# TOWARDS DESIGN RATIONALES OF SOFTWARE CONFEDERATIONS

Jaroslav Král
*Charles University*
*Malostranské nám. 25, 118 00 Praha 1, Czech Republic*

Michal Žemlička
*Charles University*
*Malostranské nám. 25, 118 00 Praha 1, Czech Republic*

Keywords: Service orientation, software paradigm, user involvement, autonomous component, user-oriented component interface, software confederations.

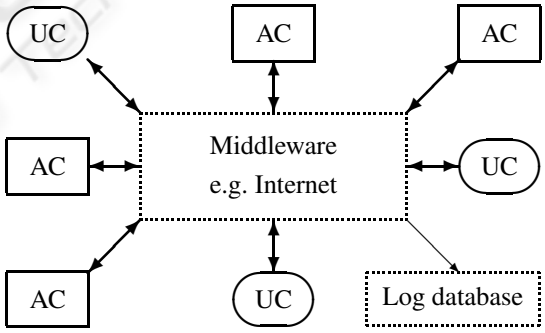
Abstract: The paper discuss reasons why service-oriented architecture is a new software paradigm and the consequences of this fact for the design of enterprise information systems. It is shown that such systems called confederations need not use web services in the sense of W3C. It is, however, more or less a necessity in *e*-commerce. Confederations (service-oriented systems with known set of services) are typical for manufacturing systems. As business processes supported by enterprise systems must be supervised by businessmen, the same must hold for communication inside service-oriented systems. It implies that the interfaces of the services must be user-oriented (user-friendly). It can be easier achieved in confederations than in *e*-commerce systems. User oriented interface has positive consequences for the software engineering properties of the confederation. Confederations should sometimes include parts based on different implementation philosophies (e.g. data orientation). Pros and cons of it are discussed. Open issues of service orientation are presented.

## 1 INTRODUCTION

Enterprise information systems (EIS) tend to have the service-oriented architecture (SOA), i.e. a peer-to-peer (virtual) network of almost independent (autonomous) software units providing some services. EIS differ from the systems supporting *e*-commerce as EIS have

- known and almost fixed collection of services,
- user interface to whole system (often called portal), and
- known set of multistep business processes involving the services.

Information systems (and software systems in general) having such properties are called *(software) confederations* whereas the SOA systems supporting *e*-commerce are called *alliances* (see (Král and Žemlička, 2003a) for details). Note, that in *e*-commerce the set of business partners is not known and the partners must to be looked for. The emphasis of SOA on peer-to-peer principles is not commonly accepted (see e.g. (Tůma, 2003)). Peer-to-peer is a crucial property of SOA.




AC ... autonomous component
UC ... user interface component
(e.g. XSLT component or portal)

Figure 1: Software confederation

The structure of a confederation is given in Fig. 1. It can be meaningful not to limit middleware to be Internet-based. The middleware can even be based on a combination of different communication technologies and/or communication infrastructures. It follows that the services (peers) in confederations need not be web services in the sense of W3C (W3 Consortium,



Král J. and Žemlička M. (2004).
TOWARDS DESIGN RATIONALES OF SOFTWARE CONFEDERATIONS.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 105-112
DOI: 10.5220/0002639301050112

2002; Jablonski and Petrov, 2003). We use SOA in this broader sense.

Software confederations appear quite often e.g. in $e$-government, health care systems, and public organizations. The user involvement in the activities of software confederation is deeper and wider than in classical logically monolithic software systems[1]. Confederations open new variants of user involvement into human interaction with the software systems.

A proper solution of human involvement in the development and the use of confederations usually results into better functions of the confederations and into enhancement of software engineering properties of the system (see section 3).

The application of service orientation is nowadays rather a paradigm than simply a new technique only. By a paradigm we understand according to the Webster Dictionary 'a generally accepted perspective of a particular discipline at a given time'. By a software paradigm we understand a consistent collection of methods, tools, examples of good practices, and development/activity patterns governed by a specific philosophy. New paradigms always require a new way of thinking and new methods and tools. Any new paradigm requires therefore a lot of effort and time to be governed and properly used. Old habits and intuitions must be changed or replaced by new ones. There should be changes in education and training of software experts (Král and Žemlička, 2004a; Král and Žemlička, 2004c).

