View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Huddersfield Repository

Accepted Manuscript

Software Sustainability: Research and Practice from a Software Architecture Viewpoint

Colin C. Venters , Rafael Capilla , Stefanie Betz ,
Birgit Penzenstadler , Tom Crick , Steve Crouch ,
Elisa Yumi Nakagawa , Christoph Becker , Carlos Carrillo

PII: S0164-1212(17)30307-2
DOI: 10.1016/j.jss.2017.12.026
Reference: JSS 10097

To appear in: *The Journal of Systems & Software*

Received date: 13 December 2017
Accepted date: 19 December 2017

Please cite this article as: Colin C. Venters , Rafael Capilla , Stefanie Betz , Birgit Penzenstadler , Tom Crick , Steve Crouch , Elisa Yumi Nakagawa , Christoph Becker , Carlos Carrillo , Software Sustainability: Research and Practice from a Software Architecture Viewpoint, *The Journal of Systems & Software* (2017), doi: 10.1016/j.jss.2017.12.026

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

Highlights

- An overview of software sustainability approaches and trends
- Challenges and insight on current software sustainability research
- A set of different approaches as school of thought
- A perspective of sustainability academy research and practice

Software Sustainability: Research and Practice from a Software Architecture Viewpoint

Colin C. Venters[a], Rafael Capilla[b], Stefanie Betz[c], Birgit Penzenstadler[d], Tom Crick[e], Steve Crouch[f], Elisa Yumi Nakagawa[g] Christoph Becker[h], Carlos Carrillo[i]

[a]University of Huddersfield, UK
[b]University Rey Juan Carlos, Spain
[c]Karlsruhe Institute of Technology, Germany
[d]California State University Long Beach, USA
[e]Cardiff Metropolitan University, UK
[f]University of Southampton, UK
[g]University of São Paulo, Brazil
[h]University of Toronto, Canada
[i]Technical University of Madrid, Spain

Abstract

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 vaporisation 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.

1. Introduction

Modern societies are highly dependent on complex software systems, which are deeply embedded into the "unconsciousness" of every facet of daily living, from commerce, communication, education, energy, entertainment, finance, governance, healthcare, transportation, as well as defence and security [Kitchin, 2011]. Fashioning complex conceptual constructs is the fundamental essence of software engineering. For example, modern modes of transportation such as the Airbus A380, which has an estimated operational lifespan of 25 years includes 120 millions of lines of mission-critical code [Charette, 2009]. However, there are increasing concerns regarding the fragility of these systems, which operate in a highly connected ecosystem with emergent properties, whose interdependencies can lead to cascading failures [Cerf, 2017]. Despite the emergence of clear and systematic approaches, the design and development of high-quality, sustainable software systems are still extremely challenging for software engineers involved in their design, development, and maintenance [Brooks, 1986; Lehman, 1998; Somerville, 2007; Taivalsaari and Mikkonen 2017]. The challenges are further exacerbated by change [Bener, 2014]. It is estimated that approximately 50%–70% of a system's total lifecycle cost is spent on its evolution [Ecklund, 1996] and maintenance [Garcia, Ivkovic and Medvidovic, 2013]. Similarly, continuous evolution and deployment of systems are heralded as the new "stairway to heaven" of software engineering, where systems and organizations evolve together to satisfy more agile customer demands [Oreizy et al., 1998; Bosch, 2014; Ameller et al., 2017, Fitzgerald and Stol, 2017; Rodríguez et al., 2017]. In this era of post-deployment, many systems are reconfigured several times on the client side or are updated automatically based on third-party software providers (e.g. mobile apps updated at any time on a smartphone). While the emergence of continuous software engineering allows development teams to release the current development version of their software to users at any time in the development cycle [Fitzgerald, 2017], this continuous cycle of redeployment is affected by how well prepared the systems are for integrating new requirements that must be satisfied within hours or days. As a result, this can lead to unexpected increases in memory and CPU usage that can lead to a significant decrease in system performance and regression failures in stable parts of the system [Tarvo, 2009]. As a result, how to design more sustainable software systems that can endure is one of the grand challenges in the field of software engineering.

Modern society's reliance on 'dangerously fragile' software [Booch, 2015] has resulted in the emergence of software sustainability as a growing area of interest in the field of software engineering [Venters et al., 2014a]. The Karlskrona Manifesto [Becker et al., 2014] reflects this new trend by providing a focal point for establishing a common ground for the software engineering community to engage. It argues that designers of software technology are responsible for the long-term consequences of their designs - a position also supported by Cerf [2017] - and proposes a set of key principles and commitments that underpin sustainability design. These include the importance of recognising that sustainability is an explicit consideration even if the primary focus of the system under design is not sustainability, i.e. a concern independent of the purpose of the system, which requires action on multiple levels. While consensus on what sustainability means in the field of software engineering is still emerging [Venters et al., 2014b], there has been a focus towards understanding technical sustainability whose overarching goal is for software developers to

achieve maintainable and extendable systems [Amri and Saoud, 2014]. Koziolek [2011] postulates that software systems are sustainable if they can be cost-efficiently maintained and evolved over their entire life-cycle, which is arguably determined by the software architecture. It is widely accepted that software architectures are the foundation of any software system as they provide 'the fundamental organisation 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]. As such, they provide a mechanism for reasoning about key software qualities (e.g. maintainability, extendability, scalability, security, performance, reliability, portability etc.) [Garlan, 2000]. However, while a plethora of software metrics can be used to measure and understand code complexity issues, the ability to determine the sustainability of a legacy or new system from an architectural perspective in terms of its cost and energy with regards to maintenance and evolution cycles is an open research area. There is still a fundamental lack of metrics to estimate architecture sustainability or the sustainability of architectural design decisions; this and other challenges are the primary focus of this article. We highlight the current state of the practice of software sustainability and identify the problems that new metrics and tools must address in the future.

The remainder of this paper is structured as follows. In Section 2, we outline the different dimensions of software sustainability and discuss their applicability to the broader field of software engineering to frame the discussion with regards to the area of software architectures. Section 3 addresses sustainability from a software architecture point of view and how it affects reference and software architectures, while Section 4 describes how the architecture design decisions must also be sustainable as long-term and stable decisions. In Section 5, we address software metrics related to estimating the sustainability in architecture and code from a practical point of view, and Section 6 provides perspectives from academia, concerning how and where an awareness of software sustainability is developed with regards to scientific and engineering research software. Finally, the paper concludes by identifying some open issues and research challenges in Section 7.

2. Software and Sustainability

Before software sustainability can be measured, it must be understood [Seacord et al., 2003]. In modern English, sustainability refers to the 'capacity' of a system 'to endure' [Oxford 2010]. The term's Latin origin *sustinere* was used as both *endure* and as *uphold, furnish* [something] with means of support.[1] This suggests that longevity as an expression of time and the ability to maintain are key factors at the heart of understanding sustainability. A closely related concept, sustainable development, was defined by the Brundtland Commission [Brundtland, 1997] 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, as well as a direct reference to acknowledging changing requirements of stakeholders and evolution

[1] http://en.wiktionary.org/wiki/sustineo#Latin

In relation to software, there exist at least two distinct viewpoints for the topic area of software and sustainability: sustainable software and software engineering for sustainability (SE4S). The former is concerned with the principles, practices, and processes that contribute to software endurance, i.e. technical sustainability, and the latter focuses on software systems to support one or more dimensions of sustainability, concerning issues outside the software systems itself [Penzenstadler, 2013]. The Karlskrona Manifesto [Becker et al., 2014] recognises both viewpoints of software sustainability as an emerging concern of central relevance and advocates that sustainability must be viewed as a construct across five dimensions: environmental, economic, individual, social and technical [Becker et al., 2015]. These dimensions are defined as follows:

- The economic dimension focuses on assets, capital and added value that comprises wealth creation, prosperity, profitability, capital investment, income, etc.
- The environmental dimension is concerned with the long-term effects of human activities on natural systems, which includes natural ecosystems and resources, the climate, pollution and waste, etc.
- The individual dimension refers to the well-being of humans as individuals, which includes mental and physical well-being, education, freedom, self-respect, mobility, agency etc.
- The social dimension covers societal communities (groups of people, organisations) and the factors that erode trust in society. The concepts analysed here encompass social equity, justice, employment, democracy, etc.
- The technical dimension includes the concept of the longevity of information, systems, and infrastructure and their adequate evolution within changing environmental conditions, which covers inter alia, system maintenance, obsolescence, and data integrity.

Nevertheless, interdependencies exist between these dimensions including tradeoffs that may have to be negotiated for a system under analysis [Becker et al., 2016]. For example, consider a car sharing system composed of a fleet of private cars that are being shared, a client/server software application that allows users to connect and sign up for rides, and a database server that stores the information in the background. We can identify the following details for the five dimensions:

- Economic: Sharing rides as opposed to having to own a car, or offering up rides in one's car can save costs for the user and (accumulated) for the user community. The service can only be sustainable if it is economically sustainable in terms of the continued supply of income streams sufficient to keep it operational. Cost efficiency for the software system's development, maintenance and operations will be affected by such choices as using open source components and applying architectural patterns to avoid incurring technical debt.
- Environmental: IT systems as well as cars require energy and therefore have an impact on the environment, e.g. through emissions. Furthermore, there is a lifecycle for the respective hardware parts that have to be resourced from somewhere, manufactured, maintained, and eventually disposed of or recycled. In a sharing system, the bottom line usage of resources often decreases, which reduces its environmental impact [Wadud et al., 2016].
- Individual: A user may benefit individually from access to individual mobility and an improved sense of responsibility linked to environmentally conscious behaviour.

- Social: Depending on the system and its mechanisms, a new community of users can form with a focus on helping each other out in choosing less carbon-intensive modes of transport; or social ride-sharing communities can erode as slowly grown personal connections are replaced by routing algorithms.
- Technical: Both the connecting IT system and the cars have to be maintained over time. The system's longevity will be influenced by such factors as technical debt, the ability of its architecture to evolve, and the lifecycles of supporting technologies.

Relationships between specific dimensions arise instantly; however, all of these dimensions -- and their intersections -- have to be analysed with regards to what their impact is for a long-term vision of the system and how it can be ensured that they are well supported. The research community is increasingly aware of the need to move towards a more comprehensive view of sustainability, which embraces these different dimensions. In addition, the impact on sustainability in these five dimensions manifests in three orders of effect [Hilty and Aebischer, 2015], defined as follows:

- First order effects appear when software systems are built and used for their direct purpose, e.g. the resourcing, manufacturing, installing and usage of the hardware and software needed for the car sharing system;
- Second order effects appear when the use of the system over time induces new types of behaviour or expectations from the previous system, e.g. users of the car sharing system organise themselves into a community resulting in smaller individual environmental footprints;
- Finally, third order effects appear due to a large-scale, longer-term use of the system, e.g. less downtown parking space problems, improved air quality in cities, etc.

