ZESZYTY NAUKOWE INSTYTUTU ADMINISTRACJI AJD W CZĘSTOCHOWIE
Gubernaculum et Administratio 1(11)/2015, s. 77–88

http://dx.doi.org/10.16926/gea.2015.01.06

Jindrich KALUZA
Akademia im. Jana Długosza w Częstochowie

Public Sector Information Systems Development Methodologies in Current Progress

Summary

Strategic management and information systems are very closely interconnected today. Success in the company information system (IS) implementation is a key factor for the whole strategic management success. The IS development process represents still a lot of individual human effort, although a lot of methodologies, methods, and software tools have been invented namely during the last decade. On the other side, it is not easy to absorb and, then, select the optimal methodology from a variety of choices existing "in the market" for the purpose of particular project. The chapter is focused on summarisation of the current achievements and trends specification that are emerging in this area.

Keywords: information systems, methodologies, public sector, life cycle, SOA, component-based software, agility.

1. Systems development methodologies today

1.1. How to classify existing methodologies

Systems development methodology could be defined according to (Hoffer et al. 2002) as a "standard process followed in an organisation to conduct all the steps necessary to analyse, design, implement, and maintain information systems". Due to a number of factors influencing the process, namely

— various sizes and natures of projects being developed
— various sizes and structures of project teams
— various techniques and methods offered in the market
— various organisational cultures representing the institutions for which the systems are being developed

— various constraints given to the project teams in advance (budgets, time, people),

a number of methodologies currently exist; some are mutually similar, some bring quite new procedures and methods. Methodologies are usually named and/or represented by various abbreviations that could mislead the observer and cause the false expectations. On the other side, some developers still utilise the individual company's methodologies or, even, no ones. According to the research carried out by (Buchalcevova & Leitl 2006) among 21 software companies in the Czech Republic it represented 14% of respondents.

When trying to classify and characterise the existing methodologies, various authors apply different classifications and classifying terms. The purpose of this chapter is not to present an exhausting review of all possible structures. However, two viewpoints are emerging here; they both offer an appropriate tool for the stratification following the principal features that could be observed in any individual case:

a) iterativity
b) agility.

Any practical methodology application leads time to time to some ***iteration*** caused by changing environment, changing requirements specification, and the initial mistakes repair. Despite that, some methodologies do not take it into account and define the pure ***waterfall*** (Hoffer et al. 2002) life cycle (see fig. 1). It means that any next phase of the cycle is started as soon as the previous one is completed.

Quite opposite approach is offered by the ***iterative*** life cycle called by some authors also as the ***prototyping*** methodology – see (Stair & Reynolds 2008). Here (see fig. 2) the developmental procedure is composed of a set of iterations (partial waterfalls), each of them passing all phases of the waterfall. At the end of any iteration we receive a prototype being reviewed afterwards and, then, repeated, improved, and refined to a new prototype etc. Finally, the resulting prototype should represent the new system or, in case of so called ***throwaway prototyping*** – see (Hoffer et al. 2002), it could serve as a model for the ultimate development in end-user programming environment.

The aspect of iterativity fits more appropriately to real situations during the system development process. Practical implementations of the waterfall methodologies led in almost all cases to some iteration reflecting various changes and mistakes repairs. This was rather understood as something not belonging directly to the methodology but a product of practical implementation. The methodology *Rational Unified Process* (RUP) is a good example of iterative approach in the object-oriented environment based on UML – see (RUP 2009).

