Confluent Factors, Complexity and Resultant Architectures in Modern Software Engineering

A Case of Service Cloud Applications

Leszek A. Maciaszek[1,2] and Tomasz Skalniak[1]
[1]*Wrocław University of Economics, Komandorska 118/120, 53-345 Wrocław, Poland*
[2]*Macquaire University, Sydney, Australia*
leszek.maciaszek@ue.wroc.pl, tomasz.skalniak@ue.wroc.pl

Keywords: Software Engineering, Service-oriented Cloud-based Applications, Software Complexity, Adaptive Systems, Architectural Design, Dependency Relationships, Meta-Architecture, Resultant Architecture.

Abstract: There is a wealth of evidence that contemporary landscape of software development has been resisting the disciplined, rigorous, formally managed, architecture-driven, forward-engineering practices. The whole field of traditional software engineering needs a re-definition alongside the practices widely used in production of modern software systems, in particular service-oriented cloud-based applications. This paper argues that contemporary software engineering must re-focus and re-define its theoretical foundations and base it on acknowledgment that quality software and systems can (and by and large should) be constructed using principles of resultant architectures and roundtrip engineering.

1 INTRODUCTION

Software engineering has never matured enough to match theory and practice of traditional engineering disciplines, such as civil engineering. Based on computer science as its foundation, software engineering has struggled to ensure software production with predictable outcomes. The main culprit is the "soft" nature of software and associated demands of users for change and evolution. When combined with the ever growing complexity of application domains that software systems solve, a need for new software engineering has been finding many vocal supporters (e.g. Jacobson and Seidewitz, 2014).

Such a need is additionally propped up by the demands placed by the fact that we live in service economy (Chesbrough and Spohrer, 2006). Almost every modern agricultural or manufacturing product is combined with services, and it is the joint product-service experience that is judged by service requestors, thus truly generating real value for individuals and profit growth for businesses. Interaction and collaboration between actors of a service (suppliers, consumers, and intermediaries) create value-in-context, employment and economic growth. A supplier offers a value proposition that can be realized in a separate process involving requestors and intermediaries. The benefits to all actors define the context of value co-creation.

Service economy exerts new business and pricing models for using information systems without owning them. Such systems are delivered to users over Internet (the cloud) as Software-as-a-Service (SaaS). Services (e-services) in SaaS systems are running software instances, which can be dynamically composed and coordinated to provide executable applications.

The delivery of Service Cloud Applications (SCA) to actors is performed (typically) on Everything-as-a-Service models (Banerjee, 2011), in which software, platform and infrastructure are made available as services paid for according to the usage. This creates ubiquitous marketplace where commercial, social, government, health, education and other services are facilitated, negotiated, coordinated and paid for through marketplace platforms.

We recognize that e-marketplaces for services (such as Airbnb, OpenTable, or BlablaCar) are governed by different business and technology principles than e-marketplaces for products (such as Alibaba, eBay, MarcadoLivre, or Amazon). However, we also recognize that e-service systems

Maciaszek L. and Skalniak T.
Confluent Factors, Complexity and Resultant Architectures in Modern Software Engineering - A Case of Service Cloud Applications.
DOI: 10.5220/0005885300370045
In *Proceedings of the Fifth International Symposium on Business Modeling and Software Design (BMSD 2015)*, pages 37-45
ISBN: 978-989-758-111-3

narrow the differences between services and products. The dichotomy between these two concepts has been replaced by a service-product continuum (Targowski, 2009). On one hand, software products are servitized; on the other hand, software services are productized (Cusumano, 2008). On one hand, vendors of traditional "boxed" software products use the cloud as a means of servitizing the product (and using it without owning it); on the other hand, productized services (i.e. automation of services, such as movies over Internet) enable "people to participate in a growing number of service-related activities without having to be physically present" (Targowski, 2009, p.57).

The service-product continuum has posed new challenges on the very idea of complexity and change management in a modern-age service enterprise. The responsibilities for complexity and change management have been placed squarely in the hands and minds of the producers and suppliers/vendors of service systems and applications (but much of the risk is still endured by the enterprises and consumers receiving/buying the services).

