Position Paper: WSSSPE'2, Second Workshop on Sustainable Software for Science: Practice and Experiences, 2014

The Nebuchadnezzar Effect: Dreaming of Sustainable Software through Sustainable Software Architectures

Colin C. Venters, [2]Michael K. Griffiths, [1]Violeta Holmes, [1]Rupert R. Ward and [3]David J. Cooke

[1]School of Computing and Engineering
University of Huddersfield
Huddersfield, UK
{v.holmes; r.r.ward}@hud.ac.uk

[2]Corp. Information & Computing Services
University of Sheffield
Sheffield, UK
m.griffiths@sheffield.ac.uk

[3]School of Applied Sciences
University of Huddersfield
Huddersfield, UK
david.j.cooke@hud.ac.uk

Abstract—**Sustainability is emerging as an area of growing interest in the field of software engineering. While there is no agreed definition of the concept it is increasingly being considered as a non-functional requirement; a desired quality of a software system. One of the principal challenges in defining sustainability as a non-functional requirement is how to develop appropriate metrics and measures to demonstrate that the software is sustainable. Software architectures are the foundation of any software system and provide a mechanism for reasoning about quality attributes. However, architectural design is in part a creative process based on the level expertise of the software architect and tacit architectural knowledge, and is often made in an unsystematic and undocumented manner. Given the high dependency of non-functional requirements on their software architecture, this paper proposes that sustainable software architectures are fundamental to the development of technical sustainable software to address architectural drift and erosion, and architectural knowledge vaporization.**

Index Terms—**Architectural sustainability, non-functional requirements, software architectures, software quality, sustainability, software sustainability, technical sustainability.**

I. Introduction

Sustainability has been identified as an important future topic in the field of software engineering as software systems become increasingly more complex and operate in evolving, distributed eco-systems [1]. However, the concept of software sustainability is an elusive and ambiguous term with diametrical opposed views and interpretations [2]. As a result, there is a considerable amount of mystification and divergence regarding what software sustainability means, how it can be measured or demonstrated, and how to train and educate the broad spectrum of domain scientists or advance the skills of software engineers to develop software that is sustainable [3].

Change is inevitable [4]. It estimated that approximately 50–70% of the total lifecycle cost is spent on evolving a system [5]. If change is an inevitable feature of the software lifecycle this raises the question, *what is the most efficient and effective method or approach for managing change and evolution in terms of software's sustainability*?

The biblical tale of Nebuchadnezzar's dream relates to a statue composed of different types of metal built on a foundation of clay and iron [6]. However, iron and clay are materials that cannot bond to form a long-lasting foundation and will deteriorate overtime. The analogy of the feet of clay is now commonly used to refer to a weakness or flaw. The central thesis of this paper is that sustainable software should be built from a strong and solid foundation that allows efficient and effective maintenance and evolutionary change. To achieve this, this paper proposes that sustainable software architectures are fundamental to the development of technical sustainable software. The principle aim of this paper is to explore existing work to provide the theoretical foundation to support our thesis. Section 2 examines the concept of sustainability and its relationship to sustainable software. Section 3 examines the relationship between non-functional requirements, sustainability and software architectures. Section 4 explores emerging work in the area of developing sustainable software architectures. In Section 5, conclusions are drawn and future directions are outlined.

II. Sustainable Software

Before software sustainability can be measured as an attribute it must be understood [7]. The word *sustainability* is derived from the Latin *sustinere*. The Oxford English Dictionary [8] defines sustainability as '*the quality of being sustained*', where sustained can be defined as '*capable of being endured*' and '*capable of being 'maintained*'. This suggests that longevity and the ability to maintain are key factors at the heart of understanding sustainability.

As a part of the concept of sustainable development, the most widely adopted definition of sustainability was that proposed by the Brundtland commission [9], which was defined as '*meeting the needs of the present without compromising the ability of future generations to meet their own needs*'. The word *'need'* is central to this definition and includes a dimension of time, present and future. However, this definition is problematic for several reasons including that it is broad in its scope, it is open to interpretation, and it is difficult, if not impossible, to quantify.

