Interface Representation Patterns — Crafting and Consuming Message-Based Remote APIs

OLAF ZIMMERMANN, University of Applied Sciences of Eastern Switzerland, Rapperswil
MIRKO STOCKER, University of Applied Sciences of Eastern Switzerland, Rapperswil
DANIEL LÜBKE, innoQ Schweiz GmbH
UWE ZDUN, University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria

Remote Application Programming Interfaces (APIs) are technology enablers for major distributed system trends such as mobile and cloud computing and the Internet of Things. In such settings, message-based APIs dominate over procedural and object-oriented ones. It is hard to design such APIs so that they are easy and efficient to use for client developers. Maintaining their runtime qualities while preserving backward compatibility is equally challenging for API providers. For instance, finding a well suited granularity for services and their operations is a particularly important design concern in APIs that realize service-oriented software architectures. Due to the fallacies of distributed computing, the forces for message-based APIs and service interfaces differ from those for local APIs – for instance, network latency and security concerns deserve special attention. Existing pattern languages have dealt with local APIs in object-oriented programming, with remote objects, with queue-based messaging and with service-oriented computing platforms. However, patterns or equivalent guidance for the structural design of request and response messages in message-based remote APIs is still missing. In this paper, we outline such a pattern language and introduce five basic interface representation patterns to promote platform-independent design advice for common remote API technologies such as RESTful HTTP and Web services (WSDL/SOAP). Known uses and examples of the patterns are drawn from public Web APIs, as well as application development and software integration projects the authors have been involved in.

CCS Concepts: •**Software and its engineering** → **Patterns;** *Designing software;*

ACM Reference Format:

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns - Crafting and Consuming Message-Based Remote APIs. *EuroPLoP* (July 2017), 36 pages.
DOI: https://doi.org/10.1145/3147704.3147734

1. INTRODUCTION

Object-Oriented Programming (OOP), Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA) established practices for designing modular and distributed software solutions more than a decade ago [Zimmermann et al. 2004]. Since then, the field has further evolved and matured [Pautasso et al. 2017]. As a consequence, application designs have become more and more distributed. This trend has reached a preliminary peak with cloud-native microservices and distributed Internet-of-Things (IoT) architectures that are based on small deployment units communicating via

EuroPLoP '17, July 12-16, 2017, Irsee, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-4848-5/17/07...$15.00
https://doi.org/10.1145/https://doi.org/10.1145/3147704.3147734

remote *Application Programming Interfaces (APIs)* for delivering the overall solution [Fehling et al. 2014; Lewis and Fowler 2014; IERC 2017].

The above mentioned developments require sophisticated and sustainable API designs. However, API design is hard; inexperienced designers will make mistakes that are avoidable. Fixing a mistake in a published API is much harder than fixing an error inside a Web application due to the related impact on a potentially huge number of only partially known systems that are using the API, increased communication and testing efforts, as well as compatibility issues (for breaking changes). As a consequence of these observations, software architects and developers seek related advice. The perceived knowledge gap can be narrowed by patterns addressing remote API-specific design challenges such as service discovery and configuration, message construction and consumption, service granularity, versioning and security.

Many books and pattern languages exist on distributed system design, SOA, Web Services, Representational State Transfer (REST), messaging, cloud computing and related topics (see Section 2 for a summary). However, few (if any) of these cover actual service design, i.e., look into the messages and their content: Which remote API endpoints (i.e., services in SOA, resources in REST) should be exposed? How are API calls structured and what do they mean? Which roles do the API calls play in the overall system-of-systems landscape? We make a first step to tackle this gap in this paper and focus on the *responsibility* of API calls and the representation *structure* of the messages exchanged in an (integration) architecture; our patterns also address aspects related to service delivery *quality*.

The patterns reported in this paper are envisioned to become part of a larger pattern language on *Message-Based Remote APIs*.[1] Let us first illustrate the *scope of the whole pattern language*. The overarching objective of this pattern language is to provide actionable, but also sustainable design guidance for API and service designers as well as system integrators and maintainers. We will discuss forces for interface cuts (emphasizing quality attributes and decision criteria) and suggest designs that have proven useful in multiple project contexts – in terms of structure and meaning of functional interface contracts and in terms of quality concerns such as usability, reliability, performance, and supportability. An example of a recurring problem related to a structural concern (taken from the problem addressed by the *Pagination* pattern described in Section 4.5) is: How to split up large responses (with repeating elements) into multiple messages to address constraints regarding capabilities of message size, network, and/or client and server environments? Furthermore, service identification and evolution concerns are also included in our overall language scope. While many candidate patterns and their known uses stem from experiences in architecting business information systems (a.k.a. enterprise applications), system landscapes, and Web applications such as portal services, all patterns are intended to be applicable in other domains as well.

This paper focuses on an *initial sub-scope of the whole pattern language*: basic interface representations and pagination, i.e., the structure of the message content. Faithful to the ambition of most (if

[1]The term *message-based* here means that communication is facilitated through remote message passing [Daigneau 2011]: A message is sent to a receiver, which must then serve it (e.g., dispatch to an object running inside it, or delegate the message to an alternate receiver). The dispatch/delegation policy and its implementation is hidden from the sender, who can not make any assumptions about the receiver-side programming model and service instance lifecycles. In contrast, Remote Procedure Calls (RPC) do not only model remote service invocations, but also bind server-side subprogram runs to the service instances they are invoked on; such instances (e.g., remote objects) are visible across the network. In summary, while we deal with remote calls, we do not assume that these remote calls are remote *procedure* calls. Message-based remoting services can be realized in multiple technologies and platforms including RESTful HTTP, Web services (WSDL/SOAP), WebSockets, and even gRPC (despite its name). For instance, the HTTP 1.1 specification explicitly states that HTTP is a request-response protocol, but it does not dictate any server-side implementation paradigm (such as object-oriented or functional programming); API calls come as HTTP *methods* operating on *resources* identified by URIs and appearing in messages as resource *representations*.

not all) pattern languages to provide timeless knowledge, our patterns are presented in an technology-independent way; technology-specific design elements such as good practices for resource design in RESTful HTTP appear in our examples. The presented patterns primarily target integration architects, service designers, and Web developers with an interest in platform-independent architectural knowledge. Both backend-to-backend integration specialists and frontend architects and developers as API consumers are supposed to benefit from the captured knowledge. Secondary target audiences include API product owners, cloud offering providers (developers, operators), and API reviewers.

We established several non-goals as well. Object-oriented remoting, for instance, is out of scope; therefore, no remote call stubs (as used in RMI and IIOP), object handlers, or remote garbage collection (as provided in CORBA) are discussed. Due to our focus on representations and message content, we do not distinguish between synchronous and asynchronous communication. Technology-specific advice such as REST recipes, hypermedia design advice, and WSDL/SOAP best practices are already documented elsewhere ([Allamaraju 2010; Zimmermann et al. 2003] are just two of such references) and therefore also excluded here. Application-level protocols, composed workflows, and messaging conversations are covered by other pattern languages that we reference (rather than replicate or rival). Finally, other organizational concerns that go beyond versioning and basic lifecycle management are not part of the core focus of our language either.

The paper is structured in the following way. Section 2 discusses relations to patterns in other languages, as well as differences to them. Section 3 outlines the scope of our pattern languages in terms of design decisions to be made and forces to be resolved; it also presents the preliminary language organization into categories. Section 4 then introduces five structural representation patterns. Section 5 identifies and lists additional pattern candidates across all categories and outlines the patterns currently emerging in selected categories. Section 6 summarizes and gives a brief outlook.

2. RELATIONS TO OTHER PATTERNS AND PATTERN LANGUAGES

The design of message-based remote APIs can benefit from many existing pattern works on various kinds of distributed systems, especially those related to services, as well as pattern works related to API design (e.g., API design in object-oriented programming) and enterprise integration.

In POSA vol. 4, Buschmann et al. [Buschmann et al. 2007] introduce a pattern language that glues together patterns from many different pattern languages on distributed systems, ranging from architectural concerns to low-level design details of distributed systems. The Remoting Patterns language [Voelter et al. 2004] specifically deals with the Broker-based design and the internal details of a middleware, but also covers API-related aspects like remote objects, servants, lifecycle management at runtime, and asynchronous invocations. Asynchronous invocation plays a central role in message-oriented middleware and is covered in depth in Enterprise Integration Patterns (EIP) [Hohpe and Woolf 2003]. While EIP focuses on asynchronous messaging systems, it also covers some aspects of message-based API design. In his general treatment of enterprise application architecture, Fowler [Fowler 2002] touches on many aspects of remote API design such as Remote Facades or Data Transfer Objects (DTOs). Similarly, Evans [Evans 2003] covers functional API design with some of his Domain-Driven Design (DDD) patterns such as Bounded Context, Aggregate, and Service. Even basic design patterns [Gamma et al. 1995] are relevant for remote API design, as they can be useful for some design aspects such as introducing a Facade or a Proxy in the API. We adopt and refine such patterns for message-based remoting.

General data modeling patterns [Hay 1996] cover data representations and meaning, but do so in the context of data storage and presentation (rather than data transport); therefore, the discussed forces and solutions to them differ from ours. Domain-specific modeling archetypes for enterprise information systems also can be found in the literature [Arlow and Neustadt 2004]. Neither the general nor the domain-specific data model catalogs introduced in these two books cover the specific aspects of data representation in message-based remote API communication.

Services provided by message-based remote APIs can be seen as components with remote interfaces. Such components and distributed objects are covered by the pattern languages listed above. More specifically, the work by Daigenau [Daigneau 2011] provides patterns for service-based designs at the level of existing service platforms and technologies such as both REST and WSDL/SOAP-based Web services. Contract versioning for backward compatibility is one of the problem sets that is addressed in this book. In contrast, process-driven SOA patterns [Hentrich and Zdun 2011] reside on a higher abstraction layer. They describe orchestrations of services based on business process or workflow engines. SOA Patterns by Arnon Rotem-Gal-Oz is largely SOA infrastructure- and platform-centric as well [Rotem-Gal-Oz 2012]; his patterns do not investigate message content and structure in depth.

Many other forms of interactions or message exchanges can be summarized by service interaction patterns [Barros et al. 2005]. Basic and advanced conversations such as Polling are covered in the ongoing work on a Conversation Patterns language [Hohpe 2007] that discusses stateful interactions composed of multiple message exchanges between loosely coupled services. In addition, Pautasso et al. [Pautasso et al. 2016] describe conversations specific to RESTful services. The application of the interface representation patterns from this paper is closely related to the application of conversation patterns and vice versa: Each message exchange in a conversation requires request and response messages which need interface representations. Coarse-grained APIs often are used in simple conversations, whereas fine-grained ones lead to more chatty conversations.

Emerging distributed system architectures like cloud computing and the microservices approach to service-based systems [Lewis and Fowler 2014] require many distributed system patterns to build remote APIs, but also bring their own patterns or flavors of related patterns with them [Richardson 2017]. The Data Abstractor pattern in [Fehling et al. 2014] is an example.

Complementary to pattern languages, platform-specific best practices and design guides have also been published, e.g., recipes in a RESTful HTTP Cookbook [Allamaraju 2010] and decisions required in Web services design and related best practices [Zimmermann et al. 2003].

3. LANGUAGE SCOPE AND ORGANIZATION

3.1 Motivation

When designing interface representations (i.e., request and response message content) to realize remote APIs and SOAs, the required *architectural decisions* [Zimmermann et al. 2008] include:

1. Message exchange pattern: request-response vs. one way and request-acknowledge [Daigneau 2011; Pautasso et al. 2017].
2. Message exchange format: JSON, XML, or other [Pautasso et al. 2017].
3. Data contract to be exposed, i.e., *Published Language* in DDD terminology [Evans 2003].

In this paper, we concentrate on the exposed data contract (and underlying domain model). All patterns are described to work with any textual message exchange format; our examples use JSON and XML. We use the request-response message exchange pattern in our examples due to its widespread usage; the patterns are written in such as way that they are also eligible when another message exchange pattern has been chosen. Designing under the specific constraints of particular integration styles such as asynchronous queue-based messaging is covered by other pattern languages and additional knowledge sources (see Section 2).

During the architectural decision making, the following primary forces (i.e., decision drivers and quality attributes) have to be taken into account and must be resolved:

- *Latency* (from API consumer/client point of view), influenced by network behavior (e.g., bandwidth and low-level latency) and endpoint processing effort including (un-)marshalling of the payload.
- *Throughput* and *scalability* (primarily an API provider concern), meaning that response times do not degrade even if provider-side load grows because more clients use it, or because existing clients cause more load.
- *Learning effort* and *modifiability* as an important sub-concern of supportability and maintainability (e.g., backward compatibility to promote parallel development and deployment flexibility, in particular).
- *Security* (e.g., confidentiality and integrity of sensitive information) and other quality-of-service concerns such as data sensitivity, reliability, and manageability.
- Amount of *coupling* and knowledge that must be shared by provider and consumer, which has an impact on changeability as another sub-concern of maintainability.

