ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645
Indian Journal of Science and Technology, *Vol 9(47), DOI: 10.17485/ijst/2016/v9i47/109080, December 2016*

Making a Formal Case for the Development of Components of Modern Enterprise Information Systems

Alexander Ryndin* and Sergey Sapegin

Voronezh State Technical University, Voronezh, Russia;
alexandr.ryndin@icloud.com, svsapegin@mail.ru

Abstract

Background: The article covers the specifics of streamlining the design of software components for modern information systems, proposes a multiple-path integration approach intended for obtaining optimal and suboptimal solutions in multidimensional problems. **Methods**: The article proposes an approach to establishing a smooth pipeline for the development of software components within enterprise Information Systems (IS) based on the principles of Service-Oriented Architecture (SOA) that intend to reduce the complexity of the IS components themselves and the way they communicate with each other. To come up with the most practical solutions, the authors have used multiple-path integration methods that can help provide solutions to multidimensional problems within a reasonable time. **Findings**: The proposed formalization of the task to design an IS component serves as a base for studying the characteristics of the life cycles of different IS components, strategies and approaches to their design and development, as well as the general IS structure by means of various strategies of interaction between the existing software components and the ones being designed. **Improvements/Application:** The proposed approach can be used as a tool for building large-scale information systems in order to reduce the time and resources spent on the project. The article suggests a use case for this approach involving the building of IS components for telecommunication companies.

Keywords: Enterprise IS, Multiple-path Integration, Software Development

1. Introduction

Given the today's level of advancements in technology and methodology, software development is a complicated, creative task that is generally nontrivial and often leads to unpredicted results. That is why for decades, there has been a significant interest towards establishing an efficient software development pipeline.

The problems associated with establishing smooth software development pipelines, designing efficient software components and choosing the right project solutions are among the most crucial in the software industry[1]. Despite the numerous attempts to formalize software design and development, in practice the high level of problem ambiguity and the complexity of the solutions at hand results in only about 29% of IT-projects fitting into their deadlines and budgets[2]. Even the stats show that the key reason for derailed projects is their complexity.

Brooks[1] made the earliest, most widely known attempts at understanding the specific experience in the design and development of IS software components. In general, there are several generations of approaches to the design and development of software components, namely the following:

**Author for correspondence*

1. The generation of structural development, which heralded the emergence of CASE tools and the use of the Waterfall model as the fundamental one[3].
2. The generation of Object-Oriented Programming (OOP), accompanied by the adoption of the spiral model, adaptive development processes, UML and OOP patterns[4–6].
3. The generation shaped by the Agile Manifesto, whose underlying concept is about the maximum agility in following the customer's interests and any changes in them, aiming to focus the customer's attention on the product quality[7–9].
4. The SEMAT initiative that bridges the practical gap between various software methodologies by enabling development teams to choose the tools that have the biggest relevance to the project's scope[10].

In the course of shaping these software development approaches, a new paradigm of designing and implementing components has emerged, with its key principles aggregated in the following groups:

1. Maximization of quality in terms of functionality. This includes iterative development practices, close collaboration with the customer (or customer's representative) and aggressive testing (to the extent of adopting Test-Driven Development (TDD)).
2. Minimization of the cost of change, both through the use of flexible object-oriented architectures and the introduction of parameters to system configurations, and through the adherence to coding standards and popular architectural patterns (such as MVC) in order to simplify the reading of source code.
3. Minimization of development time, through a partial code generation, use of frameworks and general-purpose libraries, functional code re-use and the development and application of typical project solutions and patterns.

It is easy to see that generally speaking, these groups of principles contradict one another, so the fundamental task in front of any developer is to find a reasonable compromise in the extent to which the aforementioned principles are to be implemented. Yet the growing complexity of the designed software systems makes this task increasingly harder, and the solutions to this task — increasingly less certain, dependent in large part on the expertise of the people making project-related decisions, which gets reflected on the software's performance in the long run. According to SEMAT visionaries, in case the software industry maintains the existing trends, the mankind should brace for a disaster in the form of global software degradation by 2025 at the latest.

