Supporting Architecture Elaboration
A Use-Case-Based Checklist

Technical Report
RA-01/10

Bartosz Michalik, Mirosław Ochodek

Institute of Computing Science
Poznan University of Technology

Project operated within the Foundation for Polish Science Ventures Programme co-financed by the EU European Regional Development Fund.

Contents

1 Introduction **4**

2 Use-Cases-Based Requirements Specification **6**

2.1 Use-Case Transactions 7

3 Functional-Specification-Based Checklist **9**

3.1 Interoperability Issues 10

3.1.1 Actors 10

3.1.2 Business objects 11

3.2 Object State Management Issues 11

3.3 Data Access Issues 12

3.3.1 Transactions profile 12

3.3.2 Business objects 12

3.4 Dynamic Retrieve Issues 13

3.4.1 Dynamic Retrieve Transactions 13

3.5 Consistency Issues 14

3.5.1 Transfer and Link Transactions 14

3.6 Security Issues 15

3.6.1 Actors 15

3.6.2 Business Objects 16

4 Conclusions **18**

Chapter 1

Introduction

Software architecture can be documented with the use of viewpoint or design decisions models. In viewpoint centric approach, architecture is presented in multiple perspectives, which address specific concerns. One of the well known examples of viewpoint centric approach is the 4+1 view model proposed by Kruchten [10]. With the use of this framework system can be observed in five views: logical (functionality decomposition), development (modules organization), process (performance, distribution, integrity), physical (system topology), and scenarios (components defined in other views in action). The viewpoint centric frameworks were a reference point for the definition of architecture model presented in IEEE 1471 [5].

Recently shift in the direction of design decision can be observed. While viewpoint centric models present the structure of architecture, design decision centric (DD) models are constructed to capture broader view on architectural knowledge. In addition to design decisions, DD models capture "rationales, the design rules, design constrains and additional requirements" [7].

Typically, design decisions are made by architect to address requirements specified for the system under development. The term *requirements* covers both functional and non-functional aspects of systems, however non-functional requirements are perceived as the main factor influencing the shape of software architectures [9, 11, 20].

Functional requirements are also perceived as important from the software architecture-design point of view [14, 18]. However, so far, their main role was to visualize architectural decisions, rather than stimulate architect to make them. For instance, in Kruchten's 4+1 view model, functional requirements can be presented in the use case view. Their goal is to illustrate how the system components cooperate during the scenario execution. Similar role of functional requirements can be observed in the Architecture Tradeoff Analysis Method [9] (ATAM). In ATAM functional requirements are used in use-case scenarios to animate discussion regarding architectural concerns in the context of quality attributes. Again, only a chosen subset of functional requirements defined for a system is considered as the background for discussion.

Although, functional requirements are considered as less important than non-functional requirements from the software architecture point of view. It is hard to imagine that architect could propose a set of design decissions for a system, without getting at least an overview of its functionality. Typically this is done through a kind of *informal, non-systematic process* of reading software requirements specification and discussing main system-features with analyst. As a result, architect should be able to propose a software architecture which enables

implementation of the defined functionality.

Relying on both functional, and non-functional requirements seems to be a good idea, because requirements are often incomplete. (What is in fact one of the most important reasons for projects' failures [2, 12, 19, 21].) Therefore, a question arises whether it is possible to use *functional requirements* to mitigate the risk of designing architecture on the incomplete set of non-functional requirements.

In this study we would like to propose a *systematic approach* to analysis of use-case-based functional requirements specifications, which can be used by architect to support analysis of functional requirements from the architecture-design point of view. We would like to propose a *checklist* covering issues that can have a direct impact on the shape of architecture, and/or can help to reveal *not articulated* non-functional requirements, which otherwise could remain unidentified (and might become harmful later on).

The proposed checklist might be also used to support the analysis stage of the ATAM method [9] or during the construction of so-called utility-tree (a tree-like structure presenting quality criteria for the system).

This report is organized as follows. In Chapter 2, elements of use-case-based requirements specification are described. The use-case-based checklist for architecture quality improvement is presented, and discussed in Chapter 3. Finally, conclusions are presented in Chapter 4.

