Web Service Growing Pains: Understanding Services and Their Clients

Tiago Espinha

© 2014, Tiago Espinha

Thesis style design: Tiago Espinha
Cover design: Tiago Espinha
Printed by: CPI Koninklijke Wöhrmann

In memory of my grandfather,
José Aurélio Mourinho Rodrigues

Web Service Growing Pains: Understanding Services and Their Clients

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 20 maart 2015 om 10:00 uur

door

Tiago ESPINHA

Master of Science
University of Leicester, United Kingdom
geboren te Marinha Grande, Portugal.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. A. van Deursen

Copromotor Dr. A.E. Zaidman

Samenstelling promotiecommissie:

Rector Magnificus	Voorzitter
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. A.E. Zaidman	Delft University of Technology, copromotor
Prof. dr. G.-J. Houben	Delft University of Technology
Prof. dr. ir. M.F.W.H.A. Janssen	Delft University of Technology
Prof. dr. B. Adams	École Polytechnique de Montréal, Canada
Dr. S. Jansen	Utrecht University, The Netherlands
Dr. H.-G. Gross	Esslingen University, Germany

This work was carried out as part of the Jacquard ScaleItUp project, sponsored by the Netherlands Organisation for Scientific Research (Nederlandse Organisatie voor Wetenschappelijk Onderzoek — NWO).

ISBN: 978-94-6186-406-2
Copyright © 2014, Tiago Espinha

Thesis style design: Tiago Espinha
Cover design: Tiago Espinha
Cover artwork: © Depositphotos.com/lightsource
Printed by: CPI Koninklijke Wöhrmann

Contents

1 Introduction **3**
1.1 From the Internet to Web Services . . . 3
1.1.1 Simple Object Access Protocol (SOAP) . . . 4
1.1.2 Representational state transfer (REST) . . . 4
1.1.3 Web Services . . . 4
1.2 Terminology . . . 5
1.3 Problem Statement . . . 7
1.3.1 Research Questions . . . 8
1.3.2 Research Methods . . . 9
1.4 Thesis Outline . . . 11
1.5 Origin of Chapters . . . 11
1.6 Additional Publications . . . 11

2 Web APIs: Loosely Coupled yet Strongly Tied **13**
2.1 Introduction . . . 13
2.2 Terminology . . . 15
2.2.1 SOAP . . . 15
2.2.2 REST . . . 15
2.2.3 JSON-RPC & JSON . . . 16
2.3 Experimental Setup . . . 16
2.3.1 Experimental Setup for the Client-Side Investigation . . . 17
2.3.2 Experimental Setup For The End-to-End Analysis . . . 20
2.4 Client-side Analysis . . . 21
2.4.1 Interviews With Client Developers . . . 21
2.4.2 Web API Characteristics . . . 26
2.4.3 Impact on Client Code . . . 28
2.5 End-to-End Analysis . . . 34

2.5.1 VirtualBox . . . 34
2.5.2 XBMC . . . 37
2.6 Discussion . . . 41
2.6.1 Answering the Research Questions . . . 41
2.6.2 Recommendations . . . 43
2.6.3 Threats to validity . . . 46
2.7 Related work . . . 46
2.8 Conclusion . . . 48

3 Web API Clients: How Robust is Yours 51
3.1 Introduction . . . 51
3.2 Approach . . . 53
3.2.1 Mutation Analysis — Mutant Generation . . . 53
3.3 Experimental Setup . . . 57
3.3.1 Application Selection . . . 57
3.3.2 Applying the Mutations . . . 58
3.3.3 Caching and Versioning . . . 60
3.3.4 Developer Interviews . . . 60
3.4 Experimental Results . . . 61
3.4.1 Application Behavior . . . 62
3.4.2 Behaviors Per Mutation Type . . . 62
3.4.3 Data Caching . . . 66
3.4.4 Versioning . . . 66
3.4.5 Developer Interviews . . . 67
3.5 Threats to Validity . . . 69
3.6 Related Work . . . 70
3.7 Conclusion . . . 70

4 SOA: Proposing a Standard Case Study 73
4.1 Introduction . . . 73
4.2 Background Research . . . 74
4.3 Stonehenge . . . 78
4.3.1 Motivation . . . 78
4.3.2 System Description . . . 78
4.3.3 Usage Scenarios . . . 80
4.4 Research Agenda . . . 81
4.4.1 Online Updating and Versioning . . . 81
4.4.2 Online Diagnosis and Testing . . . 82
4.5 Summary . . . 82

5 SOA: Understanding its Runtime Topology 85
5.1 Introduction . . . 85
5.2 Approach . . . 87

5.2.1 Required data 87
5.2.2 Data extraction 89
5.2.3 Data presentation 90
5.2.4 Serviz 91
5.3 Experimental Setup 91
5.3.1 One-group pretest-posttest 91
5.3.2 Assignment 94
5.3.3 Pilot 96
5.4 Results 96
5.4.1 Pretest Data 96
5.4.2 Posttest Data 98
5.5 Discussion 101
5.5.1 Revisiting the Research Questions 101
5.5.2 Threats to validity 102
5.6 Related work 103
5.7 Conclusion 104
5.7.1 Future work 104

6 SOA: Users and Versions in Multi-Tenant Systems 107
6.1 Introduction 107
6.2 Approach 109
6.2.1 Data Requirements 110
6.2.2 Data Extraction 110
6.3 Serviz 111
6.3.1 User Filtering 112
6.3.2 Service/Version Filtering 112
6.3.3 Combined Filtering 112
6.3.4 Histograms 113
6.4 Experimental Setup 113
6.4.1 Case Study System 114
6.4.2 Questionnaire 115
6.4.3 Participants 116
6.5 Results 116
6.5.1 General Questions 116
6.5.2 Generic Software Engineering Questions 118
6.5.3 User Filtering 119
6.5.4 Service Filtering 120
6.5.5 Version Filtering 120
6.5.6 Combined Filtering 121
6.6 Discussion 122
6.6.1 The Research Questions Revisited 122
6.6.2 Lessons Learned 123
6.6.3 Threats to Validity 123

6.7 Related Work . . . 124
6.8 Conclusion . . . 125
6.8.1 Future Work . . . 125

7 Conclusion 127
7.1 Summary of Contributions . . . 128
7.2 The Research Questions Revisited . . . 129
7.2.1 RQ1 — How do web service APIs evolve and what are the consequences for clients of web APIs? . . . 129
7.2.2 RQ2 — How well-prepared are Android mobile applications with regard to changes in response messages from the web API? . . . 131
7.2.3 RQ3 — How can the topology of a running SOA-based system help in its maintenance? . . . 132
7.2.4 RQ4 — Does the combination of user, service-version and timing information projected on a runtime topology help in the understanding of SOA-based multi-tenant software systems? . . . 133
7.3 Recommendations For Future Work . . . 133
7.3.1 Automated Web API Evolution . . . 133
7.3.2 Versioning Data versus Versioning Interfaces . . . 134
7.3.3 Runtime Topology . . . 134
7.3.4 Dead Code Warnings . . . 136
7.3.5 Web API Documentation Mining . . . 136
7.3.6 Metrics for Web API Frequency of Change . . . 137
7.3.7 Coupling Metrics for Organizational Co-evolution . . . 137
7.3.8 Per-user, At-will Web API Version Migration . . . 137
7.3.9 Closed versus Open-source . . . 138

Acknowledgement 139

Summary 141

Samenvatting 145

CHAPTER 1

Introduction

1.1 From the Internet to Web Services

The creation of the Internet in the 1980s paved the way for the connected world as we know it today. The Internet, which as of December 2013 was connecting in excess of 2.8 billion people[1], opened the door not only for the relatively fast sharing of any sort of data between any number of individuals but it also currently allows for an increasing globalized market. Indeed, it is now possible for any individual equipped with an ever-pervasive Internet connection to be able to engage in business with any store providing an *online* storefront.

Similarly, the Internet is also nowadays used as a channel for machine-to-machine interaction. This interaction, while seemingly simple, through the usage of an inherently heterogeneous range of different operating systems, programming languages and hardware architectures poses in fact a challenge [Vinoski, 1997]. For example, unless the software system behind the online storefront is able to automatically interact with the software system of a payment processing entity (e.g., a bank), all orders must still be manually verified and physically submitted for further payment processing. In this example, it is also unrealistic to expect all online storefronts and all banks to use exactly the same technologies for the sake of easier software interoperability.

Throughout time multiple technologies emerged to facilitate communication between such heterogeneous systems. The Common Object Request Broker Architecture (more commonly known as CORBA) is a well known example of such a technology [Vinoski, 1997]. However, a number of reasons condemned CORBA to lose ground to other more open technologies. The lack of standard implementation in the wide array of existing programming languages, the existence of competing technologies such as Microsoft's Dis-

[1] Internet World Stats — http://www.internetworldstats.com/stats.htm, last visited: August 25, 2014

tributed Component Object Model (DCOM) and the advent of the Extensible Markup Language (XML) all contributed to CORBA's eventual fall into disuse [Henning, 2008].

After it became clear that CORBA, Microsoft's DCOM, and a panoply of other similarly flavored middleware technologies did not fully fulfill the need of homogenizing software interactions, two technologies were created which are nowadays widely used in machine-to-machine interaction under the terminology of *"web services"*.

1.1.1 Simple Object Access Protocol (SOAP)

Microsoft together with DevelopMentor made use of XML to build what would become SOAP (Simple Object Access Protocol). SOAP is a protocol which enables machine-to-machine message exchanging as well as remote procedure calls between two devices [Curbera et al., 2002]. This protocol, which would later become a W3C[2] standard, brings with it the terminology of *"web services"*. Web services as defined by the W3C consortium[3] are *"software systems designed to support interoperable machine-to-machine interaction over a network"*. The full definition goes further and specifies that a web service must also make use of the SOAP approach for the network interactions. The SOAP approach for remote procedure calls does make use of the *web service* terminology. Each SOAP web service resorts to a so-called WSDL file which, named after the language it is written in (*Web Services* Description Language), is used to describe web service interfaces [Henning, 2008].

1.1.2 Representational state transfer (REST)

In 2000, Roy Fielding created another usable style for the development of web services [Nolan and Lang, 2014]. Whilst originally devising an approach to solve a different issue altogether (designing the architecture of a distributed hypermedia system [Fielding, 2000]), the Representational State Transfer architectural style is nowadays used by some web service providers as the *only* web services approach (a prominent example is Twitter which provides no SOAP support). Generally, RESTful web services serve much of the same end-goal as their SOAP counterpart: message exchange and remote procedure calls [Pautasso et al., 2008]. Academic research in the field of RESTful interfaces recognizes this architectural style can also be used for the development of *"web services"* [Pautasso et al., 2008] [Christensen, 2009] [Daigneau, 2011].

1.1.3 Web Services

With the two aforementioned major architectural styles setting the guidelines for what *web services* are and with multiple implementations existing for both web service ap-

[2]The W3C is the World Wide Web Consortium which together with several member organizations such as Microsoft, Apple, Google (to name a few) sets standards for the World Wide Web. Examples of such standards are HTML, XML and CSS which are widely used in web development.

[3]Web service — http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice

proaches, web services escalated from being yet another middleware approach to being the center of attention in both mobile and browser-based Internet-enabled applications. Indeed, due to the increasingly central role Internet connectivity plays in mobile applications, Google has now hidden the *"Internet permission"* from the main installation screen in the Android platform[4]. Moreover, some of the major Internet companies (e.g. Amazon, Facebook, Twitter, Microsoft, and many others) nowadays provide web services to facilitate access to their data and to harness third-party functionality into their own software systems.

The challenges that come with the continued usage of web services from both the perspectives of web service providers and consumers are explored in this thesis.

1.2 Terminology

In this section we clarify some of the terminology used in this thesis. Whenever available, an established academic definition is provided. When such a definition is not available and for terms whose meaning may lead to ambiguity, we clarify the meaning that is used in the subsequent chapters.

Service-Based Systems

Existing literature and research on web services bring in many instances the terms *web services* and *Service Oriented Architectures* (SOA) together. However, as Lewis and Smith point out [Lewis and Smith, 2008], SOA *"is a way of designing, developing, deploying and managing systems"* which does not cover all the uses industry and academia alike have found for web services. In many cases, businesses make use of web services to expose functionality from their existing architectural implementation. When this is the case, the existing architecture is not necessarily *"designed, developed, deployed and managed"* with service-orientation in mind and it would then be a misnomer to refer to those systems as making use of a Service Oriented Architecture. For this reason, throughout the subsequent chapters, we make use of the more generic term *"service-based system"*. This term is used in existing literature [Mahbub and Spanoudakis, 2004] [Calinescu et al., 2011] and despite never being formally defined, it is used to refer to any software system which makes use of web services.

Web Application Programming Interface (Web API)

An Application Programming Interface (API) provides a consistent and predictable software interface between its provider and consumer [Jacobson et al., 2011]. In essence, web services which are defined as *"software systems designed to support interoperable machine-to-machine interaction over a network"* restrict the concept of the interface to

[4]Simplified permissions on Google Play — `https://support.google.com/googleplay/answer/6014972?p=app_permissions`, last visited October 30th 2014

be bound by a network (or colloquially, a web). The combination of these two existing terms generated another commonly used term for web services: by drawing the *web* from web services and *API* from a consistent interface with providers and consumers, we arrive at the term *web API*. While no academic literature appears to exist which clearly states web services and web APIs refer to the same underlying idea, Wikipedia states that a web API *"in [the context of server-side] is sometimes considered a synonym for web service"*[5]. For convenience, on a case-by-case basis in the subsequent chapters we reuse the terminology already being used in the systems under study, thus frequently interchanging between web services and web APIs.

Multi-Tenant Software Systems

Another term used in parts of this thesis is *multi-tenant software systems*. When referring to these, we use the definition provided by Kabbedijk et al. [2014a] who define multi-tenant software systems as systems *"where multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant"*. An example of this is the Amazon Webstore[6] where Amazon allows third-parties to build customized online shops both using Amazon's core eCommerce technology (used on Amazon's own online shops) and running on Amazon's hardware.

Runtime Topology

In this thesis we make use of the term *"runtime topology"*. The topology, in the context of computer networks, is defined by Groth and Skandier [2005] as *"the physical and/or logical layout of the transmission media specified in the physical and logical layers"* of the network. In the context of the thesis, we analyze the topology of service-based systems as a means to identify how the web services in such a system interact. Our addition of the *"runtime"* terminology stems from such a system being able to, at runtime, reconfigure itself into a different *"topology"*. The *"runtime topology"* is then defined as the configuration of a distributed, dynamically composable software system which describes what services are available, how they depend on each other and interact with each other.

Breaking Changes

We often refer to *breaking changes* throughout the thesis. The definition of a breaking change is a change to an interface which is not backwards compatible with previous versions of the same interface. The exact changes we consider as breaking changes are aligned with those identified by Dig and Johnson [2006] in the context of statically linked APIs. Whenever we refer to breaking changes in the context of web APIs we do filter out

[5] Web API — http://en.wikipedia.org/wiki/Web_API, last visited September 9th 2014
[6] Amazon Webstore — http://webstore.amazon.com, last visited September 9th 2014

the changes which are not applicable in the web API context (e.g., pulled up method or new hook method).

Our reference for what types of changes are more common is also supported by the work of Wang et al. [2014]. In the context of RESTful web APIs where the authors performed a frequency analysis to identify which are the most common breaking changes.

1.3 Problem Statement

As more and more systems start to provide web service interfaces as a means for integration with other services whilst themselves integrating services from third-parties, the software dependencies and interactions in the emerging field of web services opened the door to an interesting research field.

Starting from the findings of Lehman and Belady [1985] who observe that software systems must evolve to stay successful, we ask what actually happens if one of these web services changes? An even graver question is then: what if the change breaks backwards compatibility? This is therefore an example of the *growing pains* that affect web service developers and client developers alike. By allowing web services to grow and mature, web service providers may sometimes be compelled to change the behavior of their web services which then results in added development effort for client developers.

This added effort for client developers is further fueled by a major shift in who controls the pace of software evolution. So far, developers making use of a statically linked API could choose not to migrate to a newer version (and thus postpone any additional effort for integration with the new version). The findings of Laitinen [1999] carry an even stronger message and state that unless the return on investment is high, typically developers will not migrate to a newer version. However, when the web service provider decides to roll out a new version of a web service it is the provider who decides if and how long older versions of the web service remain available. Depending on how *considerate* the web service provider is of their client developers, this power shift means that in some cases client developers can no longer afford the inertia found by Laitinen and it may therefore be a source of distress for such client developers.

The findings of Lehman and Belady on the inevitability of software evolution together with those of Laitinen stating that developers will only migrate to newer versions of software libraries if there is a high return on investment reveal that such web service growing pains bring web service providers and web service client developers onto a collision course. On one side the web service providers are eager to evolve their web service and potentially push breaking changes while on the other side there are web service client developers who would normally not have migrated to the newer version and who now are forced to migrate.

As the evolving web services become an inalienable reality for both providers and client developers, another concern arises: how can web service providers better understand the interactions of their own service-based systems in order to, with near-zero-downtime for web service clients, carry out the inevitable evolution process? The need for near-

zero-downtime whilst evolving and maintaining such systems is emphasized by the high financial cost which downtime has on software systems (estimated at over $5,000 per minute on average[7]). This focus on near-zero-downtime is further emphasized by the growing body of academic literature [Cheng et al., 2005] and patents [Lin, 2006] [Borissov et al., 2010] which set forward methodologies to achieve such near-zero-downtime.

1.3.1 Research Questions

In this thesis we investigate questions regarding the evolution and maintenance of service-based systems as well as the pains client developers face when such systems start to evolve. This thesis is then centered around our **main research question**, which asks:

> *"What can web service providers and client developers alike, do to minimize the pains of web service evolution?"*

In order to find answers to this research question we rely on four subsidiary research questions:

RQ1 asks *"how do web service APIs evolve and what are the consequences for clients of web APIs?"*. In order to study *how* web service APIs evolve we attempt to draw commonalities in the evolution policies of high-profile web API providers. We also investigate the *consequences* caused by these evolution steps by investigating what are concrete pains client developers experience when their client had to deal with such an evolution task instigated by a web API provider.

This question is answered in Chapter 2 where we show the results of our interviews with six professional developers regarding their experiences on dealing with web API evolution and where we present our findings regarding web API evolution policies.

RQ2 asks *"how well-prepared are Android mobile applications with regard to changes in response messages from the web API?"*. Mobile applications' growth in usage and general reliance on web APIs turn them into indicators of the current state of the practice on what concerns readiness (or lack thereof) to the sometimes unexpected changes pushed by web API providers.

This question is answered in Chapter 3 where we present the results of our exploratory study consisting of inserting disturbances in the web API responses of 48 Android applications as a means to probe their robustness and resilience to change. This study is complemented by interviews with three developers of such applications.

RQ3 asks *"How can the topology of a running SOA-based system help in its maintenance?"*. With this research question we step over to the web API *provider's* point of view. While maintaining a web service-based system, it is not always trivial to debug such a system or to identify which services are involved in a specific use case. Another aspect explored in the context of this research question is whether analyzing the runtime topology as it changes along the axis of time helps in the goal of near-zero-downtime.

[7] "How much does downtime cost?" blog post with infographic data drawn from sources at the Ponemon Institute, Standish Group, Gartner and Dataquest — http://www.appdynamics.com/blog/devops/how-much-does-downtime-cost/, last visited August 20th 2014

In order to answer this research question we first investigate which service-based system would be suitable to test the aforementioned hypothesis. Our investigation of existing research on the area of web services reveals the lack of a standard platform on which different researchers could experiment. This need led us to create Spicy Stonehenge which is presented in Chapter 4.

This question is then further answered in Chapter 5 through our implementation of the runtime topology as well as a pretest-posttest experiment aimed at evaluating the usefulness of such an approach.

RQ4 asks *"does the combination of user, service-version and timing information projected on a runtime topology help in the understanding of SOA-based multi-tenant software systems?"*. The motivation behind investigating the usefulness of user, service-version and timing data in a runtime topology stemmed from how multi-tenant software systems are often used by different users (potentially in different timezones) who in turn use slightly modified versions of the existing web services to address their specific use cases. We therefore extended the runtime topology mentioned in the previous research question to include this data. Therefore, we investigate whether these added dimensions of users and versions as well as the time help system maintainers in understanding how a particular multi-tenant software system is used by its clients in order to facilitate such tasks as debugging and maintenance.

This question is answered in Chapter 6 through a contextual interview performed with software engineering professionals who were given a chance to use this enhanced version of the runtime topology as a means to answer a number of questions relevant to the software evolution and maintenance tasks.

1.3.2 Research Methods

In this thesis we use a palette of research approaches ranging from empirical research to both behavioral and design science practices. In many cases, these methods are used in a complementary manner. Where empirical research provides insight on hard facts, e.g., the amount of code changed on average per commit per developer of a certain project, it does not necessarily answer why developer A commits, on average, more code per commit than developer B. Whenever we want to find the reason *why* people act in a certain way, we must then step into behavioral science practices.

When it comes to behavioral science, Hevner and Chatterjee [2010] remind us of how it is tied in a *"complementary research cycle"* with design science. The question between these two scientific facets is then: which should be applied first? Do we start by analyzing how a target group performs a task and use this as the input for design science? Or do we instead start with design supported by a theoretical need and then evaluate the designed outcome through behavioral science?

Throughout the different chapters of this thesis, the answers to these questions vary. Upon starting Chapter 2, existing sources hinted at the existance of a number of complaints from developers who maintain and evolve the integration of clients with web APIs. Our focus was then to first investigate the nature of these complaints by interviewing those

developers. Following the guidelines for social research by Babbie [2007] we opted for a semi-structured interview as it provides the possibility to use existing work (e.g., academic research, surveys, etc) as a starting point whilst allowing interviewees to expand on their answers even if they go outside the boundaries of the initial questionnaire. As behavioral interviews are limited by the experiences of the interviewees and may not always be complete, we chose to expand on these results through the use of empirical methods as an attempt to find potential causes for the aforementioned complaints. This is achieved by analyzing factual data, such as client code churn caused by web API evolution, the different web API evolution policies and the impact web API evolution has on client code.

The research approach in Chapter 3 is partly a continuation of the previous chapter. Aware of client developers' resistance to change, we wanted to test the hypothesis that not all web API clients are built to withstand unexpected web API evolution. This hypothesis could then only be tested empirically by simulating such scenarios where a web API has evolved and observing the behavior displayed by the clients. The reason why we then interview some of the client developers and add a behavioral facet to this work through semi-structured interviews is supported by the need to clarify the *why* of some of our findings. Also in this chapter we start with a set of questions and allow the interviewees to further expand with their experience on web API integration.

In Chapter 5 we then chose to start with an approach based on design science to develop Serviz. While a behavioral approach could have provided rich input into the design process, the limited number of subjects available with relevant experience on the use cases targeted by Serviz led us to begin with the design fueled by our own insights. Furthermore, supported by the work of Jerding et al. [1997] who report that *"interactive visualizations can present (...) voluminuous information much more effectively than textual representations"* and due to the large amounts of data collected at runtime, we chose to develop Serviz as a graphical tool. In order to complement the initial design science approach, we stepped over to behavioral science and evaluated Serviz through a user-study. The choice for a pre-experimental one-group pretest-posttest [Babbie, 2007] [Campbell et al., 1963] (over an experimental approach with a control group) was made due to a small number of potential participants available with relevant experience.

The same reasoning is applied in Chapter 6 where we first developed additional features into Serviz and then resort to a user-study to assess whether Serviz helps in understanding a running system. Again looking to obtain detailed participant insights on Serviz and stymied by the relatively small number of participants, we chose a contextual interview [Holtzblatt and Jones, 1995; Matthijssen et al., 2010; Zaidman et al., 2013] as it is an approach which allows us to obtain more meaningful and detailed insights from the participants. To this end, we set up a field study where we demonstrated the tool to groups of two participants and further ask that they *"think aloud"* [Ericsson and Simon, 1998] while exploring the tool. After the exploration stage, the participants were asked to fill in a questionnaire rating whether Serviz helped in a number of software engineering tasks. The purpose of the questionnaire was to serve as a starting point for the contextual interview where we asked the participants about the score given in each of their answers in order to find potential improvement areas.

1.4 Thesis Outline

This remainder of this thesis is divided into five chapters. Chapter 2 consists of developer interviews on the pains of web API evolution and our code analysis of some relevant web API clients which have dealt with web API evolution. Chapter 3 contains our experiment using 48 Android mobile applications on how mutation analysis helps in understanding whether web API client developers have taken the necessary precautions to alleviate the pains described in Chapter 2. Chapter 4 presents Spicy Stonehenge, a service-based system used as a case-study for Chapters 5 and 6. In Chapter 5 we present and evaluate Serviz, a tool which provides the runtime topology of a running service-based system. Lastly, in Chapter 6 we show how Serviz can be extended to provide more insight in understanding the interactions in a service-based system.

1.5 Origin of Chapters

The subsequent chapters in this thesis are composed of peer-reviewed work. Each chapter is self-contained in that it presents an introduction which may be at times overlapping with other chapters. The following list shows how each of the chapters map to each of the peer-reviewed publications:

Chapter 2 contains our paper initially published in the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014) [Espinha et al., 2014c] which was then extended and published in the Journal of Systems and Software [Espinha et al., 2014b].

Chapter 3 contains our paper published as a technical report *TUD-SERG-2014-009* [Espinha et al., 2014a] which was submitted to the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015).

Chapter 4 contains our paper published in the proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012) [Espinha et al., 2012a].

Chapter 5 contains our paper published in the proceedings of the 19th Working Conference on Reverse Engineering (WCRE 2012) [Espinha et al., 2012c].

Chapter 6 contains our paper published in the proceedings of the 2013 International Workshop on Principles of Software Evolution (IWPSE 2013) [Espinha et al., 2013].

1.6 Additional Publications

Although not published in this thesis, two publications set some of the ground work for this research. These are:

- Our paper published at the 2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2011) [Espinha et al., 2011] where we make use of dynamic analysis for the inference of causality between services involved in a single transaction.

- Our paper published at the 4th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS 2012) [Espinha et al., 2012b] where we showcase Spicy Stonehenge as a common case-study system.

CHAPTER 2

Web APIs: Loosely Coupled yet Strongly Tied

Web APIs provide a systematic and extensible approach for application-to-application interaction. Developers using web APIs are forced to accompany the API providers in their software evolution tasks. In order to understand the distress caused by this imposition on web API client developers we perform a semi-structured interview with six such developers. We also investigate how major web API providers organize their API evolution, and we explore how this affects source code changes of their clients. Our exploratory qualitative study of the Twitter, Google Maps, Facebook and Netflix web APIs analyzes the state of web API evolution practices and provides insight into the impact of service evolution on client software. In order to complement the picture and also understand how web API providers deal with evolution, we investigate the server-side and client-side evolution of two open-source web APIs, namely VirtualBox and XBMC. Our study is complemented with a set of observations regarding best practices for web API evolution. [1]

2.1 Introduction

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs) [Burns et al., 2012; Raemaekers et al., 2012]. Software developers access APIs as interfaces for code libraries, frameworks or sources of data, to free themselves from low-level programming tasks and/or speed up development [Dagenais and Robillard, 2008]. In contrast to statically linked APIs, a new breed of APIs, so called web service APIs, offer a systematic and extensible approach to integrate services into (existing) applications [Curbera et al., 2002; Vinoski, 2008]. However, what happens when

[1]This chapter contains our work together with Andy Zaidman and Hans-Gerhard Gross, published in the 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014) [Espinha et al., 2014c] which was then extended and published in the Journal of Systems and Software [Espinha et al., 2014b].

these web APIs start to evolve? Lehman and Belady [1985] emphasize the importance of evolution for software to stay successful, and updating software to the latest version of its components, accessed through APIs [Dig and Johnson, 2006]. In the context of statically linked APIs, Dig and Johnson [2006] state that *breaking changes* to interfaces can be numerous, and Laitinen [1999] says that, unless there is a high return-on-investment, developers will not migrate to a newer version.

In the context of web APIs, developers can no longer afford the inertia that was noted by Laitinen, as it is the web API provider that sets the pace when it comes to migrating to a new version of the web API. In the statically linked API context, developers could choose to stay with an older version of e.g. libxml, which meets their needs, yet, with web service APIs the provider can at any time unplug a specific version (and functionality), thus forcing an upgrade. In 2011, a study by Lämmel et al. showed that among 1,476 Sourceforge projects the median number of statically linked APIs used is 4 [Lämmel et al., 2011]. Should developers have no control over the API evolution (as is the case with web APIs), this would represent a heavy burden for client developers as it causes an endless struggle to keep up with changes pushed by the web API providers.

Also in 2011, a survey among 130 web API client developers entitled "API Integration Pain" [Blank (YourTrove), 2011] revealed a large number of complaints about current API providers. The authors reported the following regarding web API providers: *"[...] There's bad documentation. [...] APIs randomly change without warning. And there's nothing even resembling industry standards, just best practices that everyone finds a way around. As developers, we build our livelihoods on these APIs, and we deserve better."*

Pautasso and Wilde [2009] present different facets of *"loose coupling"* on web services. Indeed, all web APIs which make use of REST interfaces can be easily integrated with through a single HTTP request. However, a facet not considered in Pautasso and Wilde's work is that of how clients end up tightly tied to the evolution policies of the web API providers. This motivated us to investigate how web service APIs evolve and to study the consequences for clients of these web APIs.

In this exploratory qualitative study, we start by investigating [**RQ1.1**] what some of the pains from client developers are when evolving their clients to make use of the newest version of a web API. We do this by interviewing six professional developers that work with changing web APIs. Subsequently, we investigate the guidelines provided by 4 well-known and frequently used web API providers to find out [**RQ1.2**] what are the commonalities in the evolution policies for web APIs? Ultimately, we turn our attention to the source code. We do so by analyzing the code of several web API clients to find out [**RQ1.3**] what the impact on the source code of the web API clients is when the APIs start to evolve. We also turn our attention to the impact of evolution at the server-side of a web API, more precisely, we are asking ourselves [**RQ1.4**] whether web API providers take precautions in order to ease evolution pains of web APIs? We do so by analyzing the source code impact on two different case studies of web API provider and its respective client.

The remainder of this chapter is structured as follows: in Section 2.2 we first explain some terminology regarding web APIs. Section 2.3.2 describes our experimental setup

for both the client-side and end-to-end studies. Section 2.4 presents the results of our client side analysis including the interviews with the client developers and the lessons learned across different domains, as well as an overview of the web API characteristics and the impact on client code. Section 2.5 presents our end-to-end analysis for the two case studies. We then frame the results with our research questions and provide a list of recommendations for web API providers in Section 2.6. Lastly, we discuss related work in Section 2.7 and present our conclusions in Section 2.8.

2.2 Terminology

Throughout this chapter we refer to different terms in the context of web APIs. Indeed, the concept itself of a web API is somewhat ambiguous and is, in our definition, no different from a web service. Already Alonso et al [Alonso et al., 2010] report that *"the term Web services is used very often nowadays, although not always with the same meaning"*. The authors then resort to the W3C definition which states that a web service is a *"software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts"*. While this definition is for the most part correct, it restricts the technology to XML for which there are currently alternatives (such as JSON) as it is shown in this chapter. In the context of web APIs, different technologies also translate into different challenges faced by the developers which should be considered. In our study we encountered three major implementation approaches which we will briefly describe in the sub-sections below.

2.2.1 SOAP

The invocation of SOAP[2], originally defined as Simple Object Access Protocol, web APIs is most commonly performed through the sending of an XML document (where the method name as well as arguments are defined) over HTTP to the server. Before the invocation occurs, clients request the WSDL file (Web Service Description Language) which defines both which methods are available for invocation as well as which data types the web API expects.

2.2.2 REST

Representational state transfer (REST)[3] is an architectural style originally defined by Roy Fielding [Pautasso and Wilde, 2011]. Apigee's booklet on web API design[4] provides a clearer overview of what constitutes a REST web API as well as a set of practical guidelines.

Essentially, a REST web API relies on entities (referred to in the REST context as *resources*) and basic CRUD (Create, Read, Update, Delete) actions on those resources.

[2]SOAP —`http://www.w3.org/TR/soap12-part1/`
[3]REST — `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`
[4]Web API Design — `http://bit.ly/apigee-web-api-design`

For example, when a web API client would like to get data on the 'Customer' resource with id '1337' it would send a HTTP GET request to the address /customer/1337.

Such an approach does not allow for method invocation and Apigee supports that in certain particular cases, the 'resource' can actually be an action (i.e. a method) to be invoked by the web API server. In such cases, the client can then send a payload (specifying which method to invoke and its arguments) typically using JSON (c.f. next subsection).

An important distinction between SOAP and REST is the fact that while a WSDL equivalent exists (the WADL file), it is seldom used in practice [Maleshkova et al., 2009]. The more common alternative is human-readable documentation usually through means of a wiki. This lack (in comparison to SOAP) becomes evident by the existence of software which specializes in generating documentation for REST web APIs[5].

2.2.3 JSON-RPC & JSON

The JSON-RPC[6] (JavaScript Object Notation — Remote Procedure Call) approach shares a similarity with REST: it uses JSON as the data format for requests. In fact, oftentimes web APIs claim to be RESTful while in fact a flavor of JSON-RPC is used. JSON-RPC (where RPC stands for Remote Procedure Call) provides a mechanism for one software system to be able to invoke methods on another software system over the network. It is therefore, not an architectural pattern as REST and is bound to fewer restrictions.

The remote procedure calls are also made over HTTP and use JSON to specify which method should be invoked as well as the payload (arguments and responses). JSON in turn is used to dynamically construct types composed of key:value pairs, where the key is a string and the value is either an object or an array of objects. This contrasts with SOAP's XML-based invocations where types are defined statically and interactions are much more verbose.

2.3 Experimental Setup

Our exploratory study is comprised of two subsidiary studies. In the first part of our study we start by investigating the impact web API evolution has on clients for a group of high-profile web API providers. However, due to the closed nature of these web API providers, nothing can be learned from the potential pains web API providers also face when their web APIs must evolve. Therefore, with the second part of our study, we selected two open-source projects as well as a client for each of these projects as an attempt to shine a light on the web API providers' side of the story.

The two subsections below describe the experimental setup for each of these two parts of the study.

[5]MireDot — `http://www.miredot.com/`
[6]JSON-RPC — `http://www.jsonrpc.org/specification`

2.3.1 Experimental Setup for the Client-Side Investigation

In order to perform our client-side study which is exploratory in nature, we divided the study in three steps. We started by interviewing six developers (Table 2.1) who maintain clients for web APIs as to obtain anecdotal evidence of developers who had to undergo web API evolution in their clients. While we expected the developer interviews to provide fruitful insight into the evolution of web APIs, we also knew in advance that developer interviews are always subject to some degree of personal bias. The second step of our study therefore focused on analyzing objective software evolution metadata regarding the web APIs. Namely, we analyze the evolution policies (i.e. deprecation periods, breaking change notifications, etc) from four major web API providers. This allows us to identify potential best practices. The last step of our study is to analyze the main artifact where the web API evolution task may cause more or less impact: the source code. We measure and interpret the impact web API evolution has on client code by analyzing code churn and identifying the commits related to web API evolution.

In this section we provide more insight on how we selected the developers to be interviewed, how we selected the projects under analysis as well as how we measure the impact on the client code.

2.3.1.1 Interviews With The Developers

Our experiment included interviews with several developers who have at some point dealt with evolving web APIs. In order to find suitable candidates we e-mailed the developers of all the clients under study (see Section 2.4.3) and sent out public calls for participation on social networks. Ultimately, due to a low response rate from the approached developers, we interviewed all developers who accepted to participate in the

API	Developers	Observations
Google Maps	1 developer	Single developer. Creator and sole developer of the client since its creation in 2009.
Google Search & Bing	1 developer	Single developer. Creator and sole developer of the client since its creation in 2012.
Redmine API	1 developer	Developer is part of a larger team of 13 developers. Started in 2005.
Google Calendar	1 developer	Single developer. Creator and sole developer of the client since its creation in 2006.
Unnamed Payments Aggregator	2 developers	Two professional developers, part of a larger team.

Table 2.1: Interviewed Developers

study.

Additionally we had the opportunity to interview the client developers of a multinational payment aggregator company whose software system interacts with several financial institutions through web APIs.

Table 2.1 provides an overview of the web APIs each of the 6 interviewees developed clients for.

The interviews took on average thirty minutes per developer and were performed by the author of the thesis either face-to-face or via Skype in the format of a semi-structured interview [Babbie, 2007]. The ten starting questions that we used during the interview are listed in Table 3.2 and cover several web API-related issues, such as: maintenance effort, frequency of version upgrades, security, developer communication and implementation technologies.

As for the analysis of the collected data, we followed the guidelines set forward by Creswell [Creswell, 2009] for qualitative research. The interviews were recorded and further transcribed by the first author. Subsequently, the first and second authors read the transcriptions and established a basic coding in dialogue. This basic coding resulted in the identification of the categories or overall themes as they are reflected in Sections 2.4.1.1 through 2.4.1.6. Once we identified the categories, the first author did the actual coding.

2.3.1.2 Selecting Web APIs

In order to perform our code analysis we required web APIs with a large number of clients. To find such web APIs we resorted to ProgrammableWeb's[7] web services directory. From this list, sorted by popularity, we picked the top most popular web APIs and quickly verified which web APIs contained the largest number of references in GitHub. This led us to choose Twitter, Google Maps and Facebook. The projects using the Netflix web API were found while investigating projects on GitHub.

2.3.1.3 Selecting The Projects With Web API Evolution

Once we have selected a set of web APIs that are known to have evolved, we have to find candidate projects integrating with those web APIs. Candidate projects for our analysis need to meet the criteria of having performed maintenance due to the web API having changed. In order to have access to projects which contain this evolution step and thus shine a light on the number of changes involved in web API evolution we devised a mechanism to identify the evolution step.

This mechanism was then applied on GitHub as it contains a large collection of potentially suitable open-source projects.

For web API providers such as Twitter, Google Maps and Netflix, where an explicit versioning system is provided, the approach consists of two steps. They are: 1) compiling a list of all the projects on GitHub which contain references to the latest version of their

[7] Web Services Directory — `http://bit.ly/web-services-directory`, last visited October 3rd 2013

specific web API, and 2) for each project found, filter the Git diffs which contain references to the old version of the web API.

Facebook required a different approach. Even though a booklet on web API design by Apigee[8] emphasizes the importance of versioning by dubbing it *"one of the most important considerations"* and advising developers to *"never release an API without a version"*, Facebook violates this principle. Because there is no version number involved in the requests, our search is done by querying the GitHub repositories for small pieces of code which were reported in Facebook Developer's blog[9] as having been changed.

2.3.1.4 Impact evaluation

The goal of the chapter is to investigate how web service APIs evolve, and how this affects their clients. So, for each project, we looked at the commits right before and right after the first commit containing references to the new version of a web API. This was done to identify potential initial preparations prior to bringing a new API online, as well as to check for a potential fallout effect caused by switching to the new API.

In order to estimate the impact involved in maintaining the clients of a web service API, we start by using the code churn metric [Munson and Elbaum, 1998], which we define for each file as

$$FileCodeChurn = \frac{LOCAdded + LOCChanged}{TotalLOC} \tag{2.1}$$

The code churn we analyze and display in Table 2.4 (Avg. Churn) represents the average code churn for each commit. Of note is the fact that the churn presented does not count added files. Additionally, the *evolution churn* presented in the table consists of the churn caused by the evolution-related code changes. This churn is determined manually and through visual inspection of the evolution-related commits. This is done manually to ensure that all the churn considered in the evolution commits is indeed related to the evolution task. The percentage presented is then how this evolution churn compares to the average. With the data we collected we are also able to plot graphs showing the code churn per commit. This way we can also visually identify abnormally high code churn peaks as well as churn peaks surrounding the evolution related commits. These peaks are potential candidates for web API-related maintenance and are then investigated in more detail by looking at the source code and commit messages.

