Analyzing the Impact of Refactoring on Bad Smells

Cleiton Tavares, Mariza Bigonha, Eduardo Figueiredo
Federal University of Minas Gerais
Belo Horizonte, Brazil
cleiton.silva,mariza,figueiredo@dcc.ufmg.br

ABSTRACT

Refactoring aims to remove bad smells and increase software maintainability by improving the software structure without changing its behavior. However, some studies show that refactoring tools may introduce new bad smells into the source code, but to the best of our knowledge, we have not been able to find a complete catalog that states the bad smells introduced from refactoring. To bridge this gap, this paper goal is to evaluate the impacts of refactoring on the detection of bad smells in open-source Java systems. Hence, we want to know if and when the automated refactoring removes or introduces bad smells.

CCS CONCEPTS

• **Software and its engineering** → **Empirical software validation**; **Software defect analysis**; **Software evolution**.

KEYWORDS

Bad Smell, Refactoring, Impacts of Refactoring

ACM Reference Format:
Cleiton Tavares, Mariza Bigonha, Eduardo Figueiredo. 2020. Analyzing the Impact of Refactoring on Bad Smells. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3422392.3422408

1 INTRODUCTION

A software system requires great effort from developers to maintain its source code [17]. For the system to continue meeting its requirements and to evolve or adapt to new technologies, it must follow good development practices to facilitate the team's work. Thus, it is expected to decrease the number of bad smells, which are poor code implementations that indicate the need to be refactored [3, 9, 18]. Refactoring is a strategy used to increase the maintainability of the code by changing its internal structure without changing its behavior, it is highly recommended to solve bad smells [8].

We may find in the literature numerous tools for detecting bad smells [4, 5, 10, 14] and refactoring [13, 16]. However, studies show that refactoring often does not solve the bad smells in the source code [1, 20]. Such a problem may lead the developer not to trust in automated refactoring since there are high chances of not completely removing the bad smells present in the code.

Even though there are some indications that refactoring may not remove bad smells, there is no complete catalog that shows which ones may even introduce bad smells. A catalog containing information about which bad smells may be introduced by refactoring may assist developers in performing refactoring, whether manual or automated. Once they have this information, developers may take care and perform the most efficient and robust refactoring process to avoid introducing new bad smells.

Therefore, the goal of this paper is to conduct empirical research to assess the impacts of automated refactoring on detecting bad smells. To assess this impact, we selected seven open-source Java systems available at Qualitas Corpus. We analyzed the impacts of three different types of refactorings on ten different bad smells proposed by Fowler et al. [8].

To perform refactoring, we use JDeodorant [6] that makes it possible to perform refactoring in an automated way. Refactorings of the type Replace Type Code with State/Strategy and Replace Conditional with Polymorphism [8] are treated together by JDeodorant, which for documentation criteria we refer to them as Replace Refactoring. Therefore, for the remaining of this paper, we treat the refactorings operation analyzed as being of two type: Move Method and Replace Refactorings.

We applied a total of 80 refactorings provide by JDeodorant. By analyzing the results, we found that refactoring impacts different types of bad smells. As expected, the refactorings sometimes solve a bad smell. Surprisingly, we also found cases where refactorings may introduce new bad smells. Our main contributions are:

- a comparative study that prioritizes detection performed by five bad smell tools
- a catalog that presents which bad smells are introduced or solved by refactoring
- results of an evaluation exhibiting the impact of refactoring on bad smells

2 RESEARCH METHOD

This section presents the research method adopted in our study.

2.1 Goal and Research Questions

This study goal is to evaluate the impact of refactoring on bad smells. Research questions (RQs) are defined as follows.

RQ1 What are the impacts of refactoring on bad smells?
RQ1.1 Does automated refactoring remove bad smells?
RQ1.2 Does automated refactoring introduce bad smells?

To answer RQ1, we used an automated refactoring tool that created a new refactored version from each system's original version.

After creating the new refactored version, we used bad smell detection tools in the original and refactored versions. We then assess the impact of refactoring on the bad smells, making it possible to answer RQ1.1 and RQ1.2.

