A Meta-architecture for Service-oriented Systems and Applications

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

Keywords: Meta-architecture, Architectural Design, Service-oriented Systems and Applications, System and Software Complexity, Dependency Relationships, Software Quality, Software Adaptability, Holon Abstraction.

Abstract: The paper proposes a new meta-architecture as a reference model for developing service-oriented systems and applications. The seven-layer meta-architecture is called STCBMER (Smart Client - Template - Bean - Controller - Mediator - Entity - Resource). The purpose of it is to reduce software complexity and ensure the quality of adaptability defined as the degree to which an information system or application is difficult to understand, maintain and evolve. The main difficulty stems from complex interactions (dependencies) between system elements. The dependencies can be minimized if the system under development adheres to the architectural design and can be verified by analysing the implementation code. The paper reinforces the proposition that an architectural intent for adaptive complex systems requires some sort of hierarchical layered structure (according to the holon abstraction as an approach to restraining software complexity).

1 INTRODUCTION

The main concern and objective of software architectural design is to manage complexity in resulting systems and applications. Software complexity must not be higher than the complexity of the problem domain addressed by the software. If it is higher, we say that the software solution is over-complex (unnecessarily complicated). The main condition for lowering software complexity is to base its architectural design on a complexity-minimizing architectural framework or reference model (i.e. a meta-architecture).

Complexity is an axiomatic, but relative concept, which can only be properly interpreted by its relation to its contrary notion of simplicity (Agazzi, 2002). Something is complex because it is not simple, and vice versa.

Complexity is also a multi-faceted concept – what is complex from one point of view may be simple from another point of view. In other words, complexity is the combination of several attributes, which need to be examined separately "so that we can understand exactly what it is that is responsible for the overall "complexity". Nevertheless, practitioners and researchers alike find great appeal in generating a single, comprehensive measure to express "complexity"" (Fenton and Pfleeger, 1997).

In our opinion, a complexity measure, if one can be generated, should be seen as an overriding measure of systems and software quality. Therefore, complexity is a derivative of characteristics constituting system/application quality. As noted by Robert Glass (2005) "the task of building quality into software is almost the same as the task of making it maintainable" (or adaptable in our parlance).

The SQuaRE standard (ISO, 2011) identifies eight quality characteristics, of which the quality of maintainability represents the instrumentation side of complexity. The standard identifies further five sub-characteristics of maintainability: modularity, reusability, analysability, modifiability, and testability. We believe that a better term for these sub-characteristics is adaptability (or adaptiveness) rather than maintainability. Adaptability is a broader concept combining understand-ability as a precondition of maintainability and maintainability as a precondition of evolve-ability.

System/software adaptability is underpinned by its complexity, measured as the count of (permitted) dependency relationships in the system/software, where: "A 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

Maciaszek L., Skalniak T. and Biziel G.
A Meta-architecture for Service-oriented Systems and Applications.
DOI: 10.5220/0005423900200028
In *Proceedings of the Fourth International Symposium on Business Modeling and Software Design (BMSD 2014)*, pages 20-28
ISBN: 978-989-758-032-1

semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).”(OMG, 2009).

In our research, we address the last of the five deep questions in computing identified by Jeannette Wing (2008): “(How) can we build complex systems simply?”. We have argued that a valid answer to this question is to construct system/software according to dependency-minimizing meta-architecture (e.g. Maciaszek and Liong, 2005).

The rest of the paper is organized as follows. Section 2 summarizes the PCBMER meta-architecture and makes a case for adjusting and extending it to suit modern service-based systems and applications. Section 3 defines the "service enterprise" viewpoint on complexity and change management in systems and applications. This section introduces a new meta-architecture Smart Client - Template - Bean - Controller - Mediator - Entity - Resource (STCBMER). The meta-architecture refers to the technology-specific frameworks (used and validated on a large e-marketplace project in the domain of Ambient Assisted Living (AAL), but not used here as a case study for the lack of space). The related work, the conclusion and the future work sections close the paper's discussion, and they are followed by the list of references.

