Are RESTful APIs Well-designed? Detection of their Linguistic (Anti)Patterns

Francis Palma[1,2], Javier Gonzalez-Huerta[1], Naouel Moha[1], Yann-Gaël Guéhéneuc[2], and Guy Tremblay[1]

[1] Département d'informatique, Université du Québec à Montréal, Canada
{moha.naouel,gonzalez_huerta.javier,tremblay.guy}@uqam.ca
[2] Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
{francis.palma,yann-gael.gueheneuc}@polymtl.ca

Abstract. Identifier lexicon has a direct impact on software understandability and reusability and, thus, on the quality of the final software product. Understandability and reusability are two important characteristics of software quality. REST (REpresentational State Transfer) style is becoming a *de facto* standard adopted by many software organisations. The use of proper lexicon in RESTful APIs might make them easier to understand and reuse by client developers, and thus, would ease their adoption. *Linguistic antipatterns* represent poor practices in the naming, documentation, and choice of identifiers in the APIs as opposed to *linguistic patterns* that represent best practices. We present the DOLAR approach (Detection Of Linguistic Antipatterns in REST), which applies syntactic and semantic analyses for the detection of linguistic (anti)patterns in RESTful APIs. We provide detailed definitions of ten (anti)patterns and define and apply their detection algorithms on 15 widely-used RESTful APIs, including Facebook, Twitter, and YouTube. The results show that DOLAR can indeed detect linguistic (anti)patterns with high accuracy and that they do occur in major RESTful APIs.

Keywords: REST · Patterns · Antipatterns · Detection · Semantic Analysis

1 Introduction

Service-Oriented Architecture (SOA) has changed the way software systems are developed, deployed, and consumed [6]. The REpresentational State Transfer (REST) architectural-style [7] is becoming a *de facto* standard, adopted by large software organisations like Facebook, Twitter, Dropbox, and YouTube, for developing and publishing their services, *a.k.a.* their RESTful APIs.

In REST, well-designed URIs (Uniform Resource Identifiers) facilitate maintenance and evolution for APIs developers. Moreover, well-designed and named RESTful APIs may attract client developers more than poorly designed or named ones [14] because client developers must understand the providers' APIs while designing and developing their Web-based systems that use these APIs. Therefore, in the design and development of RESTful APIs, their understandability and reusability are two major quality factors.

Source code lexicon is shown to be an influential factor on the understandability, reusability and, overall, on the quality of software systems [12]. APIs designers use related natural names—natural language words—to name software entities [11]. In REST, linguistic relations among resources, services, and parameters are crucial [8] and the lack of such linguistic relations and–or poor naming may degrade the overall design of RESTful APIs and translate into *linguistic antipatterns*. Linguistic antipatterns are *poor* solutions to common recurring naming problems, which may hinder the consumption of RESTful APIs. In contrast, *linguistic patterns* are *best* solutions to common naming problems and may facilitate the consumption of RESTful APIs.

A number of *best* and *poor* linguistic practices for RESTful APIs design are listed in the literature [4, 8, 14] but they do not provide clear and detailed descriptions. In this paper, we represent those best and poor practices as patterns and antipatterns, respectively. For example, ✖*Contextless Resource Names* [8] is a linguistic antipattern that describes a URI composed of nodes from different semantic contexts as in the URI `www.example.com/newspaper/player` where "`newspaper`" and "`player`" do not belong to the same semantic context. On the contrary, ✓*Contextualised Resource Names* [8] is a linguistic pattern describing a URI composed of nodes that belong to the same semantic context and helping developers to understand better the resources or the interaction context with the server, and thus, increasing the understandability and reusability of an API. An example URI is `www.example.com/newspapers/media` because "`newspapers`" and "`media`" belong to the same semantic context. For RESTful APIs, the automatic detection of such linguistic patterns and antipatterns is a means to assess their understandability and reusability. However, no previous work analysed linguistic (anti)patterns in RESTful APIs.