For some of these forces, their impact on representations appearing in interfaces is obvious; for others, the relationship will become clear when taking a closer look. For instance, security and manageability primarily are runtime concerns. However, there is a resulting need to configure monitoring/metering and billing via API calls (or certain parameters in API calls), and providing security might require certain credentials to appear in the messages.

3.2 Basic Abstractions and Concepts (Language Foundations)

In this paper, we will use the following terminology: A *request message* from an *API client* goes to an *API provider*, which processes the request message and, if the message exchange pattern is request-response, constructs a *response message*. If the difference between client and provider does not matter, the generic term *communication participant* (or party) is used. If needed, the *sender* and the *receiver* of a message are distinguished: request messages are sent by the client and received by the provider; response messages are sent by the provider and received by the client. An API provider exposes one or more *API endpoints*, each of which has a unique address (such as a Uniform Resource Locator, URL); API endpoints expose one or more *API calls*.

The top-level data elements of request messages are called *in parameters*, and *out parameters* appear in response messages (picking up terminology from programming and the Web); these parameters may or may not be ordered and (statically or dynamically) typed. All parameters defined for a message constitute its *message signature*, which becomes part of the technical *API contract*. API contracts are accompanied by actionable *Service Level Agreements (SLAs)* that contain measurable *Service Level Objectives* to govern and classify the API usage terms and conditions. API contract and SLA represent the knowledge shared by provider and consumer that determines the amount of coupling between them.

We call the wire-level equivalent of a program-level Data Transfer Object (DTO) [Fowler 2002; Daigneau 2011] a *Data Transfer Representation (DTR)*; DTOs and DTRs contain request and response message content. Unlike DTOs, the DTR abstraction does not make any assumption about client- and server-side programming paradigms (such as object-oriented, imperative, or functional programming); the client-server interactions are plain messages (i.e., they do not contain any remote object stubs or handlers). Two types of representations are distinguished, e.g., *request representation* and *response representation*. The process of converting a programming language DTO into a DTR that can be sent over the wire is called *marshalling* (also known as serialization); the opposite operation is called *unmarshalling* (or deseralization). These terms are commonly used in distributed computing technologies and platforms.

Fig. 1 illustrates this setup of these basic remote messaging concepts in an architecture overview diagram. The figure also shows the position of APIs in the overall integration architecture as well as selected API client internals from a logical component point of view (e.g., context and error handling).

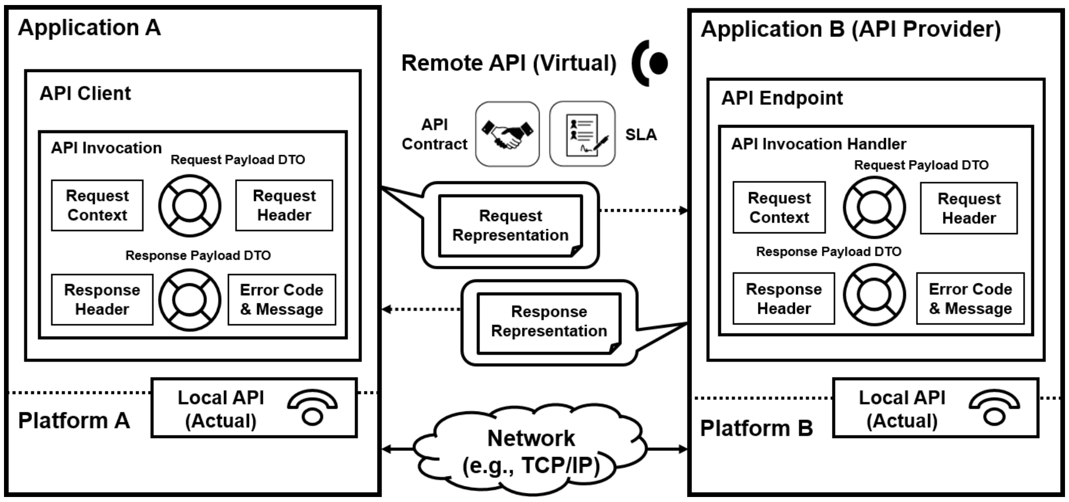

Fig. 1. Architecture of message-based remoting with key API design concepts

In SOA, the terms *service consumer* and *consumer* are synonyms for API client; the API provider acts as a *service provider*.[2] Depending on the integration syle that is used, the abstract concept of an API call can be realized as (i.e., is refined into) remote *methods*, *operations* or *resources*. The detailed API designs for specific technologies differ; for instance, when using a RESTful integration style, one resource can return multiple representations, e.g., to satisfy the information need of different clients (in terms of message syntax and verbosity) or to support backward compatibility and overcome interoperability issues.

3.3 Language Organization

To help integration architects make the decisions and resolve the forces identified in Section 3.1 in a way that is appropriate for their given project context and requirements, and to help them orient themselves in the language, we propose to organize it in seven categories (Fig. 2): *foundations*, *identification*, *structure*, interface *responsibility*, delivery *quality*, service *evolution*, and API *management*. The core of the language is provided by the three categories in the circle in the middle of the figure, Responsibility, Structure, and Quality (RSQ), answering "why","what", and "how" questions about API design, respectively[3]. The Foundations category establishes the language context and its vocabulary, beginning with the concepts introduced above in Section 3.2, as well as coupling criteria such as those compiled in [Gysel et al. 2016]. The Identification category contains process patterns that supports API designers in finding well-suited API call candidates and service abstractions. The Evolution category deals with lifecycle management concerns such as versioning and governance.

[2]In UML terminology, the API client *requires* a particular API *provided* by a component.
[3]It might be surprising at first glance that quality is a core topic in a language that deals with external service/data representations; however, such representations contain quality-related information and their size and structure has an impact on delivery

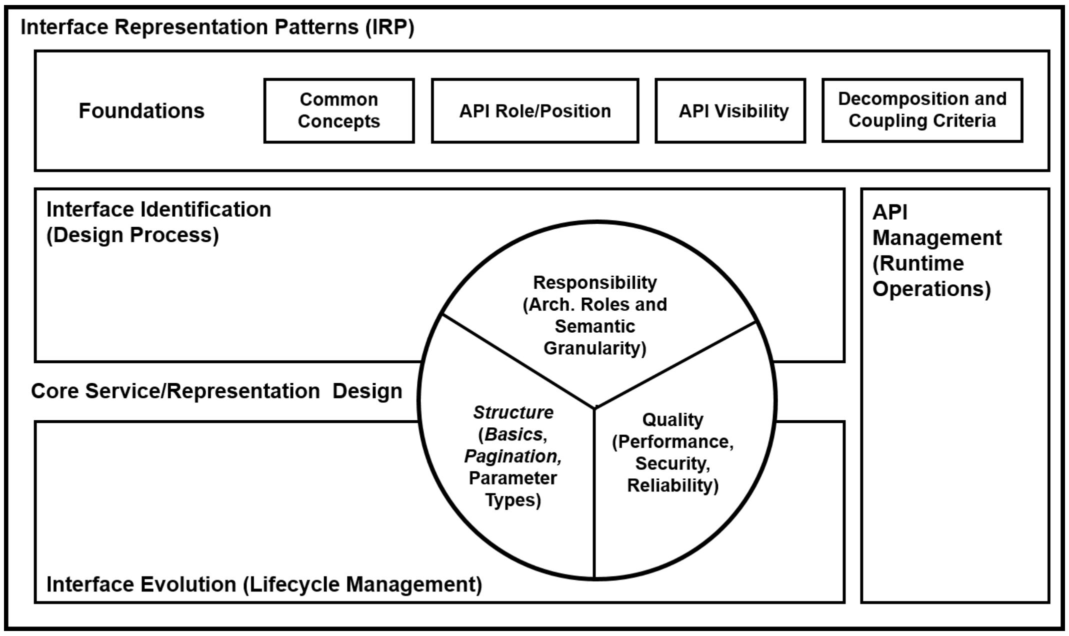
Interface Representation Patterns (IRP)
Foundations
Common Concepts
API Role/Position
API Visibility
Decomposition and Coupling Criteria
Interface Identification (Design Process)
API Management (Runtime Operations)
Responsibility (Arch. Roles and Semantic Granularity)
Core Service/Representation Design
Structure (Basics, Pagination, Parameter Types)
Quality (Performance, Security, Reliability)
Interface Evolution (Lifecycle Management)

Fig. 2. Pattern language categories. The core focus of this paper is on structural representation patterns (highlighted in *italics*).

In this paper, we introduce basic patterns from the Structure category (Section 4) and outline additional patterns across categories (Section 5).

4. STRUCTURAL REPRESENTATION PATTERNS

This section introduces basic structural patterns that respond differently to the same overall design concern in interface representation design:

What is an adequate amount and structure of the ‘in‘ parameters and the ‘out‘ parameters of API calls?

For instance, in a RESTful HTTP context: how many URI parameters should be defined and which media types should be used to represent resources in request and response bodies, and how are these parameters and media types formatted and structured (e.g., in JSON or another notation)? For instance, in a WSDL/SOAP context: how should the message part(s) in the service contract be organized and which data types are used to define the corresponding element(s) in XML Schema (XSD)?

The four patterns *Atomic Parameter*, *Atomic Parameter List*, *Parameter Tree*, and *Parameter Forest* present alternative solutions to two fundamental questions that can be derived from the overall design concern from above:

- Should simple atomic or structured data types be used to define the parameter(s)?
- Should one or more parameters be included in the request and response message contracts?

The four pattern names aim at sketching the message structure/call signatures that result from the 2x2 combinations of the outlined alternative representation design solutions (simple vs. structured

quality. For instance, security can only be achieved if certain security information is transported, and large and complex DTRs may harm latency and throughput (two facets of performance).

data, single vs. multiple parameters). Each pattern has a problem statement derived from this overall design goal, which is slightly adapted to its pattern context.

We also introduce a fifth pattern *Pagination* that leverages the four basic patterns to present how to process queries and query responses efficiently in remote APIs. This recurring design concern has been identified by members of our target audience as particularly important and promising; due to its frequent use and technical complexity (i.e., advanced and challenging forces apply), related architectural knowledge, captured in pattern form, is expected to be a welcome contribution to the body of integration design knowledge.

Figure 3 visualizes how the five structural representation patterns introduced in this section relate to each other. *Atomic Parameters* can be part of *Atomic Parameter Lists*, which in turn can be refactored into *Parameter Trees*. A *Parameter Forest* does not have a single root (unlike a *Parameter Tree*), but contains multiple *Parameter Trees* and/or *Atomic Parameter Lists*.

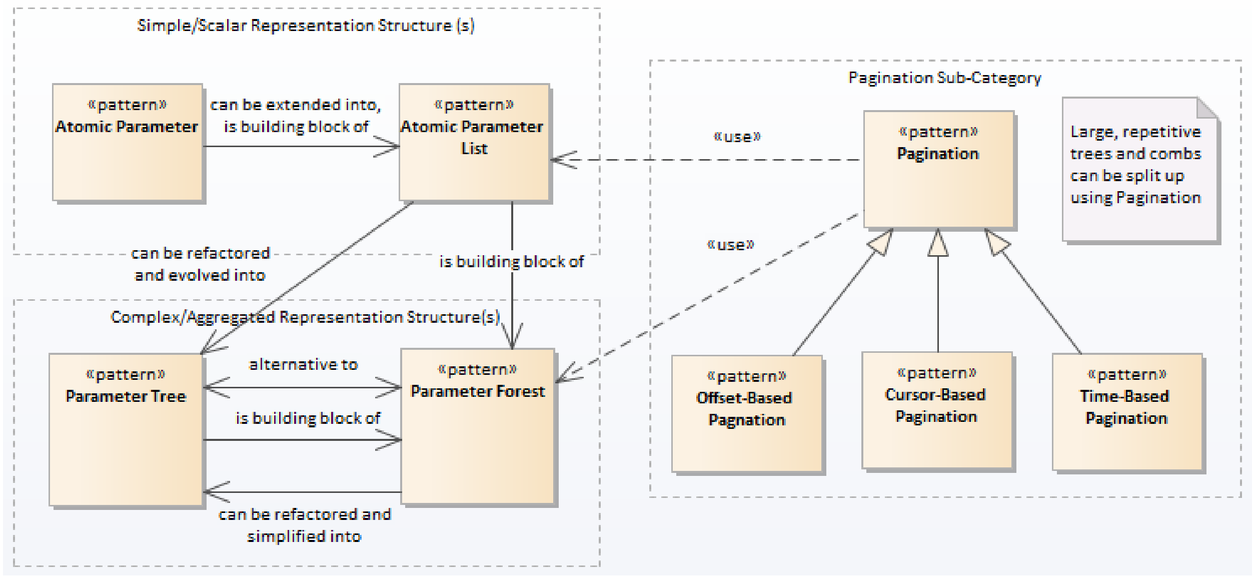
Simple/Scalar Representation Structure (s)
«pattern» Atomic Parameter
can be extended into, is building block of
«pattern» Atomic Parameter List
can be refactored and evolved into
is building block of
Complex/Aggregated Representation Structure(s)
«pattern» Parameter Tree
alternative to
is building block of
can be refactored and simplified into
«pattern» Parameter Forest
Pagination Sub-Category
«use»
«use»
«pattern» Pagination
Large, repetitive trees and combs can be split up using Pagination
«pattern» Offset-Based Pagnation
«pattern» Cursor-Based Pagination
«pattern» Time-Based Pagination

Fig. 3. Relationships between structural representation patterns and pagination variants (UML class diagram)

Two governing top-level forces for the patterns in this category are: a) message size and verbosity (due to impact on network resource consumption vs. network capability and b) security concerns. General decision drivers in service representation design include the functional requirements (e.g., use cases or user stories, domain model), operational model (e.g., network topology and number of deployment artifacts per deployment unit), proxy code generation principles and policies, strong vs. weak typing philosophy, and capabilities of XML and JSON processors ([Zimmermann et al. 2004]). Five particularly important quality attributes are:

- *Interoperability* on protocol and message content (format) level, as influenced by the communication platforms and the programming languages used by consumer and provider implementations (e.g., during parameter marshalling and unmarshalling).
- *Performance* (latency in particular), i.e., outbound processing (marshalling), network travel time, inbound processing (unmarshalling) as influenced by message verbosity.

