Institue of Software Engineering

Masterarbeit

Analyzing the Quality of RESTful APIs in a Migration Context: A Case Study in the E-Commerce Domain

Florian Frank Grotepaß

Course of Study:	M.Sc. Informatik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Justus Bogner
Commenced:	November 3, 2020
Completed:	May 3, 2021

Abstract

In this work, we compare two RESTful API versions with respect to their software product quality. These APIs are maintained by the Cologne-based company Trusted Shops. Both API versions provide similar functionality. The company wants to replace the old API version with the new version and ensure that the software product quality has improved. Therefore, we conducted a three-step case study where we first interviewed both API users and maintainers. We then conducted a qualitative data analysis of these interviews. Second, we manually detected (anti)-patterns and violations of common design rules and best practice in both API specifications. Lastly, we evaluated the two specifications with the help of four different analysis tools, three rule-based analysis tools and one tool based on quality metrics. All in all, the new API version eTrusted was perceived as superior over the old API version myTS by both API maintainers and users. However, only the manual search for (anti-)patterns and violations of API rules and best practices came to a similar conclusion as the qualitative data analysis. We identified eleven violations for myTS and only ten for eTrusted. The tool-supported evaluations showed different results. We found only 18 rule violations for the old API version but 19 for the new API version with the help of rule-based tool analysis. Also, seven out of ten metrics reported a better result for the old API version. Therefore, rule-based analysis and metric-based evaluation cannot be taken as an absolute indicator for software product quality. However, these techniques are still powerful to find low-level flaws in API specifications. Furthermore, with the help of metric-based evaluation, specific partial aspects of quality attributes can be determined. However, more research towards a uniform ruleset of service-interface (anti-)patterns and best practices is needed. Also, some metrics have to be validated.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Service-oriented architecture and web services and especially microservices have experienced a lot of exposure in the world of software development in the last decade. Amazon states that they deployed 23,000 changes per day to their web services in 2012 [24], for example. This extensive form of continuous deployment is only made possible due to a modern web service architecture, the latest hype being about microservices with RESTful APIs. Netflix, a video streaming service with about 50 million subscribers in 2015 and meanwhile more than 200 million subscribers [15], has over 500 microservices that make up their system architecture [46]. All of these web services and microservices expose pre-defined functionality through operations that are described in a RESTful API specification, a service contract.

But these API specifications are not of static manner. API providers further change and improve the provided operations. Evolution of APIs presents a lot of problems, especially for developers who have already integrated these APIs into their systems. Breaking changes of an API can make an existing integration and therefore a whole application useless. Developers claim that maintaining an existing integration of an API takes far more time and effort than the initial integration itself [12], [47], [41]. The software product quality of a specification is very important in this regard. Good usability makes interaction with an API and therefore integration and migration easier. Furthermore, good maintainability is necessary to able to update the specification easily and efficiently. It is therefore important to be able to assess the software product quality of an API.

1.1 Context and motivation

In the case of this work, Trusted Shops[1] - a Cologne based company that offers a quality seal and a rating system for mostly internet shops - provides a rare chance of a research opportunity. The Trusted Shops developers are replacing a RESTful web service API with a completely new API that features almost the same functionality - without the usual API evolution steps. The new API version shows a paradigm change in the design. Usually, API specifications evolve slowly so that only migrations to new major API versions feature complex changes that are relevant for the API clients.

These RESTful APIs can be used by shops that use the services of Trusted Shops in order to place Trusted Shops's quality seal on their own website. So, these shops can receive ratings and comments on their own services through this RESTful API, for example.

[1] **www.trustedshops.de**.

The Trusted Shops team now wants to find out if their new API version offers better software product quality. Also, through the given study design, we compare the perceptions of the stakeholders with both the manual analysis of (anti-)patterns and violations of design rules and best practices as well as the tool-supported evaluations.

1.2 Objectives

The goal of this case study is to analyze and compare the two major RESTful API versions of Trusted Shops as well as the perceptions of both API developers and users. Also, we compare the derived quality perceptions of the two API versions to the results of existing REST API analysis approaches, using tool-supported evaluations as well as manual search for (anti-)patterns and violations of design rules and best practices in the API specifications.

Our study can be seen as a holistic case study with a single unit of analysis - one business environment in the form of Trusted Shops - where we analyze two RESTful API specifications. It can also be seen as an embedded case study with two units of analysis, namely the two API versions. [40]

Our research questions concern perceptions of both API maintainers and API clients. As no actual third-party API clients who were willing to be interviewed could be identified, Trusted Shops developers fulfill both roles. Two maintainers of the new API version represent the maintainer side. Also, three developers who have previously invoked the new API as clients but never maintained it act as the API clients respectively.

So, we defined the following research questions **RQ1-5** together with the Trusted Shops team.

RQ1: How do developers and users perceive the *overall software quality* of the new API version (eTrusted) in comparison to the old version (myTS)?
More specific, this question aims to also answer if *usability* and *maintainability* of the new API version is perceived as an improvement over the old version by developers and users. Both API maintainers and users are interviewed about their perception of the two quality attributes but the maintainers' opinions are mostly relevant for the maintainability and the API users' opinions mostly for the usability. The collected qualitative data is then processed with the help of grounded theory. Answering this question fulfills the main purpose of this work as the title already implies.

RQ2: How do developers and users evaluate the API migration process from the old version to the new version?
This question investigates how hard the migration process is perceived by mainly users but also by maintainers. By answering this question, negative aspects as well as improvement potential of the whole process can be unfolded.

RQ3: Which established service-interface (anti-)patterns and violations of design rules and best practices can be found in the two API versions?
We manually analyze the two API specifications for common (anti-)patterns described in the literature. Furthermore, we also search for violations of API design rules and best practices. The

occurrence of (anti-)patterns and rule violations does not necessarily reflect the perception of software quality perfectly. So, by comparing the results, we can determine if manual search for (anti-)patterns can act as a good indicator for software product quality.

RQ4: How do existing RESTful API analysis tools evaluate both API versions?
In order to answer this question, evaluation tools for RESTful API specifications are determined by strategic search and review. These evaluations are then compared to the findings of the qualitative data analysis of the interviews to find out if the tool analyses come to similar conclusions. If this is the case, this might support the assumption that software product quality can be predicted with the help of such tools.

The questions **RQ1-4** can be seen as knowledge questions that help answer a more open technical research question relevant for the Trusted Shops team as follows (cf. [48]).

RQ5: How can the new API version as well as integration- and migration-related artefacts be further improved?
Answering this question, we propose improvement suggestions based on the outcome of the other analyses. By implementing the suggested changes, we expect the software product quality of the new API version to improve.

1.3 Structure of the thesis

First, the technical background of this work is described in chapter 2. We describe principles of service orientation as well as the architectural style REST. Second, related work is presented in chapter 3. Third, the functionality and the documentation of the two Trusted Shops API versions are presented in chapter 4. Next, the design, results, discussion, and threats to validity of the conducted case study are presented in chapters 5, 7, 6, and 8. Lastly, we give a conclusion with chapter 9.

2 Technical background

This chapter gives an overview of the technical concepts that are further addressed and used in other chapters of this work.

2.1 Software product quality

The international standard ISO/IEC 9126 was introduced in 1991 [23] to provide a standardized definition of software product quality and define respective quality models. The models differentiate between quality in use, external, and internal quality. The quality in use model e.g. describes, how effective and efficient a user interaction in a specific context is. Both the external and internal quality models are made up of the quality factors functionality, reliability, usability, efficiency, maintainability, and portability. Further differentiation between the two models is not made. The quality models for external and internal quality with its sub-quality-factors can be seen in figure 2.1.

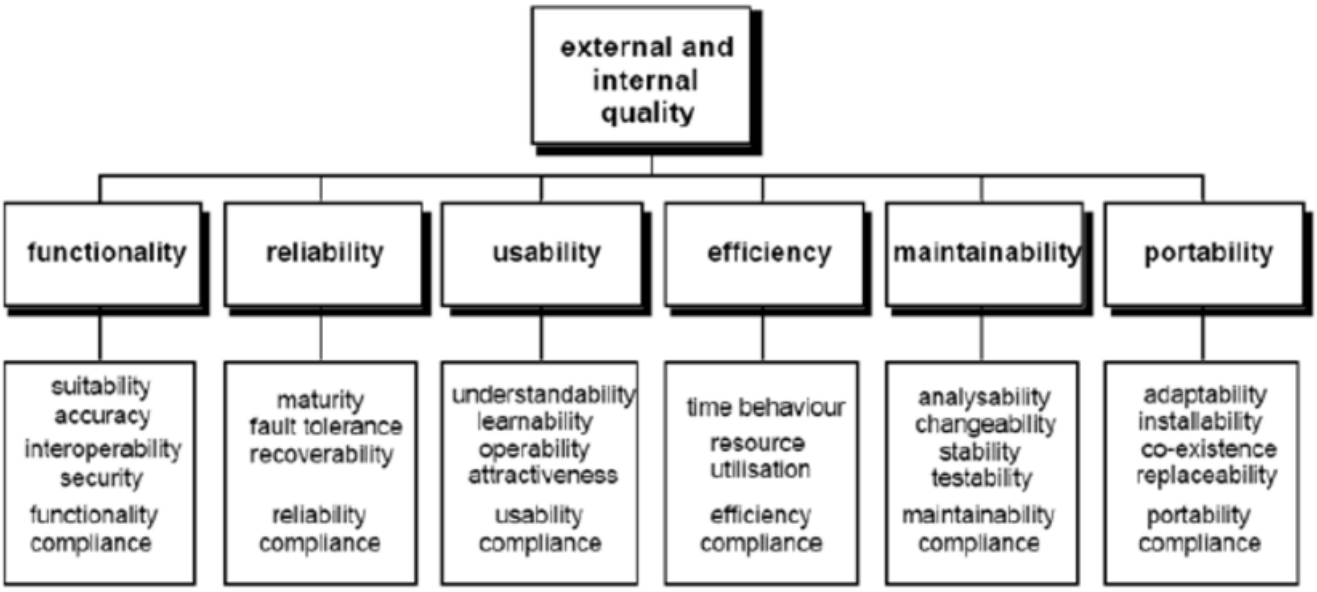

Figure 2.1: ISO/IEC 9126 quality model for external and internal quality [7]

Figure 2.2: ISO/IEC 25010 quality model for software product quality [22]

The latest standard ISO/IEC 25010 replaced the older standard in 2011 [23]. It is divided in a model about product quality as well as quality in use. Product quality is made up of the characteristics functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. This quality model is applicable to both software products as well as computer systems. [45], [23], [7]

Special interest lies in the sub-properties usability and maintainability, as those quality attributes are especially important in the context of our case study (cf. chapter 5.1).

Maintainability is described as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [45] in the product quality model of ISO/IEC 25010. Maintainability is made up of the fine-grained quality attributes modularity, reusability, analysability, modifiability, and testability. As these attributes together form maintainability, each one has an effect on the general quality of maintainability. So, parts of a software system built in a modular manner can therefore be easily updated as the system is made up of distinct modules with internally coherent functionality, for example. Consequently, switching to another database vendor might not be a problem, as all code concerning the database interfaces is located in a single module and not spread all over the system. Better maintainability therefore decreases the cost of changing parts in a computing system.

Usability is defined as the "degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [45] in ISO/IEC 25010. The sub-characteristics appropriateness recognizability, learnability, operability, user error protection, user interface aesthetics, and accessibility together make up usability. Therefore, an implementation that is hard to "learn" and understand decreases the general usability, for example. Also, if the protection for the user regarding errors is high, the user never gets to see an error because they are handled appropriately which improves usability.

2.2 Service-oriented architecture

Service-oriented computing is described as an "umbrella term that represents a distinct distributed computing platform" by Erl et al. [11]. Therefore, special design paradigms and principles etc. are included. The authors see seven different qualities that come with service-oriented computing. First of all, it causes "**Increased Intrinsic Interoperability**". Basically, services are designed in an interoperable manner that is attained intrinsically through given standards, as they are "designed to be naturally compatible" [11]. So, these pieces of software can be reused in different configurations. Services can be reorganized and used again when the business context changes. Also, "**Increased Federation**" is achieved due to having standardized contracts. Services can be built differently but offer a uniform way of interacting with them. The computing platforms should not be dependent on specific vendors and technologies so that "**Increased Vendor Diversification Options**" make it possible to change single services and technologies with products from other vendors. Furthermore, the services can be designed conceptually close to the business domain itself. So, these services can change together with the business or the whole business domain. Services are built to deliver "**Increased ROI**" (Return On Investment), as they are conceived for this reason. Also, this architectural paradigm comes with "**Increased Organizational Agility**" as well as "**Reduced IT Burden**". Services can be interchanged very fast which improves the possibility to change the system to business requirements agilely. Lastly, these goals help to reduce the cost and burden of applying these concepts.

Furthermore, **service orientation** is a design paradigm that aims at the "creation of solution logic units" [11] that help fulfilling these qualities that are supposed to come with service-oriented computing. So, service orientation introduces services as logical units of limited functionality. Working together, services are able to provide useful functions to the users. Service orientation is also influenced by object orientation known from software engineering. The design paradigms of service orientation are described by eight design principles, e.g. "**Service Loose Coupling**". This principle demands services to be decoupled from the other services. Also, service consumers should not be coupled to the services. The services are supposed to be loosely coupled in order to achieve a high grade of cohesion within a service and a better modularization: Functions that often invoke others should be in the same service, those that rarely communicate which each other should be in different services. Services should furthermore have "**Standardized Service Contract[s]**". Due to "**Service Abstraction**", these exposed contracts only keep necessary information. "**Service Statelessness**" however assures that the services do not have to manage e.g. client state. "**Service Reusability**" demands the services to be reusable due to high abstraction about e.g. data processing. Also, due to "**Service Autonomy**", services should be autonomous and in control of their own runtime environment. Furthermore, services should show "**Service Discoverability**". Therefore, they expose meta data which lets them be discovered. So, they can inform the outside about their functionality. Lastly, services should offer "**Service Composability**" so that the size and complexity of a single service does not matter. Due to their design, big and complex services interact the same way as small and easy to comprehend services.

Finally, **service-oriented architecture** (SOA) is an architectural paradigm that fulfills the goals and requirements of service-oriented computing and service orientation. So, the specific technologies and products that are used to apply service orientation are defined in the SOA implementation.

Fulfilling these goals leads to an architectural model where services are designed with limited functionality as a next step of modularization known in monolithic architectural styles. With the help of this architectural style, service-oriented computing and therefore "distributed computing platforms" are made possible. [11]

2.3 Microservices

Microservices are the next logical step onward from SOA. Microservices are smaller and therefore even more cohesive than services defined in SOA. They offer a very limited set of functionality that is more logically coherent. Furthermore, microservices should share as little resources and information as possible, whereas SOA implies sharing as much as possible [38]. So, each microservice usually stores the necessary data in its own database, for example. However, SOA services would, e.g. in an online shop context, all use the same master database that stores orders. A service that manages orders could use this database, but also a service that manages the warehouses might need this information. Furthermore, not only the database but also the invocation and processing infrastructure would be shared in an SOA context. This leads to less duplication of functionality and code but also increases coupling between the components. Microservices on the other hand are therefore decoupled in this regard but increase the overhead as a lot of functionality is duplicated. [38]

2.4 Representational state transfer

REpresentational **S**tate **T**ransfer (REST) is an architectural design paradigm proposed by Roy T. Fielding in his Ph.D thesis "Architectural Styles and the Design of Network-Based Software Architectures" [29], [14].

This architectural design proposal for web services helps to implement SOA and especially microservices as the constraints that come with these styles can be met with the help of REST.

2.4.1 REST constraints

REST introduces six constraints that are implemented to achieve the characteristic REST goals.

The following constraints are defined. [11], [33]

- **Client-Server** RESTful distributed computing systems are characterized by a design with a clear client-server-relationship. The server exposes its functionality and listens for requests. The response is returned to the client as well as possible error messages.
- **Interface/Uniform contract** The interface constraint demands that all services in a RESTful architecture provide a uniform interface. These are mostly addressed with HTTP. The interfaces act as pre-defined contracts of exposed functionality. They should be as abstract and generic as possible so that e.g. data models can be reused in different services. So, one

unified data model can be used in all services so that this model has to be implemented and comprehended only once. Also, this improves reusability and therefore maintainability in case of changing business concerns. These are isolated from the service contract itself but defined and handled in the services. Nevertheless, this might introduce unnecessary overhead, as shared data models often contain too much information for a specific service because each service might need different attributes, for example.

- **Stateless** RESTful services should be stateless. This obviously helps to fulfill the "Service Statlessness" design proposal for SOA, furthermore the "Service Loose Coupling" aspect of SOA discussed in chapter 2.2. The client attaches all necessary information to the queries which prevents unnecessary communication between services. So, the client might attach all session state data of an older interaction with the service to the request. Therefore, the service itself does not have to invoke another service where this session state data might be saved, which would make the interaction stateful.
- **Cache** A message can be flagged as cacheable in its HTTP header. This is especially important for big computational infrastructures that contain a middleware where incoming queries are distributed to the according services. The middleware can be ordered to store single responses so that the next client with the same query can get the cached response. The respective service does not have to be invoked once again. The time frame in which a cached response can be used is added to the HTTP header as well so that the response does not get stale [13]. This spares the services from queries and therefore from computational load, as some of the requests can already be answered by the middleware.
- **Layered System** RESTful distributed computing systems have to be designed in a layered manner. One architectural layer can only communicate with its neighbouring layers. The layers themselves do not know about the other layers apart from the neighbouring layers. So, a consumer does not know if the response message was cached by the middleware or if it was processed by the actual service, for example.
- **Code-On-Demand** This constraint is optional and not necessary for a system to be labelled RESTful. Web services can provide further executable code for e.g. applets and other client-sided applications.

2.4.2 REST goals

With the help of the REST constraints, see chapter 2.4.1, systems implementing and following these now offer a set of qualities - the seven REST goals [11]:

- **Performance** A few of the REST constraints help to improve performance of respective distributed computing systems. As service responses can be cached by a middleware, performance as well as response time improves because the main system does not receive the cached queries in the first place. So, the load of the main system stays low so that the delay to the client is minimized. Also, the services in a RESTful architecture are stateless and therefore unnecessary interactions with other services that save the session state are prevented.

However, the uniform contract constraint might impair performance as this can introduce overhead due to redundancy. Furthermore, the messages themselves shared by the client can become bigger and more costly as they now contain the client state.