In this paper, we present DOLAR (Detection Of Linguistic Antipatterns in REST), an approach supported by SOFA (Service Oriented Framework for Antipatterns) [17], which integrates syntactic and semantic analyses of RESTful APIs for detecting linguistic (anti)patterns. Semantic analyses are used to infer meaning and relationships among language elements whereas syntactic analyses focus on structural properties [9]. We propose (1) a detailed definition of ten common (anti)patterns for RESTful APIs [4, 8, 14] and their corresponding detection algorithms; (2) the DOLAR approach relying on the SOFA framework [17] extended with syntactic and semantic analyses based on WordNet[3] and Stanford CoreNLP[4]; (3) an empirical validation of DOLAR in which we analyse ten REST linguistic (anti)patterns on a set of 15 well-known RESTful APIs—including Facebook, Twitter, and YouTube—invoking over 300 methods. The validation results show that (1) DOLAR has an average precision and recall over 75% and (2) out of the 15 analysed RESTful APIs, most of them involve syntactical URIs design problems and they do not organise URIs nodes in a hierarchical manner. Moreover, we also observed that the REST APIs designers,

[3] wordnet.princeton.edu
[4] nlp.stanford.edu/software/corenlp.shtml

in general, use appropriate contextual resource names and they do not use verbs in URIs, which is a good URIs design practice in REST.

The remainder of the paper is organised as follows: Section 2 discusses related work. Section 3 presents the ten linguistic (anti)patterns. Section 4 presents the DOLAR approach. Section 5 presents a validation of DOLAR. Finally, Section 6 concludes the paper and sketches future work.

2 Related Work

Over the last years, several researchers (*e.g.*, [1, 2]) used semantic analyses to detect linguistic antipatterns and to check for consistency between source code and comments in object-oriented (OO) systems.

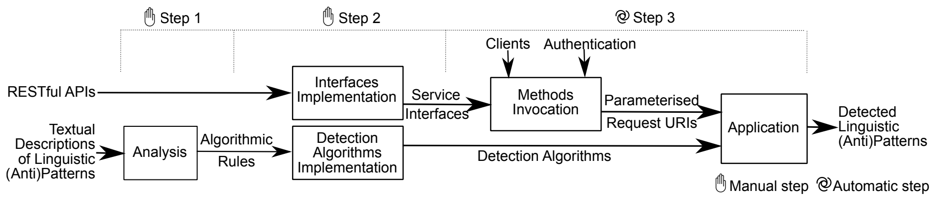
Abebe *et al.* [1] present a first set of lexicon bad smells in OO source code and a tool-suite that uses semantic analyses for their detection. Arnaoudova *et al.* [2] present a first definition of *linguistic antipatterns*, define 17 linguistic antipatterns in OO programming, and implement their detection algorithms. The authors search for the differences between the naming used for software entities (*e.g.*, method names and return types) and their implementation and/or documentation. For example, one antipattern they define *"Is" returns more than a Boolean*, which analyses the name of a method starting with "Is" and checks whether the method returns a boolean or not [2].

Semantic analyses are also applied to Web services design and development [15, 21]. Rodriguez *et al.* [21] present a study on bad linguistic practices identified on a set of WSDL descriptions and provide a catalog of Web services discoverability antipatterns. These antipatterns focus on the comments, elements names, or types used for representing the data models in WSDL documents. Mateos *et al.* [15] present a tool to detect a subset of antipatterns presented in [21].

Other researchers also use semantic analyses in different aspects of the software development life-cycle [3, 13, 20]. For example, Lu *et al.* [13] define an approach to improve code searches by identifying relevant synonyms using the WordNet English lexical database. Arnaoudova *et al.* [3] perform analyses on identifiers renaming in OO systems and classify them. Finally, Rahman and Roy [20] present an approach to automatically suggest relevant search terms based on the textual descriptions of software change tasks.

These approaches are tailored to OO identifiers and their consistencies with comments [1, 2] or to traditional SOAP-based Web services interfaces [15, 21], and therefore, they cannot be applied to RESTful APIs due to their intrinsic nature. For example, the invocation of RESTful services relies on a uniform interface formed using HTTP methods to access or modify resources via URIs.