- *Developer convenience and experience* (e.g., learning and programming effort) both on consumer and on provider side; the wants and needs of these two sides often are conflicting. For instance, a data structure that is easy to create and populate might be difficult to read; a compact format that is light in transfer might be difficult to document, understand, and parse).
- *Maintainability*, especially extensibility of existing messages, ability to deploy and evolve API clients and providers independently of each other. Loose coupling [Leymann 2016] is a related internal quality of the structural design of a distributed system and its components; as an architectural principle, it can be seen to reside half way between a requirement (problem) and a design element (solution). An API call by definition/inherently couples client and server; however, the looser the coupling, the easier it is to evolve client and server independently from each other. Two APIs from one provider should not be coupled unnecessarily, e.g., via hidden dependencies (they may share design guidelines and use the same patterns, though).
- *Security* and data privacy in particular: data in transit should not be tampered with, and it should not be possible to pretend to be somebody else when reading and writing messages. Requirements such as Confidentiality, Integrity and Availability (also known as the CIA triad) [Julisch et al. 2011]; remote APIs may be subject to denial of service attacks.

Concrete, detailed forces differ for public APIs and APIs that are internal to a community or a solution (e.g., a billable SaaS offering vs. a company-internal backend system integration). We will cover such forces in the individual pattern texts that follow. Recipe 2.2 in RESTful Web Services Cookbook [Allamaraju 2010] also discusses decision criteria such as network efficiency, size of representations, client convenience; cacheability, frequence of change, and mutability.

4.1 *Atomic Parameter* Pattern

also known as: Single Scalar Representation, Dot

Context. An API endpoint such as a REST resource or a WSDL/SOAP port has been defined and its API calls (e.g., HTTP methods, WSDL operations) have been specified initially, possibly having applied one or more service identification patterns such as Contract First and a message exchange pattern such as Request-Response [Daigneau 2011]. However, API consumer and provider have not yet agreed on the structures of request and response messages; the data contract between them has not been specified.

Problem. How can an API provider define a single data unit for the `in` parameter in request messages and/or the `out` parameter in response messages?

Forces. The data structures of the request and response messages are an essential part of the technical *service contract*. All patterns dealing with structural interface representation design and technical *service granularity* [Zimmermann et al. 2004] share the same high-level forces (as elaborated previously):

- Interoperability
- Performance (of network and endpoints)
- Processing effort (at development time and at runtime)
- Learning effort and maintainability (versioning and backward compatibility in particular)
- Security

In addition to the API/service endpoint address and call name (if any), the request message and response message structures are important elements of the technical service contract between provider and consumer; they contribute to the shared knowledge of the communication participants/parties.

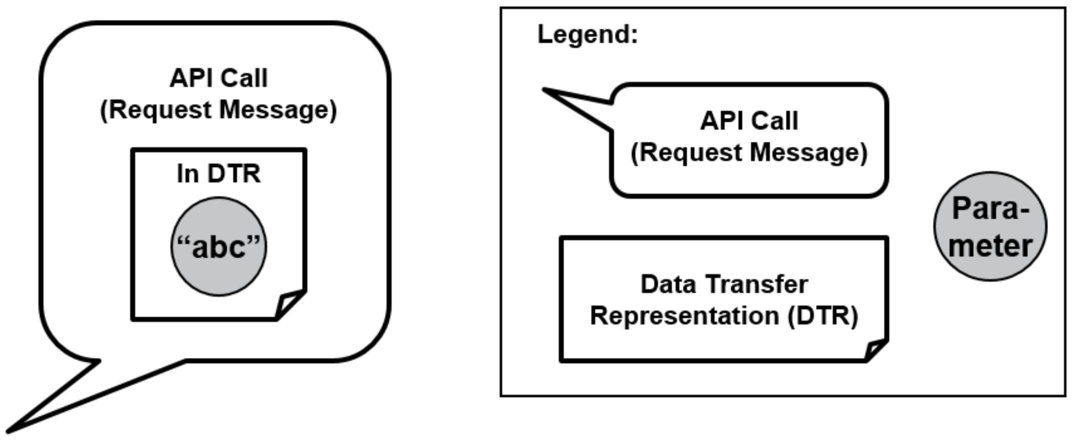

Fig. 4. Atomic Parameter pattern: single scalar parameter (here: in parameter in request message)

This shared knowledge contributes to the coupling between consumer and provider, which is discussed as the *format autonomy* aspect of loose coupling [Leymann 2016]. If the contract is underspecified, interoperability issues may arise, e.g., when dealing with optionality (which can be indicated by absence but also by dedicated `NULL` values) and other forms of variability (e.g., choosing between different representations). If the contract is over-specified, it becomes inflexible and backward compatibility becomes hard to preserve. Simple data structures lead to fine-grained service contracts; complex ones are often used for coarse-grained services (that cover a large amount of business functionality).

Sometimes, only little data has to be exchanged to satisfy the information need of the communication (integration) partner, e.g., when checking the status of a processing resource (in the form of a distinct value defined in an enumeration) by identifier (e.g., a number or a string). One could think of always exchanging strings or key-value pairs, but such generic solutions increase the knowledge implicitly shared between consumer and provider, which complicates testing and maintenance, or might bloat the message content unnecessarily.

Solution.

How it works. To exchange only simple, unstructured data such as a number or plain text, define a single scalar `in` and/or `out` parameter in the API contract. Give the parameter a name and specify a value range for it. Make this type information explicit (statically or dynamically). When doing so, use one of the primitive types provided and supported by the underlying application protocol and transport infrastructure if possible (e.g. UTF-encoded strings to exchange plain text); define custom type extensions and new types only if absolutely necessary (to minimize integration, testing, and maintenance effort as well as implicit coupling).

Like in any integration effort that has to overcome heterogeneity, describe the meaning of the transported values at least informally (including, for instance, a unit of measure); optionally, provide *data provenance* information in the API documentation (note that such information may increase coupling because the message receiver might start interpreting and depending on it, which makes the API harder to change). Test with valid and invalid data, with special emphasis on the edges of the value range; validate received data.

Figure 4 visualizes a string parameter as a single instance of the pattern appearing in a request message.

Example. Our RESTful HTTP and JAX-RS examples[4] use instances of this pattern, for instance in the request message used to GET (and PUT) claims by ID:

```
@GET
@Path("/{claimId}")
public ClaimDTO getClaimById(@PathParam("claimId") UUID claimId) {
  return claims.findById(claimId).map(ClaimDTO::create).orElseThrow(noSuchClaim);
}
```

```
curl http://localhost:8080/claims/a1e00494-e982-45f3-aab1-78a10ae3e3bd
```

In RESTful HTTP, HTTP headers are simple key-value pairs of *Atomic Parameters* (with some format, like `Accept: application/json`); hence, they are often used to exchange simple, unstructured data (but may also carry more elaborate structures like an →*Atomic Parameter List* as seen in the `Accept-Language` header). Both HTTP headers and body can be secured on the transport level (e.g., with TLS/SSL).

Implementation hints and pitfalls to avoid. Architects and developers that decide to apply and realize this pattern should take the following advice into consideration:[6]

- Make sure to use an interoperable data type to define the single scalar parameter. This is less of an issue when communicating over HTTP due to protocol-level Media Type Negotiation, captured as a pattern in [Daigneau 2011]. However, date and time formats can still cause interoperability issues. Hence, it is suggested to stick to standard formats such as those specified in standards such as ISO 8601 and RFC 3339. Generally speaking, use common data types only, avoid defining custom conventions for the meaning of the values because such conventions add to the coupling between the communicating parties, and do not age well. Such shared knowledge might vaporize over time, especially if its documentation is incomplete and not kept current.
- Specify any notion of "undefined" or NULL values explicitly, including their expression/operation semantics (e.g., comparison, equality).
- Consider restricting the value range of the single scalar parameter; if you do so, make this restriction explicit in API documentation, sample code, and tests. Make sure that the API implementation obeys the same restrictions as its interface (validation, database schema, etc.).
- Do not use instances of this pattern to wrap binary data (such as zip files) or encode data structures (such as SOAP envelopes or `base64` encoded URI parameters in REST) unless there is a strong business or technical need for such tunneling (to be approved on a lead or even enterprise architect level). Such encodings are wasteful (e.g., seven bits instead of eight bits). Generally speaking, tunneling a complex data structure in a custom string is considered an integration anti-pattern.
- Acknowledge that a resource URL in RESTful HTTP is an opaque identifier which should not be used to embed parameters (except for identity information); additional parameters should be made part of the query string (i.e., appear after the question mark `?`).

[4]The example is publicly available on GitHub: https://github.com/web-apis/riskmanagement-server[5].
[6]While some of the hints are rather generic and also apply to local, program-internal interfaces it is important to remember in our remote messaging and contract design context and this basic pattern due to the remoting-specific forces at work (e.g., think of the fallacies of distributed computing, including assumptions about security and reliability) and the consequences of forgetting about them.

The API Stylebook[7] collects additional nuggets of advice, e.g., about consumer input (for in parameters in request messages).

Discussion. Simple scalars are easy to read, write and process in all programming languages. They are also easy to secure due to their simplicity and intrinsic cohesion. Simple scalars are often seen to be highly interoperable (assuming that common data types are used) and easy to consume (only little network capacity unless very long strings are transferred). However, their impact on interoperability and bandwidth consumption is not obvious on the platform-independent pattern level; see implementation hints for some related plaform-specific thoughts (for HTTP).

However, the expressiveness of such simple scalars is limited; a rich Published Language [Evans 2003] cannot be exchanged easily this way, but would have to be encoded in the single scalar parameter (e.g., with `base64` encoding). If many API calls (such as service operations) use this pattern, it might be required to compose rather complex and chatty, stateful message conversations to achieve a certain business (or integration) goal as expressed in a user story (or integration story). Consider to switch to the →*Atomic Parameter List* or the →*Parameter Tree* pattern if this happens and becomes unmanageable.

Not all of the general forces that apply to structural representation design can be resolved by this simple introductory pattern; service contract design involves many additional concerns. For instance, result customization has to be dealt with (e.g., using the `Accept` header in HTTP) and offset/limit parameters be introduced if →*Pagination* is used.

Known Uses. Most if not all message-based remote APIs use this pattern frequently, for instance when requesting the status of a long running business transaction (activity) by process id. For example, the `createBucket` call in the Web API of Amazon Web Service (AWS) Simple Storage Service (S3) uses this pattern to define a simple `bucketName` string as its `in` parameter (see the S3 API Console and SDK[8], the response message of the WSDL of S3 Web Service API[9] and the XML Schema for S3[10]).

In RESTful HTTP, URIs and in particular URI query strings can carry instances of this pattern (particularly for GET method requests). For instance, the Facebook Graph API[11] uses a single scalar `in` parameter `event-id` in its endpoint URI; it also provides an example of an *Atomic Parameter*.

The scalar value types[12] in/of messages created with the Protocol Buffers[13] data interchange format originally developed by Google and open sourced at GitHub[14] can also be seen as instances of this pattern. The same holds true for the primitive types in Apache Avro[15], which are serialized into JSON.

Related Patterns and References. This pattern refines the three patterns *Command Message*, *Document Message*, and *Event Message* in [Hohpe and Woolf 2003] by looking into the actual message content (payload). It can also be seen as a building block of its sibling representation pattern →*Atomic Parameter List* and may also appear as leaves of →*Parameter Trees* (which also introduce complex types and parameter nesting).

[7]http://apistylebook.com/design/topics/guiding-input
[8]http://docs.aws.amazon.com/AmazonS3/latest/dev/create-bucket-get-location-example.html
[9]http://doc.s3.amazonaws.com/2006-03-01/AmazonS3.wsdl
[10]http://doc.s3.amazonaws.com/2006-03-01/AmazonS3.xsd
[11]https://developers.facebook.com/docs/graph-api/reference/event
[12]https://developers.google.com/protocol-buffers/docs/proto3#scalar
[13]https://developers.google.com/protocol-buffers/
[14]https://github.com/google/protobuf
[15]http://avro.apache.org/docs/current/spec.html

The Web API specification language Swagger[16] has the notion of a *parameter object* to "describe a single operation parameter".