One can object that service orientation is a quite old technique. It is true for e.g. soft real-time systems like flexible manufacturing systems (see e.g. (Král and Demner, 1979; Král et al., 1987)) but service orientation was not a paradigm in the above sense – it was no generally accepted philosophy at that time. Let us show that service orientation is becoming a leading software engineering paradigm now.

We exploit the experience with the development of the systems having some properties of SOA. The systems were several successful projects of Flexible Manufacturing System (later integrated into CIM systems), several automated warehouse projects (Kopeček, 2003; Skula, 2001) and analysis of the systems for municipal authorities. We often speak about services as about components to point out that a service is provided by a specific software component beeing often a complete application.

## 2 NEW PARADIGM

The concept of service-oriented architecture in the form of confederation is known for decades. The first application of SOA was in soft real-time system like the flexible manufacturing systems dated back in the seventies (compare e.g. (Král and Demner, 1979)). Some features of SOA were even present in COBOL systems. One can therefore argue that SOA is not a new paradigm. It is not so due to the following arguments:

1. SOA was not ten years ago in a leading edge of software development. The classical object-oriented paradigm was the main topic. Information systems typically had no service-oriented architecture.
2. The SOA-based philosophy differs from the object-oriented one like the data-oriented philosophy is different from the object-oriented one. First SOA systems used a very simple not too powerful middleware. SOA were therefore not able to be used in large distributed information systems. New middleware opens new ways for software systems how to be developed and/or used. These challenges are not investigated enough yet.
3. The conferences, publications and standards on SOA-related topics/problems has appeared recently only.
4. The concept of SOA was not generally accepted and strictly speaking there is now no complete consensus what is SOA about.
5. The practice indicates that it is not easy for the object-oriented people to accept and properly use the service-oriented philosophy.

Let us discuss the last point in more details as it also confirms the conjecture that the service-oriented paradigm is a new paradigm different e.g. from the object-oriented one. It is indicated e.g. by the following facts (compare (Brown et al., 1998)):

- The antipattern "Stovepipe Enterprise/Islands of Automation" ((Brown et al., 1998), pp. 147–157) is in service-orientation rather a pattern if a proper gate connecting the island to a middleware is available.
- The basic attitude in SOA is the decomposition of the system into large components providing complex functions. It is a turn near to the antipattern "Functional Decomposition". The modeling of such systems can use a modification of data flow diagrams, a tool that disappeared from object-oriented methodology during the last ten years (compare (Rumbaugh et al., 1991) and from the UML standard (Object Management Group, 2001)).
- Encapsulation and autonomous development of applications (e.g. legacy systems, third party products) is the main design pattern in SOA. It is, however, near to the stovepipe system antipattern

[1] Even if sometimes physically distributed.

((Brown et al., 1998), pp. 159–190, although the recommendations from p. 162 can be applied).

Service-orientation simplifies refactoring of or even excludes the antipatterns: Lava Flow (the technology changes can be hidden inside the services behind their interfaces), Vendor Lock-In (possible but can be easily overcome), Reinvent the Wheel (legacy systems and third-party products can easily integrated), Corncob (unpleasant hacker can develop his own service), etc.

# 3 HUMAN INVOLVEMENT IN SERVICE-ORIENTED SYSTEMS

The lack of user involvement in requirements specifications is the main reason of software project failures. The novelty of service orientation is that the involvement of users in the system activities is deeper than in classical systems. The activities of human beings are not limited to UC from Fig. 1 only. The communication (dialog) in $e$-commerce is usually stateless. The dialog starts with looking for a service (autonomous component) that provides the contact and other information on potential business partners (e.g. supplier). This is often done using UDDI and WSDL. It implies that the communication between partners must be supported by a widely accepted middleware (Internet) and must use generally accepted message formats based on appropriate standards (XML).

The dialog is usually in a SOAP-based formats. The dialog is used to perform a business process (e.g. business agreement conclusion) under the (possible) personal supervision of businessmen of the partners. The business documents produced in this way are to be often formally confirmed by businessmen (signed traditionally or electronically).