In recent years, a triple bottom line perspective of sustainability has been adopted which considers sustainability to include three dimensions: environment, society and economy [10]. Environmental sustainability is concerned with minimizing the impact on the environment and natural resources. Social sustainability is concerned with building social equity. Economic sustainability is concerned with economic growth and its impact on social or natural resources.

It is argued that by addressing the three dimensions it can lead to more sustainable outcomes [11]. Goodland [12] and Penzenstadler and Femmer [13] extend these dimensions to include individual and technical sustainability where *individual sustainability* is '*the maintenance of the private good of individual human capital*' i.e. health, education etc., and *technical sustainability* is '*the long-term usage of systems and their adequate evolution with changing surrounding conditions and respective requirements'*. In addition, we can also consider the five dimensions in relation to three orders of impact or effects of software systems [10]:

- First-order: direct impacts created by the physical existence and the processes involved including design, production, distribution, maintenance and disposal;
- Second-order: indirect impacts created by ongoing usage;
- Third-order: systemic impacts aggregated over the medium to long term.

Koziolek [14] takes a literal interpretation of the concept of sustainability and suggests that sustainable software can be defined as '*a software-intensive system that operates for more than fifteen years*'. This is a position supported by Tamai and Torimitsu [15] who suggest that the average software lifetime is ten years, with a minimum of two years, and a maximum of thirty. In addition, they highlight that longevity requirements can be embedded in a number of domains where there are large [financial] investments. Koziolek [14] extends this definition where sustainable software is defined as *'a long-living software system which can be cost-efficiently maintained and evolved over its entire life-cycle'*. This definition suggests that maintainability and extensibility are key features of sustainability, which are tightly coupled with an economical dimension.

Similarly, a number of definitions have emerged from the field of software engineering, which focuses on the sustainability of the software artifact where maintainability and evolution are key factors of sustainability [7, 16-18]. In this context *maintainability* can be defined as '*the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment*' [19]. Maintainability is also related to *extendability* and *flexibility*, which the former is concerned with increasing storage or functional capacity, and the later is concerned with integration with other applications or environments respectively. The main objective of software evolution is ensuring *reliability* and *flexibility* of the system where the former is concerned with *functional performance* and is also related to *availability*. This strongly suggests that software sustainability is a composite attribute.

There has also been a parallel interest in defining sustainable software from a software development perspective, which is concerned with the broader direct and indirect impacts on the economy, society and the environment [20-21]. However, Fenner et. al., [22] argue that for sustainable engineering to be successful it requires a paradigm shift in thinking to embrace a holistic approach enshrined in the field of complex systems science.

A diametrically opposed position of an absolute definition of software sustainability is that it is simply an emergent property of a software system determined by market forces [23]. This suggests that sustainability cannot be designed or engineered and quantified until after the software system is operational. While it could be argued that software sustainability is an emergent property similar to safety [24], it is a highly probable that a software artifact that has endured over time will have at the very minimum been maintained [25]. This suggests that software sustainability is at the very minimum a measure of maintainability metrics such as the maintainability index, which can be derived from such measures as lines-of-code, McCabe or Halstead complexity [26]. In addition, the software artifact may have evolved or been ported on to different platforms. This strongly suggests that software sustainability is not an emergent property and is a composite attribute with a number of sub-characteristics related to maintenance and evolution.

Increasingly, software sustainability is being considered a first class, non-functional requirement [24]. In the field of software engineering, non-functional requirements or software quality attributes can be defined as '*the degree to which a system, component or process meets a stakeholders needs or expectations*' [19]. This aligns with the Brundtland [9] definition of sustainability addressing needs.