- **Scalability** Due to the nature of interchangeable services in SOA and REST, scaling the entire system is easy. Computing systems can be scaled vertically by adding computing resources such as RAM and storage to a single computing instance. Horizontal scaling is a technique where scaling is achieved by adding new computing instances to the already existing ones. So, the queries are distributed to running services, e.g. with the help of a middleware in form of a load balancer. This REST goal is supported by the REST constraints for stateless services, as the load balancing in the middleware is able to send requests to any redundant "scaled out" service because the state information is attached to the query. Furthermore, the layered architectural style itself enables this load balancing - also with the help of the statelessness constraint - because clients do not know which of the redundant services answered their queries. A request might even be cached in another layer, e.g. the middleware, and never reach a single service. Most REST constraints themselves therefore support scaling out.
- **Modifiability** Modifiability as a sub-quality attribute of maintainability which is in a special focus of this work. It is of high relevance in RESTful distributed computing systems. Modifiability benefits from the RESTful architectural style due to the layered system and stateless services constraints. This is also relevant for the evolution of APIs. As each layer only communicates to the neighbouring layers and does not even know about different layers, other layers and parts of them can be exchanged. So, an entire middleware can be exchanged for a different middleware by another vendor, as long as the points of integration are adapted. These integration logic implementations should be hidden from the main domain logic. Furthermore, single services can easily be developed, deployed, and operated by one team - this principle is called DevOps [8]. Also, when changing a single service it is not necessary to redeploy the entire computing system but just this specific service.
- **Simplicity** RESTful distributed computing systems are easy to understand due to the design paradigm of "separation of concerns" [11]. Small services with separated responsibilities instead of a monolithic structure make the functionality easily understandable. As the services are "stateless", each request contains every information needed. There are no complex back-and-forth-conversations within the system to e.g. receive the application state for a current client from a service managing client state. Also, common media types are used between the services due to the "uniform contract" constraint which simplifies the interactions.
- **Visibility** As a result of the "uniform contracts" constraint, the middleware can monitor and regulate interactions between services more easily. Without knowing any service-specific contract details, the middleware can extract common generic information.
- **Portability** The "code-on-demand" constraint helps to provide portability. JavaScript code provided by a server is highly portable and can be run in almost any browser, for example.

- **Reliability** RESTful computing systems should be built in a reliable manner. Bottlenecks due to the lack of scalability potential in a monolithic architecture can be prevented. Also, the design offers monitoring mechanisms (e.g. between the layers) so that failures can be recognized early on.

2.4.3 Richardson Maturity Model

Richardson proposes a four-stage levelling system that classifies the maturity of REST principles [29], [33].

Level 0 only demands the use of HTTP as the transport protocol in communication with a web service.

Level 1 requires the API to expose individual resources. They are addressed with the help of different identifiers. Also, various operations can be invoked on a single resource.

To achieve **level 2**, APIs have to support HTTP methods such as POST and PUT in a semantically correct way instead of only GET. So, e.g. GET and DELETE operations are of idempotent manner so that they can safely be executed more that once without changing the state after the first successful try. Also, e.g. a GET request as a safe operation would not create or delete a resource.

Level 3 describes APIs that implement the principle of HATEOAS (Hypertext As The Engine Of Application State) [33]. Such APIs e.g. return hyperlinks or forms that can be used to send other requests that could be relevant for the initial queries. So, if a new instance of a resource is created with a POST operation, the response should contain a link to the GET operation for the respective resource. Furthermore, if a GET operation with a specific ID is sent, a link to deeper nested resources that also use this ID should be attached to the response.

The maturity level also has an effect on the software product quality of RESTful APIs. By strictly implementing HATEOAS principles in an API, one might improve usability. If links to associated resources are always added, it helps the user navigate through the API without hard-coded links to specific resources. Furthermore, this also supports loose coupling between clients and APIs which leads to better maintainability on the client side, as no hard-coded addresses have to be updated after a change of the API. Applying the right HTTP verbs to the right operations leads to better understandability, as e.g. updating a resource with a GET operation would be too confusing. Also, the API itself can be seen as more maintainable because better understandability improves analysability, a sub-property of maintainability.

2.4.4 RESTful API description languages

There is a great number of RESTful API description languages. Di Martino et al. [10] compare the four most common RESTful API description languages, namely API Blueprint, RAML, WADL, and OpenAPI specification. Most of the description languages are based on similar language formats and serialization languages, e.g. both RAML and OpenAPI can be expressed in YAML as well as JSON - JSON is a subset of YAML. API Blueprint documents are written in Markdown syntax which is easily readable. Only WADL is based on the more complicated XML format. [9], [10]

Listing 2.1 Exemplary meta information of an OpenAPI specification

```
openapi: "3.0.0"
  info:
  version: 1.0.0
  title: Swagger Petstore
  description: A sample API that uses a petstore as an example to
demonstrate features in the OpenAPI 3.0 specification
  termsOfService: http://swagger.io/terms/
  contact:
    name: Swagger API Team
    email: apiteam@swagger.io
    url: http://swagger.io
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0.html
servers:
  - url: http://petstore.swagger.io/api

```

The two Trusted Shops API versions we compare to each other in this work are provided as OpenAPI or Swagger specification documents, an early version of the OpenAPI specification. So, the following paragraph goes into detail about the necessary parts of this description language.

OpenAPI Specification

The first version of the OpenAPI specification[1] was released in 2011 as the Swagger Specification. The specification has been named OpenAPI Specification since its version 3.0 in 2017. The current version is 3.0.3. All example listings are taken from the OpenAPI initative exemplary specification[2] and might be shortened.

The specification can be represented in either JSON or YAML format. The entire specification document can be seen as one JSON object. The documentation consists of different segments. As seen in listing 2.1, OpenAPI specifications start with the meta information about the document, such as the title of the API and its version.

Resources can be manipulated with respective endpoints. Listing 2.2 describes a GET operation that can be performed on the resource “pets”, for example. Request parameters are described in the parameters field. Parameters can be added as query parameters to the end of the URI with a “?” and the respective parameter name. Also, path parameters can be used that act as path elements themselves without the need of the parameter name in the URI. They are surrounded by curly

[1] **https://swagger.io/specification/**.
[2] **https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/petstore-expanded.json**.

Listing 2.2 Exemplary endpoint in an OpenAPI specification

```
{
   "paths":{
      "/pets/{id}":{
         "get":{
            "description":"Returns a user based on a single ID, if the
user does not have access to the pet",
            "operationId":"find pet by id",
            "parameters":[
               {
                  "name":"id",
                  "in":"path",
                  "description":"ID of pet to fetch",
                  "required":true,
                  "type":"integer",
                  "format":"int64"
               }
            ],
            "responses":{
               "200":{
                  "description":"pet response",
                  "content":{
                     "application/json":{
                        "schema":{
                           "$ref":"#/components/schemas/Pet"
                        }
                     }
                  }
               }
            }
         }
      }
   }
}
```

brackets in the path of the OpenAPI specifiation. In listing 2.2, {id} acts as such a path variable. When sending a request to this endpoint, this item is substituted with the actual ID. Responses are defined in the response objects of the respective endpoints. Here, the status code as well as the possible response models are declared.

Furthermore, reusable parts of the specification such as response models, here the model for "'Pet", can be denoted with $ref and can then be defined in the components section of the document, see listing 2.3.

Listing 2.3 Exemplary components element of an OpenAPI specification

```
{
  "components": {
    "schemas": {
      "Pet": {
        "type": "object",
        "required": [
          "id",
          "name"
        ],
        "properties": {
          "id": {
            "type": "integer",
            "format": "int64"
          },
          "name": {
            "type": "string"
          },
          "tag": {
            "type": "string"
          }
        }
      }
    }
  }
}

```

2.5 (Anti)-patterns

Gamma et al. [16] propose a documentation style for “Design Patterns” as early as 1995. The authors claim that designers should reuse solutions from previous problems in order to solve similar new issues. This leads to patterns that can be reused regularly. “These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable” [16]. The authors compare software design patterns to actual architectural patterns that bring the same beneficial effect: “Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context” [16]. They propose a reporting style with four main elements: The name of the pattern should already be expressive of the problem domain. This enables designers to talk about problem solutions on a higher abstraction level, as the name is optimally self explanatory. Furthermore, the problem has to be reported. The context of the problem is important as well. The solution then “describes the

elements that make up the design, their relationships, responsibilities, and collaborations" [16]. This should not be too specific as a general design problem is discussed. Finally, consequences should be reported. Positive effects as well as the negative side-effects should be included.

In opposite to positive design patterns, Ganesh et al. [17] propose a template that can be used to report anti-patterns, often also referred to as (architectural) "bad smells". Generally, the term bad smells is also used for lower-level flaws in code. Anti-patterns therefore describe recurring structural flaws in the design of an API. The proposed reporting template is similar to the formal style of reporting the positive patterns. Overall, they describe twelve elements of such a report as e.g. name of the anti-pattern, short description, long description, and example(s). In opposite to reporting positive effects of patterns, here the "rationale", "violated principles", and "impacted quality attributes" report elements describing the negative effects of such an anti-pattern. A solution can be suggested with the "detected strategy" and "suggested refactoring" elements.

3 Related work

A lot of research endeavors have been made towards analysis and evaluation of web APIs and especially their documentation. Palma et al. [31] propose a three-step approach for detecting and analyzing linguistic patterns and anti-patterns. The authors implemented this detection mechanism for ten different linguistic patterns and anti-patterns. Many of these (anti-)patterns concern the URIs of REST resources, such as "Hierarchical Nodes". This pattern requires the resources or nodes in an URI to be ordered in a hierarchical manner. The lack of this requirement would then lead to the "Non-hierarchical Nodes" anti-pattern. Other (anti-)patterns also take HTTP verbs such as GET and DELETE into account and check if they are used in the right context and do not change the semantics and therefore the expected behavior. In a second step, the authors implemented dummy services for real-world RESTful APIs to receive and analyze the HTTP requests. This is necessary for some (anti-)patterns to get the "fully-parameterized requests URIs" that they receive after the HTTP request has been made. So, e.g. "Amorphous URIs" can be found. Therefore, they also implemented API clients that send requests to the dummy services. Finally, a set of patterns and anti-patterns analyzed with the help of the clients' requests is returned. Other (anti-)patterns, e.g. "Hierarchical Nodes", can already be determined with the help of the static RESTful API specification. In a similar work, Petrillo et al. [36] also take description fields in API documentations into account. The authors used natural language processing algorithms to find out about relationships between terms within the API documentation and the URIs. So, they were able to find occurrences of "Pertinent vs. Non-pertinent Documentation". This pattern describes URIs that contain terms which are not semantically related to their documentations. They claim to be more cloud domain specific than the work by Palma et al. [31] as the results in the other work were not easily comparable due to the variety of domains of the examined APIs, e.g. Facebook, Twitter, and Bestbuy.

Brabra et al. [6] propose several interface (anti-)patterns for the Open Cloud Computing Interface (OCCI), a standard for cloud service communication that also follows RESTful principles. Furthermore, the authors investigated five web APIs about REST (anti-)patterns as well as the OCCI (anti-)patterns. Most of the analyzed APIs stick to the RESTful patterns, the OCCI patterns are more often violated.

Bogner et al. [5] followed a metric-based research approach. They implemented an API specification evaluation tool for RESTful API descriptions that can process three formats, namely OpenAPI, RAML, and WADL. So, API specifications can be analyzed with the help of metrics concerning different properties. The team so far implemented metrics for analysis of complexity, cohesion, and size. The tool is implemented in a modularized manner so that new metrics can be added easily. Furthermore, the authors evaluated more than 1700 public API specifications with the help of this tool to obtain thresholds for the metrics. They determined the thresholds by ordering the

metric results by numerical value and splitting the number of metric results of all specifications into quartiles. The authors report the lower and upper bound value of those quartiles respectively for each metric.

Espinha et al. [12] give an insight on evolution of APIs in the industry. They interviewed six developers who integrate external APIs into their applications. Furthermore, the authors analyse both the server-side and client-side code base evolution of two open-source APIs. They found that maintaining an integration in regard to evolving APIs might just take more time than the initial integration of the respective external API itself.

Wang et al. [47] researched which kind of API changes would cause more questions in the developer community. They analyzed questions on the developer forum StackOverflow[1] and the respective number of answers and views on each type of question. They found that adding a method causes the most absolute number of question threads. This type of posts also gains the most views. Furthermore, questions about deleted API methods generally initiate the most comments on the related posts. The authors argue therefore that those threads cause the most discussions.

Sohan et al. [41] conducted a case study in order to find out about versioning, documentation, and communication about APIs from different industries in an evolution context. They found that version identifiers do not offer enough information about backward compatibility when a new version is released. They criticize the lack of standards in this regard. Furthermore, they show that many API providers maintain the documentation of evolving APIs by manually editing the descriptions.

In summary, research has been made towards analysis of web APIs with both low-level design patterns and best practices as well as high-level service-interface (anti-)patterns. Also, research on software quality metrics for web APIs and even specifically for RESTful APIs can be found. These (anti-)patterns and metrics are often not validated which shows the research gap in this area. Furthermore, many publications about API evolution in a continuous context can be found. However, research on API evolution with major version change modifying the entire API design is lacking.

[1] **www.stackoverflow.com**.

4 Trusted Shops APIs

In this chapter, both Trusted Shop API versions are presented with respect to their functionality and documentation. The publicly available documentations of the old version myTS[1] and the new version eTrusted[2] are reviewed.

Trusted Shops[3] is a Cologne-based company that provides a quality seal for (online) shops (see figure 4.1 taken from the Trusted Shops website[3]). Trusted Shops assures the customers of these shops a safe purchase. So, customers have the right of cancellation through Trusted Shops. Also, they can rate the services of the shops afterwards.

In this chapter, we present the functionality and documentation of both versions.

Figure 4.1: Trusted Shops's quality seal

[1] **https://api.trustedshops.com/documentation/**.
[2] **https://developers.etrusted.com/etrusted-api.html**.
[3] **https://www.trustedshops.de/**.

4.1 Functionality

Customers of Trusted Shops, e.g. online shops, can integrate the API to gain access to reviews on their own services and products as well as comments on these reviews. After a purchase, the customer can be requested to rate and review the products and services. These review collection endpoints make up the main part of these APIs.

4.1.1 myTS

The old API version myTS is provided as a Swagger 1.2 document in its version 2.2.3. Through myTS, the API consumer can query a GET request to `/public/v2/shops?url=` with a URL as a query parameter where a list of shops that contain this string in their URLs is returned. This is a public endpoint which can be used to test the API. The requested return object contains a shop ID called "tsId". The API can be further used with the help of this ID, e.g. reviews can be queried and review requests can be sent to the customers. Also, through `/restricted/v2/shops/{tsId}` further information on a specific shop can be received, see listing 4.1. The name and the URL of a specific shop are returned, for example. This endpoint is restricted and credentials have to be provided to be able to invoke this endpoint.

Information about retailer shops can be requested with a query to `/restricted/v2/retailers/shops`. Customers of the API consumers can write reviews about the services of those shops that appear on the Trusted Shops website. These reviews can also be integrated into the API consumers' own websites through the Trusted Shops API. They can be queried with a GET request to `/restricted/v2/shops/{tsId}/reviews`. The results can be filtered by earliest and latest possible date and best and worst rating. Also, the results are provided as pages. The API consumer can browse through these pages by providing a page number and a page size which equals the amount of returned items. No hyperlinks to the next or previous page are attached. The user has to increase or decrease the page number by one. Each review has a review ID, in the paths for shop review comments referred to as {rvId}. Nevertheless, the response object returns this ID as "uID", nested as an attribute of the "review" object. Also, comment IDs that are later referred to as "coId" are returned with an attribute name of "uID". Furthermore, the array of review items is called "review" instead of "reviews". The entire nested "statement" array is optional, as it is only attached if a comment on a review has been made by the owners of the reviewed shops. It is not flagged as optional in the specification, however.

With the help of the comment ID and review ID received in the earlier described operations, the endpoint for review comments can be used. A comment on a review can be received with a GET request to `/restricted/v2/shops/{tsId}/reviews/{rvId}/comments/{coId}`. The API also offers POST and PUT functionality on this endpoint. So, API users can post comments with the help of this endpoint and edit them. The POST request for comments is not expected to carry a request object as payload but instead form parameters. So, the comment as well as the boolean "informBuyerForShopComment" is transmitted as a form parameter.

Listing 4.1 Return object for GET request to `/public/v2/shops/{tsId}` in myTS

```
{
  "data": {
    "shop": [
    {
      "tsId": "",
      "url": "",
      "name": "",
      "languageISO2": "",
      "targetMarketISO3": ""
    }
    ]
  },
  "message": "CallStatusMessage",
  "status": "CallStatus",
  "code": 0,
  "responseInfo": {
    "apiVersion": "",
    "count": 0
  }
}
```

With a GET request to `/restricted/v2/shops/{tsId}/quality`, aggregated information about reviews and complaints in a specific time span can be retrieved. This is called a shop quality indicator. Also, `/review` and `/complaints` can be attached to the URL to get specific aggregated information about either only reviews or complaints. With the help of `/restricted/v2/shops/benchmarks`, entire benchmarks of quality indicators can be retrieved as a map. Shop owners can generate hyperlinks by providing a customer and an order object as a payload sending a POST request to `/restricted/v2/shops/{tsId}/reviews/requests`. Such a hyperlink leads to a submission form for a shop review. This hyperlink can then be sent in an email or embedded in the API client's website. Furthermore, a similar endpoint is implemented, `/restricted/v2/trigger/submitReviewCollectorRequest`. The inputs are similar. However, the POST request does not return a link to a submission form but instead Trusted Shops itself sends an email to the customer with the hyperlink.

Also, product reviews can be requested in a similar manner as shop reviews (GET request to `/restricted/v2/shops/{tsId}/products/reviews`). This endpoint also offers pagination. Lastly, comments on product reviews can be posted and deleted as well as edited. (`/restricted/v2/shops/{tsId}/products/reviews/{productReviewUuid}/comments/{shopCommentUuid}`). No method for querying existing comments is defined, however. In opposite to creating a shop review comment, here the product review comment is attached

as a payload and not as a form parameter. In the documentation, the path parameters are now called “productReviewUuid” and “shopCommentUuid”, whereas before they were shortened to “tId”, “coId”, and “rvId”. This naming style is inconsistent. No hyperlinks required by the principles of HATEOAS (see chapter 2.4.3) are provided.

4.1.2 eTrusted

The new API version eTrusted in its current 1.0.0 release is the new API platform introduced by Trusted Shops. Instead of changing just a few endpoints and models that might already introduce breaking changes, an entirely new API is introduced with novel resources. The main functionality is similar, a few features are missing and a few new features are added.

Resources can be accessed through channels. Many resources are queried with the help of a channel ID. Channels can be seen as a filter. So, only relevant information for e.g. different online shops or countries can be predetermined.

Also, API clients can use locales to specify a language for the response. The locale is provided by the API client as a field in the payload. The new platform eTrusted also introduces a cursor-based pagination feature. With the help of the paging object that is part of the response body, queries with a lot of returned entities can be iterated easily. Cursors in the form of UUIDs that represent the current page are provided in the response. Furthermore, hyperlinks that contain the UUIDs are also attached. Those hyperlinks represent queries to the previous and next page. This is a more consequent way of implementing HATEOAS principles (cf. chapter 2.4.3) as opposed to myTS’s pageable resources. Invoking myTS, the user has to create the hyperlink to the next page themselves with the query parameter for the page number. An exemplary paging object can be seen in figure 4.2. Filters can also be used in pageable responses. So, reviews can e.g. be filtered by rating and date. The UUIDs in the hyperlinks represent the same filters that were used by the original query.

Channel objects can be queried via `/channels` with the help of a GET request. All channels that are associated with the requestor’s account are returned. Those channels can be updated with a PUT request to `/channels/{id}`.