2 A RECAP OF THE PCBMER META-ARCHITECTURE FOR ENTERPRISE INFORMATION SYSTEMS

The architecture informs how system/software elements are interlinked. It abstract away from implementation and it omits information not related to interactions between elements. There can be many levels of architectural abstraction. We distinguish between a meta-architecture as a desired holonic structure and concrete instantiations of it in system/software under development. Those concrete instantiations (or architectures) must conform to the chosen meta-architecture so that the complexity-minimization objective is achieved.

A layered, ideally holonic-like structure is the first sine qua non condition for an architectural solution leading to the production of adaptive systems. The PCBMER is our original meta-architectural proposal for such architectural instantiations. The second sine qua non is the use of managerial dependency analysis tools to ascertain adaptability in concrete instantiations. The DSM is our managerial tool of choice for dependency analysis.

An architectural division into layers, apart from complexity reduction, has many other advantages. Without much trouble we can exchange components within a layer, e.g. within the Presentation layer we can change HTML pages to dynamic JSP pages. Moreover, a layer can only communicate with neighbouring layers and only in a single-directional way (i.e. cyclic references are not permitted). As a result, changes in a layer do not require changes in independent layers (i.e. layers that do not depend on the modified layer).

Figure 1 illustrates the PCBMER meta-architecture modelled in UML and showing layers as UML packages. There are six layers: Presentation, Controller, Bean, Mediator, Entity, Resource (e.g. Maciaszek, 2007). Figure 1 shows also Utility Data Sources (typically databases) accessible exclusively from the Resource layer.

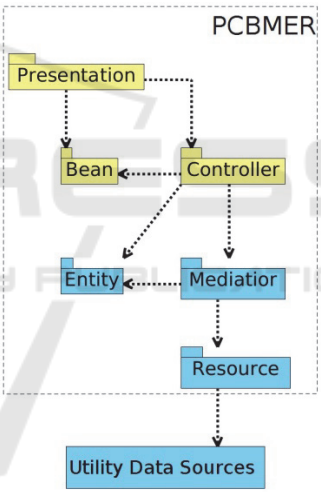

Figure 1: The original PCBMER meta-architecture.

The *Presentation* layer represents the graphical user interface (GUI) objects on which the data (beans) from the Bean layer can be rendered. It is responsible for maintaining consistency in its presentation when the beans change. So, it depends on the Bean layer.

The *Bean* layer represents the data classes and value objects that are destined for rendering on GUI. Unless data is entered by the user, the bean data is built up from the entity objects (the Entity layer).

The *Controller* layer represents the application logic. Controller objects respond to the Presentation requests resulting from user interactions with the system.

The *Entity* layer responds to Controller and Mediator. It contains business objects retrieved from the database or created for successive storage in the database. Many entity classes are container classes (i.e. they contain business objects and methods for adding and removing objects as well as methods to iterate over objects).

The *Mediator* layer mediates between Entity and Resource classes. It manages business transactions, enforces business rules, instantiates business objects in the Entity layer, and in general manages the memory cache of the application. Architecturally, Mediator serves two main purposes. Firstly, to isolate the Entity and Resource layers so that changes in any one of them can be introduced independently. Secondly, to mediate between the Controller and Entity/Resource layers when Controller requests data, but it does not know if the data has previously been loaded from the database into memory.

The *Resource* layer is responsible for all communications with external persistent data sources (databases, web services, etc.). This is where the connections to the database servers are established, queries to persistent data are constructed, and the database transactions are instigated.

The downward arrows between the PCBMER layers signify acyclic dependency relationships. Cyclic dependencies are the main characteristic of over-complex systems and the culprit of the lack of adaptability in such systems. The Downward Dependency Principle (DDP) and the Cycle Elimination Principle (CEP) are two main architectural principles of PCBMER (Maciaszek and Liong, 2005).

The DDP principle ensures that all message dependencies (function calls) have downward direction (message dependencies signify tightly coupled communication, such as in Remote Method Invocation (RMI) - not to be confused with asynchronous messaging, such as in Java Messaging Service (JMS)).

