

Software Architecture Enabling Effective Control of Selected Quality Aspects

Michal Žemlička[1,2(✉)] and Jaroslav Král[3]

[1] Faculty of Mathematics and Physics, Charles University,
Malostranské nám. 25, 118 00 Prague 1, Czech Republic
zemlicka@sisal.mff.cuni.cz
[2] AŽD Praha, Závod Technika,
Žirovnická 3146/2, Záběhlice, 106 00 Prague 10, Czech Republic
zemlicka.michal@azd.cz
[3] Faculty of Informatics, Masaryk University,
Botanická 68a, 602 00 Brno, Czech Republic
kralq@seznam.cz

Abstract. The number of requirements on information and control systems features grows. They often contain flexibility, safety, reliability, and security. Balancing all these features is not easy. We propose architectural turn enabling to control flexibility and security (and to some extent also reliability/availability). The proposed solution enables agile involvement of people into processes. It simplifies solution of rare cases or emergency situations. It can be used during system development and maintenance. We discuss some implementation details.

Keywords: Software quality aspects control · Software confederations · System flexibility

1 Introduction

There is a strong tendency to digitalize a growing part of human society as well as its autonomous subsystems like industry, social processes, ecology, education, artificial intelligence, robotics, or e-government.

Achieved solutions are enhanced and extended very quickly due to changing needs, aims, and advances in technical domain. The result is dynamic computerization of human society. Many human activities are computerized already now. It can be surprising that the importance of human involvement into the processes grows. Human must be able to formulate aims, implement systems, maintain them, and control (supervise) them. It implies that the system must be understandable for involved people and must be structured as a network of collaborating autonomous components having properties of services.

We propose in the area of information and control systems a solution enabling all human-oriented rules. The proposed solution is advantageous for all human-oriented roles (front-end users, IT specialists, customers, managers, experts, etc.) needed during system development, maintenance, and use.

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11623, pp. 536–550, 2019.
https://doi.org/10.1007/978-3-030-24308-1_43

Current trends towards Industry 5.0 [20] and Society 5.0 [19] increase the importance of human involvement in software system development and use. Human involvement increases the necessity to use specific types of service-oriented architecture. The service orientation with emphasized human involvement must be designed/structured as a complex of cooperating autonomous entities (services).

They must have understandable interface and function and must communicate in the way understandable to the involved people (e.g. domain experts) – typically using messages having easily understandable semantics. To achieve it, the messages must be of textual nature and the services should not be too complex. In order to keep the system structure understandable to people, the system granularity should be adequate to human limits mentioned e.g. in [14] (there should be cooperation of a limited number of partners at any level).

The systems grow. It is they consist from increasing number of services, the services become more complex and their communication becomes also more complex. The system becomes less understandable and more difficult to be maintained and properly used.

With respect to 7 ± 2 rule [14] it makes sense to limit the number of interfacing services. As we need to handle many services at the same time, solution of this issue can be to build the systems having hierarchical or very similar architecture. We propose a specific variant of service-oriented architecture (SOA) having combinable services. Their combinations can be treated again as combinable services. It also enables reuse of existing services.

With the increasing number of requirements being often the requirements on quality aspects [10,26] there is an increasing chance that the requirements can be contradictory and must be orchestrated. With the growing number of requirements it grows also the chance that some of the requirements could be at least to some extent contradictory. Example of such "contradictory" requirements are security and flexibility or openness. Flexibility and openness require that data and functions should be easily available. From security point of view it is like opening a door to attacks.

We discuss design and implementation principles enabling effective orchestration of requirements and develop systems and having many further advantages [12]. We introduce a turn simplifying balancing between various requirements as well as between quality aspects of the final product. Our goal is creation of systems that are able to work (at least partially) also under emergency or unexpected conditions.

It is advantageous for users that the communication between services is understandable and they can therefore take part in debugging and maintenance. It is surprisingly advantageous also for managers and developers. The solution brings some security advantages. It also simplifies analysis of incidents. It is advantageous for both users and developers.