2. Concepts

Within their study of the issues related to building modern software systems, the authors have conducted the sampling analysis of software development projects realized in various industries, involving a range of technologies, over the last 10 years. The sampling criterion required that the developed products are still used today (with due account for modifications which however should not have affected more than half of the original source code). A separate research was applied to a group of projects that were supposed to automate some processes in a certain industry at the launch of the developed software, which later underwent a significant upgrade or completely replaced with a similar product in the subsequent years. The project analysis has revealed the following tendencies clearly identified to a varied degree in almost every project from the studied group:

1. Excessive spending on equipment and platform products attributable to the volatility of requirements arising in the process of the development of mathware and software within the enterprise computing infrastructure;
2. Excessive labor input into the development of software components used for solving short-term tasks;

3. Excessive efforts into integrating software products from different vendors and development teams;
4. Unnecessary abundance of features in the implemented software systems, the lack of deep integration between their components;
5. Misaligned user experience caused by an inconsistent IS growth strategy.

Our analysis shows that the above-listed examples of excess are for the most cases associated with the attempts at improving the efficiency of the software environment, achieving a certain level of performance of its components (functionality, agility, feasibility) without taking into account their underlying systemic relations and the rate at which the system evolves. Therefore, it makes a good sense to consider the process of building software components in terms of the general information systems development strategy, in view of the influence caused by structural aspects of the particular IS.

Let's take a look at building a software component for an information system based on the Service-Oriented Architecture (SOA)[11,12]. The development path for the information system itself can be formulated as a sequence of procedures aimed at the continuous improvement of the existing and the implementation of new IT services. It has to be noted that in real life, there are complex, multilayer relations between the existing and new services within an IS which determine their mutual influence. In its turn, the design, implementation and exploitation of each IT service can be carried out using various techniques making use of different structures and approaches, depending on the nature of the service, the utilized technologies, methods, the scale of the service, user capacity, etc.

The strategy that is to be adopted for the development of the components of an enterprise IS must work towards achieving the maximum economic benefits from using a particular component (service). The authors propose to formalize this task in the form of the following expression:

$$\int_{T_0}^{T_0+T} f(S(t); F(t)) \rightarrow \max, \quad (1)$$

where $S(t)$ is a set of requirements for the IS component, $F(t)$ is the implemented functionality, T is the time of the actual usage of this component as part of the system, T_0 is the starting time of the usage of the component, f is the function assessing the adequacy of the component's functionality $F(t)$ with regards to the actual requirements $S(t)$, $f \in [0,1]$. With this in view, execution strategies can comprise the following basic concepts:

1. Statement of software requirements $S(t)$ in the way that makes them easier to implement. This task can be accomplished through the procedures of business modeling, requirements management and written specifications, which are part of the majority of modern software engineering methodologies. In terms of client-to-vendor relations, the methods for stating the most practical set of requirements can be broken down to the following groups:

   – the use of a range of methods and tools for structuring the stipulated requirements and preventing their uncontrolled "evolution" within the project scope (iterative review of requirements, prototyping, gathering non-formal data on the project's structure, establishing the mechanism for changing the requirements, preparing documents that record the requirements at a certain level).
   – streamlining the implementation strategy (prioritizing requirements and arranging them by the time and effort needed to implement them, by their having user dependencies, etc.);

– preparing counter-offers (exploring the possibilities for implementing certain requirements by means of components and solutions the developers already have in their toolkit, making suggestions as to the reorganization of inessential business processes in order to simplify automation).

2. Maximization of the concerned component's functionality *F(t)* by the time the developed component starts being used. The component's features should be designed with a view to an up-to-date set of requirements *S(t)* and the anticipated changes in them, besides the existing development technologies and developers' skills should also be taken into account. The problem of creating a component with a set of features *F(t)* that closely match the current set of requirements *S(t)* is in fact the problem of setting up an environment for an effective team-based software development. This problem has been widely addressed in the literature over the last few decades. It is worth noting that the majority of strategies proposed for solving this problem are based on empirical data. It is generally considered a difficult task to design and use mathematical models for organizing a smooth software development process, and here are the main reasons:
   – high significance and unpredictability of the human factor;
   – inefficiency of formal approaches to setting up the collaboration within small teams;
   – heavy reliance of the outcome on the emerging development technology and methodology.