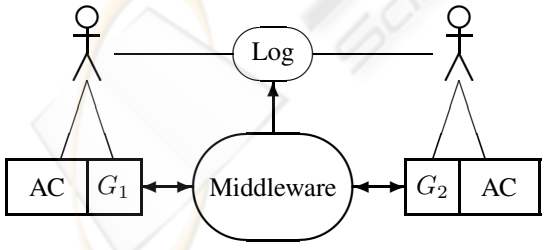



Figure 2: Semi-automated contract dialog scheme: Users should be able to supervise communication and to analyse Log datastore

Business processes are internal matters of the communicating parties. The coordination of the business processes of several partners can be therefore a quite complicated task. Optimal requirements specification on the supervising activity of individual steps of business processes is a corner stone of requirement specification of the alliance as a whole.

As the businessmen must able to understand the communication between the autonomous components, they must be able to understand the formats of the messages produced and/or accepted by the gates $G_1$ and $G_2$ in Fig. 2. This is easy only if the message formats are based on the languages of the user knowledge domains. We say that the interfaces are user-oriented. It can be more easily achieved in confederations where the partners know each other and can agree message formats according their needs. As their knowledge domains are usually based on a long term activities and they are therefore in some sense "tuned", the user-oriented interfaces of $G_1$ and $G_2$ (i.e. the languages defined by message formats) tends to be stable (not so varying as IT technologies). It has important software engineering advantages described below.

In confederations users must be able to define and to supervise business process being networks of steps performable by particular services. The communication traffic in service-oriented systems can and should be logged. The confederations admit, however, a wider set of implementation principles than alliances.

The data defining business processes are best to store in user interface components (portals). The business steps are then supervised via the portal. An extreme solution is that there are no business processes (workflows) defining data and the business processes are completely controlled by users.

Business data analysis (via e.g. OLAP) must be based on data-oriented view. In this case the users (usually managers) must have a transparent access to data (tiers) of the components.

The notions and/or languages specifying the actions of components (e.g. bookkeeping) are often remarkably stable. The same can hold for the languages of the interfaces of corresponding services. It is quite possible that the interface of a component does not vary in spite of the fact that the component implementation varies. The stability of interfaces is important for the communication partners of the given component and for the stability of the whole confederations. It has further substantial software engineering advantages (modifiability, outsourcing opportunities, reduction of number of messages, integration of products of various vendors, openness, etc.).

It is not feasible to limit the philosophy and the software architecture to one choice. It is especially important for the confederations. So the crucial point of the requirements specification for confederations is what philosophy or combination of philosophies will be used. Examples of the philosophies are classical data flow batch systems (see (Your-

don, 1988)), data driven systems (see (Donnay Software Designs, 1999)), object-oriented systems (see (Rumbaugh et al., 1991; Jacobson and et all, 1995)). This issue is not addressed in existing CASE tools and only a little addressed by research.

# 4 DATA-FLOW BATCH SYSTEMS (DFBS)

DFBS were typical for the COBOL era more than thirty years ago when due to the limits of hardware (and to some degree to then state-of-art at that time) the possibilities of on-line (interactive) elaboration as well as of the use of modern database system were limited, if any. Such systems were usually written in COBOL. Note that COBOL compilers are still well supported.

DFBS has many important advantages. The case Y2K has shown that DFBS are extremely stable. In many enterprises they had been in use without almost any maintenance for years – sometimes even for decades. The enterprises had even no COBOL programmers about year 2000 although they had been using systems written in COBOL and DFBS's intensively.

DFBS is typically a collection of autonomous programs providing some services usually written in COBOL. The tasks of the programs were the transformation of massive data from several input files into (massive) data in several output files. The programs can be developed almost independently. The batch character of the systems simplifies the development and reduces maintenance effort. There are still many situations when DFBS can and should be used. Examples are massive data correctness control, data replication, off-line data filtering and transformations, and (statistic) data analysis. Note that the philosophy of DFBS can be easily adapted to the case when the data are in data stores (e.g. copies of databases), not necessarily in files. It is, however, assumed that the data elaboration can be off-line (i.e. performable in the batch mode). Batch systems can be necessary when data analysis is very time consuming.

