Transdisciplinary Engineering: Crossing Boundaries
M. Borsato et al. (Eds.)

doi:10.3233/978-1-61499-703-0-818

Toward Systematic Software Reuse: From Concept to Modular Software Implementation

Eric C. SOUZA, Fabio K. TAKASE[1], Rafael L. COSTA and Fabio S. AGUCHIKU
Atech – Negócios em Tecnologias S/A (Grupo Embraer)
Rua do Rocio, 313, Vila Olímpia, São Paulo – SP, Brazil

Abstract. A considerable part of the development effort of systems nowadays is related to the design and implementation tasks of the software components of the system. The main motivation for this work is the expected benefits from a successful implementation of systematic software reuse by the organization, as described in IEEE 1517 - 2010: Increase software productivity; Shorten software development and maintenance time; Reduce duplication of effort; Move personnel, tools, and methods more easily among projects; Reduce software development and maintenance costs; Produce higher quality software products. This work will focus attention on what is needed for an organization to establish systematic software reuse from a process and standards point of view, to repeatedly exploit reuse opportunities in multiple software projects or products, and on how this can be implemented and harnessed on real world projects. To accomplish this, some development cases were selected from the available literature and effective approaches to the execution of tasks from processes of the Atech product life cycle model were exercised in two case studies. The application of effective approaches to development proved to be challenging, as whole new set of tools and processes are being demanded to adress the complexity of modern systems.

Keywords. Software reuse, architectural mismatch, configuration management.

Introduction

The technology market of today demands the development of large systems, with increasing complexity, within a shorter development time and with assured quality. This demand-pressing market requires from system providers high responsiveness and great flexibility during development of solutions. As a result, productivity and quality issues are increasing in the software development and implementation phases. The resulting impact on development intended for software intensive systems is significant. Higher product throughput and quality may be achieved by the successful construction of systems from already developed and tested components; a development concept introduced into the software engineering arena during the late 1960s [1,2].

This work intends to characterize the context and issues addressed in the challenge of achieving practical systematic reuse at Atech S.A. during development of a software intensive simulation system. To accomplish this goal, this note presents the standards view of processes that supports the systematic reuse of software; a

[1] Corresponding Author, E-Mail: ftakase@atech.com.br

bibliographic review of the software reuse challenge from the software architecture & components points of view. In particular, the use of a component-based development method [2,3] is applied to solve the reuse problem in-house, where components are managed through building block specification, classification and retrieval.

The difficulty inherent in the reuse problem can be briefly described by the mismatch of interface of the two software components depicted as *A* and *B* in Figure 1: they don't work together if they mismatch, i.e. if they were originally designed to work under distinct architectural assumptions.

1. Software Reuse from the Standards' Point of View

Software reuse, as stated in IEEE 1517-2010 [4], is concerned with the creation of new products from existing software and systems. This is not restricted to the creation and use of a library of assets, however. To achieve the successful reuse, activities related to the reuse must be included in the life cycle processes used to create the reuse assets. It is important to note that the term reuse considered here conveys the meaning given by the IEEE 1517 definition as *systematic reuse*. Simply stated, systematic reuse is the avoidance of multiple versions of otherwise common elements. From the organization point of view, what is desired is to repeatedly exploit reuse opportunities in multiple software projects or products.

In this work, our focus is set on the integration of reuse in software specific life cycle processes, although the organization may benefit from integration of reuse in the whole system life cycle processes. The software specific life cycle processes may be decomposed into Software Implementation and Support Processes.

Software Implementation Processes comprises the activities and tasks performed by the developer when developing software products with reuse of assets. These standards include the activities for implementation requirement analysis, architectural design, detailed design, construction, integration and qualification testing. These processes will not be reproduced here for the sake of brevity. This listing also presents additional expected outcomes from each processes and tasks added to some activities. Refer to IEEE 12207-2008 [5] to obtain a better understanding of the processes, tasks and activities that form the baseline to which the added features refer to.

Additional tasks also apply to the development process. These are the Software Support Processes, [4]. Application of these Support Process is explained in Section 3.

Figure 1. Composition of two software components and the problem of component interface mismatch.