Some researchers have dealt with the linguistic aspect of RESTful APIs. For example, Hausenblas [10] performs a subjective analysis on RESTful APIs to assess the quality of the URIs naming. However, he does not perform an automatic nor a systematic analysis. Moreover, he does not search for specific antipatterns. Parrish [19] also performs a subjective lexical comparison between



Step 1
Step 2
Step 3
Clients
Authentication
RESTful APIs
Interfaces Implementation
Service Interfaces
Methods Invocation
Parameterised Request URIs
Application
Detected Linguistic (Anti)Patterns
Textual Descriptions of Linguistic (Anti)Patterns
Analysis
Algorithmic Rules
Detection Algorithms Implementation
Detection Algorithms
Manual step
Automatic step

Fig. 1. DOLAR approach.

two well-known RESTful APIs, *e.g.*, Facebook and Twitter. In the comparison, the author analyses, for example, the use of verbs and nouns in URIs naming.

Although the above two deal with linguistic aspects of RESTful APIs, they only rely on the subjective view on a set of good linguistic practices and recommendations. Thus, there is no dedicated approach to automatically assess the linguistic quality of RESTful APIs by detecting poor and best practices.

3 REST Linguistic Patterns and Antipatterns

Table 1 presents the ten linguistic (anti)patterns that we consider in this paper and that have been extracted from existing literature [4, 8, 14, 23].

4 The DOLAR Approach

We now present the DOLAR approach (Detection Of Linguistic Antipatterns in REST) for the analysis and detection of linguistic (anti)patterns in RESTful APIs. DOLAR proceeds in three steps, as shown in Figure 1.
Step 1. Analysis of Linguistic (Anti)Patterns: This step consists in analysing the description of REST linguistic (anti)patterns from the literature to identify their relevant properties. We use these relevant properties to define *algorithmic rules* for (anti)patterns.
Step 2. Implementation of Interfaces and Detection Algorithms: This step involves the implementation of detection algorithms for (anti)patterns based on rules defined in Step 1 and the service interfaces for RESTful APIs.
Step 3. Detection of Linguistic (Anti)Patterns: This step deals with the automatic application of detection algorithms implemented in Step 2 on RESTful APIs for the detection of linguistic (anti)patterns.

4.1 Analysis of Linguistic Patterns and Antipatterns

We analyse the definitions of the (anti)patterns listed in Section 3 to identify their linguistic aspects. A linguistic aspect for the detection of the *Contextless Resource Names* antipattern is, for example, to check if a URI nodes belong to the same semantic context.

Table 1. List of ten linguistic (anti)patterns in REST.

1. ✓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/newspapers/media/page?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 [8].
2. ✓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/university/faculty/` 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 [8].
3. ✓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 [14], a URI is amorphous if it contains: (1) upper-case letter (except for Camel Cases [16]), (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 [4]. (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.
4. ✓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) [8]. The use of such terms as resource names or requested actions is highly discouraged [14].
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* [23].
5. ✓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 [8].
Example: The first example URI is a POST method that does not use a pluralised resource, thus leading to ✗*Pluralised Nodes* antipattern. On the other hand, for the ✓*Singularised Nodes* pattern, the DELETE request acts on a single resource for deleting it. ✗`DELETE https://www.example.com/team/players` or ✗`POST https://www.example.com/team/player` ✓`DELETE https://www.example.com/team/player` or ✓`POST https://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 [8].

```
1: CONTEXTLESS-RESOURCE-NAMES($Request\text{-}URI$)
2:    $URINodes \leftarrow$ EXTRACT-URI-NODES($Request\text{-}URI$)
3:    for each $index = 1$ to LENGTH($URINodes$)-1
4:       $Set1 \leftarrow$ CAPTURE-CONTEXT-BY-SYNSETS($URINodes_{index}$)
5:       $Set2 \leftarrow$ CAPTURE-CONTEXT-BY-SYNSETS($URINodes_{index+1}$)
6:       if $Set1 \cap Set2 = \emptyset$
7:          print "Contextless Resource Names detected"
8:          break
9:       end if
10:   end for
```

Fig. 2. Algorithmic rule of the *Contextless Resource Names* antipattern.

Figure 2 shows the algorithmic rule we define for the *Contextless Resource Names* antipattern. We compare the context of every pair of nodes or resources in a URI, lines 4-6. We report a URI as an occurrence of this antipattern if we find at least one contextless relation among all possible resource pairs. Conversely, we report an occurrence of the corresponding pattern *iff* all possible resource pairs share at least one common context and are relevant for that particular URI.