The first DFBS's were based on the analysis of the data (originally paper documents) flows already existing in the organization. Such analysis is known as function decomposition. Function decomposition reflects the basic features of workflows in enterprises. It is an advantage that some other specification philosophies do not have. On the other hand the use of DFBS does not properly support the horizontal communication in organizations. It has further drawbacks especially in an object-oriented environment. It can make the business process (re)construction more difficult.

From technical point of view it can be difficult to synchronize DFBS with the every-day activities of on-line (interactive) systems. So DFBS are not often used but it does not mean that they are useless. If a DFBS can be used, we should specify the requirements as transformations of input data into output data. It is not equal to the way we specify the data and data operations in data-driven system and object-oriented system described below. DFBS use data-flow diagrams as a main system visualization tool and structured analysis and design as a system development tool (Yourdon, 1988). If a DFBS can be used, it can save a lot of effort and it can simplify the maintenance and increase the reusability of the system. DFBS has the architecture of a network of autonomous services – it is a property that has recently returned in a modified form in software confederations.

DFBS are still usually written in COBOL. The classical COBOL thinking is not obsolete here. The thinking oriented towards a network of autonomous services partly applicable during the implementation of DFBS substantially simplifies the task, provided that the network based architecture is possible. It is the case of DFBS being formed by a virtual network of autonomous programs (a predecessor of services). The autonomous programs can be autonomously developed using different methodologies and/or programming languages if necessary. The programs can be reused or they can be third party products. It is a great advantage having its counterpart in software confederations. It need not therefore be a good idea of some CIO's to avoid 'COBOL thinking' (compare (Sneed, 2002)) and to require every DFBS to be reengineered to be an object-oriented system.

Batch systems are usually very stable, well maintenable and have good software engineering properties. Such systems can be applied in confederations in the cases of massive data reconstructions, data replications and even in the time consuming cases of data analysis. In these cases the batch-oriented attitude is more or less a practical necessity. We know banks performing due to safety reasons financial transactions in two steps. In on-line mode the transaction data are stored in a temporal database. The transactions are completed in batch mode by DFBS.

The use of batch system can impose some limitations on the system performance as some operations cannot be done on-line. The importance of such limitations is often overestimated and advantages of the application of batch system underestimated. The consequence is that the batch philosophy is wrongly considered to be obsolete and useless. Due to this prejudice some opportunities are missed as the analysis does not include the question whether some applications should not be implemented via a batch subsystem.

DFBS attitude can be an optimal solution in mas-

sive data filtering, presentation (report generation) and other task performable in batch mode.

The modeling of batch systems is possible via data-flow diagrams known from structured analysis and design (Yourdon, 1988) or by the tools of functional design.

# 5 DATA DRIVEN SYSTEM (DDS)

Data driven systems appeared after the success of modern database systems. The aim at that time was usually to computerize operative level of an enterprise (warehouses, bookkeeping, business data). Data types were known and various operations could be defined over the same data. Moreover properly designed data structures can be used for many other potential operations – i.e. the data enable many further operations not specified yet. It was and still is supported by the power of the SQL language.

The substantial properties of the architecture of the developed system was the data model expressed by ER-diagrams often accompanied by data-flow model (Donnay Software Designs, 1999). The condition for the application of such a strategy was a proper quality of hardware (servers, client workplaces, data networks).

Data driven systems are still a good solution in the situations when the kernel of the problem is in the data area, i.e.:

- The main problem is to define data and data structures and to collect data in an electronic form.
- The operations can be easily derived from data and data concepts, there are many possible operations over the same data (i.e. a value can be used by many different operations in different ways).
- The properties of the system depend heavily on the optimality of the operations on data.
- There is a need to work with and analyze massive data (statistics, OLAP, etc.).

The requirements specification starts from the data specifications and/or data model and from basic operations. As data enables a broad set of operations the system is open in this sense.

Data oriented philosophy is appropriate in the case when a system is built from scratch starting from data tier.