2. Software Reuse from the Practitioner's Point of View

The inherent difficulties that arise when building software applications, see [6] for example, from existing parts are not new. In 1995, Garlan and co-workers [7] identified

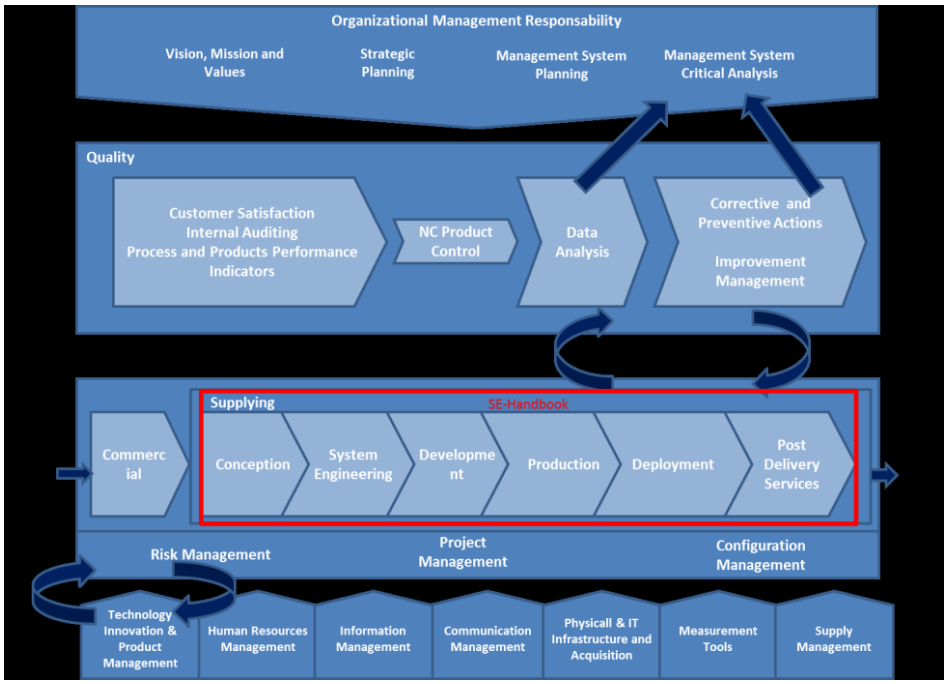

Figure 2. The Atech System Life Cycle Model.

what they called the *architectural mismatch*. In their work, the traditional solution to the reuse problem was reviewed and the root cause of the unsuccessful approaches identified, [7]:

> *"Clearly some blame can be attributed to the lack of existing pieces to build on, or our inability to locate the desired pieces when they do exist. Over time, we may expect progress in this area through the creation of more and better component libraries as well as improved mechanisms to access their contents…*
>
> *… even when the components are in hand, there remain other fundamental problems that arise because the chosen parts do not fit together well. In many cases mismatches may be caused by low-level problems of interoperability, such as incompatibilities in programming languages, operating platforms, or database schemas. These are hard problems to overcome..."*

In another work, Garlan and co-workers [8] showed that the system was built from parts that were designed to be reused. Developers of both, parts and system, were aware of implementations details and the implementation language. The execution platform did not originate new issues, as all the parts were written in C/C++. System development began with the expectation that a single person during a 6-month period would accomplish the job. It soon became apparent that their endeavor would require 2-year of work with 5-person-year. With far longer development time, the expected system with the expected functionality was built, but the *"…resulting system was sluggish, huge, brittle, and difficult to maintain*", [8]. The usual excuses to development failure did not apply to this project. The parts were engineered for reuse, the implementers were skilled, the requirements and functionalities were well known,

the development team was familiar with the source code and implementation languages and the parts were used in accordance with their functional purpose. Fourteen (14) years later, in 2009, Garlan and co-workers [9] revisited the problem and contrary to their expectations, they showed that this problem still persists and new issues were added to this landscape. These issues were summarized as: excessive code size, poor performance, need to modify external packages, need to reinvent existing functions, unnecessarily complicated tools, error-prone construction process. Once more they were "*failing miserably*" to employ reuse to its fullest.

The architectural mismatch may be defined, in a simplistic form, as the intrinsic incompatibilities arising when attempting to connect and make operate together the software components that were originally devised under distinct architectural styles. To overcome this mismatch, adaptations need to be present to connect reusable assets that were built to be part of solutions in different architectural styles. Refer to [9] and the references therein for a detailed explanation of architectural styles. Shaw [10] presented the architectural mismatch as a packaging problem and identified a set of integration patterns. Although convenient, this approach introduces a lot of compatibility code in the application.

