Evolution of Software Architecture over Decades

R. Rashmi, Dr. G N Srinivasan
Research Scholar, Research Guide, VTU,
RV College of Engineering,
Bengaluru, Karnataka, India
{rashmir, srinivasangn} @rvce.edu.in

K. B. Shubha Raj
Research Scholar, VTU, PES University,
Bengaluru, Karnataka, India
shubharaj.kb@gmail.com

***Abstract* – Through this paper, an attempt is made to portray the evolution of Software Architecture over a significant period of time with a perception of how it revamped itself to meet the changing trends in IT Industry. The current generation software systems are getting rich in features and exhibit complex behavior. Representing these scenarios poses several challenges at the architectural level. To address these challenges, dynamic design decisions are recommended in creating the Software Architecture. In the present work, vital research activities are highlighted along with examples and streamlined addressing the architectural capabilities relevant to the Industry 4.0 generation systems.**

I. INTRODUCTION

Software is the term used to refer to an application, data and program or script constituting set of instructions written within computers to solve specific tasks. Whereas, a software system is typically a combination of hardware devices and intercommunicating software components constituting of programs, configuration files, user manuals and so on.

A relatively new term is coined called as 'Software Intensive System' to identify the system in which software interacts with other software components, computer systems, hardware devices, IoT sensors, data source and with people[1]. This clearly indicates that software has become crucial part of all the applications, products and services contributing to the industrial growth. Therefore, comprehension of how such software behaves is important for a Software Engineer to appreciate/realize the purpose of the software intensive system being built. The software needs an architectural design to represent the behaviour of the software system. This architectural design, termed as Software Architecture (SA) [2], depicts the behaviour and servers as a blueprint of the entire system. It is referred by all the stakeholders involved in development and usage of the software intensive system to know how the project team is developing the software, what is the work assigned to the design and development teams, the key quality attributes the system has to possess, and so on. In total, SA acts as a communication medium among all the stakeholders [2],[7] and enables them to analyse and evaluate the correctness of the system during the early stages of Software Development Life Cycle (SDLC). This helps in identifying and mitigating design errors and risks even before the system is built thus leading to reusable, cost-effective, and time-to-market system development.

It is observed that, the term software design and software architecture are interchangeably used sometimes and the software design is depicted as 'inclusive of software architecture'. Though related to each other, these terms have to be distinctly used. Software design represents the design of individual modules/components. Whereas, the highest level of abstraction of the system is represented as software architecture.

Software design focuses on implementation details of specific module/basic part of the system being built. They are represented as UML diagrams, charts, flowcharts, algorithms and data structures of specific module/component. Whereas, Software Architecture focusing on the purpose of the system and structured as a 'skeleton' of the system. The Software Architecture involves abstract representation of the software system with specific properties (also called as quality attributes). Examples include Dataflow Systems, Call-and-Return Systems, Independent Components, Virtual Machines, Data-Centred Systems and Layered Architectures.

A. Design Levels in SDLC

In SDLC, Software Design follows the Software Architecture. Finalization of software architecture goes through several reviews and iterations as shown in Fig. 1. Once the requirement specification, architecture and design are approved, implementation phase begins followed by other phases of SDLC.

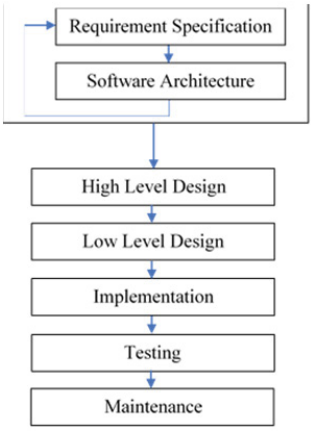

Fig. 1. SDLC depicting Three Levels of Design

As evident in Fig. 1, Software Architecture representing the abstract design of the system must be built first based on its requirement specification during the software development process. The Software Architecture must then be levelled down to High Level Design and Detailed Design (Low Level Design) to characterize their structural and behavioural aspects. The transition between these design levels determine the

success of a software system development. It also enables the system to meet its gross requirements including functional and non-functional attributes.

The features of three design levels are detailed below with illustrations.

Abstract Design Level: Software Architecture represented at this level enables realization of quality attributes in the software system. It is a generic and vendor neutral design abstraction.

The design issues at this level involves composition of overall system elements into components and connectors. Rules of governance, business policies, technological limitations, and changing requirements decide how these elements interact with each other. These constraints are addressed by the Software Architect instantiating an architectural style/pattern. The architectural patterns/styles are discovered from practical implementation and based on his expertise.

High Level Design: It consists of recurring design elements which are reusable across the system called as high level design patterns. These patterns are well proven and tested solutions that provide clarity to Software Architecture. They are represented using UML diagrams.

High level Design patterns deal with issues like object creation, organization of objects into larger and flexible modules, and provide interface through communication patterns.

