OBJECT-ORIENTED FRAMEWORK FOR SPECIFIC ARCHITECTURE OF SOFTWARE

Dr. Narendra Kumar Sharma

Assistant Professor, Department of Computer Science and Engineering,
Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- narendra@sanskriti.edu.in

ABSTRACT

Architectural understanding has played a part in discussion on design, reuse, and adaptation for over a decade. The phrase has gained a lot of popularity in recent years, and efforts are being made to determine specific what is meant by architectural knowledge. The latest developments in architectural performance management are covered in this chapter. Following the results of a thorough literature study, we present four major perspectives on architectural knowledge. We describe major kinds of architectural knowledge and analyses four different outcomes for the business that have their roots in the abovementioned views, all of which are based on software architecture and knowledge organizational theory. State-of-the-art approaches take a more comprehensive stance and integrate various viewpoints in a single architectural knowledge management approach, in contrast with traditional approaches, which were limited to a single metaphysics when it came to tools, methods, and methodologies for architectonic performance management.

KEYWORDS: Computer Software, IoT, Optical Sensors, Sensors, Wireless Sensors.

INTRODUCTION

Object-oriented frameworks are application skeletons, which reflect the basic characteristics of a particular application domain. When developing applications from such a domain, it will probably be more efficient to use such a framework rather than to start from scratch. A framework is a kind of 'instant program', that sometimes even may be a complete, ready-to-run application, but it will normally allow you to customize its look and feel to your own taste. Object-oriented frameworks is an attempt to capture the common characteristics within a certain application domain, and make them available for reuse. Only those characteristics that are common are hardwired into the code. Therefore, users of a framework are still free to handicraft those parts that give their applications the individual touch. The first more commonly used framework was the Model-View-Controller framework found in the Smalltalk-80 user interface. It allowed users to connect different visual presentations to the state of a Model object. These Views were automatically notified each time the state was changed, and were able to ask the Model for the new values of the properties they were representing[1].

A change in the Model object were thus immediately reflected on the screen. Today, frameworks are considered a very promising technology for reifying proven software designs, targeting particular functionality's such that the user interfaces and operating systems and particular application domains such that fire-alarm systems and real time avionics. Frameworks like MacApp; ET++; Interviews; ACE; Microsoft's MFC and DCOM; JavaSoft's RMI, AWT and Beans; OMG's CORBA play an increasingly important role in contemporary software development. Early Frameworks were normally monolithic, i.e., object-oriented software architectures making up an entire application within some specific

domain, but later versions are also restricting themselves to various subsystems. Due to the fact that these smaller frameworks are serving the role as design elements, they may seem to coincide with the Design Pattern concept, as specified in. There is, however, an important difference between the two, because these smaller grained frameworks still contain executable code, while design patterns are merely codeless descriptions of how to implement certain features. In addition, patterns are more universal tool in the sense that they are normally not tied to a particular application domain[2], [3].

Domain-Specific Development Environment

A domain-specific development environment (DSDE) supports the application development based on a DSSA. A DSDE has its own architecture that usually has three levels.

i. Productivity Tools

On top of a formal component model, there are a number of tools that facilitate a convenient application development, e.g., cogitation editors, semantic checkers, component repositories, generators, etc. An important tool is the constraint checker. Possible approaches to checking design constraints include attribute grammars, temporal logic, and a special type of first order logic.

ii. Formal Component Model

The formal component model is defined through the reference architecture and lies at the heart of a DSDE. The mapping of an application architecture onto the underlying layer is done by a generator. One has to decide whether to use compositional or transformational generator technology.

iii. Support Frameworks

Support frameworks implement the application component model. Both the frameworks and the reference architecture could be developed at the same time on an evolutionary basis. Support frameworks could already be portable, which would simplify the generation process. A critical aspect of the design for any large software system is its gross structure represented as a high-level organization of computational elements and interactions between those elements. Broadly speaking, this is the software architectural level of design. The structure of software has long been recognized as an important issue of concern. However, recently software architecture has begun to emerge as an explicit field of study for software engineering practitioners and researchers. Evidence of this trend is apparent in a large body of recent work in areas such as module interface languages, domain specific architectures, architectural description languages, design patterns and handbooks, formal underpinnings for architectural design, and architectural design environments.

