Experiences from Monitoring Effects of Architectural Changes

Ulf Asklund, Martin Höst, Krzysztof Wnuk

Department of Computer Science
Lund University, Sweden
{ulf.asklund, krzysztof.wnuk, martin.host}@cs.lth.se

Abstract. A common situation is that an initial architecture has been sufficient in the initial phases of a project, but when the size and complexity of the product increases the architecture must be changed. In this paper experiences are presented from changing an architecture into independent units, providing basic reuse of main functionality although giving higher priority to independence than reuse. An objective was also to introduce metrics in order to monitor the architectural changes. The change was studied in a case-study through weekly meetings with the team, collected metrics, and questionnaires. The new architecture was well received by the development team, who found it to be less fragile. Concerning the metrics for monitoring it was concluded that a high abstraction level was useful for the purpose.

Keywords: software architecture, software metrics

1 Introduction

Architectural changes are often introduced to improve some aspects of a software product or a software development project. The selection of changes and their introduction need to be systematic and well planned [1], followed by a follow-up analysis if the applied changes resulted in the desired improvements. Software metrics can support both change planning and evaluation [2]. These metrics need to accurately describe the principles behind the changes and the main objects of these changes.

Developing software products via prototyping is nowadays widely used. The first prototype version is usually rather small, and then the product and the number of included functions grow. However, during such incremental development a problem can occur that the changes made to the product affect many parts of the product, resulting in that changes can result in unpredictable software faults. This is typically the effect of an immature architecture for the purpose and/or because of a development process with insufficient quality assurance practices. The result is a suboptimal architecture that needs refactoring to bring its quality to an acceptable level. However, architecture changes can not be be carried out in isolation. There is a relationship between the business, the architecture, the process, and the organization, as described by the BAPO model, e.g. [3], and

currently further analyzed in the ITEA project SCALARE[1]. This means that, for example, the development process and the organization also might need to be changed at the same time as the architecture.

This paper presents a case study where requirements changes and a more large scale usage of the product triggered an architectural change. The introduction of the architecture change and a related process change is monitored with a set of object oriented design metrics, inspired by Martin [4]. Versions of these metrics are available in several metrics collection tools, and the objectives of this study include evaluating to what extent they can be useful in the context of an architectural change.

2 Related work

Software refactoring is an integral and important part of software maintenance and evolution and often associated with restructuring [5]. It is a way to restore quality after frequent changes [6], improve extensibility, modularity, reusability, complexity, maintainability and efficiency [6] or transform centralized software components into distributed [7]. Software restructuring is a form of "perfective maintenance" with the goal to modify the structure of the source code and facilitate correctly previously undetected errors [8]. Moreover, it is rather straightforward to estimate the payoffs of restructuring in terms of time and money saved, and shorter development cycles [9]. Refactoring can be achieved with the help of assertions (pre-conditions, post-conditions and invariants), graph transformations, model transformations with semantic annotations [10], aspect oriented concepts [11]. However, these methods are rarely empirically evaluated.

Several authors focused on software architecture stability. Among them, Aversano et al. proposed a set of instability metrics combined with thresholds when the architecture can be considered fully stable, leveling, improving, fully unstable [12]. Tony et al. suggested a metric-based approach for evaluating architecture stability based on: growth rate, change rate, cohesion and coupling and evaluated them on several open source projects [13]. Interestingly, Bahsoon and Wolfgand suggested using real options theory for evaluating architectural stability and estimating volatility, exemplified on ten architectural changes [14]. Figueiredo et al. focused on design stability of software product lines in terms of modularity, change propagation and feature dependency identifying a number of positive and negative scenarios [15]. However, they analyzed two small product lines with 10 KLOC and 3 KLOC.

3 Case description

3.1 Overall architecture

The case system is a client system for server software, intermediate software, specifically developed hardware, and other units. The system architecture is depicted in Figure 1.