Higher PCBMER layers depend on lower layers, but not vice versa (at least not from the viewpoint of message dependencies). As a result, managing change in lower layers is more troublesome and we need to endeavour to apply extra care to designing lower layers, so that they are more stable (i.e. more resilient to changes).

The DDP principle is further constrained by the Neighbour Communication Principle (NCP). This principle ensures that objects can communicate with distant layers only by utilizing chains of message passing through neighbouring layers. Occasional claims in the literature that such message passing impacts performance are misguided, in particular in the context of enterprise information systems in which performance is invariably related to input/output data transfers to/from databases (performance penalty of in-memory processing is negligible in this context).

The CEP principle demands that cycles of messages are disallowed between objects. The principle applies to objects of any granularity (methods, classes, components, services, packages, subsystems, etc.). This does not mean that call-backs are disallowed. It just means that call-backs must be implemented using other than straight message passing techniques. The two principal techniques are event processing and the use of interfaces, sometimes combined to achieve a desired effect. Additionally, clustering and de-clustering of objects can result in elimination of some cycles. Maciaszek and Liong (2005) contains a detailed description of cycle-elimination techniques.

The Upward Notification Principle (UNP) is a separately-listed principle to counteract the stringent DDP rule and to enforce the CEP principle in communications between layers. This principle requires that lower layers rely on event processing (publish/subscribe protocols) and interfaces to communicate with objects in higher layers.

The PCBMER meta-architectural framework has been created for and validated in development of large scale "stovepipe" enterprise information systems and applications. The software production in such projects is entirely in the hands and minds of the software development team. However, modern software production is not "stovepipe" any more. Software development projects are not standalone undertakings - they are endeavours in systems integration. Complexity management and delivery of adaptable solutions takes on a new dimension.

Firstly, the shift from systems development to systems integration manifests itself on the software level by the shift from synchronous message passing to asynchronous event processing (Maciaszek, 2008a). This has an obvious business explanation. Integration implies dependency on the code that is not our own and not under direct control of the developers (or rather integrators, to be precise). Frequently, this is the code of our business partners who are unlikely to open it up for synchronous message passing from/to our code. But even in case of the integration projects within the same organization, the independent nature of separate business processes (and the software supporting

them) is unlikely to permit or warrant synchronous interoperability. Moreover, whether integrating with external systems or with internal systems, synchronous message passing typically would require some level of intervention in the source code of the system we integrate with. Clearly, this is almost never an option.

Secondly, and related to the systems integration issue, another paradigm shift has been observed in modern software production - the shift from in-house software ownership to trusted provisioning of service-based systems and applications. Grounded in the Service Oriented Architecture (SOA) model of computation, this shift has created a new dimension to our understanding of software complexity and delivery of adaptable Software as a Service (SaaS) solutions. The first and foremost concern are the implications for architectural design of such systems and applications. This is discussed next.

3 THE STCBMER META-ARCHITECTURE FOR SERVICE ENTERPRISE

Founded on cloud computing, the SaaS phenomenon exerts new business and pricing models for using information systems without owning them. Service-oriented systems have emerged as a new scientific abstraction allowing orchestration of service resources and processes according to value propositions (co-creation of value).

Service systems and applications have become a commodity - like telephone, water, energy, gas, etc. Associated with this observation, several dichotomies have emerged. On one hand, software products are servitized; on the other hand, software services are productized (Cusumano, 2008). On one hand, vendors of Component of the Shelf (COTS) enterprise information systems use Internet as a service delivery mode; on the other hand, productized services are delivered over Internet as enablers and productivity enhancers in the service economy.

The above dichotomies have 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 shifted to producers and suppliers/vendors of service systems and applications, but much of the risk is endured by the enterprises receiving/buying the services. It comes as no surprise that enterprises seek to alleviate the risks and try not to lose control over their own destiny.

