2014 9th International Conference on the Quality of Information and Communications Technology

A comparative study on the effectiveness of patterns in software libraries and standalone applications

Panagiotis Sfetsos[1], Apostolos Ampatzoglou[2,3], Alexander Chatzigeorgiou[4], Ignatios Deligiannis[1], Ioannis Stamelos[2]

[1] *Department of Information Technology, AlexanderTechnological Educational Institute Thessaloniki, Greece*
[2] *Department of Informatics, Aristotle University, Thessaloniki, Greece*
[3] *Department of Computer Science, University of Groningen, The Netherlands*
[4] *Department of Applied Informatics, University of Macedonia, Greece*

***Abstract* — The existence of design pattern instances is often regarded as an indication of elaborate software design, since patterns have been reported in many studies as techniques that improve software quality properties. Driven by the widespread belief that software libraries excel in terms of design quality compared to standalone applications, this study investigates first whether this claim is confirmed and second whether the improved quality can be attributed to the use of patterns. In particular we examine: (a) whether libraries exhibit improved design quality in terms of metrics compared to standalone applications, (b) the intensity of use of design patterns in the two software categories and (c) whether there is any correlation of design patterns usage and design quality at system level. The results of the study suggest that, some of the quality properties are improved in library software although no significant difference in the use of patterns have been observed. Moreover, there is an important number of GoF design patterns that appears to be correlated to software quality metrics.**

Keywords – Design Patterns; Design Quality; Software Libraries; Standalone applications

I. INTRODUCTION

Software systems evolve by adding new features and modifying existing functionality over time. Through evolution, the underlying structure of software artifacts gradually degrades, leading to a substantial reduction of its understandability and maintainability. Refactoring is considered to be a prominent technique for solving the abovementioned problem. Software refactoring is a disciplined process of improving the design/code quality of existing systems, by changing its internal structure, without affecting its external behavior [6 and 11].

Another way of addressing the declining quality of software is to employ design patterns, i.e. well-grounded solutions to common, recurring problems in software design. In the mid-90s the Gang-of-Four (GoF) [7], defined a catalogue of design patterns as *"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"*. These solutions *"have developed and evolved over time, in a succinct and easily applied form"*. The use of design patterns can provide reusable solutions and improve software qualities, such as flexibility, understandability and maintainability [6 and 7]. In [6], Fowler suggests that there is a natural and close relation between patterns and refactorings: patterns describe the design situation where you want to be, whereas refactorings represent the possible ways to get there [8]. This statement conforms to the observation made by the Gang-of-Four: *"Our design patterns capture many of the structures that result from refactoring. Design patterns thus provide targets for refactorings* [7]*"*. To this end, Kerievsky introduced the concept of *Refactoring to patterns* [8] by expounding the interplay of design patterns and refactorings, in a detailed and explanatory way, focusing on the benefits and liabilities of each refactoring to pattern.

Among software practitioners and researchers there is a widespread belief that software libraries, which are accessible through well-defined Application Programming Interfaces (API) excel in terms of design quality. The improved quality is usually related to the need for continuous evolution of libraries calling for high-levels of flexibility and extendibility without however breaking the compatibility of clients. Driven by this premise in this paper we compare the design quality between software libraries and standalone applications. Moreover, we aim at quantitatively investigating if possible differences in the design quality can be attributed to the use of GoF design patterns. Moreover, we attempt to achieve these goals, by empirically and quantitatively (using software metrics) examining differences between software libraries and standalone applications, with respect to: (a) design quality, (b) use intensity of GoF design patterns, and (c) the effect of patterns instances on quality attributes.

The paper is structured in eight sections: Section II provides an overview of software libraries. Section III discusses design patterns in relation to quality and software libraries. In Section IV we describe the case study design. In Section V we present the results of data analysis, and discuss them in section VI. Finally, we present threats to validity in Section VII, and conclusions and future work in Section VIII.