2.2 Research Phases

In this study, we define three phases. The description of each phase is presented as follows.

Phase 1 - Selection of Systems. To conduct our research, we initially selected a set of seven systems to compose the study sample. We prioritize systems available through the Qualitas Corpus[1] [19], which contains a curated collection of open-source Java software systems. We focus our research on these seven different systems: Checkstyle-5.6, Commons-codec, Commons-io, Commons-logging, JHotDraw-7.5.1, Quartz-1.8.3, and Squirrel_sql-3.1.2.

Phase 2 - Selection of Tools. This research relies on tools to automate refactoring and bad smell detection. We selected five tools for bad smell detection: Decor[2] [11], Designite[3] [15], JDeodorant[4] [6], JSpIRIT[5] [21], and Organic[6] [12].

Phase 3 - Selection of Bad Smells and Refactorings. We focused on a sample composed of ten bad smells and two refactorings available in the Fowler et al. [8] catalog. We choose this catalog because it is the most completed one. Data Class (DC), Feature Envy (FE), Large Class (LC), Lazy Class (ZC), Long Method (LM), Long Parameter List (LP), Message Chains (MC), Refused Bequest (RB), Shotgun Surgery (SS), and Speculative Generality (SG) are the bad smells evaluated and detected by the five different tools. We selected two types of refactorings: Move Method and Replace Refactoring. JDeodorant supports the automated refactoring.

2.3 Assessment of Refactoring Impact

We choose to create a refactored version of each system with JDeodorant. Figure 1 shows the seven steps followed to assess how refactoring impacts on bad smells, described as follows.

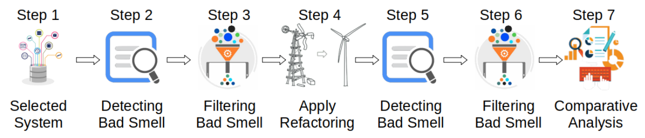

Figure 1: Steps

In Step 1, we selected the system to evaluate. In Step 2, we compute the results identified by the five bad smell detection tools in the original version of the system. In Step 3, we filter the results returned by the detection tools to find only the ones that are the focus of this study.

In Step 4, we select a refactoring strategy to be applied. For this step, we carry out all refactoring suggestions provided by JDeodorant, aiming to use the entirety tool support. In Step 5, we compute the results identified by the five bad smell detection tools in the refactored version of the system. In Step 6, we filter the results returned by the tools to find only the ten bad smells that are the focus of this study.

In Step 7, we performed a comparative analysis of the results obtained from steps 3 and 6. By performing this comparison, we analyze the impacts of automated refactoring on bad smells. For instance, we may see if the automated refactoring removed bad smells or introduced new ones. Moreover, we may list the bad smells removed and introduced by a particular refactoring.

We performed Step 1 seven times; i.e., for the seven systems used in this research. For each execution of Step 1, steps 2 and 3 performed five executions, representing the identification provided by the five tools used to detect bad smells in the original system. For each execution of Step 1, Step 4 performed two executions, representing the two refactoring strategies.

For each execution of Step 4, steps 5 and 6 performed five executions. Steps 5 and 6 represent the identification of the five tools used to detect the bad smells in the refactored system. For Step 7, we carry out the comparative analysis for each applied refactoring taking into account the detection of bad smells performed by each tool in the original and refactored versions of the system.

3 RESULTS

We analyzed individually the impact of two different types of refactoring on ten types of bad smells evaluated in seven systems from Qualitas Corpus. Table 1 presents these bad smells and the five different tools used to detect them. The first column of Table 1 shows the names of the bad smells analyzed and the tools used to detect them. The next seven columns show the detection of bad smells by tools for each system analyzed.

We may observe in Table 1 several zero (0) values, representing three different cases presented by the five tools (discussed below). There is no case of those analyzed in which two or more tools detected the same number of bad smells, except in the non-detection of bad smell.