While code churn provides a good starting point for assessing the impact of a maintenance task, it does not provide the whole picture: the nature of the code change, the number of files involved and their dispersion also play a role in determining the impact of a change. Hence, we also provide a more in-depth view of how the API migration affects a particular project. This is done by looking at the number of source code files changed, and analyzing the nature of the changes (e.g. file dispersion, actual code changes, whether the API-related files are changed again). This analysis also allows us to mitigate the code churn's indifference to the complexity of code changes.

[8]Web API Design — `http://bit.ly/apigee-web-api-design`
[9]Completed Changes — `http://bit.ly/fb-completedchanges`

2.3.2 Experimental Setup For The End-to-End Analysis

While it is important to consider the impact web API changes have on clients, this relationship is bidirectional and minimizing client-side impact can be achieved if the web API provider is mindful of changes to the web API.

This is particularly important due to, again, to how in a web API relationship the provider and clients are tied to the provider's web API evolution policy.

To further our study, we perform an end-to-end analysis, meaning that we study how the web API changes from the provider's perspective and how this impacts the client. Ideally, we would have liked to analyze the server-side code of the web APIs mentioned before. However, due to the closed nature of such software systems, we resort to case studies using open-source systems (where both the provider and clients are open-source) to study these added aspects of web API maintenance.

2.3.2.1 Project Selection

For this part of our experiment we chose Oracle Virtualbox[10] and XBMC[11] since both are large projects (∼4 million and ∼2.2 million SLOC respectively) and for both there is a client available that is actively maintained and used. Additionally these two projects use different web API technologies (VirtualBox uses SOAP whereas XBMC uses JSON-RPC) which, on both the server and client side, also plays a role in how much code needs to be changed and under what circumstances.

Together with these two projects we analyzed a web API client for each case. For VirtualBox we studied phpVirtualBox[12] which is a feature-complete web-based GUI for VirtualBox and is endorsed by Oracle as a *"hot pick"* on the main page (even though it is developed by a third party developer). For XBMC we analyzed Android-XBMC as it is the official client for the XBMC web API (developed by a subset of the XBMC developers).

2.3.2.2 Source Code Analysis

Our source code analysis is aimed at better understanding how the code of both the servers and clients is organized, how it evolves, and how the web API is implemented. We look at technologies used for the implementation (as these may have an impact on maintenance) as well as how the web API source code is organized. Specifically we look at the encapsulation of the code (e.g. is the whole source code of the client tied to the web API or is the functionality abstracted into a translation layer), at its size[13] and at the structure of the web API itself (e.g. is the different business logic also well encapsulated).

[10] Oracle VirtualBox — `https://www.virtualbox.org/`
[11] `http://xbmc.org/`
[12] phpVirtualBox — `http://sourceforge.net/projects/phpvirtualbox/`
[13] SLOCCount — `http://www.dwheeler.com/sloccount/`

2.3.2.3 Co-change analysis

With our co-change analysis we identify which files consistently change with the web API-related files. Our initial goal with this analysis is to identify to what extent the web API-related code is self-contained. If we can establish that the web API-related code is (relatively) self-contained, we expect the web API to be more stable, which in turn would be helpful for the web API client developers.

Continuing our reasoning: in most cases, changes to a web API involve changing more than just the web API interface. For instance, if new methods are added, then the types used as method parameters will also have to be added elsewhere. Similarly, when a web API is changed due to a change in method parameters, these changes are often a result of deeper changes in the business logic of the software system.

These deeper changes to files which co-change with the web API interface also provide an interesting view of how a system evolves. If a non-web API file (e.g. in a different package) consistently changes whenever the web API changes, it provides a hint that such file contains web API-related functionality. This might be an indication for a refactoring opportunity.

For our analysis we make use of association rule mining to identify co-evolving entities, similar to how Zimmermann et al. have applied it previously [Zimmermann et al., 2005]. We make use of the Apriori algorithm as implemented in the *Sequential Pattern Mining Framework*) (SPMF) tool[14]. Because this tool requires the input to be numeric, we mapped each filename to a number and considered each commit as a transaction where the files (i.e. the numbers) are the items of the transaction. In addition, because we are particularly interested in finding association rules that indicate a change to the web API, we add one item at the end of each line/transaction: 1 if the commit contains changes to the web API files, 0 if it does not. The parameters (support and confidence) used for the Apriori algorithm are explained in the analysis section.

2.4 Client-side Analysis

In the sub-sections below we present the results regarding the first part of our study. Namely the results of the interviews with client developers, our findings regarding web APIs evolution policies and lastly the source code impact on clients caused by web API evolution.

2.4.1 Interviews With Client Developers

This study aims at understanding how web API evolution impacts client developers through the forced nature of the web API changes. To do so we first performed interviews with client developers for well known web APIs (Table 2.1).

The most interesting findings obtained through the interviews are presented in the subsections below. These subsections represent the major themes (or codes) which the

[14] SPMF — `http://www.philippe-fournier-viger.com/spmf/`

participants had experience and commented on. As additional remarks we present the results which do not fit in the predefined questions.

2.4.1.1 Web API Stability

We asked the client developers *"how does the effort of initial integration with a web API compare with the effort of maintaining this integration over time"* (Q1). Two of the interviewed developers (one for Google Maps and one for Google Calendar) were very peremptory and claimed that it takes them far more time maintaining the integration than it does integrating with a web API in the beginning.

The developer behind the integration with Redmine web API claimed that the effort involved in these two tasks is divided *"at least 50% into each task, with possibly even more time going into maintaining the integration"*.

What also came to light from all the participating client developers was the fact that in the beginning, the web APIs are very unstable and generally prone to changes.

This results in two-fold advice for web API providers and client developers alike when it comes to web API stability:

- From a provider's point of view, more thought should be put towards the early

Q1	How does the effort of initial integration with a web API compare with the effort of maintaining this integration over time?
Q2	How often does your web API provider push changes?
Q3	How dependent is your client on the 3rd party web APIs you are currently using?
Q4	Does your project also make use of statically linked libraries and do you feel there is a difference on how its evolution compares with web APIs'?
Q5	How do you usually learn about new changes being pushed to the web API your client is making use of?
Q6	Do implementation technologies make a difference to you?
Q7	How do you learn how to use an API? (Documentation? Examples? Do errors play a role in this learning?)
Q8	Is having different versions of a web API useful when integrating with your client?
Q9	When using 3rd party APIs, did you ever find that particular thought was put into an API behavior?
Q10	As a web API client developer, given your development life cycle, how many versions should the API provider maintain? And for how long?

Table 2.2: Questions Asked During the Developer Interviews

versions of the web API. In the event the web API requires some instability, then an approach as suggested by one of the interviewed client developers is recommended: the Redmine API developers clearly mark which features are prototype/alpha/beta (i.e. features which are very likely to change).

- As for web API client developers, because of this inherent instability in the early versions of web APIs, the need for separation of concerns and good architectural design becomes more urgent than ever. Integration with static libraries can be maintained for as long as the client developer wishes but since a third party is now in charge of pushing changes, making sure the changes are contained to a small set of files should become a top priority.

2.4.1.2 Evolution Policies

When asked about evolution policies (Q2, Q3), the participants presented us with different insights.

While different web API providers establish different timelines for deprecation of older versions of their web API, the client developer using Google Calendar's APIs was generally happy with the two year window provided by Google and in fact favored longer periods for this evolution. This developer claimed that *"we got now two years for updating to the [new] Google Calendar API, I think it should be even longer because a year is nothing anymore"*. Of consideration is the fact that this developer works on his project as a hobby (even though he is a professional developer) and therefore favors having a longer time to migrate to newer versions of the API. As he himself states *"if you have other projects, if you have to make money on other projects, even in two years it is difficult to find time to implement [the changes]"*.

The developer interviewed in the context of the Redmine API claimed that *"[while] it is quite a difficult question which depends on many factors in the project, four months time before deprecating would be fine"*. Because the Redmine API is still under development, he would rather have shorter cycles with functionality added more often.

Despite this developer's preference for shorter cycles, the nature of the changes should also be considered. In the case of the Redmine API, the evolution process consists mostly of feature addition and the features of the web API that are likely to change are clearly marked accordingly. However, looking at the comments in the 2011 survey [Blank (YourTrove), 2011] regarding Facebook's similar four-month deprecation policy, developers complained about how *"Facebook continually alters stuff thus rapidly outdating my apps"* and *"as I only use Facebook[...], [the biggest headache] is the never ending changes to the API"*. This is an indicator that more than just the frequency of the changes, web API providers should take also into consideration how invasive are the changes being pushed.

Also interviewed were two client developers for web APIs provided by financial institutions. An important distinction in this context is the fact that the web APIs being used are not available for free, as opposed to the others under study. Perhaps for this reason and according to the interviewed developers because *"the stakes are too high in the*

financial context", the web API providers maintained all the older versions of the web API indefinitely. This allows for client developers to never have to make any changes unless they require the features made available in the new web API version. While this is the ideal scenario from a web API client developer's point of view, whether this is feasible for all web API providers and the effort it takes to maintain several versions simultaneously is still something we would like to investigate in future research.

2.4.1.3 Static Libraries versus Web APIs

We asked all the interviewed developers how does, in their experience, the evolution of static libraries compare with the evolution of web APIs (Q4). While only one of the developers was simultaneously using static libraries as well as web APIs, his experience was that the static library he used had always maintained backwards compatible methods even after adding new features.

The developer interviewed in the context of Google Maps also mentioned that while his projects do not resort to statically linked libraries, he is using Drupal (a content management system) as the basis for his Google Maps integration and admitted that with Drupal and PHP he was in control of when to migrate to newer versions in contrast with those pushed by Google Maps. This is particularly relevant seeing as PHP itself introduced breaking changes in versions 5.3 and 5.4.

2.4.1.4 Communication Channels

Another issue touched upon in the interviews with the client developers has to do with how the web API providers notify their clients of upcoming changes (Q5). The client developers integrating with financial institutions' web APIs said that while it is a rare event, they will be notified by e-mail of any upcoming changes pushed by their web API providers. What was also mentioned was that while it ultimately does not affect them (because the web API providers do not force them to migrate to newer versions), it would be unfeasible to keep up with changes (should they be mandatory) from all providers due to the unreliable nature of e-mail (e.g. messages can be lost, automatically filtered as spam or simply missed altogether by the recipient).

Nonetheless, the web API providers under analysis have changed their communication channels over time. For instance, Google and Twitter nowadays force all client developers to request an API key and by doing so, they are added to a mailing list on which the upcoming changes are announced.

While this is what is currently considered the state of the practice, client developers for these web APIs will still get e-mails even if their code is not affected by the changes.

Facebook goes further and dynamically determines what parts of the web API a specific client is using in order to send e-mails only when changes are planned for that particular functionality.

2.4.1.5 Implementation Technologies

Even though all the web APIs under study use JSON-based technologies, we asked the interviewed developers whether they believe that the choice of technology from the web API provider can have an impact on the effort it takes to both integrate and maintain the integration with a web API (Q6).

One of the developers integrating with financial institutions using both SOAP and REST interfaces claimed both come with advantages and disadvantages. For instance, while integrating with a SOAP interface there is generally a WSDL file available which gives an overview of which methods and types are available and how to invoke them. The downside is the extreme verbosity of such an interface which is hardly ever human-readable. On the other side, REST, while allowing for less wordy interactions lacks anything similar to the WSDL file and the client developer is left to rely solely on the documentation which is usually written manually by the web API providers (and is thus, not as reliable as an automatically generated WSDL file).

An interesting remark by the same developers was that while some web API providers claim to provide a REST interface, this is in fact not the case. In his experience the interface is simply an HTTP endpoint which outputs JSON content but which does not, for example, meet the criterion of being stateless.

The developers integrating with Google Calendar and Google Maps expressed negative opinions on XML as a language for message exchange (thus, SOAP). Specifically, the developer integrating with Google Calendar claimed that *"the simpler the [better]. I hate XML because XML is such an open standard, it is very complex."* whereas the developer integrating with Google Maps claimed that *"SOAP is gone and dead"*. The developer behind the integration with Google Calendar went further and commented on the effort involved in maintaining the two technologies, namely *"at the beginning [he] had a wall to climb [when switching from SOAP to JSON] but now because I have everything it is certainly much easier to switch another API from XML to JSON"*.

2.4.1.6 Additional Remarks

An interesting remark from the interview with the client developer for Google Maps was his concern for vendor lock-in. In fact, when dealing with web APIs, a client is *tightly coupled* with a particular web API provider. The same developer highlighted the dangers of such dependencies with the example of Google Translate which Google officially discontinued in December 2011 (even if later on the web API was made available once more).

Additionally, even though the feedback provided by the developers integrating with the financial web APIs was limited due to the providers maintaining all the old web API versions, these developers also contributed with an additional anecdotal story. During their integration with financial institutions worldwide, they are often faced with web API documentation in foreign languages. This causes great distress and requires the developers to resort to either unreliable machine translation or to eventual colleagues who happen to speak the language, both of which come with the cost of time.

2.4.2 Web API Characteristics

In the aforementioned survey performed in 2011, the authors claimed that in the web API world, *"there's nothing even resembling industry standards"* [Blank (YourTrove), 2011]. We also found this to be the case amongst the chosen web API providers.

In fact, each of the four web API providers under study in this chapter adhere to different policies on what concerns web API evolution. These are explored in detail in the following sub sections.

2.4.2.1 Google Maps

The Google Maps API allows client developers to, amongst other things, display maps for specific regions, calculate directions and distances between two locations.

This API[15] falls under the global Google deprecation policy, i.e., whenever products are discontinued or backwards-incompatible changes are to be made, Google will announce this at least one year in advance. Exceptions to this rule regard whenever it is required by law to make such changes or whenever there is a security risk or *"substantial economic or material technical burden"*. To summarize, save for security-related bugs, Google claims to provide a 1-year window for the transition to a new API.

In practice, however, Google is much more lenient, e.g. analyzing the migration of Google Maps version 2 to version 3, Google provided a 3-year period for this transition rather than the announced 1-year deadline. Additionally, before the deadline arrived for version 2 going offline, Google prolonged this period for another 6 months, effectively offering a 3.5-year period for the transition. Why they offered such long period is not certain. However, anecdotal evidence from Google's user forums shows that many developers waited until the last moment to upgrade. In March 2013, an unnamed developer asked *"I'm working on upgrading to v3 but I'm expecting to finish 2 or 3 weeks after 19 May [initial deprecation date], so I was wondering if we can get an official answer about this"*. Similarly, when earlier in 2013 Google experienced an outage in all its Maps APIs' versions, several developers also asked whether v2 had already been taken offline, thus revealing that a number of developers were still using it. Google's provision of a very long transition period may have led the developers to be too relaxed about the deprecation, leading them to migrate at the latest moment.

2.4.2.2 Twitter

The Twitter API allows client developers to manage a user's tweets as well as the timeline. While this web API[16] has no official deprecation policy, the announcement for the current API version set a 6-month period to adjust to the change. Since it implies a different endpoint URI, both versions could in fact be maintained in parallel indefinitely, and it means that once the old endpoint is disabled, all applications using it will break. Despite the 6-month period, Twitter did not follow the original plan. The new API

[15] Google Maps Terms — https://developers.google.com/maps/terms
[16] Twitter API v1.1 — https://dev.twitter.com/docs/api/1.1/overview

version, announced in September 2012, was intended to fully replace the old version by March 2013. However, rather than fully take it offline, they decided to approach the problem by starting to perform "blackout tests"[17], both on the date the API was supposed to be taken offline and twice again two weeks apart after the original deadline. These blackout tests last for a period of one hour and can occur at random during the days they are announced. They act as an indicator for unsuspecting users, that they should migrate.

This approach contrasts that of Google and Facebook but gathers appreciation in its own right. The blackout tests have been very well received, with developers claiming *"These blackout tests will be super helpful in the transition. Thanks for setting those up!"*[18]

2.4.2.3 Facebook

The Facebook API is extensive and allows for client developers to access many data related to users' posts and connections. Facebook's[19] approach to web API evolution is substantially different as it does not use an explicit versioning system. Instead, the introduction of new features is done by an approach referred to as "migrations," which consists of small changes to the API that each developer can enable/disable at will during the roll-in period. After this period the changes become permanently enabled for all clients. The Facebook Developers website claims that Facebook provides a 90-day window for breaking changes. Like Google's, this policy also explicitly excludes security and privacy changes, which can come into effect at any time without notice. Unlike Google, however, Facebook has proven to not be lenient and the 90-day window is consistently enforced.

Facebook is also in the process of changing this policy. While so far there have been breaking changes put into place every month from January 2012 to May 2013 (with the exception of March 2012)[20], Facebook has announced that from April 2013, all the breaking changes will be bundled into quarterly update bulks (except security and privacy fixes).

In addition, Facebook has an automated alert system in place, which sends e-mails[21] to developers whenever the features they use are affected by a change. Dynamically determining which developers are relying on which features of the API goes along our research presented in Chapter 6 ([Espinha et al., 2013]) where we investigate to which extent such a mapping affects system maintenance.

[17] https://dev.twitter.com/blog/planning-for-api-v1-retirement
[18] Discussion API v1's Retirement — http://bit.ly/apiv1-retirement
[19] Facebook Breaking Change Policy — `http://bit.ly/fb-changepolicy`
[20] Completed changes — `http://bit.ly/fb-completedchanges`
[21] Example e-mail: `http://bit.ly/so-migrationemail`

2.4.2.4 Netflix

The Netflix API is a public web API which allows client developers to access text catalogues of movies and tv shows available in the Netflix collection. Netflix, much like Twitter, has no official deprecation policy. Additionally, to date only two versions have been released and both versions do still work. What makes this web API stand out is the fact that all the versions released to date still work and have no planned deprecation date. This highlights that, albeit at a potential cost to the web API provider, it is possible to simultaneously maintain several versions of the same web API.

It should be noted, however, that over time Netflix has released breaking changes across all versions of its web API. For example, on June 2012 Netflix announced a new web API endpoint to which all clients had to migrate within three months. Additionally, in March 2013 Netflix announced that it *"is not accepting new developers into its public API program"*. This suggests that the public Netflix API is on a path to discontinuation.

Because no deprecation of older versions exists and the migration to the newer version is fueled by the client developers' will to access the latest features, very little can be said regarding the amount of time given to client developers to migrate. Nonetheless, the migration witnessed in both clients under study stems from the web API endpoint change for which three months were given to migrate.

2.4.3 Impact on Client Code

When analyzing the impact web API evolution has on different clients, several considerations must be made regarding each project's code base. In Table 2.3 we present each of the projects under analysis for each web API along with its age and number of developers per their GitHub history. In Table 2.4 we present different metrics for the projects under analysis for each web API. All projects are different in nature and the number of lines of code (LOC) for the projects under consideration varies from 1.2KLOC to 479KLOC. We expect that this, coupled with the file count for each project, has an influence on both

Web	Project	First Commit	Number of developers
Twitter	rsstwi2url	March 2012	1
	TwiProwl	August 2009	3
	netputweets	January 2011	6
	sixohsix/twitter	April 2008	46
Google Maps	hobobiker	November 2009	1
	cartographer	May 2008	5
	wohnungssucherportal	January 2010	1
Facebook	spring-social-facebook	August 2010	9
Netflix	pyflix2	June 2012	3
	Netflix.bundle	July 2010	8

Table 2.3: Project Metadata

the average code churn for each project as well as the code churn required to implement evolution-related changes.

In our study we use code churn, a measure of how much code has been changed, as a first indicator of commits which should be further investigated manually. The raw data used for this study, namely a spreadsheet per project including a list of commits and commits which are tagged as being related to the web API evolution task, is also available online[22].

In the following subsections we analyze the data per web API provider and present our findings.

2.4.3.1 Twitter

The web API evolution step under analysis for Twitter consists of a minor version upgrade. It is, nonetheless, described by Twitter as "the first major update of the API since its launch". This new version does indeed bring many changes. For instance, clients are now forced to authenticate, XML support was discontinued in favor of JSON (until

[22]Raw commit data — `http://figshare.com/articles/JSS_Web_API_Data/1192860`

Web API	Project	LOC	Commits	Avg. Churn	Evol. Churn (% of Avg.)		File dispersion	Evol. commits
Twitter	rsstwi2url	2101	366	0.008203	0.008251	0.59	3	1
	TwiProwl	1199	156	0.007530	0.030629	306.75	1	1
	netputweets	8853	218	0.001679	0.005521	228.80	15	3
	sixohsix/twitter	3866	375	0.00871	0.004509	-48.21	11	7
Google Maps	hobobiker	478994	37	0.0000607	0.000147	142.27	4	2
	cartographer	1895	36	0.009328	0.115652	1139.84	17	2
	wohnungssucherportal	35119	208	0.00026	0.000127	-51.34	4	1
Facebook	spring-social-facebook	30362	1042	0.001222	0.000277	-77.33	14	1
Netflix	pyflix2	3433	49	0.008032	0.008409	4.70	8	2
	Netflix.bundle	1724	80	0.007530	0.002115	-71.91	2	1

Notes: *Avg. Churn* defines the average churn of modified files per commit (for all the commits). *Evol. Churn* defines the churn caused by the evolutionary step (manually inspected within the commits which set forward changes for the evolutionary step). *File dispersion* consists of how many files are changed in the context of the evolutionary step. *Evol. commits* consists of how many commits were involved in implementing the changes for the evolutionary step.

Table 2.4: Statistics Per Project

then, developers were given the option for either XML or JSON) and changes have been made to rate limiting (which can penalize clients who query the web API too often).

netputweets — The netputweets project is an alternative web interface for Twitter on mobile phones. Because it implements a wide range of features from the Twitter web API, it is also the Twitter project with the highest LOC.

In Figure 2.1 we present the code churn data compiled for the netputweets project. The figure shown concerns only the netputweets project although we did compile the data for all the projects. Doing so helped us in identifying potentially interesting hotspots in the projects' commits. In this figure we highlighted the commits involved in the web API evolution task. Here it is possible to visually assert that these commits are not exceptional in terms of code churn when compared to the remaining commits. Table 2.4 does show, however, that the evolution-related code churn is approximately 228% higher than average and the changes span across several files.

From the same table we also learn that netputweets is the Twitter project with the largest codebase, yet, its evolution-related churn is lower than TwiProwl (the smallest Twitter project by LOC). The netputweets project contains approximately eight times more LOC than TwiProwl and it took three commits over 15 files to implement the evolutionary changes. Such an increase in file dispersion may signal a tight coupling with the web API. Through manual inspection of the netputweets project we confirmed our hypothesis. Many of the changed files contain on themselves a static reference to

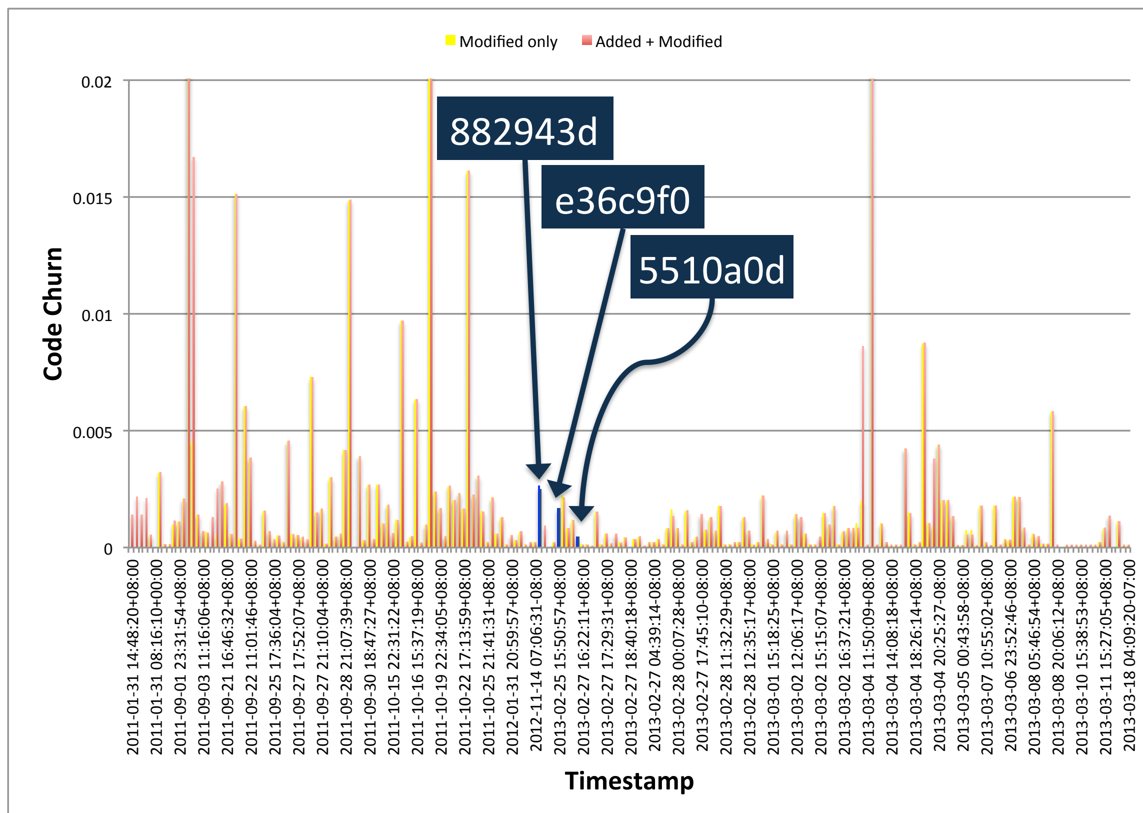


Figure 2.1: Code churn per commit netputweets

the Twitter endpoint which, should it change, requires these files to be also changed. Additionally, several changes are also made to the code handling the web API data. Because the data is used directly throughout the code (which implies a tight coupling with the specific data format), several changes are required throughout several files.

TwiProwl — As the client with the lowest LOC (compared to all analyzed Twitter projects), TwiProwl is also the one that implements the changes for the new API version by changing one file in a single commit. This project is a one file script which explains the file dispersion of 1. The 300% code churn compared to the average churn comes from implementing a new feature in the Twitter API (user lookup) and from adjusting several lines of code which directly iterate through the data provided by the web API (which was changed in this version).

sixohsix/twitter — Another project which has an elevated file dispersion is a Twitter library for Python (sixohsix/twitter). Manual inspection resulted in a different finding from that of netputweets. This client tucks away all the web API-specific integration into one file and even after the changes for the newest version had been implemented, the project was still using the older version. This is possible because of how the developer implemented a mechanism to allow him to choose the version of the web API by changing an argument in the method calls. This also justifies the file dispersion. While normally having to change a several number of files would be a task developers wanted to avoid, the only change to be done in this case is an argument that specifies which web API version to use.

rss_twi2url — The rss_twi2url project is a small script which provides tweets as an RSS feed. Because of its limited focus on a small subset of Twitter's web API, we expected the changes caused by web API evolution to be small. This was confirmed through the low code churn (approximately the same as the average). The changes span across three files, although manual code inspection revealed one of the files is a configuration file (changed due to Twitter's rate limit) and the other two files were directly impacted by Twitter's change on the returned data.

What We Learn

While in a general way Twitter pushed extensive changes with its latest web API version, our findings are that when dealing with web APIs a good architecture matters more than ever. This finding is supported by two projects: sixohsix/twitter and netputweets. While both integrate extensively with the Twitter web API, netputweets, by virtue of a poorer architecture, contains larger evolution-related churn. For instance, the Python library presents an evolution related churn that is 48% smaller than the average code churn which supports the hypothesis of a more carefully thought architecture, versus the 228% higher than average churn seen on netputweets.

2.4.3.2 Google Maps

The changes put forth by Google in version 3 of its Maps web API are extensive. Google says so itself in its thorough upgrading Google Maps guide: "as you start working

with the new API, you will quickly find that this is not simply an incremental upgrade”[23]. In our study we noticed that simple activities such as creating an instance of the web API are now done using entirely different constructors.

hobobiker — It took the hobobiker project changes in 4 files to implement the integration with the new web API. Through manual inspection we concluded that despite the low file dispersion, all the components of this project which require Google Maps integration are tightly tied with its web API. This tight connection is observed as each component of this project which requires web API integration establishes its own direct dependency to the web API. This, coupled with the 142.27% size of the evolution commits compared to the average churn may indicate poor architectural design.

wohnungssucherportal — The wohnungssucherportal, by comparison with hobobiker, despite requiring the same number of files to be changed, it took -51% of the average churn to implement the same changes. The LOC of both these projects is rather high and is justified by both being web applications and containing a large amount of boilerplate code (hobobiker relies on Drupal whereas wohnungssucherportal relies on Ruby on Rails). While the elevated LOC influences the absolute average churn (0.00026 compared to 0.009328 for cartographer with 1895 LOC) it should not interfere with the percentual code churn required by the web API evolution task.

cartographer — The cartographer project stands out in the elevated file dispersion it presents (17 files changed). As a library which allows other projects to integrate with Google Maps, this project also maintains backwards compatibility with the previous version of the Google Maps web API. Because of this and because this project’s architecture clearly separates the connection to the two API versions, several files were touched to provide support to the newer version. Namely, 8 files were copied and became the basis for the older version 2 support and the same files were copied and modified to enable the integration with the latest version. It is then not surprising that the code churn involved in the evolution task represents a 1139% increase over the average code churn spread out across 17 files.

What We Learn

The three projects under study present different lessons learned for client developers. While the change introduced by Google with version 3 of Maps is overarching and requires substantial changes (as stated by Google and proven by the creation of an extensive migration guide), projects like hobobiker and wohnungssucherportal suffered the sharpest pains (hobobiker in code churn and wohnungssucherportal in file dispersion). This is so as these two projects reveal poor design choices where every reference to the Google Maps web API was hardcoded.

The cartographer project on the other hand continuously implements support for both the old and new versions of Google Maps and despite the higher code churn, tucks away the web API concerns in a way that the core library does not require changes as extensive as the other projects.

[23] Upgrading Your Google Maps JavaScript Application To v3 — https://developers.google.com/maps/articles/v2tov3

2.4.3.3 Facebook

The only Facebook project available (as more could not be found using our approach) is a fairly large plugin for the Spring Framework which provides Facebook integration. What can be learned from this project is that even though Facebook's migrations are said to be smaller (and happen more frequently than in other web API providers), in fact the changes cause many files to require maintenance.

Considering the changes analyzed are relative to a migration and therefore not a major version change, and considering the churn percentage of this evolution task compared to the average is lower by 50%, we expected to encounter an underlying architecture with a good separation of concerns. This was confirmed through manual inspection. The web API-related code is encapsulated in Java classes specifically built for the web API communication, which were also the only files that required changes. By analyzing the changes we also realized the changes concern two major modifications in the Facebook web API. Namely, Facebook changed the way it handles images and simultaneously changed the way it refers to "check-ins" (i.e. checking in at a geographical location) and the way to retrieve them. What also contributes to the high file dispersion of these changes is the existence of an extensive test suite. In fact, for this specific commit there are six changed files (out of the 13) which are test-related. For this particular project we conclude that despite the changes pushed by Facebook being actually intrusive and require change, the way these particular client developers designed their architecture by isolating web API access into single-feature classes mitigates this problem. The changes span across several files but are generally small and confined to the web API-specific files.

2.4.3.4 Netflix

The Netflix web API pushed extensive changes with version 2. Amongst them are changes in the returned data, changes in API conventions and addition of new features.
pyflix2 — The pyflix2 project presents a rather high file dispersion of eight files. While the file dispersion is the first indicator of a potential poor architectural design, the meagre 4% increase in code churn versus the average suggests the changes are not very extensive. Nonetheless, manual investigation shows that part of the changes consist of unit tests and database migrations which are stored in text files. The majority of the remaining changes transform the hardcoded Python strings to Unicode, as presumably Unicode became mandatory on the new Netflix API. However, there are no mentions to this in the Netflix web API documentation.
Netflix.bundle — The second project which makes use of the Netflix API is a bundle for Plex (a media center) which contains all its web API references in the same two files. This justifies how all the evolution related changes are contained in two files as only these two particular files have the necessity to make web API calls.

What We Learn

Both projects analyzed in the context of the Netflix web API are small and relatively young. This somewhat justifies the small number of commits. The code churn caused by the web API evolution is rather small (with the projects staying around or below the code

churn average). The pyflix2 project is a Python library which requires a more extensive integration than the one provided by Netflix.bundle, hence the larger file dispersion and evolution-related churn.

2.5 End-to-End Analysis

In this section we analyze how the web API is implemented and whether there is evidence of special design and implementation considerations on the web API source code.

2.5.1 VirtualBox

Our analysis of the VirtualBox case study is split into two: the server-side and client-side analysis. In the server-side we cover the implementation, versioning and source code analysis of the web API as well as a co-change analysis of the web API-related files. On the client side we strictly look at the source and how it integrates with the VirtualBox web API.

2.5.1.1 Server-side

Oracle VirtualBox is a software system which allows the deployment of virtualized operating systems. Through its web API it allows full control over the features of each virtual machine deployed within an instance of VirtualBox.

Implementation In the VirtualBox web API all the methods are available under one single endpoint. This results in a single "class" which contains 1888 methods (one single file with approximately 54 kLOC) from different business entities (e.g. INATNetwork, IDHCPServer, IAppliance). This is symptomatic of a web API suffering from the multi-service anti-pattern as described by Dudney et al. [Dudney et al., 2002] where one single web API provides functionality pertaining to multiple disjoint business entities.

The web API is implemented using SOAP and thus, its interface is available as a WSDL file. This file is generated based off a XIDL[24] (Extended Interface Definition Language) file which is also used to define the interface for all the APIs VirtualBox provides (webservice bindings for Java, Perl, PHP and Python as well as static library bindings for C, Microsoft COM and XPCOM interfaces) in all the different languages. The fact alone that the SOAP interface is treated as any other (non-web) API raises a warning that VirtualBox developers take no special precautions when pushing changes to the web API.

Versioning As far as versioning is concerned, VirtualBox's web API is assigned the same version of the application itself. The implication is that regardless of whether a major, minor or patch release has pushed breaking changes to the web API, the web API will also have its version number increased. While VirtualBox does follow the semantic

[24]XIDL — `http://www.xatlantis.ch/xidl.html`, last visited December 13th 2014

versioning[25] approach, with a major.minor.patch versioning scheme, in fact, it applies a twist on this approach. With semantic versioning only a major version change is allowed to introduce breaking changes to the (web) API. In VirtualBox, the documentation states that both major and minor releases may introduce breaking changes, thus departing from the semantic versioning approach. In order to clarify the reason why semantic versioning was not being strictly followed, we asked the developers in the official VirtualBox mailing list for clarification. One of the lead developers stated it was simply a naming convention and that they *"consider the MAJOR.MINOR.0 releases as major"*. In a different post on the mailing list, a developer also stated that *"when [they] do a major update (first and/or second digit change), [they] may change APIs in incompatible ways. This is a burden on the API consumer but it allows [them] to keep a clean design and not accumulate legacy. Within one major version (e.g. 3.1.x), [they] guarantee API stability."*

Source code analysis In order to compile an overview of the breaking changes to the VirtualBox web API we analyzed all the 73 provided SDK versions on the VirtualBox website. Each SDK downloadable is a ZIP file containing (amongst other files) a WSDL file. The WSDL file from each SDK was extracted in order to then extract WSDL changes from subsequent version pairs of the file. While the SDK downloadables contain both a version number and a SVN revision number, the SVN revision number is from an internal SVN server which means it does not match that of Oracle's public SVN repository. This means that we have a WSDL file, its external version number (in the Major.Minor.Revision format obtained from the ZIP), the internal SVN revision but do not know what *public* revision of the whole VirtualBox source code the SDK belongs to. Because we are interested in source code changes, it is a requirement to know which commit was used to compile a specific SDK (and thus a specific WSDL). In order to obtain this information, we had to devise a matching mechanism to map WSDL files from the SDK downloadables to SVN revisions of the public repository. Because the WSDL is not versioned but rather generated, for each commit in SVN we compile the XIDL file into a WSDL file and compare to each of the WSDL files in the SDK.

By using a tool to compile the fine grained changes between two WSDL files (WSDLDiff[26]) we can then know what changes were made at the web API level as well as connect these changes to other files which also had to be modified to make the web API changes possible. This tool not only mines the changes from the WSDL files, but also identifies for each change whether it breaks backwards compatibility for clients. In the scope of our study, we are only interested in changes that break backwards incompatibility as those are the only ones which cause clients to be forced to change.

With resort to this tool and through analyzing the 73 officially released versions following the Major.Minor.Patch semantics, we were able to gather that VirtualBox has pushed:

- 3 major releases, all of which contain breaking changes.
- 7 minor releases, all of which contain breaking changes.
- 63 patch releases, two of which contain breaking changes.

[25] SemVer — `http://semver.org/spec/v2.0.0.html`

[26] `http://www.membrane-soa.org/soa-model-doc/1.4/cmd-tool/wsdldiff-tool.htm`

This data is not surprising except for the two patch releases which bear breaking changes (version 4.0.2 to 4.0.4 and 3.0.0 to 3.0.2). Developers claim that patch releases are free from backwards incompatible changes but this principle was violated twice. We tried to identify the nature of the changes introduced and the rationale behind this decision.

Our analysis has found that version 3.0.2 removed web API methods and changed a data type, whereas version 4.0.4 included a small change where a field was renamed.

As an attempt to clarify what was the reasoning behind this decision we reached out to the project's mailing list. Klaus Espenlaub, one of the VirtualBox developers, claimed that in 4.0.4 a field was renamed for the sake of clarity and in 3.0.2 there were *"more intrusive"* changes but that the removed methods were *"not really useful"* and because this functionality had been made available for a 10-day period (between version 3.0.0 and 3.0.2) they found *"the probability of breaking existing third party code was negligible"* and thus *"decided it was worth it"*.

Co-change analysis In order to identify co-changes, we applied association rule mining techniques. To do so, we first identified the file which, should it be changed, would cause a change to the web API. While VirtualBox makes use of WSDL for its web services, the WSDL file is generated and not initially available under versioning. For this reason, we perform our analysis on the file which is used to generate the WSDL interface, that is "src/VBox/Main/idl/VirtualBox.xidl". We then grab all the commits where this file is changed and treat each commit as a *"transaction"* for the Apriori algorithm. By doing so, we obtain a list of which files are associated with (and thus, tend to co-evolve) which other source files.

In Table 2.5 we present the results for a minimum support of 10% (lowest was 24 occurrences) and a confidence of 10%. The results of this table have been filtered to include only association rules where the web API is involved. The values for minimum support and confidence were chosen by experimentation. We started with significantly higher values (80% confidence and support) and gradually lowered until the occurrences were in the range of 20.

For VirtualBox we see then that 4 out of the 5 files which contain an association rule with the web API interface file are in the "Main" package and are all core implementation files (e.g. HostImpl.cpp deals with the implementation of host-related functionality, MachineImpl.cpp contains implementation regarding the virtual machines).

Rule	Sup.	Conf.
src/VBox/Main/MachineImpl.cpp ⇒ VirtualBox.xidl	30	14.8%
include/VBox/settings.h ⇒ VirtualBox.xidl	21	25.3%
src/VBox/Main/ConsoleImpl.cpp ⇒ VirtualBox.xidl	29	20.6%
src/VBox/Main/HostImpl.cpp ⇒ VirtualBox.xidl	14	23.3%
src/VBox/Main/include/VirtualBoxImpl.h ⇒ VirtualBox.xidl	15	15.6%
src/VBox/Main/include/HostImpl.h ⇒ VirtualBox.xidl	12	16.9%

Table 2.5: Association Rules VirtualBox

2.5.1.2 Client-side

As a client we study phpVirtualBox, a feature-complete web client for the VirtualBox software system. This project is developed by one single developer who is not affiliated with the development of VirtualBox itself.

This client links the WSDL file statically into their own source code. For the particular situation of phpVirtualBox this static approach works well, as each version of phpVirtualBox targets one specific version of the web API. However, SOAP-based web APIs allow for the WSDL to be queried dynamically [Alonso et al., 2010, p. 186] and in a dynamic scenario where the web API client provides support for web APIs with different versions, this approach is recommended as a way to ensure the web API does indeed support the required version.