We rely on WordNet and Stanford CoreNLP 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—*synsets*—each representing unique concepts which can be used interchangeably in a certain context. WordNet is useful in finding semantic similarity between words using its underlying *hypernym-hyponym* and *meronym-holonym* relations as Figure 3 depicts. In Figure 3a, `medium` is one of 11 synsets of 'media' and there exist different types of `medium` including `newspaper`, `film`, `telecommunication`, and so on defined in WordNet. Based on WordNet, `medium` is thus the *hypernym* of `newspaper` and `newspaper` is the *hyponym* of `medium`. Such relations also exhibit contextual relevance between words and can be useful for analysing *Contextless Resource Names* antipattern [8] in URIs. In addition, there exist *part-of*, *i.e.*, *holonym-meronym*, relations between words defined in WordNet (see Figure 3b). For example, a `university` consists of `faculty member`, `student`, and `department` and the `department` may include `biology` and `chemistry`. Thus, `university` is a *holonym* of `faculty member` and `faculty member` is a *meronym* of `university`. Such hierarchical relations defined in WordNet between words can be useful in analysing *Non-hierarchical Nodes* antipattern [8].

Moreover, 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). We also define algorithmic rules for nine other linguistic (anti)patterns.

4.2 Implementation of Interfaces and Detection Algorithms

This step includes the implementation of services' interfaces and the implementation of detection algorithms of linguistic (anti)patterns. We implemented the service interfaces of RESTful APIs under study using JAVA, which contain the

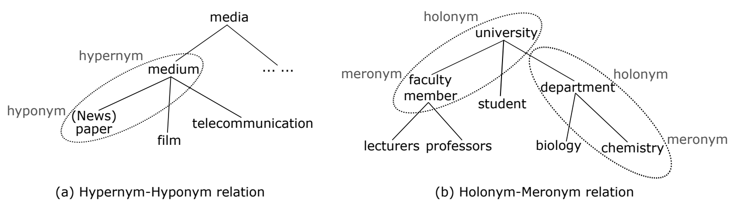


Fig. 3. *Hypernym-Hyponym* and *Meronym-Holonym* relations in WordNet.

methods callable to access or modify services' underlying resources. Each of the interface methods is mapped to a HTTP method. Using the appropriate HTTP methods, our DOLAR approach sends HTTP requests to real RESTful APIs and receives HTTP responses. Linguistic (anti)patterns, for example, *Amorphous URIs* (or *Tidy URIs*) require the fully-parameterised request URIs to be detected, which can only be obtained after HTTP requests are made. For each RESTful API, the details required to implement its service interfaces, *i.e.*, resources, HTTP actions to perform on its resources, and the parameters for each HTTP request, can be found in its online documentation as shown in Table 2. For other linguistic (anti)patterns, it is enough to extract URIs from the documentation of the RESTful APIs and then to analyse them.

Like the REST service interfaces, the detection algorithms for linguistic (anti)patterns are also written in JAVA. In fact, we manually transform the algorithmic rules defined the previous section into the executable programs.

4.3 Detection of Linguistic Patterns and Antipatterns

Methods Invocation: For each RESTful API, besides the service interfaces, we also implement clients to call the methods in the service interfaces, which perform read, write, update, or delete operations on resources. These explicit calls are done at detection time to obtain fully parameterised request URIs sent to the servers, which are required for detecting (anti)patterns like *Amorphous URI*. In REST, a resource may be related to multiple Java methods because any of the four basic operations (GET, POST, PUT, and DELETE) can be performed. As for the clients authentication, large companies often requires clients authentication to accept secured HTTP requests. Thus, we also implement the *OAuth 2.0* authentication protocol. In the end, this step produces the set of all parameterised requests URIs and their responses.

Application of Detection Algorithms: The SOFA framework (Service Oriented Framework for Antipatterns) [17] automatically applies the algorithmic rules in the form of detection algorithms on the parameterised requests URIs from the clients, collected in the previous step. Finally, the SOFA framework returns a set of detected REST linguistic (anti)patterns.