Three cases of zero values. In the first case, represented by Decor, the detection is performed and presented the number of bad smells as zero for the analyzed system. In the second case, represented by JDeodorant, the tool did not present a result for a bad smell. We considered this situation might indicate either no bad smell in the system or an internal error in the tool that failed to analyze the system. Both situations were documented the same since the JDeodorant log is not easily accessible to the end-user; we decided not to investigate the reason for this occurrence further. Finally, in the third case, represented by Designite, JSpIRIT, and Organic, a complete list of all bad smells detected is returned. We consider that the absence of a specific bad smell represents the non-existence of it.

Same bad smell detection. We did not compute the intersection of the detection results for two reasons: (i) there was no case where two or more tools detected the same number of bad smells, except in the non-detection; (ii) fluctuation in the number of bad smells of the same type detected by different tools. For this reason, we analyzed all data individually, not prioritizing any tool.

[1]http://qualitascorpus.com/
[2]https://github.com/ptidejteam/SmellDetectionCaller
[3]https://github.com/tushartushar/DesigniteJava
[4]https://github.com/tsantalis/JDeodorant
[5]https://sites.google.com/site/santiagoavidal/projects/jspirit
[6]https://github.com/opus-research/organic

Table 1: Original Detection

Bad Smell	S1	S2	S3	S4	S5	S6	S7
DC[T4 | T5]	0 | 25	3 | 0	0 | 2	1 | 2	22 | 52	5 | 10	0 | 8
FE[T3 | T4]	18 | 275	15 | 102	3 | 0	0 | 38	13 | 528	7 | 195	7 | 47
LC[T1]	0	0	0	0	0	0	0
ZC[T1 | T5]	3 | 260	4 | 36	0 | 63	1 | 18	30 | 176	4 | 67	1 | 17
LM[T1 | T2 | T3]	123 | 4 | 243	27 | 13 | 117	0 | 6 | 74	19 | 1 | 53	126 | 37 | 0	45 | 11 | 0	12 | 1 | 99
LP[T1 | T2]	12 | 32	6 | 9	0 | 29	0 | 0	70 | 155	17 | 56	1 | 10
MC[T1]	0	1	0	0	40	0	1
RB[T1 | T4 | T5]	52 | 51 | 18	0 | 11 | 4	0 | 0 | 4	6 | 3 | 5	146 | 100 | 16	21 | 18 | 1	0 | 0 | 0
SS[T4]	39	1	0	9	167	49	2
SG[T1 | T5]	2 | 39	0 | 5	0 | 7	1 | 2	9 | 24	3 | 6	0 | 0

S1: Checkstyle-5.6, S2: Commons-codec, S3: Commons-io, S4: Commons-logging, S5: JHotDraw-7.5.1, S6: Quartz-1.8.3, S7: Squirrel_sql-3.1.2, T1: Decor, T2: Designite, T3: JDeodorant, T4: JSpIRIT, and T5: Organic

Non-detection of bad smell. In some situations, the same tool does not detect a specific bad smell in more than one system. However, we kept these bad smells in our analysis because the tool detects smells in other systems. In the specific case of Large Class and Message Chains, where only Decor detects these bad smells, we decided to document it to identify if these bad smells were introduced after refactoring.

We provide a website[7] containing: (i) all systems analyzed, (ii) all data detected by the refactoring tools, (iii) all versions of the refactored systems, (iv) notes of particular cases, and (v) all analyses we carried out.

3.1 Impacts of Refactorings

After the detection of bad smells, our focus was on the following refactoring: Move Method solving Feature Envy, Replace Type Code with State/Strategy, and Replace Conditional with Polymorphism solving Type Checking. The last two were considered together and named Replace Refactoring.

Table 2 shows the number of refactorings carried out for each system, where the first column represents the systems used in our research. The next two columns represent the two types of refactoring performed. The last column shows the total number of refactorings performed for each system. Each line represents the data from an analyzed system, and the last line represents the total number of refactorings performed for each type of bad smell.