This client maintains its versions synchronized with those of VirtualBox. The approach taken by phpVirtualBox means that with every release, it does not have to keep any code related to deprecated or modified features. Ultimately, the client developer claims that regardless of the *patch* version, an instance of phpVirtualBox from a specific minor release should work with all the patch releases under the same minor release (which is coherent with the evolution policy of VirtualBox itself).

Source code The phpVirtualBox client is written in PHP and Javascript. All the interactions with the VirtualBox web API happen in two files: one PHP file which is automatically generated based on the server's XIDL file and a second PHP file, this one manually created, which contains all the web API interaction logic. While at a first sight this is a sound design decision as it results in one single point which potentially needs to be changed should the server push breaking changes, our analysis revealed a different scenario. Indeed, a single point for change exists but all the source code directly relies on this file which is a 1-to-1 mapping of the server-side methods and data types. In other words, the client's source code is directly tied to specific methods of the server-side.

2.5.2 XBMC

2.5.2.1 Server-side

XBMC is a media player developed as open-source and has been under development since 2002. It provides a web API through a JSON-RPC implementation which allows for full control of the media player's operations.

Implementation Unlike the SOAP approach used by VirtualBox, it provides no schema similar to the WSDL. Instead, human-readable documentation is provided in the form of wiki pages on the website of the project which as of right now does provide information about the latest version of the web API. Additionally, the source code of XBMC does have data types defined for server-side use. These types as well as their names are never provided to the web API clients. This in turn means that XBMC clients are not bound to type names or type references but rather the developers must compose JSON objects out of primitive types (strings, integers, etc).

Additionally, the 120 different methods are also packaged within 13 different classes.

The encapsulation of the XBMC web API functionality is more carefully thought out than that of VirtualBox and it stands as a major difference between the two. The VirtualBox web API provides 1888 methods under one single class.

Versioning The API is implemented with its own separate versioning which, in its early stages, was represented by a single digit. Recently, at version 6 (also the current version), the semantic versioning approach was adopted. Additionally, the developers warn potential client developers that only even version numbers represent stable states of the web API. This versioning approach is similar to the one used in the Linux kernel up to version 2.6 where minor releases with an odd version number are not meant to be used in production[27][28].

Source code analysis On the server side of the XBMC scenario, the web API maintains its own separate version. To perform our analysis we must then first resort to the ServiceDescription.h file to identify which web API version corresponds to each snapshot of the source code.

While this analysis is possible on the server side, the Android XBMC client does not check which version of the web API is available on the server and it is therefore not trivial to identify which version is being targeted by the client.

On the client side the data sent through JSON-RPC as well as the responses are accessed through reflection. This makes it very difficult to identify which arguments of which data type are being used in each commit of the client's source code.

On the server side we attempted to use different approaches to identify which methods exist in each commit. We attempted to do so by performing textual mining on the JSON file used as the definition of the web API as well as mining a C header file which also contains all the web API methods. Both approaches yielded no usable results. The JSON file is large (142 methods, 2221 LOC) and encompasses many different web API methods. It also relies on a separate file where the data types are declared (1610 LOC). This makes manual inspection of these files across the many thousands of commits infeasible. Automatic diffing of this file in pairs of consecutive commits also resulted in many false positives which would also be too time consuming to analyze manually.

Co-change analysis For the XBMC server we used a similar approach as the one used for VirtualBox. The JSON-RPC-based web API is declared as a JSON file which is then used to generate the actual interface. Therefore, changes applied to this file are a potential indicator of breaking changes. However, it is not trivial to identify what constitutes a breaking change for a JSON client. For instance, adding fields will not break backwards compatibility whereas removing fields depends on whether the client is actually using those fields.

For this reason, we considered all the commits where changes are applied on the interface definition files (methods.json and types.json) and use temporal references as a guideline to match changes on the server side to those on the XBMC Android client.

The association rules obtained from the XBMC project tell a different story from

[27] Linux Kernel Versioning — `http://www.tldp.org/FAQ/Linux-FAQ/kernel.html#linux-versioning`

[28] Compiling a Linux kernel — `http://lpic2.unix.nl/ch02s02.html`

that of VirtualBox. All the rules involve three files only (VideoLibrary.cpp, JSONServiceDescription.cpp and ServiceDescription.h) in addition to the two web API interface files which are used to identify changes to the web api (methods.json and types.json). Furthermore, the files implicated as co-changing with the files used as reference for the web API are also all related to the web API, as can be seen by the package to which they belong (json-rpc). Table 2.6 shows us all the tight co-change relationship involving all the aforementioned files (web API interface plus the co-changed files). Nonetheless, the conclusion to be drawn is that changes to the web API are self-contained to the web API packaging.

Rule	Support	Confidence (%)
{./interfaces/json-rpc/VideoLibrary.cpp,./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/ServiceDescription.h	24	100
{./interfaces/json-rpc/JSONServiceDescription.cpp, ./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/ServiceDescription.h	40	100
{./interfaces/json-rpc/JSONServiceDescription.cpp, ./interfaces/json-rpc/ServiceDescription.h} ⇒ ./interfaces/json-rpc/methods.json	40	88.9
{./interfaces/json-rpc/ServiceDescription.h, ./interfaces/json-rpc/JSONServiceDescription.cpp} ⇒ ./interfaces/json-rpc/methods.json	24	82.8
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ ./interfaces/json-rpc/ServiceDescription.h	24	77.4
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ ./interfaces/json-rpc/methods.json	24	77.4
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/ServiceDescription.h	91	75.8
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ ./interfaces/json-rpc/methods.json	40	66.7
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ {./interfaces/json-rpc/ServiceDescription.h,./interfaces/json-rpc/methods.json}	40	66.7
./interfaces/json-rpc/types.json ⇒ ./interfaces/json-rpc/ServiceDescription.h	63	58.3
./interfaces/json-rpc/ServiceDescription.h ⇒ ./interfaces/json-rpc/methods.json	91	51.7
{./interfaces/json-rpc/ServiceDescription.h,./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	40	44
./interfaces/json-rpc/ServiceDescription.h ⇒ ./interfaces/json-rpc/types.json	63	35.8
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	40	33.3
./interfaces/json-rpc/methods.json ⇒ {./interfaces/json-rpc/JSONServiceDescription.cpp, ./interfaces/json-rpc/ServiceDescription.h}	40	33.3
{./interfaces/json-rpc/ServiceDescription.h,./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	24	26.4
./interfaces/json-rpc/ServiceDescription.h ⇒ {./interfaces/json-rpc/JSONServiceDescription.cpp,./interfaces/json-rpc/methods.json}	40	22.7
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	24	20
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/ServiceDescription.h	24	20
./interfaces/json-rpc/ServiceDescription.h ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	24	13.6

Table 2.6: Association Rules XBMC

In both the VirtualBox and XBMC case studies we also attempted to find other possible links between the web API and the remaining source code. Specifically, we investigated whether there was any single package which was more likely to co-change with the web API. However, no such connection was found.

2.5.2.2 Client-side

The client application under study, the official XBMC client for Android, is developed by a subset of the XBMC developer community. This application actively keeps up with the latest version of the web API provided, thus breaking compatibility with previous versions of the server-side software. This is particularly troubling as the end-users who installs the Android XBMC client via the Google Play Store on an Android device cannot choose which version to install. This means that an end-user running an older version of the XBMC media player will have to either manually prevent the application from auto-updating or manually download the particular version through the XBMC client's previous releases website in order to have a functioning client. Looking at the comments left on the Play Store regarding the XBMC client, we found clear evidence of web API evolution causing trouble to end-users. Users of the client commented *"it worked until the last update with no warning. Apparently then it requires the latest version of XBMC. This is highly unfortunate, as it's hard to install older versions of software on Android devices. This basically means that if you want to use the Android remote you have to always use the last version of XBMC. In my opinion that's worse behavior towards your users than is usual amongst even open source, and I really hope this is not going to be repeated in the future."* and *"don't update the entire app without keeping backwards compatibility in mind.(...) I have 6 XBMCs running and have no time to update them all when a new release comes out of beta."*.

2.6 Discussion

In this section we use our findings to address our three research questions and present a list of nine do's and don'ts for developers of API web services.

2.6.1 Answering the Research Questions

We start by answering the research questions laid out in the introduction regarding the three different API providers.

2.6.1.1 RQ1

"What are some of the pains from client developers when evolving their clients to make use of the newest version of a web API?". Through our interviews, client developers highlighted how the early versions of web APIs are invariably unstable and change-prone. While some web API providers offer indicators of particularly unstable functionality in

their web API, by default web API providers push breaking changes across the whole feature set. It also became clear that no standard policy exists on what concerns deprecation periods and that the ideal amount of time is dependent on the developer. Ideally longer periods would be provided but further study is required to establish what the cost would be for the web API provider to keep two versions of a web API active for a longer period of time. The technology being used also plays a role in the developers satisfaction with an observed preference for REST and JSON amongst the interviewed developers.

2.6.1.2 RQ2

"What are the commonalities in the evolution policies for web APIs?". In a survey on "Web API Evolution Pains" the authors concluded that "there's nothing even resembling industry standards, just best practices that everyone finds a way around". When it comes to evolution policies, this seems to be true as well. Google and Twitter make use of versioning and give ample periods of time (∼2 years and 6 months respectively) for the client developers to migrate. Facebook opts for not providing versioning altogether and pushes breaking changes every three months. Lastly, Netflix with already two existing versions continues to maintain both versions simultaneously. Twitter also stands out for the "blackout tests" which serve as warnings for developers that eventually the old web API version will be shutdown.

2.6.1.3 RQ3

"What is the impact on source code when web APIs start to evolve?". As expected, the impact on source code depends greatly on both the breadth of the changes pushed by the web API provider and on the quality of the clients' architectural design. An example of this is two projects which integrate with the Twitter web API. While sixohsix/twitter provides an extensive integration with the web API, the churn caused by the changes is much lower than that of TwiProwl which performs basic web API tasks. This same observation applies to the two Netflix projects. The code churn and file dispersion metrics have also had limited usefulness. For instance, the cartographer project contains changes in excess of 1000% of average churn and reports having 17 files changed, yet, the architectural design is robust as this project maintains support for multiple web API versions. The lesson learned is that the impact can be high (e.g. Google Maps pushed changes which affect the smallest of tasks) and that for this reason, developers should take caution and design for change. Lastly, our evidence also suggests that web APIs are significantly more change prone in their early versions.

2.6.1.4 RQ4

"Do web API providers take precautions in order to ease evolution pains of web APIs??". To answer this question we focus on the VirtualBox and XBMC projects since they are the only ones for which we have access to the source code.

VirtualBox In the case of VirtualBox the fact that the web API is generated from the same interface as other statically linked APIs provides an early indication that no special care is taken regarding the web API. This is further fueled by a web API which suffers from a severe case of the multi-service anti-pattern where the web API and all its 1888 methods (and respective request and response types) from different business entities are provided under one single class. Perhaps as a result of this, and to the fact that the backwards compatibility responsibility has been delegated fully to the client side, the main client for this web API (phpVirtualBox) simply disregards the issue of backwards compatibility by releasing a new version for each of the VirtualBox releases. From the code analysis performed on VirtualBox it has also arisen that every single major and minor version has pushed breaking changes to the web API which reveals a great deal of external instability. Furthermore, the association rules mined out of the source code showed that multiple core implementation files consistently change together with the web API, revealing a potentially high fan-in between implementation and web API.

XBMC The XBMC web API was developed from the beginning as a web API. There are generally fewer breaking changes, with most breaking changes representing refactorings. Also from the association rule mining, only web API-related files (specifically, the files used for web API implementation) were observed to co-change with the main web API files. In this case study, it is then the client's fault for the poor backwards compatibility management. Namely, the Android XBMC client is publicly available in the Google Play Store and multiple end-users complain that the latest version simply does not work with older XBMC servers. The client developers do make older versions of the client available through their website but this requires manual installation and require multiple clients to be installed for controlling different servers with different versions.

In sum, referring back to RQ4, our analysis results in a mixed picture. On the one hand VirtualBox shows no particular arrangements made in order to maintain its web API more or less stable than any other statically linked API, whereas XBMC contains a purpose-built web API with its own versioning and human-readable documentation. As a result, the clients under analysis also deal with their web API's differently (albeit neither with an ideal approach). The phpVirtualBox client simply releases one client version per web API version and Android XBMC keeps up with the latest version of the web API, disregarding backwards compatibility.

2.6.2 Recommendations

Based on our investigation and additional insights obtained from observing developer forums, we compiled a list of nine recommendations for web service API providers with regard to easing the evolution task for developers of API clients.

2.6.2.1 Do not change too often

Facebook is pushing monthly "breaking changes", yet a recent survey on API integration pain [Blank (YourTrove), 2011] revealed that this policy has caused distress amongst

developers. It is unclear whether this has played a role in Facebook moving to quarterly updates (starting April 2013). It stands to reason that some business domains require more frequent breaking changes than others, nonetheless, frequency of change would be an interesting gauge to include in future research on metrics for service quality which to the best of our knowledge currently does not exist.

2.6.2.2 Old versions of the API should not linger too long

Looking at the scenarios where the web API provider *will* deprecate older versions of their web API, Google started off with a 1-year timeframe for the deprecation of Google Maps's version 2, and ended up extending it to 3 years. Yet, reaching the 3-year mark, many developers still flocked to the developer forums in hopes that the deadline would be extended further (which happened for another 6 months). Seeing as maintaining legacy code often implies a high level of effort, time and cost the message is: longer periods leave developers too relaxed about the change. While we expect a large company like Google to perhaps be able to invest in long periods for backwards compatibility, this may not be true for all other web API providers.

It should be noted that this advice is not applicable to web API providers which decide not to deprecate their old web API versions.

2.6.2.3 Keep usage data of your system

By knowing which users are using which features, system maintainers can target those particular users via e-mail to remind them about upcoming changes. In Chapter 5 ([Espinha et al., 2012c]) we have investigated how this could be achieved and indeed developed Serviz which provides an overview of the system in terms of time, users and versions. This approach was also adopted by Facebook as can be seen through the targetted e-mails sent to client developers who use a particular feature of a web API which will be affected by an upcoming backwards incompatible change.

2.6.2.4 Blackout tests

Before taking the old versions offline permanently, try it for short periods of time. While it requires no special tooling, Twitter's *blackout tests* approach has been successful in reminding developers that a change in the API is upcoming; the approach has also been appreciated by developers.

2.6.2.5 Provide an example of interaction with the API

Something not gathered directly from the analysis presented in this chapter but rather collected from the API integration Pain Survey, is the developers' need for an (up-to-date!) example of how to interact with the API. Maleshkova et al. [Maleshkova et al., 2010] also recognized this need stating that *"most [web] API descriptions are characterized by*

under-specifications". Indeed, recognizing this need, industry has now starting to create tools which aid in automated web API documentation (e.g. Miredot[29]).

2.6.2.6 Stability Status per Web API Feature

As a web API provider, tagging each of your web API's features with a *"stability status"* which indicates whether a feature is stable for production use or instead it is alpha/beta is welcomed by the interviewed developers. This way, developers aiming for stability are able to know which features to be wary of. This recommendation can also be the starting step for further investigation on automatically determining the stability status of a feature through e.g., repository mining techniques or unstructured data mining of bug trackers.

2.6.2.7 Lookout for Young Web APIs

An observation recurring from nearly all the developer interviews warns client developers about how young web APIs tend to be very change-prone. This should be taken into account by client developers who are advised to, from early on, implement good separation of concerns between web API interaction and the core of the client. Web API age is another factor which may influence web API/service quality and it should be further investigated for inclusion in potential future service quality metrics.

2.6.2.8 Version Accessibility

Ideally, a client will provide some degree of backwards compatibility. However, when that is not done, the client developers should make sure all previous versions of their client are as easily accessible. While not an ideal situation (e.g. an end user will require different clients to control servers with different versions as is the case for XBMC) it is still a better practice than simply discontinuing the older versions. In practice, Google for example provides a setting on the Android platform which automatically updates all applications to the latest version. Should this be used, and it is indeed important as in some cases using outdated versions represent a security risk, end users will be left with an application which is no longer compatible with an older web API. This calls for either easier switching to older versions of the same client or a better effort at not pushing backwards incompatible changes in a single e.g. Android application.

2.6.2.9 Robustness Against Changing Web APIs

Of particular use to client developers who integrate with web APIs is ensuring their client application is as robust as possible when web API providers push unexpected changes. In Chapter 3 ([Espinha et al., 2014a]) we have investigated mechanisms to support this and provide preliminary results of how mutation analysis can be used for this purpose.

[29]Miredot — `http://www.miredot.com/`

To summarize, web service APIs drive the evolution of software. Clients are forced to update by the API providers which contrasts with the statically linked libraries. However, in order to ease that evolution, we think the nine aforementioned guidelines should be taken into account.

2.6.3 Threats to validity

We now identify factors that may jeopardize the validity of our results and the actions we have taken or intend to take.

External validity. While we have quite some variety in terms of (1) developers working on web APIs, (2) API providers, as well as in (3) API client projects, it remains to be seen whether our observations still hold for (a) API providers who charge money for usage of the API, as they might be more reluctant when deprecating older version of the API which in turn might imply losing customers, and (b) for closed source API clients, whose developers might be inclined to upgrade quicker in order to satisfy their (paying) customer base with the latest security fixes and/or features. In future work, we will expand our investigation in this direction.

With regard to the generalizability of the end-to-end analysis, we studied two projects which are vastly different in terms of implementation technology, software architecture and versioning principles. They also exhibited differences in concerns with regard to backward compatibility. We acknowledge that two case studies systems form a limited basis to draw conclusions from and as such we will investigate more systems in future work.

Construct validity. We have measured the impact of evolving APIs on clients by investigating the code churn. While code churn is very valuable, it does not sufficiently take into account the relative complexity, nor the time needed to perform change tasks. In future work, through developer interviews we will investigate the actual *effort* of these maintenance tasks.

Reliability validity. There might be bias in the manual interpretation of the impact of change. To minimize bias the lead author who performed the investigation, thoroughly discussed all findings with the co-authors. Also a potential threat to reliability validity with the interviews is the existence of potential inaccurate memories, misunderstandings, and miscommunications and misrepresentations. We attempt to mitigate this threat by recording and transcribing all the interviews.

2.7 Related work

Maintenance of service-based systems. Lewis and Smith were among the first to recognize that maintenance of service-based software systems is different from maintaining other types of software systems [Lewis and Smith, 2008]. In particular, they highlight the importance of *impact analysis* for service providers as they have to consider a potentially unknown set of users.

In order to help mitigate the issues highlighted by Lewis and Smith, Espinha et al. address this lack of knowledge regarding the user-base of services by tracking how different users use a service-based system in different ways (Chapter 6, [Espinha et al., 2013]). Through collecting runtime data of a web service-based system, the authors are then able to plot an overview of which services depend on which other services, as well as being able to do so for a specific period of time as well as for a specific user.

Fokaefs et al [Fokaefs et al., 2011] also recognize the added challenges of web service evolution. The authors devised and evaluated VTracker, an algorithm for XML differencing, and based on its results analyze what actually changes between subsequent versions of web services and what is the effect on the maintainability of their clients. While very useful for XML-based services, some of the web APIs we analyzed such as Twitter or Facebook make use of JSON and have no equivalent of the WSDL document available.

Also Fokaefs and Stroulia [Fokaefs and Stroulia, 2014] expanded on their previous work and defined a classification for the different types of changes which afflict different versions of web service interfaces through the usage of a tool (WSDarwin). This tool was also extended in further work [Fokaefs and Stroulia, 2012] by the same authors where it is then able to automatically adapt clients to changed service interfaces.

Pautasso and Wilde study the different facets along which web services can be described as *"loosely coupled"* and analyze different implementation technologies [Pautasso and Wilde, 2009].

Maleshkova et al. study the state of the practice on what concerns web API implementation and amongst the findings, discovered that the majority of the web APIs are actually underspecified [Maleshkova et al., 2010].

Evolution of APIs. Robillard and DeLine conducted a large-scale investigation among 440 professional developers at Microsoft to establish what makes APIs hard to learn [Robillard and DeLine, 2011]. Their observations are that the most severe obstacles developers face pertain to the documentation and other learning resources.

Dig and Johnson try to understand the nature of changes to APIs [Dig and Johnson, 2006]. From the five case studies that they analyzed in detail, they found that over 80% of the API-breaking changes can be classified as being refactorings.

Dagenais and Robillard present *SemDiff*, tool-support for recommending API-method replacements for methods that were broken during the evolution of the API [Dagenais and Robillard, 2009].

McDonnell et al. through a study on API stability and adoption in the Android ecosystem have found that, despite the added benefits of newer versions of APIs, developers tend to be slow in adopting the newer versions [McDonnell et al., 2013].

An interesting non-peer reviewed work in this field is a survey [Blank (YourTrove), 2011] conducted on the pains of web API integration which presents many complaints from web API client developers.

Daigneau focuses specifically on the brittleness of web APIs in his book on service design patterns [Daigneau, 2011]. He proposes the *Single Message Argument* pattern, which suggests to refrain from creating signatures with long parameter lists. Daigneau further states that long parameter lists *"[...] signal the underlying framework to impose a*

strict ordering of parameters which, in turn, increases client-service coupling and makes it more difficult to evolve the client and service at different rates."

Mileva et al. [Mileva et al., 2009] present an interesting analysis on the usage of different versions of statically linked APIs. As an example they show that the the 3.8.1 version of the junit library is more popular than the latest 4.4 version. They explain this usage trend through the fact that there has been a big API change between *3.x* and *4.x* and that developers were reporting "lots of work". The choice to remain with an older version of an API that developers have when working with statically linked libraries is in most cases not present when working with web APIs.

Romano and Pinzger [Romano and Pinzger, 2012], through the analysis of web service WSDL files, make use of fine-grained changes applied to service interfaces as an attempt to measure how often these changes happen and what types of changes happen between versions of services.

Kaminski et al [Kaminski et al., 2006], mindful of the pains of web service evolution, propose the *"chain of adapters"* technique as a means for unmanaged web service evolution where the older web service interfaces are pushed into different namespaces and a translation layer translates and forwards the calls as necessary to the older versions. This approach is also supported by an Eclipse plugin which facilitates the use of this technique on WSDL-based web services.

The work of Xing and Stroulia [Xing and Stroulia, 2007] also supports the task of web service evolution with resort to a tool (Diff-CatchUp) which through the use of heuristics, attempts to automatically suggest replacement APIs for such APIs which broke backwards compatibility.

Web APIs. Ly et al. [Ly et al., 2012] note that despite the availability of a number of best practices, e.g., REST principles, and a plethora of software components and technologies, discovering and exploiting Web APIs requires a significant amount of manual labour. Notably developers need to devote efforts to interacting with general purpose search engines, filtering a considerable number of irrelevant results, browsing some of the results obtained and eventually reading and interpreting the Web pages documenting the technical details of the APIs in order to develop custom tailored clients. In this light, Ly et al. attempt to automate the extraction of relevant technical information from web pages.

2.8 Conclusion

In this chapter we perform an exploratory study regarding the impact of web service API evolution. Our contributions are:

- An interview with six professional developers to ask them about their experiences with web APIs that evolved.
- A study into the evolution policies of four high-profile web APIs (Google Maps, Twitter, Facebook and Netflix).
- An investigation of ten open source clients integrating the aforementioned four web

APIs to see the impact of web API evolution on source code.
- A list of nine recommendations for developers of web APIs and client applications integrating web APIs.
- A study on the code impact on both server and client-side code for both VirtualBox and XBMC.

Our findings suggest that web APIs still fall short of an industry standard. Different web API providers adhere to different practices and what would seem like essential features (e.g. versioning), are in fact neglected (e.g. by Facebook).

Our study also stresses the importance of developing clients for change on what concerns web API integration. The promise of loosely coupled web service APIs comes, in fact, at the cost of having changes forced upon the client developers. Should developers fail to implement proper separation of concerns, switching to different web API providers may also prove more difficult than what *"loosely coupled"* would otherwise suggest. While some web API providers may allow developers to use their old web API versions for extended periods of time, in general, all web API providers will sooner or later impose changes on their clients.

From our two end-to-end case studies we have also found that still some web API providers consider their web API just as any other statically linked API. Changes are pushed regularly and clients will also resort to per-version client releases which in turn implies requiring different versions of the client when communicating with different versions of the web API.

From these observations stems our claim that while technology-wise web APIs seem to offer a "loosely coupled" way of connecting web API provider and client, organizationally web API provider and client are "strongly tied" when the web API starts to evolve as in most cases the client is forced to "co-evolve".

As the evolution is indeed inevitable, we also found that the different evolution policies impact the satisfaction of web API client developers. To help mitigate this problem, we provide a list of recommendations such as not changing the API too often and performing blackout tests.

Future work. We aim to extend our investigation to a wider range of API providers and a larger selection of projects using these APIs. Another aspect we want to consider in future work is to make use of a change distilling tool such as proposed by Fluri et Gall [Fluri and Gall, 2006] and categorize the different types of changes as to be able to determine the nature of each change as well as to quantify its relative impact. Additionally, we aim to analyze whether web service API changes impact open-source and closed-source applications differently. Do these closed-source projects apply more urgency to their changes due to their paying customers? While the proposed recommendations stem from having been successful in real-world projects, an interesting aspect which we would like to further investigate is under which conditions these recommendations may in fact not be applicable or may require adjustments.

Finally, we also want to investigate whether the closed-source web API providers' policies differ from those of open source web APIs in which client developers are not the

same and/or have no say in the evolution process.

CHAPTER 3

Web API Clients: How Robust is Yours

Web APIs provide a systematic and extensible approach for application-to-application interaction. A large number of mobile applications makes use of web APIs to integrate services into apps. Each Web API's evolution pace is determined by their respective developer and mobile application developers are forced to accompany the API providers in their software evolution tasks. In this paper we investigate whether mobile application developers understand this and how they deal with the added distress of web APIs evolving. In particular, we studied how robust 43 high profile mobile applications are when dealing with mutated web API responses. Additionally, we interviewed three mobile application developers to better understand their choices and trade-offs regarding web API integration. [1]

3.1 Introduction

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs) [Raemaekers et al., 2012]. Software developers access APIs as interfaces for code libraries, frameworks or sources of data, to free themselves from low-level programming tasks or speed up development [Dagenais and Robillard, 2008]. In contrast to statically linked APIs, a new breed of so-called web service APIs offer a systematic and extensible approach to integrate services into (existing) applications [Curbera et al., 2002; Vinoski, 2008]. However, what happens when these web APIs start to evolve? Lehman and Belady emphasize the importance of evolution for software to stay successful [Lehman and Belady, 1985], and updating software to the latest version of its components, accessed through APIs [Dig and Johnson, 2006]. In the context of statically linked APIs, Dig and

[1] This chapter contains our work together with Andy Zaidman and Hans-Gerhard Gross, published as a technical report *TUD-SERG-2014-009* [Espinha et al., 2014a] which was submitted to the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015).

Johnson state that *breaking changes* to interfaces can be numerous [Dig and Johnson, 2006], and Laitinen says that, unless there is a high return-on-investment, developers will not migrate to a newer version [Laitinen, 1999].

When integrating with a web API however, developers can no longer afford the inertia that was noted by Laitinen. The web API provider sets the pace for migrating to newer versions (eventually removing older ones altogether) and client developers are forced to migrate. In the statically linked API context, developers could choose to stay with an older version of e.g. libxml, which meets their needs, yet, with web service APIs the provider can at any time unplug a specific version (and functionality), thus forcing an upgrade.

Indeed, while Laitinen claims that client developers will postpone migration to newer versions until there is a high return-on-investment, in Chapter 2 ([Espinha et al., 2014b]) we found some web API providers are eager to push breaking changes and force client developers to migrate to a newer version within a period as short as 4 months.

Through their loose coupling [Pautasso and Wilde, 2009] and REST interfaces, web APIs can easily be integrated into applications with a single HTTP request [Pautasso et al., 2008]. However, as the integration becomes as simple as exchanging HTTP requests, do client-side developers consider the consequences of ever-evolving web APIs?

We choose to perform our investigation into web APIs in the realm of mobile applications. This was a conscious decision instigated on the one hand due to the continued growth of mobile Internet usage[2], and on the other hand because mobile apps connected through web APIs are an integral part of the mobile computing experience [Christensen, 2009], thus ensuring relevance. Being aware of the ever-growing importance of and reliance on web APIs [Maleshkova et al., 2010], particularly in the mobile computing domain, we wonder how well-prepared some of the most popular Android mobile applications are regarding:

- changes and faults in the web API response from the server due to evolution of the web API
- interrupted HTTP requests due to e.g. loss of Internet connectivity as investigated by McIntosh et al. [2011]
- empty response messages due to server overload

In order to steer our research, our main research question is *how well-prepared are Android mobile applications with regard to changes in response messages from the web API* which is then divided into the following sub-research questions:

[RQ1] How robust are mobile apps when the web APIs being used return unexpected responses?

[RQ1.1] How can we simulate unexpected responses from web APIs?

[RQ2] Have web API client developers developed resilience against changes in or failure of the web API?.

To address these questions we performed a study on 43 Android mobile applications

[2] Already in the United States of America, reports show that specifically mobile applications' Internet usage has in 2014 surpassed Internet usage on desktop computers — `http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/`, last visited June 6th, 2014.

which make use of at least one web API. In our study we perform fuzz testing [Miller et al., 1990] through a series of manual mutations aimed at mimicking potential real-world scenarios where the web API changed its behavior either through (communication) failure or software evolution. We then report the different reactions displayed by the mobile applications when dealing with such mutated responses and interview three developers of some of the aforementioned applications as a means to gather deeper insight on whether these developers are aware of the added challenges introduced by web APIs.

The remainder of this chapter is structured as follows: in Section 3.2 we describe our approach for analyzing web API client robustness, in Section 3.3 we describe our experimental setup including how the mobile applications were selected, the added dimensions we analyzed and details regarding the developer interviews, Section 3.4 describes the results of our experiment with mutation analysis as well as the insights from the developer interviews, Section 3.5 discusses potential threats to validity of our work, in Section 3.6 we present related work and lastly we present our conclusions and future work in Section 3.7.

3.2 Approach

In order to investigate how robust Android mobile applications are when dealing with changing and/or faulty web APIs, we employ a mutation analysis approach (also found in the area of software testing [Jia and Harman, 2011]). We apply this approach on web API responses by intercepting such messages before they are received by the Android application.

We chose mutation analysis as it allows for the creation of potential mutant web API responses and therefore makes it possible to simulate real-world web API pains. Because it allows us to create a wide range of different mutants it is also ideal in an environment where we lack metadata on each web API response and therefore a more targeted approach (e.g. removing only *"non-optional"* fields) would be impossible.

In this section, we first introduce our mutation analysis in Section 3.2.1, after which we explain the technical setup that we have used to apply the mutations in Section 3.3.2.

3.2.1 Mutation Analysis — Mutant Generation

Our mutation analysis consists of mutating the web API response for a particular web API request sent by a mobile application. Xu et al. [Xu et al., 2005] set forward four perturbation primitive operators for mutating XML documents: two insertion operators and two deletion operators where the difference is the position in the XML tree where new nodes are added and deleted. One of the addition operators which inserts nodes at the same level as existing nodes is also included in our study. The other addition operator relates to datatype insertion and since none of the studied web API responses contain datatype definitions, it was excluded from our study. The same reasoning was applied to the deletion operators.

Thus, for the purpose of generating web API response mutants, we devise two operators from the aforementioned work: removal of existing nodes and addition of new unrelated

nodes (referred to in this chapter as *field removal* and *field addition*, respectively). We extend these operators with four other operators: *malforming a response*, *replying with an empty message*, *changing the implicit data type of a field* and *disrupting the data formatting*.

The choice of mutation operators is also supported by the work of Wang et al. [Wang et al., 2014]. Through a study on the evolution of some of the most popular RESTful APIs, the authors found that all of the proposed mutations (with the exception of the data formatting disruption) are in fact common change types to the APIs' interfaces. The authors also analyzed StackOverflow questions and concluded that questions related to the deletion of parameters (or *field removals*) are in fact one of the three most common questions regarding web API changes.

Field removal mutations consist of removing fields from the web API response. This particular mutation is used as a means to test robustness against *breaking changes*. Whether using a structured approach such as semantic versioning [3] where only major versions are allowed to bear breaking changes, or using a more lenient approach, when dealing with a web API it is possible that some fields are removed from web API responses. Examples of breaking changes caused by the removal of fields can be found in Chapter 2 ([Espinha et al., 2014b]), in particular regarding the Facebook web API [4]. Such

[3] Semantic Versioning — `http://semver.org/`
[4] Facebook Completed Changes — `http://bit.ly/fb-completedchanges`

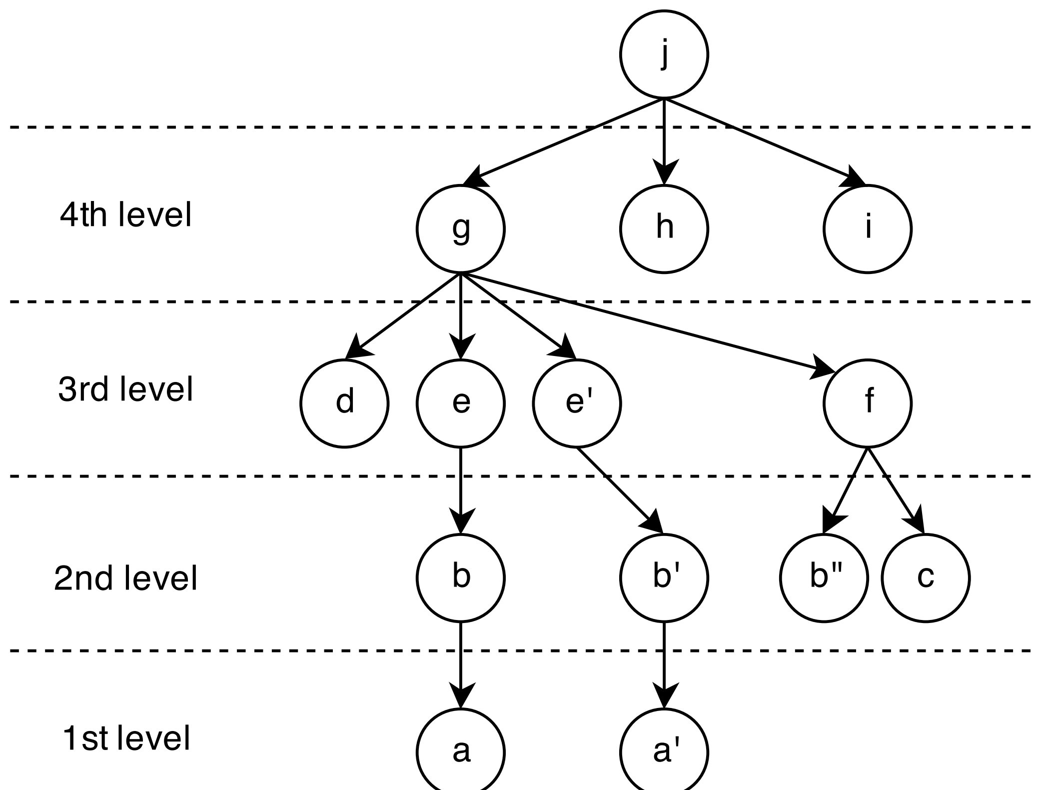


Figure 3.1: Field removal mutation applied on a sample node tree

breaking changes, as defined by Dig and Johnson [Dig and Johnson, 2006] (in the context of statically linked APIs), remind us that when fields are moved, changed, renamed or replaced, a field is always inevitably removed (e.g. a rename is a removal plus an addition with a different name). Also Li et al. [Li et al., 2013] show that indeed more providers rename parameters which also results in breaking changes.

While applying this mutation, removing fields in different parts of the web API response has the potential to result in different behaviors. For this reason we perform a step by step removal where after each step, the mobile application is tested against the resulting mutated web API response for that step. Our step by step removal is performed using the following removal guidelines to ensure that each child node of each node type is removed at least once. Since no data type definitions were encountered, we assume a node is of the same type as a different node when they share the same children structure.

Removal Guidelines

- *Rule #1* — A node of each type is removed in its own step once.
- *Rule #2* — If more nodes of the same type (as an already removed node) exist at the same level, they are left to be removed upon the removal of their parent node.
- *Rule #3* — After each bottom-most level has been emptied (or *only* contains nodes, the type of which, one node has already been removed), we move up a level.
- *Rule #4* — In each level, if nodes of different types exist but neither have children (or have children of *repeated* types), all these nodes are removed in one step.

Example

As an example, we analyze the node tree such in Figure 3.1 where each node represents a web API response field. In this node tree, the nodes are named after their type.

- *Step 1* — In the bottommost level (i.e. 1st level) two nodes exist of the same type. According to Rule #1 we remove **node a** and due to Rule #2, **node a'** is left for later removal.
- *Step 2* — Due to Rule #3 we move up to the 2nd level. At this stage, **node b** is removed and **node b'** is *ignored*. While **node b"** is *also* of the same type as **node b**, its parent node is of a different type (**node f**, which has one more child). Therefore, **node b"** and **node c** are removed in separate mutations.
- *Step 3* — Having cleared the 2nd level, we move to the 3rd level and all the nodes are child nodes of **node g**. Therefore, according to Rule #4, **nodes d, e, e' and f** are removed in one mutation.
- *Step 4* — Again due to Rule #3, we move up to the 4th level and apply Rule #4 to remove the remaining nodes (**nodes g, h and i**) in one mutation. After this step, only the root node is left, which is never removed.

Of note is the fact that despite removing fields, special care was taken to maintain the web API response semantically valid. This was done through the usage of online validators for both JSON [5] and XML [6] documents.

Malformed responses can happen for a number of reasons. While not necessarily a direct result of software evolution, such responses happen e.g. if the data encoder of the web API fails to properly sanitize strings. Such failure could then lead to floating reserved characters which in turn break the document's data format. This can happen while, for example, encoding a JSON string which contains double quotes and these are not properly escaped (i.e. “foo”:“b“ar” as opposed to the valid “foo”:“b\“ar”). In order to mimic these failures, our malformed response mutation consists of mutating a random node of the web API response as to make it malformed in its respective data format. In XML we mutate the response by breaking an XML tag (e.g. ‘<data>’ becomes ‘<data’) and in JSON this is achieved by leaving a dangling double-quote in a JSON string (e.g. “foo”:“bar” becomes “foo”:“bar).

Empty responses can be a symptom of different types of ailments on the web server. For instance, should the web server be at the edge of its maximum capacity, some requests may receive an empty response. Similarly, if the connection is terminated due to communication issues, it could also lead to empty responses being returned to the mobile application. Our empty response mutation consists simply of replacing the response with an empty-bodied HTTP response.

Changing data types mutations consist of two changes: selecting at random a numeric field and replacing it with a string as well as performing the reverse operation on one other, randomly selected, string field. As all the mobile applications under study run on Java (by force of the Android platform) and since Java is a statically typed language, if special care is not taken when parsing the web API response, type mismatches could occur. Also as a part of software evolution, web APIs may at some point change the (implicit) data types of certain fields. For instance, while the price field can be a string which includes the currency symbol, at a later stage the price field can become a purely numerical field.

An important remark is that no metadata is available for any of the web API responses under study. Therefore, this particular mutation is done by changing a field which visually appears to be of a type into the other type.

Field addition mutations consist of adding one or more fields to the web API response which contain irrelevant data for the mobile application. While less likely to cause issues as the web API response remains semantically valid and the applications may simply ignore the data, it is also a possible occurrence in a software evolution scenario.

Data formatting mutations consist of adding line breaks and white space as an attempt to verify whether the mobile applications are sensitive to this type of changes. While JSON is not sensitive to white space characters [7], XML may in some cases be sensi-

[5] JSON Editor Online — `http://www.jsoneditoronline.org`

[6] XML Viewer — `http://codebeautify.org/xmlviewer/`

[7] JSON RFC 4627 — `http://www.ietf.org/rfc/rfc4627.txt`

tive [8]. We therefore test this mutation as a means to further test the mobile applications' robustness.