The SOFA framework, uses a Service Component Architecture (SCA) [5]. It relies on FraSCAti [22] for its runtime support. We added 13 REST (anti)patterns related to the design of REST requests/responses in a previous work [18]. We extend SOFA with detection support of REST linguistic (anti)patterns using linguistic analyses based on WordNet and Stanford CoreNLP.

Specifically, we extend the *REST Handler* component to facilitate the detection of REST linguistic (anti)patterns by wrapping each RESTful API in an SCA component and applying the detection algorithms on the SCA-wrapped RESTful APIs. By wrapping each API, we can introspect each full request URI with its actual runtime parameters, relying on *FraSCAti IntentHandler*, a runtime interceptor. We invoke methods from a service interface defined with an *IntentHandler* to introspect the request details, which allows on-the-fly syntactic and semantic analyses of parameterised request URIs.

5 Validation

In this section, we assess the effectiveness of DOLAR approach by showing the accuracy of the defined algorithmic rules, the extensibility of our SOFA framework, and the performance of the detection algorithms.

5.1 Hypotheses

We define three hypotheses to assess DOLAR's effectiveness:
H_1. Accuracy: *The set of all defined rules have an average precision and recall of more than 75%, i.e., more than three out of four are true positives and we do not miss more than one out of four of all existing (anti)patterns.*
H_2. Extensibility: *Our SOFA framework is extensible for adding new service-oriented and REST-specific (anti)patterns. In addition, SOFA facilitates an easy integration of new RESTful APIs.*
H_3. Performance: *The concretely implemented detection algorithms perform with a low detection times, i.e., on an average in the order of seconds.*

5.2 Subjects and Objects

The subjects of our study are the ten REST linguistic (anti)patterns described in Section 3. The objects are 15 common and well-known RESTful APIs for which we found documentations. We choose APIs whose underlying HTTP methods, APIs end-points, and authentication mechanisms are well explained, for example Facebook, Twitter, Dropbox, or YouTube, as summarised in Table 2.

5.3 Validation Process

We followed the instructions in the online documentation for APIs and implemented their (authenticated) clients. We invoked a set of 309 REST methods from 15 RESTful APIs to access their resources. We collected all fully parameterised request URIs from the clients and responses from the servers. Later, we applied our algorithmic rules in the form of detection algorithms implemented

Table 2. List of 15 analysed RESTful APIs and 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`

manually on the REST requests URIs and report (anti)patterns detected by our SOFA framework. We validated the results 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 (anti)patterns. We chose those four APIs based on our previous findings [18], 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 to define a ground truth for a predefined subset of the analysed URIs. 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 [18]. 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. We used precision and recall to measure the detection accuracy. Precision is the ratio between the true detected (anti)patterns and all detected (anti)patterns. Recall is the ratio between the true detected (anti)patterns and all existing true (anti)patterns.

5.4 Interpretation of the Results

The mosaic plot in Figure 4 shows the pattern-wise representation of the detection results on the 15 RESTful APIs. Columns correspond to each (anti)pattern

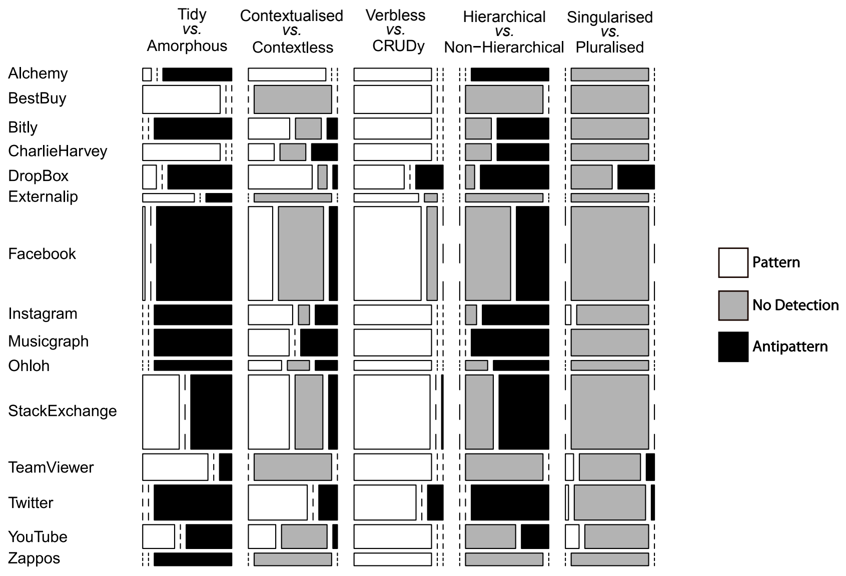


Fig. 4. Linguistic (anti)patterns detected in each RESTful API.