Communication between services must be textual to allow involved people to understand them. Properly chosen message format and processing it has many technical as well as other advantages.

Textual messages can have various formats and application rules and there are several procedures how to achieve it. Different formats as well as different forms of transformations could fit to various situations. Individual variants are discussed below.

2 Selected Quality Aspects

We focus on selected software quality attributes. We describe them and show how they influence each other.

Architectures discussed here allow improvement of many characteristics of quality according ISO 25010:2011 [10], ISO 9126-x [7–9] (compare also [26, Table 2]). They enable easier balancing e.g. between flexibility and security or between easiness of programming and user understandability of interfaces. Some issues could appear with efficiency but this issue is for many reasons weakening now.

This architecture significantly improves development and maintenance processes. It weakens the danger of failures. Surprisingly, it could be important that the system could be built and deployed stepwise and protecting proven existing interfaces – inter-human relations inclusive.

Importance of understandability and learnability of service interfaces grows. Proper design and use of service interfaces positively influences technical aspects of system – including stability, maintainability, and many others. It moreover enables to design systems providing interfaces easily usable for real-world business processes. Another advantage is that such systems could be built incrementally and some of their parts could be temporarily or forever based on human involvement or even human processing.

It leads to growing similarity between development and maintenance processes. It allows easier implementation of support of modern business processes and their management.

Modern information and control systems often integrate Internet of Things (IoT, see e.g. [3]). It is crucial that they use implementation and design principles and rules used by IoT systems. Crucial point is that IoT use messages in structured (tagged) text. The text format encodes structure understandable by human beings either directly or after simple transformations. It is possible to use results from theory of formal languges and linguistcs. It enables collaboration of different professions during system life cycle. Proper use of message formats is a precodition of high quality of the system.

It is also quite popular to connect services through middleware. Then it is virtually possible to let cooperate any service with any of the others. It is advantageous in an environment under single supervision. When there are multiple companies or organizations working together, there could be a problem with setting access rights for the shared middleware.

From various service-oriented approaches the one recommended by Thomas Erl and his team e.g. in [5] appears to be the most popular now. Moreover, service orientation as a whole loses its popularity toward cloud and edge computing.

Clouds are difficult to be integrated in a secure way. We propose a solution encapsulating cloud systems to be standard services. It enables e.g support of humans in the form supposed by Industry 5.0.

3 Real-World and Software Services

Many of software services are built as a support for real-world services or real-world agendas. Some of the real-world processes are used the same way for years, some for decades, some even for centuries. It is, the processes and agendas had enough time to be deeply studied and optimized. Their business processes could be polished to be optimal. The optimality of the processes is in their level of abstraction (they are as precise as necessary and as general as possible) in their granularity, in their procedurality (they are as procedural as necessary and as declarative as possible), etc. Similarly, there have been developed language dialects specific for given tasks. These dialects are for stable processes to some sense also optimal.

The dialects could be based on different languages – there are specific dialects covering the same purpose in all languages where the agenda makes sense (and is used for long time and within the same or similar legal framework).

When software services are created using knowledge and structure of corresponding real-world services, it is possible to consult non-software issues (system extensions and modifications inclusive) with problem domain experts: they, in such situation, understand system structure, system communication and individual processes. They can therefore recognize anything improper in the system and have better chance to find solution of the issues and challenges.

The necessary level of similarity between real-world communication and its software counterpart is still open. For programmers it is easier to use languages closer to machine processing (XML [21], JSON [11], or another language well supported by development tools). For domain experts it is easier if the language is easy to read and write – it is, if it is closer to used real-world languages. Such languages are usually simple and slightly redundant (to avoid situation when it has not been said or heard properly).

For taking full advantages of service orientation, support for human involvement is crucial – compare the differences between Industry 4.0 and 5.0 [20]. Human involvement must be supported at various levels and for various tasks. Let us mention at least some of them:

- coordination of business processes,
- system orchestration and supervision,
- processing of business processes and their individual steps.