The established disciplines of software engineering (e.g. Maciaszek and Liong, 2005) and systems analysis and design (e.g. Maciaszek, 2007) have advocated architecture-driven forward-engineering processes. Modern practices challenge the merits and economics of the architecture-first approach to software engineering (Booch 2007). They also challenge the rigid top-down development epitomized in three consecutive phases of systems analysis, design and implementation. They do not, however, challenge the traditional architectural design role of managing system complexity expressed in terms of dependencies between system elements.

The paper is organized as follows. The next Section considers service cloud applications as a significant disruptive technology and it explains the main confluent factors that shape the future of modern software engineering.

Section 3 reiterates the fact that complexity of modern software systems lies in the interactions and dependencies between software elements, which – by the object-oriented paradigm – are internally relatively simple (or at least they should be simple). This section classifies dependencies and explains how SoaML can be used to model SCA software structures.

Section 4 discusses the idea of resultant architecture as a replacement for the architecture-first paradigm. The section proposes a meta-architecture for service cloud applications and it positions the roundtrip engineering as a modus operandi of modern software engineering.

The final Section contains concluding remarks. It emphasizes the necessity of designing for change as opposed to just programming for change. It makes also a clear distinction between emergent and resultant architectures.

2 CONFLUENT FACTORS

The contemporary practice of development of service-oriented cloud-based web and mobile applications changes the pressure points and creates new expectations with regard to modern software engineering. Figure 1 is a Venn diagram that names the main confluent factors that bring about a need for redefinition of software engineering as a discipline. The overlapping between factors is significant. It emphasizes that the factors frequently come together in various combinations.

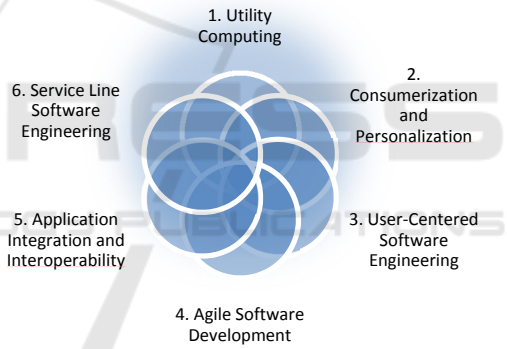

Figure 1: Confluent factors of modern software engineering.

The SCA are delivered over Internet as a kind of utility/commodity similar to energy, water, gas, telephony, and alike. They are a fact of live and are omnipresent - they are available on mobile devices at any time and any place and they can adjust to the context of use (including the current user needs, the geographical position, the temporal information, the weather conditions, the signals from the Internet of Things (IoT) sensors and actuators, etc.). Utility computing is the first confluent factor of modern software engineering.

Service innovation is consumer-focused. Consumer market, as the primary driver of CSA innovation, challenges the way companies innovate and evolve with IT. The innovation ideas need to tap into the phenomenon of consumerization and

personalization as the tendency for new IT solutions to emphasize consumer-focused service provision and to emerge first in the personal consumer market and then spread into business and government organizations. Consumerization and personalization open up an opportunity for new business models and ways of value creation, and it is the second confluent factor of modern software engineering.

By centering on consumerization, personalization, and collaborative context-dependent value creation, the modern software engineering shifts decisively towards user-centered engineering (Richter and Fluckinger, 2014; Ko, et al., 2011) and what Brenner et al. (2014) call user, use & utility research (or 3U research, to use another parlance). Although software engineering has always been recognizing significance of user's acceptance of a solution, the development of SCA systems not just recognizes, but focusses on users by emphasizing the software quality of usability (ISO, 2011) and delivery of a great User eXperience (UX) instead of delivery of software product. User-centered software engineering is the third confluent factor of modern software engineering.

The user-centered software engineering synchronizes nicely with the agile software development that dominates contemporary software engineering practice (Moran, 2015). The agile development methods, such as Scrum, are responsible for a real shift from the architecture-first approach - if not in theory, then certainly in practice. The agile software development is the fourth confluent factor of modern software engineering.

Although the agile methods are called development methods, in reality a great majority of software projects are undertakings in value-added software integration and interoperability (Maciaszek, 2008a). Most new software applications must integrate and interoperate with existing applications and databases, thus making them value-added applications. The whole technology of web service orchestration is about integration and interoperability of SCA-s. Application integration and interoperability is the fifth confluent factor of modern software engineering.

