Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 5.258

IJCSMC, Vol. 5, Issue. 3, March 2016, pg.801 – 815

SOFTWARE ENGINEERING: ARCHITECTURE, DESIGN AND FRAMEWORKS

Mohammad Imran
Department of Computer Science, College of Science and Humanities, Shaqra University, Al-Dawadmi, KSA

Dr. Abdulrahman A. Alghamdi
College of Computing and Information Technology Shaqra University, Shaqra, KSA

Bilal Ahmad
College of Computing and Information Technology, Shaqra University, Shaqra, KSA

Abstract— *Software architecture is the high level structure of a software system, the discipline of creating such a high level structure, and the documentation of this structure. The architecture of a software system is a metaphor, analogous to the architecture of a building.*
Documenting software architecture facilitates communication between stakeholders, captures early decisions about the high-level design, and allows reuse of design components between projects.
This unit builds on introductory units to analysis and design. It provides the professional software engineer with advanced knowledge and skills in high-level architectural design, its theoretical foundations, industrial best practice, and relevant application context. In the software life-cycle, software architecture sits between analysis/specification and design/implementation. The field of software architecture has come of age with a thriving research community and numerous high-level models, methods, tools and practices widely used in industry.

Keywords— *Computer architecture, Systems architecture and Enterprise architecture, Aspect-oriented software development,, Design rationale, Interaction design, Icon design, Search-based software engineering, Software Design Description (IEEE 1016),Software development, User experience, User interface design.*

I. INTRODUCTION

The term *software architecture* was first used in the late 1960s , but became prevalent only in the beginning of the 1990s. The field of computer science had encountered problems associated with complexity since its formation. Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns. Although the term “software architecture” is relatively new to the industry, the fundamental principles of the field have been applied sporadically

by software engineering pioneers since the mid-1980s. Early attempts to capture and explain software architecture of a system were imprecise and disorganized, often characterized by a set of box-and-line diagrams.

Software architecture as a concept has its origins in the research of Edsger Dijkstra in 1968 and David Parnas in the early 1970s. These scientists emphasized that the structure of a software system matters and getting the structure right is critical. During the 1990s there was a concerted effort to define and codify fundamental aspects of the discipline, with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods.

Research institutions have played a prominent role in furthering software architecture as a discipline. Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled *Software Architecture: Perspectives on an Emerging Discipline* in 1996, which promoted software architecture concepts such as components, connectors, and styles. The University of California, Irvine's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

While in IEEE 1471, software architecture was about the architecture of "software-intensive systems", defined as "any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole", the 2011 edition goes a step further by including the ISO/IEC 15288 and ISO/IEC 12207 definitions of a system, which embrace not only hardware and software, but also "humans, processes, procedures, facilities, materials and naturally occurring entities". This reflects the relationship between software architecture, Enterprise Architecture and Solution Architecture.

II. ARCHITECTURE

To Pree, software frameworks consist of *frozen spots* and *hot spots*. *Frozen spots* define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework. *Hot spots* represent those parts where the programmers using the framework add their own code to add the functionality specific to their own project.

In an object-oriented environment, a framework consists of abstract and concrete classes. Instantiation of such a framework consists of composing and subclassing the existing classes.

When developing a concrete software system with a software framework, developers utilize the hot spots according to the specific needs and requirements of the system. Software frameworks rely on the Hollywood Principle: "Don't call us, we'll call you." This means that the user-defined classes (for example, new subclasses), receive messages from the predefined framework classes. Developers usually handle this by implementing superclass abstract methods.

