IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com
ITP4940

Some Requirement Specification Issues for Service-Oriented Architectures

Jaroslav Král
Charles University, Faculty of Mathematics and Physics, Malostranské nám. 25, 118 00 Prague, CZ, jaroslav.dral@mff.cuni.cz

Michal •emlièka
Charles University, Faculty of Mathematics and Physics, Malostranské nám. 25, 118 00 Prague, CZ, michal/zemlicka@mff.cuni.cz

ABSTRACT
Service orientation is now becoming a widely used powerful software engineering paradigm. The requirement specifications of the systems having any service-oriented architecture and therefore a peer-to-peer structure depend crucially on the properties of the interfaces of the peers. The interfaces should be user-oriented and coarse grained. Another important feature of service oriented systems is the dynamics of the network of services. Extreme cases are the systems of e-commerce *and of* e-government*. It is crucial for feasibility of some requirements and techniques. User-oriented interfaces are good for the development of systems having advantageous software engineering properties.*

INTRODUCTION
Current software systems development methodology is changing from the development of logical monoliths, possibly distributed, to the systems having the character of networks of relatively autonomous components (services) interconnected via a middleware as black boxes (their interface only is known) and behaving like peers in a virtual peer-to-peer network. Such systems are told to be *service-oriented software systems* (SOSS). We say that SOSS have *service-oriented architecture* (SOA). The concept of SOSS is not always properly understood.

This paper is based on the experience with the development and use several SOSS systems (five flexible manufacturing systems, several automated warehouses, etc.) and with the development and use of customizable warehouse control software. All the projects were finished successfully and were retired after 10-15 years. The reason was that the systems were not needed any more, it was not due software problems. At least two of the systems implemented in seventies were still in use in 2002 without any maintenance for many years. Similar projects known to the authors not using the SOA-like philosophy failed.

It is well known that the crucial point of software development and successful use is the requirement specification (RS). The quality of RS depends substantially on user involvement [SG94]. The functions of SOSS must be specified via cooperation of the functions provided by the services. The functions are specified via the interface(s) of the services. If the users are obliged to specify the functions of the whole SOSS, the user must be able to understand the interfaces and to use them properly. The interfaces should therefore be user oriented. We shall show that the user-orientation of the interfaces implies crucial software engineering advantages of the resulting system (stability, reusability, modifiability, etc.).

Our concept of SOSS is wider than the concept of SOA discussed in literature (see e.g. the Microsoft point of view in [Wat03]) as SOSS need not use Internet or the web services. The cases of SOSS mentioned above indicate that SOA can be applied not only for large projects, not only in *e*-commerce and not only with web service.

WHEN SOSS SHOULD BE DEVELOPED
The globalization of organizations led to requirement of integration of legacy systems and third party products. Large on-line systems cannot be switched off for a longer time. If such a system is to be integrated as a component of a new system, it must be integrated 'almost unchanged'. The only change is an addition of gates (ports) providing interface functions allowing other components to communicate with the given one. Examples are:

1. *e*-government (peers are cooperating information systems of offices, e.g. information system of customs office),
2. large enterprises (cooperating IS of autonomous divisions),
3. process control (cooperating drivers of I/O devices),
4. military systems (cooperating IS of military bodies, services, weapon technologies, soldiers),
5. systems supporting *e*-commerce.

We call the systems similar to the *e*-commerce systems *software alliances*. The communication via common database is preferable if data analysis should be done. Such a solution is not easy in the case of alliances and is not without problems otherwise. In *e*-commerce the communication between communicating parties starts with the search for an appropriate partner. Such a partner must be accessible and must understand the messages generated by the initiator of the communication. *E*-commerce must therefore use standards like SOAP and Internet.

