On Information System Architecture Supporting Acceptance Testing

Csaba Szabó and Veronika Szabóová

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovak Republic
csaba.szabo@tuke.sk, veronika.szaboova@tuke.sk

Abstract

Our goal is to extend common information system architecture by a systematically built element that will directly support acceptance testing. To be able to support further activities such as maintenance, we decided to develop a permanent system layer or component. This paper presents detailed description of our goals and the abstract model of the proposed architecture. The paper ends with future directions on possible implementation and practical usage areas.

Keywords: *Acceptance Testing, Horizontal Traceability, Information System Architecture, Requirement, and Vertical Traceability.*

1. Introduction

Nowadays, there are lots of program systems used in many different areas of human life. When data are the main subject of these programs, the program system is called an information system (IS). A plenty of methodologies exist for IS development, most of them offer complete analysis of requirements on data structure and processing to provide proven methods in implementation. Weakness of these methodologies is the lower level of flexibility, because one significant change in the requirements implies another full iteration of the methodology beginning with requirements analysis and validation using theoretical proofs [9].

Agile methods [10] are frequently used in software development when rapid delivery and flexibility on changing user requirements is of higher priority than the usage of (semi) formally proven processes.The most popular agile methods include Extreme Programming, SCRUM, Test Driven Development, LEAN development, Crystal methods, Dynamic System Development Method and Feature Driven Development.

In general, the process of software development and testing is divided into several stages. The main difference between these stages from the project organization point of view is the role of customers (respectively customer proxy) and end users.

There is a so-called R-I-E (requirements-implementation-evaluation) tripartite as we presented in [6], that represents the possible roles in software development by their category. The requirement role is the one defining requirements to the software before and during its

development (implementation). The implementation role is the one understanding the requirements (and modifying them in several cases) and implementing the system based on the knowledge derived from these requirements during the process of understanding [11]. The evaluation role is the one verifying and evaluating the final system or its prototypes (as a whole and/or by its modules and components) based on its own knowledge discovered during the process of requirements' understanding. In addition to that, it might use also other not specified domain specific knowledge as well.The roles in the R-I-E tripartite are shown on Fig. (1).

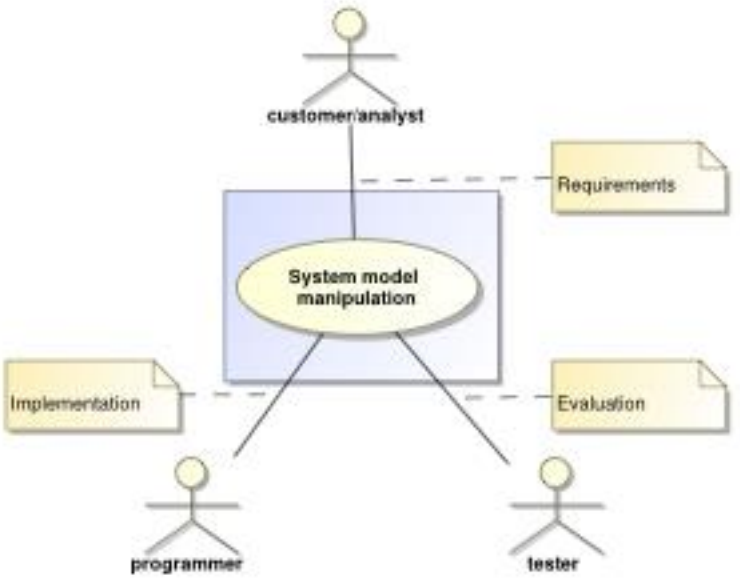

Figure (1): The R-I-E tripartite [6]

In the phase of acceptance testing, end users play an active role, because they are using the product and are reporting defects and other issues. In the most cases, the complete set of requirements in not available to these users, but they are equipped with serious domain specific knowledge [4].

Our goal is to provide an architecture that will support identification of relations between the explicitly defined requirements, their implementation and the hidden ones discovered from the domain experts during acceptance testing.

The organization of the paper is as follows. In Section 2, we describe the problem in detail. Section 3 is devoted to expression of basic types of information system architectures. Section 4 denotes the acceptance testing process. We present our method and related architecture proposal in Section 5, and we present future directions of our related research in Section 6.

2. Problem Description

The requirements are changing. A selection of them is being created during the requirements activity. These serve as the basis for implementation, which, in general, might modify them by creating its own explanations and knowledge about them. The relation between this knowledge and the initial requirements is expressed by vertical traces that join implementation details with specific requirements these implementations are developed for.

