Impact of Unanticipated software evolution on development cost and quality: an empirical evaluation

Rodrigo Vilar
Exact Sciences Department
Federal University of Paraíba
Rio Tinto, Brazil
rodrigovilar@dce.ufpb.br

Anderson Lima, Hyggo Almeida, Angelo Perkusich
Embedded Systems and Pervasive Comp. Lab.
Federal University of Campina Grande
Campina Grande, Brazil
anderson.lima, hyggo, perkusic@embedded.ufcg.edu.br

***Abstract*—Most techniques to aid maintenance and evolution of software require to define extension points. Generally, developers try to anticipate the parts that are more likely to evolve, but they can make mistakes and spend money in vain. With Unanticipated Software Evolution, developers can easily change any element of the software, even those that are not related with an extension point. However, we have not found empirical validations of Unanticipated Software Evolution impact on development cost and quality. In this work, we design and execute an experiment for Unanticipated Software Evolution (specifically, using the COMPOR platform), in order to compare its results metrics -- time, lines of code, test coverage and complexity -- using OO systems as baseline. 30 undergraduate students were subjects in this experiment. We concluded that COMPOR have significant impact on the Lines of code and Complexity metrics, reducing the amount of lines changed and the McCabe cyclomatic complexity on evolution of a small system.**

Keywords-Unanticipated Software Evolution, Cost, Quality, Empirical software engineering, Software Evolution.

I. INTRODUCTION

Some studies estimate that maintenance and evolution tasks spend between 50% and 90% of software development budget [13, 8]. Thus, Software Engineering researchers invest considerable resources in order to create new techniques that ease and reduce the cost of software evolution. Most of these techniques require developers to anticipate extension points (EP), which are flexible structures to hold new functionality and changes.

However, there is a trade-off: defining an EP is abstract and expensive; conversely, it is even more expensive to change software pieces that are not prepared for it. So, for each EP created, we expect a ROI (return of investment), deriving out of reducing the cost of later changes that use the same EP. For this reason, developers try to discern and isolate software chunks inclined toward change. Nevertheless, sometimes they do not predict EP correctly and ROI is zero.

Unanticipated Software Evolution (USE) is a Software Engineering approach, which aids developers to change any software fragment, even without EP [12]. It considers that is possible to reduce the cost of software evolution and preserve its quality, even when there is not investment to create EP. As a result, it would eliminate the trade-off we cited above and developers would not worry to create EP.

In an effort to confirm USE hypothesis, we have analyzed all articles published on USE events [12, 10, 11]. Nevertheless, none of these studies have validated the influence of USE on software development metrics such as quality and cost. In face of this gap, we define a business problem for this work.

Business Problem: *There is no convincing evidence on how USE influences software development metrics.*

In this paper, we perform an early evaluation of COMPOR [5, 6], a USE platform developed by Embedded Laboratory at UFCG [1], whose code is open source and is available online[2]. COMPOR is a container for components that communicate with each other indirectly, through a specific message mechanism. So that components have weak coupling and can be easily changed. In fact, COMPOR can even change components at run time.

We have defined a technical problem, reducing our scope to COMPOR and using more specific software development metrics.

Technical Problem: *There are not empirical studies that investigate COMPOR influence on software development cost and quality.*

We propose an experiment to fulfill this gap on USE validation. Since COMPOR is a USE platform, its experimental outputs are also USE results. So, we try to evaluate USE impact over software development through COMPOR.

Cost and Quality are abstract metrics. So we choose concrete metrics for our experiment. We measure Cost as the time spent and lines of code changed in order to complete a software evolution task. Likewise, we assess Quality being Cyclomatic complexity and Test coverage of code after evolution.

In an ideal configuration, the experiment should use professional developers to implement systems with two alternatives – coding using only Object oriented code or using

[1] http://www.embeddedlab.org
[2] http://bit.ly/COMPOR

DOI: 10.18293/SEKE2015-074

COMPOR – and compare the Time, LoC, Complexity and Coverage results. This way we would infer COMPOR impact on software development, considering OO results as baseline.

Due to resources and time limitations, we performed our experiment with undergraduate students, during an OO Design course. Carver et. al. [3] state that the risk of using inexperienced students is justifiable for pilot experiments. This kind of experiment would not be generalizable, but it contributes to fix experiment design problems and to guide future replications on professional development environments.

At this point, we can define our objective and hypotheses using the GQM template.