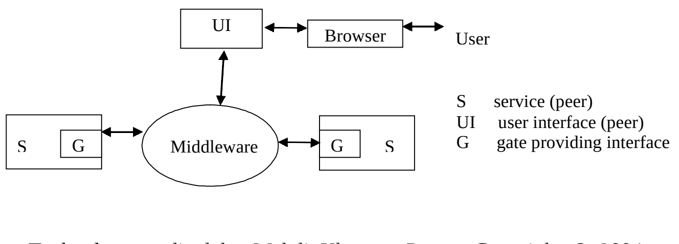
The points 3 and 4 show that 'service' does not always mean 'information providing service' or 'web service' – it can be any arbitrary properly encapsulated activity (including activities of human beings) – and the middleware need not be Internet. It is even possible that some parts of the middleware are based on TCP/IP protocol, and others are implemented via tools of the operating system or via a common database. This turn can be applied in the cases 1-4. We call such systems *software confederations* [K•03a].

In software confederations a proprietary solution could be preferable. In confederations, e. g. due to the effectiveness reasons, some parts of the middleware must use data oriented communication and other means not based on Internet (e.g. interprocess communication supported by an operating system).

Service-oriented software systems (SOSS) have many software engineering advantages [K•03a]. Software architectures in general and SOA in particular significantly influence the feasibility of some requirements (i.e. decentralization, outsourcing, etc.) [K•03a].

Requirements specifications of SOSS should include – according to the experience of the authors –explicit statement that SOSS is to be developed. It should be specified what service types will be integrated into SOSS (data service, activity, supporting middleware, services powered by people, legacy system or third party product, etc.). As the service interfaces should be user-oriented the specification of peers

Figure 1: The structure of an SOSS



should be formulated via message formats obeying a user-oriented language that defines the interface of the components. Last but not least, the principles, tools, and services of the infrastructure (middleware) should be specified or enhance (see the concept of FEG mentioned below). It includes the properties of the service infrastructure provided by the gates connecting services to the middleware.

The prevailing (but not exclusive) form of communication in SOSS is command/operation. It is usually not necessary and even possible to build central data store. There often are also technical reasons why to avoid centralized means. The structure of SOSS is in Fig. 1.

USER-ORIENTED INTERFACES

Until now we have treated the service from the software developers point of view: as a black box used as a node of a peer-to-peer network (i.e. permanently active process) with some interface. It is known that main reason for software project failures (compare e.g. [SG94]) is flaws in requirements specification. In SOSS the component interfaces must be understandable by the users, i.e. it should be user knowledge domain oriented and declarative rather than procedural. It must be user-oriented. The requirements on SOSS must then be defined via the service interfaces with collaboration with users. So the interfaces must be understandable to their users. It implies that the service operation should coincide with well-defined business operations and the interface is based on commands reflecting the structure and semantics of user knowledge domain. As the users are used to use few semantically rich commands they are unable to accept/react on too long sequences of simple programming-oriented commands. The user-oriented communication tends therefore to be coarse-grained. Coarse-grained interface is good for prototyping of peers via an operator console.

The peers are often properly encapsulated already existing information systems. The use of user-oriented interface has then further advantages. As the actions and other activities of users do not usually vary quickly (compare bookkeeping), there is good chance that the corresponding service interface will be fixed for a quite long period of time. This is a crucial software engineering advantage as stable interface simplifies the integration and the modifications of the system and increases its stability.

Business communication cannot be fully automated, as the businessmen only are responsible for business agreements. Some details of the agreements like prices, terms, and other parameters must be often agreed by human beings. It is yet another reason for user orientation of the interfaces.

If the messages are user oriented the communication traffic between communicating parties is low and can be therefore easily logged and analyzed. The semantics of the messages is clear and usually there are not too many messages.as the messages are semantically rich and we need few messages to perform an activity. Such messages will not be influenced by frequent changes of XML standard-based message formats. It simplifies debugging, prototyping, maintenance, and modification [Krá98].

The drawback is that user-oriented messages will have a format that is too specific to be quickly standardized - e.g. in the form of SOAP-based protocols. This problem is not too important in the case of software confederations.