[1] `http://scalare.org`

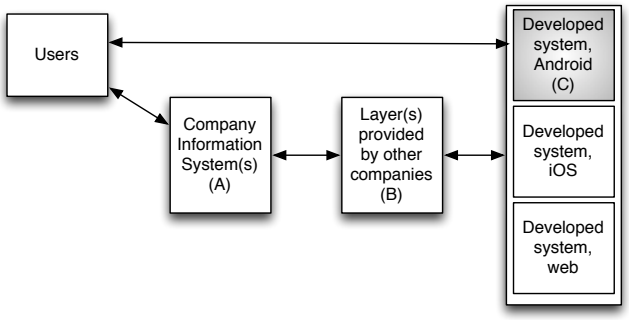

Fig. 1. High level architecture of the whole system

The overall project is managed by Company A and includes a large customer base. Company A has a number of Information Systems (marked A in Figure 2) in order to manage the customer and user data. Other companies, marked as B in Figure 2 access this data in different ways in their applications. The customers receive a number of functions for observing, and taking actions upon the data in the Company Information System (A). The application is developed in three versions, one for iOS, one for the web and one for Android. We studied the Android application in this paper, further called "developed system" in this paper (C). Other companies (B) can influence and extend the functionality provided by the system using the benefits of the layered design. Often, the new functionality development is done in both layers.

The layered design combined with the interaction between the development company and other companies impose several requirements and constraints. Firstly, many companies are involved in the project with some significantly influencing the project scope (Company A). Secondly, the division of the work to be done between the layers is not always straightforward. Thirdly, development cycles should be short since users expect new functionality frequently delivered. Fourthly, reliability requirements are high due to a large amount of users. Finally, the company wants to keep the current maintainability and lead-time levels. The result of high release frequency is limited functionality and complexity of the early versions. Therefore, the case company does not see the initial architecture of the developed system as good as it should be and they are trying to obtain a better architecture which will provide better maintainability.

3.2 Organization and process at the case company

On average, 2-3 developers work full time on the system. About the same number of developers work on the web version and the iOS version of the developed system, and the development must of course be synchronized. Since the project has existed for rather long time there has been some change of personnel, which also puts requirements on maintainability.

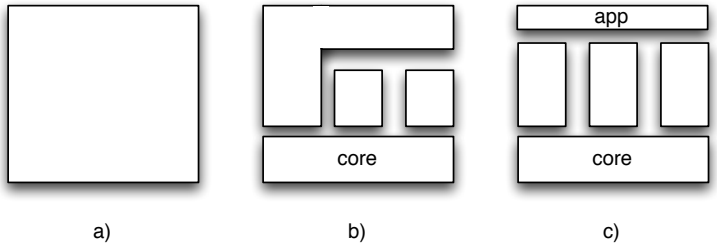

Fig. 2. Architecture change of application

The project followed an agile approach, mainly based on Scrum with collective code ownership where the developers assign the tasks to themselves to the next task independent of what it is and what part of the system it affects.

There were two main reasons for the company to make a change. There was a negative trend of quality issues like old bugs being re-introduced and too many errors found late in testing. There were also several future development activities planned in a near future, including new usage scenarios, targeting new market segments, and developing new business models.

Based on the overall system architecture, the development time requirements remained high and considered together with the business drivers (Extended functionality). The detailed analysis of the change drivers revealed that the developers spend too much time browsing, for them uninteresting, files and documents. Moreover, the developers struggled to find the relevant parts of the system to add new functionality. Finally, extensive dependencies make the developed product quite fragile, i.e. a change to one part of the system has non intuitive dependencies to other parts, which are not always considered by the developers.

3.3 Introduced architectural changes to the developed system

The architecture was designed with focus on reuse, i.e. when new functionality is added, existing classes are reused as much as possible. Extensive reuse may lead to an architecture with many dependencies resulting in a, more or less, monolithic system. This was identified as the major reason to the problems mentioned above, and in order to better structure the dependencies and make the design less fragile, the architecture was divided into modules. Each module implemented one specific function provided to the user, implemented as separate projects in the development environment (Eclipse). Functional decoupling allowed the developers to make corrections or updates of existing functions, e.g. only the code valid for the function was browsed, understood, and updated, which makes the change fast and with high quality. It also allowed for parallel updates of different functions, and new functionality can be added independent of the existing.

The architecture guidelines were changed to focus on independent modules and how to manage them individually. Previously, all developers worked on the

whole code base when changes were implemented, which often required changes to a large part of the system. The new process allows developers to avoid change request in modules they have not yet know - but also to deliberately choose to work in a module for the first time in order to increase system knowledge. From an organizational perspective, the new architecture makes it is easier to scale, letting new developers joining the project to start work on one function in one module, learning the system function-by-function.