There are patterns that are both static and dynamic [2]. Design patterns are also Platform / Framework Specific (based on the developing and operating environment) and independent of them. J2EE Design Patterns - Front Controller, Composite View, Service Locator Pattern, etc. used in enterprise application development are examples of platform specific high level design patterns. Whereas, Loosely Coupled Control Patterns used in designing dynamic system structures like smart parking system may be platform independent. The pattern may be used in a scenario where intelligent parking spots are to be located and monitored based on occupancy of each location corresponding to region [3].

There is an extensive scope for cross platform design patterns in present situation. These patterns separate and encapsulate the implementation of platform specific functionalities using platform neutral interfaces [4], [32].

Low Level Design: The modules within a system are represented using UML diagrams that form basis of work assignment for developers. The design concerns here are choice of algorithm, data structures. They are directly implemented using specific programming languages to realize quantitative, functional requirements of the system.

Table I summarizes the artefacts used to represent a software system at all these design levels with examples. It must be observed that, each level focuses on specific system concerns to address the issues at respective levels. Through this, it is obvious that software architecture is rendered after requirement specification and before high level and low level design in SDLC.

In summary, it must be comprehended that, any software built for computer-based systems will exhibit one/ two or more combinations of the many software architectures/architectural styles (discussed in the following sections) and each architectural style will consist of:

- **Components**: The components are the core computational parts of the system and can be heterogeneous; developed on different platforms, using different programming languages. Examples of components are clients, servers, databases, filters, layers of hierarchical system and sensors in IoT.
- **Connectors**: The connectors correspond to an interaction mechanism among the heterogeneous components. Connectors will help to establish communication and coordination between components. They include procedure call, shared variable access, client-server and database accessing protocols, asynchronous event multicast and piped streams, communications protocols including Near Field Communication, Wi-Fi, Bluetooth, and so on.
- **Constraints**: The elements of Software Architecture are constrained on how these elements interact with each other based on the context. These constraints reflect the design decisions made within the technological, business and environmental limitations under which the system is intended to work. One of the constraints could be deciding how components should be integrated to form a system in distributed environment. Trade-off between overall system properties (e.g., security vs performance) while developing semantic models.

Subsequent sections of the paper emphasis on the aspects leading to systematic evolution of Software Architecture. The concepts are narrated with an intent of enabling Professional software developers and research scholars to assent with the idea of architecting system models and appreciate the need for Software Architecture in SDLC to achieve high quality.

The review is supported with examples to help academicians and students gain accumulated knowledge of Software Architecture taxonomy and its accomplishments.

The paper is organized as follows: Section 2 puts forth various definitions of Software Architecture in order to comprehend the technology from different perspectives. Section 3 narrates representation of Software Architecture. Section 4 details the Impact of Software Architecture on the business, technology and operational environment; the influence of these on SA is also appraised. The 6th section elaborates the literature review on - evolution of SA with business and technological advancements in IT industry.

In remaining sections, i.e., section 7, current architectural challenges are identified and future directions are traced based on the research gaps observed; section 8 recommends software architectural solutions for current industry generation software systems. The paper concludes along with an insight to future work in section 9.

TABLE I. DESIGN LEVELS OF SOFTWARE SYSTEM

Representation	System Structure	Design level	Artefacts	Focus on	Example
Software Architecture / Architectural Style / Architectural Pattern	Dynamic	Abstract	Components Connectors	Realising – Quality Attributes / System concerns / Non-Functional Requirements	Pipes & Filters, Layered Systems, Repositories, Broker, Model-View-Controller
Design Patterns	Static and / or Dynamic	High	Recurring design elements	Providing reusable solution – to commonly occurring problems within given design context	Master-Slave, Loosely coupled control pattern, J2EE design patterns
UML diagrams	Static	Low	Modules	Implementing – System functionality (Functional Requirements) using algorithms and data structures	Interaction diagrams like Sequence Diagram

II. SOFTWARE ARCHITECTURE TERMINOLOGIES

Though Software Architecture has high substantive value for software engineers, they have been endeavouring to optimize Software Architecture in terms of taxonomy and development process [4].

Several stakeholders at different levels in Information Technology Industry have been defining the term 'Software Architecture' from different perspectives. More details get added as and when they experience new technological challenges and innovations. A collection of such definitions is put together in a bibliography by SEI. As mentioned from SEI, these definitions are taken from various books, papers and articles published since 1990s and are available in its website http://www.sei.cmu.edu [5]. Our comprehensive definition for the term 'Software Architecture' is likely to be in compliance with the terminologies forming 'Basic Repertoire' of Software Architectures [1], [6].

There are several terminologies often linked with the term Software Architecture. A brief comprehension of some of the related terminologies is tabulated in Table II to give more clarity about the concept.

All of these technologies address the problem of structuring a system at a very high, abstract level. The difference exists in the scope and its representation. The scope of architectures includes System-wide, Processor-wide, Collaboration-wise or Intra-object representation and other target environment in which the system is operational.

Throughout our work, we consider Architectural Patterns and Architectural Styles as similar since both terms convey same intent though representational autonomy exists. We must agree that both pictorial representation and descriptive documentation is necessary to understand Software Architecture and communicate the idea effectively among stakeholders. Hence, we use the terms interchangeably.