Without explicit reference to specific non-functional requirements, the GREENSOFT model proposed by Naumann et. al., [21] is designed to incorporate a range of non-functional requirements within the three categories of the sustainability criteria and metrics section of the reference model. This separation allows the examination of first-, second- and third-order impacts from an environmental perspective that result from effects of supply, effects of usage and systemic effects. However, they suggest that the fundamental question at the heart of the model is not, in which phase are metrics applied or in which phases are they taken in order to improve the quality attributes? The principal question is, *in which life cycle phase can the related effects be observed*?

Venters et. al., [27-28] defined software sustainability as a composite, non-functional requirement which is '*a measure of a systems extensibility, interoperability, maintainability, portability, reusability, scalability, and usability'*. Several of the metrics are directly related to the concept of evolution of the software system. The rationale for including usability as a metric of sustainability is that it is directly related to perceived usefulness from a stakeholder's perspective and thereby aligns sustainability with the issue of need. In addition, several of the quality attributes specify the '*effort required*' to achieve a particular outcome. This suggests that the concept of sustainability is strongly coupled to other quality attributes such as energy and cost efficiency, and resource utilization over the software's entire lifetime and aligns with the dimensions of environmental and economic sustainability.

Defining software sustainability as a composite, non-functional requirement is also a position supported by Calero, Bertoa, and Moraga [29] who suggest that software sustainability is related to a number of the main quality attributes and their sub-characteristics defined in ISO/IEC 25010 [30]; the standard has eight product quality characteristics and thirty one sub-characteristics. However, they suggest that sustainability can be considered from two perspectives: *energy efficiency* and *perdurability*. Energy efficiency is related to consumption and resource optimization, which aligns to the dimension of environmental sustainability. Based on the sub-characteristics of *reusability*, *modifiability*, and *adaptability* they define *perdurability* as the '*degree to which a software product can be modified, adapted and reused in order to perform specified functions under specified conditions for a long period of time*'. However, this significantly narrows the view of software sustainability as a composite, non-functional requirement and potentially eliminates important software quality attributes related to software evolution. Similarly, it is not clear why attempting to redefine software sustainability in terms of its perdurability i.e. very durable, is different from the overall aim of making software sustainable, at least in terms of the artifact, as the basic definition of sustainability is underpinned by the idea of enduring. Similarly, Koziolek et. al., [31] define software sustainability from a maintainability perspective in terms of its modifiability, reusability, modularity and testability. These definitions of software sustainability strongly suggest that it can be categorized as a composite, non-functional requirement. At the very minimum, technical software sustainability should address two core quality attributes including appropriate sub-characteristics: maintainability and extendibility. Nevertheless, maintainability as defined by ISO/IEC 25010 has new sub-characteristics of modularity, reusability, and modifiability, which address the issue of software evolution. As a result, what metrics and measures are suitable to demonstrate software sustainability is an open research problem.

However, one of the principal challenges in defining software sustainability as a non-functional requirement is how to demonstrate that the quality factors have been addressed in a quantifiable way. How this might be achieved is discussed in the following section.

III. Sustainability & Software Architectures

To achieve technical sustainable software, we postulate that [sustainable] software architectures are fundamental to their development [28]. A software architecture is '*the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*' [ISO/IEC 42010-2007]. The rationale for this position is that it is argued that successful software systems development and evolution is highly dependent on making informed decisions at the architectural level as the architecture is the primary carrier of system qualities such as maintainability, modifiability, reusability, portability and scalability, none of which it is argued can be achieved without a unifying architectural vision [33-34]. In addition, Koziolek [35] argues that software architectures determine sustainability as they influence how developers are able to understand, analyze, extend, test and maintain a software system. As a result, software architectures are not only the blueprint of how the software system will be built, they hold the key to post-deployment system understanding, maintenance, and evolution. As a result, this strongly suggests that software architectures are fundamental to achieving software sustainability.