3. Maximization of the time frame $[T_0, T_0+T]$ within which the developed component is being used. The useful life of a component is determined by the following factors:

   – carefully elaborated requirements reducing the chance of the *S(t)* set changing in the future and, consequently, extending the time interval *T* during which the component is being used;
   – use of programming methodology that is open to changes (object-oriented approach, agile architecture, etc.);
   – predetermined maintenance and support strategy, as well as a plan for the component revision during its operation (managing user requirements, documenting solutions and technology, taking into account the human factor).

The following macro-system indicators can be used to rate the performance of the system architecture as a whole:

- the degree of diversity of technologies and methods used throughout the system;
- the degree of integration of system components;
- the degree of functional redundancy;
- the degree of inconsistency between the system functions.

1. One of the possible indicators of the diversity of technologies and methods used for implementing the components of an enterprise IS could be the entropy of the decision tree for each component. Systems with low-entropy decisions are highly "monolithic," with a relatively strictly fixed set of technologies available for application. On the one hand, this contributes to a more coherent and consistent structure of the enterprise IS, while on the other hand it increases the risk of the components falling short of user requirements, which can result in a considerable shortening of the useful life of the system. Conversely, high entropy can both indicate a well-designed architecture open to changes and signal the inconsistency of the system components, the lack of unified standards and, consequently, lead to an exponential rise of the cost of development for each new component.

2. To measure the degree to which the system components are integrated, it is possible to use the following indicators:

- average number of relations between the system components:

$$\tilde{N}_{\text{ср}} = \sum_{i=1}^{n} C_i \Big/ n, \quad (2)$$

where, $\tilde{N}_i$ is the number of relations the component *i* has with other components;

- average number of components that take part in the implementation of a particular business process:

$$B_{\text{ср}} = \sum_{j=1}^{m} B_j \Big/ m, \quad (3)$$

where, B_j is the number of the system components that take part in the business process *j*;

In addition to calculating averages, it is sometimes useful to evaluate the distribution dynamics of $\tilde{N}_i$ and B_j in order to identify bottlenecks in the system, as well as set the goals for the refactoring of the IS and the reengineering of the business processes.

3. The degree of functional redundancy is an indication of the percentage of unused features in the system. Depending on the system architecture and the purpose of the assessment, the following criteria can be used for that assessment:

- the proportion of interfaces that are not used for maintaining business processes (in relevant architectures);
- the proportion of procedures and functions that are not used for IS operation (including those that have been "temporarily" switched off at the final stage of functional tuning);
- the proportion of processes implemented in the EIS that are nevertheless keep running without interacting with other IS components.

4. The degree of inconsistency between different business functions of the system indicates the dissimilarities in the way various system business processes are implemented, the inconsistency of data and behavior logic of the system components. Let us assume that V_i is the *i*-th identified fact of inconsistency between the implementations of the same function within two different system processes. Let us define $f(V_i)$ as the function measuring the degree of importance of the inconsistency detected in the *i*-th fact, V_i. In that case, the problem of the integration and reengineering of the IS can be formalized as

$$\sum_{i=1}^{n} f(V_i) \to \min, \quad (4)$$

where, *n* is the total number of detected inconsistencies in the system. Under favorable circumstances, it is recommended to strive for a complete elimination of the detected inconsistencies, yet in some cases the process of integrating various applications into a single enterprise IS does not allow for the total elimination of inconsistencies, which would significantly exceed the budget or simply prove unpractical for cost reasons. At any rate, the decision on each identified inconsistency, as well as the $f(V_i)$ measurement, will rely entirely on the input by the experts who work on the development of the architecture and the integration of different components under one EIS.

The general best practices for the development and effective operation of an EIS are not limited to designing

a static architecture with the required agility and performance, but also concerns the identification of the best path for developing the enterprise IS and making the EIS stay on that path.