Such involvement is a very strong tool: It must be therefore used with respect and evidence.

4 From Software System to Software Service

Complex software support for real-world services and agendas usually does not start from scratch. There are usually various applications and system covering part of the work. The questions are how to interconnect them to cooperate and whether they fulfil our expectations about interfaces (and what to do if they do not fulfil them).

We will show the creation of a complex system step by step in the bottom-up manner. We start with ensuring connectivity of the existing software, then converting the interface to problem-oriented and user understandable one, ensuring access control, creation of a hierarchy, etc. It is possible to modify classical design turns for use in our systems, see below.

4.1 Ensuring Connectivity

Existing applications are typically equipped with interface to human users. For interconnecting applications we need them to be equipped by interface reachable by other applications – ideally by a network interface supporting sending and receiving messages. The basic question is whether the yet not connected application is intended for single user or multi user use and whether it is single process or multi-process one. These features imply some basic limits like availability of concurrent use of the applications by people and computers or possible security settings and functions.

The existing user interface is usually possible to wrap by a new code to provide message connectivity. According to [25] such piece of code is called *primary gate* (PG) as it provides primary access (from the integration point of view) to the integrated (legacy, third-party, or newly-developed one) application/system (software artefact) A. Its purpose is to provide access to all functionality of A.

Technically we can use existing interfaces if they match our needs or we can accommodate the existing interface. At first, we must ensure that we can consume its outputs and provide its inputs. For such task wrapping suits well.

Textual interfaces, menus, and dialogs could be wrapped. The wrapper could be provided as an integrated part of the system (but then source codes of the wrapped system must be available) or the wrapper can work on top of the application – e.g. as its host. It can catch creation of the input and use the presented data for generating output for partners of the system. Similarly, it can capture any input opportunity and fill in the data prepared and sent by partners.

For such purpose suit better systems with textual input and output than the GUI (graphical user interface) ones – it is just simple to wrap them. The resulting situation could look like in Fig. 1 or in Fig. 2.

4.2 Providing Understandable Interface

The communication in real world is optimized for human users. It uses structured or simplified natural language. Such language is understandable and intuitive

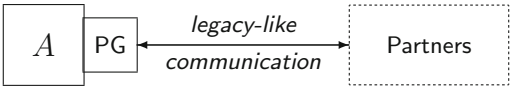


Fig. 1. Very simple application wrapped for network cooperation

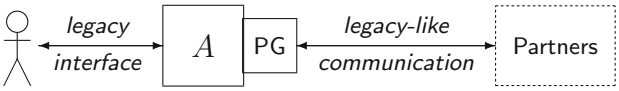


Fig. 2. Wrapped application with preserved legacy user interface

for the problem domain experts (at least the ones using the same basic natural language).

According our experience the communication languages are usually so simple that they can be parsable by LL-techniques[1]. For such languages there are tools, parser generators (like bison [1], ANTLR [18], or KindCons [22]), that can generate very efficient decoders of such languages using their formal description (LL or LR (attribute) grammars).

The programmer usually like communication using means that are well supported by available libraries – it is sending data using XML, JSON or other machine-oriented format. It is easier to let generate the serializer from record/object definition than to think about matching the stored data with existing problem-oriented language. In practice it could be reasonable to think about balancing between problem orientation and easiness of programming.

Now we can expect that the integrated application or system is in principle reachable from other applications. The issue is that the logic and interface of the integrated system could significantly differ from the logic and interfaces of its prospective partners. Accidentally, it is often the case – compare service-oriented architectures recommended by OASIS [15], Open Group [16], or Progress Software.

The interface based on existing user interface is usually revealing too much from the implementation details of A. It makes the interface dependent on the implementation of A. It can be source of change propagation of it is necessary to change some implementation details (e.g. to correct some errors). When also the cooperating partners use implementation revealing interfaces, the change propagation could hit significant part of the system.