The existence of data orientation in modern systems is confirmed by the existence of resource definition framework (RDF (W3 Consortium, 1999)) and (meta)data oriented research. Data orientation is typical for management information systems. Such systems assume a flexible open system of operations over massive data.

The fact that the collection of operations is open can be felt sometimes disadvantageous (optimality issues, some important operations need not be detected). The practice indicates that DDS are usually not felt as a system of collaborating autonomous services. Note, however, that common database with triggers can be quite easily adapted to serve as middleware. The message transfer can be implemented as a storing some data including an activation of a trigger causing proper activation of the addressee of the message. The integration of software pieces is in this case via a common data tier. The used tools are SQL, ER-diagrams, RDF, data-flow-diagrams. A strong point of DDS is their openness and support for data analysis. Weak points are:

- the problems with definition of common data structure/database,
- issues connected with data replication and unification (common format/structure),
- large systems are difficult to reconstruct,
- inflexibility of “middleware” services (data routing) compared with the middleware implementation and message passing via e.g. Internet.

Elements of DDS must be often applied in the parts performing on-line (interactive) data analysis (e.g. OLAP/ROLAP[2]) and generally the management tier of information systems. Elements of data orientation can appear on quite low level. In (Král and Žemlička, 2003b; Král and Žemlička, 2004b) a practical implementation of the interface between Scheduler and an automated machine tool workshop is described. The interface was based on a common database and its presentation tool having features of data analysis. The implementation facilitated the success of several projects of several flexible manufacturing systems.

# 6 IMPLEMENTATION OF SOFTWARE CONFEDERATIONS

Technically a legacy system $AC$ is integrated into a confederation via adding a gate $G$ to $AC$ (Fig. 3). $G$ connects $AC$ to a middleware. $AC$ is also redesigned to be a permanent process (if not already one). The software confederation has then the structure from Fig. 3.

$AC$ can be implemented using object-oriented philosophy. Sometimes it can happen that a system can (seemingly) be designed as a confederation or as a

[2]OLAP = on-line analytical processing, ROLAP = relational OLAP.

monolithic system in which all autonomous components should be completely rewritten and "dissolved" in the system.

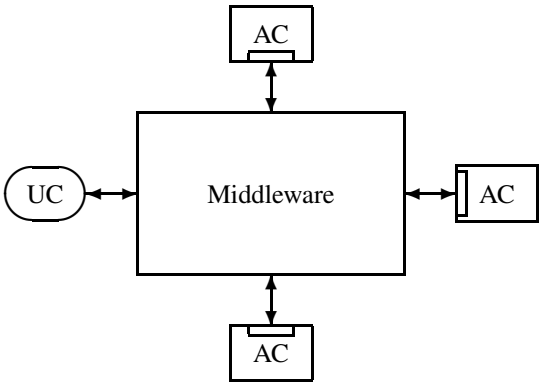



Figure 3: Software confederation with gates

Although the choice between the two alternatives seems to be a quite technical one, it should be included into requirements specification. The requirements specification should be adapted to the fact whether the monolithic or confederated philosophy is to be used as it influences substantially user properties of the system and the set of feasible solutions. If a particular autonomous component $AC$ "belongs" to some department/division (being e.g. its local information system), the department is satisfied with $AC$, and $AC$ can provide all required information/services to other collaborating autonomous components, then $AC$ can and often should be integrated. Note that it reduces development effort as substantial parts of code are reused. The department people feel that they still (mentally) 'own' $AC$. It is well known that of such ownership feeling is advantageous and often necessary.

Confederative architecture can make feasible requirement types raised by CEO like the transformation of the enterprise organization structure into a decentralized form, selective outsourcing of some parts of the information system, selling out some divisions/departments, integration of newly purchased firms/divisions, support for supply chain management, and forming various business coordination groups (Král and Žemlička, 2002). Autonomous components can be developed as a "black boxes" using any of the developing strategies mentioned above.

# 7 BUSINESS PROCESSES IN CONFEDERATIONS; USERS AS SERVICES

The business processes in confederations can be quite complex, they can be a complex networks of activities (variants of workflow). There can be various levels of automation of business processes and their particular steps. It is highly desirable that the processes can be (in principle) supervised by users. Users can:

- define or modify the definition of processes,
- can start processes, supervise/influence/perform their steps, on-line redefine and reschedule process steps,
- change the run of the running (active) processes (change the sequence of the steps, etc.),
- replace some services intended to be provided by software by 'manual activities'.