The owner/supplier of a SCA platform sets up instances for the SCA users. Customization and variability of instances is based on the technology of multi-tenancy and uses the emerging principles of Service Line Engineering (SLE) (Mohabbati et al., 2013; Walraven et al., 2014)). Service Line Software Engineering is the sixth confluent factor of modern software engineering.

3 COMPLEXITY IN-THE-WIRES

The times when software complexity could be measured in lines of code or function points are long gone. Since the object-oriented paradigm has replaced the structured programming reminiscent of Cobol systems, the monolithic size of a program ceased to be an indication of software complexity.

Complexity of modern modularized systems comes from the relations between the modules (be them objects, classes, components, packages, services). Complexity is in-the-wires between the modules, not in the modules themselves.

In our past research, we have extensively discussed the complexity in-the-wires principles for large on-premise enterprise information systems (e.g. Maciaszek and Liong, 2005; Maciaszek, 2007). We have demonstrated that complexity minimization is synonymous with the minimization of the inter-module dependencies, where dependency is "a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s)."(OMG, 2009)

It is important that complexity management revolves around software metrics that monitor and measure dependencies in the engineered code. The Design/Dependency Structure Matrix (DSM) (e.g. Eppinger and Browning, 2012) is an excellent method for visualizing, measuring and analyzing dependency relationships in software. Today many tools exist that support the DSM method, e.g. Structure101 (Structure, 2015).

In Maciaszek (2008b) and elsewhere we have discussed the ways of using DSM for the analysis and comparison of software complexity in large systems. We have applied the DSM analysis to the PCBMER meta-architecture consisting of six hierarchical-ordered software layers: Presentation, Controller, Bean, Mediator, Entity, Resource (e.g. Maciaszek, 2007).

When using DSM or other software metrics to calculate dependencies, it is important to consider various categories of dependencies and their relative importance (weight) in measuring complexity and adaptability. At a relatively high level of abstraction pertaining to complexity analysis of traditional enterprise applications, four categories need to be considered: message dependencies, event dependencies, inheritance dependencies and

interface dependencies (Maciaszek and Liong, 2005).

Complexity of modern service-oriented cloud-based applications can also be discussed based on these four categories of dependencies, but better classifications seem to be those that put services at the forefront of the discourse. One possibility is to consider just three categories of dependencies: services, references, and properties (only services are discussed in any depth in this paper).

Since "a service is value delivered to another through a well-defined interface" (SoaML, 2015, p.7), we need to concentrate on interface dependencies when engineering SCA-s. To this aim, we can adopt the SoaML (Service oriented architecture Modeling Language) standard (SoaML, 2015). The standard distinguishes three ways of service interaction: a simple interface, a service interface, and a service contract.

A simple interface is a UML-style interface as supported by popular object-oriented languages, such as Java, and web services called via RPC (Remote Procedure Call). Simple interfaces are uni-directional – the consumer calls a provider's service and the provider does not callback the consumer and may not even know it.

A service interface involves bi-directional communication between provider and consumer. "The service interface may also specify the choreography of the service - what data, assets and obligations are sent between the provider and consumer and in what order. ... The consumer must adhere to the provider's service interface, but there may not be any prior agreement between the provider and consumer of a service." (SoaML, 2015, pp.8-9).

A service contract defines how participants (providers, consumers, and other roles) work together to exchange value. To this aim, service specifications are defined in a service contract. The contract determines the participants, the interfaces, choreography, and any other terms and conditions for the enactment of the service. Service contracts are therefore encapsulation (implementation handling) mechanisms.

Service interactions via interfaces and contracts are principle communication means between software architectural layers. The layers can be represented as UML collaborations. They can contain the SoaML service capabilities, which "identify or specify a cohesive set of functions or resources that a service provided by one or more participants might offer." (SoaML, 2015, p.29).

Capabilities may be related to show intra-layer dependencies. They can aslo be used to visualize and define intra-layer dependencies, in particular they can specify the behavior and structure of interfaces (realized by the capability, which in turn is realized by a service participant). Capabilities can be nested/combined to form larger capabilities.