A. Architecture activities:

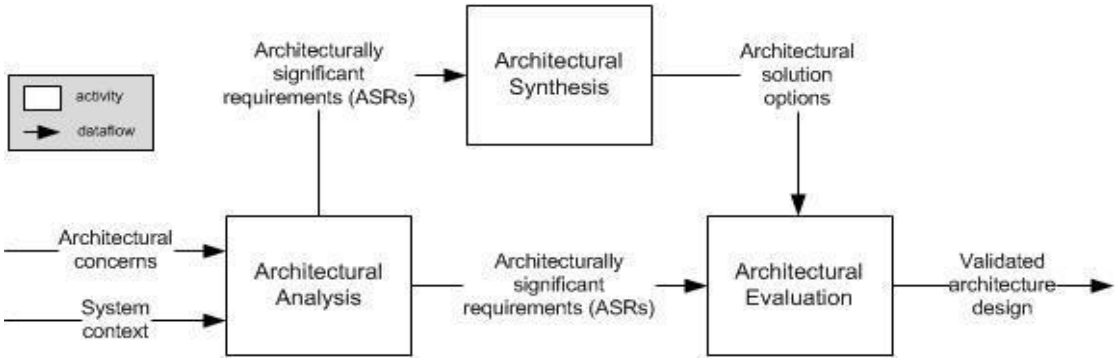

Fig : Three core activities of software architecting are depicted in the boxes.

There are many activities that a software architect performs. A software architect typically works with project managers, discusses architecturally significant requirements with stakeholders, designs software architecture, evaluates a design, communicates with designers and stakeholders, documents the architectural design and more. There are four core activities in software architecture design. These core architecture activities are performed iteratively and at different stages of the initial software development life-cycle, as well as over the evolution of a system.

Architectural Analysis is the process of understanding the environment in which a proposed system or systems will operate and determining the requirements for the system. The input or requirements to the analysis activity can come from any number of stakeholders and include items such as:

1. what the system will do when operational (the functional requirements)
2. How well the system will perform runtime non-functional requirements such as reliability, operability, performance efficiency, security, compatibility defined in ISO/IEC 25010:2011 standard .
3. development-time non-functional requirements such as maintainability and transferability defined in ISO 25010:2011 standard
4. business requirements and environmental contexts of a system that may change over time, such as legal, social, financial, competitive, and technology concerns

Architectural Synthesis or design is the process of creating architecture. Given the requirements determined by the analysis, the current state of the design and the results of any evaluation activities, the design is created and improved.

Architecture Evaluation is the process of determining how well the current design or a portion of it satisfies the requirements derived during analysis. An evaluation can occur whenever an architect is considering a design decision, it can occur after some portion of the design has been completed, it can occur after the final design has been completed or it can occur after the system has been constructed. Some of the available software architecture evaluation techniques include *Architecture Tradeoff Analysis Method (ATAM)* and TARA. Frameworks for comparing the techniques are discussed in frameworks such as SARA Report and Architecture reviews: practice and experience.

Architecture Evolution is the process of maintaining and adapting existing software architecture to meet requirement and environmental changes. As software architecture provides a fundamental structure of software system, its evolution and maintenance would necessarily impact its fundamental structure. As such, architecture evolution is concerned with adding new functionality as well as maintaining existing functionality and system behavior.

Architecture requires critical supporting activities. These supporting activities take place throughout the core software architecture process. They include knowledge management and communication, design reasoning and decision making, and documentation.

Architecture supporting activities

Software architecture supporting activities are carried out during core software architecture activities. These supporting activities assist a software architect to carry out analysis, synthesis, evaluation and evolution. For instance, an architect has to gather knowledge, make decisions and document during the analysis phase.

1. **Knowledge Management and Communication** is the activity of exploring and managing knowledge that is essential to designing a software architecture. A software architect does not work in isolation. They get inputs, functional and non-functional requirements and design contexts, from various stakeholders; and provides outputs to stakeholders. Software architecture knowledge is often tacit and is retained in the heads of stakeholders. Software architecture knowledge management activity is about finding, communicating, and retaining knowledge. As software architecture design issues are intricate and interdependent, a knowledge gap in design reasoning can lead to incorrect software architecture design.

   Examples of knowledge management and communication activities include searching for design patterns, prototyping, asking experienced developers and architects, evaluating the designs of similar systems, sharing knowledge with other designers and stakeholders, and documenting experience.

2. **Design Reasoning and Decision Making** is the activity of evaluating design decisions. This activity is fundamental to all three core software architecture activities. It entails gathering and associating decision contexts, formulating design decision problems, finding solution options and evaluating tradeoffs before making decisions. This process occurs at different levels of decision granularity, while evaluating significant architectural requirements and software architecture decisions, and software architecture analysis, synthesis, and evaluation.