The drawback of the new architecture is less reuse and more double-maintenance of "similar" code in different modules. However, the case company believed that changes in the new architecture can be limited to one module and therefore better fulfill the architecture maintainability requirements. Both architectural and process changes were gradually introduced by the case company. This was done by adding one module after another to the initial codebase. This means that there were different versions of the system during the research project.

The gradual changes that were made to the application architecture are sketched in Figure 2. In the beginning, there was a monolithic architecture (Figure 2.a). The objective of the changes was to achieve an architecture with separate modules and a limited of common functionality, as shown in Figure 2.c. The main common layer in Figure 2.c is marked "core" and handles parts of the product that is common to all modules. It also serves as the interface to layers provided by other companies (B). There is also a small common part marked "app" which is the Android application, which, for example, is responsible for configuring and launching the modules. The other parts of the architecture is made up of separate and independent modules. The current situation is that a bottom layer has been formed, and a few independent modules introduced as described above (Figure 2.b). In the current version of the architecture there are still parts of the old architecture remaining.

3.4 Selected metrics for monitoring the changes

Several metrics are available in the literature for monitoring stability and abstractness of a design. Martin has presented a number of metrics based on the coupling between classes and packages (code categories) (e.g. [16, 4]) as described below. One aspect that is of interest for a package is to what extent it depends on other packages. The fewer other packages it depends on, the more stable it is (does not break due to changes outside the package). Martin [16, 4] defines a metric for efferent coupling (C_e) as the number of classes in a package that depend on classes outside the package. We make an alternative definition of efferent coupling of a code segment as the number of other code segments that it depends on, where a code segment can be a package or a project. This level we found sufficient for our purpose in order to measure how dependent a code segment is of other code segments. It can also be noticed that it is the same definition of efferent coupling as is used in the JDepend metrics tool[2] for analysis of coupling between java packages.

[2] `http://clarkware.com/software/JDepend.html`

Another aspect that is of interest for a package is its responsibility. The more other packages are dependent on it the more responsible it is, and the more responsible it is the more stable it is forced to be. Martin [16, 4] defines a measure of afferent coupling (C_a) as the number of classes outside the package that depend on classes inside the package. In this study we define this metric as the number of code segments outside a code segment that depend on the code segment, where a code segment can be a java package or a project in Eclipse.

We use Martin's [16, 4] definition of instability $I = C_e/(C_a + C_e)$, but with our definition of C_e and C_a. $I = 0$ indicates a maximally stable segment, and I=1 indicates a maximally unstable segment. I.e. instability, I, can be seen as a measure of to what extent changes that are made in other parts of the code affects the code and to what extent it is likely to be changed based on new requirements, etc.

An interesting question to discuss is on what abstraction level the metrics for coupling and instability should be collected on. This can either be between classes, between packages, or between projects in Eclipse. Since the goal of the organization in the study was to get independent "pipes", which are implemented as projects in Eclipse we choose to measure the coupling between projects. However, this means that the metrics can only be collected for the latest version where a division into different projects has been made.

4 Research methodology

The research methodology follows a case study approach (e.g [17]), i.e. it is a flexible research approach [18] where some detailed are left undecided before all data collection is performed. The high-level goal of the research was to understand how to monitor the architectural transformation and to be able to provide objective evidence regarding the positive impact of the suggested changes. The main research questions are:

1. What are the motivations for introducing the changes described in Section 3.3, and what are the experiences of introducing them?
2. What are the experiences of using the set of metrics as described in Section 3.4 for the purpose of monitoring (and keeping) this kind of change?