Event objects can be created via `/events` and and can be updated with a PUT request to `/events/{eventRef}`. When customers receive an invitation via email to report a review, this is caused by an event. Events are triggered by so called “touchpoint[s] along your customer journey”[4] which are represented by the event types. Therefore, event types are used to define the triggers of events. An event type can be e.g. a checkout or a delivery. Checkout is the default event type. Event types can be queried and posted via `/event-types` and updated with a PUT request to `/event-types/{id}`. Furthermore, configurations can be made with invite rules. Each event type includes one invite rule. Invites represent the review invitations that are sent out to the customer via email. With the help of the invites endpoint, the API consumer can check the status of an invite, if it is sent out yet or not, and also see the respective event type. Invites can be queried via `/channels/{id}/invites`. There is no POST and PUT

[4] **https://developers.etrusted.com/etrusted-api.html**.

Listing 4.2 Exemplary paging object in a response body from eTrusted

```
{
  "paging": {
    "count": 10,
    "cursor": {
      "before": "rev-xxxxxxxa-yyyy-xxxx-yyyy-xxxxxxxxxxxx",
      "after": "rev-xxxxxxxb-yyyy-xxxx-yyyy-xxxxxxxxxxxx"
    },
    "links": {
      "previous": "https://api.etrusted.com/channels/{channelId}/
customer-reviews?count=10&before=rev-xxxxxxxa-yyyy-xxxx-yyyy-
xxxxxxxxxxxx",
      "next": "https://api.etrusted.com/channels/{channelId}/customer-
reviews?count=10&after=rev-xxxxxxxb-yyyy-xxxx-yyyy-xxxxxxxxxxxx"
    }
  }
}

```

functionality available, as the API client changes configurations through the earlier discussed event types. Invite rules however can be used to define exactly when an invitation is sent out to the customer. So, e.g. a specific time can be defined when an invite is to be sent after the event has been triggered. Invite rules can be queried and posted via `/invite-rules`. Specific invite rules can be queried, updated and deleted via `/invite-rules/{id}`. Invite rules are associated with the entire account and therefore manipulating them does not necessarily require a channel ID. However, invite rules for specific channels can also be configured. All invite rules for a specific channel can be queried via `/channels/{channelId}/invite-rules` and updated via `/channels/{channelId}/invite-rules/{inviteRuleId}`. Here, the path parameter name is now "channelId" instead of "id" which is inconsistent. An invite rule is always linked to a respective questionnaire template. In this template, the questions that are sent to the customers in the review requests are defined. They are so-called "blueprints" for questionnaires. Templates can be accessed via `/templates`. Also, one specific template can be queried with `/templates/{id}`. Furthermore, templates can be filtered either by template ID (GET request to `templates/{id}`) or by a template ID and a locale with a GET request to `templates/{id}/locales/{locale}`. Template metadata that is linked to a channel can be queried with the help of `/channels/{channelId}/template-metadata` and deleted via `/channels/{channelId}/template-metadata/{key}`.

Furthermore, questionnaires can be rendered which returns a questionnaire as a JSON object. This function can be invoked with a POST request to `/questionnaire-templates/render`.

Reviews, similar as in the old API version, can be queried with a GET request to the endpoint `/reviews`. Parameters for filtering can be attached, e.g. the date and rating of the reviews. Furthermore, a single specific review can be queried with a GET request to `/reviews/{reviewId}`.

A veto can be submitted with a POST request to `/reviews/{reviewId}/vetos` if a specific review is found inappropriate.

Also, aggregated ratings can be requested via `/channels/{channelId}/service-reviews/aggregate-rating`. Average ratings over different time spans etc. are returned.

Lastly, questionnaires, similar to the review request endpoint in myTS, can also be accessed via hyperlinks that can be embedded in emails etc. by the API consumer themselves. So, Trusted Shops does not send emails to the customers through this endpoint. Questionnaire links can be created with a POST request to `/questionnaire-links`. Here, the channel identifier is embedded in the payload object.

Table 4.1 shows a comparison between similar endpoints in myTS and eTrusted.

4.2 Documentation

The old platform myTS uses a Swagger-UI-like interface to demonstrate the API. The user interface is enriched with some descriptions. The API user can actually execute queries by either just requesting the publicly available resources or by providing credentials to the webpage for restricted resources. Furthermore, there are inconsistencies in the documentation style. So, e.g. the dropdown item for the endpoint of product reviews is named “product_reviews: Product reviews”, whereas “reviews: Shop reviews” is missing the word “shop” before the colon. “Shop quality indicators” and “Shop benchmarks” show the same inconsistencies.

On the other hand, the eTrusted documentation was created with the help of ReDoc[5], a tool to customize OpenAPI documentations. The documentation is divided into a section about API concepts such as pagination and locales and a section reporting the actual endpoints like channels, reviews, and events. More context on the use of the principles and resources is provided.

Overall, the combined specification (see chapter 5.3) for eTrusted is with ca. 3400 lines about twice as big as the one for myTS with only around 1700 lines - compared as YAML files.

[5] **https://github.com/Redocly/redoc**.

Table 4.1: Comparison of the domains and resources in myTS and eTrusted

Resource in myTS	**Resource in eTrusted**	**Comparison**
Shops, Retailer Shops	Channels	In myTS, the tsId that identifies shops and retailer shops is used to manipulate every other resource as well. In eTrusted however, channels can represent shops but also different language regions etc. and therefore act as a filter that is not always necessary.
Reviews	Reviews	The endpoints for customer reviews are similar in both versions.
Trigger (Review Collector)	Invite Rules, Event Types	With the help of the trigger endpoint in myTS, the API consumer can request Trusted Shops to send out emails to the customers. In eTrusted however, review requests are configured with invite rules and event types. Events such as a checkout trigger invites. With the help of invite rules, invites can be configured so that e.g. emails are only sent at a specific time.
Review Request	Questionnaire Links	In myTS, links to review submission forms can be requested that can then be embedded in e.g. an email by the API consumer themselves. The questionnaire links endpoint in eTrusted serves a similar purpose and returns a link to a rendered questionnaire.

5 Research design of the case study

This chapter gives an overview of the research design of our case study. The main part of the case study consists of three steps: qualitative data analysis of interviews with Trusted Shops developers, analysis of (anti-)patterns, best practices, and API design rules through manual search, and tool-supported analysis.

A high level flow diagram of the entire study design can be seen in figure 5.1. The first step is to decide on research questions (cf. chapter 1). The qualitative data that comes mostly as the transcripts of the expert interviews is analyzed with the help of grounded theory, cf. [1], [43]. The interviews and the grounded theory coding and analysis steps are conducted simultaneously. So, while there are still new interviews scheduled, analysis of the results of finished interviews can already be started. Literature about the topic should not be reviewed until the most part of the qualitative data analysis has been conducted, see [43], [1]. Otherwise, this might lead to biases in the analysis as this could make it harder to come up with "a completely new core category that has not figured prominently in the research to date" [1]. In this case, literature about the quality of Trusted Shops APIs does not exist, but this principle is still important for the manual detection of (anti-)patterns and the tool-supported evaluations. So, these steps are done after the interviews have already been conducted to keep the qualitative data analysis free of bias in this regard. Both the manual and the tool-supported analysis can be conducted simultaneously. Lastly, with the help of the findings of the earlier analyses, we propose how to improve the new API specification. Therefore, e.g. the results of the (anti-)pattern analysis and the tool-supported evaluations can be considered to fix possible anti-patterns and violations against best practices. Also, we take the results of the qualitative data analysis into consideration.

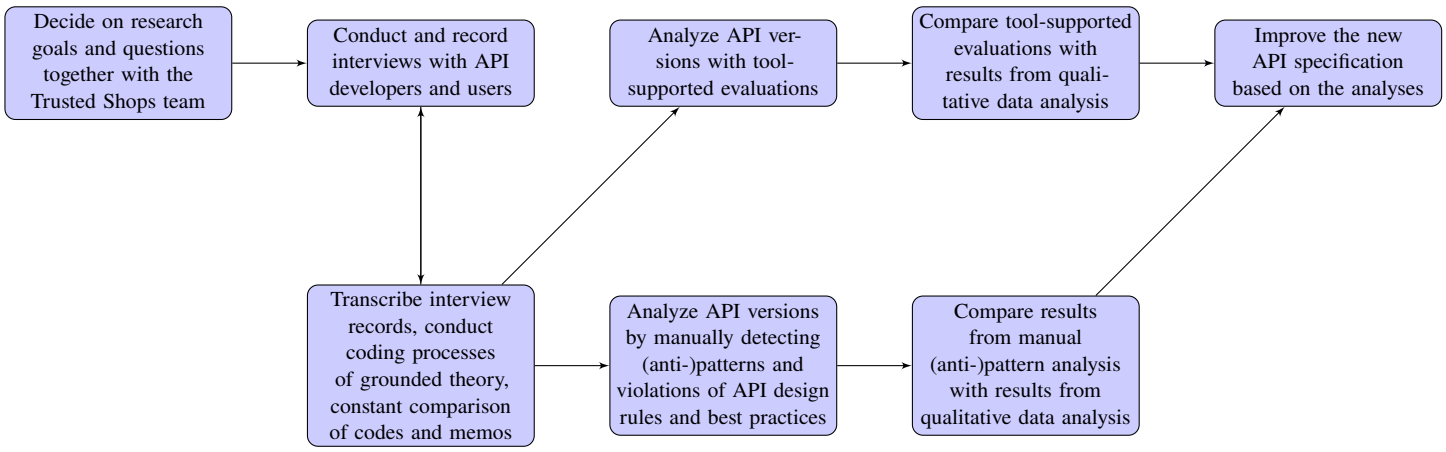

Figure 5.1: High-level design of the conducted case study

Table 5.1: List of interviewees

Interviewee ID	Years of professional work experience	Years of work experience for Trusted Shop	Role at Trusted Shops	Role in the case study
1	12	2	Software developer	API user
2	12	2	Software developer	API user
3	9	2	Software developer	API user
4	12	6	Software architect	API maintainer
5	7	7	Software developer	API maintainer

5.1 Qualitative data analysis

This case study is conducted with the help of semi-structered interviews (see [21], [40]). We interviewed five developers working for Trusted Shops, illustrated in table 5.1. Three of these developers represent the API client point of view, as they were not involved in the API design process and mostly know the new API version eTrusted from interacting with it as a client. The other two developers represent the API maintainer side, as they were more involved in designing and maintaining the new API. All interviewees are familiar with both the old API version and the new API version (see chapter 4) as the research questions of this work are focused on migration aspects. Each interviewee has at least seven years of professional work experience, so they can be seen as experts in their fields. The average work experience is 10.4 years. Three interviewees have worked for twelve years in their discipline. All interviewees have worked for Trusted Shops for at least two years. With seven years, interviewee 5 has been working for Trusted Shops for the longest time of all interviewees. So, they all have sufficient experience at Trusted Shops with respect to this case study.

Stol et al. [43] compare Glasserian, Straussian, and constructivist grounded theory that differ in their ways of coding and findings of theories. In the following, an overview of the applied processing techniques is given and the reasoning is discussed. We mostly applied constructivist grounded theory techniques.

Initial coding is used as a first step to organize the qualitative data, in our case mostly interview transcripts. We processed the transcripts line by line so that one concept can be found for each line. This step is similar in all three grounded theory variants.

After all of the interview transcripts have been initially coded, **focused coding** follows: One or more core categories are extracted from the initial codes. Therefore, the codes are categorized by their importance, relevance, and connections to other codes and categories. This is similar to Straussian axial coding and Glaserian theoretical coding. However, Glaserian theoretical coding is

the last step in this version of grounded theory after selective coding, whereby mostly a single core variable is to be discovered in order to build a core theory. Constructivist focused coding allows multiple categories.

Lastly, the relationships between the discovered categories are defined. We did not implement Straussian selective coding but the constructivist **theoretical coding**. As the distinctions between the two API versions are complex and many design properties are different, there cannot be expected to be one main issue with either one of both versions that is avoided by the other version. Different influences of different design decisions are too complex. Many of the design properties have to be discussed and compared, with potentially varying "winners". So, theoretical coding is more appropriate, as this gives the chance to find more than one core theory that explain the quality perception of both API versions.

Writing **memos** while analyzing the data and creating the codes is important in all grounded theory variants. Whatever conclusion or thoughts come to mind about interconnections between codes and theories can therefore be written down and saved in memos.

When new codes are found, all codes and other findings as e.g. memos are in **constant comparison** to each other. So, the new codes can either be assigned to already existing codes and categories or uniquely new codes can be created. Also, new interconnections and (core-)theories in the data can be found and old interconnections, theories, and codes can become obsolete. These last two principles are necessary in all grounded theory variants.

5.2 Analysis of (anti-)patterns, design rules, and best practices

In this section, we describe the manual search for common (anti-)patterns, design rules, and best practices in the API specifications of the two API versions, for the results see chapter 6.2. The findings are then compared to the results of the qualitative data analysis (cf. chapter 7.1). This is done to find out how expressive anti-pattern analysis is and if it comes to similar conclusions as the qualitative data analysis.

This next part describes the search for adequate (anti-)patterns, design rules, and best practices and the following evaluation and interpretation of the findings. Analysis is done in a mostly qualitative manner. We search for violations against patterns and rules and for occurrences where positive patterns are implemented as well as "missed chances" where a pattern could have been applied.

Criterions for relevant (anti-)patterns, best practices, and design rules are listed below.

- (Anti-)patterns must address the service interface/(RESTful) API specification of a web service.
  So, (anti-)patterns that take the underlying code base or different documents other than the specification into account violate this criterion. Also, (anti-)patterns that analyze operational properties of a running system, e.g. "Rate Limit" [42], are not considered.

- (Anti-)patterns must fit the cause and domain of the API. A violation of this property would exist e.g. in the "Wish List" pattern [42] as the response objects of both analyzed API versions are generally small with few attributes. With so few attributes, adding a wish list to the request just in order to implement this pattern is not necessary.
- (Anti-)patterns must be associated with either maintainability or usability or a respective sub-property, as these quality attributes are most important to our research goals.
- (Anti-)patterns must address a single publicly available API documentation.
  E.g., (anti-)patterns that address the dataflow within a microservice architecture are not considered (e.g. Megaservice or Nanoservices [4]). The number of operations per service cannot be determined in this work. Also, cyclic dependencies between services cannot be detected.

The papers are searched via Google Scholar with a combination of keyword search strings. Further referenced papers are considered as well.

The **keywords** are: REST, restful, antipattern, anti-pattern, pattern, service interface, OpenAPI, best practice, best practices.

The (anti-)patterns that are described in the literature have an impact on different software quality attributes. Those affected attributes should be described in the given source paper or - in case the pattern is originally described in another work - the possibly mentioned original paper.

As a first step, we discover the (anti-)patterns and violations of API design rules and best practices through manual analysis. Then three RESTful API experts from the Trusted Shops team validate the findings. Therefore, we present our findings to the experts in a virtual workshop. They are asked to give an importance rating on a standard Likert scale. So, they are asked to decide if they agree that a specific rule is important by either giving one point, "I disagree", two points, "I partially disagree", three points, "I neither agree nor disagree", four points, "I partially agree", or five points, "I agree". This is done before they receive the list of occurrences of these (anti-)patterns in order to minimize the possibility of biases in favor of the new API version. They are encouraged to discuss the ratings with each other. Then they receive the list of findings on these (anti-)patterns. This way, the results can be validated and discussed to ensure that they are valid. However, only patterns and anti-patterns for which findings could have been made out beforehand are presented to the experts. Through this pre-filtering, it cannot be assured that there are actually no (anti-)pattern occurrences for the other instances. Also, it cannot be assured that we found all occurrences of a specific (anti-)pattern that we actually presented to the experts. We chose this study design, as it is less effort for the experts, but the positive findings can still be considered validated.

We examine the specifications for universally applicable patterns, design rules, and best practices together. If those rules are violated, it can be seen as a bad practice that leads to the worsening of a specific software product quality property compared to abiding the respective pattern or best practice. Most of these patterns and principles are described in [35]. Furthermore, Kotstein and Bogner [25] conducted a delphi study with all of these rules and even more in order to find out which rules are perceived as important. Only those rules with a high importance rating from that list are therefore considered here. However, similar rules as the rules with low importance ratings can still be considered if they are identified in other sources as well. The naming of these universally

applicable rules as "best practices" is not consistent in the given literature. E.g., Brabra et al. [6] list the inappropriate use of HTTP methods as an anti-pattern whereas Petrillo et al. [35] label this as a best practice. Furthermore, some authors describe "best practices" by our definition with a dualistic pair of a pattern and an anti-pattern, e.g. Palma et al. [30]. Only occurrences of the anti-patterns (violations of the patterns and best practices) are listed. Instances where the respective practices are followed are not displayed. The lack of abidance to a rule is more expressive of the software product quality than the occurrences where a universally necessary pattern is implemented. Sticking to those universally applicable practices and patterns can be seen as the baseline, so the violations are more interesting to this study.

Furthermore, "standalone" service-interface anti-patterns are analyzed. Those anti-patterns do not necessarily have a respective positive pattern but are defined by themselves. Occurrences of those anti-patterns can be more expressive of negative impacts on software product quality than findings of the dualisticly defined anti-patterns. These anti-patterns are on a higher abstraction layer and can therefore give feedback about the structural quality of an API specification. Here, only the detected occurrences of the anti-patterns are listed and not statements where an anti-pattern could have been implemented but was not. Not implementing an anti-pattern can as well be seen as the baseline.

Lastly, we examine the specifications for "standalone" service-interface patterns. Those patterns do not have a respective anti-pattern but are defined by themselves. These patterns are very domain-specific, so implementing them is not always recommended. But to analyze the software product quality of the present RESTful API documentations, detected patterns as well as detected sections where this pattern could have been implemented but was not are listed. As these patterns are not universally applicable, careful consideration must be applied to finding occurrences of "missed chances" of implementing a respective pattern. Not applying a pattern can be seen as a negative factor on the software product quality. Implementation of such a pattern might result in a better RESTful API. This negative impact has to be assessed carefully, however. The number of "missed chances" to implement a specific pattern does not necessarily give conclusions about the software product quality of a specification, for example. Therefore, relevance to our specific RESTful specifications has to be argued carefully. So, implementing a specific pattern does also not necessarily lead to better software product quality. These "missed chances" are mostly associated with **RQ5**. Many of the reviewed (anti-)patterns describe actions and properties within a microservice architecture and are therefore not applicable to our case. Often, interactions between services are discussed and not interactions between the publicly available RESTful API specification and the API user. Information on maintainability and usability of the system's API is therefore limited.

Not all of the examined publications report the proposed (anti-)patterns in such an extensive manner as described in chapter 2.5. The effects on the software product quality attributes is of most importance here, as this can give the most conclusions on the associated research questions. Bogner et al. [4] provide a list of anti-patterns collected through other publications, for example. The referenced sources then are examined to find the missing information. Nevertheless, also primary sources do not always stick to the recommended reporting style, e.g. Stoll et al. [44] report the patterns rather informally. Many of the collected papers about novel (anti-)patterns provide the documentation in a similar formal manner however, see e.g. Zimmermann et al. [51].

Furthermore, we collect quantitative data on the software product quality of both API versions during the expert interviews, see chapter 5.1. This data is then used to compare the findings through manual (anti-)pattern analysis with the expert opinions.

5.3 Tool-supported analysis

This section describes the search for RESTful API evaluation tools. In a next step, the findings of the tools, see chapter 6.3, are then compared to the outcome of the qualitative data analysis, see chapter 7.2.