Examples of reasoning activities include understanding the impacts of a requirement or a design on quality attributes, questioning the issues that a design might cause, assessing possible solution options, and evaluating the tradeoffs between solutions.

3. **Documentation** is the activity of recording the design generated during the software architecture process. A system design is described using several views that frequently include a static view showing the code structure of the system, a dynamic view showing the actions of the system during execution, and a deployment view showing how a system is placed on hardware for execution. Kruchten's 4+1 view suggests a description of commonly used views for documenting software architecture; Documenting Software Architectures: Views and Beyond has descriptions of the kinds of notations that could be used within the view description.

Examples of documentation activities are writing a specification, recording a system design model, documenting a design rationale, developing a viewpoint, documenting views.

4. Software architecture description

Software architecture description involves the principles and practices of modeling and representing architectures, using mechanisms such as: architecture description languages, architecture viewpoints, and architecture frameworks.

5. Architecture description languages

An architecture description language (ADL) is any means of expression used to describe software architecture (ISO/IEC/IEEE 42010). Many special-purpose ADLs have been developed since the 1990s, including AADL (SAE standard), Wright (developed by Carnegie Mellon), Acme (developed by Carnegie Mellon), xADL (developed by UCI), Darwin(developed by Imperial College London), DAOP-ADL (developed by University of Málaga), and ByADL (University of L'Aquila, Italy).

B. Architecture viewpoints:

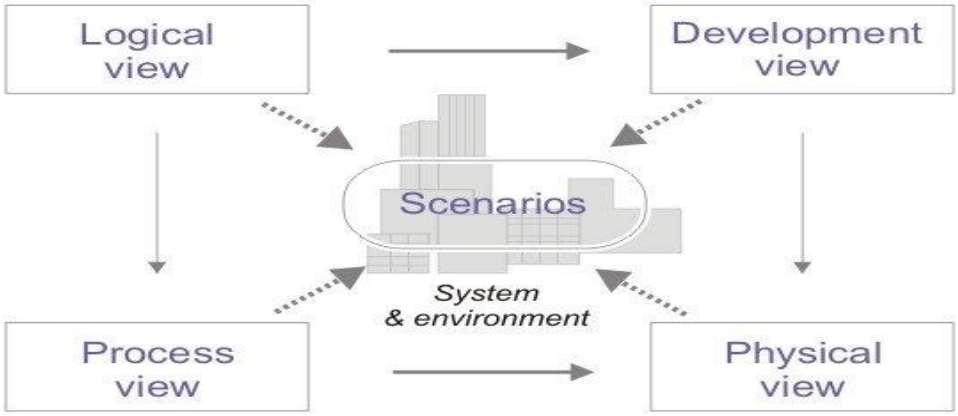

Fig 2: 4+1 Architectural View Model.

Software architecture descriptions are commonly organized into views, which are analogous to the different types of blueprints made in building architecture. Each view addresses a set of system concerns, following the conventions of its *viewpoint*, where a viewpoint is a specification that describes the notations, modeling and analysis techniques to use in a view that express the architecture in question from the perspective of a given set of stakeholders and their concerns (ISO/IEC/IEEE 42010). The viewpoint specifies not only the concerns framed (i.e., to be addressed) but the presentation, model kinds used, conventions used and any consistency (correspondence) rules to keep a view consistent with other views.

6. Architecture frameworks

An architecture framework captures the "conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders" (ISO/IEC/IEEE 42010). A framework is usually implemented in terms of one or more viewpoints or ADLs.

7. Architectural styles and patterns

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are often documented as software design patterns.

Following traditional building architecture, a 'software architectural style' is a specific method of construction, characterized by the features that make it notable" (Architectural style). "An architectural style defines: a family of systems in terms of a pattern of structural organization; a vocabulary of components and connectors, with constraints on how they can be combined." "Architectural styles are reusable 'packages' of design decisions and constraints that are applied to an architecture to induce chosen desirable qualities."

