2014 IEEE 7th International Conference on Service-Oriented Computing and Applications

Investigating the Change-proneness of Service Patterns and Antipatterns

Francis Palma*†, Le An‡, Foutse Khomh‡, Naouel Moha† and Yann-Gaël Guéhéneuc*
*Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
†Latece, Département d'informatique, Université du Québec à Montréal, Canada
‡SWAT, DGIGL, École Polytechnique de Montréal, Canada
Email: {francis.palma, le.an, foutse.khomh, yann-gael.gueheneuc}@polymtl.ca, moha.naouel@uqam.ca

***Abstract*—Like any other software systems, service-based systems (SBSs) evolve frequently to accommodate new user requirements. This evolution may degrade their design and implementation and may cause the introduction of common bad practice solutions—*antipatterns*—in opposition to *patterns* which are good solutions to common recurring design problems. We believe that the degradation of the design of SBSs does not only affect the clients of the SBSs but also the maintenance and evolution of the SBSs themselves. This paper presents the results of an empirical study that aimed to quantify the impact of service *patterns* and *antipatterns* on the maintenance and evolution of SBSs. We measure the maintenance effort of a service implementation in terms of the *number of changes* and the *size of changes* (*i.e.*, *code churns*) performed by developers to maintain and evolve the service; two effort metrics that have been widely used in software engineering studies. Using data collected from the evolutionary history of the SBS FraSCAti, we investigate if (1) services involved in patterns require less maintenance effort; (2) services detected as antipatterns require more maintenance effort than other services; and (3) if some particular service antipatterns are more change-prone than others. Results show that (1) services involved in patterns require less maintenance effort, but not at statistically significant level; (2) services detected as antipatterns require significantly more maintenance effort than non-antipattern services; and (3) services detected as *God Component*, *Multi Service*, and *Service Chain* antipatterns are more change-prone (*i.e.*, require more maintenance effort) than the services involved in other antipatterns. We also analysed the relation between object-oriented code smells and service patterns/antipatterns and found a significant difference in the proportion of code smells contained in the implementations of service patterns and antipatterns.**

Keywords*—*SOA, services, patterns, antipatterns, maintenance, change-proneness, empirical software engineering.

I. Introduction

Service Oriented Architecture (SOA) is a dominant architectural choice in the industry today [1]. SOA offers the ability to develop low-cost, reusable, and scalable distributed systems by composing ready-made services, *i.e.*, autonomous, reusable, and platform-independent software units that clients can search and invoke through a network, such as the Internet. Like any complex software systems, service-based systems (SBSs) also evolve to accommodate new user requirements both in terms of functionality and quality of service (QoS). These frequent changes often degrade the design and QoS of SBSs and cause the introduction of antipatterns which are common bad practice solutions—in opposition to patterns which are good solutions to common recurring design problems. A degradation of the design of an SBS means that it fails to follow one of the eight SOA design principles [2], including loose coupling, composability, and reusability. *Multi service* [3], an example of service antipattern corresponds to a service that implements a multitude of business and technical abstractions. Its reusability is low because it aggregates too much into a single service, resulting in methods with low cohesion. This service is often unavailable to end-users because of its overload, which may induce a high response time. *Proxy pattern* [4], an example of service pattern, is a well-known service design pattern that adds an additional indirection level between the client and the invoked service, *e.g.*, to support adding non-functional behaviors. Despite the relatively large body of work on the detection of service patterns and antipatterns in SBSs (*e.g.*, [5], [6], [7], [8]), to the best of our knowledge, there are very few studies that empirically investigated the impact of service patterns or antipatterns on the maintenance and evolution of SBSs. To perform such a study, one needs detailed information about the implementations of services, which is not easy to obtain because of the scarcity of open-source SBSs. We believe that service antipatterns do not only affect the clients of SBSs but also the maintenance and evolution of the SBSs, for example by making it harder for developers to modify existing functionalities, or to implement new ones. Several works exist in the object-oriented (OO) literature relating code smells and antipatterns to the change-proneness of software systems [9], [10], [11]. However, because of the dynamic nature of service patterns and antipatterns [6] and because of the difference in granularity, results obtained for OO systems cannot be simply transferred to SBSs. Service antipatterns and OO antipatterns are two very different concepts. Indeed, one of the root causes of OO antipatterns is the adoption of a procedural design style in OO systems, whereas service antipatterns often stem from the adoption of OO design style in SBSs.

In this paper, using data collected from the evolutionary history of the SBS FraSCAti, we perform an empirical study aimed at quantifying the impact of service *antipatterns* on the maintenance and evolution of SBSs. To measure the change-proneness of a service, in terms of its implementation, we rely on two widely used effort metrics: (1) *number of changes* and (2) *code churns*; which capture the frequency and the size of changes on a service. We address the following three research questions:

RQ$_1$: *What is the relation between service patterns and change-proneness?*
Finding: The total number of source code changes and code

978-1-4799-6833-6/14 $31.00 © 2014 IEEE
DOI 10.1109/SOCA.2014.43

churns performed during the maintenance and evolution of services involved in patterns is less than the total number of source code changes and code churns performed in other services—the difference is not statistically significant.