Chapter 2

Use-Cases-Based Requirements Specification

Functional requirements can be specified from the user's and system perspectives [22]. In case of the system perspective, the idea is to define the functions which system should provide (i.e. "System should enable uploading a paper"). If the user's point of view is considered, a focus is put on defining the goals which a given user would like to obtain by using the system.

One of the most popular approaches to present functional requirements from the user perspective, is to write use cases [6] (according to the survey conducted by Neil and Laplante in 2003 [13], 50% of projects have their functional requirements presented as scenarios or use cases).

Use cases present interaction between so-called main actor (end-user, system, or device) and the system under development in terms of user-valued transactions. That kind of use cases are called system-level use cases. There are also business-level use cases, which describe interaction between people who cooperate to obtain a business goals.

Typically, use cases are presented with the use of natural language. According to guidelines for writing use cases [1, 3] the most important parts of a use case are: *name/title* which describes the goal; *actors* participating in use case (people, devices, or co-operating systems); *main scenario* which is the most common sequence of steps leading to obtaining the goal; and *alternative scenarios, extensions, exceptions* to the main scenario describing alternative behaviour.

A use-case-based functional requirements specification (see Figure 2.1) contains not only actors and use cases, but also a set of *business objects*. Business objects are entities existing in the domain of problem, which are processed within the scenarios of use cases. They also constitute a kind of conceptual database of the system being developed.

Definitions of actors and business objects might be available even before use cases are defined if a *context diagram* [22] has been developed for the system.

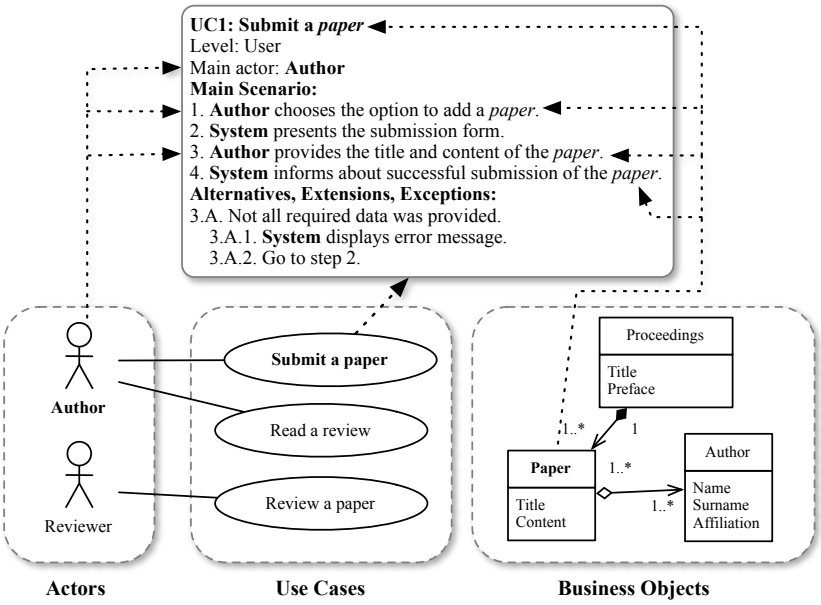

Figure 2.1: Use-case-based functional requirements specification (with an example of the textual representation of a use case)

2.1 Use-Case Transactions

The smallest unit of interaction between actors in use cases that is meaningful from the actor point of view, is called a use-case transaction.

The definition presented by Diev [4] states that use-case transaction has to satisfy two conditions:

- UCT-C1. Use-case transaction is the smallest unit of activity that is meaningful from the actor's point of view.
- UCT-C2. Use-case transaction is self-contained and leaves the business of the application being sized in a consistent state.

Use-case transactions can be available at early stages of software development, as they can be used for effort estimation [8]. In addition they can be identified automatically with the use of NLP techniques [15].