Overall, we identified four tools, three of which are rule-based evaluation tools. The fourth tool is a metric-based evaluation tool. Two tools, Zalando's Zally[1] and IBM's OpenAPI validator[2] were detected through OpenAPI.tools[3] that lists both commercial and non-commercial tools for OpenAPI specifications. Also, we used the prototype described in [25]. Lastly, we used a metric-based evaluation tool called RAMA[4] to analyze both API versions, cf. Bogner et al. [5]. This tool implements ten different metrics. It contains seven complexity metrics, two cohesion metrics, and one size metric.

We found more tools, e.g. the soda-w tool[5] by Palma et al. [32] that can be used for detection of linguistic anti-patterns. We could not compile this project because necessary jar-files seemed to be missing. Furthermore, we tested out CloudLex[6] by Petrillo et al. [36]. This tool is not easy to use and has to be customized for every new OpenAPI specification. There is no real documentation attached but the reports for already processed specifications seem to be rule-based such as the other tools we used. Anti-patterns that cannot be easily implemented with a rule-based approach but need linguistic context, e.g. non-pertinent documentation (see [36], [30]), are not implemented.

Both Zalando's Zally and IBM's OpenAPI Validator are documented as rule-based linters. They come with a custom set of already implemented rules. Zally implements the "Zalando RESTful API and Event Scheme Guidelines"[7]. For the OpenAPI Validator, rules based on the OpenAPI specification by the OpenAPI Initative for version 2[8] and 3[9] are implemented.

Zally offers a web-based frontend and a Kotlin based backend, whereas the OpenAPI Validator - implemented in Javascript - can be used as a node module or via command line interface. Both linters can be extended with a custom ruleset and support OpenAPI 2 and 3. The two tools report only violations of the defined rules. Both are not promoted as API evaluation tools but rather as extendable rule-based linters. However, they have a great number of default rule implementations.

[1] **https://github.com/zalando/zally**.
[2] **https://github.com/IBM/openapi-validator**.
[3] **https://openapi.tools/**.
[4] **https://github.com/restful-ma/rama-cli**.
[5] **http://sofa.uqam.ca/soda/**.
[6] **https://github.com/Spirals-Team/CloudLexicon**.
[7] **https://opensource.zalando.com/restful-api-guidelines/**.
[8] **https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md**.
[9] **https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md**.

Also, the rules are not explicitly associated with either usability or maintainability. Therefore, the quality attributes are assigned by us and validated by three RESTful API experts. Also, they are asked to give an importance rating on a standard Likert scale. They are encouraged to discuss with each other and give one rating. This is done together with the validation of the manual (anti-)pattern findings, see chapter 5.2. As these two tools only report warnings and errors according their specific rulesets, we did not exactly know beforehand which rules are implemented. So, we already manually checked some of the rule violations reported by the tools. Nevertheless, these rules cannot process higher-level linguistic context. So, the "restricted" node in URIs for myTS is flagged as a violation even though this is not a singularized noun but a verb, for example.

The specification for eTrusted has one super reference file where the other files are referenced. We merged all these files into one big specification with the help of Swagger Combine[10] after each file has been converted to OpenAPI 3.0 with the help of the OpenAPI editor[11]. We also removed references to missing files, e.g. description files that are not public, with the help of the OpenAPI editor. Furthermore, the old API's specification has enumeration fields that are not valid and had to also be removed. As the API specification for myTS is in form of Swagger version 1.2, in a first step we converted it to OpenAPI 3.0 with the help of the API Spec Converter[12]. The specification for myTS does not offer a proper reference file so we created a custom reference file and merged all API files with the help of the node package OpenAPI Merge CLI[13].

Furthermore, we use the quantitative data we collected during the expert interview, see chapter 5.1, in order to compare the tool evaluations with the expert opinions.

[10] **https://www.npmjs.com/package/swagger-combine**.
[11] **https://editor.swagger.io/**.
[12] **https://www.npmjs.com/package/api-spec-converter**.
[13] **https://www.npmjs.com/package/openapi-merge-cli**.

6 Results of the case study

In this chapter, we present the results of the analyses. In a first step, we show the results of the qualitative data analysis. Then we present the findings of the manual search for (anti-)patterns and violations of API design rules and best practices. Lastly, the results of the tool-supported evaluations are listed, divided into a section about rule-based tool analysis as well as metric-based evaluation.

6.1 Qualitative data analysis

An abstract overview of the grounded theory findings in form of a graph can be seen in figure 6.1. Core theories are marked as red ovals. Categories are represented by the blue boxes. Categories are hierarchically connected with black arrows if we found sub categories that were logically connected to a super category. Core theories can be connected with red arrows.

One of the core theories is that the new API platform eTrusted is better than the old API version myTS. This core theory is complemented by another core theory, “Migration is rewarding”. A third core theory is displayed as “Problems with operations” that represent the current difficulties with running two different API versions in production. The other two core theories provide an implication for “Migration is rewarding”, displayed with red arrows. These theories could also be hierarchically merged into one core theory where “Migration is rewarding” is the only core theory as the root node and “eTrusted is the better API” and “Problems with operations” are hierarchically subordinated theories or categories. This would be too coarse-grained however, as “eTrusted is the better API” and “Problems with operations” can be seen as independent theories that contribute to the other core theory “Migration is rewarding”. Numerous categories and sub-categories make up these theories, they can be seen as independent forces within the entire grounded theory.

The new API version eTrusted features better usability and better maintainability. These categories can be found by organizing the codes. Better usability however is caused by more standards, a better documentation, and better user error protection. Better maintainability is mostly associated with more standards. “Size of API client’s company” is a category of the core theory “Migration is rewarding”. Finally, “Operating API switch” and “Shut down old project” make up the third core theory of “Problems with operations”. Furthermore, we identified positive codes about the new features and authorization system. However, this is more relevant to the functional suitability and security attributes of the software product quality, whereas this work is concerned with the usability and maintainability of both API versions. Therefore, such codes are not displayed here.

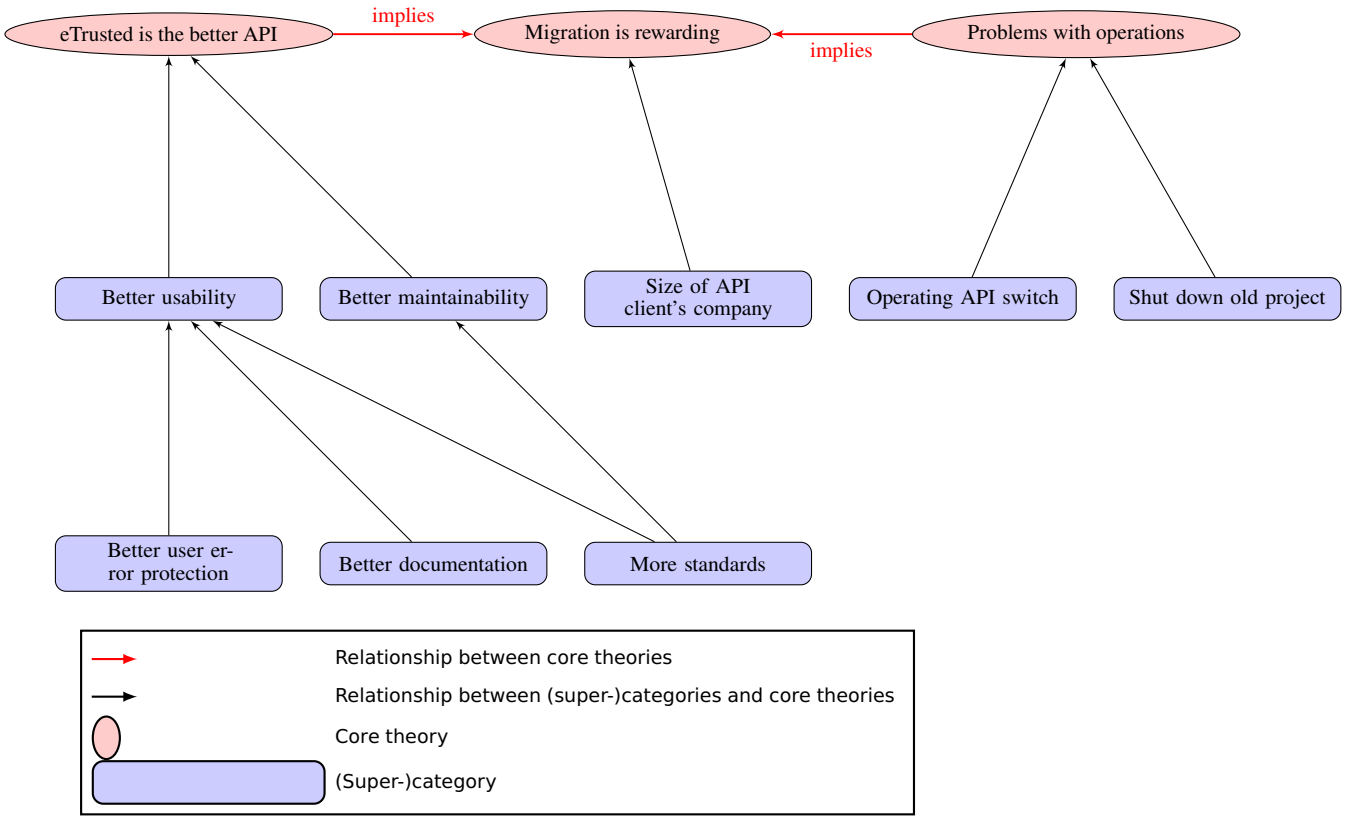

Figure 6.1: Final core theories and respective code categories found through grounded theory

Concerning **RQ1**, the interviewees generally said that the new API version is an improvement over the old API version. So, the usability of the new API version eTrusted is perceived as an upgrade. One big reason for this seems to be the documentation.

Documentation which is relevant for the usability of an API (**RQ1**) is also relevant for **RQ5** as an integration- and migration-related artefact. Whereas the documentation for the old version is seen as "crude" by interviewee 1 or even non-existent ("there is no old documentation", interviewee 3), the documentation for eTrusted is perceived as better structured by interviewee 1 and 2. Together with the help of the Trusted Shops team, the documentation was perceived as sufficient for API clients to integrate the API into their systems, according to interviewee 3. Also, the provided code samples for building requests for each operation were praised by interviewee 3 and 5. Interviewee 4 - who cannot be seen as an API client but acts as a maintainer in this study - noted that further context on how to interact with the API is provided by the new documentation compared to the old one. Nevertheless, interviewee 4 and 5, who are API maintainers, think that the new API version is missing a test interface such as Swagger UI. The old version has a few public resources that can be invoked with such an interface.

Moreover, interviewee 5, an API maintainer, described the documentation for eTrusted as "even more complicated and not necessarily better". Also, interviewee 3, 4, and 5 criticized that there was no further context on how to use the API as an API consumer, e.g. by providing actual business use cases. The reason for this might be that - as interviewee 2 claimed - Trusted Shops developers

do not know how the API consumers actually use the API. Whereas interviewee 4 and 5 wish to have those use cases for their customers, interviewee 3 wants more use cases for their development team themselves. There are also no such use cases reported in the old documentation for myTS as interviewee 5 remarked. Also, interviewee 3 criticized missing features, e.g. an authorization system is not yet implemented in eTrusted.

The new API version eTrusted itself is seen or was at least built for the purpose to be "easier and more convenient", said interviewee 3. Interviewee 4 said that the usability was better due to clearer "cross-cutting concerns" in the entire API. The API documentation of eTrusted provides a separate section about recurring concepts such as pagination and locales, see chapter 4. Furthermore, all five interviewees said that new features, e.g. custom templates for review invites, improved the new API version. Nevertheless, interviewee 5 noted that some use cases did not really fit the RESTful architectural style, e.g. "posting" review invites. Also, the HATEOAS concepts (see chapter 2)) were not fully implemented. Both points of criticism are relevant for the old and new API.

Especially the error handling of the new API version is seen as superior over the old version. Interviewee 1 said that the old API version myTS often returned empty responses if an internal error occurred. This was handled better with the new API version, they said.

Interviewee 2 also claimed that guaranteed delivery was implemented in the new versions, whereas in the old version requests would have been unanswered in a case of system failure. Nevertheless, Interviewee 2 and 4 said that normally if an API was used the right way no errors would come up once the API client has been fully integrated.

The most problems were seen with the bad maintainability of the old API version myTS. The most opinions were rather on the lack of maintainability of that version than on the respective better maintainability of the new version. Interviewee 3 claimed that maintainability "is of course hard as it is an old monolith", for example. Moreover, interviewee 4, a software architect and maintainer for the new API version, complained that the documentation would not automatically update when the underlying services for the old API version myTS change. Therefore, it cannot be guaranteed that the documentation adequately represents what is implemented and exposed, which can also be seen in chapter 6.2. This impairs maintainability badly. Interviewee 1, acting as an API consumer in this study, and interviewee 4, an API maintainer, said that the new API platform could be extended with new features more easily which improved maintainability. Nevertheless, Interviewee 1 and 4 said that communication was harder as due to the microservice architecture the project now was split up over many teams. Also, interviewee 4, a maintainer, said that this made it harder to enforce consistency within the API specifications with many teams involved. They suggested better tool support.

The old API version myTS lacks of standards and guidelines. So, interviewee 1 said that "if any [guidelines] exist, they have not been implemented". Interviewee 1 and 4 claimed that eTrusted met broadly used common practices. Interviewee 4 said that external APIs implemented those rules better as internal APIs might sometimes violate these.

The integration process, associated with **RQ2**, is mostly seen as simple but differs for different API clients. If an API client is too small to have an own IT department, they might have trouble migrating to the new API version, especially with the new oAuth2 authorization technique which

Table 6.1: Overview of extracted categories and codes associated with either myTS or eTrusted

Category	myTS		eTrusted	
	Number of positive codes	Number of negative codes	Number of positive codes	Number of negative codes
Maintainability	0	7	7	4
Usability	2	5	7	4
Error handling	0	2	1	0
Documentation	3	4	5	5
Standards, Patterns, Guidelines	0	3	7	1
Overall	5	21	27	14

was described as a “paradigm change” by interviewee 1. Interviewee 2 and 5 also think that this step might be hard for the API consumers. Bigger clients however should not have much trouble with the immigration process: Interviewee 1 and 2 claimed that it was just a matter of invoking another API with different URLs.

Interviewee 5 said that any new integration or migration of an API holds “invisible cost” for the API consumer as to understanding new wordings, concepts, and approaches. So, the entire API introduced a “change of concepts” according to interviewee 5. In contrast, interviewee 4 sees less problems from this point of view as the “concepts stayed the same”. So, interviewee 4 also sees a form of backward compatibility in regard to the comprehension of concepts. Just the naming had been changed (“There are still reviews. Shops become channels.”).

Asked on improvement potential of the migration process (see **RQ5**), the interviewees have similar ideas. Interviewee 1 and interviewee 3 both suggested a system where the API user would not even have to integrate the API themselves. They proposed a “plugin” (interviewee 1) which can be exchanged for each new API version. Interviewee 2 had a similar idea: they suggested a service where the API client uploads their data and Trusted Shops would do the integration for them.

Furthermore, both interviewee 1 and 5 proposed a sandbox system where API clients could test the API and their integration in a non-productive environment. Also, as interviewee 4 and 5 criticized missing context in form of e.g. use cases in the documentation, interviewee 5 proposed customer surveys in order to find out how they use the API. So, actual use cases could be identified with real-world customers.

It becomes clear that the Trusted Shops team wants their customers to migrate as soon as possible as they only offer the option to use the old API for an “interim period” (interviewee 4). The switch that is operated by the Trusted Shops team used for this task caused a lot of overhead (interviewee 1 and 2). Hence, interviewee 2 and 4 said that they wanted the customers to migrate to the new API version.

Table 6.1 shows the respective number of codes by category for myTS and eTrusted. For the interviewee IDs of interviewees who contributed to the particular positive and negative codes, see table A.1. The number of negative codes for e.g. usability are surprisingly close for both versions. This can partially be explained due to the fact that a couple of negative codes were found through questions on eTrusted that also could fit myTS but the old version was not explicitly mentioned.

Table 6.2: Overview of extracted categories and codes associated with both myTS and eTrusted

Category	Number of positive codes	Number of negative codes
Migration	5	13
Customer	0	3
Operations	2	4
Overall	7	20

Table 6.3: Quantitative expert quality ratings

Interviewee ID	myTS		eTrusted	
	Usability rating	Maintainability rating	Usability rating	Maintainability rating
Interviewee 1	7	7	10	10
Interviewee 2	10	7	10	9
Interviewee 3	8	7	9	7
Interviewee 4	8	3	8	8
Interviewee 5	7	5	7	7
Median	8	7	9	8

So, one interviewee criticized missing use cases in the documentation of the new API version, for example. However, there are no use cases in the documentation of the old version either. Also, this might be caused by collecting too few data points as we only interviewed five developers. More categories were found that are not shown in this table. Codes concerning security and functionality go beyond the scope of this work that focuses on the usability and maintainability of the both API versions, for example. Only categories that are relevant for each API version separately are displayed. So, codes from categories such as migration, communication, and operations cannot be assigned to this style of table with positive and negative codes for each API version. An overview of these codes can be seen in table 6.2. The categories and the number of generally positive and negative codes can be found. For the full table with the IDs of interviewees contributing to these codes, see table A.2.

Overall, there are 27 positive codes about eTrusted and only five about myTS. Also, there are seven more negative codes for myTS than for eTrusted, specifically 21 versus 14. Nevertheless, eTrusted worsened in a few attributes, e.g. as already discussed, interviewees missed the possibility to try out the API in a Swagger-UI-like environment.

We also asked the interviewees to rate the usability and maintainability of both API versions on a one to ten point scale. A rating of ten points represents the best rating, whereas a rating of one point represents the worst. The expert ratings can be seen in table 6.3. The median ratings for both usability and maintainability are one point higher (with 9.0 and 8.0 points) for the new API version eTrusted compared to myTS. This result fits to the outcome of the qualitative data analysis, although the difference of the ratings could have been expected to be higher. Nevertheless, the new version eTrusted received better results than myTS as also in the qualitative data analysis.

In summary, it becomes clear that the new API version is mostly seen as a progress. Disadvantages are perceived in maintenance and operation processes that naturally become harder with time. Better tool support can help with this issue. Nevertheless, a few structural problems about the new API version are also criticized. Interviewee 5 sees the complexity of the new API version on the same level as the complexity of the old API version myTS, for example. This might be caused by more extensive functionality as eTrusted offers more features than myTS.

6.2 Analysis of (anti-)patterns, design rules, and best practices

This section describes the results of the manual search for (anti-)patterns in both API versions. The old API version myTS violates ten of the 32 examined basic API design rules and best practices, see table 6.4. The new API version eTrusted only violates nine of these examined rules. Only rules where at least one violation is found are displayed. For the entire list of examined design rules, see table A.3. Violating those rules can be seen as an impairment of the respective quality attribute, as ideally these rules should always be observed. So, the amount of violated rules can give a good conclusion on the software product quality of the RESTful API specification.

Overall, the new API version violates less of these principles. There are no violations in eTrusted that cannot be found in myTS. But myTS contains more violations that cannot be found in eTrusted. All path parameter names in both version violate the rule that only lowercase letters should be used. E.g.,“tsId” for myTS and “channelId” are written in camel case. As these variables are just placeholders for integer values, the actual URIs will not have upper case letters in them.