In [5], vertical traceability is defined as the tracing of requirements through the layers of development documentation to components or test components, respectively.

In unit testing, horizontal traces are used to maintain connection between the tests and related components under test, and vertical traces to maintain connection between tests and related requirements [12]. Our goal is to develop a similar property for acceptance testing while focusing on automation of the traceability management.

Problem 1: There is no sufficient horizontal traceability in acceptance testing

The first problem is that there is no definition for the component under test in acceptance testing, only for the system under test, i.e. the whole system. This means, that there is no explicit implementation unit that could be given as reference for found defects in the system. The module name and the process description using domain specific language might be insufficient to identify a single source of error.

Problem 2: There is no sufficient management of vertical traceability in acceptance testing

The second problem lies in the language used: domain experts might use in their feedback many formulations of the problem, and they might not have knowledge about existing requirements. In other words, simple text processing (searching) within requirement database will not always find a matching existing requirement, so the requirements phase needs to re-run to identify existing vertical traces or to introduce new requirements into the knowledge base of the project.

3. Information System Architectures

Literature presents many software architectures that could be implemented for information systems [3]. The common feature of them is the usage of a database (or more databases) to store the processed information. Another common property is the so-called business logic representing information processing routines, access roles, and integrity constraints.

We present several architectures for multiuser information systems:

- Multi-layer architecture consists of separated layers for presentation, business logic, and data.
- Modular resp. component-based architecture is characterized by a new behavior achieved by integrating its components that provide separate functionalities.
- Service-oriented architecture (SOA), where independent services are combined to a system to fulfill user requirements.

Business logic, data processing or presentation of information [1] might use artificial intelligence in implementation [2]. Some parts of the system could be generated based on design models or integrated using automated tools [7,8].

From the point of view of our proposal, these presented architectures are specific in their way. SOA distributes functionality over independent services. Components combine their functionalities to fulfill the requirement, but it is more important that these could use (or be used by) a common component. Multi-layer systems eliminate traces between data and presentation or limit the number of these traces (might also mask some traces).

4. Acceptance testing

Acceptance testing is defined in [5] as formal testing with respect to user needs, requirements, andbusiness processes conducted to determine whether or not a systemsatisfies the acceptance criteria and to enable the user, customers orother authorized entity to determine whether or not to accept the system.

As end users, customers and/or customer proxies provide this type of testing, the direct relation to the initial requirements cannot be guarantied.The professional qualities and used domain specific language of the domain experts in the role of acceptance testers might imply identification of inconsistencies in the requirement model and of missing requirements.

During acceptance testing process, testers fill defect report documents. This is very important for our proposal as well, because the content of these reports is aimed to be preserved, but the way of collecting data is modified by automation based on traceability extensions.

5. Proposed Architecture and Method

The information system architectures have a common unit representing data storage medium with a kind of its management, mostly implemented as a database schema with defined integrity criteria running on a database management system. Our proposal is based on this assumption.

The method

We propose a method, where additional database objects representing knowledge about the system behavior extend the system's database. To separate data logic of the system itself and behavior records, there cannot be a direct connection (e.g. relation) to the real application data. Therefore, data should be logged as part of behavior records.

The process of the method is as follows:

1. Allow the user to report a defect anytime by filling a feedback form.
2. Allow the user to fill the feedback form if the system discovers a defect.
3. The user fills only domain specific text fields (describing a possibly new requirement), but is also allowed to select from the list of requirements related to the actually used unit of the system.

4. When sending the form (i.e. storing information into behavior records), the information is extended by data from the application itself – these are used to precisely locate the implementation unit the report belongs to, and data from the last operation (to improve knowledge for better defect identification).
5. In the case of hidden error recovery in the system (i.e. defect is not reported to the user), the behavior defect is recorded too.
6. As behavior records are stored independently, data consistency is not violated.
7. The stored data can be used after connecting the reporting/maintenance tool to the system (as it is not intended to be a part of the production information system).

The method workflow is shown on Fig. (2).

