Titre: Title:	Unifying Service Oriented Technologies for the Specification and Detection of their Antipatterns
Auteur: Author:	Francis Palma
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Palma, F. (2015). Unifying Service Oriented Technologies for the Specification and Detection of their Antipatterns [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1915/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1915/
Directeurs de recherche: Advisors:	Yann-Gaël Guéhéneuc, & Naouel Moha
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

UNIFYING SERVICE ORIENTED TECHNOLOGIES FOR THE SPECIFICATION AND
DETECTION OF THEIR ANTIPATTERNS

FRANCIS PALMA
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

UNIFYING SERVICE ORIENTED TECHNOLOGIES FOR THE SPECIFICATION AND
DETECTION OF THEIR ANTIPATTERNS

présentée par : PALMA Francis
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. MULLINS John, Ph. D., président
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et directeur de recherche
Mme MOHA Naouel, Ph. D., membre et codirectrice de recherche
M. DESMARAIS Michel C., Ph. D., membre
Mme STROULIA Eleni, Ph .D., membre externe

To...

Dr. Hermann Gmeiner,
the founder father of SOS Children's Villages International

Mr. Helmut Kutin,
the honorary president and ex-president of SOS Children's Villages International

SOS Children's Villages International
a loving home for every child and my forever home

ACKNOWLEDGEMENTS

PUBLICATIONS

Portions of the material in this dissertation have previously appeared in the following publications.

The list of international journal article(s) (peer reviewed) :

1. **Francis Palma**, Naouel Moha, and Yann-Gaël Guéhéneuc, *Unifying SBSs Technologies for the Specification and Detection of their Antipatterns.* IEEE Transactions on Software Engineering (**IEEE TSE**, **2015**) (**under review**)

2. **Francis Palma**, Mathieu Nayrolles, Naouel Moha, Yann-Gaël Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel, *SOA Antipatterns : An Approach for their Specification and Detection.* International Journal of Cooperative Information Systems (**IJCIS**, **2013**).

The list of international conference proceeding(s) (peer reviewed) :

1. **Francis Palma**, Javier Gonzalez-Huerta, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc : *Are RESTful APIs Well-designed ? Detection of their Linguistic (Anti)Patterns.* International Conference on Service Oriented Computing (**ICSOC**), Goa, India (November **2015**).

2. **Francis Palma**, Naouel Moha, and Yann-Gaël Guéhéneuc : *Specification and Detection of Business Process Antipatterns.* In Proceedings of the 6th International **MCETECH** Conference, May 12-15th, **2015**, Montreal, Canada. (May 2015).

3. **Francis Palma**, Le An, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc : *Investigating the Change-proneness of Service Patterns and Antipatterns. (Best Paper Award)*. In Proceedings of the 7th IEEE International Conference on Service Oriented Computing and Applications (**SOCA**), Matsue, Japan. (November **2014**).

4. **Francis Palma**, Johann Dubois, Naouel Moha, and Yann-Gaël Guéhéneuc : *Detection of REST Patterns and Antipatterns : A Heuristics-based Approach.* In Proceedings of the 12th International Conference on Service Oriented Computing (**ICSOC**), Paris, France. Springer (November **2014**).

5. **Francis Palma**, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc : *Specification and Detection of SOA Antipatterns in Web Services.* In Proceedings of the 8th European Conference on Software Architecture (**ECSA**), Vienna, Austria. Springer (August **2014**).

6. Naouel Moha, **Francis Palma**, Mathieu Nayrolles, Benjamin Joyen Conseil, Yann-Gaël Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel : *Specification and Detection of SOA Antipatterns. (Best Paper Award–Runner up)*. In Proceedings of the 10th International Conference on Service Oriented Computing (**ICSOC**), Shanghai, China. Springer (November **2012**).

The list of workshop, symposium, and tool demo paper(s) (peer reviewed) :

1. **Francis Palma** and Naouel Moha, *A Study on the Taxonomy of Service Antipatterns.* In the proceedings of 2nd on Patterns Promotion and Anti-patterns Prevention (**PPAP 2015**) co-located with 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (**SANER** 2015), Montréal, Canada.

2. **Francis Palma**, *Specification and Detection of SOA Antipatterns*, PhD Symposium, in conjunction, with 30th International Conference on Software Maintenance and Evolution (Victoria, Canada), **ICSME 2014**, September 28 - October 3, Victoria, Canada.

3. **Francis Palma**, Naouel Moha, and Yann-Gaël Guéhéneuc, *Detection of Process Antipatterns : A BPEL Perspective.* Workshop on Methodologies for Robustness Injection into Business Processes (**MRI-BP**), in conjunction with the 17th IEEE International EDOC Conference (EDOC 2013), "The Enterprise Computing Conference", 9–13 September **2013**, Vancouver, Canada.

4. **Francis Palma**, *Detection of SOA Antipatterns.* 8th PhD Symposium (Shanghai, China), in conjunction with **ICSOC 2012**, (10th International Conference on Service Oriented Computing), November 12-16, Shanghai, China, 2012.

5. Mathieu Nayrolles, **Francis Palma**, Naouel Moha and Yann-Gaël Guéhéneuc, *SODA : A Tool Support for the Detection of SOA Antipatterns.* ICSOC Demonstration Track, in conjunction with **ICSOC 2012**, (10th International Conference on Service Oriented Computing), November 12-16, Shanghai, China, 2012.

6. **Francis Palma**, Hadi Farzin, Yann-Gaël Guéhéneuc and Naouel Moha, *Recommendation System for Design Patterns in Software Development : An DPR Overview.* 3rd International Workshop on Recommendation Systems for Software Engineering (**RSSE 2012**), in conjunction with **ICSE** 2012, Zurich, Switzerland.

RÉSUMÉ

Les logiciels basés sur les services (SBSs) sont développés en utilisant l'Architecture Orientée Services (SOA), comme les SOAP Web services, Service Component Architecture (SCA) et REST. Pourtant, comme tous les autres systèmes complexes, les SBSs sont sujets aux changements. Les changements peuvent être fonctionnelles ou non fonctionnelles et peuvent apparaître à la conception ou a l'implémentation. Ces changements peuvent dégrader la qualité de la conception et la qualité de service (QoS) dans les SBS en introduisant des mauvaises solutions—*antipatrons de services*. La présence d'antipatron de services dans les SBSs peut entraver l'entretien futur et l'évolution de SBSs. L'évaluation de la qualité de la conception et de la qualité de service de SBSs via la détection des antipatrons de services peut faciliter leur maintenance et leur évolution. Bien qu'il existe quelques points communs entre les différentes technologies SBS, ils varient dans leurs (1) *building blocks*, (2) les styles de composition, (3) la méthodologie de développement et (4) les styles de communication ou d'interaction client ; ce qui posent des défis pour les analyser d'une manière unique. La littérature actuelle manque une approche unifiée pour évaluer la qualité de la conception et de la QoS SBS.

Pour répondre à ce besoin, cette thèse présente un méta-modèle unifiant trois technologies SBSs : Web services, SCA et REST. Nous utilisons ce méta-modèle, pour créer une approche unifiée, SODA (Service Oriented Détection for Antipatterns), soutenue par un framework, SOFA (Service Oriented Framework for Antipatterns), pour évaluer la conception et la QoS des SBSs. En utilisant l'approche SODA, nous définissons des règles de détection pour 31 antipatrons de service et 10 patrons de service indépendamment de leurs technologies. Basé sur ces règles, nous générons automatiquement (pour SCA et Web services) ou implémentons (pour REST) leurs algorithmes de detection. Nous appliquons et validons ces algorithmes en termes de precision et de rappel sur (1) deux systemes de SCA, (2) plus de 120 Web services et (3) un ensemble de 15 services REST largement utilisés, incluant Facebook, Twitter et YouTube.

Les résultats de détection fournissent des preuves de la présence d'antipatrons de services dans les SBS. Notre méthode de détection possède une précision et un rappel élevées et une performance de détection acceptable en termes de temps. Notre approche SODA et l'outil sous-jacent peuvent aider les praticiens à évaluer leur SBS, ce qui peut entraîner un SBS (1) avec une meilleure qualité de conception et un entretien facilité et (2) une QoS améliorée pour les utilisateurs finaux comparé à un SBS contenant les antipatrons de services.

ABSTRACT

Service-based Systems (SBSs) are developed on top of diverse emerging Service-Oriented Architecture (SOA) technologies and architectural choices, including SOAP Web services, SCA (Service Component Architecture), and REST. Yet, like any other complex systems, SBSs are subject to change. The changes can be functional or non-functional and can be at design or implementation-level. Such changes may degrade the quality of design and quality of service (QoS) of the services in SBSs by introducing poor solutions—*service antipatterns.* The presence of service antipatterns in SBSs may hinder the future maintenance and evolution of SBSs. Assessing the quality of design and QoS of SBSs through the detection of service antipatterns may ease their maintenance and evolution. With a few commonalities among various SBSs implementation technologies, they vary in their (1) building blocks, (2) composition styles, (3) development methodology, and (4) communication or client interaction styles; which pose challenges to analyse them in a unique manner. However, the current literature lacks a unified approach for evaluating the design quality and QoS of SBSs.

To address this need, this dissertation presents an abstraction unifying three SBSs technologies: Web services, SCA, and REST. Using this abstraction, it describes a unified approach, SODA (Service Oriented Detection for Antipatterns), supported by a framework, SOFA (Service Oriented Framework for Antipatterns), for assessing the design and QoS of SBSs. Using the SODA approach, we define detection rules for 31 service antipatterns and 10 service patterns independent of their technologies. Based on these rules, we automatically generate (for SCA and Web services) or implement (for REST) their detection algorithms. We apply and validate these algorithms in terms of precision and recall on (1) two SCA systems, (2) more than 120 SOAP Web services, and (3) a set of 15 widely-used RESTful services, including Facebook, Twitter, and YouTube.

The detection results provide evidence of the presence of service antipatterns in SBSs. The reported detection accuracy exhibits a high precision and recall and an acceptable detection performance in terms of detection time. Our SODA approach and the underlying tool can help practitioners to evaluate their SBSs, which may result in an SBS (1) with improved quality of design and easy maintenance and (2) with improved QoS for the end-users than an SBS with antipatterns.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

BNF	Backus Normal Form
BPEL4WS	Business Process Execution Language for Web Services
CBS	Component-based Systems
DSL	Domain Specific Language
GQM	Goal Question Metric
HATEOS	Hypermedia as the Engine of Application State
ISBSG	International Software Benchmarking Standards Group
JSON	JavaScript Object Notation
OMG	Object Management Group
OOP	Object-Oriented Programming
QoD	Quality of Design
QoS	Quality of Service
REST	REpresentational State Transfer
ROI	Return On Investment
SBSs	Service-based Systems
SCA	Service Component Architecture
SCDL	Service Component Definition Language
SDLC	Software Development Life Cycle
SDO	Service Data Objects
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SODA	Service Oriented Detection for Antipatterns
SODA-BP	Service Oriented Detection for Antipatterns in Business Processes
SOFA	Service Oriented Framework for Antipatterns
SOMAD	Service Oriented Mining for Antipattern Detection
SOP	Service-Oriented Programming
UDDI	Universal Description, Discovery, and Integration
URIs	Uniform Resource Identifiers
WSDL	Web Service Description Language
XML	eXtensible Markup Language

LIST OF APPENDICES

CHAPTER 1   INTRODUCTION

1.1   Research Context

Service-Oriented Architecture and Service-Oriented Programming: *Software engineering* is defined as "*the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software*" [ISO/IEC/IEEE (2010)]. The *technological knowledge* and *methods* applied for software development evolve regularly to develop more flexible, high-performance, and high-maintainable software systems. Therefore, the existing software development paradigms vary in facilitating the reusability and ease of maintenance and evolution for software.

One of the earliest software development paradigms is the *procedural development*—a simple but powerful development paradigm—which followed the sequential execution of a program. However, it has several drawbacks: (1) it is hard to relate real world concepts programmatically; (2) systems are difficult to maintain when the code grows; (3) data are publicly exposed, *i.e.*, not secured. To overcome those pitfalls, Object-Oriented Programming (OOP) was introduced with an increased software development productivity and reuse, with reduced software maintenance cost, and improved data security [Champeaux *et al.* (1993)]. Moreover, software performance and quality of software improved significantly.

After the Internet was invented, and industries required conducting their business and transactions remotely over the Web, a new software development paradigm was inevitable. Thus, a new programming paradigm—Service-Oriented Programming (SOP)—appeared for developing remotely accessible and platform independent autonomous functional entities— *services* [Papazoglou *et al.* (2003); Erl (2004); Turner *et al.* (2003)]. Compared to the procedural and OOP paradigms, SOP introduced an additional layer of software abstraction— the service layer. A benefit of having this additional layer of service abstraction is that it helps propagate business changes rapidly and can significantly reduce the maintenance efforts [Woods et Mattern (2006)]. A service-based system relies on services as its building blocks and on the Service-Oriented Architecture (SOA) as its underlying architecture [Erl (2005)].

Service-Oriented Architecture (SOA) [Papazoglou *et al.* (2003); Erl (2005); Heffner *et al.* (2007)] is a collection of principles and methodologies for designing and developing service-based systems (SBSs). SOA helps IT organisations to meet their business needs by composing loosely-coupled, platform-independent, reusable functional *services*, which encapsulate application and business logics [Erl (2005)]. The rapid growth of SOA is strongly supported by

the major software vendors, *e.g.*, IBM, Sun, and SAP, offering Service-Oriented middleware platforms, SBSs development and deployment tools with development environments.

SBSs are designed and developed applying SOA design patterns and principles [Erl (2009)] and using a number of different technologies and architectural styles, typically REpresentational State Transfer (REST, [Fielding (2000)]), Service Component Architecture (SCA, [Chappell (2007)]), and SOAP Web services [Alonso *et al.* (2003)]. In this dissertation, we refer these technologies as *SBSs technologies*. Google Maps, Amazon, eBay, PayPal, and FedEx are examples of industry-scale SBSs. Spanoudakis and Mahbub [Spanoudakis et Mahbub (2004)] defined 'SBSs' as composite systems, which are dynamically composed with autonomous Web services and whose composition is controlled by some composition processes. The definition of SBSs provided by Spanoudakis and Mahbub focuses only on the compositional aspect of SBSs. Therefore, this dissertation further generalises the definition:

"*SBSs are built on top of SOA design principles and are composed of autonomous services implemented with heterogeneous technologies as their building blocks.*" [Palma *et al.* (2013)].

Systems developed with these *SBSs technologies* exhibit high flexibility and agility, expedite rapid business changes, and more importantly, have high reusability [Erl (2007)]. However, services and SBSs are not exempt of some common software engineering challenges— maintenance and evolution. Maintenance and evolution take place when new or changed user requirements appear. More specifically, the maintenance and evolution can take place due to (1) *functional* changes, *i.e.*, changes at design and implementation-level; (2) *non-functional* changes, *i.e.*, changes in the execution contexts or in service-level agreements.

Maintenance and Evolution of SBSs: For any software systems including the SBSs, *maintainability* and *evolvability*—the two most important quality characteristics—represent the ability of a software system to be modified and evolved at ease [Khurana (2007); ISO/IEC (2011)]. Among the several iterative and long-running development phases, software maintenance and evolution phase is the most resource-intensive phase (*i.e.*, consumes more than 70% of overall project resources) and involves continuous *corrective*, *preventive*, *adaptive*, and *perfective* activities performed on existing software components [Lientz et Swanson (1980); Khurana (2007); Pigoski (1996); Rombach *et al.* (1992); Holgeid *et al.* (2000); Coleman *et al.* (1994); Zuse (1997)]. These studies showed the importance of having well-maintainable software systems. Thus, developing a software that is difficult to maintain can lead to *halting* the project in the long-run due to the increased maintenance cost and effort [Khurana (2007)]. Boehm *et al.* [Boehm *et al.* (1976)] suggested that the main benefit of having a software system of good quality is its cost-effective maintenance and evolution. This dissertation

considers software quality as the combination of (1) *quality of design* that facilitates future maintenance and evolution and (2) end user's satisfaction in terms of *quality of service.*

From the SOA perspective, developing highly-maintainable services and SBSs of good quality is crucial due to their continuous-evolving nature. Due to the rapid changes in (growing and global) business needs, SBSs must adapt to client requirements by changing their underlying business logics and rules with the shortest delay [Newcomer et Lomow (2004)]. A highly maintainable *service* can facilitate better adaptation. Moreover, firm time constraints, *i.e.*, shorter software product release times, might lead to producing less-maintainable systems. SBSs operate in a highly dynamic execution environment, and, thus, SBSs developers need to take an extra care of the desired functional (*i.e.*, system's functional capabilities) and non-functional behaviors (*i.e.*, throughput, latency, availability, etc.), after performing all kinds of maintenance activities.

More than 50% of the efforts during the maintenance activities are typically spent on the perfective maintenance [ISBSG (2005)]. Perfective maintenance activities involve modifications of a system to accommodate new user requirements or change the existing functionalities. Perfective maintenance is highly relevant to the service-oriented development, which, in general, incorporates a large number of business rules integrated within a business process. A business process may involve a number of partner remote services and plays as a media between the end-users and the services providing desired functionalities. Such complex business processes are the most unstable part of SBSs [Wan-Kadir et Loucopoulos (2004)]. This instability of SBSs causes them going under frequent perfective maintenance to seamlessly cope with the changed business requirements. Thus, developing SBSs exhibiting high maintainability is one of the key factors towards a successful adoption of SOA.

Antipatterns to Assess Software Quality: During the maintenance and evolution of SBSs the various *functional* and *non-functional* changes may degrade the quality of design and implementation and the quality of service (QoS) of SBSs and may cause the appearance of poor solutions to recurring design problems—*service antipatterns.* These poor solutions, *i.e.*, service antipatterns, are documented in the literature as the common bad design practices considered during the design and development of services and SBSs. In other words, service antipatterns typically capture some aspects of design inelegance, component complexity, lack of cohesion, and–or high coupling among a group of services (or components). Assessing the design quality and QoS of any systems by detecting antipatterns may help to ease their maintenance and evolution [Bennett et Rajlich (2000)].

Multi Service and *Tiny Service* are examples of two typical service antipatterns [Dudney *et al.* (2003)] in SBSs. *Multi Service* represents a service that implements a long list of

operations varying in business abstractions. A service implemented as a *Multi Service* is not easily reusable and exhibits a low cohesion among its operations. Being overloaded by many different client requests, the *Multi Service* might become frequently inaccessible to its end-users. This excessive load, however, can be reduced by deploying multiple instances of the service, which is not an inexpensive resolution. On the contrary, *Tiny Service*, a small service with very few operations, implements only a part of an abstraction, thus requiring several other tightly coupled services to complete an abstraction, increasing the design and development complexity. Researchers suggest that *Tiny Service* is the cause of a number of SOA failures [Král et Žemlička (2008)]. Detecting such service antipatterns requires the rigorous assessment of the design and QoS of SBSs.

However, such assessment of the design and QoS of SBSs is challenging because (1) service antipatterns do not have a formal specification, *i.e.*, they are only textually described in the current literature and (2) each kind of SBSs technology: Web services, SCA, and REST, includes common and differing concepts. Diverse SBSs technologies vary in their (1) building blocks, (2) composition styles, (3) development methodology, and (4) communication or client interaction styles, which pose some challenges to analyse them in a unique manner.

Assessing the design of an SBS involves its static analysis while assessing its QoS involves measuring various dynamic aspects (*i.e.*, performance and availability). Detecting service antipatterns in SBSs is one of the effective means of analysing and assessing the design and QoS criteria of SBSs. The detection of service antipatterns within SBSs is also a part of maintenance activities: (1) to improve their quality of design and QoS and (2) to decrease the effort and cost of their future maintenance and evolution.

1.2 Problem Statement and Thesis

Despite their importance to assess the design and QoS of SBSs, service antipatterns have not received much attention in the literature. Till today, a little research effort was dedicated to considering how service antipatterns in SBSs impact their maintainability. More importantly, the detection mechanisms of service antipatterns in various SBSs technologies are still immature. Pressman in his book *Software Engineering: A Practitioner's Approach* [Pressman (2010)] suggested that the detection and correction of design problems (in this dissertation the service antipatterns) in any software systems can reduce the cost and effort of their future maintenance and evolution activities. He also argued that software systems with least possible design problems are easier to implement and cope with the change requests [Pressman (2010)].

To support software engineers and developers in the assessment of software quality, a number of quality models were proposed in the literature, for example [Boehm *et al.* (1976); Kitchenham (1987); Dromey (1995)]. However, the existing literature on the quality assessment is mostly subjective, incomplete and–or not strong enough to gain wide acceptance [Mattsson *et al.* (2006)]. Moreover, due to the unique characteristics of SOA paradigm described in Section 2.2, the existing procedural and OO approaches are not applicable for the identification of bad design practices in SBSs [Perepletchikov *et al.* (2005)]. Therefore, this dissertation specifies, detects, and empirically evaluates a suite of service antipatterns. These service antipatterns provide a means to assess the structural properties of service interfaces and behavioral aspects of services while invoking them. Moreover, in the literature, there exists no formal specification of service antipatterns to facilitate their detection.

We identified the four following potential problems from the literature on the detection of service antipatterns in SBSs:

– **Problem 1. No unified abstraction of various SBSs technologies:** To detect service antipatterns in various SBSs technologies (*e.g.*, SCA, Web services, and REST), we need a unified abstraction, first, to understand those technologies well, and, second, to apply generic approaches to specify and detect service antipatterns. Combining various technologies brings potential challenges because despite their commonalities, these technologies have many architectural differences and they differ how clients consume services.

– **Problem 2. No specification of service antipatterns:** To detect service antipatterns, we must specify them in a machine-processable and human-understandable format. Without proper specifications, service antipatterns are ambiguous and cannot be detected automatically. The present service antipatterns literature is still in textual format, which is hard to handle and use.

– **Problem 3. No dedicated unified approach and framework for the detection of service antipatterns:** SBSs operate in the Internet-based dynamic environment. The complex execution context of SBSs, *i.e.*, scenarios based on choreographed services and their dynamic nature, *i.e.*, physical availability and service-level agreements between services and clients make the analysis of services and detection of service antipatterns challenging. There are numerous contributions in the literature for the detection of object-oriented (OO) antipatterns. Yet, there is no consolidated method and technique for such detection in SBSs to assess their QoS and design. There is no unified framework for the detection of various service antipatterns.

– **Problem 4. No empirical evidence on the impact of service antipatterns**

on service-based systems: An empirical study aimed at quantifying the impact of service patterns and antipatterns on the maintenance and evolution of service-based systems can provide a concrete evidence. No study quantitatively assesses the impact of service patterns and antipatterns on specific quality attributes. Based on such evidence, the developers and engineers would be aware of introducing antipatterns, intentionally or either way. There are some works in the Object-Oriented domain [Khomh *et al.* (2009, 2012b); Romano *et al.* (2012)]. However, no such strong evidence exists in the service-oriented literature.

In the literature, no solution exists dealing with the above four gaps. Thus, with the problems stated above, we formulate our thesis statement as following:

> **Thesis:** "A unified approach to assessing the design and quality of service (QoS) of SBSs, supported by a framework for specifying and detecting service antipatterns, can facilitate the maintenance and evolution of SBSs, with the conjecture that service antipatterns may degrade the design and QoS, and hinder the future maintenance and evolution of SBSs."

To support the above thesis statement, we answer the two following research questions:

RQ$_1$ – *Can we efficiently specify and detect service antipatterns in different development technologies and architectural styles of service-based systems in terms of detection accuracy and performance?*

By answering RQ$_1$, we show that we can specify any service antipatterns and detect them with high accuracy and low performance overhead in terms of detection time. Moreover, the proposed detection framework must be extensible for new service antipatterns and new SBSs technologies.

A positive answer to **RQ**$_1$ solves **Problem 1**, **Problem 2**, and **Problem 3** identified in the literature:

Solution to Problem 1: A unified abstraction of various SBSs technologies for their better understanding and for the specification and detection of service antipatterns in various SBSs technologies.

Solution to Problem 2: Specification of service antipatterns in an automatable format for their automatic detection.

Solution to Problem 3: A unified approach based on a framework that can facilitate the detection of service antipatterns in SBSs regardless of their underlying technologies and can ensure the reusability and extensibility of the proposed framework.

RQ$_2$ – *What are the impact of service antipatterns and patterns on the maintenance and evolution of service-based systems?*

By answering RQ$_2$, we empirically show that services involved in antipatterns are more change-prone, *i.e.*, antipatterns require more maintenance effort. In contrast, services involved in patterns are less change-prone than those not involved in any patterns, *i.e.*, patterns require less maintenance effort. To conclude with RQ$_2$, we investigate the following:

1. **_RQ_**$_{2.1}$ – *What is the relation between service antipatterns and change-proneness?*
   Finding: The total number of source code changes and code churns performed during the maintenance and evolution of services involved in antipatterns is higher than the total number of source code changes and code churns performed on other services—the difference is statistically significant.

2. **_RQ_**$_{2.2}$ – *What is the relation between particular kinds of service antipatterns and change-proneness?*
   Finding: Services found to be involved in *God Component*, *Multi service*, and *Service Chain* antipatterns are more change-prone than services involved in other antipatterns—the difference is statistically significant.

3. **RQ**$_{2.3}$ – *What is the relation between service patterns and change-proneness?*
   Finding: The total number of source code changes and code churns performed during the maintenance and evolution of services involved in patterns is less than the total number of source code changes and code churns performed in other services—the difference is not statistically significant.

An answer to **RQ**$_2$ with a statistically significant evidence solves the **Problem 4** identified in the literature:

Solution to Problem 4: A strong evidence of the relationship between service antipatterns in SBSs and their maintainability, *i.e., change-proneness*. Such strong evidence on the negative impact of service antipatterns on maintainability will raise awareness among SBSs designers and developers.

1.3   The Unified Abstraction and SODA Approach

To answer RQ$_1$, which concerns the efficient specification and detection of service antipatterns in SBSs regardless of SBSs technologies, we propose a unified abstraction of different SBSs technologies. Later, using this unified abstraction, we propose and follow a unified approach, SODA (Service Oriented Detection for Antipatterns), to specify antipatterns at higher-level of abstraction and perform their detection on a number of SBSs developed using diverse SBSs implementation technologies, namely Web services, SCA, and REST.

Figure 1.1 presents an overview of our proposed SODA approach with the three steps. Our novel and innovative SODA approach, relies on a unified framework, SOFA (Service Oriented Framework for Antipatterns), for the specification and automatic detection of service antipatterns in SBSs. Our SOFA framework facilitates the static and dynamic analyses of SBSs where the static analysis concerns quantifying design-related structural properties and the dynamic analysis refers to quantifying runtime properties while executing an SBS. The SOFA framework is also capable of performing the syntactic and semantic analyses of services interfaces based on WordNet [Miller (1995)] and Stanford CoreNLP [Manning *et al.* (2014)]. Using the SODA, we perform a series of experiments intended for the empirical validation of the presence of service antipatterns in SBSs.

The three steps of the SODA approach include:

– **Step 1:** Our proposed unified SODA approach relies upon a domain specific language (DSL) to specify service antipatterns in terms of metrics, both static and dynamic. The DSL is defined after a thorough domain analysis of service antipatterns from the literature and provides the means to specify service antipatterns at higher-level of abstraction;

– **Step 2:** With the help of DSL and SOFA, we automatically generate (for SCA and Web services) or implement (for REST) antipatterns detection algorithms from the rules of service antipatterns;

– **Step 3:** Finally, we apply them on a target SBS for the detection of service antipatterns and report suspicious services.
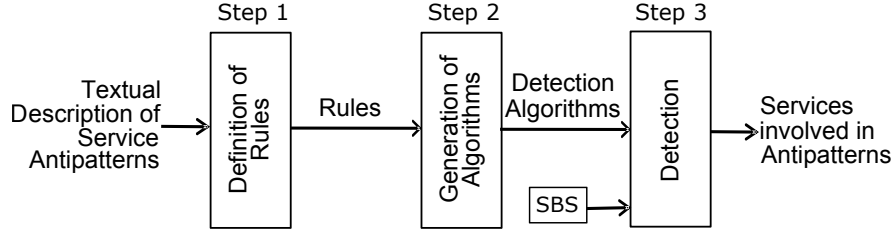
Figure 1.1: The Overview of the Proposed Unified SODA Approach.

We show the usefulness of SODA by defining rules for 31 service antipatterns from SCA, REST, and Web services and by performing their detection. We validate the detection results in terms of precision, recall, and F_1-measure on:

1. FraSCAti, the largest open-source SCA system with 130 services and *Home-Automation*, a demo SCA application with 13 services;

2. more than 120 Web services collected from a Web services search engine: `www.programmableweb.com`;

3. 15 well-known RESTful APIs, including Facebook, Twitter, Dropbox, and YouTube.

We show that SODA allows the specification and detection of a representative set of service antipatterns of different types with an average precision and recall of more than 75%.

1.4 Contributions

This dissertation makes a contribution to the field of SOA by presenting a dedicated approach for specifying and detecting bad design practices, *i.e., service antipatterns*, in SBSs.

The main contributions of this dissertation are:

1. A unified abstraction combining different SBSs technologies and architectural styles showing the differences and commonalities among them (**solving Problem 1**);

2. On top of the unified abstraction, a service DSL to specify service antipatterns regardless of SBSs technologies with higher-level of abstractions (**solving Problem 2**);

3. Using the proposed unified abstraction and the service DSL, a unified SODA approach for the specification and automatic detection of service antipatterns in SBSs technologies (**solving Problem 3**);

4. An extensive validation of the SODA approach using precision, recall, and F_1-measure on the largest SCA system—FraSCAti, more than 120 Web services, and 15 well-known REST services (**solving Problem 3**);

5. An empirical evidence on the impact of service antipatterns on the maintenance and evolution of SBSs, in particular on SCA systems (**solving Problem 4**).

However, this dissertation contributes towards the detection of service antipatterns and does not consider removing or refactoring service antipatterns. Demonstrating the elimination of service antipatterns requires significant research effort investigating the refactoring methodology from the literature based on the best practices for each service antipattern. We consider the refactoring of service antipatterns as one of our future works.

1.4.1 Other Related Contributions

Detection of Process Antipatterns: As an additional contribution, we also performed detection of business process antipatterns in BPEL4WS business processes [Curbera *et al.* (2006); Alves et et al. (2007)]. BPEL4WS (Business Process Execution Language for Web Services) is a *de facto* business process definition language in the industry built on top of the Web services model. However, business processes may evolve, *i.e.*, changes may take place (1) by modifying the existing tasks and–or adding new tasks or elements (2) by modifying the flow in the processes. This evolution of business processes may deteriorate their designs over time and introduce poor but recurring solutions to process design problems—*process antipatterns*. Process antipatterns describe common design problems in SBPs that may hinder their maintenance and evolution and result in poor quality of design (QoD) [Koehler et Vanhatalo (2007)]. Therefore, for SBPs, the automatic detection of such antipatterns is an important activity by assessing their design (1) to ease their maintenance and evolution and (2) to improve their QoD. We proposed SODA-BP (Service Oriented Detection for Antipatterns in Business Processes) supported by our underlying framework, SOFA, to specify and detect process antipatterns. Please refer to the full paper in [Palma *et al.* (2015)] for the detail description and validation of our SODA-BP approach.

Taxonomy of Service Antipatterns: A better understanding of service antipatterns is a must prerequisite to perform their detection. The study in [Palma et Moha (2015)] presents a taxonomy of service antipatterns in Web services and SCA (Service Component Architecture), the two state-of-the-art SBSs implementation technologies. The presented taxonomy will facilitate engineers their understanding on service antipatterns. Other substantial benefits of the presented taxonomy include: (1) assisting in the specification and detection of service antipatterns, (2) revealing the relationships among various groups of service antipatterns, (3)

grouping together antipatterns that are fundamentally related, and (4) providing an overview of various system-level design problems ensemble.

1.5   Organisation of the Dissertation

The remainder of this dissertation provides the following content:

Chapter 2: Background (P. 13) provides a background on Service Oriented Architecture and on the SBSs technologies we studied in this dissertation. It also compares among those technologies from the architectural point of view. Finally, it compares services antipatterns and classify them to show their distribution among technologies.

Chapter 3: Literature Review (P. 25) performs a literature review on the existing contributions in both Object- and Service-Oriented domain. Finally, it identifies the gaps in the current literature.

Chapter 4: Abstraction of SBSs Technologies (P. 34) presents the proposed unified abstraction and the meta-abstraction that we use for the specification of service antipatterns across various SBSs technologies.

Chapter 5: Specification and Detection of Service Antipatterns (P. 40) presents our proposed unified approach, SODA, relying on the unified abstraction, for the specification of service antipatterns. It also presents the unified framework, SOFA that is used by SODA for the automatic detection of service antipatterns.

Chapter 6: Validation (P. 54) presents the results of the empirical validation of SODA approach and discusses the results in detail for each antipattern.

Chapter 7: An Impact Study Of Service Antipatterns (P. 90) explores the impact of service patterns and antipatterns on SBSs change-proneness, and provides quantitative evidence of the negative impact of antipatterns on services change-proneness on an SCA system.

Chapter 8: Conclusion (P. 110) presents the conclusion of this dissertation and outlines some directions of future research.

Appendix A: Antipatterns in SBSs Technologies (P. 127) presents the brief descriptions of all service antipatterns and their rule cards or heuristics studied in this dissertation.

Appendix B: Transformation of REST Heuristics to Rule Cards: (P. 146) presents the transformation of five REST heuristics into their corresponding rule cards—using those rule cards we can automate the generation of their detection algorithms.

Appendix C: Design Overview of FraSCAti and Home-Automation (P. 149) presents the overall design for the two SCA systems *FraSCAti* and *Home-Automation* analysed in this dissertation for the detection of service antipatterns in SCA.

Appendix D: List of SOAP Web services (P. 153) presents the list of 123 service interfaces analysed in this dissertation for the detection of service antipatterns in SOAP Web services.

CHAPTER 2   BACKGROUND

2.1   Chapter Overview

This chapter provides a short introduction to Service-Oriented Architecture (SOA) and its various key concepts. Section 2.3 briefly describes the three Service-based Systems (SBSs) implementation technologies, namely SOAP Web services, REST, and Service Component Architecture (SCA). We provide a detailed comparison among those three SBSs implementation technologies in Section 2.4. We compare those technologies to show their differences and commonalities in various aspects. Finally, in Section 2.5, we show a classification of service antipatterns using a Venn diagram. The classification shows that different SBSs implementation technologies may share common service antipatterns, and there are antipatterns that are technology-specific. This classification is important because antipatterns that exist across technologies can be detected using similar detection algorithms whereas technology-specific antipatterns might require dedicated detection algorithms.

In the next section, we briefly discuss SOA and its various key concepts.

2.2   Service-Oriented Architecture

Primarily, a Service-Oriented Architecture (SOA), as shown in Figure 2.1, considers these four elements as its key: (1) an application front-end, (2) a set of services, (3) a service repository holding services specifications, which facilitates searching for services, and (4) a service bus that provides a mechanism to the services for their interaction [Rosen *et al.* (2008)]. Each service has a contract defining its capabilities. A service implements at least one interface, which lists and exposes service's functional capabilities. Overall, a service implements a business abstraction. In the following, we identify and discuss the four key concepts in SOA-based systems.

A Service: A service in a service-oriented architecture (SOA) is an (entirely) autonomous functional entity and several autonomous services can be composed to achieve higher level business goals [Erl (2005)]. Services in the context of SOA hold two main notions: (1) they perform business-related tasks and are not visible elements and (2) services are black-box entities and only expose their functionalities while hiding their implementation details. A service has end-point(s) through which the prospective clients will connect to it. An end-point holds a set of related operations. Each end-point compulsorily defines a binding type,
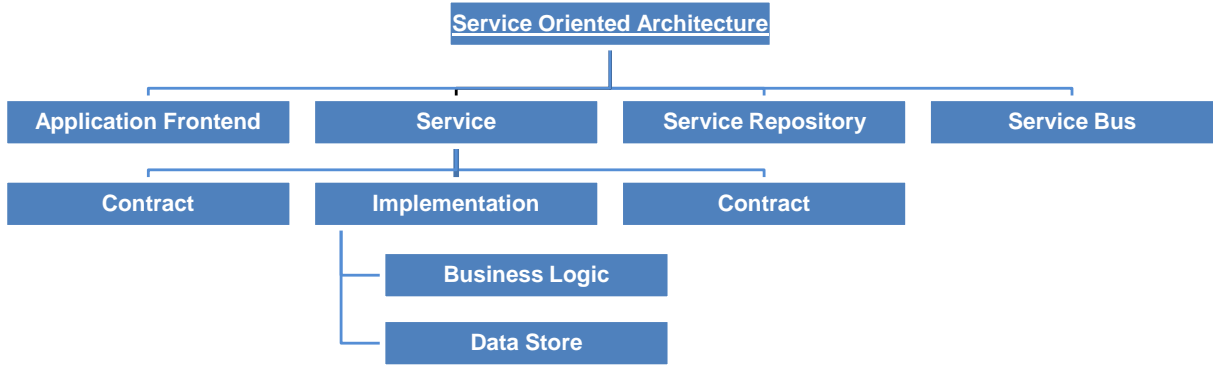
Figure 2.1: The Four Key Elements in SOA and Their Sub-elements.

i.e., how a client will communicate with the service and a physical address, *i.e.*, where the end-point is hosted.

Client-Server Model: As depicted in Figure 2.2, the SOA-based systems rely on the client-server communication model and their communications rely on SOAP messages or HTTP requests/responses. In general, the client can be an application front-end or a Web browser. The server holds the service(s) and manages and serves the client's requests. The details on SOAP messages and HTTP requests/responses and their formats are not discussed in this dissertation (more on SOAP and HTTP requests/responses can be read from here [Daigneau (2011)]). However, the services' interfaces, provided and exposed in a registry by a service provider is our subject of interest in this dissertation. We briefly discuss on WSDL interfaces.
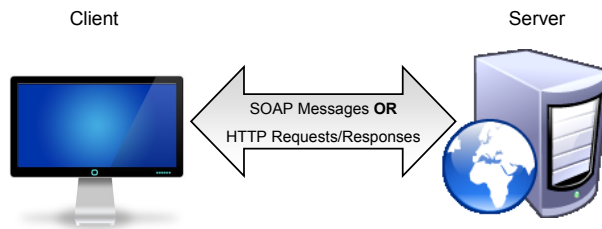
Figure 2.2: The Simple Client-Server Model.

WSDL Interface: Once a service provider creates a service, he must describe the service using a specification language and provides the physical location (*i.e.*, end-point) of the service for potential clients to connect and consume functionalities provided by the service.

W3C defines a service end-point as "*an association between a fully specified interface binding and a network address, specified by a URI that may be used to communicate with an instance of a Web service*" [Christensen *et al.* (2011)].

WSDL (Web Service Description Language) specifications are written in XML format and are published over the network, *i.e.*, the Internet. WSDL contains a set of end-points associated with a set of messages, *i.e.*, exchanged data and operations, which are bound to a binding protocol. Thus, a WSDL specification has three main constituents: (1) *definitions* of data types and–or messages, (2) *operations* of four different types, namely *one-way request*, *request/response*, *one-way response*, and *notification*, and (3) *service bindings* to connect a service's port by the clients.

UDDI Registry: UDDI (Universal Description, Discovery, and Integration), a platform-independent and XML-based registry, plays a major role in helping clients to search for services already published over the Internet. The UDDI is formed with three components: (1) *white pages* containing providers' addresses and contacts, (2) *yellow pages* providing industry-standard services categorisations, and (3) *green pages* containing technical details on services, *i.e.*, their functional capabilities and communications protocols. UDDI relies on SOAP messages to interact with clients.

Finally, if we combine the above individual SOA key concepts, it works together to fulfill clients business requirements from searching for an appropriate service to an actual interaction with the service, as shown in Figure 2.3. The steps from service's search to its consumption include:
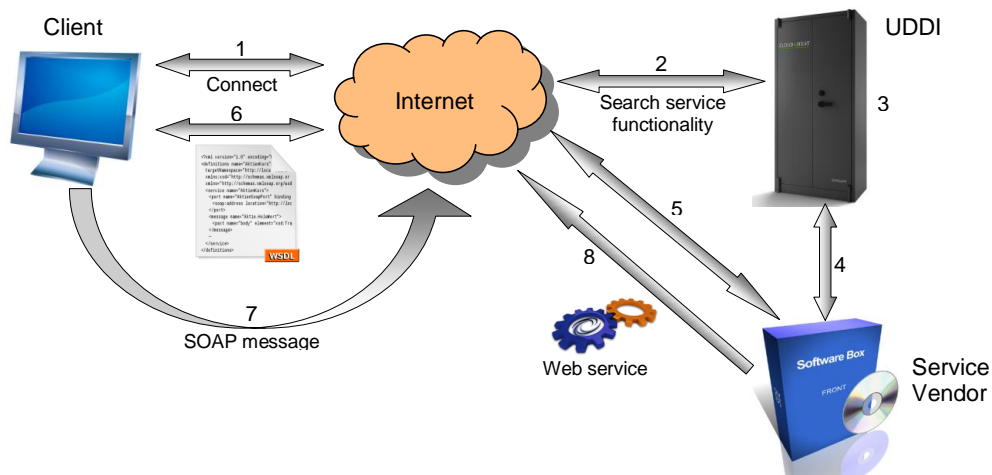
Figure 2.3: A Typical SOA Service Interaction Scenario.

Step 1: A client requires a Web service and, therefore, the client searches for a directory;

Step 2: The client connects to a UDDI directory to search for relevant services;

Step 3: The client and UDDI directory determine if the desired service is available;

Step 4: The UDDI directory contacts the service provider for the service availability;

Step 5: The service provider responds with a WSDL document if the service is available;

Step 6: The client prepares to consume the service, for example, by creating a client stub;

Step 7: The client interacts with the actual service using the SOAP or HTTP messages;

Step 8: The service responds with the client's business data to be processed by the client;

However, the UDDI registries and the underlying SOAP messaging protocol are getting desolated. Because, at present, the UDDI registries are not well-maintained and are not complete. Moreover, the SOAP messaging protocol is losing its popularity due to its verbosity. Marshalling and unmarshalling SOAP messages are computationally expensive [Daigneau (2011); Weerawarana *et al.* (2005)].

In Figure 2.3, the service vendors might implement services using different service implementation technologies. In the next section, we briefly highlight various implementation technologies for Service-based Systems (SBSs).

2.3   Different SBSs Technologies

This section briefly introduces the three SBSs technologies considered in this dissertation: Service Component Architecture (SCA) [Chappell (2007)], SOAP Web services [Alonso *et al.* (2003)], and REpresentational State Transfer (REST) [Fielding (2000)]. Later, a detailed comparison among them follows.

2.3.1   Web Services

W3C defines a Web service as "*a software system designed to support interoperable machine-to-machine interaction over a network*". Developers rely on Web services as the remote interfaces for their software systems. The two main categories of Web service are (1) SOAP Web services and (2) RESTful Web services.