An example of how software architectures are fundamental to the development of sustainable software was reported by Berriman et. al., [36]. They present a case study of an approach to sustainable software applied over a ten year period to astronomy software services at the NASA Infrared Processing and Analysis Center. In this context, software sustainability is implicitly defined as long living. Their approach involved using a component-based architecture, which consisted of approximately one hundred core components, and was designed from its inception to support sustainability i.e. longevity, extensibility, and portability. The rationale for implementing the architectural style was to enable reusability, adaptability and portability. Their approach demonstrates that longevity can be embedded in a software architecture at the outset to support changing stakeholder requirements and technical obsolesce. In addition, they recommend a number of best practices for software sustainability based on their experience:

- Design for sustainability, extensibility, reusability and portability from the outset;
- Adopt a component-based architecture;
- Assess impact and risk of adopting new or emerging technology;
- Build a user community;
- Encourage stakeholder engagement;
- Adopt rigorous software engineering practices;
- Develop open source software.

However, architectural design is in part a creative process where the design of a system must provide a balance between the functional and non-functional requirements. As a result, Sommerville [37] suggests that it is useful to think of this process as a series of decisions where a number of fundamental questions are answered rather than a sequence of activities. In addition, Avgeriou, Stal, and Hilliard [38] highlight that while the software architecture is the foundation of a software system encompassing a system architects and stakeholders strategic decisions, these are often made in an unsystematic and undocumented manner. As a result, this can lead to architectural drift and erosion, resulting in a decrease in software quality, which in turn leads to increased costs and dissatisfied stakeholders. This suggests the architectural design and reasoning is based on the level expertise of the software architect and tacit architectural knowledge. As a result, it is essential to consider the sustainability of the software architecture as an intrinsic part of overall software sustainability.

IV. Sustainable Software Architectures

Architecture sustainability is the capacity of a software architecture to endure different types of change through efficient maintenance and orderly evolution over its entire life cycle [38]. Koziolek [14] conducted a systematic review of the literature in order to address a number of key questions related to architectural sustainability:

- How do scenario-based architecture evaluation methods used industry support the sustainability of software architectures?
- Which are the most appropriate architectural-level metrics to analyze the sustainability of software architectures?

Twenty scenario-based methods were initially identified. Of these only two, ATAM [39] and ALMA [40], met the criteria for inclusion; were active and applied in industrial settings. While ATMA was not specifically designed for sustainability evaluation it offered a range of techniques in the context of sustainability evaluation. In contrast, ALMA was specifically designed for modifiability and offers a number of techniques for change scenario elicitation. However, the overall results suggest that existing scenario-based methods do not provide sufficient support for the systematic analysis of ripple effects, or the integration with reverse engineering tools and knowledge management support. Similarly, forty architectural-level metrics were identified, which could potentially assist sustainability evaluation of implemented architectures. However, many were based on plausibility and have not been systematically validated. As a result, their value in addressing sustainability is an open research question.

Zdun et. al., [41] suggest that software architectures not only comprise a systems structure but essential design decisions based on architectural knowledge. They argue that to achieve sustainable architectures, requires capturing significant sustainable design decisions and their rationale as failure to do so can lead to decision rationale erosion. They derive five key criteria to define decision sustainability:

- Strategic consequences;
- Measurable and manageable;
- Achievable and realistic;
- Rooted in requirements;
- Timeliness.

They suggest that the five criteria are strongly related to the decision life cycle. As a result, the evolution of decisions across the life cycle affects the degree of sustainability achieved at any given time period. Based on the results of a case study, they propose a set of guidelines to identify and capture a minimalistic set of relevant decisions and trace links including

- Establishing explicit traceability links between decisions and requirements;
- Establishing traceability links among decisions, architecture, and code;
- The use and application of design rationale;
- Minimal decision documentation.

As a result, it is suggested that capturing and maintaining relationships between sources and architectural design decisions can prevent 'architectural knowledge vaporization'. How this can be achieved in practice is unclear and provides further avenues for research.

Avgeriou, Stal, and Hilliard [38] identify several causes of change that are significant for architecture sustainability:

- New requirements emerge while older requirements change;
- Interdependence between requirements and architecture;
- Changes in business strategies and goals;
- Environment changes;
- Architecture erosion or drift;
- Accidental complexity;
- Technology change;
- Deferred decisions to meet near-term goals;
- Human error.