The main objective and sine qua non in such service enterprises must be to ensure the adaptability of received service systems and applications. This in turn implies a demand for a layered, modular and dependency-minimizing architecture in such systems and applications, so that the service enterprise can understand, maintain and evolve its software solutions. In this context, it does not matter if a service system or application is delivered as a complete SaaS solution or it is delivered as componentized web services from which a system or application is constructed. In all cases a level of trust between providers and recipients of services is necessary, and in all cases we need to ensure the quality of adaptability in service solutions.

Interestingly, but also paradoxically, the service systems and applications 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. In the remainder of the paper, we propose a meta-architecture for adaptable architectural design of SOA systems and applications. The meta-architecture has evolved from the PCBMER meta-architecture and it is called Smart Client - Template - Bean - Controller - Mediator - Entity - Resource (STCBMER).

The seven layers of the STCBMER meta-architecture can be grouped into three main architectural modules as shown in Figure 2. The three modules - Smart Client Logic, Application Logic, and Business Logic - work in different address spaces separated by the technology of web services. The SOA technology is responsible for discovering web services, providing service binding, and orchestrating an exchange of information through web service interactions. The service discovery dependencies can be realized through WSDL (Web Services Description Language). The service binding dependencies can be realized through SOAP (Simple Object Access Protocol) or REST (Representational State Transfer) invocations.

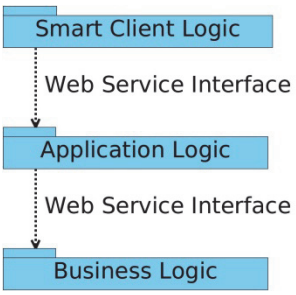

Figure 2: The main modules of the STCBMER meta-architecture.

Figure 3 shows the layered model of the STCBMER meta-architecture. Layers are represented as the UML packages. In the discussion that follows we identify possible technologies for the packages and sub-packages (based on the ones that we have used in a specific instantiation of the meta-architecture in a large project that has served as a validation platform for our architectural vision).

The arrows between the STCBMER packages and sub-packages signify message dependencies. Figure 3 shows also the connectivity from the Smart Client layer to a Web Browser as a typical user interface and the connectivity from Resource to Utility Data Sources.

The most independent and therefore most stable layer is *Resource*. The Resource is a layer responsible for communication with Utility Data Sources (relational databases, NoSQL databases, LDAP directories, etc.). It contains tools to communicate with the database, manage database sessions, construct database queries, etc. Being the most stable layer, it allows easy switching between data sources without making changes in higher layers. The Resource connects to a data source, constructs queries and allows building Entity objects (by Mediator) based on various data sources. The SQL-Alchemy framework is a possible technology for the Resource layer.

The *Entity* layer contains two sub-layers: Entity Object and Entity Object Adapter. The Entity Object package holds business entities, which are mapped (loaded) from data sources. They can be mapped from one or more database tables or views using well known mapping patterns.

ORM (Object-Relational Mapping) frameworks, such as SQL-Alchemy, provide two ways of defining concrete mappers: mapping can be defined as an external class or it can be defined directly in an entity object class. In theory, better and cleaner way is to define the mapper as the external mapping class. In practice, mapping directly in the entity object class may be preferred because in the external mapping all database relationships are added dynamically to the entity object class and are not directly visible in the code as accessible attributes (when for example SQL-Alchemy is used).