Objective: The purpose of this study is to measure the impact of COMPOR on evolution cost (time spent and lines of code changed) and quality (tests coverage and complexity) [16], from the point of view of software developers, in the context of evolution tasks for a small system implemented by undergraduate students, using plain Object Oriented implementations as baseline.

Hypothesis 1: COMPOR systems require less time to complete evolution tasks than plain OO systems;

Hypothesis 2: COMPOR systems change less lines of code to complete evolution tasks than plain OO systems;

Hypothesis 3: COMPOR systems have better test code coverage after evolution tasks than plain OO systems;

Hypothesis 4: COMPOR systems have lower cyclomatic complexity after evolution tasks than plain OO systems.

In the remaining of this paper, we show the related work, describe COMPOR features, detail the experiment design, analyze the experiment results and point out our conclusions and future work.

II. RELATED WORK

We have divided this section into two parts. Firstly, we review the literature about USE, looking for concrete tools and their empirical validation. After that, we show some experimental works for software evolution, similar to our experiment.

A. Unanticipated software evolution

We have found some USE works in literature. Oreizy et. al. defined an architecture for run-time software evolution [14]. Keeney and Cahill created a framework for dynamic adaptation [9]. Wurthinger et. al. modified a Java virtual machine to allow arbitrary runtime changes at any point at which a Java program can be suspended [19]. Piechnick et. al. propose a role-based composition system that enables the adjustment of unanticipated, dynamic self-variation of applications in a fine-grained manner [15]. However these works did not evaluated empirically the impact of USE on software cost and quality. Therefore, we expanded the scope of our literature review to experimental evaluation of software evolution.

B. Software evolution experiments

While investigating software evolution literature, we found several experimental studies that evaluated aspects of evolution.

Arisholm and his partners worked three times with alternative designs for a coffee machine simulator, which sells and prepares drinks [4], in order to: evaluate changeability of systems with good and bad design [2]; measure the effect of sequence in which maintenance tasks are performed on the time required to perform them and on the functional correctness of the changes made [18]; evaluate the effect of centralized versus delegated control design on software maintainability [1].

Deligiannis et. al. have replicated the [1] study, using other programming language, enhancing the evolution tasks and collecting more metrics [7]. Sfetsos et. al. used the coffee machine project to investigate the impact of developer personalities and temperaments on communication, pair performance and pair viability-collaboration [17].

In spite of not focusing on USE, these works helped us to design a comparative experiment for COMPOR.

III. COMPOR

This section explains the Unanticipated Software Evolution features of COMPOR that we evaluated in an experiment. COMPOR has a generic and formal specification for a component container. It defines the components structure and their indirect communication, decoupling components and easing their change. For example, in Fig. 1, a component A needs to invoke a service of another component B, A does not invoke B directly. Instead, B should declare a service named as s and A could use the COMPOR API to invoke service s. The COMPOR container discovers automatically where s is declared and invokes it.

In a hypothetical evolution scenario, the system client requires to change some functionality of s. The system developer only needs to deploy another component C that also declares a s service, replacing the former service from B. A and other components that invoke s are not aware of that change, since they do not know B and C components directly.

Currently, there are four COMPOR implementations, for Java, C#, C++ and Python languages. In this experiment, we used the Java Component Framework for COMPOR, which defines two API classes: `ComporFacade`, that must be extended to create the system entry point and to deploy components; and `Component`, that must be extended too, in order to declare services and invoke services of other components.

IV. EXPERIMENT DESIGN

A. Experimental Units

With a view to run this experiment using evolution tasks, we needed to choose a system to be implemented by the experiment subjects. We decided to use a small system, which can be completely implemented by just one developer, rather than a big one that demands several developers working together.

The coffee machine problem fits this small-system requirement. It has also been replicated on several experimental studies. So, we have decided to use the same project (with some adaptations) for this experiment.