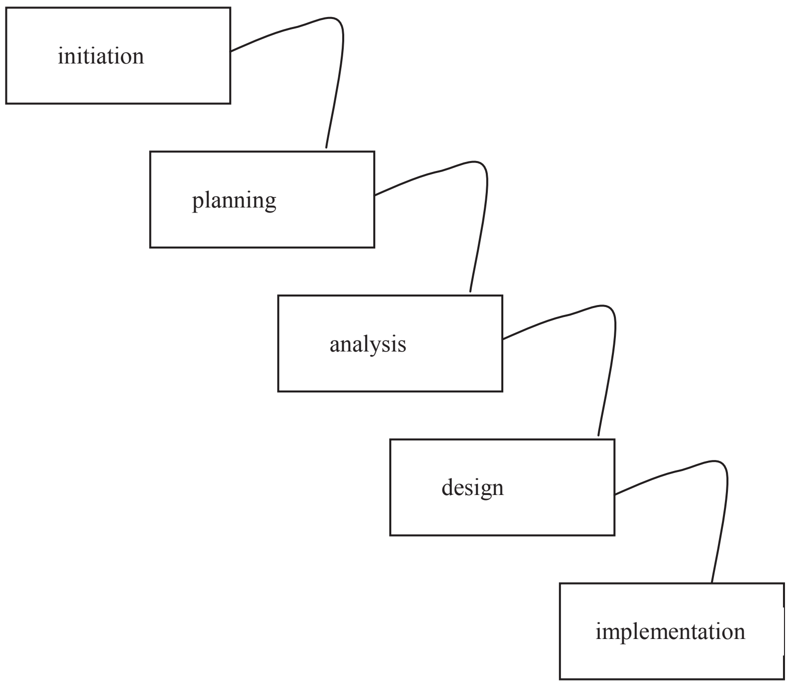

Figure 1. Waterfall life cycle

Agility allows systems to change during the process of development. While ***traditional*** methodologies are concentrated on processes specification, development tools application, planning, documentation completion, the ***agile*** ones prefer communication, co-operation with customers, quick changes. The objective here is a speed of the whole process. Mutually various methodologies belonging to this group differ, however, the principles are same: i.e. to offer the first prototype to users as soon as possible, to work very intensively – designers, programmers and users in one team, personal communication within the team is preferred, quick changes of prototypes are expected.

The typical representative of agile methodologies is the *eXtreme Programming* published by (Beck 2000). It is based on the development life cycle composed of coding, testing, listening, and designing. Small teams of programmers work together with users in short cycles, the requirements specification and development is based on the “planning game” where customers first specify particular requirements as “story cards”, programmers estimate the necessary capacities then, all cards are assessed by necessity (customers) and risks (programmers), selected cards are finally put into the new system release. Programmers transform “story cards” into the “task cards”, accept responsibility for them, balance the workload, write code and test that. The process carries on in further cycles. Documentation is represented mostly by the code.

The concluding question here could be: will the agile methodologies replace the traditional ones in the future? The discussion carries on; some authors present both approaches without any prediction, others predict either the agile methodologies as the main stream of future software development (Mullaney & Davidson 2007), or argue some limitations of that (Turk et al. 2008):
— limited support for distributed development environment
— limited support for subcontracting
— limited support for building reusable artefacts

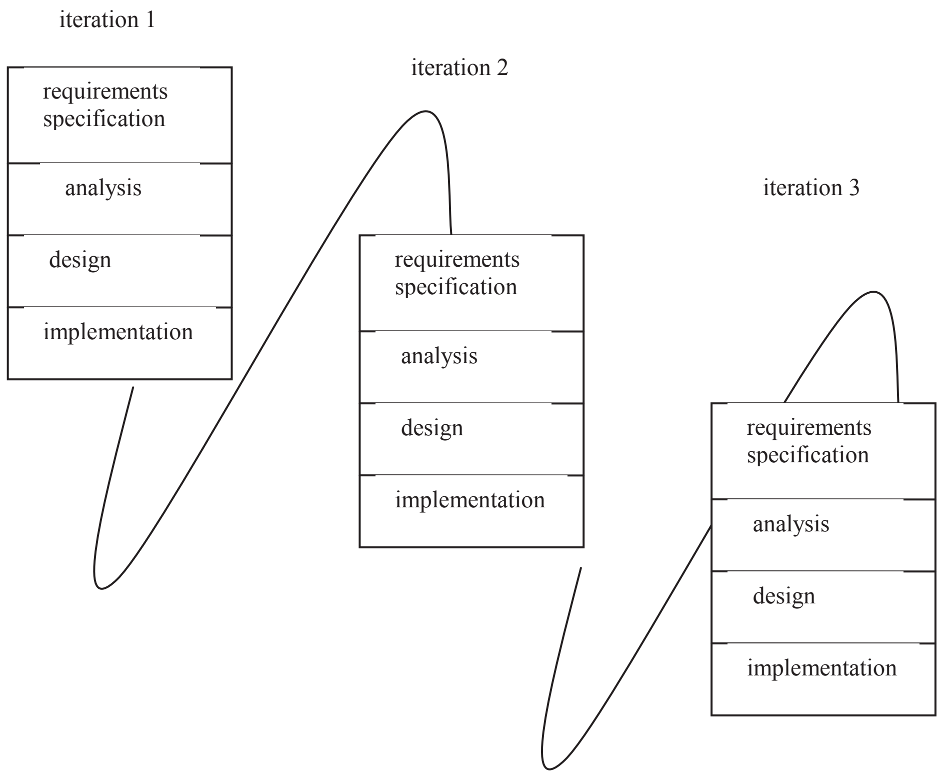

Figure 2. Iterative life cycle

— limited support for development involving large teams
— limited support for developing safety-critical software
— limited support for developing large, complex software