To improve architecture sustainability they propose '*systematic architecting'* which considers a system in its total environment, where environment is defined as the '*context determining the setting and circumstances of all influences upon a system including developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences*' [42]. As a whole, these elements form the basis for establishing the forces that architects must consider when making decisions and identifying the risks to be mitigated throughout the systems lifecycle. To support this they distinguish between three types of approaches for handling change systematically which are listed in order of severity:

- Refactoring: modifying existing components without impacting functionality.
- Renovating: rebuilding one or more essential components from scratch;
- Rearchitecting: creating a new architecture.

While maintainability and evolution are usually treated individually they advocate making explicit the differences between the approaches as they suggest that mixing the three types obfuscates the challenges of architecture sustainability for practicing architects and offers no clear direction to researchers.

To address the cost-effective and sustainable evolution of industrial software systems, Koziolek et. al., [43] proposed MORPHOSIS; a holistic, sustainable, software architectural method that incorporates evolution scenario analysis to deal with technological change and unexpected redesign; architecture enforcement to avoid architectural erosion; and architectural-level code metrics framework to assess trends of sustainability. Architectural-level code metrics were based on existing metrics and selected on the basis of their relevance to maintainability [44]. They argue that alternative approaches have only gained limited adoption in practice because the return on investment is unknown. However, they acknowledge that the limitations of their approach are that the metrics are not widely used in practice, some metrics conflict with each other, and that they could not quantify the return on investment of applying their method.

In a follow on study, Koziolek et. al., [45] argue that it is difficult to express a software architecture's sustainability in a single metric as relevant information may be spread across a range of related factors including requirements and architecture design documents, technology choices, source code, systems context, and the software architectures tacit knowledge. As a result, software architecture sustainability should consider multiple perspective including volatile requirements, technology decisions, architecture erosion, and modularization. Applying the MORPHOSIS approach [46], they report the results of a two-year longitudinal study, which tracked selected sustainability measurements of a large-scale, distributed industrial control system. The results suggest that a number of predicted evolution scenarios had occurred and those that had not were still valid. Similarly, the use of architecture enforcement to avoid architectural erosion created a higher awareness for the architecture specification resulting in fewer dependency violations. Finally, the architectural-level code metrics framework led to an improvement in the overall code quality at the design level because a measurement instrument was in place. This suggests that assessments can be conducted with limited effort and that through regular assessment code can be improved through refactoring to achieve improved sustainability. Nevertheless, further work is required to correlate software maintenance costs with the architectural metrics to enable quantitative cost-benefit analysis. However, it was noted that software maintenance remained a challenge.

Sehestedt, Cheng, and Bouwers [47] state that software architectures and their representations in models are instrumental in achieving sustainability, and the fulfillment of functional and non-functional requirements. They suggest that the quality of a software architectural model can be measured by evaluating it against the following criteria: *completeness*, *consistency*, *correctness* and *clarity*. In addition they propose seven, system independent metrics, against which the four-C's criteria can be judged:

- Decomposition quality;
- Best practices adherence;
- View consistency;
- Rationalization completeness;
- Requirement fulfillment;
- Change scenario robustness;
- Decision traceability.

The proposed metrics address quality attributes from three views: architecture models; architectural decisions; and requirement specifications. This approach differs from traditional methods as it indirectly assesses the quality of the architecture through its documentation. The rationale for this is that architecture models and related documentation are generally not formal models through which an architect can evaluate the architectural model in a consistent and repeatable way. However, they acknowledge that coverage of the proposed metrics is limited and requires validation to test the limits of their approach.

In contrast, Ameller et. al., [48] present an interesting counter argument to our position and the general consensus on the symbiotic relationship between non-functional requirements and software architectures. The aim of the study was to investigate how architects deal with non-functional requirements and focused on four questions:

- What types of non-functional requirements are relevant to software architects?
- How are non-functional requirements elicited?
- How are non-functional requirements documented?
- How are non-functional requirements validated?