Some situations in Table 2 represent the no application of refactoring for a given type; i.e., the zero cases (0, 0*). The first zero (0) represents the situation where JDeodorant provided possibilities for refactoring, but it could not be performed. For instance, the cases found when it tries to perform the refactoring, Eclipse IDE returned an error, making it impossible to finish it. The other case was when Eclipse IDE showed compilation errors in the refactored system. We did not perform the refactoring when the compilation error was not trivial to solve.

The second zero (0*) represents situations in which JDeodorant did not suggest any refactoring. Probably, the cause of this situation might be no existing refactoring for that type. Alternatively, there may have been an internal error in the tool that made refactoring impossible. As the log system generated by the tool is not easily accessible to the end-user, we decided not to further investigate the reasons for this.

[7]https://cleitonsilvat.github.io/sbes2020/

Table 2: Refactorings Applied

System	MM	RR	**Total**
Checkstyle-5.6	18	5	**33**
Commons-codec	15	0	**15**
Commons-io	3	2	**05**
Commons-logging	0*	1	**01**
JHotDraw-7.5.1	13	0*	**13**
Quartz-1.8.3	6	11	**17**
Squirrel_sql-3.1.2	6	0*	**06**
Total	**61**	**19**	**80**

MM: Move Method; RR: Replace Refactoring

In a brief data summary, we identified a total of 80 applied refactorings, where the Replace Refactoring presented the lowest number of refactorings. Commons-logging was the system with the lowest number of refactorings (01), while Checkstyle-5.6 was the highest (33). The results obtained after a comparative analysis between the original and refactored versions of the system may be classified into three different types: decrease, increase, and neutral number of bad smells.

Decrease. This result type represents the decrease in the number of bad smells found in the system's refactored version compared to its original version.

Increase. This result type represents the increase in the number of bad smells found in the system's refactored version compared to its original version.

Neutral. This result type represents the non-existence or no-change in the number of bad smells found in the system's refactored version compared to its original version.

3.2 Comparative Analysis

Move Method Refactoring. According to our data, we have Move Method in 133 situations. The number of bad smells decreased in 13.53%; increased in 4.51%, and remained the same in 81.95%. This refactoring did not impact on Large Class, Long Parameter List, and Speculative Generality in any of the evaluated systems. In these cases, the number of bad smells remained the same. Message Chains only *increased* in one system, while Lazy Class only *decreased* in one system. In other systems, both bad smells remained the same. Data Class presented divergence of identification in the same system by different tools. That is, JSpIRIT computed an increase in the number of bad smells for JHotDraw-7.5.1, while Organic computed a decrease.

Replace Refactoring. This refactoring has been applied in 133 situations, where the number of bad smells decreased in 0.75%, increased in 13.53%, and remained the same in 85.71%. Large Class, Long Parameter List, and Message Chains were not impacted by Replace Refactoring. In all cases, the number of bad smells detected remained the same. Shotgun Surgery and Data Class only increased in one system; they remained the same in the others. Refused Bequest for Quartz-1.8.3 presented an increase of 190.5% and 4,000%. Their absolute values represent an increase from 21 to 61 and from 1 to 41, respectively.

3.3 Results Summarization

The applied refactorings highly impacted the number of analyzed bad smells. In the case of Move Method, the number of different bad smells removed was higher than the number of different bad smells introduced. Table 3 shows the removed and introduced bad smells after refactoring the seven systems. Therefore, we use Table 3 to answer **RQ1.1** and **RQ1.2**.

The first column of Table 3 shows the refactoring name. The second and third columns present the removed and introduced bad smells, respectively. We show the percentage of systems where bad smells were removed or introduced. With these results, we answer the RQs as follows.

RQ1. The impacts of automated refactoring may be positive and negative. As we evaluate different refactoring and bad smell detection tools, we may analyze them from different perspectives.

RQ1.1. The answer is positive. A single refactoring may remove more than one bad smell.

RQ1.2. The answer is positive. A single refactoring may introduce more than one bad smell.

4 THREATS TO VALIDITY

We discuss the threats to validity listed by Wohlin et al. [22]: construct, internal, conclusion, and external.