4.2 *Atomic Parameter List* Pattern

also known as: Multiple Scalar Representations

Context. API consumer and API provider have not yet agreed on a data contract between them; the structures of request and response messages have not been specified yet. In RESTful HTTP, the parameters and request/response MIME types of HTTP methods (verbs) still have to be defined; in Web services, the elements and parts of SOAP envelopes still have to be defined in WSDL and XML Schema.[17] More than one distinct information item might may have to be transmitted.

Problem. How can the API provider inform the API consumer about multiple primitive information items it expects in request messages and/or supports in response messages?

Forces. The data structures of the request and response messages are an essential part of the technical *service contract*. All patterns dealing with structural interface representation design and technical service granularity share the same high-level forces (as elaborated previously):

- Interoperability
- Performance (of network and endpoints)
- Processing effort (at development time and at runtime)
- Learning effort and maintainability (versioning and backward compatibility in particular)
- Security

In many scenarios, a single scalar piece of information is not sufficient to inform the provider about the consumer's need or to provide a meaningful response. The more information items are transmitted, the more significant the forces become (for instance, multiple parameters expose security threats, both in isolation and in combination).

One could send several messages each sending a single scalar →*Atomic Parameter*, but that would lead to chatty conversations and waste of network capacity. Such approach would also waste server-side resources if the server is stateless and has to restore the session state from the request message and protocol headers each time. One could also think of always sending and returning a complex data structure that leaves all fields empty that are not required in the current call; however, such approach would lead to unnecessary interoperability and evolution issues, increased testing efforts, and waste of processing capacity in the endpoints (because unnecessary, empty data structures have to be marshalled and unmarshalled).

Solution.

How it works. To transmit two or more simple, unstructured information items, define multiple →*Atomic Parameters* (e.g., numbers, strings, boolean values) in an ordered list. Populate and process these items separately in sender and receiver. Name these parameters in an expressive way and, optionally, order them logically for better human-readability (to indicate and promote high cohesion within a parameter and low coupling between parameters). Mark optional parameters as such; provide representative examples for the permitted combinations (i.e., instances of the parameter list).

[16]https://swagger.io/specification/

[17]This pattern and its sibling pattern →*Atomic Parameter* have very similar context sections and problem statements because they provide alternate solutions to the same general design concern.

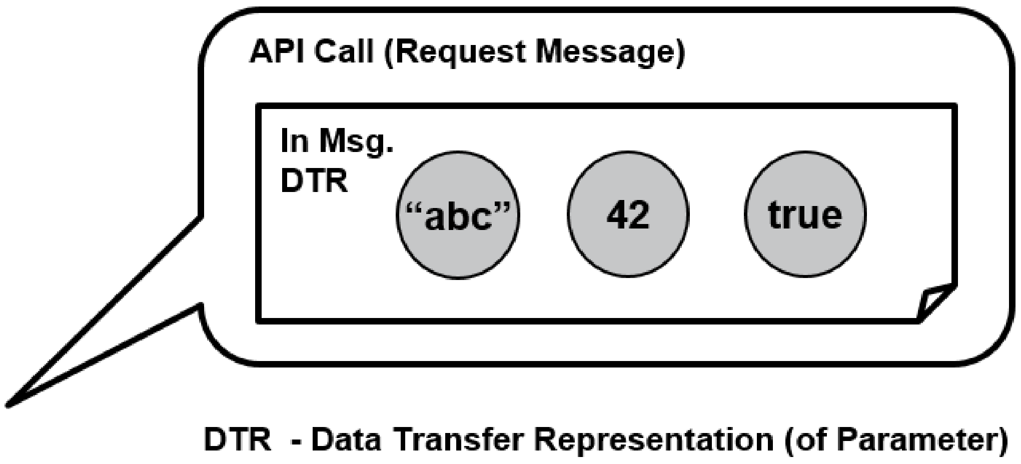
API Call (Request Message)
In Msg. DTR
"abc"
42
true
DTR - Data Transfer Representation (of Parameter)

Fig. 5. Iconic visualization of Atomic Parameter List pattern: multiple simple scalars are sent (here: in parameter of a request message)

Figure 5 sketches an application of the pattern in a request message. The DTR has three →*Atomic Parameter* entries, which form a list.

Multiple patterns and multiple instances of the same pattern can be applied in the same API call. For instance, in RESTful HTTP, both the URI (in particular, URI parameters) as well as form parameters can be used to realize the pattern. If both options are used in one API call, two *Atomic Parameter Lists* can be observed.

Some integration platforms do not allow the communication parties to send multiple scalars in a particular type of message, For instance, many programming languages only allow one return value or object (i.e., out parameter) in the method signatures, and the default mappings from these languages to JSON and XML schema follow this convention (e.g., JAX-RS and JAX-WS in Java). If this is the case, you can define a message-specific sequence (a.k.a. record, associative array) of scalars as a variant of this pattern, which may or may not require custom marshalling. This variant can also be seen as a special, primitive variant of the →*Parameter Tree* pattern that only uses scalar leaves underneath the root of the tree; therefore, we call it *Atomic Parameter Tree*.

Example. The in parameter of the `getRiskReport` operation in the insurance Web services example that we adopt from [Zimmermann et al. 2003] is an instance of this pattern:

```
<xs:complexType name="getRiskReportViaMultipleScalar">
    <xs:sequence>
      <xs:element name="startYear" type="xs:int"/>
      <xs:element name="numberOfYears" type="xs:int"/>
    </xs:sequence>
</xs:complexType>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
      <ns2:getRiskReportViaMultipleScalar xmlns:ns2="http://reports.insurance.irp.org/">
         <arg0>2015</arg0>
         <arg1>3</arg1>
      </ns2:getRiskReportViaMultipleScalar>
    </soap:Body>
</soap:Envelope>
```

The RESTful HTTP implementation of the risk management server[18], implemented with JAX-RS on the provider side, also applies this pattern in its request messages (and the →*Parameter Tree* pattern in its response messages):

```
GET /riskreport?firstYear=2017&noOfYears=1 HTTP/1.1

[...]

HTTP/1.1 200 OK
Date: Tue, 14 Feb 2017 08:52:29 GMT
Content-Type: application/json
Vary: Accept-Encoding
Content-Length: 102

[{"year":2017,"policyCount":42,
  "totalClaimValue":2000.0,"claimCount":1,
  "totalInsuredValue":1000000.0}]
```

Implementation hints and pitfalls to avoid. Adopters of this pattern should take the following advice into consideration:

- Minimize the size of the list and keep its elements consistent; make sure to include only scalars for which clients have a need (i.e., apply a use case- and consumer-driven API design approach).
- Avoid semantic dependencies between different scalars. The value of one scalar should not influence the meaning of another scalar with the exceptions of values and their units (e.g., `{"amount"=123.00, "currency"="EUR"}`). If such dependencies are inherent to the data exchanged (the published language), consider to switch to an implementation of the →*Parameter Tree* pattern.
- Define a security protection level for each parameter and for the entire message (or API call) and design security means that respond to related security threats in risk- and cost-driven manner.

The API Stylebook[19] collects additional nuggets of advice, e.g., about consumer input (for in parameters in request messages).

Discussion. The solution balances the information need of the receiver with the desire to minimize processing and communication overhead; since it uses base types, it has good interoperability characteristics in most integration platforms and paradigms (e.g., WSDL/SOAP Web services, RESTful HTTP). In some platforms, this pattern cannot be realized in its pure form, or its realization does not differ significantly from its sibling pattern →*Parameter Tree* (e.g., if there is no notion of request message parts or if there is a single response data element only, see introduction of *Atomic Parameter Tree* variant above). In some technologies and platforms, such atomic parameter trees can be marshalled automatically based on reflection; in others they require a customer marshaller.

[18]https://github.com/web-apis/riskmanagement-server
[19]http://apistylebook.com/design/topics/guiding-input

In some technologies, such as Web Sockets, there is no type system to be assumed (such as XML Schema or MIME types); consumer and provider have to agree on data marshalling/unmarshalling (details of which are out of scope here).

From a security standpoint, usage of this pattern may cause more design and processing work than its →*Atomic Parameter* sibling; this depends on the security classification (level) of the different parameters and their *semantic proximity* as discussed in [Gysel et al. 2016].

Known Uses. Attaching multiple URI parameters to an HTTP GET method call in RESTful HTTP can be viewed as an instance of this pattern, as well as *URI Templates* as defined in RFC 6570[20]. Multiple plain parameters in an HTTP POST request also qualify as pattern instances.

Facebook's Graph API to GET information about events per user[21] applies this pattern in its response message.

Twitter heavily uses this pattern in its API, for example to post a new tweet[22].

```
POST
https://api.twitter.com/1.1/statuses/update.json?status=Hi%20there&lat=47.2266&lon=8.8184
```

Messages created with the Protocol Buffers[23] data interchange format (originally developed by Google and open sourced at GitHub[24]) that only contain simple data types as field value types can also be seen as instances of this pattern.

Swagger[25] has the notion of a *parameters definitions* object to "to hold parameters to be reused across operations".

Related Patterns and References. This pattern has three siblings →*Atomic Parameter*, →*Parameter Tree*, →*Parameter Forest*; these interface representation patterns continue the coverage of *Command Message*, *Document Message*, *Event Message* in [Hohpe and Woolf 2003] by discussing the syntactic structure of the content of such messages. The →*Atomic Parameter* pattern can be seen as a simpler alternative to this pattern, but also as its building block. Once an →*Atomic Parameter List* becomes too large, switching to an (atomic) →*Parameter Tree* often is the next step in the evolution of an API call and its representations.

4.3 *Parameter Tree* Pattern

also known as: Single Complex Representation, Tree Representation, Bar

Context. A simple message format has been defined, consisting of one or more scalars that form an →*Atomic Parameter* or an →*Atomic Parameter List*. However, this simple message format does not fully satisfy the information need of the message receiver (i.e., the provider for request messages and the consumer for response messages), or contains undesired semantic dependencies between the list elements.

Problem. This pattern solves a variation of the general representation design problem also addressed by its sibling patterns →*Atomic Parameter* and →*Atomic Parameter List*:

How do you exchange repetitive or nested data between consumer and provider in a message-based remote API? For instance, how to include such data in the `in` and `out` messages of WSDL/SOAP Web

[20]https://tools.ietf.org/html/rfc6570
[21]https://developers.facebook.com/docs/graph-api/reference/event/
[22]https://dev.twitter.com/rest/reference/post/statuses/update
[23]https://developers.google.com/protocol-buffers/
[24]https://github.com/google/protobuf
[25]https://swagger.io/specification/%3E

service operations and the parameters and body of requests/responses in RESTful HTTP (e.g., GET, POST, PUT)?

Forces. All representation patterns dealing with technical service granularity and structural data contract design share a common set of top-level forces (whose significance increases when the representation structures become more complex):

- Interoperability
- Performance (latency in particular)
- Processing effort at development time and at runtime
- Learning effort and maintainability (versioning and backward compatibility in particular)
- Security is a concern of increasing importance as more data is exchanged (data privacy level, attribute-based access control)

If repetitive or nested data is to be transmitted, the number of data items and its nesting depth are particularly relevant; expressiveness and efficiency have to be balanced.

One could send several messages each sending a single or multiple scalar parameters, as described in the →*Atomic Parameter* and →*Atomic Parameter List* patterns, but that would lead to chatty conversations and waste of network capacity if the information need of the message receiver exceeds the expressiveness of such simple data formats (e.g., query results usually are repetitive and may consist of structured information). It also runs the risk of violating the loose coupling principle due to the semantic dependencies between calls and call parameters.

Solution.

How it works. Define a single root that contains one or more subordinate composite/aggregate data structures such as *tuples* or *arrays* as available in the concrete syntax used as the message exchange format (e.g., JSON objects or XML complex types using sequences). A tuple assembles data of different types (e.g., the ZIP code and the name of a city in an address). An array is typically used if all elements to be aggregated share the same structure; in JSON arrays are a first-class citizen of the notation, and in XML repetitive sequences can be defined via cardinalities of complex type sequences higher than 1 (e.g., `maxOccurs="unbounded"`). To structure the subordinate further, the pattern can be applied recursively to create nested structures (if this can be justified in the domain model/data to be exchanged).

Figure 6 sketches two different applications of the pattern, one in a request message and one in a response message.

If all tree leaves on level 1 are →*Atomic Parameters*, the resulting Data Transfer Representation (DTR) structure and pattern variant is called *Atomic Parameter Tree*.