To expose the architectural mismatch, a simple system architectural model should be used. According to this simple model, an architecture is composed by Components – primary computational and storage elements of the system – and by Connectors, that do not in general correspond directly to compilation units, see Fig. 1. From the above definition, four categories of architectural mismatch can be identified, details in [9]:

1. Assumptions about the nature of the components: infrastructure-assumptions; control model-assumptions; model-assumptions about how environment will manipulate data managed by a component.
2. Assumptions about the nature of the connectors: protocols-assumptions about the patterns of interaction characterized by a connector; data model-assumptions about the kind of data that is communicated.
3. Assumptions about the global architectural structure: Related to the topology of the system communications and the presence of particular components and connectors.
4. Assumptions about the construction process: In many cases the components and connectors are produced by instantiating a generic building block.

The above mismatch categories proved helpful during conception and application efforts of the solution targeting SW reuse to the development processes.

3. Application of an Effective Approach to Development Processes

The task of making developers aware of the tasks and activities to be performed to achieve the systematic reuse in the organization is not a simple one. For this initiative, a more agile and responsive approach was adopted, given a single assumption was made to make small but effective interventions in a running development project. This approach considered a compromise solution, harmonizing good development practices and rigid application of norms.

In this section the development project selected for this work is briefly introduced and its relevant characteristics are described. The approach to implement the changes for systematic reuse and the stakeholders directly involved in this work are presented. A brief description of the CM processes implemented at Atech is described first

followed thereafter by the application IEEE 1517 standard goals, presented in [4] as lists of Implementation and Support Processes, in the Atech context, in Section 3.1.

The running development project selected for this initiative comprises the development of a simulator to stimulate the Atech Air Traffic Control, or ATC, solution for training purposes. The complete ATC solution of Atech comprises more than 50 modules operating in distributed environment and with very different deployment combinations to provide services and support to control, training, planning and simulation activities. The modules were developed by different teams and depending on the constraints in different technologies and programming languages such as C, C++, and Java. This solution comprises large blocks of Atech expertise knowledge, and this knowledge is already coded into software blocks with very similar application, but originally intended for slightly different business domains. From the functionality point of view, almost 60% of the solution could be reused. The server side of the available component candidates for reuse is coded in C programming language and the available GUI component candidates for reuse are implemented in C/C++ (Motif) and on Java platform.

Tasks and activities were prioritized with emphasis on the software components design and on the control over components versioning and usage (configuration). To enforce the application of the recommended practices and to be more responsive to the needs of changes, an architecture office was created to perform the design of subsystems (system architecture design) and the design of the software components. The architects allocated to this office were also responsible for the detailed design of the components, selection of tools to support and automate repetitive tasks and give implementation orientation to the software developers. All the support needed to the configuration management of the software components was given by the Atech CM – Configuration Management – area. This close relationship was fundamental as the CM supporting area had to change some procedures to allow more fine grained control over the source code, building procedures and binary asset versioning.

From the configuration management point of view, the practice adopted in Atech is in accordance with the definition given in the Configuration management guidance military handbook [11], as a *"management process for establishing and maintaining consistency of a product's performance, functional, and physical attributes with its requirements, design and operational information throughout its life."* It is supported by four processes to establish and maintain this consistency: Configuration identification, Configuration Control, Configuration Status Accounting, and Configuration Audit. Refer to [11] for in-depth description of how each one of associated processes work in configuration management.

Configuration Identification is the process that permits to identify the system, its subsystems, modules, related documentation and configuration items. A configuration item is a part of a system that has a specified function and has its development individually controlled. Normally those items are primary items (modules) that will be reused in many projects and critical items but they can also be items used in the installation of the product and items that a client desires to control. In a hardware-software system, software is always considered a configuration item. Another function of the configuration identification is the creation of unique IDs for version and tags to allow the development team control its advancement and know what was delivered to the client. Tags are snapshots of a moment of the project that can be used to store, access and recover data generated on that particular point in time.

Configuration control is the coordination of all the asset changes that occur in a project, as well as all new versions that are created by these changes. It has great impact in the reuse of software since it allows the development team to track the impacts of each change of a module on every system configuration that may depend on it. Additionally, configuration status accounting is the process where all the data generated by the project is stored, accessed and recovered if needed. The status accounting store the data in the form of tags and baselines so a developer can restore its work form a predetermined moment in time, or a client can always recover the exactly same data for a product.