Use-case transactions can be also analyzed from the semantics point of view. Some of the transactions share similar goals (although their exact actions are different). Based on such similarities of goals, 12 semantic transaction-types has been defined so far [16] :

- Create (C) – create an instance of object;
- Retrieve (R) – retrieve and present the object;
- Update (U) – modify stored object;

- Delete (D) – remove stored object;
- Link (L) – connects two or more objects, which are not a composite;
- Delete Link (DL) – remove connection between linked objects;
- Asynchronous Retrieve (AR) – retrieve object and provide it to actor in the future (not as direct response);
- Dynamic Retrieve (DR) – retrieve objects based on dynamically defined criteria;
- Transfer (T) – transfer objects between actors (they are replicated or change the owner);
- Check Object (CO) – object is checked against some business rule;
- Complex Internal Activity (CIA) – launch not trivial internal processing operation (typically observed in systems with simple GUI and complex algorithms beneath);
- Change State (CS) – similar to Update but the modification changes also object behaviour in the system (e.g. change the state of article from draft to published state).

For instance, in the use-case “Submit a paper" presented in Figure 2.1, one can identify a *Create* transaction (instance of object *paper* is added to the system).

Chapter 3

Functional-Specification-Based Checklist

As presented in Section 2, use-case-based functional requirements specification consists of actors, use cases (which are built from use-case transactions), and business objects. Based on the data from several *software intensive systems*, we investigated the influence of each of this elements on design decisions, and non-functional requirements, which were defined (or were not initially articulated, but should be, and emerged later on).

As a result of this analysis a checklist was constructed, which covers issues related to data access, consistency of data, object-state management, dynamic retrieve of data, interoperability, and security. These issues will be presented and discussed in the following sections of the report, together with examples coming from the projects being subjects of this study.

Unfortunately, reasoning process and checklist generation can differ depending on application area, and technique used to express functional requirements. For the purpose of this report several middle size (up to the 200 KLOC) software intensive systems were analysed. Therefore the proposed checklist should be treated as an example of how functional requirements (e.g. use cases) can support development of architecture for the system (e.g. web-based software intensive systems).

The examples presented in this report were chosen from two projects, for which authors had a full insight into the production process.

System M is a university e-recruitment system. It was developed by the software development unit at the Poznan University of Technology (PUT). It supports students' admission process. System supports the documents flow between candidates and the university; definition of the recruitment process; fee collection; and ranking process. The System M cooperates with multiple legacy systems at PUT. One of them is the System C which is a system supporting administrative work of each department at the university. (Each department has its own instance of the system.) Recently, PUT started migrating their systems towards the SOA architecture. Therefore, System M communicates with some services at PUT using the Web Services technology. Unfortunately, System C was not SOA-enabled, and System M had to

use other existing communication mechanisms.

System T is a production system which is developed for the mailing service company. System supports complete flow of the orders starting from the transportation management, through the service, up to the billing activities. System supports mobile couriers as well as service groups, quality control, and accounts divisions. System was developed and is still maintained by the SDG-BI company.

3.1 Interoperability Issues

Interoperability issues relate to communication between the system being developed and its environment. The main task is to identify communication protocols and data that is exchanged. This process can be done through the analysis of use-case-based requirements specification. After interfaces are identified, questions regarding non-functional requirements relating to performance, interoperability, security, or fault tolerance might be asked. This can help to mitigate the risk regarding incompatible interfaces definition.

3.1.1 Actors

Identify actors (look at use cases and context diagram), and answer following questions:

Q1: What type of interfaces actors use? Human actors usually communicate with the system through a kind of UI (e.g. GUI, terminal console). If actor represents device or external system, architect should investigate what kind of interfaces such devices or system provide (which can be potentially used as communication channel).

Q2: Does a single actor use many types of interfaces? Q3: Are different interfaces used for exchanging the same business object? Answer to this questions can help architect to define a proper communication abstraction layer. For instance, if a single object can be provided through different interfaces, architect might investigate data formats which are acceptable by such interfaces, and e.g. consider inclusion of the transparent conversion layer.

Example: System M communicated with two external systems. Although at the university a shift towards SOA was observed, only one of the systems provided a web-service interface. Direct database access was the only possibility to access an instance of System C. From the functional requirements, one could read that System M had to communicate with several instances of System C, working at different departments (different versions problem). Moreover, one of the non-functional requirements was to be prepared for a change to web-service communication. Therefore, a combination of these requirements motivated decision about modularization of communication modules.