It could be agreed with the last opinion that "practical processes lie somewhere in between the purely agile and purely predictive (i.e. traditional) spectrum extremes". The nature of particular system – its size, complexity, uniqueness, personal support, etc. should influence the selection.

1.2. How to apply existing methodologies

Potential methodologies for the information system development were discussed in previous section. Further step in these considerations is the projection of a set of potential methodologies into the practical problem solving, i.e. into the concrete software development projects. Some authors talk here about the "tailoring software development methodologies". In accordance with (deCesare et al. 2008) we could distinguish four different possibilities:

— individual methodology is tailored for particular project
— complete takeover of existing methodology
— mixing "best-of-breed" parts of various methodologies
— adaptation of one methodological framework to particular projects.

The first possibility represents a very expensive solution bringing the risk of mistakes and misunderstandings caused by not proven methods application, risk of reinventing things generally known and being utilised. This approach probably occurs very rarely in real world situations.

The second approach walks on proven path, however, some problems could happen in two directions: a disposal of enough qualified staff being able to utilise efficiently a full portfolio of tools and methods brought by particular methodology implementation, and consistency of all aspects of a given methodology with the nature of a concrete project. Nevertheless, this approach represent safer route and more robust utilisation than the previous one.

The "mixing" approach enables to put together best fitting set of methods and tools from various methodologies to particular project. This solution brings a benefit of advantages of more methodologies to a concrete application. The disadvantage here is a problem of harmonisation of different frameworks, paradigm transformations, leading to difficulties in mapping of incompatible constructs. Practical implementation of this approach should be, then, very careful with constraints caused by the above mentioned problems.

Following some research referenced in (deCesare et al. 2008), the fourth possibility offers probably the most applicable solution. Particular institution or project team adopt a methodological framework that tailor to specific projects. It brings an advantage of a cosistent framework and flexible adaptation to individual needs. Modern methodologies like current versions of RUP are based on building blocks (methods) enabling the assembly of some selected parts into a specific whole. The positive aspect of this approach is also a know-how keeping for the benefit of future projects.

2. Systems architecture building

2.1. Component-based software

The idea of components forming the whole is not new. It was one of greatest inventions of humans to break down the complex problem that is difficult to understand into smaller parts (possibly repeatedly) easier to be described and computerised. Some thirty years ago the conception of ***modular programming*** has been created. Same principle is a part of ***Yourdon's structured method*** of system analysis and design, i.e. hierarchical decomposition of functions.

Later on, in early 2000s, some authors talk about the ***Component-Based Software Engineering*** – see e.g. (Brown 2000) or (Heineman 2001). Basic idea of that is to build up the software systems by assembling components already developed and prepared for integration. Eventually people are finding out that permanent re-design of information systems following the development of new hardware and system software platforms is enormously costly, time consuming and from the implementation point of view less flexible.

The component is a key element here. Basically, the component is a part of something. More precisely and specifically towards the software, adopting (Szyperski 1998) the software component *is a unit of composition with contractually specified interface and explicit context dependencies.* The most valuable aspect of that is a separation of component's interface from its implementation. The integration of a component into the application is required to be independent on the component development; there should be no need to rebuild the application when updating with a new component.

Such understanding of a component has an impact on the whole *software architecture* related to particular system. Any software system could be viewed in terms of the decomposition into components which are in mutual relationships. While the traditional architecture (i.e. non-component based one) is of monolithic structure at the execution time (although possibly composed of some logical parts or "components"), the component-based system has an architecture recognizable during the system execution, the system still consists of clearly separated components. Traditional approach thus utilises "components" during the developmental process, not in terms of the final product. The components at the execution time are possibly logically visible but they are hardly re-usable without the code modification. It should be emphasised here that components are meant in something like commercial sense, not as rather the "technical" modules dealing with frequently used small parts of code. The software architecture is then concerned with components specification and interactions among components. Some architectural definition languages exist (e.g. ACME) and could be utilised for the component-based systems design.