The concept of sustainability is not widely understood in the field of software engineering, with opposed views on its meaning in the software engineering community. [Venters et al., 2014a; Chitchyan et al., 2016; Manotas et al., 2016; Kasurinen et al., 2017; Groher and Weinreich, 2017]. Nevertheless, a number of contributions have proposed a formal definition of software sustainability, generally focused on the software system's capacity to endure [Koziolek, 2011; Penzenstadler, 2013; Calero, Moraga, and Bertoa, 2013]. As a result, the term has been described in the literature as a first-class, non-functional requirement or software quality [Penzenstadler et al., 2014]. For example, Venters et al., [2014] defined software sustainability as a composite, non-functional requirement which is ‘a measure of a system's 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 utilisation over the software's entire lifetime and aligns with the dimensions of environmental and economic sustainability. However, consensus on what sustainability means in the field of software engineering is still emerging [Venters et al.,

2014] and further research is required to confirm or refute this position. In addition to the simultaneous consideration of several interrelated dimensions of sustainability [Becker et al., 2016], it is argued that the concept of sustainability requires context -- such as that proposed by Tainter [2006] -- and (social) structure [Ramsey, 2015]. Similarly, it is suggested that rather than seeking broad conformity of definitions, the aim should be to clarify how different communities use the terms to have a shared and common understanding [Knowles et al., 2013].

The primary focus of research in the field of software engineering with regards to sustainability has focused on reference models to develop sustainable software and approaches for software sustainability evaluation. In the area of software engineering and sustainability, a number of reference models to develop sustainable software (i.e. a system that requires less maintenance effort to be changed or reduces its energy consumption during execution saving the resources as well) have been proposed. For example, Naumann et al., [2011] proposed the GREENSOFT model for the development of sustainable software; a conceptual reference model, which includes a cradle-to-grave product life cycle model for software products, sustainability metrics and criteria for software, and extensions for software engineering. The model covers development, distribution, usage, deactivation and disposal of software systems, and offers two categories of sustainability criteria and metrics for software products covering direct and long-term impact. Similarly, Mahmoud and Ahmad [2013] propose a development model aimed at supporting (environmentally) sustainable software engineering that includes a list of metrics to measure the environmental sustainability of each software engineering phase. While these approaches primarily focus on environmental sustainability, they acknowledge that the identification of impacts of the software systems on sustainable development should not be limited to a single dimension but should also include other sustainability dimensions such as that proposed by Penzenstadler and Femmer [2013]. Their proposed method comprises a generic sustainability reference meta-model and instances derived for specific processes and software systems that are primarily designed to aid software developers by demonstrating how environmental sustainability can be aligned with the other dimensions of sustainability, i.e. economic, individual, social, and technical.

In addition to the development of a number of reference models for sustainable software, a number of approaches for software sustainability evaluation, which focus on evaluating the longevity of software systems have also been proposed. Cabot et al., [2009] proposed the i* framework as a sustainability taxonomy for modelling and integrating stakeholders' sustainability issues, which can be used for exploring alternative design options during the development of a software system where decisions may have a potential impact on sustainability. However, the extent to which this approach can be utilised beyond the case study used to develop the taxonomy is unclear but provides a useful basis to explore its limits and generalizability. Jansen, Wall and Weiss [2011] focus on sustainability from an economic perspective and consider how a system can remain economically viable over its entire lifetime. To address this, they propose TechSuRe as a method for reasoning about sustainability in assessing software evolution and technology integration from three perspectives: time, risk and cost benefit. Sustainability is defined in terms of 'sustainability risk' which is an estimated value based on nine high-level indicators: lifetime in production; lifetime; competence risk; technology evolution risk; risk of changing business model; market risk; lifetime certainty; complexity risk; and technology evolution-fitness. The output of the

assessment is an indication of the expected lifetime of the technology's economic sustainability. Koziolek et al., [2013] developed a multi-perspective approach to analyzing architecture sustainability (i.e. it could be understood in terms of (i) the design of sustainable systems, (ii) the capacity of a software architecture to evolve and cope with new changes without affecting the fundamental structure of the design) using scenario analysis, architecture compliance checking, and architecture metric tracking. This multi-perspective approach enables tracking changes in requirements and technology as well as to prevent architecture erosion. Durdik et al., [2012] developed a catalogue of sustainability guidelines for different stakeholders such as managers, architects and programmers covering the software development life-cycle from system design to maintenance. The guidelines include software engineering methods, techniques and tools to enhance the longevity of systems in a cost-efficient way. Both approaches focus on the technical and the economic dimension of sustainable software systems and do not discuss the meaning of different sustainability dimensions. In addition, other related work on sustainability in the field of software and requirements engineering has focused on sustainability requirements elicitation [Mahaux et al., 2011] and modelling [Roher and Richardson, 2013a; Roher and Richardson, 2013b].

In this article, we distinguish the five dimensions that articulate concerns of relevance and provide a scope for the indicators and concepts required to understand the capacity of real-world cyber-physical and socio-technical systems to endure. We focus the discussion more specifically on two particular aspects of software architecture sustainability. To do this, we focus more narrowly on three concepts:

1. Software sustainability. The capacity of the software-intensive system itself to endure will be a concern for the operating organisation and the community [Becker et al., 2015].
2. Software architecture sustainability. The capacity of that software system to endure in turn is contingent upon its structures and their ability to evolve. This is often referred to as architecture sustainability (i.e. the degree to which the architecture of the software system supports its continued maintenance and evolution over time without requiring substantial and expensive restructuring) [Koziolek, 2011].
3. Sustainable software architecture design decisions. This architecture, in turn, reflects the foundational design decisions that structure the system and its elements, so decision making is increasingly a focus of attention (van Vliet & Tang, 2016). Because architectural decisions have long-lasting effects and are expensive to revise, the capacity of each decision to remain valid is a primary concern to the architects. This notion has been described as the sustainability of software architecture design decisions (Zdun et al., 2013).

The following sections provide an overview of fundamental concepts and highlight some of the recent work in the area of software architecture sustainability (Section 3), architectural decisions (Section 4), and metrics to estimate sustainability (Section 5).

3. Software Architectures and Sustainability

Software systems are directly dependent on their architectural design to ensure their long-term use, efficient maintenance, and appropriate evolution in a continually changing execution environment [Kruchten et al., 2016]. Bass et al., [2012] state that software

architectures are the critical factor in the capacity of software systems to endure and evolve. Architecture sustainability refers to the ability of the architecture to tolerate changes resulting from shifts in requirements, environment, technologies, business strategies and goals throughout software system life cycles [Avgeriou et al., 2013; Kim et al., 2014]. However, the sustainability of any system architecture is degraded by two related phenomena: architectural drift and erosion [Taylor et al., 2009]. Architectural erosion often appears when the source code becomes sub-optimal compared to the designed architecture. In contrast, architectural drift is considered a divergence of the code of a system from its underlying architecture. Both problems are the product of unsystematic, unintended addition, removal, and modification of architectural design decisions and can arise during evolution and maintenance cycles of the system [Garcia et al., 2013]. Many factors can lead to architecture erosion and drift, from the accumulation of wrong or sub-optimal design decisions to communication problems between design and development teams [Jaktman et al., 1999]. The field of software architectures as a distinct discipline within the broader context of software engineering is still embryonic despite its rise to prominence over the last fifteen years [Woods, 2016]. Despite the pivotal role that software architectures play in the design of software systems, the topic of software architecture sustainability has only recently emerged as a specific area of research. This has primarily focused on investigating the role of technical debt and architectural metrics to measure the sustainability of architectural designs [Avgeriou et al., 2013; Durdik et al., 2012; Giesecke et al., 2011; Koziolek et al., 2012; Koziolek et al., 2013; Sehestedt et al., 2014].

One approach to designing flexible, open software architectures that are sustainable is to pre-emptively design them to accommodate future changes to a greater extent without significant change to the basic structure of the system with minimal cost [Kim et al., 2014]. For this, adherence to established design principles (e.g. separation of concerns and conceptual integrity etc.) and the avoidance of poor evolution decisions are critical factors [Garlan, 2000]. The emergence of software reference architectures, which embody the architectural knowledge of structures, elements and the relations of many successful architectural implementations, can provide templates for designing sustainable architectures albeit constrained to a particular domain or a family of software systems [Nakagawa, 2014]. Well-known reference architectures include: AUTOSAR[2] for the automotive sector; Continua[3] for health systems; OASIS Service-Oriented Foundation; IBM Service-Oriented Solution Stack (S3) for Service-Oriented Architectures (SOA); and the recent Industrial Internet Reference Architecture (IIRA). For example, AUTOSAR has brought a number of significant benefits related to standardisation, interoperability facilitation, knowledge reuse, and improvement in communication among interested parties (e.g. vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry) [Martinez et al., 2015]. However, sustainability has not been explicitly or adequately addressed in these reference architectures. While software architectures constitute the design solution for specific systems, reference architectures operate at a higher level of abstraction for a set of systems in specific domains. Both types of architecture embody elements of reusable knowledge of critical design decisions that endure in favour of standardisation and the sustainability of proven solutions over time. One of the most

[2] http://www.autosar.org
[3] http://www.pchalliance.org

apparent and visible indicators of the success of these architectural solutions is their longevity. However, this is dependent on their resilience to design degradation and the ability to detect degradation symptoms [Bertran, 2014]. To address this, AUTOSAR adopts an update policy with release and version control of its documentation to manage evolution; as each document is continually updated taking parts of different releases. The main evolution phases and releases of AUTOSAR are presented in Figure 1.

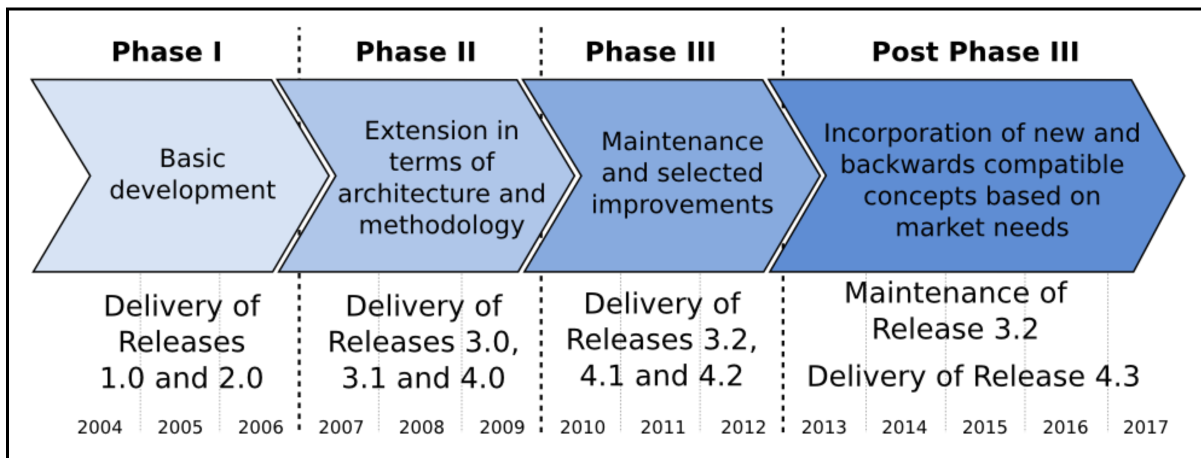

Figure 1: Evolution Phases and Main Releases of AUTOSAR

Other reference architectures also present a continuous delivery of new versions, as shown in Figure 2. However, while new releases include refinements and extension, this has resulted in a significant increase in the amount of documentation. For example, the first version of Continua in 2008 contained 231 pages, while the current version has almost three times the number of pages (654), similar to IIRA.