There are many recognized architectural patterns and styles, among them:

- Blackboard
- Client-server (2-tier, 3-tier, n-tier, cloud computing exhibit this style)
- Component-based
- Data-centric
- Event-driven (or Implicit invocation)
- Layered
- Monolithic application
- Peer-to-peer (P2P)
- Pipes and filters
- Plug-ins
- Representational state transfer (REST)
- Rule-based

- Service-oriented
- Shared nothing architecture
- Space-based architecture
- Persistence Free architecture

Some treat architectural patterns and architectural styles as the same, some treat styles as specializations of patterns. What they have in common is patterns and styles are idioms for architects to use, they "provide a common language" or "vocabulary" with which to describe classes of systems.

Software architecture and agile development

There are also concerns that software architecture leads to too much Big Design Up Front, especially among proponents of agile software development. A number of methods have been developed to balance the trade-offs of up-front design and agility. IEEE Software devoted a special issue to the interaction between agility and architecture.

Software architecture erosion

Software architecture erosion (or "decay") refers to the gap observed between the planned and actual architecture of a software system as realized in its implementation. Software architecture erosion occurs when implementation decisions either do not fully achieve the architecture-as-planned or otherwise violate constraints or principles of that architecture. The gap between planned and actual architectures is sometimes understood in terms of the notion of technical debt.

As an example, consider a strictly layered system, where each layer can only use services provided by the layer immediately below it. Any source code component that does not observe this constraint represents an architecture violation. If not corrected, such violations can transform the architecture into a monolithic block, with adverse effects on understandability, maintainability, and evolvability.

Various approaches have been proposed to address erosion. "These approaches, which include tools, techniques and processes, are primarily classified into three generic categories that attempt to minimize, prevent and repair architecture erosion. Within these broad categories, each approach is further broken down reflecting the high-level strategies adopted to tackle erosion. These are: process-oriented architecture conformance, architecture evolution management, architecture design enforcement, architecture to implementation linkage, self-adaptation and architecture restoration techniques consisting of recovery, discovery and reconciliation."

There are two major techniques to detect architectural violations: reflexion models and domain-specific languages. Reflexion model (RM) techniques compare a high-level model provided by the system's architects with the source code implementation. Examples of commercial RM-based tools include the Bauhaus Suite (developed by Axivion), SAVE(developed by Fraunhofer IESE) and Structure-101 (developed by Headway Software). There are also domain-specific languages with focus on specifying and checking architectural constraints, including .QL (developed by Semmle Limited) and DCL (from Federal University of Minas Gerais).

III.SOFTWARE ARCHITECTURE RECOVERY

Software architecture recovery (or reconstruction, or reverse engineering) includes the methods, techniques and processes to uncover a software system's architecture from available information, including its implementation and documentation. Architecture recovery is often necessary to make informed decisions in the face of obsolete or out-of-date documentation

and architecture erosion: implementation and maintenance decisions diverging from the envisioned architecture.

Design

Architecture is design but not all design is architectural. In practice, the architect is the one who draws the line between software architecture (architectural design) and detailed design (non-architectural design). There aren't rules or guidelines that fit all cases, although there have been attempts to formalize the distinction. According to the *Intension/Locality Hypothesis*, the distinction between architectural and detailed design is defined by the *Locality Criterion*, according to which a statement about software design is non-local (architectural) if and only if a program that satisfies it can be expanded into a program that does not. For example, the client–server style is architectural (strategic) because a program that is built on this principle can be expanded into a program that is not client–server—for example, by adding peer-to-peer nodes.

Software design:

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design. The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process. Davis [DAV95] suggests a set of principles for software design, which have been adapted and extended in the following list:

1. **The design process should not suffer from "tunnel vision."** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.
2. **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
3. **The design should not reinvent the wheel.** Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. **The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.** That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
5. **The design should exhibit uniformity and integration.** A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
6. **The design should be structured to accommodate change.** The design concepts discussed in the next section enable a design to achieve this principle.
7. **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Well- designed software should

never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

8. **Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
9. **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts and design measures are available to assist the designer in assessing quality.
10. **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

Design Concepts:

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

1. Abstraction - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.
2. Refinement - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.
3. Modularity - Software architecture is divided into components called modules.
4. Software Architecture - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. Good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule and cost.
5. Control Hierarchy - A program structure that represents the organization of a program component and implies a hierarchy of control.
6. Structural Partitioning - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.
7. Data Structure - It is a representation of the logical relationship among individual elements of data.
8. Software Procedure - It focuses on the processing of each modules individually
9. Information Hiding - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information

