API Refactoring to Patterns

Catalog, Template and Tools for Remote Interface Evolution

Mirko Stocker
Olaf Zimmermann
mirko.stocker@ost.ch
olaf.zimmermann@ost.ch
Eastern Switzerland University of Applied Sciences (OST)
Rapperswil, Switzerland

ABSTRACT

Refactoring is an essential agile practice for software evolution. While program-internal code-level refactoring is well established, architecture-level refactoring has been researched but not yet widely adopted in practice. As a result, application programming interface (API) refactoring is not well understood, and practitioners consequently struggle with the evolution of distributed systems using Web APIs and other remoting technologies. To fill this knowledge gap, we propose to apply refactoring to the problem of designing and developing adaptive APIs. This paper introduces an Interface Refactoring Catalog (IRC) and presents eight of its refactorings. IRC has been available online since 2021 and collects 22 refactorings at present. Eleven of these patterns leverage Patterns for API Design, originating from our previous work; the remaining ones deal with the number and size of API endpoints and their operations, cover renaming of these API building blocks and message representation elements, and deal with architectural change.

CCS CONCEPTS

• **Software and its engineering → Patterns**; *Designing software*.

KEYWORDS

application programming interface, cloud computing, design patterns, enterprise application integration, refactoring

ACM Reference Format:
Mirko Stocker and Olaf Zimmermann. 2023. API Refactoring to Patterns: Catalog, Template and Tools for Remote Interface Evolution. In *28th European Conference on Pattern Languages of Programs (EuroPLoP 2023), July 05–09, 2023, Irsee, Germany*. ACM, New York, NY, USA, 32 pages. https://doi.org/10.1145/3628034.3628073

1 INTRODUCTION

Refactoring is the practice of improving a software system without changing its external/observable behavior, such as making sure that names are well chosen or breaking a long sequence of code into several parts. Refactoring may as well align the software with a design pattern to improve its understandability [19]. Software design patterns collect and distill proven solutions and "communicate wisdom and insight in computer/software systems design" [23]. They follow a common template to address a specific problem and discuss different forces and how they are resolved when applying the pattern solution in a context; known uses make validity and applicability of each pattern evident. In this paper, we adopted the pattern concept to describe and structure our refactorings in a uniform template.

Many of the patterns that are the target to which we refactor are described in our pattern language for microservice and remote API design, first published in the EuroPLoP proceedings 2017 to 2020 [22, 36, 43, 44, 47] and finalized in "Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges" [48]. Appendix A provides an overview of all our API design patterns referenced in this paper. The initial inspirations for the current set of API refactorings came from our work on these API design patterns: for instance, patterns such as Embedded Entity and Linked Information Holder provide alternative solutions for a problem (here: management of nested, referenced data in the request and response payload); this observation led to the identification of *Extract Information Holder* and its inverse, *Inline Information Holder*. Other inspiration came from literature like "Refactoring: Improving the Design of Existing Code" by Fowler [12] and online resources such as refactoring.guru that describe code-level refactorings that we projected to the API/architectural level. Earlier work on architectural refactoring [42] and our own professional experience developing software also contributed to the current version of our catalog. We prototyped most of the refactorings in a tool and provide many application examples as well.

We have collected a total of 22 refactorings so far and maintain a backlog of additional candidates. Eight of these refactorings are featured in this paper; the entire catalog is available at interface-refactoring.github.io. Eleven of the refactorings in the catalog use API design patterns as targets. In this paper, we present two refactorings to API design patterns (*Add Wish List* and *Introduce Pagination*), four refactorings changing API structure and names (*Introduce Data Transfer Object*, *Split Operation*, *Merge Operation*, *Rename Representation Element*), as well as two refactorings adding architectural components (*Segregate Commands from Queries* and *Introduce Version Mediator*) to ensure that the selection is representative of the catalog without growing the paper to an excessive length. Note that we use the terms API refactoring and interface refactoring interchangeably.