Based on semi-structured interviews with thirteen software architects at twelve software intensive organizations covering a variety of business and application domains, the results revealed a number of interesting findings. Firstly, non-functional requirements were principally defined by architects rather than being driven by stakeholders needs. Secondly, non-functional requirements are generally not documented or precise in their representation. Finally, non-functional requirements were only partially validated, if at all. Overall, the results suggest that there is a significant mismatch between software engineering theory and practice. However, Buschmann et. al., [49] suggests that this apparent mismatch provides valuable insights into how to deal with non-functional requirements in software development; a prime indicator is their business value. As a result, software architects need to be pragmatic in how they balance the inclusion of non-functional requirements as a prime driver of architecture design.

V. Summary & Conclusions

In this paper we propose that sustainable software architectures are fundamental to the development of technical sustainable software as a basis for discussion in order to consider how we can address the challenge of developing and achieving sustainable software. The paper explores previous research to provide the theoretical foundation to support our thesis.

Examination of the concept of sustainability and its relationship to software demonstrates that software sustainability has been defined from a number of different perspectives including that of the software artifact and the software engineering development process. While there is no agreement on an absolute definition of software sustainability there is growing consensus that software sustainability should be considered a first-class, non-functional requirement that is a measure of a number of core quality attributes. We suggest that at the very minimum, a software's technical sustainability should address two core quality attributes: maintainability and extendibility. To what extent existing metrics and measures of quality attributes defined within existing standards are appropriate for measuring a software artifact's technical sustainability is an open research question and provides further avenues for research. In addition, how to make software sustainable both in terms of the software artifact, the development process, and how these relate to the wider concerns of environmental, economic, social, individual, and technical sustainability remains an open area of research

Software architectures can be considered the Quoins of sustainable software. While research into the relationship between software architectures and sustainability is strictly limited, emerging evidence suggests that the architecture plays a critical role in satisfying non-functional requirements. As a result, software architectures are not only fundamental in understanding how the software system will be built in the first instance, they are critical to post-deployment system maintenance and evolution, which in turn leads to software that is sustainable. While it is suggested that architectural design is more of a creative process, which requires a degree of expertise, the principal challenge is how to embed software architectural practice into software engineering best practice rather than viewed as a by-product of the software engineering process; which is particularly true of software developed in academic environments.

An emerging area of interest in the field of software architectures is architectural sustainability, which aims to address architectural drift and erosion that can result in a decrease in software quality. While a small number of approaches have been proposed their value is unknown and require validation to test their limits. Critical to architectural sustainability is capturing decision viewpoints and their rationale as first-class elements of architectural descriptions. How this can be achieved in practice is unclear and is an area ripe for research. However, based on our previous work into the relationship between trust, provenance, and high-value decision making in data intensive, service-oriented computing environments, we suggest that integrating requirements traceability methods and provenance potentially provides an avenue for further research.

If we accept the premise that software architectures are the key to developing technical sustainable software then ensuring architectural sustainability is critical to its success. Future work will focus on the development of a software sustainability evaluation framework that will assist in facilitating a greater holistic view of sustainable hybrid, GPU-enabled magnetohydrodynamcs (MHD) code in the field of solar plasma physics and serial and parallel molecular dynamics simulation software in the field of computational chemistry.

References

[1] A. Geist, and R. Lucas. "Major computer science challenges at exascale," International Journal of High Performance Computing Applications, 23(4): pp: 427-436, 2009.

[2] C. C. Venters, et. al., "Software Sustainability: The Modern Tower of Babel," RE4SuSy: Third International Workshop on Requirements Engineering for Sustainable Systems, 2014.

[3] B. Penzenstadler, and A. Fleischmann. "Teach sustainability in software engineering?" Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training, IEEE Computer Society, pp: 454-458, 2011.