Capabilities can be related by 'usage' relationships. An 'expose' relationship can be used to indicate what capabilities (required or provided by a participant) should be exposed through a service interface. However, the operations and properties of a service interface may differ from operations and properties of a capability it exposes. "It is possible that services supported by capabilities could be refactored to address commonality and variability across a number of exposed capabilities" (SoaML, 2015, p.47).

The UML interface realization can be used to denote service interfaces that a capability 'realizes' (implements). As with the 'expose' relationship, the operations and properties of a service interface and a capability may differ.

SoaML also defines the notion of a service channel as a communication path between consumer requests and provider services. The service channels between and within the architectural layers determine the complexity and adaptability of a service system.

4 RESULTANT ARCHITECTURE

In our past research we have argued that a valid answer to the software complexity and adaptability is the architecture-first design, i.e. that the architecture should be designed into the system. However, we have always recognized that the proactive forward-engineering architecture-first approach requires a parallel contribution from a reactive reverse-engineering approach (Maciaszek, 2005). In other words, we have recognized that the architecture should result from roundtrip engineering. The concept of a resultant architecture, used in the titles of this paper and this section, is a consequence of of the above interpretation of roundtrip engineering.

The primary purpose of an architecture is to minimize complexity of expected outcome and to lead to a solution that is adaptive, i.e. understandable, maintainable, and extendable. Software engineering and management has struggled to properly address systems complexity and adaptability. The reason is twofold: deficient

architectural design and/or nonconformance of software implementation to the architectural design.

The paradigm shift to SCA-s has introduced new threats and opportunities with regard to complexity management and delivery of adaptive solutions. On one hand, the SCA-s assume dynamic composition of services and tenant variability and, therefore, they emphasize implementation over architecture (and over project management at large). On the other hand, the SCA-s are built on the technologies that, by their very nature, support adaptability. The concepts such as loose coupling, abstraction, orchestration, implementation neutrality, configurability, discoverability, statelessness, immediate access, etc. are exactly the ideas of adaptable architectural design.

Figure 2 represents our meta-architecture (i.e. an architectural reference model) for SCA-s, called Meta-SCA. The model retains the conceptual thrust of the PCBMER meta-architecture and it provides roundtrip engineering perspective on our recently defined STCBMER (Smart Client, Template, Controller, Bean, Mediator, Entity, Resource) meta-architecture (e.g. Maciaszek et al., 2015).

The Meta-SCA model recognizes and even emphasizes the fact that SCA engineering activities are facilitated by software toolkits and frameworks. Toolkits enable code reuse. They support programmers in writing the real code – the main body of the program. Frameworks enable design reuse. They provide to programmers a skeleton of the program and inform programmers which code to write, so that the framework can call it.

Toolkits and frameworks deliver reusability at the level of software implementation, but they need to be chosen to facilitate the forward engineering objective of minimizing complexity and maximizing adaptability. Frameworks need to facilitate implementation of the meta-architecture; toolkits need to facilitate implementation of patterns and principles. In the reality of SCA-s, toolkits and frameworks can be encapsulated within the technology of Platform-as-a-Service (PaaS).

The Meta-SCA model defines four hierarchical layers for the application code placed on top of a Data Storage layer. The four layers are called Client Front-end, Client Back-end, Business Service and Data Access. They are modeled as SoaML collaborations and stereotyped as <<ServicesArchitecture>>.

Each layer contains single SoaML <<Participant>> realizing specific capabilities. Participant at a higher layer requests services implemented in participants in lower layers. This is represented on the model by the UML notation of required and provided interfaces (so called lollypop notation). It also indicates a top-down single directional interface dependency between layers.