II. SOFTWARE LIBRARIES / APIS

The design of software libraries is generally believed to

978-1-4799-6133-7/14 $31.00 © 2014 IEEE
DOI 10.1109/QUATIC.2014.26

be of a high quality since certain design guidelines have to be followed, due to the need for continuous evolution. According to Tulach [14], designing a shared library is a far more complicated task than building closed application software because of the numerous clients depending on the library's interface. This dependency necessitates the consideration for backward compatibility so that evolution is performed in a way that does not disturb clients. These requirements for an Application Programming Interface (API) impose a strict design and development style, making the conformance to design rules and the use of best practices in software design such as design patterns, even more important. Consequently, libraries are characterized by particular properties [14]:

- *Evolution* is one of the most important aspects to consider when architecting a library and its interface.
- The *behavior* of an API is the most important part of the API contract: Only if the behavior of a component *remains unchanged,* can its users cluelessly [14] replace versions of the component in their applications. Clients should have confidence that functionality will not be compromised by upgrading to a newer version of the library.
- Programmer productivity can be boosted just by having a working knowledge of an API and knowing where to find the appropriate documentation. In other words, coding can be largely facilitated by learning a tip of an iceberg. The cornerstone of this architectural approach is the *abstraction* that wraps around every library or framework. This abstraction—the API— hides all the complexities. Thus, as Tulach states [14], the more selectively clueless you are, the more reliable the system is.

These features of libraries focusing on the cluelessness model allow development teams to concentrate on the most important aspect of their work: the actual logic of their own application. In this perspective, the study of APIs can offer valuable guidance on how to improve the existing design of any software project. Driven by this viewpoints we examine the difference between APIs and standalone systems with respect to the use of design patterns and the effect that patterns have on the design quality of the underlying software.

III. Design Patterns

According to the definition of refactorings by Mens and Tourwe, in [9], refactorings aim at improving product quality of software systems. Thus, a prerequisite for GoF design patterns to be eligible as refactoring activities is to prove the hypothesis that pattern application has a positive effect on system quality. Thus, in section III.A, we present a brief overview of the literature findings on the effect of GoF design patterns on software quality. In addition to that, in section III.B, we present related work that investigates the use of GoF design patterns in software libraries and APIs.

A. Design Patterns and Quality

In [1], the authors suggest that the most active field in pattern research is the investigation of the effect of GoF design patterns on software quality. Thus, although, in the original introduction of the GoF pattern catalogue [7], design patterns have not been linked to specific quality attributes, researchers tend to believe that they affect it. However, design pattern are expecting to have a local effect on software quality, by improving the structure of a limited amount of classes. Therefore, an interesting and open research question that arises and which might be valuable to agile development teams is: *does the extensive use of GoF design patterns can have a system wide effect on quality*?

B. Design Patterns and Software Libraries

The literature that concerns the use and the evaluation of GoF design patterns on software libraries, is quite limited. In [10] the author describes the design principles that he found useful, while developing a real-time imaging framework in Java. The results suggest that in such a framework five patterns are applicable, namely *Singleton*, *Abstract Factory*, *Observer*, *Façade* and *Strategy*. Apart from the exact description of how each design pattern has been instantiated, the author presents the reusability benefits that these patterns offered to the framework.

Concerning the effect of GoF design patterns on the quality of software libraries, Ellis et al. suggest that one of the most interesting quality attributes to investigate, is usability, in the sense that in order for APIs to be valuable to their clients, they should be highly usable [4]. Additionally, in the same study Ellis et al. investigate the usability that *Factory* patterns offers to APIs, compared to the use of constructors, through an experiment. The results of the study suggest that the time needed to use an API, based on *Factory* patterns, is statistically significantly lower than the time needed to use an API, based on alternative design solutions [4].

IV. Case Study

In order to compare the effectiveness of GoF design pattern instances in software libraries and standalone applications, we performed a case study, i.e. an observational empirical method that is used for monitoring projects and activities in a real-life context [12], on 26 Java open source software (OSS) projects. The case study of this paper has been designed and is presented according to the guidelines of Runeson et al. [12]. In this section, we present: (a) research objectives and research questions, (b) cases and units of analysis, (c) data collection methods, and (d) data analysis methods.

Research Objectives & Research Questions: The goal of this study is to investigate differences among software libraries and standalone applications in terms of: (a) the levels of design quality, (b) the use intensity of design patterns, and (c) the effect of GoF design pattern on design quality. Intuitively, since software libraries are expected to be more heavily reused, they are expected to exhibit better design quality than standalone software, and more frequently employ established techniques, such as GoF design patterns, rather than standalone applications.