[4] A. B. Bener, M. Morisio, A. Miranskyy. "Green software," IEEE Software, 31(3), 2014, pp: 36-39.

[5] E. F. Ecklund, Jr., L. M. L. Delcambre, and M. J. Freiling. "Change cases: use cases that identify future requirements," In: OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp: 342-358, 1996.

[6] The Bible, Daniel 2:34-45.

[7] R. C. Seacord, J. Elm, W. Goethert, G. A. Lewis, D. Plakosh, J. Robert, L. Wrage, and M. Lindvall. "Measuring software sustainability," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2003.

[8] Oxford English Dictionary. 2012. Oxford Dictionaries.

[9] United Nations: World Commission on Environment and Development: Our Common Future. Oxford Univ. Press, 1987.

[10] European Information Technology Observatory. "The impact of ICT on sustainable development," pp: 250-283, 2002.

[11] J. Elkington. "Enter the triple bottom line," In: R. Henriques and A. Richardson (eds.). The triple bottom line: Does it all add up? pp: 1-16, 2004.

[12] R. Goodland. "Sustainability: Human, social, economic and environmental," Encyclopedia of Global Environmental Change, John Wiley & Sons, 2002.

[13] B. Penzenstadler and H. Femmer. "A generic model for sustainability with process- and product-specific instances," In International Workshop on Green In Software Engineering and Green By Software Engineering at AOSD, 2013.

[14] H. Koziolek. "Sustainability evaluation of software architectures: A systematic review," Proceedings of the joint ACM SIGSOFT conference on quality of software architectures. Boulder, Colorado, USA, ACM: 3-12, 2011.

[15] T. Tamai and Y. Torimitsu. "Software lifetime and its evolution process over generations," Proceedings of the Conference on Software Maintenance, pp: 63-69, 1992.

[16] Software Sustainability Institute. Available: http://www.software.ac.uk/about.

[17] Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss. "Sustainability guidelines for long-living software," ICSM 2012: Proceedings of the 28th IEEE International Conference in Software Maintenance, Trento, Italy, September 23-28, 2012.

[18] B. Penzenstadler. "Towards a definition of sustainability in and for software engineering," SAC'13: Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, March 18-22, pp: 1183-1185, 2013.

[19] IEEE. 1990. "IEEE standard glossary of software engineering terminology," IEEE Std. 610.12-1990.

[20] N. Amsel, Z. Ibrahim, A. Malik, B. Tomlinson. "Toward sustainable software engineering," ICSE: Proceedings of the 33rd International Conference on Software Engineering. Waikiki, Honolulu, HI, USA, pp: 976-979, 2011.

[21] S. Naumann, M. Dick, E. Kern, T. Johann. "The GREENSOFT model: A reference model for green and sustainable software and its engineering," Sustainable Computing: Informatics and Systems, 1, pp: 294-304, 2011.

[22] R. A. Fenner, C. M. Ainger, H. J. Cruickshank, P. M. Guthrie. "Widening engineering horizons: Addressing the complexity of sustainable development," Proceedings of the ICE: Engineering Sustainability, 159 (4), pp: 145-154. 2006.

[23] WSSSPE Collaborative notes. Available bit.ly/wssspe13

[24] B. Penzenstadler, A. Raturi, D. Richardson, and B. Tomlinson. "Safety, security, now sustainability: The nonfunctional requirement for the 21st century," IEEE Software, 31(3), pp: 40-47, 2014.

[25] F. Brooks, "The mythical man-month," Addison-Wesley, 1995.

[26] D. M. Coleman, D. Ash, B. Lowther, and P. W. Oman, "Using metrics to evaluate software system maintainability." IEEE Computer, 27 (8), pp. 44–49, 1994.

[27] C. C. Venters et. al., "The blind men and the elephant: Towards an empirical evaluation framework for software sustainability," WSSSPE'1: First workshop on sustainable software for science: practice and experiences, SC'13, 17 November 2013, Denver, CO, USA, 2013.