Construct and Conclusion Validity. The comparative analysis represents the authors viewpoint. To minimize this threat, we used five different bad smell detection tools to present the point of view from different dimensions. We performed the comparative analysis individually, taking into account only the original version, and applying a single refactoring strategy. The results found were all documented, without favoring any information or tool.

Internal Validity. The detailed specification of the process guaranteed the study replication. Possible limitations may cause changes made to the tools or setups. We performed all refactorings suggested by JDeodorant. The Eclipse IDE itself performed all adjustments made to solve trivial errors caused by some refactorings. For the bad smell detection tools, we used only their standard configuration without any modification in their process or results.

External Validity. This threat is related to the representativeness of the analyzed systems to generalize results to any other system. To reduce this threat, we chose a known database, the Qualitas Corpus. The number and systems evaluated were selected randomly, and the seven systems were sufficient to bring us preliminary insights into how refactoring can impact on certain bad smells.

5 RELATED WORK

Bavota et al. [1] mined the evolution history of three Java open-source projects to investigate whether refactoring activities occur on code components, suggesting there might be a need for refactoring operations. Their results show that quality metrics usually do not show a clear relationship with refactoring; 42% of refactoring operations are performed on code entities affected by code smells, and only 7% of the performed operations remove code smells.

Similarly, Cedrim and Garcia et al. [2] analyze how 16,566 refactorings distributed in ten different types affect the density of 13 types of code smells and the version histories of 23 projects. Results reveal that 79.4% of refactorings touched smelly elements, 57% did not reduce their occurrences, 9.7% of refactorings removed smells, and 33.3% induced new ones.

Fontana and Spinelli [7] analyze the impact of refactoring applied to remove code smells. They select four bad smells detected by three tools and apply the refactoring automatically using two tools. They performed this process in one open-source, object-oriented system of about 400 classes. They report a summary of the impact according to six metrics proposed to evaluate the code and design quality of a system.

Similarly to these studies, we evaluate the impact of refactoring on bad smells. However, we used seven open-source Java systems, in which we applied two different types of refactorings in an automated way. We also used five tools to detect ten types of bad smells. As a result, we provide a catalog of bad smells removed and introduced by refactoring.

6 CONCLUSION

This paper presented an empirical research analyzing the impact of refactoring on bad smells. To conduct this research, we selected seven open-source Java systems available in the Qualitas Corpus. We applied two refactorings and measured their impact on ten bad smells detected by five different tools. We observed that both refactorings may decrease, increase, or have neutral impact on the number of bad smells. Surprisingly, the number of decrease cases was the lowest compared to the others. Replace Refactoring

Table 3: Anwsering the Request Questions

	RQ1.1. Does the automated refactoring remove bad smells? Answer: Affirmative		**RQ1.2. Does the automated refactoring introduce bad smells? Answer: Affirmative**	
Refactoring	**Bad Smell**	**% of system**	**Bad Smell**	**% of system**
Move Method	Data Class	28.57%	Data Class	28.57%
	Feature Envy	85.71%	Long Method	14.29%
	Lazy Class	14.29%	Message Chains	14.29%
	Long Method	28.57%	Refused Bequest	14.29%
	Refused Bequest	14.29%	Shotgun Surgery	14.29%
	Shotgun Surgery	14.29%		
Replace Refactoring	Data Class	14.29%	Feature Envy	28.57%
			Lazy Class	57.14%
			Long Method	42.86%
			Refused Bequest	57.14%
			Shotgun Surgery	14.29%
			Speculative Generality	28.57%

presented the lowest decrease (0.75%) while Move Method showed the highest decrease (13.53%).

To better help developers, we investigated which bad smells tend to be introduced and removed by refactoring. For instance, Feature Envy was removed in 85.71% of the systems by Move Method. We found that Replace Refactoring may introduce more bad smells than remove, while the situation is opposite with Move Method. Our findings may assist developers in different ways. For instance, developers who want to apply refactoring, automatically or manually, are better informed now of which bad smells may be introduced or removed by refactoring

We foresee several options for future work. First, in the construct of the data, we consider extending this research to more systems and refactoring tools, different bad smells, and refactoring. Second, one may replicate this research with manual refactoring to evaluate the differences found between the automated and manual refactoring. Third, in the analysis process, one may evaluate the impacts on the intersection of detected bad smell to compose a different set of bad smells. Finally, we are working on a tool that considers this catalog for better refactoring.