SOAP Web Services:

Web services rely on an XML-based messaging protocol SOAP (Simple Object Access Protocol) [Alonso *et al.* (2003)]. Such Web services operate using customised operations and

the communications between clients and Web services are based on standards: (1) XML (eXtensible Markup Language) as the service data format, (2) HTTP as the transport protocol, (3) SOAP as the reliable and secured messaging protocol, (4) UDDI (Universal Description, Discovery, and Integration) as the service discovery mechanism, and, finally, (5) WSDL (Web Services Description Language) as the formal service contract.

RESTful Web Services:

REST allows resource-centric remote services [Fielding (2000)]. A Web service created based on the REST architecture can be referred as RESTful Web service. Strictly speaking, RESTful Web services must adhere to the rules and standards of HTTP. Unlike Web services, which operate using customised operations, REST services use standard HTTP operations, *e.g.*, GET, POST, PUT, and DELETE, to access and manipulate resources. It is a matter of great importance selecting the appropriate HTTP method for a certain context. REST architecture has an increased data transport efficiency and reduced data handling complexity, which come from its light-weight design and simple usage scheme. Among many others, the unique characteristics of REST architectures are: (1) the explicit use of HTTP methods, (2) the statelessness and cacheability, thus scalability, (3) the exposure of directory-like URIs (Uniform Resource Identifiers), (4) the ability to transfer data in many Web formats including XML and JSON (JavaScript Object Notation), *a.k.a.*, the media-types or MIME types. The lack of proper knowledge on REST can lead to the creation of SOAP-REST hybrid services.

In the past years, SOAP Web services were strongly adopted in the industry. In the recent years, many large companies are moving towards using RESTful services. In 2011, a study by ProgrammableWeb on the types of services used in the industry (among its 3,200 listed Web APIs) showed that more than 70% of listed Web APIs are currently RESTful whereas around 17% services rely on SOAP-based protocols. This clearly shows the present dominance of REST over SOAP. Figure 2.4 shows the type distribution of Web APIs (from the source: programmableweb.com).

2.3.2 Service Component Architecture

SCA is a software technology that provides a model to compose applications on top of SOA design principles [Chappell (2007)]. SCA provides a complete model for the service construction, assembly, and deployment. The composition, *a.k.a.*, *SCA composite*, is described using a standard XML-based language, SCDL (Service Component Definition Language) where a set of related *SCA components* are orchestrated. The components provide the actual desired business functionalities in the form of *services*. SCA defines a technology-agnostic
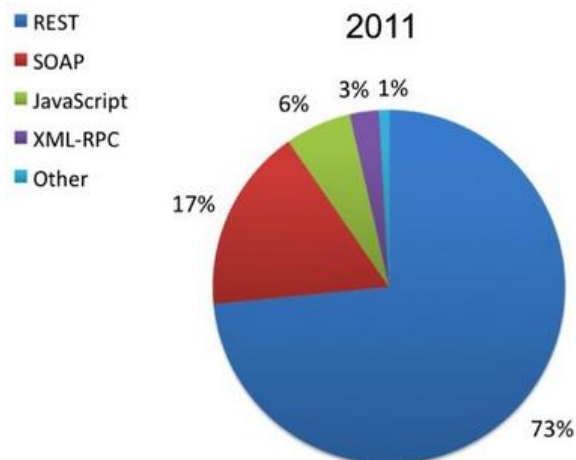
Figure 2.4: Type Distribution of Web APIs Protocols and Styles.

model for composing diverse interface definition languages (WSDL or Java), implementation languages and frameworks (Java, BPEL, C/C++, Spring, or OSGi), and bindings (SOAP, JMS, or REST).

The next section compares the above three SOA implementation technologies in detail.

2.4 Comparison among Technologies

Differences at architectural, design, and implementation-level exist among the above SBSs technologies, summarised in Table 2.1. An extensive review of the literature helped us to identify and classify the various technology-specific properties highlighted in Table 2.1. Hence, we must consider such varying properties when analysing SBSs developed relying on them.

The next two sections discuss the basic differences among three SBSs technologies in two aspects: their core design elements and service consumption styles.

Table 2.1: Non-trivial Architectural Differences among SCA, Web services, and REST.

Criteria	Web services	SCA	REST
Cacheability	no	no	cacheable
Contract design	contract first/last	contract last	contract-less
Dynamic configuration management	no	yes	no
Dynamic deployment	no	yes	no
Error handling	no	built-in	built-in
Message encoding	yes	yes	no
Messaging support	yes	within domain vendor-specific	no
Policy	WS-Policy	SCA policy framework	no standard
Operations invoking protocol	*SOAP*	*SOAP, JMS, RMI, so on*	*HTTP*
Reliability	WS-Reliability	non-standard	no
Representation of information	*XML-standard*	*XML-standard*	*JSON, XML, MIME, so on*
Security	WS-Security	SCA security policy	HTTP, SSL
Standards based	yes	yes	no
Statelessness	mostly stateful	by default stateless	completely stateless
Transactions	WS-AtomicTransaction	WS-AtomicTransaction	no standard
Transport protocol	HTTP, TCP, SMTP, JMS, etc.	same as SOAP	HTTP
Verbosity	more	more	less
Service composition	*WS-BPEL*	*SCDL*	*mashups*
Service/resource identification	*WS-Addressing*	*no*	*URI*
Core design elements	*service*	*component*	*resource*
Focus	accessing named operations	accessing components as service units	accessing named resources
Human intelligible payload	no	no	yes
Hypermedia/hyperlinking	*no support*	*no support*	*natural support*
Interface	different interfaces for services	different interfaces for components	uniform interface for resources
Service discovery	UDDI registries	not applicable	no standard
Service invocation	*through calling RPC method*	*same as SOAP*	*via URL path*
Standardised interface definition	*Web Services Description Language*	*Service Component Definition Language*	*no*
Interface exposure	public	neither	neither
Method callability	exposed as remotely callable operation	no	no
Specification	JAX-WS	SCA-J	JAX-RS
Written documentation	*no dependency*	*no dependency*	*highly dependent*

2.4.1 Core Design Elements

A first major difference among the three SBSs technologies is in their core design elements. SCA relies on *component* as its building block that provides a specific service and implements at least one interface [Chappell (2007)]. The SCA components communicate among themselves by passing the data as service data objects (SDO) and can be composed to achieve higher-level business goals. A collection of related components are specified in an SCA composite to achieve a higher-level business goal.

A *service* in Web services is operation-centric and exposes an arbitrary set of customised operations. Clients can search their desired customised interfaces. One use of such services is by orchestrating using a well-defined structured language, BPEL4WS [1]. The communications in Web services are verbose, *i.e.*, the message encoding and decoding are computationally expensive [Kumar (2004); Papazoglou (2008)].

REST relies on resources that include from one single data (*e.g.*, name, salary) to a file (*e.g.*, JPEG or PDF). Resources are identified using URIs (Uniform Resource Identifiers) and are accessible via standard HTTP methods. Thus, RESTful services rely on a uniform interface with common HTTP methods that can be applied on any resources. A resource has several essential properties including a location and a representation. REST resources can be a collection of other resources.
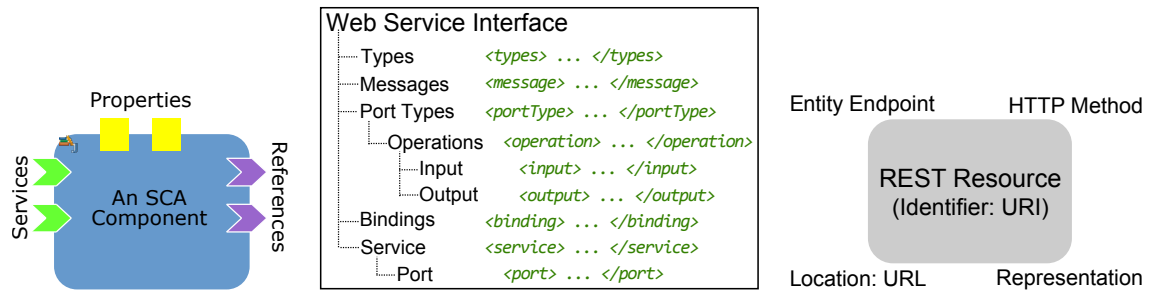
Figure 2.5: Building Blocks of the Three SBSs Implementation Technologies.

Figure 2.5 shows the building blocks of the three SBSs implementation technologies, namely a *component* for SCA, a *service* interface for SOAP Web services, and a REST *resource* with its different properties.

---

1. www.oasis-open.org/committees/wsbpel/

2.4.2 Service Consumption Styles

Differences among the three SBSs technologies from the service consumption view-point include:

1. Web services have publicly discoverable WSDL-based contracts. SCA systems have SCDL-based specifications that are private and non-discoverable. REST has no standardised contract or specification. Generating or writing those contracts and specifications must follow standardised conventions and best practices to allow their better understandability and re-usability.

   Summary: Depending on the technologies, the appearance of service antipatterns related to service contracts might vary.

2. The Web services clients invoke services relying on SOAP protocols. SCA also executes its component services relying on SOAP or REST protocols when various SCA composites are not within a single machine. As for REST, the service invocation is completely HTTP dependent and relies on client requests based on resources URIs.

   Summary: Antipatterns in REST depend on how well the HTTP client requests are formed following best practices for REST described in the literature from the client side and how well the HTTP responses are designed from the server side. Thus, depending on the technologies the types of antipatterns may vary.

3. Finally, the response data and exchanged messages are available only in the XML-form for Web services, in XML or SDO (Service Data Objects) for SCA systems, and in any Web formats like JSON, MIME, YAML, or PDF for REST.

   Summary: For Web services and SCA, having only one data representation is not a bad practice at all; whereas REST must facilitate multiple representations of the same resource, and, if not, it might be considered as one of the bad practices in REST.

In the next section, we compare among service antipatterns based on their appearances in various SBSs technologies.

2.5 Comparison among Service Antipatterns

Designers or developers may follow common bad practices while designing or developing SBSs using different SBSs technologies. Thus, service antipatterns exist in different SBSs technologies, including SCA, Web services, and REST.

Figure 2.6 shows the Venn diagram relating 31 service antipatterns found in SCA, Web services, and REST. We identify and relate these 31 service antipatterns from the literature [Král et Žemlička (2008); Rodriguez *et al.* (2010a); Modi (2006); Dudney *et al.* (2003); Jones
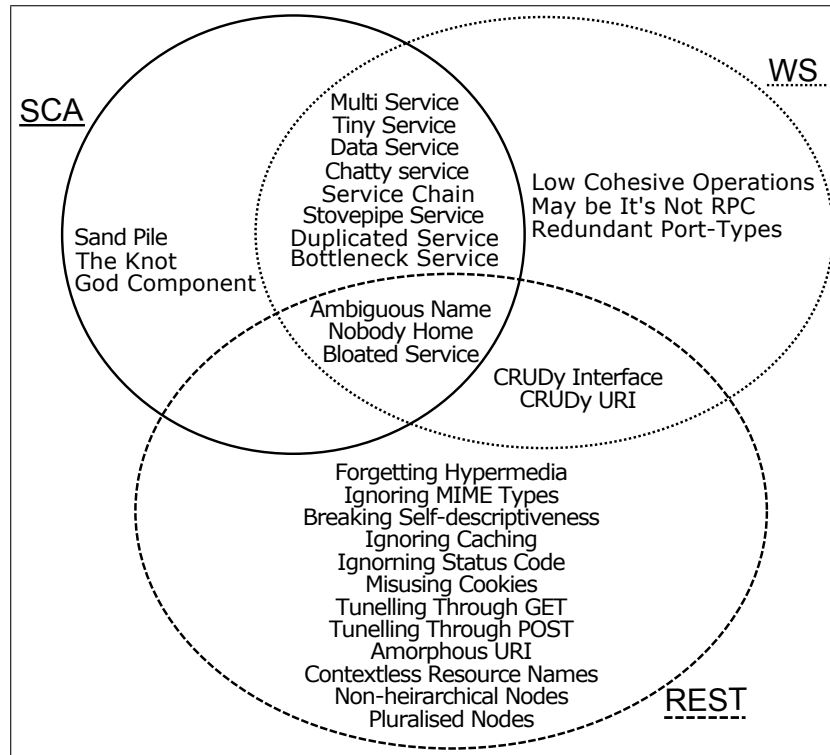
Figure 2.6: The Set Relation Among Service Antipatterns Found in SCA, Web services, and REST.

(2006); Evdemon (2005); Massé (2012); Tilkov (2008)]. This diagram shows the specific and common antipatterns shared by SBSs technologies. Antipatterns may be of types inter-service (involve other services in the system) or intra-service (do not depend or impact other services directly) and may require static, dynamic, or hybrid analyses of services to detect them.

The antipatterns in SCA and Web services are described in the literature based on various criteria related to services design and implementation and their runtime behavior (*e.g.*, *Multi Service*, *Tiny Service*, and *Chatty Web service*) because in SCA and Web services, clients consume services by invoking operations defined in their services' interfaces, and that the runtime behavior of services depend on how well the services are designed and implemented.

In contrast, the antipatterns in REST are described based on *resources*, *client requests*, and *server responses*. REST clients are unaware of the concrete service interfaces and can only send requests using well-known HTTP methods. REST antipatterns are defined in the literature focusing on best practices of making requests by clients and sending responds

by servers. For example, *Forgetting Hypermedia* and *Ignoring MIME Types* antipatterns indicate the scarcity of HATEOS (Hypermedia as the Engine of Application State) principle and content negotiation mechanism [Tilkov (2008)].

The detection algorithms for common antipatterns (see Table 2.2) should be similar across technologies, but require a unified abstraction of the SBSs technologies. However, the detection mechanism for technology-specific antipatterns will vary. Table 2.2 shows a comparison among antipatterns in SCA, Web services, and REST. In general, antipatterns in Web services are defined at the service interface-level, whereas antipatterns in SCA are at component-level. *Components* in SCA are at the higher-level of granularity than the *services* in Web services.

In this dissertation, we consider Web services as individual entities, *i.e.*, no composition among them. Therefore, none of the discussed antipatterns for Web services are at the service composition-level. Nevertheless, there are antipatterns that spread across several Web services, for example, *Stovepipe Service* or *Single-Schema Dream* [Dudney *et al.* (2003)], which we plan to analyse as our future work because analysing a set of services requires at least one execution scenario that can be defined using a process language[1].

2.6   Discussion

Service-Oriented Architecture (SOA), with its own design principles, is already a dominant architectural model for designing and developing service-based systems (SBSs). SOA has several distinct concepts including—services, a remote client-server model, publicly available service interfaces, and searchable services registries. Besides, the implementation of SOA requires various SBSs implementation technologies. In this chapter, we compared those different SBSs implementation technologies and identifies a list of non-trivial differences that describe the technology-specific characteristics. These characteristics also provide a ground for antipatterns to be different from one technology to another. We also showed the differences and commonalities among the services antipatterns in various SBSs implementation technologies with which we can conclude:

Summary: Despite the presence of some common antipatterns across diverse SBSs technologies, there exist also antipatterns that are technology-specific, and, therefore, their specification and detection may differ, which poses the challenge to have a unified and technology-independent approach for the detection of service antipatterns in different SBSs technologies.

Table 2.2: Comparison among Antipatterns in SCA, SOAP Web service, and REST.

Architectural Choices	Antipatterns Names	Existence Level	Distribution Level
SCA ∩ REST ∩ Web services	Ambiguous Name	Interface	Intra-service
	Nobody Home	Interface	Intra-service
	Bloated Service	Interface	Intra-service
SCA ∩ Web services	Multi Service	Interface	Intra-service
	Tiny Service	Interface	Intra-service
	Data Service	Interface	Intra-service
	Chatty service	Interface	Intra-service
	Service Chain	Interface	Inter-service
	Duplicated Service	Interface	Intra-service
	Stovepipe Service	Interface	Intra-service
	Bottleneck Service	Interface	Intra-service
REST ∩ Web services	CRUDy Interface	Interface	Intra-service
	CRUDy URI	Resource	Intra-resource
SCA	Sand Pile	Composition	Inter-service
	The Knot	Composition	Inter-service
	God Component	Component	Inter-component
Web services	Low Cohesive Operations	Interface	Intra-service
	May be It's Not RPC	Interface	Intra-service
	Redundant Port-Types	Interface	Intra-service
REST	Forgetting Hypermedia	Resource	Intra-resource
	Ignoring MIME Types	Resource	Intra-resource
	Breaking Self-descriptiveness	Resource	Intra-resource
	Ignoring Caching	Resource	Intra-resource
	Ignorning Status Code	Resource	Intra-resource
	Misusing Cookies	Resource	Intra-resource
	Tunelling Through GET	Interface	Intra-service
	Tunelling Through POST	Interface	Intra-service
	Amorphous URI	Resource	Intra-resource
	Contextless Resource Names	Resource	Intra-resource
	Non-heirarchical Nodes	Resource	Intra-resource
	Pluralised Nodes	Resource	Intra-resource

In the next chapter, we highlight various research contributions in the literature on the detection of service antipatterns in various SBSs implementation technologies. Moreover, we briefly discuss various research contributions on the detection of OO antipatterns, which, we believe, would inspire us to derive a methodology that can be applied uniformly on various SBSs implementation technologies.

CHAPTER 3   LITERATURE REVIEW

3.1   Chapter Overview

Design of quality is essential for building easily maintainable and evolvable Service-based Systems (SBSs). Service antipatterns are potential ways to measure the design quality of SBSs. The most famous book on antipatterns by Brown *et al.* [Brown *et al.* (1998)] described an antipattern as a "*literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences*", and introduced a collection of 40 Object-Oriented (OO) antipatterns.

This chapter on the literature review presents notable studies on antipatterns that have been performed in the OO, Service-Oriented, and Component-based Systems (CBS) domains. Section 3.2 presents the various catalogs of antipatterns. Section 3.3.1 highlights the relevant studies on OO antipatterns detection that may inspire our research in the service domain. Section 3.3.2 discusses the relevant works on the detection of antipatterns in Web services, SCA, and REST—the three emerging SBSs implementation technologies. Section 3.3.3 highlights relevant works from the CBS domain. Finally, Section 3.4 summarises the chapter and identifies the existing gaps in the literature.

Since, this dissertation focuses only on antipatterns (and not on design patterns), in the below, we provide an exhaustive catalog of antipatterns proposed in the literature.

3.2   Catalog of Antipatterns

The catalog of OO antipatterns has already became enriched and matured. However, unlike the OO antipatterns, fewer books, and articles deal with service antipatterns: most references are Web sites where SOA practitioners share their experiences in service design and development [Král et Žemlička (2008); Jones (2006); Modi (2006); Fredrich (2012); Tilkov (2008); Pautasso (2009)].

Dudney *et al.* [Dudney *et al.* (2003)] in their book on service antipatterns provided a catalog of 53 antipatterns related to the architecture, design, and implementation of systems based on J2EE technologies, such as EJB, JSP, Servlet, and Web Services. Most service antipatterns described in this book cannot be detected automatically and are specific to a technology and correspond to variants of the Tiny and Multi Service. Král *et al.* [Král et Žemlička (2007)] described seven service antipatterns, which are caused by an improper

usage of SOA standards and improper practices borrowed from the OO design style. The catalog of service antipatterns is still growing. Moreover, the current literature suggests that due to the smaller 'catalog size' and limited available resources, *i.e.*, articles, books, journals, etc., the specification and detection of service antipatterns were not considered with greater importance by the SOA community.

The catalog of Web services-specific antipatterns and their specification and detection are still in their infancy. Antipatterns in the book *J2EE Antipatterns* [Dudney *et al.* (2003)] are described informally. Rotem-Gal-Oz *et al.* [Rotem-Gal-Oz *et al.* (2012)] also informally described several service antipatterns. All the above works contributed to the existing catalog of service antipatterns, but did not discuss their specification nor their detection.

As for the REST, there are few books [Erl (2009); Daigneau (2011); Erl *et al.* (2012)] that discussed a number of REST patterns. In addition, a number of online resources [Pautasso (2009); Tilkov (2008); Fredrich (2012)] by REST practitioners provided a high-level overview of REST patterns and antipatterns related to API design with simple examples and discussed how they are introduced by the developers.

For example, Erl in his book [Erl (2009)] discussed 85 SOA patterns related to service design and composition. Erl *et al.* [Erl *et al.* (2012)] also explained the REST and RESTful service-orientation, and discussed seven new REST patterns, thus in total, the catalog defines 92 REST patterns. Daigneau [Daigneau (2011)] introduced 25 design patterns for SOAP (Simple Object Access Protocol) and RESTful services related to the service interaction, implementation, and evolution. Moreover, various online resources [Pautasso (2009); Tilkov (2008); Fredrich (2012)] defined a limited number of REST antipatterns. Moreover, the definitions of a number of linguistic patterns and antipatterns in REST are proposed in the literature [Massé (2012); Berners-Lee *et al.* (2005); Fredrich (2012); Tilkov (2008)]. Beyond those studies, however, the detection of REST patterns and antipatterns require a concrete approach, to support their rigorous analysis, which is still lacking in the current literature.

In the next section, we highlight the relevant works in the literature (in the ascending chronological order) on the detection of antipatterns in the OO, service, and CBS domains.

3.3 Detection of Antipatterns

Detection of antipatterns might help in improving programs design and performance, and facilitate better evolution and maintenance in general. There are numerous approaches dealing with the detection of OO antipatterns. However, unlike the research in the OO domain, research on methods and techniques for the detection of service antipatterns is in

their infancy.

3.3.1 Studies on the Detection of OO Antipatterns

Table 3.1 highlights notable works, *e.g.*, [Smith et Williams (2000); Peiris et Hill (2014); Cortellessa *et al.* (2012, 2014); Marco et Trubiani (2014); Coscia *et al.* (2013); Khomh *et al.* (2011); Mateos *et al.* (2011); Moha *et al.* (2010); Salehie *et al.* (2006)] in the OO domain inspiring our work on the detection of service antipatterns in SBSs. Below, we discuss few of them briefly in connection to our research.

Smith and Williams [Smith et Williams (2000, 2002)] proposed a catalog of performance antipatterns exploiting well-defined templates. The authors coined the term *performance antipatterns* as '*the common performance mistakes made in software architectures or designs, which may have negative impacts on other quality attributes like reusability or modifiability*' [Smith et Williams (2000, 2002)]. They empirically showed the negative consequences of performance antipatterns, *i.e.*, performance antipatterns have a negative impact on the runtime performance of OO systems.

Salehie *et al.* [Salehie *et al.* (2006)] proposed a framework based on metrics and heuristics to detect OO antipattern with a direct comparison with the good OO design heuristics at the classes and their interaction-levels. The authors argued that the key benefit of their proposed detection framework is its extensibility, *i.e.*, new quality factors and their related heuristics and metrics can be added and detected incrementally.

Moha *et al.* [Moha *et al.* (2010)] proposed the first rule-based DECOR approach for the specification and detection of OO antipatterns. Using the DECOR, engineers can specify OO antipatterns at higher-level of abstraction and automatically generate their detection algorithms. The authors validated DECOR with four well-known OO antipatterns, namely *Blob*, *Functional Decomposition*, *Spaghetti Code*, and *Swiss Army Knife*. However, the proposed DECOR approach suffered from the low precision [Moha *et al.* (2010)].

Later, Khomh *et al.* [Khomh *et al.* (2011)] proposed BDTEX, a GQM-based (Goal Question Metric) method, for the detection of antipatterns in OO systems. The authors relied on a Bayesian Belief Network (BBN) built using symptoms—a set of attributes of an antipattern—and investigated the probabilistic relationships between symptoms and antipatterns. The BDTEX outperformed its *state of the art* approach, DECOR [Moha *et al.* (2010)] in terms of precision and recall, and minimised the effort required by the quality analysts, as suggested by the authors [Khomh *et al.* (2011)].

In another recent work, Maiga *et al.* [Maiga *et al.* (2012)] proposed the SMURF approach

(using SVM and feedback from engineers) to detect OO antipatterns. The authors identified several limitations in the literature, including the fact that existing studies were demanding an extensive knowledge of OO antipatterns and those studies suffered from low precision and recall, which could be tackled through receiving feedback from the engineers at detection time. The authors showed improvement in terms of accuracy over the two *state of the art* methods DECOR [Moha *et al.* (2010)] and BDTEX [Khomh *et al.* (2011)].

Other relevant studies on the detection of OO antipatterns include [Peiris et Hill (2014)] and [Marco et Trubiani (2014)] where the authors monitored system performance metrics, for example, the CPU utilisation metrics. They supported the argument of using system performance metrics because the source code analysis requires highly-skilled domain knowledge and source code may not always be available for such analysis.

Although a number of OO methods exist in the literature for the detection of OO antipatterns, these detection techniques cannot be directly applied to SOA. Indeed, SOA focuses on *services* as first-class entities, whereas OO focuses on *classes*, which are at a lower level of granularity. Moreover, the highly dynamic nature of SOA environment raises several challenges that are not faced in OO development and requires more dynamic analyses than OO systems. However, all those previous works on OO systems may form a sound basis of expertise and technical knowledge for building methods for the detection of service antipatterns in SBSs.

3.3.2 Studies on the Detection of Service Antipatterns

Various studies were carried out for the detection of technology-specific antipatterns, for example, in SCA [Nayrolles *et al.* (2013)] and Web services [Tripathi *et al.* (2014); Torkamani et Bagheri (2014); Anchuri *et al.* (2014); Zheng et Krause (2006); Rodriguez *et al.* (2010b); Mateos *et al.* (2011); Rodriguez *et al.* (2013)]. There are a number of works for discovering bad practices in writing WSDLs [Torkamani et Bagheri (2014); Anchuri *et al.* (2014); Zheng et Krause (2006); Rodriguez *et al.* (2010b); Mateos *et al.* (2011); Rodriguez *et al.* (2013)]. Some works contribute to the catalog of service antipatterns by defining new antipatterns, but do not focus on their specification and detection, for example, [Král et Žemlička (2008, 2007, 2009); Tripathi *et al.* (2014)]. However, to get the full benefit of these newly defined service antipatterns, not only the definitions of new service antipatterns, but also there should be a generic way to specify and automatically detect them within the SBSs regardless of their underlying implementation technologies.

Detection of Service Antipatterns in SCA

A fewer studies were performed on the detection of antipatterns in SCA. For example, Nayrolles *et al.* [Nayrolles *et al.* (2013)] proposed SOMAD approach (Service Oriented Mining for Antipattern Detection) for the detection of antipatterns in SCA systems by mining execution traces. SOMAD mines strong associations between sequences of service/method calls from the execution traces of an SBS and further filters them by using metrics. Nevertheless, the SOMAD approach requires the instrumentation of the SBSs under analysis. However, the source codes of the SBSs for instrumentation are not always publicly available, which is the main drawback of SOMAD.

Detection of Service Antipatterns in Web Services

Král and Žemlička [Král et Žemlička (2008, 2007)] presented a list of the most risky antipatterns, *i.e.*, antipatterns occurring very often and having crucial consequences (consequences are measured using he process risk assessment). In another work, Král and Žemlička [Král et Žemlička (2009)] analysed frequently used SOA development practices by the developers and showed that the practices implicitly apply some known service antipatterns. The authors concluded that it is important to avoid developing fine-grained (very small) services and interfaces. They also suggested that the antipatterns in SOA are mainly caused by improper design principles borrowed from OO domain. However, the authors did not discuss the detection strategies of those newly defined service antipatterns.

Rodriguez *et al.* [Rodriguez *et al.* (2010b, 2013)] presented a catalog of eight Web Service discoverability antipatterns and conducted an empirical study on the retrieval performance of three Web services discovery systems. The study was performed using the Web Services with antipatterns and the Web services with refactored antipatterns. The results showed that the refactoring of the antipatterns eases the discovery process by allowing the Web services discovery systems better rank more relevant Web services, given the same search query.

Coscia *et al.* [Coscia *et al.* (2013)] empirically showed that for Web Services, there is a significant statistical correlation between OO metrics and WSDL-related metrics. These OO metrics and WSDL-metrics measure the quality of source code and the complexity of services' WSDL interfaces, respectively. However, the detection of antipatterns in service interfaces was not aimed in the study.

Torkamani and Bagheri [Torkamani et Bagheri (2014)] presented a repository of 45 general antipatterns in SOA from the literature and proposed a systematic approach to assist architects for the detecting and avoiding antipatterns in the service development process. The

authors, however, did not offer their automatic detection strategies.

Anchuri *et al.* [Anchuri *et al.* (2014)] presented a framework to detect hot-spots in SBSs relying on objective functions. The proposed framework combines service metrics data, both the historical and current, for ranking hot-spots in the services call graph, which recommends a set of services to be optimised. However, the assessment of the quality of the structural design for services were not their main goal.

Finally, one highly relevant study to our research, recently Ouni *et al.* [Ouni *et al.* (2015)] introduced a novel search-based approach for detecting antipatterns in Web services. In the proposed approach, the detection rules are automatically inferred from a set of examples of Web service antipatterns after selecting the best combination of metrics (and their threshold values) from a set of candidate metrics. This work is interesting as the authors represented the *detection* problem as an *optimisation* problem relying on genetic programming [Koza (1992)]. The authors conducted experiments on a benchmark of 310 Web services from various domains. However, their detection accuracy did not improve compared to one of our recent studies in [Palma *et al.* (2014b)].

In summary, the studies on Web services either focus on providing the catalog of service antipatterns [Král et Žemlička (2008, 2007, 2009); Torkamani et Bagheri (2014)] or only on the structural aspects [Rodriguez *et al.* (2010b, 2013)], or solely on the behavioral aspects [Coscia *et al.* (2013); Anchuri *et al.* (2014); Ouni *et al.* (2015)] of service interfaces. However, in the literature, no work focused on both the structural and behavioral aspects of Web services, which we considered in this dissertation.

Detection of Service Antipatterns in REST

After a thorough literature survey, we found no direct study on the antipatterns in REST. However, some works, for example, dealt with the fault tolerance of RESTful applications [Edstrom et Tilevich (2012)] or with the evolution patterns of RESTful APIs [Wang *et al.* (2014)] classifying the types of changes over several versions of RESTful APIs. Thus, our study on REST antipatterns will bring direct benefit to the REST community by showing the evidence of the presence of antipatterns in real-world RESTful APIs.

3.3.3   Other Related Studies

Also, some works on the detection of antipatterns in the CBS have been carried out [Garcia *et al.* (2009); Parsons et Murphy (2008); Cortellessa *et al.* (2010); Trubiani *et al.* (2014); Wert *et al.* (2014); Zhang *et al.* (2012)].

For example, Parsons and Murphy [Parsons et Murphy (2008)] proposed a framework for automatically detecting performance antipatterns and assessing the impact of poor performance designs in CBS. The authors relied on run-time analysis of CBS. One notable benefit of their framework is the ability to visualise the detected antipatterns, which facilitates the better comprehension of the identified design problems and promotes faster resolutions. However, their framework is dedicated to only component-based J2EE systems.

Garcia *et al.* [Garcia *et al.* (2009)] introduced the notion of *architectural smells* and defined their characteristics. The authors argued that *architectural smells* are different from the antipatterns. According to the authors, the *architectural smells* are the group of potential problems caused by the presence of design fragments that frequently change over the whole duration of software development and maintenance. The proposed approach is applicable on both the newly developed or reversed engineered architecture. However, these architectural smells works only for CBS.

The approach proposed by Cortellessa *et al.* [Cortellessa *et al.* (2010)] ranks the possible causes of performance antipatterns based on their *guiltiness* where the guiltiness is a given score for an antipattern for its violation against a set of performance requirements. The main goal of this study was to generate a new software model free of performance deficiencies. For the detection of antipatterns, the authors defined rules using first-order logic and relied on the Java Rule Engine API as their rule engine and later applied those rules on a system model. However, the implementation part of the rules were done manually and the proposed work focused only on antipatterns related to performance degradation and the quality of structural design was not of their interest.

Among the other related works: Trubiani *et al.* [Trubiani *et al.* (2014)] investigated the relations between the *bottleneck analysis* and *performance antipatterns* detection. The detection of performance antipatterns were performed based on defined heuristics. Zhang *et al.* [Zhang *et al.* (2012)] reported the SM@RT tool for the detection of Java EE antipatterns in a system model at runtime and experimented with 35 antipatterns on six real Java EE applications.

3.4 Summary on Literature Review

We performed a thorough literature review following the guidelines by Kitchenham [Kitchenham (2004)] and retrieved relevant research works related to service antipatterns detection and provide a summary. In Table 3.1, we list works that specifically deal with antipatterns in CBS, OO and SO domains. For the SO domain, Table 3.1 shows the indi-

Table 3.1: Relevant Works in the Literature (in the Chronological Order) on the Catalog and the Detection of Antipatterns in Component-based Systems (CBS), Object-Oriented Systems (OO), and Service-based Systems (SBSs)—SCA, SOAP Web services, and REST.

Component-based Systems (CBS)	Object-Oriented Systems (OO)	Service-based Systems (SBSs)			
		SCA	SOAP Web services	REST	Unified
Cortellessa *et al.* (2010)	Peiris et Hill (2014)	Nayrolles *et al.* (2013)	Ouni *et al.* (2015)		
Trubiani *et al.* (2014)	Cortellessa *et al.* (2014)		Anchuri *et al.* (2014)		
Wert *et al.* (2014)	Marco et Trubiani (2014)		Torkamani et Bagheri (2014)		
Zhang *et al.* (2012)	Ouni *et al.* (2013)		Tripathi *et al.* (2014)		
Garcia *et al.* (2009)	Maiga *et al.* (2012)		Coscia *et al.* (2013)		
Parsons et Murphy (2008)	Cortellessa *et al.* (2012)		Rodriguez *et al.* (2013)	X	X
	Khomh *et al.* (2011)		Mateos *et al.* (2011)		
	Stoianov et Sora (2010)		Rodriguez *et al.* (2010b)		
	Moha *et al.* (2010)		Král et Žemlička (2009)		
	Salehie *et al.* (2006)		Král et Žemlička (2008)		
	Smith et Williams (2002)		Král et Žemlička (2007)		
	Smith et Williams (2000)		Zheng et Krause (2006)		

vidual studies performed on providing antipatterns catalogs or on the detection of service antipatterns in SCA, Web services, and REST.

3.5 Discussion

From the literature survey performed in Sections 3.2 and 3.3, we identified the following gaps in the literature:

1. Numerous contributions are presented in the literature for analysing OO design quality to detect antipatterns in OO systems, whereas the analysis of SBSs was exploited a little. Those OO approaches are not applicable to SBSs;

2. Several approaches were proposed in the literature to analyse SCA systems, relying on the analysis of execution traces [Nayrolles *et al.* (2013)] or metric-based quantitative analysis [Demange *et al.* (2013)]. However, these approaches are strict to SCA;

3. A number of empirical validations were performed in the literature on the detection of antipatterns in Web services interfaces, *i.e.*, discovering bad practices in writing WSDL or identifying best practices [Rodriguez *et al.* (2010b, 2013); Tripathi *et al.* (2014)]. However, these analyses are static and do not incorporate services runtime aspects (*e.g.*, *availability* or *response time*);

4. Moreover, there exists no approach in the literature dealing with the detection of antipatterns in RESTful APIs. A high demand of RESTful APIs and their increased usage require rigorous assessment for the improved consumption and maintenance of RESTful APIs;

5. There exists no unified abstraction that combines different SBSs technologies and provides a means to detect service antipatterns in SBSs developed using diverse technologies in a generic way;

6. A unified (framework-based) approach is missing in the literature for the static and dynamic analyses of SBSs to detect service antipatterns in SBSs independent of SBSs technologies.

In this dissertation, we fill the above gaps in the literature and contribute by proposing a unified abstraction that can model and represent any SBSs implementation technologies and a unified detection approach, relying on an underlying detection framework that can support the specification of various service antipatterns.

However, before proposing any solution, a good understanding of various SBSs technologies and their relations is a prerequisite. Modeling various technology-specific concepts and relating those concepts (where applicable) will facilitate the better comprehension of various SBSs implementation technologies. Such a model—a unified abstraction—will also help having a common language for the specification of service antipatterns with a higher-level of abstraction.

In the next chapter, we present a unified abstraction and a meta-abstraction of three SBSs implementation technologies: Service Component Architecture (SCA), Web services, and REST.

CHAPTER 4   ABSTRACTION OF SBSs TECHNOLOGIES

4.1   Chapter Overview

This chapter introduces a unified abstraction and a meta-abstraction for SBSs technologies. We combine various SBSs technology meta-models, as shown in Figure 4.1, showing their commonalities and differences at the level of their design, implementation, and consumption to build the unified abstraction. Then, we abstract the concepts in an *abstract* abstraction to reduce heterogeneity, which is applicable to each of the three SBSs technologies discussed in Section 2.3. The meta-abstraction we present in Section 4.3 helps viewing all the technology-specific models from a single perspective and thus facilitates reasoning about antipatterns in an abstract way. The meta-abstraction in Figure 4.3 presents the strong inter-technological relations among the various design components in the service domain.

4.2   The Unified Abstraction

We introduce the unified abstraction for Web services, SCA, and REST in Figure 4.1. Individual meta-models of SCA, REST, and Web services can be found in [Abid *et al.* (2011)], [Valverde et Pastor (2009)], and [WWW-Consortium (2006)], respectively. We present the simplified versions of their meta-models by hiding optional details not related to services concrete design and implementation. Our proposed unified abstraction has five parts, each representing a distinct area within the model.

Part 1.   The first part includes only the REST-related section and excludes those that are common between REST and Web services. More specifically, the concepts `Resource`, `HTTP Method`, `Request`, and `RESTService` based on URI conventions reside with REST-style architecture where a `RESTService` is defined as a collection of multiple `Resources` that can be accessed via a `baseURI`. To access a `Resource` at least one `Method` is defined by the REST developer. HTTP methods (*e.g.*, GET, POST, PUT or DELETE) are used to make user requests on the `Resource`.
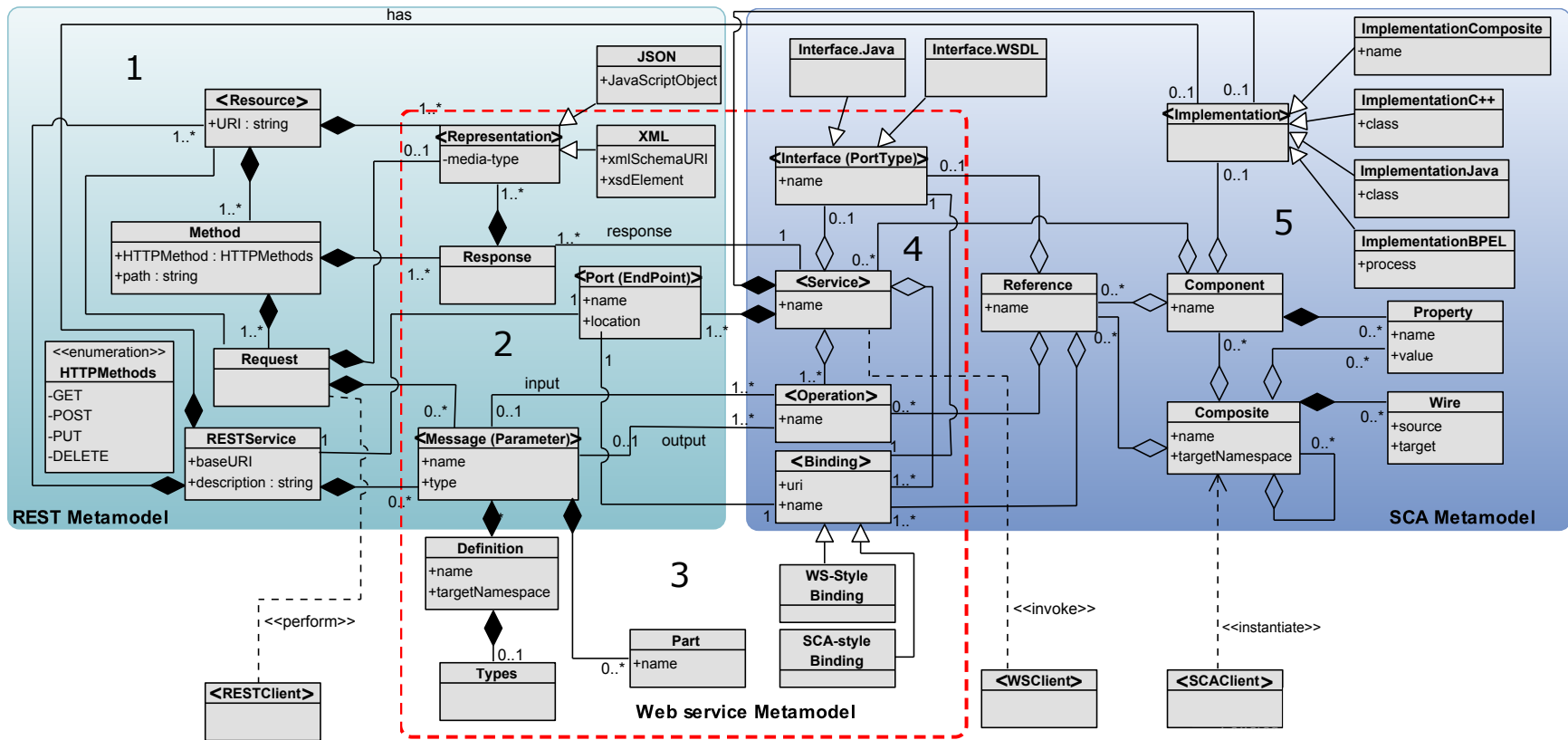
Figure 4.1: The Unified Abstraction for Web services, SCA, and REST (the elements related to meta-abstraction are shown inside angle brackets).

Part 2. The second part includes the intersection between REST and SOAP Web service meta-models. A set of arguments, *a.k.a.*, `Parameter`, must be specified to make an HTTP `Request`. The last state of the requested resource is returned as a `Response` message that must have a `Representation`. A `Representation` is generally in a globally accepted form, *i.e.*, XML, JSON, and so on. Thus, in the meta-model, the `Representation` is defined as an abstract entity, which is generalised in various data representation formats.

Part 3. The unique concepts in this third part are: `Definition` of message, `Binding`, `Parts`, and `Types` of messages. For Web services, the `Types` and `Parts` of the `Messages` and `Binding` types are required during the services invocation. Also, the way `Messages` are defined in Web services interfaces is unique to the meta-model of Web services—each message related to different operations has a particular type and message-parts are defined independently within the service interface. This part of the unified abstraction excludes the common parts of REST and SCA from the Web services.

Part 4. The fourth part includes the common concepts between Web services and SCA where a `Service` implements an `Interface` and each service has at least one `Operation` defined and at least one `Binding` is required to be specified. Therefore, the concepts `Interface/PortType`, `Service`, `Operation`, and `Binding` are common between Web services and SCA architectural meta-models.