3.3 Experimental Setup

For our experiment, we first required a body of mobile applications to be analyzed. How these applications were chosen is described in detail on section 3.3.1. We then introduce mutations in web API responses through the use of a transparent web proxy, described in subsection 3.3.2. These mutations, previously described in subsection 3.2.1, are a means to probe the robustness of the mobile applications by emulating breaking changes and faults on the web API.

We also expand this quantitative view with a more qualitative approach. To do so we interviewed 3 developers, each from a different application under study. More details regarding these interviews are also presented.

3.3.1 Application Selection

With our study we aim to understand what is the current state of web API integration issues. We do so by analyzing some of the most popular Android applications. All the applications under study were required to meet two criteria.

The first criterion is that each application must use at least one web API. Important to note is that our definition of web API excludes mobile applications which simply load HTML or RSS feeds. These exclusions are due to the fact that with an HTML response, no processing is required on the mobile application. With this we exclude also any mobile application which may use screen-scraping techniques [Martin et al., 2003] as screen-scraping is performed by extracting data from user interface elements which are not necessarily designed with the same backwards compatibility concerns of an API. Similarly, RSS feeds have a fixed structure which means they are not susceptible to software evolution changes which make them not applicable for our study.

The second criterion is related to how the mutation analysis is applied in the web API response. Tampering with such web API responses is only possible if the web API communication happens over an insecure channel such as HTTP. Indeed, this is particularly important as an encrypted protocol such as HTTPS does not allow for changes to be made to the content of its messages (without a security certificate).

For this study we picked candidate applications from the top 100 free applications available in the Google Play Store of the Netherlands, United Kingdom, United States of America, Canada, Australia, Belgium, Brazil, Spain, Germany and France. In total we installed 198 applications. We had to exclude 68 of these because they use HTTPS, while a number of other applications use proprietary binary data formats for the communication. Similarly, other applications which use compressed files as a means to transfer the web

[8]The XML FAQ, How does XML handle white-space in my documents — `http://xml.silmaril.ie/whitespace.html`

API responses (e.g. ZIP files) were not mutable using our approach as it would require the response to be decompressed and recompressed on the fly. Ultimately, our study corpus is composed of 43 applications (see Table 3.1 [9]).

Of note is the fact that, despite not having been a criterion for selection of the applications, all of the analyzed applications make use of either XML or JSON as a data format for the web API requests. This is particularly relevant as a data format like XML requires an additional XSD schema document to enforce data types. For this to happen, the XML document would require references to the XSD [10] document which can be used to validate it. Such cases have not been encountered in this study which means no XML documents were being (explicitly) validated against an XSD schema. During our mutation analysis, changing data types was still attempted on XML documents (i.e. changing a field with a numeric value to a string) even though none of the fields were specifically numeric as it happens with JSON where numeric fields can be identified by the absence of quotation marks.

3.3.2 Applying the Mutations

In order to mutate the responses from each mobile application's respective web API, we start by installing the chosen applications for our study on a Google Nexus 7 tablet (running Android KitKat 4.4.2) and configure the tablet to redirect all the network traffic through a transparent proxy (Charles Web Proxy) setup in a separate machine on the same network.

For each mobile application studied, before starting the mutation analysis, we follow a series of manual steps:

1. While using the transparent proxy, we identify a repeatable action (e.g., the push of a button) which causes a request and response interaction with the web API.
2. We then collect a standard response (all the standard responses, many of which several hundreds of lines long, are available online [Espinha, 2014]) for that particular web API request made by the application under study and configure the transparent proxy to replace the response of all other similar requests (for the purposes of the Charles web proxy, similar requests means requests sent to the same endpoint) with a customized response.
3. The customized response is in fact the original web API response although slightly modified. The modification first introduced is used as validation to ensure the customized response is being loaded and is therefore *not* part of the mutation analysis (n.b. this first customized response remains a valid response).

After ensuring the customized responses are replacing the standard response, the original web API response is then further disturbed with a number of different types of mutations (explained in detail in section 3.2.1). For each mutation we observe how the Android application reacts to such changes. Our observations are then categorized into

[9] The Buienalarm and Eurosport mobile applications make use of two distinct web APIs and therefore appear twice on the list.

[10] XSD Schemas — `http://www.w3.org/TR/xmlschema11-1/`

Application	Version Number	Crashes	Versioning	Caching
NS.nl	3.2.1	√	√	√
yr.no	3.0.1	√		
Skyscanner	2.0.9		√	
Buienalarm (own API)	2.3			
Buienalarm (OpenWeatherMap)	2.3			
BBC News	2.5.2 WW	√		
Daily Mail Online	3.2		√	
WeatherBug	3.5.97		√	
The Weather Channel	5.0.3	√		
Reddit is fun	3.3.12	√		
Ozsale	3.3.12		√	
tramTracker	1.3	√		
NRL - League Live	4.5.7			
The Masters Gold	4.1			
mobile.de	4.2.0	√		
Wetter App	2.3.3			
MeinProspekt	7.19		√	
TV Movie	1.3.1	√	√	
Wetter.com	1.4.9.4	√		
McDonald's Deutschland	1.4.0.1			√
H&M	2.6.1			
eltiempo.es	1.1.2	√		
Liga de Futbol Professional	5.2.12	√		
TecMundo	1.6.1d	√	√	
Trivago	1.9.4		√	
BeSoccer	3.0.3	√		
La Chaine Meteo	1.1.3		√	√
Resultats Foot en Direct	2.8			
RATP	3.0.10			
ViaMichelin	3.5.0		√	
Le Figaro	3.5.1		√	√
Le Parisien	3.3.2			√
Eurosport (XML)	3.7.6			
Eurosport (JSON)	3.7.6			√
Tele Loisirs	4.4.1			
NU.nl (stocks)	5.4.1		√	√
Just Eat	1.4.1.63			
Couverts	2.8.3			
Trulia	5.7.2		√	√
24Kitchen	2.2			
NL Treinen	2.0.10			√
Huizen	1.71		√	√
Kieskeurig	0.9.9.2		√	
Jumbo FoodMarket	1.1		√	√
Pull&Bear	1.2.3		√	
	Total	13	18	11

Table 3.1: List of Mobile Applications Studied

different types of behaviors (e.g. crashing or indefinitely loading without a timeout) and turned into a report (a sample is available online [11]) which is two-fold in its content: it starts with a statistical overview displaying how many applications behave in each of the identified categories, and culminates with a report specific to that particular application.

[11]Sample status report — `http://bit.ly/report-web-api`

3.3.3 Caching and Versioning

After beginning our experimental study, we found that for some applications, after the *"sample web API response"* was collected to be used as basis for the mutations, the mutated data (even with a string for string mutation) would not be loaded. This hinted at the usage of caching on some of the mobile applications. As caching is of particular interest for mobile applications where the Internet connectivity may sometimes experience slow bandwidth and where the web API may not respond due to high peaks of server load, we collected data on whether each mobile application uses caching and whenever possible, how the caching is used.

On what concerns versioning, in Chapter 2 ([Espinha et al., 2014b]) we found that some of the high-profile state of the practice web APIs (e.g. Twitter, Google Maps, Netflix) make use of some form of versioning. Still, we also found a major web API provider (Facebook) which does not make use of any form of versioning in their web API. Having studied these two different approaches and how client developers perceive each of the aforementioned web APIs, we also collect data on which web APIs are versioned and which type of versioning is used.

3.3.4 Developer Interviews

While the empirical study described above provides interesting insight on how a large body of mobile applications react when web APIs experience different failures and changes, it does not provide an explanation as to the choices of their respective developers. As an attempt to shed some light on the developer perspective of developing and testing an application which integrates with a web API, we aimed at interviewing the developers of some of the mobile applications under study. These interviews took the format of a semi-structured interview [Babbie, 2007] using the questions in Table 3.2 as a basis to stimulate the exploratory discussion.

We selected 14 applications which stood out either due to a special versioning mechanism, because they crashed or because of some particular behavior that other mobile applications did not demonstrate. For these select applications we composed an application-specific *status report* with our findings. These reports were then sent to their respective developers along with an invitation to participate in an interview.

Ultimately we were able to interview three software developers: the mobile software architect for *OZsale*, the product manager for the *Trivago* mobile application and lastly, the sole developer of the *NS.nl* Android application. The interviews lasted 15 minutes on average.

The *status report* we compiled and sent to the developers contains statistical information on all the applications under study. We provide data on how many applications crash due to a mutation, how many use versioning and the divide between JSON and XML implementations. Also for each type of mutation we provide statistics on the different observed behaviors. We also present statistics on how many applications use caching and complement the report with application-specific findings (e.g. some applications still

load malformed data). It is also these outlier findings of behaviors that are not common which we aim at clarifying with the developers through the interviews.

3.4 Experimental Results

We present our findings regarding each observed behavior in the Android applications under study upon applying the different mutations to the web API response. Specifically, we report on four of the six initially proposed mutations as the field addition and data formatting mutations did not elicit any unexpected behavior. We provide an analysis of the different behaviors displayed by the mobile applications and whenever relevant, provide anecdotal examples.

We also present our findings on the different types of data caching and web API versioning encountered as well as developer input on some of the choices used in web API integration. Of note is that the results are valid for the respective versions studied (see Table 3.1), as each of the mobile applications and their respective web API may change, so may the mobile applications' reaction to web API mutations.

Q1	What was the design decision behind choosing HTTP over HTTPS?
Q2	Why are no caching mechanisms used?
Q3	Is versioning not used for a particular reason?
Q4	Is the web API used by other (third-party and or mobile) applications?
Q5	Is the mobile application native to Android or generated with a mobile development framework?
Q6	Is the mobile application developed by the same team as the web API?
Q6.1	In particular when it is not developed by the same team, how does the mobile application team learn about the web API changes?
Q7	How frequently are the web API and mobile application updated?
Q8	Have there been problems in the past with breaking changes causing the mobile application to break?
Q9	Are there automated tests in either the mobile application or web API?

Table 3.2: Questions Asked During the Developer Interviews

3.4.1 Application Behavior

When each of the different types of mutants was applied to the web API responses, we observed four distinct behaviors from the mobile applications:

1. Force close — The *force close* is Android terminology for applications which crash by throwing uncaught Java runtime exceptions. Such exceptions are then caught by the Android platform and the application is *force closed*.
2. Error message — Showing an *error message* is a graceful way of letting the end-user know that something did not go as expected and what his or her course of action should be (e.g. try again or check the Internet connectivity). Some applications show an error message whenever a number of disturbances afflict the connection to the web API server. Whenever these error messages are shown, the robustness criteria is met as the applications do not crash and it is therefore a preferable behavior to a force close. However, in the highly dynamic domain of web APIs where new versions are released regularly and often without the client developers' knowledge, we expected some applications to provide recommendations in the error messages as to what the end-user should do. This was never the case and in fact, the end-user is at times misinformed about the nature of the problem.
3. No indication — By opposition to showing an error message, some applications silently deal with the mutated web API response. In some cases the data is partially loaded, in other cases a clue is provided that the loading has stopped, but no information is given to the end-user as to what has happened. By not showing any reaction to the end-user's input, the mobile application may induce confusion. When the application simply does not react even though the request has been made to the web API and the mutated web API response has been received, it is impossible for the end-user to know whether the data is still being loaded (as in these cases there was also no visual indication of loading) or whether he or she should try again to refresh the data.
4. No timeout — timing out is an important part of reporting a failure. At times however, applications remain indefinitely loading. It is then up to the end-user to decide when to stop waiting and close the application. Such behavior indicates that whatever exceptions may have been thrown are being handled (since the application did not crash) but are potentially being muffled. Ultimately, the application never closes the loading screen and fails to provide the end-user with insight as to what happened.

We now present each of the mutant types and how each of the behaviors have been observed per mutant.

3.4.2 Behaviors Per Mutation Type

3.4.2.1 Field Removal

As explained previously in subsection 3.2.1 we expected field removal mutations to be the mutations which more faithfully represent a web API evolution scenario. This

particular mutation was applied in a particularly invasive way (i.e. all the child nodes of each type have been removed at least once) which coupled with its disruptive nature (i.e. the message is valid but is missing data) lead us to expect such mutations to be the most challenging in testing the robustness of the mobile applications.

Force Close. Our results show that 13 applications out of the 43 under study (approximately 30%) when faced with field removal mutations crash with a *force close*. Using our approach also allowed us to narrow down on exactly which fields were causing the applications to crash. In fact, 12 out of the 13 crashing applications crash with *one single field* being removed (the particular field is found following the steps in Section 3.2.1). The remaining application required two fields to be removed in order to crash. While we have no metadata for any of the web API responses, all the 13 crashing applications allowed for *some* fields to be removed without crashing.

This result also confirmed our initial expectation. As can be seen in Table 3.3 the field removal mutation is the one that resulted in the most applications crashing.

Web API ownership may also play a role in hardening a client application against web API changes. For example, the *NS.nl* web API is also used by a third party application (*NL Treinen 2 - NL*) which does not crash with field removal mutations (as opposed to the *NS.nl* application). When asked about this, the interviewed developer for the *NS.nl* application claimed he was in control of the web API and therefore knew when the web API would change. Unfortunately the developer for *NL Treinen 2 - NL* did not react to our interview request and we cannot verify our hypothesis.

Error Message. When certain fields were missing from the web API response, there were 4 out of the 43 applications which did show error messages. In some cases the error message is generic (e.g. *"connectivity issue"*) and not specially tailored to inform the user on how to proceed.

No Indication. More troubling than showing an incomplete error message is the behavior displayed by 39 out of the 43 mobile applications under study (n.b. the 39 applications which show no indication include the 13 which force close). These applications show no indication of completion or failure whenever one or more fields of the response are removed.

3.4.2.2 Malformed Response

Mutations of the type malformed response resulted in the same three different behaviors as the field removal mutation, with an added behavior where applications failed to timeout.

Mutation	# of apps w/force close	# of apps w/o force close
Field Removal	13 applications	30 applications
Malformed Response	1 applications	42 applications
Empty Response	1 application	42 applications
Changing Data Type	3 applications	40 applications

Table 3.3: Force Closing Applications per Mutation

Force Close. Having one single application crashing upon receiving a malformed response (namely *Wetter.com*) is an indicator that the majority of the applications are accounting for the scenario where the document parser fails due to a faulty document. One other application (*Le Parisien*) would become unusable upon facing a malformed response but rather than force closing, it would report to the user that it cannot continue and then gracefully close itself.

Error Message. The *NS.nl* application, when facing a malformed response makes use of the Android native dialog mechanism to show a message informing the end-user that it *"cannot retrieve data from server"*. While certainly better than crashing or remaining silent about the failure, it is also an example of a generic message which does not offer an indication of how the end-user should proceed.

No Indication. While malformed responses are an unmistakable case of failure somewhere along the connection with the web API, it is then surprising that 31 applications (72%) give no indication of the unrecoverable error or of what is the right course of action for the end-user.

Also in this category, we found two different types of applications. In some cases, such as the *tramTracker*, the application still attempts to load whatever data is available in the damaged response (whilst giving no indication that it was damaged). In contrast, applications such as the *Resultats Foot en Direct* load directly onto a screen where if the response is malformed, nothing is shown. In such a case, the end-user may be endlessly waiting, expecting the data to be loaded while the application has in fact silently stopped loading.

No Timeout. Applications which indefinitely stay loading and never timeout was the most common occurrence when dealing with malformed responses. Indeed, 8 applications (19%) never time out and are left stuck in a loading screen.

3.4.2.3 Empty Response

When applying the empty response mutation, all four behaviors were observed. While other mutations consist of turning the web API responses into crash-inducing mutants, we expected an empty response to be a fairly trivial occurrence without serious repercussions.

Force Close. Indeed, while only one application demonstrated this behavior, it stands to reason that the source code of *TecMundo* does not account for empty web API responses. When presented with an empty web API response, this application would immediately force close. All the other 42 applications did not crash when faced with such an empty response.

Error Message. When dealing with an empty response, 9 applications of those under study did show an error message. Of special note are 4 of these applications (*mobile.de*, *Resultados Futbol*, *Couverts* and *Trulia*) which show a message claiming that no results were found and that the end-user should change the search criteria. In fact, all these applications always return a boilerplate JSON or XML result *even* in the event no results had been found. This may then indicate that the application did not recognize the empty response as a fault. The remaining 5 applications showed generic *"network error"*

messages, with special attention to the *yr.no* application. This was the only application which actually reported an *"empty response"*.
No Indication. In contrast with the low number of applications which display an error message, more than two thirds (34 out of 43) of the applications under study provide no indication that an empty response has been received from the web API. More specifically, these mobile applications would stop loading and never present the user with a message describing why the loading had stopped and why no data had been loaded.
No Timeout. In part overlapping with the applications which provide no indication of receiving an empty response, 8 out of the 43 applications remain indefinitely loading and never time out. While it is not possible to verify the reason for this behavior in the applications' source code due to their closed source nature, it is likely the affected applications always expect a reply with content. When the content is not present, the applications hang until there is content.

In fact, RESTful web APIs may indeed at times reply with an empty message, for example with HTTP status codes 304 (Not Modified) or 204 (No Content) [12]. Although it was not the case for any of the aforementioned 8 applications, in our study we experienced web APIs which replied with an empty HTTP message having a status code of 301 (moved permanently), indicating that a request should be made to a different URL.

3.4.2.4 Changing Data Type

Due to the lack of metadata on all the web API responses, it is impossible to know with certainty whenever a field is of the numeric or string type. Nonetheless, with the exception of the timeout issues, our mutated web API responses were able to cause all of the other aforementioned behaviors (force close, error message and no indication).
Force Close. Through the use of the changing data type mutation, 3 applications crashed. While we were not able to investigate the exception being thrown, it is possibly related to the parsing of the message and to a type mismatch between Java's statically typed variables and the field values being parsed into the wrong types.
Error Message. The changing data type mutation resulted in only 3 applications actually showing an error message. Out of the 3 applications, none provided an error message which offered an explanation or solution for the problem.
No Indication. As with the other faults, some of the mobile applications silently fail without informing the end-user about the fault. In fact, 40 out of the 43 applications did not report an error even when the data would not load (in which case we were certain to have disrupted the loading of the data).

[12]REST Patterns, HTTP Status Codes — `http://bit.ly/restpatterns`

Mutation	# of apps w/timeout	# of apps indefinitely loading
Malformed Response	35 applications	8 applications
Empty Response	35 applications	8 applications

Table 3.4: Timeout

3.4.3 Data Caching

While applying the mutations to the web API responses, some applications would not initially attempt to load the mutated data. This was due to the fact that because the data had just successfully been loaded (from our first execution, probing for a testable action), the data would be stored in cache. Until the cache timeout was met, new data would then not be loaded. Some applications would still send a request, but the response would be discarded because of the cache. While this approach to caching does not help with minimizing web API interaction and thus network data usage, it may help as a backup whenever network connectivity is not available.

This led us to investigate (1) how many of the mobile applications under study were making use of caching and (2) how the caching is implemented. While one of the main uses of caching is to reduce network traffic [Wang, 1999], its usage can also help in cases where connectivity is temporarily lost, which is a potential risk when using wireless networks. In particular, caching data does help to some degree in dealing with a web API whose responses should not be taken for granted (as opposed to what happens with a statically linked API).

Our results show that the majority (32 out of 43 applications) do not use caching (see Table 3.1). Amongst all the applications which do make use of caching (with the exception of *Huizen*), quitting the application would not clear the cache. We had to manually clear all data of the Android application using the built-in Android data clearing function. We therefore assume these applications implemented time-based caching.

3.4.4 Versioning

In Chapter 2 ([Espinha et al., 2014b]) we highlighted the importance of versions in the web API context. Especially when the client developers have no control over when changes happen to the web API behavior, versioning that behavior allows the client developers to know what behavior to expect from a particular web API. Versioning, either in the URL (e.g. `www.weather.com/v1/report`) or through variations of semantic versioning (as demonstrated by *OZSale*), allows client developers to know *when* to expect changes. Indeed, in the case of *OZsale*, the web API currently at version 3.4 is guaranteed to not introduce breaking changes in all the minor versions of the 3.X release.

It is then surprising how such a high percentage of mobile applications make use of the web API without any form of versioning (58%). When a non-versioned web API introduces changes, all the clients which have not yet migrated to the latest version will

Mutation	# of apps ⇒ error message	# of apps ⇒ silently fail
Field Removal	4 applications	39 applications
Malformed Response	12 applications	31 applications
Empty Response	9 applications	34 applications
Changing Data Type	3 applications	40 applications

Table 3.5: Mutations versus Error Messages

be interacting with a changed web API, which may not be compatible. Evidence of a scenario where this would potentially happen would it not be for versioning comes from one of the interviewed developers. The developer interviewed in the context of the *OZsale* application claimed that indeed, some end-users do not update their mobile applications and that 5% of their user-base (of 100,000∼500,000 users according to the Google Play Store) was still using their mobile application's very first version.

3.4.5 Developer Interviews

We conclude our study with the findings gathered from interviewing the three participants in our study. In the paragraphs below we refer to the questions shown in Table 3.2.

Insecure HTTP

Referring to question **Q1** regarding the design decision of using HTTP over HTTPS, an intriguing finding of our study is how such a large number of mobile applications (indeed, all the 43 under study) still make use of insecure HTTP. Data sent over HTTP allows for the data to be both eavesdropped upon and, indeed, tampered with in the same fashion as is performed in this study. When confronted with this question, one of the developers claimed he did not in fact know why their web API was using HTTP because the web API is developed by a different team. According to the developer their mobile application does make use of HTTPS for login and payment interactions.

Caching

While all the interviewed developers perceive caching (question **Q2**) as a useful mechanism to reduce network usage, specifically the developer of the *NS.nl* application raised a concern about the necessity for *"fresh data"*. Indeed, this application provides information on the Dutch train departure and arrival times which are at times susceptible to delays. It is thus crucial to always display the latest data.

Another interviewee (the mobile software architect for *OZSale*) justified the lack of caching as it being a lower priority requirement. While such a feature is already present in the iOS version of the mobile application, at the time the Android version started being developed *"the libraries available for caching in the Android platform were not yet mature enough"*. The iOS application goes a step further and makes all of the data available for offline browsing.

Also the developer responsible for the web API at *Trivago* stated that caching would in fact stay in the way of the mobile application's performance for their specific case. Caching would require the mobile application to keep track of which data it has available and only request the delta between what it already has and the results it still needs to fetch. To do so with caching and without state would make for chatty communications. The mobile application would have, with every request, to report what it already has in cache and what it requires. *Trivago* contains a more pragmatic approach where sessions

(i.e. stateful exchanges) are used which allows the cache to be on the server-side and thus lower the chattiness which is desirable for both performance and data usage.

Versioning

Two of the three interviewed developers (for the *Trivago* and *OZsale* mobile applications) have versioning mechanisms implemented in their respective web APIs.

For instance, the *Trivago* web API makes use of HATEOAS [13] (Hypermedia as the Engine of Application State) versioning approach. The HATEOAS approach makes use of HTTP headers (Accept-Type and Content-Type) as a way to handle versioning and description of the data since it stands central to being *the* way a RESTful web API should be versioned. When asked about why this particular versioning mechanism was used, the answer was that even though the Trivago web API is still in its first version, HATEOAS was chosen as a way to future-proof the evolution of the web API.

The developer of the *OZsale* application also stressed the importance of their versioning system. While the data itself is not versioned (as it happens with HATEOAS), a version number must be used in the URL to inform the server of which version of the web API the mobile application requires.

An interesting divide between the two aforementioned developers is how old versions of the web API are handled. The Trivago software architect underlined that they try to avoid maintaining different versions in parallel due to costs, while the *OZsale* developer claimed that the different versions were a core part of their different platforms: while the website was running on the latest version of the web API, the different mobile platforms were lagging at least 5 minor versions behind (all of which were still available and fully functional). A reason for this was the delay between submitting a new version of the mobile application to the respective application store and the application actually being available (e.g. the developer claimed a 1 week delay in the iOS App Store).

The developer of the *NS.nl* mobile application claims that over the course of four years of development, no breaking changes have been applied to the web API, thus making a versioning mechanism unnecessary.

Evolution & Communication fragility

Another interesting finding is anecdotal evidence of communication issues between the different teams involved in the development process. Indeed, one of the developers claimed that at least twice in their project, changes were pushed to the web API which inadvertently broke backwards compatibility. The result was having a mobile application which was crashing. This anecdote raises an issue which is also supported by our analysis of the 43 mobile applications: not all mobile applications are built with the consideration that the web API can change *at any time*. This was especially relevant as in this very same project, changes were being pushed daily with breaking changes taking place every two months highlighting the need for excellent communication between the mobile teams

[13] Versioning REST Services — `http://bit.ly/versioningrestservices`

and the web API team should these teams not be one and the same (questions **Q4**, **Q6** and **Q6.1**).

Integration Testing

While using a static library it is possible to test it and expect it to behave the same. However, when using web APIs where the behavior can change due to a simple patch which fixes what was buggy (but expected) behavior can cause the mobile application to suddenly misbehave. This highlights the importance of both positive and negative testing, that is testing both scenarios which are part of the use cases as well as unexpected but potential scenarios.

Our empirical data suggests that Çalıklı and Bener's [Çalıklı and Bener, 2013] observation on confirmation bias regarding testing may indeed affect some of the studied applications. Indeed, while some of the applications may have automated tests (which we cannot confirm due to their closed source nature), they may be positive tests which *"make their program work rather than breaking the code"* as would be the case with negative tests. In our interviews, we questioned the participants on whether their application makes use of any kind of testing (**Q9**). Our results show that for some of these applications a simple mutation such as malforming the web API response caused a crash. Considering the *OZsale* application as an example, the interviewed mobile software architect claimed they do perform automated testing for some bad scenarios which may potentially happen, this very same application would remain loading indefinitely when faced with a malformed response.

3.5 Threats to Validity

External validity. Our study which includes 43 applications, is composed solely of Android applications. Other mobile platforms which make use of web APIs such as iOS or Windows Phone should also be explored. Perhaps in some platforms it is more or less difficult to cause the whole application to crash. As such, in future work we will perform a similar study on both the iOS and Windows Phone platforms.

Similarly, our study can only be applied to mobile applications which make use of the insecure HTTP protocol. This both limits the number of mobile applications which can be used. Mobile developers who intentionally chose to use HTTPS over HTTP are perhaps more conscious regarding the differences which make *web* APIs different from the non-web counterparts. Indeed, without modifying the Android platform itself, nothing can be done to mitigate this threat.

Internal validity. While our study intercepts and mutates web API responses and analyzes mobile applications' reactions to these mutations, we did not consider whether these mobile applications send failure data back to the respective software developers for further analysis. Such data, should it exist, may complement and aid the debugging task.

Another threat to validity stems from potentially long timeouts (e.g., several minutes) when reacting to mutated web API responses. In such case an application could poten-

tially lead to a misclassified application. While a threat, unnecessarily long timeouts would also potentially hinder the usage of such applications for end-users.

Reliability validity. Our mutation analysis approach requires human intervention when capturing a standard web API response and replacing the response with its mutated counterpart. This raises a possible reliability threat. We mitigate it by starting with a slightly mutated response (e.g., changed string) whilst maintaining its validity as a means to ensure that the mutated response is indeed being loaded.

3.6 Related Work

Li et al. [Li et al., 2013] highlight the evolution challenges of web APIs over statically linked APIs and provide a set of potential changes which web APIs may implement. They analyze what are common changes applied to web APIs and propose the creation of a tool for automated client migration.

McDonnell et al. study API stability and adoption in the Android ecosystem and have found that, despite the added benefits of newer versions of APIs, developers tend to be slow in adopting the newer versions [McDonnell et al., 2013], thus further highlighting the awareness required when web API changes are inevitable.

An interesting non-peer reviewed work in this field is a survey [Blank (YourTrove), 2011] conducted on the pains of web API integration which presents many complaints from web API client developers.

Daigneau focuses on the brittleness of web APIs and proposes to refrain from creating signatures with long parameter lists [Daigneau, 2011]. Daigneau further states that long parameter lists *"[...] signal the underlying framework to impose a strict ordering of parameters which, in turn, increases client-service coupling and makes it more difficult to evolve the client and service at different rates."*

3.7 Conclusion

In this chapter we perform a study on the impact that changes to web API behavior can have on mobile applications. Our contributions are:

- An approach using mutation analysis for simulating unexpected responses from web APIs.
- A study on how 43 high profile mobile applications react to a set of predefined mutations in web API responses.
- Insight on caching and versioning approaches of some of the web APIs under study.
- An interview with three developers of some of the studied mobile applications.

Referring back to our research questions proposed in the introduction, we set out to find how robust mobile applications are when facing unexpected responses from web APIs. The first question we answer is [**RQ1.1**] on *"how can we simulate unexpected responses from web APIs"*. The mutation analysis presents a structured approach to simulate web APIs afflicted either by failure or by changes caused by software evolution.

Using mutation analysis we are then able to address [**RQ1**] which asks *"how robust are mobile apps when the web APIs being used return unexpected responses?"*. Our results present a mixed answer to this question. Indeed, most of the mobile applications studied are fairly robust to mutations in the web API response as seen by only 30% of the applications studied crashing through the field removal mutation. Nonetheless, all but one of the aforementioned applications can be crashed through the removal of one single field which presents a serious concern for some web API client developers. Less serious but also worrying is how for all the mutations, more than half of the applications silently hide the faulty web API response. This behavior should be made more informative and user-friendly, which can be achieved through better understanding potential changes to web APIs.

Also [**RQ2**] which asks *"have web API client developers developed resilience against changes in or failure of the web API?"* is answered with mixed results. Some of the applications studied make use of state of the art approaches (e.g. the HATEOAS versioning) to ensure a smooth evolution of their web API client, where others do not use versioning altogether (which as reported in Chapter 2 [Espinha et al., 2014c] may cause long-term pains) and allow the application to crash. The need for this resilience exists also outside of the source code. One of the interviewed developers raised concerns with inter-team communication, highlighting the need for clear and concise documentation from web API providers to client developers.

Our main research question asks *"how well-prepared are Android mobile applications with regard to changes in response messages from the web API"*. We conclude that while the majority of the studied applications are capable of dealing with such changes without major issues, some applications still use web APIs as if their behavior can be expected to never change, which as we have seen does not always happen. Rather than trying to generalize the results for all the web API clients, our goal is rather to raise awareness to the fact that amongst some of the most popular Android applications, a fair share still allow their web APIs to significantly affect their behavior.

Future work. We aim to extend our investigation to paid mobile applications and other platforms (e.g. iOS) as we want to understand whether the underlying platform provides more or less support for web API integration.

As a result of this study, we also would like to investigate the frequency with which each type of change appears in practice for each of mobile applications' web APIs.

Another aspect worth investigating is whether the owner/creator of the web API influences client developers to use validity checks to the web API response.

CHAPTER 4

SOA: Proposing a Standard Case Study

Maintenance research in the context of Service Oriented Architectures (SOA) is currently lacking a suitable standard case study system that can be used by scientists in order to develop and assess their research ideas, and for comparison, and benchmarking purposes. It is also well established in different fields that having such a standard case study system brings many benefits, in that it helps determine which approaches work best for specific problems. For this reason, we decided to build upon an existing open-source system and make it available for other researchers to use. This system, Spicy Stonehenge, provides many advantages for carrying out maintenance research: it is complex, extensible and, most importantly, openly available for anyone to use and build upon. With this paper, we introduce Spicy Stonehenge as the standard case study SOA system, and we also present our research vision in the field of maintenance, reengineering, evolution and testing of SOA systems, and how these goals fit together with Spicy Stonehenge.[1]

4.1 Introduction

While the actual term Service-Oriented Architecture (or SOA) was first coined in the mid 1990's by Gartner [Natis, 2003; Josuttis, 2007], the ideas behind it, namely building software systems that are composed out of loosely coupled, interoperable components or services, goes back further. It was, however, the technology of *web services*, launched in 2000 as a set of standards to allow computers to communicate with each other [Josuttis, 2007], that acted as a catalyst for both industry and academia to really start investigating the possibilities of Service-Oriented Architectures. In particular, SOAs promise to (1) allow businesses to be more flexible as business needs change and (2) ease evolution due to the loosely coupled nature of the system [Gold et al., 2004].

[1] This chapter contains our work together with Cuiting Chen, Andy Zaidman and Hans-Gerhard Gross, published in the proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012) [Espinha et al., 2012a]

When looking at the past decade of research in service-orientation, we can observe that although a lot of fruitful research has been carried out (e.g., see [Benatallah and Motahari Nezhad, 2008; Benbernou et al., 2008]), many of the research efforts are isolated in nature. While this isolation is not bad per se, it does hinder progress. Symptomatic of the isolated nature of research in this area is the absence of a common case study that can be used as a benchmark. Indeed, Sim et al. report that benchmarking, when embraced by a community, has a strong positive effect on the scientific maturity of a discipline [Sim et al., 2003]. In particular, it allows to easily compare solutions and to perform replication studies. In many fields of software engineering, researchers have resorted to benchmarking in order to compare approaches and ultimately advance the field. Prime examples being the aspect-mining community that settled on *JHotDraw* as a standard case study system [Ceccato et al., 2006], or the refactoring community that introduced the *LAN simulation* [Demeyer et al., 2002].

In order to unify the SOA community around a single case study, we propose a system that is at the same time realistic, easy to understand, and which most researchers should be able to use as a "standard case study system". The system we propose — Spicy Stonehenge — is based on Apache Stonehenge and consists of an application composed out of several web services. The open-source nature of Spicy Stonehenge and its availability should stimulate researchers in the area of SOA, that normally resort to small examples built specifically for the context of their research, to choose for Spicy Stonehenge, thus enabling the benchmarking process that the community needs.

This chapter makes the following contributions:

- We introduce Spicy Stonehenge, an open-source service-based Java software system as a possible standard case study system for researchers working in the area of SOA.
- A brief survey of existing research initiatives in the area of SOA from which we extract criteria that need to be specified when performing a case study in order to allow future comparison and/or replication.
- A research agenda for online evolution and online testing in the area of SOA.

This chapter is structured as follows: in Section 4.2 we present a background of similar research being done in this field, Section 4.3 describes our service-oriented system (Apache/Spicy Stonehenge) in detail, Section 4.4 provides an overview of our future research agenda and lastly, Section 4.5 presents a summary of what is discussed in this chapter.

4.2 Background Research

In our reconnaissance of the research area of Service-Oriented Architectures, we noticed that there is no standard case study system being used by researchers. Furthermore, during our exploration of the field we also got the impression that a wide variety of small and/or closed source systems were being used as case studies for evaluating the research.

In order to get a better feeling of how research in the area of SOA is conducted, we have performed a small literature survey where we specifically focused on the software systems that are being used in case study research.

In order to characterize the case study systems being used in SOA research, we compiled Table 4.1, which represents a small subset of research papers in the area of SOA. The papers that we selected for this overview originate from:

- The state-of-the-art report on service monitoring from the European S-Cube[2] project on software services [Benbernou et al., 2008]. We selected this survey because our research goals are aligned with many of the papers mentioned in this report.

- A selection of recent papers published at venues like CSMR, ICSE and ESEC/FSE, from which we expect a thorough validation.

The 14 papers listed in Table 4.1 are all representatives of case study research [Wohlin et al., 2000]. We now list some of our observations:

Self-created case study systems. From this selection of papers we noticed that some authors created their own simple non-industrial examples as case systems, which contain a very small number of services, e.g., [Domenico and Carlo, 2007] and [Ardissono et al., 2006] have one and three services respectively. It is arguable whether these small case study systems are representative of real service-based software systems. Some self-created systems also appear more complex. For example, Baresi et al. [Baresi et al., 2004] describe an IT certification system which gives enrolled students a chance to try a certification test for free. However, the paper only describes the conceptual details of the system.

An important issue with self-created systems is that their set-up might be favoring the researched technology, which becomes extra hard to verify when these self-created systems are not publicly available. Looking at Table 4.1 we see that unfortunately, almost all systems are not publicly available.

Closed-source systems. Some researchers are cooperating with industry and have the chance to get a real-world system as their research vehicle. For example, Momm et al. [Momm et al., 2007] apply their approach to a practical scenario developed in a project aiming to redesign a university's business process; Nasr et al. [Nasr et al., 2011] provide an industry case study supported by a business service IT company. Also, in the paper by Pistore et al. [Pistore and Traverso, 2007], the authors mention that their approach was applied to some real applications, but no more details are provided.

[1] AECS: Amazon E-Commerce Services — `http://webservices.amazon.com/ AWSECommerceService/AWSECommerceService.wsdl`

[2] MPS: Monte dei Paschi di Siena Group(an important Italian financial Group) — `http://www.mps.it/`

[3] KIM project — `http://kim.cio.kit.edu/`

[4] del.icio.us — `http://delicious.com/help/api`

[5] OpenSocial — `http://code.google.com/apis/opensocial/docs/0.7/reference`

[2] S-Cube — `http://www.s-cube-network.eu`

Industrial case studies are extremely important in software engineering research, however, due to the closed-source nature of these software systems they cannot be obtained by other researchers. This means their results cannot be reproduced or compared, which strengthens our call for a common case study to compare techniques on.

Implementation technology. During this survey, we also focus on investigating the implementation technologies used in those case study systems, such as the programming language, the underlying frameworks, the communication protocols, etc. These pieces of information are necessary in different situations, e.g., (1) when practitioners want to use the experimental results and want to verify whether the results are applicable in specific circumstances or (2) when researchers want to replicate a study or perform a meta-analysis [Kitchenham et al., 2002].

However, as Table 4.1 shows, most papers do not provide implementation details. The notable exceptions are paper [Schmerl et al., 2011] and paper [Nasr et al., 2011], which clearly mention that their systems are built on the Apache CXF framework and the IBM WebSphere respectively. In the case of industrial case studies, sometimes the APIs are open, but the implementation techniques are kept confidential.

Summary and recommendations. The small survey that we present in this section makes it clear that comparative studies or replications are difficult to perform considering the fact that many (implementation) details are not presented in the papers considered. While this is perfectly understandable in the case of closed-source software systems, this is less so in other cases. These observations reinforce our stance that the SOA (maintenance) community would benefit from having a standard case study system in order to benchmark solutions.

When reflecting on the case studies that we came across during our small survey, we established a number of details that we would ideally want to know from all case studies:

- The implementation technology (e.g., the programming language or the communication protocol) and the used frameworks.
- The complexity of the service-based system (e.g., the number of services or interfaces).
- The availability of the system.

With these criteria in mind, we will introduce and describe Stonehenge, the standard case study system that we propose in the next section.

Table 4.1: Selected SOA research papers with case studies

Paper	**Description**	**Complexity**	**Impl. Tech.**	**Availability**
[Barbon et al., 2006]	Runtime monitoring of web service compositions	3 web services	N/A	No
[Pistore and Traverso, 2007]	Assumption-based composition and monitoring of WS	1 web service	N/A	No
[Domenico and Carlo, 2007]	Monitoring functionality of conversational services	1 web service with 3 interfaces	N/A	No
[Heward et al., 2010]	Assessing the performance impact of service monitoring	1 web service	N/A	No
[Marconi and Pistore, 2009]	Synthesis and composition of web services	2 services: AECS[1] and MPS[2] e-payment service	N/A	Industry, API avail.
[Ardissono et al., 2006]	Exception handling for web service orchestration	3 web services	N/A	No
[Baresi et al., 2004]	Dynamic monitoring service compositions	Unknown	N/A	No
[Mahbub and Spanoudakis, 2005]	Runtime monitoring requirements for service composition	Unknown	N/A	No
[Momm et al., 2007]	Monitoring process for SOA	Unknown, KIM[3] project	N/A	No
[Nasr et al., 2011]	Adopting and evaluating SOA in industry	700+ services	J2EE, IBM WebSphere etc.	Industry case
[Ahmad and Pahl, 2011]	Transformation-driven evolution for SOA	Unknown	N/A	No
[Schmerl et al., 2011]	A case study in SOA-based platform design	120+ services	Apache CXF etc.	No
[Bertolino et al., 2009]	Automic sysnthesis for composable web services	1 service: AECS[1]	N/A	Industry, API avail.
[Denaro et al., 2009]	Ensure interoperability for service-oriented systems	del.icio.us[4]and OpenSocial[5]	N/A	Industry, API avail.