We have seen the in the case of alliances the use of standards is unavoidable. The existing standards like SOAP are cumbersome and rather procedural and are therefore better understandable to programmers if to anybody than to users. It is then difficult to test whether some SOAP protocol implements user requirements properly..

The experience with systems having main features of SOA indicates that there is yet another bonus of service-oriented architectures. SOSS can be very reliable [KD79, KD89]. COBOL systems have the property that applications can be developed autonomously like services in SOA systems. The problem Y2K has shown that in many enterprises the systems were used for years without any support.

SOAP WSDL, and UDDI are based on object-oriented philosophy, on remote procedure calls. These standards are general in the sense that like programming languages they can define a very wide class of interfaces. On the other hand, the messages used in SOAP framework tend to be fine-grained programming-oriented. They tend not to be user-oriented.

The high-level declarative user-oriented communication specification must be translated into SOAP messages in the cases when SOAP-based communication must be used. This is a well-known problem of compilation [•K00]. SOSS developers must, however, in many cases write manually programs transforming declarative-type commands into sequences of SOAP-commands. This is source of errors and delays. These disadvantages can be weakened by the application of compiler like tools that simplify the translation the high level messages into SOAP ‘programs’. Unfortunately the tools like XSLT are unstable and very inefficient. Another solution can be found in [•K00].

A service in SOA can be a standard three-tier information system with its own users and user interfaces equipped by an additional interface – by a gate. If the gate is implemented as recommended in e.g. ASP.NET, then there is a danger that the communication will be too dependent on the implementation philosophy and/or implementation details of the component. It is undesirable as implementation varies too quickly.

It can be easily deduced that in software confederations the following requirements can be easily fulfilled:

- Integration of legacy systems and third party products via adding a proper gates, Fig. 1;
- The use of existing interfaces of integrated legacy systems;
- Simple dialogs between services (just a few steps);
- Support of different level of service automation (even of manual ones) and of the is implemented as a component working as a peer in a peer-to-peer network;
- Stability of the interfaces due to the fact that the complex problems that people have to solve do not change as fast as their solutions and related standards ;-
- Internal architecture and implementation details of the compo nents can be hidden;
- Software engineering advantages as simpler development, easier and cheaper maintenance, incremental development, openness, modifiability etc.;
- Changes are local and peers can be developed/modified almost independently (in an autonomous way).

The interfaces can be made even more flexible and less implementation dependent if we implement them as a two-tier structure [K•02] consisting of *Primary gate* G being a part of the peer and one or more *front-end gates (FEG)*. FEG is a service serving as message format transformers.

Problems for developers:

- Declarative interface can hide some potential functions of the service provided by the implementation (e.g. useful methods).
- SOAP does probably not support the user orientation of inter faces well.
- Standards are changing too fast, so their use need not be good in the case on confederations. This can be solved using FEG.
- Interface choice strategy can be different for *e*-commerce and for the interconnection of the services within a company [K•03b].

FURTHER SOSS DEVELOPMENT ISSUES

Requirements specifications for SOSS is in comparison with usual software development lifecycle more structured and it often require the involvement of company top-management in requirements specification. Typical life cycle of a SOSS is incremental. It is, the development is via development, reuse, or purchasing of new services (applications) and their integration into the system. Confederative systems can make easier business process reengineering, the restructuring of the whole enterprise inclusive [K•03a]. Examples are outsourcing (a division is outsourced with its autonomous information systems), decentralization of the enterprise, incremental development with short service development life cycles and via agile programming [BB+01]. Top-management should therefore formulate its long-term requirements or, at least, its expectation about the company future. The developers should understand what opportunities SOSS offers to enterprise strategy.

Middle management should be together with end-users involved into the formulation of operation requirements. The specification of service functions should be possible via description of user-oriented interface. Development of such an interface cannot be task of a project manager only (as recommended in [SG94]) as this activity requires a significant part of the SOSS development effort. It therefore cannot be the task for one person only. It must a common task of users and developers. It especially needed in the case of legacy systems as their functions are best known to their users. The cooperation between users and developers similar to the one known from extreme programming and from agile forms of development is needed and useful.