The last requirement is especially important during the system development as it can serve as the prototyping tool and even substitution of not yet developed (implemented) services. It is a very useful tool for run-time of the system as it can happen that some services will be always provided by human beings or that users will have to simulate or provide services not available due a failure at the given moment. The condition is that user interface components (UC, portals) are able to accept the messages for the missing services, present it to the users and to produce answers defined by users. It is often a quite simple task to implement it (Král et al., 1987) – provided that the interfaces of gates are user-oriented in the above sense.

More complicated business processes should use (should be controlled) by some process defining data. There is a problem where to store business process' states and data. The most promising seems the strategy to include the data into the data of the associated user 'owning' (responsible for) the process. It is also possible to generate a new service $S$ associated with UC. The purpose of $S$ is then to serve as an agent controlling business processes owned by some user.

# 8 SOFTWARE ENGINEERING ISSUES

The user-oriented interfaces of components enhance the following software engineering properties of the service-oriented systems:

1. User involvement. The governance of business processes by users is easier than in classical systems.
2. Modifiability

- implementation changes in services can be hidden, it simplifies maintenability as well;
- some autonomous components can be purchased, insourced or outsourced.

3. Incremental development as well as iterative development.
4. Prototyping and temporal replacement of the services that are currently not available.
5. Short milestones (autonomous services can be not too large software units).
6. Solution/refactoring of many antipatterns (Brown et al., 1998), see above.
7. New turns, e.g. testing the response time of real-time control systems (Král, 1998).
8. Reduction of development effort due:
   - integration of legacy systems,
   - purchasing autonomous component from vendors,
   - decomposition the system into small units that due to dependency between the system size and the development effort reduces the effort (Effort $\doteq c \cdot \text{Size}^{1+a}$, $a > 0$), see COCOMO II,
   - shorter feedback between developers and users.

The are open issues:

1. It is not clear what services should be good to be provided as a central (infrastructure) ones. Central services can be a bottleneck (access traffic, too strong influence of big vendors, data correctness checking, service modification, etc.).
2. Confederative architecture can make customers less dependent on large vendors. The vendors are not happy with it and try to resist it.
3. As a new paradigm the confederations need the developers to precisely understand the user knowledge and needs. This opposes the strong computer-oriented feeling and interests of software experts. To avoid it they should understand and accept the thinking and philosophy of experimental sciences (e.g. physics, to some degree economy) and mathematical statistics. It is quite difficult to achieve it.
4. Data intensive functions (for management and DDS in general) can benefit from service orientation but there is no good methodology available for it now.
5. In some parts of confederations some seemingly obsolete techniques (COBOL) can be used. The traditional integration (into object-oriented or database applications/systems) of such services need not be a simple matter.
6. There are problems with security and effectiveness. Note, however, that the user-oriented interfaces imply reduction of the number of messages and therefore can enhance system performance and can limit threats due to misuse of RPC.

# 9 MAIN POINTS OF DESIGN OF SERVICE-ORIENTED SYSTEMS

As there is no common agreement what service orientation and service-oriented architecture are about (Barry and Associates, 2003), it is necessary to decide first whether the architecture of the system will be a virtual peer-to-peer network. We believe that the choice of peer-to-peer is the best one and we shall discuss this case. The components (peers of the network) should be quite large applications behaving like services in human society. The services should be specified via their interfaces and integrated therefore as black boxes.

The services (their interfaces) should mirror the real-world services provided by the organizational units of enterprises or by people. Their interfaces should be (and due to the above conditions can be) user-oriented, i.e. understandable to system users. The system then behaves like the human society being also a network of autonomous services.

Users should be deeply involved in the specification and design of the interfaces.

The development process of SOA system depends on the variants of decomposition and integration. The implementation of interfaces can be different in confederations (known collection of services) and alliances ($e$-commerce).

There can be problems with the acceptance of the philosophy in the above sense, as service orientation is a new paradigm for many people especially for the strongly object-oriented beings. The acceptance and governing of a new paradigm need many years.