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 4, 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. However, the conclusions drawn above are based on the analysis results obtained applying the DOLAR approach.

Table 3 presents detailed detection results for the ten linguistic (anti)patterns on 15 RESTful APIs. The table reports the (anti)patterns in the first column followed by the analysed RESTful APIs in the following fifteen columns. For each RESTful API and for each (anti)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. The detailed detection results for all the 309 tested methods from 15 RESTful APIs are available on our project Web site `http://sofa.uqam.ca/dolar/`.

As shown in Table 3, 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

Table 3. Detection results of the ten REST lexical (anti)patterns (numbers in parenthesis show the number of methods tested for each API).

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**
Lexical Antipatterns/Patterns																	
✗Amorphous URI	8	0	15	0	14	2	65	14	19	7	28	3	25	10	9	**219**(71%)	0.984s
✓Tidy URI	1	20	0	12	3	4	2	0	0	0	25	16	0	7	0	**90**(29%)	0.968s
No Detection	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**(0.0%)	–
✗Contextless Resource Names	0	0	2	4	1	0	7	4	9	2	6	0	6	1	0	**42**(14%)	0.565s
✓Contextualised Resource Names	9	0	8	4	14	0	21	8	10	3	28	0	19	6	0	**130**(42%)	0.66s
No Detection	0	20	5	4	2	6	39	2	0	2	19	19	0	10	9	**137**(44%)	–
✗CRUDy URI	0	0	0	0	6	0	0	0	0	0	1	0	5	0	0	**12**(4%)	0.737s
✓Verbless URI	9	20	15	12	11	5	58	14	19	7	52	19	20	17	9	**287**(93%)	0.677s
No Detection	0	0	0	0	0	1	9	0	0	0	0	0	0	0	0	**10**(3%)	–
✗Non-hierarchical Nodes	9	0	10	8	15	0	28	12	19	5	34	0	25	6	0	**171**(55%)	0.584s
✓Hierarchical Nodes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**(0.0%)	0.592s
No Detection	0	20	5	4	2	6	39	2	0	2	19	19	0	11	9	**138**(45%)	–
✗Pluralised Nodes	0	0	0	0	8	0	0	0	0	0	0	2	1	0	0	**11**(4%)	0.668s
✓Singularised Nodes	0	0	0	0	0	0	0	1	0	0	0	2	1	3	0	**7**(2%)	0.656s
No Detection	9	20	15	12	9	6	67	13	19	7	53	15	23	14	9	**291**(94%)	–

pattern correctly—APIs providers use contextual resources names as nodes in URIs design—an important factor that affects the understandability of RESTful APIs. 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. However, a domain specific dictionary might reason about URIs contexts more accurately.

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.

The occurrences of *CRUDy URI* antipattern 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*. 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. 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 for this paper). 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:

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

5.5 Further Discussion of the Results:

Table 4 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%.

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:

```
1. {baseURI}/{media|revisions|shares}/dropbox/MyDropboxFolder/...
2. {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

Table 4. 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**							**Validation 2**				
	DOLAR		**Validated**	**Precision**	**Average Precision**	**Recall**	**Average Recall**	**DOLAR**		**Validated**	**Precision**	**Average Precision**
	P	TP						P	TP			
✖ 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%**

easy to infer the hierarchical relations among those pairs simply because they use a natural naming scheme [11]. 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:

```
1. {baseURI}/help/privacy.json
2. {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 4: two cases were identified as *Contextless 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%.

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

5.6 Discussion on the Hypotheses

We now discuss the hypotheses defined in Section 5.1.