Q4: Is the system actor active or passive one? System actors in use cases can appear in active or passive roles. For instance, in the use-case step "System X sends data to system Z", system X is an active actor, and system Z is a passive one. If the active system is the system under development, it can use one of the interfaces provided by the passive system (or introduce a new one if it is possible to modify the passive system). On the other hand, if the passive system is the system to be developed, it has to provide at least one of the communication interfaces which is supported by the active system.

Q5: Is the communication with external actors synchronous or asynchronous? Asynchronous communication may involve storing temporally data. Asynchronous interfaces can be identified directly through the analysis of asynchronous retrieve (AR) transactions. However, transactions of other types, e.g., C,R,U,D,T,L might be investigated, because they can also involve asynchronous communication. Creating an account at the web forum can be given as an example. Quite often, the activation of account is done through e-mail confirmation.

3.1.2 Business objects

Business objects represent an abstraction of the future data that will be exchanged through the interfaces. A following question, related to business objects and interoperability of systems, should be considered:

Q6: Is communication interface capable of transporting business object between actors? This question can be decomposed to following sub-questions:

- *What format is required by business objects shared or processed by actors?*
- *What is the amount of data being exchanged?* - performance requirements recognition.

Example: System M communicated through a Web Service with the System E. System M was an active side of the communication (Q4). After analysis of business objects which were supposed to be exchanged between both systems, it appeared that the System E did not have the interface capable of exchanging one of the business objects. As a result, management of the project A decided to send a request to the team responsible for maintenance of the System E, to implement the new interface.

3.2 Object State Management Issues

Object state management is an issue closely related to the *change state* (CS) transactions in the functional specification of the system. The most important problem in this context is:

Q7: How to handle object state changes?

To tackle with the state management three approaches can be considered: to employ COTS workflow/state management library (like Commons SCXML); to implement state management layer; or to handle objects change-of-states manually in a business layer. To choose one of these approaches, one can consider answering to following questions relating to the *change state* transactions:

- *Q7.1: How many CS transaction exist for the key business objects?*
- *Q7.2: How complex is the change state diagram?*
- *Q7.3: What actions accompany the change of states?*
- *Q7.4: How many actors can change the state of a given business object?*

If analysis leads to the conclusion that complex change state process exists for one or many business object, the COTS workflow/state management library seems to be a proper choice. In addition non-functional requirement concerning flexibility or modifiability can be defined.

Example: System T had several objects that could change their states. For some of the change-of-state operations, additional actions were supposed to be performed together with the change of state (e.g. sending an e-mail). Some of the changes-of-state transactions involved also performing undo operations. Unfortunately, this problem was not identified correctly. (There were no non-functional requirements available that could justify the need of a workflow system; in addition the analysis of change state transactions was not performed as well.) As a consequence, a wrong decision was made to implement the change-of-state management in a business logic. As a result, further reconfiguration of the flow was a cumbersome task, because the change management code was scattered in multiple places of the business layer.

3.3 Data Access Issues

Data access issues relate to the data persistence, performance and resource utilization. Common architectural design decisions concern the number of abstraction layers, choice of the data persistence mechanism (relational database, object database, repository, file system etc.), compression, caching, query optimization.

3.3.1 Transactions profile

After use-case transactions are identified, the analysis of transactions profile could be performed as a simple technique for the data-access issues recognition. A transactions profile is a distribution of different types of transactions that were identified in all use cases describing system.

Q8: What is the transaction profile of the system being developed? When the number of CRUD transactions is visibly higher than the number of transactions of other types, we are likely to deal with the data intensive system. (If not, architect should decide whether data-access issues are still important for the system.) As the transaction profile could be available at the very beginning of the project development life-cycle, an appropriate architectural style can be chosen early on. Other analysis of the transaction profile can be also performed. For instance, one can investigate the transaction profile from the perspective of business objects. For example, if *retrieve transactions* dominate other data access transactions, the introduction of caching mechanism might be considered.

3.3.2 Business objects

Business objects describe the domain related entities processed during the end-user interaction with the system. Most of them constitute (directly or after decomposition) the data model of the system being built. Therefore, business-objects analysis can be helpful when dealing with the data access related issues.