EuroPLoP 2023, July 05–09, 2023, Irsee, Germany

ACM ISBN 979-8-4007-0040-8/23/07.
https://doi.org/10.1145/3628034.3628073

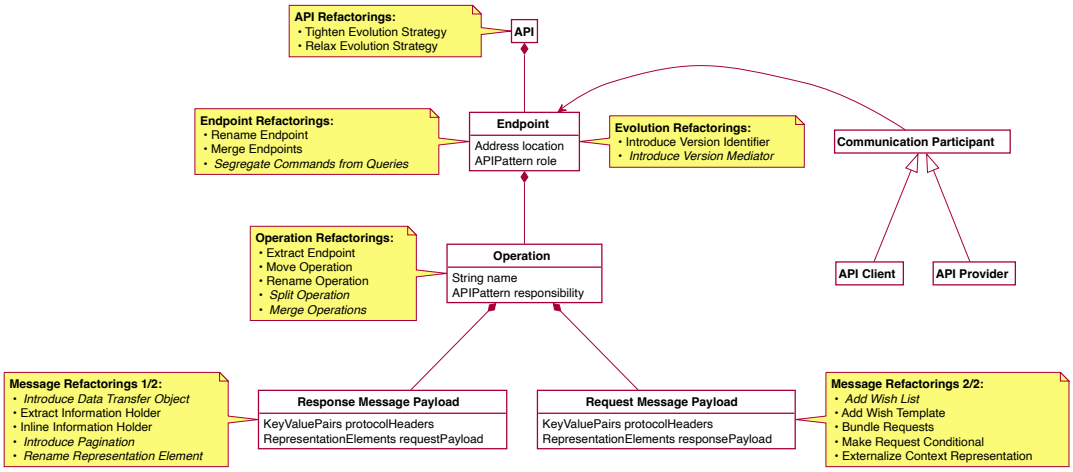

Figure 1: Refactorings by targeted API element; refactoring names in italics indicate the scope of this paper. Also see Table 1 that introduces the refactorings with a goal-statement expressed in the form of user stories.

Just like code-level refactorings affect different targets (e.g., variables, methods, or classes), API refactorings can also be performed on different elements of an API. According to the API domain model in [48], "an API is a collection of endpoints" that offer "operations" to "communication participants" (also called "API clients" and "API providers" depending on their role in the communication). API clients and API providers exchange structured request and response messages. Figure 1 shows the targets of our refactorings.

Patterns are mined from known uses, whereas our refactorings are derived from experience and literature. To give concrete usage examples of how the refactorings affect the API implementation and architecture, we are showing them in the context of the Lakeside Mutual sample application. Lakeside Mutual is a fictitious insurance company that serves as an example scenario to demonstrate microservices [46] and domain-driven design [6]. We use it for our teaching of bachelor-level lectures Application Architecture [17] and Cloud Solutions; it also demonstrates many of the API design patterns in Zimmermann et al. [48]. The application comprises several Spring Boot microservices that offer APIs to frontends. Supporting Online Transaction Processing, these APIs expose operations to create, read, update, and delete insurance policies as well as product and customer master data; search capabilities are provided as well. For example, the *Customer Self-Service Frontend* single-page application uses two backend APIs to let insurance customers manage their policies. Figure 2 provides an overview of the microservices and frontends in the application scenario.

The remainder of this paper is structured in the following way. Section 2 discusses related patterns and pattern languages. Section 3 gives an overview of our interface refactoring catalog and presents the eight refactorings mentioned above. Section 4 discusses the practical adoption of our refactorings in tools and practices. Section 5 summarizes and concludes the paper. Our layout conventions are as follows: Refactorings are set in *italics*, whereas pattern names are set in Small Caps. Links to additional information are printed as footnotes for better readability, in print copies in particular.

2 RELATED WORK

The design and refactoring of message-based remote APIs can benefit from many existing patterns on various kinds of distributed systems, especially those related to services, as well as patterns related to API design (e.g., API design in object-oriented programming) and enterprise integration. Related research includes "Mapping the Space of API Design Decisions" by Stylos and Myers [38]. API usability redesign is the focus of a case study by Stylos et al. [37]. "WEBAPIK: A body of Structured Knowledge on Designing Web APIs" by Sadi and Yu [29] presents design techniques to address non-functional requirements in the design of Web APIs, such as evolvability and performance. Refactoring helps to align an architecture with such non-functional requirements.