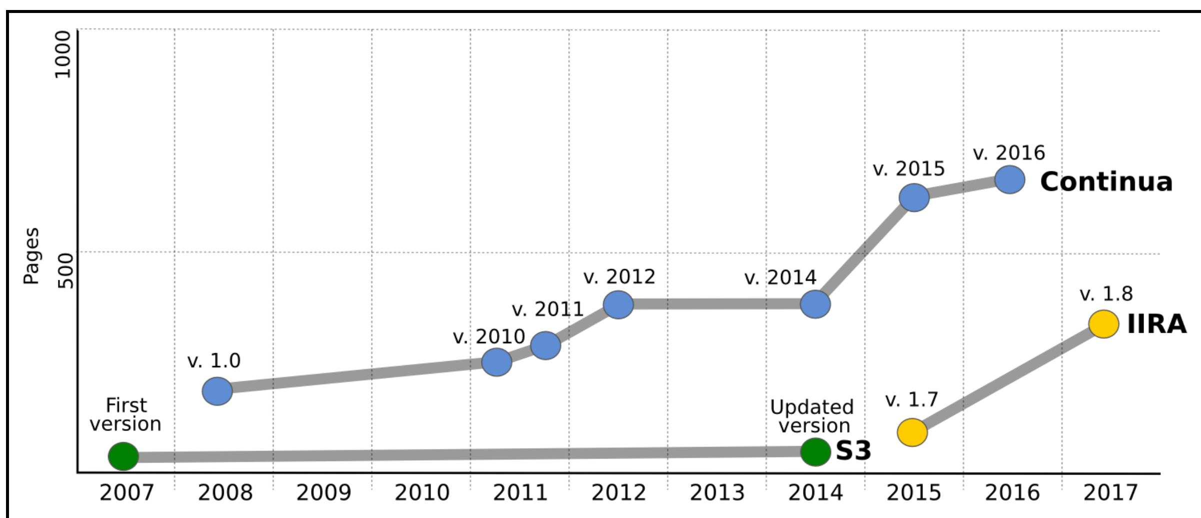

Figure 2: Updates in Reference Architectures

Nevertheless, while a number of reference architectures have been proposed for a diverse range of application domains, many of them have not been adopted or survived. For example, Oliveira et al., [2010] analysed sixteen reference architectures focusing on the Service Oriented Paradigm for systems development. The results highlighted that thirteen of them did not present any evidence of use (i.e., no website, projects, or related publications).

However, as new challenges demand changes in these architectures to support new requirements (e.g. smart car functions), references architectures must be ready to integrate further design decisions that endure over time. From a broader perspective, some of the principal factors that make reference architectures sustainable are:

- Alignment to state-of-practice use case scenarios and the widely accepted and adopted technologies in the target domains (including architectural patterns and styles, domain standards and legislation, communication protocols, etc.);
- Regular updates and releases;
- Decisions that endure over time when the architectures evolve with the addition of new requirements without having a detrimental effect on existing decisions;
- The existence of a community around the architecture, which can in some instances be strengthened via a consortium of companies, research centres, and universities.

Examining these features suggests that a critical factor in the long-term existence of reference architectures is that their communities sustain them. Sustainability should also be considered as the primary, overarching quality attribute in the process of the design and evolution of reference architectures such as ProSA-RA taking into account the significant effort and time required in their establishment [Nakagawa, 2014]. Similarly, as reference architectures provide the foundation for the design of a number of derived systems architectures, sustainability of the subsequent systems should be the central concern. Hence, revisiting existing reference architectures to re-architect them for sustainability is a rational middle to long-term investment for both the reference architectures sustainability and the sustainability of derived architectures.

4. Software Architecture Decisions and Sustainability

The multi-faceted concept of "sustainability debt" (i.e. how technical debt can be used to identify and communicate about the effects of software design decisions on sustainability) discussed in Betz et al., [2015] reflects the hidden effect of past design decisions as a negative factor affecting the five sustainability dimensions including the economic issue of long-term costs. As a result, design decisions strongly influence the longevity of systems and their architecture. However, the sustainability of designs rely not only on the quality of optimal design decisions but also on economic, individual, social, and technical factors required to capture those decisions including lack of motivation or incentive, lack of adequate tools, the effort in capturing architectural knowledge (AK), disrupting the design flow, lack of stakeholder understanding, and knowing what knowledge is relevant and valuable to capture which is especially challenging in Agile projects where documentation is reduced to a minimalistic set of data [Zimmermann, 2007]. Nevertheless, it is suggested that the many barriers related to capturing AK stem from the significant and ongoing effort required to efficiently store large quantities of codified knowledge that can not only be difficult to manage but also to efficiently use [Capilla et al., 2016]. As a result, there is a need to capture a sustainable body of design decisions that are easier to maintain and that can be usefully applied. For example, Zdun [2013] proposed capturing a minimal set of significant architectural design decisions using configurable AK templates to reduce AK documentation effort. Consequently, achieving sustainable decisions should be based on timeless and strategic knowledge, which can extend the longevity of systems and their architectures,

combined with minimalistic and lightweight approaches that can succinctly reduce the documentation and capture the salient decisions. To address this, Carrillo et al., [2015] proposed a meta-model and set of guidelines to enable the construction of flexible Architectural Knowledge Management[4] (AKM) tools, which allows designers to create more sustainable architectural design decisions. Their approach is based on a flexible and configurable new meta-model that overcomes the inflexibility of previous approaches [Zimmermann et al., 2007; Capilla et al., 2011] by combining a set of AKM tools for maintaining a minimal set of AK, a set of configurable entities to extend and customize new AK, and a set of related metrics that can be used to measure the sustainability of the design decisions. Quality attributes such as timeliness, changeability, complexity and cost of the AK are strongly related to the metrics and proposed criteria to estimate AK sustainability [Capilla et al., 2017]. As a result, it is suggested that the model and the set of proposed criteria can be used to build flexible and configurable AKM tools that can measure the sustainability of the size of the decisions captured (e.g. number of AK items captured, number of trace links etc) during the design process, and how well the decisions evolve (i.e. estimate the impact of changes using ripple effect and instability metrics). As an improvement to the meta-model proposed by Carrillo et al., [2015], Carrillo [2017] proposed a modification that suggests a refinement to the sustainability model and the extensions used for capturing additional AK items (Figure 3).

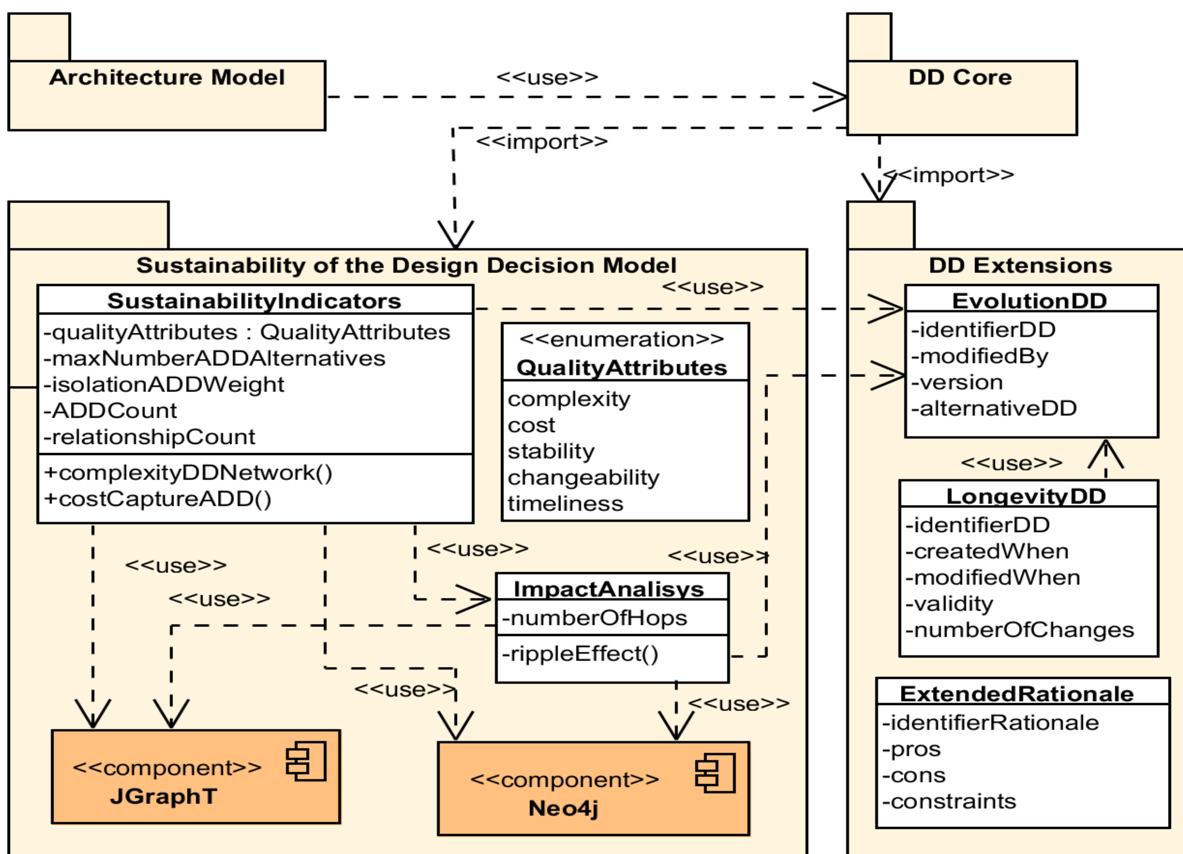

Figure 3: Refined meta-model to measure decisions that endure and other sustainability estimators related to AK

[4] Architecture Knowledge Management (AKM) can be understood as the process of creating, sharing, using and managing architectural design decisions and other coded and reusable knowledge.

In Figure 3, the classes belonging to the DD Extensions package add a list of configurable AK items that can be used to capture different amounts of AK (i.e. the smaller the number of AK items captured reduces the burden and maintenance cost of the AK capturing process and leads to more sustainable decisions networks), and those attributes can then be used to estimate the longevity, timeliness and validity of decisions (e.g. *createdWhen*, *modifiedWhen*, *validity*, *version*, *numberOfChanges*). In the Sustainability of the Design Decision Model package, a number of quality attributes related to the estimation of sustainability are proposed such as the number of decisions and edges as useful indicators to estimate the complexity of a decision network. Other metrics related to evolution include ripple effect measures as a way to understand the impact of changes and to highlight which decisions are frequently modified. In addition, the Sustainability of the Design Decision Model package was extended with two new components (i.e. JGraphT and Neo4j) to visualise the decision network as well as new methods to estimate sustainability based on the complexity of the decision networks.