Q9: How many data sources are present in the system? For each object, one should investigate which actors access business object to read (R, DR) or write them (C,U,D,T,L,CS). Especially valuable could be investigating what actors manage the objects (i.e. on side of which actor, objects are stored). If some data sources are not local (e.g. managed by actors that represent external systems) interoperability issues should be also investigated (see Section 3.1).

Q10: How complex the business objects are, and how intensively they are used? Complex business object are likely to be decomposed during the domain model modeling phase. If

these object are extensively used (e.g. there are multiple *retrieve transactions* performed by different actors), the decomposition can lead to the performance problems in the system. Therefore, availability of the performance requirements relating to those transaction should be investigated.

Example: System T had a visible CRUD transactions profile. In addition, business objects processed in the system were stored in two separate data sources. Therefore, the data-access layer separation was one of the design decisions made. In this layer data access object (DAO) was constructed for each business object. As the CRUD operations for every business object had the same semantics, the *generic DAO pattern* was used for the layer implementation. This allowed for the simple implementation of the security and data-access transaction mechanisms.

3.4 Dynamic Retrieve Issues

The dynamic retrieve (DR) issues concern multiple problems of how users search for the data managed by the system. In many systems the dynamic retrieve is the main functionality they provide (e.g. Google search, eBay querying mechanism). The non-functional requirements accompanying this functionality can be related to usability (user interface for defining search criteria), performance, resources utilization, security etc.

In data intensive systems two main patterns of dynamic retrieve can be observed: customized search performed with the use of query language; and filtering of data presented to end-users. The dynamic retrieve affects multiple layers of the system, e.g., presentation layer, data-access layer, and data-storage layer.

3.4.1 Dynamic Retrieve Transactions

During the analysis of dynamic retrieve issues performed based on use cases, the first step should be directed towards the *dynamic retrieve* (DR) transactions. Quantitative evaluation can help to investigate whether dynamic retrieve is an important issue in the considered system. The questions which should be answered are as follows:

Q11: For which business objects dynamic retrieve should be available (none, all, or a subset of them)? The answer can indicate whether the chosen mechanism should be general (e.g. dedicated module for all business objects) or it can be handled directly (e.g. user query only for one of the domain objects).

Q12: Should indexing mechanism be present in the system? This is a first of the patterns architect can choose. Indexing mechanism is a way of supporting customized queries (e.g. expressed in natural language). The decision should be made with regard to the selection of business objects attributes to be indexed, and selection of business objects to index. Wrong selection can lead to decrease of search-results accuracy. In addition, one should make decisions regarding the index storage, and access optimization. In this context analysis of CRUD transactions can be helpful.

Q13: Should the filtering mechanism be provided? It happens that motivation for mechanisms, like filtering, are covered directly in the use cases. Architect may look at the screen sketches if available, or at the content of the *DR* transactions.

- *How to build a secured filtering-mechanism for the data-access mechanism which was chosen for the system?* This question should be analysed with respect to the data access mechanism. As dynamic-retrieve deals with the end user data, discussion may reveal the risks of given solution in the context of different attack types.
- *Is the list of filtering attributes static or dynamic?* First of all, answer to this question may help to define the data layer interface. If static-attributes approach was identified, the risk relating to the model evolution should be assessed. Next (especially when DR is common for the most of business object), the performance issues should be investigated.

Example: Large part of the System T was a form-based web application to manipulate the data. For almost all business objects *dynamic retrieve* transactions were identified. Dynamic retrieve in this case was a dynamic filtering function for all attributes of each business object. As the data access was realized by Hibernate ORM, there was a need for a query building mechanism. To reduce the size of glue-code and increase modifiability of the service layer, the query generation module was designed (using the builder pattern). The query was automatically constructed from the list of attributes and simple relations (e.g. range, equality, similarity) defined for each attribute.

3.5 Consistency Issues

Consistency of the data model is an important issue in the software intensive system. The main risks regarding consistency of data relate to the choice of method used to integrate different data sources; the number of a different ways the data model is used; or the evolution of the model.

In addition, many non-functional requirements concerning interoperability, changeability, fault tolerance, or co-existence can affect the consistency of data.