Finally, the configuration audit is the configuration analysis of all that was done so far. It inquires whether functions observed during the beginning of the project were applied at the end of the project, if the physical product is the same as the one documented or, if not, whether all the changes were accepted and registered. In a way, it is a final analysis given by the development and configuration management teams.

3.1. Results of Application of an Effective Approach to Development Processes

The following paragraphs detail application of the reuse approach as described in IEEE 1517 [4] to process tasks in software development processes. The application of Implementation Processes is listed first.

Software Implementation Process:

- life cycle process model: the Atech system life cycle model is already defined and shown in Figure 2. Phases are defined according to the organization management view with the related milestones and decision gates. For the design and development processes, tasks and activities are already performed in this framework, tasks and activities related to the reuse that were performed are already in accordance with this life cycle model.
- standards, methods, tools and programming languages: tools and languages for the project were carefully selected; for the sake of brevidy these are not listed here. Special emphasis is given to the adoption of the NAR plugin to the Apache Maven tool. This plugin gave the needed support to the definition of software components based on configuration of configuration items, or CIs, in an hybrid environment with components developed in C/C++ language and in Java language. Further details on this tool can be found in a description of the use of this tool in the LHC at CERN [12].
- communication issues: in this work, the communication mechanism is still informal and highly relies on the communication skills of the members of the architecture team.

Software Requirements Analysis Process: although not the main purpose of this initiative, this process had to be addressed as the project control and audit procedures relies on artifacts related to the software specification and traceability to requirements.

Software Architectural Design Process: all the tasks were performed by the team of architects. The identification of subsystems and system elements, the messages and interfaces gave sufficient information to setup the system domain architecture. In this phase it was possible to identify the configuration items that were part of the solution. In this process an architectural style, based on data distribution services specified by the OMG, or the Object Management Group, was chosen and, from this point on, all the assets were designed to play their roles in this integration environment.

Software Detailed Design Process: the tasks related to the detailed design were also performed by the team of architects. During the realization of some activities it was identified that new configuration items could be defined and with the support of CM staff it was just a matter of creating new CIs in the overall project structure.

Software Construction Process: the software design was done with testing in mind. Wherever possible, unit testing was implemented and through the continuous build/integration, supported by the tools considered for development, the build process was continuously exercised and enhanced.

Software Integration Process: this process was simplified by the fact that all the assets, such as the SW components themselves, were designed for reuse in a very specific architectural style. The interfaces could be separated in compiled binary configuration items and each system element could be described as a configuration of CIs, and the system itself could be described as a configuration of system elements. This represents the adopted solution to problem in Fig. 1.

Software Qualification Testing Process: This process and its specificities were not addressed in this work.

The application of the Support Processes of [4] is considered next.

Software Documentation Management Process: Document reuse was a goal at first, but the obsolescence and availability of software applications to edit and update old documents made this impossible to do. Diagrams had to be redrawn.

Software Configuration Management Process: The process of configuration management is already running at Atech, as shown in the previous session. It was needed to add support to the management of binary assets, in order to guarantee the reconstruction of a given build environment (compilation) and in order to allow the construction of configuration without the need to recompile all the modules every time that a new version of the system is generated. The main difficulty was the hybrid development environment with C, C++ and Java modules.

Software Quality Assurance Process: This process and its specificities were not addressed in this work.

Domain Engineering Process: The application domain was identified and an integration framework was developed.

Reuse asset Management Process: A formal process is still not established, but the requirements for the activities and tasks for this are being gathered.

Reuse has been observed and validated at Atech with software components reuse independently realized by different development groups and by independence of component version management through the appropriate tools. In particular, the composition of two components of Fig.1 is a binary used as the data distribution mechanism. This binary is a CI and serves as a component interface even for hybrid systems, i.e. systems developed under different platforms (e.g., C/C++ and Java).

4. Application to Development Case Studies

A conceptual framework devised at Atech for the development of complex, software intensive systems. It embodies the idea that complex systems are the sum of smaller and simpler parts, or system components. This platform also borrows ideas present in the architectural frameworks of the U.S. Department of Defense (DoDAF) and the British Ministry of Defence (MoDAF); [13-14]. These frameworks focus on the effective sharing of architectural “data” – rather than a focus on a product-centric

architecture development process – to support decision making during development. Visualization of architectural data is accomplished through models with graphical representation thus facilitating communication and understanding among stakeholders. In particular, the DoDAF v2.0 defines Architecture development into a 6-step process. These frameworks are intended to guide the development of large systems, involving complex integration of many parts, and both generate development artifact models, called viewpoints, to meet the specific needs of the many business interests being represented. Each viewpoint represents a detailed description of the architecture from a different perspective. These views correspond to architecture Operational Capabilities.