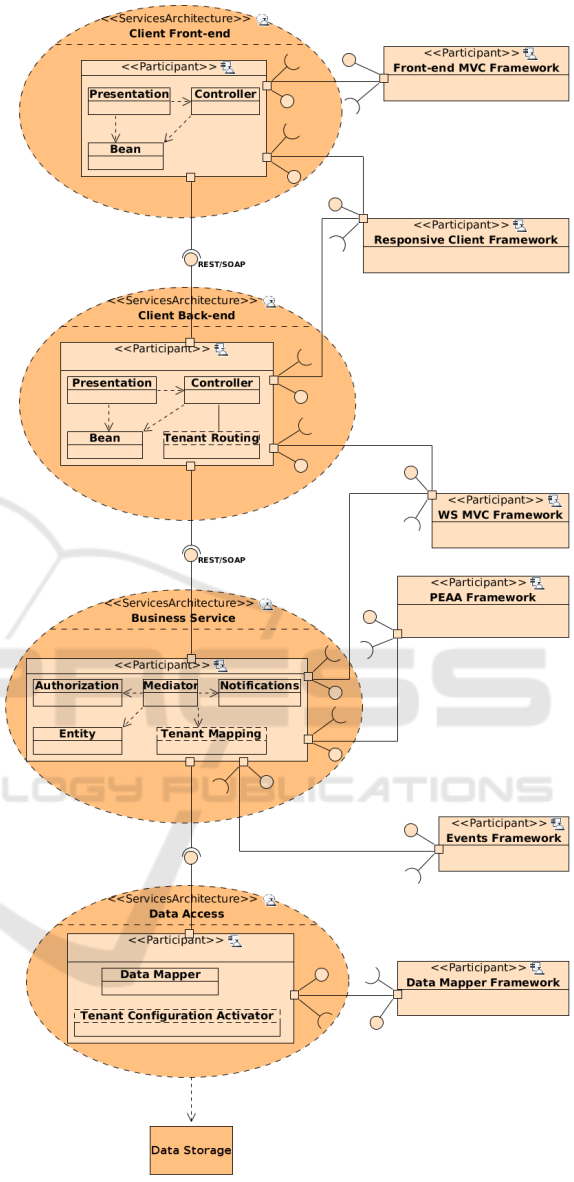

Figure 2: Meta-architecture for service cloud applications (Meta-SCA).

The service discovery can be realized through WSDL (Web Services Description Language). The service binding can be realized through SOAP (Simple Object Access Protocol), but even better degree of software adaptiveness can be achieved if the statelessness of the system is not an issue. If it is not, then the REST (Representational State Transfer) architecture might be more desirable than the SOAP.

In parallel to the layers, the Meta-SCA model defines other participants, which are third-party frameworks and toolkits supporting the system's implementation. Each framework is understood as a service offering a special “framework interface” that is needed by some of the components of the proposed meta-architecture.

Relations between frameworks and layer participants are defined as bi-directional service interfaces. This is because to use a framework you to have to deliver to it a code, which satisfies strict conditions defined by the framework. Also, the frameworks have to offer a given set of functionalities to the layer participants.

Some frameworks in Meta-SCA – Responsive Client Framework and SW MVC Framework – are needed and used by two different layers. In such a case, each framework has its own framework instance (this way the Meta-SCA does not introduce to the model disallowed dependencies).

Each layer participant has various components inside. The dependencies between those components can be organized in various ways – according to different need of a specific environment, technologies and project as well as chosen frameworks (which support the implementation). The organization of those dependencies follows the guidelines and propositions consistent with and based on the PCBMER and STCBMER studies.

The Client Front-end and Client Back-end layers form an abstract application layer, while the Business Service and Data Access layers together form an abstract business layer (e.g. Maciaszek et al., 2015).

The Front-end MVC framework is a JavaScript framework needed to implement complex, off-line, widget-based asynchronous web applications. The Responsive Client Framework is a presentation framework supporting implementation of Responsive Web applications.

The Web Services MVC Framework is a server-side framework, which offers building the web services communication as well as layering the code into separate parts based on different variants of the MVC pattern. These types of frameworks are often called just web frameworks.

The PEAA Framework means a framework containing different, needed implementations of the patterns from the Catalog of Patterns of Enterprise Application Architecture by Martin Fowler (2013).

The Events Framework refers to frameworks and toolkits offering project-specific event-based implementations. This includes authorization tools and notification schemes (e.g. SMS notifications), but also any other event-driven mechanisms allowing interoperability with cyber-physical monitoring systems (IoT sensors and actuators) and integration with enterprise, government and social networking systems.

The Data Mapper Framework is a type of framework, which offers connecting to a database or other data stores, mapping of database entities to programming objects, transaction management, etc.

Table 1 presents examples of Meta-SCA frameworks and toolkits.

Table 1: Examples of Meta-SCA frameworks.

