular-architectures-53ec61f88ff4.
[5] (2021). Accelerator (software) - Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Accelerator_(software)
[6] Mehta M. R, Lee S, & Shah J. R, Service-Oriented Architecture: Concepts and Implementation. In Proceedings of the Information Systems Education Conference (ISECON). 23(2335) (2006) 1.
[7] Tutisani T, Effective Software Development for Enterprise Beyond DDD, Software Architecture, and XP 1st edition: Leanpub. (2020).

[8] Narduzzo A, & Rossi A, Modular Design and the Development of Complex Artefacts: Lessons from Free/Open Source Software, Quaderno, DISA. 78 (2003).
[9] Knoernschild K, Java Application Architecture: Modularity Patterns with Examples Using OSGi. Prentice-Hall Press. (2012).
[10] (2021). Tutorialspoint, Software Architecture & Design Tutorial - Tutorialspoint. [Online] Available: https://www.tutorialspoint.com/software_architecture_design/index.htm
[11] Giakoumidis N, Bak J. U, Gómez J. V, Llenga A, & Mavridis N, Pilot-Scale Development of a UAV-UGV Hybrid with Air-Based UGV Path Planning. In 2012 10th International Conference on Frontiers of Information Technology IEEE. (2012) 204-208.
[12] Schilling M. A, Toward a General Modular Systems Theory and its Application to Interfirm Product Modularity, Academy of Management Review. 25(2) (2000) 312-334.
[13] (2020). Mustafic A, Modular Application Architecture. [Online]. Available: Goetas.com https://www.goetas.com/modular-application-architecture-intro
[14] (2021). Package Development - Laravel - The PHP Framework for Web Artisans. [Online]. Available: https://laravel.com/docs/8.x/packages
[15] Soothram S, Efficient Techniques for Partitioning Software Development Tasks. (2010).
[16] (2021). Kibabii, Kibabii University. [Online]. Available: https://kibu.ac.ke/kibu-automation-dream-accomplished-in-2019-2020-academic-year.
[17] (2021). Moodle, Moodle Plugins Directory. [Online]. Available: Moodle.org https://moodle.org/plugins
[18] (2021). OpenMRS I, OpenMRS. [Online]. Available: Openmrs.org https://openmrs.org
[19] (2020). Brinkman S, & Delamore D, The 5 Essential Elements of Modular Software Design, Medium. [Online]. Available: https://shanebdavis.medium.com/the-5-essential-elements-of-modular-software-design-6b333918e543.
[20] Nesnas I. A. D, Simmons R, Gaines D, Kunz C, Diaz-Calderon A, Estlin T, Madison R, Guineau J, McHenry M, Shu I.-H, & Apfelbaum D, CLARAty: Challenges and Steps toward Reusable Robotic Software, International Journal of Advanced Robotic Systems. (2006). https://doi.org/10.5772/5766
[21] Sant'Anna C, Figueiredo E, Garcia A, & Lucena C. J, On the Modularity of Software Architectures: A Concern-Driven Measurement Framework. In European Conference on Software Architecture, Springer, Berlin, Heidelberg. (2007) 207-224.
[22] Santos D, de Resende A. M. P, Lima E. C, & Freire A. P, Software Instability Analysis Based on Afferent and Efferent Coupling Measures. J. Softw., 12(1) (2017).
[23] Budd T, Introduction to Object-Oriented Programming, Pearson Education India. (2008).
[24] Page-Jones M, The Practical Guide to Structured Systems Design. Yourdon Press. (1988).
[25] (2021). Coupling (computer programming) - Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Coupling_(computer_programming
[26] Northrop L, Trends and New Directions in Software Architecture, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst. (2014)
[27] Woods E, Software Architecture in a Changing World, IEEE Software. 33(6) (2016) 94-97.
[28] Garlan D, Software Architecture: A Travelogue, In Future of Software Engineering Proceedings. (2014) 29-39.