The nature of the software development process is also changing here. In component-based development the process is oriented to re-using of existing components. Some difficulties could occur with components interaction and with fitting to all features of the stated requirements.

2.2. Service oriented architecture

Service Oriented Architecture (SOA) is a hit of last couple of years. There are a number of definitions of SOA in relevant literature. For example, in (Sun Microsystems 2005) the following definition is offered:

> A service-oriented architecture is an information technology approach or strategy in which applications make use of (perhaps more accurately, rely on) services available in a network such as the World Wide Web. Implementing a service-oriented architecture can involve developing applications that use services, making applications available as services so that other applications can use those services, or both.

A *service* is understood as a specific function, typically a business function (e.g. processing a purchase order). It can provide a single discrete function or a set of related business functions. One way of looking at an SOA is then the approach to connecting applications (exposed as services) so that they can communicate with (and take advantage of) each other. In other words, a service-oriented architecture is a way of sharing functions (typically business functions) in a widespread and flexible way.

What distinguishes an SOA from other architectures is *loose coupling*. Loose coupling means that the client of a service is essentially independent of the service. The way a client (which can be another service) communicates with the service doesn't depend on the implementation of the service. The client communicates with the service according to a specified, well-defined interface, and then leaves it up to the service implementation to perform the necessary processing. If the implementation of the service changes the client communicates with it in the same way as before, provided that the interface remains the same. Loose coupling enables services to be document-oriented (or document-centric). A document-oriented service accepts a document as input, as opposed to something more granular like a numeric value or Java object. The client does not know or care what business function in the service will process the document. It's up to the service to determine what business function (or functions) to apply based on the content of the document.

More detailed specification is in (W3C 2004): *SOA is a form of distributed systems architecture that is typically characterised by the following properties:*

— Logical view: The service is an abstracted, *logical* view of actual programs, databases, business processes, etc., defined in terms of what it *does*, typically carrying out a business-level operation.

— Message orientation: The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The internal structure of an agent, including features such as its implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns so-called legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application that can be "wrapped" in message handling code that allows it to adhere to the formal service definition.

— Description orientation: A service is described by machine-processable meta-data. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.
— Granularity: Services tend to use a small number of operations with relatively large and complex messages.
— Network orientation: Services tend to be oriented toward use over a network, though this is not an absolute requirement.
— Platform neutral: Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is the most obvious format that meets this constraint.

It does not make sense to bring more and more definitions of SOA. The above specified ones bring enough information to realize the basic framework.

One of frequently discussed features here is the level of services ***granularity***. Basically, ***fine-grained*** services cause more complex interactions and higher network overheads. On the other side, using services with extremely ***coarse-grained*** interfaces externalizes complex data structures, creates interdependencies, and potentially creates overlapping functionality. Some critical experience with this problem is published in (Subramanian 2006). He reports that "... we quickly realized that a system based on such fine grained services will have unwanted development, deployment, and performance overhead. What we have learnt is that a service ... is something that the company wants to manage independently". Services must be specified at the correct level of abstraction and granularity. It is not sufficient to proclaim only that services should be coarse-grained and have well-defined interfaces. The relevant design methodology is necessary guiding the whole process of reusable services (being something like building blocks for business-level composite services) design. Further extensive research here should be expected.

2.3. Service component architecture

The ***Service Component Architecture*** (SCA) represents an industry effort by consortium OASIS sponsored by IBM, BEA, SAP, Sun and Primeton (OASIS 2005) to provide a set of specifications which describe a model for building applications and systems using a SOA. SCA extends and complements prior approaches to implementing services, and builds on open standards such as Web services.

SCA encourages a SOA organisation of business application code based on ***components*** that implement business logic, which offer their capabilities through service-oriented interfaces and which consume functions offered by other components through service-oriented interfaces, called service references. SCA divides up the steps in building a service-oriented application into two major parts:

(1) ***implementation*** of components which provide services and consume other services;

(2) ***assembly*** of sets of components to build business applications, through the wiring of service references to services.

SCA emphasizes the decoupling of service implementation and of service assembly from the details of infrastructure capabilities and from the details of the access methods used to invoke services.