4.1 Data collection

The data collection steps are outlined in Figure 3. The data was continuously collected during the architectural changes. To the left of Figure 3, the "normal improvement work" of the case company is shown, i.e. how the company goes from an initial version of the architecture to a "final version", i.e. the last version during this case study. The data collection started when the change process had been initiated and some changes were already introduced. Thus, the authors were not involved in the decisions or selecting the goals of the architectural transformations. During the data collection, information that is the basis for

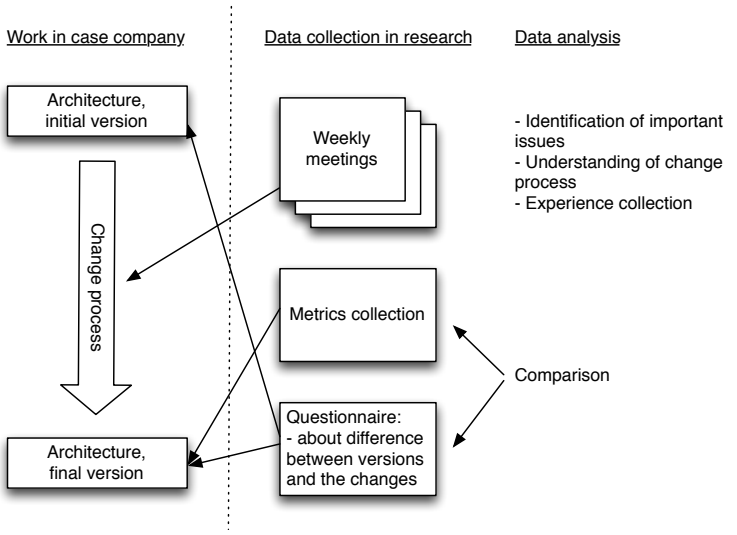

Fig. 3. Main steps of data collection and analysis

understanding the background, the type of changes and the introduced changes as presented in Section 3.1 – Section 3.3 were gathered.

Weekly meetings. The researchers held approximately every week, or at least bi-weekly, meetings with a case company contact person of the project. The objectives of the meetings were to understand what happened in the project and what decisions are taken at different points in time. The meetings were also held in order to understand more about why different decisions were taken. During the meetings questions about the business, architecture, process, and organization were asked. The meetings were informal and held either in the company premises or over skype/telephone. The following list of questions were used to guide the meetings:

- What has happened since the last meeting?
- What important events or decisions have been taken, with respect to i) business, ii) architecture, iii) process, iv) organization?
- What are your plans, with respect to i) business, ii) architecture, iii) process, iv) organization?

Since the meetings were held informal, not all questions were asked every time. However, they were used to ensure that no important dimension was missed but the discussions always covered these aspects. During the meetings, notes were taken by the researchers. No audio recordings were carried out. There were also a few meetings with the person in charge of collecting the metrics at the company where different metrics were discussed. This is further discussed in Section 3.4.

Metrics collection. Metrics were collected using the Eclipse plugin CodePro AnalytiX[3]. After some initial analysis, it became clear that collecting metrics on the class and package levels is unfeasible. Firstly, the main objective of the case company was to have a clear separation between the projects (i.e. the "pipes" in Figure 2). Therefore, package separation was not necessary since this would not show the difference between the projects. Secondly, the metrics are only collected for the latest version, since in the previous versions there was no division into projects in this way.

Questionnaires. A questionnaire was also sent to developers in the Android project. The questions were in most cases formulated as open questions where the participants answers in free-text, while in a few pre-decided answer questions. The questionnaire included the following questions:

1. General questions, e.g. name, experience, and role
2. Characteristics about the team, e.g. collaboration approach, division of tasks
3. Questions concerning the architecture changes, e.g. perceived motivation for change, and observed benefits and drawbacks
4. Questions concerning the development process, with sub-questions about noticed change in product quality, noticed change in how easily the code can be browsed and searched, and perceived change regarding how much code a developer must be able to work with
5. Questions about testing, e.g. how the architectural change affected test-case selection and the number of faults found
6. Question about how changes negatively affect other parts of the system, before and after the architectural change
7. Question about how the amount of duplicate code has changed after the architectural change

Analysis. The metrics analysis was carried out by collecting the metrics on the last available version of the system and analyzing them, see Section 5.1. Qualitative data analysis included summarizing the answers to each major question category.