There is to say that the principles on HTTP status codes cannot be examined well just by inspecting the specification. The servers might e.g. respond with a “200” status code which is just not reported in the API specification. Only error codes can be found in myTS and no codes starting with “2” except for the endpoint product reviews comments. Also, the new API specification of eTrusted does actually report status codes starting with “2”. Therefore, it can actually be seen as intended to not only report error codes but also report intended codes furthermore. Not using the “200” status code at all does not violate the rule ”200 (“OK”) must not be used to communicate errors in the response body ”, see table A.3. However, we found instances for both API versions, wherein the codes “201” and “204” were not used. Not using “201” can partly be explained, as also criticized by a maintainer in chapter 6.1, that sending a post request to trigger an invite request is not a typical use case of REST. This applies to both API versions. Nevertheless, we also found the same issue for shop reviews comments, where a “201” actually could be returned. But as already discussed, most of the endpoints in myTS do not return “20X” codes. “204” also is only used in product reviews comments in myTS. It could have also been applied to e.g. GET requests for shops and product reviews. The new API version eTrusted does not return a 201 status code after a POST request has been made to create invite rules. Also, e.g. a GET request to the channels endpoint does not list a 204 response, whereas some DELETE and UPDATE requests do. “403” is used in every endpoint in myTS expect for shops and retailer shops. In eTrusted, this code is missing for the endpoints of

Table 6.4: Detected violations of API design rules and best practices

API design rule or best practice	Violations in myTS	Violations in eTrusted	Associated quality attributes	Sources
Lowercase letters should be preferred in URI paths	all path parameters names	all path parameters names	usability	[31], [6], [35], [39]
A plural noun should be used for document names	POST review collector	POST render	usability	[31], [6], [39], [35], [30]
201 (“Created”) must be used to indicate successful resource creation	POST review request, review collector, shop reviews comments	POST invite-rules	usability	[6], [35], [37]
204 (“No Content”) should be used when the response body is intentionally empty	GET shops, shop reviews, shop quality indicators, benchmarks, retailer shops, product reviews	GET channels, invites, invite-rules, events, event-types	usability	[6], [35], [37]
403 (“Forbidden”) should be used to forbid access regardless of authorization state	shops, retailer shops	events, `events/{eventRef}`	usability	[6], [35], [37]
415 (“Unsupported Media Type”) must be used when the media type of a request’s payload cannot be processed	all POST operations	all POST operations	usability	[26]
500 (“Internal Server Error”) should be used to indicate API malfunction	all operations	all operations except POST events	usability	2[6], [35]
Pertinent documentation	GET shop reviews: pagination description missing	-	usability and maintainability	[30]
Links in Representations	GET shops to GET shop reviews etc., pagination	GET channels to GET invites etc.	usability	[6], [44], [39], [37]
Forms in Representations	missing	missing	usability	[37]

`/events` and `/events/eventRef`. “415” is not applied at all in both specification. It could be applied to all POST operations. The old API version myTS does not list the code “500” in its entire specification. The new API version eTrusted lists it only for possible POST operation responses.

Also, the old version myTS does not contain any “Links in Represenation”, whereas eTrusted contains links in pagination response objects. Both versions do not provide further hyperlinks from e.g. GET operations to a respective PUT or DELETE operation. Such hyperlinking would be possible in many responses, e.g. from GET `channels/{id}` to GET `channels/{id}/invite-rules` in eTrusted or from GET `/restricted/v2/shops/{tsId}` to GET `GET/restricted/v2/shops/{tsId}/reviews` in myTS. This can be seen as a violation to level 3 of Richardson’s maturity model (see chapter 2.4.3). As the new version, eTrusted, provides links at least in the pagination response objects, it follows the principles of HATEOAS better than the old version myTS. When pagination is used, the client can just follow the returned links in order to browse through the response items. This violation addresses usability as an abstract property of “navigational help” [37].

The old API version myTS shows a violation of the “Pertinent documentation” pattern at the GET operation for shop reviews (`/restricted/v2/shops/{tsId}/reviews`). The other pageable operation GET product reviews (`/restricted/v2/shops/{tsId}/products/reviews`) offers a minimal description explaining that the resource supports pagination. This violation impairs the usability and reusability, a sub quality attribute of maintainability, of the API specification [30].

Also, XHTML forms as another returnable media type is not used in either API version. Nevertheless, it is questionable if this practice is meant as a best practice by our definition. Admittedly, the authors label their entire list as “State of the Art and Best Practices”. But still they rank this practice as “highly relevant” [37].

Most of those principles are met as we examined 32 rules. So, both API versions can be seen as sufficient in this regard.

Next, findings on service-interface anti-patterns are presented.

Occurrences of the examined service-interface anti-patterns can directly impair the software product quality of a RESTful API specification. Overall, the specifications have been analyzed for three anti-patterns, see table A.4. Only one of the examined anti-patterns could be found, deep path, cf. [44]. This anti-pattern occurs when unnecessary IDs as path parameters are listed in the URIs. This comes up in myTS in the GET and PUT endpoint for comments on reviews (`/shops/{tsId}/reviews/.{rvId}/comments/coId`), as the IDs rvId and tsId are not necessary to manipulate a specific comment with a comment ID coId. Furthermore, the same anti-pattern can be found for the product reviews comments operations DELETE and PUT for path `/shops/{tsId}/products/reviews/{productReviewUuid}/comments/{shopCommentUuid}`. As the product review IDs and shop comment IDs are universally unique IDs, it would be possible to shorten the URIs and just use the necessary identifiers. This would improve understandability as it makes URI calls unnecessarily complex. Therefore, usability that is the super-property of understandability defined in ISO 9126 (cf. chapter 2) and also learnability, a sub-property of usability in ISO 25010, are negatively affected.

Table 6.5: Detected patterns and missed opportunities to implement these patterns

Service-interface pattern	Occurrences in myTS	"Missed opportunities" in myTS	Occurrences in eTrusted	"Missed opportunites" in eTrusted	Associated quality attributes	Sources
State Creation Operation	POST shops/tsId/review-collector	-	POST invite-rules	-	usability and maintainability	[50]
State Retrieval Operation	-	GET invites	GET channels/id/invites	-	usability and maintainability	[50]
State Transition Operation	-	PUT invites	PUT invite-rules	-	usability and maintainability	[50]

Not adding these identifiers to the path would not be possible if e.g. the review ID was not universally unique. So, it would just be clear in combination with the shop ID which review is addressed exactly. Reviews would not be an independent resource. This is not the case, however.

The new API version eTrusted's analogous endpoint for customer reviews shows the same anti-pattern. So, channelId would not be necessary for customer reviews (`/channels/{channelId}/customer-reviews/{customerReviewId}`). However, getting all customer reviews, the channel ID is needed as the only ID in the path. Removing it when requesting a specific customer review might me confusing. The same holds for invite rules and template metadata, where requesting a single specific instance the channel ID could be left out. Also, both channelId and customerReviewId would not be necessary for vetos (`/channels/{channelId}/customer-reviews/{customerReviewId}/vetos`).

Both API versions are therefore negatively affected by this anti-pattern. The old API version myTS only shows two occurrences, whereas the new API version eTrusted shows four occurrences of this pattern. But the specification for eTrusted is essentially larger and therefore has more possibilities to apply this anti-pattern. So, this does not imply that eTrusted is actually worse than myTS in this regard. Also, both versions show no instance where this anti-pattern is not implemented where it could have been implemented. Therefore, it can be seen as a structural design issue. As both API versions only implement one of the three examined anti-patterns, they can be seen as qualitatively good in this regard.

The API versions have been analyzed for three service-interface patterns, see table 6.5. Both API versions show occurrences of the pattern "State Creation Operation". This pattern is useful for situations in which the API client wants to inform the API with new data without expecting a response right away, e.g. in long-running operations. Such an operation could be described as "in essence write-only" [50].

Both eTrusted and myTS make use of this pattern in a similar way: With the help of a POST request to `/shops/tsId/reviewcollector`, a new review request can be sent to myTS so that Trusted Shops sends out an email to the customer of the API client. POST requests to `/shops/tsId/reviews/requests` cannot be seen as State Creation Operations, as this way just links to review requests are directly returned. The API clients themselves then can send these links out to the customers. This is not considered a long running operation. In eTrusted, invite rules can be created which set the conditions when a review invite is sent to the customers (POST to `/invite-rules`).

The old API version myTS does not have a respective "State Retrieval Operation" [50], whereas eTrusted offers the retrieval of existing invite rules (GET `/invite-rules`) as well as invites (GET `/channels/{id}/invites`). So, this can be seen as a missed opportunity for myTS as an operation of requested but not yet sent out review requests is conceivable. This operation could then be used to request the current state of the just discussed "State Creation Operation". This also applies to the "State Transition Operation" [50] through which the state of a long running operation is updated. In eTrusted, the API client can change existing invite rules (PUT `/invite-rules`). The old version myTS offers no such operation.

Zimmermann et al. [50] claim that these three patterns are associated with compatibility, evolvability, learnability, and manageability which makes the outcome important for our case. Both maintainability and usability are therefore affected by these state patterns.

6.3 Tool-supported analysis

In this section, we present the results of the tool-supported evaluations. It is divided into a part about the results of the three rule-based linting tools and a part about the metric-based evaluation tool.

6.3.1 Rule-based tool analysis

As some references had to be deleted and merging and converting might falsify the specifications, all reported errors are also checked against the original specifications. Only the violations that can actually be found in the original specifications which are not caused due to merging or conversion are listed in table 6.6. Similar rule violations of the tools are reported together. For the complete list of violations of each rule for all tools, see tables A.5, A.6, and A.7.

The prototype presents its findings in a CSV file where only the number of valid, invalid, and unapplicable occurrences of each rule is reported. The tool finds actual rule validations in the rules for "Document Singular Noun", "Plural Noun Collection/Store Name", "Use hyphens", and "ContentType".

"Document Singular Noun" and "Plural Noun Collection/Store Name" check path items with the help of a language processing library if they are singularized or pluralized nouns. Therefore, the last element of a path is checked for a singular noun if no array is returned and for a plural noun if

Table 6.6: Aggregated reported errors and warnings

Rule	Tools	Violations in myTS	Violations in eTrusted	Associated quality attributes
204 responses do not include a message-body	OpenAPI Validator	2 occurrences, GET and DELETE product review comments	-	usability
Parameter names + property names + path segments + operationIds + ENUM values follow snake case convention	OpenAPI validator + Zally	all parameter names, e.g. tsId; all operationIds, e.g. retrieveReviews	all parameter names, e.g. channelId; all property names, e.g. createdAt; all path segments,e.g. customer-reviews; all operationIds, e.g. getTemplate; all operationIds, e.g. getReviews.; 53 out of 55 ENUM values, e.g. AUTOMATIC	usability
HTTP header fields follow Hyphenated-Pascal-Case	Zally	-	all HTTP header fields, e.g. token	usability
Hyphens (-) should be used to improve the readability of URIs	Protoype	6 occurrences, all path parameter names	4 occurrences, all path parameter names	usability
Arrays MUST NOT be returned as the top-level structure in a response body	OpenAPI Validator + Zally	-	4 occurrences, e.g. GET `/channels`	usability
Operation do not have non-empty ‘tags‘ array	OpenAPI Validator	all operations	-	usability
operationIds follow naming convention: operationId verb should be list, replace, add or create	OpenAPI Validator	5 OperationIds, e.g. retrieveReviews	12 operationIds, e.g. getTemplate	usability, maintainability
Required parameters appear before optional parameters	OpenAPI Validator	tsId after acceptLanguage at GET `/shops/benchmarks`	4 occurrences, e.g. id (UUID) after Channel-Id (channel name) in invites endpoint	usability, maintainability
Common path parameters are defined on path object	OpenAPI Validator	all operations	all operations	usability, maintainability
Schema + schema properties have a non-empty description	OpenAPI Validator	all schemas, e.g. dateiso8601, all schema properties, e.g. format	one schema property, invite-rules property rules	usability, maintainability
Server URL does not have a trailing slash	OpenAPI Validator	-	missing slash	usability, maintainability
Operations have a non-empty ‘operationId‘	OpenAPI Validator	-	5 occurrences, e.g POST `/events`	usability, maintainability
Security scope is used when defined	OpenAPI Validator	-	e OAuth2: [eTrusted]	usability
API Audience + API Identifier + API Meta Information is provided	Zally	missing	missing	maintainability
Numeric properties must have valid format specified	Zally	1 occurrence, property amount	1 occurrence, contact email	usability, maintainability
URL does not contain version number	Zally	v2	-	usability, maintainability
No singular nouns in path	Zally + prototype	35 occurrences, e.g. quality and v2	3 occurrences, e.g. aggregate-rating and render endpoints	usability
Not more than 8 resources are used	Zally	10 resources	13 resources	usability, maintainability
Maximum three sub resources	Zally	8 occurrences, e.g. shops products reviews comments	-	usability, maintainability
Custom media types are only be used for versioning	Zally	18 occurrences, e.g. product reviews comments etc.	1 occurrence, POST `/event-types`	usability, maintainability
Array property names are plural: parameter	Zally	13 occurrences, e.g. array RequestParameter	2 occurrences, e.g. inviteTemplatePriority	usability
operation contains the default response	Zally	all operations	all operations	usability
Input property is an extensible enum	Zally	-	2 occurrences, e.g. user_defined	usability
Use Additional Ressource Formats	Prototype	4 occurrences, endpoints do not accept XML, e.g. review collector	-	usability
Use Forbidden	Prototype	2 occurrences, e.g. `/shops/{tsId}`	5 occurrences, e.g. `/events/{eventRef}`	usability

an array of items is returned. If the last element is not a singularized noun or respectively not a pluralized noun, the tool adds this path to the invalid results. So, the prototype finds instances in which the URI ends with a pluralized noun but does not return an array or returns an array but ends with a singular noun. The new API version eTrusted for example returns paging objects instead of arrays but uses pluralized nouns. This would therefore violate the "Document Singular Noun" rule. Also, myTS returns e.g. the list of reviews embedded in an object and violates this rule. Furthermore, the tool reports three violations for "Plural Noun Collection/Store Name", as it e.g. does not recognize "event-types" to be a plural noun. Also, the "aggregate-rating" endpoint cannot be detected this way as it is displayed with a singular noun but does not return an array. However, it also replies with a cumulated return object.

Actual violations of these rules can only be detected with context of the API. That knowledge is necessary to know if a specific path element actually references a document which should then be represented by a singular noun or a collection where a pluralized noun is recommended.

Other rules such as the "Substitue Variable Path Rule" check each path for path parameters. If a path parameter is found, it is added to the "valid results" list. Otherwise, it is added to the "unapplicable results" list. Through this implementation, no path can be assigned to the "invalid results" list. This makes sense, as there is no rule to actually use path parameters for every path there is. Nevertheless, this rule is not very expressive. Effectively, all it does is count the number of paths with at least one path parameter. Other rules such as "Encourage Caching" return only invalid and unapplicable results and no valid results. If an endpoint returns the value "no-cache" for the head field "Cache-Control", it is added to the "invalid results" list, otherwise to the "unapplicable results" list. This behavior makes the results of the prototype inconsistent. Therefore, only actual results of rule violations are of interest and just the results of the invalid lists are considered.

The rule "Use hyphens" flags all twelve occurrences for myTS and ten occurrences for eTrusted. As both versions do not stick to this style, there are obviously many violations, so this outcome is not surprising. This type of rule though can be very useful adapted to the specific style guide that is used. So, actual violations can be reported that might be missed when checking manually. The "ContentType" rule checks the content type parameter value of each operation with a pre-selected list of content types. This rule compares the string values to a list of content types. The prototype finds four invalidities for myTS and one invalid content type for eTrusted. It also checks if the response content type is defined. This rule only applies to POST and PUT operations. The reported violations are not actually errors as the wildcard media type "* / *" is used [13]. This is not included in the list of media types and therefore reports an error. However, this rule also checks if content type is used for both the expected input media type as well as the response media type. The input and output media types of some POST operations were sometimes not merged correctly and are therefore missing in the combined file. So, this rule falsely reports instances where an input is missing due to merging. This type of rule is useful to find spelling errors in an API specification, however. Also, some violations are not fully clear as to what they are supposed to report. E.g., Zally identified more than three sub resources in myTS. This is caused by the fact that every path starts with "/restricted/v2" which is counted as two separate resources. Furthermore, the paths for product review comments contain "/products/reviews" even though products is not an independent resource. So, even for the longest path with nine path segments there can only three separate actual resources be identified. However, as this still gives good feedback on which paths might be too long, the rule

is also listed here. Moreover, some results are only relevant for specific use cases and related to company-specific guidelines, e.g. Zally's rules for addressing the API audience. This field can be defined to differentiate between API specifications that are of internal use, e.g. "company-internal" or "business-unit-internal", and of external use for "external-partner" and "external-public". This rule could probably be applied here, however it is also very specific to Zalando's guidelines that are not universally applied. Furthermore, this practice might not be necessary for API specifications of services that are solely public where no private hidden services are used.

The results of the OpenAPI Validator are a bit confusing. Errors and warnings are reported separately. All errors are in the form of a "must"-sentence, but warnings are reported both as "should"- and "must"-sentences. Zally's reports are more consistent in this regard: either "must"-, "should"- or "may" - sentences are returned. They are reported as such in the online documentation for Zalando's guidelines as well. Some reports do not contain such a key word, e.g. "URL contains version number" which is a "must not" - sentence in the Zalando guidelines. We did not find any substantial differences between these different types of rule violations, so we list all types of violations in the same table.

Some rules are not met by the two API versions simply because they designed with another style guide. So, both Zally and OpenAPI Validator require variable names etc. to be in snake case. Both API versions e.g. use camel case instead. Here is to say that enumerations in myTS do not follow the snake case syntax either, but we had removed them because they were not syntactically correct. So, the tools do not report these violations. Other rules are more impactful as e.g. five operation do not have operation IDs in eTrusted. The old API version myTS on the other hand misses a tag-array for all operations. This array can be used to attach metadata. Furthermore, also Zally reports that both API versions have array names with singularized nouns.

A few errors or warnings are reported by both Zally and OpenAPI Validator. E.g., both tools report arrays that are returned as top-level structure. One rule violation from the prototype and one from OpenAPI Validator are even mutually exclusive: The prototype demands all path elements in a URI to be in lowercase and hyphenated, whereas the OpenAPI validator checks if path parameter names are in lower snake case. Also, the prototype checks for top-level arrays when a single noun is used but OpenAPI Validator returns a violation for such instances. Overall, eTrusted violates 19 rules and myTS only 18. Also, eTrusted violates one more rule associated with usability than myTS, namely 18. Both tools violate ten rules associates with maintainability.

All in all, the tools reported 760 violations in total for myTS and only 420 for eTrusted with a few duplicate violations of similar rules reported by more than one tool (see tables A.5, A.6, and A.7).

6.3.2 Metric-based tool analysis

The results of the metric-based analysis with RAMA are shown in table 6.7. Bogner et al. [5] also report quartiles for all metrics by analyzing the metric value ranges of 1,737 API specifications. So, they describe the threshold values for all quartiles. They are also presented in the table. These quartile thresholds take into account if a value is better if it is low or high. Therefore, one is the best quartile with a value that is in the top 25% and four is the worst quartile with a value in the top 75%-100%.