Similarly, Carrillo [2017] investigated the effort required to capture as well as the quality of key architectural design decisions using different architectural templates: short (seven items), medium (ten items), and long (fourteen items). The experiment included sixty-four participants, randomly divided into eleven different groups; nine groups with six participants and two groups with five. Each group was composed of three different roles (junior, senior, and cognitive software architects) to capture the key design decisions of a given system over a four week period. As a research methodology, we used an exploratory case study to identify what would happen if different stakeholders chose different AK templates to capture architectural design decisions and how the subjects can cooperate during the decision-making activity. We did not use test and control groups because we were not testing an independent variable but instead estimating how much effort the subjects employ for capturing the AK using the different templates and how many alternative decisions they captured on average. The results suggest that participants worked more efficiently with a few decisions as these were easier to manage during evolution cycles and that short and medium templates proved more useful for capturing the AK (Figure 4). From the results shown in Figure 4, we can conclude that groups using the short template spent less time in general than those using the medium and long templates. However, in the case of group G1, the team members spent almost three times more than groups G7 and G10 using the same template because they captured much more decisions. Groups using the medium template produced the expected results compared to those using the short one, but only group G4 spent more time than G3, G6 and G11 capturing less number of decisions. This dissonance is the product of the wrong answers given by the team members during the individual tests we ran to check the accuracy of the results, which in some cases were exaggerated by the subjects. Finally, group G5 exhibits an anomalous result compared to G2 as the members spent less effort than G2 in capturing more decisions, which can be attributed to an incorrect answer during the tests.

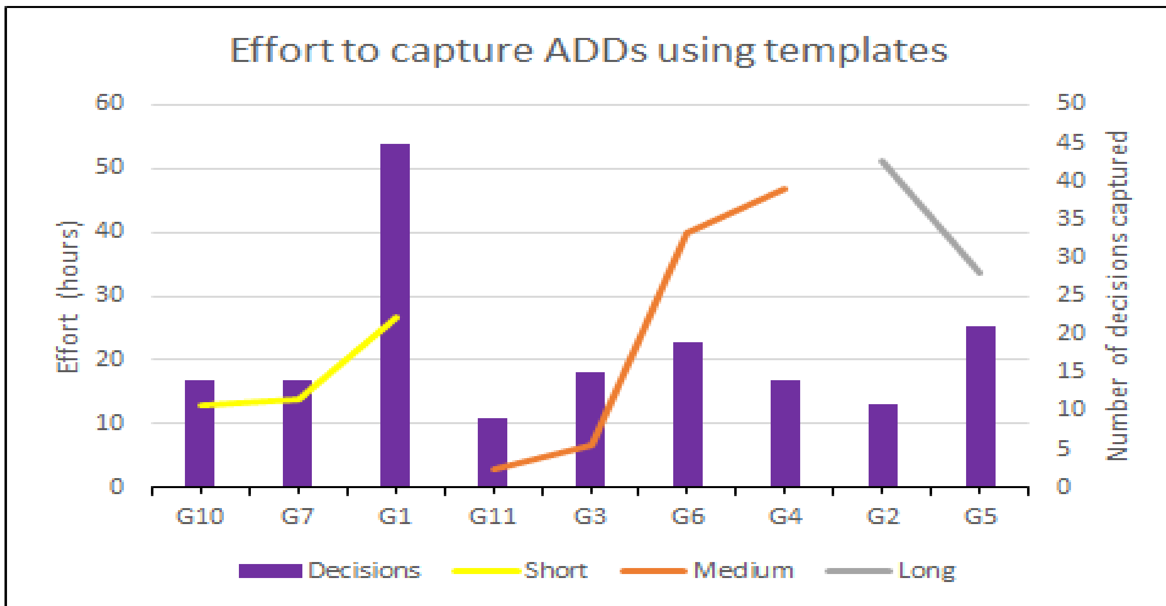

Figure 4: Capturing effort architecture design decisions using three different templates

With regard to the capturing effort, we measured the effort spent by the groups in capturing the different number of alternative decisions. The results revealed that groups capturing between two and four design alternatives also required less effort than those capturing more than four (Figure 5). As a result, an increase in the number of decision alternatives revealed an exponential increase in the time taken to make, deliberate and capture more alternative decisions, which may have a definite impact on agile projects [Martin, 2003]. Similarly, the results highlight that groups G2 and G10 exhibit lower values in capturing effort when they captured decisions with more than four alternatives compared to decisions that only included two and four alternative decisions. This suggests that because they capture fewer decisions with more than four alternatives than those between two and four the overall number of decision points captured is lower and hence, the overall capturing effort.

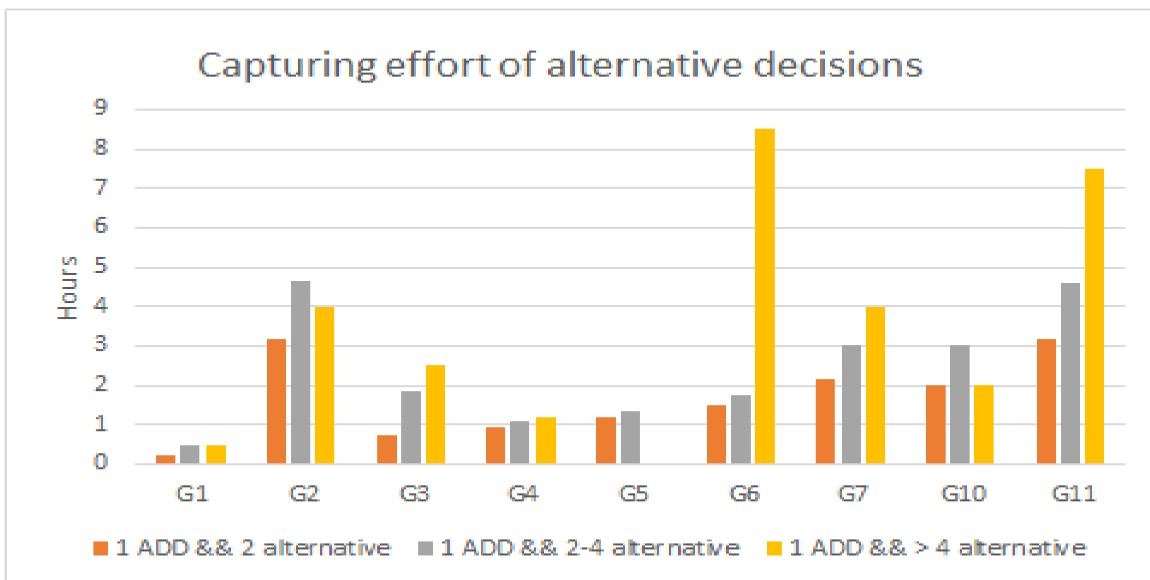

Figure 5: Capturing effort of alternative decision per group

Figure 6 shows the total effort spent in capturing decisions with a different number of alternative decisions. The results reveal that time increases accordingly with the number of alternatives considered during the decision making and evaluation activity.

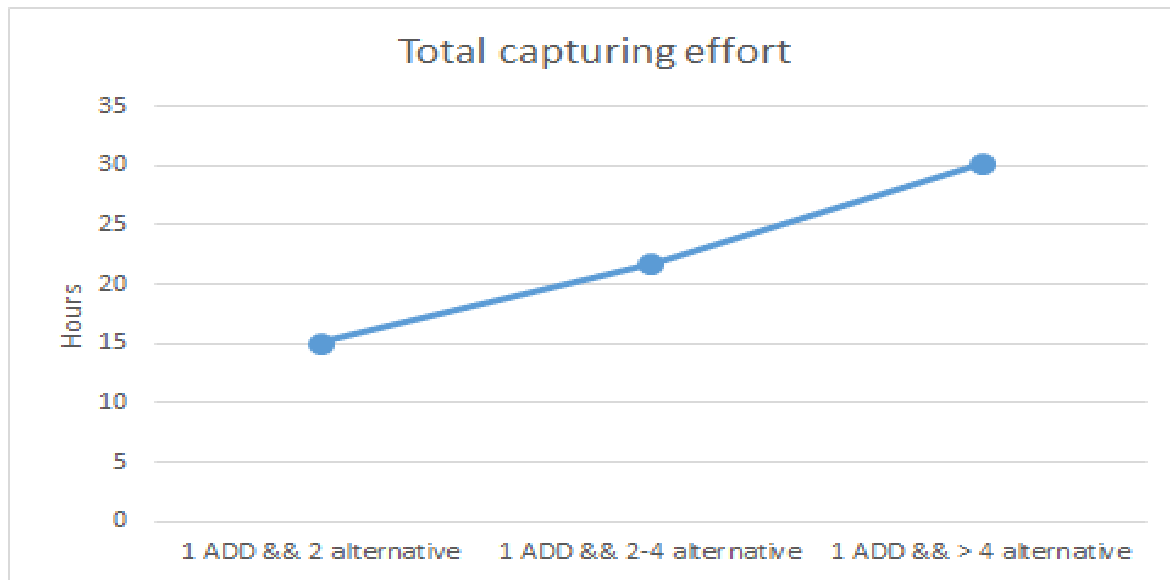

Figure 6: Total time spent in capturing decisions with different number of alternatives

The results also highlighted that groups reused Enterprise architecture design patterns suitable for the target problem, so many of the decisions made were technically sound and grounded on established knowledge.

The software architecture of a system is the product of a set of architectural design decisions. However, the knowledge of the architectural design decisions of a software-intensive system is easily lost, which leads to increased maintenance and evolution costs, and design erosion [Tofan, Galster and Avgeriou, 2011]. The benefits of capturing, sharing, and reusing AK is widely acknowledged in mitigating architectural knowledge vaporiszation [Ali Babar et al., 2009]. While the field of architectural knowledge (AK) management can be traced back to the early 90's and has resulted in a range of models, approaches, and research tools the cost of capturing relevant knowledge has been a significant barrier to widespread adoption. Critical to architectural sustainability is capturing decision viewpoints and their rationale as first-class elements of architectural descriptions. The ability to understand architectural design decision is crucial for the evolution of the system in measuring how sustainable the AK is during architectural changes and can provide an estimation of the maintenance and documentation effort needed when new requirements trigger new decisions.

5. Metrics to Estimate Sustainability

The potential loss of quality of a system must be estimated using appropriate indicators and metrics that can smell that the quality is decreasing during evolution cycles. The appearance of the different dimensions of technical debt [McConnell, 2007] [Fowler, 2009] as a quality indicator of suboptimal design decisions and coding practices, led software developers to keep this debt under control [Letouzey, 2012] to reduce the remediation cost of technical

debt management. Therefore, we need to discover the root and sources of the debt [Kazman, 2015] and those "hot-spots" [Mo, 2015] in long-living systems to be able to measure software sustainability both in architecture and code [Le, 2016]. A number of approaches have investigated modularity metrics as an indicator of technical debt [Liang, 2014], but many more quality attributes are often affected when the debt is not repaid. In this section our intention is not to provide an extensive list of the plethora of metrics that could be used individually or be combined to address sustainability or other quality indicators, but to highlight the different types of metrics according to different quality goals that could be chosen to estimate technical sustainability from different perspectives and abstraction levels.