RQ$_2$: *What is the relation between service antipatterns and change-proneness?*

Finding: The total number of source code changes and code churns performed during the maintenance and evolution of services involved in antipatterns is higher than the total number of source code changes and code churns performed on other services—the difference is statistically significant.

RQ$_3$: *What is the relation between particular kinds of service antipatterns and change-proneness?*

Finding: Services found to be involved in *God Component*, *Multi service*, and *Service Chain* antipatterns are more change-prone than services involved in other antipatterns—the difference is statistically significant.

We examined the confounding impact of OO code smells contained in the implementations of the services, by comparing the proportion of code smells in the implementations of service patterns and antipatterns. Results show that the implementations of service antipatterns contain significantly more code smells than the implementations of service patterns. Hence, the higher change-proneness observed for service antipatterns implementations is likely due in part to the presence of code smells, since previous studies have found code smells to be more change-prone than codes that do not contain any smell. In summary, service antipatterns (respectively patterns)—which indicate poor (respectively good) service designs—do not only affect the clients of SBSs but also the cost of maintenance of the SBSs themselves. Developers and maintainers should therefore avoid implementing antipatterns in their SBSs since it will significantly increase the maintenance effort and hence the maintenance cost of the system.

The remainder of this paper is organised as follows. Section II presents background information on the FraSCAti project. Section III describes the approach used to extract change and churn information, and to identify FraSCAti services involved in patterns and antipatterns, and the design of our study. We present the findings in Section IV. Section V discusses the confounding effect of code smells while Section VI reports the threats to the validity of our results. Finally, Section VII presents related work followed by Section VIII, which concludes and sketches future work.

II. FraSCAti

FraSCAti [12] is a Java-based open-source implementation of the Service Component Architecture (SCA) standard [13]. FraSCAti is based on the OW2 Fractal[1] component model and provides an open architecture for the integration and binding of SCA components. SCA defines a technology-agnostic model for composing diverse interface definition languages (`WSDL`, Java, `WADL`, etc.), implementation languages and frameworks (Java, `BPEL`, `C/C++`, Spring, `OSGi`, etc.), bindings (`SOAP`, `JMS`, `REST`, etc.). To date, FraSCAti is the largest service-oriented SCA system for which the source code and change commits are publicly available. Table I summarises the main attributes of the FraSCAti project for its entire revision history. FraSCAti offers 130 distinct services. The size of the FraSCAti project is around 170 KLOC excluding any supporting and configuration files. We collected more than 15,000 changed files from the entire FraSCAti revision history, more than 9,000 of which are Java source files. The patterns and antipatterns studied in this paper were detected for 62 services, thus around 3,700 Java source files were involved directly or indirectly with these services implementations. Moreover, a few more than 1,800 and 2,100 analysed Java source files were directly involved with the studied patters and antipatterns in [5] and [14], respectively. As shown in Table I, we extracted more than 71,000 changes and approximately 62.6 million code churns from the entire FraSCAti commits.

[1]`http://fractal.ow2.org/`

III. Study Design

This section presents the design of our study, which aims to address the three research questions stated in Section I.

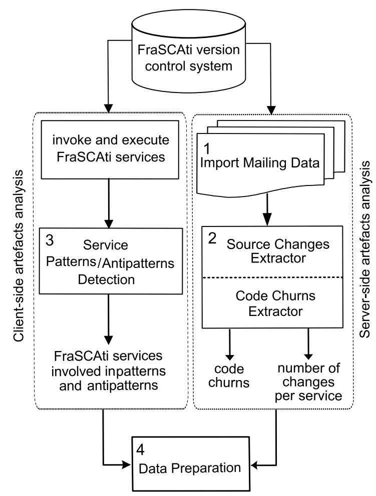

Figure 1. An overview of our approach to study the impact of service patterns and antipatterns on the change-proneness of SBSs.

A. Data Collection and Processing

In this study, we analyse FraSCAti services over their entire revision history. Our data set contains, for each FraSCAti service: (1) all the source code changes performed and (2) the code churns in its entire revision history. We also gather the type and the number of service patterns and antipatterns in which a FraSCAti service is involved, using SODOP [5] and SODA [14] detection techniques. Figure 1 shows an overview of our data collection and processing approach. The remainder of this section elaborates on each of its steps.

*1) **Step 1: Collecting Mailing Data**:* The first step consists in mining change data information from the complete FraSCAti-commits mailing list archives. The FraSCAti-commits mailing list archive is available online[2]. We developed a Python script to recursively download all the change histories for the entire FraSCAti project. For each commit message in the mailing list, we extract (1) the revision number, (2) the author, (3) the date of the commit, (4) the log message, (5)

[2]`http://mail-archive.ow2.org/`

Table I. SUMMARY OF THE CHARACTERISTICS OF THE FRASCATI PROJECT V1.4 (THE ENTIRE REVISION HISTORY).

Total Services	Total Size	Total Changed Files	Total Java Source Files	Total Changes	Total Code Churns
130	170 KLOC	15,863	9,020	71,151	62,676,363
Analysed Services	Analysed Java Source Files	Java Source Files Related to Patterns	Java Source Files Related to Antipatterns	Java Source Files Related to Both	Java Source Files Related to None
62	3,717	1,860	2,114	1,840	18

the modified paths, *i.e.*, the list of changed files, (6) the added paths, *i.e.*, the list of newly added files, (7) the removed paths, *i.e.*, the list of removed files, and (7) the *diff* which contains detailed information about the changes that were performed on each of the modified, added, or removed files. We stored all these information in a MySQL database for analysis.

*2) **Step 2: Extracting Source Code Changes and Code Churns**:* The second step involves the extraction of the number of changes made and the number of code churns for a certain service artefact. In this case, we also used a Python script to query our database and calculate the number of times that each file involved in the implementation of a service appeared in the commit. For each file and for each commit containing the file, we parse the *diff* contained in the commit and extract information about the number of lines of code that were added and removed. We use this information to compute the Code churn of the file for that commit, as the total number of added, modified and deleted lines of code in the file. In a *diff*, the modification of a line is recorded as a line deletion followed by a line addition. We aggregate the code churns values of the file obtained for all commits in which the file was involved to obtain the *total code churn* of the file.

*3) **Step 3: Service Patterns and Antipatterns Detection**:* The third step in our approach involves detecting service patterns and antipatterns in FraSCAti. This detection is done on the client-side by analysing service compositions, system design, and the quality of service (QoS). We perform the detection of service antipatterns using SODA (Service Oriented Detection for Antipatterns) [14]. We asked the core development team of FraSCAti to manually validate all the antipatterns that were found in FraSCAti before their usage in our study. We also used the SODOP approach (Service Oriented Detection Of Patterns) proposed by Demange *et al.* [5] to perform the detection of service patterns. The patterns found in FraSCAti were also manually validated by the core development team of FraSCAti. A detailed description of the service antipatterns and patterns analysed in this study is available at http://sofa.uqam.ca/impact/. Table II shows the summary of the detection results for service-oriented patterns and antipatterns in FraSCAti v1.4.

*4) **Step 4: Data Preparation**:* Once we extracted the changes and code churns and performed the detection of service patterns and antipatterns, in this last step of our approach, we grouped the source code changes and code churns to link them with the detected patterns and antipatterns. In our current study, we do not consider the types of changes (that we plan to investigate in the future) and only focus on the number of changes and code churns as the measures of change-proneness of a service implementation.

We map each service to the corresponding artifacts or source files from the entire FraSCAti project in the form of $s_i \rightarrow f_{1\ to\ k}$, where for each i from 1 to 130, a service s is associated with different numbers of artefacts or source files f up to k. We classify the entire FraSCAti project into three groups: (1) Java source files that underwent any number of changes and are part of any patterns or antipatterns, (2) Java source files that underwent any changes and are not related to any patterns or antipatterns, and (3) Java source files that did not undergo any changes. This classification helps us to restrain relevant source files while we compare among changed *vs.* unchanged and pattern *vs.* antipattern service groups. We also manually gather various details about the patterns and antipatterns considered in this study, *e.g.*, their categories, the levels of their appearance, root causes, and symptoms. Finally, we collected the feature details from the FraSCAti feature model[3] for each FraSCAti service, *i.e.*, which particular features are implemented by a service in FraSCAti. This feature information helps us to better understand the changes made in services that are involved in patterns or antipatterns. Using these data, we perform a series of statistical analysis to examine the relation between service patterns/antipatterns and change-proneness. The following sections discuss the details of our analysis method.

Table II. SUMMARY OF THE DETECTION RESULTS FOR SERVICE-ORIENTED PATTERNS AND ANTIPATTERNS IN FRASCATI V1.4.

	Names	Detected Instances	Involved Java Source Files
Patterns	Adapter	1	14
	Basic Service	5	54
	Facade	3	62
	Proxy	3	61
Antipatterns	Bloated Service	3	25
	Bottleneck Service	2	24
	God Component	2	4
	Multi Service	1	5
	Nobody Home	4	12
	Service Chain	3	10
	The Knot	1	24
	Tiny Service	1	24
	OO Code Smells	26,381	3,717

B. Variable Selection

We identify the following dependent and independent variables to test the *null* hypotheses (defined in Section IV) corresponding to each research question.

1) Independent Variables: The total set of service patterns and antipatterns that we considered in this study are the independent variables. We investigate the presence of eight different service antipatterns and four different service patterns. For $\mathbf{RQ}_1$, we use the boolean variable f_i^1 to indicate whether a file i was involved in the implementation of at least one pattern. For $\mathbf{RQ}_2$, we use a similar boolean variable f_i^2 to indicate whether a file i was involved in the implementation of at least one antipattern. Finally, for $\mathbf{RQ}_3$, we use the boolean variables $f_{i,j}^3$ to indicate whether a file i was involved in the implementation of the antipattern j.

[3] http://frascati.ow2.org/doc/1.4/ch12s02.html

2) Dependent Variables: The dependent variables measure the phenomena related to services participating in service antipatterns or patterns. Our dependent variable for research questions $\mathbf{RQ}_1$ to $\mathbf{RQ}_3$ is the change-proneness of the files involved in the implementation of a service. We measure the change-proneness of a file *i* using the total number of changes c_i and the total number of code churns d_i that the file *i* underwent in its entire revision history.

C. Analysis Method

We apply the Wilcoxon rank sum and Kruskal-Wallis tests [15] to compare the proportion of source code changes and code churns in the different categories of services (*i.e.*, service antipatterns, service patterns, and others), using a 95% confidence level (*i.e.*, p-value<0.05). For any comparison exhibiting a statistically significant difference, we further compute the Cliff's δ effect size [16] to quantify the importance of the difference because Cliff's δ is reported to be more robust and reliable than Cohen's d [17].

The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical tests make no assumptions about the distributions of assessed variables. The Kruskal-Wallis test is an extension of the Wilcoxon rank sum test for more than two groups. Cliff's δ is a non-parametric effect sizes measure (*i.e.*, it makes no assumptions of a particular distribution) which represents the degree of overlap between two sample distributions [16]. It ranges from -1 (if all selected values in the first group are larger than the second group) to +1 (if all selected values in the first group are smaller than the second group). It is zero when two sample distributions are identical [18].

IV. Case Study Results

In this section, we present and discuss the answers to our three research questions. For each research question, we present the motivation behind the question, our analysis approach, and a discussion on our findings.

RQ*$_1$*: What is the relation between service patterns and change-proneness?

Motivation: The SOA paradigm has a specific set of design principles associated with it. Over the past years, patterns for SOA (*i.e.*, service patterns) have been proposed to guide developers through the application of these design principles, in order to help them reap the benefits of SOA, which includes fast and cost-effective responses to changes [19]. Each SOA service pattern affects and influences the application of one or more SOA design principles. There are also adverse relationships, where the results and trade-offs of some service patterns have a negative impact on a design principle [2]. A violation of some design principles can in turn result in a degradation of the quality of the SBSs. A better understanding of relations between service patterns and SBSs software quality is therefore important to guide development teams in making good design decisions. Yet, to date, no empirical evidence is available to validate the positive or negative impact of service patterns on the quality of SBSs. In this research question, we investigate the effect of service patterns on code change-proneness. Code change-proneness is an important quality attribute since it captures the effort required to modify and evolve the code of the SBSs, which translates into maintenance costs.

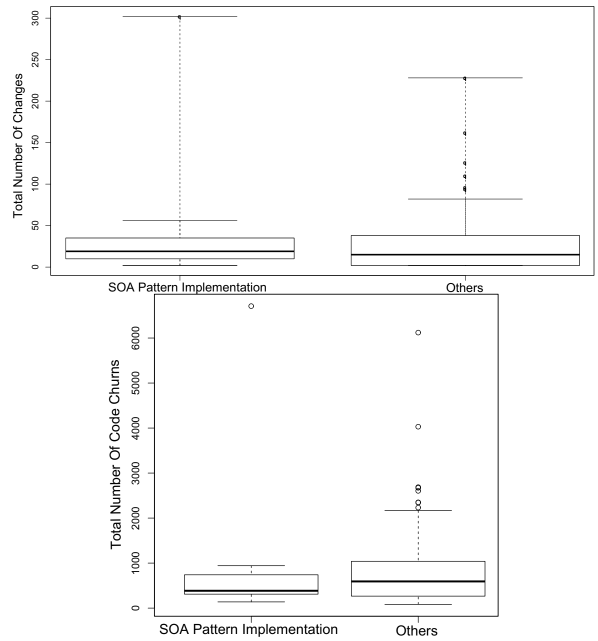

Figure 2. Comparison between pattern services and non-pattern services in terms of number of changes (top) and code churns (bottom).

Approach: We answer this research question in three steps: First, we perform the detection of service design patterns using SODOP [5] and obtain a set of FraSCAti services involved in different design patterns. We manually validate the results of our detection as discussed in Section III-A. Next, for each file implementing a FraSCAti service, we measure the change-proneness of the service's implementation using the following two metrics:

- **Total number of changes**: the total number of times that the file was changed in its entire revision history.
- **Total number of code churns**: the total number of churns (*i.e.*, lines added, deleted, and modified) that the file underwent in its entire revision history.

To compare the change-proneness of files involved in the implementation of service patterns with the change-proneness of files implementing services that are not involved in a pattern, we test the two following *null* hypotheses:

H^1_{01}*: there is no difference between the total number of changes experienced by files involved in the implementation of a service pattern and other files.*

H^2_{01}*: there is no difference between the total number of code churns experienced by files involved in the implementation of a service pattern and other files.*

We use the Wilcoxon rank sum test to examine H^1_{01} and H^2_{01}. H^1_{01} and H^2_{01} are two-tailed since they investigate whether service patterns are related to higher or lower change and code churn rates. All the tests are performed using the 5% significance level (*i.e.*, p-value<0.05).

Findings: Services involved in service patterns are less change-prone than the services not involved in any service pattern, but not at statistically significant level. Figure 2 presents the box-plots showing the median difference between pattern services and non-pattern services, both for the total number of changes (top) and the total number of code churns (bottom). In Figure 2, we observe the difference between the median values of the two groups. However, this difference is not statistically significant as the Wilcoxon rank sum test yielded p-values of 0.487 (>0.1) and 0.603 (>0.1), for respectively the total number of changes and the total number of code churns (see Table III). The Cliff's δ effect size values presented in Table IV also show a negligible difference.

Table III. THE WILCOXON RANK SUM TEST BETWEEN SERVICE PATTERNS AND OTHER SERVICES.

Treatment Groups	Treatment Types	p-value
patterns $\sim$ non-patterns	total number of changes	0.487
patterns $\sim$ non-patterns	total number of code churns	0.603

Table IV. THE NON-PARAMETRIC CLIFF'S δ EFFECT SIZE MEASURE BETWEEN SERVICE PATTERNS AND OTHER SERVICES.

Treatment Groups	Treatment Types	Cliff's δ
patterns $\sim$ other-services	total number of changes	-0.075 (negligible)
patterns $\sim$ other-services	total number of code churns	-0.075 (negligible)

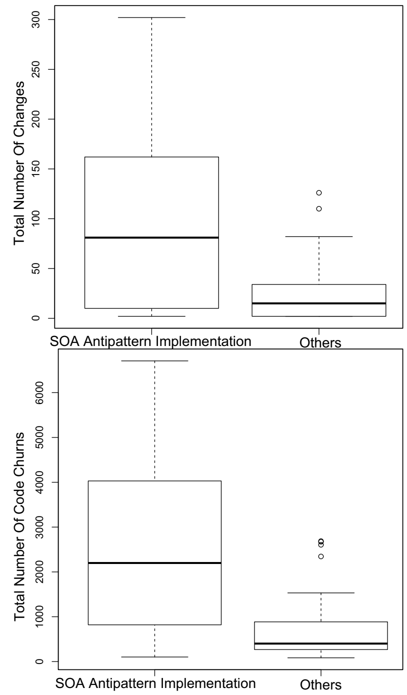

Figure 3. Comparison between antipattern services and non-antipattern services in terms of number of changes (top) and code churns (bottom).

RQ$_2$: What is the relation between service antipatterns and change-proneness?

Motivation: In **RQ**$_1$, we found that the services involved in patterns are less change-prone (although, not statistically significant) than other services (which is a positive impact). Since service antipatterns represent *poor* designs, it is very likely that they negatively impact the quality of SBSs, for example by making them more prone to changes, which may result in an increase of maintenance costs. Clearing up the interaction between service antipatterns and change-proneness is important from both researchers' and practitioners' points of view. For researchers, a quantitative analysis of the impact of service antipatterns on change-proneness will contribute to proving or refuting the conjecture about their negative impact. For practitioners, knowing how service antipatterns affect the change-proneness of their code will help them make educated decisions about which antipattern to remove first. In this research question, we investigate the effect of service antipatterns on code change-proneness.

Approach: Similar to **RQ**$_1$, we answer this research question in three steps: First, we perform the detection of service antipatterns using SODA [14] and obtain a set of services involved in different antipatterns. We manually validated the results of our detection as discussed in Section III-A. Next, for each file implementing a FraSCAti service, we measure the change-proneness of the service's implementation using the same metrics as in **RQ**$_1$ (*i.e.*, *total number of changes* and *total number of code churns*). Finally, to compare the change-proneness of files involved in the implementation of service antipatterns with the change-proneness of files implementing services, but not involved in a service antipattern, we test the two following *null* hypotheses:

H^1_{02}: *there is no difference between the total number of changes experienced by files involved in the implementation of a service antipattern and other files.*

H^2_{02}: *there is no difference between the total number of code churns experienced by files involved in the implementation of a service antipattern and other files.*

Similar to **RQ**$_1$, we use the Wilcoxon rank sum test to examine H^1_{02} and H^2_{02}, which are also two-tailed since they investigate whether service antipatterns are related to higher or lower change and code churn rates. For any comparison exhibiting a statistically significant difference, we compute the Cliff's δ effect size [16] to quantify the importance of the difference. All the tests are performed using the 5% significance level (*i.e.*, p-value<0.05).

Findings: Services involved in service antipatterns are more change-prone than the services that are not involved in any service antipattern. Figure 3 presents the box-plots showing the median difference between antipattern services and non-antipattern services both for the total number of changes (top) and the total number of code churns (bottom). This difference is statistically significant as the Wilcoxon rank sum test yielded p-values of 0.011 (<0.05) and 0.015 (<0.05) for respectively the total number of changes and the total number of code churns (see Table V). Therefore, we reject both H^1_{02} and H^2_{02}. The Cliff's δ effect size values presented in Table VI shows that the difference is large for both the total number of changes and the total number of code churns (p-value <0.01).

RQ$_3$: What is the relation between particular kinds of service antipatterns and change-proneness?

Motivation: In this research question, we investigate whether certain kinds of service antipatterns are more change-prone

Table V. THE WILCOXON RANK SUM TEST BETWEEN SERVICE ANTIPATTERNS AND OTHER SERVICES.

Treatment Groups	Treatment Types	p-value
antipatterns $\sim$ non-antipatterns	total number of code churns	0.015
antipatterns $\sim$ non-antipatterns	total number of changes	0.011

Table VI. THE NON-PARAMETRIC CLIFF'S δ EFFECT SIZE MEASURE BETWEEN SERVICE ANTIPATTERNS AND OTHER SERVICES.

Treatment Groups	Treatment Types	Cliff's δ
antipatterns $\sim$ non-antipatterns	total number of code churns	0.515 (large)
antipatterns $\sim$ non-antipatterns	total number of changes	0.496 (large)

than others. Knowing which service antipatterns are more change-prone could help development teams and managers better focus their limited resources toward the correction of the most change-prone antipatterns, thereby reducing the maintenance cost of their SBSs. Researchers working on antipatterns detection tools could also use this information to prioritise the results of their detection tools and guide their users toward service antipatterns with high change-proneness.

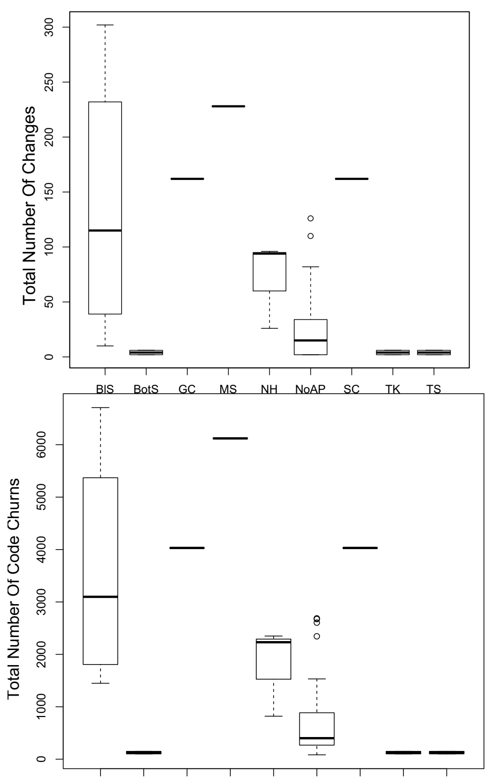

Figure 4. Comparison among antipattern services in terms of total number of changes (top) and code churns (bottom).

Approach: To answer this research question we proceed as follows: First, using the results of the antipattern detection performed in $\mathbf{RQ}_2$, we divide the files involved in the implementation of service antipatterns in different categories corresponding to the 13 kinds of antipatterns that are considered in this study. For each kind of antipattern A_i, we create a group GA_i containing files that are involved in the implementation of an antipattern of type A_i. In total, we obtained eight groups of files ($GA_i, i \in \{1, \dots 8\}$ since only eight kinds of service antipatterns were detected and validated in FraSCAti. We also create a group $GNoAP$ containing files implementing services that are not antipatterns. Next, for each file implementing a FraSCAti service, we measure the change-proneness of the service's implementation using the same metrics as in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ (*i.e.*, *total number of changes* and *total number of code churns*). Finally, to compare the change-proneness of files involved in the implementation of different kinds of service antipatterns with the change-proneness of files implementing services that are not antipatterns, we test the two following *null* hypotheses:

H^1_{03}: there is no difference between the total number of changes experienced by files from groups $(GA_i)_{i \in \{1, \dots 8\}}$ and $GNoAP$.

H^2_{03}: there is no difference between the total number of code churns experienced by files from groups $(GA_i)_{i \in \{1, \dots 8\}}$ and $GNoAP$.

We use the Kruskal-Wallis test to examine H^1_{03} and H^2_{03}. The two hypotheses are two-tailed (as in $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$). We test H^1_{03} and H^2_{03} using the 5% significance level (*i.e.*, p-value<0.05).

Findings: The eight kinds of antipatterns investigated in this study are not equally change-prone. Figure 4 presents the boxplots showing the medians of the total number of changes (top) and total number of code churns (bottom) in the nine groups (eight groups for the eight kinds of antipatterns and the no antipattern group). The result of the Kruskal-Wallis test presented in Table VII suggests that the difference is statistically significant. Hence, we reject both H^1_{03} and H^2_{03}. From Figure 4, we observe that *God Component* (GC), *Multi service* (MS), and *Service Chain* (SC) antipatterns have code churn values greater than 4000, while most other kinds of antipatterns have churn values less than 2000. As for the number of changes, the highest median value for a kind of antipattern is between 160 and 230. Overall, *God Component*, *Multi service*, and *Service Chain* antipatterns are more change-prone than other kinds of antipatterns.

Table VII. KRUSKAL-WALLIS TEST FOR THE DIFFERENT KINDS OF SERVICE ANTIPATTERNS.

Test Types	*p-value*
total number of code churns $\sim$ antipattern	0.0002
total number of changes $\sim$ antipattern	0.01003

We now discuss the possible reasons behind the high change-proneness of these three kinds of antipatterns. The service `ComponentFactory`, identified as *Service Chain* [14] and *God Component* [14] antipattern, is implemented by the `component-factory` FraSCAti component. The main role of this service is to generate and instantiate SCA components, which is one of the major steps to execute an SCA application. When an SCA application executes, it follows several sequential steps, including loading the SCA configuration file, parsing it, instantiating the SCA components, resolving the bindings, and so on. Therefore, the `ComponentFactory` service is in that invocation chain and highly related to other collaborating services. Because of this strong dependency, if others change, there is a high possibility that this `ComponentFactory` will also change frequently. Being a *God Component*, the `ComponentFactory` service also has a high number of encapsulated services with many methods and parameters.

The service `MembraneGeneration`, identified as *Multi Service* antipattern, is implemented by the `component-factory-juliac` FraSCAti component. The `MembraneGeneration` service wraps SCA components with the help of `ComponentFactory` service, in this way it helps each SCA components to be treated as an individual entity. According to the specification of *Multi Service* [14], we found that the `MembraneGeneration` service had a high number of low cohesive methods defined in its interface, which might cause its frequent and large changes. Among the less change-prone antipatterns: *Bottleneck Service* (BotS), *The Knot* (TK), and *Tiny Service* (TS) show the very low number of changes and code churns. Also, the *Bloated Service* (BlS) antipattern change with a significant variation, *i.e.*, large interquartile range, in the number of changes and code churns. Based on these findings, development teams could decide to prioritise the code of services involved in the *Service Chain* (SC), *Multi Service* (MS), and *God Component* (GC) antipatterns, for special reviews and refactoring, since as shown in Figure 4, they have a high change-proneness. We have also investigated the change-proneness of the four kinds of service patterns found in FraSCAti (*i.e.*, *Basic service*, *Adapter*, *Facade*, and *Proxy pattern*) and did not find a statistically significant difference between them. Overall, as shown in $\mathbf{RQ}_1$, the services involved in service patterns are less change-prone than other services.

V. Service Patterns, Antipatterns and Object-Oriented Code Smells

Since previous studies [20], [21] have reported that classes containing object-oriented (OO) code smells change more frequently than other classes, we examine the potential confounding impact of the OO code smells contained in the implementations of the services. For each Java class implementing a FraSCAti service, we perform a detection of code smells using the well-known source code analyser PMD [22]. PMD is capable of identifying a large set of code smells based on more than 300 pre-defined rules. The main benefit of using PMD is its support for the analysis of uncompiled source code. We performed a Wilcoxon rank sum test between the different groups of Java classes, *i.e.*, (1) antipattern-classes *vs.* pattern-classes, (2) antipattern-classes *vs.* non-antipattern classes, and (3) pattern-classes *vs.* non-pattern classes and obtained statistically significant results for pattern and non-pattern classes (see Table VIII).

Table VIII. Wilcoxon rank sum test for different groups.

Comparison Classes	*p-value*
antipattern classes ∼ pattern classes	0.066
antipattern classes ∼ non-antipattern classes	0.071
pattern classes ∼ non-pattern classes	0.018

Findings: In FraSCAti, the implementations of service antipatterns contain significantly more code smells than the implementations of service patterns, and other services.

VI. Threats to Validity

In this section, we discuss the threats to validity of our study based on the guideline in [23]. *Construct validity* threats refer to the relation between theory and observation, which is apparent by the measurement errors. The identification of changes and code churns in this study is reliable because we rely on the FraSCAti mailing list archives. In this study, we only look for the number of changes and code churns for a service artefact. We plan to investigate and quantify the types of changes in the future. SODOP [5] and SODA [14] reflect their authors' subjective understanding of the service patterns and antipatterns, but they have good detection accuracy. Moreover, the service patterns and antipatterns instances used in this study were manually validated by the developers of FraSCAti, which minimises the threats to construct validity. However, other tools and techniques should be used to confirm our findings. *Internal validity* threats concern the smell detection accuracy of the PMD tool [22]. We relied on PMD because of its effectiveness in detecting code smells and duplicate codes [24]. PMD has the detection precision of more than 60% and the recall of more than 90% [25]. Another threat to this validity that might affect us, in RQ_1, for some service patterns we had very few data points. We did not investigate the reason of the introduction of service patterns or antipatterns analysed in SODOP [5] and SODA [14]. The threats to *reliability validity* concern the possibility of replicating this study. To minimise this threat, we provide all the details required to replicate the study, including the source code repositories and the raw data used to compute the statistics on our website[4].*External validity* threats concern the possibility to generalise our findings. Further validation should be done on other service-based systems (SBSs) to better analyse the impact of service patterns and antipatterns on the change-proneness. One major challenge to minimise the threat to the external validity is the very limited availability of open-source SBSs. The FraSCAti project that we have studied is the largest open-source SBS available presently. It contains 130 services and 91 SCA components. Also, we have used a representative set of service patterns and antipatterns in our study.Finally, the *conclusion validity* threats refer to the relation between the treatment and the outcome. We paid full attention not to violate the assumptions of the performed statistical tests. We mainly used non-parametric tests that do not require making any presumption about the data distribution.

VII. Related Work

In this section, we discuss the relevant literature on service patterns and antipatterns in relation to the maintenance of service-based systems (SBSs).

A number of studies have been done for the detection of service patterns and antipatterns in SBSs [5], [6], [7], [8]. For example, Tsantalis *et al.* [8] used a data mining-based approach to detect behavioral patterns by analysing execution traces. Di Penta *et al.* [7] followed a model checking-based approach for the detection of service patterns where the authors built the models from the SOAP messages exchanged between services. In another recent work, Demange *et al.* [5] performed the detection of five service patterns in SCA systems relying on the rule-based SODOP approach. Several works exist in the OO literature associating *code smells* and *antipatterns* with the change- and fault-proneness [9], [10], [11]. However, these studies are not replicable on SBSs due to several facts, including: (1) SOA is concerned with services as building blocks, whereas OO is concerned with classes, *i.e.*, services

[4] `http://sofa.uqam.ca/impact/`

are coarser than classes in terms of granularity—composed of many classes; (2) OO development mainly focuses on marshalling parameters, while SO development mostly handles request-response payloads; (3) interface development and description in OO are mostly middleware specific, on the other hand, for SO programming, it is mostly protocol specific; and, finally (4) OO development deals with homogeneous platforms and execution environments, whereas SO development deals with heterogeneous platforms and distributed execution environments. All these non-trivial differences between OO and SBSs development make it infeasible to replicate early OO studies on the SBSs. Therefore, new studies are required to relate service patterns and antipatterns to the maintenance, *e.g.*, the change-proneness, of SBSs.

VIII. Conclusion and Future Work

This paper reports on the results of an empirical study aimed at quantifying the impact of service *patterns* and *antipatterns* on the change-proneness of service-based systems (SBSs). We have performed the detection of five service patterns and 13 service antipatterns using SODOP [5] and SODA [14], respectively, and answered three research questions $\mathbf{RQ}_1$ to $\mathbf{RQ}_3$. Results show that the services involved in patterns, in terms of their implementations, are less change-prone than other services; however, this difference is not statistically significant ($\mathbf{RQ}_1$). Results also show that the services involved in antipatterns, in terms of their implementations, are more change-prone than the services that are not involved in any antipattern ($\mathbf{RQ}_2$). The services involved in *God Component*, *Multi service*, and *Service Chain* antipatterns, in terms of their implementations, are more change-prone than services involved in other kinds of service antipatterns ($\mathbf{RQ}_3$). Moreover, we observed a strong correlation between object-oriented code smells and service patterns and antipatterns—the implementations of service antipatterns contain significantly more code smells than the implementations of service patterns, and other services.

Future work includes replicating this study on other SBSs and with different service patterns and antipatterns. However, one major challenge is the availability of open-source SBSs. We are also interested in investigating the types of changes made in each FraSCAti commit and their impact on service patterns and antipatterns. Furthermore, we want to explore, using FraSCAti bug reports, the possible relation between service patterns/antipatterns and fault-proneness.

Acknowledgment

The study is supported by NSERC and FRQNT grants.

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, August 2005.

[2] ——, *SOA Design Patterns*. Prentice Hall PTR, January 2009.

[3] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf, *J2EE AntiPatterns*. John Wiley & Sons Inc, August 2003.

[4] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, November 2011.

[5] A. Demange, N. Moha, and G. Tremblay, "Detection of SOA Patterns," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds., vol. 8274. Springer Berlin Heidelberg, 2013, pp. 114–130.

[6] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and Detection of SOA Antipatterns," in 10^{th} *International Conference on Service Oriented Computing*. Springer, November 2012.

[7] M. D. Penta, A. Santone, and M. L. Villani, "Discovery of SOA Patterns via Model Checking," in *2nd International Workshop on Service Oriented Software Engineering: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. IW-SOSWE '07. New York, NY, USA: ACM, 2007, pp. 8–14.

[8] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transaction on Software Engineering*, vol. 32, no. 11, pp. 896–909, November 2006.

[9] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in *15th European Conference on Software Maintenance and Reengineering*, March 2011, pp. 181–190.

[10] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, June 2012.

[11] M. V. Mäntylä and C. Lassenius, "Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, September 2006.

[12] L. Seinturier, P. Merle, D. Fournier, V. Schiavoni, C. Demarey, N. Dolet, and N. Petitprez, "FraSCAti - Open SCA Middleware Platform v1.4 : http://frascati.ow2.org/," November 2008.

[13] M. Edwards, "Service Component Architecture (SCA)," USA, April 2011. [Online]. Available: http://oasis-opencsa.org/sca

[14] F. Palma, M. Nayrolles, N. Moha, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "SOA Antipatterns: An Approach for their Specification and Detection," *International Journal of Cooperative Information Systems*, vol. 22, no. 04, 2013.

[15] D. J. Sheskin, *Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, January 2007.

[16] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate Statistics For Ordinal Level Data: Should We Really Be Using T-Test And Cohen's D For Evaluating Group Differences On The NSSE And Other Surveys?" in *Annual Meeting of the Florida Association of Institutional Research*, February 2006, pp. 1–33.

[17] J. Cohen, *Statistical Power Analysis For The Behavioral Sciences*, 2nd ed. Lawrence Erlbaum, January 1988.

[18] N. Cliff, "Dominance Statistics: Ordinal Analyses To Answer Ordinal Questions," *Psychological Bulletin*, vol. 114, no. 3, pp. 494–509, November 1993.

[19] C. Koch, "A New Blueprint For The Enterprise," *CIO Magazine*, March 2005. [Online]. Available: Retrieved2014-05-13

[20] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes," in *19th Working Conference on Reverse Engineering*, ser. WCRE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 437–446.

[21] F. Khomh, M. Di Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, Oct 2009, pp. 75–84.

[22] A. Dangel and R. Pelisse, "PMD," Tech. Rep., May 2013. [Online]. Available: http://pmd.sourceforge.net/

[23] R. K. Yin, *Case Study Research: Design and Methods*. SAGE Publications, Inc, May 2013.

[24] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ser. ISSRE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 245–256.

[25] H. Hata, O. Mizuno, and T. Kikuno, "Comparative Study of Fault-Proneness Filtering with PMD," in *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, Nov 2008, pp. 317–318.