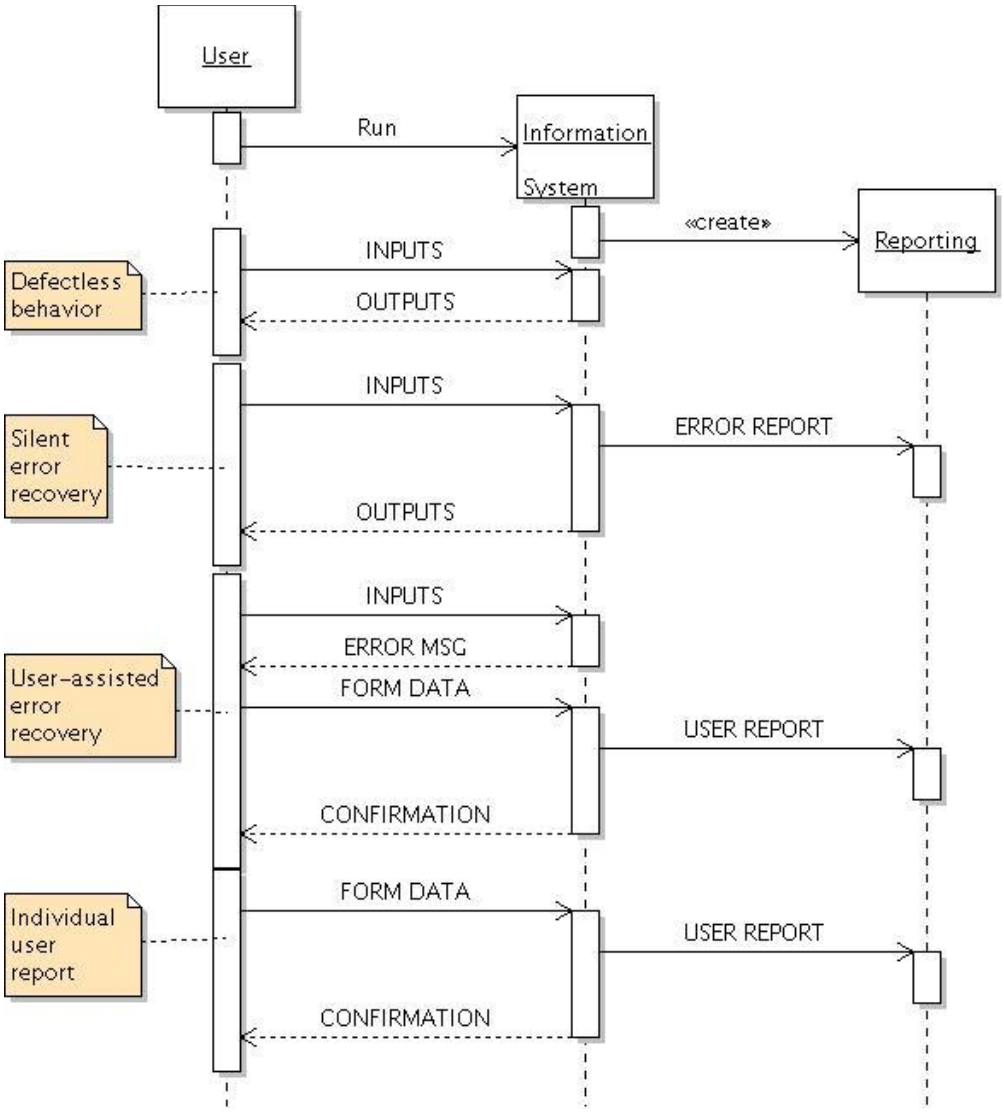

Figure (2): The proposed method

Notes on the different architectures

For the presented three information system architectures, we present the following implementation units:

1. Component-based architecture offers the solution using a shared reporting component that will be used as the base for other components and for error recovery, equipped with user interface element as defect reporting form. Implementation is possible in many languages, because there isa number of components for logging that seem to be extendable. Depending on the type of defect handling (presented to the user or not), the records might contain besides component data a reference to an existing requirement as well.
2. Multi-layer architecture makes almost impossible to implement the proposed method for it. A reasonable solution might be the execution of each of its functions in a reporting environment – e.g. in the frame of a special program, that has access to each layer. This would violate architecture. Second alternative is to separate reporting into layers:
    a. Database layer reports are fully automated and are related only to invisible error recovery (i.e. not propagated to upper layers in the form of messages). These include relation definition to the tables, events and triggering procedures at database level only.
    b. Business layer (application logic) reports will be the ones, where automatic error recovery for business logic solutions is implemented without showing an error or information message to the user. These records include data information extended by business layer related information.
    c. Presentation layer reports are of two types: automatic/hidden and user reports. The automatic in recording is related to the error recovery at this layer that does not show up to the user. All other reports need user interaction. The automatic reports include all available data. The user report will have an optional property assigning the report to a specific requirement. For that, requirements must be defined and stored together with the system implementation. The requirements must be expressed using their original, domain specific form. As an option, agile models such as user stories could be used.
3. For SOA, only separation of reporting is acceptable: with separated records related to the specific service only; as these services are independent. To ensure communication for defect recording, a specific type of method must be provided by each service. If the defect is discovered only by the specific service, there is an implementation detail whether it will inform the caller about it. If the caller (master application) discovers a defect, it could ask the user for assistance and might record information related to requirements as well. This is done only for the master and will not be sent to the used services, because these must be independent. This architecture needs a kind of requirement tracing, where requirements defined for the master system only are traced to the implementing services.