5 Results

5.1 Metrics collection

Since the study focused on the relations between the different projects, the metrics were collected on the higher abstraction looking at the separation of projects. The results are presented in Table 1. The calculation of C_e was based on the number of the referenced projects. Figure 4 extends the general sketch in Figure 2.b with some more details. The figure consists of a number of parts:

[3] `https://marketplace.eclipse.org/content/codepro-analytix`

Table 1. Metrics results

Project	Referenced projects	C_e	Referencing projects	C_a	I
P1	P2, P4, P6	33	P2, P4, P6	3	0.5
P2	P1, P3	2	P1	1	0.67
P3	–	0	P2, P4, P5, P6	4	0
P4	P1, P3	2	P1	2	0.5
P5	P3	1	–	0	1
P6	P1, P3	2	P1	1	0.67

- P1: This part denotes the "original project", i.e. the architecture according to Figure 2.a. This part has evolved by breaking out some of the functionality of the large architecture in Figure 2.a when developing other parts, and it has been improved in general.
- P3: This part denotes the "core" functionality that is intended to be used by the other projects. It is intended to be stable, which is also reflected in the value of $I = 0$.
- P2, P4, P5, P6: These parts are individual "pipes" which are dependent on P3, but not on each other. Since they are dependent on P3 they are not stable in respect to I.

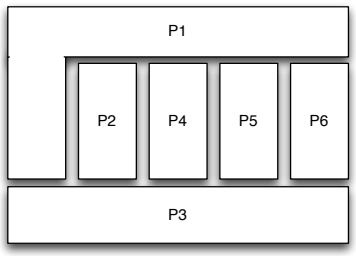

Fig. 4. Architecture of the current version

5.2 Interviews/questionnaires

The questionnaire was answered by five persons with different types of roles. The answers to the questions can be summarized as follows:

1. Two developers, one architect, are customer representative and one manager (core reviewer) answered the survey. Their experience in the project was between 4 and 19 months (median 8) and industrial experience between 1 and 10 years (median 9).

2. The team consisted of 6-7 persons working as a cohesive team and physically in the same location. The development methodology is agile and based on Scrum with work allocation performed by the developers. Developers worked together with the same feature, i.e. they take a feature "together" and work with that until it was done. The feature work is divided into tasks that take 1-8 hours to develop. Sometimes, front-end and back-end development efforts are split.
3. The studied architectural change is considered as an improvement. The understanding of the necessity and the detailed of the changes among the participants is high. The benefits of the changes include that changes will not spread to other parts of the code, it is easier to get an overview, which means that the maintainability is better, and unit test is easier. One participant also highlighted better discussions about the code and the architecture as a benefit. The drawbacks that are seen include that some code may be duplicated since the focus is so much on independent architectural parts. One person also thought that the setup process of the projects were more complicated.
4. The developed code consists fewer faults and it is easier to find what changes have been made to the code.
5. The participants think that it is easier to formulate the right tests. This may however be due to an overall code improvements.
6. Some of the introduced changes may actually negatively impact other parts of the software. The participants think that this problem has decreased in the current version.
7. The participants admitted that the amount of duplicate code has increased, however they found it challenging to accurately estimate the amount of it due to the lack of reliable estimates.

6 Discussion

The architectural changes were considered positive since no participant was clearly negative to the changes, despite additional duplicate code. One explanation could be that the potential negative effects are yet to be discovered. It can also be that this type of project is suitable for this kind of architectural changes and therefore no negative effects occur. The studied code focuses on providing outputs based on input without complicated algorithms, which may be one reason the solution is suitable.

The organizational set up in this case made it possible to move the logic to the server (implemented in decoupled "modules") and focus on thin clients, allowing for decoupled implementation of new features. The clients got thicker than needed, which also may be the reason there is still different teams for iOS and Android, something that can be avoided by having a pure feature oriented organization cross different OS. This fits well the studied context, see B in Figure 1, since different companies are responsible for client applications and the server layers. Moreover, canonical data forced the client code to process data not valid for them. To summarize, other contexts can also benefit from this type of

architectural changes, e.g. in a telecom service provider (corresponding to company A) with customers who have outsourced information systems management to other companies (corresponding to layers here).

The used metrics were suitable for a rather high level of abstraction (between Eclipse projects and not on one specific project). They successfully measured if the architectural changes were achieved. Once in place, these metrics can also serve as continuous verification of the quality of the architecture.