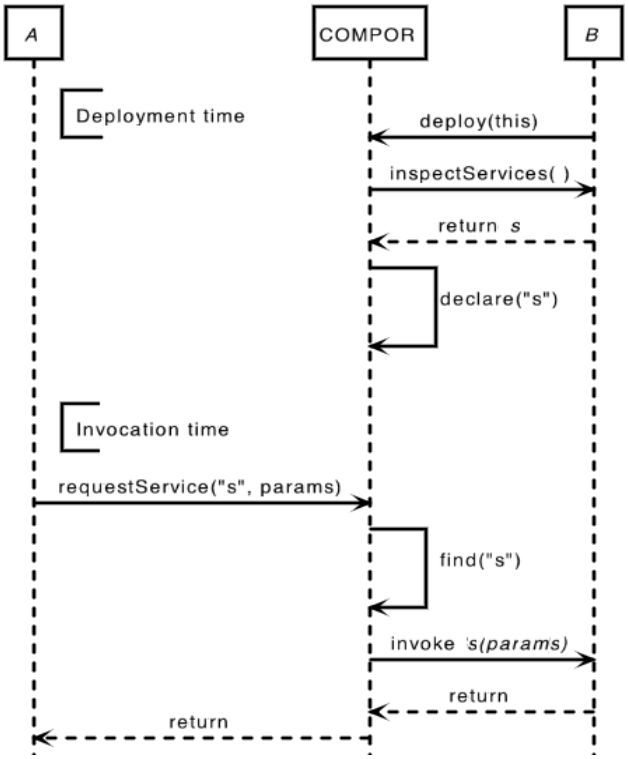

Figure 1. COMPOR: Declaring and invoking.

We planned its development as evolution tasks, which are the Experimental Units of our experiment.

The original coffee machine problem has four phases [4]: Payment with coins, four types of drink with the same price, cancel drink and return coins; add a new drink type with another price; use employees badges to directly debit the cost of drink purchases from paychecks; dynamic drink configuration.

We have partitioned these phases into 50 small tasks, so that developers can achieve better success rates on evolution tasks. Among these tasks there are also some new tasks that we have added to fill some missing functionality, e. g., loading coins on machine start in order to provide change.

Since COMPOR is a tool that starts operating from software design, the experiment does not need to measure COMPOR influence on requirements and analysis development phases. Therefore, we have simulated that requirements and analysis phases were already finished, and provided automatic functional tests for each evolution task. The functional tests run against a specific coffee machine Facade, which can be implemented using COMPOR or plain object orientation.

Next subsections detail the treatments designed to run with the experimental units.

B. Input: Independent variables

Factor: Technology

The main focus of this experiment is to compare result metrics of evolution tasks, between implementations that used COMPOR versus other versions that used plain OO. Therefore, the experiment contains a simple design with only one interesting factor, which has two levels: using COMPOR or using plain OO. The other sources of variation are undesired. So, we have designed the experiment in order to neutralize their effect over dependent variables.

Undesired controlled variable: Participants

The experiment engaged 30 third-year students of Licentiate in Computer Sciences at UFPB, which were taking an OO Design course. The OO Design teacher used this experiment to grade the students in a practical project. Each student has different levels of experience in OO programming. While some of them work as junior developers on start-ups, others have almost no programming skills. Before the experiment, all students already had classes of refactoring techniques and design patterns.

To reduce the effect of developer experience, we have allocated them into random pairs. In fact, when we compared the final grading score in OO Design class, the score standard deviation for individuals was 0.89. In other hand, the score standard deviation for the experiment pairs was 0.61. Therefore, we suggest that pair randomization really reduced the developer experience effect. Each pair performed pair programming during the evolution tasks. Moreover, the experiment design uses replication for participants, because each pair should carry out all evolution tasks. We divided the 15 student pairs randomly into two groups. The first five pairs should use COMPOR in the coffee machine implementation and the other ten pairs must use plain OO.

Undesired not controlled variable: Environment

There are some events that we cannot control and would impact the experiment results, such as, climate, holidays, students transportation problems, etc. In order to reduce the environment effect, we have designed the experiment in a controlled manner. All students worked in a laboratory at UFPB with similar schedule, resources and instructions to execute

C. Procedures

Preparing

We have prepared some guidelines to guide students through experiment:

- An Experiment Manual [3], explaining experiment conditions, purpose, resources, steps, auxiliary documentation and glossary;
- A Web Form to manually collect experimental unit configuration and time spent metric;
- Auxiliary documentation containing pseudo code, because instead of evaluating algorithms, we want to analyze design decisions;
- A Github repository for a coffee machine specification, containing a sequence of 50 tags (one for each evolution task). Each tag defines the functional tests,

[3] http://bit.ly/CoffeeMachineExperiment

using jUnit [4] and Mockito[5] , for its respective evolution task;

- A tutorial which we have used to give a class about COMPOR;
- A Github repository with the last COMPOR version.

The students were also trained on: Git, to manipulate Github repositories; Maven, to manage projects dependencies; JUnit and Mockito, to understand and execute the functional tests; Facade design pattern, which was used by the functional tests.