Framework	Examples
Front-end MVC Framework	Angular.js, Backbone.js, Knockout.js
Responsive Client Framework	Bootstrap, Foundation
WS MVC Framework	Django, Pyramid, Rails, Spring MVC, ASP .NET, Symfony
PEAA Framework	Spring, .NET, Django REST Framework
Events Framework	JMS (Java Message Service), Activiti, Kinoma
Data Mapper Framework	Hibernate, SQL-Alchemy

Some of the meta-architecture components (Tenant Routing, Tenant Mapping, Tenant Configuration Activator] are placed in the diagram to emphasize that modern tenant-oriented systems often need to have tenant-specific code in different layers. In addition, this special code has to cooperate with the frameworks used in a specific project. As a result there is a need of having the code organized in independent modules that can be easily used when needed.

There are several ways of building the tenant-based systems. Separation of data and functionalities can be implemented on different layers of the system (also the databases often are playing a role in it). That is why the proposed meta-architecture has different tenant-specific components placed in different layers. In reality not all of them might be needed – based on the chosen design patterns, different layers might or might not need the tenant-specific code.

Concrete instances of architectures derived from the Meta-SCA in the roundtrip engineering process need to conform to the additional rules and principles over and above the layering shown in Figure 2. Because of the space limitations of this paper we cannot describe them in any details. Nevertheless most of the twelve principles defined for the STCBMER meta-architecture (Maciaszek et al., 2015) remain relevant for the Meta-SCA.

Three most fundamental principles from the PCBMER and STCBMER - DDP (downward dependency principle), UNP (upward notification principle) and CEP (cycle elimination principle) - fully apply to the Meta-SCA. This means, that message dependencies are only allowed in downward communication between layers and the upward communication requires event notifications from lower layers, possibly combined with the use of interface inheritance. Implementation inheritance is disallowed between layers; it is restricted to intra-layer implementations. Cycles of message communications (method invocations) are disallowed between layers, inside layers, and for any granularities of objects (classes, components, packages, web services). Cycles need to be eliminated using well-known software engineering rules, exhaustively explained for example in Maciaszek and Liong (2005).

In reality – when programmers start to code – it is very hard to stick to the abstract meta-architecture without breaking some of its principles and assumptions. This is because of different patterns chosen to base the frameworks on and because of various technologies used by the frameworks.

Since the frameworks in modern software engineering are more and more becoming the backbones of the software solution, it is very important to choose the right ones for implementing a concrete instance of the architecture so that it fully conforms to the abstract meta-architecture. There are two ways of how to do this. The first is by choosing the frameworks which are compatible with the meta-architecture. The second is by choosing the frameworks which are modular and flexible enough to set them up in a way that is satisfying the meta-architecture. Nowadays many frameworks have sufficient modularity and flexibility so that the programmers can easily replace predefined framework's components with other implementations – written by themselves or by third-party organizations.

The conformance of the project's resultant architecture to the meta-architecture and its principles should be evaluated by an in-depth analysis of dependencies. This must involve some reverse-engineering of code to establish factual dependencies to compare them against the allowed dependencies of the meta-architecture. The DSM method briefly discussed in Section 3 provides an excellent vehicle for dependency analysis and calculation of dependency metrics.

Unfortunately measuring the dependencies in a project built with the help of a number of frameworks and toolkits is a difficult task. It constitutes a new and important research challenge. We intend to focus on this problem in our future work and to measure the impact of contemporary frameworks and toolkits on architectural design of SCA-s from the viewpoint of software complexity and adaptiveness.

5 CONCLUDING DISCUSSION

Traditional software development lifecycles assume architecture-first design (Booch, 2007; Maciaszek, 2007). However, the confluent factors of modern software engineering have led to practices where software architecture evolves in parallel with software construction. As noted by Jacobson and Seidewitz (2014), "...agile development have made it possible to create high-quality software systems of significant size using a craft approach - negating a major impetus for all the up-front activities of software engineering" (p.50).

Moreover, modern multi-tenant SaaS applications (Walraven et al., 2014) demand the built-in capability of dynamic software adaptation (Kakoutsis et al., 2010). This in turn requires inventing new architectural styles that respond to and embrace the dynamic runtime software adaptability (not addressed in this paper, but refer e.g. Kakoutsis et al., 2010).