Part 5. This part includes the concepts specific to SCA, including `Reference`, `Component`, `Wire`, `Property`, `Composite`, and so on. The design and implementation of SCA systems conform such architectural components, which are not found in REST and Web services architectural meta-models. In SCA, a `Component` provides at least one `Service` and several `Components` may reside within a `Composite`. Multiple `Composites` may be connected via `Wires` and `Components` might be dynamically reconfigured via various `Property` values.

This first unified abstraction for SBSs technologies brings the following benefits:

1. It is helpful in analysing various service antipatterns and SBSs by providing a common language to specify service antipatterns at high-level of abstraction beyond any ambiguity;

2. It clearly separates the technology-specific concepts and relates them where applicable, thus is helpful in understanding the commonalities and differences among various SBSs technologies;

3. Finally, it allows building a meta-abstraction using inter-related concepts.

Our unified abstraction is applicable to any SBSs developed with SCA, REST, or Web

services. It is, however, extensible for new technologies by integrating their shared concepts, *i.e.*, architectural components and communication styles.

In the next section, we discuss some of our observations based on our experience while trying to build a technology-neutral unified abstraction.

4.2.1 Our Observations on the Unified Abstraction

– The presented unified abstraction (in Section 4.2) plays an important conceptual role. It provides a clear understanding of the mutual exclusiveness and inclusiveness among different SBSs implementation technologies. However, our ultimate goal is to use this unified abstraction towards the specification of service antipatterns.

– According to the unified abstraction, it is evident that SBSs technologies are not so similar in terms of their technical specifications and domain-specific concepts. The three presented SBSs technologies have a number of entities and concepts that are unique. This uniqueness poses the challenge to have a unified approach for the specification and detection of service antipatterns in a unique manner, *e.g.*, to rely on the same grammar and use the same specification language for service antipatterns.

– The proposed unified abstraction can be seen as the composition of different SOA schemas with some relations among elements (or concepts) from different technology domains. In the literature, this unification was never been attempted. In fact, due to its complexity to build and the possibility to reuse this abstraction, very few contributions are made.

– One of the complexities arises from the fact that these three technologies (*e.g.*, SOAP Web services, SCA, and REST) have unique building blocks as their first class design elements. SOAP Web services rely on SOAP *services* and WSDLs to publish and consume their services, SCA relies on its own specific types of *components*, and REST relies on *resources* to conceptually design and fully operationalise an SBS. The technical specifications of these three building blocks differ significantly.

– Finally, acquiring such a unified abstraction was not so easy and straightforward. A significant amount of literature review effort and paper works were involved before coming up with a final unification, which conforms to all the three technologies and does not negate their individual syntactic and semantic meanings.

In the next section, we introduce the proposed meta-abstraction of the three SBSs implementation technologies.

4.3 The Meta-abstraction

We propose a meta-abstraction that combines all the base but mandatory elements of different SBSs technologies (*i.e.*, SCA, Web services, and REST). Figure 4.2 shows our proposed meta-abstraction, which embodies a `Service` implementing an `Interface` and has a concrete implementation. Services incorporate a list of operations where each `Operation` is associated with some `Messages`, *i.e.*, parameters. Every service has some `Bindings` defined that denote which transport protocol and data format to use. A service is also related to a dedicated port, *i.e.*, service's physical location, referred as `Service Endpoint`. An operation deals with a set of `Messages` and–or `Resources`, which have some schematic object-style representation and are transferable through a network, *i.e.*, the Internet.
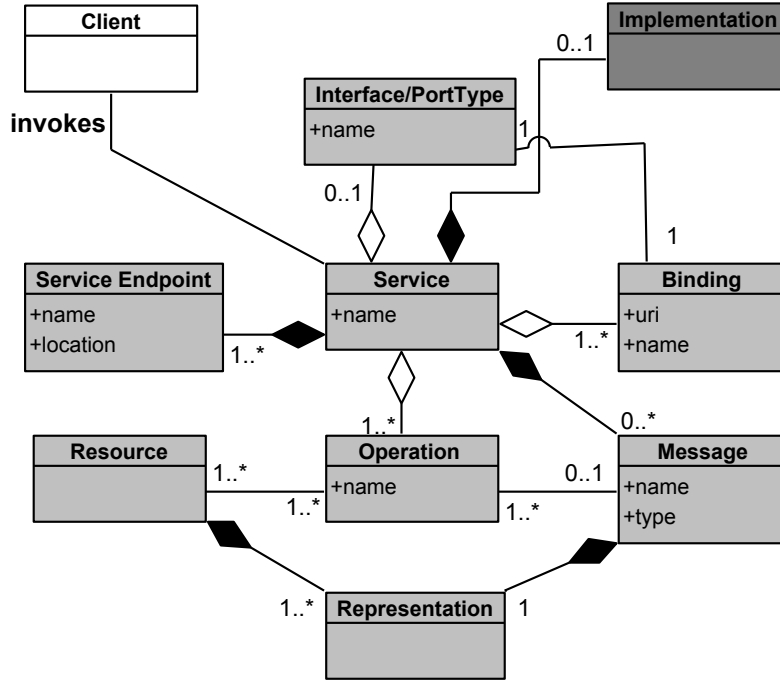
Figure 4.2: The Meta-abstraction of Web services, SCA, and REST.

4.4 Discussion

The proposed meta-abstraction conforms to any current SBSs implementation technologies and we believe will fit upcoming (if any) SBSs technologies. Each of the SBSs technologies discussed in Section 2.3 shares these common service-related concepts, and, thus, this conceptual abstraction for technologies helps us to work with different SBSs tech-

nologies. Moreover, the proposed unified abstraction (see Section 4.2) facilitates us to have a common specification language to represent various service antipatterns with higher level of abstraction.

Relying on the proposed unified abstraction (in Section 4.2) and the meta-abstraction (in Section 4.3), in the following chapter, we present a unified approach for the specification and detection of service antipatterns in different SBSs implementation technologies. The unified abstraction comes into effect in introducing a common specification language, which is the first step for the detection of service antipatterns.

CHAPTER 5   SPECIFICATION AND DETECTION OF SERVICE ANTIPATTERNS

5.1   Chapter Overview

This chapter presents and discusses our proposed unified approach in detail. The different steps of our approach are described in Sections 5.2.1 to 5.2.3. In particular, Section 5.2.1 elaborates the process of antipatterns specification and further discusses how we specify heuristics for REST-specific service antipatterns. Section 5.2.1 also presents our common domain specific language that we use for the specification of antipatterns. Section 5.2.2 describes the process of automatically generating detection algorithms for service antipatterns in SCA and Web services, and the process of manually implementing detection algorithms for REST antipatterns. Finally, Section 5.2.3 presents our unified detection framework and its various components. The unified framework is the core of our detection process.

5.2   Proposed Unified Approach

Figure 5.1 shows our proposed unified approach, SODA (Service Oriented Detection for Antipatterns), for specifying and detecting service antipatterns in service-based systems (SBSs). Our SODA approach encompasses the proposed unified abstraction presented in Sections 4.2 and 4.3 (see Figures 4.1 and 4.2). Starting with the textual description of service antipatterns, the specification and the generation of detection algorithms to the detection phase, we follow through with a validation at the end of the detection process. The three main steps of the proposed SODA approach include:
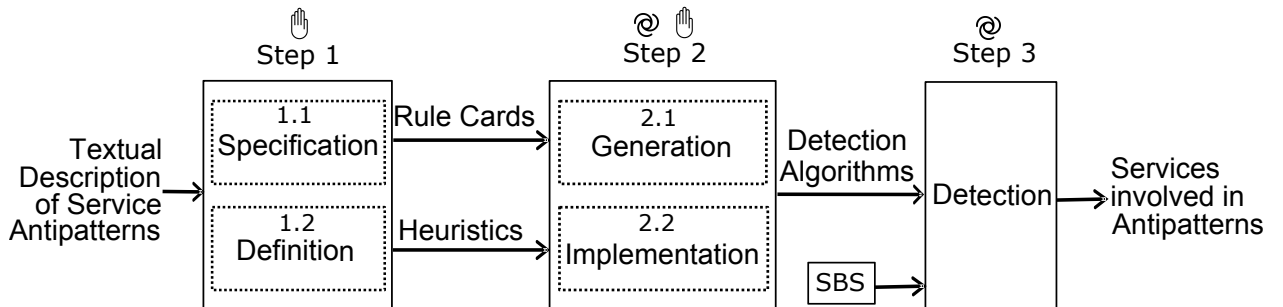
Figure 5.1: The Unified SODA Approach.

Step 1: Specification. This step includes performing a thorough domain analysis by studying the definitions and textual descriptions of antipatterns from the literature to identify relevant static and dynamic properties to specify them. The identified properties represent (1) measurable attributes of the proposed unified abstraction elements and (2) the inter-element relations among them. We use these properties as the basis of the vocabulary to define a common domain specific language (DSL) and formalise antipatterns with rule cards. Rule cards are representation of antipatterns at a high-level of abstraction, which are both machine processable and human understandable. Figure 5.3 shows examples of rule cards for *Multi Service* and *Tiny Service*. For REST antipatterns, we define detection heuristics that are applicable on REST requests and responses (see Figure 5.10).

Step 2: Generation. From the rule cards in the previous step, we intend to automatically generate their detection algorithms using a simple *template*-based technique (for SCA and Web services) or implement the detection algorithms for defined heuristics (for REST). Templates are defined with well-defined *tags* that can be replaced with values at runtime.

Step 3: Detection. For the detection of antipatterns, we introduce an underlying framework, SOFA (Service Oriented Framework for Antipatterns). The computations of all static and dynamic metrics, *i.e.*, identified relevant properties of antipatterns and related analyses are performed in SOFA framework. It also assists in syntactic and semantic analyses of services interfaces based on WordNet [Miller (1995)] and Stanford CoreNLP [Manning *et al.* (2014)]. In this step, we apply the detection algorithms automatically generated in the previous step on different SBSs and report suspicious services. Moreover, for REST services, SOFA provides the means to concretely send HTTP requests and to automatically apply the detection algorithms on both HTTP requests and responses.

The next two sections describe the process of the specification of service antipatterns and the generation of their detection algorithms. We describe our detection framework, SOFA, in Section 5.2.3. The validation of the approach is discussed in Chapter 6.

5.2.1 Step 1. Specification of Service Antipatterns

Current literature does not provide the specification of any service antipatterns. As the first step towards their specification, we perform a thorough domain analysis of service antipatterns by studying their definitions and textual descriptions in the literature [Král et Žemlička (2008); Dudney *et al.* (2003); Král et Žemlička (2009)] and in online resources and articles [Rodriguez *et al.* (2010a); Evdemon (2005); Cherbakov *et al.* (2006); Jones (2006); Modi (2006)]. The process of domain analysis involves identifying, capturing, and organising reusable information for using them in software development [Prieto-Díaz (1990)]. The

```
1  rule_card    ::= RULE_CARD:rule_cardName {(rule)+};
2  rule         ::= RULE:ruleName {content_rule};

3  content_rule ::= metric | relationship | operator ruleType (ruleType)+
4                 | RULE_CARD: rule_cardName

5  ruleType     ::= ruleName | rule_cardName

6  operator     ::= INTER | UNION | DIFF | INCL | NEG

7  metric       ::= id_metric ordi_value
8                 | id_metric comparator num_value

9  id_metric    ::= ALS | ANAM/ANAO | ANIM | ANP | ANPT | ARIM | ARIO | ARIP
10                | COH | CPL | NCO | NI | NIR | NMD/NOD | NOPT
11                | NOR | NPT | NSE | NUM | NVMS | NVOS | RGTS | TNP
12                | A | NMI | NTMI | RT

13 ordi_value   ::= VERY HIGH | HIGH | MEDIUM | LOW | VERY LOW
14 comparator   ::= < | ≤ | = | ≥ | >

15 relationship ::= relationType FROM ruleName cardinality TO ruleName cardinality
16 relationType ::= ASSOC | COMPOS
17 cardinality  ::= ONE | MANY | ONE_OR_MANY | num_value NUMBER_OR_MANY

18  rule_cardName, ruleName, ruleClass ∈ string
19  num_value ∈ double
```

Figure 5.2: BNF Grammar of Rule Cards for SODA.

domain analysis allows us to identify properties relevant to service antipatterns, including static properties related to their design (*e.g.*, cohesion and coupling) and also dynamic properties, such as various QoS properties (*e.g.*, response time or availability). Static properties are properties that apply to the static descriptions of SBSs, such as WSDL files for Web services, whereas dynamic properties are related to the dynamic behavior or nature of SBSs as observed during their execution. Our proposed unified abstraction (see Figure 4.1) and meta-abstraction (see Figure 4.2) represent the properties related to their static aspects. However, we do not show the dynamic or behavioral aspects of services in these meta-models because all the three SBSs technologies share the same quality of service concepts. Table 2.2 lists selected antipatterns in the SCA, Web services, and REST from the literature, which are commonly found in SBSs and well-explained with their related examples. A brief description of those service antipatterns are presented in Appendix A where we highlight various relevant properties in *bold-italic*. We use these properties as the base vocabulary to define our own DSL, in the form of a rule-based language for specifying service antipatterns. The DSL offers software engineers high-level domain-related abstractions and variability points to express different properties of service antipatterns based on their own judgment, experiences, and context.

We manually identify and organise relevant domain concepts and properties essential for specifying service antipatterns via rule cards at a high-level of abstraction using a DSL. Our proposed DSL (see Figure 5.2) allows the specification of antipatterns in a declarative way, relying on the compositions of multiple rules. We define DSL using a *Backus Normal Form* (BNF) grammar as shown in Figure 5.2. A *rule card* is identified by the keyword `RULE_CARD`, followed by a rule card name and a set of rules (line 1). A rule describes a set of static or dynamic properties, *e.g.*, metrics (lines 9–12), and may have relationships with other rules, such as via association (`ASSOC`) (lines 15-17), and may combine with other rules via set operators such as union (`UNION`) or intersection (`INTER`) (line 6). A metric may define a numerical value (line 7) or an ordinal value defined using five-point Likert scale (line 13). In SOFA framework, we define ordinal values, by relating ordinal values with concrete numerical values to avoid manually setting threshold values with the box-plot statistical technique [Chambers *et al.* (1983)]. We identify, as listed in Table 5.1, a set of 27 metrics (lines 9-12) after a thorough domain analysis, which can be easily extended by adding new metrics and can be used to specify various SCA and Web services antipatterns.

From Table 5.1, for example, the `ARIP`, `ARIO`, and `ARIM` metrics combine both the structural and semantic similarity computation for Web services. Structural similarity uses the well-known Levenshtein Distance algorithm, whereas semantic similarity uses WordNet [Miller (1995)] and Standford CoreNLP [Manning *et al.* (2014)]. WordNet is a widely used lexical database that groups nouns, verbs, adjectives, etc. into the sets of synsets, *i.e.*, cognitive synonyms, each representing a distinct concept. We use WordNet to find the cognitive similarity between two (sets of) operations, messages, or port-types. We use Stanford's CoreNLP: (1) to find the base forms of a set of signatures of operations, messages, or port-types and (2) to annotate them with the part-of-speech (`POS`) tagger after we split the signatures based on the CamelCase.

As for REST, we concretely define detection heuristics for REST-specific antipatterns. Thus, we do not rely on the proposed DSL for REST antipatterns.

With our domain analysis, antipatterns specifications are made in a consistent high-level abstraction and capture all relevant domain expertise. Thus, for the domain experts, it becomes easy to understand and modify the specifications without prior knowledge of the underlying detection framework. We provide the specifications of all service antipatterns analysed in this dissertation in Appendix A (see P. 127).

Figure 5.3 shows the rule cards for *Tiny Service* and *Multi Service* antipatterns. The *Multi Service* antipattern is characterised by very high response time, high number of operations, low availability, and low cohesion. A *Tiny Service* corresponds to a service that

```
1 RULE_CARD: MultiService {
2    RULE: MultiService {INTER MultiOperation HighResponse LowA LowCohesion};
3    RULE: MultiOperation {NOD VERY_HIGH};
4    RULE: HighResponse {RT VERY_HIGH};
5    RULE: LowA {A LOW};
6    RULE: LowCohesion {COH LOW};
7 };
```

```
1 RULE_CARD: TinyService {
2    RULE: TinyService {INTER FewOperation HighCoupling};
3    RULE: FewOperation {NOD VERY_LOW};
4    RULE: HighCoupling {CPL HIGH};
5 };
```

Figure 5.3: Rule Cards for *Multi Service* and *Tiny Service* antipatterns in SCA and Web services.

declares a very low number of operations and has a high coupling with other services. Thus, the *Multi Service* and *Tiny Service* antipatterns rely, for example, on the `Operation` and `Message` or `Parameter` elements and their measurable attributes from the proposed unified abstraction (see Figure 4.1).

Using a DSL offers greater flexibility than implementing *ad-hoc* detection algorithms manually. Indeed, the DSL is independent of any implementation concern, such as the computation of static and dynamic metrics and the multitude of underlying SBSs technologies. Moreover, the DSL allows the adaptation of the antipattern specifications to the context and characteristics of the analysed SBS by adjusting the metrics and associated values and considering the unified abstraction for SBSs technologies.

Heuristics of REST-specific Antipatterns

Unlike SCA and Web services, we do not formally specify REST antipatterns using rule cards because the specification of REST antipatterns does not imply the use of metrics. Moreover, the antipatterns in the literature defined for REST are of the nature that require observation rather than measurement, to be detected. For example, as presented in Figure 5.10, *Forgetting Hypermedia* [Tilkov (2008)] represents a case where the link to an entity, *i.e.*, *entity links*, are absent in the response body or header provided by the server. More specifically, for the HTTP GET requests, such *entity links* are provided in the response body or header, hence, checking missing links in the body or header of the response is adequate (see line 4, Figure 5.10). As for HTTP POST requests, the server provides an external *location* in the response header or a *link* in the response body. Therefore, looking for the absence of such location in the response header (see line 5, Figure 5.10), or missing link in the response

Table 5.1: List of 27 Service Metrics for Specifying Service Antipatterns.

Metrics	Full Names	Static/Dynamic
A	Availability of a Service	dynamic
ALS	Average Length of Signatures	static
ANP	Average Number of Parameters in Operations	static
ANPT	Average Number of Primitive Type Parameters	static
ANIO	Average Number of Identical Operations	static
ANAO	Average Number of Accessor Operations	static
ARIP	Average Ratio of Identical Port-Types	static
ARIO	Average Ratio of Identical Operations	static
ARIM	Average Ratio of Identical Messages	static
COH	Service Cohesion	static
CPL	Service Coupling	static
NCO	Number of CRUD Operations	static
NOD	Number of Operations Declared	static
NOPT	Number of Operations in Port-Types	static
NI	Number of Interfaces	static
NIR	Number of Incoming References	static
NMI	Number of Method Invocations	dynamic
NOR	Number of Outgoing References	static
NPT	Number of Port-Types	static
NTMI	Number of Transitive Methods Invoked	dynamic
NSE	Number of Services Encapsulated	static
NUM	Number of Utility Methods	static
NVMS	Number of Verbs in Message Signatures	static
NVOS	Number of Verbs in Operation Signatures	static
RGTS	Ratio of General Terms in Signatures	static
RT	Response Time of a Service	dynamic
TNP	Total Number of Parameters	static

1: Forget-Hyper-media(*response-header*, *response-body*, *http-method*)
2:    *body-links*[] ← Extract-Entity-Links(*response-body*)
3:    *header-link* ← *response-header*.getValue("Link")
4:    **if**(*http-method* = GET **and** (*length*(*body-links*[]) = 0 **or** *header-link* = NIL)) **or**
5:    (*http-method* = POST **and** ("Location:" ∉ *response-header*.getKeys() **and**
6:     *length*(*body-links*[]) = 0))) **then**
7:      **print** "*Forgetting Hypermedia detected*"
8:    **end if**

Figure 5.4: Heuristic of *Forgetting Hypermedia* Antipattern.

body (see line 6, Figure 5.10) is enough to detect *Forgetting Hypermedia* antipattern.

Thus, we rely on defining detection heuristics in the form of pseudo-code to ease their comprehension and detection. Using various static and dynamic properties identified from
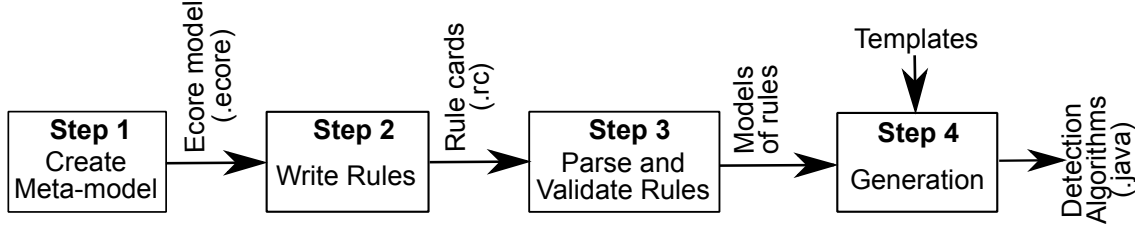
Figure 5.5: Different Steps Involved in the Automatic Algorithms Generation Process.

REST requests and responses, we define detection heuristics for REST antipatterns. The heuristic presented in Figure 5.10 is more suitable than a rule card for the detection of REST antipatterns because of their more intuitive nature. Moreover, an engineer's knowledge and experience on REST plays an important role in defining such heuristic.

5.2.2 Step 2. Generation of Detection Algorithms

This step follows a procedure to generate detection algorithms automatically using a simple template-based technique. From the previous step, for the SCA and Web services, using the specified rule cards, we generate detection algorithms for the service antipatterns. More specifically, we rely on Eclipse Modeling Framework [EMF-Eclipse (2010)] and code generation facility based on a predefined Ecore [Sciamma *et al.* (2013)] model. The EMF project, a modeling framework and code generation facility, provides tools and runtime support to generate compilable code.

To follow with the generation process, first, we create a meta-model of our DSL in Ecore format (Step 1). We detail the meta-model in Section 5.2.2. Then, we use EMFText [EMFText (2007)] to write and validate rule cards on-the-fly (Step 2). For the generation of the detection algorithms, first, we parse the rule cards of each antipattern and represent them as models. Then, we use Ecore to syntactically validate the rule card models against the meta-model of our DSL (Step 3). Ecore guarantees the correctness of the rule card models. We use a template-based code generation technique provided by Acceleo [Obeo (2005)] (Step 4). To do this, we define a unique template for all rule cards consisting of well-defined tags to be replaced with the values of different metrics defined in the rule cards of antipatterns. Finally, the template is applied to a rule card model resulting in a Java class, which is directly compilable and executable without any manual involvement. Figure 5.5 shows the different steps involved in the algorithm generation process. Figure 5.6 shows the snapshot of automatically generated detection algorithm for *Multi Service* antipattern. We show the class template for automatically generated detection code in Appendix B (see P. 149). Also,

```
package com.soda.antipatterns;

import com.sofa.metric.Metric;
import com.sofa.motifs.AMotif;
import com.sofa.rulecard.comparators.Comparator;
import com.sofa.rulecard.numoperators.NumOperator;
import com.sofa.rulecard.setoperators.SetOperator;
import com.sofa.rulecard.values.OrdinalValue;
import com.sofa.rulecard.smells.*;

public class MultiService extends AMotif {

    public MultiService() {

            // SMELL 1
            MetricValue metricValueSmell1 = new MetricLeaf(Metric.NOD);
            Smell smell1 = new SmellNumericValue(metricValueSmell1, Comparator.EQUAL, OrdinalValue.VERY_HIGH);

            // SMELL 2
            MetricValue metricValueSmell2 = new MetricLeaf(Metric.RT);
            Smell smell2 = new SmellOrdinalValue(metricValueSmell2, Comparator.EQUAL, OrdinalValue.VERY_HIGH);

            // SMELL 3
            MetricValue metricValueSmell2 = new MetricLeaf(Metric.A);
            Smell smell2 = new SmellOrdinalValue(metricValueSmell2, Comparator.EQUAL, OrdinalValue.LOW);

            // SMELL 4
            MetricValue metricValueSmell2 = new MetricLeaf(Metric.COH);
            Smell smell2 = new SmellOrdinalValue(metricValueSmell2, Comparator.EQUAL, OrdinalValue.LOW);
            // ROOT SMELL
            this.rootSmell = new SmellComposite(SetOperator.INTER);
            this.rootSmell.addChildSmell(smell1);
            this.rootSmell.addChildSmell(smell2);
            this.rootSmell.addChildSmell(smell3);
            this.rootSmell.addChildSmell(smell4);
    }
}
```

Figure 5.6: Automatically Generated Detection Algorithm of *Multi Service* Antipattern Represented as a Java Class.

the concrete EMF syntax of the rule cards is presented in Appendix B (see P. 150).

In Figure 5.6, the class `Multi Service` consists of four objects corresponding to four metrics defined in its rule card (see Figure 5.3). The four metric objects, *e.g.*, `NOD`, `RT`, `A`, and `COH`, are composed together and added to a `rootSmell`, which is the root container of the Multi Service rule card. Each of the metrics is implemented as a singleton class. When the `Multi Service` class is instantiated, the underlying metrics will be calculated and the values will be filtered accordingly, *i.e.*, based on the provided `OrdinalValue`.

This generative process is fully automated to avoid any manual tasks, which are usually repetitive and error-prone. This process also ensures the traceability between the specifications of antipatterns with the DSL and their concrete detection in SBSs using our underlying
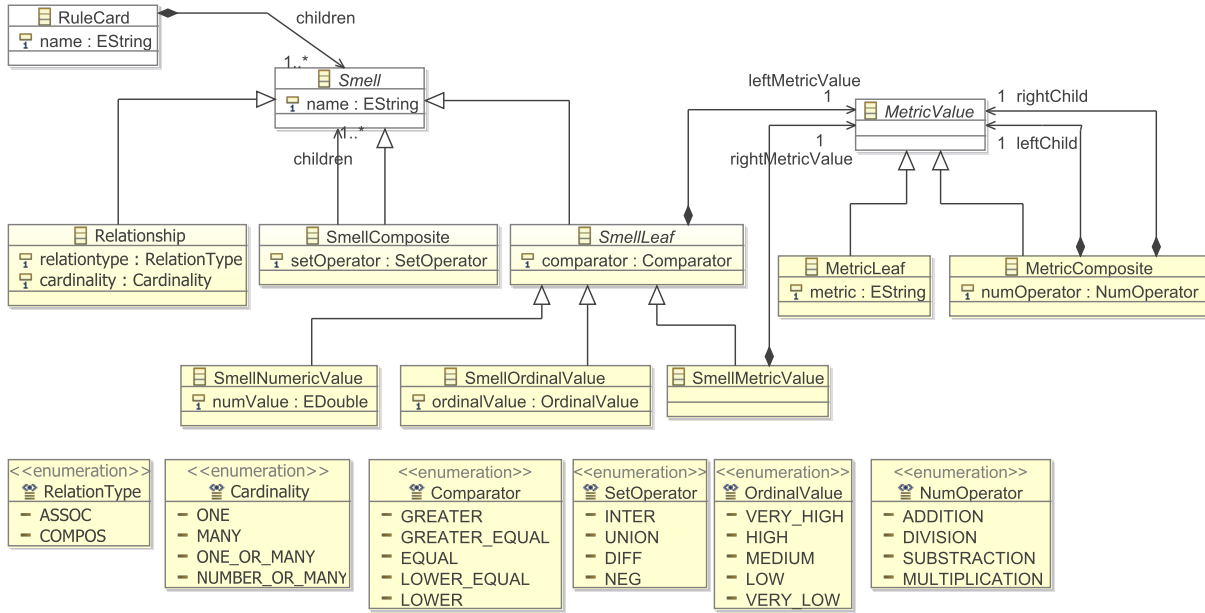
Figure 5.7: The Meta-model of Rule Cards.

framework. Consequently, software engineers can focus on the specification of antipatterns, without considering any technical aspects of the underlying framework.

As for REST antipatterns, from the heuristics defined in Section 5.2.1, we implemented their detection algorithms, which is fundamental to REST. These detection algorithms, by their nature, conform to their corresponding heuristics. We also require to implement REST services' interfaces for consuming REST services. Such interfaces comprise a list of methods mapped to their respective HTTP requests. We formulate these requests based on their online documentation (see Table 6.1), which contain a list of resources with their corresponding HTTP methods, specific end-point for each HTTP request, and a set of well-defined request parameters.

The next section discusses the meta-model of rule cards that we use to automatically generate detection algorithms for service antipatterns in SCA and Web services, in particular.

The Meta-model of Rule Cards

The meta-model of rule cards, as shown in Figure 5.7, defines different constituents to represent our rule cards, rules, set operators, relationships among rules, and various static and dynamic properties, *i.e.*, metrics. A rule card is specified concretely as an instance of Ecore *EClass* `RuleCard`. An instance of `RuleCard` is composed of `Smell` objects, which

describe rules that can be either simple or composite. A composite rule, `SmellComposite`, is composed of other rules, using the *Composite design pattern* [Gamma *et al.* (1994)] and different set operators. Various set operators are defined as an Ecore `ENum` and the structural relationships among rules are defined with `RelationType` and `Cardinality`. We have defined three other `ENum` classes: `Comparator`, `OrdinalValue`, and `NumOperator` for comparing and handling metric values.

In the next section, we describe SODA's underlying unified framework and its different components.

5.2.3   Step 3. Detection of Service Antipatterns: The SOFA Framework

Figure 5.8 shows the underlying framework, SOFA (Service Oriented Framework for Antipatterns), that supports the specification and detection of service antipatterns in SBSs. Our SOFA framework is also capable of performing the syntactic and semantic analyses of services interfaces based on WordNet [Miller (1995)] and Stanford CoreNLP [Manning *et al.* (2014)]. SOFA has eight modules, programmatically each of which represents a component providing a stand-alone service. The components include:

– **Detection** component represents the main detection engine that initiates and controls the overall detection process. It also provides an interface to the clients to run the detection process and help the clients visualise the detection results;

– **Metric** component provides the computation of both the static and dynamic metrics of our metric suite. This component also stores the static metric values in a repository to be used on the fly. The values of dynamic metrics cannot be restored, as they may change over executions;

– **Rule Specification** component is responsible for specifying rule cards using the **Rule** component and **Operator** component. All the rule cards are also restored in a repository used by the **Algorithm Generation** component;

– **Algorithm Generation** component generates the detection algorithms automatically from the specified rules. Then, these detection algorithms will be executed by the clients using the **Detection** component;

– **Rule** component represents a repository of all the singleton rules that are composed of metrics, and depends on **Metric** component to get required metrics;

– **Operator** component provides all the boolean and comparison operators to merge or group the rules to form a rule card;

– **Boxplot** component provides the means for computing boundary values and threshold values. It provides all statistical analyses during the detection phase of our approach;
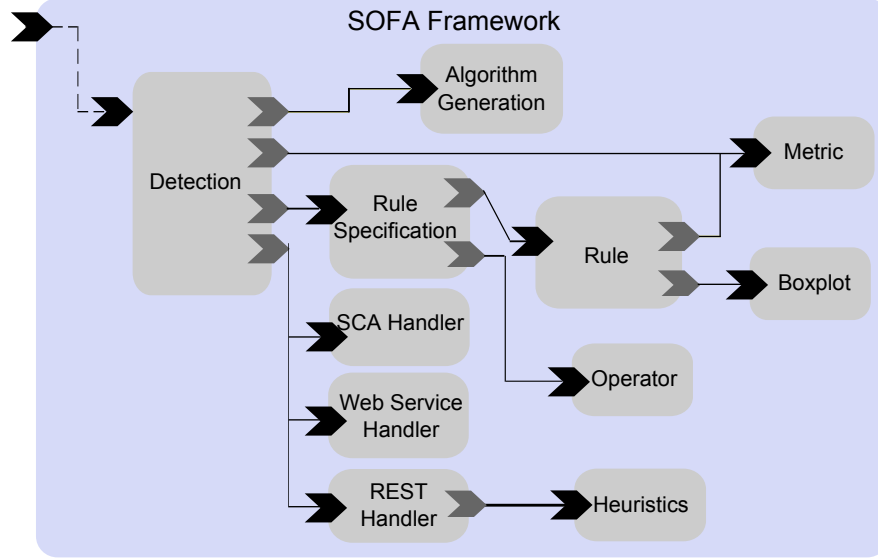
Figure 5.8: The SOFA Framework.

– **Heuristics** component provides the implementations for the defined heuristics to be applied on REST services for detecting REST antipatterns;

With respect to the computation of metrics, the generated detection algorithms call sensors and triggers implemented using the services provided by *FraSCAti*. These sensors and triggers, implemented as join points in an aspect-oriented programming style, allow, *at runtime*, the introspection of the interface of services and the triggering of events to add non-functional concerns, such as transactions, debugging, and, in our case, the computation of metrics such as response time.

These sensors and triggers are provided at the deployment of the SBS under analysis. The code excerpt shown in Figure 5.9 presents the computation of the response time as a join point at the service interface level. The sensor `RTIntentHandler` (line 1) corresponds to an aspect that will intercept a service call and monitor the service response time. An intent join point (line 2) corresponds to the interface where a service invocation has been intercepted. The code enabling the computation of the response time is inserted before and after the invocation of the service (line 5). This new monitoring aspect is then declared as a service and added to the SOFA framework within the metric module (line 7).

The SOFA itself is a *service*-based framework and developed with SCA technology [Chappell (2007)]. We also have three other components: `SCA Handler`, `Web Service Handler`, and `REST Handler` dedicated to the analyses of different technology-specific systems. More

```
 1: public class ‹RT›IntentHandler implements IntentHandler {

    2: public Object ‹invoke›(‹IntentJoinPoint ijp›) {

       3: long ‹startTime = System.currentTimeMillis();›
       4: Object ret = null;
       5: ‹ret = ijp.proceed()›;
       6: long ‹estimatedTime = System.currentTimeMillis() - startTime;›

       7: ‹Metrics.setValue("RT",estimatedTime);›

       8: return ret;

    9: }

10: }
```

Figure 5.9: The Example of the Usage of Sensors and Triggers.

specifically, the `SCA Handler` is responsible for executing the use-case scenarios of *FraSCAti* and *Home-Automation* systems. The different functionalities performed by the `Web Service Handler` component include: (1) given keywords, it returns a list of SOAP Web services from a search engine, (2) it then filters broken service descriptions or unavailable services, and finally (3) for all Web services, it generates a list of SCA components. Concretely, these SCA components wrap Web services because our SOFA framework can only introspect SCA components. On the other hand, the `REST Handler` component supports the detection of REST (anti)patterns by (1) wrapping each REST API with an SCA component and (2) automatically applying the detection heuristics on the SCA-wrapped RESTful APIs. This wrapping allows us to introspect each request and response at runtime by using a FraSCAti IntentHandler.

5.3   Discussion

Our proposed unified detection approach—SODA—involves three main steps from the specification of service antipatterns to the automatic generation (or manual implementation) of detection algorithms, and, finally, the detection of service antipatterns by applying automatically generated (or implemented) detection algorithms on SCA systems, SOAP Web services, or on RESTful APIs. For the specification of service antipatterns, we rely on a common service domain specific language, which was designed after a thorough domain analysis with the available resources, *i.e.*, research articles, proceedings, Web sites, and blogs of SOA

practitioners.

Our SCA-based unified detection framework, SOFA, is in the core of the detection process. We chose SCA as our framework implementation technology because SCA encompasses many other service/interface types including Java RMI, SOAP, REST, JSON-RPC, JNA, and UPnP, and it facilitates introspection of any SCA-wrapped components where the components can wrap a REST API, or a Web service, or even an SCA component itself.

5.3.1 Extension of the DSL for REST Antipatterns

Although, for REST-specific antipatterns (in Section 5.2.1) we relied on detection heuristics and implemented their concrete detection algorithms manually, the generation process can be automated by taking the advantage of same model-driven engineering approach. For such extension, we extend our proposed language and its underlying grammar (see Section 5.2.1). To extend the proposed BNF grammar, we introduce new metrics, operators, comparators, and rule constructs. We can thus convert our heuristics into rule cards. In this dissertation, we list some new REST-specific metrics and provide some rule cards that are translated from the corresponding heuristics.

Table 5.2: List of 12 REST-specific Metrics for Specifying Antipatterns in RESTful APIs.

Metrics	Full Names
AC	Authentication Cookie
AK	Action Keywords
CC	Client Cookie
CCV	Client Caching Value
HL	Header Link
HM	Http Method
RRF	Resource Representation Format
SC	Server Cookie
SCV	Server Caching Value
TLB	Total number of Links in response Body
VRB	Verbs in Request Body
VRU	Verbs in Request URI

For example, Table 5.2 shows the list of 12 metrics that we use to specify REST antipatterns. However, this is not an exhaustive list of metrics and addition of new antipatterns may require to add or define new metrics. In the following, we give an example of the heuristic for

Forgetting Hypermedia antipattern and its corresponding translated rule card. We provide the rule cards of five other REST antipatterns in Appendix B (see P. 146).

```
1: FORGET-HYPER-MEDIA(response-header, response-body, http-method)
2:     body-links[] ← EXTRACT-ENTITY-LINKS(response-body)
3:     header-link ← response-header.getValue("Link")
4:     if(http-method = GET and (length(body-links[]) = 0 or header-link = NIL)) or
5:     (http-method = POST and ("Location:" ∉ response-header.getKeys() and
6:      length(body-links[]) = 0))) then
7:         print "Forgetting Hypermedia detected"
8:     end if
```

```
1 RULE_CARD: ForgetHyperMedia {
2     RULE: ForgetHyperMedia { UNION GetRequestLink PostRequestLink };
3     RULE: GetRequestLink { INTER HttpMethodGet NoLinkGet };
4     RULE: PostRequestLink { INTER HttpMethodPost NoLinkPost };
5     RULE: NoLinkGet { UNION NoBodyLink NoHeaderLink };
6     RULE: NoLinkPost { INTER NoBodyLink NoLocationHeader };
7     RULE: HttpMethodGet { HM = 'GET' };
8     RULE: HttpMethodPost { HM = 'POST' };
9     RULE: NoBodyLink { TLB = 0 };
10    RULE: NoHeaderLink { HL = NULL };
11    RULE: NoLocationHeader { 'Location' ∉ ResponseHeader };
12 };
```

Figure 5.10: Heuristic of *Forgetting Hypermedia* antipattern (top) and the Corresponding Rule Card (bottom).

As our future work, we plan to translate all the heuristics from this dissertation into rule cards and automatically generate their corresponding detection algorithms. This transformation ensures the full use of our SODA approach, which relies on a model-driven engineering technique and benefit from our unified abstraction as sound basis to build the grammar of the rule cards.

With our elaborated unified approach, a thorough validation will rationalise its effectiveness and efficiency. The next chapter discusses the thorough validation of our proposed unified SODA approach where we state our four experimental assumptions, the subjects, the objects, the overall validation process, and, finally, discuss the antipatterns detection results in details.

CHAPTER 6   VALIDATION

6.1 Chapter Overview

This chapter discusses the validation of the proposed unified SODA approach (Service Oriented Detection for Antipatterns) through a series of experiments. The goals of our experiments are, despite of differences and commonalities among the various SBSs technologies, to show that we can: (1) specify service antipatterns using the proposed domain specific language (DSL) and that our DSL is extensible for new antipatterns and SBSs technologies and (2) efficiently detect antipatterns across various SBSs technologies in terms of the accuracy and performance of the detection algorithms. Section 6.2, 6.3, and 6.4 describe respectively the assumptions, subjects, and objects of our experiments. Section 6.5 describes the overall validation process. Section 6.6 presents detailed detection results for all service antipatterns while Section 6.7 discusses our experimental assumptions. Finally, Section 6.8 explores the threats to the validity of our results.

6.2 Assumptions

The objective of our experiments is to positively support the four following assumptions through which we can answer our first research question **RQ**$_1$: "*Can we efficiently specify and detect service antipatterns in different development technologies and architectural styles of service-based systems in terms of detection accuracy and performance?*"

A1. Generality: *Our DSL allows the specification of different service antipatterns, from simple to more complex ones, in various SBSs technologies using the proposed unified abstraction.* This assumption supports the applicability of SODA using the rule cards on 31 service antipatterns, composed of 27 static and dynamic metrics.

A2. Accuracy: *Our automatically generated (or implemented) detection algorithms have a precision and recall of greater than 75%, i.e., more than three-quarters of detected antipatterns are true positive and more than three-quarters of all existing antipatterns are detected, respectively.* Given the trade-off between precision and recall, we assume that 75% precision is significant enough with respect to 75% recall. This assumption supports the precision of the rule cards and the accuracy of the algorithm generation and of the SOFA framework.

A3. Extensibility: *Our DSL and the proposed framework, SOFA is extensible for adding new service metrics and technology-specific or technology-neutral antipatterns.* Through this

assumption, we show how well the DSL, and in particular the metrics, with the supporting SOFA framework, can be combined to specify and detect new antipatterns.

A4. Performance: *The computation time required for the detection of service antipatterns using the generated (or implemented) algorithms is reasonably very low, i.e., in the order of seconds in various technologies.* This assumption supports the performance of the services provided by the SOFA framework for the detection of antipatterns.

6.3   Subjects

In our validation, we use 31 service antipatterns and 10 patterns from the domains of SCA, Web services, and REST from the literature [Dudney *et al.* (2003); Král et Žemlička (2008); Rotem-Gal-Oz *et al.* (2012); Jones (2006); Cherbakov *et al.* (2006); Modi (2006); Rodriguez *et al.* (2010a); Evdemon (2005); Hess *et al.* (2004); Tilkov (2008); Erl *et al.* (2012); Pautasso (2009); Fredrich (2012); Berners-Lee *et al.* (2005)], which are commonly found and well-explained with related examples. The list of service antipatterns that we analysed and detected is presented in Table 2.2. We put a brief description for all the service (anti)patterns and their specifications (for SCA and Web services) or heuristics (for REST) in Appendix A.

6.4   Objects

For our validation, we use the following systems:

– **SCA:** We perform experiments on two SCA systems: *Home-Automation* [FraSCAti (June 2013)] and *FraSCAti* [Seinturier *et al.* (2012)]. The *Home-Automation* is developed independently in Spirals Project-Team, INRIA, France to simulate a digital home for controlling basic household tasks remotely. *Home-Automation* is designed with 13 SCA components, each providing unique services with seven use cases. *FraSCAti* has a total of 91 SCA components providing 130 distinct services. To the best our knowledge, *FraSCAti* is currently the largest open-source SCA system implementing the SCA standard. We present the design of *Home-Automation* and *FraSCAti* in Appendix B. The detailed design of *FraSCAti* system can be found on `http://frascati.ow2.org/doc/1.4/ch12s04.html`.

– **Web services:** Since most of Web services are proprietary, it is hard to find freely available services for the validation. However, some existing Web services search engines, *e.g.*, `eil.cs.txstate.edu/ServiceXplorer` or `programmableweb.com`, facilitate search for service-interfaces and are limited in number and have the risk of

Table 6.1: List of 15 RESTful APIs with Their Online Documentations.