4.3 Stonehenge

Apache Stonehenge[8] is a simulation of the stock market consisting of a web application and several web services. Stonehenge provides the possibility to buy and sell shares in a single stock market, with a single currency. Apache Stonehenge was built as a joint cooperation between Microsoft and the Apache Software Foundation to showcase service interoperability between different technologies.

Our goal, however, is not to explore the field of interoperability but that of maintenance in SOA, and all that it entails. We chose Stonehenge as it provides a real world example of how services can interact together to compose a software system. However, conscious of its size, we decided to extend it in order to make it more realistic and complex. We have extended it with several new features to make the system more complex on what concerns business logic and number of services. That is, we added the possibility to maintain several wallets in different currencies, to exchange money amongst the different currencies, and to use real-world data from the stock market. The result of our changes is called Spicy Stonehenge[9] which relies substantially on the business logic of the original implementation. We have also ported the original JAX-WS-based implementation to the Turmeric SOA platform[10] due to our research agenda. More on this can be found in Section 4.4.

4.3.1 Motivation

In our background research (see Section 4.2) we have established that in service-oriented research there is no case study which researchers can use to compare their approaches and results. For this reason, we decided to bring forth a system that meets the criteria needed for a standard case study. For such a system we feel it is necessary that: a) it reproduces the behavior of a real-world system, b) is large or at least provides many extension possibilities that all researchers can build upon and c) it must be easy to port to different frameworks.

With Spicy Stonehenge we feel we have met these three criteria. Spicy Stonehenge provides similar behavior to that of the stock market, it is already fairly large in number of services and we plan on extending it to make it even more similar to a real system. This way, we believe Spicy Stonehenge can become the standard system which every researcher in this field can use as the "common software system" mentioned in the work of Sim et al. [2003].

4.3.2 System Description

The current version of the system is composed out of five different services and two databases (Fig. 4.1). In this section we provide an overview of what each service does and

[8] Apache Stonehenge — https://cwiki.apache.org/STONEHENGE/
[9] Spicy Stonehenge — `https://github.com/SERG-Delft/spicy-stonehenge`
[10] Turmeric — `https://www.ebayopensource.org/index.php/Turmeric/`

further into the section, what data is stored in each table.

Also referring to Figure 4.1, solid arrows represent one service invoking another whereas the dashed arrow represents a publish/subscribe connection where the Order Processing service can subscribe to topics on the external service.

Services:

- The **Configuration Service** acts as a registry for all the deployed instances of the other services. All the other services need, therefore, to know in advance the endpoint of at least one instance of the *Configuration Service*.

- The **Business Service** mediates the interaction of the web application with the business logic of the system. For this reason, the Business Service contains all the operations the web application is capable of performing. They are: the buying and selling of stocks, user registration, statistical information about the market and information about stock prices.

- The **Order Processing Service** is solely responsible for processing the buying and selling of shares. It is meant to be invoked by the *Business Service* whenever a user performs a purchase or sale of shares in the web application.

- The **Exchange Service** makes use of Google's API for currency exchange. This service is invoked whenever the user explicitly requests for currency to be exchanged from a wallet in a certain currency into another wallet, with a different currency. In

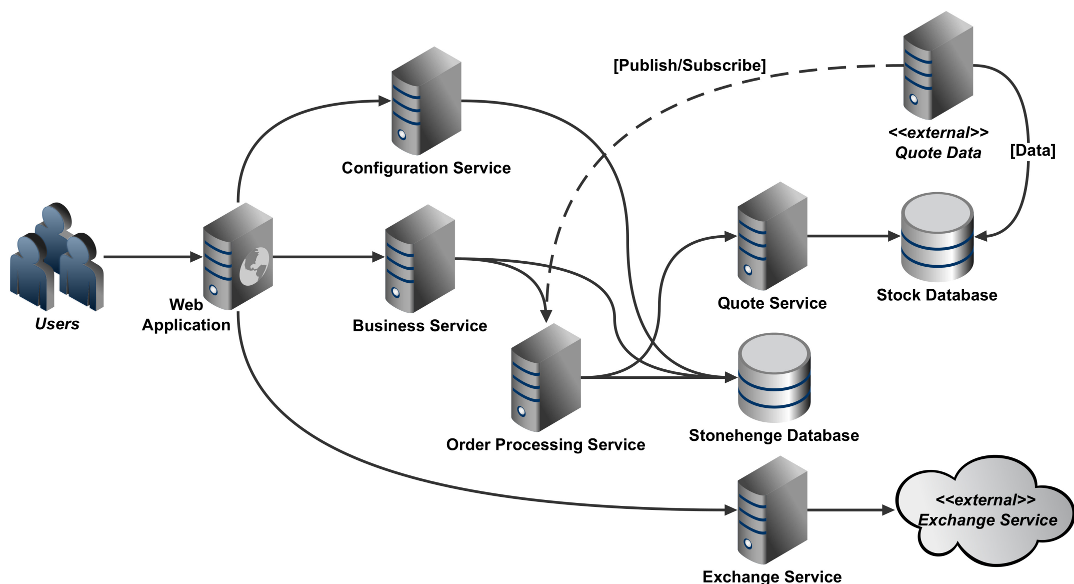


Figure 4.1: Spicy Stonehenge

the future, this service will also become part of the purchase request for the cases when the user wants to buy shares in a currency A but chooses to use currency B.

- The **Quote Service** is in fact composed of two services. Referring to Figure 4.1, the service described as *Quote Service* is a normal *pull-based* service with a SOAP interface that the *Order Processing Service* can invoke to obtain data about specific stocks on-demand. On the other hand we also have the *Quote Data* service which performs two tasks: 1) it fills the *Stock Database* table with data and continuously updates it with data from Yahoo Finance, and 2) it provides a publish/subscribe interface (implemented using the ZeroMQ library) which other services, such as the *Order Processing Service* can bind to in order to be notified for price changes in specific stock symbols.

Databases:

- The **Stonehenge Database** contains the information necessary for the basic operation of the system. Namely it contains user information, including how much money and which stocks each user owns. It also contains information about the services' endpoints and the mapping between service instances (which instance should each service use).

- The **Stock Database** contains solely information about stock prices. This table is kept separately as it is meant to be filled by an external service which continuously checks whether there is new data and pushes it to the database.

4.3.3 Usage Scenarios

With these services we can then have different usage scenarios. These are summarized in Table 4.2.

Table 4.2: Features available/planned for Spicy Stonehenge

Currently available features	**Planned features**
Purchase and sale of stocks	Automatic conversion of currencies
Price information about stock symbols	Multiple stock markets
Wallets in different currencies	External bank entities
Management of service endpoints	Stock options
User registration	

Our planned extensions to the existing system aim at making the interactions amongst web services more complex. For example, the existence of multiple stock markets will create the need for different instances of the Order Processing service. Similarly, having an automatic conversion of currencies will add a possible additional step to the purchase of stocks.

4.4 Research Agenda

The purpose of compiling and proposing Stonehenge as a standard case study system is tightly connected with our research goals. In past research we performed dynamic analysis on SOA systems in order to identify dependencies amongst services [Espinha et al., 2011]. Chen et al. have also investigated the challenges of performing runtime monitoring of SOA systems [Chen et al., 2011]. Our preliminary results are promising but our goal is to further investigate these topics.

In order to support this earlier research, we have also ported the original Apache Stonehenge to a different platform, i.e., Turmeric SOA. This platform was built and recently open-sourced by eBay and it already provides many of the features that we require for runtime monitoring of SOA systems. This platform is also highly scalable as proven every day by supporting eBay's own business. With it we hope to replicate as closely as possible a real-world setting.

We have also been collaborating with the engineers from Intalio who are in charge of making Turmeric SOA open-source. We plan on leveraging this collaboration to learn about the pains of online SOA maintenance, reengineering and testing.

In the future we would like to investigate online evolution of SOA in more detail. This entails two major research tracks: online updating of services and versioning, that is performing an evolutionary step, and online diagnosis and testing, which consists of assessing it. Following are a few of the research questions we want to explore in the field of reengineering, evolution and maintenance.

4.4.1 Online Updating and Versioning

Performing online updates of services comes with several challenges. For instance, if the external behavior of a service changes, this might cause other services to fail (since they expect the older behavior). For this reason it is important to know exactly which services depend on which, as to have a clear picture of which services a code change can impact.

Similarly, when interfaces of services change, it might be the case that a new version of the service is created and the older one is kept for backwards compatibility. When this happens, eventually there will be a large number of versions of the same service. With runtime information about which services are no longer used we can provide software engineers with information to help mitigate this problem.

In an online system it is also important to have knowledge about the usage patterns over time. This way, maintainers can attempt to reduce downtime to a minimum by choosing the periods of time with less usage on specific services.

These concerns can be summarized as three research questions:

- How can we determine which services depend on which, at runtime?
- How can SOA maintainers determine the best periods to perform maintenance?
- When can a version of a service be deprecated based on its usage profile?

4.4.2 Online Diagnosis and Testing

For our research, we assume that the business logic of all deployed services is tested individually, and that the integration of the deployed service in the SOA has been checked [Bertolino and Polini, 2009]. Checking the SOA during runtime then comes down to observing its behavior (through monitors) and waiting for a failure to appear. The root cause of a failure can be pinpointed through a specific lightweight diagnosis technique referred to as spectrum based fault localization (SFL) [Piel et al., 2011]. This can be used during runtime to convict or exonerate a potentially faulty service [Reps et al., 1997]. SFL is based on a matrix of service involvement in transactions carried out in the entire SOA. Based on this matrix SFL can deduce likely faulty services during runtime after a number of observations have been recorded.

However, applying SFL online creates a number of challenges:

- How can we determine service involvement for SFL through monitoring and tracing of the SOA? Our goal is to determine the current architecture of the running SOA, and, therefore the width of the SFL matrix. In other words, how many services are there, and how are they interconnected at a certain point in time?

- How to limit a spectrum for *online* SFL? The spectrum of a service represents which transaction it has been invoked in. Therefore, in the online case, a spectrum can become indefinitely large. This question is related to length of the SFL matrix. In other words, how much execution history is required for the diagnosis?

- How can we generate test cases for observations that are required for a particular diagnosis, but that have never been made?

- What is the overhead of applying online SFL to SOA? How can we determine the trade-off between the overhead of the online testing and diagnosis and the systems' performance?

Currently with Turmeric it is possible to visualize usage profiles of different services over time allowing us to have an idea of which services are most used and at what times. This feature already partly addresses two of the previous research questions as it provides information about which pairs of services are invoked. One of our aims is to extend this feature and provide more detailed information about the runtime system. This way, developers in charge of maintaining SOA systems can, for example, schedule maintenance tasks appropriately and therefore lower the perceived downtime for customers.

4.5 Summary

The motivation in many organizations for deploying and using service oriented architectures is the loosely coupled nature of their services, facilitating flexibility in adapting systems to changing business requirements and alleviating constant system evolution. Therefore, SOAs are predestined to realize easily maintainable distributed systems.

Our research agenda addresses some of the challenges encountered when service oriented architectures are maintained while they are operating and being used by various stakeholders. We refer to this as online evolution and maintenance.

Working in this area and devising techniques for online maintenance of SOA brings with it the requirement of having a realistic case study system for experimentation, assessment, and comparison. However, when venturing into the subject, we noticed a chronic lack of suitable applications to be used as case studies. We could have used an industrial application, which would have facilitated our research goals, thereby sacrificing open access to other researchers and obstructing open discussion and exchange of ideas. On the other hand, open source SOAs are not readily available, which might well be attributable to the fact that the typical scalability requirements of organizations that lead to SOA deployment, are not apparent in the open source community.

In this chapter, we have, therefore, addressed this lack by proposing an open source SOA system, i.e., Spicy Stonehenge, which we developed out of an existing application, and now put forward as standard case study system. We hope that in the future more researchers will use and contribute to this case in order to suit their particular research interests, but also in order to facilitate comparison between all ideas and techniques developed based on this system. That way, we hope to contribute to the overall maturity of software maintenance and reengineering research in general.

In addition to developing a suitable case study for our research, we have also started to use it for our purposes, as outlined in our research agenda.

CHAPTER 5

SOA: Understanding its Runtime Topology

Through their dynamic and loosely coupled nature, service-oriented architectures (SOA) are ideal for realizing runtime evolvable systems. However, runtime evolution demands understanding of the configuration of SOA-based software systems over time. In order to keep system disturbance as low as possible while replacing existing services with their newer versions, engineers require an adequate illustration of how the connections and dependencies of services change in the running system. That way, they can identify the most appropriate point in SOA-based systems' operation time for maintenance. In this paper, we make use of the runtime topology to help software engineers understand SOA-based systems, and we describe a tool, Serviz*, that visualizes how services are activated, and how much they interact over time. A user study demonstrates to which extent the runtime topology can support the analysis and understanding of SOA-based software systems.*[1]

5.1 Introduction

The fact that software systems need to evolve in order to remain successful has been long recognized [Lehman and Belady, 1985]. With the advent of Service Oriented Architectures, the maintenance problem was said to become easier [Gold et al., 2004], as a Service Oriented Architecture (SOA) would be composed of several loosely coupled (and smaller) services. These smaller services are said to be easier to evolve and more flexible to compose, thus easing overall evolution. However, in 2004 already, Gold et al. warned that even though the actual evolution was possibly easier, the understanding of SOA-based systems would still be a major and costly challenge [Gold et al., 2004]. This is mainly due

[1] This chapter contains our work together with Andy Zaidman and Hans-Gerhard Gross, published in the proceedings of the 19th Working Conference on Reverse Engineering (WCRE 2012) [Espinha et al., 2012c].

to the shift from understanding a monolithic application to understanding a distributed system composed of many entities (services).

Thus, program comprehension remains an absolute and important prerequisite to evolving systems [Zaidman et al., 2010], also for systems built around the concept of SOA. In this context Corbi even reported that up to 60% of the time effort of a maintenance task lies in understanding the system at hand [Corbi, 1989]. Strangely enough then, a large-scale survey of scientific literature in the area of program comprehension using dynamic analysis from 2009 did not reveal any major advances in the area of understanding SOA based systems using dynamic analysis. As the authors note, this seems strange as dynamically orchestrated compositions of services would seem to benefit from dynamic analysis for understanding them [Cornelissen et al., 2009].

In particular, when understanding a SOA-based software system, one of the challenges lies in the uncertainty of the software construction [Gold et al., 2004]. This uncertainty is instigated by the fact that SOA-based systems are composed of loosely coupled components or services. As the composition of the entire SOA-based system only happens at runtime, the services composing the system may only be known at execution time [Canfora and Di Penta, 2007]. Furthermore, the service-composition can be changed at runtime, a situation that can become even more complicated when services can be automatically discovered [Canfora and Di Penta, 2007; White et al., 2013]. This leads to systems that are highly dynamic. It comes as no surprise then that Kajko-Mattsson et al. identified the understanding of the SOA infrastructure as one of the most important challenges when evolving SOA-based software systems [Kajko-Mattsson et al., 2008].

In order to support the understanding task, we propose to reverse engineer [Chikofsky and II, 1990] the *runtime topology* of the SOA system. The runtime topology is the configuration of a distributed, dynamically composable software system and describes which services are available and how they depend and interact with each other. Knowing exactly *which* services depend on each other, and *when* they are interacting which each other, enables a better understanding of how the SOA system is composed *and* enables a more efficient maintenance strategy as insight is obtained in when an upgrade of a service should ideally occur. As such, we consider the time-dimension of this runtime topology to be important.

In this chapter, we demonstrate how the runtime topology of a SOA can be reverse engineered from observing its operations, and investigate how the topology, augmented with a time-dimension, enables a better understanding. More specifically, our research is steered by the following research questions:

RQ3.1 Does the runtime topology help in identifying services involved in a specific use case?

RQ3.2 Does the runtime topology help in debugging a SOA-based system?

RQ3.3 Does the runtime topology augmented with the time-dimension help to identify services that are used more often (e.g., to optimize them)?

RQ3.4 Does the runtime topology help to identify periods in time when the usage of a particular service is low, e.g., for evolving it with minimal disturbance to its users?

Our proposed runtime topology reverse engineering approach is implemented in a tool called *Serviz*. In order to evaluate the approach, we set up a user study that involves Spicy Stonehenge (Chapter 4 [Espinha et al., 2012a]), a stock-market simulator, as subject system. The user study is conducted as a one-group pre-test post-test experiment and involves 8 participants.

The remainder of this chapter is structured as follows: Section 5.2 describes our approach (the data requirements, how we collect the data and how it is presented). Section 5.3 outlines the experiment we performed and Section 5.4 presents the results of our experiment. In Section 5.5.1 answer the research questions, in Section 5.5.2 we reason about possible threats to validity and we finish with related work and conclusion in Sections 5.6 and 5.7.

5.2 Approach

Central to our understanding approach stands the runtime topology of a SOA-based software system, representing the configuration of a set of services that are deployed within an environment. Before developing our approach, we compiled a number of maintenance scenarios representative for SOA-based systems. They are described in Table 5.1. Further down in Subsection 5.2.1, we present the data requirements inherent to creating a runtime topology of a SOA system that supports the maintenance tasks described.

5.2.1 Required data

In order to create a runtime topology, a significant amount of data must be collected from the SOA. We must first look at exactly which data we require to be able to build a complete picture of the SOA system, and then look at how it must be extracted and used. The runtime topology can contain different amounts of data, depending on the goal at hand. We focus, however, on the data required to satisfy our maintenance scenarios.

First, we present the different data requirements for the different scenarios outlined in Table 5.1. We then proceed to describe how the data can be retrieved and presented.

Static data

Data collected at runtime provides insight into which services are dependent on each other. This technique, however, is unable to provide a full picture of the services deployed in the system, therefore making it impossible to satisfy Scenario 1 based on this data alone. Also, in order to be able to detect unused services and satisfy Scenario 2, we can not rely on a technique which requires the code to be executed. For this reason, we also require static snapshots of what the system looks like in terms of which services are deployed in the platform. These snapshots are collected whenever a change occurs in the system.

Only by storing the latest static configuration can we know whether there are unused services.

Causal relationship

In order to enable Scenario 3 we require the causal relationships between services to be stored. To satisfy this scenario, different levels of granularity can be provided. For example, if a web service A invokes a web service B, the event describing this invocation relationship can be stored as a pair relating web service A to web service B. However, this data is too coarse. Narrowing the scope down to the method level provides system maintainers with a more focused aid as to which method is involved in a specific invocation pair thus making it easy to identify "use case flows".

These invocation pairs on their own still fall short of fully fulfilling Scenario 3. This data tells us that web service A invokes web service B and that web service B invokes

Scenario 1: Which services are available?
Motivation: Since services are loosely coupled, they are ideal to be reused in different service compositions realizing various business goals. While working on new business functionality, a maintainer requires to know which services are already deployed, and which of them can be reused in the service composite for realizing a new business case.
Scenario 2: Which services are actually used?
Motivation: An important maintenance requirement is the knowledge about the used and disused services in a SOA. When replacing services or adding new versions it is important to know whether these services are still being used. Otherwise, engineers could be attempting to treat the wrong (disused) versions of system components. Knowledge about used/disused services also represents the first step in identifying dead services that should be removed.
Scenario 3: When are certain services least used?
Motivation: On top of knowing which services there are and which are used, it is also important to know when they are used, and to which extent. If services must be updated or their new versions installed, the knowledge of their usage patterns, based on historical data, helps to determine when system updates may be conducted with the least impact on the users of the SOA system.
Scenario 4: How do services interact for realizing a particular use case?
Motivation 1: In a large system, composed of many services, an important maintenance task is the deployment of new versions of services. An essential requirement, therefore, is the knowlegde about which services are normally required in a particular use case, so that dependent services can be configured to use the new version. **Motivation 2**: When a particular feature offered by the SOA system needs to be changed, it is important to understand which other services are involved in the provision of this feature, and thus which individual services, possibly, must be updated or reconfigured.

Table 5.1: Motivating scenarios

web service C. It does not, however, tell us whether these two invocations are related. This makes it impossible to trace a single use case to all the pairs involved in providing its functionality.

In sum, to satisfy the requirements of inferring the causal relationship between web services and to tie it to use cases in a way which is useful to system maintainers, we require the invocation pairs at method level and a common request identifier which binds all the pairs to a single request trace.

Historical usage

The historical usage data is required to determine the periods of time when services have been used the least in the past (Scenario 4). This requirement is easily satisfied by adding a timestamp to each invocation pair, thus adding information about when an invocation occurred.

5.2.2 Data extraction

Dynamic data. Different web services platforms provide different means on what concerns data extraction. For our approach we chose the Turmeric SOA[2] platform, the open source version of eBay's services platform, as it already provides some of the information we described in the previous subsection. This data is not provided by default in Turmeric SOA and in order to collect it, we make use of one of the framework's features. This framework allows us to intercept each incoming request by means of a Java class which can then access and manipulate information regarding the request. Since this data is only available during the web service's invocation lifetime, the data must be stored persistently. For this reason, the request handler stores the data in an Apache Derby database in order to keep a historical track of the system's usage.

In practice, we are able to collect all the required data by handling the incoming requests from the point of view of each web service. This event provides us with all the information mentioned in Subsection 5.2.1. On the event of an incoming request, the following data is stored:

- **Request ID** - A unique identifier per invocation trace. This ID allows us to link pairs of services as belonging to the same trace.
- **Timestamp** - The current timestamp at the time of the request, as seen by the database server. This timestamp is a crucial piece of data as it allows us to analyze exactly when the web services were busiest.
- **Consumer name and method** - The name of the caller service (i.e., client) and the respective method which caused this request pair.
- **Service name and method** - The name of the callee service in this request pair and the respective web method being called.

[2]Turmeric SOA — https://www.ebayopensource.org/index.php/Turmeric/

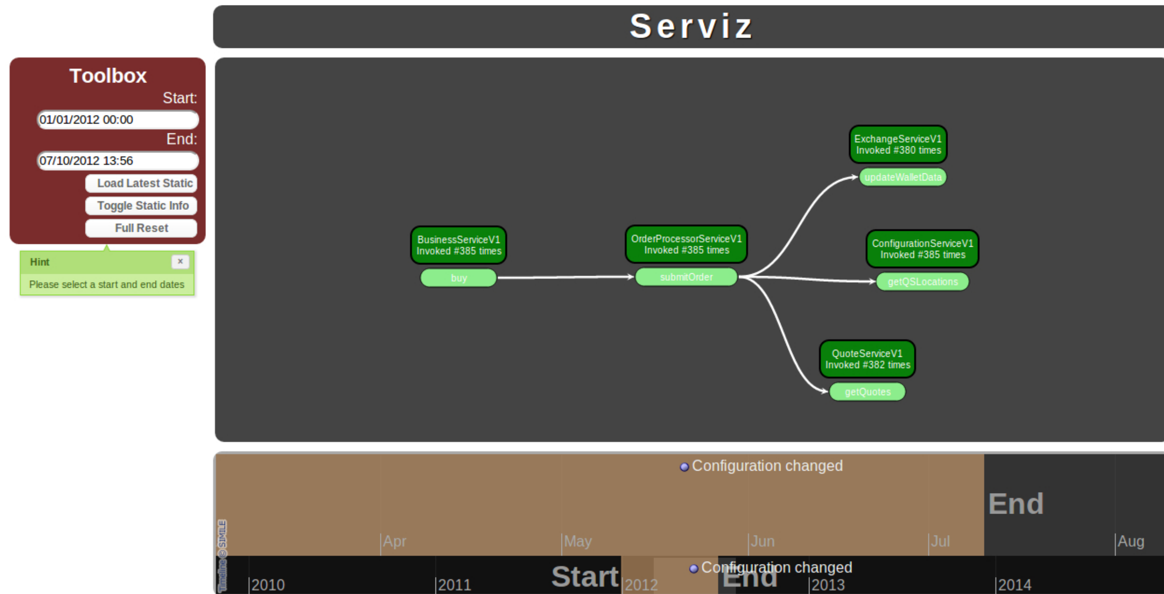


Figure 5.1: Screenshot from Serviz topology visualization

This data is readily available in string format through Turmeric SOA's handlers, except for data about the methods involved in the invocation pairs. This must be added manually by appending it to the existing request string.

Another challenge comes from how different instances of the handler are presented with different data. When a web service A is invoked, the request string available on this service does not yet contain the data identifying a possible AB invocation pair. We must then collect the data provided by all the instances of the request handler (i.e. also collect the request string from web service B, C, etc.) and then decide, based on comparing the request ID, which data is the most complete. This is stored it in the database and then queried by our visualization tool, Serviz.

Static data. In order to store the static information we created a small script which periodically inspects the application server's folder to check whether services have been added or removed; this is possible by comparing the current set of services with the latest stored information in the database. If the deployed services change (a service is removed or added), this event is stored, along with the new configuration.

5.2.3 Data presentation

The data described in the previous steps is meaningless unless transformed into information which can be consumed by system maintainers. In order to satisfy the motivating scenarios presented before and given the size of modern software systems, we resorted to visualization to present this data to maintainers. In particular, we create a dynamically updateble time-based dependency graph, thus creating a runtime topology of the system.

5.2.4 Serviz

Serviz is our main contribution putting into practice the implementation of the two aforementioned steps. It encompasses both a data collection component and the visualization component. Data collection relies on the Turmeric SOA platform, while the visualization is web-based and makes use of open-source Javascript visualization and graphing libraries. Serviz is an open-source tool and its user-interface allows system maintainers to visualize and inspect the runtime data collected from web-service based systems. All the code and detailed instructions on how to use Serviz are available on GitHub[3].

By using Serviz, maintainers are presented with a topology of a running SOA system (Fig. 5.1), and are therefore able to analyze the system's usage over user-defined periods of time. This way it becomes easier to identify possibly affected services whenever maintenance is performed in a specific deployment of a certain web service. Similarly, by introducing the time dimension to the runtime topology and allowing maintainers to scan the web service usage over past periods of time, we are able to estimate which periods of time are less busy and thus more suitable to perform maintenance.

5.3 Experimental Setup

In the experiment we performed, we assessed to which extent Serviz can help the maintenance tasks of distributed software systems. In particular, we gauged how useful, adequate and effective Serviz is throughout a series of maintenance tasks performed on a web-service-based system. The experiment is organized as a one-group pretest-posttest pre-experimental user study design [Campbell et al., 1963]. The subject system that we used for doing the maintenance tasks on is *Spicy Stonehenge*, a simulation of the stock market (Chapter 4 [Espinha et al., 2012a]).

This type of experiment is called pre-experimental, to indicate that it does not meet the scientific standards of experimental design [Babbie, 2007], yet it allows us to report on facts of real user-behaviour, even those observed in under-controlled, limited-sample experiences. In particular, the pretest-posttest design does not allow to identify an event related to the dependent variable that intervenes between the pretest and the posttest where the effects could be confused with those of the independent variable [Babbie, 2007].

5.3.1 One-group pretest-posttest

In this type of experiment, there is no control group. Instead, every participant is required to fill out a questionnaire before the assignment takes place. This questionnaire helps understanding the participant's opinions and expectations before the tool is used. After the participant uses the tool, a posttest questionnaire brings back some of the same questions from the pretest in order to compare whether the tool has in fact been useful.

[3]Serviz — http://git.io/serviz

For both questionnaires, close-end matrix questions are used where the participants can rate each question on a 1 to 5 scale, based on whether they strongly disagree up to strongly agree with the question (Likert scale).

Pretest design. The pretest questionnaire (Table 5.2) is divided into five main sections.[4]

[4]Both the pretest and posttest questionnaires are available at http://goo.gl/0Rs2r

i-a	I am an experienced Java developer
i-b	I often use Eclipse to write Java code
i-c	I am familiar with the Maven build process
i-d	I am familiar with the concepts behind web services
i-e	I have developed web services before
i-f	I am familiar with service-oriented architectures
ii-a	I have performed software maintenance before
ii-b	I am often involved in software maintenance tasks
ii-c	I mostly maintain software developed by other people
ii-d	I have used software visualization tools before
ii-e	I have maintained a distributed system before
ii-f	For this task I used static analysis
ii-g	For this task I used dynamic analysis
iii-a	Software maintenance in general is a difficult task
iii-b	Maintaining distributed software is more difficult than monolithic software
iii-c	I mostly maintain software developed by other people
iv-a	During software maintenance, I often use static analysis
iv-b	During software maintenance, I often use dynamic analysis
iv-c	I have performed dynamic analysis on a software system before
iv-d	Dynamic analysis provides an added value to static techniques
iv-e	I have used dynamic analysis tools on distributed systems before
Tool description: Serviz is a tool that provides you with a topology of the system created based on both dynamic and static information. With this tool, you can see which services are being and have been used throughout time. You can also see whenever services have not been used and which services are used the most.	
v-a	By displaying runtime artifact relationships, Serviz will help me identify artifacts involved in a specific use case
v-b	By displaying runtime artifact relationships, Serviz will help me debug a distributed system
v-c	With the dimension of time, Serviz can help me identify services that are used more often
v-d	Serviz will stand in the way of maintenance of distributed systems instead of aiding it

Table 5.2: Pretest Questionnaire

Each of these sections helps us filter out conditioning factors that might influence the user's experience with Serviz. These factors are divided into:

1. Experience with software development: has the participant used the underlying technologies supporting the case study system? (Java, Eclipse, Maven, web services, etc)

2. Opinion on software maintenance: did the participant perform maintenance tasks before? Is he/she familiar with such tasks on software written by third parties?

3. Opinion on distributed software: is the participant familiar with distributed software? Does he/she think distributed software is more difficult to maintain than its monolithic counterpart?

4. Views on dynamic analysis: has the participant used dynamic analysis techniques to perform maintenance before? Does he/she understand the added value of such techniques?

5. Expectation regarding Serviz: after reading a short description of Serviz, what are the participant's expectations from Serviz?

Posttest design. After the completion of the pretest and the assignment, participants are asked to fill in a posttest questionnaire (also composed of close-end matrix questions based on the Likert scale). Its goal is to understand how useful the participants found Serviz and whether it met their initial expectations.

The posttest (Table 5.3) is divided into five categories. We ask the participants to identify the following aspects of their experience with Serviz:

1. Overall experience: participants are asked to report on their overall experience with the assignment.

2. Usability: was Serviz clear and intuitive for the participants?

3. Usefulness: how did Serviz help in identifying different runtime aspects of the running system? In other words, how useful was it in aiding the task of system maintenance?

4. Favorite feature: in this question, participants are asked to rank their favorite Serviz features.

5. Further remarks: the last question is an open-end question where we allow participants to provide more detailed feedback about their experience and possible suggestions to improve Serviz.

vi-a	The assignments were too hard for me
vi-b	I felt a lot of time pressure
vi-c	The assignments were very interesting to do
vi-d	I felt enthusiastic about the proposed assignments
vi-e	I got enough guidance for completing the assignments
vii-a	It was clear that the thickness of the arrows was proportional to the number of calls
vii-b	It was clear I was supposed to input dates before any data was shown to me
vii-c	It was clear I could scroll the timeline to access different periods of time
vii-d	It was clear that the arrows represent dynamic dependencies
viii-a	Serviz helped me identify which services were involved in a specific use case
viii-b	Serviz helped me identify where the fault was in the system
viii-c	Using Serviz I could easily infer the dependencies between the different services
viii-d	Serviz also helped me investigate such dependencies during runtime and for different periods of time
viii-e	It was clear, using Serviz, which services are used more often
viii-f	Serviz helped me identify dominating usage patterns of the system (e.g. when is it most often used)
viii-g	By identifying dominating usage patterns, Serviz also allows me to determine the best periods for performing maintenance
ix - Favorite feature	
x - Further remarks	

Table 5.3: Posttest Questionnaire

5.3.2 Assignment

In order to evaluate Serviz we designed an assignment which aims at simulating common maintenance tasks. As an attempt to align our research with realistic maintenance tasks, we refer to the nine software comprehension activities as described by Pacione et al. [Pacione et al., 2004]. These activities are:

A1. Investigating the functionality of (part of) the system.

A2. Adding to or changing the system's functionality.

A3. Investigating the internal structure of an artefact.

A4. Investigating dependencies between artifacts.

A5. Investigating runtime interactions in the system.

A6. Investigating how much an artefact is used.

A7. Investigating patterns in the system's execution.

A8. Assessing the quality of the system's design.

A9. Understanding the domain of the system.

The scope of our tool forces us to discard three out of the nine activities, i.e., activities A3, A8 and A9. They refer to internal artefact structure, system design quality and system domain, respectively. All other activities have been specifically targeted by our assignment.

The assignment starts with an anecdotal story of a recently hired developer (the participant) who receives a call from a customer currently unable to purchase stock using Stonehenge. The participant is then asked to attempt to purchase stock and is faced with an error.

Afterwards, the participant is asked to load Serviz and analyze the runtime information collected last year (when the system was working normally) and compare it with the information collected during the previous few minutes. By comparing the two, the participant should be able to figure out that the `QuoteServiceV1` is never invoked.

In order to have more truthful results, the assignment is divided in two parts and the participants are only handed the second part after they completed the first part. This is done because the second part of the assignment reveals the nature of the fault in the system. The second part of the assignment is related to fixing the fault. The fault is then revealed in order to let the participant continue with the assignment, in case the participant was not able to pinpoint it in the first part.

The second part of the assignment contains two questions: one of which asks the participant to pick a suitable time to deploy the repaired service into the production system, and the other tests whether the user is able to identify the periods of time when the system is not used.

The assignment was designed to be performed in two hours and it has been carefully tailored to target Pacione's maintenance activities as described in Section 5.3. How the assignments match Pacione's six activities that we consider is shown in Table 5.4.

	Question	Pacione's Task
Q1.1	Which services are involved in the buy functionality?	**A1**
Q1.3	Based on what Serviz shows you, can you identify which services depend on the OrderProcessorService and which services BusinessService depends on?	**A4**
Q2.2	Did your changes repair the system's functionality to provide correct behavior?	**A2**
Q3	Which period did you choose [to perform maintenance] and why?	**A5**
Q4.1	How are services used over a long period of time?	**A6**
Q4.2	Which services and which periods are those [when services never get used]?	**A7**

Table 5.4: Assignment questions matched to Pacione's tasks

5.3.3 Pilot

In order to find and eradicate possible problems with the tool and the experimental set-up, we performed a pilot run. This pilot run allowed us to minimize unexpected problems with our tool. For instance, it allowed us to amend a problem with the experimental set-up where the data was not being logged correctly by the services framework. It also allowed us to polish the assignment questions to make them clearer and easier to follow.

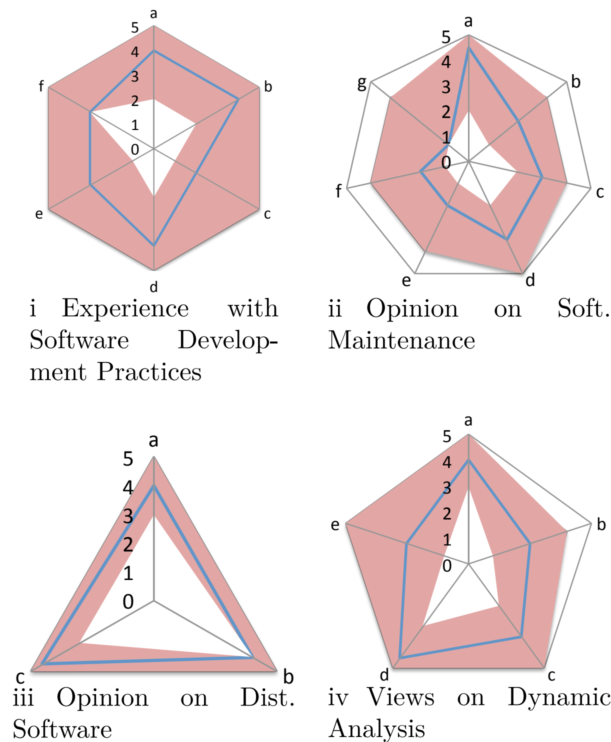
5.4 Results

Following is the data obtained from the pretest (Fig. 5.2)[5] and posttest (Figs. 5.3 and 5.5) questionnaires, along with a description of the collected data.

5.4.1 Pretest Data

In this section we analyze the data collected from the pretest questionnaires. We look at the participants' background, focusing on questions that might have had a direct impact on their experience with the assignment.

[5]The radar charts should be interpreted as follows: each axis of the chart represents one question, the minimum and maximum values of the respondents answers are marked by the colored area, the median score is indicated by the (blue) line.



i Experience with Software Development Practices

ii Opinion on Soft. Maintenance

iii Opinion on Dist. Software

iv Views on Dynamic Analysis

Figure 5.2: Pretest Data

Subject Profile. For our user study involving Serviz, we recruited eight volunteers in the computer science faculty of the Delft University of Technology. The participants have various backgrounds and the group is composed of four MSc students, two PhD students (one of which with five years of experience as a software engineer), one post-doctoral researcher and a software engineer. All participants are male with ages ranging between 23 and 32 years of age.

Experience with Software Development. We refer now to the participants' experience with software development, shown in Fig. 5.2i. Based on this data we can infer the participants have a fair amount of experience with Java development (Fig. 5.2i-a, median 4) and while most participants are familiar with web services and SOA (Fig. 5.2i-d, median 4 and 5.2i-f, median 3) not all of them had participated in web service development before (Fig. 5.2i-e, median 3, but some reported to have very little experience). From these results we can also gather that participants are fairly acquainted with Eclipse for Java development purposes (Fig. 5.2i-b, median 4). The participants reported to be less experienced with Maven (Fig. 5.2i-c, median 2). This particular observation is unlikely to have an impact on the outcome of the experiment as step-by-step instructions on how to use Maven were provided (Maven was required to recompile any changed web services). Furthermore, the participants were encouraged to ask questions during the experiment in case of uncertainty.

Opinion on Software Maintenance. As part of the pretest questionnaire, we also gathered the participants' opinion on software maintenance. From our data we observed that most participants had performed software maintenance before (Fig. 5.2ii-a, median 4.5) even though this is not something they do often (Fig. 5.2ii-b, median 2.5). Based on the participants' input, this is also a task mostly performed on their own software (Fig. 5.2ii-c, median 3). When asked in particular about maintaining distributed systems, the results reveal this is also something not often performed by the participants (Fig. 5.2ii-e, median 2). This is further made clear by the responses to whether they had used dynamic or static analysis for this task (Fig. 5.2ii-f, median 2 and Fig. 5.2ii-g, median 1). Also in the context of software maintenance, we asked the participants whether they had used software visualization tools before (Fig. 5.2ii-d, median 3.5) and based on the responses we can assume the participants are, at least, familiar with the concept of such tools.

Opinion on Distributed Software. Looking at the participants' opinion regarding distributed software, the results show that the participants see software maintenance as a difficult task (Fig. 5.2iii-a, median 4) made even more difficult by the distributed dimension (Fig. 5.2iii-b, median 4). The general consensus was also that making use of runtime information can help maintain distributed software (Fig. 5.2iii-c, median 4.5).