Design considerations

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

1. **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
2. **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
3. **Fault-tolerance** - The software is resistant to and able to recover from component failure.
4. **Maintainability** - A measure of how easily bug fixes or functional modifications can be accomplished. High maintainability can be the product of modularity and extensibility.
5. **Modularity** - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
6. **Reliability** - The software is able to perform a required function under stated conditions for a specified period of time.
7. **Reusability** - the software is able to add further features and modification with slight or no modification.
8. **Robustness** - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with resilience to low memory conditions.
9. **Security** - The software is able to withstand hostile acts and influences.
10. **Usability** - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.
11. **Performance** - The software performs its tasks within a user-acceptable time. The software does not consume too much memory.
12. **Portability** - The usability of the same software in different environments.
13. **Scalability** - The software adapts well to increasing data or number of users.

Modeling language

A *modeling language* is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual. Examples of graphical modeling languages for software design are:

- Business Process Modeling Notation (BPMN) is an example of a Process Modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across a number of layers.
- Flowchart is a schematic representation of an algorithm or a step-wise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.

- Jackson Structured Programming (JSP) is a method for structured programming based on correspondences between data stream structure and program structure
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Unified Modeling Language (UML) is a general modeling language to describe software both structurally and behaviorally. It has a graphical notation and allows for extension with a Profile (UML).
- Alloy (specification language) is a general purpose specification language for expressing complex structural constraints and behavior in a software system. It provides a concise language based on first-order relational logic.
- Systems Modeling Language (SysML) is a new general-purpose modeling language for systems engineering.

Design patterns

A software designer or architect may identify a design problem which has been solved by others before. A template or pattern describing a solution to a common problem is known as a design pattern. The reuse of such patterns can speed up the software development process, having been tested and proven in the past.

Usage

Software design documentation may be reviewed or presented to allow constraints, specifications and even requirements to be adjusted prior to computer programming. Redesign may occur after review of a programmed simulation or prototype. It is possible to design software in the process of programming, without a plan or requirement analysis,[6] but for more complex projects this would not be considered feasible. A separate design prior to programming allows for multidisciplinary designers and Subject Matter Experts (SMEs) to collaborate with highly skilled programmers for software that is both useful and technically sound. Making of robots is also a huge use of software design

Requirements Engineering

Requirements engineering and software architecture can be seen as complementary approaches: while software architecture targets the 'solution space' or the 'how', requirements engineering addresses the 'problem space' or the 'what'. Requirements engineering entails the elicitation, negotiation, specification, validation, documentation andmanagement of requir ements. Both requirements engineering and software architecture revolve around stakeholder concerns, needs and wishes.

There is considerable overlap between requirements engineering and software architecture, as evidenced for example by a study into five industrial software architecture methods that concludes that *"the inputs (goals, constrains, etc.) are usually ill-defined, and only get discovered or better understood as the architecture starts to emerge"* and that while *"most architectural concerns are expressed as requirements on the system, they can also include mandated design decisions"*. In short, the choice of required behavior given a particular problem impacts the architecture of the solution that addresses that problem, while at the same time the architectural design may impact the problem and introduce new requirements. Approaches such as the Twin Peaks model aim to exploit the synergistic relation between requirements and architecture.

Software framework:

A **software framework** is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. A software framework is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions. Software frameworks may include support programs, compilers, code libraries, tool sets, and application programming interfaces (APIs) that bring together all the different components to enable development of a project or solution.

Frameworks contain key distinguishing features that separate them from normal libraries:

- *inversion of control*: In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.[1]
- *default behavior*: A framework has a default behavior. This default behavior must be some useful behavior and not a series of no-ops.
- *extensibility*: A framework can be extended by the user usually by selective overriding or specialized by user code to provide specific functionality.
- *non-modifiable framework code*: The framework code, in general, is not supposed to be modified, while accepting user-implemented extensions. In other words, users can extend the framework, but should not modify its code.

The designers of software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time.[2] For example, a team using a web application framework to develop a banking web-site can focus on writing code particular to banking rather than the mechanics of request handling and state management.