As myTS is divided in a public and restricted part, “/restricted/v2” is prepended with every path. This work only discusses the restricted API as the public API can be seen as a showcase API with few functionality. Therefore, both “Average Path Length” and “Longest Path” are exactly raised by 2.0 due to this fact. Obviously, a small number is desirable for this metric. Both metrics deal with the paths and simply count all ”/”. “Average Path Length” returns the average of all path lengths, whereas “Longest Path” just returns the length of the path with the biggest amount of slashes [19]. Both metrics rate eTrusted better. The average path length value for eTrusted is in the first quartile, myTS’s value is classified in the third one. Also, eTrusted is classified in the best “Argument per Operation” quartile with a value of 3.125, whereas myTS is classified in the second quartile with 3.625 arguments per operation. This metric counts the average number of both path parameters and path query variables and divides it by the number of operations. A low number supposedly implicates less complexity than a high number in this regard [3].

Both APIs are in the best quartile in the “Distinct Message Ratio” metric with just a slightly better rating for eTrusted with 0.1878 versus 0.1505. This metric returns the number of distinct messages regarding the amount of parameters for an operation. A low number is to be preferred, as it implies a high amount of similarly built operations which decreases complexity [3].

Three of the complexity metrics return a better result for myTS. So, “Data Weight” [3] returns only 137.0 for myTS which is in the second quartile of the analyzed APIS but 210.0 for eTrusted which is in the third quartile. This metric rates the complexity of an object by counting how many simple types are contained in this object. The metric returns a summed up value for all objects in a specification. A low number is to be desired.

“Biggest Root Coverage” [19] returns the ratio of the number of paths with the same so-called root elements compared to the total number of paths. A Root element is the first element in a path. As all paths in myTS start with “/restricted”, this value is 1.0. This is also the best value that can be achieved and therefore fits in quartile one. The new API version eTrusted only has a ratio of 0.375 and is classified into quartile four. By removing the first two path elements for each path in myTS, this ratio becomes 0.9375 and falls only under the second worst quartile three. For the same reason, the “Number of Roots” [19] is 1.0 for myTS which is classified into quartile one. If the first two path elements for all paths are removed, the number of roots is still lower for myTS with only two root elements. This value would fall under quartile two. The new version eTrusted is more complex with eight root paths which classifies it into the worst quartile four.

“Lack of Message-Level Cohesion” [2] returns a value based on the lack of edges in a graph where interface operations are mapped and interconnected. A low number implies high cohesion within the operations . Therefore, a low number is to be desired. Both APIs are classified into quartile four although with only a slightly better rating of 0.7559 for myTS with only about 0.16 difference. Furthermore, “Service Interface Data Cohesion” [34] that computes the ratio between operations with same interface inputs to all other sets of operations returns a way better result for myTS. It receives a rating 0.8167 and therefore almost 0.6 better than myTS. A high number implies a better result. Therefore, myTS’s value is classified in the 2nd best quartile, whereas eTrusted is classified into the fourth quartile.

Table 6.7: Results of the metric-based analysis tool RAMA

		myTS		eTrusted	
Metric name	**Property**	**Metric value**	**Quartile**	**Metric value**	**Quartile**
Average Path Length	Complexity	5.9286	3	2.44	1
Arguments per Operation		3.625	2	3.125	1
Biggest Root Coverage		1.0	1	0.375	4
Data Weight		137.0	2	210.0	3
Distinct Message Ratio		0.1505	1	0.1879	1
Longest Path		9.0	3	5.0	2
Number of Roots		1.0	1	8.0	4
Lack of Message-Level Cohesion	Cohesion	0.7559	4	0.9197	4
Service Interface Data Cohesion		0.8167	2	0.2177	4
Weighted Service Interface Count	Size	16.0	3	32.0	4

"Weighted Service Interface Count" [20] returns the overall number of operations. A low value indicates a smaller API size and therefore represents a better rating than a high value. So, myTS has 16 operations and eTrusted has 32 operations. Even though eTrusted's value is double as high as myTS's, the new version is classified into quartile four, whereas myTS is classified just a bit better into quartile three.

7 Discussion

In this chapter, the findings of the manual search for (anti-)patterns and violations of common design rules and best practices, see chapter 6.2, and the tool-supported evaluations (chapter 6.3), are compared to the results of the qualitative data analysis (see chapter 6.1). With the help of this comparison, we can argue if the manual pattern analysis and tool-supported analysis are good indicators for the prediction of software product quality. This can therefore be seen as a comparative discussion of **RQ1**, **RQ3**, and **RQ4**.

7.1 Analysis of (anti-)patterns, design rules, and best practices

As can be seen in table 7.1, (anti-)pattern analysis comes to similar conclusion as the qualitative data analysis does.

Firstly, there are more rule violations and anti-pattern occurrences found in the old API version myTS than in the new API version eTrusted, namely eleven and ten. In eTrusted three positive patterns could be found and in myTS only one. This is supported by the quantitative data that was collected during the expert interviews. The median expert rating for usability is 8.0 for myTS but 9.0 for eTrusted on a scale from one to ten. Moreover, myTS has one more negative code regarding usability, namely four for eTrusted versus five for myTS. These numbers are still very similar. This can be caused by the fact that some interviewees explicitly criticized the new API version about aspects that could also be criticized about the old API version when talking about improvement aspects. As the difference between positive codes is pretty high, namely five (seven versus two positive codes), this can be seen as an expressive hint anyway.

Moreover, myTS shows two missed opportunities where positive patterns could have been implemented, whereas eTrusted shows none. The old API version myTS could have implemented both a state retrieval and a state transition operation but overall lacks this functionality fully.

All of the determined rules and (anti-)pattern violations are associated with usability. Only four of them are associated with maintainability. Three of these are the state creation, retrieval, and transition pattern. The other one is pertinent documentation. Therefore, only one positive maintainability pattern is found for myTS, whereas eTrusted implements all three state patterns. Implementing this pattern would not actually improve maintainability for myTS simply because myTS does not have such a feature at all. Reviews can only be requested but information about pending reviews cannot be retrieved and there are no invite rules that could be updated. The new API version eTrusted offers such functionality. Also, the pertinent documentation pattern is violated

Table 7.1: Overview of qualitative data analysis and manual detection of (anti-)patterns and violations of API design rules and best practices. Better values in each row are marked green and worse ones red.

	myTS		eTrusted	
Software quality attribute	**Positive patterns/-codes**	**Negative patterns/codes**	**Positive patterns/-codes**	**Negative patterns/codes**
Number of patterns regarding usability	1	11	3	10
Median expert Usability rating	8.0		9.0	
Number of codes regarding Usability	2	5	7	4
Number of patterns regarding maintainability	1	1	3	0
Median expert Maintainability rating	7.0		8.0	
Number of codes regarding Maintainability	0	7	7	4

by myTS but not by eTrusted. So, there is not much data on maintainability. However, the median maintainability expert rating for eTrusted, 8.0, is one point better the median rating for myTS with 7.0 points. Lastly, the number of codes analyzed with the help of qualitative data analysis shows a similar picture: seven positively connoted codes about maintainability were found for eTrusted and none for myTS. Also, we found only four negatively connoted codes about maintainability for eTrusted but seven for myTS. All four negative codes for eTrusted are also provided by interviewee 4, a maintainer. A few of them are also supported by interviewee 1, an API user.

Table 7.2 shows a similar overview but we only counted (anti)-patterns that received an importance rating of at least four points ("partially agree that this is important") given by the API experts. This is interesting because the experts think that a lot of rules are not very important. The rule "Links in Representations" for example only receives a rating of three. Comparable rules from the delphi study conducted by Kotstein and Bogner [25] show similar results and receive only low importance ratings. The new API version eTrusted implements none of these important maintainability anti-patterns, whereas myTS shows one, non-pertinent documentation. The old API version implements more usability anti-patterns with eight versus only seven for eTrusted. This result slightly favors eTrusted. This is a similar result as to counting all (non)-important usability rules that also favors eTrusted. Furthermore, the median expert rating regarding usability is one point better for eTrusted. Also, eTrusted implements more positive maintainability and usability patterns, namely three versus one. For the importance ratings for each (anti-)pattern, see tables 7.3 and 7.4. The only detected service-interface anti-pattern "deep path" received an importance rating of five points.

Table 7.2: Overview of qualitative data analysis and manual detection of (anti-)patterns and violations of API design rules and best practices with importance ratings of at least 4 ("partially agree that this is important"). Better values in each row are marked green and worse ones red.

	myTS		eTrusted	
Software quality attribute	**Positive patterns/-codes**	**Negative patterns/codes**	**Positive patterns/-codes**	**Negative patterns/codes**
Number of important patterns regarding usability	1	8	3	7
Median expert Usability rating	8.0		9.0	
Number of codes regarding Usability	2	5	7	4
Number of important patterns regarding maintainability	1	1	3	0
Median expert Maintainability rating	7.0		8.0	
Number of codes regarding maintainability	0	7	7	4

All in all, detecting patterns and anti-patterns might be a powerful technique to find out about the software product quality of a RESTful API specification. The sheer number of anti-patterns and best practice violations can be seen as very expressive as these violations do not have to be seen in context in opposite to higher level structural patterns. Those best practices should never be violated and therefore give a good insight on which rules an API might apply poorly. However, low-level best practices are only useful if they are accepted as common rules in a domain. So, it has to be figured out first which best practices actually should be applied. Then violations of those practices are a good indicator of software product quality. The experts rank the importance of using status codes such like 500 with five points ("fully agree that they are important"), for example. However, actually demanding these at every operation in the specification, they give an importance rating of two ("partially disagree that this is important"). This rule might be enforced differently in other companies.

Moreover, more high level patterns such as service-interface patterns indeed require some interpretational context, as there might be reasons why a pattern is not implemented. A specific pattern might just not fit the domain. So, assumptions on the software product quality have to be argued carefully. Also, there are not many service-interface (anti-)patterns described in the literature that just work on one public RESTful specification in a non-operational context. Nevertheless, these patterns receive high importance ratings from the experts. All high-level service-interface (anti-)patterns

Table 7.3: Detected violations of API design rules and best practices and respective expert ratings

API design rule or best practice	Violations in myTS	Violations in eTrusted	Associated quality attributes	Median expert importance rating	Sources
Lowercase letters should be preferred in URI paths	all path parameters names	all path parameters names	usability	4	[31], [6], [35], [39]
A plural noun should be used for document names	POST reviewcollector	POST render	usability	5	[31], [6], [39], [35], [30]
201 ("Created") must be used to indicate successful resource creation	review request, review collector, shop reviews comments	POST invite-rules	usability	4	[6], [35], [37]
204 ("No Content") should be used when the response body is intentionally empty	GET shops, shop reviews, shop quality indicators, benchmarks, retailer shops, product reviews	GET channels, invites, invite-rules, events, event-types	usability	4	[6], [35], [37]
403 ("Forbidden") should be used to forbid access regardless of authorization state	shops, retailer shops	events, `events/{eventRef}`	usability	4	[6], [35], [37]
415 ("Unsupported Media Type") must be used when the media type of a request's payload cannot be processed	all POST operations	all POST operations	usability	4	[26]
500 ("Internal Server Error") should be used to indicate API malfunction	all operations	all operations except POST operations	usability	2	[6], [35]
Pertinent documentation	GET shop reviews: pagination description missing	-	usability and maintainability	5	[30]
Links in Representations	GET shops to GET shop reviews etc., pagination	GET channels to GET invites etc.	usability	3	[6], [44], [39], [37]
Forms in Representations	missing	missing	usability	1	[37]

Table 7.4: Detected patterns and missed opportunities to implement these patterns and respective expert importance ratings

Service-interface pattern	Occurrences in myTS	"Missed opportunities" in myTS	Occurrences in eTrusted	"Missed opportunites" in eTrusted	Associated quality attributes	Median expert importance rating	Sources
State Creation Operation	POST shops/tsId/review-collector	-	POST invite-rules	-	usability and maintainability	4	[50]
State Retrieval Operation	-	GET invites	GET channels/id/invites	-	usability and maintainability	4	[50]
State Transition Operation	-	PUT invites	PUT invite-rules	-	usability and maintainability	4	[50]

receive a rating of at least four points. So, these (anti-)patterns might be influential on the software product quality. As only three experts contributed to these ratings, our outcome might not represent a universally valid result.

7.2 Tool-supported analysis

In this section, the results of the tool-supported analyses are discussed. It is divided into an analysis part about rule-based tool analysis and metric-based tool analysis.

7.2.1 Rule-based tool analysis

Table 7.5 shows a comparison between the tool findings and the expert opinions of both API versions. Here, the number of rules that have been violated is displayed and not the absolute number of violations, e.g. two rule violations of the same rule only count once. We chose to display this number, as many rules were violated at every possible instance, e.g. 227 schema properties without content were found in myTS. Other rule violations are only found once or twice. So, adding up these numbers is therefore not very meaningful.

The comparison table also shows the median expert quality ratings as well as the respective number of negative codes for these categories. Only the amount of negative codes is listed as most of these tools only report violations and do not list rules that have been followed. So, this number is better comparable to the number of negative codes found through grounded theory. The amount of negative codes has to be interpreted carefully however, as this value does not include the information if the same arguments is made by only one interviewee or by more than one. Also, the absolute number of occurrences and repetitions of an argument is not represented by this value, e.g. important arguments can be found more than once in single interview.

Table 7.5: Overview of qualitative data analysis and tool-supported rule violation findings. Better values in each row are marked green and worse ones red.

Quality attribute	myTS	eTrusted
Total number of rule violations	18	19
Usability rule violations	17	18
Median expert Usability rating	8.0	9.0
Number of negative codes regarding Usability	5	4
Maintainability rule violations	10	10
Median expert Maintainability rating	7.0	8.0
Number of negative codes regarding Maintainability	7	4

Combining and merging similar rules of all three rule-based tools together, eTrusted shows one more violation than eTrusted, namely 19 for eTrused and 18 for myTS. So, the absolute number of violations is not a good indicator for the software product quality. The median values for usability and maintainability are with 9.0 versus 8.0 and 8.0 versus 7.0 similar. But this might be an issue due to the small number of collected data points.

All except one of the reported rule violations - missing API audience, identifier, and meta information affect maintainability - are associated with usability. So, eTrusted shows 18 rule violations associated with usability and myTS one less with 17. We found five negatively connoted codes for usability through grounded theory for the old API version but only four for eTrusted. However, all negative usability codes for eTrusted were raised by interviewee 5, an API maintainer. Also, the median expert rating for usability is slightly higher for eTrusted with 9.0 instead of 8.0. These results are still pretty similar but in favor of myTS instead of eTrusted.

For each API version, ten of these findings can be associated with maintainability as well. Maintainability is rated slightly better by the experts with a median rating of only 7.0 for myTS but 8.0 for eTrusted. Also, myTS shows three more negative grounded theory codes for maintainability than eTrusted, namely seven versus four. So, the tool reports are not fully expressive of the respective quality attributes.

Nevertheless, simply counting the number of rules that have been violated can give a hint towards how the software might actually be perceived in terms of software product quality. As both API versions show similar results in the tool reports as well as the expert ratings, this result might still be in the margin of error.

Still, one has to be careful, as e.g. eight out of the 22 cumulated reports receive an importance rating of three or less. If an analyzed API specification only violates less important rules due to expert opinions, these results might not be as expressive.

Counting only rule violations that are considered at least partly important by the experts, see table 7.6, this gives an almost identical picture: Here, myTS shows twelve violations against rules that have been considered important. The new API version eTrusted violates two more rules, namely 14.

Table 7.6: Overview of qualitative data analysis and tool-supported rule violation findings with a least importance rating of 4 (“partially agree that this is important”). Better values in each row are marked green and worse ones red.

Quality attribute	myTS	eTrusted
Total number of important rule violations	12	14
Important usability rule violations	12	14
Median expert Usability rating	8.0	9.0
Number of negative codes regarding Usability	5	4
Important maintainability rule violations	7	7
Median expert Maintainability rating	7.0	8.0
Number of negative codes regarding Maintainability	7	4

As there is only one rule associated with exclusively maintainability which is also not considered important (API audience metadata), the same numbers are valid for rules associated with usability. This is slightly different than the median expert rating that favors eTrusted with 9.0 over 8.0. So, neither the total number of important rule violations nor the usability rule violations represent the expert perception well. But the amount of rule violations as well as the expert ratings are still pretty close for both API versions.

Lastly, both API versions show the same amount of violations of important maintainability rules, namely seven. This is similar to the median expert maintainability ratings of 7.0 for myTS and 8.0 for eTrusted.

For the distinct importance ratings for each rule, see table 7.7. It has to be noted that the rule “Not more than 8 resources are used” triggered a long discussion about the usefulness of this rule. The experts found a consensus and rated the adjusted rule that as least resources are to be used as possible with four points. They could not agree with the absolute number of eight resources that Zally proposes.

So, collecting the sheer number of rules that are reported can give a hint towards software product quality. Moreover, categorizing the rules by importance might give a better insight. Less important rules that have been implemented for specific domains, e.g. Zally’s rules for addressing the API audience that is mostly important to be able to differentiate between public and private API specifications, can therefore be left out. However, counting only important rule violations, myTS shows even a better results than eTrusted compared to counting all rule violation - in opposite to the expert opinions. Also, only counting violations of important usability rules causes a better result for myTS. Here, eTrusted shows two more rule violations. Counting also non-important rule violations, the difference is only one. In regard to maintainability, the same amount of violations of important rules can be found for both API versions. This is a similar result as also counting non-important rules returns. Counting all important rule violations associated with either usability

Table 7.7: Aggregated reported errors and warnings with expert importance ratings

Rule	Tools	Violations in myTS	Violations in eTrusted	Associated quality attributes	Median expert importance rating
204 responses do not include a message-body	OpenAPI Validator	2 occurrences, GET and DELETE product review comments	-	usability	5
Parameter names + property names + path segments + operationIds + ENUM values follow snake case convention	OpenAPI validator + Zally	all parameter names, e.g. tsId; all operationIds, e.g. retrieveReviews	all parameter names, e.g. channelId; all property names, e.g. createdAt; all path segments,e.g. customer-reviews; all operationIds, e.g. getTemplate; all operationIds, e.g. getReviews.; 53 out of 55 ENUM values, e.g. AUTOMATIC	usability	5
HTTP header fields follow Hyphenated-Pascal-Case	Zally	-	all HTTP header fields, e.g. token	usability	5
Hyphens (-) should be used to improve the readability of URIs	Protoype	6 occurrences, all path parameter names	4 occurrences, all path parameter names	usability	5
Arrays MUST NOT be returned as the top-level structure in a response body	OpenAPI Validator + Zally	-	4 occurrences, e.g. GET `/channels`	Usability	3
Operation do not have non-empty 'tags' array	OpenAPI Validator	all operations	-	usability	3
operationIds follow naming convention: operationId verb should be list, replace, add or create	OpenAPI Validator	5 OperationIds, e.g. retrieveReviews	12 operationIds, e.g. getTemplate	usability, maintainability	5
Required parameters appear before optional parameters	OpenAPI Validator	tsId after acceptLanguage at GET `/shops/benchmarks`	4 occurrences, e.g. id (UUID) after Channel-Id (channel name) in invites endpoint	maintainability, usability	4
Common path parameters are defined on path object	OpenAPI Validator	all operations	all operations	Usability, Maintainability	3
Schema + schema properties have a non-empty description	OpenAPI Validator	all schemas, e.g. dateiso8601, all schema properties, e.g. format	one schema property, invite-rules property rules	usability, maintainability	5
Server URL does not have a trailing slash	OpenAPI Validator	-	missing slash	usability, maintainability	3
Operations have a non-empty 'operationId'	OpenAPI Validator	-	5 occurrences, e.g POST `/events`	usability, maintainability	5
Security scope is used when defined	OpenAPI Validator	-	e OAuth2: [eTrusted]	usability	5
API Audience + API Identifier + API Meta Information is provided	Zally	missing	missing	maintainability	3
Numeric properties must have valid format specified	Zally	1 occurrence, property amount	1 occurrence, contact email	usability, maintainability	5
URL does not contain version number	Zally	v2	-	usability, maintainability	2
No singular nouns in path	Zally + prototype	35 occurrences, e.g. quality and v2	3 occurrences, e.g. aggregate-rating and render endpoints	usability	5
Not more than 8 resources are used	Zally	10 resources	13 resources	usability, maintainability	4
Maximum three sub resources	Zally	8 occurrences, e.g. shops products reviews comments	-	usability, maintainability	5
Custom media types are only be used for versioning	Zally	18 occurrences, e.g. product reviews comments etc.	1 occurrence, POST `/event-types`	usability, maintainability	3
Array property names are plural: parameter	Zally	13 occurrences, e.g. array RequestParameter	2 occurrences, e.g. inviteTemplatePriority	usability	5
operation contains the default response	Zally	all operations	all operations	usability	5
Input property is an extensible enum	Zally	-	2 occurrences, e.g. user_defined	usability	4
Use Additional Ressource Formats	Prototype	4 occurrences, endpoints do not accept XML, e.g. review collector	-	usability	1
Use Forbidden	Prototype	2 occurrences, e.g. `/shops/{tsId}`	5 occurrences, e.g. `/events/{eventRef}`	usability	5

or maintainability, the result for eTrusted is yet even worse than the myTS's result with two less rule violations - in opposite to the expert opinions. Counting also non-important rules, eTrusted only shows one more rule violation. This is contrary to the number of negative codes collected through qualitative data analysis and the quantitative expert ratings.