Implementation techniques evolve in time quite quickly: every several years there are significant changes and every about ten – fifteen years there are changes also of the used programming paradigm. Updating entire system at once (to keep compatibility between parts) is usually not possible – technically it is too complex and moreover Big Bang (replacement of a big system by another one

[1] LL parsing techniques are discussed in literature on formal languages and automata theory as well as in literature on parsing and compiler construction literature – compare e.g. any of [2,6,23,24].

at once) is known as a frequent problem source [4]. Coupling systems using implementation-oriented interface and being developed using different implementation techniques or even different implementation paradigms is a very hard (if not solvable) task.

The defence against it is known – information hiding as proposed by Parnas e.g. in [17]. The use of problem-oriented languages fulfils this task well – it hides all commands and other technical details. The use of problem-oriented language makes the connection between cooperating applications more stable as the problem-oriented language is stable in time and can hide changes of implementation of any of the partner.

It is necessary to convert the available interface following the existing interface logic (typically detailed procedural and often also implementation-oriented one) to the one used in real world (usually declarative and problem-oriented one). Such conversion can be provided by special convertors (services providing the interface conversion) first described in [13] and called *front-end gate* (FEG). Generally it could convert sequences of messages from the partner to sequences of other messages to the integrated application/system (Fig. 3).

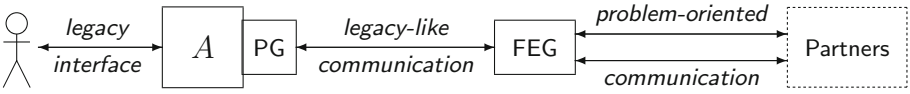


Fig. 3. Service with a front-end gate

If there are partners with different requirements on the interface, it is possible to construct for each partner group specific front-end gate (Fig. 4).

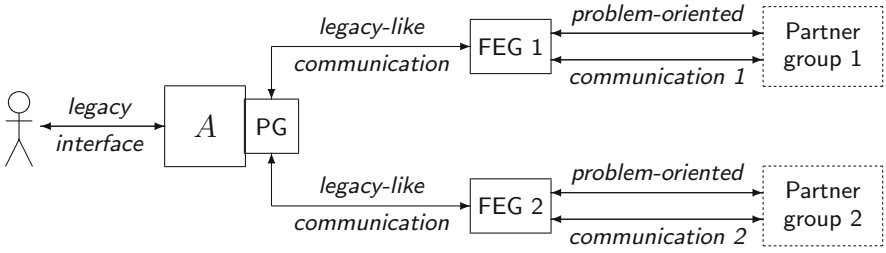


Fig. 4. Service with multiple front-end gates

Such construction rises the chance that implementational changes in any of the partners will not affect A or that implementational changes of A will not affect its partners. It increases stability of the system and reduces maintenance costs.

Let us note that the problem-oriented communication could be based on simple commands, event notifications or even forms containing complex applications as well as entire documents. The message-based communication is more common in control systems and at lower levels of information systems, whereas the document-based communication suits better higher levels of information system (e.g. for communication between cooperating institutions).

4.3 Birth of a Service

The service is expected to transparently serve multiple partners that do not need not know (and usually do not know) the requests of the other service partners, respective the other partners themselves. As the partners access outer interface of service front-end gate(s), such behaviour is expected at last at this border.

Where the border should be, depends also on the application from which the service is created: If it is already a multi-user information system, then it is easy to reach all advantages of service at the primary gate interface. If it is a single-user (and single-thread) application, it depends on number of attached front-end gates: When there is just one, primary gate can be simple – just converts existing user interface to network-reachable interface. Such interface must be at least able to be locked for a single front-end gate at a moment. The serialization of the requests can be done either by extending such simple primary gate by request queue or the request queue(s) can be in the front-end gate(s). In all cases at the outer interface of the front-end gates the expected service behaviour should be provided. In the case when PG has features of a service, front-end gates are full-featured services themselves.

4.4 Access Control

One of the most popular cyber attacks are denial of service (DoS) ones. It is, the service is blocked by enormous load that it is not able to fulfil. The correct users than have no chance that their requests could be solved.