Roundtrip engineering activities that aim at resultant architectures for SCA-s are based on and driven by various reuse strategies (e.g. Maciaszek, 2007). The forward engineering activities are driven by an assumed meta-architecture. Associated with the meta-architecture are matching architectural patterns and architectural principles. A meta-architecture delivers reusability at the level of a solution idea. Patterns and principles deliver reusability at the level of software design.

The reverse engineering activities aim at software architecture recovery (e.g. Solms, 2015) and at measurably validating the conformance of a system's resultant architecture to the meta-architecture (Maciaszek, 2008b). An ultimate

objective is a SCA that minimizes complexity and maximizes adaptability.

Information systems in general, and service cloud applications in particular, need to be designed for change. They need to be adaptive. Ideally, they need to be self-adaptive, but such an aim is unreachable as yet in practical software engineering (Maciaszek, 2012).

The confluent factors of modern software engineering reflect the necessity of designing for change. Unfortunately, this is not sufficiently reflected in contemporary practice of software engineering. Current practice is full of ideas, methods and tools to facilitate development/programming for change, but lacks systematic and rigorous approach to designing for change.

The development/programming for change is exemplified, for example, by the growing popularity of DevOps, which is an approach to merging development and operations (Huttermann, 2012). Another example, on the level of user interface and web programming, are approaches known as responsive development, progressive enhancement, graceful degradation (Overfield et al., 2013).

The design for change must revolve around the software architecture, which sets a fundamental structural organization for a software system. Such an organization must determine hierarchical layers of software elements (components, objects, services) ensuring separation of concerns and resulting in a tractable/adaptive complexity of the solution.

There seem to be three approaches to considering software architecture in software engineering projects. The architecture:

1. can be designed into the system,
2. can emerge from the implementation,
3. can result from roundtrip engineering

The first approach is synonymous with the architecture-first approach. It is a commendable approach, but increasingly impractical in the fast-paced world demanding immediate software solutions.

The second and third approach can be best understood by reference to complexity theory (e.g. Agazzi, 2002). Since complexity entails existence of relations/dependencies between elements, then - by opposition - simplicity (something that is analytically simple) entails no internal relations.

Further, the complexity theory distinguishes between emergence and resultance. We speak of emergence when a complex structure emerges from the properties of the "analytic simples" in a way that is not completely understandable and explainable. In this sense, software architecture can emerge in a bottom-up fashion from the implementation of software elements.

By contrast, we speak of resultance when a complex structure results from the properties of the analytic simples by the guidance of the relations between software elements. This means that a meta-architecture exists prior to the implementation and it guides software engineers in designing concrete system architecture (an instance of meta-architecture) in parallel with software implementation. This is a roundtrip engineering effort leading to, what we call, a resultant architecture.

REFERENCES

Agazzi, E. (2002). What is Complexity? In Agazzi, E., Montecucco, L. (Eds) *Complexity and Emergence. Proceedings of the Annual Meeting of the International Academy of the Philosophy of Science,* pp. 3-11, World Scientific.

Banerjee, P. et al. (2011). *Everything as a Service: Powering the New Information Economy*, Computer (IEEE), March, pp.36-43

Booch, G. (2007). The Economics of Architecture-First, *IEEE Software*, Sept./Oct., pp.18-20

Brenner et al. (2014). *User, Use & Utility Research. The Digital User as New Design Perspective in Business and Information Systems Engineering*, Business & Information Systems Engineering, 1, pp.56-61

Chesbrough, H. and Spohrer J. (2006). *Research Manifesto for Services Science*, Comm. ACM, Vol. 49, No. 7, pp.35-40

Cusumano, M.A. (2008). The Changing Software Business: Moving from Products to Services, *IEEE Computer*, January, pp.20-27.

Eppinger, S.D., Browning T.R. (2012). *Design Structure Matrix Methods and Applications*, The MIT Press.

Fowler, M. (2003). *Patterns of Enterprise Application Architecture*, Addison-Wesley.

Huttermann, M. (2012). *DevOps for Developers*, Apress.

ISO (2011). *International Standard ISO/IEC 2510: Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models*, ISO/IEC.