III. REPRESENTATION OF SOFTWARE ARCHITECTURE

A typical Software Architecture can be represented as boxes and lines depicting the system elements. Here, the boxes represent components and the lines are connectors. This representation may include idioms and phrases to convey meaning behind the symbols and provide some rationale for the specific choice of components and interactions. These idioms and symbols form common vocabulary across the organization and forms basis for understanding broader system level concerns such as patterns of communication, execution control structure, and quality attributes such as scalability and security [1].

Consider an example of a traditional compiler. Fig. 2 depicts its Software Architecture. The design diagram embodies abstract representation of subsystems of the traditional compiler as components. The lines represent the connectors indicating sequential order of execution during generation of machine code from source code.

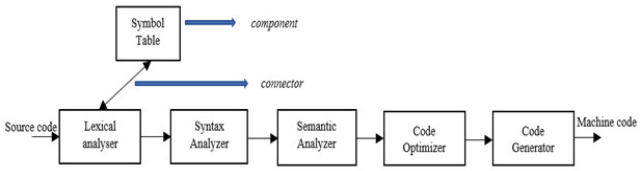

Fig. 2. Software Architecture of Traditional Compiler

In essence, a well-designed macro level software system is a collection of well-formed modules at the micro level that are integrated to form overall software structure. Software architecture diagrammatically represents both levels of such software systems.

At the at macro level, software architecture structures the system as a sub-total of the components/subsystems and the addresses issues at system-level structural composition in an enterprise. Though, at the micro level, the subsystems are levelled down to modules to be implemented in later stages. Software architecture, at the macro level of a system, deals with finely grained design issues like functionality performance optimization and security and so on. The goal of the software architecture is to develop reusable and cost-effective software systems.

TABLE II. SOFTWARE ARCHITECTURE TERMINOLOGY

Terminology	Description / Representation	Examples
Architectural Style/ Software Architecture	The structural organization of Software System is represented using vocabulary of components. Interaction among components are represented using connectors and constraints on how they can be combined [1].	Dataflow systems, Call-and-Return systems, Independent components, Virtual Machines, Data-Centred Systems [1].
Architectural Patterns / Pattern Oriented Software Architecture	"Results of previous experiences learnt through different case studies are presented as well-proven generic solution scheme. They are structurally documented and reused on commonly occurring problems in specific design context. Also defines the implementation strategies of the components, their responsibilities and relationships, and the ways in which they collaborate" [7].	Layers, Pipes and Filters, Blackboard, Broker, Model-View-Controller, Presentation-Abstraction-Control, Microkernel Architecture, Reflection Architecture [7].
Reference Architecture	Reference Architecture is a "reference model mapped onto software elements and the data flows between them" [8]. It facilitates a shared understanding across multiple products, organizations, and disciplines about the current architecture and the vision on the future direction. Diagrammatic representation includes integration of various Architectures in context of Business, Technical and Customer aspects [8], [9].	OASIS Reference -Architecture for Service Oriented Architecture [10], The Open Group SOA Reference Architecture [11], Global Justice Reference Architecture (JRA).
System Architecture	A collection of components that are connected to accomplish a specific purpose described as one or more models possibly with different Architectural Views.	Integrated AUTODIN System Architecture [12], DARPA funded project known as ARPANET[13], Engineered systems like Airline system, Energy distribution system.
Enterprise Architecture	Aligned with goals of the Enterprise. Captures business processes and workflows. Encompasses various systems, software and technical architectures. Addresses Enterprise-wide issues such as security, inter-operability, integrity, consistency and reuse across multiple systems within the organization.	Federal Enterprise Architecture [14].

IV. TAXONOMY OF ARCHITECTURAL PATTERNS

Architectural Patterns can be broadly categorized into Routine and Innovative patterns based on their novelty.

Routine design solves repeatedly occurring problems while architecting a system. Architects create a repository of design artefacts by capturing their problem-solving skills and experiences in the form of design documents and code libraries (APIs). Software Engineers pickup relevant, reusable parts of these solutions from the repository and use them to solve similar recurring problem. This helps them realize specific system property with accuracy.

Table III lists a few of such architectural styles and typical forms of applications implemented from them.

TABLE III. ARCHITECTURAL STYLES AND CORRESPONDING FORMS OF APPLICATIONS

Architectural Style	Forms of Application
Pipes and Filters	Compilers – where sequential processing is involved
Main Program and Subroutine	Stand-Alone Applications – simple and medium level software
Hierarchical Layers	Operating Systems and Network Layers
Model View Controller	Web Applications – provides interactive user interface

Innovative patterns find novel solution to unfamiliar problems. These patterns are essential for building original software systems operating in uncertain situations. Such systems require frequent reconfiguration to adapt themselves to frequently changing requirements. Such applications are domain specific; with deliberative and reactive behaviour.

Innovative patterns find novel solution to unfamiliar problems. These patterns are essential for building original software systems operating in uncertain situations. Such systems require frequent reconfiguration to adapt themselves to frequently changing requirements. Such applications are domain specific; with deliberative and reactive behaviour.

Table IV lists innovative architectural styles and typical forms of applications implemented from them.