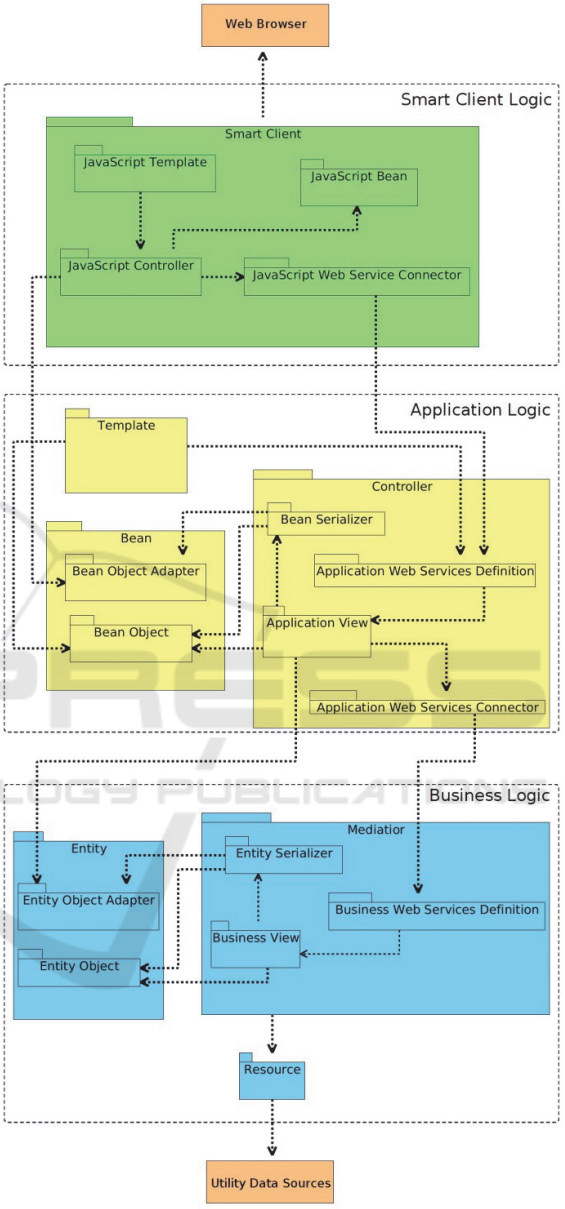

Figure 3: The STCBMER meta-architecture.

The Entity Object Adapter package is a set of classes, which represent entity objects which are serialized and ready to send via a web service. Also every entity object adapter class decides which attributes of the original entity object should be visible to external applications (web services consumers). JSON-based (JavaScript Object Notation) representation might be a good choice,

especially if the web service is built with a REST Web Service. JSON is a native JavaScript type, so it suits well web programming, and it is a reasonable alternative to the XML (eXtensible Markup Language).

The *Mediator* layer is responsible for managing business transactions and business rules as well as loading and unloading business objects (entity objects). This layer manipulates entity objects and defines a kind of Facade pattern, which offers access to them: getting, saving, creating, deleting, editing and caching.

As a technology-specific example, the Mediator could use the SQL-Alchemy or other ORM framework to communicate with the Resource layer (which also could be based on the SQL-Alchemy) to load/unload entity objects. Entity objects manipulation (the Mediator) could be available as a set of simple Python functions as well as a set of web services defined with the Pyramid web framework (as in our platform of choice) and accessible via the REST interface. Those functions should be defined in the Business View sub-package and are called “business views”.

To construct a web service (view) from a Python function, a programmer can use a special decorator (Decorator pattern) provided by the Pyramid framework. Since all web methods (views) are available via the REST (Representational State Transfer) interface, every web service should be accessible with a given URI (Uniform Resource Identifier). Routing from a given URI to a specific web service is done by the Pyramid itself. This functionality in the STCBMER meta-architecture is realized by the Business Web Service Definition package (and analogously by the Application Web Service Definition package in the Application Layer).

In the SOAP-based web service the Business Web Service Definition package should also build a WSDL document describing the web API (Application Programming Interface) of the Business Logic. If the API is built as a REST interface, this package should also define the mappers (routes) from a specific URI to a given view (web service). In the Pyramid web framework all the routes are defined by programmers using regex (regular expression) patterns. To serialize and send business objects via the REST interface, the Mediator uses the Entity Serializer. Every serialized entity object is a JSON object, with structure defined in the Entity Object Adapter package.

The *Controller* layer defines the application logic (different from the Mediator's business logic). In our technology-specific scenario, the application logic is captured in a set of functions (Application View sub-package) accessed as pure Python functions or web services. Each function (web service) is called a view (just like in the Mediator layer). The Controller uses the Mediator to get entity object adapters to create and operate on Bean objects. Mapping between the Entity Object Adapter and the Bean Object classes is done by the Application View package. Because the Mediator is accessed via the REST interface, there is a need to cover the REST communication with a Facade component responsible for a networking communication.