3.5.1 Transfer and Link Transactions

To mitigate the risk of making a bad design decision affecting consistency of data, one can look at the *transfer*, *link*, and *delete link* transactions. Quantitative analysis of this transactions helps to investigate how important is the consistency assurance problem. Then, more precise questions can be asked:

Q14: How to deal with delete link and delete business object transactions? Deleting objects can have a negative impact on consistency of the data model. *Delete link* transactions can be especially harmful when association is removed between objects maintained by different actors (e.g. objects shared by different systems). Additional issue relate to storing historical versions of objects. Architect should investigate, which objects can be deleted permanently, and for which of them deletion means, e.g., hiding them for some actors.

Example: In the System T some business objects were imported from the external CMS system. These objects have to remain associated (and synchronized) with the corresponding objects in the CMS system. Preserving consistency of associations was crucial for the correct behaviour of the system. (Inconsistency of associations could have serious consequences for the production process in the company). Unfortunately due to legal issues, only a read-only database access to CMS was available (the modification of the CMS system was not possible). Therefore, the risk of data inconsistency was high (e.g. as a result of re-numeration of artificial

keys of the CMS data tables; or change in the DB schema – what happened). To limit the risk, a multi-attribute association key was chosen, and periodical consistency-checking mechanism was implemented.

Q15: What data-access transaction mechanism should be used? When *transfer transactions* (T) are identified in use cases, it usually indicates the need for supporting the atomicity of operations. Depending on the nature of transfer, the appropriate transaction-support mechanism should be chosen. For example, if transfer of business objects is performed within a single data source, adequate design decisions might be to choose solutions like JPA transaction mechanism. In multi-sources environment, or in situation where a transfer transaction is supposed to be performed asynchronously, a different solutions, like e.g., JMS might be considered.

Example: One of the features of the System M was to be capable of exporting lists of future students (accepted candidates) to each instance of the System C. This function was implemented as a batch export-process. During that process, System M communicated with the external System E management service to obtain IDs for the new students. Therefore, the transfer transaction involved three parties (three systems), and had to be performed entirely or not at all. For instance, it could happen that the student's ID was obtained for the accepted candidate (from the System E), but the export of his/her data to the System C failed. As a result, accepted candidate would have its student's ID reserved, but would not appear as a student for the administration of faculty (in System C). Unfortunately, the System E did not provide any failure recovery mechanisms. Therefore, it was not possible to rollback the assignment of student's ID. To solve this problem, architect decided that the System M should store the information about the operations that failed, and periodically resume/redo them.

3.6 Security Issues

Security requirements are mainly expressed by non-functional requirements, however analysis of use-case model can provide some additional information regarding security issues, which can be valuable to architect.

3.6.1 Actors

Actors represent people, devices, and other systems co-operating with the system under development. Analysis of actors can provide information regarding the nature of the access control mechanism (roles, permissions etc.), as well as information concerning the security of communication interfaces.

Q16: Is the access control mechanism static or dynamic? Actors can be understood as roles available in the system. In case of some systems they can map one to one to the roles identified within the organization, for which the system is developed. If no additional non-functional requirements were defined concerning flexibility of the access control mechanism, actors can be used directly to define, and implement explicit roles available in the system (*static* roles). Another approach is to use *dynamic* roles definition mechanism. In this approach, roles can be dynamically composed from the permissions available in the system. Typically if the dynamical access control mechanism is considered, use-case model will include use cases defined by Övergaard and Palmkvist [17] as a dynamic security units blueprint (e.g. "Manage access rights", "Manage security unit").

Example: System M was an example of system with dynamic access control. Administrator could create new roles dynamically. There was even a use case "Manage roles", which described how new roles could be created.

In the System T many roles could be assigned to a user, however a set of available roles was explicitly based on actors (a static list of roles was available).

Q17: How secure communication interfaces between actors should be? Communication interfaces that system should provide, can be identified through the analysis of interaction between actors. Available interfaces can be identified based on the context diagram and/or use cases. (Appearance of interaction between actors implies existence of communication interface between them.) Architect should investigate whether non-functional requirements regarding security were defined for all interfaces, as well as if there are any constraints regarding the type of an interface which can affect security.