Jacobson, I. and Seidewitz, E. (2014). *New Software Engineering*, Comm. ACM, Vol. 57, No. 12, pp.49-54

Kakoutsis, K., Paspallis, N., Papadopoulos, G.A. (2010). A Survey of Software Adaptation in Mobile and Ubiquitous Computing, *Enterprise Information Systems*, Vol. 4, No. 4, pp.355-389

Ko, A.J. et al. (2011): *The State of the Art in End-User Software Engineering*, ACM Computing Surveys, Vol. 43, No. 3, pp.21:1-21:44

Maciaszek, L.A. (2005). Roundtrip Architectural Modeling, In Hartmann, S. and Stumper, M. (Eds), *Second Asia-Pacific Conference on Conceptual Modelling*, Newcastle, Australia, Australian Computer Science Communications, Vol. 27, No. 6, pp.17-23.

Maciaszek, L.A. (2007). *Requirements Analysis and System Design*, 3rd ed., Addison-Wesley

Maciaszek, L.A. (2008a). Adaptive Integration of Enterprise and B2B Applications. In Filipe, J., Shishkov, B., M. Helfert, M. (Eds.) *ICSOFT 2006, CCIS 10*, pp. 3–15, Springer-Verlag Berlin Heidelberg.

Maciaszek, L.A. (2008b). Analiza struktur zależności w zarządzaniu intencją architektoniczną systemu, In Huzar, Z., Mazur, Z. (Eds), *Inżynieria Oprogramowania – Od Teorii do Praktyki,* pp.13-26, Wydawnictwa Komunikacji i Łączności, Warszawa.

Maciaszek, L.A. (2012). An Architectural Style for Trustworthy Adaptive Service Based Applications, In Ardagna, C.A. Damiani, E. Maciaszek, L.A. Missikoff, M.M. and Parkin, M. (Eds), *Business System Management and Engineering. From Open Issues to Applications*, Lecture Notes in Computer Science, Vol. 7350, pp.109-121, Springer.

Maciaszek, L.A., Liong, B.L. (2005). *Practical Software Engineering. A Case-Study Approach*. Addison-Wesley.

Maciaszek, L.A. Skalniak, T. and Biziel, G. (2015). Architectural Principles for Service Cloud Applications, In Shishkov, B. (Ed.), *Business Modeling and System Design*, Lecture Notes in Business Information Processing LNBIP 220, 21p., Springer, (to appear)

Mohabbati, B. Asadi, M. Gasevic, D. Hatala M. and Mueller, H.A. (2013). *Combining Service-Orientation and Software Product Line Engineering: A Systematic Mapping Study*, Information and Software Technology, http://dx.doi.org/10.1016/j.infsof.2013.05.006, 15p.

Moran, A. (2015). *Managing Agile. Strategy, Implementation, Organisation and People*, Springer.

Overfield, E., Zhang, R., Medina, O. and Khipple, K. (2013). Responsive Web Design and Development. In Overfield, E., Zhang, R., Medina, O. and Khipple, K. (Eds), *Pro SharePoint 2013 Branding and Responsive Web Development*, pp.17-46, Apress

Richter M. and Fluckinger, M. (2014). *User-Centred Engineering. Creating Products for Humans*, Springer.

OMG (2009). *Unified Modeling Language™* (OMG UML), Superstructure, Version 2.2.

Shaw, M. (2009). *Continuing Prospects for an Engineering Discipline of Software*, IEEE Software, November/December, pp.64-67

SoaML (2015). *Service Oriented Architecture Modeling Language (SoaML)*, Retrieved from: http://www.omg.org/spec/SoaML/

Solms, F. (2015). A Systematic Method for Architecture Recovery, In Filipe, J. and Maciaszek, L. (Eds), *ENASE 2015 10th International Conference on Evaluation of Novel Approaches to Software Engineering Proceedings*, pp.215-222, SciTePress

Structure (2015). *Structure101*, Retrieved from: http://structure101.com/.

Targowski, A. (2009). *The Architecture of Service Systems as the Framework for the Definition of Service Science Scope*, International Journal of Information Systems in the Service Sector, 1(1), pp.54-77

Walraven, S., Landuyt, Van D., Truyen, E., Handekyn, K., Joosen, W. (2014). *Efficient Customization of Multi-Tenant Software-as-a-Service applications with Service Lines*, The Journal of Systems and Software, 91, pp.48-62