H$_1$. Accuracy: From Table 4, for the first validation on Dropbox API, we obtained an average precision of 81.4% and recall of 78% (Validation 1). As for the second validation, on a partial set of tested methods on Facebook, Dropbox, Twitter, and YouTube (*i.e.*, 50 out of 125 tested methods), we obtained an average precision of 79.7% (Validation 2). However, for the second validation, we cannot calculate recall because we validated only a part of all tested methods. Moreover, for the manually validated subset of URIs, we had a lower precision ranging between 16.67% and 21.43% only for *Non-hierarchical Nodes* antipattern due to the limitations of WordNet dictionary. Thus, despite lower precision for one specific antipattern, with an average precision of 81.4% and 79.7%, and a recall of 78% for all (anti)patterns, we can positively support our first hypothesis on the accuracy of our defined set of rules and the detection algorithms.
H$_2$. Extensibility: We added to SOFA ten new REST linguistic (anti)patterns, which required semantic analyses for their detection. At present, SOFA can detect a set of 23 REST (anti)patterns from both syntactic and semantic aspects. Furthermore, we added three new RESTful APIs (*i.e.*, Instagram, StackExchange, and Externalip), and more than 190 new HTTP requests from [18]. To add new (anti)patterns, one needs to implement and integrate their detection algorithms within SOFA architecture. To add a new RESTful API, one must add its service interface, the underlying methods of the service, an authenticated client that can invoke these methods, and a wrapper SCA component, which specifies the bindings, base URI, and various runtime properties. Thus, it is possible to add new (anti)patterns, which supports our second hypothesis.
H$_3$. Performance: Table 3 (last column) shows the detection time for each pattern and antipattern, ranging between 0.565s and 0.984s, with an average of 0.709s. In fact, the total required time also includes the execution time, *i.e.*, sending requests and receiving responses (ranges from 2.074s to 20.656s, with an average of 6.92s). We performed our experiment on an Intel Core-i7 with a processor speed of 2.50GHz and 8GB of memory. The reported detection times are comparatively low (on an average, 10% of the total required time). However, the total required time also depends on the number of tested methods for each API. With such a low average detection time of 0.709s and execution time of 6.92s, we can positively support our third hypothesis on performance.

5.7 Threats to Validity

To minimise the threat to the *external validity* of our results, we performed experiments on 15 well-known APIs by invoking over 300 methods. We used WordNet for lexical and semantic analyses of URIs. However, one limitation of WordNet is that it does not include information on the semantic similarity between words. In addition, the number of defined relationships among words is limited and it lacks compound concepts or words. For example, we found URIs with compound resource identifiers that, when split, may cause loosing contextual information. This threat to the *internal validity* affected our detection results. However, we plan to incorporate other similarity measure techniques like second order similarity to improve our semantic analysis.

The detection results may deviate depending on the defined algorithmic rules of linguistic (anti)patterns. Engineers may have their own views and levels of expertise on REST linguistic (anti)patterns, which may affect the definition of algorithmic rules. We tried to minimise this threat to the *construct validity* by defining all rules after a thorough review of definitions in existing literature on REST linguistic (anti)patterns. We also involved three professionals in the validation of the results and involved a third expertise if conflicts arose. Finally, to minimise the threat to *reliability validity*—the possibility to replicate this study—we gather the details to replicate this study, including the algorithmic rules and the client request URIs, on our Web site.

6 Conclusion and Future Work

REST client developers need to understand well RESTful APIs while designing and developing their own Web-based systems. Understandability and reusability are thus two major factors that APIs providers must consider. This paper presented DOLAR (Detection Of Linguistic Antipatterns in REST), an approach supported by the SOFA framework [17] extended with syntactic and semantic analyses, for the detection of linguistic (anti)patterns in RESTful APIs.

We applied DOLAR to specify ten linguistic (anti)patterns. We validated DOLAR by analysing 15 RESTful APIs and invoking 309 methods and showed its accuracy: (1) an average precision of 81.4% and recall of 78% on Dropbox and (2) an average precision of 79.7% for a partial validation on Facebook, Dropbox, Twitter, and YouTube. We also observed that out of the 15 analysed RESTful APIs, most of them involve syntactical URIs design problems and do not organise URIs nodes in a hierarchical manner. However, the REST APIs designers, in general, use appropriate resource names fit for a context and they do not use verbs in URIs, which is a good URIs design practice in REST.