What exactly do we mean by the term software architecture? As one might expect of a field that has only recently emerged as an explicit focus for research and development, there is currently no universally-accepted definition. Moreover, if we look at the common uses of the term architecture in software, we find that it is used in quite different ways, often making it difficult to understand what aspect is being addressed. Among the various uses are is that the architecture of a particular system, as in the architecture of this system consists of the following components and an architectural style, as in this system adopts a client-server

architecture and the general study of architecture, as in \the papers in this journal are about architecture.

As definitions go, this is not a bad starting point. But definitions such as this tell only a small part of the story. More important than such explicit definitions, is the locus of effort in research and development that implicitly has come to define the field of software architecture. To clarify the nature of this effort it is helpful to observe that the recent emergence of interest in software architecture has been prompted by two distinct trends. The first is the recognition that over the years designers have begun to develop a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems[4], [5].

For example, the box and line diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a pipeline," a blackboard-oriented design or a client-server system. Although these terms are rarely assigned precise definitions, they permit designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provide significant semantic content that informs others about the kinds of properties that the system will have: the expected paths of evolution, its overall computational paradigm, and its relationship to similar systems.

The second trend is the concern with exploiting specific domains to provide reusable frameworks for product families. Such exploitation is based on the idea that common aspects of a collection of related systems can be extracted so that each new system can be built at relatively low cost by instantiating the shared design. Familiar examples include the standard decomposition of a compiler which permits undergraduates to construct a new compiler in a semester, standardized communication protocols which allow vendors to interoperate by providing services at different layers of abstraction, fourth-generation languages which exploit the common patterns of business information processing, and user interface toolkits and frame- works which provide both a reusable framework for developing interfaces and sets of reusable components, such as menus, and dialogue boxes.

Generalizing from these trends, it is possible to identify four salient distinctions:

i. Focus of Concern

The first distinction is between traditional concerns about design of algorithms and data structures, on the one hand, and architectural concerns about the organization of a large system, on the other. The former has been the traditional focus of much of computer science, while the latter is emerging as a significant and different design level that requires its own notations, theories, and tools. In particular, software architectural design is concerned less with the algorithms and data structures used within modules than with issues such as gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

ii. Nature of Representation

The second distinction is between system description based on definition use structure and architectural description based on graphs of interacting components. The former modularizes a system in terms of source code, usually making explicit the dependencies between use sites of the code and corresponding definition sites. The latter modularizes a system as a graph, or

configuration, of components and connectors. Components define the application-level computations and data stores of a system. Examples include clients, servers, filters, databases, and objects. Connectors define the interactions between those components. These interactions can be as simple as procedure calls, pipes, and event broadcast, or much more complex, including client-server protocols, database accessing protocols, etc.

iii. Instance Versus Style

The third distinction is between architectural instance and architectural style. An architectural instance refers to the architecture of a specific system. Box and line diagrams that accompany system documentation describe architectural instances, since they apply to individual systems. An architectural style, however, defines constraints on the form and structure of a family of architectural instances. For example, a pipe and filter architectural style might define the family of system architectures that are constructed as a graph of incremental stream transformers. Architectural styles prescribe such things as a vocabulary of components and connectors (for example, filters and pipes), topological constraints (for example, the graph must be acyclic), and semantic constraints (for example, filters cannot share state). Styles range from abstract architectural patterns and idioms (such as \client-server" or \layered" organizations), to concrete \reference architectures" (such as the ISO OSI communication model or the traditional linear decomposition of a compiler).