Example. Our JAX-RS example available at GitHub[26] uses an instance of this pattern when returning updated claims in response to HTTP PUT requests:

```
public class ClaimDTO {
    private final UUID id;
    private final String dateOfIncident;
    private final double amount;
    private final List<Evidence> evidence;
    [...]
```

[26]https://github.com/web-apis/riskmanagement-server

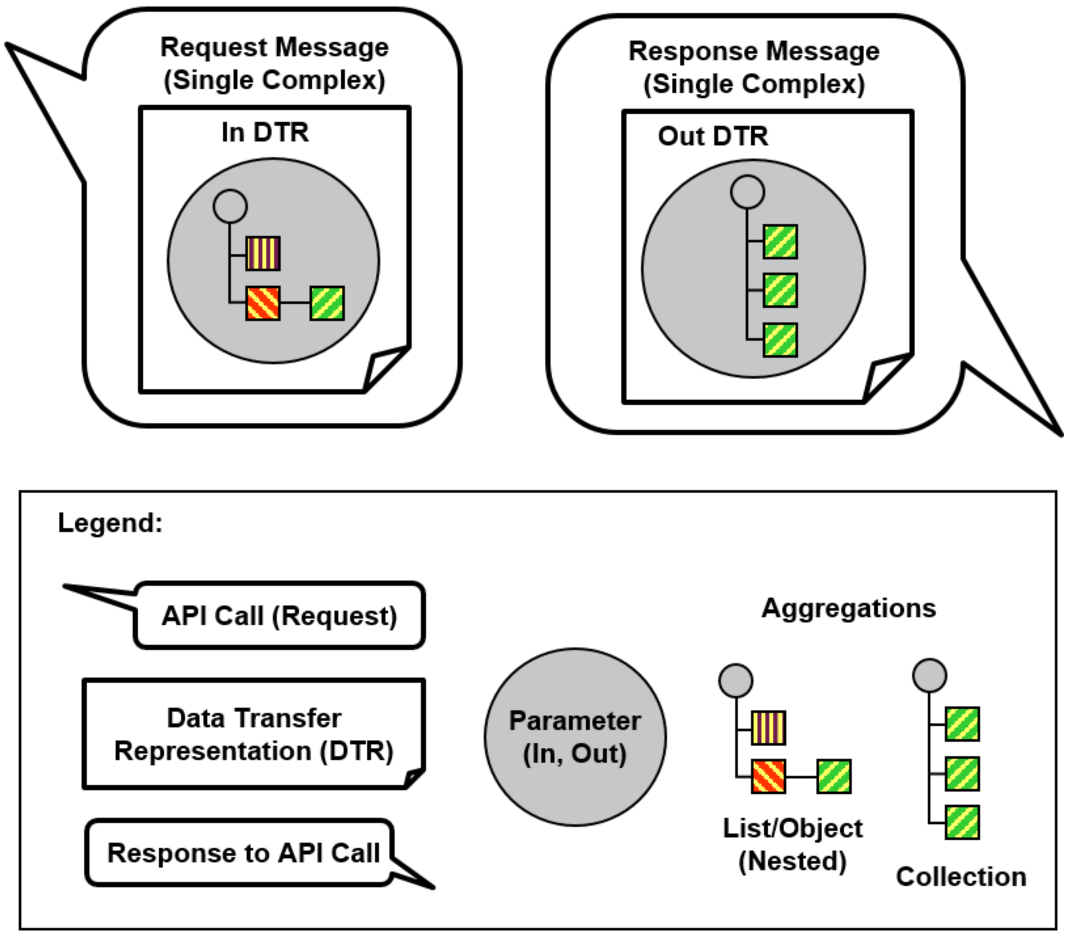

Fig. 6. Parameter Tree pattern with two variations, nested list and homogeneous, flat collection (iconic visualization)

```
}

@PUT
@Path("/{claimId}")
public ClaimDTO updateClaim(@PathParam("claimId") UUID claimId,
                            @NotNull @Valid Claim claim) {
    boolean result = claims.update(claim);
    if (!result) {
        throw noSuchClaim.get();
    }
    return ClaimDTO.create(claim);
}
```

A sample GET request and instance of the `ClaimDTO` returned to the consumer looks like this:

```
GET http://localhost:8000/claims/0afeb849-6d63-40b6-b52f-21dee16fdda5
```

```
{"claim":
  {"id":"0afeb849-6d63-40b6-b52f-21dee16fdda5",
   "dateOfIncident":"2017-02-14",
   "amount":2000.0,
   "evidence":[],
   "links":[{"uri":"http://localhost:8080/claims/0afeb849-6d63-40b6-b52f-21dee16fdda5",
            "params":{"rel":"self"},
             "type":null,
             "rel":"self",
            "uriBuilder":{"absolute":true},
            "rels":["self"],
            "title":null}]}}
```

The `ClaimDTO` with its attributes constitutes the *Parameter Tree* in the example. One of these attributes (`evidence`) applies the pattern again. The resulting Parameter Tree is marshalled into a JSON object that contains an array (which is empty in the above JSON/HTTP snippet).

Implementation hints and pitfalls to avoid. When applying and realizing this pattern, the following advice should be taken into consideration (note that similar hints apply to the sibling patterns):

- "Be liberal in what you accept and conservative in what you do/sent" according to J. Postel's robustness principle in RFC 761[27] for network protocol implementations, which (like all principles) should be applied with a sense of pragmatism in the given context [Allman 2011]; validate outgoing data according to a schema as/if defined by concrete syntax in use and validate incoming data as lax as possible while still making sure that the request can be processed successfully.
- Specify the upper and lower boundaries, e.g., for arrays and elements of sequences; be explicit about NULL values and optionality (just like when using more basic structures uch as →*Atomic Parameters*).
- Resist the temptation to represent the real world exactly and completely (with all variations and exceptions modeled explicitly) to minimize message verbosity; the general "if in doubt, leave it out" rule for modeling (and other specification efforts) also applies to data modeling and interface representation design. Consider using compression (as for instance supported in HTTP2). Avoid overly deep nesting of data structures unless minimizing the number of calls has high priority (e.g., in a mobile applications) and the API consumer is expected to follow domain model links to reference data anyway (e.g., a customer is referenced in a contract or a purchase, and customer details have to be displayed)
- Be reluctant to introduce fully generic data structures (e.g., key-value pairs to be built and interpreted dynamically); the promised flexibility might backfire and cause difficulties in interface comprehension by the developers that lead to additional debugging and testing efforts in the long run. Flexibility comes at a price; domain-specific abstractions and names make tools such as code completion and test automation more powerful.
- When dealing with an Object-Oriented (OO) domain model in the service implementation (backend), consider using an OO-to-XML mapper (but conduct a thorough proof-of-technology before deciding for one strategically).

[27]https://tools.ietf.org/html/rfc761

While technically possible and suggested/considered by E. Wilde in a blog post[28][29], a *Parameter Tree* contained in the request message of an HTTP GET is not supposed to have any effect on the provider and is therefore useless according to the HTTP/1.1 specification (see a related Stack Overflow discussion[30][31]).

Discussion. Like the →*Atomic Parameter List* pattern, this solution balances the information need of the client with the desire to minimize processing and communication overhead; it is more expressive since it can be applied recursively. It is also more cohesive to due the presence of the single root. Learning and processing effort depend heavily on detailed design (e.g., depth and breadth of the tree) and integration platform in use: For instance, in many frontend development Software Development Kits (SDKs) such as those used to write mobile applications, JSON array processing is a native platform capability that requires very little programming effort.

When being combined with the sibling pattern →*Parameter Forest*, highly expressive DTRs can be created if complex or advanced information needs have to be satisfied. Such structures that might be complex to process. Tree navigation comes in as an additional challenge; the breadth and the depth of the data structures have to be decided carefully. Bloated data structures increase processing overhead and might waste network capacity. For instance, data that is not needed by the client to perform its task (e.g., to realize a certain user story) but still included in the data structure is unnecessarily sent over the wire.

From a security standpoint, it is good that only a single data structure has to be analyzed and possibly secured; however, its content may contain data with different protection needs (e.g., person name and credit card number), which complicates the task of providing field-level security means (e.g., attribute-based access control, encryption). Selective or declarative attribute-based access control might be necessary to satisfy security requirements such as data privacy (confidentiality) and data integrity (no tampering) as these requirements might differ by sub-tree or individual tree nodes.

Many additional data structures exist in XML schema, JSON schema and programming languages (e.g., vectors, hash maps, and associative arrays); in order to minimize platform coupling, one should stay away from proprietary and overly complex data structures (assuming explicit static typing here).

If the structure of the *Parameter Tree* (e.g., the amount and/or size of array entries and/or the number of elements in record structures) increases beyond about five to seven top-level elements, consider splitting the tree structure and apply the →*Parameter Forest* pattern. You may also consider →*Pagination* for selected sub-trees that have a repetitive structure; this might require refactoring the API into several calls.

Known Uses. When JAX-RS is used to implement a message-based remote API, `@Consumes` and `@Produces` annotations that refer to custom media types (e.g., nested JSON objects with a single root) may indicate instances of this pattern.[32]

The JIRA Cloud REST APIs[33] use this pattern in the requests of its `issue-createIssue`[34] call; note that it uses an →*Atomic Parameter List* for the corresponding responses. The JIRA Cloud REST API

[28]http://dret.typepad.com/dretblog/2007/10/http-get-with-m.html
[29]http://dret.typepad.com/dretblog/2007/10/http-get-with-m.html
[30]http://stackoverflow.com/questions/978061/http-get-with-request-body
[31]http://stackoverflow.com/questions/978061/http-get-with-request-body
[32]See this online tutorial for examples: http://www.mkyong.com/webservices/jax-rs/integrate-jackson-with-resteasy/. Note that the tutorial does not feature all platform-specific design guidelines and recommended practices for REST (e.g., the URI should not include an action code in verb form because this action code is already given by the HTTP methods such as GET and POST).
[33]https://docs.atlassian.com/jira/REST/cloud/
[34]https://docs.atlassian.com/jira/REST/cloud/#api/2/issue-createIssue

also uses concepts such as expansion and pagination, which are covered elsewhere in our pattern language.

The API call GET collections/list in the Twitter REST API contains a single nested JSON object called "objects" with subordinates that list users, timelines, etc.

The messages created with the Protocol Buffers[35] data interchange format (originally developed by Google and open sourced at GitHub[36]) can also be seen as instances of this pattern. The same holds true for the complex types in Apache Avro[37], which are serialized into JSON.

Related Patterns and References. This pattern has three siblings: *→Atomic Parameter*, *→Atomic Parameter List*, *→Parameter Forest*, all refining *Command Message*, *Document Message* and *Event Message* from [Hohpe and Woolf 2003]. An *→Atomic Parameter List* can be refactored into a *Parameter Tree* when it becomes too complex; a *→Parameter Forest* consists of multiple *→Parameter Trees*. *Content Enrichers* and *Content Filters* as described in [Hohpe and Woolf 2003] operate on *→Parameter Trees*.

A similar pattern called *Single Message Argument*[38] appears in the Service Design Patterns book by [Daigneau 2011] (making the point that parameter order should not be determined by the communications infrastructure).

Abstract Data Types are a closely related general concept in computer science.

4.4 *Parameter Forest* Pattern

also known as: Parameter Comb, Hybrid Parameter List

Context. An API endpoint such as a REST resource or WSDL/SOAP port has been defined and its calls (e.g., HTTP methods, WSDL operations) have been specified initially. Requirements engineering and business domain analysis efforts have unveiled that a rich set of information has to be exchanged to process the API call (service operation/method) successfully.

Problem. How do you exchange rich repetitive or nested data between consumer and provider in a message-based remote API?[39] For instance, how do you exchange such deeply structured data between message sender and message receiver in a SOA message exchange?

Forces. General forces for structural representation design were discussed on the category level, and the forces section of the *→Parameter Tree* pattern covered additional forces that apply for more complex data structures (assembled from atomic ones such as strings and integers, as well as other complex ones):

- Interoperability
- Performance (latency in particular)
- Processing effort at development time and at runtime
- Learning effort and maintainability (versioning and backward compatibility in particular)
- Security is a concern of increasing importance as more data is exchanged (data privacy level, attribute-based access control)

[35] https://developers.google.com/protocol-buffers/
[36] https://github.com/google/protobuf
[37] http://avro.apache.org/docs/current/spec.html
[38] http://www.servicedesignpatterns.com/WebServiceEvolution/SingleMessageArgument
[39] Another reason might be that you want to stress test the message processors in endpoints (JSON, XML, other).

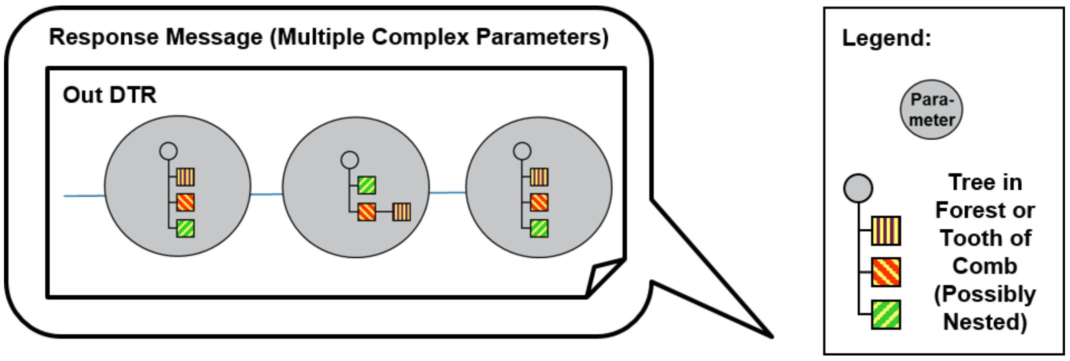
Response Message (Multiple Complex Parameters)
Out DTR
Legend:
Para-meter
Tree in Forest or Tooth of Comb (Possibly Nested)

Fig. 7. Parameter Forest pattern: message anatomy in iconic representation (here: response message consisting of three trees)

One could send several messages each sending a single complex parameter, as described in the →*Parameter Tree* pattern, but this might still not be sufficient to satisfy the information need of the message receiver.