ACKNOWLEDGMENTS

This research was partially supported by Brazilian funding agencies: CNPq (Grant 424340/2016-0), CAPES, and FAPEMIG (Grant PPM-00651-17)

REFERENCES

[1] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* (2015), 1–14.

[2] D. Cedrim and A. Garcia et al. 2017. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proc. of Foundations of Soft. Engineering.* 465–475.

[3] D. Cruz, E. Figueiredo, and A Santana. 2020. Detecting Bad Smells with Machine Learning Algorithms: an Empirical Study. In *Proc. of Int. Conf. on Technical Debt (TechDebt).*

[4] W. Fenske, S. Schulze, D. Meyer, and G. Saake. 2015. When code smells twice as much: Metric-based detection of variability-aware code smells. In *Int. Conf. on Source Code Analysis and Manipulation.* 171–180.

[5] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo. 2016. A Review-based Comparative Study of Bad Smell Detection Tools. In *Proc. of Int. Conf. on Evaluation and Assessment in Software Engineering.*

[6] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. 2012. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software* (2012), 2241–2260.

[7] F. A. Fontana and S. Spinelli. 2011. Impact of refactoring on quality code evaluation. In *Proc. of the 4th Workshop on Refactoring Tools.* 37–40.

[8] M. Fowler, K. Beck, J. Brant, and W. Opdyke. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

[9] V. Gupta, P. K. Kapur, and D. Kumar. 2016. Modelling and measuring code smells in enterprise applications using TISM and two-way assessment. *Int. Journal of Syst. Assurance Eng. and Management* (2016), 332–340.

[10] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien. 2015. Tracking the software quality of android applications along their evolution (t). In *30th IEEE/ACM Int. Conf. on Automated Software Engineering.* 236–247.

[11] N. Moha, Y.G. Gueheneuc, L. Duchien, and A.F. Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* (2009), 20–36.

[12] W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, A. B. Agbachi, R. Oliveira, and C. Lucena. 2018. On the identification of design problems in stinky code: experiences and tool support. *JBCS* (2018), 13.

[13] A. Ouni, M. Kessentini, M. Cinnéide, H. Sahraoui, K. Deb, and K. Inoue. 2017. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *JSEP* (2017).

[14] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze. 2016. Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In *Int. Conf. on Automated Soft. Eng.*

[15] T. Sharma, P. Singh, and D. Spinellis. 2020. An empirical investigation on the relationship between design and architecture smells. *Accepted in EMSE* (2020).

[16] M. Shomrat and Y. A. Feldman. [n.d.]. Detecting Refactored Clones. In *ECOOP,* Giuseppe Castagna (Ed.). 502–526.

[17] E. Sobrinho, A. De Lucia, and M. Maia. 2018. A systematic literature review on bad smells—5 W's. *Trans. on Software Engineering* (2018).

[18] C. S. Tavares, A. Santana, E. Figueiredo, and M. A. S. Bigonha. Accepted in 2020. Revisiting the Bad Smell and Refactoring Relationship: A Systematic Literature Review. In *Experimental Software Engineering (ESELAW).*

[19] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M.s Lumpe, H. Melton, and J. Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Asia Pacific Software Engineering Conference (APSEC2010).* 336–345. https://doi.org/10.1109/APSEC.2010.46

[20] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. 2015. When and why your code starts to smell bad. In *Proc. of the 37th Int. Conference on Software Engineering.* 403–414.

[21] S. A Vidal, C. Marcos, and J A. Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Soft. Engineering* (2016), 501–532.

[22] C. Wohlin, P. Runeson, M. Höst, M. C Ohlsson, B. Regnell, and A. Wesslén. 2012. *Exp. in software engineering.* Springer Science & Business Media.