Code and architecture metrics: Software metrics used to estimate the quality of systems by examining defects in source code are the most popular approaches to identifying "code smells" and increasing the maintainability of system [Briand, 1993]. However, there is a plethora of code metrics that can be used to estimate different types of defects and bad programming practices affecting important quality attributes such as complexity, maintainability, and reusability etc. Metrics that are employed to estimate the complexity, coupling, cohesion and dependencies between modules are commonly combined to provide meaningful indicators on the quality of code and to estimate the technical debt ratios. Koziolek et al. [2011] suggested a number of more than forty metrics that could be used to estimate software sustainability of software architecture including evolution concerns. Further work by Koziolek et al. [2013] limited the number of metrics to twelve. Garcia et al. [2009] introduced the concept of "*architectural bad smell*" (i.e., an anti-pattern or inadequate design practice that violates a well-established design principle) exemplified by anti-patterns, architectural mismatch, defects and code smells that impact on the quality of designs, detailing four of these architectural smells as a class of cumulative design problems that lead to architectural refactoring. Studying the evolution of architectural decay is a primary concern for the maintainability of systems, as highlighted by the study with open source systems described in Le et al. [2015].

Architecture knowledge metrics: Metrics to estimate the sustainability of architectural knowledge and the design decisions captured is still an emerging area of research where only a small number of metrics have emerged. For example, Zimmermann [2015a] suggests counting the number of design problems solved and the number of options considered per problem as an architectural decision metric. A more elaborate version was suggested by Capilla et al. [2017] who proposed a taxonomy of quality attributes and metrics for maintaining and evolving the design decisions with sustainability in mind. They state that complexity of the decision networks can help to estimate its sustainability based on the granularity of the design decisions and the number of traceability links among them. They also suggest the estimation effort for capturing AK items, such as the results of the study described in Section 4 [Carrillo, 2017], as another form to compute how much AK should be captured to make the set of design decisions sustainable during maintenance and evolution cycles. Finally, metrics to compute the impact of changes in the design decisions are based on ripple effect, instability and change-proneness metrics [Ampatzoglou, 2015]. For example, Ampatzoglou et al. [2015] use instability and change-proneness metrics to estimate the stability of design patterns (i.e. stability in this context is defined as the resistance of a software system to the ripple effect), where the resistance of classes to changes helps to determine the stability of the classes participating in a change. These early

attempts at the estimation of the sustainability of AK highlight that this is an open and fruitful area of research.

Aggregated metric sets: A combination of metrics often has a more significant impact to provide more accurate quality indicators when we estimate the sustainability of the system's architecture. For instance, in a renewed study, Koziolek et al., [2015] categorise software metrics ranging from modularisation to volatility with their potential to impact on estimating architecture sustainability. Similarly, Le et al., [2016] suggest other combinations of metrics and introduce three new metrics (i.e., BDCC: Bi-Directional Component Coupling, ASD: Architectural Smell Density, ASC: Architectural Smell Coverage) to understand technical sustainability and estimate when architecture start to erode or decay. They state in order to relate architectural qualities and smells, the combination of Concern Diffusion over Architectural Components (CDAC) with Component-level Interlacing Between Concerns (CIBC) metrics is suitable to estimate the scattered parasitic functionality as a concern-based issue for maintenance, which has a definite impact on modifiability and reusability. However, one of the challenges is to find out which combination of metrics provides more meaningful indicators, as some tools only offer coarse-grained indicators about technical debt and other quality properties.

As an overview of metrics that could be potentially be used to estimate and understand the sustainability of systems at different levels of abstraction, we summarise a representative number of these metrics that can be combined or used in isolation to relate the qualities desired with the smells in Table 1. This classification is based on previous work [Koziolek, 2011; Koziolek et al., 2013; Le, 2016; Capilla et al., 2017], which summarises the seminal work on architecture level metrics. The information in Table 1 can serve as a guide on which metrics to combine in order to evaluate a particular quality attribute that may help to estimate technical sustainability. However, we do not provide specific relationships indicating which concrete metrics measure each particular quality attribute. It should be noted that conflicts between different metrics are not discussed in this table.

Table 1: Overview of software metrics that can be used to estimate architecture sustainability.

Architecture level metrics			
	Smells	**Metrics**	**Quality attributes**
Maintenance	Smells about ambiguous and unused interfaces, when functionality of modules are rather small or big and those smells concerning delegation of functionality	Module interaction index, Attribute hiding factor, API function usage index, Module Size Uniformity Index, Module Size Boundedness Index	Complexity, Modularity [Mitchell 2006], Analyzability, Effectiveness, Understandability
	Smells that effect to duplicate functionality and coupling between components	Clone detection, Coupling between object, Ratio of cohesive interaction, Modularization Quality	Reusability, Complexity, Modifiability, Modularity

	Smells where multiple components realise the same concern or a component implements an excessive number of concerns. Therefore, we can identify components with a suitable percentage of methods	Concern diffusion over architectural components, Component-level interlacing between concerns, Number of concerns per component, Well-sized Methods Index	Reusability, Modifiability, Understandability, Modularity
	We identify components with an excessive number of dependencies, cyclic dependencies and dependencies that crosscut layers	Cyclic dependency index, API function usage index, Layer Organization Index, Cumulative component dependency, Excessive structural complexity	Modularity, Understandability, Changeability Modifiability
	Other cross-cutting smells affecting any part of the architecture	Architectural smell coverage Architectural smell density	Cost
Evolution	Elements that change too often, Number of elements impacted by a change	Instability, Ripple effect, Distance from Main Sequence, Module Interaction Stability Index	Stability, Evolvability
	Likelihood of components that evolve together	Bi-directional coupling component	Complexity, Evolvability
Architecture knowledge level metrics			
Maintenance	Excessive number of decisions and trace links	NodeCount, EdgeCount	Complexity, Stability
	Too many AK ítems and decision alternatives	Cost of AK capturing effort	Cost
Evolution	A change impact on many decisions	Ripple effect, instability, change proneness	Changeability, Stability
	Obsolete decisions and frequent changes	Decision volatility	Timeliness

6. Sustainability in Academic Research and Practice

In addition to the increasing dependency of modern society on software in general, it also now plays a critical role in the advancement of knowledge, with the paradigm shift in research towards large-scale, data-intensive computational science and engineering [Hey, Tansley, and Tolle, 2009]. Software's increasing importance in the field of research has led for calls for it to be classified as a first-class, scientific instrument [Goble, 2014; Allen et al., 2017; Crick et al., 2017]. While Hettrick et al., [2014] demonstrated the importance of software in research -- 59% of respondents claimed that software was fundamental to their research -- the study highlighted that 56% of researchers developed code with 21% of them having no training in software development. As a result, this raises serious questions

regarding the overall quality of the software per se, the potential implications on the reliability and validity of the research output, as well as the sustainability of essential codebases critical to the research communities.

The use of metrics to estimate software architectural sustainability serves to highlight the presence of a range of underlying issues. The correct use and interpretation of these metrics - and subsequent identification and resolution of the causes of sustainability problems - requires appropriate knowledge and expertise in developing useful and usable software artefacts. Progress in scientific research is dependent on the quality and accessibility of software at all levels, and it is now critical to address the variety of challenges related to the development, deployment, and maintenance of reusable software. Despite significant and continuing efforts across academia, from undergraduate students through to senior academics and research leaders, to draw attention to the importance of developing and maintaining software that underpins a growing proportion of the research base, there is increasing recognition that a lack of software development skills exists in the education and research pipeline that needs to be addressed [Brown et al., 2014; Murphy et al., 2017].

A critical issue in research is that methods used to generate research output are required to continually adapt to an evolving research agenda, potentially discipline-specific, where new hypotheses are generated and subsequently tested. Thus, the software that underpins that research is consequently required to evolve with that agenda. This continual change presents a significant challenge to developing sustainable software in itself, and one that is exacerbated by a lack of relevant education, skills and professional development for researchers. However, issues of sustainability, or indeed architecture sustainability and associated metrics and dimensions, are rarely encountered or highlighted as important factors for developing useful and usable software. What is required is a partnership between researcher and software experts, to ensure that the software not only meets its initial research requirements but can evolve as those requirements change over the lifetime of a research project.

The emergence of international community-driven initiatives such as WSSSPE[5] (*Workshop on Sustainable Software for Science: Practice and Experiences*) promotes sustainable research software by addressing challenges related to the full lifecycle of research software through shared learning and community action have catalysed change, especially in the broader context of open science and research. In October 2016, a Knowledge Exchange Workshop on Research Software Sustainability [Hettrick, 2016] brought together key European stakeholders that deliver infrastructure and services for higher education and research, identifying five recommendations to improve the sustainability of research software:

1. Academics must raise awareness of the fundamental role of software in research;
2. Research software should be recognised as a valuable research object in line with the investment it receives and the research it makes possible;
3. Funders should use their position to promote software sustainability;

[5] http://wssspe.researchcomputing.org.uk/

4. Skills related to software sustainability must be embedded in the research community;
5. The creation of organisations (centralised or distributed) to act as focal points for software sustainability expertise.

A number of initiatives have emerged to address some of these issues such as Software Carpentry, which aims to teach the fundamental computational skills to doctoral and postdoctoral researchers to fill this skills gap [Wilson, 2016]. Founded in 1998, Software Carpentry has evolved from a US National Laboratories course into a global volunteer-based effort, which focuses on practical two-day courses teaching attendees task automation, program design and version control, along with techniques that are considered standard practice in the software industry. Similarly, in 2014 the increasing need for researchers to improve their skills in managing and analysing data led to the development of Data Carpentry, a sibling organisation that adopts the same two-day course approach, but focused on research data. These initiatives have met with significant demand and uptake worldwide (for example, 400 workshops for over 12,000 researchers between January 2013 and July 2015) and have made notable gains in addressing the lack of skills endemic within the research community. However, these are retrospective initiatives to address a problem that -- in the longer term -- should ideally be handled much earlier in the career path of researchers and practitioners. We, therefore, need to consider education and skills development, especially for undergraduate students across computational science and engineering disciplines.

The issue of recruiting software expertise in academia is further exacerbated through a ‘hotchpotch' of different solutions that have been developed to meet the disparate needs of local human resources and finance departments, university culture and restrictions from funders' [Hettrick 2016]. The lack of a formal career path -- with software experts associated with careers unrepresentative of their work -- makes it difficult to recruit and retain such valuable staff. In the UK, a powerful disincentive to employ software experts is that they do not attract overheads on funding bids, so universities are unable to recover the full economic cost for such positions. In addition, funder review panels are biased against recruiting software experts over "traditional" researchers. In turn, this leads to a lack of these experts in senior roles that would function to raise recognition of the role and effect cultural and institutional change [Brett et al., 2017]. Thus, what is needed is a recognised career path for these Research Software Engineers (RSEs), which would enable overheads to be recouped and their efforts gauged against metrics appropriate to their role (as opposed to just publications). This would also raise awareness and recognition for the role, and help persuade review panels of the legitimacy of recruiting software experts. A supported recruiting, and retention process would increase access to skills fundamental to modern research while improving the sustainability and reproducibility of the software that underpins that research. Research software Engineering is far from a niche community: the RSE State of the Nation report [Brett et al., 2017] estimates there are between 1,000-14,000 RSEs in the UK, although it was acknowledged that due to software experts being ‘hidden' in other roles, an upper limit is hard to estimate.