We have shown that the requirements specification of SOSS must be more structured than the specifications of logically monolithic possibly distributed systems. SOSS specifications should include the specifications of communication rules, message formats, and infrastructure services. The tools, methods, and best practices for this task are not well established yet.

It is, however, not the main problem. SOA is a new paradigm for many developers. By a paradigm we understand the philosophy, methods, tools, skills, and best practices proven to be good for the solution of certain class of problems. The history of object-oriented paradigm shows that years are necessary to apply a new paradigm widely in practice.

SOA requires a specific intuition, skills, and problem solving patterns. One cannot expect that the people used to develop systems from scratch using e.g. object-oriented techniques will be ready to apply turns not only different from the object-oriented ones but sometimes against them. An example is the principle to build services as quite large components starting from their interfaces that should be user oriented. Object-oriented systems tend to use many simple collaborating methods. It results into long method call sequences of rather simple methods. This problem is only partially reduced if we use components in the sense of UML. The services in SOSS can be (should be) developed using OO attitude.

The acceptance of SOSS is not an easy task for the data-oriented people as well. The services in SOSS do not provide only data manipulation functions like data storing, exchange, analysis, and presentation.

The services can support and often support real-life activities sometimes involving human activities (process control in manufacturing or in weapon control systems, activities of soldiers in terrain, businessmen in business processes, etc). The communication protocols in SOSS cannot be therefore fully automated. The communication should include the supervision of human beings and human beings only are often able to resume/restart a business process when some of its steps failed. A proper distribution of activities between automated processes and human activities increases substantially the system stability [Krá98]. An improper application of data-oriented techniques, like the tendency to develop common databases and/or data concepts for substantially different applications and to overuse database triggers and stored procedures for the communication between some services can be risky.

CONCLUSIONS

Service orientation is becoming a very important paradigm. It is confirmed by the by the success of tools supporting service orientation (see XML and its dialects), and explosively growing interest of big software providers and by the success of already implemented SOSS's. There are many domains (e.g. global systems, *e*-government) where the service orientation is the only feasible solution.

Service orientation can have significant advantages even for quite small systems as the decomposition into services brings substantial resource savings and increase the system quality. Reasons for these savings are in broader opportunity to integrate legacy systems and third party products and the effects of decomposition itself [K•02]. Elements of service-orientation were long time ago present in batch systems written in COBOL and especially in many soft real-time systems (see e.g. [KD79]). SOA has grown into an important software development paradigm only in the last decade an especially in the last years. The promises and limits of service orientation have not been investigated enough yet. SOA is often difficult to accept for the people trained in object-oriented work.

If good service interfaces are used SOSS's can have significant software engineering advantages (incremental development, modifiability, better requirements specification process, agile programming). Practical experience of the authors from realizations of systems starting from mid 70-ies [KD79, K•00] indicates that SOSS can have the advantages mentioned above. The use of service orientation requires some effort to start with. Some existing stereotypes and prejudices must be overcome.

Any newly applied paradigm requires developing of specific best practices, tools, and development patterns. It is to be still developed. Each paradigm has its own domain of its optimal application. Typical area of application of SOSS is in large systems development like *e*-government, customer relationship management and *e*-commerce. Surely there exist some problem domains where SOSS cannot be reasonably used. It can be expected in the case of very small (but not medium size) systems, for some applications of scientific or technical computations, and probably some data intensive systems. SOA has many features common with grid computing. This fact was not studied enough yet. The limits of the applicability of SOA are still to be investigated

The use of SOA is not always positively influenced by marketing strategies of large software vendors. Open problem is the marketing strategy (what to buy, reuse, and build). In our opinion, the orientation toward user-oriented coarse-grained interfaces of application services/components is a crucial condition of the success of any SOSS:.

REFERENCES

[BB+01] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: *Agile programming manifesto* (2001) http://www.agilemanifesto.org/.