Counting the actual number of violations instead of only the rules that have been violated might give a more expressive result. This is not helpful in our case however, as eTrusted has a way larger specification, about twice as long, which would falsify the result.

All in all, rule-supported analysis is good for automatically finding syntactical rule violations. The reports even helped finding anti-patterns that have been missed by manual inspection as already discussed. Also, we noticed other inconsistencies that have not been checked manually with the help of the reports. As the tools expect snake case instead of camel case for variable names, e.g. all operation IDs are listed. However, in eTrusted they exist both in camel case as well as separated with hyphens. It is very hard to find such inconsistencies manually. So, it is advisable to use a custom ruleset that is actually of relevance in the specific domain. Also, rules that take linguistic context into account can be error-prone. So, some of the reported errors about singular nouns and plural nouns do not take into account if a response list is embedded as a response object. The prototype, for example, checks if an array is returned or not and then looks up if the last path element is singularized or pluralized. This last step also reported false errors. Therefore, e.g. Zally demands all path elements to be singular. But this also reports some false errors in the case of myTS with the prepended elements "/restricted/v2".

7.2.2 Metric-based tool analysis

The results of the metric-based tool analysis with the help of RAMA together with the median expert ratings collected during the interviews (cf. chapter 5.1) are shown in table 7.8. The experts were asked to rate both API versions regarding usability and maintainability on a one to ten scale where ten indicates the best quality rating in this regard. The three metric property categories, complexity, cohesion, and size can be expected to affect both usability and maintainability.

The median values for both usability and maintainability are 1.0 points higher for eTrusted with 9.0 and 8.0. This is interesting because myTS has the most negative codes for any category split in two categories for both versions. It shows seven negative codes for its maintainability category. So, the median maintainability rating for myTS is unexpectedly high. Not collecting enough data points might cause this result.

Three of the seven complexity metrics show a better result for eTrusted. Average path length, arguments per operation, and longest path all favor eTrusted. Looking at the quartiles, the biggest difference in the favor of eTrusted can be found for the average path length. The old version myTS has almost six path elements on average, whereas eTrusted has 2.44. Therefore, the value for eTrusted falls under quartile one and myTS's value under quartile three. Not counting the first two elements ("/restricted/v2") for myTS, the average amount of path elements would be around four which would be in the second quartile of the determined value ranges. Arguing with the respective

Table 7.8: Results of the metric-based analysis tool RAMA compared to the median expert ratings. Better values in each row are marked green and worse ones red.

	myTS		eTrusted	
Metric name/quality attribute	**Metric value**	**Quartile**	**Metric value**	**Quartile**
Average Path Length	5.9286	3	2.44	1
Arguments per Operation	3.625	2	3.125	1
Biggest Root Coverage	1.0	1	0.375	4
Data Weight	137.0	2	210.0	3
Distinct Message Ratio	0.1505	1	0.1879	1
Longest Path	9.0	3	5.0	2
Number of Roots	1.0	1	8.0	4
Lack of Message-Level Cohesion	0.7559	4	0.9197	4
Service Interface Data Cohesion	0.8167	2	0.2177	4
Weighted Service Interface Count	16.0	3	32.0	4
Median expert Usability rating	8.0		9.0	
Number of positive codes regarding Usability	2		7	
Number of negative codes regarding Usability	5		4	
Median expert Maintainability rating	7.0		8.0	
Number of positive codes regarding Maintainability	0		7	
Number of negative codes regarding Maintainability	7		4	

quartile affiliation, one has to be careful. These quartile thresholds just project the number of API specifications which returned respective values in those ranges. No qualitative statement can be given on how expressive the differentiation between these quartiles is.

So, these three metrics report a similar result as the expert ratings.

The other four complexity metrics show a better results for myTS. This cannot be supported by the result of the expert ratings.

Both cohesion metrics return better values for myTS. The values for lack of message-level cohesion fall under the same quartile however, as they are pretty close. Nevertheless, service-interface data cohesion reports a considerably higher value for myTS. So, these metric values neither fit to the quantitative expert estimations nor to the number of extracted codes for usability or maintainability collected through grounded theory. Also, a lot of the discovered sets with same parameters do not really have a similar context and therefore do not provide better cohesion: e.g., getComment and submitReviewRequest are returned as one of the sets with same parameters. Both operations do not provide similar functionality however.

Lastly, the number of exposed operations (weighted service interface count) is double as high for myTS. Nevertheless, the value for myTS falls under quartile three, only one quartile better than eTrusted. However, this result does not represent the expert ratings for either usability or maintainability well.

Overall, only three of the ten metrics return a better results for eTrusted over myTS. The expert ratings however show a better result in regard to usability and maintainability. The numbers of positive and negative codes extracted through grounded theory show a similar picture. So, in this case, the metrics' results do not predict the software product quality of both API versions well. The metrics describe small and very specific aspects of their respective quality attributes. So, e.g. a higher data weight might impair the complexity and therefore both the usability and maintainability of an API. However, a single metric only describes a small factor of many factors that together make up the complexity and other properties of an API specification. Therefore, these values cannot be seen as absolute indicators for the property. Other metrics, e.g. the total number of operations, probably describe the respective properties more generally valid. The size of an API is not as abstract as the complexity, which is subjectively rated differently. Nevertheless, these metrics might give an indication towards issues of an API specification. So, e.g. easy to comprehend metrics such as average path length can be used to improve the API. Also, the quartile thresholds can give away how good a value is compared to other public specifications. However, metrics which explicitly or implicitly take the length of an API into account do not work well in our case. Weighted service interface count returns the number of operations, for example. However, also data weight returns a higher value applied to specifications with more operations, as this metric returns the total data weight of a specification. So, as the new API version eTrusted has a bigger specification, myTS is favored in this regard.

7.3 Improvement of the new API version

In this section, we propose improvement suggestions for the new API version eTrusted and the migration process based on the results of the qualitative data analysis, the manual search for (anti-)patterns and violations of API design rules and best practices as well as the tool-supported evaluations. This is associated with **RQ 5**.

Three of the interviewees already gave ideas on how to improve the migration process. Two interviewees proposed a sandbox for the new API and one interviewee even had the idea of a service through which Trusted Shops developers would integrate or migrate the applications for their customers themselves. This would probably go beyond the scope of the Trusted Shops services. But another issue with eTrusted would also be solved with this approach. The old API version myTS has publicly available endpoints integrated in a Swagger-UI-like interface which is missed by the two API maintainers, for example. The new API version eTrusted does not offer any public resources. So, it is definitely advisable to offer a test platform of any kind and expose some resources to the public.

Also, the documentation has no real world use cases but just an explanation on how to invoke them. So, just like one interviewee even proposed, a customer survey could be conducted to find out about actual business use cases which can then be added to the documentation. This would also improve the usability of the API.

Maintainability could be improved by applying tool-supported automatic rule checks. As one maintainer said, the syntactical rules are not enforced with the help of tool-support. This could also improve another point of criticism, as the new architectural style of microservices with many teams involved causes a lot of communication overhead. Better tool-support, both about enforcing style rules as well as communication between teams, might resolve this issue.

Finally, the API specification itself can be improved by consequently removing all applied anti-patterns. So, both positive HTTP status codes as well as HTTP error codes have to be implemented consistently, for example. Also, e.g. the deep path anti-pattern that is found by both the manual anti-pattern analysis as wells as the tool-supported evaluations should be fixed. Customer-reviews could be exposed as a top-level resource for the customer reviews veto endpoint, for example. This would shorten some URIs and improve readability and therefore usability of the API specification. Lastly, rule violations found through the linting tools should be fixed, especially the rules that are rated as important by the experts themselves. Therefore, missing operation IDs should be added, for example. Also, the third REST maturity level by Richardson which demands implementing HATEOAS, see chapter 2.4.3, could be applied consequently by e.g. adding missing hyperlinks in responses. Right now, there are only hyperlinks attached to pagination responses. Further hyperlinks to deeper nested operations can improve navigation through the API. However, both the Trusted Shops experts as well as the results from Kotstein and Bogner [25] hint that usability might not benefit a lot from such hyperlinking.

8 Threats to validity

As a main part of this thesis was to conduct a case study in an industrial context, there are also some threats to validity in this process that we discuss in this chapter.

We try to avoid *internal validity threats* such as biases in favor of the new API version - there may be hidden biases that the new API version should be confirmed as the better one - with the help of the study design. In order to prevent such biases, we processed the interview transcripts in an anonymous manner so that the Trusted Shops developers can speak freely. Since the interviews were processed anonymously and we asked for permission to publish passages of direct speech, internal pressure to unjustified praising of the new API version on the interviewees was minimized. This ensured that the interviewees did not feel pressured to only commend and not criticize the new API version, as they felt secure about expressing their actual opinions. So, this can also be seen in the interviews, as a lot of constructive criticism towards the new API version eTrusted came up. It can therefore be assumed that most of the responses were real opinions that have not been biased by the interviewees' relationship to Trusted Shops. Processing the data anonymously also addressed possible ethical issues about the study design. Furthermore, the interview questions had been made up before the interviews and were not changed after the first interview had been conducted. So, biases towards the opinions of the first interviewees can be minimized. Furthermore, all interviewees had received the interview questions before the interviews were conducted. So, all interviewees received the same question base even if interview questions and follow-up questions turned out differently due to the semi-structured interview style.

No external API consumers were interviewed for this study. It was planned to conduct interviews with third party client developers who have actually migrated from the old API version myTS to the new API version eTrusted. However, time challenges on the clients' side have made those interviews impossible. Nevertheless, as three of the interviewees from Trusted Shops were not involved in design processes of the new API version and mostly got in touch with it as API consumers themselves, they can be seen as an appropriate replacement. One of the client-side interviewees admitted that they did not really know how eTrusted was built but only invoked the API once, however. As three of the Trusted Shops developers were interviewed as API clients, this does not impair the results too much. However, analyzing the results of the manual search for (anti-)patterns and the tool-supported analysis, we took the outcome of the qualitative data analysis as the ground truth. As we only interviewed five Trusted Shops developers, this might be a threat to the validity of our results. Nevertheless, the perceptions of the interviewees seemed to be homogenous and were rarely contradictory.

Also, the experts rated the importance of the (anti-)patterns and rule violations before we showed them our findings. So, biases towards the new API version eTrusted can be minimized. The experts could still have a small bias, as they might know by themselves which (anti-)patterns have been

implemented. However, the experts e.g. rated “deep path” as an important anti-pattern with the highest rating of five points (“I agree that this is important”). While discussing the importance, the three experts already acknowledged that they did implement this anti-pattern in both versions. But there is no instance of a high-level service-interface anti-pattern that is only implemented in the new API version eTrusted. So, this bias cannot be fully ruled out, as the experts might rate such a rule as less important just because it would be only implemented in eTrusted. However, the analysis for only important patterns and rules showed an even worse result for eTrusted compared to considering all (non-)important findings. So, the influence of such a possible bias must be small.

Furthermore, the number of detected codes per category collected through grounded theory was sometimes similar for both API versions. So, another person processing the transcripts might have come to a different result in favor of myTS or maybe to an outcome favoring eTrusted even more. However, the total number of codes cannot describe the software product quality qualitatively. This number does not represent how often a single code was found in the transcripts, as some codes can be referenced more often than others. Therefore, we also collected quantitative ratings on usability and maintainability during the interviews. However, this also poses the risk of trying to validate the theory found with the help of grounded theory, cf. Adolph et al. [1]. Nevertheless, we used the qualitative data mostly for comparison with the tool-supported analysis as well as the manual search for (anti-)patterns and design rule violations. We did not try to validate the theory itself with the help of the quantitative data.

Furthermore, in grounded theory data is collected until no more novel codes and theories emerge from the data, ideally. The theory then is saturated, cf. [1], [43]. In our case however, new codes came up through the last interview. The codes still fit the theory that was already emerging. So, the theory was probably close to being saturated. Also, the theory can be assumed to have reached “theoretical saturation” [43] which indicates that the data was collected until the theory itself does not substantially change any more.

Also, some of the violations of low-level patterns, design rules, and best practices were not found by hand but were added after the tool evaluation returned similar errors. So, finding such low-level violations manually is extensive and a lot of instances can be overlooked. Therefore, it can be useful to do low-level analysis with tool support, e.g. looking for every possible status code. Some of these best practices obviously need some individual context, e.g. right use of HTTP verbs. This cannot be achieved with a simple rule-based linting tool.

Furthermore, merging and converting the API specifications for each single endpoint into one big OpenAPI 3.0 specification for both API versions has broken some of the endpoints. Therefore, some tool reports might have missed violations of rules or returned violations that only existed due to merging and converting.

External validity threats can emerge if findings of this work are generalized. As this work is domain-specific, not all findings can be transferred to other disciplines or domains. Our findings indicate that the used analysis tools cannot provide a good indication for software product quality, for example. However, the two compared APIs show different sizes and functionalities. So, a comparison in this manner by only counting rule violations might not be appropriate. The new API version eTrusted has a lot more possibilities to implement rule violations and anti-patterns. Therefore, we counted the number of rules that have been violated instead of all occurrences of rule

violations. However, this way rules that have been violated more than once are not weighted more in the analysis. So, computing the mean number of violations per endpoint is conceivable. However, there can also be multiple rule violations of the same rule in a single endpoint, e.g. missing schema property descriptions were reported 227 times for myTS, cf. A.5. A fairer way could be to calculate the ratio between rule violations and all possibilities in which a specific rule could have been implemented. This would return a number between zero and one for each rule. Identifying all sections in which a rule has not been implemented might be very error-prone, however.

9 Conclusion

We conducted a case study about comparing two API versions developed and maintained by the Cologne-based company Trusted Shops. Both API versions provide similar functionality but are designed differently. The case study consisted of three steps. First, we interviewed both API maintainers and users about the software product quality of both versions. Second, we manually searched for (anti-)patterns and violations of design rules and best practices in the API specifications. Third, we conducted a tool-supported analysis on both API versions. The new API version is perceived as the better one by the interviewees. This is caused by better usability and maintainability. Also, the documentation is seen as superior over the documentation of the old API version. Nevertheless, there are also points of criticism about the new API version. The interviewees criticize that there is no public interface that can be used to test out the new API, for example. The old API version exposes some endpoints to the public for this reason. Such points of criticism also go into the recommendations on how to improve the new API version that we propose. Only the manual search for (anti-)patterns comes to similar conclusions as the qualitative data analysis does. The tool-supported evaluations mostly favor the old API version myTS. Therefore, these approaches do not seem to be able to asses software product quality well. However, these analyses can be a good way of finding low-level syntactical (anti-)patterns that might also impair usability and maintainability of a specification. Also, the metric-based evaluation can be a good indicator on specific aspects of size, cohesion, and complexity of the analyzed APIs. These results can also be compared to the metric values of other API specifications from different domains with the help of the reported quartiles.

9.1 Outlook

Further research endeavours can be made to evaluate the new API version once the improvement recommendations have been applied. So, an improved version can be compared to the current version. Also, it could be investigated how much these changes actually improve the new API version.

Furthermore, more research is needed about the software product quality, especially usability and maintainability, of publicly available service-interface (anti-)patterns. Most of the (anti)-patterns we analyzed are of syntactical and low-level nature. Only a few (anti-)patterns that actually analyze the structural issues of a public API specification could be determined. Many of the common service-interface (anti-)patterns use operational contexts that cannot be applied to an API specification without further information. Also, a lot of the (anti-)patterns described in the literature analyze interactions between both private and public microservices and evaluate the granularity of these interactions instead of analyzing only the publicly available API specifications. Within the

scope of this work, we are mostly interested in (anti-)patterns that are concerned with usability and maintainability. Also, the proposed quality metrics should be validated, as many of the metrics came to different conclusions than our qualitative data analysis.

Lastly, a universally acceptable set of common rules could be defined. The tools we analyzed are a good basis to automatically enforce low-level rules. A lot of the implemented rules are very domain specific, however. A universally applicable ruleset that is automatically checked would improve usability and maintainability immensely, as we found a lot of inconsistencies even in the new API version. A survey or case study could be conducted to find out about universal relevance of rules. Also, with advanced progress in AI and deep learning, reliable automatic detection of higher-level (anti-)patterns is conceivable. These (anti-)patterns take a lot of linguistic and structural context into account, e.g. pertinent documentation. Petrillo et al. [36] already implemented this pattern but only report a precision of about 85% and recall of around 64%. Furthermore, not just automatically checking these rules but also automatically enforcing them for every deployment would improve the API. Therefore, these rule-based linting tools could be further developed. So, they could e.g. analyze merge requests of potential changes in an API specification. This would solve problems due to having various teams that work on many microservices and operate one big system together.

Bibliography

[1] S. Adolph, W. Hall, P. Kruchten. “Using grounded theory to study the experience of software development”. In: *Empirical Software Engineering* 16 (Aug. 2011), pp. 487–513. DOI: 10.1007/s10664-010-9152-6 (cit. on pp. 39, 76).

[2] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, P. Vassiliadis. “Cohesion-Driven Decomposition of Service Interfaces without Access to Source Code”. In: *IEEE Transactions on Services Computing* 8.4 (2015), pp. 550–562. DOI: 10.1109/TSC.2014.2310195 (cit. on p. 60).

[3] D. Baski, S. Misra. “Metrics suite for maintainability of extensible markup language Web Services”. In: *IET software* 5.3 (2011), pp. 320–341 (cit. on p. 60).