[28] C. C. Venters et. al., "The blind men and the elephant: Towards an empirical evaluation framework for software sustainability," Journal of Open Research Software. 2(1):e8, pp: 1-6, 2014.

[29] C. Calero, M. A. Moraga, and M. F. Bertoa. "Towards a software product sustainability model," WSSSPE'1: First workshop on sustainable software for science: practice and experiences, SC'13, 17 November 2013, Denver, CO, USA, 2013.

[30] ISO/IEC 25010:2011: Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models.

[31] H. Koziolek, D. Domis, T. Goldschmidt, and P. Vorst. "Measuring architecture sustainability," IEEE Software, 30 (6), pp: 54-62, 2013.

[32] ISO/IEC 42010:2007 Systems and software engineering -- Recommended practice for architectural description of software-intensive systems.

[33] P. Clements, R. Kazman, and M. Klien, "Evaluating software architectures: methods and case studies," Addison-Wesley Professional, 2002.

[34] Software Engineering Institute. Defining software architectures. Available at: http://www.sei.cmu.edu/architecture/

[35] Koziolek, H. "Sustainability evaluation of software architectures: a systematic review," In: QoSA-ISARCS'11: Proceedings of the joint ACM SIGSOFT conference on quality of software architecture and architecting critical system, New York, NY, USA, pp: 3-12, 2011.

[36] G. B. Berriman, J. Good, E. Deelman, A. Alexov. "Ten years of software sustainability at the infrared processing and analysis center," Philosophical Transactions of the Royal Society A. pp: 3384-3397, 2011.

[37] I. Sommerville. "Software engineering," 9th edition, Pearson, 2011.

[38] P. Avgeriou, M. Stal, and R. Hilliard. "Architecture sustainability," IEEE Software, 30 (6), pp: 40-44, 2013.

[39] P. Clements, R. Kazman, and M. Klein. Evaluating software architeture analysis methods. IEEE Transactions on Software Engineering, 28(7), pp: 638-653, 2002.

[40] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. "Architecture-level modifiability analysis (ALMA)," Journal of Systems and Software, 69(1-2), pp: 129-147, 2004.

[41] U. Zdun, R. Capilla, H. Tran, and O. Zimmermann. "Sustainable architectural design decisions," IEEE Software, 30 (6), pp: 46-53, 2013.

[42] Systems and Software Engineering-Architecture Description, IEEE Std. ISO/IEC/IEEE 42010, 2011.

[43] H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst, and R. J. Weiss. "MORPHOSIS: A lightweight method facilitating sustainable software architectures," WICSA-ECSA: Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), Helsinki, Findland, pp: 253-257, 2012.

[44] S. Sakar, G. M. Rama, and A. C. Kak. "API-Based and information-theoretic metrics for measuring the quality of software modularization," IEEE Transactions on Software Engineering, 33 (1), pp: 14-32, 2007.

[45] H. Koziolek, D. Domis, T. Goldschmidt, and P. Vorst. "Measuring architecture sustainability," IEEE Software, 30(6), pp: 54-62, 2013.

[46] H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst, and R. J. Weiss. "MORPHOSIS: A lightweight method facilitating sustainable software architectures," WICSA-ECSA: Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), Helsinki, Findland, pp: 253-257, 2012.

[47] S. Sehestedt, C-H. Cheng, and E. Bouwers. "Towards quantitative metrics for architecture models. First International Workshop on Software Architecture Metrics," WICSA 2014: 11th Working IEEE/IFIP Conference on Software Architecture, Sydney, Australia, April 7-11, 2014.

[48] D. Ameller, C. Ayala, J. Cabot, and X. Franch. "Non-functional requirements in architectural decision making," IEEE Software, 30 (2), pp: 61-67, 2013.

[49] F. Buschmann, D. Ameller, C. P. Ayala, J. Cabot, and X. Franch. "Architecture quality revisited," IEEE Software, 29, 4 pp: 22-24, 2012.