RESTful APIs	Online Documentations
Alchemy	alchemyapi.com/api
BestBuy	developer.bestbuy.com/documentation
Bitly	dev.bitly.com/api.html
CharlieHarvey	charlieharvey.org.uk/about/api
Dropbox	dropbox.com/developers/core/docs
Externalip	api.externalip.net
Facebook	developers.facebook.com/docs/graph-api
Instagram	instagram.com/developer
Musicgraph	developer.musicgraph.com/api-docs/overview
Ohloh	github.com/blackducksw/ohloh_api
StackExchange	api.stackexchange.com.docs
TeamViewer	integrate.teamviewer.com/en/develop/documentation
Twitter	dev.twitter.com/rest/public
YouTube	youtube.com/yt/dev/api-resources.html
Zappos	developer.zappos.com/docs/api-documentation

    providing broken or dislocated results. We performed our validation with more than 120 Web services collected from `programmableweb.com`. The list of all Web services interfaces is available in Appendix C.

– **REST:** We use 15 widely-used and popular RESTful APIs that we found well-documented as shown in Table 6.1. These 15 REST experimental objects well-define their underlying HTTP methods, service end-points, authentication details, and client-request parameter details.

6.5 Overall Process

    Using the SOFA framework, we generated the detection algorithms corresponding to the rule cards (for SCA and Web services) or implemented the detection algorithms corresponding to the detection heuristics (for REST) of the 31 service antipatterns. Then, we applied those detection algorithms at run-time on the target SBSs. Finally, we validated the detection results by analysing the suspicious services manually (1) to validate that these suspicious services are true positives, and (2) to identify false negatives (if any), *i.e.*, missing antipatterns. To validate the results on Web services and SCA we involved graduate students who are not the part of any experiment steps. As for the validation on *FraSCAti*, the core developers from the FraSCAti team assisted us. To measure the accuracy, we use the mea-

sures of precision, recall, and F_1-measure. Precision estimates the ratio of true antipatterns identified among the detected antipatterns (*cf.* Equation 6.1), while recall estimates the ratio of detected antipatterns among the existing antipatterns (*cf.* Equation 6.2). F_1-measure is the harmonic mean of precision and recall, to conclude the detection accuracy with a single value (*cf.* Equation 6.3).

$$precision = \frac{|\{existing\_antipatterns\} \cap \{detected\_antipatterns\}|}{|\{detected\_antipatterns\}|} \tag{6.1}$$

$$recall = \frac{|\{existing\_antipatterns\} \cap \{detected\_antipatterns\}|}{|\{existing\_antipatterns\}|} \tag{6.2}$$

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{6.3}$$

As for the validation of REST detection results, we performed the validation in two phases: (1) all the Dropbox URIs and (2) four representative APIs, *i.e.*, Facebook, Twitter, Dropbox, and YouTube, for which we randomly selected some candidate request URIs detected as patterns or antipatterns. We chose those four APIs based on our findings in [Palma *et al.* (2014a)], which concluded that Twitter and Dropbox are more problematic APIs, whereas Facebook and YouTube were well-designed.

We involved three professionals manually evaluated the URIs to identify the true positives and false negatives. The professionals have knowledge on REST and did not take part in the detection step. We provided them with the descriptions of REST linguistics (anti)patterns and the sets of all requests URIs collected during the service invocations. We resolved conflicts at the majority.

Due to the large size of the data-sets, we performed the validation on two sample sets because it is a laborious task to validate all APIs and all (anti)patterns and because Facebook, Dropbox, Twitter, and YouTube are representative APIs [Palma *et al.* (2014a)]. Therefore, in the first phase, we choose one medium sized API, Dropbox, to calculate the recall on one API (the entire validation would have required 1,545 questions for 309 test methods). In the second phase, we randomly selected 50 validation questions (out of 630 possible candidates) to measure overall accuracy. Here, we also used precision and recall to measure the detection accuracy.

6.6 Detection Results and Discussions

In this section, we discuss detection results of different groups of antipatterns presented in Table 2.2 (see Section 2.5). More specifically, Section 6.6.1 presents results on the detection of service antipatterns commonly found in SCA, Web services, and REST. Section 6.6.2 details the detection results of service antipatterns common in SCA and Web services while Section 6.6.3 presents results for service antipatterns in Web services and REST. Finally, Sections 6.6.4, 6.6.5, and 6.6.6 present the detection results of service antipatterns exclusively found in SCA, Web services, and REST only. A more detailed representation of the detection results for all 31 service antipatterns is available on our Web site `http://sofa.uqam.ca/soda/`.

6.6.1 Detection of Antipatterns Common in SCA, Web services, and REST

In this section, we detail the detection of three service antipatterns commonly found in SCA, Web services, and REST, namely *Ambiguous Name*, *Nobody Home*, and *Bloated Service* antipatterns. Table 6.2 shows the services involved in the service antipatterns (Column 3), the associated metrics and their values (Column 4), required detection times (Column 5). The last three columns in Table 6.2 show the precision, recall, and F1-measure.

Ambiguous Name: The `AIP3_PV_ImpactCallback` as reported in Table 6.2 was detected as the *Ambiguous Name* antipattern. This Web service offers operations with a set of signatures that (1) are extremely long (`ALS`=0.675), (2) use high number of general terms for naming (`RGTS`=0.85), (3) contain many messages having verbs (`NVMS`=26), and (4) having multiple verbs or action names within a single signature (`NVOS`=7). In comparison to the median values (*e.g.*, median of `ALS`=0.463, `RGTS`=0.0, `NVMS`=6, and `NVOS`=3) as calculated by *BoxPlot* service, the computed values measure high. The Web services having *Ambiguous Name* antipattern—which represents poor interface elements naming with (1) very short or long identifiers, (2) too general terms as identifiers, and (3) improper use of verbs—are not published semantically and syntactically sound interface over the Web and, thus, impact the discoverability and the reusability of a service. Applying SODA can help service developers in detecting and refactoring *Ambiguous Name* antipattern in SBSs.

Nobody Home: We detected `NativeCompiler`, `ServletManager`, `WsdlCompiler`, and `BPELEngine` components from *FraSCAti* as *Nobody Home* antipatterns. The SCA component, `BPELEngine`, for example, is detected as an antipattern because its implementation does not support the weaving of sensors and triggers to introspect at runtime [Seinturier

et al. (2012)], which is discarded by default, by the *FraSCAti* design. The `UselessService` component in *Home-Automation* is also identified as *Nobody Home* antipattern because it was not invoked in any executed scenarios (*i.e.*, `NIR`=0 and `NMI`=0) even though the component was orchestrated with other components in *Home-Automation*. The presence of such components or services not in use within an SBS may increase its maintenance cost. In REST, we did not detect corresponding *Deprecated Resources* antipattern because we implemented only a part of entire resources and tested them all.

Table 6.2: Detection Results of the Three Service Antipatterns: *Ambiguous Name*, *Bloated Service*, and *Nobody Home* commonly found in the three SBSs Implementation Technologies SCA, Web services, and REST.

Service Antipatterns	Applicable SBS Technology	Identified Service(s)	Metrics/Occurrences	Detection Time	Precision	Recall	F$_1$
Ambiguous Name	Web services	AIP3 PV ImpactCallback	ALS=0.675;RGTS=0.85;NVMS=26;NVOS=7;	0.855s	[9/9] 100%	[9/9] 100%	100%
		Bliquidity	ALS=0.576;RGTS=0.682;NVMS=42;NVOS=7;				
		CurrencyServerWebService	ALS=0.136;RGTS=0.682;NVMS=42;NVOS=5;				
		… …	… …				
		ProhibitedInvestorsService	ALS=0.158;RGTS=0.684;NVMS=12;NVOS=4;				
	SCA	*none detected*	*n/a*				
	REST	*none detected*	*n/a*				
Bloated Service	Web services	*none detected*	*n/a*	0.071s	[3/3] 100%	[3/3] 100%	100%
	SCA *(FraSCAti)*	component-factory	NOI=1;NMD=7;TNP=12;COH=0.066;				
		factory	NOI=1;NMD=7;TNP=12;COH=0.066;				
		frascati-binding-http	NOI=1;NMD=5;TNP=8;COH=0.065;				
	REST	*none detected*	*n/a*				
Nobody Home	Web services	*none detected*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*
	SCA *(Home-Automation)*	UselessService	NIR>0;NMI=0;	0.606s	[4/5] 80%	[4/4] 100%	88.89%
	SCA *(FraSCAti)*	NativeCompiler	COH=0.1;NMD=5;RT=1018ms;				
		ServletManager	NMI=0;NIR>0;				
		WsdlCompiler	NMI=0;NIR>0;				
		BPELEngine	NMI=0;NIR>0;				
	REST	*none detected*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*
Average				**0.511s**	**[16/17] 93.33%**	**[16/16] 100%**	**96.3%**

The next section presents the detection results of eight service antipatterns commonly found in SCA and Web services.

6.6.2   Detection of Antipatterns Common in SCA and Web services

Table 6.3 presents the detection results of eight service antipatterns commonly found in SCA and Web services. Table 6.3 shows the services involved in the service antipatterns (Column 3), the associated metrics and their values (Column 4), required detection times (Column 5). The last three columns in Table 6.3 show the precision, recall, and F1-measure.

In the below, *Bottleneck Service*, *Service Chain*, *Multi Service*, and *Tiny Service* antipatterns are discussed in detail.

Bottleneck Service: The `sca-composite` and `sca-parser` services in *FraSCAti* were detected as *Bottleneck Service* with a high response time (*i.e.*, `RT` values are 41ms and 45ms, respectively) and a very high coupling (*i.e.*, `CPL` values are 0.96 and 0.84). The coupling `CPL` was calculated by means of `NIR` and `NOR`, which are also very high for these two services. As estimated by the *Boxplot* service, the median for `CPL` and `RT` were 0.68 and 5ms respectively. After manually analysing the FraSCAti design, the engineer found that only the `sca-parser` service is highly used by other services, and validated `sca-parser` as the only *Bottleneck Service*. The *FraSCAti* development team also agreed with this manual detection result, and confirmed that if too many external clients try to invoke the `sca-parser` service, they might wait to get the access. The precision and recall for *Bottleneck Service* are 75% and 100% respectively (see Table 6.3), with one false positive. The `sca-composite` is detected as *Bottleneck Service* and is a false positive again because of the value of availability (`A`), which is always very high, *i.e.*, 100%. For an SBS with several Composites, while the *FraSCAti* invokes the `sca-parser` service to parse all the relevant Composites, the `sca-parser` service naturally has the low availability. This low availability is mainly due to multiple invocations simultaneously by several clients to parse their Composites using the same `sca-parser`. Since, we had the availability (`A`) of 100% for all the components and did not consider the value of availability while reporting suspicious services, the `sca-parser` was reported as *Bottleneck Service*.

Service Chain: In *Home-Automation*, we detected a consecutive chain of invocations of `IMediator` → `SunSpotService` → `PatientDAO` → `PatientDAO2`, which forms a *Service Chain*, whereas engineers validated `IMediator` → `PatientDAO` → `PatientDAO2`. The `SunSpotService` was not validated by the engineers and thus, it was considered as a false

positive. However, the detected chain exists in the system but only in one scenario. Engineers did not consider the full chain as harmful and therefore did not classify it as an antipattern.

We also identified {`MembraneGeneration, TypeFactory,` and `Processor`} as involved in a *Service Chain* in *FraSCAti*. The `BPELEngine` was detected as a *Nobody Home* antipattern because its implementation does not support the weaving of non-functional code such as sensors and triggers. In fact, all the BPEL implementations of *FraSCAti* are, by default, discarded from the weaving algorithms provided by *FraSCAti*. We also detected one false positive for the *Service Chain* (`Processor` → `Processor` → `BindingFactory` → `PluginResolver`) that was not confirmed by the engineers. This detection was due to the fact that the `Processor` service calls itself using a public method and artificially extends the chain of calls. We may consider the modification of the *Service Chain* detection algorithm in order to eliminate self calls.

In summary, our detection algorithms did not detect any *Duplicated Service*, *Chatty Service*, *Sand Pile*, and *Data Service* antipatterns in *FraSCAti*. The absence of *Sand Pile* and *Chatty Service* were obvious, as they are related to *Data Service* and evidently, there were no such antipatterns in *FraSCAti*, *i.e.*, as confirmed by the *FraSCAti* team and our detection results. Our detection algorithms did not identify any suspicious services, after calculating the metric values we define in rule cards for those antipatterns. Indeed, as validated by FraSCAti team, there were no *Duplicated Service* and *Data Service* antipatterns in *FraSCAti*, and our detection algorithms did not filter any services as false positives either.

Indeed, very few services, *i.e.*, 10 were actually involved in 6 antipatterns (4 antipatterns are not present) in *FraSCAti*, in comparison to the high number of services, *i.e.*, 130 services. *FraSCAti* is well designed with continuous maintenance and evolution. Mostly, services (*e.g.,* `sca-parser` and `sca-composite-*`) related to parsing and handling the composite file were involved in the antipatterns. The presence of such antipatterns in a system is not surprising because there is no other way to develop a parser without introducing a high coupling among services.

Table 6.3: Detection Results of the Eight Service Antipatterns Commonly Found in the Two SBSs Implementation Technologies SCA and Web services.

Service Antipatterns	Applicable SBS Technology	Identified Service(s)	Metrics/Occurrences	Detection Time	Precision	Recall	F₁
Bottleneck Service	SCA (Home-Automation)	IMediator	NIR=7;NOR=7;CPL=1.0;RT=40ms;	0.167s	[3/4] 75%	[3/3] 100%	85.71%
		PatientDAO	NIR=4;NOR=4;CPL=0.57;RT=2ms;				
	SCA (FraSCAti)	sca-composie	RT=41ms;CPL=0.96;NIR=16;NOR=8;				
		sca-parser	RT=45ms;CPL=0.84;NIR=16;NOR=5;				
	Web services	none detected	n/a				
Chatty Service	SCA (Home-Automation)	IMediator	ANP=1.0;ANPT=1.0;NMI=3;ANAM=100%;COH=0.167;	1.14s	[3/4] 75%	[3/3] 100%	85.71%
		PatientDAO	ANP=1.0;ANPT=1.0;NMI=3;ANAM=100%;COH=0.167;				
	Web services	ForeignExchangeRates	COH=0.16;ANAO=50;NOD=24;RT=3286ms;		[1/2] 50%	[1/1] 100%	66.67%
		TaarifCustoms	COH=0.12;ANAO=72.22;NOD=18;RT=4105ms;				
Data Service	SCA (Home-Automation)	PatientDAO	ANAM=100%;COH=0.167;ANPT=1.0;ANP=1.0;	0.268s	[1/1] 100%	[1/1] 100%	100%
	SCA (FraSCAti)	none detected	n/a	n/a	n/a	n/a	n/a
	Web services	none detected	n/a	n/a	n/a	n/a	n/a
Duplicated Service	SCA (Home-Automation)	CommunicationService vs. IMediator	ANIM=25%	0.215s	[2/2] 100%	[2/2] 100%	100%
	SCA (FraSCAti)	none detected	n/a	n/a	n/a	n/a	n/a
	Web services	none detected	n/a	n/a	n/a	n/a	n/a
Multi Service	SCA (Home-Automation)	IMediator	COH=0.027;NMD=13;RT=132ms;	78.67s	[2/2] 100%	[2/3] 66.67%	74.63%
	SCA (FraSCAti)	juliac	COH=0.1;NMD=5;RT=1018ms;				
		Explorer-GUI	n/a				
	Web services	none detected	n/a				
Tiny Service	SCA (Home-Automation)	MediatorDelegate	NOR=4;CPL=0.44;NMD=1;	0.194s	[6/7] 85.71%	[6/6] 100%	92.31%
	SCA (FraSCAti)	sca-parser	NMD=1;CPL=0.56;	0.067s			
	Web services	SrtmWsPortType	NOD=2;COH=0.0;	0.945s			
		Hydro1KWsPortType					
		ShadowWsPortType					
		XigniteTranscripts	NOD=4;COH=0.125;				
		BGCantorUSTreasuries	NOD=3;COH=0.083;				
Service Chain	SCA (Home-Automation)	Imediator > SunSpotService > PatientDAO > PatientDAO2	NTMI=4	0.143s	[3/4] 75%	[3/3] 100%	85.71%
	SCA (FraSCAti)	MembraneGeneration > rocessor > ComponentFactory > MembraneGeneration	NTMI=4				
		TypeFactory > Processor > ComponentFactory > MembraneGeneration	NTMI=4				
		Processor > Processor > ComponentFactory > MembraneGeneration	NTMI=4				
	Web services	none detected	n/a	n/a	n/a	n/a	n/a
Stovepipe Service	SCA	none detected	n/a	n/a	n/a	n/a	n/a
	Web services	none detected	n/a	n/a	n/a	n/a	n/a
Average				9.09s	[21/26] 82.59%	[21/22] 95.83%	86.34%

Multi Service and Tiny Service: From Table 6.3, we briefly discuss detection results of *Tiny Service* and *Multi Service*. In particular, the `IMediator` SCA component was identified as a *Multi Service* antipattern in *Home-Automation* due to its very high number of interface methods (`NMD`=13) with a very low cohesion among its methods (`COH`=0.027) and a very high response time (`RT`=132ms). All these values were assessed as high or low by the *Boxplot* component in our SOFA framework compared to the values from other components in *Home-Automation*. For example, the *Boxplot* component estimated the median value of `NMD` in *Home-Automation* as 2, compared to which 13 is quite high. Similarly, the detected *Tiny Service* antipattern, *i.e.*, `MediatorDelegate`, has a very low number of methods (`NMD`=l) with a high coupling (`CPL`=0.44) with respect to other *Home-Automation* components. The cohesion (`COH`) and coupling (`CPL`) metrics range between 0 and 1.

We also detected `juliac` as *Multi Service* antipattern because of its very high response time (`RT`=1,018ms), low cohesion (`COH`=0.1), and high number of methods declared (`NMD`=5). The median values estimated by the *Boxplot* service, *i.e.*, median of `RT` is 4ms, `COH` is 0.1, and `NMD` is 1 compared to other services in *FraSCAti* classified `juliac` as *Multi Service*. `juliac` is likely a true positive because it implements six different features belonging to two different abstractions. Indeed, `juliac` provides services for the membrane generation and the Java compilers. Therefore, it is low cohesive and highly used because each component needs a membrane, and each composite file needs to be compiled by the Java compiler. Moreover, these actions are resource-consuming and requires more execution time than other services. Manual inspection by the engineer also validated this detection.

Subsequently, the inspection of *FraSCAti* also allowed the identification of `Explorer-GUI` as a *Multi Service*. The *FraSCAti* development team confirmed that this service uses a high number of other services provided by *FraSCAti*. Indeed, this component encapsulates the graphical interface of `FraSCAti Explorer`, which aims to provide an exhaustive interface of FraSCAti functionalities. SOFA was not able to detect it because the execution scenarios did not involve the graphical interface of `FraSCAti Explorer`. Therefore, with two services detected as true positives, and with one missing occurrence of *Multi Service*, *i.e.*, `Explorer-GUI`, we had a precision of 100% and recall of 66.67%.

We also detected `sca-parser` as the *Tiny Service* with its small number of methods (`NMD`=1) and a high coupling (`CPL`=0.56). The engineer and the FraSCAti team also validated this detection. The boxplot median values are respectively 1 and 0.11 for `NMD` and `CPL`. After the manual inspection of *FraSCAti* implementation, the independent engineer identified that `sca-parser` contains only one method, *i.e.*, `parse(QName `*`qname,`*` ParsingContext`

*parsingContex*t). In some cases, for a given metric, the median value and the high and–or low value might be identical if the values of most services are equal. For example, the median and low values are the same for `NMD` because out of 86 analysed services, 50 services have the `NMD` value of 1. While concerned about the coupling CPL, `sca-parser` has dependency references to five other services (*i.e.*, `sca-metamodel-*`) that give a high coupling. In fact, the coupling CPL values presented here were calculated on the logarithmic scale, *i.e.*, the more references a service has to other services, the more highly coupled it is. The FraSCAti development team also validated `sca-parser` as a *Tiny Service.* However, according to them, `sca-parser` is invoked alone when only a reading for an SCA composite file is requested. However, *FraSCAti* performs more tasks than just reading and–or parsing an SCA composite file, and these other tasks are performed by other services such as `AssemblyFactory`. These several delegation also explains the high outgoing coupling..

Moreover, we detected `SrtmWsPortType`, `ShadowWsPortType`, and `Hydro1KWsPortType` as *Tiny Service* antipatterns in Web services since they possessed very low values for `NOD` (*i.e.*, 2) and `COH` (*i.e.*, 0.0). As computed by the *Box-Plot* component in SOFA framework, `NOD` values of 2 are rather low compared to the median of 5.5. Moreover, the `COH` values are less-significant compared to other Web services whose `COH` values are in the range of 0.216 and 0.443. Such small services implemented as *Tiny Service* antipatterns often require other services to be used together, resulting in higher development complexity and reduced usability of Web services. The manual validation with the Web services interfaces confirmed the detection of this antipattern only for `ShadowWsPortType` and `Hydro1KWsPortType` Web services. Two other Web services, *i.e.*, `XigniteTranscripts` and `BGCantorUSTreasuries` were also detected as *Fine Grained Web Service.* Both those Web services have a very small number of operations defined (`NOD` is 3 and 4) and have a low cohesion (`COH` between 0.083 and 0.125), compared to the maximum values (*i.e.*, 70 for `NOD`, and 0.667 for `COH`) from other Web services. However, no occurrences of *Multi Service* were detected in Web services.

In the next section, we discuss two service antipatterns commonly found in the Web services and REST domains.

6.6.3 Detection of Antipatterns Common in Web services and REST

Table 6.4 presents detection results of service antipatterns in Web services and REST. The table shows the services involved in the service antipatterns (Column 3), the associated metrics and their values or their occurrences (Column 4), required detection times (Column 5). The last three columns in Table 6.4 show the precision, recall, and F1-measure. In the below, we discuss *CRUDy Interface* and *CRUDy URIs* antipatterns in detail.

CRUDy Interface and CRUDy URIs: `ForeignExchangeRates` and `TaarifCustoms` are both identified as *Chatty Web Service* and *CRUDy Interface* antipatterns because of their low cohesion (`COH`=0.16 and 0.12, respectively), high average number of accessor operations (50≤`ANAO`≤72.22), high number of operations (18≤`NOD`≤24), and high response time (`RT`>3s), compared to other Web services. The manual validation did not confirm `ForeignExchangeRates` as a *Chatty Web Service* because the order of operations invocation could be inferred from the service's interface. The specification of *CRUDy Interface* includes *Chatty Web Service*. Therefore, the detection of `ForeignExchangeRates` as a *CRUDy Interface* was not confirmed.

Thus, having a chatty Web service in an SBS, which exhibits low cohesion among its operations and causes high response time, impacts the maintainability (since, inferring the order of invocation is difficult and many interactions are required) and the overall performance. SODA can automatically detect such bottleneck services within SBSs and, therefore, facilitates their maintenance.

Table 6.4: Detection Results of the Two Service Antipatterns: *CRUDy Interface* and *CRUDy URI* Commonly Found in the Two SBSs Implementation Technologies Web services and REST.

Service Antipatterns	Applicable SBS Technology	Identified Service(s)	Metrics/Occurrences	Detection Time	Precision	Recall	F_1
CRUDy Interface	Web services	ForeignExchangeRates	COH=0.16;ANAO=66.67;NOD=24;RT=3113ms;NCO=9;	37.23s	[1/2] 50%	[1/1] 100%	66.67%
		TaarifCustoms	COH=0.12;ANAO=72.22;NOD=18;RT=4105ms;NCO=18;				
CRUDy URI	REST	DropBox	POST /fileops/move POST /1/fileops/delete POST /1/fileops/create_folder POST /1/fileops/copy	0.737s	[3/3] 100%	[3/3] 100%	100%
		StackExchange	GET /2.2/suggested-edits				
		Twitter	GET /1.1/users/show.json POST /1.1/statuses/update.json GET /1.1/statuses/show.json POST /1.1/account/update_profile_colors.json				
Average				18.98s	[4/5] 75%	[4/4] 100%	83.33%

As shown in Table 6.4, the occurrences of *CRUDy URI* antipattern in RESTful APIs were detected in only 4% (12 out of 309) tested URIs. In contrast, 93% (287 out of 309) of the tested URIs are *Verbless URI*—the corresponding REST pattern. In other words, APIs designers seem aware of not mixing the definition of traditional Web service operations and resource-oriented HTTP requests in REST. In traditional Web services, operation identifiers

reflect what they are doing, whereas in REST, actions to be performed on a resource should be explicitly mentioned only using HTTP methods and not within a URI through a CRUDy term.

In the next section, we discuss three service antipatterns commonly found only in the SCA technology.

6.6.4 Detection of SCA-specific Antipatterns

Table 6.5 presents the detection results of eight service antipatterns commonly found in SCA and Web services. The table shows the services involved in the service antipatterns (Column 3), the associated metrics and their values (Column 4), required detection time (Column 5). The last three columns in Table 6.5 show the precision, recall, and F1-measure.

As shown in Table 6.5, the `Home-Automation` itself was detection as *Sand Pile* antipattern by our detection algorithm. After the manual investigation, we found that it contained a high number of service (`NCS=13`) with an average number of parameter per operation and an average number of primitive-type parameter per operation as very high. Moreover, the container service and the contained (child) services were highly cohesive, in our case, `COH=0.17`, which is high in comparison to other services. Moreover, `IMediator` and `PatientDAO` components in *Home-Automation* system are detected as *The Knot* antipatterns. Those services exhibit a very low cohesion (`COH=0.027`), a very strong coupling with partner services (`CPL=1`), and a high response time (`RT=57ms`) in comparison to other services. In Table 6.5, our detection algorithm reported `PatientDAO` service as *The Knot* antipattern because the availability (`A`) value was discarded, which was high, *i.e.*, 100%.

Table 6.5: Detection Results of the Three Service Antipatterns: *God Component*, *Sand Pile*, and *The Knot* Commonly Found in SCA.

Service Antipatterns	Applicable SBS Technology	Identified Service(s)	Metrics/Occurrences	Detection Time	Precision	Recall	F$_1$
God Component	SCA *(FraSCAti)*	FraSCAti	NOSE=6;NMD=12;TNP=12	0.069s	[2/2] 100%	[2/2] 100%	100%
		component-factory	NOSE=5;NMD=7;TNP=12				
Sand Pile	SCA *(Home-Automation)*	HomeAutomation	NCS=13;ANP=1;ANPT=1;ANAM=100%;COH=0.17	0.184s	[1/1] 100%	[1/1] 100%	100%
The Knot	SCA *(Home-Automation)*	IMediator	COH=0.027;NIR=7;NOR=7;CPL=1.0;RT=57ms	0.412s	[2/3] 66.67%	[2/2] 100%	80%
		PatientDAO	COH=0.027;NIR=7;NOR=7;CPL=1.0;RT=57ms				
	SCA *(FraSCAti)*	sca-parser	CPL=0.84;COH=0.08;RT=44ms	0.07s			
Average				0.184s	[5/6] 88.89%	[5/5] 100%	93.3%

In the next section, we discuss three service antipatterns commonly found in the domain of Web services.

6.6.5   Detection of Web services-specific Antipatterns

Table 6.6 presents the detection results of eight service antipatterns commonly found in SCA and Web services. The table shows the services involved in the service antipatterns (Column 3), the associated metrics and their values (Column 4), required detection time (Column 5). The last three columns in Table 6.6.5 show the precision, recall, and F1-measure. In the below, the *Redundant PortTypes* antipattern is discussed in detail.

Redundant PortTypes: As shown in Table 6.6, we identified `wsIndicadoresEconomicos*` groups of PortTypes as the *Redundant PortTypes* antipatterns in Web services with multiple identical port-types (*i.e.*, `NPT`>1 and `NOPT`>1) defined in their service interfaces, thus have `ARIP`=1.0, *i.e.*, a very high value compared to the median of 0.465. If a Web service has redundant *port-types*, it is a good practice to merge them, while making sure that this merge does not introduce a *God Object Web Service* antipattern.

Low Cohesive Operations: We detected seven other Web services as *Low Cohesive Operations* antipatterns (see Table 6.6). A Web service is said to have low cohesive operations when it declares a large number of low-cohesive operations, which affects the comprehension, reusability, and overall, the maintainability of the Web service. For example, in Table 6.6, eight Web services were detected as *Low Cohesive Operations* antipatterns, which had from 12 to 37 operations defined in their interfaces. Moreover, they were not very cohesive, both syntactically and semantically, with the `ARIO` values ranging between 0.177 and 0.268. The maximum value of `ARIO` is 1 and such low values of eight Web services suggests that their interfaces were not well-designed, which may hinder their discoverability.

Table 6.6: Detection Results of the Three Service Antipatterns: *Low Cohesive Operations, May be It's Not RPC,* and *Redundant PortTypes* Commonly Found in Web services.

Service Antipatterns	Applicable SBS Technology	Identified Service(s)	Metrics/Occurrences	Detection Time	Precision	Recall	F_1
Low Cohesive Operations	Web services	ndfdXMLPortType	NOD=12;ARIO=0.221;	121.81s	[8/8] 100%	[8/8] 100%	100%
		ServiceSoap	NOD=24;ARIO=0.253;				
		XigniteSecuritySoap	NOD=25;ARIO=0.177;				
		… …	… …				
		XigniteSecurityHttpPost	NOD=25;ARIO=0.177;				
		XigniteCorporateActionsSoap	NOD=37;ARIO=0.268;				
May be It's Not RPC	Web services	*none detected*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*
Redundant PortTypes	Web services	AIP3 PV Impact vs. AIP3 PV ImpactCallback	NOPT=9;ARIP=0.378;	167.62s	[5/5] 100%	[5/5] 100%	100%
		wsIndicadoresEconomicosHttpPost vs. wsIndicadoresEconomicosSoap vs. wsIndicadoresEconomicosHttpGet	NOPT=2;ARIP=1.0;				
Average				**144.72s**	**[13/13] 100%**	**[13/13] 100%**	**100%**

However, we did not detect any occurrences of *May be It's Not RPC* antipattern among the 123 Web services. This suggest that none of the analysed Web services provided CRUD operations with a large number of parameters. In general, the consequence of this antipattern is a poor system performance because the clients often wait for the synchronous responses.

In the next section, we discuss service antipatterns commonly found only in REST.

6.6.6 Detection of REST-specific Antipatterns

Table 6.8 presents detailed detection results for the eight REST antipatterns related to the syntactic design of REST requests/responses. The table reports the antipatterns in the first column followed by the analysed RESTful APIs in the following twelve columns. For each REST API and for each antipattern, we report: (1) the total number of validated true positives with respect to the total detected antipatterns by our algorithms, *i.e.,* the precision, in the first row and (2) the total number of detected true positives with respect to the total existing true positives, *i.e.,* the recall, in the following row. The last two columns show, for all APIs, the average precision-recall and the total detection time for each antipattern.

Since, we perform the detection for both REST antipatterns and patterns, some of the antipatterns can be mapped to their corresponding patterns. Table 6.7 shows the such mapping between REST antipatterns and patterns.

Table 6.7: The Mapping between REST Antipatterns and Patterns.

REST Patterns	Corresponding REST Antipatterns
Content Negotiation	Ignoring MIME Types
Contextualised Resource Names	Contextless Resource Names
End-point Redirection	-
Entity Endpoint	-
Entity Linking	Forgetting Hypermedia
Hierarchical Nodes	Non-hierarchical Nodes
Response Caching	Ignoring Caching
Singularised Nodes	Pluralised Nodes
Tidy URI	Amorphous URI
Verbless URI	CRUDy URI

A more detailed representation of the REST detection results is available on our Web site http://sofa.uqam.ca/soda/, which presents the detection results for each REST resource and each REST request from 15 RESTful APIs.

Overview of REST Request/Response Syntactic Antipatterns

Overall, RESTful APIs that follow patterns tend to avoid corresponding antipatterns and *vice-versa.* For example: BestBuy and Facebook are found involved respectively in 0 and 8 instances of *Forgetting Hypermedia* antipattern; however, these APIs are involved in 11 and 21 corresponding *Entity Linking* pattern. Moreover, DropBox, Alchemy, YouTube, and Twitter APIs had 27 instances of *Ignoring Caching* antipattern, but they were involved in 8 instances of the corresponding *Response Cashing* pattern. Finally, we found Facebook, DropBox, BestBuy, and Zappos APIs involved in only 3 instances of *Ignoring MIME Types* antipattern, which conversely are involved in more than 55 instances of corresponding *Content Negotiation* pattern.

In general, while detecting antipatterns related to REST request/response syntactic design, among the 12 analysed RESTful APIs with 115 methods tested and eight antipatterns, we found Twitter (32 instances of four antipatterns), DropBox (40 instances of four antipatterns), and Alchemy (19 instances of five antipatterns) are more problematic, *i.e.*, contain more antipatterns than others (see Figure 6.1). On the other hand, considering the five REST patterns, we found Facebook (49 instances of four patterns), BestBuy (22 instances of two patterns), and YouTube (15 instances of three patterns) are well designed, *i.e.*, involve more patterns than others (see Figure 6.1).
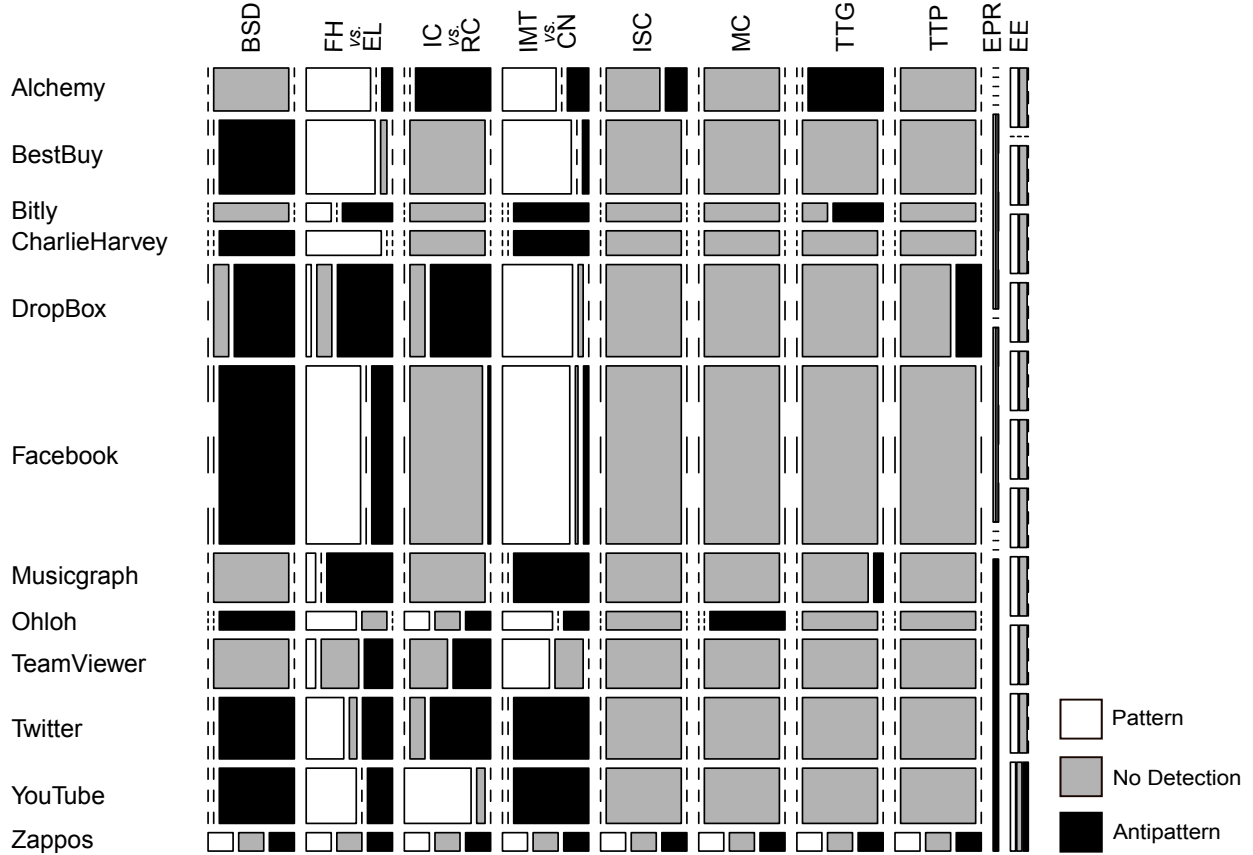
Figure 6.1: Overview on the Detection Results of REST Request/Response Syntactic Antipatterns. [BSD→Breaking Self-descriptiveness; FH→Forgetting Hypermedia; EL→Entity Linking; IC→Ignoring Caching; RC→Response Caching; IMT→Ignoring MIME Types; CN→Content Negotiation; ISC→Ignoring Status Code; MC→Misusing Cookies; TTG→Tunnelling Through GET; TTP→Tunnelling Through POST; EPR→End-point Redirection; EE→Entity Endpoint.]

In the below, *Breaking Self-descriptiveness*, *Forgetting Hypermedia*, *Ignoring Caching*, and *Ignoring MIME Types* REST antipatterns are discussed in detail.

Breaking Self-descriptiveness: REST developers tend to rely on their own customised headers, formats, and protocols, and thus introduce *Breaking Self-descriptiveness* antipattern. The analysis on the 12 RESTful APIs shows that developers used non-standard header fields and protocols in most APIs including BestBuy, DropBox, Facebook, and Twitter. For example, Facebook used `x-fb-debug` and `x-fb-rev` header fields, which are mainly used to track a request id for their internal bug management purpose. Similarly, we found Drop-

Box using the `x-dropbox-request-id` and Twitter using `x-tfe-logging-request-*` and `x-xss-protection` header fields. In general, the designers and implementers often distinguish the standardised and non-standardised header members by prefixing their names with "x-" (*a.k.a., eXperimental*). Indeed, the "x-" convention was highly discouraged by the Internet Society in RFC822 [RFC2822 (2001)]. The manual validation reveals that all our detection was true positives and we reported all existing non-standard header fields and protocols, except two in DropBox where the manual validation considered them as non-standard practice. This leads to the precision of 100% and the recall of 98.21% for this detection.

Table 6.8: Detection results of the Eight REST Antipatterns Related to the Syntactic Design of REST Requests/Responses Obtained by Applying Detection Algorithms on the 12 RESTful APIs (numbers in the parentheses show total test methods for each API).

REST API	(7)Alchemy	(12)BestBuy	(3)Bitly	(4)CharlieHarvey	(15)DropBox	(29)Facebook	(8)Musicgraph	(3)Ohloh	(8)TeamViewer	(10)Twitter	(9)YouTube	(7)Zappos	precision-recall	(115) Total	Average Precision-Recall	Detection Time
REST Antipatterns																
Breaking Self-descriptiveness	0/0	12/12	0/0	4/4	12/12	29/29	0/0	3/3	0/0	10/10	9/9	7/7	p	86/86	100%	21.31s
	0/0	12/12	0/0	4/4	12/14	29/29	0/0	3/3	0/0	10/10	9/9	7/7	r	86/88	98.21%	
Forgetting Hypermedia	1/1	0/0	2/2	0/0	9/10	8/8	7/7	0/0	3/3	4/4	2/3	0/0	p	36/38	94.58%	19.54s
	1/1	0/0	2/2	0/0	9/9	8/8	7/7	0/0	3/3	4/4	2/2	0/0	r	36/36	100%	
Ignoring Caching	7/7	0/0	0/0	0/0	12/12	1/1	0/0	1/1	4/4	8/8	0/0	0/0	p	33/33	100%	18.99s
	7/7	0/0	0/0	0/0	12/12	1/1	0/0	1/1	4/4	8/8	0/0	0/0	r	33/33	100%	
Ignoring MIME Types	2/2	1/1	3/3	4/4	0/0	2/2	8/8	0/0	0/0	10/10	9/9	0/0	p	39/39	100%	19.39s
	2/2	1/1	3/3	4/4	0/0	2/2	8/8	0/0	0/0	10/10	9/9	0/0	r	39/39	100%	
Ignoring Status Code	1/2	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	p	1/2	50%	21.22s
	1/2	0/0	0/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	r	1/3	25%	
Misusing Cookies	0/0	0/0	0/0	0/0	0/0	0/0	0/0	3/3	0/0	0/0	0/0	0/0	p	3/3	100%	19.1s
	0/0	0/0	0/0	0/0	0/0	0/0	0/0	3/3	0/0	0/0	0/0	0/0	r	3/3	100%	
Tunnelling Through `GET`	5/7	0/0	0/2	0/0	0/0	0/0	0/1	0/0	0/0	0/0	0/0	0/1	p	5/11	17.86%	28.26s
	5/5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	r	5/5	100%	
Tunnelling Through `POST`	0/0	0/0	0/0	0/0	5/5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	p	5/5	100%	28.64s
	0/0	0/0	0/0	0/0	5/5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	r	5/5	100%	
Average													p	208/217	82.81%	22.06s
													r	208/212	90.4%	

Forgetting Hypermedia: Any RESTful interaction is driven by *hypermedia*—by which clients interact with application servers via URL links provided by servers in resource representations [Fielding (2000)]. The absence of such interaction pattern is known as *Forgetting*

Hypermedia antipattern [Tilkov (2008)], which was detected in eight APIs, namely Bitly, DropBox, Facebook, and so on (see Table 6.8). Among the 115 methods tested, we found 38 instances of this antipattern. Moreover, RESTful APIs that do not have this antipattern well applied the corresponding *Entity Linking* pattern [Erl *et al.* (2012)], *e.g.*, Alchemy, BestBuy, and Ohloh, which is a good practice. This observation suggests that, in practice, developers sometimes do not provide hyper-links in resource representations. As for the validation, 36 instances of *Forgetting Hypermedia* antipattern were manually validated; therefore, we have an average precision of 94.58% and a recall of 100%. For *Entity Linking* pattern, the manual validation confirmed 66 instances whereas we detected a total of 65 instances, all of which were true positives. Thus, we had an average precision of 100% and a recall of 98.81%.

Ignoring Caching: Caching helps developers implementing high-performance and scalable REST services by limiting repetitive interactions, which if not properly applied violates one of the six REST principles [Fielding (2000)]. REST developers widely ignore the caching capability by using *Pragma: no-cache* or *Cache-Control: no-cache* header in the requests, which forces the application to retrieve duplicate responses from servers. This bad practice is known as *Ignoring Caching* antipattern [Tilkov (2008)].

In contrast, *Response Caching* [Erl *et al.* (2012)] REST pattern supports response cacheability. We detected six RESTful APIs that explicitly avoid caching capability, namely Alchemy, DropBox, Ohloh, and so on (see Table 6.8). On the other hand, cacheability is supported by YouTube and Zappos, which were detected as *Response Caching* patterns. The manual analysis of requests and responses also confirmed these detections, and we had an average precision and recall of 100% (see Table 6.10).