Validity can, for example, be discussed with the respect to construct validity, internal validity, external validity, and reliability, e.g. [17]. Construct validity was strengthened by having a long time contact with weekly meetings, which can reduce the risk of misunderstandings. Internal validity threats in terms of other factors affecting the values of the architectural measures were minimized by studying the whole change and trying to understand what was actually changed, e.g. the way of working was changed somewhat at the same time. However, a risk remains that the changes were made as a result of positive attitude and just because the participants were not happy with the old architecture. Moreover, the study was conducted with a rather small team for an Android application threatening external validity. Thus, further replications are needed for this type of projects, and if other types of projects are consider additional research is probably needed. Finally, we address reliability threats by having regular meetings with the team and recording all research steps in the case study protocol.

7 Conclusions

Concerning the motivation for introducing the changes described in Section 3.3, the main motivations are that it can decrease the risk of making changes that negatively affect other parts of the system, and that it makes it possible to divide the work between different people in natural way. There was also a need to refactor the architecture since the software has grown and future anticipated requirements include further growth of the software and increased differentiation between customer segments. The experiences of introducing the changes are in general positive, which indicates that it in this case was correct to prioritize modularization over the avoidance of code duplication. If the same change is made in another project, conclusions from this study may be relevant if the project is similar in e.g. size and the type of code (e.g. with respect to how complicated algorithms etc. it involves).

One conclusion that can be drawn concerns the level of abstraction of the metrics. Even if this is only one study it seems like it is reasonable to study the metric on a high level of abstraction, since the focus is on the whole project and not on a specific part when this change is made.

Acknowledgement

This work was funded by Vinnova in the ITEA2 project 12018 SCALARE.

References

1. Bergman, B., Klevsjö, B.: Quality, from Customer Needs to Customer Satisfaction. third edn. Studentlitteratur (2010)
2. Fenton, N., Pfleeger, S.L.: Software Metrics, a Rigorous and Practical Approach. second edn. PWS Publishing Company (1997)
3. Betz, S., Wohlin, C.: Alignment of business, architecture, process, and organisation in a software development context. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM). (2012) 239–242
4. Martin, R.C.: Agile Software Development Principles, Patterns, and Practices. second edn. Prentice-Hall (2003)
5. Chikofsky, E., Cross, J.H., I.: Reverse engineering and design recovery: a taxonomy. IEEE Software **7**(1) (Jan 1990) 13–17
6. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2) (2004) 126–139
7. Seriai, A., Bastide, G., Oussalah, M.: Transformation of centralized software components into distributed ones by code refactoring. In: 6th Int. Conf. on Distributed Applications and Interoperable Systems. (2006) 332 – 346
8. Eloff, J.: Software restructuring: Implementing a code abstraction transformation. In: Proceedings of the 2002 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology. SAICSIT '02, Republic of South Africa (2002) 83–92
9. Arnold, R.: Software restructuring. IEEE Software **77**(4) (Apr 1989) 607–617
10. Ivkovic, I., Kontogiannis, K.: A framework for software architecture refactoring using model transformations and semantic annotations. In: 10th European Conf. on Software Maintenance and Reengineering (CSMR). (March 2006)
11. Rizvi, S., Khanam, Z.: A methodology for refactoring legacy code. In: International Conference on Electronics Computer Technology (ICECT 2011). (2011) 198 – 200
12. Aversano, L., Molfetta, M., Tortorella, M.: Evaluating architecture stability of software projects. In: Working Conf. on Reverse Eng. (2013) 417 – 424
13. Tonu, S.A., Ashkan, A., Tahvildari, L.: Evaluating architectural stability using a metric-based approach. In: Proceedings of the European Conference on Software Maintenance and Reengineering, (CSMR), Bari, Italy (2006) 261 – 270
14. Bahsoon, R., Emmerich, W.: Evaluating architectural stability with real options theory. In: IEEE International Conference on Software Maintenance, ICSM, Chicago, IL, United states (2004) 443 – 447
15. Figueiredo, E., Cacho, N., Garcia, A., Ferrari, F., Khan, S., Sant'Anna, C., Monteiro, M., Soares, S., Filho, F.C., Kulesza, U., Dantas, F.: Evolving software product lines with aspects: An empirical study on design stability. In: Int. Conference on Software Engineering, Leipzig, Germany (2008) 261 – 270
16. Martin, R.C.: OO design quality metrics. Technical report, Object Mentor (1994)
17. Runeson, P., Höst, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering - Guidelines and Examples. Wiley (2012)
18. Robson, C.: Real World Research. second edn. Blackwell (2002)