According to the previously stated goal we extracted and formulated three research questions that will guide the case study design and the reporting of the results:

RQ_1: Are there differences between software libraries and standalone applications, in terms of design quality?

RQ_2: Are there differences between software libraries and standalone applications, in the use intensity of GoF design patterns?

RQ_3: Are there differences between software libraries and standalone applications, in the effect of GoF design patterns instances on quality attributes?

Cases and Units of Analysis: To answer the above mentioned questions we performed a holistic multiple-case study, where the cases and units of analysis are open source projects. We note that according to Runeson et al., a case study is holistic, if from every case, we extract only one unit of analysis [12], as opposed to embedded case studies, in which we extract multiple units of analysis from one case. As units of analysis we used the 26 OSS projects explored in [3], for similar reasons. More specifically, we analyzed 13 open source standalone applications, and 13 open source software libraries, all written in java. For pattern detection we used the tool created by Tsantalis et al. [13].

Data Collection: The dataset that was created after selecting the cases, consisted of 19 variables, as follows:

[A_1] *software name*

[A_2-A_{12}] *design pattern use intensity*. We recorded one variable for each type of design pattern that was explored (Factory Method, Proxy, Prototype, Adapter, Singleton, Composite, Decorator, Observer, Template Method, Visitor and State – Strategy). These variables have been normalized over the number of classes of a project, in order to filter out the confounding factor *project size*. Thus, the variables are calculated as the fraction of number of classes, divided by the number of pattern instances. Thus, the lower the value is, the more intensive the use of GoF design patterns is.

[A_{13}-A_{18}] *metric scores on design quality*. We adopted the measures on design quality from the QMOOD suite [2]. Thus, we recorded one variable for each high-level quality attribute (Functionality, Effectiveness, Extendibility, Reusability, Understandability, Flexibility). We note that in all measures, the higher the metric score is, the higher the levels of design quality become.

[A_{19}] *type* of software (standalone/library).

Data Analysis: In the data analysis phase of our case study we have employed descriptive statistics (mean values and standard deviation, Spearman correlation), graphs (3D area charts) and hypothesis testing (Mann Whitney U-test) [5].

V. Results

In this section we present case study results, organized by research question. We note, that this section only deals with presenting results, whereas a discussion of findings (interpretation of results, and implications for researchers and for practitioners), will be provided in Section VI.

RQ_1: Differences in design quality

Regarding the differences between software libraries and standalone applications, in terms of design quality, the descriptive results are presented in Table I. Next, in order to investigate if the results on our sample can be generalized to a larger population, we performed a Mann-Whitney U-test, i.e. a non-parametric test for testing differences between mean values (see Figure 1).

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of Reusability is the same across categories of type.	Independent-Samples Mann-Whitney U Test	,081[1]	Retain the null hypothesis.
2	The distribution of Flexibility is the same across categories of type.	Independent-Samples Mann-Whitney U Test	,113[1]	Retain the null hypothesis.
3	The distribution of Understandability is the same across categories of type.	Independent-Samples Mann-Whitney U Test	,113[1]	Retain the null hypothesis.
4	The distribution of Functionality is the same across categories of type.	Independent-Samples Mann-Whitney U Test	,014[1]	Reject the null hypothesis.
5	The distribution of Extendibility is the same across categories of type.	Independent-Samples Mann-Whitney U Test	,000[1]	Reject the null hypothesis.
6	The distribution of Effectiveness is the same across categories of type.	Independent-Samples Mann-Whitney U Test	,002[1]	Reject the null hypothesis.

Fig. 1. Hypothesis testing for RQ_2

TABLE I. DESCRIPTIVE STATISTICS FOR RQ_1

Quality Attribute	Type	Mean
Reusability	*Library*	4.404
	Standalone	3.455
Flexibility	*Library*	0.901
	Standalone	0.524
Understandability	*Library*	-4.142
	Standalone	-3.369
Functionality	*Library*	2.281
	Standalone	1.618
Extendibility	*Library*	1.243
	Standalone	0.476
Effectiveness	*Library*	0.586
	Standalone	0.362

From both Table I and Figure 1, it can be concluded that there is evidence for a difference in the levels of design quality among software libraries and standalone applications for some quality attributes (functionality, extendibility and effectiveness).