Fowler, author of "Refactoring: Improving the Design of Existing Code" [12] and maintainer of Refactoring.com defines refactoring as "a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior-preserving transformations." "Refactoring to Patterns" by Kerievsky [19] describes improving a system with pattern-directed refactorings. "Refactoring for Software Design Smells" by Suryanarayana et al. [39] describes how to apply refactorings to remove design smells. The Refactoring.Guru website focuses on code refactorings. Some are API design-related; the website also works with design smells. Fowler popularized code-level refactoring, but the concept and the activity have a broader appeal. Refactoring can also be applied to databases, as Ambler and Sadalage [2] show in "Refactoring Databases: Evolutionary Database Design." Yoder and Merson [40] present "Strangler Patterns" that can be used to evolve a monolith into microservices.

The Open Agile Architecture™ has an activity called "Continuous Architectural Refactoring." The importance of connecting architecture-centric practices with agile ones, such as refactoring, is identified by Hohpe et al. [17]. An emerging collection of Domain-Driven Refactorings by Henning Schwentner includes tactical and strategic refactorings in and around bounded contexts and socio-technical refactorings to reorganize teams.

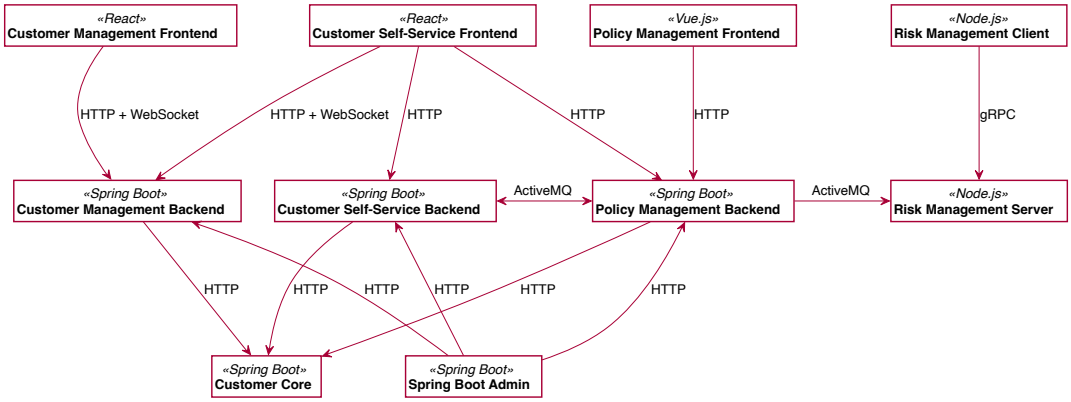

Figure 2: Lakeside Mutual components and their relationships

The Cloud Adoption Patterns website describes several "Cloud Refactoring" steps to "minimally adapt an existing application to work on the Cloud" without having to rewrite it. A Serverless Land blog post by Hohpe and Pillai introduces "Refactoring to Serverless" as "improve the design of your serverless application by replacing application code with automation code, while using the same programming language."

In our previous work, we show how decision models can guide API designers through their pattern selection process [41, 48]. We defined the term "API refactoring," reported on desired and missing qualities, and outlined the catalog that this paper elaborates upon in Stocker and Zimmermann [35]. We also implemented automated refactoring tools for the Ruby and Scala programming languages [4, 34]. Finally, we feature related practices for agile architecting and service design in our "Design Practice Reference" [45].

3 THE INTERFACE REFACTORING CATALOG (FIRST SLICE)

Our Interface Refactoring Catalog is a collection of API refactorings and related architectural refactorings that impact the API of a system. Stal [33] proposed architectural refactoring to improve system-wide software evolvability. Architectural refactoring may also target other quality attributes and non-functional requirements; therefore, Zimmermann [42] defined *Architectural Refactoring* as follows:

> "… a coordinated set of deliberate architectural activities that removes a particular architectural smell and improves at least one quality attribute without changing the scope and functionality of the system."

In Stocker and Zimmermann [35], we then defined the term *API Refactoring*. A slightly adapted version of this definition is:

> An API refactoring evolves the remote interface of a system or system component without changing the feature set and semantics of the interface implementation to improve at least one quality attribute. Interface clients may or may not have to be changed when this evolution takes place.