The Controller is equipped with the Application Web Service Connector sub-package used by application views to realize the Mediator communication and orchestration. All web services (views) are available through the REST interface. This is why the Controller contains the Application Web Service Definition sub-package, which can be built with the Pyramid framework, and works in the same way as an analogue package in the Mediator layer.

Controller's views return different types of data. Sometimes they pass prepared data to the Template package (bean objects) to get from it an HTML document. Sometimes views provide only pure bean objects in the serialized (JSON) notation (bean object adapters – analogously to entity object adapters). This kind of data can be used by different web services, for example JavaScript Controllers or other applications.

In Figure 3 we present only one application consuming the Business Layer (plus the Smart Client application), but in the STCBMER model the Business Layer can serve the business services (as web services) to more than one application written in various technologies.

The *Bean* layer is just a set of classes that define application objects. Objects of those classes can be used by the Template layer to generate the web front-end (HTML, CSS, eventually JavaScript). But in some cases Bean objects are just returned as a result of invoking an application view (a web service). In this case they are mapped by the Bean Serializer to the bean object adapters. Bean objects are defined dynamically by Controller views and can be stored in JSON notation, which is close to a native type of Dictionary in Python and it is a native type for the JavaScript language. The JSON notation is nowadays widely used in web systems because the text representation of JSON objects (which in the end is sent via HTTP) is quite lightweight and easy to parse in various technologies.

The *Template* layer is responsible for generating a web front-end using Bean objects (prepared by the Controller module in views). While views (Controller) construct data to be displayed, the Template is responsible for how data will be displayed. In our technology-specific solution, the Template layer uses the Mako template library written in Python and is responsible for generating HTML documents (sometimes with some additional CSS and JavaScript, if the documents have to be prepared dynamically). In general the Template layer is used also to generate different types of documents which might be needed by various remote applications/systems.

The *Smart Client* layer consists of the JavaScript Controller, JavaScript Template, JavaScript Bean and JavaScript Web Service Connector. In our e-marketplace project (not described here, as mentioned in passing), all modules except the JavaScript Web Service Connector are provided by the Angular.js framework, which is based on the M-V-VM (Model-View-ViewModel) pattern. This pattern is used by a large number of web frameworks, also by JavaScript frameworks, working usually in a homogeneous memory environment (all objects can access each other).

The ViewModel listens to the Model object (usually as a Subscriber), and after triggering an event, does some application logic (for example changing the state of other Model objects). In the end, the ViewModel can publish its own event object, so that the View (which is usually a Subscriber) could re-render the user interface based on ViewModel attributes (which the ViewModel defines for each Model – similar to the Adapter pattern). Of course, the Angular.js framework is just our platform of choice for the Smart Client layer in our e-marketplace project and it could be realised with different technologies based on various patterns (M-V-VM is just an example).

The STCBMER meta-architecture is an extension of the PCBMER meta-architecture to cater for service-oriented systems and applications. Both meta-architectures share the same complexity-minimizing architectural principles. The four principles discussed earlier (namely CEP, DDP, UNP, and NCP) are all honoured by the STCBMER meta-architecture.

4 RELATED WORK

The word "architecture" is an overloaded term in computing. It is used to denote physical architectural design as well as logical architectural design. In its physical meaning, it refers to the allocation of software components, and communication patterns between them, to computing nodes forming architectural tiers. In its logical meaning (as addressed in this paper), it refers to the allocation of software components, and communication patterns between them, to computing packages forming architectural layers. In between these physical and logical meanings, there are various mixed uses of the word "architecture", including SOA, ADL (Architecture Description Language), Enterprise Architecture, etc.

Although the term "architecture" is overloaded and even overused in the literature, it comes as a surprise that very little research has been reported on layered architectural design for the development of software systems and applications. While complete meta-architectural proposals are difficult to find, the literature is full of architectural guidelines and patterns of which the Core J2EE Patterns (Alur et al., 2003) and the PEAA (Patterns of Enterprise Application Architecture) (Fowler, 2003) have made most impact on our work.