At Atech, system components are called Operational Capabilities where each represents a self-contained implementation of a specific set of functionalities and analysis tools intended for operational use on a given domain application. As an example, in the ATC domain, the functionalities and analysis tools required for a thorough, efficient and effective air traffic control are realized by *Building Blocks*, which together make up or implement the ATC Operational Capability. The use of building blocks, inspired by the method presented in [15], is justified as a means to exploit reuse in system development, in compliance with IEEE 1517-2010 [4]. The use of building blocks for reuse becomes even more appealing when efficiency in large systems is a sought for requirement.

Perhaps the first thing that comes to mind when delving into the ATC domain is the picture of a large controller console with a myriad of visual elements crowding the display screen. This is implemented by the graphical geospatial reference interface along with accompanying support air traffic control functionalities. Another indispensable Building Block of this same domain is entrusted with the task of managing the position of flying aircraft, or flight tracks, on the control display screen. Additionally, alert managers, NOTAM managers, and controller issued objects manager are also present in the ATC Operational Capability. Notice, moreover, that systems in the ATC domain are essentially distributed systems where distributed computing is heavily employed and, thus, one important Building Block needed in the ATC system implements communication between nodes in a system network.

The application of the above processes has yield tangible benefits and software component reuse is already a present reality at Atech. The first system is a concept ATC system used to demonstrate specific functionalities with international partners. These functionalities included flight track, aeronautical, and boundary coordination sharing of information. It is estimated that as much as 30% of the development and implementation effort was saved because of software component reuse.

Two other systems were developed under the conceptual platform described above. The first system provides several types of information in order to increase situational awareness and improve the decision making process regarding the surveillance of the Brazilian coast. The main concept of this system is to present several types of information, including those provided by other external parties, in an integrated form. The adopted development framework allowed for parallelization of efforts naturally: different components were developed concomitantly and, therefore, the components integration to achieve the system was greatly simplified. The system presents contacts information, such as vessel and aircraft, on the HMI. Vessel information is acquired from system external tracking services. These services provide several information about a vessel, including position, speed, course, identification, timestamp of the detection, size, callsign and others. The HMI display the vessels' current positions, speed and courses, identification and a history of vessels position.

Vessels can be classified through HMI according to the military standards. This application was one of the first developed after introduction of the processes and benefited little from the reuse. It served, nevertheless, to pinpoint several potential components and interfaces to be re-used in subsequent development projects. The second application is a distributed system solution for tactical Command and Control missions. In the typical mission scenario, three types of land vehicles are used collaboratively. Each vehicle has a different objective and is, therefore, configured with a distinct set of system functionalities. They represent different system network nodes. As a result, the required data for each vehicle to carry out mission objectives is different since each type of vehicle employs a distinct system configuration. Vehicles' systems integration translates to publishing and reading subscribed data and presenting data on the HMI appropriately in a way that improves the decision making process. Situational awareness improves tactical decision making on the field. The system installed on the vehicles achieves this with information available from several sensors. This information is acquired by partners systems, i.e. systems outside the mission network, and the C2 system integrates data obtained from local sensors with those acquired from other systems and presents sensor information on a single user interface. Network data comprises GPS location data, video streaming, other sensor data, where each one is transmitted using different protocols. Approximately 40% of application development originated from the reuse of existing components.

An extra system development project also benefited from the processes described above for reuse. In-depth detail about this system development will not be presented here, sufficive to say that it is related to ATC and training operations. Development of the first version of this system achieved 40% of component reuse for the training module and 70% for the full ATC system implementation.

It is believed that greater measures of software reuse will be achieved as the components implementations or building blocks specifically, and process mature in time. Developer awareness about existing components also contributes; although this is not critical given CM tools indicate availability of components for development. Finally, appropriate project scope definition contributes significantly to reuse, given the influence of requirements specification on development and implementation phases.