One might argue that evolving an interface violates the "without changing its external behavior" part of the code refactoring definition from Refactoring.com. However, we argue that it is a matter of scope and boundaries and that we have to look at the entire system and each message exchange, including the API provider and API client. Let us consider a simple code refactoring example where a local variable is renamed. It is highly unlikely that the external behavior of the software will be affected after this change. But refactorings that involve multiple classes, such as moving a method from one class to another, change the interfaces of those classes – possibly including public ones – in a backward incompatible way; the code that uses the changed class part has to be changed as well. Refactoring tools can fix the clients of those classes, but only if they are known to the tool. If these classes are part of a library and are exposed to unknown third parties, a refactoring tool won't be able to fix them automatically. Whether a refactoring at the interface level changes external behavior depends on the perspective and expectations of the observer: an API client developer may notice and be directly affected by the change caused by an API refactoring, but the end user of the software may not. This is similar to refactoring databases; when table structures change, the data access layer code (e.g., JDBC and SQL) also changes, but other parts of the application and its external behavior do not. We aim to ensure that refactorings are backward-compatible if possible, and many refactorings can indeed be applied without affecting the clients of an API (see the "Backward Compatibility" column in Table 1).

The focus on improving quality attributes is also evident in the template we propose to document API refactorings. Refactorings start from an *initial position sketch* along with *smells* that a series of *instructions (steps)* transforms into a *target solution sketch*. The full refactoring template is shown in Appendix B. Fowler [12] uses a template for code-level refactorings that describes them with a name, solution sketch, motivation, mechanics, and examples. Our template also covers these information elements. However, we decided to structure the template further and add elements such as stakeholder concerns (that make quality attributes explicit) and design smells (that indicate problems with an existing solution).

While the refactorings captured in this template are not patterns in the classical sense [15], they share many properties with software

design patterns: refactorings are applied in a specific context to solve a particular problem. Different forces apply, requiring trade-offs and decisions.

The refactoring activities are described in a technology-neutral way for the most part, but their examples and implementation hints mainly stem from HTTP resource API design and evolution. Most examples are written in Java, using the Spring Framework, and some use the Microservice Domain-Specific Language (MDSL). MDSL is a domain-specific language that uses API design patterns to describe APIs in a technology-independent way. Code generators are available to generate interface descriptions, for example targeting OpenAPI. See [30] for an introduction to MDSL.

Table 1 outlines the eight refactorings in this paper that we will present in the remainder of this section.

3.1 Refactoring: Introduce Data Transfer Object

also known as: Map and Wrap Representation Structure, Ubiquitous Language Wrapper

3.1.1 Context and Motivation. An API offers one or more operations that return Data Elements [48]. For example, the API of a customer relationship management service might contain an endpoint that returns detailed information about customers and the interactions with them. The structure of these responses might have been derived from the API implementation classes and domain model data structures, with possibly deeply nested structures of elements [31]. For example, an API implementation might use an Object-Relational Mapper (ORM) to manage the data and might return serializations of instances of the ORM classes in the response messages.

> *As an API provider, I want to encapsulate my internal data structures so that I can freely change them without breaking backward compatibility of my clients. In domain-driven design terms, I want to keep the integration-level published language separate from the application-level ubiquitous language*[1].

3.1.2 Stakeholder Concerns (including Quality Attributes and Design Forces).

#modifiability, #evolvability, and #information-hiding API providers want the freedom to change the API implementation without revealing such changes to clients. Such information hiding [26] is crucial for the independent evolvability of API providers and clients.

#developer-experience API clients want to navigate the required data with minimum effort, taking as few coding steps (expressed as statements and expressions) as possible.

#cohesion, #coupling API providers strive for low coupling and high cohesion in their endpoint, operation, and message designs.

3.1.3 Initial Position Sketch. The refactoring is eligible for any API operation that uses an implementation type (for example, a domain class or a database-mapped entity) in a request or response message. Note that this implementation type may also appear as a subordinate element within a message structure hierarchy.

[1]Ubiquitous Language is one of the core patterns in Domain Driven Design [6].

For instance, an API provider might return data from a repository directly to the client, as shown in Figure 3. The structure and content of the data are not changed; it is simply passed through.