Beyond support for the RSE role itself, support from institutions is needed for their activities. To address this need, a new model for organising university software expertise has

emerged: the Research Software Group. Different research groups will have different requirements for software development, so recruiting a permanent Research Software Engineer within that group is not the answer for every situation. Where larger researcher groups with a high degree of software development may employ many RSEs, those small groups with requirements for less development may seek to use such expertise on a short-term basis only, or leverage the expertise of centralised groups of these engineers. Thus, Research Software Groups permanently employ RSEs who collaborate with researchers. Through funding a core capability and leadership team via *top-slicing* research funding, additional RSEs are then funded through paid-for services. This model allows the group to develop and grow over time. University College London pioneered this model, and over a five-year period has grown its team from three centrally funded posts to six grant-funded posts. Other research software groups have subsequently emerged internationally across institutions and universities[6].

More broadly, there is the need for national support for research software communities. Founded in 2010, the Software Sustainability Institute is a partnership of the universities of Edinburgh, Manchester, Oxford and Southampton [Crouch, 2014]. Initially funded by the UK's Engineering and Physical Sciences Research Council (EPSRC), its second phase of funding in 2015 has also attracted funding from two other UK research funding councils, namely the Biotechnology and Biological Sciences Research Council (BBSRC) and the Economic and Social Research Council (ESRC). The establishment of an institute highlights the growing recognition and support for cultivating sustainable software development across a range of academic disciplines in the UK, which was the first country to invest in such an organisation. EPSRC is also supporting the community and role of RSEs through a number of Research Software Engineer Fellowships, starting with a pilot call in 2015 and investing up to £3m for five-year Fellowships, awarded to "exceptional individuals with combined expertise in programming and a solid knowledge of the research environment"[7]. This fellowship scheme has led to the formation of new RSE groups at the Universities of Sheffield and Bristol.

The Software Sustainability Institute is organised into five teams, to address the broad issue of software sustainability within the UK. In addition to a Communications team, the Community team runs workshops and a fellowship programme. The Training team runs training events and is the UK administrator for Software and Data Carpentry workshops. The Policy team investigates community issues related to software and runs campaigns to raise awareness and solve these issues. The Software team helps to assess and improve research software and practices directly. Given the scale of potential work in the area of software improvement, the Software team periodically runs an "Open Call for Projects" which reviews and prioritises applications for selection, with a key criterion being the potential benefit to the broader research community. The assessments take the form of a software evaluation, which analyses the software itself, the use of development and operational infrastructure, and the processes used to develop the software and manage that development, and is supplied as an experience-based report of observations and recommendations for improvement. An analysis of the areas in which the Software team

[6] http://rse.ac.uk/community/international-rse-groups/
[7] https://www.epsrc.ac.uk/funding/calls/rsefellowships/

assisted with Open Call projects from May 2013 to October 2015 is shown in Figure 7, representing a total of 19 projects from the first five rounds of the Institute's Open Call.

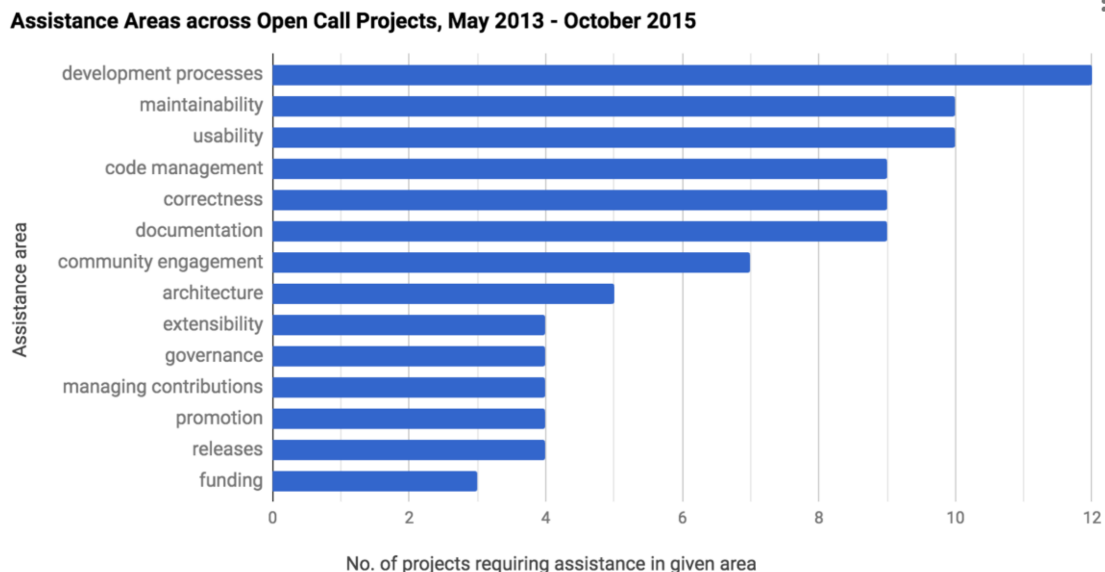

Figure 7: Areas that the Institute Software team assisted with Open Call Projects that were undertaken from May 2013 to October 2015

These results are typical of the issues commonly highlighted by the research community[8], and serve to reinforce the core cultural and technical problems already identified: namely, the need to address the lack of skills in the areas of research software development, deployment (and subsequent use), and maintenance of reusable software. As previously highlighted, there is a relationship between architectural issues and other problems, such as code maintainability, modifiability (in part, extensibility), and system evolution (which is related to many of these areas). However, looking at architecture directly, aside from aspects of process and supporting infrastructure (i.e. development processes, code management and community engagement), and focusing on areas of improvement that relate to the technical properties of the software, we can see that software architecture features within the top five assistance areas, featuring as an issue in over a quarter of projects.

Beyond the Software team, the Institute's Policy team has campaigned to raise awareness of the RSE role and formed a community of RSEs that became the UK RSE Association[9]. Launched in 2014, the Association is a democratic organisation that campaigns for 'the recognition and adoption of the RSE role within academia along with the need for appropriate reward and career opportunities for RSEs'. The RSE State of the Nation report [Brett et al., 2017] indicates there are 780 RSE Association members as of February 2017.

Finally, the Knowledge Exchange workshop report [Hettrick, 2016] highlights that other models to approach national support for sustainability exist, e.g. in the Netherlands, an

[8] Note that 'funding' in this figure refers to help provided by the Software team to projects concerning how to fund specific development tasks, not the much broader community issue of how to fund RSEs
[9] http://rse.ac.uk/

approach involving shared responsibility across the DANS[10] and SURFsara[11] centres are being considered. The report also highlights broad agreement at the workshop for an EU-level organisation that should be investigated, acting as an 'umbrella group that shares expertise across national organisations and campaigns for software sustainability at an EU level'. The impact of the various national and international community initiatives have catalysed awareness and cultural change of software sustainability within and outside of academia, permeating into broader professional and industrial practice through new models, methodologies and metrics. However, there is still significant work to be done, particularly for software architecture sustainability, as well as building stronger translation, adoption and embedding of academic research into real-world practice.

7. Outlook and Future Directions

Modern societies are dependent upon complex software systems that operate continuously in highly connected and distributed ecosystems with emergent behaviour, that are difficult to change (i.e. brittle), and tend to break in multiple places when a single change is made (i.e. fragile). This environment results in challenging requirements for availability, resilience, and sustainability. As a result of changing stakeholder requirements, software systems are the product of accidental complexity that presents significant challenges with regards to maintenance and evolution. While Brooks [1986] argues that complexity is an essential property of software it should not be accidental; this presents significant challenges and threats to ensure the dependability and longevity of software systems.

The concept of sustainability has emerged as a growing area of interest in the field of software engineering to address the challenges of designing software systems that can endure. This raises the question, what is the most efficient and effective method or approach for managing change and evolution regarding a software system? Sustainable software requires a solid foundation that allows efficient and effective maintenance and evolutionary change over its entire life-cycle. Software architectures are the foundation of any software system and provide the mechanism for reasoning about key software quality attributes. Moreover, it is essential to understand and, if needed, minimise the effects software systems under design have on their socio-technical environment over time.

This article argues that software architectures are fundamental to the development of sustainable software since they manifest the major design decisions that determine a system's initial development and deployment and its evolutionary change. The rationale for this position is 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 etc. None of these can be achieved without a unifying architectural vision. Software architectures strongly influence sustainability, because they affect how developers can understand, analyse, extend, test and maintain a software system. As a result, software architectures the blueprint of how the software system will be built; they hold the key to post-deployment system understanding, maintenance, and evolution. This suggests that software

---

[10] http://dans.knaw.nl/en

[11] http://surf.nl/en/about-surf/subsidiaries/SURFsara

architectures are fundamental to achieving software sustainability if they can be cost-efficiently maintained and evolved over their entire life-cycle. Since its inception as a discipline, the field of software architectures has implicitly focused on the concept of sustainability. However, in the last decade research has increasingly focused explicitly on the concept of architectural sustainability.

It is argued that sustainability should be an explicit consideration in the design of a system, even when the primary purpose of the software system is not sustainability [Becker et al., 2014]. The principal aim of this paper was to explore emerging work to provide the theoretical foundation to support this position by examining some of the recent trends in software engineering and software architectures in relation to the concept of sustainability.

The primary focus of research in the field of software engineering with regards to sustainability has focused on reference models to develop sustainable software and approaches for evaluating sustainability. At their heart are a set of metrics that measure the degree to which a software system exhibits some property and are widely considered as an approach to achieving higher quality software. However, a plethora of software metrics have been proposed to estimate code complexity including cyclomatic complexity, Halstead metrics, source lines of code (SLOC), Fagan inspection, defect counting, etc. which has resulted in a software quality "quagmire" [Voas and Kuhn, 2017].

Considered as a broader quality attribute, architecture sustainability can unfold in other quality attributes including maintainability (i.e. analyzability, stability, testability, understandability), modifiability, portability, and evolvability. We suggest that at the very minimum, software sustainability should address two core quality attributes: maintainability and extensibility. However, to what extent existing metrics and measures of the quality attributes defined within existing standards are appropriate for measuring a software artefact's technical sustainability is an open research question that provides further avenues for research. Similarly, a number of architecture-level metrics have also been proposed to estimate and understand the sustainability of systems at different levels of abstraction. However, many were based on plausibility and had yet to be systematically validated. A key issue in assessing the value of software metrics is whether they support decision-making. As a result, which are the most appropriate architectural-level metrics to analyse the sustainability of software architectures is an open research question. In addition, how to make software sustainable both in terms of the software artefact, 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.

Architectural-level code metrics frameworks have led to an improvement in the overall code quality at the design level because measurement instruments are in place. This suggests that while software maintenance remains a challenge, assessments can be conducted with limited effort through regular assessment code can be enhanced 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. Similarly, while a number of existing methods exist to assess sustainability, they do not provide sufficient support for the systematic analysis of ripple effects, or the integration with

reverse engineering tools and knowledge management support. How this can be achieved in practice is an open research area.