[BMS98] Bernus, P., Martins, K., Schmidt, G.: *Handbook on Architectures of Information Systems*. Springer, Berlin (1998)

[Bra02] Bray, I.K.: *An Introduction to Requirements Engineering*. Addison-Wesley, Harlow, United Kingdom (2002).

[FHLW03] Fensel, D., Hendler, J., Lieberman, H., Wahlster, W.: *Spinning the Semantic Web: Bringing the World Wide Web to its Full Potential*. MIT Press, Cambridge, MA (2003).

[Got02] Gottesdiener, E.: Requirements by Collaboration. Addison-Wesley/Pearson Education, Boston, MA (2002).

[Krá98] Král, J.: *Informa*
ní systémy, (Information Systems, in Czech). Science, Veletiny, Czech Republic (1998)

[Krá99] Král, J.: Architecture of open information systems. In •upan
i
, J., Wojtkowski, W., Wojtkowski, W.G., Wrycza, S., eds.: *Evolution and Challenges in Systems Developments*, Kluwer Academic/Plenum Publishers (1999) 131-138, presented on 7th int. Conf. on Information Systems, Bled, Slovenia, Sept. 21-23., 1998.

[KD79] Král, J., Demner, J.: Towards reliable real time software. In: *Proceedings of IFIP Conference Construction of Quality Software*, North Holland (1979), pp. 1-12.

[KDK89] Král, J., Demner, J., Koste
ka, V.: Synchronization primitives for mass service like control software. *Polytechnica* 7 (IV,1) (1979), pp. 11-21, also in *Proceedings of IFIP-IFAC 3rd SOCOCO Conference*, Praha, 1989.

[K•00] Král, J., •emli
ka, M.: Autonomous components. In Hlavá
, V., Jeffery, K.G., Wiedermann, J., eds.: *SOFSEM'2000: Theory and Practice of Informatics*. Number 1963 in LNCS, Milovy, Springer V. (2000), pp. 375-383.

[K•02] Král, J., •emli
ka, M.: Necessity, Challenges, and Promises of Peer-to-Peer Architecture of Information Systems. In G. Harindranath, et all, eds.: *New Perspectives on Information Systems Development: Theory, Meth-*

ods and Practice. Kluwer Academic/Plenum Publishers, New York, 2002.

[K•03a] Král, J., •emli

ka, M.: Software confederations - an architecture for global systems and global management. In Kamel, S., ed.: *Managing Globally with Information Technology*, Hershey, PA, USA, Idea Group Publishing (2003). pp. 57-81.

[K•03b] Král, J., •emli

ka, M.: Software confederations and alliances. In: *CAiSE'03 Forum: Information Systems for a Connected Society*, Maribor, Slovenia, University of Maribor Press (2003).

[Lan01] Lanzerini, M.: *Data integration is harder than you though* (2001), CoopIS 2001, www.science.unitn.it/coopis, choice videos/slides.

[Pet77] Peterson, J.L.: Petri nets. *ACM Computing Surveys* **9** (1977), pp. 223-251.

[SG94] Standish Group: *The chaos report*. (1994), http://www.pm2go.com/smapleresearch/chaos1994_1.php.

[Udd] UDDI Initiative: *Universal definition, discovery, and integration, version 3*. An industrial initiative, http://uddi.org/pubs/uddi_v3.htm.

[W3b] W3 Consortium: *Web service definition language*. A proposal of W3 Consortium. http://www.w3.org/TR/wsdl.

[W3c] W3 Consortium: *Simple object access protocol*. A proposal of W3 Consortium. http://www.w3.org/TR/SOAP.

[Wat03] Watling, N.: *Web services in enterprise computing* (2003) Talk at SI'2003 in Prague.

[•K00] •emli

ka, M., Král, J.: Semitop-down parsing in information systems. In: *Proceedings of SCI/ISAS 2000 Conference*, Orlando, Florida (2000).