The basic artefact is the ***Module***, which is the unit of deployment for SCA and which holds ***Services*** which can be accessed remotely. A module contains one or more ***Components***, which contain the business function provided by the module.

Components offer their function as services, which can either be used by other components within the same module or which can be made available for use outside the module through ***Entry Points***. Components may also depend on services provided by other components – these dependencies are called ***References***.

References can either be linked to services provided by other components in the same module, or references can be linked to services provided outside the module, which can be provided by other modules. References to services provided outside the module, including services provided by other modules, are defined by ***External Services*** in the module. Also contained in the module are the linkages between references and services, represented by ***Wires***.

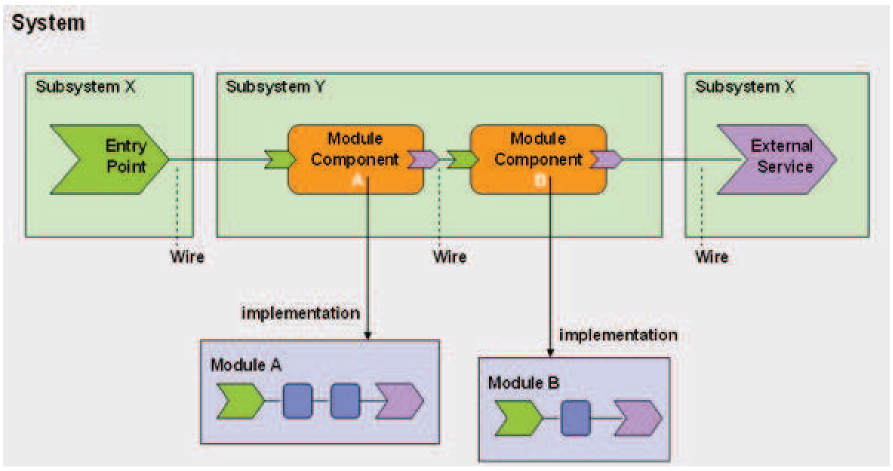

Figure 3. Service Component Architecture

A Component consists of a configured ***Implementation***, where an implementation is the piece of program code implementing business functions. The component configures the implementation with specific values for settable ***Properties*** declared by the implementation. The component can also configure the implementation with wiring of references declared by the implementation to specific target services.

Modules are deployed within an ***SCA System*** that is represented by a set of services providing an area of business functionality that is controlled by a single organisation. To help build and configure the SCA System, ***Subsystems*** are used to group and configure related modules. Subsystems contain module components, which are configured instances of modules. Subsystems, like modules, also have entry points and external services which declare external services and references which exist outside the system. Subsystems can also contain wires which connect together the module components, entry points and external services.

SCA supports service implementations written using any one of many programming languages, both including conventional object-oriented and procedural languages or declarative languages. SCA also promotes the use of service data objects to represent the business data that forms the parameters and return values of services, providing uniform access to business data to complement the uniform access to business services offered by SCA itself.

2.4. Components and services

When comparing the term “component” specified both in component-based approach and in SOA we could summarize it is practically equal. The SOA approach brings a conception considered in more detailed form and in more practically oriented approach. Component–based approach adopts the software component as a primary element then leading to something like services. On the other side SOA was primarily based on services and further methodological development led to the components specification. So we could see the current state of methodology of SOA as the adoption and further development of component-based approach.

2.5. Fusion in SOA

Leading software producers substantially changed their strategies recently. They left the original strategy of purely in-house application software development offering to users as much as possible the broadest spectrum of applications (services) enabling also parametric modifications following their various needs.

Currently in accordance with (Pomazal 2007) they carry out the strategy based on:

a) ***acquisitions*** of (not necessarily) smaller software houses offering in the market some specific products
b) ***building the open software platforms*** enabling users to add some specific external components to existing packages.

Good example of a) is Oracle Corp. uniting under single roof the Siebel's Enterprise CRM, JD Edwards's World and PeopleSoft's Enterprise.

The b) aspect in my opinion witnesses of something like a new trend in SOA development currently (is this a revolution?). Originally the SOA conception was oriented to the design of completely new systems and solutions. The Oracle Fusion brings the opposite strategy: to utilise the SOA architecture as the framework for implementation of various enterprise systems not only from its own production but also externally. So the user should implement the platform and then proceed following the "best-of-breed" approach.