One approach to designing software architectures that are sustainable is adherence to established design principles enshrined in reference architectures, which embody the wisdom of reusable architectural knowledge of key design decisions and provide a common vocabulary and template solution for an architecture for specific domains or a family of software systems. As a consequence, significant benefits could be achieved, including a reduction in the cost and effort related to software maintenance and evolution. An open research issue is how to re-architect existing reference architectures for sustainability, which will subsequently affect the sustainability of derived architectures. A key factor in the continued long-term existence of reference architectures is that their communities sustain them. However, despite their apparent value in providing a foundation for the design of derived systems architectures the significant increase in the amount of documentation of new versions presents a considerable threat to their adoption by the wider community.

As the foundation of software systems, software architectures encompass the architects' and stakeholders' strategic decisions, often made in unsystematic and undocumented manners. This can lead to architectural drift and erosion, a decrease in software quality, and in turn increased costs and dissatisfied stakeholders. Software architectures not only comprise a system's structure but essential design decisions based on architectural knowledge. As such, software architectures are the product of architectural-level reasoning and manifest design decisions, which strongly influence the longevity of systems and their architecture. How to make informed and systematic design decisions is one of the grand challenges in software engineering. Numerous approaches have been developed by the research community over the last decade in relation to Architectural Knowledge and its management, but these methods have not yet found widespread adoption in practice. It is suggested this stems from the significant and ongoing effort required to efficiently store large quantities of codified knowledge that can not only be difficult to manage but also to use effectively. To achieve sustainable architectures requires capturing significant sustainable design decisions and their rationale as failure to do so can lead to decision rationale erosion. A key challenge is how to capture a minimalistic set of salient design decisions combined with lightweight approaches that can succinctly reduce the documentation. How this can be achieved in practice is unclear and is an area ripe for research.

The challenges related to the development and maintenance of reusable software for science and engineering are a growing concern, as many scientists' research is increasingly dependent on the quality and availability of software upon which their work is based. The development of software in the field of computational science and engineering research by a significant proportion of researchers with no or a minimal educational background in software engineering principles and practice is alarming as it raises serious concerns regarding the overall quality of the software per se, as well as having severe implications for the reliability and validity of the research output. In a number of instances, the consequences of poorly designed software have led to retractions of scientific papers [Miller, 2006]; it is essential that researchers be able to learn and adopt software-related skills and methodologies. While a number of initiatives exist to teach fundamental software engineering skills, the limited exposure to fundamental software engineering knowledge

cannot bridge the chasm. A key challenge is how to train and educate the broad spectrum of domain scientists or advance their skills to develop software that is sustainable. As such, the education sector as a whole has an important role to play in ensuring that software designers of the present and future fully understand the concept of sustainability and its integral relationship to the field of software engineering by bridging the chasm between domain scientists and software engineering theory, principles, and practice. Similarly, educators need to consider how to integrate sustainability into software engineering curricula and articulate the competencies required for successful sustainability design. A 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. This is particularly true of software developed in academic environments, where the concepts of software architectures are often merely accidental.

Architecture sustainability is the capacity of a software architecture to endure different types of change through efficient maintenance and orderly evolution over its entire lifecycle. 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 addressing sustainability in software systems. Software architectures are fundamental in understanding how the software system is built in the first instance; they are also essential to post-deployment system maintenance and evolution, which in turn leads to software that is sustainable. Garlan [2000] declared that while the field of software architectures had experienced considerable growth over the past decade in developing the technological and methodological base for treating architectural design as an engineering discipline much remained to be done. Today, the changing face of technology presents a number of new challenges for the field of software architecture. This article provides a foundation of emerging research themes in the area of sustainable software architectures highlighting recent trends, and open issues and research challenges.

Acknowledgements

Part of this work is supported by the Brazilian funding agency FAPESP (Grant: 2017/06195-9), and the Canadian NSERC through RGPIN2016-06640.

8. References

A. Allen, C. Aragon, C. Becker, J. Carver, A. Chis, B. Combemale, M. Croucher, K. Crowston, D. Garijo, A. Gehani, C. Goble, R. Haines, R. Hirschfeld, J. Howison, K. Huff, C. Jay, D. S. Katz, C. Kirchner, K. Kuksenok, R. Lämmel, O. Nierstrasz, M. Turk, R. van Nieuwpoort, M. Vaughn, J. J. Vinju, “Engineering Academic Software (Dagstuhl Perspectives Workshop 16252),” Dagstuhl Manifestos, vol. 6, no. 1, pp. 1–20, 2017.

D. Ameller, C. Farré, X. Franch, D. Valerio and A. Cassarino, "Towards continuous software release planning," 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, pp. 402-406, 2017.

R. Amri and N. B. B. Saoud, "Towards a Generic Sustainable Software Model," Fourth International Conference on Advances in Computing and Communications, Cochin, pp. 231-234, 2014.

A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou. The effect of gof design patterns on stability: A case study. IEEE Transactions on Software Engineering, 41(8):781–802, 2015.

P. Avgeriou, M. Stal, R. Hilliard, Architecture sustainability. IEEE Software, 30(6), 40-44, 2013.
M. Ali Babar, T. Dingsyr, P. Lago, and H. van Vliet, Software architecture knowledge management: Theory and practice. Springer, 2009.
L. Bass, P. Clements, R. Kazman, Software architecture in practice. 3rd ed. Addison-Wesley, 2012.
C. Becker, R. Chitchyan, L. Duboc, S. Easterbrook, M. Mahaux, B. Penzenstadler, G. Rodriguez-Navas, C. Salinesi, N. Seyff, C. C. Venters, C. Calero, S. Akinli Kocak, S. Betz. The Karlskrona manifesto for sustainability design. 2014. Available at: http://sustainabilitydesign.org/.

C. Becker, R. Chitchyan, L, Duboc, S. Easterbrook, B. Penzenstadler, N. Seyf, and C. C. Venters. 2015, May. Sustainability design and software: the Karlskrona manifesto. In IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE) (Vol. 2, pp. 467-476), 2015.

C. Becker, S. Betz, R. Chitchyan, L. Duboc, S. M. Easterbrook, B. Penzenstadler, N. Seyff, and C. C Venters. Requirements: The key to sustainability. IEEE Software, 33(1), 56-65, 2016..

S. Betz, C. Becker, R. Chitchyan, L. Duboc, S.M. Easterbrook, B. Penzenstadler, N. Seyff, C. C. Venters, Sustainability Debt: A Metaphor to Support Sustainability Design Decisions, Fourth International Workshop on Requirements Engineering for Sustainable Systems, RE4SuSy, CEUR Workshop Proceedings, pp. 45-54, 2015.

G. Booch, @Grady_Booch. "I do not fear the rise of super intelligent AI as do Stephen, Bill, & Elon; what I do fear is the fragile software on which society relies. " 2015.

J. Bosch, Continuous Software Engineering. Springer, 2014.

A. Brett, M. Croucher, R. Haines, S. Hettrick, J. Hetherington, M. Stillwell, C. Wyatt. Research Software Engineers: State of the Nation Report. Engineering and Physical Sciences Research Council. DOI: 10.5281/zenodo.495360, 2017.

L. Briand, S. Morasca, and V. Basili. Measuring and assessing maintainability at the end of high level design. In Proceedings of the Conference on Software Maintenance (ICSM), 1993.

F. P. J. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering," IEEE Computer, 20(4), pp. 10-19, 1986.

N. C. C. Brown, S. Sentance, T. Crick and S. Humphreys. Restart: The Resurgence of Computer Science in UK Schools. ACM Transactions on Computer Science Education, 14(2), 1–22, 2014

G. H. Brundtland and UN World Commission on Environment and Development, Our common future. Oxford University Press, 1987

J. Cabot, S. Easterbrook, J. Horkoff, L. Lessard, S. Liaskos, J. Mazón. Integrating sustainability in decision-making processes: A modelling strategy. ICSE: 31st International Conference on Software Engineering. pp: 207-210, 2009.

C. Calero, M. A. Moraga, and M. F. Bertoa. “Towards a software product sustainability model,” WSSSPE1: First workshop on sustainable software for science: practice and experiences, SC’13, 17 Denver, CO, USA, 2013.

C. Carrillo, R. Capilla, O. Zimmermann, and U. Zdun. Guidelines and metrics for configurable and sustainable architectural knowledge modelling. In Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSA’15, pp. 1–5. ACM, 2015.

C. Carrillo, "A Sustainable for Architectural Design Decisions Management". Doctoral Thesis in Systems Engineering and Services for the Information Society, Technical University of Madrid (UPM), Madrid, Spain, 2017.

R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou and J.M. Küster. An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artefacts in the Software Engineering Lifecycle. ECSA 2011, LNCS 6903, pp. 303-318, Springer-Verlag Berlin Heidelberg 2011.

R. Capilla, A. Jansen, A. Tang, P. Avgeriou, M.A. Babar, 10 years of software architecture knowledge management: Practice and future. Journal of Systems and Software 116(6), 191-205, 2016.

R. Capilla, E.Y. Nakagawa, U. Zdun, C. Carrillo, Toward Architecture Knowledge Sustainability: Extending System Longevity. IEEE Software 34(2), 108-111, 2017.

V. Cerf. A Brittle and Fragile Future. Communications of the ACM, 60(7), 2017

T. Crick, B. A. Hall and S. Ishtiaq. Reproducibility in Research: Systems, Infrastructure, Culture. Journal of Open Research Software 5(1), 2017.

R. N. Charette. This Car Runs on Code. IEEE Spectrum, 2009

R. Chitchyan, C. Becker, S. Betz, L. Duboc, B. Penzenstadler, N. Seyff, and C. C. Venters, "Sustainability design in requirements engineering: State of practice," in Proceedings of the 38th International Conference on Software Engineering Companion, ser. ICSE-SEIS '16, pp. 533–542, 2016.

S. Crouch, N. Chue Hong, S. Hettrick, M. Jackson, A. Pawlik, S. Sufi, L. Carr, D. De Roure, C. Goble, M. Parsons. "The Software Sustainability Institute: Changing Research Software Attitudes and Practices," Computing in Science & Engineering, vol.15, no.6, pp.74,80, 2014.

Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, R. Weiss. Sustainability guidelines for long-living software systems. In Proc. 28th IEEE Int. Conf. on Software Maintenance (ICSM'12), Industry Track, IEEE DL, 1-10, 2012.

B. Fitzgerald and K-J. Stol. Continuous software engineering: A roadmap and agenda. Journal of Systems and Software, Vol. 123, pp. 176–189, 2017.

M. Fowler, "Technical debt quadrant,". Available: http://martinfowler.com/bliki/TechnicalDebtQuadrant.html, 2009.

D. Garlan. "Software architecture: a roadmap". In: ICSE: Proceedings of the Conference on The Future of Software Engineering (ICSE '00). ACM, New York, NY, USA, 91-101, 2000.

J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying Architectural Bad Smells. In 13th European Conference on Software Maintenance and Reengineering, pages 255–258, 2009.

J. Garcia, I. Ivkovic and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 486-496, 2013.

S. Giesecke, J. Friebe, M. Frenzel, Long-term software architecture management with multi-technology tool support. In: 15th European Conference on Software Maintenance and Reengineering (CSMR 2011), p p. 321-324, 2011.