6. Conclusion

We presented related work in the field of information system architectures and requirement engineering with emphasis on vertical and horizontal traceability in software projects.

After presenting the requirements-implementation-evaluation tripartite and the problems of acceptance testing with the lacking traceability property, we presented our proposal of an architecture, which provides the functionality to not only report a defect in the system, but also automatically identifies its relations to the existing parts of the system.

The advantage of such a system is the ability to increase maintenance speed and effectiveness, and to introduce new requirements together with traces onto existing implementation units.

A disadvantage might be the extension of the whole system by additional database objects reflecting its structure and implementing the required knowledge representation for internal behavior.

The next step of our research is the implementation of experiments to ensure applicability of the presented abstract architecture and method.

The future of the architecture and the whole approach is in its extendibility to a kind of self-reflecting system by replacing the defect reporting interface by a hidden automated behavior recording component. Further, the self-reflection could be used as a basis for self-adaptation or self-healing after finding an appropriate evaluation and adaptation or healing mechanism.

Acknowledgment

This work was supported by the Cultural and Educational Grant Agency of the Slovak Republic, Project No. 050TUKE-4/2013: "Integration of Software Quality Processes in Software Engineering Curricula for Informatics Master Study Programme at Technical Universities - Proposal of the Structure and Realization of Selected Software Engineering Courses."

References

[1] Szabó, Cs., "Concept of the Automated Information System Adaptation to Users' Requirements," *5th PhD Student Conference and Scientific and Technical Competition of Students of Faculty of Electrical Engineering and Informatics Technical University of Košice, Proceedings from Conference and Competition*, Košice, Slovakia, 2005, pp. 109-110.

[2] Szabó, Cs., Pločica, O., Havlice, Z., "Application of AI in the Multi-Layer Architecture of Information Systems," *Proceedings of the Seventh International Scientific Conference Electronic Computers and Informatics ECI 2006*, Herľany, Slovakia, September 20-22, 2006, Košice, VIENALA Press, pp. 52-57.

[3] Adamuščinová, I., Révés, M., Havlice, Z., "Using Architectural Knowledge in Process of Software Maintenance," *SAMI 2010 The 8th International Symposium on Applied Machine Intelligence and Informatics*, Herlany, Slovakia, pp. 83–88, 2010.

[4] Kreutzová, M., Porubän, J., "Domain Usability of User Interfaces," *Proceedings of CSE 2012 International Scientific Conference on Computer Science and Engineering*, Stará Lesná, High Tatras, Slovakia, October 3-5, 2012, pp. 39-46.

[5] *Standard glossary of terms used in software testing, version 2.2*, October 19, 2012. Produced by the 'Glossary Working Party' International Software Testing Qualifications Board.

[6] Samuelis, L. et al., *Software Testing Fundamentals: Introduction to Software Verification Theory*, Faculty of Electrical Engineering and Informatics of the Technical University of Košice, 2013.

[7] Pataki, N. et al., "Features of C++ Template Metaprograms," *Proceedings of the 8th International Conference on Applied Informatics (ICAI 2010)*, Vol. 2., 2010.

[8] Simon, M., Pataki, N., "SQL Code Complexity Analysis," *Proceedings of the 8th International Conference on Applied Informatics (ICAI 2010)*, Vol. 1., 2010, pp. 353-359.

[9] Samuelis, L., "Notes on the Emerging Science of Software Evolution," *Handbook of Research on Modern Systems Analysis and Design Technologies and Applications*, USA, Idea Group Inc., 2008, pp. 172-179.

[10] Abrahamsson, P. et al. "*Agile software development methods – Review and analysis*," Otamedia Oy, Espoo: VTT Publishing, 2002.

[11] Egyed, A., Grünbacher, P.,"Supporting Software Understanding with Automated Requirements Traceability,"*International Journal of Software Engineering and Knowledge Engineering (JSEKE)*, Volume 15, Number 5, 2005, pp. 783-810.

[12] Medvidovic, N., Grünbacher, P., Egyed, A., Boehm, B.,"Bridging Models across the Software Life-Cycle,"*Journal for Software Systems (JSS)*, Volume 68, Issue 3, December 2003, pp.199-215.