Executing

In the Coffee machine Github repository, we have created a tag for each evolution task, with an X.YY format. Where X means the coffee machine phase (from 1 to 4) and YY is the evolution task inside of the phase.

Beginning on tag 1.01 until tag 4.15, the student pair has to follow this procedure:

a) Merge the current implementation code with the next tag (except for tag 1.01, which has not implementation);
b) Set the task start time;
c) Evolve the code until all functional tests pass;
d) Set the task finish time;
e) Commit the task final code and send it to Github;

Submit the Web Form with task data, such as start and finish times, pair id, task id, technology and subjective questions about difficulties.

The results of first five evolution tasks were fragile, since we consider it as training for experiment modus operandi.

D. Output: Dependent Variables

After the Executing procedure detailed above, we can collect several data about each evolution task. Github provides a diff report for each commit, so we can calculate the amount of lines of code changed by the evolution task. The time spent is collected from a spreadsheet populated by the web form.

Collecting Complexity and Coverage data is harder, since we need to access each evolution task final code and run the Cobertura Maven plugin[6]. It generates an HTML report with total test code coverage and mean McCabe cyclomatic complexity.

V. ANALYSIS

In this section, we show the experiment result data and its transformations, in order to try to obtain normal-distributed data. We also perform some statistical tests and interpret the models results. After all, we check the hypotheses defined in Section 1.

A. Results and Transformations

The initial 26 experiment tasks (1.01 to 1.26) represent the first coffee machine requirement. While the subsequent ones represent evolution tasks. Due to software evolution importance and COMPOR evolution nature, we focused our analysis on evolution tasks (2.01 to 4.15).

Each student pair worked 35 hours in this experiment, but only one pair finished all evolution tasks successfully. All teams submitted approximately 500 experimental task logs.

We have ignored some data due to the following reasons:

- After task 4.01 there is not enough data to perform statistical tests;
- Three COMPOR teams used COMPOR poorly. Their Facades are replete of OO code and invoke COMPOR only three times. By comparison, the two remaining teams have smaller Facades and invoke COMPOR 8 and 23 times, respectively;
- The 3.02 Task alone weakened all COMPOR metrics. Since, in the 3.03 task, the metrics returned to normal levels, we consider the former task as an outlier.

This resulted in 338 observations that we have analyzed using the R statistical language. The tasks data, R code and program output are available online[7].

In the scope of this experiment, tasks size vary a lot and absolute data for response variables did not tend to be normally distributed. With this in mind, we have transformed raw experiment data into relative values based on the mean OO metrics for each task. For example, the mean Time for all OO teams on task 1.01 was 35 minutes. So, instead of using the absolute Time value for Team 01 on this task (106 minutes), we have made statistical tests using the respective relative value (302% of OO mean). The relative data became closer to the normal distribution than the absolute data.

After that, we applied a log transformation into Time and LOC metrics and they become almost normally distributed. We did not find any transformation that made Coverage and Complexity data normal. So, these variables were tested with non-parametric methods.

Since the experiment generated a lot of observations, we have decided to split the observations into seven sequential task groups of about 50 observations. After that, we made separated statistical tests for each task group. Therefore, we compared OO and COMPOR metrics seven times during experiment execution and got temporal conclusions for experiment results.

The task groups had different configurations for each response variable (Time, LOC, Coverage and Complexity). We made some group adjustments in order to find group boundaries where response variables change behavior. This approach optimized the results of statistical tests.

Firstly, we tested the normality and homoscedasticity of data for each combination of task group and response variable, for both OO and COMPOR teams. In the sequence of analysis, we used t tests for normal data and Wilcox tests for non-normal data, in an effort to discover significant relations between COMPOR and OO data: Do COMPOR metrics differ of OO metrics? Are COMPOR metrics lower than OO metrics? And

[4] http://www.junit.org
[5] http://code.google.com/p/mockito/
[6] cobertura.github.io/cobertura/
[7] http://bit.ly/ComporExperimentResults1

are COMPOR metrics greater than OO metrics? At least, we performed Power tests to analyze the probability of type II errors.

B. Interpretation

The relative Time spend to complete tasks (Table 1) had an alternating behavior in the experiment beginning. Between 1.04 and 1.24 tasks, 12 tasks spent less time with OO and 9 tasks spent fewer time with COMPOR. These results have considerable significance (p-value < 0.03) and power above 0.7.