RQ_2: Differences in design pattern use intensity

In order to investigate if the aforementioned differences can be attributed to the use of patterns, we first explore if there are differences in the use intensity of GoF design patterns between software libraries and standalone applications (see Table II).

TABLE II. DESCRIPTIVE STATISTICS FOR RQ_2

Pattern	Type	Mean
Factory Method	*Library*	69.2583
	Standalone	109.7500
Prototype	*Library*	45.3519
	Standalone	82.3611
Singleton	*Library*	35.0513
	Standalone	53.8544
Adapter	*Library*	17.6733
	Standalone	32.9210
Decorator	*Library*	16.9896
	Standalone	236.3125
Proxy	*Library*	137.0000
	Standalone	325.3333
Observer	*Library*	87.9167
	Standalone	212.9714
Template Method	*Library*	92.6667
	Standalone	49.8021
State – Strategy	*Library*	15.7985
	Standalone	21.1458

As it is observed, all design pattern types, except from Template Method, are more frequently occurring in software libraries than standalone applications. To examine whether there are significant differences between the mean values presented in Table II, we performed a Mann-Whitney U-test. The results suggested that the only differences, which are statistically significant at the 0.05 level concern Decorator (sig: 0.01), and at the 0.10 level concern Adapter (sig: 0.06).

RQ_3: Differences in the effect of design patterns on design quality

Concerning the investigation of differences in the strength of possible correlations between the number of employed design pattern instances and the design quality metric values, with respect to the software type (software library or standalone application), the results are summarized in Table III (due to space limitations, in Table III we only present statistically significant correlations at the 0.05 level). For example, the correlation coefficient for the first row of Table III implies that the use intensity of the Adapter pattern has a positive effect on reusability (the negative sign is due to the fact that the intensity of pattern use is obtained as the fraction of number of classes, divided by the number of pattern instances). Moreover, the effect is stronger for standalone applications.

TABLE III. DESCRIPTIVE STATISTICS FOR RQ_3

Pattern:: QA	Type	Correlation Coefficient (sig)
Adapter :: Reusability	*Library*	-0.167 (0.66)
	Standalone	-0.883 (0.03)
Adapter :: Flexibility	*Library*	-0.467 (0.20)
	Standalone	-0.883 (0.05)
Adapter :: Understandability	*Library*	0.200 (0.60)
	Standalone	0.733 (0.25)
Adapter :: Functionality	*Library*	-0.250 (0.51)
	Standalone	-0.867 (0.03)
State-Strategy :: Flexibility	*Library*	-0.663 (0.04)
	Standalone	-0.697 (0.02)
State-Strategy :: Understandability	*Library*	0.867 (0.00)
	Standalone	0.018 (0.96)
State-Strategy :: Functionality	*Library*	-0.883 (0.00)
	Standalone	-0.358 (0.31)
Singleton :: Flexibility	*Library*	-0.664 (0.02)
	Standalone	0.191 (0.57)

One of the most interesting observations from Table III, is that actually only three design patterns, namely *State-Strategy, Adapter* and *Singleton* are correlated to design quality at system level (probably because these are the most heavily used design patterns and therefore, their aggregate effect is more evident at system level). However, the quality attributes, to which these design patterns have effect on, are in principal not among those that differ between libraries and standalone applications. Thus, such differences, at least from the results of this study, cannot be attributed to the use of design patterns. However, other interesting findings (although orthogonal to the original research question), can be derived from the exploration of the 3D area chart of Figure 2, by contrasting the differences for libraries and standalone software systems.

These plots can be interpreted as follows, considering for example the top right chart concerning the effect of the Adapter pattern on reusability. As the use of the Adapter pattern becomes more frequent (the value 20 on the Adapter axis indicates the occurrence of one Adapter every 20 classes) in the case of standalone applications, reusability increases. On the other hand, this observation is not so evident for libraries.