The task of developing a software component can generally be expressed in the following words. Suppose there is the set of requirements S_i for the component, with its parts expressed through the following formula:

$$S_i = (1 + A(t)) \cdot S_{i\,tech} + (1 + B(t) + \tilde{N}) \cdot S_{i\,user} + S_{D(t)} \quad (5)$$

where $S_{i,tech}$ represents the technologies used to enable the component (including the technologies used to interact with other system components); $S_{i\,user}$ represents the user requirements for the component; $A(t), B(t)$ are multipliers determining the rate of changes introduced within the component during its useful time; C is the factor of consistency of requirements for the component set out by the different users of the component within the enterprise IS; $S_{D(t)}$ is a set of requirements stipulating the compatibility of various features inside the component (the conditions for the multipliers $A(t), B(t)$). Therefore, the general task of developing a software component can be described as meeting, within a limited time frame, a set of requirements S_x that is as close as possible to a certain ultimate (ideal) set of requirements S_{ideal}. Generally speaking, it is impossible to achieve a full compliance with S_{ideal} during development because of the following reasons[13]:

1. The software component must be completed within a limited period.
2. The requirements for the software component to be developed are often set out by a whole group (or even several groups), and not by a single user, so each group member can put forward requirements which are hard to align with the requirements of other group members.
3. The requirements for the software component can change with time, from both the user side and the side of the software with which it interacts.
4. In many cases, the assessment of the feasibility of all user requirements is done without taking into account the subjective nature of this process, i.e. the skills of the developers working on the software component.

3. Results and Discussion

Based on the suggested concept, the problem of achieving the maximum cumulative effect from the use of a particular software component in a system can be expressed as

$$\int_{T_0}^{T_0+T} f(S(t); F(t)) \rightarrow \max, \quad (6)$$

where, $S(t)$ is the set of requirements, $F(t)$ is the set of implemented features, T is the useful life of the component, T_0 is the moment when the component started being used, f is a function reflecting the correspondence between the component features $F(t)$ and the actual requirements $S(t)$, $f \in [0,1]$. The simplest interpretation of the f function is the following expression

$$f = \frac{D(S(t), F(t))}{|S(t)|}, \quad (7)$$

where, D is the cardinality of the symmetric difference between the sets $S(t)$ and $F(t)$ at the moment of time t; $|S(t)|$ is the cardinality of the set $S(t)$. Experience has shown that

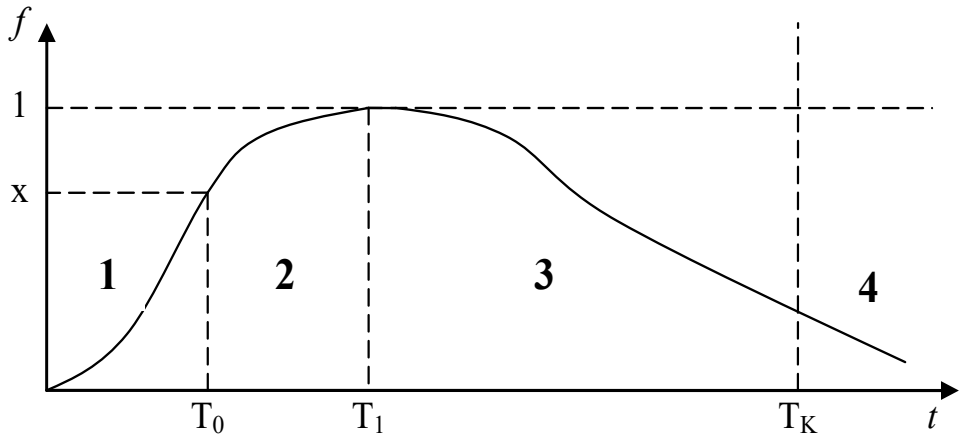

Figure 1. Component life cycle.

the dependence diagram for the distance between the sets *S(t)* and *F(t)* (i.e. the degree of correspondence between the component features and requirements) turns into an S-curve, so long as the development process is vigorous (task-oriented). The reason for this is that at the beginning of the development cycle, resources are dedicated not so much to implementing functions, as to building the architecture of the software tool. Therefore, the variations of the *f* function that are closer to statistical data should be looked for amongst the family of S-functions of varied curvature. Figure 1 shows a sketchy diagram of the *f* function reflecting the most common life cycle of such component.