Opinion on Dynamic Analysis. Lastly, we gather the participants' opinions on dynamic analysis. When asked in general whether the participants used static or dynamic analysis, the tendency is vastly in favor of static analysis (Fig. 5.2iv-a, median 4 vs Fig. 5.2iv-b, median 2.5). This makes it clear that the participants are not extremely familiar with dynamic analysis techniques. Despite this fact, the participants in general have performed dynamic analysis at least once (Fig. 5.2iv-c, median 3.5) and a smaller

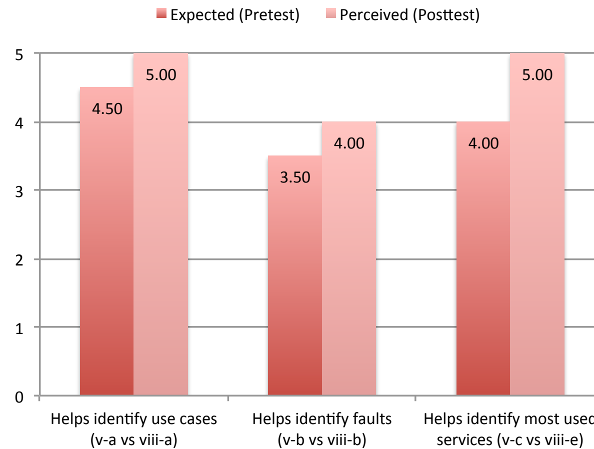


Figure 5.3: Expectation vs Perception (Median)

percentage has used dynamic analysis tools on distributed systems (Fig. 5.2iv-e, median 2.5). It is also worth noting that participants are well aware of the added value dynamic analysis provides over static analysis (Fig. 5.2iv-d, median 4.5).

5.4.2 Posttest Data

In this section we discuss the data collected after the participants performed the assignment. For some questions we also cross-check their perceived values with those expected in the pretest questionnaire in order to measure how well Serviz meets the participants' expectations.

Overall Experience. Our posttest results regarding the overall experience with Serviz have been mostly positive with small exceptions. When asked whether the assignments were too hard to do or if there was time pressure, the participants responded with a median of 2 and 1 respectively (questions vi-a and vi-b). This excludes difficulty and lack of time as potential factors which affect the final outcome of the experiment. This is by all accounts expected, as all the participants managed to finish the assignment within the allotted two hours. The participants were also quite interested in the experiment and were enthusiastic about it (questions vi-c and vi-d, with medians of 4 in both accounts) which is further supported by the fact that all participants but one left further remarks at the end of the assignment. The consensus was also that they got enough guidance for completing the assignments (question vi-e, median 4).

Identifying use cases. One of the purposes of making use of the runtime topology of a SOA system is to identify web services that together participate in providing a specific functionality. We measured how well Serviz achieves this by asking the following question before and after the assignment "Serviz helped me identify which services were involved in a specific use case" (Fig. 5.3, v-a vs viii-a).

We can now compare what the participants expected of Serviz before using the tool and how they perceived Serviz after using it. In particular, for question v-a, we see that although the participants already had quite high expectations (median 4.5), the perceived usefulness of Serviz after having used the tool rose to a median of 5. This rise

in the expected vs. perceived score means that not only were the participants confident that such a feature would likely help, they were even more convinced after having used Serviz.

Manual fault identification. Before each participant started the assignment, a fault was intentionally added to the software system under maintenance. This fault consisted of disabling the invocation from the `OrderProcessorService` to the `QuoteService`, resulting in the *purchase stock* action of the system to fail with an error displayed to the participant.

One of the tasks the participants were asked to solve during the assignment was to fix this fault. One of the goals of adding this fault was to infer whether the runtime topology of a SOA system aids in manually identifying such faults in an effective manner.

This facet is assessed through our posttest questionnaire where the participants are asked whether "Serviz helped [them] identify where the fault was in the system" (Fig. 5.3, v-b vs viii-b). This question was met with a median of 3.5 in the pretest which rose to a median of 4 after the participants performed the assignment. From this we understand that, perhaps based on the wording (Table 5.2, Tool description) which does not explicitly mention fault detection, the participants were not extremely confident Serviz would help in this task. However, after using Serviz they were convinced that Serviz does indeed help in identifying where the fault was located.

Identifying most used services. Directly linked to RQ3 is the need to identify the most used services in a SOA system. This aspect of our approach is measured in both the pretest and posttest with the question "It was clear, using Serviz, which services are used more often".

The results show that the participants were already expecting Serviz would help them identifying the most used services and this was confirmed after the assignment with a rise of the median from 4 to 5 points (Fig. 5.3, v-c vs viii-e).

Identifying dependencies between different services. In the posttest we also asked the participants whether "using Serviz [they] could easily infer the dependencies between the different services" (Fig. 5.5, viii-c). This question, somewhat related to question viii-a, is aimed at making a more direct assumption on whether Serviz can at least provide an overview of which services depend on which other services. In this specific case, having already obtained the maximum score for question viii-a with a median of 5, the participants' opinion remains consistent also with a median of 5 for question viii-c. This makes it clear that Serviz does indeed help identifying dependencies as well as identifying the use cases related to these dependencies.

We elaborated on this question by asking whether the above is also possible whilst including the time dimension (Fig. 5.5, viii-d). While the median for this particular question lowered to 4 points, it is still clear that Serviz helps when the time dimension is added to the equation. The reason for the median being lower compared to not including the time dimension might be related to the participants' frustration with the date picking controls which were, at times, buggy. This is discussed further down, in the additional remarks from the participants.

Identifying periods of time for maintenance. Another goal of using the system's

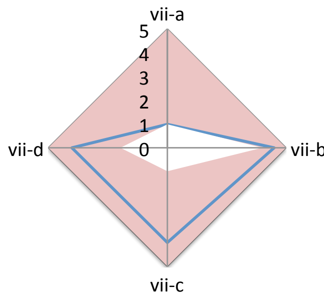


Figure 5.4: Usability Questions (Median)

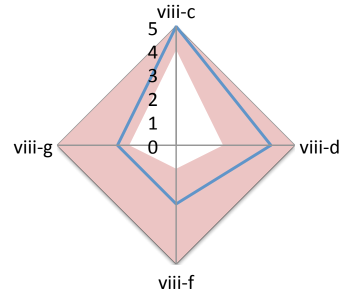


Figure 5.5: Additional Posttest Questions (Median)

runtime topology is that of finding the most suitable time for performing maintenance. By following our assignment, the participants were also expected to find such periods of time. The results related to this task were asked only in the posttest due to the more complex nature of the questions (viii-f and viii-g).

The results for these particular questions (Fig. 5.5) have been somewhat disappointing with the median for both cases lying at 2.5 points. This result might be related to either the data shown which is not enough to provide this information, or due to the tool which does not do a good enough job at displaying such data. Based on the participants' comments (further discussed in the additional remarks), we are inclined towards a shortcoming of the tool itself.

Usability. Regarding the usability of Serviz, a very disappointing factor comes to light. Our data shows that the participants did not notice that the thickness of the arrows connecting the web services was proportional to the number of calls (Fig. 5.4, vii-a, median 1). This might have had a negative impact on how the participants used the tool, as understanding how many requests are made per web service is a crucial means to an end in order to find which web services are used less often and when. In hindsight the relationship between the arrow thickness and number of invocations is not always obvious; in order to overcome this issue, the next version of Serviz will contain a legend which explicitly describes line thickness as a metric.

Despite this disappointing outlier in our data, the remaining features of Serviz seem to have been obvious and intuitive enough. The participants claimed it was clear they were supposed to input dates before any data was shown (Fig. 5.4, vii-b, median 4.5), it was clear they could scroll the timeline to access different periods of time (Fig. 5.4, vii-c, median 4) and lastly that it was clear the arrows represent dynamic dependencies between services (Fig. 5.4, vii-d, median 4).

Additional remarks. From the open question at the end of the posttest we obtained more detailed feedback from the participants on what can be improved for future work. Several participants were perplexed by the purpose of the timeline, claiming it had no practical usefulness. Some participants suggested the timeline should include a histogram so it becomes easier to visualize at a glance when is the system not being used.

This timeline was initially envisioned to visualize all the static configuration changes

and it allows maintainers to load each of these changes. This feature was, however, not targeted by the assignment as the "load latest static data" button is more intuitive for our maintenance scenarios.

Some participants also mentioned that the date picking process is sometimes buggy (e.g., when the participants input a date manually rather than using the date pickers) and moving the services around in the graph can also be quite slow. This is mostly due to the prototype nature of the tool and is something we will improve in further versions. We also believe that while these minor bugs might have somewhat frustrated the participants, they have clearly not had a direct impact on the results which remained generally positive.

Another common remark was the fact that Serviz can only be used when the browser is maximized. Any other window configuration of the browser forces the date pickers to hide which in turn makes it impossible to use the tool.

Finally, the last remark has to do with the positioning of the web services in the graph. By default, and whenever the dates are changed, the web services revert to their initial position, where they overlap on top of each other. This was reported by the participants as being very cumbersome to manage. Whenever the dates are changed, the participants were forced to rearrange the services to visualize the runtime topology of the system in that particular interval. Looking for particular periods of time when specific services do not show is certainly a task that requires copious amounts of patience and time. This is something we also want to fix in future versions by providing a usage histogram per service.

5.5 Discussion

5.5.1 Revisiting the Research Questions

Our first research question (**RQ3.1**) is aimed at determining whether "*the runtime topology helps in identifying services involved in a specific use case?*" Based on our experiment results and referring to Fig. 5.3 (v-a vs viii-a), the participants replied with a median of 5 points when asked this question directly after using Serviz. This allows us to infer that indeed, the runtime topology of a system (more specifically, the one provided by Serviz) helps in addressing RQ3.1.

As for the second research question (**RQ3.2**), we wanted to find out if "*the runtime topology helps in debugging a SOA-based system?*" While the term *debugging* can describe a whole range of techniques and approaches aimed at finding and removing faults from a software system, we were interested in investigating whether our approach would help with this task on a SOA-based system. Our approach has indeed helped the participants, as demonstrated by the median of 4 points in Fig. 5.3 (v-b vs viii-b). In fact, all the participants managed to successfully find and fix the intentional fault of the system.

The third research question we presented (**RQ3.3**) asks whether "*the runtime topology augmented with the time-dimension is suited to identify services that are used more often (e.g., to optimize them)?*" The answer to this question is also positive, supported by the results shown in Fig. 5.3 (v-c vs viii-e). The participants have given Serviz a median score

of 5 for this particular feature, which we can interpret as a positive outcome of Serviz in addressing this challenge.

Lastly, the fourth reseach question (**RQ3.4**) tries to identify if *the runtime topology helps to identify periods in time where the usage of a particular service is low, e.g., for evolving it with minimal disturbance to its users?* This question was, unfortunately, met with disappointing results. The participants have, for these two posttest questions, replied with a median of 2.5 points as shown in Fig. 5.5 (viii-f and viii-g). These results are, in our opinion, not related to the approach but to how we implemented the visualization of this particular data. Judging by the further remarks left by the participants, we understood that using our implementation of this particular feature is not trivial and requires a significant amount of time. This is something we will mitigate in further versions of Serviz by providing a histogram to better visualize usage peaks.

5.5.2 Threats to validity

In this section we present a discussion of how the results of our experiment might be challenged. We discuss internal validity, namely whether the participants might have been directly affected by factors we did not consider, and external validity, meaning whether our results are generalizable.

5.5.2.1 Internal Validity

The participants in this experiment were being told to use Serviz. It remains to be known whether such a tool fares better than other means of system analysis. While the tool seems to have, indeed, helped the participants find and fix the problem, it is unknown whether they could have achieved the same without the tool. It should also be said that Serviz simply aims at making it easier to achieve the solution and not to simply provide the solution.

Another threat to the internal validity of our study concerns the participant group chosen. The participants have been told to be impartial when evaluating the tool but, despite this, they might have felt emotionally biased to give favorable results.

Also from reading the questions in the pretest questionnaire, the participants may have deduced the goal of the study and this may have, therefore, influenced their behavior when using Serviz.

5.5.2.2 External Validity

Our main concern with external validity has to do with the performance impact of our approach. We are particularly concerned with the performance overhead added by the data collection step. However, the data collection is supported by a robust framework used by a company with a large web services infrastructure (eBay).

The storage requirement of our approach is also quite high. In a system with a large traffic of web service requests, a large amount of storage space is required in order to maintain the system's state over time. This threat can be mitigated by using compression

techniques, e.g., as applied in the Compact Trace Format (CTF) [Hamou-Lhadj and Lethbridge, 2012].

The applicability of our results to larger systems is also something we tried to mitigate by using Spicy Stonehenge. Spicy Stonehenge, despite its small size, contains all the ingredients of an industrial system, including complex interactions between services and a well-specified domain.

Another issue is that the group of participants was mostly composed of students. Despite this, two of the students have working experience in industry. Another factor is that all participants in the experiment should be considered *software immigrants* [Elliott Sim and Holt, 1998] and as such, the findings that we report upon are based on developers trying to find their way in a previously unknown software system and do not reflect situations were developers are already familiar with the system. We acknowledge that a follow-up study should also investigate the usefulness of Serviz in situations where the developers are already familiar with the domain.

Lastly, the tasks chosen for the assignment might not be realistic. We tried to mitigate this by looking at the nine principle maintenance activities identified by Pacione et al. [Pacione et al., 2004] and tailoring the assignment tasks to cover six of these activities.

5.6 Related work

In general, the increased maintenance complexity of SOA-based systems has been acknowledged and emphasized by Lewis and Smith [Lewis and Smith, 2008], requiring for instance, impact analysis for an unknown set of users, or increased number of externally accessed services to be considered in maintenance. These are asking for a readjustment of current maintenance practice for SOA-based systems at large.

For this chapter, we started by analyzing the survey of Cornelissen et al. [Cornelissen et al., 2009] on program comprehension through dynamic analysis, which, among others, lists the work of De Pauw et al. [De Pauw et al., 2005]. They describe a web services navigator for generating service topologies, and focus on detecting incorrect implementations of business rules and ”excessively chatty” communications. Our approach is different w.r.t. two significant improvements to the service topology: it allows to identify the method of an invoked service plus its invoking client, and it includes the time dimension providing a historic view on the topology.

White et al. [White et al., 2013] present a dynamic analysis approach to aid the maintenance of SOA-based composite applications where they propose using a feature sequence viewer to recover sequence diagrams from such systems. This approach, however, does not seem to clearly provide a good basis for understanding the topology of a running service-oriented system and rather focuses on mapping features to software artifacts.

In other work, White et al. [White et al., 2012] investigated the information of developers during the maintenance of SOA-based systems. They established that the first question a maintainer must ask is “how does the software work now?” The authors also motivated that maintainers require immediate additional assistance in understanding an

application's data types.

Finally, we investigated citations to the aforementioned papers, in particular the systematic survey by Cornelissen et al. [Cornelissen et al., 2009] in order to find more recent additions to the body of knowledge. Unfortunately, this search has yielded no extra relevant related work.

5.7 Conclusion

In this chapter, we investigate how the topology of a running SOA-based system can help in its maintenance. More specifically, our contributions are:

- The runtime topology augmented with the time dimension.
- Serviz, an open-source implementation of this approach.
- A user-study with 8 participants evaluating the effectiveness of such an approach.

We now summarize how our approach and the tool address our original research questions formulated in Section 5.1:

RQ3.1 *Does the runtime topology help in identifying services involved in a specific use case?* Our experiment participants agreed that by providing them with a runtime view of what the system looks like at any point in time, they can easily discover which services are involved in providing a specific functionality.

RQ3.2 *Does the runtime topology help in debugging a SOA-based system?* Similarly, participants have also indicated that they strongly agree that by comparing normal circumstances of a system to periods when a fault is occurring, the runtime topology aids them in debugging.

RQ3.3 *Does the runtime topology augmented with the time-dimension help to identify services that are used more often (e.g., to optimize them)?* Our user study indicates that participants also found that the runtime topology augmented with an invocation count helps them to quickly identify the most used services in a specific time interval.

RQ3.4 *Does the runtime topology help to identify periods in time when the usage of a particular service is low, e.g., for evolving it with minimal disturbance to its users?* Our experiment participants were not convinced Serviz helps them with this task, which we mainly attribute to our own implementation of the runtime topology falling short. In particular, this task involves manually scrolling the data hour by hour, while the participants would like to see this task automated.

5.7.1 Future work

As future work we propose to bring the dimension of users to the runtime topology. This way we can determine which users are using which services the most. This added information makes it even easier to filter the data provided by the runtime topology and it makes it also easier to find periods of time which minimize perceived downtime.

We also want to explore how the runtime topology can help us understand service versioning. In that line of research we aim to make statistical assertions (based on histor-

ical usage data) about whether a certain version of a service has become dead code and should be undeployed.

Another goal is to keep on developing Serviz, make it more user-friendly and improve its visualization. Subsequently, we aim to perform a full-fledged controlled experiment [Cornelissen et al., 2011] aimed at measuring the actual time-gain developers have when using the runtime topology during maintenance.

CHAPTER 6

SOA: Users and Versions in Multi-Tenant Systems

Multi-tenant systems represent a class of software-as-a-service (SaaS) applications in which several groups of users, i.e. the tenants, share the same resources. This resource sharing results in multiple business organizations using the same base application, yet, requiring specific adaptations or extensions for their specific business models. Each configuration must be tended to during evolution of a multi-tenant system, because the existing application is mended, or because new tenants request additional features. In order to facilitate the understanding of multi-tenant systems, we propose to use a runtime topology augmented with user and version information, to help understand usage patterns exhibited by tenants of the different components in the system. We introduce Serviz*, our implementation of the augmented runtime topology, and evaluate it through a field user study to see to which extent Serviz aids in the analysis and understanding of a multi-tenant system.*[1]

6.1 Introduction

Multi-tenant systems represent a class of software-as-a-service (SaaS) applications in which several groups of users, i.e. the tenants, share the same (software) resources [Bezemer and Zaidman, 2010]. These multi-tenant systems allow to make full use of the economy of scale, as several SaaS customers access the same application at the same time, while the instances are configured according to the diverse requirements of the various tenants [Kwok et al., 2008]. Multi-tenant systems are characterized by high demands on configurability and evolvability, which go well beyond multi-instance and multi-user applications, or software product lines [Clements and Northrop, 2001]. These demands are

[1] This chapter contains our work together with Andy Zaidman and Hans-Gerhard Gross, published in the proceedings of the 2013 International Workshop on Principles of Software Evolution (IWPSE 2013) [Espinha et al., 2013].

instigated by the multiple tenants, or business organizations, that all use the same base application, yet, require specific adaptations or extensions in order to suit their specific business models.

Since flexibility is key to effective multi-tenant applications, their deployment should go along with inherently accommodative architectures, such as service oriented architectures (SOA) [Wang et al., 2008]. For example, SOA permit to isolate tenant-specific business needs in a separate version of a service, which raises the challenge of managing versions of services. This is evidenced by the work of Fang et al. [Fang et al., 2007] who describe an approach to manage different service versions in a web service directory.

It has long been recognized, that software systems must evolve in order to remain successful [Lehman and Belady, 1985]. This is no different in multi-tenant software systems, and we hypothesize that evolution might even be more important. This is due to the large number of stakeholders involved, each demanding new features to be added to (their version of) the multi-tenant software system. This brings about the challenge of having to understand the impact of changes on different versions of the code (deployed as different versions of services) that might be specific to a single tenant or a group of tenants.

Even though SOA provides the flexibility for realizing one of the different types of multi-tenant software systems as presented by Kabbedijk et al. [2014b], we know from Gold et al. that the promise of easier software maintenance is not completely met [Gold et al., 2004], because even though conducting the actual evolution in terms of exchanging services is possibly easier, the understanding of their interactions and usage is still an issue [Gold et al., 2004]. This comes from understanding a monolithic application versus understanding a distributed system composed of many entities (services).

With the increased complexity of multi-tenant software systems [Bezemer and Zaidman, 2010], their understanding is likely to become more demanding. Corbi claims that up to 60% of the maintenance effort lies in understanding the system [Corbi, 1989]. Strangely enough then, a large-scale survey of scientific literature in the area of program comprehension using dynamic analysis from 2009 did not reveal any major advances in the area of understanding SOA based systems using dynamic analysis. "This seems strange, as dynamically orchestrated compositions of services would benefit from dynamic analysis for understanding them [Cornelissen et al., 2009]", because the composition of the entire SOA-based system only happens at runtime and its configuration can change during operation [Canfora and Di Penta, 2007; White et al., 2013], leading to highly dynamic systems. It comes as no surprise then, that Kajko-Mattsson et al. identified the understanding of the SOA infrastructure as one of the most important challenges when evolving SOA-based software systems [Kajko-Mattsson et al., 2008].

In Chapter 5 ([Espinha et al., 2012c]) we presented the runtime topology as a means to aid the understanding and maintenance of service based systems. Our preliminary results show that the runtime topology is indeed a good means to understand highly dynamic SOA-based software systems. In this chapter we extend the runtime topology to also incorporate user and version information, since we believe that these help to understand the role of multi-tenancy in SOA-based software systems. They show how different tenants

use the system, and how several versions of similar services are customized for particular business needs in different ways.

This leads us to our **main research question**: *Does the combination of user, service-version and timing information projected on a runtime topology help in the understanding of SOA-based multi-tenant software systems?*

In order to steer our research, we will investigate these subsidiary research questions:

RQ4.1 Does the user information added to the runtime topology help in the understanding of SOA-based systems?

RQ4.2 Does the service version information added to the runtime topology help in the understanding of SOA-based systems?

RQ4.3 Do usage graphs help to understand SOA-based systems?

In order to evaluate the approach, we set up a user study that involves our research prototype *Serviz* and a case study system called Spicy Stonehenge (Chapter 4 [Espinha et al., 2012a]), a stock-market simulator. The user study is set up as a contextual interview and it involves four software engineering professionals.

The remainder of this chapter is structured as follows: Sections 6.2 and 6.3 describe our approach (the data requirements and how we collect the data) and the tool implementing it. Section 6.4 outlines the experimental setup, and Section 6.5 presents its results. Section 6.6 discusses the results and potential threats to validity. The chapter is rounded off with related work and conclusions in Sections 6.7 and 6.8.

6.2 Approach

Central to our understanding approach stands the runtime topology of a service-oriented software system, representing the configuration of a set of services that are deployed within an environment. Based on previous research (Chapter 5, [Espinha et al., 2012c]), we found to be essential to include information which allows us to link service invocations to request traces and associate each request with its timestamp. This information alone allows us to infer the runtime topology of a running service-based system. In addition, and due to our focus on multi-tenant systems, usage scenarios require different versions of the same service to co-exist in the same platform. These versions can then be used by different tenants who, regardless of the client used to invoke the service, may cause different usage patterns.

In order to satisfy this requirement, we also included information regarding which user caused a particular invocation as well as which version of a service was invoked. In the following subsection we present the data requirements inherent to creating the initial runtime topology of a service-oriented system, plus the additional data required in order to be able to add the user and service version dimensions.

6.2.1 Data Requirements

The runtime topology on its own has specific data requirements which we already identified earlier, and whose extraction in our platform of choice (Turmeric SOA[2]) is briefly described below in subsection 6.2.2. These requirements represent the essential set of data required for plotting a runtime topology diagram representing the service invocations. The collected data are:

- **Request ID** - A unique identifier per invocation trace. It allows us to link pairs of services as belonging to the same trace.
- **Timestamp** - The current timestamp at the time of the request, as seen by the database server. This is crucial for analyzing exactly when the web services were busiest.
- **Consumer name and method** - The name of the caller service (i.e. client) and the respective method which caused this request pair.
- **Service name and method** - The name of the callee service in this request pair and the respective web method being called.

More in-depth detail on the reasoning behind the usage of this data can be found in Chapter 5, Sections 5.2.1 and 5.2.2 ([Espinha et al., 2012c]).

In our current research, we analyze the usefulness of the runtime topology for multi-tenant systems. This type of systems, due to its inherent usage patterns (i.e. multiple users simultaneously use multiple service versions), adds two important dimensions which need to be taken into consideration upon performing maintenance. For this reason, our approach requires additional data to be added per service request. Therefore, our approach now also includes:

- **Consumer and service versions** - For each pair of calls, the consumer and service versions are stored.
- **Username** - The username of the user who originally caused the request to happen.

Having identified all the data requirements for our particular goal, in the next section we propose an approach to collect the data from a running system.

6.2.2 Data Extraction

The data extraction step required to enable the runtime topology is highly dependent on the platform used. Some service platforms may already provide parts of the required data whereas others may require deeper changes in order to obtain such data. Ultimately, however, this is a completely automated step after it has been enabled for a specific platform. This means the data is automatically pulled from the running system rather than requiring manual intervention.

[2]Turmeric SOA —
https://www.ebayopensource.org/index.php/Turmeric/

For our implementation we chose the Turmeric SOA platform, the open source version of eBay's services platform, as it already provides some of the information we described in the previous subsection. This data is not provided by default in Turmeric SOA and in order to collect it, we make use of one of the framework's features. This allows us to intercept each incoming request by means of a Java class which can then access and manipulate information regarding the request. In order to retrieve information on invocation pairs, we had to follow another solution: we appended information to the request string, thus making sure that information on the caller got to the callee. Since this data is only available during the web service's invocation lifetime, the data must be stored persistently. For this reason, the request handler stores the data in a MongoDB database in order to keep a historical track of the system's usage.

In practice, we are able to collect all the required data by handling the incoming requests from the point of view of each web service. This event provides us with all the information mentioned in Subsection 6.2.1.

6.3 Serviz

Serviz[3] is an open-source visualization tool which allows software engineers to visualize periods of high and low usage in a service-based system. Additionally, Serviz allows this information to be filtered for a set of users, per service and version, and according to a specific time-frame.

[3]https://github.com/etiago/serviz

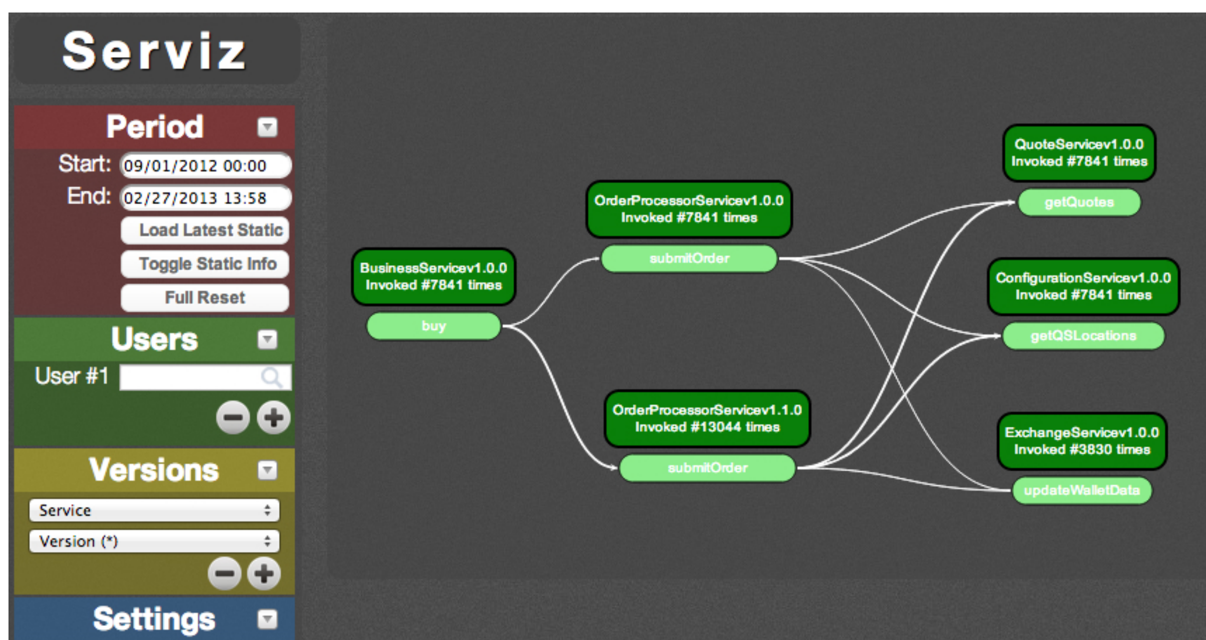


Figure 6.1: Screenshot of Serviz

In Chapter 5 ([Espinha et al., 2012c]) we performed a pre-test post-test experiment on whether the use of a runtime topology of a service-based system would help in understanding such a system. From this experiment we gathered that while the runtime topology alone provides a step forward in understanding service-based systems, it can be enhanced by providing more runtime information to system maintainers.

With this in mind, we enhanced Serviz with new features. They are: the capability to filter the information displayed based on which users were using the system at the time, and similarly, the capability to filter such information based on a specific service name and version. Simultaneously, we added histograms to more easily visualize service usage, per service, over time. Having these added filtering capabilities provides the maintainers with more detailed insight into how the system has behaved and is behaving. A screenshot of Serviz can be seen in Figure 6.1. On the left hand side of the screenshot, we see the filtering options for the collected data: the time-period under consideration, the users and also the versions of interest. The right hand side of the screenshot shows the different services and the service methods called per service (below the service name and invocation frequency). Of interest here is that the `OrderServiceProcessor` has two versions, which each play an active role in the functioning of this service-based system.

We now discuss some of the main features of Serviz.

6.3.1 User Filtering

It is important for a service provider managing a service-based system to be able to filter the runtime usage based on specific users. For instance, after a system maintainer has inferred that a service is only used by a specific user (e.g. in the case when a specific version was created to cater the specific needs of a particular user), the maintainer can then pinpoint high and low peak usages for that specific user by filtering this data.

6.3.2 Service/Version Filtering

By filtering the runtime topology with the service and version axes, maintainers are able to find services which are direct dependences to a particular service version. In the same way, by selecting extended periods of time, maintainers are also able to determine when a particular version of service can be exchanged for a newer version with minimal interruption.

6.3.3 Combined Filtering

Our runtime topology also allows system maintainers to perform combined filtering using both the user and version filtering axes. This way maintainers can find out information such as “which service versions have never been used by a particular user” or through knowing that a particular version is only used by a specific user, maintainers can find out periods suitable for maintenance.

6.3.4 Histograms

Another feature of Serviz are the time-based histograms with service usage over time (see Figure 6.2). The histograms are created per service and also take into consideration the filtering defined by the maintainer. The histograms are to be combined with the filtering features, so that maintainers can use them not only to determine periods of high and low usage for the whole system, but also on a per user and per service basis. This allows for a very fine grained view of which users use which services the most and at which times.

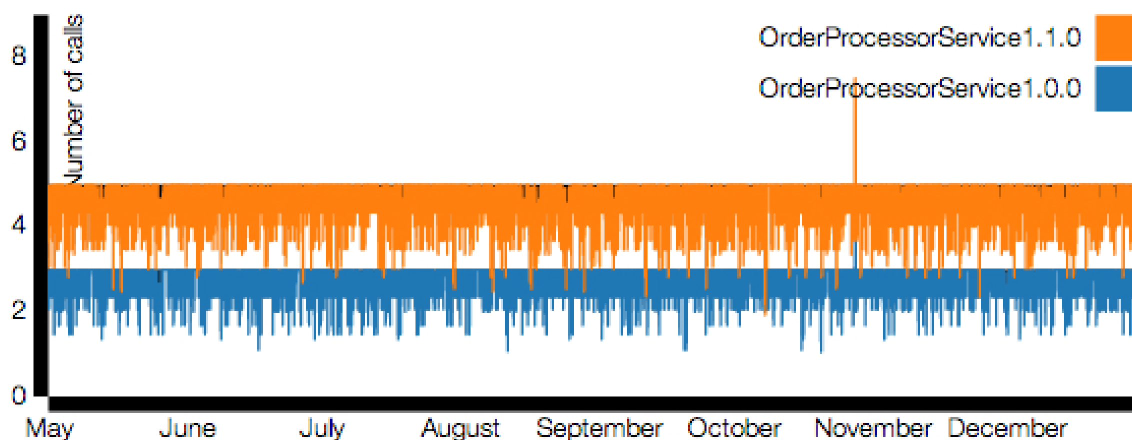


Figure 6.2: Histogram showing the usage (requests per minute) of two versions of the same service.

6.4 Experimental Setup

In order to evaluate our implementation of a runtime topology augmented with time, user and version information, we performed a field user study with a total of four software engineers from two IT companies in the Netherlands, Adyen B.V. and Exact N.V. The user study was done in two sessions, one session at each company.

This augmented runtime topology is especially aimed at multi-tenant systems, where different users use different parts of the system in different periods of time. Because of this added complexity, we involved software engineers from IT companies which are dealing with multi-tenant scenarios as part of their own software systems. This user study was organized as a contextual interview [Holtzblatt and Jones, 1995; Matthijssen et al., 2010; Zaidman et al., 2013].

Step 1: Demo of Serviz — We started the session with a short demo of Serviz in which we showed the developers all features of Serviz.

Step 2: Free Exploration — In this step we asked the developers to freely explore Spicy Stonehenge, which we had pre-installed and ready to be used with Serviz. We

gave them the goal of getting a good understanding of the implementation of the major functionality in Stonehenge *and* how the system is used based on the accompanying usage data. We told them that we would discuss the implementation details of this functionality later on during the session.

This phase of free exploration took in both cases less than one hour and was done in pairs, as to stimulate communication between the participants during exploration, thereby simulating the think-aloud protocol [Ericsson and Simon, 1998]. We recorded the conversation between the software engineers and provided assistance whenever the information shown was not clear enough. As an example of this, Serviz was loaded with data solely for the year 2012, so to prevent the participants from losing time, this information was provided whenever the participants tried to input dates in 2013. Similarly, whenever the directional arrows linking the service were not clear enough, this information was provided to the participants.

Step 3: Questionnaire — When the developers were satisfied with their reconnaissance of Spicy Stonehenge, we presented them our questionnaire (shown in Table 6.1) which had to be answered individually. This questionnaire is not so much meant to gather quantitative data, but rather serves as a means to steer the discussion in the next step, which is aimed at gathering qualitative data.

Step 4: Contextual Interview — While we already gained quite a lot of information during the free exploration phase, we intensified the interview once the developers felt they were comfortable with Serviz and the case study system Spicy Stonehenge. In particular, we used a contextual interview [Holtzblatt and Jones, 1995]. This contextual interview already started during the second step, where we observed how the developers explored the system. In particular, we took note of which questions they were asking and how they were using Serviz to answer these questions. Subsequently, we continued the interview and we aimed to further explore the possibilities of Serviz and identify circumstances in which Serviz can be of benefit. In order to steer this conversation, we used the questionnaire from Table 6.1 as a basis.

Again, this discussion was recorded. The interview took on average one hour and a half and a time limit was not imposed on the participants. This excludes any possibility for time-related pressure to finish the experiment.

6.4.1 Case Study System

The subject system that we pre-loaded into Serviz is Spicy Stonehenge, a simulation of the stock market (Chapter 4 [Espinha et al., 2012a])[4]. Spicy Stonehenge is composed of 6 services and the user data that we preloaded concerns 3 users. An overview of Stonehenge can be seen in Figure 6.3. The figure depicts two services with multiple versions (BusinessService and ConfigurationService), which are in fact fully independent implementations of each service. This particular setup was chosen due to the resemblance it bears to multi-tenant systems. In such systems, user-specific configuration is created

[4]https://github.com/etiago/spicy-stonehenge

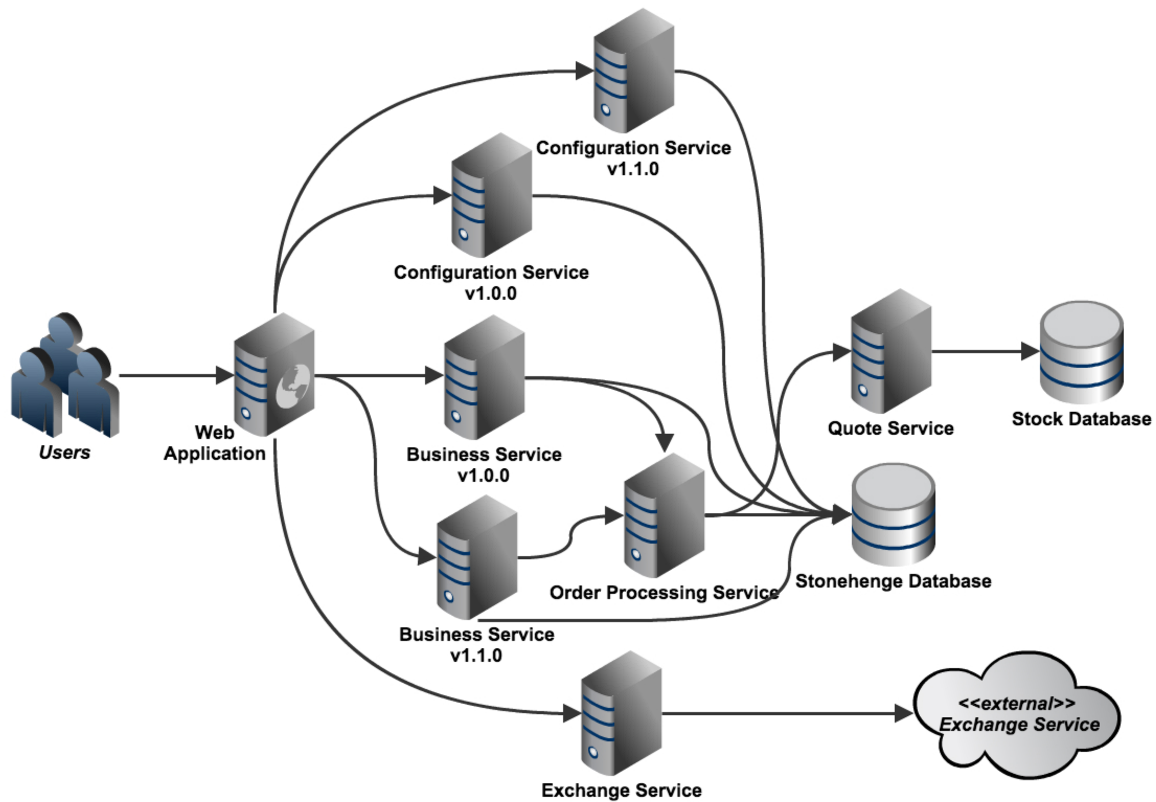


Figure 6.3: Spicy Stonehenge

by branching versions of services containing the business logic required by a specific user. Important to note is that none of the participants was familiar with (Spicy) Stonehenge.

6.4.2 Questionnaire

The origin of the questionnaire can be traced back to the work of Sillito et al. [Sillito et al., 2008], in which the authors present a set of 44 typical questions that software engineers have when performing maintenance on a software system. However, a number of questions from this work was either not relevant for the context of service-based systems or had to be adapted to this context. For example, the original question "What is the behavior that these types provide together and how is it distributed over the types?" is now rephrased as "What is the behavior these services provide together and how is it distributed over the services?" (SE11). Despite these changes, we tried to preserve the main phrasing of these questions as much as possible. The questions should then be interpreted as whether Serviz *helps* in achieving each of those goals (i.e. finding a particular behavior or pattern). From the original 44 questions from Sillito et al. we kept 19 questions that are particularly relevant for Serviz, e.g., questions that dealt with static analysis were removed. Additionally, we introduced 20 new questions, which we feel are important questions when trying to understand a service-based system. All questions are

listed in Table 6.1 and they are organized as follows: G1 through G4 are general questions about the usability and usefulness of Serviz, SE1 through SE19 are questions extracted from Sillito's work, U1 through U4 are questions related to the user filtering feature, S1 through S4 concern the service filtering, V1 through V4 relate to the version filtering and lastly, C1 through C3 identify the user experience with combined filtering of the previous features.

The participants were asked to fill in the questionnaire using a 5-point Likert scale ranging from *strongly disagree* to *strongly agree*. At the end of the questionnaire there were 2 open questions gauging for the most-liked features of Serviz and asking for any additional comments or suggestions.

6.4.3 Participants

For this study we approached four software engineers from two distinct software companies. All four participants have extensive experience with web service development (> 2 years). However, the technologies used and the context in which they are used are quite different. This is reflected in the results of the experiment where clear disagreements exist and are explored later in the chapter.

Adyen, a SMB, makes extensive use of web services in their business practice and the majority of the technologies used are developed in-house. This leads to a deep knowledge of every single component of their own system. It also means that every developer has a very good understanding of how the system interacts to bring together functionality.

As for Exact, while this is by comparison a larger company with around 3000 employees, service orientation is being actively pursued. There is also a larger reliance on proprietary third-party technologies which can sometimes obscure the runtime details of a software system.