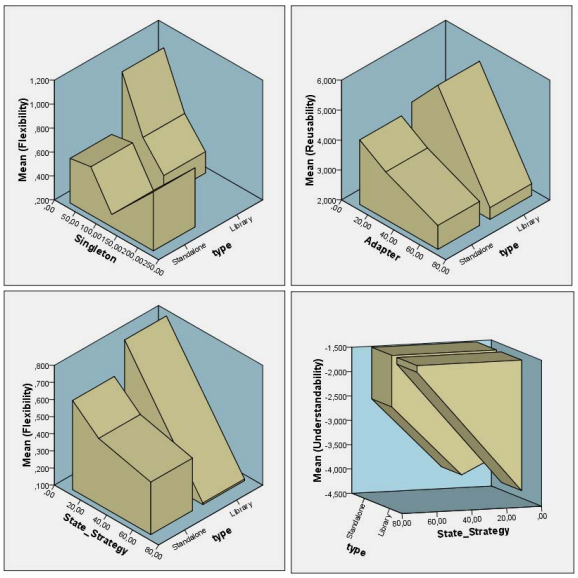

Fig. 2. Descriptive Graphs for RQ_3 (3D Chart Areas)

VI. Discussion

In this section we discuss the results of the case study, based on the three research questions, and organized through two different perspectives: (a) interpretation of results, (b) implications for researchers and practitioners.

A. Interpretation of results

Although the statistical significance is relatively low, the results weakly confirm the intuition that design libraries excel both in terms of design quality (i.e. Functionality, Extendibility, Effectiveness), as well as in terms of the number of employed design patterns. The fact that the investigated three qualities exhibit higher values for libraries is rather reasonable. These measures depend (positively) on abstraction which is higher for APIs, negatively on coupling, which for the independent pieces of functionality in a library is usually low, and positively on the use of inheritance which by definition is extensive in libraries. The low statistical significance regarding the use of patterns can possibly be attributed to the small number of pattern instances in the examined systems.

The results on the effect of patterns on the design quality are rather mixed: for example, the effect of the Adapter design pattern on reusability, flexibility, understandability and functionality is more intense for standalone applications despite the lower number of Adapter instances. On the other hand, the impact of State-Strategy pattern on understandability and functionality is higher for libraries. Although no safe explanations can be derived for these differences, the results imply that patterns are exploited in a different manner between software libraries and standalone applications. In any case, further empirical evidence should be analyzed with respect to the particular design decisions, which are responsible for the improved design quality of libraries.

B. Implications to researchers and practitioners

Concerning implications for researchers and practitioners, the results can be helpful for providing: (a) pointers to interesting areas for future work, and (b) advices on how detailed-design decisions could benefit from considering GoF design patterns, as follows:

- researchers could further investigate the cause of the observed differences in the instantiation of the same pattern between software libraries and standalone applications
- researchers could further investigate the different effect of GoF design patterns between software libraries and standalone application, in the class level, rather than a system-wide level. It is expected that locally the differences will be more evident
- agile software developers, based on the quality attributes that they are interested in, can select GoF design patterns which might have an effect on system level quality attributes, if they are heavily applied, e.g. State/Strategy improves flexibility
- the developers of standalone applications should seek opportunities to apply design patterns more systematically (at this point they don't seem to use a lot pattern instances), by studying design pattern instantiation in well-known libraries

VII. Threats to Validity

In study we classified threats to validity based on [12] and [15], distinguishing them among four types. In this section the validity threats are presented, accompanied with the approaches that we followed to mitigate them.

Threats to construct validity: These threats concern the design of the study and especially the identification of the correct measures for the concepts being studied. The case study was designed and contacted in separate stages, strictly following the detailed case study protocol and the approved methodology [12]. We used multiple sources of evidence, for the data collection process and the QMOOD metrics suite, which is a proper and empirically validated metrics suite for Object Oriented Designs. The 26 OSS projects explored (13 software libraries and 13 standalone applications) were collected from the most representative software areas such as Communications, Graphics, Audio & Video and Business & Enterprise.

Threats to internal validity: These threats concern the identification of cause-and-effect relationships and the evidence of causality. In our study, we consider as threats to internal validity those factors that may cause interferences regarding the relationships that we are trying to investigate [15]. As in [3], a threat is concerned with the extent to which the design quality of the examined systems has been accurately captured by the selected metrics. We believe that the QMOOD metrics suite we chose, predicts well the quality of an OO software design, minimizing this threat. Another concern is that software systems from different domains might have fundamental differences in inherent complexity, affecting the design and implementation. In our study, the categorization of software projects and their domains were varied randomly.