TABLE IV. INNOVATIVE ARCHITECTURAL STYLES AND CORRESPONDING FORMS OF APPLICATIONS

Architectural Style	Forms of Application
Reference Architecture	Robotics – with highly flexible and adaptive features, Automatic Vehicle Management – driverless cars.
Reflection Architecture	IoT – with self-healing capability
Microkernel Architecture	Portable Operating Systems, Plug-and-Play extensions.

It may be observed that, an architectural pattern can be used to develop different forms of applications of similar kind; provided, suitable changes are allowed to be made to that software architecture according to the requirements. Similarly, several architectural patterns may be combined together to develop such applications. In such scenarios, software engineers must remember that each pattern enables specific property in a system being built. Hence, it is necessary for them to explore and compare alternative styles/patterns which yield different consequences.

As said, it is obvious that, most software systems support several quality attributes and must be addressed by a set of combined architectural patterns. These patterns used within a software intensive system must be compatible with each other.

For example, both microkernel and MVC architectural styles may be combined to develop adaptable web applications with highly interactive User Interface [7].

Another example is, while building enterprise applications, the complex domain logic may be implemented using a rich Domain Model pattern. The patterns most suitable for data source layer in this case is combination of Query Object and Repository patterns. This is more relevant in situations where developers cannot tell whether the objects were retrieved from in-memory or from the database (high abstraction between database and view is needed). MVC is the best choice for building the presentation layer here too [33].

In total, the selection of architectural pattern should be based on above knowledge and driven by prioritised non-functional requirements of the system being built.

Combining aspects of both routine and innovative design mechanism improves productivity in an organization by leverage existing system capabilities.

Architects need to identify system elements that could be routine, and innovatively develop supporting artefacts appropriately. This approach enables software architecture to adapt to evolving technologies and changing system requirements.

Following sections provide detailed description of impact of software architecture on the system and its evolution based on several aspects.

IMPACT OF SOFTWARE ARCHITECTURE

The significant aspect of Software Architecture is that, it identifies the properties of individual components and its interfaces that form the basis for the design of a good Software System [6], [9]. A Good Software Architecture, when included in the SDLC, ensures realization of System Quality Attributes (non-functional requirements) like Reliability, Security, Performance, Modifiability, etc., [6]. These quality attributes are evaluated earlier at the architecture level to reduce development effort and time-to-market. Hence, Software Architecture predicts the efficiency of the software product and also helps Architects to understand the complexity of the system. In contrast, a bad architecture leads to system disasters.

From previous sections, it is comprehended that structural compositions of software architecture are guided by its constraints. These constraints are influenced by technical and business aspects. Local and international governance and policies, social concerns also will influence the business and technical decisions of an organization.

Business strategies affect project development process, time and its cost. Development process also depends on the technical skills of the employees. Changing customer requirements determine the features of the system. Other stakeholder's concerns contradict with each other leading to trade-off of quantitative and qualitative attributes of the system. For example, performance of an embedded system with minimal resources will be affected dynamically with selection of cryptographic algorithms to secure the system. Such issues should be addressed at architectural level, early in stages of SDLC.

Hence, software architecture of a system is influenced by the organization's business strategies, product requirements and system's operational environment. It is the responsibility of the software architect to solicit the needs of the stakeholders through trade-off, understand the technical and business goals and prioritise the constraints at system level. The architect is expected to review the architecture based on his experience and expertise. This enables possibilities of expanding the enterprise goals; taking advantages of previous investments in architecture. Thus, software architecture significantly reduces the cost and effort of system building through reusability.

Consequently, in the process of getting influenced, the software architecture impacts the business and technological advancements too. It ensures that the supporting business solutions reflect the technological expansions stimulated at architectural levels.

Industrial trends too, have been changing drastically since invention of computers that manifested 3rd industry generation. Advent of the internet, ascent of data and large globalised businesses, growth in computer power and interlinked financial systems let to evolution of sophisticated software engineering process that was directed by software architectural patterns.

Rapid technology change in fields of artificial intelligence, data analytics are leading to automation and hyper connectivity. Due to high agility, industries are moving towards Internet of Things, Internet of everything. Over all, there is a drastic shift from enterprise to digital enterprise and to intelligent enterprise. In such circumstances, the role of software architecture is indebted because of its capabilities.

The notion of devising software architecture for a system and implementing from that have several advantages. A few of them are listed below [7]:

- SA acts as a communication media between various stakeholders to share their perspectives about the system.
- Allows architects to incorporate quality attributes into the system being built and evaluated at the early development stage.
- Helps architects in "capturing design decisions and past experiences of stakeholders in the form of a document that serves as shared, semantically rich vocabulary for any organization".

- Enables architects to reuse idioms, patterns and styles at abstract level through 'Routine Design'.
- Makes it possible for organizations to build the products at reduced cost through reusability.
- Reduces the efforts of developers and testers by facilitating them through architecture-based frameworks and tools.
- Enables organizations to achieve Time to Market through faster and efficient development techniques.