Solution.

How it works. Send multiple simple and/or composite/aggregate data structure representations such as tuples, arrays or other complex types defined in the concrete message exchange format that has been decided upon (e.g., JSON or XML). Each of these structures qualify either as →*Atomic Parameters* or as →*Parameter Trees*.

Figure 7 sketches an applications of the pattern (in a response message).

Example. The following interface demonstrates all four basic representation patterns using alias names for the four patterns: `Dot` (for →*Atomic Parameter*), `Dotted Line` (for →*Atomic Parameter List*), `Bar` (for →*Parameter Tree*), and `Comb` (for *Parameter Forest*):

```
@WebService
public interface IRPService {
  boolean dotInDotOut(int singleScalarParameter);
  int dottedLineInDotOut(String scalarParameter1, String scalarParameter2);
  ResponseDTO barInBarOut(RequestDTO singleComplexParameter);
  ResponseDTO combInBarOut(RequestDTO complexParameter1,
                           AnotherRequestDTO complexParameter2);
}

public class RequestDTO {
  private float value;
  private String unit;

  [...]
}

public class AnotherRequestDTO {
  private int id;
  private NestedRequestDTO[] toothOfComb;
  [...]

public class ResponseDTO {
  private int id;
  private String dataField1;
  private String dataField2;

  [...]
}

<xs:complexType name="combInBarOut">
    <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="tns:requestDTO"/>
      <xs:element minOccurs="0" name="arg1" type="tns:anotherRequestDTO"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="requestDTO">
    <xs:sequence>
      <xs:element minOccurs="0" name="unit" type="xs:string"/>
      <xs:element name="value" type="xs:float"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="anotherRequestDTO">
    <xs:sequence>
      <xs:element name="id" type="xs:int"/>
      <xs:element maxOccurs="unbounded" minOccurs="0"
        name="toothOfComb" nillable="true" type="tns:nestedRequestDTO"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="nestedRequestDTO">
    <xs:sequence>
      <xs:element minOccurs="0" name="key" type="xs:string"/>
      <xs:element minOccurs="0" name="valie" type="xs:string"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="responseDTO">
    <xs:sequence>
      <xs:element minOccurs="0" name="dataField1" type="xs:string"/>
      <xs:element minOccurs="0" name="dataField2" type="xs:string"/>
      <xs:element name="id" type="xs:int"/>
    </xs:sequence>
</xs:complexType>
```

The `in` message of the `combInBarOut` call has two parameters each forming one "tooth" of the "comb" (in the metaphor); both of these parameters are further structured into Data Transfer Objects (DTOs).

Implementation hints and pitfalls to avoid. Architects and developers that decide to apply and realize this pattern should take the following advice into consideration (which picks up on that given for →*Parameter Trees*):

- Limit the number of trees and leaves in the forest to a few (say three to five) unless minimizing the number of calls has high priority (e.g., in certain mobile applications) and the API consumer is expected to follow domain model links to reference data anyway (e.g., a customer is referenced in a contract or a purchase, and customer details have to be displayed).[40]
- Just like for all nontrivial data structures (including standalone →*Parameter Trees*), provide sample data and test cases, but also machine-readable specifications such as schemas (e.g., using JSON Schema or XML Schema) to support regression testing and long-term maintenance.
- Be careful with optional "spikes" (i.e., trees degenerated to leaves or holes a.k.a. teeth in the comb structure); particularly if placed somewhere in the middle (because such definitions complicate the unmarshalling and security policy processing); in extreme cases, ambiguities and misinterpretations of the data may occur (leading to incorrect processing results and audit failures).

If the length of the *Parameter Tree* (or the size of the array/the number of elements in the record structure) gets high, consider applying →*Pagination*.

Discussion. This pattern has similar forces resolution characteristics as its sibling pattern →*Parameter Tree*, but takes the content structuring one step further by listing multiple scalars and/or trees as message parameters. In some technologies and platforms such as JAW-WS, the applications of these two patterns are hard to distinguish (depending on the way request and response messages are realized in these platforms). Some readers might remember the `rpc/encoded vs. (wrapped) document/literal` discussions in the early days of Web services [Zimmermann et al. 2003]: A wrapped document/literal SOAP envelope qualifies as instance of →*Parameter Tree* (single root), while the `rpc/encoded` style can also create in messages that contain *Parameter Forests*.

Depending on the breadth and depth of the trees in the forest (or "teeth" of the "comb" in the metaphor used in the alias name), performance might be poor when applying this pattern (just like its sibling →*Parameter Tree*); consider to simplify the structure via refactoring and/or applying performance improvement concepts such as *expansion* as demonstrated by the JIRA Cloud REST API[41] in such cases.

Complex data structures are harder to maintain in public APIs than simple ones; once exposed, existing contracts should not be broken to avoid an unnecessary coupling between consumers and providers from a deployment and evolution perspective.

Complex data structures are also harder to secure against tampering and other security threats; on the other hand, dedicated fields can be defined that contain signatures or public key information. If this is done, the pattern can also be seen as a security enabler/facilitator, e.g., with one branch of a tree (or a "tooth" of the comb) being dedicated to security information.

Known Uses. The response messages of the core banking integration solution described in [Brandner et al. 2004] use this pattern by providing domain-specific complex types in XML Schema (XSD).

[40]Another reason might be that you want to stress test the message processors in endpoints (JSON, XML, other).
[41]https://docs.atlassian.com/jira/REST/cloud/

The Flickr App Garden[42] uses combs in multiple calls, e.g., in the responses of its read access to collections: `flickr.collections.getInfo`[43]. Note that the Flickr API and App Garden also use the →*Atomic Parameter List* pattern, e.g. for `Upload Photos`[44]. The API supports multiple message exchange formats in its request and response messages, including SOAP, RESTful mime/media types, and even XML-RPC.

The response structure used in the Twitter REST API also qualifies as a *Parameter Forest*, with one tree containing an array of objects and a second one containing control information and metadata such as cursors and page tokens (see →*Pagination* pattern).[45]

A message created with the Protocol Buffers[46] data interchange format, originally developed by Google and open sourced at GitHub[47], that contains another message that is tagged with the keyword `repeated` can also be seen as an instance of this pattern.

JSON API[48] responses also qualify as instances of this pattern, with three mandatory members (data, errors, meta) and three optional ones (jsonapi, links, included). Each one is a →*Parameter Tree*.

Related Patterns and References. This pattern refines *Command Message* and *Document Message* from [Hohpe and Woolf 2003] . The pattern can utilize its sibling pattern →*Parameter Tree* to create complex, deeply nested structures; however, some inhabitants of the forest (or "teeth" of the comb in the alias name) might also be →*Atomic Parameters* (scalars).

A *Parameter Forest* can be refactored into a →*Parameter Tree* by introducing a single root when it becomes too complex to prepare and process. If this happens at runtime rather than design time, a *Splitter* can be used ([Hohpe and Woolf 2003]). As discussed already, a →*Parameter Tree* can not only be seen as an alternative to a *Parameter Forest*, but also as a building block of instances of this pattern.

4.5 *Pagination* Pattern

also known as: Query with Partial Result Sets, Response Sequence

Context. API consumers often query (retrieve) data to display to the user or to be processed in other applications. When processing such a consumer query, which may include query parameters, the API provider often has to respond with a large data set that consists either of identically structured data elements (e.g., rows fetched from a relational database or line items in a batch job executed by an enterprise information system in the backend) or of heterogeneous data not adhering to a common schema (e.g., parts of a document from a document-oriented NoSQL database such as MongoDB).

Problem. How can a provider progressively return large amounts of repetitive or inhomogeneous data (in response to a consumer enquiry) if this data does not fit well in a single message?

Forces. Key design criteria when dealing with large amounts of repetitive response data include:

- Data set size and data access profile (user needs), especially number of data records required to be available to a consumer (immediately and over time)
- Variability of data: Are all result elements identically structured? How often do data definitions change?

[42]https://www.flickr.com/services/api/
[43]https://www.flickr.com/services/api/flickr.collections.getInfo.html
[44]https://www.flickr.com/services/api/upload.api.html
[45]https://dev.twitter.com/rest/collections/responses.
[46]https://developers.google.com/protocol-buffers/
[47]https://github.com/google/protobuf
[48]http://jsonapi.org/format/

- Memory available for a request (both on provider and on consumer side) and data currentness requirements vs. change dynamics
- Network capabilities (server topology, intermediaries)
- Security and robustness/reliability concerns

Especially when returning data for human consumption, not all data may be needed immediately. Network and endpoint processing capabilities should be used efficiently, but all results transferred and processed accurately (consistently). A single large response message might be inefficient to exchange and process.

Common text-based message exchange formats (e.g., expressively tagged XML, but also JSON) incur high parsing cost and transfer data size due to verbosity and overhead of the textual representations of the data. Some of this can be significantly reduced by using compact binary formats such as Apache Avro, Protocol Buffers, etc. However, many of these formats require dedicated marshalling libraries which may not be available in all consumer environments, for example Web browsers.

Underlying network transports such as IP networking transport data in packets, which leads to non-linear transfer times with data size. For example, 1500 bytes fit into a single IP packet transmitted over Ethernet.[49] As soon as the data is one byte longer, two separate packages have to be transmitted and coordinated on the receiver side.

Retrieving and encoding large data sets can incur high effort/cost on the provider side and can open up an attack vector for a denial-of-service attack. Moreover, transferring large data sets across a network can lead to interruptions as most networks are not guaranteed to be reliable, especially cellular networks.

One could think of sending the entire large response data set in a single response message, but such simple approach might waste endpoint and network capacity; it also does not scale well. Sending a data query, which can result in a result set whose size is unknown in advance, can be too large to be processed on the consumer/client or the provider/server side. Without mechanisms to limit such queries, processing errors such as out-of-memory exceptions may occur and the client or the endpoint implementation may crash. Developers and API designers often underestimate the memory requirements imposed by unlimited query contracts. These problems often go unnoticed until concurrent workload is placed on the system and/or the database size increases. In shared environments, it is possible that unlimited queries cannot be processed efficiently in parallel, which leads to similar performance, scalability, and consistency issues – only combined with concurrent requests which are hard to debug and analyze anyway.

Solution.

How it works. Divide large response data sets into manageable and easy-to-transmit chunks (“pages”). Send only partial results in the first response message and use metadata and semantic links (“hypermedia”) to inform the consumer how additional results can be obtained/retrieved incrementally. The page size or limit, i.e., the number of data elements in a chunk, can be either a fixed size (which is part of the service contract) or can be specified by the consumer as part of the request. Process some or all partial responses on the consumer side iteratively as needed; agree on a request correlation and intermediate/partial results termination policy (possibly requiring session state management). Inform the client about the total and remaining number of elements in the result set. Provide optional filtering capabilities. Allow consumers to request a random selection from the result set.