Ignoring MIME Types: We also detected *Ignoring MIME Types* in eight REST services. According to REST principle [Fielding (2000)], the server should represent resources in multiple formats, which allow clients a more flexible service consumption. Yet, the server-side developers often intend to have a single representation of resources (or rely on their own formats), which limits the resource and the underlying service accessibility and reusability. We detected ten instances of Ignoring MIME Types antipattern in Twitter and nine instances in YouTube. Moreover, we found Facebook, DropBox, and BestBuy, APIs involved in only three instances of Ignoring MIME Types antipattern, *i.e.*, they mostly support the good design practice of multiple resource representations.
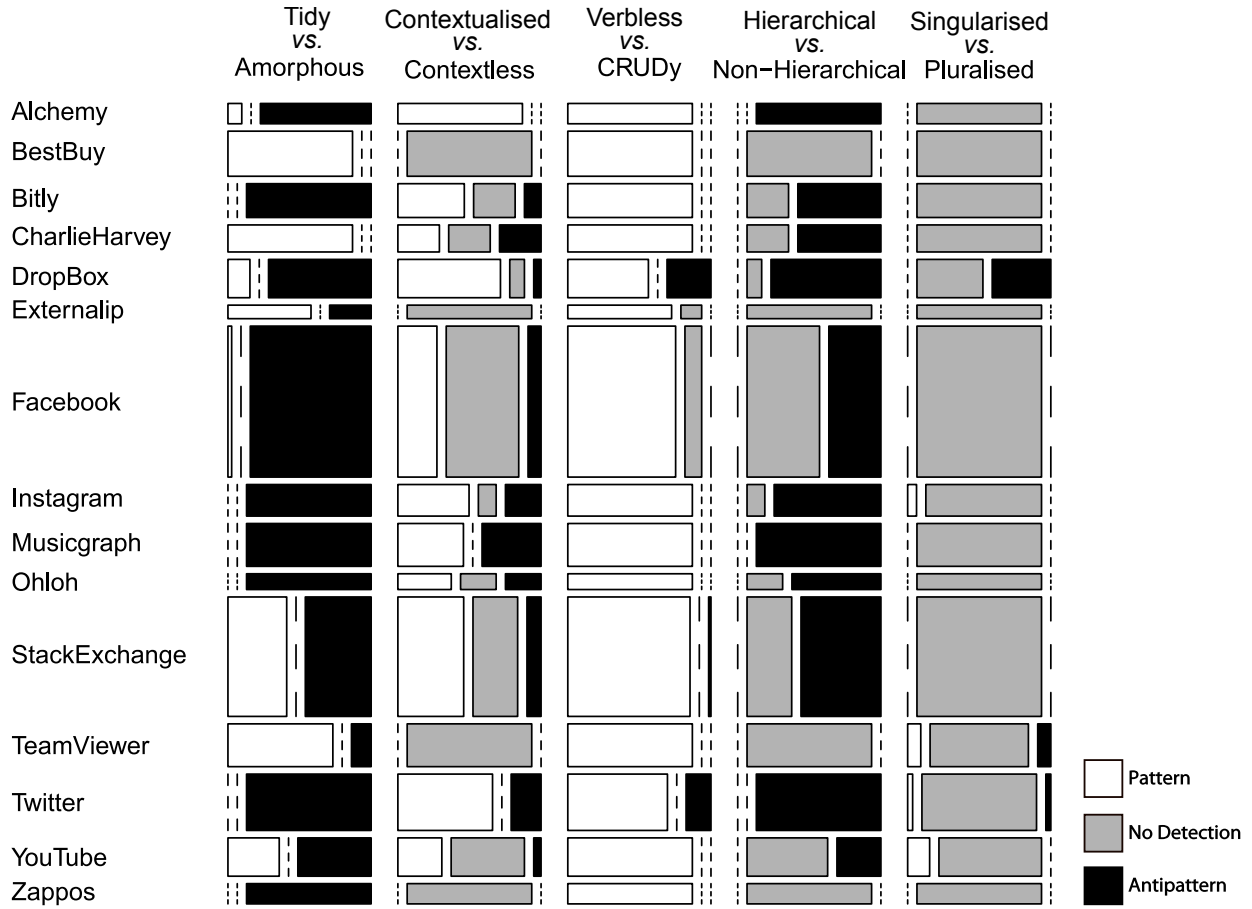
Figure 6.2: Linguistic (Anti)Patterns Detected in each REST API.

Overview of REST Linguistic Antipatterns

The mosaic plot in Figure 6.2 shows the pattern-wise representation of the detection results of ten linguistic patterns and antipatterns on the 15 RESTful APIs. Columns correspond to each (anti)pattern while rows represent the detected (anti)patterns on each API. In each row, the height of the mosaic represents the size of the method suite we tested for an API. In Figure 6.2, the most frequent patterns are *Verbless URI* and *Contextualised Resource Names*—the majority of the analysed APIs did not include any CRUDy terms or any of their synonyms and the nodes in these URIs belong to the same semantic context. In contrast, the most frequent antipatterns are *Amorphous URI* and *Non-Hierarchical Nodes*—the majority of the analysed APIs involve at least one syntactical problem and that URI nodes for those APIs were not organised in a hierarchical manner.

Table 6.9 presents detailed detection results for the four REST linguistic antipatterns related to the semantic design of REST URIs on 15 RESTful APIs. The table reports the

antipatterns in the first column followed by the analysed RESTful APIs in the following fifteen columns. For each REST API and for each antipattern, we report the total number of occurrences reported as positives by our detection algorithms. The last two columns show the total detected occurrences across 15 APIs (with percentage) and the average detection time.

As shown in Table 6.8, more than 70% of analysed URIs (219 out of 309) show amorphousness. Exceptionally, the Bestbuy API has all the URIs detected as *Tidy URI*. In contrast, all the URIs in Instagram and Twitter, for example, have syntactic problems and all of them are detected as *Amorphous URI*. As for the *Contextualised Resource Names* pattern, most of the APIs applied this pattern correctly—APIs providers use contextual resources names as nodes in URIs design—an important factor that affects the understandability of RESTful APIs. We rely on WordNet [Miller (1995)] and Stanford CoreNLP [Manning *et al.* (2014)] to capture contexts and perform semantic analyses. WordNet is a widely used lexical database, which groups nouns, verbs, and adjectives into sets of *cognitive synonyms*, each representing unique concepts. WordNet helps finding the semantic similarity between two nodes or resource names. Stanford's CoreNLP annotate nodes (after splitting CamelCase nodes) with its underlying POS (part-of-speech) tagger to differentiate verbs (*i.e.*, actions) and nouns (*i.e.*, resources).

However, our dictionary-based analyses did not relate contexts among URI nodes for 137 cases because the dictionaries we used are general English dictionaries and do not relate to specific domains like social networks such as Twitter and Facebook. A domain specific dictionary might reason about URIs contexts more accurately. We plan to build API-specific ontologies to better capture the context.

We observe the same detection results for *Hierarchical Nodes* pattern, *i.e.*, the dictionaries could not find hierarchical relations among URIs nodes. Indeed, we have zero detection for *Hierarchical Nodes* pattern because: (1) around 50% of tested URIs used only one node (excluding the base URI) in which case we cannot check the hierarchical relation and (2) more than 20% URIs contain digits or numbers as nodes, which again do not fall under any hierarchical relations. Finally, there is a significant amount of *No Detection* for *Singularised vs. Pluralised Nodes* since about 90% of our tested requests used HTTP GET method. HTTP GET requests can retrieve both single and multitude of resources. However, for the remaining 10%, the *Pluralised Nodes* antipattern appeared more frequently than the *Singularised Nodes* pattern.

Here, we discuss the *Contextless Resource Names* antipattern in detail (since, it is our running example). Out of 309 tested URIs, 14% (42 occurrences) of them are detected as

Contextless Resource Names antipatterns, 42% (130 occurrences) are detected as *Contextualised Resource Names* patterns, and 44% (137 occurrences) are detected as *None.* More specifically, for example, in Bestbuy, most of the URIs have only one node followed by parameters. We ignore parameters while we capture the context. Thus, if there is only one node in URIs, it is not possible to find any contextual relationship. Therefore, all the Bestbuy URIs are detected as *No Detection.*

In contrast, the Dropbox, Facebook, StackExchange, Twitter, and YouTube involve a high number of contextualised URIs naming. These good practices may help their APIs clients better understand and reuse. The following snippet shows two request URIs from Facebook where the URI nodes are considered to be in the same semantic context:

```
https://graph.facebook.com/v2.2/{user_id1}/mutualfriends/{user_id2}?access_token=CAATt8..
https://graph.facebook.com/v2.2/{user_id1}/friendlists?access_token=CAATt8..
```

For Facebook, our DOLAR approach reported 21 tested methods (out of 67) as *Contextualised Resource Names* patterns.

Table 6.9: Detection Results of the Four REST Linguistic Antipatterns Related to the Semantic Design of REST URIs Obtained by Applying Detection Algorithms on the 15 RESTful APIs (numbers in the parentheses show total test methods for each API). The Detection Time Excludes the Execution Time—Sending Requests and Receiving Responses.

RESTful APIs	(9)Alchemy	(20)Bestbuy	(15)Bitly	(12)CharlieHarvey	(17)DropBox	(6)Externalip	(67)Facebook	(14)Instagram	(19)Musicgraph	(7)Ohloh	(53)StackExchange	(19)TeamViewer	(25)Twitter	(17)YouTube	(9)Zappos	(309) Total	Detection Time
REST Linguistic Antipatterns																	
Amorphous URI	8	0	15	0	14	2	65	14	19	7	28	3	25	10	9	**219**(71%)	0.984s
Contextless Resource Names	0	0	2	4	1	0	7	4	9	2	6	0	6	1	0	**42**(14%)	0.565s
Non-hierarchical Nodes	9	0	10	8	15	0	28	12	19	5	34	0	25	6	0	**171**(55%)	0.584s
Pluralised Nodes	0	0	0	0	8	0	0	0	0	0	0	2	1	0	0	**11**(4%)	0.668s
Average																	0.7s

In the next section, we discuss service patterns commonly found in REST related to the

syntactic design of REST requests/responses.

6.6.7 Detection of REST Patterns

Table 6.10 presents detailed detection results for the five REST patterns related to the syntactic design of REST requests/responses. The table reports the patterns in the first column followed by the analysed RESTful APIs in the following twelve columns. For each REST API and for each pattern, we report: (1) the total number of validated true positives with respect to the total detected patterns by our algorithms, *i.e.*, the precision, in the first row and (2) the total number of detected true positives with respect to the total existing true positives, *i.e.*, the recall, in the following row. The last two columns show, for all APIs, the average precision-recall and the total detection time for each pattern.

Table 6.10: Detection results of the Five REST Patterns Related to the Syntactic Design of REST Requests/Responses Obtained by Applying Detection Algorithms on the 12 RESTful APIs (numbers in the parentheses show total test methods for each API).

REST API	(7)Alchemy	(12)BestBuy	(3)Bitly	(4)CharlieHarvey	(15)DropBox	(29)Facebook	(8)Musicgraph	(3)Ohloh	(8)TeamViewer	(10)Twitter	(9)YouTube	(7)Zappos	precision-recall	(115) Total	Average Precision–Recall	Detection Time
REST Patterns																
Content	5/5	11/11	0/0	0/0	14/14	26/26	0/0	3/3	5/5	0/0	0/0	7/7	p	71/71	100%	19.63s
Negotiation	5/5	11/11	0/0	0/0	14/14	26/26	0/0	3/3	5/5	0/0	0/0	7/7	r	71/71	100%	
Entity	6/6	11/11	1/1	4/4	3/3	21/21	1/1	2/2	1/1	5/5	6/6	4/4	p	65/65	100%	19.90s
Linking	6/6	11/11	1/1	4/4	3/3	21/21	1/1	2/2	1/1	5/5	6/7	4/4	r	65/66	98.81%	
End-point	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/1	0/0	0/0	0/0	0/0	p	2/2	100%	20.36s
Redirection	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/1	0/0	0/0	0/0	0/0	r	2/2	100%	
Entity	1/1	0/0	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	0/0	p	10/10	100%	23.06s
Endpoint	1/1	0/0	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	0/0	r	10/10	100%	
Response	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	8/8	4/4	p	13/13	100%	19.23s
Caching	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	8/8	4/4	r	13/13	100%	
Average													p	161/161	100%	20.44s
													r	161/162	99.76%	

In the next section, we discuss five REST antipatterns commonly found in RESTful APIs.

Detection of Linguistic Patterns in REST

Table 6.11 presents detailed detection results for the five REST linguistic patterns related to the semantic design of REST URIs on 15 RESTful APIs. The table reports the patterns in the first column followed by the analysed RESTful APIs in the following fifteen columns. For each REST API and for each pattern, we report the total number of occurrences reported as positives by our detection algorithms. The last two columns show the total detected occurrences across 15 APIs (with percentage) and the average detection time.

Table 6.11: Detection results of the Five REST Linguistic Patterns Related to the Semantic Design of REST URIs Obtained by Applying Detection Algorithms on the 15 RESTful APIs (numbers in the parentheses show total test methods for each API). The Detection Time Excludes the Execution Time—Sending Requests and Receiving Responses.

RESTful APIs	(9)Alchemy	(20)Bestbuy	(15)Bitly	(12)CharlieHarvey	(17)DropBox	(6)Externalip	(67)Facebook	(14)Instagram	(19)Musicgraph	(7)Ohloh	(53)StackExchange	(19)TeamViewer	(25)Twitter	(17)YouTube	(9)Zappos	(309) Total	Detection Time
REST Linguistic Patterns																	
Tidy URI	1	20	0	12	3	4	2	0	0	0	25	16	0	7	0	**90**(29%)	0.968s
Contextualised Resource Names	9	0	8	4	14	0	21	8	10	3	28	0	19	6	0	**130**(42%)	0.66s
Verbless URI	9	20	15	12	11	5	58	14	19	7	52	19	20	17	9	**287**(93%)	0.677s
Hierarchical Nodes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**(0.0%)	0.592s
Singularised Nodes	0	0	0	0	0	0	0	1	0	0	0	2	1	3	0	**7**(2%)	0.656s
Average																	0.656s

Further Discussion of the REST Results:

Table 6.12 shows the validation results on Dropbox (Validation 1) and on four representative APIs (Validation 2). For the first validation, the average precision is 81.4% and recall is 78% for all (anti)patterns. For the second validation, the average precision is 79.7%.

Table 6.12: Complete validation results on Dropbox (Validation 1) and partial validation results on Facebook, Dropbox, Twitter, and YouTube (Validation 2). 'P' represents the numbers of detected positives and 'TP' the numbers of true positives.

Antipatterns/ Patterns	Validation 1 DOLAR P	TP	Validated	Precision	Average Precision	Recall	Average Recall	Validation 2 DOLAR P	TP	Validated	Precision	Average Precision
✘Amorphous URI	13	12	12	92.31%		100%		4	4	4	100%	
✔Tidy URI	3	3	4	100%	96.2%	75%	87.5%	3	3	3	100%	100%
No detection	0	0	0	-		-		0	0	0	-	
✘Contextless Resource Names	0	0	0	-		-		2	0	2	0%	
✔Contextualised Resource Names	14	14	14	100%	100%	100%	100%	5	3	5	60%	53.3%
No detection	2	2	2	100%		100%		3	3	3	100%	
✘CRUDy URI	6	6	8	100%		75%		2	2	2	100%	
✔Verbless URI	10	8	8	80%	90%	100%	87.5%	9	9	9	100%	100%
No detection	0	0	0	-		-		0	0	0	-	
✘Non-hierarchical Nodes	14	3	3	21.43%		100%		6	1	3	16.67%	
✔Hierarchical Nodes	0	0	11	-	60.7%	0%	66.7%	0	0	11	-	58.3%
No detection	2	2	2	100%		100%		4	4	5	100%	
✘Pluralised Nodes	6	3	4	50%		75%		1	1	4	100%	
✔Singularised Nodes	0	0	2	-	60%	0%	48.3%	1	1	6	100%	86.7%
No detection	10	7	10	70%		70%		10	6	10	60%	
Average				Precision	81.4%	Recall	78%				Precision	79.7%

In the first validation of Dropbox, two occurrences of *Verbless URI* are false positives. The terms 'copy' and 'search' (or their synonyms) were not considered CRUDy by our algorithm in /1/copy_ref/dropbox/MyDropboxFolder/ and /1/search/dropbox/MyDropboxFolder/. However, the manual validation considered those terms CRUDy. Thus, on Dropbox, we had a precision of 100% and a recall of 75% for *CRUDy URI* and a precision of 80%, recall of 100% for *Verbless URI.*

The *Non-hierarchical Nodes* antipattern was detected by our detection algorithm in 14 cases whereas the manual validation suggested only three of them actually are organised in a non-hierarchical order. We manually investigated the causes of such discrepancies, and found that the URIs that we identified as antipatterns by our detection algorithms and, later, were (manually) validated as patterns have the following URI pattern:

```
{baseURI}/{media|revisions|shares}/dropbox/MyDropboxFolder/...
{baseURI}/fileops/{copy|delete|move|create_folder}/?root=dropbox&path=...
```

Our dictionary-based analyses did not find any hierarchical relations between {media,revisions,shares} and dropbox, between MyDropboxFolder and dropbox, and so on. Yet, these hierarchical relations are obvious for developers and it was easy to infer the hierarchical relations among those pairs simply because they use a natural naming scheme [Laitinen

(1996)]. It is the same for the second example, where fileops and {copy, delete, move, create_folder} are validated to be in hierarchical relation and the English dictionaries could not find any hierarchical relations, thus DOLAR considered them as *Non-hierarchical Nodes* antipatterns. Therefore, for this antipattern, we had a low precision of 21.43%.

In the second validation, also for the *Non-hierarchical Nodes* antipattern, DOLAR faces a similar problem for Twitter as illustrated in these examples:

```
{baseURI}/help/privacy.json
{baseURI}/statuses/{show.json|user_timeline.json}?screen_name=...
```

The dictionary-based analyses did not find any hierarchical relations between 'help' and 'privacy' or between 'statuses' and {show,user,timeline} and reported them as non-hierarchical. The precision for *Non-hierarchical Nodes* antipattern is therefore 16.67%, due to this limitation with the analyses.

Finally, an interesting observation from Table 6.12: two cases were identified as *Context-less Resource Names* antipatterns that were manually validated as *Contextualised Resource Names* pattern. Our investigation shows that the English dictionaries suggested 'Canucks' and 'albums' in Facebook and 'followers' and 'list' in Twitter to be in two different contexts. However, three professionals validated them as patterns, which caused the precision down to 0% for this antipattern in four representative APIs, with an average precision of 53.3%.

```
https://graph.facebook.com/Canucks/albums?access_token=CAA2...
https://api.twitter.com/1.1/followers/list.json?screen_name=...
```

In the next section, we assess the four assumptions stated in Section 6.2. The assessment helps us to show that the SODA approach can efficiently and effectively detection service antipatterns in SBSs.

6.7 Discussion on Assumptions

We now discuss the four assumptions stated in Section 6.2.

A1: Generality

Using our DSL, derived from our proposed unified abstraction, we specified 13 SCA antipatterns (as listed in Table 2.2). Please refer to Appendix A for the full set of antipatterns and their specifications. These antipatterns range from simple ones, such as the *Tiny Service*

and *Multi Service*, to complex ones such as the *Sand Pile*, which involve several services and complex relationships. In particular, *Sand Pile* has both the `ASSOC` and `COMPOS` relation types. Also, both *Sand Pile* refers, in its specification, to another antipattern, *i.e.*, *Data Service*. As for the Web services antipatterns, we specified ten antipatterns from the literature (see Table 2.2) where we specified complex antipatterns with composite rules, such as *CRUDy Interface* composed of another rule card, *i.e.*, *Chatty Web Service*. We also specified antipatterns combining six different rules, *Ambiguous Name* antipattern, for instance. The generality of defined heuristics is fulfilled because engineers can define their own heuristics based on their needs. Thus, we show that we can specify from simple to complex antipatterns regardless of the SBSs technology, which supports the generality of our DSL. As for the REST, we define only pseudocode-style detection heuristics and we do not rely on the DSL. Thus, the definition of REST heuristics is not related to the generality of our DSL.

A2: Accuracy

Concerning the accuracy of our detection algorithms, as shown in Table 6.13, we obtained an average recall of **95.5%**, precision of **88%**, and an average F_1-measure of **90.66%** for all service antipatterns and patterns. The achievement of such detection accuracy was possible because we defined the rule cards (for Web services and SCA) and the heuristics (for REST) after a thorough literature review and a careful analysis of relevant properties for all service antipatterns. Consecutively, the detection algorithms were generated or implemented precisely according to the rule cards and heuristics.

Thus, we can positively support our second assumption on the accuracy of our automatically generated (or implemented) detection algorithms.

A3: Extensibility

Our proposed DSL was initially designed for specifying SCA antipatterns described in the literature (see Table 2.2). Such design requires inclusion of various service metrics, mathematical, logical, and set operators. Table 5.1 provides the list of service metrics among which the metrics specific to SCA, namely `COH`, `CPL`, `NIR`, `NOR`, `NUM` were initially defined and extracted from the literature. Later, for specifying new Web services antipatterns, we reused several pre-existing metrics, for example `ANP`, `ANPT`, `COH`, `CPL`. Moreover, we added some new Web services metrics to our metric suite, namely `ALS`, `ARIP`, `NOPT`, `RGTS`, and so on, which are very specific to Web services. Our defined DSL is flexible in the integration of new service metrics and antipatterns. However, the underlying SOFA framework should also be extended to provide the operational implementations of the new service metrics. Such an

Table 6.13: Average Precision, Recall, and F1-measure for the Different Antipatterns Groups.

Antipatterns Groups (Tables)	Average Precision	Average Recall	Average F-1 measure
SCA ∩ REST ∩ Web services (Table 6.2)	93.33%	100%	96.3%
SCA ∩ Web services (Table 6.3)	82.59%	95.83%	86.34%
REST ∩ Web services (Table 6.4)	75%	100%	83.33%
SCA (Table 6.5)	88.89%	100%	93.3%
Web services (Table 6.6)	100%	100%	100%
REST Syntactic Antipatterns (Table 6.8)	82.81%	90.4%	86.44%
REST Syntactic Patterns (Table 6.10)	100%	99.76%	99.88%
REST Linguistic Antipatterns (Table 6.12)	81.4%	78%	79.66%
Average	**88%**	**95.5%**	**90.66%**

addition can only be realised by developers skilled with our framework, which may require hours according to the complexity of the metrics. However, once the metrics are integrated in the SOFA framework, their use is straightforward for the specification of rule cards using the DSL.

We added SCA antipatterns (*e.g.*, *Sand Pile*, *Bloated Service*, and *Multi Service*, and so on) within our detection framework first and then extended the framework with new Web services antipatterns (*e.g.*, *CRUDy Interface*) and REST (*e.g.*, *Forgetting Hypermedia*, *Ignoring MIME Types*), which further confirms the extensibility of our SOFA framework. To add REST antipatterns, we add their detection algorithms within the framework and add each REST API to test their underlying resources. Later, we automatically apply those detection algorithms on requests and responses received from server at run-time.

Thus, with these extensibility features of our DSL and our SOFA framework, we positively support A3.

A4: Performance

We performed all our experiments on an Intel Dual Core at 3.30GHz with 4GB of RAM. For SCA, computation times include introspection delay during static and dynamic analyses, computing metric values, and applying detection algorithms.

As for the Web services, for each antipattern, the detection time also includes: (i) the

filtering of the WSDL files, (ii) the generation of the concrete services implementation, (iii) the generation of the detection algorithms, and (iv) the computation of the related metrics.

For the detection of antipatterns in RESTful APIs, the total required time includes: (i) the execution time, *i.e.*, sending REST requests and receiving REST responses and (ii) the time required to apply and run the detection algorithms on the requests and responses.

As shown in Table 6.14, the average detection time for all service antipatterns (regardless of the technologies) is 27.086s with a minimum of 0.184s (for SCA-specific antipatterns) and a maximum of 144.72s (for Web services-specific antipatterns with a high number of required pair-wise comparisons). Thus, with such low average detection time of **27.086s**, we can positively support our fourth assumption on performance.

Table 6.14: Average Detection Times for the Different Antipatterns Groups.

Antipatterns Groups (Tables)	Average Detection Times
SCA ∩ REST ∩ Web services (Table 6.2)	0.511s
SCA ∩ Web services (Table 6.3)	9.09s
REST ∩ Web services (Table 6.4)	18.98s
SCA (Table 6.5)	0.184s
Web services (Table 6.6)	144.72s
REST Syntactic Antipatterns (Table 6.8)	22.06s
REST Syntactic Patterns (Table 6.10)	20.44s
REST Linguistic Antipatterns (Table 6.12)	0.7s
Average	**27.086s**

It is important to note that the complexities of our detection algorithms are *linear*, *i.e.*, $O(n)$, where n represents the number of rules. Therefore, the cumulative detection times increase with the number of rules and with the number of antipatterns to be detected.

6.8 Threats to Validity

The main threat to the validity of our results concerns their *external validity*, *i.e.*, the possibility to generalise our approach to other SCA systems, SOAP Web services, and RESTful APIs. We minimise the external validity by experimenting with one demo SCA system and one large scale SCA system, *i.e.*, *FraSCAti*. For Web services, we experimented

with more than 120 Web services, while we analysed 15 common and widely-used RESTful APIs by invoking more than 300 methods from them to minimise external validity.

For *internal validity*, the detection results not only depend on the services provided by the SOFA framework, but also on the antipattern specifications using rule cards and–or on the heuristics defined for the REST antipatterns. To minimise the threat to the internal validity: (1) we thoroughly studied the descriptions and definitions of each antipattern from the literature and (2) we made sure that our SOFA framework itself does not introduce any antipatterns. Moreover, for RESTful APIs, we confirmed that every invocation receives responses from servers with the correct request URI, and the client authentication is done while necessary. We tested all the major HTTP methods in REST, *i.e.*, GET, DELETE, PUT, and POST on resources to minimise such threat. Also, we performed experiments on a representative set of antipatterns to lessen the threat to internal validity.

The subjective nature of specifying and validating antipatterns and defining detection heuristics is a threat to *construct validity*. We tried to lessen this threat by defining rule cards and detection heuristics based on a literature review and thorough domain analysis and by involving independent engineers in the validation.

We also tried to minimise the threat to *reliability validity* by automating the generation of the detection algorithms for SCA and Web services, such that each subsequent detection produces consistent sets of results with high precision and recall. However, for REST, the manual implementation of detection algorithms was fundamental. The threats to reliability validity also concern the possibility of replicating all the studies in this dissertation. We provided all the details required to replicate the study, including the rule cards and heuristics on our Web site `http://sofa.uqam.ca/soda/` to minimise such threats.

In the next section, we introduce our `SODA` tool-set, which relies on our proposed unified SODA approach.

6.9   Tool Support

The methods and techniques presented in this dissertation are partially and–or fully implemented and integrated with the `SODA` tool-set. The `SODA` tool-set relies on the proposed unified SODA approach and currently is a prototype implementation. The `SODA` tool-set is capable of performing the detection of service antipatterns in SCA systems, Web services, and RESTful APIs.

Following the three steps in the SODA approach, an engineer can express service antipatterns using rule cards. The rule cards must conform to the grammar defined in Figure

5.2 (see Section 5.2.1). This is the first step in detecting service antipatterns. The two key parts in this step are: (1) engineers must have certain domain knowledge on service antipatterns and (2) engineers can use their own threshold values while writing rule cards, which give them the flexibility to rely on their knowledge and experience (expected detection) over the detection results returned by the tool (actual detection).

In the second step of the detection, the SODA tool-set allows two separate mechanisms for generating detection algorithms: automatic and manual. Currently, for SCA and Web services, an EMF-driven code generation technique helps engineers to generate detection algorithms for antipatterns specific to SCA and Web services. However, for REST antipatterns, the SODA tool-set requires engineers to manually implement the detection algorithms.

The third step in the SODA tool-set is completely automatic, and both the automatically generated or manually implemented algorithms are applied either on SCA systems, Web services, or on RESTful APIs. The mandatory condition for this step is to wrap each Web service and each REST API with an SCA component because the SODA tool-set (likewise the SOFA framework) allows the introspection of only SCA components.
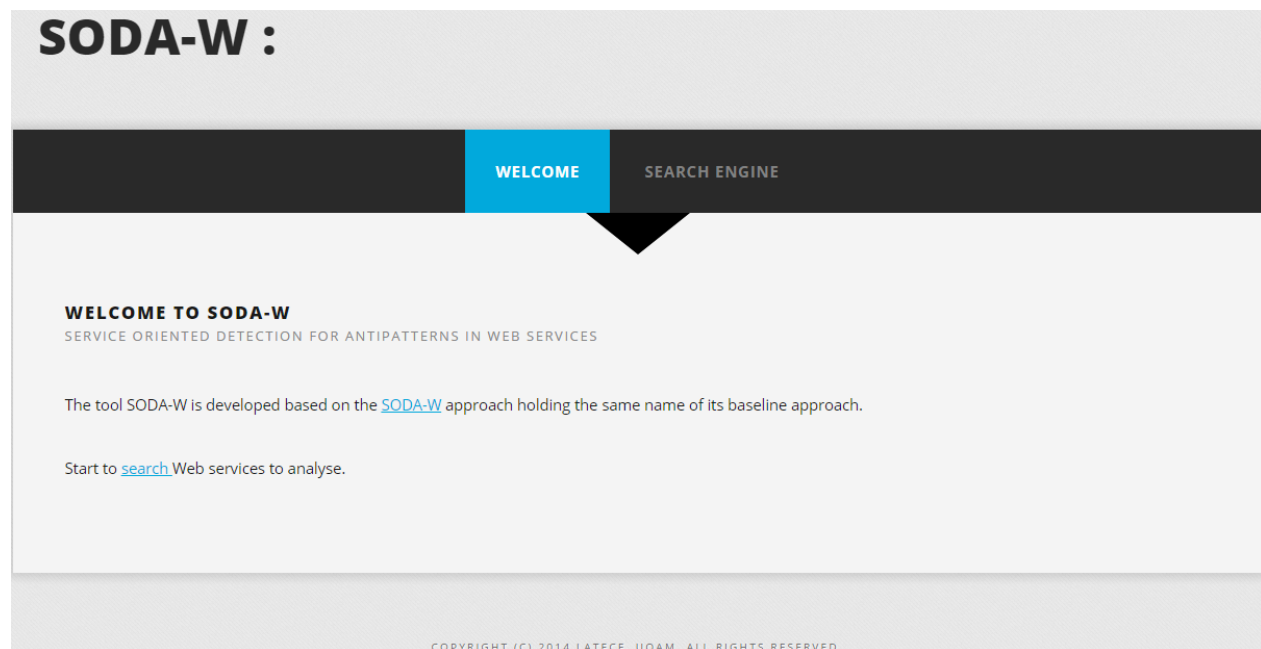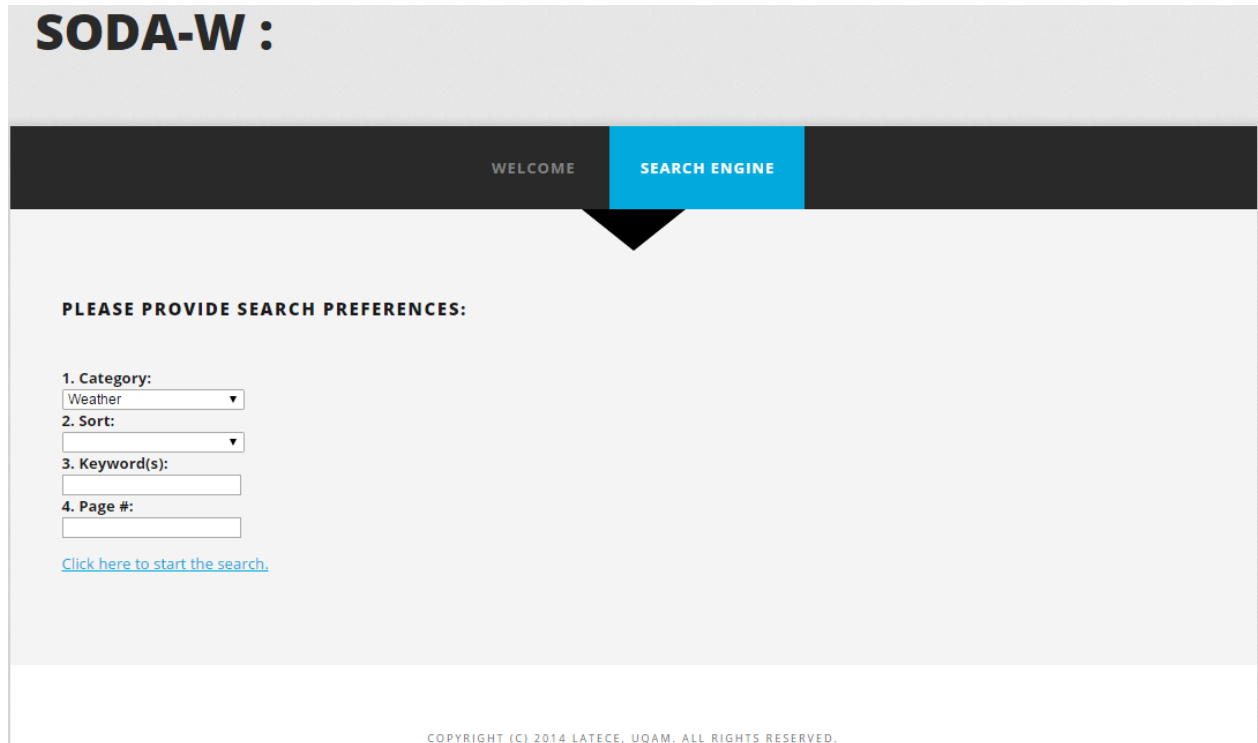
Figure 6.3: The Home Page of the Prototype Web Interface for the Detection of Web services-specific Service Antipatterns.

Thus, the SODA tool-set has three main modules: (1) the *specification* module, which

Figure 6.4: The Search and Detection Page of the Prototype Web Interface for the Detection of Web services-specific Service Antipatterns.

helps engineers specifying rule cards or heuristics (2) the *generation* module, which helps engineers generating detection algorithms and (3) the *detection* module, which performs detection and report suspicious services.

In summary, most of the dynamic and static metrics calculated by `SODA` tool-set use only the service interfaces that are freely available. The principal features of `SODA` tool-set include:

1. `SODA` does direct import of an SCA system as a Jar package, or a WSDL interface, or a REST API implemented as a Java class with methods to access its underlying resources;

2. `SODA` has a straightforward and simple detection interface for the engineers, which is handy both for beginners and experts;

3. `SODA` shows all the detection details, *i.e.*, metric values, corresponding rule cards, textual descriptions of antipatterns, and so on, in the form of a detection report for SCA and Web services, and the request and response headers and bodies for RESTful APIs;

Figure 6.5: The Prototype Web Interface for the Detection of REST-specific Service Antipatterns.

4. For the detection, the `SODA` tool-set invokes Web services on the fly and can calculate various run-time properties of a Web service, for example, response time and availability;

5. Features we plan to implement: (1) the calculation of other quality metrics, including *reliability* and *accessibility* of services and (2) the visualisation of detected antipatterns in SCA components, Web services, and REST resources.

The downloadable or Web-accessible versions of the `SODA` tool-set can be found on our Web site `http://sofa.uqam.ca/soda/`. For the detection tool on Web services, users can

download the WAR version and deploy on a local Apache Tomcat server. The tool's Web interface is presented in Figures 6.3 (tool's home page) and 6.4 (tool's search and detection page). The detection tool on SCA can be executed from command line. As the detection tool of REST antipatterns, we provide a Web-based interface from where users can perform the detection of REST antipatterns on selected RESTful APIs as shown in Figure 6.5. It is important to note that although we provide two separate tools for SCA and Web services, they share a common language for the specification of service antipatterns and rely on a generic approach for the detection of service antipatterns.

6.10 Discussion

Based on our proposed unified abstraction (see Section 4.2), we presented the unified SODA approach for the detection of service antipatterns in various SBSs technologies involving three main steps from the specification of service antipatterns to their detection via the step of automatic generation of detection algorithms (Section 5.1). We also listed various service metrics that we use to specify service antipatterns (Table 5.1). Using those metrics we defined rule cards and detection heuristics for 31 service antipatterns. We also presented the SOFA detection framework (Section 5.2.3) and discussed the detailed evaluation of our proposed methodology where we experimented with three commonly used SBSs technologies, *i.e.*, SCA, Web services, and REST.

In particular, we discussed the detection results for service antipatterns for different groups as we mentioned in Section 2.2. The discussions on the detection results confirm the presence of service antipatterns in practice. Some significant observations are: (1) the occurrence of *Ambiguous Name* antipattern is very common in Web services interfaces, (2) a significant number of eight service antipatterns are common in SCA and Web services (see Table 6.3), (3) the *Tiny Service* antipattern is also very common in SCA and Web services, which further confirms the fact that (as suggested by researchers) *Tiny Service* antipattern is the cause of a number of SOA failures [Král et Žemlička (2008)], (4) maintaining the syntactic and semantic cohesiveness among the service interface elements is still a major challenge for service designers and developers (as reflected in Table 6.6), and, (5) RESTful APIs suffer comparatively higher syntactic and lexical design challenges than SCA and Web services (as shown in Tables 6.8, 6.9).

With a detailed discussion on the detection results, we positively supported four assumptions (Section 6.7) showing the generality of our specification language (**A1**), the accuracy of our detection algorithms (**A2**), the extensibility of our language and of the framework (**A3**), and the low performance overhead of our detection algorithms in terms of detection time

(**A4**). Therefore, we can conclude that:

Summary: With our proposed unified SODA approach that encompasses the unified abstraction and the DSL, we can effectively specify and detect service antipatterns in different SBSs technologies in terms of accuracy and performance.

In this chapter, we provided the evidence of the presence of service antipatterns in SBSs in practice. However, a strong motivation showing why the engineers and developers should and need to care about service antipatterns and detect them as early as possible, is still missing.

In the next chapter, we conduct a study to observe the impact of service antipatterns on the maintainability, *i.e.*, change-proneness, of SBSs. To verify and confirm the impact of service antipatterns on maintainability, negative or positive, we study the entire evolution history of *FraSCAti* system—the largest open-source SCA system available at present [Seinturier *et al.* (2012)]. We rely on eight service antipatterns detected and discussed in the previous chapter in SCA (see Sections 6.6.1, 6.6.2, and 6.6.4). To confirm the correlation between service antipatterns and change-proneness of an SBS, we perform two well-known non-parametric statistical tests, *e.g.*, Wilcoxon rank sum, *a.k.a.*, Mann–Whitney test and Kruskal-Wallis tests [Sheskin (2007)].

CHAPTER 7   AN IMPACT STUDY OF SERVICE ANTIPATTERNS

7.1   Chapter Overview

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

The remainder of this chapter is organised as follows. Section 7.2 discusses the motivation behind this study. Section 7.3 presents background information on the FraSCAti project. Section 7.4 describes the approach used to extract change and churn information, and to identify FraSCAti services involved in patterns and antipatterns, and the design of our study. We present the findings in Section 7.5. Section 7.6 reports the threats to the validity of our results. Finally, Section 7.7 concludes the chapter.

7.2   Motivation and Research Questions

Like any complex software systems, service-based systems (SBSs) evolve to accommodate new user requirements both in terms of functionality and quality of service (QoS). These

frequent changes may degrade the design and QoS of SBSs and cause the introduction of antipatterns which are common bad practice solutions—in opposition to patterns which are good solutions to common recurring design problems. A degradation of the design of an SBS means that it fails to follow one of the eight SOA design principles [Erl (2009)], including loose coupling, composability, and reusability. *Multi service* [Dudney *et al.* (2003)], an example of service antipattern corresponds to a service that implements a multitude of business and technical abstractions. Its reusability is low because it aggregates too much into a single service, resulting in methods with low cohesion. This service is often unavailable to end-users because of its overload, which may induce a high response time. *Proxy pattern* [Daigneau (2011)], an example of service pattern, is a well-known service design pattern that adds an additional indirection level between the client and the invoked service, *e.g.*, to support adding non-functional behaviors.

Despite the relatively large body of work on the detection of service patterns and antipatterns in SBSs [Demange *et al.* (2013); Moha *et al.* (2012); Penta *et al.* (2007); Tsantalis *et al.* (2006)], to the best of our knowledge, there are very few studies that empirically investigated the impact of service patterns or antipatterns on the maintenance and evolution of SBSs. To perform such a study, one needs detailed information about the implementations of services, which is not easy to obtain because of the scarcity of open-source SBSs. We believe that service antipatterns do not only affect the clients of SBSs but also the maintenance and evolution of the SBSs, for example by making it harder for developers to modify existing functionalities, or to implement new ones. Several works exist in the object-oriented (OO) literature relating code smells and antipatterns to the change-proneness of software systems [Abbes *et al.* (2011); Khomh *et al.* (2012a); Mäntylä et Lassenius (2006)]. However, because of the dynamic nature of service patterns and antipatterns [Moha *et al.* (2012)] and because of the difference in granularity, results obtained for OO systems cannot be simply transferred to SBSs. Service antipatterns and OO antipatterns are two very different concepts. Indeed, one of the root causes of OO antipatterns is the adoption of a procedural design style in OO systems, whereas service antipatterns often stem from the adoption of OO design style in SBSs.

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

We address the following three research questions:

RQ$_{2.1}$: *What is the relation between service antipatterns and change-proneness?*
Finding: The total number of source code changes and code churns performed during the maintenance and evolution of services involved in antipatterns is higher than the total number of source code changes and code churns performed on other services—the difference is statistically significant.

RQ$_{2.2}$: *What is the relation between particular kinds of service antipatterns and change-proneness?*
Finding: Services found to be involved in *God Component*, *Multi service*, and *Service Chain* antipatterns are more change-prone than services involved in other antipatterns—the difference is statistically significant.

RQ$_{2.3}$: *What is the relation between service patterns and change-proneness?*
Finding: The total number of source code changes and code churns performed during the maintenance and evolution of services involved in patterns is less than the total number of source code changes and code churns performed in other services—the difference is not statistically significant.

In summary, service antipatterns (respectively patterns)—which indicate poor (respectively good) service designs—do not only affect the clients of SBSs but also the cost of maintenance of the SBSs themselves. Developers and maintainers should therefore avoid implementing antipatterns in their SBSs since it will significantly increase the maintenance effort and hence the maintenance cost of the system.

7.3 The Study Object: FraSCAti

Table 7.1: Summary of the Characteristics of the FraSCAti OW2 v1.4 (the entire revision history).