We decided to separate the channels to individual partners and put a control on them. There could be services playing the role of access point for given partner. It authenticates the partner. When something goes wrong, the access point can be simply made inactive what closes access to the system. When the situation restores, the access point can be restored too.

As the partner has only access to its access point, it cannot access other access points easily (they can be e.g. on proprietary connections). It is, when e.g. erroneously some of the partners produce more load than can be processed, the connection can be quickly closed and the other partners can therefore work without any interruption.

It could work also as a protection against DoS attacks (or, more precisely, reduction of their impact) led through connections over Internet. If necessary, access point could be used also for further restriction of the interface provided by front-end gate: it could limit applicable values to given range (i.e. given partner could not ask for arbitrary number of products, but only within agreed range).

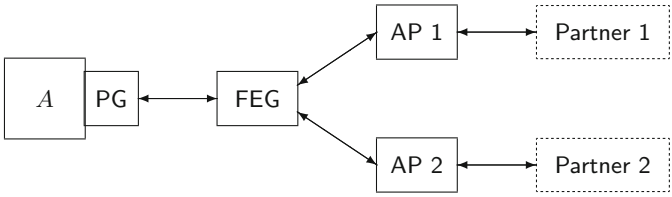


Fig. 5. Service with front-end gate and access points

It is reasonable especially if the limits are not set by supporter company limits but by agreements or legal settings (Fig. 5).

Although it is technically feasible to connect both sides of the access point to the same communication mean (like LAN, Internet, or some middleware products), for security reasons it is not good to do so. Separation of various levels of a system by the use of various separated communication means gives the system a higher overall system resistance to attacks and other network security issues.

4.5 Building Hierarchy/System Composition

Single service is usually not enough – we need to create more complex systems. One possible way is to compose multiple services into a more complex one – into a *composite service*.

Such group should be from outside visible as a sole service: For this purpose there should be created a special service *head of composite service* concentrating communication of any service in the group (incoming and outgoing) with any partner being outside the group (Fig. 6). The composite service can be built from simple service as well as from other composite services or from a mixture of simple and composite services (Fig. 7).

Any request for the composite service must be sent to its head. It decides which service should handle the request and passes the request to the selected service. The responses are usually passed back along the same path. Real composite service could have more complex structure where front-end gates and probably also access points take place – compare Fig. 8.

Sometimes the language of requests from outside do not match the language used by the services in the composite service C. In such case the service of a front-end gate that will provide the request and response transformation could be used. The composite service could be composed from simple services or other composite services. It is possible to build entire hierarchy of composite services forming a complex system.

4.6 Smart Access (Portals)

Larger organizations have dedicated entry points where it is possible to request their service. Incoming requests are evaluated and assigned to some internal service (department) that will process it.

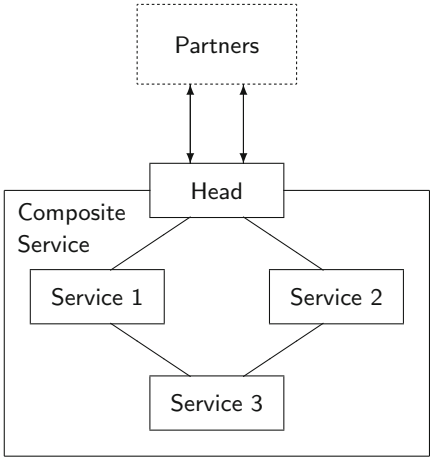


Fig. 6. Simplified composite service

We can build software service behaving like this. It will receive requests from outer world (digital or real), register them, and forward them (probably after some transformation) to the proper target (service that is or should be able to handle it). From the system point of view portal behaves similarly as head of composite service: both services concentrate communication to themselves. The difference is that portals designed to handle human-computer interaction, whereas heads are designed to serve primarily for fully computerized communication.

Many systems have multiple portals: one for inner service (intranet), one for outer service (extranet), and others for special service like supervision of business processes by their owners.