[49] Source: https://en.wikipedia.org/wiki/Maximum_transmission_unit

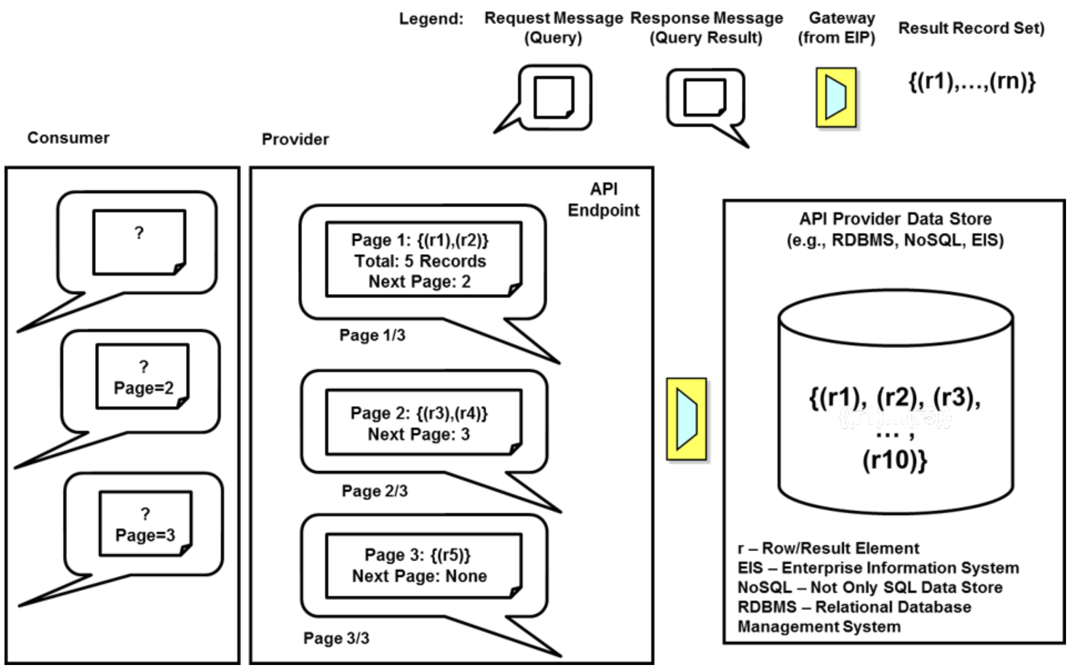
Legend:
Request Message (Query)
Response Message (Query Result)
Gateway (from EIP)
Result Record Set)
{(r1),...,(rn)}
Consumer
Provider
API Endpoint
?
? Page=2
? Page=3
Page 1: {(r1),(r2)} Total: 5 Records Next Page: 2
Page 1/3
Page 2: {(r3),(r4)} Next Page: 3
Page 2/3
Page 3: {(r5)} Next Page: None
Page 3/3
API Provider Data Store (e.g., RDBMS, NoSQL, EIS)
{(r1), (r2), (r3), ... , (r10)}
r – Row/Result Element
EIS – Enterprise Information System
NoSQL – Not Only SQL Data Store
RDBMS – Relational Database Management System

Fig. 8. Pagination: query and follow on request messages, response messages with filtered, partial result sets (pages)

Figure 8 visualizes a single instance of the pattern that results in a stateful conversation between the communication parties. The server-side data storage (backend) is also shown; it is integrated with the help of an Application Gateway here, one of the 65 Enterprise Integration Patterns (EIP) [Hohpe and Woolf 2003].

The pattern has variants such as *Offset-Based Pagination*, *Cursor-Based Pagination* (also known as *Token-Based Pagination*) and *Time-Based Pagination*, which differ in the way the consumer requests partial results and navigates from chunk to chunk. For instance, the consumer by default may request a certain "page number" to specify which data chunk the provider should return. Alternatively, the consumer might be permitted to specify an offset, i.e., how many single elements to skip.

The default page-based *Pagination* and its *Offset-Based Pagination* variant are quite similar; in most cases the basic pattern and its variant can be used interchangeably. Offset-Based Pagination is more flexible when the number of requested results/the page size changes. If done incorrectly, enlarging the page size while keeping the page index the same can cause missed entries. For example, when the requested page size doubles, the page index must be halved to stay on the same page.

Cursor-Based Pagination and *Time-Based Pagination* are similiar in that they do not rely on an element's index. In *Time-Based Pagination*, the chunk size and the population of individual chunks is driven by additional metadata to increase data currentness/liveness/freshness of the paginated responses.

Example. The insurance claim processing/reporting example[50] illustrates the Pagination pattern in its claims queries:

```
curl http://localhost:8080/claims?limit=10\&offset=0
```

[50]https://github.com/web-apis/riskmanagement-server

```
    @GET
    public ClaimsDTO listClaims(@DefaultValue("3") @QueryParam("limit") Integer limit,
            @DefaultValue("0") @QueryParam("offset") Integer offset,
            @QueryParam("orderBy") String orderBy) {

        List<ClaimDTO> result = [...]

        return new ClaimsDTO(limit, offset, claims.getSize(), orderBy, result);
    }
```

Besides limits and offset parameters, the `ClaimsDTO` class also shows how HATEOAS-style link relations [Allamaraju 2010] can be generated using Jersey annotations:

```
public class ClaimsDTO {
    private final int limit, offset, size;
    private final String orderBy;
    private final List<ClaimDTO> claims;

    public ClaimsDTO(int limit, int offset, int size, String orderBy,
      List<ClaimDTO> claims) {
        super();
        this.limit = limit;
        this.offset = offset;
        this.size = size;
        this.orderBy = orderBy;
        this.claims = claims;
    }

    @InjectLinks({
      @InjectLink(resource = ClaimManagement.class, method = "listClaims",
        style = Style.ABSOLUTE,
        bindings = {
            @Binding(name = "offset", value = "${instance.offset}"),
            @Binding(name = "orderBy", value = "${instance.orderBy}"),
            @Binding(name = "limit", value = "${instance.limit}") }, rel = "self"),

    @InjectLink(resource = ClaimManagement.class, method = "listClaims",
        style = Style.ABSOLUTE,
        condition = "${instance.offset + instance.limit < instance.size}",
        bindings = {
            @Binding(name = "offset", value = "${instance.offset + instance.limit}"),
            @Binding(name = "orderBy", value = "${instance.orderBy}"),
            @Binding(name = "limit", value = "${instance.limit}") }, rel = "next"),

    @InjectLink(resource = ClaimManagement.class, method = "listClaims",
        style = Style.ABSOLUTE,
        condition = "${instance.offset - instance.limit >= 0}",
        bindings = {
            @Binding(name = "offset", value = "${instance.offset - instance.limit}"),
            @Binding(name = "orderBy", value = "${instance.orderBy}"),
            @Binding(name = "limit", value = "${instance.limit}") }, rel = "prev") })

    private List<Uri> links;

    [...]
}
```

The JSON API[51] specification provides additional pagination examples.

Implementation hints and pitfalls to avoid. Architects and developers that decide to apply and realize *Pagination* should take the following advice into consideration:

- The pattern should be used consistently throughout an API so that API consumers do not need to learn multiple API styles and parameter sets. If consumers make incorrect assumptions because of inconsistent use of the pattern, they will surprised by the results of their service invocations.
- The maximum page size should be carefully chosen, especially if the loaded data is hold in memory before it is written to the output message, which is the standard implementation in most XML and JSON frameworks (session state management is required but difficult to scale and maintain). The possibility of introducing a *Discrete Web Data Stream* as an alternative to, or variant of, this pattern should be investigated.
- Pagination is of limited utility if the service implementation does not take advantage of it and still fetches all data from the database, e.g., SQL `LIMIT` clauses should be used in the case of relational database access.
- Pagination should be deterministic so that fetching the next page really fetches a different set of records. SQL's `ORDER BY` clauses are one possible way to achieve this (in server-side implementations of the pattern). If the data can change while the user is paging through it, offset-based pagination can cause entries to be shown twice (if an entry is added before the current location) or missed (if an entry is deleted before the current location). In such cases, a time- or token based approach should be used to provide a robust view to the client.
- "Implement consistent pagination by providing links to additional pages that are timestamped or versioned, such that you will never see duplicate results in pagination requests even if the objects involved change."[52]
- All platform-specific design guidance should be adhered to, e.g., proper URIs be defined in RESTful HTTP and hypermedia be used as the engine of application state (here: navigation within the result set).[53]

Discussion. Delivering one page at a time allows the consumer to process a digestible amount of data; a specification of which page to return facilitates remote navigation directly within the data set. Less endpoint memory and network capacity are required to handle individual pages, although some overhead is introduced because pagination management is required (see below).

The application of Pagination leads to additional design concerns:

[51] http://jsonapi.org/examples/#pagination
[52] Source: https://mathieu.fenniak.net/the-api-checklist/
[53] See for instance this online REST API tutorial http://www.restapitutorial.com/lessons/restfulresourcenaming.html.

- Where, when, and how to define the page size (i.e., the number of data elements per page)? This influences the chattiness of the API (in terms of number of messages and message size).
- How to order results, i.e., how to assign them to pages and how to arrange the partial results on these pages?
- Where and how to store intermediate results, and for how long (deletion policy, timeouts)?
- How to deal with request repetition; for instance, do the initial and the subsequent requests have to be idempotent to prevent certain errors and inconsistencies?
- How to correlate partial responses (with the original request, with the previous and the next partial response)?

Additional design concerns include the caching policy (if any), the liveness (currentness) of results, filtering, as well as query pre- and postprocessing (e.g., aggregations, counts, sums). Common data access layer concerns (e.g., isolation level, locking in relational databases) come into play here as well [Fowler 2002]. Consistency requirements differ by client type and use case: Is the client (end user) aware of the pagination (i.e., virtual pagination vs. technical pagination)? The resolution of these concerns is context specific; for instance, frontend representations of search results or product auctions in vertical integration of public Web applications differ from bulk/batch master data replication in backend-to-backend integration of enterprise information systems.

A correlation identification scheme is required so that the client can distinguish the partial results of multiple queries in arriving response messages [Hohpe and Woolf 2003].

More than a single pattern is required to solve the pagination problem; future pattern mining and writing work is required to address the above concerns.

A negative consequence is that pagination required more programming effort on the consumer side. Sometimes this can be annoying for users (here: developers consuming the API), as they have to "click through" even if there are a few results only, and they are not able to search the entire result set. All functions requiring a full record set like searching don't work (well) with *Pagination* or require extra effort (such as intermediate data structures on the consumer side).

Known Uses. The roots of the pattern and its name go back to plain Web page design, e.g., when displaying search or other query results on a series of linked Web pages. An early SOA and Web services production references that uses Pagination is [Brandner et al. 2004]. While not being message-based, remote JDBC applies sophisticated pagination concepts via its Result Set[54] abstraction.

Many public Web APIs use *Pagination*; typically both Page-/Offset-Based and Cursor-Based Pagination are supported while the Time-based Pagination variant is less common. For example, Google's search results are paginated as well as GitHub's Query API. Atlassian also features the concept of pagination explicitly and prominently in its JIRA Cloud REST APIs[55]. Regarding correlation, the Twitter REST API[56] is an interesting example because the timeline often changes, simple page/offset therefore does not work that well. Instead, a `since_id=12345` →*Atomic Parameter* can be used to only retrieve tweets that are more recent than the specified `id`.

A Swiss software vendor specializing on the insurance industry describes two variants of Pagination (page-based, offset-based) in its internal REST API Design Guidelines. Sorting and filtering of collection records is supported via operators that travel as HTTP parameters that contain control metadata.

[54]https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html
[55]https://docs.atlassian.com/jira/REST/cloud/
[56]https://dev.twitter.com/rest/public/timelines

An online API Stylebook website[57] lists eleven Web APIs and/or API design guideline books/websites that discuss *Pagination*.

Related Patterns and References. A paginated query typically defines an →*Atomic Parameter List* for its `in` messages (containing the query parameters) and a →*Parameter Tree* or →*Parameter Forest* for its `out` messages (i.e., the pages). A *Message Sequence* from [Hohpe and Woolf 2003] can be used when a single large data element has to be split up. Finally, *Incremental State Build-up*[58] in the currently emerging Conversation Patterns[59] language has the inverse intent (how can a consumer create a complex and/or large request message in multiple steps?), but a similar solution.

Chapter 10 of [Sturgeon 2016] covers the pagination types, discusses implementation approaches, and presents examples in PHP; Chapter 8 in the *RESTful Web Services Cookbook* by [Allamaraju 2010] deals with queries in an RESTful HTTP context.

The User Interface (UI) and Web design community has captured pagination patterns in/for different contexts (i.e., not API design and management, but interaction design and information visualization). See for example coverage of the topic at the Interaction Design Foundation[60] and a UI Patterns website[61].

4.6 Pattern Implementation Examples

We have implemented the patterns presented in this paper in a self-contained Java API provider based on Dropwizard and the Jetty Web server.[62] The provider exposes REST resources via JAX-RS and WSDL/SOAP Web services via JAX-WS; ones of the resources uses Pagination. Several illustrative API consumers are available in Java and JavaScript. A subset of the code snippets and wire-level data representations excerpts in the previous subsections stem from this sample implementation. The sample implementation also features the realizations of multiple collaborating patterns.

5. ADDITIONAL CANDIDATE PATTERNS (ACROSS CATEGORIES)

The five structure patterns presented in Section 4 only provide an initial foundation for the broad domain of the design and evolution of message-based remote APIs (which is the full scope of our future pattern language). More patterns are required to complete the envisioned pattern language. High priority candidates from our pattern backlog are listed in Table I.

Table I. : Overview of Candidate Patterns

Category			
Foundations	Vertical Integration Public API	Horizontal Integration Community API	Service Contract Solution-Internal API
Identification	Contract First Static Domain Analysis	Emergent Service Model Dynamic Process Analysis	Event Storming Business Artifact Analysis

[57] http://apistylebook.com/design/topics/collection-pagination
[58] http://www.enterpriseintegrationpatterns.com/patterns/conversation/IncrementalStateBuild.html
[59] http://www.enterpriseintegrationpatterns.com/patterns/conversation/
[60] https://www.interaction-design.org/literature/article/split-the-contents-of-a-website-with-the-pagination-design-pattern
[61] http://ui-patterns.com/patterns/Pagination
[62] The source code and related documentation is available at https://github.com/web-apis/riskmanagement-server.