6.5 Results

The quantitative results obtained through the four participants of the questionnaire are presented in Table 6.1. The subsequent interview was done in pairs of two participants and discussed each of the questions in the questionnaire. Our discussion below will highlight those questions that sparked the most interesting discussions.

An important note concerning the results table is that the results P1 and P2 represent the participants from Adyen whereas P3 and P4 represent those from Exact.

6.5.1 General Questions

The general questions assess the usability of Serviz, whether it would save the maintainers time while maintaining the system, and also whether the runtime topology helps with understanding service-oriented systems during maintenance.

These first results provide a somewhat mixed image of how useful Serviz is perceived to be. There is a clear divide amongst the companies, with the participants from Adyen

Table 6.1: Questionnaire

		Adyen		Exact	
		P1	P2	P3	P4
G1	The tool was easy to use	3	3	4	3
G2	A tool like Serviz will save me time	2	2	4	4
G3	Visualizing the runtime topology makes maintenance tasks on service oriented systems easier	3	5	5	5
G4	A tool like Serviz will help me understand and maintain a service oriented system	2	2	5	4
SE1	Which service represents this UI element or action.	3	3	4	4
SE2	Where is there any code involved in the implementation of a particular behavior	4	4	3	3
SE3	Where is there an exemplar for a certain behavior?	3	4	5	4
SE4	Identifying system services based on their names.	4	4	5	4
SE5	What are the interfaces of a specific service?	4	2	4	4
SE6	Where is a method called?	4	2	4	2
SE7	When during the execution is a method called?	2	1	2	2
SE8	Where are instances of a service created?	3	1	4	2
SE9	How are services composed/assembled to bring together functionality?	3	3	5	4
SE10	How is a feature implemented?	2	2	4	3
SE11	What is the behavior these services provide together and how is it distributed over the services?	2	1	4	4
SE12	How is control getting from here to there?	2	2	4	3
SE13	Why isn't control reaching this point in code?	2	2	3	2
SE14	Which execution path is being taken in this case?	4	1	4	3
SE15	How does the system behavior vary over these services?	2	1	4	3
SE16	What is the mapping between these services?	4	2	5	4
SE17	To move a certain feature, what else needs to be moved?	2	4	5	2
SE18	What will be the direct impact of a change?	3	4	5	3
SE19	What will be the total impact of a change?	3	4	4	3
U1	It was clear I could filter the runtime topology based on usernames.	4	5	5	4
U2	The username filtering adds value to the runtime topology.	5	5	5	4
U3	By using the username filtering, I could find out which users use the system the most in a specific time interval.	2	3	4	4
U4	By using the username filtering, I could find out which users have used the system the most.	2	3	4	4
S1	It was clear I could filter the runtime topology based on specific services.	1	4	4	4
S2	The service filtering adds value to the runtime topology.	1	3	5	4
S3	By using the service filtering, I could find out which services are used the most.	1	3	2	3
S4	Service filtering helps me find out which services depend on each other.	1	3	4	3
V1	It was clear I could filter the runtime topology by picking specific versions of services.	4	4	5	4
V2	The version filtering adds value to the runtime topology.	3	4	5	4
V3	By using the version filtering, I could find out which services are good candidates for dead code (never used).	1	4	4	4
V4	Version filtering will help me pinpoint which service version to perform maintenance on.	1	4	5	3
C1	With version and user filtering I could find which user has used a specific version the most.	2	3	4	4
C2	Version and user filtering allows me to find which versions a user never used.	4	3	3	4
C3	With version and user filtering I can find periods of low usage which are more suitable for software maintenance.	2	4	5	4

generally giving lower marks. In particular, when asked whether Serviz would save time and whether it would help them understand a service oriented system during maintenance, they seemed more reluctant by scoring these questions with a 2 (*disagree*). The two participants from Exact, on the other hand, gave a score of respectively 4 (*agree*) and 5 (*strongly agree*). This divide can be explained by the context in which both companies are operating, something which is also supported by the participants' comments during the interview. More specifically, P1 claimed "I think this is useful to identify the topology of your system, but what happens if you already know the topology, in the beginning you want to get to know the system then it can be useful". In this case, participants P1 and

P2 claimed during the interview that because they are well acquainted with their system's topology, they did not see the usefulness of Serviz.

However, they recognized the value for immigrants to the software system by stating "I liked it as a tool you can use at the beginning". This is in line with the observations of Sim et al. [Sim and Holt, 1998] who state that "software immigrants" need to undergo a naturalization process during which they get to know the system. This notion of "software immigrants" deserves particular attention in the case of large multi-tenant software systems. Here, interactions are determined at runtime and the services interact in various ways to satisfy the requirements of different users. Even well-seasoned software engineers who are well acquainted with the system may feel as if they were *immigrants* to a particular usage workflow of a particular user.

Additionally, from the interview we identified another reason why for participants P1 and P2 Serviz may have limited usefulness. As the participants stated, at Adyen, the user-specific variation is not done through service versioning. Instead, their system resorts to inline configurability in the source code. Because Serviz acts at the service level, it does not provide source code granularity, and it might be difficult for participants P1 and P2 to imagine how Serviz would address their scenario.

As for the Exact case, both participants P3 and P4 agreed that the tool was easy to use (4 points and 3 points), that it will save them time (4 points for both), that it makes maintenance tasks easier (5 points for both) and that it will help them understand and maintain a service oriented system (5 points and 4 points). When asked how it helps them during their understanding and/or maintenance, P3 and P4 mentioned that they do not have any tooling available which shows them the information that Serviz provides.

In contrast, in the post-questionnaire discussion, Adyen's participants revealed a prototype tool developed in-house which bears similarities to the histogram feature of Serviz. This tool was driven by their own internal needs and in the specific case of Adyen, the histogram works in realtime. Simultaneously, it also lists the users which have been using the system recently, ranked by number of requests. This demonstrates that even in a slightly different setup like Adyen's, the need exists to know how the users of a service-based system are actually using the system.

6.5.2 Generic Software Engineering Questions

In this section we analyze the results regarding the general software maintenance questions that are based on the questions found by Sillito et al. [Sillito et al., 2008]. Here the answers are also mixed and require deeper analysis.

The first major disparity in the scores begins with question SE5 (What are the interfaces of a specific service?), where participants P1 and P2 scored the question with 4 and 2 points respectively. These results are intriguing at a first sight, but through inquiring the participants about these scores, we realized participant P2 had different expectations from the "service interface" nomenclature. Namely, participant P2 did not consider the method names as the interface of a service as just the method name does not include, for example, the method signature.

Question SE7 has generally low scores with all participants scoring it with 2 points except for P2 which scored 1 point. Upon further investigation with the participants, the general opinion was that more detail was expected regarding the "when during the execution is a method called". In fact, the participants claimed they were expecting to know the exact line of code to be able to infer the "when" rather than just knowing the method in which another method is called. This appears to simply have been a mismatch between expectation and reality.

Question SE8 presents a similar scenario with participants P2 and P4 expecting more detail about the *where* the instances of a service are created. Another remark was the existence of a misunderstanding. Participant P2 said he did not know where the service instance was created, as with some implementations this happens within the service framework. Here again, a misunderstanding happened with the definition of the "creation of a service". However, the participant did agree that it was possible to identify the place where the preparation of a service call is created.

Questions SE9, SE10 and SE11 focus on service composition and how the services come together to provide functionality. In all of these questions the major divide is between the two different companies, with Adyen's participants scoring these questions much lower than Exact's counterparts. This can again be accounted for with the nature of the systems the participants are used to. Because P1 and P2 (from Adyen) deal with much more stable software systems, it can be more difficult to see the added value of using runtime information to figure out how a feature is implemented (SE10), or how a behavior is distributed over services (SE11).

On the other hand, because Exact has a larger software system with more variation, it seems easier for the participants to understand the value of finding out exactly how a certain feature is implemented in terms of services, or how a behavior is distributed across the different services. Since service-orientation is also something not yet fully in practice at Exact, it might also be the case that the participants do not have as much bias from the systems they are used to.

6.5.3 User Filtering

When asked whether it was clear that it was possible to filter the runtime topology based on usernames (U1), the results were on the overall very positive (scores of 4 or more). Also when asked whether this feature adds value to the runtime topology (U2), the participants agreed, with only one participant giving 4 points and all the rest scoring it 5 points.

Some disagreement seems to exist on whether this feature helps in finding out which users use the system the most during a specific time interval (U3). Participants P1 and P2 scored it with 2 and 3 points respectively, whereas participants P3 and P4 scored this question with 4 points. The same question was asked, but then with a broader scope regarding time rather than focusing on a specific interval (U4). The results for this question were identical.

The lower scores from participants P1 and P2 can be explained also through weighing

in the interview transcripts. In the interview, the participants claimed they would have liked the possibility to have this done automatically rather than having to find the users manually. Namely, the developers considered the task of having to manually input each username one by one to be cumbersome and time-consuming. This is a valuable insight, which we intend to implement in a future release of Serviz.

6.5.4 Service Filtering

For service filtering, there was a disagreement within Adyen's participant group. All the questions received a 1 point mark from P1 where as the remaining participants are generally more positive about the service filtering features. Namely when asked about how clear the feature's availability was (S1), all participants scored it with 4 points. As to whether it adds value (S2), the results are all positive in the overall with the participants P2, P3 and P4 ranking it 3 points, 5 points and 4 points respectively.

This feature was also combined with the histograms which the participants seemed to have had trouble with. Two participants (P1 and P3) ranked this feature with 1 and 2 points respectively, whereas P2 and P4 ranked this feature with 3 points. From the interview it transpired that this was mainly caused by the runtime data not containing actual cases of a service being used more or less than another.

As for the question on whether the service filtering feature helps in finding out which services depend on each other, participants P2, P3 and P4 ranked it with 3, 4 and 3 points respectively, which is a reasonably positive score.

The fact that P1 answered all the questions regarding service filtering with 1 point can also be attributed to the fact that P1 did not identify any usefulness for Adyen's particular environment. This is also somewhat reflected in P2's answers, which despite more positive than P1's, are also generally lower scored than those of P3 and P4.

We conclude that the ability to filter the runtime topology based on particular services is mostly useful for systems with: a) a large amount of services and b) little insight about service usage. In the particular case of Adyen where the participants already possessed a very good understanding of the software system, it would seem that such a feature would provide no added value. Furthermore, Adyen's system also does not face regular changes in topology. It is also our understanding that for this particular system, there is a great deal of logging being done down at the line level. While this provides tremendous insight for post-mortem analysis when failures occur, it also generates a great amount of data which needs to be stored.

6.5.5 Version Filtering

On what concerns version filtering, the results were on the overall positive. Despite being clear to all participants that this filtering was available (V1), disagreements exist regarding whether it adds value (V2) and the overall usefulness of Serviz for different version-related tasks (V3 and V4). Namely, participant P1 ranked once again the usefulness questions with 1 point. This was something we pursued in depth during the

interview that came after the questionnaire and it came to light that Adyen's system does not possess explicit versioning. This eliminates the needs to filter usage per version, as the per-user configurations are done through conditional code calls.

Examining the question on dead code inference based on the runtime topology filtered by service version (V3), all but one participant agreed with a 4 point score. This feature is something which during the experiment left much to the imagination. Because the data under analysis did not actually include any event of a service becoming dead code, this was something the participants had to imagine rather than something they could in fact see during the experiment. This could explain why P1 scored the feature with a lower score compared to the remaining participants.

Lastly, as to whether it helps pinpoint which services to perform maintenance on (V4), the results are mixed. Participants agreed that this is something which could be possible should there be a bug which completely disables a service call. The only remark which came to light regarding this question came from P1 who expected an automated failure detection. Currently, the approach involves comparing periods of time when the system is healthy, with periods of time when a problem occurred. Then, with the information at hand, try to find out where the problem might exist. Automated failure detection is not something we tried to achieve with our current research. For this particular question, the remaining participants (P2, P3 and P4) ranked it with respectively 4, 5 and 3 points, therefore agreeing that the version filtering feature helps figuring out which service version requires maintenance.

6.5.6 Combined Filtering

In the previous sections we analyze the results for the individual filtering options used on their own. In the last round of questions we analyze to what extent all the features combined help in different maintenance tasks. Namely, we asked the participants whether it was possible, using version and user filtering, to identify which user accesses a particular version the most (C1). The results continue the trend of previous questions with participants P1 and P2 scoring lower (2 and 3 points respectively) and participants P3 and P4 providing more positive results (4 points for both participants). The lack of explicit versions in Adyen's software system makes the participants less prone to find the usefulness of version-related features. Additionally, participants P1 and P2 had claimed previously about the user-specific questions that it is a cumbersome task to manually filter through all the users. This is a possible explanation to the lower scores of these participants compared to the 4 points attributed by participants P3 and P4.

Regarding question C2, because of its focus on a particular user, the results are generally higher across all the participants and there is no major discrepancy between Adyen's and Exact's participants (4 points, 3 points, 3 points and 4 points).

6.6 Discussion

In this section, we discuss the results of the experiment with regards to the research questions. Subsequently, we talk about our lessons learned and we identify threats to validity.

6.6.1 The Research Questions Revisited

RQ4.1: *Does the user information added to the runtime topology help in the understanding of SOA-based systems?* We asked the four participants of our user study for their opinion on the usefulness of the runtime topology for understanding SOA-based systems. They all expressed themselves positively here, giving a first clear indication that user information does indeed help to understand the system and how it is used. The participants' appreciation for this feature is likely related to the multi-tenant requirement of knowing which tenants are using which parts of the system.
RQ4.2: *Does the service version information added to the runtime topology help in the understanding of SOA-based systems?* The opinions of the four participants somewhat diverged from each other on this point. In particular, participant P1 is of the opinion that the runtime topology is mainly useful for software immigrants, and not for software engineers who already have a thorough knowledge of the system, especially for systems that do not change very often and/or do not have explicit service versioning.

However, three participants do agree that in systems that change often and that do have explicit service versioning, the ability to add service versioning information to the runtime topology — and also filter on versioning information — helps in trying to understand a software system.

The same three participants were also positive about the fact that the service filtering feature helps them to better understand which services depend on each other.
RQ4.3: *Do usage graphs help to understand SOA-based systems?* Most participants think that the usage graphs available in Serviz provide a good combination with the user filtering, this way they could get a clear view of how a (set of) user(s) uses the system throughout time.

It should also be noted that with regard to the usage graphs, this is a feature of which an experimental prototype is being independently developed at one of the companies. This on itself highlights that this particular feature represents a real-world need.

Having discussed the subsidiary research questions, we are now in a position to answer our main research question: *Does the combination of user, service-version and timing information projected on a runtime topology help in the understanding of SOA-based multi-tenant software systems?* The results give a clear indication that most of the participants to our field study are positive towards Serviz and its features to better understand how users and service-versions interact with each other over time. We also specifically gauged for whether it is possible to find periods of low usage (for particular service-versions) which would prove more suitable for performing software maintenance when version *and* version filtering is combined. The results (2, 4, 5 and 4 points) indicate that this particular

filtering combination helps in finding periods of low usage.

6.6.2 Lessons Learned

The field study with 4 participants at two different companies also taught us a number of valuable insights that go beyond the augmented runtime topology.

In particular, for a number of questions we observed quite different answers from the participants belonging to the different companies. While all four participants are experts in the area of SOA-based software systems, the particular *context* of the company plays an important role in their appreciation. This underlines the importance of performing program comprehension experiments with participants that are from different contexts.

We witnessed a generally lower appreciation of Serviz by participant P1. From the interviews that we held as the final step of the field study, we gathered from him that he thought that Serviz was not particularly useful if you already have a good understanding of the system. Yet, he also acknowledges that novice users, the so-called software immigrants, would actually benefit from the overview that Serviz could give them. In effect, this seems somewhat strange, because at the beginning of the field study, we explicitly asked them to reason about Serviz from the point of view of an engineer exploring an unknown system (hence also the choice for Stonehenge, our subject system, which was not known to any of the participants). It seems that reasoning outside of the technological or business context in which one is immersed, is sometimes actually quite difficult.

This opens up the question of whether for program comprehension experiments, one needs to select experts in the field, people that are aware of the particular difficulties faced, or novice users, who are not familiar with all of the problems in the area and that have a no bias when it comes to experience with potential difficulties in understanding systems.

6.6.3 Threats to Validity

In this section we present a discussion of how the results of our experiment might be challenged. We discuss internal validity, namely whether the participants might have been directly affected by factors we did not consider, and external validity, meaning whether our results are generalizable.

6.6.3.1 Internal Validity

One possible threat to the internal validity could have been related to a lack in the participants' competence. However, in a short pre-study interview we did gauge for their experience with SOA and we established that all participants had at least 2 years of experience with developing SOA based systems. This, as we noted in Section 6.6.2, might have been both an advantage and a disadvantage. While experience with SOA is welcomed, we theorize it may have also been a source for bias towards a particular type of SOA system.

Participants might have been inclined to rate the tool more positively than they actually value it, because they might have felt this was the more desirable answer. We mitigated this concern by indicating to participants that only honest answers were valuable. The good mix of answers we have obtained is another indication of the participants answering the questions truthfully.

6.6.3.2 External Validity

Our main concern with external validity has to do with the performance impact of our approach. We are particularly concerned with the performance overhead added by the data collection step. However, the data collection is supported by a robust framework used by a company with a large web services infrastructure (eBay).

The storage requirement of our approach is also quite high [Zaidman et al., 2006]. In a system with a large traffic of web service requests, a large amount of storage space is required in order to maintain the system's state over time. This threat can be mitigated by using compression techniques, e.g., as applied in the Compact Trace Format (CTF) [Hamou-Lhadj and Lethbridge, 2012].

The applicability of our results to other, larger systems is also a possible threat. We tried to mitigate this by using Spicy Stonehenge, which, despite its small size, contains all the ingredients of an industrial system, including complex interactions between services and a well-specified domain.

6.7 Related Work

In general, the increased maintenance complexity of SOA-based systems has been acknowledged and emphasized by Lewis and Smith [Lewis and Smith, 2008], requiring for instance, impact analysis for an unknown set of users, or increased number of externally accessed services to be considered in maintenance. These are asking for a readjustment of current maintenance practice for SOA-based systems at large.

For this chapter, we started by analyzing the survey of Cornelissen et al. [Cornelissen et al., 2009] on program comprehension through dynamic analysis, which, among others, lists the work of De Pauw et al. [De Pauw et al., 2005]. They describe a web services navigator for generating service topologies, and focus on detecting incorrect implementations of business rules and "excessively chatty" communications. Our approach is different w.r.t. two significant improvements to the service topology: it allows to identify the method of an invoked service plus its invoking client, and it includes the time dimension providing a historic view on the topology.

White et al. [White et al., 2013] present a dynamic analysis approach to aid the maintenance of SOA-based composite applications where they propose using a feature sequence viewer to recover sequence diagrams from such systems. This approach, however, does not seem to provide a basis for understanding the topology of a running service-oriented system and rather focuses on mapping features to software artifacts.

In other work, White et al. [White et al., 2012] investigated the information of developers during the maintenance of SOA-based systems. They established that the first question a maintainer must ask is "how does the software work *now*?"

Finally, we investigated citations to the aforementioned papers, in particular the systematic survey by Cornelissen et al. [Cornelissen et al., 2009] in order to find more recent additions to the body of knowledge. Unfortunately, this search has yielded no extra relevant related work.

6.8 Conclusion

In this chapter, we investigate how the analysis of the users and versions in a multi-tenant system can help its understanding for maintenance purposes. More specifically, our contributions are:

- The runtime topology augmented with the time dimension.
- The runtime topology filtered by user and service version.
- Serviz, an open-source implementation of this approach.
- A field user study with 4 participants from two software engineering companies in order to evaluate the effectiveness of such an approach.

We now summarize how our approach and the tool address our original research questions formulated in Section 6.1:

RQ4.1 *Does the user information added to the runtime topology help in the understanding of SOA-based systems?* Our participants agree that having this information available helps in the understanding of a SOA-based system, with some remarks regarding automating the identification of important highlights (e.g. which users used the system the most in a specific period).

RQ4.2 *Does the service version information added to the runtime topology help in the understanding of SOA-based systems?* On what concerns service version information, on average all but one participant agreed that this having this information at hand makes understanding highly dynamic SOA-based systems easier. On systems where the runtime topology is less prone to change, the usefulness might prove to be diminished.

RQ4.3 *Do usage graphs help to understand SOA-based systems?* For our last research question, the participants generally agreed that time-based graphs add value to understanding SOA-systems and their maintenance. Furthermore, one of the companies is creating a prototype tool which offers a similar set of features, highlighting the relevance and usefulness of such a feature.

6.8.1 Future Work

From our interview, the participants noted several aspects which we would like to improve as future work. An improvement we plan to add to our runtime topology deals with automatic identification of useful data. For example, the participants claimed that identifying peaks of high and low usage for a particular service costs a significant amount

of time. Similarly, a common remark was the lack of a service usage referential. This means the participants had to manually calculate the usage of a particular service by dividing the number of requests by the specific time interval chosen. This is something which was also suggested as beneficial, should it be automated.

Another aspect we would like to further study is an automated dead code detection for service oriented systems. This is particularly important in the context of multi-tenancy where specific versions are tailored for a particular user, who might in the future migrate to another version.

Our ultimate goal is to keep on developing Serviz, make it more user-friendly and improve its visualization. Some participants complained the graph did not retain its position every time the dates were changed, which caused great frustration.

CHAPTER 7

Conclusion

In this thesis we investigate how the relatively young field of web services is coping with its growing pains. We investigate the issue by looking at both web service providers and web service consumers and by gathering anecdotal stories of such pains from the developers on both ends of this relationship. Our investigation of the aforementioned issues consists succinctly of the three following steps:

1. Interviewing developers who have experience in maintaining service-based systems as a means to gather insight on what hinders such developers on their software maintenance task.

2. Analyzing the source code of service-based systems (both consumers and providers) in order to identify the added challenges of providing and integrating web services and whether or not developers are aware of such added challenges.

3. Creating and evaluating a tool (Serviz) which through a runtime topology provides a runtime overview of the system, namely which services compose the system and how they interact with each other at different times, with different users who possibly use different service versions.

Web services, while in principle similar to statically linked APIs, represent a major shift in who controls the software evolution pace. This, in turn, leads to repercussions for both parties. On the one hand, web service providers must apply a higher degree of parsimony when it comes to pushing breaking changes to a web service at the risk of losing customers by not doing so. On the other hand, web service consumers must, in some cases, be ready to deal with more frequent changes. Our results show that the majority of the web service clients studied proved ready for changes to web service behavior. Each web service provider on the other hand, seems to have its own view of what is a good

evolution policy. As a web service provider, being aware of which particular client depends on which web services is something which can be achieved through a runtime topology and is indeed, something already used by some professional web service providers to allow for more flexibility when rolling out new web service versions. Some industrial web service providers do make use of state of the art approaches to ease the burden on web service client developers but generally web services still receive frequent changes and, in many cases, bear very little support for backwards compatibility. The lack of backwards compatibility, while less harmful in a statically linked environment where client developers have a choice not to migrate, appears to be the major source of the growing pains.

7.1 Summary of Contributions

- The insights of six professional developers regarding their experiences with web APIs that underwent the software evolution task.
- A comparison of the evolution policies of four high-profile web APIs (Google Maps, Twitter, Facebook and Netflix).
- An investigation of ten open source clients integrating with the aforementioned four web APIs demonstrating the impact web API evolution has on source code.
- Nine recommendations for developers of web APIs and client applications integrating web APIs.
- An investigation on the code impact on both server and client-side code for VirtualBox and XBMC demonstrating how invasive server-side changes can be on clients as well as how web service providers do not always differentiate web and non-web APIs.
- An approach using mutation analysis for simulating unexpected responses from web APIs.
- An experiment detailing how 43 high profile mobile applications react to a set of predefined mutations in web API responses aimed at mimicking web API evolution and failure.
- Insight on caching and versioning approaches of some high profile web APIs.
- The insight of three developers of some of the 43 studied mobile applications.
- Serviz, an open-source implementation of a runtime topology filterable by time, user and service version.
- Two user-studies, one with 8 participants and a further study with an additional 4 paticipants evaluating the effectiveness of the runtime topology.

- Spicy Stonehenge[1], an open-source service-based system used as a case-study for the runtime topology.

7.2 The Research Questions Revisited

7.2.1 RQ1 — How do web service APIs evolve and what are the consequences for clients of web APIs?

In Chapter 2, in order to gather anecdotal evidence from client developers regarding web API evolution pains we interview six professional developers who develop clients that integrate with web APIs. In the same chapter we also analyze the evolution policies for some of the most popular web APIs as means to find commonalities in these policies. We go further and, for 10 clients of these web APIs, we investigate the actual impact web API evolution has on source code by using code metrics. Lastly, because the web APIs studied are closed-source and do not allow for a similar investigation on the web APIs' source, we select two other open-source projects on which we perform an end-to-end analysis in order to investigate whether there are precautions in place to ease the impact of such web API evolution.

RQ1.1 — *What are some of the pains from client developers when evolving their clients to make use of the newest version of a web API?*.

Through our interviews, client developers highlighted how the early versions of web APIs are invariably unstable and change-prone. While some web API providers offer indicators of particularly unstable functionality in their web API, by default web API providers push breaking changes across the whole feature set. It also became clear that no standard policy exists on what concerns deprecation periods and that the ideal amount of time is dependent on the developer. Ideally longer periods would be provided but further study is required to establish what the cost would be for the web API provider to keep two versions of a web API active for a longer period of time. The technology being used also plays a role in the developers' satisfaction with an observed preference for REST and JSON amongst the interviewed developers.

RQ1.2 — *What are the commonalities in the evolution policies for web APIs?*.

A sentiment common amongst developers on public fora highlights the lack of industry standards regarding the web API practice. When it comes to evolution policies, this seems to be true as well. Google and Twitter make use of versioning and give ample periods of time (∼2 years and 6 months respectively) for the client developers to migrate. Facebook opts for not providing versioning altogether and pushes breaking changes every three months. Lastly, Netflix with already two existing versions continues to maintain both versions simultaneously. Twitter also stands out for the "blackout tests" which serve as warnings for developers that eventually the old web API version will be shutdown.

RQ1.3 — *What is the impact on source code when web APIs start to evolve?*.

[1] Spicy Stonehenge — `https://github.com/SERG-Delft/spicy-stonehenge`

As expected, the impact on source code depends greatly on both the breadth of the changes pushed by the web API provider and on the quality of the clients' architectural design. An example of this is two projects which integrate with the Twitter web API. While one of the projects provides an extensive integration with the web API, the churn caused by the changes is actually much *lower* than that of another project which performs more basic web API tasks. This same observation applies to the two Netflix projects. The code churn and file dispersion metrics have also had limited usefulness. For instance, the cartographer project contains changes in excess of 1000% of average churn and reports having 17 files changed, yet, the architectural design is robust as this project maintains support for multiple web API versions. The lesson learned is that the impact can be high (e.g. Google Maps pushed changes which affect the smallest of tasks) and that for this reason, developers should take caution and design for change. Lastly, our evidence also suggests that web APIs are significantly more change prone in their early versions.

RQ1.4 — *Do web API providers take precautions in order to ease evolution pains of web APIs?.*

To answer this question we focus on the VirtualBox and XBMC projects since they are the only ones for which we have access to the source code.

VirtualBox In the case of VirtualBox the fact that the web API is generated from the same interface as other statically linked APIs provides an early indication that no special care is taken regarding the web API. This is further fueled by a web API which suffers from a severe case of the multi-service anti-pattern where the web API and all its 1888 methods (and respective request and response types) from different business entities are provided under one single class. Perhaps as a result of this, and to the fact that the backwards compatibility responsibility has been delegated fully to the client side, the main client for this web API (phpVirtualBox) simply disregards the issue of backwards compatibility by releasing a new version for each of the VirtualBox releases. From the code analysis performed on VirtualBox it has also arisen that every single major and minor version has pushed breaking changes to the web API which reveals a great deal of external instability. Furthermore, the association rules mined out of the source code showed that multiple core implementation files consistently change together with the web API, revealing a potentially high fan-in between implementation and web API.

XBMC The XBMC web API was developed from the beginning as a web API. There are generally fewer breaking changes, with most breaking changes representing refactorings. Also from the association rule mining, only web API-related files (specifically, the files used for web API implementation) were observed to co-change with the main web API files. In this case study, it is then the client's fault for the poor backwards compatibility management. Namely, the Android XBMC client is publicly available in the Google Play Store and multiple end-users complain that the latest version simply does not work with older XBMC servers. The client developers do make older versions of the client available through their website but this requires manual installation and require multiple clients to be installed for controlling different servers with different versions.

In sum, referring back to RQ1.4, our analysis resulted in a mixed picture. On the one hand VirtualBox shows no particular arrangements made in order to maintain its web

API more or less stable than any other statically linked API, whereas XBMC contains a purpose-built web API with its own versioning and human-readable documentation. As a result, the clients under analysis also deal with their web API's differently (albeit neither with an ideal approach). The phpVirtualBox client simply releases one client version per web API version and Android XBMC keeps up with the latest version of the web API, disregarding backwards compatibility.

7.2.2 RQ2 — How well-prepared are Android mobile applications with regard to changes in response messages from the web API?

In Chapter 3 we test the robustness of some of the most popular applications in the Android Play Store which make use of web APIs through the simulation of unexpected responses. The question of *how* to simulate unexpected responses is also addressed in the chapter. Ultimately we make use of mutation analysis to generate unexpected (mutated) responses and assess how robust mobile applications are when web APIs reply with an unexpected response and/or behavior.

RQ2.1 — *How robust are mobile apps when the web APIs being used return unexpected responses?*

Our results present a mixed answer to this question. Indeed, most of the mobile applications studied are fairly robust to mutations in the web API response as seen by only 25% of the applications studied crashing through one of the mutations. Nonetheless, some of the mobile applications are not as resilient, as some applications crashed or silently failed upon facing changes to the web API. This behavior should be made more informative and user-friendly, which can be achieved through better understanding potential changes to web APIs.

RQ2.1a — *How can we simulate unexpected responses from web APIs*

The mutation analysis presents a structured approach to simulate web APIs afflicted either by failure or by changes caused by software evolution.

Indeed it was through the usage of mutation analysis that we are able to address the aforementioned [**RQ2.1**].

RQ2.2 — *Have web API client developers developed resilience against changes in the web API or failure of the web API?*

Some of the applications studied make use of state of the art approaches (e.g. the HATEOAS versioning) to ensure a smooth evolution of their web API client, where others do not use versioning altogether (which as reported in Chapter 2 may cause long-term pains) and allow the application to crash. The need for this resilience exists also outside of the source code. One of the interviewed developers raised concerns with inter-team communication, highlighting the need for clear and concise documentation from web API providers to client developers.

Our main research question asks *"how well-prepared are Android mobile applications with regard to changes in response messages from the web API"*. We conclude that while the majority of the studied applications are capable of dealing with such changes without

major issues, some applications still use web APIs as if their behavior can be expected to never change, which as we have seen does not always happen.

7.2.3 RQ3 — How can the topology of a running SOA-based system help in its maintenance?

In Chapter 5 we developed an implementation of a *runtime topology* for SOA-based systems and through a one-group pretest-posttest experiment investigate whether the runtime topology helps in identifying which services are involved in specific use cases. Also through the use of a runtime topology we asked the experiment participants whether such a topology is useful in debugging a service-based system. Our implementation of the runtime topology was also tailored to include the web service usage and interactions along the time axis which allows us to investigate whether this added dimension helps in identifying which services are used more often. Similarly, we also make use of the runtime topology to investigate whether it helps in identifying periods of low usage for e.g. maintenance with near-zero-downtime.

RQ3.1 — *Does the runtime topology help in identifying services involved in a specific use case?*

Our experiment participants agreed that by providing them with a runtime view of what the system looks like at any point in time, they can easily discover which services are involved in providing a specific functionality.

RQ3.2 — *Does the runtime topology help in debugging a SOA-based system?*

Similarly, participants have also indicated that they strongly agree that by comparing normal circumstances of a system to periods when a fault is occurring, the runtime topology aids them in debugging.

RQ3.3 — *Does the runtime topology augmented with the time-dimension help to identify services that are used more often (e.g., to optimize them)?*

Our user study indicates that participants also found that the runtime topology augmented with an invocation count helps them to quickly identify the most used services in a specific time interval.

RQ3.4 — *Does the runtime topology help to identify periods in time when the usage of a particular service is low, e.g., for evolving it with minimal disturbance to its users?*

Our experiment participants were not convinced Serviz helps them with this task, which we mainly attribute to our own implementation of the runtime topology falling short. In particular, this task involves manually scrolling the data hour by hour, while the participants would like to see this task automated.

7.2.4 RQ4 — Does the combination of user, service-version and timing information projected on a runtime topology help in the understanding of SOA-based multi-tenant software systems?

As a means to further explore whether the runtime topology studied in RQ3 can also be of use in multi-tenant software systems, we expanded it with information regarding users and service-versions. Additionally we added usage graphs per service on which peaks of high and low usage are visible for any chosen period. We then are able to investigate whether user and service information added to the runtime topology help in the understanding of service-based systems. Also instigated by our previous subsidiary research question (RQ3.4) we investigate whether usage graphs help in the understanding of SOA-based systems.

RQ4.1 — *Does the user information added to the runtime topology help in the understanding of SOA-based systems?* Our participants agree that having this information available helps in the understanding of a SOA-based system, with some remarks regarding automating the identification of important highlights (e.g. which users used the system the most in a specific period).

RQ4.2 — *Does the service version information added to the runtime topology help in the understanding of SOA-based systems?* On what concerns service version information, on average all but one participant agreed that this having this information at hand makes understanding highly dynamic SOA-based systems easier. On systems where the runtime topology is less prone to change, the usefulness might prove to be diminished.

RQ4.3 — *Do usage graphs help to understand SOA-based systems?* For our last research question, the participants generally agreed that time-based graphs add value to understanding SOA-systems and their maintenance. Furthermore, one of the companies is creating a prototype tool which offers a similar set of features, highlighting the relevance and usefulness of such a feature.

7.3 Recommendations For Future Work

As a result of the research conducted in this thesis, many issues remain still uninvestigated and pose interesting questions for future work. In this section we will highlight those which we believe to be more relevant:

7.3.1 Automated Web API Evolution

Henkel and Diwan [Henkel and Diwan, 2005] proposed an automated evolution mechanism for clients relying on APIs. Their approach relies on the API developer capturing the changes performed on the API and allowing client developers to use and *"replay"* this data as refactorings automatically performed on the API client. This idea was further

implemented as an Eclipse plugin by Xing and Stroulia [Xing and Stroulia, 2007] but its focus remains solely on static APIs.

Given that the *web* API source code does not always exist in the same Eclipse workspace as all of its clients' source code, it would be interesting to investigate whether such an approach would be usable in the web API context. More importantly, it would be interesting to analyze whether it would indeed help both web API providers and developers to be able to continuously evolve their software whilst minimizing hassle for both parties.

7.3.2 Versioning Data versus Versioning Interfaces

While versioning data is already an integral part of the REST approach for web API architectures [Fielding, 2000], Roy Fielding (the researcher behind REST) expressed his frustration at how developers insist on using high coupling interfaces and labeling them as as a REST API[2]. In the same blog post, Fielding argues that RESTful APIs must rely on the versioning of data rather than versioning the interface itself (an approach known as Hypermedia as the Engine of Application State or HATEOAS[3]).

This approach, while also advocated in Apigee's Web API Design book by Brian Mulloy[4], was not seen in practice in any of the analyzed web APIs. An analysis is required to investigate whether this approach does make the tasks of developing, evolving and integrating with a web API easier.

7.3.3 Runtime Topology

In Chapters 5 and 6 we emphasize the usage of a runtime topology as a means to understand a running service-based system. We identify which runtime data to collect, how to collect it and ultimately, how to visualize it in order to enable the understanding task. This understanding comes in turn, as a means to identify which clients make use of a particular version of a particular service and thus which clients would be affected by software maintenance on that particular version.

7.3.3.1 Code Smell & Anti-pattern Detection

While in a monolithic software system code smell detection can be done statically through the use of appropriate tooling, when the system is composed of distributed web services a static analysis approach may not always be possible. In such a scenario, having the runtime data collected in the context of the runtime topology would make it possible to infer web service dependencies and based on this information detect some code smells and/or anti-patterns.

[2]REST APIs must be hypertext-driven — http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, last visited: August 26th 2014

[3]HATEOAS — http://en.wikipedia.org/wiki/HATEOAS, last accessed October 6th 2014

[4]Web API Design — http://apigee.com/about/resources/ebooks/web-api-design, last visited August 27th 2014

As part of the runtime topology implementation, causality data is collected for each web service invocation. Namely, at what time did it happen and which web service method caused which other web service method to be invoked (as well as their respective versions).

Such data could, for example, enable the detection of *feature envy* where one class uses large chunks of one other class' functionality or even the detection of the *God class* anti-pattern where one class could potentially be central to different web service workflows and thus its functionality should probably be broken into smaller web services.

Further investigation is needed as to whether this data could be used for the detection of these and other code smells as set out by Fowler [Fowler and Beck, 1999] and brought out as warnings within the runtime topology.

7.3.3.2 Visualization

The evaluation of our runtime topology visualization revealed that the user interface was not always presenting the information in a way developers could intuitively use to understand the system under analysis. For example, while displaying a number of transactions in the runtime topology, it becomes impossible to see which web service was involved in a particular transaction (even though the collected data contains this information).

While our implementation did enable a certain degree of understanding of a running service-based system it was highly preliminary and experimental. It became clear through the user-study that some information which the system maintainers deemed useful was in fact obscured by the interface. Therefore an interesting aspect to further investigate is to, together with system maintainers, investigate how such maintainers go about understanding their system when the need arises and, when this happens, what information would be useful to have available. Having such a study would then reveal essential aspects which could be implemented in the runtime topology's visualization and further increase its usefulness.

7.3.3.3 Fan-In Analysis

The work of Marin et al. [Marin et al., 2007] proposes the use of fan-in analysis for the purpose of feature extraction through aspect mining. The ultimate motivation is that, according to the authors, methods with a high fan-in metric are likely to represent features required in more than one particular use-case. The authors then use this knowledge to extract such functionality into aspects which can then be *woven* at compile-time into the code they were extracted from.

In a system which is purely service-oriented and every component of the system is a web service (as proposed in the context of the runtime topology), due to the potentially large number of web services running in different hardware, it is not trivial to calculate the fan-in metric. In the aforementioned study the authors make use of an Eclipse plugin for this calculation, however, in a web service-based system it is not realistic to expect all the web services to be available as Eclipse projects. Thus this calculation may prove to be impossible through the usage of such a plugin (as the software system is divided into independent web services).

Through the usage of the runtime topology we collect data regarding the causal dependencies, not only between web services but also the specific method which caused the invocation as well as which method was invoked. An interesting research path would be to investigate whether the runtime topology could be used to identify high fan-in methods and with this knowledge, whether aspects could be used in the context of web services in order to extract frequently used functionality into aspects.

7.3.4 Dead Code Warnings

Currently a large body of research exists which addresses the topic of dead code detection and/or removal [Chen et al., 1998] [Knoop et al., 1994] [Chang et al., 1991]. Indeed, in monolithic systems where all the possible components of the software system are available at compile time, such an approach is possible through either static or dynamic analysis.

When deploying web services however, it is impossible to know both at compile-time and runtime whether a particular web service is still being used by a third-party. Indeed, it often happens that web service providers deploy a new version of a web service but maintain the older version available for backwards compatibility purposes. When this happens, such older versions *may* eventually become dead code which no third-party uses anymore. When this happens, the web service provider has no means to know with certainty that this is the case (e.g. maybe some web services are only used every other month or every other year).

Through the usage of the data collected in the runtime topology, a heuristic could be devised based on previous web service usage which alerts to potentially dead web services and/or service methods. Rather than immediately remove such *potential dead code* such an approach could raise developers' awareness who could then manually further investigate such alerts.