[4] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, A. Zimmermann. “Towards a collaborative repository for the documentation of service-based antipatterns and bad smells”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2019, pp. 95–101 (cit. on pp. 42, 43, 89).

[5] J. Bogner, S. Wagner, A. Zimmermann. “Collecting Service-Based Maintainability Metrics from RESTful API Descriptions: Static Analysis and Threshold Derivation”. In: Sept. 2020. ISBN: 978-3-030-59154-0. DOI: 10.1007/978-3-030-59155-7_16 (cit. on pp. 29, 44, 59).

[6] H. Brabra, A. Mtibaa, F. Petrillo, P. Merle, L. Sliman, N. Moha, W. Gaaloul, Y.-G. Guéhéneuc, B. Benatallah, F. Gargouri. “On Semantic Detection of Cloud API (Anti)Patterns”. In: *Information and Software Technology* 107 (Nov. 2018). DOI: 10.1016/j.infsof.2018.10.012 (cit. on pp. 29, 43, 53, 66, 88).

[7] I. Castillo, F. Losavio, A. Matteo, J. Bøegh. “REquirements, aspects and software quality: The REASQ model”. In: *Journal of Object Technology* 9 (Dec. 2010), pp. 69–91. DOI: 10.5381/jot.2010.9.4.a4 (cit. on pp. 17, 18).

[8] L. Chen. “Microservices: architecting for continuous delivery and DevOps”. In: *2018 IEEE International conference on software architecture (ICSA)*. IEEE. 2018, pp. 39–397 (cit. on p. 22).

[9] M. Cremaschi, F. De Paoli. “Toward Automatic Semantic API Descriptions to Support Services Composition”. In: Sept. 2017. DOI: 10.1007/978-3-319-67262-5_12 (cit. on p. 23).

[10] B. Di Martino, M. Rak, M. Ficco, A. Esposito, S. Maisto, S. Nacchia. “Internet of things reference architectures, security and interoperability: A survey”. In: *Internet of Things* 1-2 (Sept. 2018), pp. 99–112. DOI: 10.1016/j.iot.2018.08.008 (cit. on p. 23).

[11] T. Erl, B. Carlyle, C. Pautasso, R. Balasubramanian. *SOA with REST: Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. 1st. USA: Prentice Hall Press, 2012. ISBN: 0137012519 (cit. on pp. 19–22).

[12] T. Espinha, A. Zaidman, H.-G. Gross. "Web API Growing Pains: Loosely Coupled yet Strongly Tied". In: *Journal of Systems and Software* 100 (Jan. 2014). DOI: 10.1016/j.jss.2014.10.014 (cit. on pp. 13, 30).

[13] R. Fielding, J. Reschke. *Hypertext transfer protocol (http/1.1)*. 2014 (cit. on pp. 21, 58).

[14] R. T. Fielding, R. N. Taylor. "Architectural Styles and the Design of Network-Based Software Architectures". AAI9980887. PhD thesis. 2000. ISBN: 0599871180 (cit. on p. 20).

[15] *FORM 10-K*. Washington, DC 20549, 2020. URL: https://www.sec.gov/ix?doc=/Archives/edgar/data/1065280/000106528021000040/nflx-20201231.htm (cit. on p. 13).

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612 (cit. on pp. 26, 27).

[17] S. Ganesh, T. Sharma, G. Suryanarayana. "Towards a Principle-based Classification of Structural Design Smells." In: *J. Object Technol.* 12.2 (2013), pp. 1–1 (cit. on p. 27).

[18] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic. "Toward a catalogue of architectural bad smells". In: *International conference on the quality of software architectures*. Springer. 2009, pp. 146–162 (cit. on p. 89).

[19] F. Haupt, F. Leymann, A. Scherer, K. Vukojevic-Haupt. "A Framework for the Structural Analysis of REST APIs". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017, pp. 55–58. DOI: 10.1109/ICSA.2017.40 (cit. on p. 60).

[20] M. Hirzalla, J. Cleland-Huang, A. Arsanjani. "A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures". In: *Service-Oriented Computing – ICSOC 2008 Workshops*. Ed. by G. Feuerlicht, W. Lamersdorf. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 41–52. ISBN: 978-3-642-01247-1 (cit. on p. 61).

[21] S. Hove, B. Anda. "Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research". In: vol. 2005. Oct. 2005, 10 pp.-. ISBN: 0-7695-2371-4. DOI: 10.1109/METRICS.2005.24 (cit. on p. 40).

[22] *ISO/IEC 25010*. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed: 2010-09-30 (cit. on p. 18).

[23] *ISO/IEC 9126-1:2001 Software engineering — Product quality — Part 1: Quality model*. https://www.iso.org/standard/22749.html. Accessed: 2010-09-30 (cit. on pp. 17, 18).

[24] G. Kim. *Guide The IT Revolution DevOps Guide*. URL: http://images.itrevolution.com/documents/ITRev_DevOps_Guide_5_2015.pdf (cit. on p. 13).

[25] S. Kotstein, J. Bogner. *Which RESTful API Design Rules are Important and How Do They Improve Software Quality? A Delphi Study with Industry Experts*. Zenodo, Mar. 2021. DOI: 10.5281/zenodo.4643907. URL: https://doi.org/10.5281/zenodo.4643907 (cit. on pp. 42, 44, 64, 74, 92).

[26] M. Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. Ö'Reilly Media, Inc.", 2011 (cit. on pp. 53, 66, 88).

[27] A. Neumann, N. Laranjeiro, J. Bernardino. "An Analysis of Public REST Web Service APIs". In: *IEEE Transactions on Services Computing* (2018), pp. 1–1. DOI: 10.1109/TSC.2018.2847344 (cit. on p. 88).

[28] A. Ouni, R. Kula, M. Kessentini, K. Inoue. "Web Service Antipatterns Detection Using Genetic Programming". In: July 2015. DOI: 10.1145/2739480.2754724 (cit. on p. 89).

[29] H.-y. Paik, A. Lagares Lemos, M. Barukh, B. Benatallah, A. Nat. *Web Service Implementation and Composition Techniques*. June 2017. ISBN: 978-3-319-55540-9. DOI: 10.1007/978-3-319-55542-3 (cit. on pp. 20, 23).

[30] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, Y.-G. Guéhéneuc. "Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns". In: *International Journal of Cooperative Information Systems* 26 (May 2017), p. 1742001. DOI: 10.1142/S0218843017420011 (cit. on pp. 43, 44, 53, 54, 66, 88).

[31] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, G. Tremblay. "Are restful apis well-designed? detection of their linguistic (anti) patterns". In: *International Conference on Service-Oriented Computing*. Springer. 2015, pp. 171–187 (cit. on pp. 29, 53, 66, 88).

[32] F. Palma, N. Moha, G. Tremblay, Y.-G. Guéhéneuc. "Specification and detection of SOA antipatterns in web services". In: *European Conference on Software Architecture*. Springer. 2014, pp. 58–73 (cit. on pp. 44, 89).

[33] C. Pautasso. "RESTful Web Services: Principles, Patterns, Emerging Technologies". In: *Web Services Foundations*. Ed. by A. Bouguettaya, Q. Z. Sheng, F. Daniel. New York, NY: Springer New York, 2014, pp. 31–51. ISBN: 978-1-4614-7518-7. DOI: 10.1007/978-1-4614-7518-7_2. URL: https://doi.org/10.1007/978-1-4614-7518-7_2 (cit. on pp. 20, 23).

[34] M. Perepletchikov, C. Ryan, K. Frampton. "Cohesion Metrics for Predicting Maintainability of Service-Oriented Software". In: *Seventh International Conference on Quality Software (QSIC 2007)*. 2007, pp. 328–335. DOI: 10.1109/QSIC.2007.4385516 (cit. on p. 60).

[35] F. Petrillo, P. Merle, N. Moha, Y.-G. Guéhéneuc. "Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study". In: vol. 9936. Oct. 2016, pp. 157–170. DOI: 10.1007/978-3-319-46295-0_10 (cit. on pp. 42, 43, 53, 66, 88).

[36] F. Petrillo, P. Merle, F. Palma, N. Moha, Y.-G. Guéhéneuc. "A Lexical and Semantical Analysis on REST Cloud Computing APIs: 7th International Conference, CLOSER 2017, Porto, Portugal, April 24–26, 2017, Revised Selected Papers". In: July 2018, pp. 308–332. ISBN: 978-3-319-94958-1. DOI: 10.1007/978-3-319-94959-8_16 (cit. on pp. 29, 44, 80, 88).

[37] D. Renzel, P. Schlebusch, R. Klamma. "Today's top RESTful services and why they are not restful". In: Nov. 2012, pp. 354–367. ISBN: 978-3-642-35062-7. DOI: 10.1007/978-3-642-35063-4_26 (cit. on pp. 53, 54, 66, 88).

[38] M. Richards. *Microservices Vs. Service-oriented Architecture*. O'Reilly Media, 2015. ISBN: 9781491975657. URL: https://books.google.de/books?id=Bd5mAQAACAAJ (cit. on p. 20).

[39] C. Rodríguez, M. Baez, F. Daniel, F. Casati, J. C. Trabucco, L. Canali, G. Percannella. "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices". In: *Web Engineering*. Ed. by A. Bozzon, P. Cudre-Maroux, C. Pautasso. Cham: Springer International Publishing, 2016, pp. 21–39. ISBN: 978-3-319-38791-8 (cit. on pp. 53, 66, 88).

[40] P. Runeson, M. Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14 (2008), pp. 131–164 (cit. on pp. 14, 40).

[41] S. Sohan, C. Anslow, F. Maurer. "A Case Study of Web API Evolution". In: June 2015, pp. 245–252. DOI: 10.1109/SERVICES.2015.43 (cit. on pp. 13, 30).

[42] M. Stocker, O. Zimmermann, U. Zdun, D. Lübke, C. Pautasso. "Interface Quality Patterns: Communicating and Improving the Quality of Microservices APIs". In: July 2018, pp. 1–16. DOI: 10.1145/3282308.3282319 (cit. on pp. 41, 42).

[43] K.-J. Stol, P. Ralph, B. Fitzgerald. "Grounded Theory in Software Engineering Research: A Critical Review and Guidelines". In: May 2016. DOI: 10.1145/2884781.2884833 (cit. on pp. 39, 40, 76).

[44] A. B. Stoll, M. L. Chaim, T. D. Oyetoyan, D. S. Cruzes. "OAS DB: A Repository of Specifications to Support OpenAPI Research". In: () (cit. on pp. 43, 53, 54, 66, 88, 89).

[45] *test*. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en. Accessed: 2010-09-30 (cit. on p. 18).

[46] S. Tonse. *Scalable microservices at Netflix. challenges and tools of the trade*. 2018 (cit. on p. 13).

[47] S. Wang, I. Keivanloo, Y. Zou. "How Do Developers React to RESTful API Evolution?" In: vol. 8831. Nov. 2014, pp. 245–259. ISBN: 978-3-662-45390-2. DOI: 10.1007/978-3-662-45391-9_17 (cit. on pp. 13, 30).

[48] R. J. Wieringa. "Research Goals and Research Questions". In: *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 13–23. ISBN: 978-3-662-43839-8. DOI: 10.1007/978-3-662-43839-8_2. URL: https://doi.org/10.1007/978-3-662-43839-8_2 (cit. on p. 15).

[49] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, D. Lübke. "Guiding architectural decision making on quality aspects in microservice apis". In: *International Conference on Service-Oriented Computing*. Springer. 2018, pp. 73–89.

[50] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, M. Stocker. "Interface Responsibility Patterns: Processing Resources and Operation Responsibilities". In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. 2020, pp. 1–24 (cit. on pp. 55, 56, 67).

[51] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun. “Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs”. In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. EuroPLoP ’17. Irsee, Germany: Association for Computing Machinery, 2017. ISBN: 9781450348485. DOI: 10.1145/3147704.3147734. URL: https://doi.org/10.1145/3147704.3147734 (cit. on p. 43).

All links were last followed on May 3, 2021.

A Appendix

Table A.1: Overview of extracted categories and codes associated with either myTS or eTrusted with interviewee IDs

Category	Number of positive codes for myTS	IDs of contributing interviewees	Number of negative codes for myTS	IDs of contributing interviewees	Number of positive codes for eTrusted	IDs of contributing interviewees	Number of negative codes for eTrusted	IDs of contributing interviewees
Maintainability	0	-	7	1, 3, 4	7	1,4, 5	4	1, 4
Usability	2	2,5	5	4,5	7	1, 2, 3, 4, 5	4	5
Error handling	0	-	2	1, 2	1	1, 2, 5	0	-
Documentation	3	1, 2, 4, 5	4	1, 3, 4, 5	5	1, 2, 3, 4, 5	5	1, 3, 4, 5
Standards, Patterns, Guidelines	0	-	3	1, 3, 5	7	1, 4, 5	1	4
Overall	5	1, 2, 4, 5	21	1, 2, 3, 4, 5	27	1, 2, 3, 4, 5	14	1, 2, 3, 4, 5

Table A.2: Overview of extracted categories and codes associated with both myTS and eTrusted with interviewee IDs

Category	Number of positive codes	IDs of contributing interviewees	Number of negative codes	IDs of contributing interviewees
Migration	5	1, 2, 4	13	1, 2, 4, 5
Customer	0	-	3	1, 2, 4, 5
Operations	2	2, 4	4	1,2

Table A.3: All examined API design rules and best practices

API design rule or best practice	Associated quality attributes	Sources
Forward slash separator (/) must be used to indicate a hierarchical relationship	usability and maintainability	[31], [35], [30]
Contextless resource names	usability	[31], [36], [30]
Hyphens (-) should be used to improve the readability of URIs	usability and maintainability	[39], [35]
Underscores (_) should not be used in URIs	Usability	[31], [6], [39] [35]
Lowercase letters should be preferred in URI paths	Usability	[31], [6], [39] [35]
A singular noun should be used for document names	Usability	[31], [6], [39], [35], [30]
A plural noun should be used for collection names	Usability	[31], [6], [39], [35], [30]
Variable path segments may be substituted with identity-based values	usability	[26]
Scoping information	usability	[37]
GET and POST must not be used to tunnel other request methods	usability and maintainability	[6], [44], [39], [35], [37]
GET must be used to retrieve a representation of a resource	usability and maintainability	[6], [44], [39], [35], [37]
POST must be used to create a new resource in a collection	usability and maintainability	[6], [44], [39], [35], [37]
POST must be used to execute controllers	usability and maintainability	[6], [44], [39], [35], [37]
DELETE must be used to remove a resource from its parent	Usability and maintainability	[6], [44], [39], [35], [37]
HTTP Method Override	usability and maintainability	[37]
200 (“OK”) must not be used to communicate errors in the response body	usability	[6], [35], [37]
201 (“Created”) must be used to indicate successful resource creation	usability	[6], [35], [37]
204 (“No Content”) should be used when the response body is intentionally empty	usability	[6], [35], [37]
403 (“Forbidden”) should be used to forbid access regardless of authorization state	usability	[6], [35], [37]
415 (“Unsupported Media Type”) must be used when the media type of a request’s payload cannot be processed	usability	[26]
500 (“Internal Server Error”) should be used to indicate API malfunction	usability	[6], [35], [37]
500 (“Internal Server Error”) should be used to indicate API malfunction	usability	[6], [35], [37]
Meaningful HTTP Status Codes	usability	[37]
Content-Type must be used (Content-negotiation)	usability and maintainability	[6], [39], [27], [35], [37]
Custom HTTP headers must not be used to change the behavior of HTTP methods	usability and maintainability	[35]
JSON should be supported for resource representation	usability and maintainability	[35]
A consistent form should be used to represent media type formats	usability	[35]
A consistent form should be used to represent media type schemas	usability	[35]
A consistent form should be used to represent error responses	usability	[35]
New URIs should be used to introduce new concepts	usability	[35]
Pertinent documentation	usability and maintainability	[30], [36]
Links in Representations	usability	[6], [44], [39], [37]
XHTML Forms in Representations	usability	[37]

Table A.4: All examined service-interface anti-pattern

Service interface anti-pattern	**Associated quality attributes**	**Sources**
Ambiguous Service (here not a single service but the entire public specification)	usability and maintainability	[28], [4], [18]
Chatty Service (here not a single service but the entire public specification)	maintainability	[28], [4], [32]
Deep path	usability	[44]

Table A.5: All reported violations through IBM's OpenAPI Validator

Rule	Number of violations in myTS	Number of violations in eTrusted
A 204 response MUST NOT include a message-body	2	5
Parameter names must follow case convention: lower_snake_case	42	29
Property names must follow case convention: lower_snake_case	126	130
Arrays MUST NOT be returned as the top-level structure in a response body	0	4
Path segments must follow case convention: lower_snake_case	0	15
Schema of type string should use one of the following formats: byte, binary, date, date-time, password	0	3
Operation should have non-empty 'tags' array	16	0
operationIds should follow naming convention: operationId verb should be ...	5	12
operationIds must follow case convention: lower_snake_case	16	27
Required parameters should appear before optional parameters	1	1
Common path parameters should be defined on path object	1	4
Definition was declared but never used in document	5	0
Schema must have a non-empty description	74	0
Schema properties must have a description with content in it	227	1
Server URL should not have a trailing slash.	0	1
'example' property type should be string	0	1
Operations must have a non-empty	0	5
A tag is defined but never used	0	5
Parameters must not explicitly define 'Authorization'. Rely on the 'securitySchemes' and 'security' fields to specify authorization methods. This check will be converted to an 'error' in an upcoming release	0	13
Enum values must follow case convention: lower_snake_case	0	53
Not all languages use JSON, so descriptions should not state that the model is a JSON object.	0	1
Security scope is defined but never used: eTrusted	0	1

Table A.6: All reported violations through Zalando's Zally

Rule	Number of violations in myTS	Number of violations in eTrusted
API Audience must be provided	1	1
API Identifier should be provided	1	1
MUST – Contain API Meta Information	1	1
field 'type' has type 'null' (expected type 'string')	6	0
Numeric properties must have valid format specified	1	1
URL contains version number	1	0
Resource X appears to be singular	32	2
Endpoint not secured by OAuth2 scope(s)	32	2
API has to be secured by OAuth2	1	0
Query parameter has to be snake_case	5	9
Property name has to be snake_case	48	20
must use YAML format (for API)	1	0
Property "dateöf type ßtringänd format "date-timeßhould match one of the patterns [.*_at]"	10	7
Identified 13 resource resource types, greater than recommended limit of 8	0	1
Number of sub-resources should not exceed 3	16	0
Custom media types should only be used for versioning	18	1
Array property name appears to be singular: parameter	13	2
operation has to contain the default response	32	16
Nested paths may be top-level resource	4	5
use only standardized or specified request headers	0	15
Always return JSON objects as top-level data structures to support extensibility	0	4
Property is not an extensible enum (use 'x-extensible-enum' instead)	0	2
Header has to be Hyphenated-Pascal-Case	0	3

Table A.7: All reported violations through the protoype by Kotstein and Bogner [25]

Rule	**Number of violations in myTS**	**Number of violations in eTrusted**
A singular noun should be used for document names	6	2
A plural noun should be used for collection/store names	0	3
Hyphens (-) should be used to improve the readability of URIs"	12	10
Content-Type must be used	4	1

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature