Delft University of Technology
Software Engineering Research Group
Technical Report Series

Analyzing the Change-Proneness of Service-Oriented Systems from an Industrial Perspective

Daniele Romano

Report TUD-SERG-2013-001

TUD-SERG-2013-001

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the 35th International Conference on Software Engineering, 2013.

Analyzing the Change-Proneness of Service-Oriented Systems from an Industrial Perspective

Daniele Romano
Software Engineering Research Group
Delft University of Technology
Delft, The Netherlands
daniele.romano@tudelft.nl

***Abstract*—Antipatterns and code smells have been widely proved to affect the change-proneness of software components. However, there is a lack of studies that propose indicators of changes for service-oriented systems. Like any other software systems, such systems evolve to address functional and non functional requirements. In this research, we investigate the change-proneness of service-oriented systems from the perspective of software engineers.**

Based on the feedback from our industrial partners we investigate which indicators can be used to highlight change-prone application programming interfaces (*API*s) and service interfaces in order to improve their reusability and response time. The output of this PhD research will assist software engineers in designing stable *API*s and reusable services with adequate response time.

***Index Terms*—Service-oriented systems, web services, WSDL, metrics, Antipatterns, change-proneness**

I. Research Context

Several years of research on software maintenance have produced numerous approaches to identify and predict change-prone components in a software system. Among others, source code metrics and heuristics to detect antipatterns and code smells have been widely validated as indicators of changes. However, these indicators have been mainly proposed and validated for object-oriented systems. There is still the need to define and validate indicators of changes for systems implemented in other programming paradigms such as the service-oriented one.

In recent years there has been a tendency to adopt Service-Oriented Architectures (*SOA*s) in companies and government organizations for two main reasons. First, *SOA*s allow companies to organize and use distributed capabilities (*i.e.*, services) that may be under the control of different organizations or different departments within the same organization [1]. Second, organizations benefit from the loose coupling between clients and services. However, clients and services are still coupled and changes in the services can impact negatively their clients and entire systems. The dependencies removed in *SOA*s are the dependencies between clients and the underlying technologies used to implement services. Clients and services are still coupled through *function coupling* and *data structure coupling* [2]. In fact, clients depend 1) on the functionalities implemented by services (*i.e.*, *function coupling*) and 2) on the data structures that a service's instance receives and returns (*i.e.*, *data structure coupling*) and that are specified in its interface. For this reason service interfaces are considered contracts between clients and service providers and they should remain as stable as possible. However, like any other software component, services evolve to satisfy functional and non functional requirements. To the best of our knowledge there are no studies that show the impact of badly designed service interfaces on the change-proneness of service-oriented systems. As a matter of fact software engineers encounter difficulties in designing the right interfaces for their services.

In this PhD research we investigate the change-proneness of service-oriented systems from the perspective of software engineers. Thanks to the precious input from our industrial partners we focus on the main problems software engineers face while designing a service-oriented system that can affect the change-proneness.

In particular, we investigate two main scenarios that can cause changes. First, changes in the implementation logic can cause changes in the service interfaces, especially when legacy *API*s are made available through service interfaces. Second, service interfaces can be refactored to improve two quality attributes, namely reusability and response time. Non reusable interfaces are more prone to change when new clients need existing functionalities but they request different interfaces. Response time is the main concern in designing XML based interfaces (*e.g.*, WSDL interfaces). After publishing these interfaces they can undergo changes to improve their response time.

In this paper we propose a research approach (Section II) to investigate indicators of changes (*e.g.*, heuristics and software metrics) that highlight: 1) change-prone *API*s (track **T1**) and 2) service interfaces likely to be changed to improve reusability and response time (tracks **T2** and **T3**).

The main goal of this research is to assist software engineers in designing stable *API*s and reusable services with adequate response time. The three main research questions are:

- *Which (existing) quality indicators can be used to high-*

light the change-prone parts of an API?
- *Which quality indicators can be used for highlighting change-prone service interfaces?*
- *To what extent can these quality indicators be used to improve the reusability and response time of services?*

II. Research Approach

This research is divided in three tracks that are presented in the following subsections.

T1. Analysis of the Change-Proneness of *API*s. Each service is implemented by an implementation logic that is hidden to service clients through its interface. Changes to the implementation logic can be propagated and affect the service interface. Among all the software units composing the implementation logic, *API*s are likely to be mapped directly into service interfaces. This scenario happens especially when a legacy *API* is made available through a service interface. For this reason, in the first track we analyze the change-proneness of *API*s, where we refer to *API* as the set of public methods declared in a software unit. To perform this study we use existing techniques to mine software repositories and to extract changes performed in the *API*s. We then analyze whether there is a correlation between the amount of changes an *API* undergoes and the values of source code metrics and/or the presence of antipatterns and code smells. The outcome of this study will consist in a set of quality indicators (*e.g.*, heuristics and software metrics) that can highlight change-prone *API*s and assist software engineers to design stable *API*s. In our context, these indicators are particularly useful to check the stability of *API*s when they are exposed as service interfaces.

T2. Analysis of the Change-Proneness of Service Interfaces. The second track consists in investigating the change-proneness of service interfaces through the analysis of their evolution. This analysis can help us in identifying bad design practices that can increase the probability that a service interface will be changed in the future. Among all the changes a service interface undergoes we focus on the changes performed to improve reusability and response time.

To perform this study we detect and extract changes performed in the interfaces. This task is performed by a tool that compares two subsequent versions of a service interface and extracts changes taking into account the syntax of the interface specification. In this way we can extract the type of a change performed in the interface. Knowing the type of a change is particularly useful for two reasons. First, we can see which element is affected by the change and how it changes. Second, we can classify the changes depending on the impact they can have on the clients. In fact, changes can be divided into *breaking changes* and *non-breaking changes* depending on whether service client developers need to update their code or not [2].

Once we are able to extract and classify changes we investigate heuristics and software metrics that can be used as indicators of low reusable service interfaces with inadequate response time. Similar to **T1**, we then investigate the correlation between them and the changes performed in the service interfaces.

We start measuring and investigating the impact of *coupling* and *SOA antipatterns* on change-proneness. As already described in Section I, even though services are loosely coupled they are still coupled through *function* and *data structure coupling*. We believe that *coupling* can be a good quality indicator in service-oriented systems like it has been already proved for systems implemented in other programming paradigms. We expect that a service with a higher incoming and outgoing *coupling* can show a higher response time. However, measuring *coupling* in service-oriented systems is more challenging than for systems implemented in other paradigms. This is mainly due to the dynamic and distributed nature of service-oriented systems.

Besides *coupling*, we plan to analyze other attributes that can affect change-proneness such as *granularity*, *cohesion* and *duplication*. We believe that designing a service interface with the right *granularity* is a challenging step. On the one hand the data types encoding should be designed for different clients to improve service's reusability. On the other hand the data types encoding can affect the response time of a service.

Furthermore, we believe that a service interface should be cohesive to prevent changes in the future. A low cohesive interface can affect the comprehension of the interface resulting in a lower reusability. Moreover, an interface with different responsabilities can be a bottleneck that can affect response time because of the different clients invoking it.

Finally, *duplication* of service operations can be risky similarly to code clones, widely studied in the software engineering community. Duplicated entities can impact the comprehension and, hence, the reusability.

To analyze the impact of these attributes on change-proneness we plan to start analyzing existing antipatterns defined in literature. Based on our findings we will refine them and propose new antipatterns.

The outcome of this study will consist in a set of heuristics and metrics that can assist software engineers in designing service interfaces that are reusable, show an adequate response time, and are less change-prone.

T3. Services Categorization. An important step to understand why service interfaces change over time is understanding their purpose. In this track we extend our research in **T2** taking into account the service typologies. We study and define heuristics to classify service interfaces into different typologies, as suggested by Krafzig *et al.* [3]. We expect that some service typologies change less frequently and for different reasons than others. For instance, the interface of a service that is meant to bridge a technological gap would change only when the bridged technologies change. On the other hand, the interface of a service that provides search functionalities can change every time that the search criterion changes.

To automatically classify services we can analyze two sources of information. First, we analyze the documentations

that are usually available in natural language and published on websites. For instance, Google Maps web services are documented on their website.[1] The second source of information consists of the service interface that is composed of: 1) method declarations, 2) data types needed to invoke the methods and to retrieve the results, and 3) comments to ease the comprehension of a service interface. To obtain relevant information from these two sources we plan to use information retrieval techniques, widely used in the software engineering community for similar purposes.

Similar to **T2**, the outcome of this track will consist in a refined set of heuristics and metrics to assist software engineers in designing reusable service interfaces with adequate response time. However, the heuristics and metrics investigated in this track are validated taking into account the service typologies, obtaining specific sets of heuristics and metrics for each different typology of service.

III. Validation Strategy

The research questions of each track will be addressed using the mixed-methods approach [4] which is a combination of quantitative and qualitative methods. In each track we plan to perform empirical studies with open and industrial software projects using statistics, machine learning, and information retrieval techniques, similar to those that have been already used in [5], [6]. The results will be validated with questionnaire data gathered in academic and in industrial environments. In addition to questionnaires, we plan to perform interviews and case studies with industrial software engineers.

IV. Progress

My PhD research started in November 2010 under the supervision of Prof. Dr. Martin Pinzger and in collaboration with the Software Improvement Group[2] and KPMG[3] IT departments located in Amsterdam. We initially set up this research project meeting our industrial partners to understand the problems they face in designing service-oriented systems. Based on their inputs and on our research interests we decided to start the project focusing on tracks **T1** and, partially, on track **T2**.

A. T1: Analysis of the Change-Proneness of APIs

To analyze the change-proneness of *API*s we performed two analyses aimed at validating source code metrics and the presence of antipatterns as indicators of changes in *API*s.

In our first work [5] we analyzed the change-proneness of Java interfaces. We mined the source code repositories of 10 Java open source projects extracting the fine-grained source code changes performed in Java interfaces. We then correlated the number of changes with the values of source code metrics measured in the interfaces. Among all the metrics analyzed, the *Interface Usage Cohesion (IUC)* metric [7] showed the best correlations and improved the performance of prediction models for classifying *change-* and *not change-prone* interfaces.

In our second work [6] we analyzed the change-proneness of Java classes affected by antipatterns. We extracted the fine-grained source code changes from the source code repositories of 16 Java open source projects. We then analyzed the correlation between different types of changes and different antipatterns affecting Java classes. The results showed that classes participating in the *ComplexClass*, *SpaghettiCode* and *SwissArmyKnife* antipatterns are more likely to undergo changes in their *API*s.

B. T2: Analysis of the Change-Proneness of Service Interfaces

In the second track of our project (**T2**) we investigate the change-proneness of service interfaces, in a similar way to which it is done in the first track (**T1**) to analyze change-prone Java *API*s. To perform this analysis we need two important information: 1) the changes between the different versions of a service interface, and 2) the dependencies between services to compute the *coupling* and antipatterns defined for service interfaces.

To extract changes we developed a tool called *WSDLDiff* that extracts fine-grained changes from subsequent versions of WSDL interfaces. Differently to existing XML differencing tools, *WSDLDiff* takes into account the syntax of the WSDL specification. This allows us to extract information about the element changed in the WSDL and the type of change performed. A first analysis showed the relevance of using fine-grained changes in analyzing the evolution of WSDL interfaces. We decided to focus on WSDL interfaces because the service-oriented systems developed or maintained by our industrial partners are mainly composed of WSDL interfaces. The details of our *WSDLDiff* can be found in [8].

To extract the dependencies among services we developed an approach based on vector clocks [9]. Vector clocks have been originally conceived to order events in a distributed environment. In our work we use this technique to extract dynamic dependencies among services deployed in an enterprise. Our approach has been implemented into the Apache CXF framework using the *Pipes and Filters* pattern. We use this pattern because it is widely used in the most popular web service frameworks and Enterprise Service Buses (*ESB*), making our approach portable to other SOA platforms (e.g., Apache Axis2 and Mule ESB).

C. Current and Future Work

Currently we are working on further investigating track **T1** and **T2**, while track **T3** will be investigated in the near future.

Concerning track **T1**, we are extending our previous work [5], [6]. We are analyzing which source code metrics can be used as indicators of changes in *API*s exposed by concrete Java classes, similar to what we have already done for Java interfaces [5]. Moreover, we plan to extend our dataset to investigate the change-proneness of *API*s that are mapped to service interfaces. To conclude **T1** we will perform a qualitative analysis aimed at validating our results with questionnaire data gathered in both academia and industrial environments.

[1] https://developers.google.com/maps/documentation/webservices/
[2] http://www.sig.eu/en/
[3] http://www.kpmg.com/nl/en/pages/default.aspx

Concerning track **T2**, we are analyzing the change proneness of public available web services, like the services analyzed in our previous study [8]. We are mainly interested in investigating the impact of service *cohesion* and *granularity* on the change-proneness. As part of this analysis we are integrating existing techniques into our *WSDLDiff* tool to classify changes based on their impact on existing clients. As next step we plan to visit our industrial partners to analyze their industrial service-oriented systems. This allows us to further investigate track **T2** taking into account the dependencies among services.

Having access to industrial systems will allow us to perform the research on track **T3**. In track **T3** we plan to define heuristics and techniques to classify the services according to their purpose. This is useful to refine the results achieved in the track **T2**.

V. Related Work

In literature many studies propose quality indicators for service-oriented systems. However, these indicators have been poorly validated mainly because of the lack of availability of such systems. Furthermore, to the best of our knowledge, there are no studies that propose quality indicators to estimate the change-proneness of service-oriented systems.

Perepletchikov *et al.* defined a set of *cohesion* and *coupling* metrics for service-oriented systems. In [7] they analyzed *cohesion* in the context of SOA and they proposed four different types of *cohesion* metrics for measuring analyzability. Furthermore, they proposed three different *coupling* measures for SOA [10] and they showed their impact on maintainability.

The most recent work on SOA antipatterns has been proposed by Moha *et al.* in 2012 [11]. They proposed an approach to specify and detect an extensive set of SOA antipatterns that encompass concepts like *granularity*, *cohesion* and *duplication*. Their tool is capable to detect the most popular SOA antipatterns defined in literature. Besides these antipatterns, they specified three more antipatterns, namely: *bottleneck service*, *service chain* and *data service*. *Bottleneck service* is a service used by many services and it is affected by a high incoming and outgoing *coupling* that can affect response time. *Service chain* appears when a business task is achieved by a long chain of consecutive services invocations. *Data service* is a service that performs simple information retrieval or data access operations that can affect the *cohesion*.

In 2012 Rotem-Gal-Oz [12] defined *the knot* antipattern as set of low cohesive services which are tightly coupled. This antipattern can cause low usability and high response time.

The *sand pile* defined by Král *et al.* [13] appears when many fine-grained services share common data that may be available through a service affected by the *data service* antipattern.

Cherbakov *et al.* proposed the *duplicate service* antipattern [14] that affect services sharing similar methods and that can cause maintainability issues.

In 2003 Dudney *et al.* [15] defined a set of antipatterns for J2EE applications. Among these we plan to investigate the *multi service*, *tiny service* and *chatty service* antipatterns. The *multi service* is a service that provides different business operations that are low cohesive and can affect availability and response time. *Tiny services* are small services with few methods that are used together. This antipattern can affect the reusability of such services. Finally the *chatty service* antipattern affects services that communicate with each other with small data. This antipattern can affect the response time.

VI. Contributions

The expected contributions of this PhD research can be summarized as follows:

- A set of validated quality indicators, comprising metrics, heuristics, and techniques and tools, to highlight the change-prone parts of an *API*;
- A set of validated quality indicators, comprising metrics, heuristics, and techniques and tools to highlight change-prone service interfaces;
- Design guidelines, heuristics, techniques, and tools to assist software engineers in designing reusable services with adequate response time.

The tools will be publicly available allowing other researchers to perform analyses based on the evolution of service-oriented systems.

Acknowledgment

This work has been partially funded by the NWO-Jacquard program within the ReSOS project.

References

[1] P. F. Brown and R. M. B. A. Hamilton, "Reference model for service oriented architecture 1.0," 2006.
[2] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Pearson Education, 2011.
[3] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.
[4] J. Creswell and V. Clark, *Designing and Conducting Mixed Methods Research*. SAGE Publications, 2010.
[5] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *ICSM*, 2011, pp. 303–312.
[6] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," in *WCRE*, 2012, pp. 437–446.
[7] M. Perepletchikov, C. Ryan, and Z. Tari, "The impact of service cohesion on the analyzability of service-oriented software," *IEEE T. Services Computing*, vol. 3, no. 2, pp. 89–103, 2010.
[8] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *ICWS*, 2012, pp. 392–399.
[9] D. Romano, M. Pinzger, and E. Bouwers, "Extracting dynamic dependencies between web services using vector clocks," in *SOCA*, 2011, pp. 1–8.
[10] M. Perepletchikov and C. Ryan, "A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software," *IEEE Trans. Software Eng.*, vol. 37, no. 4, pp. 449–465, 2011.
[11] N. Moha, F. Palma, M. Nayrolles, B. Joyen Conseil, Y.-G. Yann-Gael, Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and Detection of SOA Antipatterns," in *International Conference on Service Oriented Computing*, Shanghai, Chine, Nov. 2012.
[12] A. Rotem-Gal-Oz, *SOA Patterns*, 1st ed. Manning Pubblications, 2012.
[13] J. Král and M. Zemlicka, "The most important service-oriented antipatterns," in *ICSEA*, 2007, p. 29.
[14] L. Cherbakov, M. Ibrahim, and J. Ang, "Soa antipatterns: the obstacles to the adoption and successful realization of service-oriented architecture."
[15] B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, and D. Osborne, *J2EE Antipatterns*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2002.

TUD-SERG-2013-001
ISSN 1872-5392