Table I. : Overview of Candidate Patterns

Category			
Responsibility	Master Data Resource	Transactional Data Resource	Command Service
	Embedded Reference Data	Linked Reference Data	Reference Data Lookup
	Business Activity Service	Query Service	Validation Service
	Information Service	Periodic Report	Status Check
	Id Parameter	Link Parameter	Entity Parameter
	Metadata Parameter	Control Metadata, Provenance Metadata	Annotated Parameter Collection
Quality	API Key	Rate Limit	Service Level Agreement
	Request Bundle	Conditional Request	Wish List/Template
	Context Representation	Metering and Billing	Error Reporting
Evolution	Semantic Versioning	Version Identifier	Two in Production
	Eternal/Limited Lifetime Guarantee	Aggressive Deprecation	Experimental Preview
Management	not actively worked on		

These candidate patterns were mined from own system integration projects, 18 public Web APIs (including Facebook, GitHub, Google Calendar, Heroku, Instagram, JIRA, LinkedIn, Microsoft Graph, OpenWeatherMap, PayPal, Twitter, and Youtube) and supporting literature through a series of patterns ”problem jams”, ”forces jams” and ”known uses jams” in which we asked participants questions like:

- Candidate pattern (problem) identification:
  - Which (architecture) design problems recur in service design and evolution and can be described in a platform-independent fashion?
  - Which patterns do you expect a pattern language for message-based remote APIs (e.g., RESTful HTTP, WSDL/SOAP Web services; no remote objects) to feature, in topic areas such as message structure (syntax), message content (semantics), and message delivery quality?
- Forces (quality attributes, other decision drivers) elicitation:
  - What are the main architectural decision drivers ([Zdun et al. 2013]) in API design and consumption?
  - Can you name your top three quality attributes, possibly with some refinements such as quality attribute utility trees [Barbacci et al. 2002]?
  - Which forces should the pattern language focus on?
  - What are typical conflicts between these quality attributes and/or forces?
  - Which tradeoffs should be discussed?

- Known uses and pattern sources scoping:
  - Which pattern languages and APIs qualify as role models and sources of examples?
  - Which best practices documents (white papers) for RESTful HTTP and Web services design should be mined to generalize platform-specific into platform-independent advice?

We first answered these questions ourselves, followed by sessions with members of our professional and academic networks. As part of our future work, we may continue these pattern problems, forces, and known uses jams and review additional APIs such as those provided by cloud providers and enterprise information systems to identify additional known uses, forces, and future pattern candidates.

In the remainder of this section, we introduce selected candidate patterns from three more categories: responsibility, quality, and evolution.

5.1 Responsibility Category

API design does not only deal with the syntax of the request and response messages. API designers also need to address semantic concerns and find an appropriate business granularity for each API call. Simplistic statements such as "always prefer fine-grained over coarse-grained contracts" are insufficient or even irresponsible as requirements and project contexts differ [Pautasso et al. 2017; Zimmermann et al. 2004]; the architectural role and *responsibility* of each API call has to be specified.[63]

To satisfy the need for reusable knowledge about such content semantics and granularity, we distinguish between different types of service responsibilities and data access. For instance, *master data* does not change often (and has many incoming references) in contrast to *transactional data* that is created and changed frequently. This observation leads to patterns such as →*Master Data Resource* and →*Transactional Data Resource*. Transactional data often references master data (e.g., orders and contracts reference customers), and the decomposition mechanisms follow different patterns depending on project contexts and requirements: →*Embedded Reference Data*, →*Linked Reference Data*. When reference data is not embedded but linked, a →*Reference Data Lookup* service can be used to obtain the reference data. References data may include dynamic and complex master data, but also rather static, unstructured data such as country codes, currency information (e.g., in geographical information systems and enterprise applications).

Once entity-level Create, Read, Update, Delete (CRUD) operations are available, these can be composed into processing services of different types: we distinguish a data pushing →*Command Service* (that has specializations such as →*Information Service* and →*Business Activity Service*) from a data pulling →*Query Service* (with specializations/variants →*Periodic Report* and →*Status Check*) and →*Validation Service* (with variant →*Business Rule (Enforcement) Service*; the validation may pertain to the payload of the request message and/or the current internal state of the server). These services may or may not cause a server-internal state transition when being invoked; this is an important architectural decision to be made during API design (for each API call). Commands typically change server state, while queries and validations do not. The amount of server-side processing caused by the service invocations differs by service type as well.

5.2 Quality Category

The patterns in this category discuss how to achieve a certain level of quality of the offered services (in API design and usage). An API provider has to perform the balancing act of providing a high-quality service while at the same time having to use its available resources economically. The resulting compromise is expressed in a provider's →*Service Level Agreement* by the targeted service objectives

[63]The category name is inspired by the Responsibility-Driven Design method, CRC cards and the Single Responsibility Principle.

and associated penalties. If the service is paid for or follows a "freemium" model, the provider needs to come up with one or more rate plans and pricing schemes; service usage has to be monitored. The most common variations are a simple flat-rate subscription or a more elaborate consumption-based pricing scheme, which is explored in the →*Metering and Billing* pattern.

A provider needs to identify the calls it receives to decide if a call actually originates from a customer or some unknown client. An →*API Key* that identifies the client is a minimal, or even minimalistic, solution to this problem. If security is an issue, →*API Keys* are not enough and should be complemented by a proper authentication mechanism such as OAuth 2.0 (which are out of our scope).

Having identified its clients, an authenticated client could use too many resources, negatively impacting the service for other clients. To limit excessive usage, a →*Rate Limit* can be employed to restrain overusing clients.

Basic representation patterns such as →*Atomic Parameter* and →*Parameter Tree* deal with structuring the 'in' and 'out' parameters in request and response messages in message-based remote APIs. Providers may offer rather rich data contracts in their responses; not all consumers might need all of this information all the time. A →*Wish List* and a →*Wish Template* allow the client to request only the attributes in a response data set that it is interested in. A →*Conditional Request* can be used to save endpoint processing power and →*Rate Limit* usage. Building a singular →*Request Bundle* of multiple requests further reduces latency and bandwidth usage.

The requests exchanged between client and provider often span different networks and transportation technologies. To make sure that no control- or meta information (such as a →*Rate Limit*) is lost or needs to be reformatted on the way, a →*Context Representation* can be established. Finally, if something goes wrong, the provider needs to think about →*Error Reporting* to communicate any information about incorrect requests or internal server errors.

5.3 Evolution (Lifecycle Management) Category

The evolution category deals with lifecycle management concerns. This includes aspects like versioning and deprecation and how these concerns are reflected in the API design. Patterns in this category often need to balance the following forces: compatibility and developer experience, decoupling of the life-cycle of the consumer and provider, impact of changes on the consumer, freedom of the provider to change the API, and maintenance efforts on both the consumer and provider side.

Different patterns are concerned with the management of the compatibility of an API, which is expressed by its versioning strategy. The →*Version Identifier* pattern introduces an explicit version tag in the exchanged messages. Consumers and providers need to check whether they can parse and interpret this particular version and fail if that is not the case. While this allows for semantic changes in the API without risking that partners misinterpret messages, an explicit →*Version Identifier* imposes frequent updates to consumers because they need to support the new version. A →*Version Identifier* can for example be derived by using →*Semantic Versioning*, which uses a three-digit version number that expresses the change impact (major, minor and fix version).

A converse approach is described in the →*Eternal Lifetime Guarantee* pattern: The provider promises to make the API available forever and make updates either in another API or by making backwards compatible changes only. This shifts all burden and associated effort from the consumer to the provider. Because this extreme form of a compatibility guarantee is often not applicable in practice, a weakened guarantee is provided in the →*Limited Lifetime Guarantee* pattern, in which the provider promises to provide the API in its current or compatible form for a specified time. For example, consumers can rely for two years on an API, but also have to plan to migrate to the newest API version during this period. Another pattern that balances the effort for the provider and consumer more evenly is the →*Two in*

Production pattern. Instead of specifying a fixed time-frame, this patterns limits the number of API versions that need to be maintained by the provider in parallel.

If not the whole API needs to be migrated but a finer-grained approach is better suited to the problem and project context, a pattern to consider is →*Aggressive Deprecation*. By issuing deprecation notices, the API provider can signal that certain capabilities of an API will be removed at a specified date or with the next version.

Versioning and API Management can be a hassle, which is too complicated especially if an API is not yet stable or released. The →*Experimental Preview* pattern describes a possibility to exempt development previews, beta versions etc. from these restrictions and show early versions to possible users without making any guarantees and reserving the right to revoke access to them at any time..

6. SUMMARY AND OUTLOOK

In this paper, we presented first versions of four plus one representation patterns for message-based remote API design and evolution, *Atomic Parameter*, *Atomic Parameter List*, *Parameter Tree*, *Parameter Forest*, and *Pagination*. We also outlined a number of additional pattern candidates in seven categories. We have collected these patterns and candidate patterns in literature reviews and interactions with practicing architects and developers. We are currently in the process of documenting them incrementally and iteratively; about 40 pattern descriptions have already been drafted and partially reviewed. The pattern mining continues at the time of writing.

ACKNOWLEDGMENTS

We want to thank our shepherds, students and members of our professional networks who helped to investigate public Web APIs, donated candidate forces and patterns, and reviewed early drafts of pattern candidates and language structure: Thomas Brand, Nicolas Dipner, Hugo Sereno Ferreira, Alex Gfeller, Gregor Hohpe, Sebnem Kaslack, Oliver Kopp, Jochen Küster, Frank Leymann, Frank Müller, Cesare Pautasso, Peter Sommerlad, and Veith Zäch.

REFERENCES

Subbu Allamaraju. 2010. *RESTful Web Services Cookbook*. O'Reilly Media, Inc, Sebastopol.

Eric Allman. 2011. The Robustness Principle Reconsidered. *Queue* 9, 6, Article 40 (June 2011), 8 pages. DOI:http://dx.doi.org/10.1145/1989748.1999945

Jim Arlow and Ila Neustadt. 2004. *Enterprise patterns and MDA: building better software with archetype patterns and UML*. Addison-Wesley Professional.

Mario R Barbacci, Robert J Ellison, Anthony Lattanze, Judith Stafford, Charles B Weinstock, and William Wood. 2002. *Quality attribute workshops*. CMU/SEI-2003-TR-016. Software Engineering Institute, Carne- gie Mellon University, Pittsburgh, PA.

Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. 2005. Service Interaction Patterns. In *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*. 302–318. DOI:http://dx.doi.org/10.1007/11538394_20

Michael Brandner, Michael Craes, Frank Oellermann, and Olaf Zimmermann. 2004. Web services-oriented architecture in production in the finance industry. *Informatik-Spektrum* 27, 2 (2004), 136–145. DOI:http://dx.doi.org/10.1007/s00287-004-0380-2

Frank Buschmann, Kevlin Henney, and Douglas Schmidt. 2007. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons.

Robert Daigneau. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional. http://www.servicedesignpatterns.com/

Eric Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Publishing Company, Incorporated.

Martin Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 185–200.

David C. Hay. 1996. *Data Model Patterns: Conventions of Thought*. Dorset House Pub. https://books.google.ch/books?id=a7VQAAAAYAAJ

Carsten Hentrich and Uwe Zdun. 2011. *Process-Driven SOA: Patterns for Aligning Business and IT*. Auerbach Publications, Boston, MA, USA.

Gregor Hohpe. 2007. Conversation Patterns: Interactions between Loosely Coupled Services. In *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee, Germany.

Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

IERC. 2017. IoT European Research Cluster. (2017). http://www.internet-of-things-research.eu/

Klaus Julisch, Christophe Suter, Thomas Woitalla, and Olaf Zimmermann. 2011. Compliance by design–Bridging the chasm between auditors and IT architects. *Computers & Security* 30, 6 (2011), 410–426.

James Lewis and Martin Fowler. 2014. Microservices. https://martinfowler.com/articles/microservices.html/, (2014).

Frank Leymann. 2016. ESOCC Keynote. (2016). http://esocc2016.eu/wp-content/uploads/2016/04/Leymann-Keynote-ESOCC-2016.pdf

Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2016. A Pattern Language for RESTful Conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee, Germany.

Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. 2017. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software* 34, 1 (2017), 91–98. DOI: http://dx.doi.org/10.1109/MS.2017.24

Richardson. 2017. Microservices Patterns. http://microservices.io/patterns/microservices, (2017).

Arnon Rotem-Gal-Oz. 2012. *SOA Patterns*. Manning.

Phil Sturgeon. 2016. *Build APIs you won't hate*. LeanPub, https://leanpub.com/.

Markus Voelter, Michael Kircher, and Uwe Zdun. 2004. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. J. Wiley & Sons, Hoboken, NJ, USA.

Uwe Zdun, Rafael Capilla, Huy Tran, and Olaf Zimmermann. 2013. Sustainable Architectural Design Decisions. *IEEE Software* 30, 6 (2013), 46–53.

Olaf Zimmermann, Pal Krogdahl, and Clive Gee. 2004. Elements of service-oriented analysis and design. *IBM developerWorks* (2004).

Olaf Zimmermann, Mark Tomlinson, and Stefan Peuser. 2003. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer Science & Business Media.

Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, and Frank Leymann. 2008. Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. In *Proc. of WICSA*. 157–166.