The philosophical underpinning of structuring our models of meta-architectures into hierarchical layers comes from the holonic approach to science as the most promising way to take control over complexity of artificial systems (Koestler, 1967; Koestler, 1980; Capra, 1982; Agazzi, 2002). Apart from dismissing network structures as untenable for construction of complex adaptive systems, the holonic approach explains so called SOHO (Self-regulating Open Hierarchic Order) properties in biological systems. These properties provide a basis for better understanding of human-made systems and how adaptive complex systems should be modelled.

Software complexity underpins all efforts to achieve software quality. Software quality models and standards, such as SQuaRE (ISO, 2011), tend to concentrate on software product quality, but recognize that it is not possible to produce a quality product without having a quality process that defines lifecycle activities. It is in the very nature of software engineering that a major activity within a software quality process is change management.

There is a growing body of research on service change management (e.g. Wang and Wang, 2013), but we do not know of published works that would link change management in service-oriented systems to architectural design as the crux of complexity management and software adaptability.

Similarly with regard to software metrics - a huge number of generic software metrics have been proposed (e.g. Fenton and Pfleeger, 1997). There exist also proposals of metrics targeting service-oriented systems (e.g. Perepletchikov and Ryan, 2011). However, the metrics are not sufficiently linked to the quality assurance processes that would be enforcing architectural design in the software. In other words, the metrics are reactive rather than proactive.

The same observation applies to the DSM method as a visualization of software complexity as well as a vehicle for calculating complexity metrics (Eppinger and Browning, 2012; Sangal et al., 2005). The expressive power of DSM has been mostly used for discovering complexity problems in the software, and for fixing problems like cyclic dependencies, but there is a lack of tangible results reporting round-trip engineering use of DSM to control software complexity.

5 CONCLUSION

The introduction and description of the STCBMER meta-architecture is a contribution of this paper. When we started working on a meta-architecture proposal for service-oriented systems and applications, we expected a notable departure from our PCBMER meta-architecture developed for conventional enterprise systems. It has turned out that STCBMER and PCBMER are similar.

The STCBMER introduces one new layer built with JavaScript and few new sub-packages. A web browser is now an explicit part of the new model.

The Entity and the Bean layers are now defined with more details. Each consists of two sub-packages: one containing the real objects (the Entity Object and the Bean Object) and the second representing objects ready to send via a web service interface (the Entity Object Adapter and the Application Object Adapter).

To map business/application objects to proper adapters, special packages are introduced: the Entity Serializer and the Bean Serializer. Since the communication between the Smart Client Logic, the Application Logic and the Business Logic is organized with a web service technology, special web service packages are introduced. The first type of packages needed to organize a web service communication, are packages which contain the API definition: the Business Web Service Definition and the Application Web Service Definition. Those packages define how the API of each layer looks like. The second type of packages are web service connectors: the JavaScript Web Service Connector and the Application Web Service Connector.

Some differences between STCBMER and the PCBMER can be noticed in dependency relationships. New dependencies exist to reflect the fact that the new meta-architecture works in a web service environment. For example in the PCBMER the direct dependency between Controller and the Entity (Controller's objects construct Bean objects from the Entity objects) is in the STCBMER defined as a dependency between the Controller package and the Entity Object Adapter package. But since the Entity Object Adapter is a sub-package of the Entity package, dependency between the Controller and the Entity layers still exists.

Other differences can also be noticed – not in the architecture definition but in default technical environment. The PCBMER has not been defined to work in a web environment, or in a service-oriented model. The STCBMER is an elaborated version of PCBMER designed to be able to work in those environments.

6 FUTURE WORK

The STCBMER meta-architecture proposed in this paper has been validated in the field on a large project for the e-marketplace domain. However, the usability of the meta-architecture is only a partial proof of its value. In the follow-up research we need to develop concrete metrics that can be used to measure complexity of comparable versions of software designs and systems built according to the STCBMER framework.