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

FraSCAti [Seinturier *et al.* (2012)] is a Java-based open-source implementation of the Service Component Architecture (SCA) standard [Edwards (2011)]. FraSCAti is based on

the OW2 Fractal[1] component model and provides an open architecture for the integration and binding of SCA components. SCA defines a technology-agnostic model for composing diverse interface definition languages (WSDL, Java, WADL, etc.), implementation languages and frameworks (Java, BPEL, C/C++, Spring, OSGi, etc.), bindings (SOAP, JMS, REST, etc.). To date, FraSCAti is the largest service-oriented SCA system for which the source code and change commits are publicly available. Table 7.1 summarises the main attributes of the FraSCAti project for its entire revision history. FraSCAti offers 130 distinct services. The size of the FraSCAti project is around 170 KLOC excluding any supporting and configuration files. We collected more than 15,000 changed files from the entire FraSCAti revision history, more than 9,000 of which are Java source files. The patterns and antipatterns studied in this dissertation were detected for 62 services, thus around 3,700 Java source files were involved directly or indirectly with these services implementations. Moreover, a few more than 1,800 and 2,100 analysed Java source files were directly involved with the studied patters and antipatterns in [Demange *et al.* (2013)] and [Palma *et al.* (2013)], respectively. As shown in Table 7.1, we extracted more than 71,000 changes and approximately 62.6 million code churns from the entire FraSCAti commits.

7.4   Study Design

This section presents the design of our study, which aims to address the three research questions stated in Section 7.2.

7.4.1   Data Collection and Processing

In this study, we analyse FraSCAti services over their entire revision history. Our data set contains, for each FraSCAti service: (1) all the source code changes performed and (2) the code churns in its entire revision history. We also gather the type and the number of service patterns and antipatterns in which a FraSCAti service is involved, using SODOP [Demange *et al.* (2013)] and SODA [Palma *et al.* (2013)] detection techniques. Figure 7.1 shows an overview of our data collection and processing approach. The remainder of this section elaborates on each of its steps.

---

1. `http://fractal.ow2.org/`
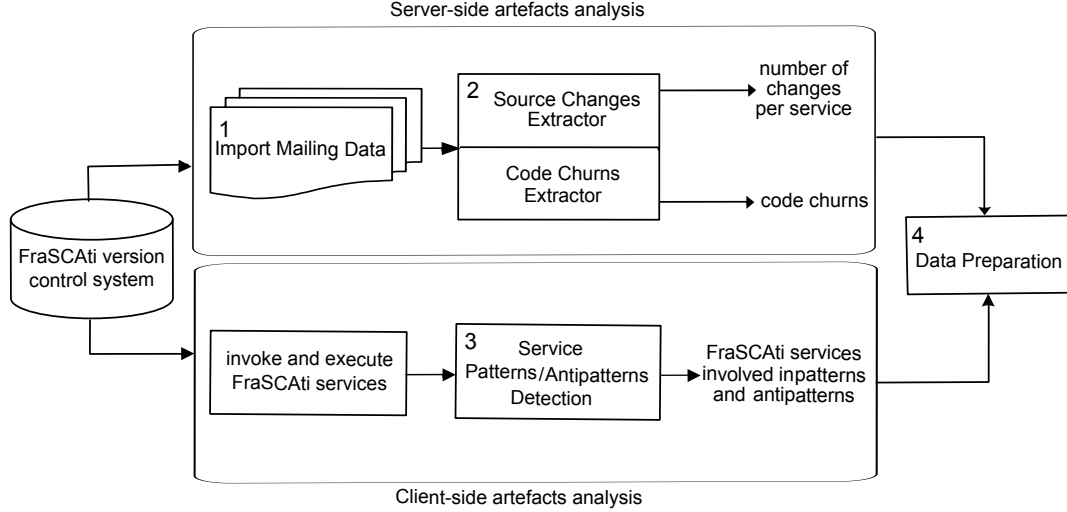
Figure 7.1: An Overview of Our Approach to Study the Impact of Service Patterns and Antipatterns on the Change-proneness of SBSs.

Step 1: Collecting Mailing Data

The first step consists in mining change data information from the complete FraSCAti-commits mailing list archives. The FraSCAti-commits mailing list archive is available online [2]. We developed a Python script to recursively download all the change histories for the entire FraSCAti project. For each commit message in the mailing list, we extract (1) the revision number, (2) the author, (3) the date of the commit, (4) the log message, (5) the modified paths, *i.e.*, the list of changed files, (6) the added paths, *i.e.*, the list of newly added files, (7) the removed paths, *i.e.*, the list of removed files, and (7) the *diff* which contains detailed information about the changes that were performed on each of the modified, added, or removed files. We stored all these information in a MySQL database for analysis.

Step 2: Extracting Source Code Changes and Code Churns

The second step involves the extraction of the number of changes made and the number of code churns for a certain service artefact. In this case, we also used a Python script to query our database and calculate the number of times that each file involved in the implementation of a service appeared in the commit. For each file and for each commit containing the file, we parse the *diff* contained in the commit and extract information about the number of lines of code that were added and removed. We use this information to compute the Code churn of the file for that commit, as the total number of added, modified and deleted lines of code

---

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

in the file. In a *diff*, the modification of a line is recorded as a line deletion followed by a line addition. We aggregate the code churns values of the file obtained for all commits in which the file was involved to obtain the *total code churn* of the file.

Step 3: Service Patterns and Antipatterns Detection

The third step in our approach involves detecting service patterns and antipatterns in FraSCAti. This detection is done on the client-side by analysing service compositions, system design, and the quality of service (QoS). We perform the detection of service antipatterns using SODA (Service Oriented Detection for Antipatterns) [Palma *et al.* (2013)]. We asked the core development team of FraSCAti to manually validate all the antipatterns that were found in FraSCAti before their usage in our study. We also used the SODOP approach (Service Oriented Detection Of Patterns) proposed by Demange *et al.* [Demange *et al.* (2013)] to perform the detection of service patterns. The SODOP approach is dedicated to the detection of service patterns in SCA systems and an essential part of our integrated SOFA framework. SODOP shares the same specification language with SODA to represent service patterns at a higher abstraction level. However, new service metrics are required by SODOP to capture various design decisions, which are considered to be good design practices to common recurring service design problems.

The patterns found in FraSCAti were also manually validated by the core development team of FraSCAti. A detailed description of the service antipatterns and patterns analysed in this study is presented in Appendix. Table 7.2 shows the summary of the detection results for service-oriented patterns and antipatterns in FraSCAti v1.4.

Step 4: Data Preparation

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

We map each service to the corresponding artifacts or source files from the entire FraSCAti project in the form of $s_i \rightarrow f_{1\ to\ k}$, where for each i from 1 to 130, a service s is associated with different numbers of artefacts or source files f up to k. We classify the entire FraSCAti project into three groups: (1) Java source files that underwent any number of changes and

Table 7.2: Summary of the Detection Results for Five Service Patterns and Eight Service Antipatterns in FraSCAti OW2 System.

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

are part of any patterns or antipatterns, (2) Java source files that underwent any changes and are not related to any patterns or antipatterns, and (3) Java source files that did not undergo any changes. This classification helps us to restrain relevant source files while we compare among changed *vs.* unchanged and pattern *vs.* antipattern service groups. We also manually gather various details about the patterns and antipatterns considered in this study, *e.g.*, their categories, the levels of their appearance, root causes, and symptoms. Finally, we collected the feature details from the FraSCAti feature model [3] for each FraSCAti service, *i.e.*, which particular features are implemented by a service in FraSCAti. This feature information helps us to better understand the changes made in services that are involved in patterns or antipatterns. Using these data, we perform a series of statistical analysis to examine the relation between service patterns/antipatterns and change-proneness. The following sections discuss the details of our analysis method.

7.4.2   Variable Selection

We identify the following dependent and independent variables to test the *null* hypotheses (defined in Section 7.5) corresponding to each research question.

Independent Variables

The total set of service patterns and antipatterns that we considered in this study are the independent variables. We investigate the presence of eight different service antipatterns

---

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

and four different service patterns.

For $\mathbf{RQ}_{2.3}$, we use the boolean variable f_i^3 to indicate whether a file i was involved in the implementation of at least one pattern. For $\mathbf{RQ}_{2.1}$, we use a similar boolean variable f_i^1 to indicate whether a file i was involved in the implementation of at least one antipattern. Finally, for $\mathbf{RQ}_{2.2}$, we use the boolean variables $f_{i,j}^2$ to indicate whether a file i was involved in the implementation of the antipattern j.

Dependent Variables

The dependent variables measure the phenomena related to services participating in service antipatterns or patterns. Our dependent variable for research questions $\mathbf{RQ}_{2.1}$ to $\mathbf{RQ}_{2.3}$ is the change-proneness of the files involved in the implementation of a service. We measure the change-proneness of a file i using the total number of changes c_i and the total number of code churns d_i that the file i underwent in its entire revision history.

7.4.3 Analysis Method

We apply the Wilcoxon rank sum and Kruskal-Wallis tests [Sheskin (2007)] to compare the proportion of source code changes and code churns in the different categories of services (*i.e.*, service antipatterns, service patterns, and others), using a 95% confidence level (*i.e.*, p-value<0.05). For any comparison exhibiting a statistically significant difference, we further compute the Cliff's δ effect size [Romano *et al.* (2006)] to quantify the importance of the difference because Cliff's δ is reported to be more robust and reliable than Cohen's d [Cohen (1988)].

The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical tests make no assumptions about the distributions of assessed variables. The Kruskal-Wallis test is an extension of the Wilcoxon rank sum test for more than two groups. Cliff's δ is a non-parametric effect sizes measure (*i.e.*, it makes no assumptions of a particular distribution) which represents the degree of overlap between two sample distributions [Romano *et al.* (2006)]. It ranges from -1 (if all selected values in the first group are larger than the second group) to +1 (if all selected values in the first group are smaller than the second group). It is zero when two sample distributions are identical [Cliff (1993)].

$$\text{Cliff's } \delta = \begin{cases} +1, & \text{Group 1} > \text{Group 2}; \\ -1, & \text{Group 1} < \text{Group 2}; \\ 0, & \text{Group 1} = \text{Group 2}. \end{cases}$$

Interpreting the Effect Sizes

Cohen's d is mapped to Cliff's δ via the percentage of non-overlap as shown in Table 7.3 [Romano *et al.* (2006)]: 0.20 (small) of Cohen's d corresponds to 0.147 of Cliff's delta δ, 0.50 (medium) corresponds to 0.33, and 0.80 (large) corresponds to 0.474. Cohen [Cohen (1992)] states that a medium effect size represents a difference likely to be visible to a careful observer, while a large effect is noticeably larger than medium.

Table 7.3: Mapping Cohen's d to Cliff's δ.

Cohen's Standard	Cohen's d	% of Non-overlap	Cliff's δ
small	0.20	14.7%	0.147
medium	0.50	33.0%	0.330
large	0.80	47.4%	0.474

Interpreting the *p-values*

In a statistical test, the p-value helps to determine the significance of the results and if the p-value is less than or equal to the threshold value (default is 5%), the *null* hypothesis can be rejected. The p-value ranges between 0 and 1. Different interpretations in our statistical tests are presented in Table 7.4.

Table 7.4: The informal interpretation of *p-values*.

p-values	Interpretation
$p < 0.001$	*Very strong* evidence against the *null* hypothesis.
$0.001 < p < 0.01$	*Strong* evidence against the *null* hypothesis.
$0.01 < p < 0.05$	*Moderate* evidence against the *null* hypothesis.
$0.05 > p < 0.1$	*Weak* evidence against the *null* hypothesis.
$p > 0.1$	*No* evidence against the *null* hypothesis.

7.5 Case Study Results

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

RQ$_{2.1}$: What is the relation between service antipatterns and change-proneness?

Motivation: Since service antipatterns represent *poor* designs, it is very likely that they negatively impact the quality of SBSs, for example by making them more prone to changes, which may result in an increase of maintenance costs. Clearing up the interaction between service antipatterns and change-proneness is important from both researchers' and practitioners' points of view. For researchers, a quantitative analysis of the impact of service antipatterns on change-proneness will contribute to proving or refuting the conjecture about their negative impact. For practitioners, knowing how service antipatterns affect the change-proneness of their code will help them make educated decisions about which antipattern to remove first. In this research question, we investigate the effect of service antipatterns on code change-proneness. Code change-proneness is an important quality attribute since it captures the effort required to modify and evolve the code of the SBSs, which translates into maintenance costs.

In general, the presence of service antipatterns cause solutions with poor quality of design and bad quality of service and hinder maintenance [Dudney *et al.* (2003)]. No one empirically validated the fact that service antipatterns have negative impact on the maintenance cost. After we detected services as antipatterns on client-side, we show that the services involved in antipatterns changed more during server-side maintenance in terms of number of changes and code churns.

Approach: We answer this research question in three steps: First, we perform the detection of service antipatterns using SODA [Palma *et al.* (2013)] and obtain a set of services involved in different antipatterns. We manually validated the results of our detection as discussed in Section 7.4.1. Next, for each file implementing a FraSCAti service, we measure the change-proneness of the service's implementation using the following two metrics:

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

Finally, to compare the change-proneness of files involved in the implementation of service antipatterns with the change-proneness of files implementing services, but not involved in a service antipattern, we test the two following *null* hypotheses:

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

H_{01}^2: *there is no difference between the total number of code churns experienced by files in-*
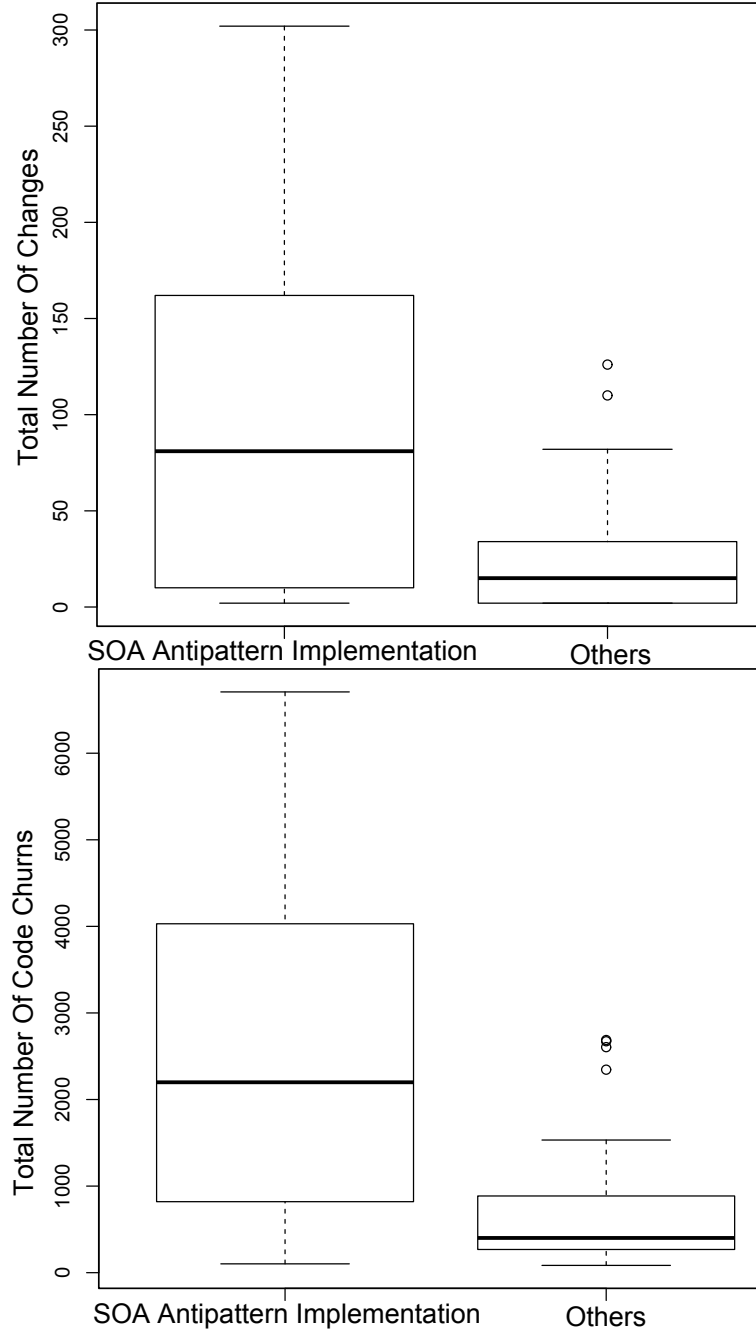
Figure 7.2: Comparison between Antipattern Services and Non-antipattern Services in Terms of Number of Changes (top) and Code Churns (bottom).

volved in the implementation of a service antipattern and other files.

We use the Wilcoxon rank sum test to examine H_{01}^1 and H_{01}^2, which are also two-tailed since they investigate whether service antipatterns are related to higher or lower change and code churn rates. For any comparison exhibiting a statistically significant difference, we

compute the Cliff's δ effect size [Romano *et al.* (2006)] to quantify the importance of the difference. All the tests are performed using the 5% significance level (*i.e.*, p-value<0.05).

Findings: Services involved in service antipatterns are more change-prone than the services that are not involved in any service antipattern. Figure 7.2 presents the box-plots showing the median difference between antipattern services and non-antipattern services both for the total number of changes (top) and the total number of code churns (bottom). This difference is statistically significant as the Wilcoxon rank sum test yielded p-values of 0.011 (<0.05) and 0.015 (<0.05) for respectively the total number of changes and the total number of code churns (see Table 7.5). Therefore, we reject both H_{01}^1 and H_{01}^2. The Cliff's δ effect size values presented in Table 7.6 shows that the difference is large for both the total number of changes and the total number of code churns (p-value <0.01), (*i.e.*, Cliff's δ=0.515 for changed LOC and 0.4962 for code churns) between two treatment groups, *i.e.*, antipattern-services and non-antipattern services.

There is a significant difference between the proportion of services undergoing at least one change and one code churn between services involved in antipatterns and other services. The Wilcoxon rank sum test in Table 7.5 confirms this observation (*i.e.*, p-value=0.0152 for changed LOC and p-value=0.011 for code churns), therefore we can strongly reject H_{02}.

Table 7.5: The Wilcoxon Rank Sum Test Between Service Antipatterns and Other Services.

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

The non-parametric Cliff's δ effect size measure in Table 7.6 shows the large difference (*i.e.*, Cliff's δ=0.515 for changed LOC and 0.4962 for code churns) between two treatment groups, *i.e.*, antipattern-services and non-antipattern services. In this test, with the significant p-value of $\sim$0.01, our Cliff's δ effect size measure is $\sim$0.5, which is expected, *i.e.*, the difference between two treatment groups is large.

Table 7.6: The Non-parametric Cliff's δ Effect Size Measure Between Service Antipatterns and Other Services.

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

Summary: Services involved in service antipatterns, in terms of their implementations, are more change-prone than the services that are not involved in any service antipattern. The total number of source code changes and code churns performed during the maintenance and evolution of services involved in a service antipattern is higher than the total number of source code changes and code churns performed on other services. The difference is statistically significant.

RQ$_{2.2}$: What is the relation between particular kinds of service antipatterns and change-proneness?

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

Approach: To answer this research question we proceed as follows: First, using the results of the antipattern detection performed in **RQ$_{2.1}$**, we divide the files involved in the implementation of service antipatterns in different categories corresponding to the 13 kinds of antipatterns that are considered in this study. For each kind of antipattern A_i, we create a group GA_i containing files that are involved in the implementation of an antipattern of type A_i. In total, we obtained eight groups of files ($GA_i, i \in \{1, \ldots 8\}$ since only eight kinds of service antipatterns were detected and validated in FraSCAti. We also create a group $GNoAP$ containing files implementing services that are not antipatterns. Next, for each file implementing a FraSCAti service, we measure the change-proneness of the service's implementation using the same metrics as in **RQ$_{2.1}$** (*i.e.*, *total number of changes* and *total number of code churns*). Finally, to compare the change-proneness of files involved in the implementation of different kinds of service antipatterns with the change-proneness of files implementing services that are not antipatterns, we test the two following *null* hypotheses:

H_{02}^1: *there is no difference between the total number of changes experienced by files from groups* $(GA_i)_{i \in \{1, \ldots 8\}}$ *and* $GNoAP$.

H_{02}^2: *there is no difference between the total number of code churns experienced by files from groups* $(GA_i)_{i \in \{1, \ldots 8\}}$ *and* $GNoAP$.

We use the Kruskal-Wallis test to examine H_{02}^1 and H_{02}^2. The two hypotheses are two-
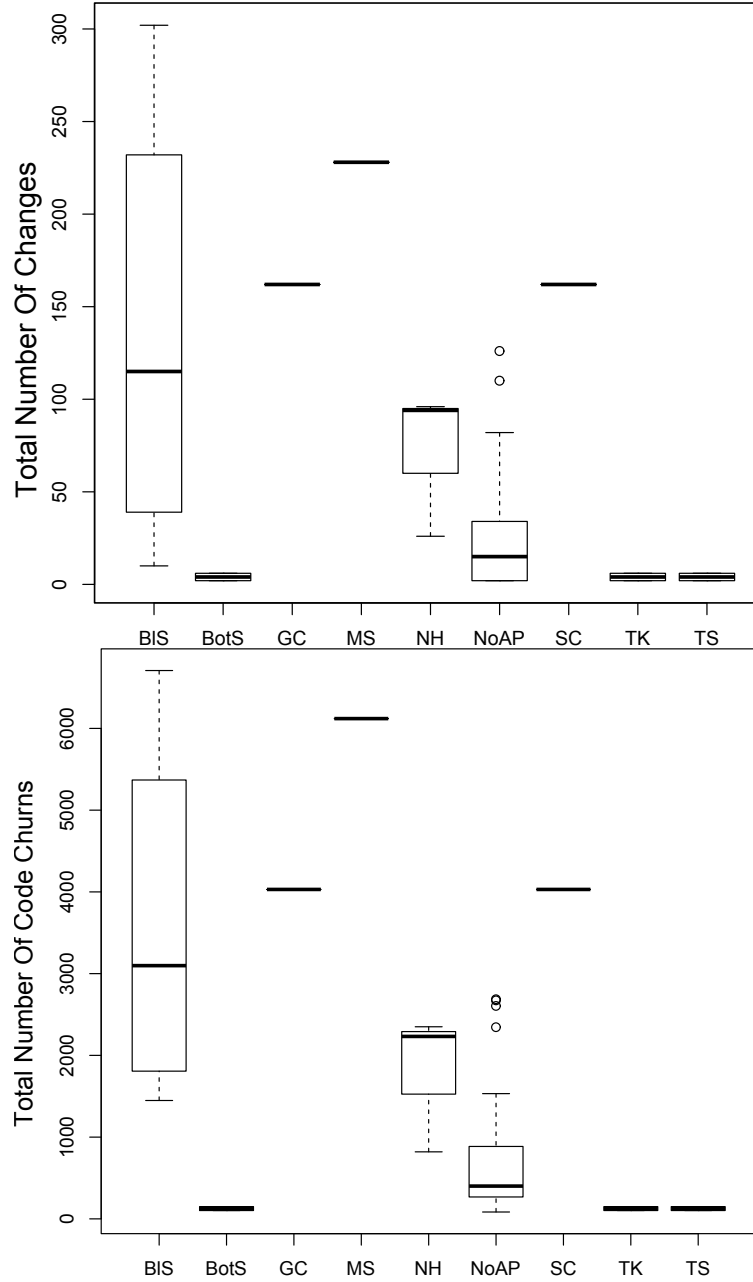
Figure 7.3: Comparison among Antipattern Services in Terms of Total Number of Changes (top) and Code Churns (bottom).

tailed (as in $\mathbf{RQ}_{2.1}$). We test H^1_{02} and H^2_{02} using the 5% significance level (*i.e.*, p-value<0.05).

Findings: The eight kinds of antipatterns investigated in this study are not equally change-prone. Figure 7.3 presents the box-plots showing the medians of the total number of changes (top) and total number of code churns (bottom) in the nine groups (eight groups for the eight kinds of antipatterns and the no antipattern group). The result of the Kruskal-Wallis

test presented in Table 7.7 suggests that the difference is statistically significant. Hence, we reject both H_{02}^1 and H_{02}^2. From Figure 7.3, we observe that *God Component* (GC), *Multi service* (MS), and *Service Chain* (SC) antipatterns have code churn values greater than 4000, while most other kinds of antipatterns have churn values less than 2000. As for the number of changes, the highest median value for a kind of antipattern is between 160 and 230 while others range from ∼10 to ∼100. Overall, *God Component, Multi service*, and *Service Chain* antipatterns are more change-prone than other kinds of antipatterns with significantly higher median than other antipatterns.

The confirms this observation (*i.e.*, p-value=0.00017 for changed LOC and p-value=0.01003 for code churns), therefore we can strongly reject H_{02}. In particular, we observed *God Component, Multi service*, and *Service Chain* antipatterns are more change-prone with significantly higher median than other antipatterns.

Table 7.7: Kruskal-Wallis Test for the Different Kinds of Service Antipatterns.

Test Types	p-value
total number of code churns ∼ antipattern	0.0002
total number of changes ∼ antipattern	0.01003

Summary: Services involved in *God Component, Multi service*, and *Service Chain* antipatterns, in terms of their implementations, are more change-prone than services involved in other kinds of service antipatterns. The difference is statistically significant.

We now discuss the possible reasons behind the high change-proneness of these three kinds of antipatterns. The service `ComponentFactory`, identified as *Service Chain* [Palma *et al.* (2013)] and *God Component* [Palma *et al.* (2013)] antipattern, is implemented by the `component-factory` FraSCAti component. The main role of this service is to generate and instantiate SCA components, which is one of the major steps to execute an SCA application. When an SCA application executes, it follows several sequential steps, including loading the SCA configuration file, parsing it, instantiating the SCA components, resolving the bindings, and so on. Therefore, the `ComponentFactory` service is in that invocation chain and highly related to other collaborating services. Because of this strong dependency, if others change, there is a high possibility that this `ComponentFactory` will also change frequently. Being a *God Component*, the `ComponentFactory` service also has a high number of encapsulated services with many methods and parameters.

The service `MembraneGeneration`, identified as *Multi Service* antipattern, is implemented by the `component-factory-juliac` FraSCAti component. The `MembraneGeneration` service wraps SCA components with the help of `ComponentFactory` service, in this way it helps each SCA components to be treated as an individual entity. According to the specification of *Multi Service* [Palma *et al.* (2013)], we found that the `MembraneGeneration` service had a high number of low cohesive methods defined in its interface, which might cause its frequent and large changes. Among the less change-prone antipatterns: *Bottleneck Service* (BotS), *The Knot* (TK), and *Tiny Service* (TS) show the very low number of changes and code churns. Also, the *Bloated Service* (BlS) antipattern change with a significant variation, *i.e.*, large inter-quartile range, in the number of changes and code churns. Based on these findings, development teams could decide to prioritise the code of services involved in the *Service Chain* (SC), *Multi Service* (MS), and *God Component* (GC) antipatterns, for special reviews and refactoring, since as shown in Figure 7.3, they have a high change-proneness. We have also investigated the change-proneness of the four kinds of service patterns found in FraSCAti (*i.e.*, *Basic service*, *Adapter*, *Facade*, and *Proxy pattern*). We report the possible relationship in the next section.

RQ$_{2.3}$: What is the relation between service patterns and change-proneness?

Motivation: The SOA paradigm has a specific set of design principles associated with it. Over the past years, patterns for SOA (*i.e.*, service patterns) have been proposed to guide developers through the application of these design principles, in order to help them reap the benefits of SOA, which includes fast and cost-effective responses to changes [Koch (2005)]. Each SOA service pattern affects and influences the application of one or more SOA design principles. There are also adverse relationships, where the results and trade-offs of some service patterns have a negative impact on a design principle [Erl (2009)]. A violation of some design principles can in turn result in a degradation of the quality of the SBSs. A better understanding of relations between service patterns and SBSs software quality is therefore important to guide development teams in making good design decisions. Yet, to date, no empirical evidence is available to validate the positive or negative impact of service patterns on the quality of SBSs. In this research question, we investigate the effect of service patterns on code change-proneness. Code change-proneness is an important quality attribute since it captures the effort required to modify and evolve the code of the SBSs, which translates into maintenance costs.

Approach: We answer this research question in three steps: First, we perform the detection of service design patterns using SODOP [Demange *et al.* (2013)] and obtain a set of FraS-
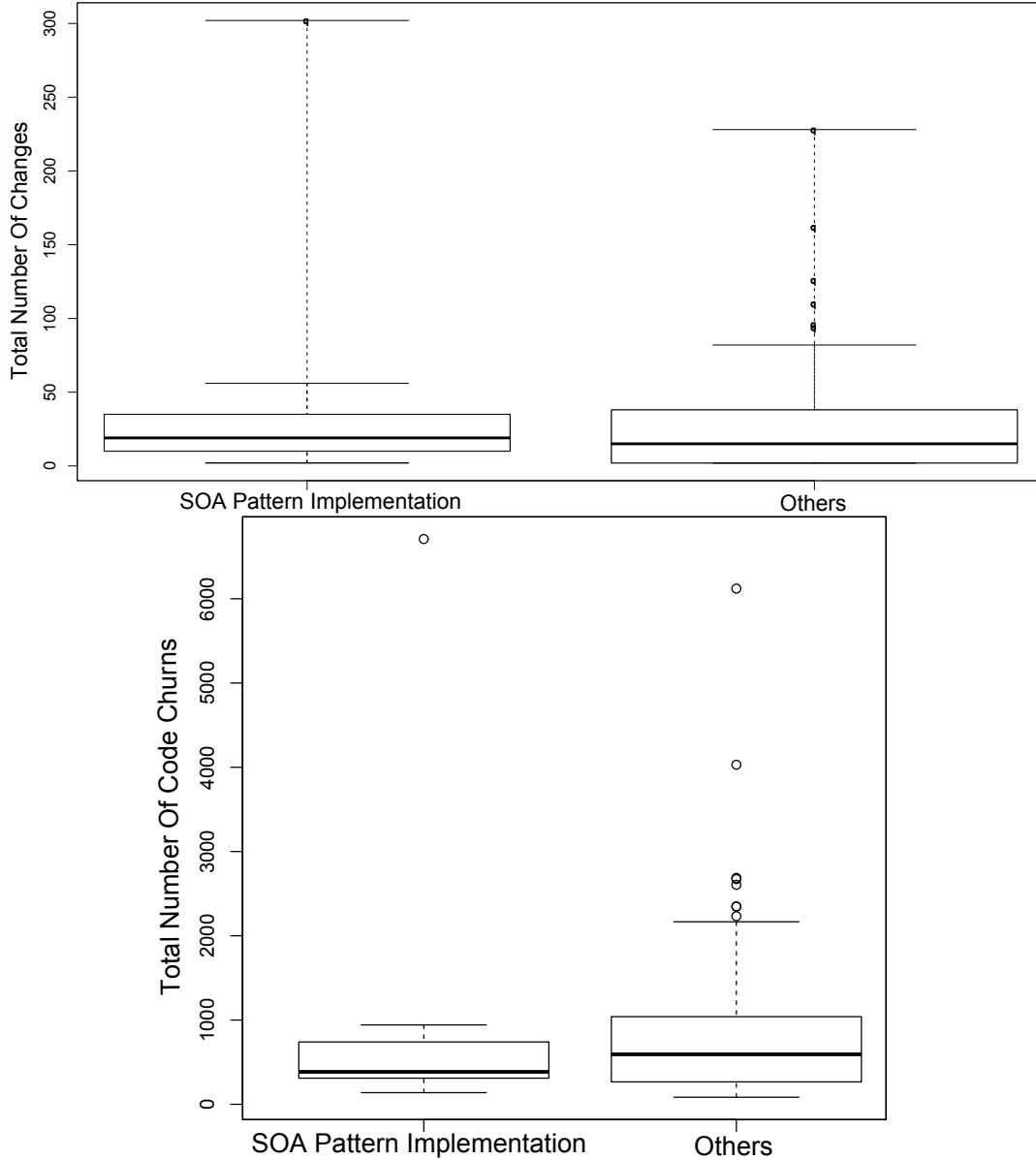
Figure 7.4: Comparison Between Pattern Services and Non-pattern Services in Terms of Number of Changes (top) and Code Churns (bottom).

CAti services involved in different design patterns. We manually validate the results of our detection as discussed in Section 7.4.1. Next, for each file implementing a FraSCAti service, we measure the change-proneness of the service's implementation using the two metrics: *total number of changes* and *total number of code churns.*

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

H_{03}^1: *there is no difference between the total number of changes experienced by files involved in the implementation of a service pattern and other files.*

H_{03}^2: *there is no difference between the total number of code churns experienced by files involved in the implementation of a service pattern and other files.*

We use the Wilcoxon rank sum test to examine H_{03}^1 and H_{03}^2. H_{03}^1 and H_{03}^2 are two-tailed since they investigate whether service patterns are related to higher or lower change and code churn rates. All the tests are performed using the 5% significance level (*i.e.*, p-value< 0.05).

Findings: Services involved in service patterns are less change-prone than the services not involved in any service pattern, but not at statistically significant level. Figure 7.4 presents the box-plots showing the median difference between pattern services and non-pattern services, both for the total number of changes (top) and the total number of code churns (bottom). In Figure 7.4, we observe the difference between the median values of the two groups. However, this difference is not statistically significant as the Wilcoxon rank sum test yielded p-values of 0.487 (>0.1) and 0.603 (>0.1), for respectively the total number of changes and the total number of code churns (see Table 7.8). The Cliff's δ effect size values presented in Table 7.9 also show a negligible difference.

Table 7.8: The Wilcoxon rank sum test between service patterns and other services.

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

Table 7.9: The non-parametric Cliff's δ effect size measure between service patterns and other services.

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

7.6 Threats to Validity

For this study, the *construct validity* threats refer to the relation between theory and observation, which is apparent by the measurement errors. The identification of changes and code churns in this study is reliable because we rely on the FraSCAti mailing list archives.

Summary: Services involved in service patterns, in terms of their implementations, are less change-prone than the services that are not involved in any service pattern. The total number of source code changes and code churns performed during the maintenance and evolution of services involved in a service pattern is lower than the total number of source code changes and code churns performed on other services. However, the difference is not statistically significant.

In this study, we only look for the number of changes and code churns for a service artefact. We plan to investigate and quantify the types of changes in the future. SODOP [Demange *et al.* (2013)] and SODA [Palma *et al.* (2013)] reflect their authors' subjective understanding of the service patterns and antipatterns, but they have good detection accuracy. Moreover, the service patterns and antipatterns instances used in this study were manually validated by the developers of FraSCAti, which minimises the threats to construct validity. Another threat to this validity that might affect us, in $RQ_{2.1}$, for some service patterns we had very few data points. We did not investigate the reason of the introduction of service patterns or antipatterns analysed in SODOP [Demange *et al.* (2013)] and SODA [Palma *et al.* (2013)]. *External validity* threats concern the possibility to generalise our findings. Further validation should be done on other service-based systems (SBSs) to better analyse the impact of service patterns and antipatterns on the change-proneness. One major challenge to minimise the threat to the external validity is the very limited availability of open-source SBSs. The FraSCAti project that we have studied is the largest open-source SBS available presently. It contains 130 services and 91 SCA components. Also, we have used a representative set of service patterns and antipatterns in our study. Finally, the *conclusion validity* threats refer to the relation between the treatment and the outcome. We paid full attention not to violate the assumptions of the performed statistical tests. We mainly used non-parametric tests that do not require making any presumption about the data distribution.

7.7 Discussion

This chapter reports on the results of an empirical study aimed at quantifying the impact of service *patterns* and *antipatterns* on the change-proneness of service-based systems (SBSs). We performed the detection of five service patterns and 13 service antipatterns using SODOP [Demange *et al.* (2013)] and SODA [Palma *et al.* (2013)], respectively, and answered three research questions $\mathbf{RQ_{2.1}}$ to $\mathbf{RQ_{2.3}}$. Results show that the services involved in antipatterns, in terms of their implementations, are more change-prone than the services that are not

involved in any antipattern (**RQ**$_{2.1}$). The services involved in *God Component*, *Multi service*, and *Service Chain* antipatterns, in terms of their implementations, are more change-prone than services involved in other kinds of service antipatterns (**RQ**$_{2.2}$). Results also show that the services involved in patterns, in terms of their implementations, are less change-prone than other services; however, this difference is not statistically significant (**RQ**$_{2.3}$).

However, for this study on the impact of service antipatterns on the change-proneness of services, we did not investigate the types of changes underwent for the services. Investigating the types of changes, *e.g.*, if they are related to new external requirements or to the improvement of existing code quality, will reveal more interesting facts on the correlation between service antipatterns and services' change-proneness.

In summary, we found that the services that involve antipatterns are more subject to change during maintenance and evolution, supporting the claim that the presence of antipatterns is an indicator of software quality. However, this observation might not be the *only* causal effect. Although, in a more general sense, this observation can be true because more skilled programmers make greater use of patterns and lesser use of antipatterns. Therefore, the developers' skills also impact the quality of design and implementation. Moreover, code reviews might tend to remove antipatterns and introduce patterns to improve the code quality. As a result, the more code reviews a service implementation will go through, the less maintenance effort it is likely to face in the future.

CHAPTER 8   CONCLUSION AND RESEARCH PERSPECTIVES

8.1   Conclusion

Service-based Systems (SBSs), relying on *services* as first-class entities, are developed on top of diverse SBSs technologies and architectural styles. SOAP Web services, SCA (Service Component Architecture), and REST are widely used by companies to design and develop SBSs [Fielding (2000); Chappell (2007); Alonso *et al.* (2003)].

SBSs are subject to *functional* and *non-functional* changes, which may degrade their design and implementation and introduce *service antipatterns*. Antipatterns in SBSs (1) may hinder their further maintenance and evolution and (2) may degrade their design quality and quality of service (QoS). Such antipatterns must be detected to improve their design and QoS, and ease their maintenance and evolution. Detecting antipatterns in SBSs developed using various technologies requires the analysis of their design and QoS. However, diverse SBSs technologies vary in their (1) building blocks, (2) composition styles, (3) development methodology, and (4) communication or client interaction styles. These differences pose challenges to develop a unified framework for the specification and detection of service antipatterns in SBSs. Moreover, the current literature did not consider service antipatterns with great importance and there exist some potential problems highlighted as follows:

– Problem 1. No unified abstraction of various SBSs technologies.
– Problem 2. No specification of service antipatterns.
– Problem 3. No dedicated unified approach and framework for the detection of service antipatterns.
– Problem 4. No empirical evidence on the impact of service antipatterns on service-based systems.

On solving the above problems, we formulated our thesis statement as below:

> "A unified approach to assessing the design and quality of service (QoS) of SBSs, supported by a framework for specifying and detecting service antipatterns, can facilitate the maintenance and evolution of SBSs, with the conjecture that service antipatterns may degrade the design and QoS, and hinder the future maintenance and evolution of SBSs."

Our Solution: To support the above thesis statement and solve the problems identified from the literature, we presented a unified abstraction and a meta-abstraction to support the comprehension and specification of service antipatterns in different SBSs technologies. Then, we proposed a unified approach, SODA (Service Oriented Detection for Antipatterns) that uses the unified abstraction and meta-abstraction, to assess the design and QoS of SBSs by means of the specification and detection of service antipatterns in SBSs. SODA is supported by an underlying framework, SOFA (Service Oriented Framework for Antipatterns), which provides a common platform to (1) specify service antipatterns at higher-level of abstraction or define detection heuristics, (2) automatically generate detection algorithms for service antipatterns, and (3) apply generated algorithms on diverse SBSs and report suspicious services involved in service antipatterns. Our SOFA framework is also capable of performing the syntactic and semantic analyses of services interfaces based on WordNet [Miller (1995)] and Stanford CoreNLP [Manning *et al.* (2014)].

In this dissertation, we investigated the two following research questions:

– **RQ$_1$:** *Can we efficiently specify and detect service antipatterns in different development technologies and architectural styles of service-based systems in terms of detection accuracy and performance?*
  We **positively** supported four assumptions:
  (1) Using our proposed domain specification language and the proposed unified abstraction, we specified 31 service antipatterns from different SBSs technologies;
  (2) The overall accuracy of our detection algorithms is high, *i.e.*, the average precision is **88%** and the average recall is **95.5%**;
  (3) Our detection framework and the proposed domain specification language are extensible for adding new service antipatterns and new SBSs technologies; and,
  (4) The average detection time for all 31 service antipatterns is **27.086s**.

– **RQ$_2$:** *What are the impact of service antipatterns and patterns on the maintenance and evolution of service-based systems?*
  We found a **negative** impact of service antipatterns on the change-proneness of SBSs:
  (1) The total number of source code changes and code churns performed during the maintenance and evolution of services involved in antipatterns is higher than the total number of source code changes and code churns performed on other services—the difference is statistically significant;
  (2) Services found to be involved in *God Component*, *Multi service*, and *Service Chain* antipatterns are more change-prone than services involved in other antipatterns—the

difference is statistically significant; and,

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

In summary, this dissertation made a contribution to the field of SOA by presenting a first unified approach, SODA, for specifying and detecting bad design practices, *i.e., service antipatterns*, in SBSs regardless of their underlying implementation technologies.

Therefore, the main contributions of this dissertation included:

1. A **unified abstraction** combining different SBSs technologies and architectural styles showing the differences and commonalities among them;

2. On top of the unified abstraction, a **service DSL** to specify service antipatterns regardless of SBSs technologies with higher-level of abstractions;

3. Using the unified abstraction and the DSL, a **unified SODA approach** for the specification and detection of service antipatterns in SBSs technologies;

4. An **extensive validation of SODA** using precision, recall, and F_1-measure on the largest SCA system, FraSCAti, more than 120 SOAP Web services, and 15 well-known RESTful APIs.

5. An **empirical evidence on the impact of service antipatterns** and patterns on the maintenance and evolution of service-based systems, in particular on SCA systems.

8.2   Research Perspectives

Service-Oriented Architecture (SOA) and Service-based Systems (SBSs) are widely adopted in practice as an architecture and as an emerged software development style, respectively. Accordingly, to get the best out of SOA and SBSs, software engineers and developers must deliver the final software product of good design quality, easy to maintain and extend. To assist SBSs developers and engineers, this dissertation proposed an approach capable of rigorous analysis of SBSs by assessing their design and quality of service. Beside this dissertation, which provides a first unified approach for assessing the design and quality of service of SBSs, we have several short and long-term research plans (in addition to the detection of service antipatterns).