A component life cycle ends when the component is taken out of service. The main reasons for taking a software component out of service are:

1. Reaching a certain critical threshold of the minimum number of features in use (the component is in fact rendered useless under new circumstances);
2. The component being incompatible with the actual architecture (in case there is a change in the operation system, data model, service interaction environment, standards, etc.);
3. Replacing the component with a new version or delegating all of its features to other components put into operation.

The need to consider systemic relations and the mutual influence of a number of uncertainties in the course of making reasonable decisions leads to the emergence of models, in which looking for a precise solution is not the most efficient approach, since in the process of searching for the optimal solution you might lose the ability to analyze an entire group of dominant decision options. It is, however, possible to expand the potential of the systems approach to the design and development of software components if you agree to measure the metric value of the rationality of your choice of solutions using an information characteristic, namely the entropy of the multiple-path integration that determines the degree of diversity of the numerous paths for integration. In the dynamically evolving and changing structure of rational decisions, it is the entropy that often is the only indicator that serves as a suitable ground for choosing a solution that will ensure the optimal development of complex software packages under the conditions of uncertainty.

Let us present the process of building a software package as a design path consisting of a series of multiple-criteria choice tasks, where the outcome of each choice affects the route of the further system growth. In the course of designing the enterprise IS, the choice is made among the items of vector $\overline{W_g}$, each of which, in its turn, contains a set of components making up the system. The choice items $w_g = \overline{1, W_g}$ will in general define the implementation choice:

$$S_l = (w_1, w_2, ..., w_g, ..., w_G) \in S, \quad (8)$$

and are described by a vector of parameters f_{Wg}.

When switching between the implementations $w_g \in \overline{1, W_g}$, the components of vector f_{Wg} are to be changed discretely.

The decision on choosing the best possible design path (based on their complexity) and the technology stack for its implementation, given the multiple existing options of the IS architecture and of the integration principles, is made based on the aggregate value of all the technical and economic indicators $F_i (i = \overline{1, I})$. The chosen design path is supposed to ensure that the software will be designed with a focus on the planned performance characteristics: reliability, efficiency, functional richness, etc. The software performance characteristics form a subset $I_{á} \in I$ that determines the choice of the variables in the design process.

To outline the principles of the multiple-path integration, let us explore the nature of the task of complex systems design. The rational synthesis task is generally about choosing the best option $s \in S$, where *S* is a general set comprising various items. The choice s is made via a step-by-step crossing off of options that do not satisfy the given requirements, i.e. it is made through limiting the diversity expressing the relations between two subsets, whereby the diversity of one of them is reduced by applying certain limitations. The currently known direct search methods intended to reduce the diversity of the S set can help find a solution within an adequate time frame only for synthesis problems limited by the determinacy of the relation between the parameters of system options and the requirements $F_i \left(i = \overline{1, I}\right)$, as well as by the clearness of the requirements for the system operation environment, which has an analytically defined criterion for evaluating the overall system efficiency.

The most effective instrument for measuring the general degree of diversity of the options bound by probability relationships is the entropy, such as the one for a subset of options $á$ that can be expressed using the following for-

$$H(\alpha) = -\sum_{n=1}^{N} P_n^{\alpha} \lg P_n^{\alpha} \quad (9)$$

and has a range of intrinsic properties:

It is symmetric with respect to the coordinates of vector $p^{á}$, i.e. it does not depend on the relative positions of $p_n^{á}$;

It peaks in vector $p^{á}$ with coordinates $p_n^{á} = 1/N \ \forall n = \overline{1, N}$, i.e. when all options are equal;

It attains its minimum in vector $p^{á}$ with coordinates $p_v^{\alpha} = 1, \ p_n^{\alpha} = 0, \forall n \neq v$, i.e. when there is only one integration path left suitable for implementation.

In addition to that, one must observe the principles of local and multiple-path integration, utmost reliability and adequacy.

4. Conclusion

The present-day practices of software development rarely involve building and using any standards of software components development within a particular project. Under these circumstances, the development standardization is mostly aimed at building a set of typical solutions used by programmers in certain cases. Among the reasons for the common neglect of standardization are such factors as the diversity of problems to be solved, the specific nature of different industries and the human factor. On the other hand, in case of an initially highly uncertain set of user requirements, or when development, testing or deployment needs to be expedited or paralleled, it is possible to achieve a positive effect in terms of both time and performance by building and using local development standards for each project. A key role here is played by the relations between the components and the estimated useful time of each component.