4.7 Service Type Summary

Integrated service/application with its primary gate, front-end gates and access points form a *logical service*. Logical service can have specific interfaces for different partner groups provided by the front-end gates, and managed access regulated by access points.

All the presented services that must be developed (like front-end gates, access points, heads or portals) determine architecture of the system (especially its structure). They play the role of *architectural services*. Architectural services could be equipped with supervising interface allowing responsible people to influence their behaviour and make, if necessary, the decision on their own. Such extension could make sense e.g. when documents are communicated and the service handles its assignment.

In some cases it makes sense to let the request handle manually. Then it could be supported by simple shell allowing receiving requests and sending

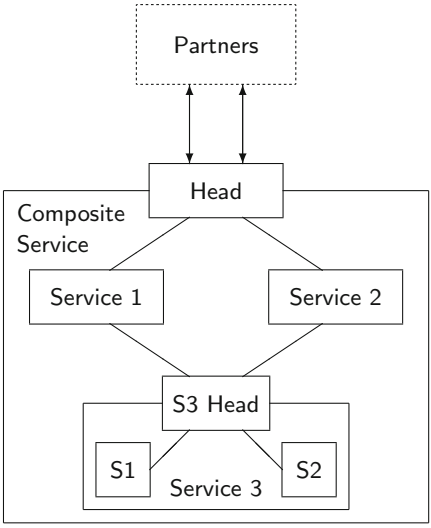


Fig. 7. Simplified hierarchical composite service

replies – *screen prototype*. It could also check correctness of the outgoing messages/documents. It is possible to arrange such tools for handling situation that were not expected or were considered to be too rare to be implemented.

5 Communication Means

Generally there can be used any communication means able to transfer messages or documents between the cooperating services. In practice, there are some aspects that should be taken into account (there are more of them but let us remember just sample of them):

1. Security,
2. Understandability,
3. Usability.

The security aspect is primarily addressed by services called access point. Their proper functionality is conditioned by the fact whether their both ends are connected to the same or different communication mean. If they are connected on the both ends to the same mean, they can be easily bypassed by direct connection on the given mean. There can be established virtual subnetworks or circuits what could improve the situation. When high security is required, the separated communication means appear to be a necessity.

The communication language and format depends on priority setting between understandability and usability (ease of use – here form the programmers' point of view): When communication should be easily understandable to domain experts and if it should be easy to generate by the users, it appears that the