Threats to external validity: As threats to external validity, we consider those factors that limit the possibility to generalize the findings beyond the sample of the study. Obviously, a different set of software projects could lead to different results. This kind of threat is always valid in an empirical study when the number of systems is limited and the criticism is related to possible differences between the projects that have been selected for analysis and other kinds of projects.

Threats to reliability: This aspect of validity is concerned with the extent by which the data and the analysis are dependent on the particular researchers. Reliability is demonstrating that the operations of a study, such as the data collection procedures and analysis, can be repeated, with the same results. We used a case study protocol, documenting the procedures that have been followed, as proposed in [12]. Moreover, we developed a case study database storing the collected data. With these operational steps we believe that an external auditor could in principle repeat the procedures and arrive at the same findings and conclusions.

VIII. Conclusions – Future Work

Design patterns are often regarded as an indication of elaborate software design, improving several aspects of quality. Driven by the assumption that software libraries exhibit improved design quality compared to standalone application, due to their need for continuous evolution without disturbing their clients, we performed an empirical study comparing the two kinds of software. In particular, we investigated possible differences between software libraries and standalone applications, with respect to: (a) their design quality, (b) the use of GoF design patterns, and (c) the effect of GoF design patterns instances on quality attributes. The study is based on 26 open source projects, equally divided between libraries and standalone applications.

According to the results: (a) the functionality, extendibility and effectiveness of software libraries are significantly higher than those of standalone applications, (b) software libraries employ more design pattern instances than standalone applications, but these results are weak, and not generalizable to a wider population, and (c) some pattern instances appear to have the stronger effects on system-wide quality attributes. The aforementioned results led us to the identification of future work directions and interesting lessons learned for practitioners.

IX. Acknowledgement

The research work is co-founded by the European Social Fund and National Resources, ESPA 2007-2013, EDULLL, “Archimedes III” program.

X. References

[1] A. Ampatzoglou, S. Charalampidou and I. Stamelos, “Research state of the art on GoF design patterns: A mapping study”, *Journal of Systems and Software*, Elsevier, 86 (7), pp. 1945-1964, July 2013.

[2] J. Bansiya and C. Davis, “A hierarchical model for object-oriented design quality assessment”, *Transaction on Software Engineering*, IEEE Computer Society, 28 (1), pp. 4–17, January 2002.

[3] A. Chatzigeorgiou and E. Stiakakis. “Benchmarking library and application software with Data Envelopment Analysis”, *Software Quality Journal*, Springer, 19 (3), pp. 553-578, September 2011.

[4] B. Ellis, J. Stylos and B. Myers, “The Factory Pattern in API Design: A Usability Evaluation”, *29th International Conference on Software Engineering* (ICSE' 07), IEEE Computer Society, pp. 302-312, 20 - 26 May 2007.

[5] A. Field, “Discovering Statistics Using SPSS”, *SAGE Publications*, 2009.

[6] M. Fowler, “Refactoring: Improving the Design of Existing Code”, *Addison Wesley Longman*, 1999.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, *Addison Wesley*, 1994.

[8] J. Kerievsky. “Refactoring to Patterns”, *Addison Wesley*, 2004.

[9] T. Mens and T. Tourwe, “A Survey of Software Refactoring”, *Transactions on Software Engineering*, IEEE Computer Society, 30 (2), pp. 126–139, 2004.

[10] C. J. Neill, “Leveraging object-orientation for real-time imaging systems”, *Real-Time Imaging*, Elsevier, 9 (6), pp. 423–432, June 2003.

[11] W. Opdyke, "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks," *PhD Thesis*, 1992.

[12] P. Runeson, M. Host, A. Rainer and B. Regnell, “Case Study Research in Software Engineering: Guidelines and Examples”, *John Wiley & Sons*, 2012.

[13] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, “Design pattern detection using similarity scoring”. *Transactions on Software Engineering,* IEEE Computer Society, 32 (11), pp. 896-909, November 2006.

[14] J. Tulach, “Practical API Design: Confessions of a Java Framework Architect”, *Apress*, 2012.

[15] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslen, “Experimentation in software engineering: An introduction”, *Kluwer*, 2000.