8.2.1   Short-term Perspectives

We plan to perform the following research activities, which will compliment this dissertation:

1. In this dissertation, we performed the detection of linguistic antipatterns related to the lexical design of resources URIs used to make HTTP requests. We plan to apply SODA on other RESTful APIs and consider linguistic antipatterns related to responses, *i.e.*, how well the REST responses are designed from lexical aspects, which may affect their comprehension and reusability. We plan also to include domain-specific ontologies in the semantic analyses to overcome the limitations of English dictionaries;

2. For the extensive validation of REST antipatterns, we want to replicate SODA on other RESTful APIs with other REST patterns and antipatterns. Moreover, we intend to enrich the catalog of REST antipatterns by thoroughly investigating a large set of RESTful APIs;

3. We plan to convert all our REST heuristics into rule cards. This conversion will extend our DSL and make it more generic—applicable to all SBSs implementation technologies. Thus, we will be able to analyse all the three SBSs technologies using the same mechanism: specifying rule cards to automatically generate detection algorithms, and, later on, automatically apply those detection algorithms on REST resources and URIs;

4. The detection of REST antipatterns is now semi-automatic, *i.e.*, the detection requires significant manual involvement from wrapping RESTful APIs with SCA components to implementing detection algorithms from predefined heuristics. We plan to automate more these wrapping and implementation steps to lessen manual efforts;

5. To analyse the impact of service antipatterns, we plan to replicate the study presented in Chapter 7 on other SBSs, *e.g.*, Web services and RESTful APIs, with different service antipatterns. However, one major challenge is the availability of open-source SBSs and the evolution history of Web services. In connection to this plan, we are also interested in investigating the types of changes made and their impact on service antipatterns. Furthermore, using bug reports we want to explore the possible relation between service antipatterns and fault-proneness;

6. For the process antipatterns, as discussed in the Introduction chapter, we analysed the business processes statically. We plan to perform dynamic analyses of business processes by executing them to collect their runtime properties. An automatic correction of process antipatterns is also in our future plan; and, finally,

7. We want to study the evolution of service antipatterns in SBSs, *i.e.*, the study on how, when, and why service antipatterns are introduced and their lifespan;

8.2.2 Long-term Perspectives

In addition to the above short-term research activities, we also plan to perform the following long-term research activities:

1. We plan to conduct more experiments and analyse results in an industrial setup using real systems. To conduct such experiments, we need to perform some preparatory steps including:
   – Instrumenting the target service-based system;
   – Deploy our SOFA framework within their (*i.e.*, industrial partners') platform; and,
   – Finally, perform the three steps in our proposed SODA approach to detect service antipatterns, and report suspicious services.
   Summary: The experiments with industrial setup will help us to fine-tune our approach and the framework more, and facilitate the approach and the framework to be more matured.

2. We also plan to propose a corrective approach, which follows the detection of service antipatterns as a long-term research activity. To perform the correction of detected service antipatterns, finding an alternative design is a crucial and mandatory step. Thus, for the refactoring, we intend to follow steps in the below:
   – To perform a thorough literature review on the treatments of service antipatterns;
   – To propose an independent refactoring approach, possibly supported by an underlying framework; finally,

 – To perform the correction and validation of the refactored service antipatterns by using the proposed refactoring approach in the previous step by the actual services developers;

 Summary: Through this research activity we will be able to rectify the detected service antipatterns in SBSs.

3. We plan to perform a study showing how refactored service antipatterns can improve the maintainability of an SBS in terms of cost and efforts where the cost can be measured as the maintenance expenses and the efforts can be measured with time spent on the program comprehension, design analysis, and required modifications.

REFERENCES

Marwen Abbes and Foutse Khomh and Yann-Gaël Guéhéneuc and Giuliano Antoniol (2011). An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. *15th European Conference on Software Maintenance and Reengineering.* 181–190.

Wided Ben Abid and Mohamed Graiet and Mourad Kmimech and Mohamed Tahar Bhiri and Walid Gaaloul and Eric Cariou (2011). Profile UML2.0 for Specification of the SCA Architectures. *Semantics, Knowledge and Grid, International Conference on, 0,* 191–194.

Gustavo Alonso and Fabio Casati and Harumi Kuno and Vijay Machiraju (2003). Web Services: Concepts, Architectures and Applications. Springer, Data-Centric Systems and Applications.

Alexandre Alves and et al. (2007). Web Services Business Process Execution Language Version 2.0. Rapport technique.

Anchuri, Pranay and Sumbaly, Roshan and Shah, Sam (2014). Hotspot Detection in a Service-Oriented Architecture. *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management.* ACM, New York, NY, USA, CIKM '14, 1749–1758.

Keith H. Bennett and Václav T. Rajlich (2000). Software Maintenance and Evolution: A Roadmap. *Proceedings of the Conference on The Future of Software Engineering.* ACM, New York, NY, USA, ICSE '00, 73–87.

Berners-Lee, Tim and Fielding, Roy Thomas and Masinter, L. (2005). Uniform Resource Identifier (URI): Generic Syntax.

Boehm, B. W. and Brown, J. R. and Lipow, M. (1976). Quantitative Evaluation of Software Quality. *Proceedings of the 2Nd International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, USA, ICSE '76, 592–605.

William J. Brown and Raphael C. Malveau and Hays W. "Skip" McCormick and Thomas J. Mowbray (1998). *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons.

John M. Chambers and William S. Cleveland and Paul A. Tukey and Beat Kleiner (1983). *Graphical Methods for Data Analysis.* Wadsworth International.

Dennis De Champeaux and Douglas Lea and Penelope Faure (1993). *Object-oriented System Development.* Lectures in Mathematics Eth Zurich. Addison-Wesley.

David Chappell (2007). *Introducing SCA*. Chappell & Associates, USA.

Luba Cherbakov and Mamdouh Ibrahim and Jenny Ang (2006). SOA Antipatterns: The Obstacles to the Adoption and Successful Realization of Service-Oriented Architecture.

Erik Christensen and Francisco Curbera and Greg Meredith and Sanjiva Weerawarana (2011). Web Services Description Language (WSDL) 1.1. Rapport technique, W3C.

Cliff, Norman (1993). Dominance Statistics: Ordinal Analyses To Answer Ordinal Questions. *Psychological Bulletin, 114*(3), 494–509.

Jacob Cohen (1988). *Statistical Power Analysis For The Behavioral Sciences.* Lawrence Erlbaum, seconde édition.

Cohen, Jacob (1992). A Power Pprimer. *Psychological Bulletin, 112*(1), 155–159.

Coleman, Don and Ash, Dan and Lowther, Bruce and Oman, Paul (1994). Using Metrics to Evaluate Software System Maintainability. *Computer, 27*(8), 44–49.

Cortellessa, Vittorio and Di Marco, Antinisca and Trubiani, Catia (2012). Software Performance Antipatterns: Modeling and Analysis. M. Bernardo, V. Cortellessa et A. Pierantonio, éditeurs, *Formal Methods for Model-Driven Engineering*, Springer Berlin Heidelberg, vol. 7320 de *Lecture Notes in Computer Science.* 290–335.

Cortellessa, Vittorio and Di Marco, Antinisca and Trubiani, Catia (2014). An Approach for Modeling and Detecting Software Performance Antipatterns based on First-order Logics. *Software & Systems Modeling, 13*(1), 391–432.

Cortellessa, Vittorio and Martens, Anne and Reussner, Ralf and Trubiani, Catia (2010). A Process to Effectively Identify "guilty" Performance Antipatterns. *Fundamental Approaches to Software Engineering*, Springer. 368–382.

Jose Luis Ordiales Coscia and Marco Crasso and Cristian Mateos and Alejandro Zunino (2013). Estimating Web Service Interface Quality Through Conventional Object-oriented Metrics. *CLEI Electronic Journal, 16.*

Curbera, Francisco and Khalaf, Rania and Nagy, William A. and Weerawarana, Sanjiva (2006). Implementing bpel4ws: the architecture of a bpel4ws implementation. *Concurrency and Computation: Practice and Experience, 18*(10), 1219–1228.

Robert Daigneau (2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley.

Anthony Demange and Naouel Moha and Guy Tremblay (2013). Detection of SOA Patterns. S. Basu, C. Pautasso, L. Zhang et X. Fu, éditeurs, *Service-Oriented Computing.* Springer Berlin Heidelberg, vol. 8274 de *Lecture Notes in Computer Science*, 114–130.

Dromey, R. Geoff (1995). A model for software product quality. *IEEE Trans. Softw. Eng.*, *21*(2), 146–162.

Bill Dudney and Stephen Asbury and Joseph K. Krozak and Kevin Wittkopf (2003). *J2EE AntiPatterns*. John Wiley and Sons.

Edstrom, J. and Tilevich, E. (2012). Reusable and Extensible Fault Tolerance for RESTful Applications. *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. 737–744.

Mike Edwards (2011). *Service Component Architecture (SCA)*. OASIS, USA.

EMF-Eclipse (2010). Eclipse Modeling Framework (EMF) - http://www.eclipse.org/emf.

EMFText (2007). http://www.emftext.org/. Rapport technique, www.eclipse.org/acceleo.

Erl, Thomas (2004). *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Thomas Erl (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR.

Erl, Thomas (2007). *SOA Principles of Service Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Thomas Erl (2009). *SOA Design Patterns*. Prentice Hall PTR.

Thomas Erl and Benjamin Carlyle and Cesare Pautasso and Raj Balasubramanian (2012). *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. The Prentice Hall Service Technology Series from Thomas Erl. Prentice Hall.

John Evdemon (2005). Principles of Service Design: Service Patterns and Anti-Patterns.

Roy Thomas Fielding (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Thèse de doctorat.

Martin J. Fowler and Kent Beck and John Brant and William Opdyke and Don Roberts (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

FraSCAti (June 2013). Home-Automation: websvn.ow2.org/listing.php?repname=frascati&path=/trunk/demo/home-automation/.

Todd Fredrich (2012). RESTful Service Best Practices: Recommendations for Creating Web Services.

Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Garcia, J. and Popescu, D. and Edwards, G. and Medvidovic, N. (2009). Identifying Architectural Bad Smells. *13th European Conference on Software Maintenance and Reengineering (CSMR)*. 255–258.

Randy Heffner and Carey Schwaber and Jonathan Browne and Tim Sheedy and Jacqueline Stone and Gene Leganza (2007). Planned SOA Usage Grows Faster Than Actual SOA Usage. *Business Data Services North America, Europe, And Asia Pacific.* Forrester.

Andreas Hess and Eddie Johnston and Nicholas Kushmerick (2004). ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services. *In Proceedings of International Semantic Web Conference.*

K.K Holgeid and J Krogstie and Dag I.K Sjøberg (2000). A Study of Development and Maintenance in Norway: Assessing the Efficiency of Information Systems Support using Functional Maintenance. *Information and Software Technology, 42*(10), 687–700.

ISBSG (2005). Application Software Maintenance and Support: An Initial Analysis of New Data. Rapport technique.

ISO/IEC (2011). Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. *Reference Number: ISO/IEC 25010:2011.*

ISO/IEC/IEEE (2010). Systems and Software Engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, 1–418.

Steve Jones (2006). SOA Anti-patterns, Available Online: www.infoq.com/articles/SOA-anti-patterns.

Khomh, F. and Di Penta, M. and Guéhéneuc, Y.G. (2009). An Exploratory Study of the Impact of Code Smells on Software Change-proneness. *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on.* 75–84.

Foutse Khomh and Massimiliano Di Penta and Yann-Gaël Guéhéneuc and Giuliano Antoniol (2012a). An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. *Empirical Software Engineering, 17*(3), 243–275.

Khomh, Foutse and Penta, Massimiliano Di and Guéhéneuc, Yann-Gaël and Antoniol, Giuliano (2012b). An exploratory study of the impact of antipatterns on class change- and fault-proneness. Kluwer Academic Publishers, Hingham, MA, USA, vol. 17, 243–275.

Foutse Khomh and Stephane Vaucher and Yann-Gaël Guéhéneuc and Houari Sahraoui (2011). BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns. *Journal of Systems and Software, 84*(4), 559–572.

Khurana, R. (2007). *Software Engineering: Principles and Practices.* Vikas Publication.

Kitchenham, Barbara (1987). Towards a Constructive Quality Model: Part 1: Software Quality Modelling, Measurement and Prediction. *Software Engineering Journal, 2*(4), 105–113.

Barbara Kitchenham (2004). Procedures for performing systematic reviews.

Christopher Koch (2005). A New Blueprint For The Enterprise. *CIO Magazine.*

Jana Koehler and Jussi Vanhatalo (2007). Process Anti-Patterns: How to Avoid the Common Traps of Business Process Modeling. *IBM WebSphere Developer Technical Journal.*

John R. Koza (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge.

Jaroslav Král and Michal Žemlička (2007). The Most Important Service-Oriented Antipatterns. *International Conference on Software Engineering Advances.* 29–29.

Jaroslav Král and Michal Žemlička (2008). Crucial Service-Oriented Antipatterns. *In Proceedings of the International Conference on Software Engineering Advances.* International Academy, Research and Industry Association (IARIA), vol. 2, 160–171.

Jaroslav Král and Michal Žemlička (2009). Popular SOA Antipatterns. *Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns.* IEEE Computer Society, Washington, DC, USA, COMPUTATION-WORLD '09, 271–276.

B. V. Kumar (2004). *Web Services.* McGraw-Hill Education (India) Pvt Limited.

Laitinen, Kari (1996). Estimating Understandability of Software Documents. *SIGSOFT Softw. Eng. Notes, 21*(4), 81–92.

Lientz, B.P. and Swanson, E.B. (1980). *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations.* Addison-Wesley.

Maiga, Abdou and Ali, Nasir and Bhattacharya, Neelesh and Sabane, Aminata, and Yann-Gaël Guéhéneuc and Aimeur, Esma (2012). SMURF: A SVM-based Incremental Anti-pattern Detection Approach. *2012 19th Working Conference on Reverse Engineering (WCRE).* IEEE, 466–475.

Manning, Christopher D. and Surdeanu, Mihai and Bauer, John and Finkel, Jenny and Bethard, Steven J. and McClosky, David (2014). The Stanford CoreNLP Natural Language Processing Toolkit. *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations.* Association for Computational Linguistics, Baltimore, Maryland, 55–60.

Mika V. Mäntylä and Casper Lassenius (2006). Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. *Empirical Software Engineering, 11*(3), 395–431.

Antinisca Di Marco and Catia Trubiani (2014). A model-driven approach to broaden the detection of software performance antipatterns at runtime. *Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, Grenoble, France.* 77–92.

Massé, Mark (2012). *REST API Design Rulebook.* O'Reilly.

Mateos, Cristian and Crasso, Marco and Zunino, Alejandro and Coscia, José Luis Ordiales (2011). Detecting WSDL Bad Practices in Code-first Web Services. *International Journal of Web and Grid Services*, *7*(4), 357–387.

Mira Kajko Mattsson and Gerardo Canfora and Dan Chiorean and Tuomas Ihme and Meir M Lehman and Rupert Reiger and Eads Deutschland Gmbh and Torsten Engel (2006). A Model of Maintainability - Suggestion for Future Research. *International Conference on Software Engineering Research and Practice.*

Microsoft:MSDN (1992). Capitalization Styles, Available On: https://msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx.

Miller, George A. (1995). WordNet: A Lexical Database for English. *Commun. ACM*, *38*(11), 39–41.

Tarak Modi (2006). SOA Management: SOA Antipatterns.

Naouel Moha and Yann-Gaël Guéhéneuc and Laurence Duchien and Anne-Francoise Le Meur (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transaction on Software Engineering*, *36*(1), 20–36.

Naouel Moha and Francis Palma and Mathieu Nayrolles and Benjamin Joyen Conseil and Yann-Gaël Guéhéneuc and Benoit Baudry and Jean-Marc Jézéquel (2012). Specification and Detection of SOA Antipatterns. C. Liu, H. Ludwig et F. Toumani, éditeurs, *Proceedings of the 10th International Conference on Service Oriented Computing (ICSOC).* Springer. Runner-up best paper. 15 pages.

Mathieu Nayrolles and Naouel Moha and Petko Valtchev (2013). Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces. *20th Working Conference on Reverse Engineering.* 321–330.

Newcomer, Eric and Lomow, Greg (2004). *Understanding SOA with Web Services (Independent Technology Guides).* Addison-Wesley Professional.

Obeo (2005). Acceleo. Rapport technique, www.eclipse.org/acceleo.

Ouni, Ali and Kessentini, Marouane and Sahraoui, Houari and Boukadoum, Mounir (2013). Maintainability Defects Detection and Correction: A Multi-objective Approach. *Automated Software Engineering*, *20*(1), 47–79.

Ouni, Ali and Kula, Raula Gaikovina and Kessentini, Marouane and Inoue, Katsuro (2015). Web Service Antipatterns Detection Using Genetic Programming.

Palma, Francis and Dubois, Johann and Moha, Naouel and Yann-Gaël Guéhéneuc (2014a). Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. X. Franch, A. Ghose, G. Lewis et S. Bhiri, éditeurs, *Service-Oriented Computing.* Springer Berlin Heidelberg, vol. 8831 de *Lecture Notes in Computer Science*, 230–244.

Palma, Francis and Moha, Naouel (2015). A Study on the Taxonomy of Service Antipatterns. *Patterns Promotion and Anti-patterns Prevention (PPAP), 2015 IEEE 2nd Workshop on.* IEEE, 5–8.

Francis Palma and Naouel Moha and Yann-Gaël Guéhéneuc (2015). Specification and Detection of Business Process Antipatterns. H. Mili, M. Benyoucef et M. Weiss, éditeurs, *Proceedings of the 6th International MCETECH Conference.* Springer International Publishing, Lecture Notes in Business Information Processing.

Palma, Francis and Moha, Naouel and Tremblay, Guy and Yann-Gaël Guéhéneuc (2014b). Specification and Detection of SOA Antipatterns in Web Services. P. Avgeriou et U. Zdun, éditeurs, *Software Architecture.* Springer International Publishing, vol. 8627 de *Lecture Notes in Computer Science*, 58–73.

Francis Palma and Mathieu Nayrolles and Naouel Moha and Yann-Gaël Guéhéneuc and Benoit Baudry and Jean-Marc Jézéquel (2013). SOA Antipatterns: An Approach for their Specification and Detection. *International Journal of Cooperative Information Systems*, *22*(04).

M. Papazoglou (2008). *Web Services: Principles and Technology.* Pearson Education. Prentice Hall.

Michael P. Papazoglou and Paolo Traverso and Schahram Dustdar and Frank Leymann (2003). Service-oriented Computing. *Communications of the ACM*, *46*, 25–28.

Trevor Parsons and John Murphy (2008). Detecting Performance Antipatterns in Component Based Enterprise Systems. *Journal of Object Technology*, *7*(3), 55–90.

Cesare Pautasso (2009). Some REST Design Patterns (and Anti-Patterns), Available Online: http://www.jopera.org/node/442.

Peiris, Manjula and Hill, James H. (2014). Towards Detecting Software Performance Antipatterns Using Classification Techniques. *SIGSOFT Software Engineering Notes*, *39*(1), 1–4.

Massimiliano Di Penta and Antonella Santone and Maria Luisa Villani (2007). Discovery of SOA Patterns via Model Checking. *2nd International Workshop on Service Oriented*

Software Engineering: In Conjunction with the 6th ESEC/FSE Joint Meeting. ACM, New York, USA, IW-SOSWE '07, 8–14.

Perepletchikov, Mikhail and Ryan, Caspar and Frampton, Keith (2005). Comparing the Impact of Service-Oriented and Object-Oriented Paradigms on the Structural Properties of Software. R. Meersman, Z. Tari et P. Herrero, éditeurs, *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, Springer Berlin Heidelberg, vol. 3762 de *Lecture Notes in Computer Science.* 431–441.

Pigoski, Thomas M. (1996). *Practical Software Maintenance: Best Practices for Managing Your Software Investment.* John Wiley & Sons, Inc., New York, NY, USA.

Roger S. Pressman (2010). *Software Engineering: A Practitioner's Approach.* McGraw-Hill higher education. McGraw-Hill Education.

Prieto-Díaz, Rubén (1990). Domain Analysis: An Introduction. *SIGSOFT Softw. Eng. Notes*, *15*(2), 47–54.

RFC2822 (2001). Internet Message Format by Internet Engineering Task Force. Rapport technique.

Juan Manuel Rodriguez and Marco Crasso and Cristian Mateos and Alejandro Zunino (2013). Best Practices for Describing, Consuming, and Discovering Web Services: A Comprehensive Toolset. *Software: Practice and Experience*, *43*(6), 613–639.

Juan Manuel Rodriguez and Marco Crasso and Alejandro Zunino and Marcelo Campo (2010a). Automatically Detecting Opportunities for Web Service Descriptions Improvement. Springer Berlin Heidelberg, vol. 341. 139–150.

Juan Manuel Rodriguez and Marco Crasso and Alejandro Zunino and Marcelo Campo (2010b). Improving Web Service Descriptions for Effective Service Discovery. *Science of Computer Programming*, *75*(11), 1001–1021.

Romano, Daniele and Raila, Paulius and Pinzger, Martin and Khomh, Foutse (2012). Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. *Proceedings of the 2012 19th Working Conference on Reverse Engineering.* IEEE Computer Society, Washington, DC, USA, WCRE '12, 437–446.

Jeanine Romano and Jeffrey D. Kromrey and Jesse Coraggio and Jeff Skowronek (2006). Appropriate Statistics For Ordinal Level Data: Should We Really Be Using T-Test And Cohen's D For Evaluating Group Differences On The NSSE And Other Surveys? *Annual Meeting of the Florida Association of Institutional Research.* 1–33.

H.Dieter Rombach and Bradford T. Ulery and Jon D. Valett (1992). Toward full life cycle control: Adding maintenance measurement to the {SEL}. *Journal of Systems and Software*, *18*(2), 125 – 138.

Michael Rosen and Boris Lublinsky and Kevin T. Smith and Marc J. Balcer (2008). *Applied SOA: Service-Oriented Architecture and Design Strategies.* Wiley.

Arnon Rotem-Gal-Oz and E. Bruno and U. Dahan (2012). *SOA Patterns.* Manning Publications Co.

Mazeiar Salehie and Shimin Li and Ladan Tahvildari (2006). A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. *Proceedings of the 14th IEEE International Conference on Program Comprehension.* IEEE Computer Society, Washington, DC, USA, ICPC, 159–168.

David Sciamma and Gilles Cannenterre and Jacques Lescot (2013). Ecore Tools. Rapport technique, www.eclipse.org/modeling/emft/?project=ecoretools.

Lionel Seinturier and Philippe Merle and Romain Rouvoy and Daniel Romero and Valerio Schiavoni and Jean-Bernard Stefani (2012). A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience, 42*(5), 559–583.

Sheskin, David J. (2007). *Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition.* Chapman & Hall/CRC.

Smith, Connie U. and Williams, Lloyd G. (2000). Software Performance Antipatterns. *Proceedings of the 2nd International Workshop on Software and Performance.* ACM, New York, NY, USA, WOSP '00, 127–136.

Connie U. Smith and Lloyd G. Williams (2002). New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. *International Computer Measurement Group Conference.* 667–674.

George Spanoudakis and Khaled Mahbub (2004). Requirements Monitoring for Service-based Systems: Towards a Framework based on Event Calculus. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on.* 379–384.

Alecsandar Stoianov and Ioana Sora (2010). Detecting Patterns and Antipatterns in Software using Prolog Rules. *International Joint Conference on Computational Cybernetics and Technical Informatics (ICCC-CONTI).* 253–258.

Stefan Tilkov (2008). REST Anti-Patterns, Available Online: www.infoq.com/articles/rest-anti-patterns.

Mohammad Ali Torkamani and Hamid Bagheri (2014). A Systematic Method for Identification of Antipatterns in Service Oriented System Development. *International Journal of Electrical and Computer Engineering (IJECE), 4*(1), 16–23.

Deepali Tripathi and Ugrasen Suman and Maya Ingle and S. K. Tanwani (2014). Towards Introducing and Implementation of SOA Design Antipatterns. *International Journal of Computer Theory and Engineering*, *6*(1), 20–25.

Trubiani, Catia and Di Marco, Antinisca and Cortellessa, Vittorio and Mani, Nariman and Petriu, Dorina (2014). Exploring Synergies Between Bottleneck Analysis and Performance Antipatterns. *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ACM, New York, NY, USA, ICPE '14, 75–86.

Nikolaos Tsantalis and Alexander Chatzigeorgiou and George Stephanides and Spyros T. Halkidis (2006). Design Pattern Detection Using Similarity Scoring. *IEEE Transaction on Software Engineering*, *32*(11), 896–909.

Turner, M. and Budgen, D. and Brereton, P. (2003). Turning software into a service. *Computer*, *36*(10), 38–44.

Valverde, Francisco and Pastor, Oscar (2009). Dealing with REST Services in Model-driven Web Engineering Methods. *V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB*, 243–250.

Wan-Kadir, W. M. N. and Loucopoulos, Pericles (2004). Relating evolving business rules to software design. *J. Syst. Archit.*, *50*(7), 367–382.

Wang, Shaohua and Keivanloo, Iman and Zou, Ying (2014). How Do Developers React to RESTful API Evolution? X. Franch, A. Ghose, G. Lewis et S. Bhiri, éditeurs, *Service-Oriented Computing*, Springer Berlin Heidelberg, vol. 8831 de *Lecture Notes in Computer Science*. 245–259.

Sanjiva Weerawarana and Francisco Curbera and Frank Leymann and Tony Storey and Donald Ferguson (2005). *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR.

Wert, Alexander and Oehler, Marius and Heger, Christoph and Farahbod, Roozbeh (2014). Automatic Detection of Performance Anti-patterns in Inter-component Communications. *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*. ACM, New York, NY, USA, QoSA '14, 3–12.

Woods, D. and Mattern, T. (2006). *Enterprise SOA: Designing IT for Business Innovation*. O'Reilly Media.

WWW-Consortium (2006). WWW Consortium, Web Services Description Language (WSDL) Version 2.0. Rapport technique.

Zhang, Lei and Sun, Yanchun and Song, Hui and Wang, Weihu and Huang, Gang (2012). Detecting Anti-patterns in Java EE Runtime System Model. *Proceedings of the Fourth*

Asia-Pacific Symposium on Internetware. ACM, New York, NY, USA, Internetware '12, 1–8.

Yongyan Zheng and Krause, P. (2006). Asynchronous Semantics and Anti-patterns for Interacting Web Services. *6th International Conference on Quality Software (QSIC).* 74–84.

Zuse, Horst (1997). *A Framework of Software Measurement.* Walter de Gruyter & Co., Hawthorne, NJ, USA.

List of Antipatterns in SBSs Technologies

Multi Service :
Multi Service also known as *God Object* corresponds to a service that implements a ***multitude of operations*** related to different business and technical abstractions. This service aggregates too many operations into a single service, such a service is not easily reusable because of the ***low cohesion*** of its operations and is often ***unavailable*** to end-users because of it is ***overloaded***, which may also induce a ***high response time***. However, the excessive load of this service can be reduced by deploying multiple instances of the service, which is not an inexpensive resolution [Dudney *et al.* (2003)].

```
1 RULE_CARD : MultiService {
2   RULE : MultiService {INTER MultiOperation HighResponse LowAvailability LowCohesion};
3   RULE : MultiOperation {NOD VERY_HIGH};
4   RULE : HighResponse {RT VERY_HIGH};
5   RULE : LowAvailability {A LOW};
6   RULE : LowCohesion {COH LOW};
7 };
```

Tiny Service :
Tiny Service is a small service with ***few methods***, which only implements part of an abstraction. Such service often requires ***several coupled*** services to be used together, resulting in higher development complexity and ***reduced usability***. In the extreme case, a *Tiny Service* will be limited to ***one method***, resulting in many services that implement an overall set of requirements [Dudney *et al.* (2003)].

```
1 RULE_CARD : TinyService {
2   RULE : TinyService {INTER FewOperation HighCouplingORLowCohesion};
3   RULE : FewOperation {NOD VERY_LOW};
4   RULE : HighCouplingORLowCohesion {UNION HighCoupling LowCohesion};
5   RULE : HighCoupling {CPL HIGH};
6   RULE : LowCohesion {COH LOW};
7 };
```

128

Sand Pile :

Sand Pile is also known as *"Fine-Grained Services"*. It appears when a service is **composed** by multiple smaller services sharing **common data**. It thus has a **high data cohesion**. The common data shared may be located in a **Data Service** antipattern (see below) [Král et Žemlička (2008)].

```
1 RULE_CARD : SandPile {
2   RULE : SandPile {COMPOS FROM ParentService ONE TO ChildService MANY};
3   RULE : ChildService {ASSOC FROM ContainedService MANY TO DataSource ONE};
4   RULE : ParentService {COH HIGH};
5   RULE : DataSource {RULE_CARD : DataService};
6   RULE : ContainedService {NRO > 1};
7 };
```

Chatty Service :

Chatty Service corresponds to a set of services that exchange a lot of **small data of primitive types**, usually with a **Data Service** antipattern. The *Chatty Service* is also characterized by a **high number of method invocations**. *Chatty Services* chat a lot with each other [Dudney *et al.* (2003)].

```
1 RULE_CARD : ChattyService {
2   RULE : ChattyService { INTER TotalInvocation DSRuleCard};
3   RULE : DSRuleCard {RULE_CARD : DataService};
4   RULE : TotalInvocation {NMI VERY_HIGH};
5 };
```

The Knot :

The Knot is a set of **very low cohesive** services, which are **tightly coupled**. These services are thus less reusable. Due to this complex architecture, the **availability** of these services may be **low**, and their **response time high** [Rotem-Gal-Oz *et al.* (2012)].

```
1 RULE_CARD : TheKnot {
2   RULE : TheKnot {INTER HighCoupling LowCohesion LowAvailability HighResponse};
3   RULE : HighCoupling {CPL VERY_HIGH};
4   RULE : LowCohesion {COH VERY_LOW};
5   RULE : LowAvailability {A LOW};
6   RULE : HighResponse {RT HIGH};
7 };
```

Nobody Home :

Nobody Home corresponds to a service, defined but actually never used by clients. Thus, the **methods** from this service are **never invoked**, even though it may be **coupled** to other services. Yet, it still requires deployment and management, despite of its non-usage [Jones (2006)].

```
1 RULE_CARD : NobodyHome {
2   RULE : NobodyHome {INTER IncomingReference MethodInvocation};
3   RULE : IncomingReference {NIR GREATER 0};
4   RULE : MethodInvocation {NMI EQUAL 0};
5 };
```

Duplicated Service :

Duplicated Service, a.k.a., *The Silo Approach*, introduced by IBM, corresponds to a set of **highly similar** services. Because services are implemented multiple times as a result of the silo approach, there may have **common** or **identical methods** with the **same names and–or parameters** [Cherbakov *et al.* (2006)].

```
1 RULE_CARD : DuplicatedService {
2   RULE : DuplicatedService {ANIM HIGH};
3 };
```

Bottleneck Service :

Bottleneck Service is a service that is highly used by other services or clients. It has a **high incoming and outgoing coupling**. Its **response time** can be **high** because it may be used by too many external clients, for which clients may need to wait to get access to the service. Moreover, its **availability** may also be low due to the traffic.

```
1 RULE_CARD : BottleneckService {
2   RULE : BottleneckService {INTER LowPerformance HighCoupling};
3   RULE : LowPerformance {INTER LowAvailability HighResponse};
4   RULE : HighResponse {RT HIGH};
5   RULE : LowAvailability {A LOW};
6   RULE : HighCoupling {CPL VERY_HIGH};
7 };
```

Service Chain :
Service Chain, a.k.a., Message Chain [Fowler *et al.* (1999)] in OO systems corresponds to a ***chain of services***. The *Service Chain* appears when clients request consecutive service invocations to fulfill their goals. This kind of ***dependency*** chain reflects the subsequent ***invocation*** of services.
1 RULE_CARD : *ServiceChain* { 2   RULE : *ServiceChain* {INTER *TransitiveInvocation LowAvailability*} ; 3   RULE : *TransitiveInvocation* {NTMI VERY_HIGH} ; 4   RULE : *LowAvailability* {A LOW} ; 5 } ;

Data Service :
Data Service, a.k.a., Data Class [Fowler *et al.* (1999)] in OO systems, corresponds to a service that contains ***mainly accessor methods***, *i.e.*, getters and setters. In the distributed applications, there can be some services that may only perform some simple information retrieval or ***data access*** to such services. *Data Services* contain usually ***accessor methods*** with ***small parameters*** of ***primitive types***. Such service has a ***high*** data ***cohesion***.
1 RULE_CARD : *DataService* { 2   RULE : *DataService* {INTER *HighDataAccessor SmallParameter PrimitiveParameter HighCohesion*} ; 3   RULE : *SmallParameter* {ANP LOW} ; 4   RULE : *PrimitiveParameter* {ANPT HIGH} ; 5   RULE : *HighDataAccessor* {ANAM VERY_HIGH} ; 6   RULE : *HighCohesion* {COH HIGH} ; 7 } ;

God Component :

God Component in SCA technology corresponds to a component that **encapsulates** a **multi-tude of services**. This component represents high responsibility enclosed by **many methods** with **many** different types of **parameters** to exchange. It may have a **high coupling** with the communicating services. Being at the component-level, *God Component* is at a higher level of abstraction than the *Multi Service*, which is at the service-level, and usually **aggregates** a **set of services** [Dudney *et al.* (2003)].

```
1 RULE_CARD : GodComponent {
2   RULE : GodComponent {INTER HighEncapsulatedService MultiMethod HighParameter};
3   RULE : HighEncapsulatedService {NOSE HIGH};
4   RULE : MultiMethod {NMD VERY_HIGH};
5   RULE : HighParameter {TNP VERY_HIGH};
6 };
```

Bloated Service :

Bloated Service is an antipattern related to service implementation where services in SOA become 'blobs' with **one large interface** and **lots of parameters**. *Bloated Service* performs **heterogeneous operations** with **low cohesion** among them. It results in a system with less maintainability, testability, and reusability within other business processes. It requires the consumers to be aware of many details (*i.e.*, parameters) to invoke or customize them [Modi (2006)].

```
1 RULE_CARD : BloatedService {
2   RULE : BloatedService {INTER SingleInterface MultiMethod HighParameter LowCohesion};
3   RULE : SingleInterface {NOI EQUAL 1};
4   RULE : MultiMethod {NMD VERY_HIGH};
5   RULE : HighParameter {TNP VERY_HIGH};
6   RULE : LowCohesion {COH LOW};
7 };
```

Stovepipe Service :

Stovepipe Service is an antipattern with **large number** of **private** or **protected** methods that primarily focus on performing infrastructure and **utility functions** (*i.e.*, logging, data validation, notifications, etc.) and few business processes (*i.e.*, data type conversion), rather than focusing on main operational goals (*i.e.*, very **few public methods**). This may result in services with **duplicated code**, longer development time, inconsistent functioning, and poor extensibility [Dudney *et al.* (2003)].

```
1 RULE_CARD : StovepipeService {
2   RULE : StovepipeService {INTER HighUtilMethod FewMethod DuplicatedCode};
3   RULE : HighUtilMethod {NUM VERY_HIGH};
4   RULE : FewMethod {NMD VERY_LOW};
5   RULE : DuplicatedCode {ANIM HIGH};
6 };
```

Ambiguous Name :

Ambiguous Name is an antipattern where the developers use the names of **interface elements** (*e.g.,* **port-types**, **operations**, and **messages**) that are **very short** or **long**, include too **general terms**, or even show the improper **use of verbs**, etc. *Ambiguous names* are not **semantically** and **syntactically** sound and impact the **discoverability** and the **reusability** of a Web service [Rodriguez *et al.* (2010a)].

```
1 RULE_CARD : AmbiguousName {
2   RULE : AmbiguousName {INTER GeneralTerm ShortORLongSignature VerbedMessage Multi-
    VerbedOperation};
3   RULE : ShortORLongSignature {UNION ShortSignature LongSignature};
4   RULE : LongSignature {ALS VERY_HIGH};
5   RULE : ShortSignature {ALS VERY_LOW};
6   RULE : GeneralTerm {RGTS HIGH};
7   RULE : VerbedMessage {NVMS > 0};
8   RULE : MultiVerbedOperation {NVOS > 1};
9 };
```

CRUDy Interface :

CRUDy Interface is an antipattern where the design encourages services the **RPC-like behavior** by creating **CRUD-type operations**, *e.g.*, **create_X()**, **read_Y()**, etc. Interfaces designed in that way might be **chatty** because multiple operations need to be invoked to achieve one goal. In general, **CRUD operations** should **not be exposed** via **interfaces** [Evdemon (2005)].

```
1 RULE_CARD : CRUDyInterface {
2   RULE : CRUDyInterface {INTER ChattyInterface HighCRUDOperation};
3   RULE : ChattyInterface {RULE_CARD : ChattyWebService};
4   RULE : HighCRUDOperation {NCO > 1};
5 };
```

Low Cohesive Operations in the Same PortType :

Low Cohesive Operations in the Same PortType is an antipattern where developers place **low cohesive operations** in a **single port-type**. From the Web services perspective, if the operations belonging to the same *prototype* do not provide a set of **semantically related** operations, the *prototype* becomes **less cohesive** [Rodriguez *et al.* (2010a)].

```
1 RULE_CARD : LowCohesiveOperations {
2   RULE : LowCohesiveOperations {INTER MultiOperation LowCohesivePortType};
4   RULE : MultiOperation {NOD HIGH};
5   RULE : LowCohesivePortType {ARIO LOW};
6 };
```

Maybe It's Not RPC :

Maybe It's Not RPC is an antipattern where the Web service mainly provides **CRUD operations** with a **large number of parameters**. This antipattern causes **poor system performance** because the clients **often wait** for the synchronous **responses** [Dudney *et al.* (2003)].

```
1 RULE_CARD : MaybeItsNotRPC {
2   RULE : MaybeItsNotRPC {INTER HighResponseTime HighCRUDOperation HighParameter};
3   RULE : HighResponseTime {RT HIGH};
4   RULE : HighCRUDOperation {NCO VERY_HIGH};
5   RULE : HighParameter {ANP HIGH};
6 };
```

Redundant PortTypes :

Redundant PortTypes is an antipattern where ***multiple port-types*** are ***duplicated*** with the ***similar set*** of ***operations***. Very often, such *port-types* deal with the ***same messages***. The *Redundant PortType* antipattern may ***negatively impact*** the ***ranking*** of the Web Services [Hess *et al.* (2004)].

---

1 RULE_CARD : *RedundantPortType* {

2   RULE : *RedundantPortType* {INTER *MultiPortType MultiOperations HighCohesivePortType*};

4   RULE : *MultiPortType* {NPT > 1};

5   RULE : *MultiOperations* {NOPT > 1};

6   RULE : *HighCohesivePortType* {ARIP VERY_HIGH};

7 };

Breaking Self-descriptiveness :

REST developers tend to ignore the ***standardised headers***, ***formats***, or ***protocols*** and use their own customised ones. This practice shatters the self-descriptiveness or containment of a message header. Breaking the self-descriptiveness also limits the ***reusability*** and ***adaptability*** of REST resources [Tilkov (2008)].

---

1 BREAKING-SELF-DESCRIPTIVENESS(*request-header*, *response-header*)

2   *std-request-headers*[] ← {"Content-Type", "Proxy-Authorization", "Host", ...}

3   *std-response-headers*[] ← {"Set-Cookie", "Last-Modified", "Location", ...}

4 **for each** $h_{req} \in$ *request-header*.getKeys() **and** $h_{res} \in$ *response-header*.getKeys()

5   **if**($h_{req} \notin$ *std-request-headers*[]) **or** ($h_{res} \notin$ *std-response-headers*[])

6   **print** "*Breaking Self-descriptiveness detected*"

7   **end if**

8 **end for**

Forgetting Hypermedia :

The ***lack*** of ***hypermedia***, *i.e.*, ***not linking resources***, hinders the state transition for `REST` applications. One possible indication of this antipattern is the ***absence*** of `URL` ***links*** in the ***resource representation***, which typically restricts clients to follow the links, *i.e.*, limits the dynamic communication between clients and servers [Tilkov (2008)].

---

1 Forget-Hyper-media(*response-header*, *response-body*, *http-method*)

2   *body-links*[] ← Extract-Entity-Links(*response-body*)

3   *header-link* ← *response-header*.getValue("Link")

4   **if**(*http-method* = GET **and** (*length*(*body-links*[]) = 0 **or** *header-link* = NIL)) **or**
    (*http-method* = POST **and** ("Location :" ∉ *response-header*.getKeys() **and**
    length(*body-links*[]) = 0))) **then**

5   **print** "*Forgetting Hypermedia detected*"

6   **end if**

Ignoring Caching :

REST clients and server-side developers tend to ***avoid*** the caching capability due to its complexity to implement. However, caching capability is one of the principle REST constraints. The developers ignore caching by setting ***Cache-Control : no-cache*** or ***no-store*** and by not providing an ***ETag*** in the ***response header*** [Tilkov (2008)].

---

1 Ignoring-Caching(*request-header*, *response-header*, *http-method*)

2   *client-caching* ← *request-header*.getValue("Cache-Control")

3   *server-caching* ← *response-header*.getValue("Cache-Control")

4   **if**((*http-method* = GET **and** ((*client-caching* = "no-cache" **or** "no-store") **or**
    (*server-caching* = "no-cache" **or** "no-store")) **and**
    "ETag" ∉ *response-header*.getKeys()))

5   **print** "*Ignoring Caching detected*"

6   **end if**

Ignoring MIME Types :

The server should represent **_resources_** in various formats, *e.g.*, **_xml, json, pdf_**, etc., which may allow clients, developed in diverse languages, a more flexible service consumption. However, the server side **_developers_** often intend to have a **_single representation_** of resources or rely on their **_own formats_**, which limits the resource (or service) **_accessibility_** and **_reusability_** [Tilkov (2008)].

1 Ignoring-MIME-Types(*request-header, response-header*)

2   *std-types*[] ← {"json", "xml", "yaml", "pdf", "jpeg", ...}

3   *client-format*[] ← *request-header*.getValue("Accept")

4   *server-format* ← *response-header*.getValue("Content-Type")

5   **if**(*server-format* ∉ *client-format*[] **and** *server-format* ∉ *std-types*[])

6   **print** "*Ignoring MIME Types detected*"

7   **end if**

Ignoring Status Code :

Despite of a rich set of defined **_application-level status codes_** suitable for various contexts, REST developers tend to **_avoid_** them, *i.e.*, **_rely_** only on **_common ones_**, namely 200, 404, and 500, or even use the **_wrong or no_** status **_codes_**. The correct use of status codes from the classes **_2xx, 3xx, 4xx, and 5xx_** helps clients and servers to communicate in a more semantic manner [Tilkov (2008)].

1 Ignoring-Status-Code(*response-body, response-header, http-method*)

2   *std-codes*[] ← {'GET-OK-200', 'POST-Created-201', 'DELETE-Unauthorized-401'}

3   *status-description* ← Extract-Status(*response-body*)

4   *status-code* ← Extract-Status-Code(*response-header*)

5   *response-status* ← Concat(*http-method, status-description, status-code*)

6   **if**(*response-status* ∉ *std-codes*[] **or** *status-code* = NIL)

7   **print** "*Ignoring Status Code detected*"

8   **end if**

Misusing Cookies :

Statelessness is another REST principle to adhere—**session state** in the server side is **disallowed** and any **cookies violate** RESTful-ness [Fielding (2000)]. Sending **keys** or **tokens** in the **Set-Cookie** or **Cookie** header field to server-side session is an example of misusing cookies, which concerns both **security** and **privacy** [Tilkov (2008)].