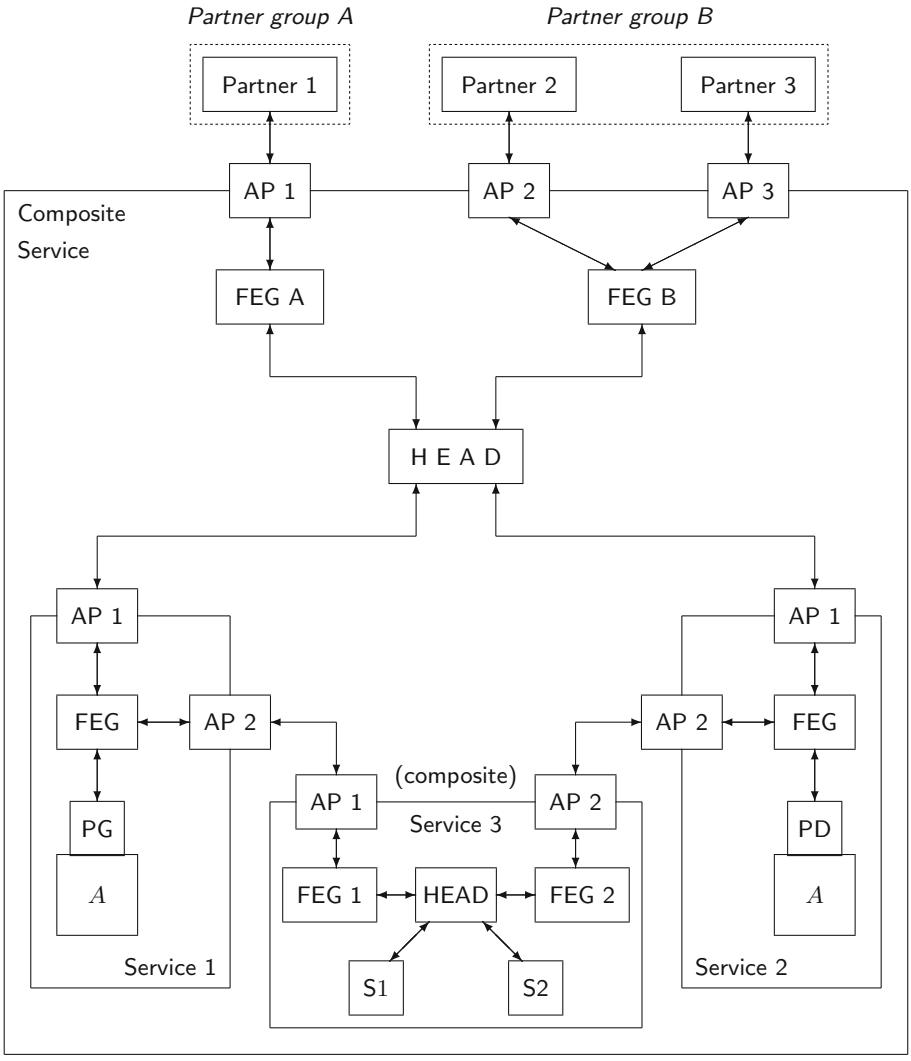


Fig. 8. More detailed view on the same composite service

communication language should be close to programming language – plain text with predefined syntax. For example LL(1) languages can be easy to read by human readers as well as by the computer ones (LL parsers and transducers are very efficient and easy to develop).

Last years communication in many systems is based on semi-structured languages like XML or JSON. They are text-based, there are many tools able to generate and analyse them (for programmers it means minimal work) but their readability for non-programmers are limited. Moreover, data exchange could result in longer messages that require for processing by domain experts additional

tools or longer time than messages in inter-human-communication-inspired LL languages.

So the final decision is on the system architect and their preferences on usability of domain experts (and then their potential use e.g. for error identification and emergency use) or on saving some work of the programmers in the coding phase.

6 Remarks

It all could work only if the organization (or network of organizations) the system should support works properly. Otherwise it can make the situation worse – as the software designed in this way usually works as an amplifier: if the real-world system works fine, the software can improve it; if it works poor it can also strengthen it.

Proposed solution has many technical as well as user advantages. Philosophy, on which it is built, allows the systems to work properly, to be available in time, to be easy to maintain and improve. It allows overall quality of the systems to be improved.

Accidentally there are situations known already from monolithic systems that computerization can be applied also to improperly working processes. Automation then works as an amplifier of the improper behaviour of the system. The chance that it be this way is relatively small. The problem could be also caused by improperly set goal(s) or by non-realistic expectations.

7 Further Research and Conclusions

We need to provide tools simplifying serialization and deserialization of exchanged data to the form close to the one (or exactly the one) used in real-world cooperation. Relation between various software quality aspects seems to be a very promising research topic.

Solution described here has been partially tested in practice. The described solution simplifies development, use, and improvements of so developed systems as well as their management.

Turns described here proven themselves in practice and could be significantly extended using services that could be used as extended middleware. It could be used for building e-government. It enables useful application of information technology at global level. It also allows relatively safe use of open systems.

Systems having described architecture could be developed and deployed by smaller IT companies. It concerns also quite big systems. It has many other technical advantages, e.g. broader spectrum of testing and building prototypes.