Conclusions

A wide spectrum of IS development methodologies exist today being utilised both in business and in public sector too. They could be compared and analysed from various viewpoints. Two viewpoints seem to be most valuable here; they both offer an appropriate tool for the stratification following the principal features that could be currently observed in any individual case: iterativity and agility. While the aspect of iterativity fits more appropriately to real situations during the system development process, the agility phenomenon brings a new paradigm (compare to traditional methodologies) suitable either for a specific type of applications as a unique methodology, or partially for remaining ones complementing the traditional approach.

Practical applications (projects) stand in front of the decision-making process – which of the disposable methodologies or their parts should be selected. Out of four potential solutions the adaptation of one methodological framework to particular projects accross the organisation seems to be the most valuable solution. It brings an advantage of a cosistent framework and flexible adaptation to individual needs, and, also, the know-how keeping for the benefit of future projects.

In terms of the system architecture, the component-based development building up the software systems by assembling already developed components is more cultivated into the service-oriented architecture conception. Currently, the SOA conception offers a revolutionary idea of building the open software platforms enabling users to add some specific external components to existing packages. Some providers (Oracle Corp.) call this, for the future very promising aspect, as a "fusion".

References

Beck K. (2000), *eXtreme programming eXplained*, Upper Saddle River, NJ: Addison-Wesley.

Brown A. (2000), *Large-Scale Component-Based Development*, Prentice Hall.

Buchalcevová A., Leitl M. (2006), Průzkum používání agilních metodik v ČR. In: Objekty 2006, Praha: ČZU.

Heineman G., Councill W. (2001), *Component-Based Software Engineering, Putting the Pieces Together*, Addison Wesley.

Hoffer J.A., George J.F., Valacich J.S., (2002), *Modern Systems Analysis and Design*, 3rd ed., Pearson Education, Inc., NJ, Prentice-Hall Int., Inc.

Kaluza J. (2002), *Information Systems Development and Implementation – Still Too Many Drawbacks and Problems*, Int. Conf. "Organisation, Informatics, Personnel, Management and the European Community", University of Maribor, Slovenia, Portoroz.

Mullaney J., Davidson M. (2007), *Software Development Trends in 2008: Outsourcing, Agile Development*.SearchSoftwareQuality.com, http://searchsoftwarequality.techtarget.com/news/article/0,289142,sid92_gci1287341,00.html.

OASIS (2005) http://xml.coverpages.org/ni2005-12-07-a.html.

Pomazal A. (2007), *Blíže potřebám zákazníků prostřednictvím SOA*, Int. Conf. World of Information Systems, Zlín 2007.

Rational Unified Process, http://www.ibm.com/developerworks/rational/products/rup/.

Stair R., Reynolds G., (2008), *Fundamentals of Information Systems*, 4th ed., Thomson Course Technology, Boston, Mass.

Subramanian S.N. (2006), *Taking SOA from Paper to Production*, Systems Integration 2006, 14th Int. Conf. Prague.

Sun Microsystems (2005), http://java.sun.com/developer/technicalArticles/WebServices/soa2/SOATerms.html#soaterms.

Szyperski C. (1998), *Component Software – Beyond Object – Oriented Programming*, Addison Wesley.

Turk D., France R., Rumpe B. (2008), *Limitations of Agile Software Process*. Agile Alliance, http://www.agilealliance.org/system/article/file/1096/file.pdf.

W3C (2004), http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

Systemy informatyczne w administracji publicznej

Streszczenie

Zarządzanie strategiczne i systemy informatyczne są dziś bardzo ściśle ze sobą połączone. Sukces w realizacji systemu informatycznego firmy jest kluczowy dla całego sukcesu zarządzania strategicznego. Proces rozwoju systemu informacyjnego reprezentuje wciąż dużo indywidualnego wysiłku ludzkiego, chociaż wiele metodologii, metod i narzędzi programowych zostało wynalezionych w ciągu ostatniej dekady. Z drugiej strony, nie jest łatwo przyjąć, a następnie wybrać optymalną metodę dla celów konkretnego projektu spośród wielu możliwości istniejących „na rynku". Artykuł koncentruje się na streszczeniu stanu obecnego oraz na trendach specyfikacji, które pojawiają się w tej dziedzinie.

Słowa kluczowe: systemy informacyjne, metodologie, sektor publiczny, cykl życia, SOA, oprogramowanie oparte na komponentach.