V. Evolution of Software Architecture

The beginning stage of evolution of Software Architecture can be traced back to the days when scientific basis was applied to engineering practice during 1960 [1]. Different programming languages were invented during this era namely, COBOL, ALGOL, PASCAL, FORTRAN and so on. Data was embedded in the programs. Each of these programming languages had their own file systems which made it difficult for the programs to deal with data in other programs and avoided programs to access data of other's. The maturity in theories led to abstraction of data structures from individual programs. Because it describes process without referring to actual data, abstract data types were born and formed a way to the beginning of software architecture.

Abstract data types gave a way to explore and address issues like specifying abstract models, representing software structures, language issues, data protection, integrity constraints and rules of composition [1]. Programming language provided solutions to these issues through predicates and language constructs bound by conventional syntax and semantics. With the proliferation of programming languages, it became necessary to describe functionality of a system without referring to its features which is second stage in evolution of software architecture.

Third stage in evolution was to describe a procedure before writing programs. This procedure was common for all programming languages like flow-chart or algorithms and lead to Single processor-multi threading, multitasking and multi-processing concepts. Examples of such programming languages are Interface Definition Language (IDL) and Module Interconnection Language (MIL).

Further, with complete separation of data and programs, separate ownership of data and programs were established. This requirement raised several issues like securing data, connecting between data and core computation, etc. Software Design alone was not sufficient to handle the issue as these concerns were beyond the selection of data structures and algorithms for computation. This provided a way to go ahead of design to address system concerns (quality attributes) like security, reliability, performance, etc. This is the fourth step of software architecture Evolution.

Through these stages, it is evident that programming constructs evolved first and then software architecture evolved through them. In concise, software architectures evolved as generic models which are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems.

Functional requirements of a system are quantitative in nature and easily attainable through programming languages. Achieving non-functional requirements is a challenge as they are both quantitative and qualitative and requires software architecture to imbibe these aspects at early stage of system development. Now that we have realized the significance of it, we write SA first and then use programming languages to put them into operation.

Further, we quote a few systems as best examples that leveraged from implementation level to architectural level imbibing general characteristics represented more abstractly.

Example 1: Transformation of compilers from traditional model to modern canonical compilers:

Initially, compilation was regarded as a sequential process of translation through lexical analysis and code generation phases. The model represented Pipes and Filters Architectural Styles belonging to batch sequential version. As this model was not completely accurate, it was improved by adding a separate symbol table and was connected to lexical analyser. Data was passed to subsequent phases in sequential order.

As the complexity of algorithms increased, intermediary code generation phase that attributed parse tree was introduced leading to Modern Canonical Compilers. These compilers represented a significant shift in the architectural styles. They were moulded as more appropriate structures that re-directed attention from sequences of passes to central shared representation of data structures. The symbol table and parse tree were now connected directly to all phases of compiler which enabled direct data access and manipulation. This system advanced into Repository architecture style that is more accurate and appropriate [1].

This process leads to following accomplishments at architecture level:

- System components could be structured to operate independently, and at the same time, interact through each other via shared data repository.
- Flexibility was introduced in determining the order of execution of operations.

Example 2: Pipes and Filters in Unix operating system:

Successful implementation of pipe and filter commands in the shell to transfer data from one command to another lead to Pipe and Filter Architectural Style [1]. Here, Filters represent components and pipes represent connectors.

Accomplishments at architecture level:

- System components were organized to form a series of operations that produce highly specific results.
- Components were structured as independent entities leading to parallel programming.
- Connectors carried specific type of data and connected output of one component to be fed as input to subsequent component.

Example 3: In certain systems, where there was a need for two or more processes to communicate in one system, Shared Memory concept evolved. Advancement in this technology led

to development of Multi-Processing and Distributed Systems. Remote Procedure Call (RPC) was introduced to send request parameters from client stub, execute the procedure in server, and send back the response to respective client through server skeleton. RPC was used only when the client and server systems are built using same platform.

To address portability issues when servers were built on multiple platforms, Remote Method Invocation (RMI) was introduced. RMI was soon replaced by CORBA technology to solve Heterogeneity issues [6]. CORBA proved to be a successful concept to handle active objects (COM and DCOM). Thus, Broker Architecture arrived [7].

Accomplishments at architecture level:

- Software Architectures allowed for flexibility in choosing underlying communication protocols based on the application domain.
- Addressed quality issues such as portability and security through inclusion of relevant platform and frameworks.

These examples are substantiating that, as the requirements for specific systems evolve, the corresponding architecture that addressed both quantitative and qualitative system attributes also advanced. This led to the conceptualization (modelling) of Architectural Patterns.

During system modelling, repeatedly occurring issues were observed in the problem space (real world environment). To solve these issues, a set of generic functionalities (including both functional and non-functional) were framed for the associated solution space (software system being developed to solve the problem).

To realize these functionalities, a set of individual and reusable software elements were found; the relationships between them were outlined and generalized into architectural patterns. Various architectural patterns were introduced based on the specific problem domain, structuring of software elements (also called as components) using interfaces (also called as connectors) which are course-grained structures.