Example: For instance, in the System M, there were many human-actors communicating with the system. After the analysis of security requirements concerning communication channels for these actors, it appeared that channels used by Administrator, Admission Manager, Admission Committee, and Examination Committee require similar, and more secure access to the system than channel for Candidate, because these actors have access to confident information (e.g. personal data of candidates). As a result additional security mechanisms were used for the part of functionality used by these actors, like e.g., encrypting communication, and restricting access to them only from the university internal network.

Q18: Do all actors will use the same access control mechanism? Investigate whether all actors are supposed to use the same access control mechanism, or maybe requirements concerning security of interfaces demand different level of security for some of them.

Example: In the System T access-right mechanism is based on roles. In addition, different groups of users communicated using one of two avaiable interfaces. Couriers with the mobile devices used REST interface, whereas other employees accessed the system through a web interface (intranet). Access application developed by 3rd part company was able to authenticate with Basic Authentication mechanism, while web users were authenticated with the Form Base Authentication. Both mechanisms were implemented in the system. Spring Security was chosen for the security layer implementation. In addition, mobile client communicated via HTTPs protocol. In order to limit the risk of data leak, only part of the system functionality was revealed through the HTTPs protocol.

3.6.2 Business Objects

Business objects are another element important from the system security point of view. The most important issues is access control of business objects.

Q19: Are permissions to business objects instance-specific or type-specific? Type-specific access permissions means that actor can perform a given operation on all objects of a given type, while in case of instance-specific access control actor has permission to perform a given operation on a chosen subset of objects of a given type.

Example: In the System M permissions were instance-specific. For instance, Admission Committee could access candidates' personal information only if they sent application to their major. However, Dean could access personal information of all candidates who applied for any major within the faculty. As a response to this requirements a design decision was

made to use a VPD feature of Oracle DBMS, and add a special layer of permission control, which enabled checking access rights in the context of a given object (e.g. accessing all majors at a given faculty). This is an example of influence of coupled functional and non-functional requirements on the design decision.

System T, however, was example of a system where security access is restricted at a function level (URL-based). Access restriction is based on the set of static roles which can be assigned to system users. As one of the actor has a remote access to the system, two security protocol are implemented.

Q20: Are object transferred between system actors? Investigate all *transfer transactions* in use cases which are performed by system actors. For each of them identify business objects that are transferred. If objects are transferred between systems there is a question whether access rules that were applied in the source system should be also transferred to the destination system (e.g same users should have access to certain objects in both systems).

Chapter 4

Conclusions

In the report we presented how the improvement of software architecture quality can be supported by the systematic analysis of functional requirements (defined in a form of use cases).

Based on the analysis of several middle-size data intensive systems, we proposed a checklist consisting of 24 questions (regarding different elements of use-case-based requirements specification), and addressing 6 areas important from the software architecture point of view (interoperability, object state management, data access, dynamic retrieve, data consistency, security).

By answering these questions architect can investigate whether proposed architecture is capable of handling the functionality of the system; make design decisions motivated directly by functional requirements; and identify additional (not articulated) non-functional requirements. The checklist might be also incorporated into the architecture evaluation methods like ATAM, as additional to non-functional requirements source of information.

The proposed approach has also some limitations. First of all, the checklist was constructed based on the analysis of the limited set of systems. Therefore, some issues could be specific only for that systems. From the same reason, there might be some additional questions related to use cases and architecture, which are important, but were not identified in this study. Therefore, further analysis of architectural issues should be performed for different classes of systems.

Acknowledgments.

First of all, we would like to thank Łukasz Konopko from the SDG-BI company and the PUT software development team, for the cooperation. We would also like to thank Jerzy Nawrocki for the fruitful discussions of the problem described in this paper.

Project operated within the Foundation for Polish Science Ventures Programme co-financed by the EU European Regional Development Fund.

Bibliography

[1] S. Adolph, P. Bramble, A. Cockburn, and A. Pols. *Patterns for Effective Use Cases*. Addison-Wesley, 2002.

[2] O. Albayrak, D. Albayrak, and T. Kilic. Are software engineers' responses to incomplete requirements related to project characteristics? In *Applications of Digital Information and Web Technologies, 2009. ICADIWT '09.*, pages 124–129. IEEE, 2009.