C. Goble, Better software, better research, IEEE Internet Computing, 18 (5), pp. 4–8, 2014.

I. Groher and R. Weinreich, "An interview study on sustainability concerns in software development projects," in Proceedings of the 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA '17, 2017.

S. Hettrick. It's impossible to conduct research without software, say 7 out of 10 UK researchers, 2014. Available at: https://www.software.ac.uk/blog/2016-09-12-its-impossible-conduct-research-without-software-say-7-out-10-uk-researchers

S. Hettrick. Research Software Sustainability: Report on a Knowledge Exchange Workshop, 2016. Available at: https://www.software.ac.uk/attach/Research_Software_Sustainability_Report_on_KE_Workshop_Feb_2016_FINAL.pdf

T. Hey, S. Tansley, K. M. Tolle, The fourth paradigm: data intensive scientific discovery. Microsoft Research Redmond, 2009.

L. M. Hilty, B. Aebischer. "ICT for sustainability: An emerging research field." *ICT Innovations for Sustainability*. Springer International Publishing, pp. 3-36, 2015.

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

C. B. Jaktman, J. Leaney, M. Liu, "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study", 1st Working IFIP Conference on Software Architecture (WICSA), Kluwer Academic Publisher, 1999, pp. 455-470.

A. Jansen, A. Wall and R. Weiss. “TechSuRe: A method for assessing technology sustainability in long lived software intensive systems,” SEAA 2011: Proceedings of the 37th EUROMICRO Conference on software engineering and advanced applications, Oulu, Finland, 2011.

J. Kasurinen, M. Palacin-Silva, and E. Vanhala, “What concerns game developers?: A study on game development processes, sustainability and metrics,” in Proceedings of the 8th Workshop on Emerging Trends in Software Metrics, WETSoM ’17, pp. 15–21, 2017.

R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, A.Shapochka, A Case Study in Locating the Architectural Roots of Technical Debt, ICSE’15, IEEE CS, 179-188, 2015.

D. Kim, J. Ryoo, S. Kim, Building sustainable software by preemptive architectural design using tactic-equipped patterns, International Conference on Availability, Reliability and Security, ARES,484-489, 2014.

R. Kitchin and M. Dodge, Code/space: Software and everyday life. MIT Press, 2011.

H. Koziolek, Sustainability evaluation of software architectures: a systematic review. 7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS, ACM DL, 3-12, 2011.

H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst, R. J. Weiss, Morphosis: A lightweight method facilitating sustainable software architectures. In: 10th Joint Working Conference on Software Architecture (WICSA 2012) 6th European Conference on Software Architecture (ECSA 2012), pp. 253-257, 2012.

H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst. Measuring architecture sustainability. IEEE Software, 30(6),54-62, 2013.

H. Koziolek. Sustainability evaluation of software architectures: A systematic review. In CM SIGSOFT Symp.—ISARCS on Quality of Software Architectures—QoSA and Architecting Critical Systems—ISARCS, QoSA’115, 2015.

B. Knowles et al., Exploring sustainability research in computing: Where we are and where we go next, in UbiComp. ACM, pp. 305–314, 2013.

P. Kruchten, H. Obbink, and J. Stafford, “The past, present, and future for software architecture,” IEEE Software, vol. 23(2), 22–30, 2006.

P. Lago, S. A. Koçak, I. Crnkovic, B. Penzenstadler. Framing sustainability as a property of software quality. Communications of the ACM, 58(10), 70–78, 2015.

D. M. Le, C. Carrillo, R. Capilla, N. Medvidovic. Relating Architectural Decay and Sustainability of Software Systems, 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, IEEE pp. 178-181, 2016.

D. M. Le. Architectural-based speculative analysis to predict bugs in a software system. In Proceedings of the 38th International Conference on Software Engineering, ICSE ’16, 2016.

D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In Proc. Mining Software Repositories, 2015.

J. Letouzey and L. Ilkiewiczr. Managing technical debt with the sqale method. IEEE Software, 29(6), 44–51, 2012.

M. M. Lehman, Software's future: Managing evolution, IEEE Software, vol. 15(1) pp. 40–44, 1998.

Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou. An empirical investigation of modularity metrics for indicating architectural technical debt. In 10th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA'14, 2014.

I. Macia Bertran, R. Arcoverde, A. Garcia, C. Chávez, and A. von Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. In 5th European Conference on Software Maintenance and Reengineering, CSMR'12, pp. 277–286, 2012.

M. Mahaux, P. Heymans, and G. Saval, "Discovering sustainability requirements: an experience report," in Intl. Working Conf. REFSQ, pp. 19–33, 2011.

S. S. Mahmoud and I. Ahmad, "A green model for sustainable software engineering," Intl. Journal of Software Engineering and Its Applications, vol. 7(4), pp. 55–74, 2013.

I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in Proceedings of the 38th International Conference on Software Engineering, ser. ICSE '16, pp. 237–248, 2016.

S. Martínez-Fernández, C. P. Ayala, X. Franch, E. Y. Nakagawa, A Survey on the Benefits and Drawbacks of AUTOSAR, 12th Working IEEE/IFIP Conference on Software Architecture Workshop, WICSAW, pp. 19-26, 2015.

R. C. Martin. Agile software development: principles, patterns, and practices. Prentice Hall PTR, 2003.

S. McConnell, "Technical debt," available: http://www.construx.com/10x Software Development/Technical Debt/, 2007.

G. Miller, "A Scientist's Nightmare: Software Problem Leads to Five Retractions," Science, vol. 314, no. 5807, pp. 314, 2006.

B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. IEEE TSE, 32(3), 193–208, 2006.

R. Mo, Y. Cai, R. Kazman, L. Xiao, Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells, 12th Working IEEE/IFIP Conference on Software Architecture, WICSA IEEE CS, pp. 51-60, 2015.

E. Murphy, T. Crick, and J. H. Davenport, An Analysis of Introductory Programming Courses at UK Universities. The Art, Science, and Engineering of Programming, vol. 1(2), no. 18, 2017

E. Y. Nakagawa, M. Guessi, D. Feitosa, F. Oquendo, J. C. Maldonado, Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures, 11th Working IEEE/IFIP Conference on Software Architecture, WICSA, pp. 143-153, 2014.

S. Naumann, M. Dick, E. Kern, and T. Johann. "The greensoft model: A reference model for green and sustainable software and its engineering," Sustainable Computing: Informatics and Systems, vol. 1(4), 2011.

The Oxford Dictionary of English. Oxford University Press, 2010.

L. Oliveira, K. R. Felizardo, D. Feitosa, E. Y. Nakagawa, Reference Models and Reference Architectures Based on Service-Oriented Architecture: A Systematic Review, 4th European Conference on Software Architecture, ECSA, pp. 360-367, 2010.

Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 1998. Architecture-based runtime software evolution. In Proceedings of the 20th International Conference on Software Engineering (ICSE '98). IEEE Computer Society, Washington, DC, USA, pp. 177-186, 1998.

B. Penzenstadler. "Towards a definition of sustainability in and for software engineering." Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, 2013.

B. Penzenstadler and H. Femmer, A generic model for sustainability with process-and product-specific instances, in Proceedings of the 2013 workshop on Green in/by software engineering. ACM DL, pp. 3-8, 2013.

O. Philippe, N. Chue Hong and S. Hettrick. Preliminary analysis of a survey of UK Research Software Engineers. 4th Workshop on Sustainable Software for Science: Practice and Experience, 2016.

J. L. Ramsey, On not defining sustainability, Journal of Agricultural and Environmental Ethics, vol. 28(6), 1075–1087, 2015.

Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, Markku Oivo, Continuous deployment of software intensive products and services: A systematic mapping study, Journal of Systems and Software, Volume 123, pp 263-291, 2017.

K. Roher and D. Richardson, “A proposed recommender system for eliciting software sustainability requirements,” in Workshop USER, pp. 16–19, 2013.

K. Roher and D. Richardson, “Sustainability requirement patterns,” IEEE Third International Workshop in Requirements Patterns (RePa), 2013 IEEE CS, pp. 8–11, 2013.

R. C. Seacord, et al. "Measuring software sustainability." International Conference on Software Maintenance. IEEE, 2003.

S. Sehestedt, C. Cheng, E. Bouwers, Towards quantitative metrics for architecture models. In: 11th IEEE/IFIP Conference on Software Architecture (WICSA 2014), pp. 51-54, 2014.

S. Sherman, I. Hadar, Identifying the need for a sustainable architecture maintenance process. In: 5th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE 2012), pp. 132-134, 2012.

I. Somerville, "Design for failure: Software challenges of digital ecosystems," 2007 Inaugural IEEE-IES Digital EcoSystems and Technologies Conference, Cairns, Australia, 2007.

J. A. Tainter, “Social complexity and sustainability,” Journal of Ecological Complexity, no. 3, pp. 91–103, 2006.

A. Taivalsaari and T. Mikkonen, "A Roadmap to the Programmable World: Software Challenges in the IoT Era," in IEEE Software, vol. 34, no. 1, pp. 72-80, Jan.-Feb. 2017.

A. Tarvo Mining Software History to Improve Software Maintenance Quality: A Case Study. IEEE Software 26(1), 34-40, 2009..

D. Tofan, M. Galster and P. Avgeriou, "Capturing tacit architectural knowledge using the repertory grid technique," 33rd International Conference on Software Engineering (ICSE), Honolulu, Hawai, pp. 916-919, 2011.

C. C. Venters, C. Jay, L. Lau, M. K. Griffiths, V. Holmes, R. Ward, J. Austin, C. E. Dibsdale, and J. Xu, Software sustainability: The modern tower of babel, in Proceedings of the Third International Workshop on Requirements Engineering for Sustainable Systems (RE4SuSy), 2014.

C. C. Venters, L. Lau, M. K. Griffiths, V. Holmes, R. R. Ward, C. Jay, C. E. Dibsdale, and J. Xu, “The blind men and the elephant: Towards an empirical evaluation framework for software sustainability,” Journal of Open Research Software, vol. 2(1), 2014..

J. Voas, and R. Kuhn, What happened to software metrics? Computer, 50(5), 88-98, 2017.

O. Zimmermann, T. Gschwind, J. M. Küster, F. Leymann, N. Schuster, Reusable Architectural Decision Models for Enterprise Application Development. Third International Conference on Quality of Software Architectures (QoSA), Springer LNCS 4880, 15-32, 2007.

O. Zimmermann, L. Wegmann, H. Koziolek, T. Goldschmidt, Architectural Decision Guidance Across Projects - Problem Space Modeling, Decision Backlog Management and Cloud Computing Knowledge. WICSA, pp. 85-94, 2015.

O. Zimmermann, Metrics for Architectural Synthesis and Evaluation - Requirements and Compilation by Viewpoint. An Industrial Experience Report. 2nd IEEE/ACM International Workshop on Software Architecture and Metrics, SAM 2015, IEEE DL, 8-14, 2015.

U. Zdun, R. Capilla, H. Tran, and O. Zimmermann. Sustainable architectural design decisions. IEEE Software, 30(6), 46–53, 2013.

G. Wilson. Software Carpentry: lessons learned, 2016, Available at: https://f1000research.com/articles/3-62/v1

E. Woods. Software Architecture in a Changing World, IEEE Software, 33(6): 94-97, 2016.