1 MISUSING-COOKIES(*request-header*, *response-header*)

2   *client-cookie* ← *request-header*.getValue("Cookie")

3   *server-cookie* ← *response-header*.getValue("Set-Cookie")

4   **if**((*client-cookie* ≠ NIL **and** CHECK-KEY(*client-cookie*) = TRUE) **or**
    (*server-cookie* ≠ NIL **and** CHECK-KEY(*server-cookie*) = TRUE))

5   **print** "*Misusing Cookies detected*"

6   **end if**

Tunnelling Through GET :

Being the most fundamental HTTP method in REST, the **GET** method **retrieves** a resource identified by a URI. However, very often the developers **rely only** on **GET** method to perform any kind of actions or operations including **creating**, **deleting**, or even for **updating** a resource. Nevertheless, HTTP GET is an inappropriate method for any actions other than **accessing** a **resource**, and does not match its **semantic purpose**, if improperly used [Tilkov (2008)].

1 TUNNELLING-THROUGH-GET(*request-uri*, *http-method*)

2   *keywords*[] ← {"method", "action", "operation", ...}

3   **if**(*http-method* = GET **and**
    (*keywords*[] ∈ *request-uri* **or** CHECK-VERB(*request-uri*) = TRUE))

4   **print** "*Tunnelling Through GET detected*"

5   **end if**

Tunnelling Through POST :

This anti-pattern is very similar to the previous one, except that in addition to the URI the **body** of the HTTP POST request may **embody operations** and **parameters** to apply on the resource. The **developers** tend to **depend** only on HTTP POST method for **sending any** types of **requests** to the server including **accessing**, **updating**, or **deleting** a resource. In general, the proper use of HTTP POST is to **create** a server-side resource [Tilkov (2008)].

---

1 TUNNELLING-THROUGH-POST(*request-uri, request-body, http-method*)

2   *keywords*[] ← {"method", "action", "operation", ...}

3   **if**(*http-method* = POST **and**

    ((*keywords*[] ∈ *request-uri* **or** CHECK-VERB(*request-uri*) = TRUE) **or**

    (*keywords*[] ∈ *request-body* **or** CHECK-VERB(*request-body*) = TRUE)))

4   **print** "*Tunnelling Through POST detected*"

5   **end if**

Content Negotiation :

This pattern supports **alternative resource representations**, *e.g.*, in **json**, **xml**, **pdf**, etc. so that the service consuming becomes more flexible with **high reusability**. Servers can provide resources in **any standard format** requested by the clients. This pattern is applied via standard HTTP **media types** and adhere to *service loose coupling* principle. If not applied at all, this turns into **Ignoring MIME Types** antipattern [Erl *et al.* (2012)].

---

1 CONTENT-NEGOTIATION(*request-header*)

2   *std-types*[] ← {"json", "xml", "yaml", "pdf", "jpeg", ...}

3   *client-format*[] ← *request-header*.getValue("Accept")

4   *server-format* ← *response-header*.getValue("Content-Type")

5   **if**(*server-format* ∈ *client-format*[] **and** *server-format* ∈ *std-types*[])

6   **print** "*Content Negotiation detected*"

7   **end if**

End-point Redirection :
The **redirection** feature over the Web is supported by this pattern, which also plays a role as the means of **service composition**. To redirect clients, servers send a new **location** to follow with one of the **status code** among **301**, **302**, **307**, or **308**. The main benefit of this pattern is—an **alternative service** remains **active** even if the requested service end-point is not sound [Erl *et al.* (2012)].
1 END-POINT-REDIRECTION(*response-header*)   2   *new-location* ← *response-header*.getValue("Location")   3   *status-code* ← EXTRACT-STATUS-CODE(*response-header*)   4   **if**(*new-location* ≠ NIL **and** (*status-code* = 301 **or** 302 **or** 307 **or** 308))   5   **print** "*End-point Redirection detected*"   6   **end if**

Entity Linking :
This pattern enables **runtime communication** via **links** provided by the server in the **response body** or via **Location :** in the **response header**. By using **hyper-links**, the servers and clients can be **loosely coupled**, and the clients can **automatically find** the **related entities** at **runtime**. If not properly applied, this pattern turns into **Forgetting Hypermedia** antipattern [Erl *et al.* (2012)].

1 ENTITY-LINKING(*response-header*, *response-body*, *http-method*)
 2   *body-links*[] ← EXTRACT-ENTITY-LINKS(*response-body*)
 3   *header-link* ← *response-header*.getValue("Link")
 4   **if**(*http-method* = GET **and** (*length*(*body-links*[]) ≥ 1 **or** *header-link* ≠ NIL)) **or**
     (*http-method* = POST **and** ("Location :" ∈ *response-header*.getKeys() **or**
     length(*body-links*[]) ≥ 1))) **then**
 5   **print** "*Entity Linking detected*"
 6   **end if**

Entity Endpoint :

Services with single ***end-points*** are too coarse-grained. Usually, a client requires at least ***two identifiers*** : (1) a ***global*** for the ***service*** itself and (2) a ***local*** for the ***resource or entity*** managed by the service. By applying this pattern, *i.e.*, using ***multiple end-points***, each ***entity (or resource)*** of the incorporating service can be ***uniquely identified*** and ***addressed*** globally [Pautasso (2009)].

---

1 ENTITY-END-POINT(*interface*)

2   *method-endpoints<http-method, end-points[]>* ← GET-UNIQUE-ENDPOINTS(*interface*)

3   **for each** *http-method* ∈ *method-endpoints*

4   *unique-end-point* ← COUNT(GET-UNIQUE-PATHS(*end-points[]*))

5   **if**(*unique-end-point* = COUNT(GET-PATHS(*end-points[]*)))

6   **print** "*Entity Endpoint detected*"

7   **end if**

8   **end for**

Response Caching :

Response caching is a good practice to ***avoid sending duplicate requests*** and ***responses*** by caching all response messages in the ***local*** client machine. In opposed to ***Ignoring Caching*** antipattern, the ***Cache-Control :*** is set to any value other than ***no-cache*** and ***no-store***, or an ***ETag*** is used along with the ***status code 304*** [Erl *et al.* (2012)].

---

1 RESPONSE-CACHING(*request-header, response-header, status-code*)

2   *client-caching* ← *request-header*.getValue("Cache-Control")

3   *server-caching* ← *response-header*.getValue("Cache-Control")

4   **if**((*client-caching* **or** *server-caching* ≠ "no-cache" **and** "no-store") **and**
    ("ETag" ∈ (*request-header*.getKeys() **or** *response-header*.getKeys())) **and**
    status-code = 304)

5   **print** "*Response Caching detected*"

6   **end if**

Contextualised vs. Contextless Resource Names :

Description : URIs should be *contextual, i.e.*, nodes in URIs should belong to semantically-related context. Thus, the *Contextless Resource Names* antipattern appears when URIs are composed of nodes that do not belong to the same semantic context.

Example : `https://www.example.com/newspapers/players?id=123` is a *Contextless Resource Names* antipattern because 'newspapers' and 'players' do not belong to same semantic context. `https://www.example.com/soccer/team/players?id=123` is a *Contextual Resource Names* pattern because 'soccer', 'team', and 'players' belong to same semantic context.

Consequences : *Contextless Resource Names* do not provide a clear context for a request, which may mislead the APIs clients by decreasing the understandability of the APIs [Fredrich (2012)].

1 CONTEXTLESS-RESOURCE-NAMES(*Request-URI*)

2   *URINodes* ← EXTRACT-URI-NODES(*Request-URI*)

3   **for each** *index* = 1 **to** LENGTH(*URINodes*)

4   *Set1* ← CAPTURE-CONTEXT-BY-SYNSETS($URINodes_{index}$)

5   *Set2* ← CAPTURE-CONTEXT-BY-SYNSETS($URINodes_{index+1}$)

6   **if** *Set1* ∩ *Set2* = ∅

7   **print** "*Contextless Resource Names detected*"

8   **break**

9   **end if**

10   **end for**

Hierarchical vs. Non-hierarchical Nodes :

Description : Each node forming a URI should be hierarchically related to its neighbor nodes. In contrast, *Non-hierarchical Nodes* is an antipattern that appears when at least one node in a URI is not hierarchically related to its neighbor nodes.

Example : `https://www.example.com/professors/faculty/university/` is a *Non-hierarchical Nodes* antipattern since 'professors', 'faculty', and 'university' are not in a hierarchical relationship. `https://www.example.com/university/faculty/professors/` is a *Hierarchical Nodes* pattern since 'university', 'faculty', and 'professors' are in a hierarchical relationship.

Consequences : Using non-hierarchical names may confuse users on the real purpose of the API and hinders their understandability and, therefore, the API's usability [Fredrich (2012)].

1 NON-HIERARCHICAL-NODES(*Request-URI*)

2   *URINodes*[] ← EXTRACT-URI-NODES(*Request-URI*)

3   **for each** *index* = 1 **to** LENGTH(URINodes)

4   **if**((IS-HIERARCHICAL-RELATION($URINodes_i, URINodes_{i+1}$) = **false**) **or**

5   IS-SPECIALISATION-RELATION($URINodes_i, URINodes_{i+1}$))

6   **print** "*Non-hierarchical Nodes detected*"

7   **end if**

Tidy vs. Amorphous URIs :

Description : REST resource URIs should be tidy and easy to read. A *Tidy URI* is a URI with appropriate lower-case resource naming, no extensions, underscores, or trailing slashes. *Amorphous URI* antipatterns appear when URIs contain symbols or capital letters that make them difficult to read and use. As opposed to good practices [Massé (2012)], a URI is amorphous if it contains : (1) upper-case letter (except for Camel Cases [Microsoft :MSDN (1992)]), (2) file extensions, (3) underscores, and, (4) a final trailing-slash.

Example : `https://www.example.com/NEW_Customer/_photo01.jpg/` is a *Amorphous URI* antipattern since it includes a file extension, upper-case resource names, and underscores. `https://www.example.com/customers/1234` is a *Tidy URI* pattern since it only contains lower-case resource naming, without extensions, underscores, or trailing slashes.

Consequences : (1) Upper/lower-case names may refer to different resources, RFC 3986 [Berners-Lee *et al.* (2005)]. (2) File extensions in URIs violate RFC 3986 and affect service evolution. (3) Underscores are hidden when highlighting URIs, decreasing readability. (4) Trailing-slash mislead users to provide more resources.

1 AMORPHOUS-URI(*Request-URI*)

2   *URINodes*[] ← EXTRACT-URI-NODES(*Request-URI*)

3   **for each** *index* = 1 **to** LENGTH(URINodes)

4   **if**(*URINodes$_i$*.contains("_") **or** *URINodes$_i$*.contains("/") **or**

5   *URINodes$_i$*.contains(FIND-FILE-EXTENSIONS(*Request-URI*)) **or**

6   *URINodes$_i$*.contains(FIND-UPPERCASE-RESOURCES(*Request-URI*)))

7   **print** "*Amorphous URI detected*"

8   **end if**

Verbless vs. CRUDy URIs :

Description : Appropriate HTTP methods, *e.g.*, GET, POST, PUT, or DELETE, should be used in *Verbless URIs* instead of using CRUDy terms (*e.g.*, create, read, update, delete, or their synonyms) [Fredrich (2012)]. The use of such terms as resource names or requested actions is highly discouraged [Massé (2012)].

Example : `POST https://www.example.com/update/players/age?id=123` is a *CRUDy URIs* antipattern since it contains a CRUDy term 'update' while updating the user's profile color relying on an HTTP POST method. `POST https://www.example.com/players/age?id=123` is a *Verbless URIs* pattern since is an HTTP POST request without any verb.

Consequences : Using CRUDy terms in URIs can be confusing for API clients, *i.e.*, in the best cases they overload the HTTP methods and in the worst cases they go against HTTP methods. CRUDy terms in a URI confuse and prohibit users to use proper HTTP methods in a certain context and may introduce another REST antipattern, *Tunnelling through GET/POST* [Tilkov (2008)].

1 CRUDY-URI(*Request-URI*)

2   *CRUDyWords*[] ← {"create", "read", "delete", "destroy", "update", "copy", "move",...}

3   *URINodes*[] ← EXTRACT-URI-NODES(*Request-URI*)

4   **for each** $i = 1$ **to** LENGTH(*URINodes*)

5   **if**((SYNONYMS(URINodes$_i$) **or** ANTONYMS(URINodes$_i$)) ∈ *CRUDyWords*)

6   **print** "*CRUDy URI detected*"

7   **end if**

Singularised vs. Pluralised Nodes :

Description : URIs should use singular/plural nouns consistently for resources naming across the API. When clients send PUT/DELETE requests, the last node of the request URI should be singular. In contrast, for POST requests, the last node should be plural. Therefore, the *Pluralised Nodes* antipattern appears when plural names are used for PUT/DELETE requests or singular names are used for POST requests. However, GET requests are not affected by this antipattern [Fredrich (2012)].

Example : The first example URI is a POST method that does not use a pluralised resource, thus leading to *Pluralised Nodes* antipattern (Example 1). On the other hand, for the *Singularised Nodes* pattern, the DELETE request acts on a single resource for deleting it (Example 2).

(1) DELETE `www.example.com/team/players` or POST `www.example.com/team/player`

(2) DELETE `www.example.com/team/player` or POST `www.example.com/team/players`

Consequences : If a plural node for PUT (or DELETE) request is used at the end of a URI, the API clients cannot create (or delete) a collection of resources, which may result in, for example, a `403 Forbidden` server response. In addition, even if the resources can be filtered through query-like parameters, it confuse the user if one or multiple resources are being accessed/deleted [Fredrich (2012)].

1 Pluralised-URI-Nodes(*Request-URI*, *http-method*)

2   *lastNode* ← Get-Last-Node(*Request-URI*)

3   *secondLastNode* ← Get-Second-Last-Node(*Request-URI*)

4   **if**(((*http-method* = PUT **or** DELETE) **and** is-Plural-Node(*lastNode*) = **true**) **or** ((*http-method* = POST) **and** (is-Plural-Node(*secondLastNode*) = **false**)))

5   **print** "*Pluralised URI Nodes detected*"

6   **end if**

Transformation of REST Heuristics to Rule Cards

---

1 : IGNORING-CACHING(*request-header*, *response-header*, *http-method*)

2 :   *client-caching* ← *request-header*.getValue("Cache-Control")

3 :   *server-caching* ← *response-header*.getValue("Cache-Control")

4 :   **if**((*http-method* = GET **and** ((*client-caching* = "no-cache" **or** "no-store")

5 :   **or** (*server-caching* = "no-cache" **or** "no-store")) **and**

6 :   "ETag" $\notin$ *response-header*.getKeys())) **then**

7 :   **print** "*Ignoring Caching detected*"

8 :   **end if**

1 RULE_CARD : IgnoringCaching {

2   RULE : IgnoringCaching { INTER HeaderCaching NoEntityTag } ;

3   RULE : HeaderCaching { INTER HttpMethodGet NoClientOrServerCaching } ;

7   RULE : HttpMethodGet { HM = 'GET' } ;

8   RULE : NoClientOrServerCaching { UNION NoClientCaching NoSeverCaching } ;

9   RULE : NoClientCaching { CCV = 'no-cache' | 'no-store' } ;

9   RULE : NoServerCaching { SCV = 'no-cache' | 'no-store' } ;

11   RULE : NoEntityTag { 'ETag' $\notin$ ResponseHeader } ;

12 } ;

---

1 : IGNORING-MIME-TYPES(*request-header*, *response-header*)

2 :   *std-types*[] ← {"json", "xml", "yaml", "pdf", "jpeg", ...}

3 :   *client-format*[] ← *request-header*.getValue("Accept")

4 :   *server-format* ← *response-header*.getValue("Content-Type")

5 :   **if**(*server-format* $\notin$ *client-format*[] **and** *server-format* $\notin$ *std-types*[]) **then**

6 :   **print** "*Ignoring MIME Types detected*"

7 :   **end if**

1 RULE_CARD : IgnoringMIMETypes {

2   RULE : IgnoringMIMETypes { INTER UnavailableClientFormat NonStandardServerFormat } ;

3   RULE : UnavailableClientFormat { *ServerFormat* $\notin$ *ClientFormat* } ;

4   RULE : NonStandardServerFormat { *ServerFormat* $\notin$ *StandardTypes* } ;

5   RULE : ServerFormat { ResponseHeader$Content-Type } ;

6   RULE : ClientFormat { RequestHeader$Accept } ;

7   RULE : StandardTypes { RRF = 'json' | 'xml' | 'yaml' | 'pdf' | 'jpeg' | ...} ;

8 } ;

---

1 : Tunnelling-Through-GET(*request-uri*, *http-method*)
2 :   *keywords*[] ← {"method", "action", "operation", ...}
3 :   **if**(*http-method* = GET **and**
4 :   (*keywords*[] ∈ *request-uri* **or** Check-Verb(*request-uri*) = TRUE)) **then**
5 :   **print**"*Tunnelling Everything Through GET detected*"
6 :   **end if**

1 RULE_CARD : TunnellingThroughGET {
2   RULE : TunnellingThroughGET { INTER HttpMethodGet VerbedRequestURI } ;
3   RULE : HttpMethodGet { HM = 'GET' } ;
4   RULE : VerbedRequestURI { UNION KeywordBasedURI URIwithVerbs } ;
5   RULE : KeywordBasedURI { Keywords ∈ *request-uri* } ;
6   RULE : URIwithVerbs { VRU = TRUE } ;
7   RULE : Keywords { AK = 'method' | 'action' | 'operation' | ...} ;
8 } ;

---

1 : Misusing-Cookies(*request-header*, *response-header*)
2 :   *client-cookie* ← *request-header*.getValue("Cookie")
3 :   *server-cookie* ← *response-header*.getValue("Set-Cookie")
4 :   **if**((*client-cookie* ≠ NIL **and** Check-Key(*client-cookie*) = TRUE) **or**
5 :   (*server-cookie* ≠ NIL **and** Check-Key(*server-cookie*) = TRUE)) **then**
6 :   **print** "*Misusing Cookies detected*"
7 :   **end if**

1 RULE_CARD : MisusingCookies {
2   RULE : MisusingCookies { UNION ClientCookie ServerCookie } ;
3   RULE : ClientCookie { INTER ClientCookieValue AuthenticationCookie } ;
4   RULE : ServerCookie { INTER ServerCookieValue AuthenticationCookie } ;
5   RULE : ClientCookieValue { CC ≠ NULL } ;
6   RULE : ServerCookieValue { SC ≠ NULL } ;
7   RULE : AuthenticationCookie { AC = TRUE } ;
8 } ;

---

1 : Tunnelling-Through-POST(*request-uri*, *request-body*, *http-method*)

2 :   *keywords*[] ← {"method", "action", "operation", ...}

3 :   **if**(*http-method* = POST **and**

4 :   ((*keywords*[] ∈ *request-uri* **or** Check-Verb(*request-uri*) = TRUE) **or**

5 :   (*keywords*[] ∈ *request-body* **or** Check-Verb(*request-body*) = TRUE))) **then**

6 :   **print** "*Tunnelling Everything Through POST detected*"

7 :   **end if**

1 RULE_CARD : TunnellingThroughPOST {

2   RULE : TunnellingThroughPOST { INTER HttpMethodPost VerbedRequestURI } ;

3   RULE : HttpMethodPost { HM = 'POST' } ;

4   RULE : VerbedRequestURI { UNION URIWithVerbs BodyWithVerbs } ;

4   RULE : URIWithVerbs { UNION KeywordBasedURI URIwithVerbs } ;

5   RULE : KeywordBasedURI { Keywords ∈ *request-uri* } ;

6   RULE : URIwithVerbs { VRU = TRUE } ;

4   RULE : BodyWithVerbs { UNION KeywordBasedBody VerbsInBody } ;

5   RULE : KeywordBasedBody { Keywords ∈ *request-body* } ;

6   RULE : VerbsInBody { VRB = TRUE } ;

7   RULE : Keywords { AK = 'method' | 'action' | 'operation' | ...} ;

8 } ;

Materials Related to the Generation of Detection Algorithms

The class template for generating the detection algorithms of service antipatterns.

```
[comment encoding = UTF-8 /]
[module generateJava('http://rulecards/1.0')]

[template public generateJava(ruleCard : RuleCard)]
[comment @main/]
[file (ruleCard.name.concat('.java'), false, 'UTF-8')]

package com.sodop.patterns;

import com.sofa.metric.Metric;
import com.sofa.motifs.AMotif;
import com.sofa.rulecard.comparators.Comparator;
import com.sofa.rulecard.numoperators.NumOperator;
import com.sofa.rulecard.setoperators.SetOperator;
import com.sofa.rulecard.values.OrdinalValue;
import com.sofa.rulecard.smells.*;

public class [ruleCard.name/] extends AMotif {

        public [ruleCard.name/]() {

        [for (smell : Smell | ruleCard.children)]
        [if (smell.oclIsTypeOf(SmellComposite))]
                [generateSmell(smell, i)/]
        [/if]
        [/for]
        }
}

[/file]
[/template]

[template public generateSmell(smell : SmellComposite, index : Integer)]
[for (child : Smell | smell.children)]
        // SMELL [i/]
        [generateSmell(child, i)/]

[/for]
        // ROOT SMELL
        this.rootSmell = new SmellComposite(SetOperator.[smell.setOperator/]);
[for (i : Integer | Sequence{1..smell.children->size()})]
        this.rootSmell.addChildSmell(smell[i/]);
[/for]
[/template]

[template public generateSmell(smell : SmellOrdinalValue, index : Integer)]
MetricValue metricValueSmell[index/] = [generateMetricValue(smell.leftMetricValue)/];
Smell smell[index/] = new SmellOrdinalValue(metricValueSmell[index/], Comparator.[smell.comparator/],
OrdinalValue.[smell.ordinalValue/]);
[/template]

[template public generateSmell(smell : SmellNumericValue, index : Integer)]
MetricValue metricValueSmell[index/] = [generateMetricValue(smell.leftMetricValue)/];
Smell smell[index/] = new SmellNumericValue(metricValueSmell[index/], Comparator.[smell.comparator/],
new Double([smell.numValue/]));
[/template]

[template public generateSmell(smell : SmellMetricValue, index : Integer)]
MetricValue metricValue1Smell[index/] = [generateMetricValue(smell.leftMetricValue)/];
MetricValue metricValue2Smell[index/] = [generateMetricValue(smell.rightMetricValue)/];
Smell smell[index/] = new SmellMetricValue(metricValue1Smell[index/], Comparator.[smell.comparator/],
metricValue2Smell[index/]);
[/template]

[template public generateMetricValue(metricComposite : MetricComposite)]
new MetricComposite([generateMetricValue(metricComposite.leftChild)/],
NumOperator.[metricComposite.numOperator/], [generateMetricValue(metricComposite.rightChild)/])
[/template]

[template public generateMetricValue(metricLeaf : MetricLeaf)]
new MetricLeaf(Metric.[metricLeaf.metric/])
[/template]

[template public generateMetricValue(metricValue : MetricValue)]
[/template]

[template public generateSmell(smell : Smell, index : Integer)]
[/template]

[template public generateSmell(smell : SmellLeaf, index : Integer)]
[/template]
```

The concrete syntax for the rule cards of service antipatterns.

```
SYNTAXDEF rc
FOR <http://rulecards/1.0>
START RuleCard

OPTIONS
{
        generateCodeFromGeneratorModel = "true";
}

TOKENS {
        DEFINE COMMENT $'//'(~('\n'|'\r'|'\uffff'))*$;
        DEFINE FLOAT $('-')?(('1'..'9') ('0'..'9')* | '0') '.' ('0'..'9')+ $;
        DEFINE ORDINAL_VALUE $('VERY_HIGH')|('HIGH')|('MEDIUM')|('LOW')|('VERY_LOW')$;
        DEFINE SET_OPERATOR $ ('INTER') | ('UNION') | ('DIFF') | ('INCL') | ('NEG')$;
}


TOKENSTYLES {
        "TEXT" COLOR #000000;
        "COMMENT" COLOR #00bb00, ITALIC;
        "ORDINAL_VALUE" COLOR #FF6600, BOLD;
        "FLOAT" COLOR #B20000, BOLD;
        "SET_OPERATOR" COLOR #CC0099, BOLD;
}


RULES {

        RuleCard ::= "RULE_CARD:" name[] "{"
                children+
        "};";

        SmellComposite ::= "RULE:" name[] "{"
                setOperator[SET_OPERATOR] children[]+
        "};";

        SmellNumericValue ::= "RULE:" name[] "{"
                leftMetricValue comparator[LOWER:"<",LOWER_EQUAL:"<=",EQUAL:"=",GREATER_EQUAL:">=",GREATER:">"]
                numValue[FLOAT]
        "};";

        SmellOrdinalValue ::= "RULE:" name[] "{"
                leftMetricValue comparator[LOWER:"<",LOWER_EQUAL:"<=",EQUAL:"=",GREATER_EQUAL:">=",GREATER:">"]
                ordinalValue[ORDINAL_VALUE]
        "};";

        SmellMetricValue ::= "RULE:" name[] "{"
                leftMetricValue comparator[LOWER:"<",LOWER_EQUAL:"<=",EQUAL:"=",GREATER_EQUAL:">=",GREATER:">"]
                rightMetricValue
        "};";

        MetricLeaf ::= metric[];

        MetricComposite ::= leftChild:MetricLeaf numOperator[ADDITION:"+",SUBSTRACTION:"-",MULTIPLICATION:"*",DIVISION:"/"]
                rightChild:MetricLeaf;
}
```
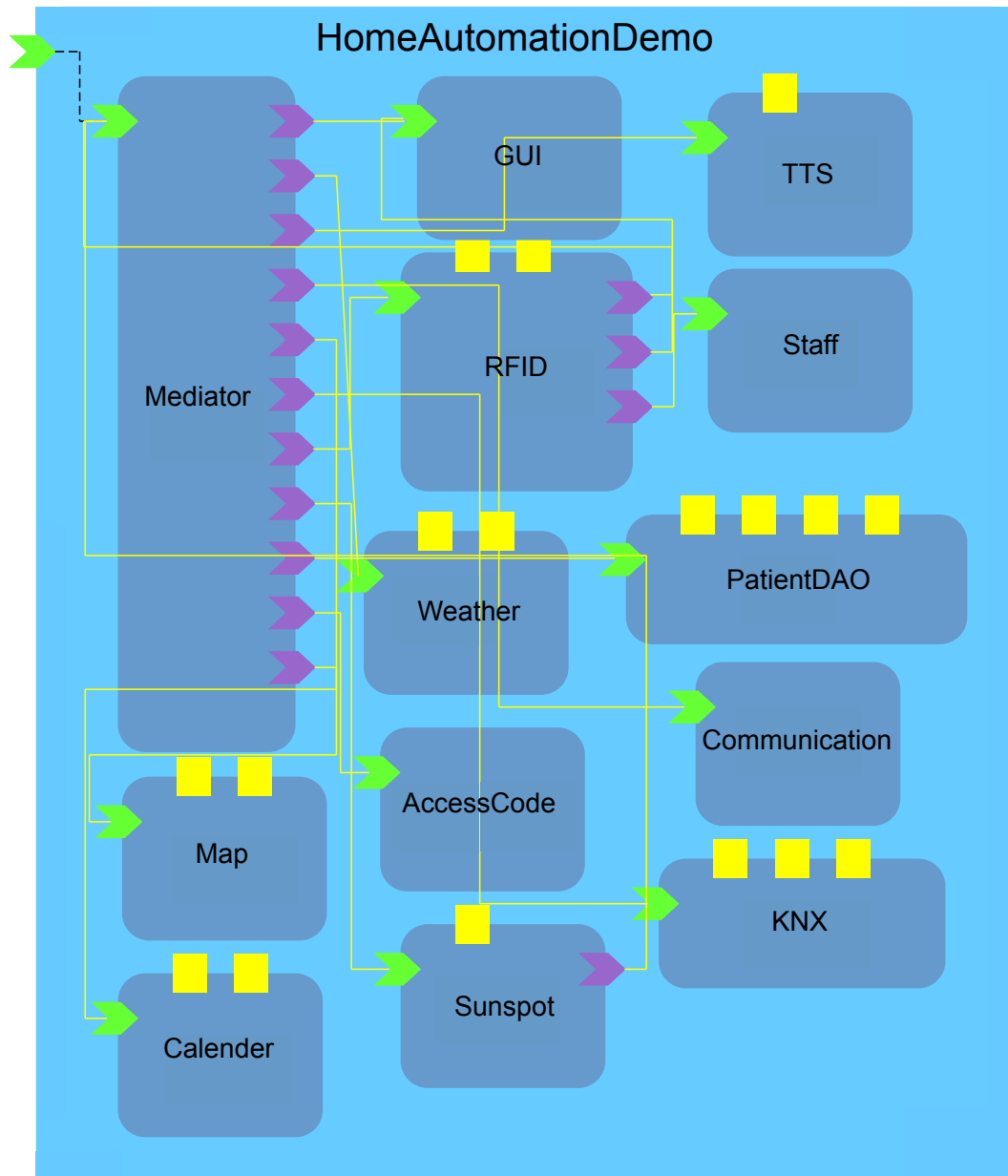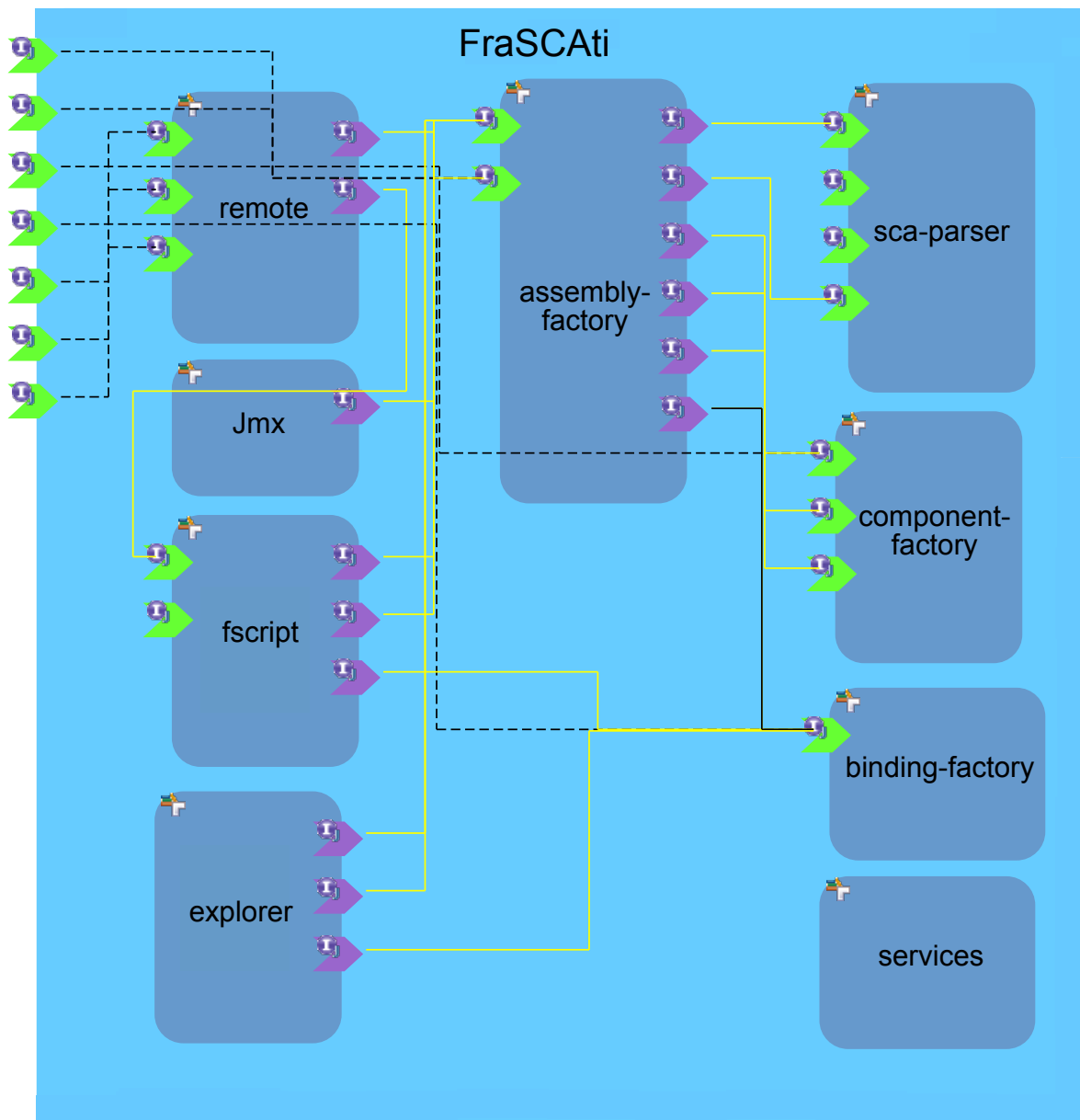
The architectural design of the *Home-Automation* system with 13 components.

The architectural design of the *FraSCAti* system with 91 components taken from FraSCAti Web site http://frascati.ow2.org/doc/1.4/ch12s04.html. Each component represents an independent SCA composite.

The List of SOAP Web services

The list of 13 weather-related Web services :

1. http://soap.webservice-energy.org/empclimat__ws/service?wsdl
2. http://toolbox.webservice-energy.org/TOOLBOX/WSDL/AIP3__PV__Impact/AIP3__PV__Impact.wsdl
3. http://soap.webservice-energy.org/hydro1k__ws/service?wsdl
4. http://websky.kma.go.kr/services/SurfaceService?wsdl
5. http://climhy.lternet.edu/wambam/soap__server.pl?wsdl
6. http://graphical.weather.gov/xml/DWMLgen/wsdl/ndfdXML.wsdl
7. http://soap.webservice-energy.org/ncepforecast__ws/service?wsdl
8. http://www.weather.gov/forecasts/xml/DWMLgen/wsdl/ndfdXML.wsdl
9. http://www.restfulwebservices.net/wcf/WeatherForecastService.svc?wsdl
10. http://developer.ebay.com/webservices/finding/latest/FindingService.wsdl
11. http://soap.webservice-energy.org/shadow__ws/service?wsdl
12. http://soap.webservice-energy.org/solemi__ws/service?wsdl
13. http://soap.webservice-energy.org/srtm__ws/service?wsdl

The list of 110 weather-related Web services :

1. http://abn-markets.customers.solvians.com/services/finance-broker/?wsdl
2. http://www.banguat.gob.gt/variables/ws/TipoCambio.asmx?wsdl
3. http://webservices.lb.lt/BLiquidity/BLiquidity.asmx?wsdl
4. http://www.cbr.ru/DailyInfoWebServ/DailyInfo.asmx?wsdl
5. http://www.mosaicsoftware.co.uk/services/bankcheck/bankcheck.asmx?wsdl
6. http://indicadoreseconomicos.bccr.fi.cr/indicadoreseconomicos/WebServices/wsIndicadoresEconomicos.asmx?wsdl
7. http://www.bgcantorondemand.com/BGCantorUSTreasuries.asmx?wsdl
8. http://ws.cdyne.com/delayedstockquote/delayedstockquote.asmx?wsdl
9. http://api.cba.am/exchangerates.asmx?wsdl
10. http://fx.cloanto.com/webservices/CloantoCurrencyServer.asmx?wsdl
11. http://www.spk.gov.tr/webservices/MutualFundsPortfolioValues/MFundsService.asmx?wsdl
12. http://www.spk.gov.tr/webservices/ProhibitedInvestors/ProhibitedInvestorsService.asmx?wsdl
13. http://ws.strikeiron.com/CorteraCreditPulse?wsdl
14. http://www.programi.net/fiskalizacija/FiskalizacijaService.wsdl

15. http://www.currencyserver.de/webservice/currencyserverwebservice.asmx?wsdl

16. http://www.dfm.ae/ws/MSData.asmx?WSDL

17. http://www2.dfm.ae/ws/TickerData.asmx?WSDL

18. http://ws.strikeiron.com/DnBQuickCheck?WSDL

19. http://ebswebtest.iab.gov.tr/EBS/VYKWS?wsdl

20. http://ws.eoddata.com/data.asmx?wsdl

21. http://mifiddatabase.cesr.eu/ws/MifId.asmx?WSDL

22. http://www.findata.co.nz/Populate.asmx?WSDL

23. http://api.efxnow.com/DEMOWebServices2.8/Service.asmx?WSDL

24. http://www.gama-system.com/webservices/exchangerates.asmx?WSDL

25. http://www.gama-system.com/webservices/stockquotes.asmx?wsdl

26. http://www.thomas-bayer.com/axis2/services/BLZService?wsdl

27. http://devweb.grants.gov/techlib/ApplicantIntegrationServices-V1.0.wsdl

28. http://www.ibanbic.be/IBANBIC.asmx?WSDL

29. http://fr.iban-bic.com/soap_noarrays/index.php?wsdl

30. http://inflationinrussia.com/DesktopModules/WebServices.asmx?WSDL

31. http://www.infovalutar.ro/curs.asmx?wsdl

32. http://invoiceclarity.com/Api/invoiceclarity.asmx?WSDL

33. http://www.iaa.gov.il/Rashat/PublicWS/IAAUtilities.asmx?WSDL

34. http://62.219.95.10/TaarifWebService/TaarifCustoms.asmx?WSDL

35. http://www.mnb.hu/arfolyamok.asmx?WSDL

36. http://www.mondor.org/ces/rates.asmx?WSDL

37. http://www.msm.gov.om/ws/MarketSummaryData.asmx?WSDL

38. http://services.nexus6studio.com/StockQuoteService.asmx?WSDL

39. http://ws.homewarranty.nsw.gov.au/HWPremiumCalcService.asmx?WSDL

40. http://www.progetica.it/GetImmagesWS.asmx?WSDL

41. http://rateticker.progressive.com/SuperCREWS.asmx?WSDL

42. http://services.prosper.com/ProsperAPI/ProsperAPI.asmx?WSDL

43. http://www.quentinsagerconsulting.com/wsdl/aba.wsdl

44. http://www.quentinsagerconsulting.com/wsdl/iban.wsdl

45. http://www.restfulwebservices.net/wcf/StockQuoteService.svc?wsdl

46. http://ws2.serviceobjects.net/sq/FastQuote.asmx?WSDL

47. http://ws.strikeiron.com/ForeignExchangeRate2?WSDL

48. http://www.superfundlookup.gov.au/xmlsearch/SflXmlSearch.asmx?WSDL

49. http://www.takasbank.com.tr/tr/Documents/fplWS.wsdl

50. http://services.taxdataservice.com/TaxEconomyService.svc?WSDL

51. http://service.taxdatasystems.net/TdsBasic.svc?WSDL

52. http://www.verifilter.com/API/VerifilterSOAP.asmx?wsdl

53. http://ec.europa.eu/taxation_customs/vies/checkVatService.wsdl

54. http://api.virwox.com/api/basic.wsdl

55. http://www.xignite.com/xBATSLastSale.asmx?WSDL

56. http://www.xignite.com/xcalendar.asmx?WSDL

57. http://www.xignite.com/xglobalhistorical.asmx?WSDL

58. http://www.xignite.com/xIndexComponents.asmx?WSDL

59. http://www.xignite.com/xRealTime.asmx?WSDL

60. http://www.xignite.com/xanalysts.asmx?WSDL

61. http://bonds.xignite.com/xBonds.asmx?WSDL

62. http://bondsrealtime.xignite.com/xBondsRealTime.asmx?WSDL

63. http://www.xignite.com/xChart.asmx?WSDL

64. http://www.xignite.com/xCompensation.asmx?WSDL

65. http://xignite.com/xCorporateActions.asmx?WSDL

66. http://www.xignite.com/xCurrencies.asmx?WSDL

67. http://xignite.com/xEarningsCalendar.asmx?WSDL

68. http://www.xignite.com/xEdgar.asmx?WSDL

69. http://www.xignite.com/xEmerging.asmx?WSDL

70. http://www.xignite.com/xEnergy.asmx?WSDL

71. http://www.xignite.com/xestimates.asmx?WSDL

72. http://www.xignite.com/xExchanges.asmx?WSDL

73. http://www.xignite.com/xfinancials.asmx?WSDL

74. http://www.xignite.com/xfundamentals.asmx?WSDL

75. http://www.xignite.com/xfunddata.asmx?WSDL

76. http://www.xignite.com/xFundHoldings.asmx?WSDL

77. http://www.xignite.com/xFunds.asmx?WSDL

78. http://www.xignite.com/xFutures.asmx?WSDL

79. http://globalbondmaster.xignite.com/xGlobalBondMaster.asmx?WSDL

80. http://xignite.com/xGlobalFundamentals.asmx?WSDL

81. http://xignite.com/xGlobalRealTime.asmx?WSDL

82. http://www.xignite.com/xHistorical.asmx?WSDL

83. http://www.xignite.com/xHoldings.asmx?WSDL

84. http://www.xignite.com/xHousing.asmx?WSDL

85. http://www.xignite.com/xIndices.asmx?WSDL

86. http://www.xignite.com/xInsider.asmx?WSDL

87. http://www.xignite.com/xinterbanks.asmx?WSDL

88. http://www.xignite.com/xLogos.asmx?WSDL

89. http://www.xignite.com/xMaster.asmx?WSDL

90. http://www.xignite.com/xmetals.asmx?WSDL

91. http://www.xignite.com/xmoneymarkets.asmx?WSDL

92. http://www.xignite.com/xNASDAQLastSale.asmx?WSDL

93. http://www.xignite.com/xNews.asmx?WSDL

94. http://www.xignite.com/xOFAC.asmx?WSDL

95. http://www.xignite.com/xoptions.asmx?WSDL

96. http://www.xignite.com/xOutlook.asmx?WSDL

97. http://www.xignite.com/xQuotes.asmx?WSDL

98. http://www.xignite.com/xRates.asmx?WSDL

99. http://xignite.com/xRealTimeOptions.asmx?WSDL

100. http://www.xignite.com/xReleases.asmx?WSDL

101. http://xignite.com/xScreener.asmx?WSDL

102. http://www.xignite.com/xSecurity.asmx?WSDL

103. http://www.xignite.com/xStatistics.asmx?WSDL

104. http://xignite.com/xTranscripts.asmx?WSDL

105. http://www.xignite.com/xVWAP.asmx?WSDL

106. http://xignite.com/xWatchLists.asmx?WSDL

107. http://xurrency.com/api.wsdl

108. http://ws.xwebservices.com/XWebACHDirectory/V1/XWebACHDirectory.wsdl

109. http://budgetassistant.axionweb.be/service.asmx?WSDL

110. http://developer.ebay.com/webservices/finding/latest/FindingService.wsdl