[3] A. Cockburn. *Writing effective use cases*. Addison-Wesley Boston, 2001.

[4] S. Diev. Use cases modeling and software estimation: applying use case points. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–4, 2006.

[5] IEEE Architecture Working Group. IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000.

[6] I. Jacobson. Object-oriented development in an industrial environment. *ACM SIGPLAN Notices*, 22(12):183–191, 1987.

[7] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.

[8] G. Karner. Resource Estimation for Objectory Projects. *Objective Systems SF AB (copyright owned by Rational Software)*, 1993.

[9] R. Kazman, M. Klein, and P. Clements. *ATAM: Method for Architecture Evaluation*. Carnegie Mellon University, Software Engineering Institute, 2000.

[10] PB Kruchten. The 4+ 1 View Model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[11] Philippe Kruchten, Rafael Capilla, and Juan Carlos Due nas. The decision view's role in software architecture practice. *IEEE Softw.*, 26(2):36–42, 2009.

[12] L.J. May. Major Causes of Software Project Failures. *CrossTalk–The Journal of Defense Software Engineering*, 11(7):9–12, 1998.

[13] C.J. Neill and P.A. Laplante. Requirements Engineering: The State of the Practice. *Software, IEEE*, 20(6):40–45, 2003.

[14] B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(2):115–117, 2001.

[15] Mirosław Ochodek and Jerzy Nawrocki. Automatic Transactions Identification in Use Cases. In *Balancing Agility and Formalism in Software Engineering: 2nd IFIP Central and East European Conference on Software Engineering Techniques CEE-SET 2007*, volume 5082 of *LNCS*, pages 55–68. Springer Verlag, 2008.

[16] Mirosław Ochodek and Jerzy Nawrocki. Enhancing Use-Case-Based Effort Estimation with Transaction Types. *Foundations of Computing and Decisions Sciences*, 35(1), 2010.

[17] G. Övergaard and K. Palmkvist. *Use Cases: Patterns and Blueprints*. Addison-Wesley, 2005.

[18] B. Paech, A.H. Dutoit, D. Kerkow, and A. Von Knethen. Functional requirements, non-functional requirements, and architecture should not be separated-A position paper. *REFSQ, Essen, Germany*, 2002.

[19] RB Rowen. Software project management under incomplete and ambiguous specifications. *IEEE Transactions on Engineering Management*, 37(1):10–21, 1990.

[20] Jeff Tyree and Art Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, 2005.

[21] J. Verner, K. Cox, S. Bleistein, and N. Cerpa. Requirements Engineering and Software Project Success: an industrial survey in Australia and the US. *Australasian Journal of Information Systems*, 13(1), 2007.

[22] K.E. Wiegers. *Software Requirements*. Microsoft Press, Redmond, WA, USA, 2003.

Appendix: A

Table 4.1: Use-case-based checklist for architecture quality improvement

Interoperability Issues
Q1: What type of interfaces actors use? Q2: Does a single actor use many types of interfaces? Q3: Are different interfaces used for exchanging the same business object? Q4: Is the system actor active or passive one? Q5: Is the communication with external actors synchronous or asynchronous? Q6: Is communication interface capable of transporting business object between actors?
Object State Management Issues
Q7: How to handle object state changes? Q7.1: How many CS transaction exist for the key business objects? Q7.2: How complex is the change state diagram? Q7.3: What actions accompany the change of states? Q7.4: How many actors can change the state of a given business object?
Data Access Issues
Q8: What is the transaction profile of the system being developed? Q9: How many data sources are present in the system? Q10: How complex the business objects are, and how intensively they are used?
Dynamic Retrieve Issues
Q11: For which business objects dynamic retrieve should be available (none, all, or a subset of them)? Q12: Should indexing mechanism be present in the system? Q13: Should the filtering mechanism be provided?
Consistency Issues
Q14: How to deal with delete link and delete business object transactions? Q15: What data-access transaction mechanism should be used?
Security Issues
Q16: Is the access control mechanism static or dynamic? Q17: How secure communication interfaces between actors should be? Q18: Do all actors will use the same access control mechanism? Q19: Are permissions to business objects instance-specific or type-specific? Q20: Are object transferred between system actors?