As the experiment reached evolution tasks, COMPOR and OO metrics equalized. In spite of the low statistical power, this data indicates the COMPOR Time spent for evolution tasks is better than its own Time for development tasks. We should replicate this experiment, in order to obtain sufficient data until the 4.15 task and analyze the metrics trends. There still is one question: will COMPOR Time tend to equate OO Time infinitely or COMPOR will overcome OO?

In relation to the **LOC** metric, the relative amount of lines changed is equal for both technologies until task 1.25. The last 9 (evolution) tasks demanded less lines for COMPOR versions. As Table 1 shows, these data is significant and has statistical power above 0.5.

Regarding Test **Coverage**, we gave equal and fixed tests, mapping each task requirements, for all teams. So, low coverage rates mean that a team created a lot of unnecessary code, which reduces the code quality.

COMPOR teams had better coverage in the first 11 tasks and equals coverage in the middle 23 tasks. However, in the last 4 tasks, OO teams got better coverage results. This means that COMPOR teams wrote more unnecessary code in the experiment end and the use of COMPOR would impact system quality.

Finally, we analyzed the **Complexity** metric, where the first 15 tasks had similar results for both technologies. In the 11 intermediary tasks, the COMPOR teams code complexity was significantly lower than OO code. The last 8 (evolution) tasks showed similar results again, but with low statistic power. This means that there is a great probability that the statistical tests, which are not significant for tasks 2.01 - 4.01, were wrong. This data needs more replication to enhance the last tests power and find out Complexity trends: does COMPOR continue to generate code with lower complexity as system increases? This answer can be found in a future work.

C. Hypothesis test

Hypothesis 1: Does COMPOR systems require less time to complete evolution task than plain OO systems?

There is no significant trend on the impact of Technology factor on the Time teams took to perform the evolution tasks. So, we REJECT this hypothesis.

Hypothesis 2: Does COMPOR systems require less lines of code to complete evolution task than plain OO systems?

COMPOR teams changed less lines of code to implement evolution tasks. So, we ACCEPT this hypothesis.

Hypothesis 3: Does COMPOR systems have better test code coverage after completing evolution task than plain OO systems?

For almost all tasks, the Technology factor did not have significant impact on test coverage after evolution tasks, in the COMPOR versus OO comparison.

TABLE I. STATISTICAL TESTS FOR TIME, LOC, COVERAGE AND COMPLEXITY RESPONSES VARIABLES.

Variable	Task group	Normal data	Equal variance	Statistical tests (p-value)			Statistical power
				COMPOR != 0	COMPOR < 0	COMPOR > 0	
Time	1.04 - 1.07	Yes	Yes	0.042	0.021	NS	0.78
	1.08 - 1.11	Yes	Yes	0.003	NS	0.001	0.96
	1.12 - 1.16	Yes	Yes	0.048	0.024	NS	0.73
	1.17 - 1.24	Yes	Yes	0.054	NS	0.027	0.75
LOC	1.26 - 3.01	No	NA	0.053	0.026	NS	0.51
	3.03 - 4.01	Yes	Yes	NS	0.064	NS	0.59
Coverage	1.01 - 1.04	No	NA	0.058	NS	0.029	0.74
	1.05 - 1.07	Yes	No	0.087	NS	0.043	0.46
	1.08 - 1.11	No	NA	NS	NS	0.091	0.63
	3.03 - 4.01	No	NA	0.016	0.008	NS	0.999
Comp.	1.16 - 1.20	No	NA	0.044	0.022	NS	0.999
	1.21 - 1.26	No	NA	0.021	0.01	NS	0.99

a. This table shows only significant statistical results.

Moreover, in the last evolution tasks, OO teams wrote code with better coverage than COMPOR ones. So, we REJECT this hypothesis.

Hypothesis 4: Does COMPOR systems have lower cyclomatic complexity after completing evolution task than plain OO systems?

In spite of the final evolution tasks inconclusive results, COMPOR teams produced less complex code for 11 intermediary tasks. Therefore, we ACCEPT this hypothesis.

D. Threats to validity

Conclusion validity

Due to the low experience of the experiment subjects, we have noticed that most pairs did not take care of system design. They just want to finish the most tasks possible. These groups have written bad code and randomly made some refactoring, which could have influenced some results. We suggest, in future replications of this experiment, to reserve a period of time, after each task, only to perform refactoring.

Random irrelevancies can affect results, such as lack of internet, hardware problems on PCs, etc. Another threat is related to human experimental subjects who can easily change their behavior over time, generating noise to the data.