As the architects got more experience in a specific domain, they documented all their previously known design solutions in handbooks and manuals. The software engineers with medium credentials started reapplying this knowledge to achieve system requirements. Later, such design solutions were identified and categorized based on requirements/goals they met. Additional information like, trade-offs they made or system structures they created made it easy for engineers to reuse large portion of prior solutions. These reusable methods were enriched, standardized and documented that led a foundation for 'Routine Design' patterns.

It is obvious that, any changes in the problem space demands alternative selection of architectural patterns. Since the quality attributes that a system must imbibe depends on changed environmental and business goals, the selection and refinement of relevant architectures becomes a crucial and significant process. This process is also iterative and depends on the "Design Decisions" made by the architects. Such design decisions include system rationale, design rules and design constraints [15].

These architectural patterns are innovative in nature and require sufficiently fine-grained software entities. The entities involve design decisions to achieve quality attributes in a system and are also termed as design tactics [6]. Different Architectural Patterns can be used to realize one or more tactics. At the same time, one tactic can be used in many patterns; provided, the solution is based on the context of related problem space.

Following examples are indication that patterns arrived from continuous observation, experience and expertise of Software Architects.

Example1: Layered architecture was applied to networks and a few good operating systems [1], [7]. This led to standardization of OSI ISO model and some of the X Window System protocols. This success influenced the pattern to be applied to model systems in other domains.

Accomplishments at architecture level:

- Reusability of architectural decisions became fundamental aspect of Software Architectures.
- The pattern support enhancement of system components with little modifications.

Example 2: Programming language constructs correlate with abstract data type architectural styles as explained in previous sections of this paper.

Accomplishments:

- Data Representation is independent of functionality as changes in data formats are easily accommodated through interfaces.
- Modularization allows changes in processing algorithm and enhancement of system components.

Based on above illustrations, it is determined that design patterns emanated from reusable entities at architectural level similar to origin of language constructs from reusability concept.

Further, a critical need to reduce complexity in software gave birth to Object Oriented Paradigm. Many architectures including Abstract Data Types and Objects, layered and hierarchical patterns may be implemented using object-oriented programming to control complexity through abstractions.

In third industrial revolution, the software design methodologies aimed at one-of-a-kind applications, designs are expressed in terms of objects and classes, and software was coded manually. Gradually, the role of software became decisive factor with advancement in industry trends and change in business priorities.

The fact that factories do not manufacture individual products, but instead create families of closely related products, lead to development of integrated systems. System functionalities such as material requirements planning was superseded by Enterprise Resources Planning tools that enabled humans to plan, schedule and track product flows through factory. Distribution of these factories across wide spread geographical locations lead to concept of Supply Chain Management.

With this advancement, software intensive systems became more complicated and required different approaches.

Numerous architectural styles/patterns were introduced which are domain specific and a few others are widespread. A few significant ones are listed below:

- Process Control Systems were intended to provide dynamic control of a physical environment [16].
- State Transition Systems organized many reactive systems and defined systems as states; denoting a set of named transitions that move system from one state to another [17].
- Product-Line Architectures were designed for families of related applications. PLA served as blue-print for creating families of related applications [18].
- Domain-Specific Software Architectures [19] were specialized to increase the descriptive power of structures.
- REST architectural styles were developed for applications that are accessed over network in the form of services identified by URIs. Restful Web Services represent stateless client server architecture [20].
- Service Oriented Architecture (SOA) style represented collection of services communicating and coordinating with each to perform activities. SOAs applications enabled applications to be more responsive and competitive [21].
- Microservices Architectures structures formed a set of fine-grained autonomous services modelled around a business domain [22].

Several architecture styles can be combined in a single design giving rise to Heterogeneous Styles. This enables sophisticated system development that imbibes good practices of all the embedded styles. The quality attributes accomplished here at the architecture level are:

- The architectural representation helped in understanding the its system complexity more clearly; serving as communication medium across all stakeholders.
- Efforts were made in Software Architectures to accomplish reusability of system artefacts over multiple products and reduce the costs of software development.
- Realized scalability and maintainability by making informed architectural decisions.

Software architectures of third generation led to refinement and generalization of components. Software systems were benefited from codeless form of programming (plug and play). Code generators were invented. Many executable systems were generated automatically or semi-automatically directly from architectural descriptions. It was achieved through sophisticated configuration of components, connectors and its constraints. This paved a way to digitization and automation of business processes.

VI. Current Architectural Challenges

New technologies in 21st century is giving rise to disruptive services, products and business models. Cutting edge technologies like Additive Manufacturing, Robotics, IIoT and Cyber Physical Systems are posing new challenges. Cloud computing, Agile Technologies, Analytics and Automation are driving the business [23]. Aspects like compliance with security standards, adherence to data privacy policy and intelligent analytics guide the 'intelligent actions' within the applications developed. User Experience defines customer satisfaction directing to business success.

Typical challenges that must be addressed by Software Architects include:

- Shorter delivery cycle – time to market.
- Collaboration with development and operation teams high level stakeholders – the stakeholders must be involved.
- Large and complex testing environment due to huge amount of unstructured data spread across networked artefacts.
- Securing collaborative systems with diversified culture – architects may have to think about new/alternative security tactics as and when existing solutions get compromised.