Frameworks often add to the size of programs, a phenomenon termed "code bloat". Due to customer-demand driven applications needs, both competing and complementary frameworks sometimes end up in a product. Further, due to the complexity of their APIs, the intended reduction in overall development time may not be achieved due to the need to spend additional time learning to use the framework; this criticism is clearly valid when a special or new framework is first encountered by development staff. If such a framework is not used in subsequent job tasking, the time invested in learning the framework can cost more than purpose-written code familiar to the project's staff; many programmers keep copies of useful boilerplate for common needs.

However, once a framework is learned, future projects can be faster and easier to complete; the concept of a framework is to make a one-size-fits-all solution set, and with familiarity, code production should logically rise. There are no such claims made about the size of the code eventually bundled with the output product, nor its relative efficiency and conciseness. Using any library solution necessarily pulls in extras and unused extraneous assets unless the software is a compiler-object linker making a tight (small, wholly controlled, and specified) executable module.

The issue continues, but a decade-plus of industry experience has shown that the most effective frameworks turn out to be those that evolve from re-factoring the common code of the enterprise, instead of using a generic "one-size-fits-all" framework developed by third parties for general purposes. An example of that would be how the user interface in such an application package as an office suite grows to have common look, feel, and data-sharing attributes and methods, as the once disparate bundled applications grow unified into a suite

which is tighter and smaller; the newer/evolved suite can be a product that shares integral utility libraries and user interfaces.

This trend in the controversy brings up an important issue about frameworks. Creating a framework that is elegant, versus one that merely solves a problem, is still an art rather than a science. "Software elegance" implies clarity, conciseness, and little waste (extra or extraneous functionality, much of which is user defined). For those frameworks that generate code, for example, "elegance" would imply the creation of code that is clean and comprehensible to a reasonably knowledgeable programmer (and which is therefore readily modifiable), versus one that merely generates correct code. The elegance issue is why relatively few software frameworks have stood the test of time: the best frameworks have been able to evolve gracefully as the underlying technology on which they were built advanced. Even there, having evolved, many such packages will retain legacy capabilities bloating the final software as otherwise replaced methods have been retained in parallel with the newer methods.

Examples: Software frameworks typically contain considerable housekeeping and utility code in order to help bootstrap user applications, but generally focus on specific problem domains, such as:

- Artistic drawing, music composition, and mechanical CAD
- Compilers for different programming languages and target machines.
- Financial modeling applications
- Earth system modeling applications
- Decision support systems
- Multimedia framework - Media playback and authoring
- CSS framework
- Ajax framework / JavaScript framework
- Web application framework
- Middleware
- Cactus Framework - High performance scientific computing
- Application framework - General GUI applications
- Enterprise Architecture framework
- Oracle Application Development Framework

IV.CHARACTERISTICS

Multitude of stakeholders: software systems have to cater to a variety of stakeholders such as business managers, owners, users and operators. These stakeholders all have their own concerns with respect to the system. Balancing these concerns and demonstrating how they are addressed is part of designing the system. This implies that architecture involves dealing with a broad variety of concerns and stakeholders, and has a multidisciplinary nature.

Separation of concerns: the established way for architects to reduce complexity is by separating the concerns that drive the design. Architecture documentation shows that all stakeholder concerns are addressed by modeling and describing the architecture from separate points of view associated with the various stakeholder concerns. These separate descriptions are called architectural views (see e.g. the 4+1 Architectural View Model).

Quality-driven: classic software design approaches (e.g. Jackson Structured Programming) were driven by required functionality and the flow of data through the system, but the current insight[3] is that the architecture of a software system is more closely related to its quality

attributes such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and other such – utilities. Stakeholder concerns often translate into requirements on these quality attributes, which are variously called non-functional requirements, extra-functional requirements, system quality requirements or constraints.

Recurring styles: like building architecture, the software architecture discipline has developed standard ways to address recurring concerns. These "standard ways" are called by various names at various levels of abstraction. Common terms for recurring solutions are architectural style, strategy or tactic, *reference architecture and architectural pattern.*

Conceptual integrity: a term introduced by Fred Brooks in The Mythical Man-Month to denote the idea that the architecture of a software system represents an overall vision of what it should do and how it should do it. This vision should be separated from its implementation. The architect assumes the role of "keeper of the vision", making sure that additions to the system are in line with the architecture, hence preserving conceptual integrity.

Overall, macroscopic system structure; this refers to architecture as a higher level abstraction of a software system that consists of high-level components and connectors, as opposed to implementation details.

8. *The important stuff—whatever that is*; this refers to the fact that software architects should concern themselves with those decisions that have high impact on the system and its stakeholders—which may include apparently low-level details.
9. *That which is fundamental to understanding a system in its environment" Things that people perceive as hard to change*; since designing the architecture takes place at the beginning of a software system's lifecycle, the architect should focus on decisions that "have to" be right the first time, since reversing such decisions may be impossible or prohibitively expensive.
10. *A set of architectural design decisions*; software architecture should not be considered merely a set of models or structures, but should include the decisions that lead to these particular structures, and the rationale behind them. This insight has led to substantial research into software architecture knowledge management.

1. There is no sharp distinction between software architecture versus design and requirements engineering. They are all part of a "chain of intentionality" from high-level intentions to low-level details.

V. REFERENCES

Examples of reference items of different categories shown in the References section include:

- "ISO/IEC/IEEE 42010:2011 - Systems and software engineering - Architecture description". Iso.org. 2011-11-24. Retrieved 2013-08-06.
- "ISO/IEC/IEEE 42010: Conceptual Model". Iso-architecture.org. Retrieved 2013-08-06.
- "Survey of Architecture Frameworks". Iso-architecture.org. Retrieved 2013-08-06.
- David Emery and Rich Hilliard (2008-02-21). "Updating IEEE 1471: Architecture Frameworks and Other Topics". Ieeexplore.ieee.org. doi:10.1109/WICSA.2008.32. Retrieved 2013-08-06.

VI. CONCLUSIONS

Overall, macroscopic system structure; this refers to architecture as a higher level abstraction of a software system that consists of high-level components and connectors, as opposed to implementation details.

11. *The important stuff—whatever that is*; this refers to the fact that software architects should concern themselves with those decisions that have high impact on the system and its stakeholders—which may include apparently low-level details.
12. *That which is fundamental to understanding a system in its environment" Things that people perceive as hard to change*; since designing the architecture takes place at the beginning of a software system's lifecycle, the architect should focus on decisions that "have to" be right the first time, since reversing such decisions may be impossible or prohibitively expensive.
13. *A set of architectural design decisions*; software architecture should not be considered merely a set of models or structures, but should include the decisions that lead to these particular structures, and the rationale behind them. This insight has led to substantial research into software architecture knowledge management.
14. There is no sharp distinction between software architecture versus design and requirements engineering. They are all part of a "chain of intentionality" from high-level intentions to low-level details.

REFERENCES

1. *Clements, Paul; Felix Bachmann; Len Bass; David Garlan; James Ivers; Reed Little; Paulo Merson; Robert Nord; Judith Stafford (2010). Documenting Software Architectures: Views and Beyond, Second Edition. Boston: Addison-Wesley. ISBN 0-321-55268-7.*
2. *Bass, Len; Paul Clements; Rick Kazman (2012). Software Architecture In Practice, Third Edition. Boston: Addison-Wesley. pp. 21–24. ISBN 978-0321815736.*
3. *SEI (2006). "How do you define Software Architecture?". Retrieved 2012-09-12.*
4. *Garlan & Shaw (1994). "An Introduction to Software Architecture". Retrieved 2012-09-13.*
5. *Fowler, M. (2003). "Design - Who needs an architect?". IEEE Software **20** (5): 11–44.doi:10.1109/MS.2003.1231144. edit*
6. *ISO/IEC/IEEE 42010: Defining "architecture". Iso-architecture.org. Retrieved on 2013-07-21.*
7. *Jansen, A.; Bosch, J. (2005). "Software Architecture as a Set of Architectural Design Decisions". 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). p. 109. doi:10.1109/WICSA.2005.61.ISBN 0-7695-2548-2.*
8. *Ali Babar, Muhammad; Dingsoyr, Torgeir; Lago, Patricia; van Vliet, Hans (2009). Software Architecture Knowledge Management. Dordrecht Heidelberg London New York: Springer. ISBN 978-3-642-02373-6.*
9. *George Fairbanks (2010). Just Enough Software Architecture. Marshall & Brainerd.*
10. *ISO/IEC/IEEE (2011). "ISO/IEC/IEEE 42010:2011 Systems and software engineering -- Architecture description". Retrieved 2012-09-12.*
11. *SARA Work Group (2002). "SARA Report". Retrieved 14 September 2012.*
12. *P. Naur and B. Randell, Eds.,, ed. (1969). "Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968.". Brussels: NATO, Scientific Affairs Division,. Retrieved 2012-11-16.*
13. *P. Kruchten, H. Obbink & J. Stafford (2006). "The past, present and future of software architecture". Retrieved 2012-011-12.*
14. *University of Waterloo (2006). "A Very Brief History of Computer Science". Retrieved 2006-09-23.*
15. *IEEE Transactions on Software Engineering (2006)."Introduction to the Special Issue on Software Architecture". Retrieved 2006-09-23.*
16. *Garlan & Shaw (1994). "An Introduction to Software Architecture". Retrieved 2006-09-25.*
17. *Kruchten, P. (2008). "What do software architects really do?". Journal of Systems and Software **81** (12): 2413–2416. doi:10.1016/j.jss.2008.08.025. edit*