References

1. Bison (1998). http://www.gnu.org/software/bison/bison.html
2. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation and Compiling: Parsing, vol. I. Prentice-Hall, Englewood Cliffs (1972)
3. Atzori, L., Iera, A., Morabito, G.: Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm. Ad Hoc Netw. **56**, 122–140 (2017). https://doi.org/10.1016/j.adhoc.2016.12.004. http://www.sciencedirect.com/science/article/pii/S1570870516303316
4. Clark, P.: Big bang rarely works. https://projectone.com/big-bang-rarely-works/
5. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River (2005)
6. Grune, D., van Reeuwijk, K., Bal, H.E., Jacobs, C.J.H., Langendoen, K.: Modern Compiler Design, 2nd edn. Springer, New York (2012). https://doi.org/10.1007/978-1-4614-4699-6
7. International Organization for Standardization: ISO/IEC 9126–1:2001 software engineering - product quality - part 1: quality model (2001). https://www.iso.org/standard/22749.html
8. International Organization for Standardization: ISO/IEC 9126–2:2003 software engineering - product quality - part 2: external metrics (2003)
9. International Organization for Standardization: ISO/IEC 9126–3:2003 software engineering - product quality - part 3: internal metrics (2003). https://www.iso.org/standard/22891.html
10. International Organization for Standardization, International Electrotechnical Commission: ISO/IEC 25010:2011 systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - system and software quality models (2011). https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en
11. Internet Engineering Task Force (IETF): The Javascript object notation (JSON) data interchange format, December 2017. https://tools.ietf.org/html/rfc8259
12. Král, J., Pitner, T., Žemlička, M.: Document-oriented middleware: the way to high-quality software. In: Gervasi, O., et al. (eds.) ICCSA 2017. LNCS, vol. 10408, pp. 607–619. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62404-4_45
13. Král, J., Žemlička, M.: Component types in software confederations. In: Hamza, M.H. (ed.) Applied Informatics, pp. 125–130. ACTA Press, Anaheim (2002)
14. Miller, G.A.: The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychol. Rev. **63**, 81–97 (1956). https://doi.org/10.1037/h0043158
15. OASIS: Reference architecture foundation for service oriented architecture version 1.0, committee specification 01, December 2012. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/
16. Open Group: Open Group standard SOA reference architecture, November 2011. https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?publicationid=12490
17. Parnas, D.L.: Designing software for ease of extension and contraction. IEEE Trans. Softw. Eng. **5**(2), 128–138 (1979). https://doi.org/10.1109/TSE.1979.234169
18. Parr, T.J., Quong, R.W.: ANTLR: a predicated-LL(k) parser generator. Softw. Pract. Exp. **25**(7), 789–810 (1995). https://doi.org/10.1002/spe.4380250705
19. Shiroishi, Y., Uchiyama, K., Suzuki, N.: Society 5.0: for human security and well-being. Computer **51**(7), 91–95 (2018). https://doi.org/10.1109/MC.2018.3011041

20. Skobelev, P.O., Borovik, S.Y.: On the way from industry 4.0 to industry 5.0: from digital manufacturing to digital society. Industry 4.0 **II**, 307–311 (2017). https://stumejournals.com/journals/i4/2017/6/307/pdf
21. W3 Consortium: Extensible Markup Language (XML) 1.0, 5th edn, November 2008. https://www.w3.org/TR/xml/
22. Žemlička, M.: Kind constructor (2002). http://www.ms.mff.cuni.cz/~zemlicka/KindCons/
23. Žemlička, M.: Principles of kind parsing. Ph.D. thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, July 2006
24. Žemlička, M., Král, J.: Run-time extensible (semi-)top-down parser. In: Matousek, V., Mautner, P., Ocelíková, J., Sojka, P. (eds.) TSD 1999. LNCS (LNAI), vol. 1692, pp. 121–126. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48239-3_22
25. Žemlička, M., Král, J.: Legacy systems as kernel of web services. Technical report 2004/1, Faculty of Mathematics and Physics, Department of Software Engineering, Charles University, Prague, Czech Republic, January 2004
26. Žemlička, M., Král, J.: Software architecture and software quality. In: Gervasi, O., et al. (eds.) ICCSA 2016. LNCS, vol. 9790, pp. 139–155. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42092-9_12