Doman Specific Issues:

- Cyber-Physical Systems (CPS) are more vulnerable to threats than ever before. There is a need for closely monitoring and synchronizing data/information between physical factory floor and cyber computational space [24].
- Systematically deployed CPS envisages the ability to perform efficiently, collaboratively and resiliently. Software architecture should cater for all such quality attributes with deliberate trade-off among them.
- Complex systems such as digital/intelligent enterprises integrate cloud computing with various wearable devices and IoT. They exhibit dynamic features such as interoperability, cross-platform functionality and dynamic organization of system structures. Since modern applications are multi-disciplinary and spread across variety of networked machines, segregation and categorization of styles and processes is a tedious task.
- Expectations are more in terms of performance, availability and scalability even for hand held devices such as smartphones in spite of their battery and processing limitations. Identifying relevant activities and integrating them into a process to achieve desired solution is highly challenging in such environments.

VII. Recommended Solution at Software Architectural Level

There is need to deliver business values as fast as it is demanded. Old approaches adapted by organizations are posing risk of losing competitive advantages.

It is need of the hour to refocus all our intelligence and efforts on constantly evolving existing architectures making suitable changes in their design and enable them to respond to

the changing requirements. Software architecture must integrate data and functionality in a better way to enable data availability and collaboration.

All above mentioned quality attributes may be achieved by tuning existing design to support following enablers in Software Architectures [25]:

- Agility – having characteristics of speed and coordination, ability to react quickly and appropriately to change.
- Velocity – fast development through automation techniques and tools.
- Modularity – building self-contained entities and integrating them based on needs in different ways.
- Self-Healing System – to monitor and repair themselves. This enables systems to be responsive, resilient, elastic and message driven [26].
- Supporting Elastic Test Infrastructure – for rapid deployment and scaling. This improves adaptability in systems like data centres by swiftly adding or removing load balancing and other application resources [27].

After discussing variety of Architectural Styles and their evolution towards fulfilling the needs of changing industrial trends, we suggest **Software Architects** to focus on following thumb rules in order to suit current scenarios:

- Constantly evolve/enhance existing architectures through Artificial Intelligence, advanced analytical and automation techniques.
- Effectively communicate converged roles of stakeholders and enhance their skills through understanding Software Architectures.
- Design Architectural Structures that are loosely coupled and provide extension points to add new components safely.

In order for software intensive systems to cater for increasing demands of the present and upcoming industrial trends Software Architectures must be built to:

- Enable components to manipulate data on itself or its execution environment or state.
  - Self-Organizing Architectures supporting Reflective Programs [28], [29] are best example that adjusts their behaviour based on their context. In other words, Architectural reflection performs computation about its own software architectures making them self-representative. This enables them to be explicitly observed and manipulated [30].
- Support reusability at all phases of system development life cycle.
  - For example, providing means to reuse named functionality to operate on different types across diversified products.
- Model an organizational structure tailored to a family of applications, such as avionics, command and control, or vehicle management systems. These specialized architectures enhance the descriptive power of structures by clearly defining technological, environmental and business constraints on them.
  - Reference Architectures for specific domains [31] are best examples of this category which make it possible to generate executable systems automatically or semi-automatically directly from architectural descriptions.

VIII. Conclusion and Future Work

The work highlights major aspects in software architecture evolution due to change in technology, business capabilities and system requirements. Insights on past and present architectural perceptions give directions for continued research related to qualitative measurement of software systems. Software issues are identified from basic stand-alone systems to collaborative systems built across various domains. Relevant examples are quoted in order to enable readers to map specific architectural styles to address their problems. Reusability at architectural level requires better understanding of architectural patterns (code reuse is already extensive; e.g., opensource). Hence, the review emphasizes on choice of architectural patterns based on several scenarios and provides recommendations for building sustainable software architecture.

It is evident that all major institutions and industries are heading towards researching new possibilities in software architecture. This justifies that all aspects of architectural activities are inevitable and must be explicitly dealt as a separate phase in SDLC. There is a need to build and evaluate software architectures in an organized manner so that every aspect of architecture and design are reused, systems are built cost-effectively and focus on time-to-market.

References

[1] Shaw M., Garlan D, “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, 1996, pp. 10-68.

[2] E. Eide, A. Reid, J. Regehr and J. Lepreau, “Static and Dynamic structure in design patterns”, Proceedings of the 24th International Conference on Software Engineering (ICSE’02), IEEE, pp. 208-218.

[3] Srikanth Narasimhan, Jagadish Chundury, “Enterprise Digitization Patterns: Designing, Building and Deploying Enterprise Digital Solutions”, Notion Press, Incorporated, 2018, pp. 246.

[4] Holmes, Benedikt, and Ana Nicolaescu. "Continuous architecting: Just another buzzword." Full-scale Software Engineering/The Art of Software Testing (2017): 1.

[5] http://www.sei.cmu.edu/architecture/start/glossary/bibliographicdefs.cfm . (website active as on date of submission).