7.3.5 Web API Documentation Mining

While SOAP-based web APIs provide a WSDL file which, through describing the web service interface provides some degree of documentation, specifically the web APIs analyzed in Chapter 3 make use of XML documents for data encoding whilst not using WSDL documentation. The same is true for the studied REST (JSON-based) interfaces.

Considering the work of Endrikat et al. [Endrikat et al., 2014] which alerts on how documentation of APIs has a positive effect on development time it is then no suprise that Subramanian et al. [Subramanian et al., 2014] propose the mining of API usage examples from online textual repositories (e.g. StackOverflow) as a means to generate links between proposed usage examples to existing API documentation. With this in mind and considering the remark from the interviewed developers in Chapter 2 who complained that at times web APIs do not provide usage examples and the documentation is unclear, it would be interesting to investigate whether documentation such as a partial web service interface could be mined from an existing and previously integrated web service client.

Indeed, we would like to investigate in future work how web API providers could mine incoming web API requests at runtime and based on such requests reconstruct a usage-based documentation of their own web API. Because such an approach would require the web API provider to take the initiative, the same approach could also be applied on the client-side. Client developers could reconstruct the web API interfaces of the web APIs with which their clients integrate as a means to understand how exactly their client depends on the web API (e.g. which methods and fields are used) and whether the client will be affected by e.g. upcoming breaking changes.

7.3.6 Metrics for Web API Frequency of Change

Previous work from Coscia et al. [Coscia et al., 2013] makes use of existing source code metrics commonly used in object-oriented code to predict quality attributes of web service interfaces. While the quality attributes under study give an indication on a number of factors (e.g. modularity, adaptability and reusability amongst others), to the extent of our investigation no metric exists which analyzes frequency of change. This is a potentially important metric as each web service provider has a different policy on what concerns how often changes are pushed to the web service API. Such a metric could then be combined with the work of Romano and Pinzger [Romano and Pinzger, 2012] which makes use of fine-grained changes for understanding web service evolution, and by having both *frequency of change* as well as the size of *fine-grained interface changes*, another metric could be devised to warn developers of particularly change-prone web service APIs.

7.3.7 Coupling Metrics for Organizational Co-evolution

This thesis highlights how web API clients are dependent on the web API providers' evolution policies. This dependency extends beyond the source code and influences organizations' development cycles. In Chapter 2 we use code churn as a means to quantify the amount of change caused by this dependency but code churn does not always present a clear picture.

For instance, code churn as a cause of web API evolution is also dependent on other factors (e.g., architectural design of the web API client). However, combining it with a coupling metric for web API clients may provide web API client developers with new insights on quantifying the impact this co-evolution has on their clients' code.

7.3.8 Per-user, At-will Web API Version Migration

One of the aspects which web service providers seemingly fail to agree upon is on how long support for older web service versions should be maintained. Our findings show that it ranges from up to two years to as short as three months. From a web API client developer's satisfaction point of view, web API client developers would be allowed to, individually, choose when to migrate to a newer version as it has happened until now with static libraries.

Indeed, Facebook allows each client developer such a choice (and amongst the web API providers studied, is unique in doing so) but at the same time, forces each client developer to migrate within three months.

It would be interesting to analyze whether such an approach could be generalized for other web API providers and what the cost would be of maintaining such a system which allows migration at will on a per-user basis. An investigation into how this could be achieved with near-zero-downtime could set forward an entire new methodology on how to reliably provide a web service API without the concern that client developers have to deal with the changes on short notice.

7.3.9 Closed versus Open-source

Most of the research in this thesis has been conducted on open-source software. Namely when investigating web API *evolution policies* and *source code impact*, closed-source web APIs may be more cautious especially if there is a financial interest involved (e.g. the client is paying for a service provided through a web API). While one of the interviewed developers suggested that this may indeed be the case, it still remains to be investigated.

Acknowledgement

To the SERG group & our industrial partners,

My thanks go first to my supervisor and co-promoter, Andy Zaidman. He made all of this possible by encouraging me to apply after I approached him while he was a visiting lecturer in Leicester, UK where I was doing my Master's. Andy is an excellent teacher and he was always available whenever I needed to pick his brains. Thank you for this opportunity. My thanks go also to Hans-Gerhard Gross. Even though he ultimately moved on with his career back to his home country, our threesome discussions were always enrichened by *Gerd's* out-of-the-box thinking. Whenever we were seemingly stuck, it was most often Gerd who came up with some seemingly crazy but ultimately interesting idea.

Omdat ik mijn dankwoord in 4 talen wil doen, bedank ik mijn promotor in het Nederlands. Professor Arie van Deursen, heel erg bedankt. Ondanks dat onze conversaties vaak kort waren, het was altijd innemend voor mij om te zien hoe jij de blijheid van mensen boven alles plaatst. Bedankt dat je mij toeliet in jouw onderzoeksgroep om onderzoek te doen en zo mijn doctorstitel te halen.

My thanks go also to my officemates, Cuiting Chen and Cor-Paul Bezemer. Thank you Cuiting for the crash-course on Chinese culture which helped me greatly in understanding how the Chinese and Portuguese cultures are not all that different. Thanks Cor-Paul for always making me laugh even in the gloomy mornings when I had nothing to laugh about. I also learned a great deal about Dutch culture from you, for which I am eternally grateful.

Daniele Romano, my first conference (and conference accommodation) was shared with you. I can still remember hot and humid Williamsburg, VA as if it was yesterday. I will also never forget "Misery" (or James!). Thanks!

I would also like to thank the rest of the SERG group for the nice research environment made possible because of you.

Also to our industrial partners at Adyen: thank you Maikel, Peter and Bert for your availability in helping out with our studies. My thanks goes also to our partners at Exact: thank you Mark, Bart and Nenad!

To my relatives,

Sócito... é com muita pena que já não me vais poder ver nesta nova etapa da minha vida. Estejas onde estiveres, obrigado por tudo e por de uma forma ou de outra, teres

contribuido para eu hoje ser quem sou. É algo que eu certamente nunca irei esquecer. A tua prematura partida deixou um vazio o qual eu não esperava de ter de enfrentar tão cedo.

Não podia também deixar de agradecer à minha avó Carmo por ser alguém sempre muito presente na minha vida. Obrigado por tudo o que sempre fizeste por mim. Este livro é o culminar do percurso que começou contigo e com o Sócio no armazém a ler caixas de copos. Agora, muitos anos depois, deixei de ler caixas e passei a escrever livros.

Ao meu pai e à minha mãe, agradeço por todos os sacrifícios que fizeram para que eu hoje possa estar onde estou e por sempre me terem encorajado e apoiado os meus mais caros caprichos. Vocês, de uma forma ou de outra, tornaram tudo isto possível e fizeram-no a partir do nada (ou de muito pouco). Talvez não o diga tantas vezes quanto devia mas acreditem no seguinte: admiro-vos muito. É em parte graças a vocês que os meus filhos não vão poder usar a frase *"não sou filho de nenhum doutor!"*. Espero ter-vos orgulhado até agora e tenciono continuá-lo a fazer, na medida do que me for possível.

Um abraço também muito especial ao meu caro amigo Pedro Sousa. Mesmo a mais de dois milhares de quilómetros de distância as nossas conversas contribuiram muito para eu ter mantido a minha sanidade mental. Muito obrigado por tudo.

Deixo também aqui uma palavra de agradecimento a todos os meus familiares que de uma forma ou de outra contribuiram e continuam a contribuir para que eu possa continuar esta aventura por mares navegados pelos meus antepassados.

Also for my parents in law, 爸爸妈妈, 别担心。你们姑娘出生的时候有了一个家，你们家。现在她有三个家：一个在中国，一个在荷兰，一个在葡萄牙。你们姑娘出生后有了一家人疼爱她。现在她有两家人疼爱她。最后到底我觉得变好了！我感谢你们接受我为你们的女婿。每次跟你们在一起的感觉和跟父母在一起的感觉一样。

To my friends,

To 胡滨 and 杨宇光, thank you for being part of our Fridays and sharing with us all those movie nights! Hope you two hang around!

To all the remaining friends: Claudia Hauff, TaoKe, Alberto González-Sánchez and Zhutian, Éric Piel and WangXin, HuYu, RenHuijun, Tina, Josselin Pello and MaHaiyan, LiZhang, JianFei, Monica Abrudan, Katy Thomas, and Kamilla Bremeraunet: my thank you for one way or another having made my life a little better.

To my wife,

My wife, 赵俏功, is left for last. The first place is reserved for the supervisor and my wife deserves the closing word. I am also acknowledging her in her native language as both an attempt to leave the non-Chinese perplexed and an attempt to leave her Chinese name in the annals of history through my thesis. 小玶玶：我非常爱你。我非常感谢你。你一直不让我放弃，一直鼓励我完成我的博士。没你的话我有可能会辞职。。。然后会后悔。生活有时候有点意思。第一次我看见你，我还不知道天已经知道的缘分：你是我的另一半，你会是我的老婆。我希望跟你一生一世在一起。一生一世"从不觉得你讨厌，你的一切都喜欢，你是白云我是蓝天。" — 筷子兄弟。

Summary

Web Service Growing Pains: Understanding Services and Their Clients
- Tiago Espinha -

At an implementation level, web services serve the basic purpose of message exchange between potentially heterogeneous software systems. Through abstracting language- and platform-specific implementations into text-based, human-readable XML and JSON-based formats, different software systems are able to execute procedures and retrieve data from remote systems, in many cases provided by a third-party. In this thesis, we analyze web services from two perspectives: web services used to provide an interface with which third-party clients can integrate, and web services used as the components used to build a software system. This distinction is made in the sections below together with the different challenges addressed in this thesis.

Web Services for Integration

When studying web services as a means for integration, the major challenge we address in this thesis relates to how software systems naturally evolve to keep up with ever-changing laws, services and technologies. When using a static software library such as a Java ARchive (JAR), it is up to the client developer to decide *when* and *if* to integrate such new *"evolved"* versions of the library. Therefore, any effort of integrating a new version could be delayed at the developer's own discretion. When integrating a client with web services, this is no longer the case. It is then the web service *provider* who decides when the web service will bear new features and/or (potentially breaking) changes in behavior. It is also the web service provider who decides whether the older versions will remain accessible and for how long. In such a scenario, client developers must deal with an added pressure of conformity. If their client is not compatible with the new version it may simply stop to work when support for the older version is removed.

This power-shift in who controls the evolution pace of service integration led us to study two facets of this client/provider relationship:

Web Service Providers. We investigated whether this power-shift actually happens in practice by studying real world examples of breaking changes pushed by high-profile web service providers (Facebook, Twitter, ...) and the impact they have on client source

code. We found that, indeed, in many of these instances the changes are breaking and invasive. Moreover, some client developers are also unhappy with both the structuring and frequency with which web service providers structure their (breaking) changes. Another interesting finding is how web service providers lack standard best-practices which all the web service providers agree on. Instead, each web service provider independently decides on which policies they will follow regarding evolution, backwards compatibility and versioning (or lack of versioning).

Web Service Client Developers. Seeing as client developers are sometimes inconvenienced by breaking changes, we also investigated whether they have then developed resilience against potentially unstable and frequently changing web services. To achieve this we make use of mutation analysis to simulate evolving and failing web services and observe the behavior of a set of Android applications which integrate with these web services. While the results are mixed, a considerable number of applications crashed upon facing these changes which hints that not all client developers are aware of the added responsibilities when integrating a third-party web service.

Web Services for Service-Orientation

Web services are also used as building blocks for software systems in the so-called Software Oriented Architectures (SOA). When that is the case, it is often difficult (and at times impossible altogether) to understand what are the repercussions of changing the functionality of a specific web service. This is due to the loosely coupled nature of web services which can, in some cases, resolve dependencies at runtime. In order to address this, we created and evaluated an implementation of a runtime topology (Serviz) which provides us with this information.

The runtime topology provides the system maintainer with an overview of which web services communicated with which other web services. By having this overview of causality in web service requests and therefore which web service methods depend on which other methods, system maintainers are then better able to understand which users and which other web service methods are affected when performing maintenance on a specific web service method. Our runtime topology provides such an overview while also allowing for different types of filtering (which can be combined):

- Time-based filtering, which allows for restricting the interactions to a specific period of time.
- Service-based filtering, which in large service-based systems helps in pinpointing all interactions and thus other web services which are invoked together with a particular web service.
- Version-based filtering, which allows the filtering to be done with a higher degree of granularity and for choice to be done on a particular version of a web service.
- User-based filtering, which shows the web service interactions only for a single user.

Besides the runtime topology, a usage graph is also computed per web service which allows system maintainers to visually identify high and low usage peaks. Such graph is potentially useful for identifying the potentially best periods for performing software maintenance.

Conclusion

Ultimately, whether for an integration or service-orientation scenario, our results suggest that the lack of backwards compatibility in an environment where services actively depend on each other is one of the major causes of pain for web service client developers. An aspect which remains yet to be explored is whether providing such backwards compatibility is feasible and at what cost and effort does it come for web service providers.

Samenvatting

Web Service Growing Pains: Understanding Services and Their Clients
- Tiago Espinha -

Op een implementatie niveau hebben web services tot doel de berichtenuitwisseling tussen mogelijkerwijs heterogene software systemen te bewerkstelligen. Door taal- en platform-specifieke implementatie details te abstraheren en gebruik te maken van tekstgebaseerde en makkelijk leesbare XML en JSON data formaten, kunnen software systemen procedures uitvoeren op en data ophalen van andere software systemen, die mogelijk niet-lokaal draaien en die worden beheerd door derden.

In dit proefschrift analyseren we web services vanuit twee invalshoeken: enerzijds kijken we naar web services als interfaces voor derde partijen die hun client applicaties willen integreren. Anderzijds focussen we op web services die als componenten in een software systeem worden gebruiken. Dit onderscheid bespreken we in de volgende paragrafen samen met de specifieke uitdagingen die we in deze thesis adresseren.

Web Services voor Integratie

Voor web services die als integratiemiddel tussen computer systemen dienen, kijken we in dit proefschrift naar de uitdagingen die gerelateerd zijn aan de evolutie van software die typisch wordt ingegeven door de continue verandering in wetgeving, diensten en technologieën. In het geval van een statische bibliotheek zoals een Java ARchive (JAR), is het aan de cliënt ontwikkelaar om te bepalen *of* en *wanneer* te integreren met een nieuwe en *"geëvolueerd"* versie van de bibliotheek. Dit houdt in dat het aan de cliënt ontwikkelaar is om te bepalen wanneer de investering voor de integratie van de upgrade wordt gemaakt.

Bij de integratie van een cliënt applicatie met web services heeft de cliënt ontwikkelaar geen controle meer over het moment van integratie van de upgrade. In dit geval is het de *provider* die bepaalt wanneer nieuwe functionaliteiten en/of "breaking changes" worden uitgebracht. Het is ook de web service provider die bepaalt of oudere versies van de web service beschikbaar blijven en voor hoe lang. In dit scenario moet cliënt ontwikkelaars omgaan met de toegevoegde druk van conformiteit: als hun client niet compatibel is met de nieuwe versie, dan bestaat de kans dat de client stopt met werken zodra de oudere versie van de web service wordt verwijderd.

Deze machtsverschuiving van wie de controle heeft over de snelheid van software evolutie heeft ons geleid tot het bestuderen van twee facetten van deze cliënt/provider relatie:
Web Service Providers. We hebben onderzocht of deze machtsverschuiving zich in de praktijk ook echt voordoet, door voorbeelden van breaking changes te onderzoeken die door bekende web service providers (Facebook, Twitter, ...) zijn doorgevoerd. We hebben bovendien in kaart gebracht hoe deze breaking changes de broncode van clients be nvloeden. De resultaten van onze studie wijzen uit dat deze breaking changes leiden tot invasieve veranderingen voor de cliënten. Bovendien hebben we vastgesteld dat sommige cliënt ontwikkelaars niet blij zijn met de structuur en de frequentie van de (breaking) changes. Een andere interessante bevinding is het gebrek aan "best-practices" met betrekking tot de evolutie van web services. In plaats daarvan voert elke web service provider een eigen beleid met betrekking tot evolutie, achterwaartse compatibiliteit en versiebeheer (of gebrek aan versiebeheer).
Web Service Cliënt Ontwikkelaars. Aangezien cliënt ontwikkelaars soms gehinderd worden door breaking changes, hebben we ook onderzocht of ze maatregelen nemen om beter om te kunnen gaan met mogelijkerwijs onstabiele en vaak veranderende web services. Om dit te bereiken gebruiken we mutation analysis om evoluerende en gebrekkige web services te simuleren. Deze mutation analysis hebben we toegepast op een verzameling van Android applicaties die integreren met web service. Hoewel de resultaten een gemengd beeld geven, geven onze resultaten aan dat een aanzienlijk aantal applicaties crasht. Dit duidt goed aan dat niet alle cliënt ontwikkelaars zich bewust zijn van hun verantwoordelijkheden bij de integratie met een web service aangeboden door een derde partij.

Web Services voor Service-oriëntatie

Web services zijn ook als bouwstenen te gebruiken voor software systemen in zogenaamde Software Oriented Architectures (SOA). In deze context is het vaak moeilijk (en soms helemaal onmogelijk) te begrijpen wat de repercussies zijn van het wijzigen van de functionaliteit van een specifieke web service. Dit komt door de losse koppeling van web services en het feit dat afhankelijkheden slechts bij runtime duidelijk worden. Om tegemoet te komen aan dit probleem, hebben we een implementatie van een runtime topology ontwikkeld en ge valueerd (Serviz), die de runtime afhankelijkheden in kaart brengt.

De runtime topology geeft de systeem ontwikkelaar een overzicht van welke web services met welke anderen web services communiceren. Door middel van dit overzicht van causaliteit in web service requests en daardoor welke web service methoden afhangen van welke anderen methoden, zijn systeem ontwikkelaars (en onderhouders) beter in staat te begrijpen welke users en welke andere web service subprogramma's worden beïnvloed door onderhoud in een specifiek web service subprogramma.

Onze runtime topology levert zo'n overzicht terwijl bieden de mogelijkheid voor verschillende soorten van filtering (die ook samen toegepast kunnen worden):

- Tijd-gebaseerde filtering, die gebruikt kan worden om interacties te beperken tot een specifiek tijdsperiode.
- Service-gebaseerde filtering, die in grote service-based systemen helpt bij het lokalis-

eren van alle interacties tussen een geselecteerde web-service en de web-services die gebruik maken of afhankelijk zijn van de geselecteerde web-service.
- Versie-gebaseerde filtering, die gebruikt kan worden om een bepaalde versie van een web-service en zijn interacties beter in kaart te brengen.
- Gebruiker-gebaseerde filtering, die toelaat om web-service interacties van een bepaalde gebruiker in kaart te brengen.

Als aanvulling op de runtime topology wordt ook een gebruiksgrafiek gegenereerd die toelaat systeem ontwikkelaars toelaat om inzicht te krijgen in periodes van hoog of laag gebruik van bepaalde services. Deze grafiek laat mogelijkerwijs toe om het beste moment voor onderhoud aan web-services te identificeren.

Conclusie

Onze resultaten geven aan dat de afwezigheid van achterwaartse compatibiliteit in een omgeving waar web-services actief van elkaar afhangen kan leiden tot ongemak bij web service cliënt ontwikkelaars. Dit fenomeen doet zich bovendien zowel in het integratie als in het service-oriëntatie scenario voor. Een belangrijk pad voor toekomstig onderzoek blijft of deze achterwaartse compatibiliteit mogelijk is en wat de kost voor web-service providers is om deze aan te bieden.

Bibliography

Ahmad, A. and Pahl, C. (2011). Customisable transformation-driven evolution for service architectures. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 373–376. IEEE CS.

Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2010). *Web Services: Concepts, Architectures and Applications*. Springer Publishing Company, Incorporated, 1 edition.

Ardissono, L., Furnari, R., Goy, A., Petrone, G., and Segnan, M. (2006). Fault tolerant web service orchestration by means of diagnosis. In *Proceedings of the Third European Workshop on Software Architecture (EWSA)*, volume 4344 of *LNCS*, pages 2–16. Springer.

Babbie, E. (2007). *The practice of social research, 11th edn.* Wadsworth Belmont.

Barbon, F., Traverso, P., Pistore, M., and Trainotti, M. (2006). Run-time monitoring of instances and classes of web service compositions. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 63–71. IEEE CS.

Baresi, L., Ghezzi, C., and Guinea, S. (2004). Smart monitors for composed services. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, pages 193–202. ACM.

Benatallah, B. and Motahari Nezhad, H. (2008). Service oriented architecture: Overview and directions. In *Advances in Software Engineering*, volume 5316 of *LNCS*, pages 116–130. Springer.

Benbernou, S., Hacid, L. C. M. S., Kazhamiakin, R., Kecskemeti, G., Poizat, J.-L., Silvestri, F., Uhlig, M., and Wetzstein, B. (2008). State of the Art Report, Gap Analysis of Knowledge on Principles, Techniques and Methodologies for Monitoring and Adaptation of SBAs. Deliverable # PO-JRA-1.2.1 of the S-Cube project.

Bertolino, A., Inverardi, P., Pelliccione, P., and Tivoli, M. (2009). Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 141–150. ACM.

Bertolino, A. and Polini, A. (2009). Soa test governance: Enabling service integration testing across organization and technology borders. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 277–286. IEEE CS.

Bezemer, C.-P. and Zaidman, A. (2010). Multi-tenant saas applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92. ACM.

Blank (YourTrove), S. (2011). Api integration pain survey results. https://www.yourtrove.com/blog/2011/08/11/api-integration-pain-survey-results/. Website last visited September 27, 2013.

Borissov, L., Brodkorb, T., Driesen, V., Georgiev, A., Mihalev, I., and Mitev, D. (2010). Zero downtime mechanism for software upgrade of a distributed computer system. US Patent App. 12/337,675.

Burns, C., Ferreira, J., Hellmann, T., and Maurer, F. (2012). Usable results from the field of api usability: A systematic mapping and further analysis. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 179–182. IEEE.

Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., and Tamburrelli, G. (2011). Dynamic qos management and optimization in service-based systems. *Software Engineering, IEEE Transactions on*, 37(3):387–409.

Campbell, D., Stanley, J., and Gage, N. (1963). *Experimental and quasi-experimental designs for research*. Rand McNally.

Canfora, G. and Di Penta, M. (2007). New frontiers of reverse engineering. In *Future of Software Engineering (FOSE)*, pages 326–341. IEEE CS.

Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., and Tourwé, T. (2006). Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231.

Chang, P. P., Mahlke, S. A., and Hwu, W.-M. W. (1991). Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321.

Chen, C., Zaidman, A., and Gross, H.-G. (2011). A framework-based runtime monitoring approach for service-oriented software systems. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications (QASBA)*, pages 17–20. ACM.

Chen, Y.-F., Gansner, E., and Koutsofios, E. (1998). A c++ data model supporting reachability analysis and dead code detection. *Software Engineering, IEEE Transactions on*, 24(9):682–694.

Cheng, F.-T., Wu, S.-L., Tsai, P.-Y., Chung, Y.-T., and Yang, H.-C. (2005). Application cluster service scheme for near-zero-downtime services. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 4062–4067.

Chikofsky, E. J. and II, J. H. C. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

Christensen, J. H. (2009). Using RESTful web-services and cloud computing to create next generation mobile applications. In *Proc. Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA-companion)*, pages 627–634. ACM.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306.

Cornelissen, B., Zaidman, A., and van Deursen, A. (2011). A controlled experiment for program comprehension through trace visualization. *IEEE Trans. Software Eng.*, 37(3):341–355.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. Software Eng*, 35(5):684–702.

Coscia, J. L. O., Crasso, M., Mateos, C., and Zunino, A. (2013). Estimating web service interface quality through conventional object-oriented metrics. *CLEI Electron. J.*, 16(1).

Creswell, J. W. (2009). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S. (2002). Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing*, 6(2):86–93.

Dagenais, B. and Robillard, M. P. (2008). Recommending adaptive changes for framework evolution. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 481–490. ACM.

Dagenais, B. and Robillard, M. P. (2009). Semdiff: Analysis and recommendation support for API evolution. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 599–602. IEEE.

Daigneau, R. (2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley.

De Pauw, W., Lei, M., Pring, E., Villard, L., Arnold, M., and Morar, J. F. (2005). Web services navigator: Visualizing the execution of web services. *IBM Systems Journal*, 44(4):821–846.

Demeyer, S., Mens, T., and Wermelinger, M. (2002). Towards a software evolution benchmark. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 172–175. ACM.

Denaro, G., Pezzè, M., and Tosi, D. (2009). Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 253–262. ACM.

Dig, D. and Johnson, R. E. (2006). How do APIs evolve? A story of refactoring. *Journal of Software Maintenance*, 18(2):83–107.

Domenico, B. and Carlo, G. (2007). Monitoring conversational web services. In *Proceedings of the 2nd international workshop on Service oriented software engineering (IW-SOSWE)*, pages 15–21. ACM.

Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., and Osborne, D. (2002). *J2EE Antipatterns*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition.

Elliott Sim, S. and Holt, R. C. (1998). The ramp-up problem in software projects: a case study of how software immigrants naturalize. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 361–370. IEEE CS.

Endrikat, S., Hanenberg, S., Robbes, R., and Stefik, A. (2014). How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 632–642, New York, NY, USA. ACM.

Ericsson, K. A. and Simon, H. A. (1998). How to study thinking in everyday life: Contrasting think-aloud protocols with descriptions and explanations of thinking. *Mind, Culture, and Activity*, 5(3):178–186.

Espinha, T. (2014). Web API Responses. http://dx.doi.org/10.6084/m9.figshare.1202060.

Espinha, T., Chen, C., Zaidman, A., and Gross, H.-G. (2012a). Maintenance research in SOA — towards a standard case study. In *Proc. of the Conf. on Software Maintenance and Reengineering (CSMR)*, pages 391–396. IEEE CS.

Espinha, T., Chen, C., Zaidman, A., and Gross, H.-G. (2012b). Spicy stonehenge: Proposing a soa case study. In *Principles of Engineering Service Oriented Systems (PESOS), 2012 ICSE Workshop on*, pages 57–58.

Espinha, T., Zaidman, A., and Gross, H.-G. (2011). Understanding service-oriented systems using dynamic analysis. In *Proceedings of the International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*, pages 1–5. IEEE CS.

Espinha, T., Zaidman, A., and Gross, H.-G. (2012c). Understanding the runtime topology of service-oriented systems. In *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, pages 187–196. IEEE CS.

Espinha, T., Zaidman, A., and Gross, H.-G. (2013). Understanding the interactions between users and versions in multi-tenant systems. In *Int'l Workshop on Principles of Softw. Evol.*, pages 53–62. ACM.

Espinha, T., Zaidman, A., and Gross, H.-G. (2014a). Web API fragility: How robust is your web api client. Technical Report TUD-SERG-2014-009, Delft University of Technology.

Espinha, T., Zaidman, A., and Gross, H.-G. (2014b). Web API growing pains: Loosely coupled yet strongly tied. *J. Syst. Software*.

Espinha, T., Zaidman, A., and Gross, H.-G. (2014c). Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 84–93. IEEE CS.

Fang, R., Lam, L., Fong, L., Frank, D., Vignola, C., Chen, Y., and Du, N. (2007). A version-aware approach for web service directory. In *IEEE International Conference on Web Services (ICWS)*, pages 406–413. IEEE CS.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis. AAI9980887.

Fluri, B. and Gall, H. C. (2006). Classifying change types for qualifying change couplings. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 35–45.

Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., and Lau, A. (2011). An empirical study on web service evolution. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 49–56.

Fokaefs, M. and Stroulia, E. (2012). Wsdarwin: Automatic web service client adaptation. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 176–191, Riverton, NJ, USA. IBM Corp.

Fokaefs, M. and Stroulia, E. (2014). Wsdarwin: Studying the evolution of web service systems. In Bouguettaya, A., Sheng, Q. Z., and Daniel, F., editors, *Advanced Web Services*, pages 199–223. Springer New York.

Fowler, M. and Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley.

Gold, N., Knight, C., Mohan, A., and Munro, M. (2004). Understanding service-oriented software. *IEEE Software*, 21(2):71–77.

Groth, D. and Skandier, T. (2005). *Network+ Study Guide: Exam N10-003*. Wiley.

Hamou-Lhadj, A. and Lethbridge, T. C. (2012). A metamodel for the compact but lossless exchange of execution traces. *Software and System Modeling*, 11(1):77–98.

Henkel, J. and Diwan, A. (2005). Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 274–283. ACM.

Henning, M. (2008). The rise and fall of corba. *Commun. ACM*, 51(8):52–57.

Hevner, A. and Chatterjee, S. (2010). Design science research in information systems. In *Design Research in Information Systems*, pages 9–22. Springer.

Heward, G., Müller, I., Han, J., Schneider, J.-G., and Versteeg, S. (2010). Assessing the performance impact of service monitoring. In *Australian Software Engineering Conference (ASWEC)*, pages 192–201. IEEE CS.

Holtzblatt, K. and Jones, S. (1995). Human-computer interaction. chapter Conducting and analyzing a contextual interview (excerpt), pages 241–253. Morgan Kaufmann.

Jacobson, D., Woods, D., and Brail, G. (2011). *APIs: A Strategy Guide*. Oreilly and Associate Series. O'Reilly Media.

Jerding, D. F., Stasko, J. T., and Ball, T. (1997). Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 360–370, New York, NY, USA. ACM.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

Josuttis, N. M. (2007). *SOA in Practice: The Art of Distributed System Design*. O'Reilly.

Kabbedijk, J., Bezemer, C.-P., Jansen, S., and Zaidman, A. (2014a). Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*.

Kabbedijk, J., Pors, M., Jansen, S., and Brinkkemper, S. (2014b). Multi-tenant architecture comparison. In Avgeriou, P. and Zdun, U., editors, *Software Architecture*, volume 8627 of *Lecture Notes in Computer Science*, pages 202–209. Springer International Publishing.

Kajko-Mattsson, M., Lewis, G. A., and Smith, D. B. (2008). Evolution and maintenance of soa-based systems at sas. In *Proc. Hawaii Int'l Conf. on Systems Science (HICSS)*, page 119. IEEE CS.

Kaminski, P., Litoiu, M., and Müller, H. (2006). A design technique for evolving web services. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '06. IBM Corp.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734.

Knoop, J., Rüthing, O., and Steffen, B. (1994). Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 147–158, New York, NY, USA. ACM.

Kwok, T., Nguyen, T., and Lam, L. (2008). A software as a service with multi-tenancy support for an electronic contract management application. In *Proc. Int. Conf. on Services Computing (SCC)*, pages 179–186. IEEE.

Laitinen, M. (1999). In Fayad, M., Schmidt, D., and Johnson, R., editors, *Object-Oriented Application Frameworks: Problems and Perspectives*, chapter 23, sidebar 9, pages 620–624. Wiley.

Lämmel, R., Pek, E., and Starek, J. (2011). Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, pages 1317–1324. ACM.

Lehman, M. M. and Belady, L. A. (1985). *Program Evolution: Processes of Software Change*. Academic Press.

Lewis, G. and Smith, D. (2008). Service-oriented architecture and its implications for software maintenance and evolution. In *Proceedings Frontiers of Software Maintenance*, pages 1–10. IEEE CS.

Li, J., Xiong, Y., Liu, X., and Zhang, L. (2013). How does web service API evolution affect clients? In *Int'l Conf. on Web Services (ICWS)*, pages 300–307. IEEE.

Lin, C. (2006). Method and system for nondisruptive deployment during upgrading of enterprise systems. US Patent 7,089,548.

Ly, P., Pedrinaci, C., and Domingue, J. (2012). Automated information extraction from web APIs documentation. In Wang, X., Cruz, I., Delis, A., and Huang, G., editors, *Web Information Systems Engineering (WISE)*, volume 7651 of *LNCS*, pages 497–511. Springer Berlin Heidelberg.

Mahbub, K. and Spanoudakis, G. (2004). A framework for requirents monitoring of service based systems. In *Proceedings of the 2Nd International Conference on Service Oriented Computing*, ICSOC '04, pages 84–93, New York, NY, USA. ACM.

Mahbub, K. and Spanoudakis, G. (2005). Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. *Proceedings of the International Conference on Web Services (ICWS)*, pages 257–265.

Maleshkova, M., Pedrinaci, C., and Domingue, J. (2009). Supporting the creation of semantic RESTful service descriptions. In *Proceedings of the 3rd International SMR2 2009 Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web, collocated with the 8th International Semantic Web Conference (ISWC)*. http://ceur-ws.org/Vol-525/.

Maleshkova, M., Pedrinaci, C., and Domingue, J. (2010). Investigating web apis on the world wide web. In *Proc. European Conf. on Web Services (ECOWS)*, pages 107–114. IEEE CS.

Marconi, A. and Pistore, M. (2009). Synthesis and composition of web services. In Bernardo, M., Padovani, L., and Zavattaro, G., editors, *Formal Methods for Web Services*, volume 5569 of *LNCS*, pages 89–157. Springer.

Marin, M., Deursen, A. V., and Moonen, L. (2007). Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Methodol.*, 17(1):3:1–3:37.

Martin, J., Arsanjani, A., Tarr, P., and Hailpern, B. (2003). Web services: Promises and compromises. *Queue*, 1(1):48–58.

Matthijssen, N., Zaidman, A., Storey, M.-A., Bull, I., and van Deursen, A. (2010). Connecting traces: Understanding client-server interactions in ajax applications. In *Proc. 18th Int. Conf. on Program Comprehension (ICPC)*, pages 216–225. IEEE CS.

McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of api stability and adoption in the android ecosystem. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*, pages 70–79. IEEE CS.

McIntosh, S., Adams, B., and Hassan, A. E. (2011). Using indexed sequence diagrams to recover the behaviour of ajax applications. In *Proc. of the 13th Int'l Symposium on Web Systems Evolution (WSE)*, pages 1–10.

Mileva, Y. M., Dallmeier, V., Burger, M., and Zeller, A. (2009). Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, pages 57–62. ACM.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44.

Momm, C., Malec, R., and Abeck, S. (2007). Towards a model-driven development of monitored processes. *Internationale Tagung Wirtschaftsinformatik (WI2007), Karlsruhe*.

Munson, J. C. and Elbaum, S. G. (1998). Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 24–33. IEEE CS.

Nasr, K. A., Gross, H.-G., and van Deursen, A. (2011). Realizing Service Migration in Industry - Lessons Learned. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*.

Natis, Y. V. (2003). Service-oriented architecture scenario. Website last visited November 30th, 2011.

Nolan, D. and Lang, D. T. (2014). Rest-based web services. In *XML and Web Technologies for Data Sciences with R*, Use R!, pages 339–379. Springer New York.

Pacione, M. J., Roper, M., and Wood, M. (2004). A novel software visualisation model to support software comprehension. In *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, pages 70–79. IEEE CS.

Pautasso, C. and Wilde, E. (2009). Why is the web loosely coupled? a multi-faceted metric for service design. In *Proc. Int'l World Wide Web Conf. (IW3C2)*, pages 911–920. ACM.

Pautasso, C. and Wilde, E. (2011). *REST: From Research to Practice*. Springer.

Pautasso, C., Zimmermann, O., and Leymann, F. (2008). Restful web services vs. "big" web services: Making the right architectural decision. In *Proc. Int'l Conf. on World Wide Web (WWW)*, pages 805–814. ACM.

Piel, É., González-Sanchez, A., Groß, H.-G., and van Gemund, A. J. C. (2011). Spectrum-based health monitoring for self-adaptive systems. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 99–108. IEEE.

Pistore, M. and Traverso, P. (2007). Assumption-based composition and monitoring of web services. In Baresi, L. and Di Nitto, E., editors, *Test and Analysis of Web Services*, pages 307–335. Springer.

Raemaekers, S., van Deursen, A., and Visser, J. (2012). Measuring software library stability through historical version analysis. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*, pages 378–387. IEEE CS.

Reps, T. W., Ball, T., Das, M., and Larus, J. R. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 432–449.

Robillard, M. P. and DeLine, R. (2011). A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732.

Romano, D. and Pinzger, M. (2012). Analyzing the evolution of web services using fine-grained changes. In *Proceedings of the 2012 IEEE 19th International Conference on Web Services*, ICWS '12, pages 392–399, Washington, DC, USA. IEEE Computer Society.

Schmerl, B. R., Garlan, D., Dwivedi, V., Bigrigg, M. W., and Carley, K. M. (2011). Sorascs: a case study in soa-based platform design for socio-cultural analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 643–652. ACM.

Sillito, J., Murphy, G. C., and De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451.

Sim, S. E., Easterbrook, S. M., and Holt, R. C. (2003). Using benchmarking to advance research: A challenge to software engineering. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 74–83. IEEE CS.

Sim, S. E. and Holt, R. C. (1998). The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 361–370. IEEE CS.

Subramanian, S., Inozemtseva, L., and Holmes, R. (2014). Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA. ACM.

Vinoski, S. (1997). Corba: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55.

Vinoski, S. (2008). Restful web services development checklist. *IEEE Internet Computing*, 12(6):96–95.

Wang, J. (1999). A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46.

Wang, S., Keivanloo, I., and Zou, Y. (2014). How do developers react to RESTful API evolution? In *Service-Oriented Computing*, volume 8831 of *LNCS*, pages 245–259. Springer.

Wang, Z. H., Guo, C. J., Gao, B., Sun, W., Zhang, Z., and An, W. H. (2008). A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *International Conference on e-Business Engineering (ICEBE)*, pages 94–101. IEEE.

White, L., Wilde, N., Reichherzer, T., El-Sheikh, E., Goehring, G., Baskin, A., Hartmann, B., and Manea, M. (2012). Understanding interoperable systems: Challenges for the maintenance of soa applications. pages 2199–2206. IEEE CS.

White, L. J., Reichherzer, T., Coffey, J., Wilde, N., and Simmons, S. (2013). Maintenance of service oriented architecture composite applications: static and dynamic support. *J. Softw. Maint. Evol.: Res. Pract.*, 25(1):97–109.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer.

Xing, Z. and Stroulia, E. (2007). Api-evolution support with diff-catchup. *Software Engineering, IEEE Transactions on*, 33(12):818–836.

Xu, W., Offutt, J., and Luo, J. (2005). Testing web services by XML perturbation. In *Proc. Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 10 pp.–266. IEEE CS.

Zaidman, A., Du Bois, B., and Demeyer, S. (2006). How webmining and coupling metrics improve early program comprehension. In *ICPC*, pages 74–78. IEEE CS.

Zaidman, A., Matthijssen, N., Storey, M.-A. D., and van Deursen, A. (2013). Understanding ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering*, 18(2):181–218.

Zaidman, A., Pinzger, M., and van Deursen, A. (2010). Software evolution. In Laplante, P. A., editor, *Encyclopedia of Software Engineering*, pages 1127–1137. Taylor & Francis.

Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.

Çalıklı, G. and Bener, A. (2013). Influence of confirmation biases of developers on software quality: an empirical study. *Software Quality Journal*, 21(2):377–416.

Titles in the IPA Dissertation Series since 2009

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and*

Distributed Services. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transforma-*

tions. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Fac-

ulty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

A.P. van der Meer. *Domain Specific Languages and their Type Systems*. Faculty of Mathematics and Computer Science, TU/e. 2014-10

B.N. Vasilescu. *Social Aspects of Collaboration in Online Software Communities*. Faculty of Mathematics and Computer Science, TU/e. 2014-11

F.D. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2014-12

N. Noroozi. *Improving Input-Output Conformance Testing Theories*. Faculty of Mathematics and Computer Science, TU/e. 2014-13

M. Helvensteijn. *Abstract Delta Modeling: Software Product Lines and Beyond*. Faculty of Mathematics and Natural Sciences, UL. 2014-14

P. Vullers. *Efficient Implementations of Attribute-based Credentials on Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2014-15

F.W. Takes. *Algorithms for Analyzing and Mining Real-World Graphs*. Faculty of Mathematics and Natural Sciences, UL. 2014-16

M.P. Schraagen. *Aspects of Record Linkage*. Faculty of Mathematics and Natural Sciences, UL. 2014-17

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06