iv. Design Methods versus Architectures

A fourth distinction is between software design methods such as object-oriented design, structured analysis, and JSD and software [6], [7]architecture. Although both design methods and architectures are concerned with the problem of bridging the gap between requirements and implementations, there is a significant difference in their scopes of concern. Without either software design methods or a discipline of software architecture design, the implementer is typically left to develop a solution using whatever ad hoc techniques may be at hand. Design methods improve the situation by providing a path between some class of system requirements and some class of system implementations. Ideally, a design method defines each of the steps that take a system designer from the requirements to a solution. The extent to which such methods are successful often depends on their ability to exploit constraints on the class of problems they address and the class of solutions they provide. One of the ways they do this is to focus on certain styles of architectural design. For example, object-oriented methods usually lead to systems formed out of objects, while others may lead more naturally to systems with an emphasis on data flow. In contrast, the field of software architecture is concerned with the space of architectural designs. Within this space object-oriented and data ow structures are but two of the many possibilities. Architecture is concerned with the trade-offs between the choices in this space the properties of different architectural designs and their ability to solve certain kinds of problems. Thus design methods and architectures complement each other: behind most design methods are preferred architectural styles, and different architectural styles can lead to new design methods that exploit them.

LITERATURE REVIEW

D. Le et al. stated that the Object-oriented domain-driven design (DDD) aims to iteratively develop software around a realistic model of the application domain, which both thoroughly captures the domain requirements and is technically feasible for implementation. The main

focus of recent work in DDD has been on using a form of annotation-based domain specific language (aDSL), internal to an object-oriented programming language, to build the domain model. However, these works do not consider software modules as first-class objects and thus lack a method for their development. In this chapter, we tackle software module development with the DDD method by adopting a generative approach that uses aDSL. To achieve this, we first extend a previous work on module-based software architecture with three enhancements that make it amenable to generative development. We then treat module configurations as first-class objects and define an aDSL, named MCCL, to express module configuration classes. To improve productivity, we define function MCCGEN to automatically generate each configuration class from the module's domain class. We define our method as a refinement of an aDSL-based software development method from a previous work. We apply meta-modelling with UML/OCL to define MCCL and implement MCCL in a Java software framework. We evaluate the applicability of our method using a case study and formally define an evaluation framework for module generativist. We also analyse the correctness and performance of function MCCGEN. MCCL is an aDSL for module configurations. Our evaluation shows MCCL is applicable to complex problem domains. Further, the MCCs and software modules can be generated with a high and quantifiable degree of automation. Conclusion: Our method bridges an important gap in DDD with a software module development method that uses a novel aDSL with a module-based software architecture and a generative technique for module configuration[8], [9].

B. Alshemaimri et al. stated that the Database code fragments exist in software systems by using Structured Query Language (SQL) as the standard language for relational databases. Traditionally, developers bind databases as back ends to software systems for supporting user applications. However, these bindings are low-level code and implemented to persist user data, so Object Relational Mapping (ORM) frameworks take place to database access details. Both approaches are prone to problematic database code fragments that negatively impact the quality of software systems. We survey problematic database code fragments in the literature and examine antipatterns that occur in low-level database access code using SQL and high-level counterparts ORM frameworks. We also study problematic database code fragments in different and popular software architectures such as Service-Oriented Architecture, Microservice Architecture, and Model View Controller. We create a novel categorization of both SQL schema and query antipatterns in terms of performance, maintainability, portability, and data integrity. This article reviews database antipatterns including SQL antipatterns and framework-specific antipatterns in terms of their impact on nonfunctional requirements such as performance, maintainability, portability, and data integrity.

M. Ghareb et al. stated that explores a new framework for calculating hybrid system metrics using software quality metrics aspect-oriented and object-oriented programming. Software metrics for qualitative and quantitative measurement is a mix of static and dynamic software metrics. It is noticed from the literature survey that to date, most of the architecture considered only the evaluation focused on static metrics for aspect-oriented applications. In our work, we mainly discussed the collection of static parameters, long with AspectJ-specific dynamic software metrics. The structure may provide a new direction for research while predicting software attributes because earlier dynamic metrics were ignored when evaluating quality attributes such as maintainability, reliability, and understandability of Asepect Oriented software. Dynamic metrics based on the fundamentals of software engineering are

equally crucial for software analysis as are static metrics. A similar concept is borrowed with the introduction of dynamic software metrics to implement aspect-riented software development. Currently, we only propose a structure and model using static and dynamic parameters to test the aspect-oriented method, but we still need to validate the proposed approach[10], [11].

M. Amor et al. illustrated that the production of maintainable and reusable agents depends largely on how well the agent architecture is modularized. Most commercial agent toolkits provide an Object-Oriented (OO) framework, whose agent architecture does not facilitate separate (re)use of the domain-specific functionality of an agent from other concerns. This paper presents Mala, an agent architecture that combines the use of Component-based Software Engineering and Aspect-Oriented Software Development, both of which promote better modularization of the agent architecture while increase at the architectural level. Malaca supports the separate (re)use of the domain-specific functionality of an agent from other communication concerns, providing explicit support for the design and configuration of agent architectures and allows the development of agent-based software so that it is easy to understand, maintain and reuse.

R. Taylor et al. stated that the objective of software development using domain-specific software architectures (DSSA) is reduction in time and cost of producing specific application systems within a supported domain, along with increased product quality, improved manageability, and positioning for acquisition of future business. Key aspects of the approach include software reuse based on parameterization of generic components and interconnection of components within a canonical solution framework. Viability of the approach depends on identification and deep understanding of a selected domain of applications. The DSSA approach, to be effectively applied, requires a variety of support tools, including repository mechanisms, prototyping facilities, and analysis tools. This curriculum module describes the DSSA approach, representative examples, supporting tools, and processes.

B. Belhomme et al. illustrated that the completely new ray tracing software has been developed at the German Aerospace Center. The main purpose of this software is the flux density simulation of heliostat fields with a very high accuracy in a small amount of computation time. The software is primarily designed to process real sun shape distributions and real highly resolved heliostat geometry data, which means a data set of normal vectors of the entire reflecting surface of each heliostat in the field. Specific receiver and secondary concentrator models, as well as models of objects that are shadowing the heliostat field, can be implemented by the user and be linked to the simulation software subsequently. The specific architecture of the software enables the provision of other powerful simulation environments with precise flux density simulation data for the purpose of entire plant simulations. The software was validated through a severe comparison with measured flux density distributions. The simulation results show very good accordance with the measured results.

R. Tu et al. illustrated that the Virtual Enterprise model affords the valid instruction for rapid establishing and successful running of Virtual Enterprise. However, authors perceive that low quality and low efficiency are serious restriction factor to the development of Virtual Enterprise model. In order to overcome above-mentioned embarrassment in Virtual Enterprise modeling, authors put forward applying software reuse technology and Domain

Engineering theory to establishing the Domain Specific Software Architecture of Virtual Enterprise, then develop application system and establish the reusable component library in terms of Domain Specific Software Architecture of Virtual Enterprise. On the one hand, the quality and efficiency of modeling can be promoted remarkably. On the other hand, the model of Virtual Enterprise can be reused in the same domain.

J. Zhu et al. illustrated that the rapid development of technology, software is rapidly evolving with emerging applications. Chips that fail to adapt to software such that the application-specific integrated circuits, ASICs suffer from a short lifecycle and high nonrecurring engineering (NRE) costs. Meanwhile, as the projection of Moore's law and Dennard scaling are decreasing, energy efficiency has shown a diminishing return with new technologies. The computing capacity of general-purpose processors is limited due to power budgets. Consequently, future chips must jointly optimize flexibility, power efficiency, and ease of programmability. Reconfigurable chips combine the high flexibility of a general-purpose processor and high energy efficiency of ASIC by providing on-demand customization of their architectures. This article thoroughly reviews the development and architecture of reconfigurable chips. Moreover, the future challenges of reconfigurable chips are analyzed. Based on these challenges, future directions are also discussed.

B. Senyapj et al. ststed that the Interior architectural education and practice employ various general-purpose software packages. This study problematizes that as none of these packages is developed specifically for interior architectural design process and purposes, both interior architecture education and market seek ways to fulfill their specific needs. It is argued that currently interior architecture does not fully benefit from digital opportunities. A specific software package for interior architecture will enable the discipline to put forth its assets and manifest its existence. Consequently, this study proposes a domain specific model for interior architectural software. Initially, general-purpose and domain specific computer aided architectural design (CAAD) software used in interior architecture are determined. Then, selected software packages are analyzed according to Szalapaj's set of features: 'drawing', 'transformation', 'view', 'rendering' and 'other'. Based on these analyses, domain specific requirements for interior architecture are obtained. Consequently, questionnaires and interviews are performed with interior architectural students and professionals in order to determine the user needs. Finally, based on the findings, a software model for interior architecture is proposed.

A. Gopalakrishnan et al. illustrated that the Software Engineering has evolved over many years but stays human centric as it relies significantly on the technical decisions made by humans. Modeling the problem statement and arriving at the architecture and design revolves in the minds of software architects and designers. Many of the decisions stays in architect's minds and are only present in the models. The abstraction structures in software design are deeper than in other disciplines, since the final design is program code. This distinction leads to software architecture and design a highly interwoven process. The early design decisions are otherwise termed architectural decisions which compose software architecture. The architectural decisions are at an intermediate abstraction level with higher probability of reuse, but still not effectively reused even within the same organization. The most effective cases of reuse in software is with architecture patterns and design patterns. The paper points to the fact that patterns are successfully reused due to the quality of the descriptions which include problem, solution pair and supporting example. The paper focuses on intra-

organizational reuse, based on Domain Specific Software Architectures and the descriptions containing domain model, decision trees, architectural schema and rationale. It further tries to analyze three different use cases in the light of these elements and analyze if major hindrance of reuse is 'Rationale of decisions not well understood' than the commonly stated 'Not Invented here', supported with a survey of software engineers.

R. Weinreich et al. stated that the Software architecture is a central element during the whole software life cycle. Among other things, software architecture is used for communication and documentation, for design, for reasoning about important system properties, and as a blueprint for system implementation. This is expressed by the software architecture life cycle, which emphasizes architecture-related activities like architecture design, implementation, and analysis in the context of a software life cycle. While individual activities of the software architecture life cycle are supported very well, a seamless approach for supporting the whole life cycle is still missing. Such an approach requires the integration of disparate information, artifacts, and tools into one consistent information model and environment. In this article we present such an approach. It is based on a semi-formal architecture model, which is used in all activities of the architecture life cycle, and on a set of extensible and integrated tools supporting these activities. Such an integrated approach provides several benefits. Potentially redundant activities like the creation of multiple architecture descriptions are avoided, the captured information is always consistent and up-to-date, extensive tracing between different information is possible, and interleaving activities in incremental development and design are supported.

O. Pedreira et al. illustrated that the gamification has been applied in software engineering to improve quality and results by increasing people's motivation and engagement. A systematic mapping has identified research gaps in the field, one of them being the difficulty of creating an integrated gamified environment comprising all the tools of an organization, since most existing gamified tools are custom developments or prototypes. In this paper, we propose a gamification software architecture that allows us to transform the work environment of a software organization into an integrated gamified environment, i.e., the organization can maintain its tools, and the rewards obtained by the users for their actions in different tools will mount up. We developed a gamification engine based on our proposal, and we carried out a case study in which we applied it in a real software development company. The case study shows that the gamification engine has allowed the company to create a gamified workplace by integrating custom-developed tools and off-The-shelf tools such as Redmine, TestLink, or JUnit, with the gamification engine. Two main advantages can be highlighted: (i) our solution allows the organization to maintain its current tools, and (ii) the rewards for actions in any tool accumulate in a centralized gamified environment.

C. Venters et al. ststed that the Context Modern societies are highly dependent on complex, large-scale, software-intensive systems that increasingly operate within an environment of continuous availability, which is challenging to maintain and evolve in response to the inevitable changes in stakeholder goals and requirements of the system. Software architectures are the foundation of any software system and provide a mechanism for reasoning about core software quality requirements. Their sustainability the capacity to endure in changing environments is a critical concern for software architecture research and practice. Problem Accidental software complexity accrues both naturally and gradually over time as part of the overall software design and development process. From a software

architecture perspective, this allows several issues to overlap including, but not limited to: the accumulation of technical debt design decisions of individual components and systems leading to coupling and cohesion issues; the application of tacit architectural knowledge resulting in unsystematic and undocumented design decisions; architectural knowledge vaporization of design choices and the continued ability of the organization to understand the architecture of its systems; sustainability debt and the broader cumulative effects of flawed architectural design choices over time resulting in code smells, architectural brittleness, erosion, and drift, which ultimately lead to decay and software death. Sustainable software architectures are required to evolve over the entire lifecycle of the system from initial design inception to end-of-life to achieve efficient and effective maintenance and evolutionary change. Method This article outlines general principles and perspectives on sustainability with regards to software systems to provide a context and terminology for framing the discourse on software architectures and sustainability. Focusing on the capacity of software architectures and architectural design choices to endure over time, it highlights some of the recent research trends and approaches with regards to explicitly addressing sustainability in the context of software architectures. Contribution The principal aim of this article is to provide a foundation and roadmap of emerging research themes in the area of sustainable software architectures highlighting recent trends, and open issues and research challenges.

J. W. Kruize et al. stated that the smart farming is a management style that includes smart monitoring, planning and control of agricultural processes. This management style requires the use of a wide variety of software and hardware systems from multiple vendors. Adoption of smart farming is hampered because of a poor interoperability and data exchange between ICT components hindering integration. Software Ecosystems is a recent emerging concept in software engineering that addresses these integration challenges. Currently, several Software Ecosystems for farming are emerging. To guide and accelerate these developments, this paper provides a reference architecture for Farm Software Ecosystems. This reference architecture should be used to map, assess design and implement Farm Software Ecosystems. A key feature of this architecture is a particular configuration approach to connect ICT components developed by multiple vendors in a meaningful, feasible and coherent way. The reference architecture is evaluated by verification of the design with the requirements and by mapping two existing Farm Software Ecosystems using the Farm Software Ecosystem Reference Architecture. This mapping showed that the reference architecture provides insight into Farm Software Ecosystems as it can describe similarities and differences. A main conclusion is that the two existing Farm Software Ecosystems can improve configuration of different ICT components. Future research is needed to enhance configuration in Farm Software Ecosystems.

DISCUSSION

The three approaches that have been discussed in the previous sections, according to the criteria, use the same terminology, only the names of the terms change, showing the lack of a unified language. They share the fact of considering that the quality characteristics wanted or expected high-level quality characteristics in a software product must be defined and quantified measured in order to be assured. External and internal quality views are considered. The high-level characteristics, that may affect the exit or failure of the final system, cannot in general be directly measured. They must be “refined” in order to get the measurable aspects. Moreover, these measures are used to link or relate the low-level

characteristics, which are measurable, with the high-level characteristics. In this way, a trade-off to detect the dependencies among these characteristics is established. The definition of these links is always performed empirically or on the basis of experience. On the other hand, the approaches differ mostly on the stage of development where the quality model is applied. However, an important issue is that at design stage, all the approaches could be used. From our point of view, this stage is very important because it concerns the definition of the system architecture, characterized by non-functional properties. Nevertheless the ABAS approach, specific to this stage, does not offer any guideline. Finally, an important research issue is the extension of the software development methods that do not consider explicitly a quality model, with one of the three quality model approaches studied. Those offering guidelines should be better candidates, or the use of an extended ABAS with ISO 9126 or Dromey's design model. Moreover, since these approaches lack a common language, the specification of the quality models studied using notational standards, such as UML (Unified Modelling Language) should be considered. In UML is used to model architectures of real-time systems, where the selection of an architecture meeting precise quality requirements is crucial.