5. Final Remarks

The subsystems and system elements of the simulation platform solution were developed following the practices recommended to support systematic software reuse. The communication issues were addressed through the creation of an architecture team responsible for making design decisions and detailed design of the software components. Without the support of the CM staff and the appropriate tools for version control, project configuration and artifact management the growing number of CIs would not be manageable. A key decision was the use of the NAR plugin to the Apache Maven tool, which allowed the smooth integration of *make* based build configuration of many legacy subsystems written in C language in an artifact management tool written to work with Java. This allowed for resolution of the overwhelming complexity related to platform architecture issues, as C compilations are not platform independent as Java byte code. Another important aspect of this work was the definition of an architectural style based on OMG's DDS. This decision was based mainly on the past experience on critical systems development at Atech. One desirable effect of the use of

DDS was the possibility of packaging the interface related code derived from IDL in compiled binary CIs, an approach that guarantees that two subsystems with the same version of the interface package in their configuration will be able to change information without problems. It is important to note that assumptions related to the data model and communication protocols may be encapsulated in the IDL and the overall data distribution mechanism is sufficiently generic to support an easy integration. The publisher/subscribe model also provides a solution with low coupling between the modules, imposing low dependency related to the flow control of processing. The effort related to the implementation of the reuse supporting tasks was due to the need of designing for reuse and the effort to maintain the design of system elements compatible with the architectural style defined for the project. The high modularity of the system, at first, presented a problem to the implementation activities, as modules were not available for use at the beginning, meaning that small developments were followed by quite complex steps to compile and publish new versions of the CIs. As the system evolved and some modules releases were available, the CIs management overhead became small, and the independence between modules and their versions was exploited and simplified the development process, mainly the building process, keeping explicit dependency relationship between modules in project model files. The documentation related to the binary assets with greater possibility of reuse is made available in an html site, and during this work, other development projects at Atech could make use of the newly deployed binary assets and their documentation.

References

[1] M. Gasparic, A. Janes, A. Sillitti, G. Succi, An Analysis of a Project Reuse Approach in an Industrial Setting, *ICSR 2015*, LNCS 8919, Springer, pp. 164–171, 2014.

[2] S.G. Shiva, L.A. Shala, Software Reuse: Research and Practice, *IEEE International Conference on Information Technology*, 2007.

[3] W.B. Frakes, K. Kang, Software Reuse Research: Status and Future, *IEEE Transactions on Software Engineering*, 31(7), pp. 529-536, July, 2005

[4] IEEE Std. 1517-2010, *IEEE standard for information technology - system and software life cycle processes - reuse processes*, IEEE, pages 1-51, 2010.

[5] IEEE Std. 12207-2008, *IEEE standard for information technology - system and software engineering – software life cycle processes*, IEEE, pages c1-138, 2008.

[6] F.P. Brooks, *The Design of Design: Essays from a Computer Scientist*, Pearson, Boston, 2010.

[7] D. Garlan, R. Allan And J. Ocklerbloom, Architectural mismatch, or, why it's hard to build systems out of existing parts, *Proceedings of the 17th Int. Conf. on Software Engineering*, pp. 179-185, 1995.

[8] D. Garlan, R. Allan, J. Ocklerbloom, Architectural mismatch: Why reuse is so hard, *IEEE Software*, Vol. 12, 1995, No. 6, pp. 17-26.

[9] D. Garlan, R. Allan, J.M. Ockerbloom, Architectural mismatch: Why reuse is still so hard, *IEEE Software*, 2009, pp. 66-69.

[10] M. Shaw, Architectural issues in software reuse: It's not just the functionality, it's the packaging, *Proceedings of the Symposium on Software Reuse*, 1995, pages 3-6.

[11] U.S. DEPARTMENT OF DEFENSE. MIL-HDBK-61A - *Military Handbook Configuration Management Guidance*, February, 2001.

[12] J.N. Xuan, B. Copy, M. Donszelmann, A C/C++ build system based on maven for the LHC control system, *Proceedings of ICALEPCS*, 2011, pages 848-851.

[13] US Department Of Defense, *The DoDAF Architecture Framework*, Ver.2.02. Online: http://dodcio.defense.gov/Portals/0/Documents/DODAF/DoDAF_v2-02_web.pdf, Acces. Feb. 25, 2016.

[14] UK Government Ministry Of Defence, *MOD Architecture Framework*. Online Portal: https://www.gov.uk/guidance/mod-architecture-framework, Accessed: Feb. 25, 2016.

[15] J.K. Muller, *The Building Block Method*, Koninklijke Philips Electronics NV, 2003.