[6] Bass, Len, Paul Clements, and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, 2003.

[7] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. “Pattern-Oriented Software Architecture, Volume 1: A System of Patterns”, 1996.

[8] Cloutier, Robert, Gerrit Muller, Dinesh Verma, Roshanak Nilchiani, Eirik Hole, and Mary Bone. "The concept of reference architectures." Systems Engineering 13, no. 1, 2010, pp.14-27.

[9] Lago, Patricia, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Antony Tang. "The road ahead for architectural languages." IEEE Software 32, no. 1, 2015, pp. 98-105.

[10] Estefan, J. A., K. Laskey, F. G. McCabe, and D. Thornton. "OASIS Reference Architecture for Service Oriented Architecture." Version 1.0, OASIS Public Review Draft 1, 2008.

[11] Lankhorst, Marc. “Enterprise architecture at work: Modelling, communication and analysis.” Springer Science & Business Media, 2009.

[12] R. Lyons, "A Total AUTODIN System Architecture," in IEEE Transactions on Communications, vol. 28, no. 9, Sep 1980, pp. 1467-1471.

[13] Leiner, Barry, Robert Cole, Jon Postel, and David Mills. "The DARPA Internet protocol suite." IEEE Communications Magazine 23, no. 3 (1985): 29-34.

[14] Bellman, Beryl, and Felix Rausch. "Enterprise architecture for e-government." In International Conference on Electronic Government, pp. 48-56. Springer, Berlin, Heidelberg, 2004.

[15] Anton Jansen, Jan Bosch, "Software Architecture as a Set of Architectural Design Decisions", WICSA, 2005, Software Architecture, Working IEEE/IFIP Conference on, Software Architecture, Working IEEE/IFIP Conference on 2005, pp. 109-120.

[16] Astrom, Karl J, B. Wittenmark, “Computer-Controlled Systems Design”. Prentice Hall, second ed., 1990.

[17] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” Science of Computer Programming, vol. 8, pp. 1987, 231-274.

[18] Batory, Don. "Product-line architectures." In Smalltalk and Java Conference. 1998.

[19] Binns, Pam, Matt Englehart, Mike Jackson, and Steve Vestal. "Domain-specific software architectures for guidance, navigation and control." International Journal of Software Engineering and Knowledge Engineering 6, no. 02, 1996, pp. 201-227.

[20] Medvidovic, Nenad, and Richard N. Taylor. "Software architecture: foundations, theory, and practice." In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, pp. 471-472. ACM, 2010.

[21] Rosen, Michael, Boris Lublinsky, Kevin T. Smith, and Marc J. Balcer. Applied SOA: service-oriented architecture and design strategies. John Wiley & Sons, 2012.

[22] Hasselbring, Wilhelm, and Guido Steinacker. "Microservice architectures for scalability, agility and reliability in e-commerce." In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 243-246. IEEE, 2017.

[23] Sniderman, Brenna, Monika Mahto, and Mark J. Cotteleer. "Industry 4.0 and manufacturing ecosystems: Exploring the world of connected enterprises." Deloitte Consulting, 2016.

[24] Lee, Jay, Behrad Bagheri, and Hung-An Kao. "A cyber-physical systems architecture for industry 4.0-based manufacturing systems." Manufacturing Letters 3, 2015, pp. 18-23.

[25] Richard Monson-Haefel, “97 things every software architect should know: collective wisdom from the experts”. O'Reilly, 2009.

[26] Pepper, Joël, and Ana Nicolaescu. "A Look at the Evolution of Software Architecture Evolution since 2010." Full-scale Software Engineering/The Art of Software Testing (2017): 25.

[27] Gambi, Alessio, Waldemar Hummer, and Schahram Dustdar. "Automated testing of cloud-based elastic systems with AUToCLES." In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 714-717. IEEE Press, 2013.

[28] Affonso, Frank José, and Elisa Yumi Nakagawa. "A reference architecture based on reflection for self-adaptive software." In 2013 VII Brazilian Symposium on Software Components, Architectures and Reuse, pp. 129-138. Ieee, 2013.

[29] Garlan, David, Bradley Schmerl, and Shang-Wen Cheng. "Software Architecture-Based Self-Adaptation." In Autonomic computing and networking, pp. 31-55. Springer, Boston, MA, 2009.

[30] Tisato, Francesco, Andrea Savigni, Walter Cazzola, and Andrea Sosio. "Architectural reflection realising software architectures via reflective activities." In Engineering Distributed Objects, pp. 102-115. Springer, Berlin, Heidelberg, 2001.

[31] E. Mettala and M. H. Graham, eds., “The Domain-Specific Software Architecture” Program. No. CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[32] ChristophRieger, Tim A.Majchrzak, “Towards the definitive evaluation framework for cross-platform app development approaches”, Journal of Systems and Software Volume 153, July 2019, Pages 175-199.

[33] Fowler, Martin. Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc., 2002.

[34] Hasselbring, Wilhelm. "Software architecture: past, present, future." The Essence of Software Engineering. Springer, Cham, 2018. 169-184.