External validity

The main threat to validity in this experiment is to generalize the results, from the sample we have chosen – undergraduate students – to the real population of programmers. In the context of a company, it is expected that employees have a reasonable leveling in relation to software development. On the other hand, the academic environment is very heterogeneous, in terms of ability and knowledge.

Another problem is the generalization of COMPOR results to the whole class of Unanticipated Software Evolution tools, because COMPOR good metrics could be result of another COMPOR characteristic apart from USE. With this in mind, we hid most COMPOR function, such as, run-time adaptation and exposed only the inter-component communication.

VI. CONCLUSIONS

In this work, we designed and executed the first experiment, as far as we know, for Unanticipated Software Evolution that measures the effects of this technique on development cost and quality.

We consider the experiment design as a valid contribution, because it can be easily replicated with better configurations in order to obtain more relevant results. In the scope of this work, we have obtained some significant results for USE influence on development cost and code quality. While COMPOR significantly reduces the amount of lines changed and code complexity, it does not affect development time and code test coverage.

As future works, we suggest the replication of this experiment with other configurations: providing with sufficient time to complete all evolution tasks, until task 4.15; inviting professionals to perform evolution tasks and given them better COMPOR training; and using other USE tools. After acquiring stronger evidences of USE hypothesis, other works may also measure COMPOR performance overhead.

REFERENCES

[1] E. Arisholm and D. I. Sjoberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. Software Engineering, IEEE Transactions on, 30(8):521–534, 2004.

[2] E. Arisholm, D. I. Sjøberg, and M. Jørgensen. Assessing the changeability of two object-oriented design alternatives–a controlled experiment. Empirical Software Engineering, 6(3):231–277, 2001.

[3] J. Carver, L. Jaccheri, R. Morasca, and F. Shull. Issues in using students in empirical studies in software engineering education. In In IEEE METRICS, page 239. Prentice Hall, 2003.

[4] A. Cockburn. The coffee machine design problem: Part 1 & 2. C/C++ Users Journal, may/june 1998.

[5] H. de Almeida, A. Perkusich, E. Costa, and R. Paes. Compor: a methodology, a component model, a component based framework and tools to build multiagent systems. CLEI Electronic Journal, 7(1), 2004.

[6] H. O. de Almeida, A. Perkusich, G. Ferreira, E. Loureiro, and E. de Barros Costa. A component model to support dynamic unanticipated software evolution. In Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2006), San Francisco, CA, USA, July 5-7, 2006, pages 262–267, 2006.

[7] I. Deligiannis, P. Sfetsos, I. Stamelos, L. Angelis, A. Xatzigeorgiou, and P. Katsaros. Assessing the modifiability of two object-oriented design alternatives– a controlled experiment replication. In Proceedings 5th EUROSIM Congress on Modelling and Simulation, 2004.

[8] L. Erlikh. Leveraging legacy system dollars for ebusiness. IT professional, 2(3):17–23, 2000.

[9] J. Keeney and V. Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework, 2003.

[10] G. Kniesel, P. Costanza, and J. L. Fiadeiro. Second international workshop on unanticipated software evolution, Apr. 2003.

[11] G. Kniesel and T. Mens. First international workshop on foundations of unanticipated software evolution, Mar. 2004.

[12] G. Kniesel, J. Noppen, T. Mens, and J. Buckley. First international workshop on unanticipated software evolution, June 2002.

[13] B. P. Lientz and E. B. Swanson. Software Maintenance Management. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[14] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In Proceedings of the 20th International Conference on Software Engineering, pages 177–186. IEEE Computer Society, 1998.

[15] C. Piechnick, S. Richly, S. Gotz, C. Wilke, and U. Aßmann. Using role-based composition to support unanticipated, dynamic adaptation smart application grids. In ADAPTIVE 2012, The Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications, pages 93–102, 2012.

[16] D. Racodon. Developers' seven deadly sins, July 2014.

[17] P. Sfetsos, I. Stamelos, L. Angelis, and I. Deligiannis. An experimental investigation of personality types impact on pair effectiveness in pair programming. Empirical Software Engineering, 14(2):187–226, 2009.

[18] A. I. Wang and E. Arisholm. The effect of task order on the maintainability of object-oriented software. Information and Software Technology, 51(2):293–305, 2009.

[19] T. Wurthinger, C. Wimmer, and L. Stadler. Unrestricted and safe dynamic code evolution for java. Science of Computer Programming, 7 2011.