It is preferable for the development, debugging and use of service-oriented system to design services/programs so that they can be substituted by discussion with human bodies via user interface. It is good to support the human involvement as much possible (i.e. automate as little as possible). The development process should be incremental using Pareto 80-20 law as much as possible. It is against the strategy computerize as much as possible promoted by software vendors and accepted by some users.

## 10 CONCLUSIONS

The service-oriented architecture is a very powerful and in many areas a quite new paradigm. There are, however, several issues to be solved yet:

1. Education of people to be able to use service orientation properly. Service orientation is not any simple and straightforward modification of the object-oriented techniques and object-oriented philosophy.
2. Design patterns as well as new tools of modeling are not available yet.
3. The SOA influences the specification of requirements more substantially than other architectures do.
4. The collaboration of users (*including their top management*) with developers as well as users involvement should be tighter than before.
5. The developers should have a deeper knowledge of the user problem and knowledge domain.

Important issues are the prejudices of software developers like insisting on full computerization, belief that users are too stupid to take part in requirements specifications, etc.

Service orientation is, however, today already an attitude promising to develop software of the quality known from the other branches of industry.

*Supported by the Czech Science Foundation, grant No. 201/02/1456.*

## REFERENCES

Barry and Associates (2003). http://www.service-architecture.com.

Brown, W. J., Malveau, R. C., Hays W. "Skip" McCormick, I., and Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York.

Donnay Software Designs (1999). Mature, portable, data-driven systems. http://www.dclip.com/datadr.htm.

Jablonski, S. and Petrov, I. (2003). Web services, workflow and metadata management as the integration means in the electronic collaboration era. Tutorial at the ICEIS'03 conference.

Jacobson, I. and et all (1995). *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley.

Kopeček, P. (2003). Private communication.

Král, J. (1998). *Informační Systémy, (Information Systems, in Czech)*. Science, Veletiny, Czech Republic.

Král, J. and Demner, J. (1979). Towards reliable real time software. In *Proceedings of IFIP Conference Construction of Quality Software*, pages 1–12, North Holland.

Král, J., Černý, J., and Dvořák, P. (1987). Technology of FMS control software development. In Menga, G. and Kempe, V., editors, *Proceedings of the Workshop on Information in Manufacturing Automation*, Dresden.

Král, J. and Žemlička, M. (2002). Autonomous components. In Hamza, M. H., editor, *Applied Informatics*, pages 125–130, Anaheim. ACTA Press.

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

Král, J. and Žemlička, M. (2003b). Software confederations and manufacturing. In Camp, O., Filipe, J., Hammoudi, S., and Piattini, M., editors, *ICEIS 2003: Proceedings of the Fifth International Conference on Enterprise Information Systems*, volume 3, pages 650–653, Setúbal. EST Setúbal.

Král, J. and Žemlička, M. (2004a). Architecture, specification, and design of service-oriented systems. In reviewing process.

Král, J. and Žemlička, M. (2004b). Service orientation and the quality indicators for software services. Accepted for EMCSR'04 in Vienna.

Král, J. and Žemlička, M. (2004c). Systemic of human involvement in information systems. Technical Report 2, Charles Univerity, Faculty of Mathematics and Physics, Department of Software Engineering, Prague, Czech Republic.

Object Management Group (2001). Unified modeling language. http://www.omg.org/technology/documents/formal/uml.htm.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey 07632.

Skula, J. (2001). Private communication.

Sneed, H. (2002). Position statement at panel discussion at CSMR 2002, Budapest, Hungary.

Tůma, P. (2003). Modern software architectures: Novel solutions or old hats? In Popelínský, L., editor, *DATAKON 2003*, pages 151–162, Brno, Czech Republic. Masaryk University.

W3 Consortium (1999). Resource description framework. A proposal of W3C Consortium. http://www.w3.org/RDF/.

W3 Consortium (2002). Web services activity. http://www.w3.org/2002/ws/ – as visited on 31st October 2003.

Yourdon, E. (1988). *Modern Structured Analysis*. Prentice-Hall, 2nd edition.