As future work, we want to apply DOLAR on other RESTful APIs and to Open Linked Data. We plan also to include domain-specific ontologies in the semantic analyses to overcome the limitations of English dictionaries and to apply other natural language processing techniques like second order similarities. We want to perform a validation of DOLAR results with RESTful APIs developers.

Acknowledgements The authors thank Charlie Faucheux for initiating the study. This study is supported by NSERC (Natural Sciences and Engineering Research Council of Canada) and FRQNT, Canada research grants.

References

1. Abebe, S.L., Haiduc, S., Tonella, P., Marcus, A.: Lexicon Bad Smells in Software. In: 2009 16th Working Conference on Reverse Engineering. pp. 95–99. IEEE (2009)
2. Arnaoudova, V., Di, M.: Linguistic Antipatterns : What They Are and How Developers Perceive Them. Empirical Software Engineering (2015)

3. Arnaoudova, V., Eshkevari, L.M., Penta, M.D., Oliveto, R., Antoniol, G., Gueheneuc, Y.G.: REPENT: Analyzing the Nature of Identifier Renamings. IEEE Transactions on Software Engineering 40(5), 502–532 (May 2014)
4. Berners-Lee, T., Fielding, R.T., Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax (2005)
5. Edwards, M.: Service Component Architecture (SCA). OASIS, USA (April 2011)
6. Erl, T.: Service-Oriented Architecture: Concepts, Technology and Design. Pearson Education, Boston, ME, USA (2005)
7. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
8. Fredrich, T.: RESTful Service Best Practices: Recommendations for Creating Web Services (May 2012), `http://www.restapitutorial.com/resources.html`
9. Goddard, C.: Semantic Analysis: A Practical Introduction. Oxford Textbooks in Linguistics, OUP Oxford (2011)
10. Hausenblas, M.: On Entities in theWeb of Data. In: Wilde, E., Pautasso, C. (eds.) REST from Research to Practice, pp. 425–440. Springer (2011)
11. Laitinen, K.: Estimating Understandability of Software Documents. SIGSOFT Softw. Eng. Notes 21(4), 81–92 (Jul 1996)
12. Lawrie, D., Morrell, C., Feild, H., Binkley, D.: Effective identifier names for comprehension and memory. Innovations in Systems and Software Engineering 3(4), 303–318 (Oct 2007)
13. Lu, M., Sun, X., Wang, S., Lo, D., Duan, Y.: Query Expansion via Wordnet for Effective Code Search. In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering. pp. 545–549. Montreal, Canada (2015)
14. Massé, M.: REST API Design Rulebook. O'Reilly (2012)
15. Mateos, C., Rodriguez, J.M., Zunino, A.: A Tool to Improve Code-first Web Services Discoverability Through Text Mining Techniques. Software - Practice and Experience (2014)
16. Microsoft MSDN: Capitalization Styles, Available On: https://msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx
17. Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.G., Baudry, B., Jézéquel, J.M.: Specification and detection of SOA antipatterns. In: 10th International Conference on Service-Oriented Computing. vol. 7636 LNCS, pp. 1–16. Shanghai, China (2012)
18. Palma, F., Dubois, J., Moha, N., Guhneuc, Y.G.: Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach. In: Franch, X., Ghose, A., Lewis, G., Bhiri, S. (eds.) Service-Oriented Computing, Lecture Notes in Computer Science, vol. 8831, pp. 230–244. Springer Berlin Heidelberg (2014)
19. Parrish, A.: Social Network APIs : A Revised Lexical Analysis (2010)
20. Rahman, M.M., Chanchal, R.K.: TextRank Based Search Term Identification for Software Change Tasks. In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering. pp. 540–544. Montreal, Canada (2015)
21. Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M.: Improving Web Service Descriptions for Effective Service Discovery. Science of Computer Programming 75(11), 1001–1021 (2010)
22. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. Software: Practice and Experience 42(5), 559–583 (May 2012)
23. Tilkov, S.: REST Anti-Patterns, Available Online: www.infoq.com/articles/rest-anti-patterns (July 2008)