As a use case, let us review the task of building a CAD system for creating data transmission networks that would help design metropolitan multi-service communication networks. The construction of modern high-speed communication networks and the installation of necessary equipment are expensive in terms of both costs and time, so when designing a data transmission network, engineers extensively use numerous methods for simulating and modeling networks, as well as for predicting the increase in their traffic load over time and the operation patterns of various equipment in a certain environment. For this reason, one of the most important requirement for the data transmission network CAD system is the availability of effective integration mechanisms maintaining the exchange of data between other design tools, as well as an open architecture that allows importing into the CAD system of different algorithms of simulating various pieces of equipment, consumers of data services, channels, etc. The overall structure of the CAD system is fairly transparent in this case, but the success in streamlining its development and implementation relies heavily on the procedure of component development and the corresponding design philosophy. The situation was analyzed based on the acquired general list of components to be developed, the evaluation of labor intensity of each component, and the assessment of the volume of implemented user requirements, including the aforementioned uncertainty factor. Based on the results of analysis of the user requirements, a component dependency matrix has been built. Then, using the gathered data and the predetermined set of common development approaches (rapid development, architecture design methods, development intended to create a reusable component), the approximate life cycle profiles was calculated for each component. The most streamlined paths for designing the components and the recommended standards were determined based on the principles of multiple-path integration. Up to two independent paths were considered for each component. The streamlining of the system development plan allowed us to clearly identify the following groups of components:

1. Key components with a low degree of uncertainty. In general, the recommendation for such components was to elaborate them with as much details as possible.
2. Key components with a high degree of uncertainty. In this case, the most optimal strategy has been the concept of "disposable design" that intends to quickly deploy the initial "stub" version to test any dependent components, followed by a detailed development of an advanced version of that component adjusted for the newly identified user requirements.
3. Non-key components. Their implementation standard is directly determined by their degree of uncertainty – the higher this degree, the more details should be factored into the component.

Time-wise, it was easy to spot the signs of risk mitigation strategies: the components with the least certain

requirements tended to migrate towards the beginning of the development process.

Thus, the use of multiple-path integration methods for a comprehensive evaluation of the possible paths for the evolution of enterprise information ecosystems, as well as for choosing the most reasonable software design methodology within the scope of the enterprise IS, is one of the ways to ensure the organization's competitive growth in today's reality.

5. References

1. Brooks FP. The mythical man-month: essays on software engineering. 2nd Edition. Addison-Wesley Professional; 1995.
2. Lynch J. Chaos report. Standish Group [Internet]. 2015. Available from: http://blog.standishgroup.com/post/50.
3. Telnov Yu. Intelligent information systems in economics. Study guide. Moscow: Sinteg; 1998.
4. Yakobson A, Buch G, Rambo G. Unified software development process. Saint Petersburg: Piter; 2002.
5. Kruchten Ph. The rational unified process: An introduction. 3rd Edition. Addison-Wesley Professional; 2003.
6. Booch G, Rumbaugh J, Jacobson I. The unified modeling language user guide. 2nd Edition, Addison-Wesley Professional; 2005.
7. Ryndin A, Khaustovich A, Dolgikh D, Mugalyov A, Sapegin S. Design of enterprise information systems. Voronezh, Kvarta; 2003.
8. Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M. Manifesto for Agile Software Development; 2001.
9. Rubin KS. Essential scrum: A practical guide to the most popular agile process (Addison-Wesley Signature Series (Cohn)), Addison-Wesley Professional; 2012.
10. Bieberstein N, Laird RG, Jones K, Mitra T. Executing SOA: A practical guide for the Service-Oriented Architect. IBM Press; 2008.
11. Jacobson I, Ng P-W, McMahon PE, Spence I, Lidman S. The essence of software engineering: Applying the SEMAT kernel. Addison-Wesley Professional; 2013.
12. Erl T. Service-Oriented Architecture (SOA): Concepts, technology, and design. Prentice Hall; 2005.
13. King W. ITIL for beginners: Simple and easy beginners guide to understanding and starting with ITIL Implementation in your organization; 2016.