18. *Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, Pierre America (2007)."A general model of software architecture design derived from five industrial approaches".*
19. *Osterwalder and Pigneur (2004). An Ontology for e-Business Models. pp. 65–97.*
20. *Woods, E. (2012). "Industrial architectural assessment using TARA". Journal of Systems and Software **85** (9): 2034–2047. doi:10.1016/j.jss.2012.04.055. edit*
21. *Obbink, H.; Kruchten, P.; Kozaczynski, W.; Postema, H.; Ran, A.; Dominick, L.; Kazman, R.; Hilliard, R.; Tracz, W.; Kahane, E. (Feb 6, 2002). "Software Architecture Review and Assessment (SARA) Report". Retrieved October 8, 2012.*
22. *Maranzano, J. F.; Rozsypal, S. A.; Zimmerman, G. H.; Warnken, G. W.; Wirth, P. E.; Weiss, D. M. (2005). "Architecture Reviews: Practice and Experience". IEEE Software **22** (2): 34. doi:10.1109/MS.2005.28. edit*
23. *Babar, M.A.; Dingsøyr, T.; Lago, P.; Vliet, H. van (2009).Software Architecture Knowledge Management:Theory and Practice (eds.), First Edition. Springer. ISBN 978-3-642-02373-6.*
24. *Tang, A.; Han, J.; Vasa, R. (2009). "Software Architecture Design Reasoning: A Case for Improved Methodology Support". IEEE Software **26** (2): 43.doi:10.1109/MS.2009.46. edit*
25. *Kruchten, Philippe (1995, November). Architectural Blueprints — The "4+1" View Model of Software Architecture. IEEE Software 12 (6), pp. 42-50.*
26. *M. Shaw and D. Garlan, Software architecture: perspectives on an emerging discipline. Prentice Hall, 1996.*
27. *Riehle, Dirk (2000), Framework Design: A Role Modeling Approach, Swiss Federal Institute of Technology*
28. *"Framework". DocForge. Retrieved 15 December 2008.*
29. *Vlissides, J M; Linton, M A (1990), "Unidraw: a framework for building domain-specific graphical editors",ACM Transactions of Information Systems **8** (3): 237–268, doi: 10.1145/98188.98197*
30. *Johnson, R E (1992), "Documenting frameworks using patterns", Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (ACM Press): 63–76*
31. *Johnson, R E; McConnell, C; Lake, M J (1992), "The RTL system: a framework for code optimization", in Giegerich, R; Graham, S L, Proceedings of the International workshop on code generation (Springer-Verlag): 255–274*
32. *Birrer, A; Eggenschwiler, T (1993), "Proceedings of the European conference on object-oriented programming",Frameworks in the financial engineering domain: an experience report (Springer-Verlag): 21–35*
33. *Hill, C; DeLuca, C; Balaji, V; Suarez, M; da Silva, A (2004), "Architecture of the Earth System Modeling Framework (ESMF)", Computing in Science and Engineering: 18–28*