The metrics will measure dependency relationships in software. To this aim, we first need to classify all kinds of dependencies in service-oriented systems and applications that have a clear impact on software complexity. At the beginning we will concentrate on coarse-grained dependencies: message dependencies (addressed in this paper, but not in the context of metrics), event dependencies and interface dependencies. For the service-oriented systems and applications, a special attention will need to be placed on the interface dependencies as they constitute the essence of web services. As an important aspect of our future research, we will need to discuss the strengths/weights of various kinds of dependencies on the complexity and adaptability of software.

We stress that the complexity metrics are not absolute measures – their value is only in

comparison to other (previous) versions of system/application architectural designs and in successive versions of software products. In Maciaszek (2008b) we discussed the ways of using DSM (Dependency Structure Matrix) for the analysis and comparison of system/software complexity. Today many tools exist that support the DSM method and that additionally integrate with popular IDE-s, such as Eclipse, Visual Studio or IntelliJ.

The tool support is important here as the complexity management has both forward and reverse-engineering dimension. The software needs to be forward-engineered according to its architectural design, but we also need to validate the code conformance with the architectural principles.

Contemporary tools offer visualization of dependencies in the code-base not just at particular levels, such as method-to-method, class-to-class, directory-to-directory, but also across levels, such as function-to-type, namespace-to-class, jar-to-method. One of such tools is Structure101 (Structure, 2014).

Structure101 and most other tools are predominantly reverse engineering tools, more reactive than proactive. Structure101 provides, however, a specialized module, called Architecture Development Environment (ADE), to define architectural rules and guide conformance inside an IDE. The "proactivity" remains at the architecture (instantiation) level and meta-architecture is offered by the tool itself, but we plan to use ADE to define the STCBMER principles for various industrial studies and software development projects.

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.

Alur, D., Crupi, J., Malks, D., 2003. *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed., Prentice Hall.

Capra, F. (1982): *The Turning Point. Science, Society, and the Rising Culture*. Flamingo.

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.

Fenton, N.E., Pfleeger, S.L., 1997. *Software Metrics. A Rigorous and Practical Approach,* 2nd ed., PWS Publishing Company.

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

Glass, R.L., 2005. The Link Between Software Quality and Software Maintenance. *IT Metrics and Productivity Journal*, November, p.29.

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.

Koestler, A., 1980. *Bricks to Babel,* Random House.

Koestler, A., 1967. *The Ghost in the Machine*, Penguin Group, London.

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

Maciaszek, L.A., 2007. An Investigation of Software Holons - The 'adHOCS' Approach. In *Argumenta Oeconomica* Vol.19, No.1-2, pp.1-40.

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

Maciaszek, L.A., 2009. Architecture-Centric Software Quality Management, In Cordeiro, J., Hammoudi, S., Filipe, J. (Eds), *Web Information Systems and Technologies, WEBIST 2008,* LNBIP 18, Springer.

Maciaszek, L.A., 2006. From Hubs Via Holons to an Adaptive Meta-Architecture – the "AD-HOC" Approach. In Sacha, K. (Ed.), *IFIP International Federation for Information Processing, Vol. 227, Software Engineering Techniques: Design for Quality,* pp.1-13, Springer.

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

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

Perepletchikov, M., Ryan, C., 2011: *A Controlled Experiment for Evaluating the Impact of Coupling on the Maintainability of Service-Oriented Software*, IEEE Trans. On Soft. Eng., Vol. 37, No. 4, pp.449-465

Sangal, N. Jordan, E. Sinha, V., Jackson, D., 2005. Using Dependency Models to Manage Complex Software Architecture, In *Procs. OOPSLA'05*, pp.167-176, ACM.

Structure, 2014. *Structure101*, http://structure101.com/, viewed February 2014.

Wang Yi., Wang Ying (2013). A Survey of Change Management in Service-Based Environments, In *SOCA*, pp.259-273, Springer

Wing, J.M., 2008. *Five Deep Questions in Computing*. Comm. of the ACM, Vol. 51, No.1, pp.58-60.