CONCLUSION

This paper presents an approach to integrate frameworks with domain specific languages (DSL). We argue that DSLs allows the domain expert to formalize the specification of a software solution immediately without worrying about implementation decisions and the framework complexity. The code for the variation points is specified in DSLs that are transformed (or compiled) to generate the framework instantiation code. During the transformation the framework instantiation restrictions may be verified. The case studies have shown that the proposed approach may enhance very much the instantiation process. It is important to note that DSLs can be transformed into other DSLs, thus creating a domain network, in a way similar to that described in, providing an easy implementation path for new DSLs. An approach for the derivation of the framework instantiation restrictions based on UML specifications is shown in, as well as tool support for the transformations. We are now working on a more elaborated version of the supporting environment, based on UML case tools and specific transformational systems.

REFERENCES

[1] M. Ozkaya en F. Erata, "A survey on the practical use of UML for different software architecture viewpoints", *Inf. Softw. Technol.*, vol 121, bl 106275, Mei 2020, doi: 10.1016/j.infsof.2020.106275.

[2] T. Gu, M. Lu, L. Li, en Q. Li, "An Approach to Analyze Vulnerability of Information Flow in Software Architecture", *Appl. Sci.*, vol 10, no 1, bl 393, Jan 2020, doi: 10.3390/app10010393.

[3] A. Baabad, H. B. Zulzalil, S. Hassan, en S. B. Baharom, "Software architecture degradation in open source software: A systematic literature review", *IEEE Access*. 2020. doi: 10.1109/ACCESS.2020.3024671.

[4] M. M. Soto-Cordova, S. León-Cárdenas, K. Huayhuas-Caripaza, en R. M. Sotomayor-Parian, "Proposal for a software architecture as a tool for the fight against corruption in the regional governments of Peru", *Int. J. Adv. Comput. Sci. Appl.*, 2020, doi: 10.14569/IJACSA.2020.0110786.

[5] S. Farshidi, S. Jansen, en J. M. van der Werf, "Capturing software architecture knowledge for pattern-driven design", *J. Syst. Softw.*, 2020, doi:

10.1016/j.jss.2020.110714.

[6] S. Moaven en J. Habibi, "A fuzzy-AHP-based approach to select software architecture based on quality attributes (FASSA)", *Knowl. Inf. Syst.*, 2020, doi: 10.1007/s10115-020-01496-7.

[7] *et al.*, "The Principle of Architecture First in Software Project Management Minimizes the Cost of Software Development Process: A Review", *Int. J. Innov. Technol. Explor. Eng.*, 2020, doi: 10.35940/ijitee.a8154.1110120.

[8] O. Sievi-Korte, I. Richardson, en S. Beecham, "Software architecture design in global software development: An empirical study", *J. Syst. Softw.*, 2019, doi: 10.1016/j.jss.2019.110400.

[9] J. Cruz-Benito, F. J. García-Peñalvo, en R. Therón, "Analyzing the software architectures supporting HCI/HMI processes through a systematic review of the literature", *Telemat. Informatics*, 2019, doi: 10.1016/j.tele.2018.09.006.

[10] Q. Q. G. Wuniri, X. P. Li, S. L. Ma, J. H. Lü, en S. Q. Zhang, "Modelling and Verification of High-order Typed Software Architecture and Case Study", *Ruan Jian Xue Bao/Journal Softw.*, 2019, doi: 10.13328/j.cnki.jos.005749.

[11] C. Ellwein, A. Elser, en O. Riedel, "Production planning and control systems - A new software architecture Connectivity in target", 2019. doi: 10.1016/j.procir.2019.02.089.