3.1.4 Design Smells.

Leaky encapsulation Domain layer language constructs (e.g., classes) or abstractions defined in the persistence layer are directly exposed in the API. Such permeable or even completely missing encapsulation of API internal data structures makes an API harder to evolve because it introduces coupling and harms backward compatibility.

Tight coupling of data contract Anything exposed will be used according to Hyrum's Law. Hence, leaky encapsulations cause undesired coupling, which may slow development and decrease modifiability and flexibility.

Confetti design Clients might have to issue many requests to get all the needed data. Fine-grained APIs that rain many small Data Elements on clients are rather flexible but can be tedious to use.

3.1.5 Instructions (Steps). To replace an implementation type in a response message with a DTO, perform the following steps:

1. Create a new data-centric wrapper (e.g., a class in object-oriented languages) that mirrors the attributes of the current message representation. Such *Data Transfer Objects* (DTOs) [11] are typically implemented as immutable Value Objects [6] with structural, value-based equality.
2. Depending on the implementation framework, add additional serialization logic or mapping configuration information. An example is the JSON-to-Java data binding offered by libraries such as Jackson.
3. Write unit tests for the DTOs and the mapping logic. If the framework generates the code, such tests might be unnecessary (or already provided by the framework).
4. Adjust the implementation of the operation to create an instance of the DTO and fill it with the necessary data.
5. Return the DTO from the operation implementation, adjusting any return types if necessary (so that the new serialization logic can pick them up).
6. Run the tests to ensure the message structure was not changed accidentally. For example, an integration test might check whether all attributes expected in a JSON object are present.
7. Include the API change in the API Description if it is visible there; for example, adjust the JSON-Schema part of the OpenAPI description of the API.
8. Align the sample data in supplemental API documentation artifacts such as tutorials so that it the new structure is featured.

These instructions assume that the DTO is introduced in a response message. If a request message is the target, steps 4 and 5 must be adapted. Instead of creating and returning a DTO, adjust the implementation of the operation to take the DTO as a parameter and convert it back to the API-internal data representation. This refactoring is fully backward-compatible because it only changes the implementation, not the structure of the message.

Refactoring	Goal Expressed as a User Story	Backward Compatibility
Introduce Data Transfer Object (p. 4)	As an API provider, I want to encapsulate my internal data structures so that I can freely change them without breaking backward compatibility of my clients.	Yes, this is an implementation level change and does not affect the API contract
Add Wish List (p. 7)	As an API client, I want exact control over response message content so that I receive just the data I require to realize an application feature.	Yes, the Wish List can be an optional parameter
Introduce Pagination (p. 10)	As the API provider, I want to return data sets in manageable chunks so that clients are not overwhelmed by a huge amount of data arriving at once.	Possible, depending on message structure, but not recommended
Split Operation (p. 12)	As an API provider, I want to offer endpoints that expose operations with distinct responsibilities so that the API is easy to understand and use for client developers and can be modified rapidly and flexibly by provider developers.	No, client has to call new operation with different message
Merge Operations (p. 15)	As an API designer, I want to remove an API operation from an endpoint and let another operation in that endpoint take over its responsibilities so that there are fewer operations to maintain and evolve and the inner cohesion of the endpoint improves.	No, client has to call new operation, operation messages change as well
Rename Representation Element (p. 18)	As a data modeler and/or Data Transfer Object (DTO) designer, I want to use expressive and domain-specific terminology so that data representations and their elements become self-explanatory to API developers on the client as well as on the provider side.	Possible, with additional effort on provider side to accept old names
Segregate Commands from Queries (p. 20)	As an API provider, I want to serve queries and process commands separately so that I can optimize the respective read and write model designs independently.	Yes, if API Gateway, Service Registry, or Version Mediator are used
Introduce Version Mediator (p. 22)	As an API client, I want to continue to call a deprecated API for some time, and I expect the provider to support me with a temporary solution for doing so. The behavior of this solution should be identical to those of the API that I have been using so far.	Yes, ensuring backward compatibility is the objective of this refactoring

Table 1: Refactorings in the order in which they are presented in this paper along with a goal statement expressed in the form of a user story and an indication of whether the refactoring is backward-compatible.

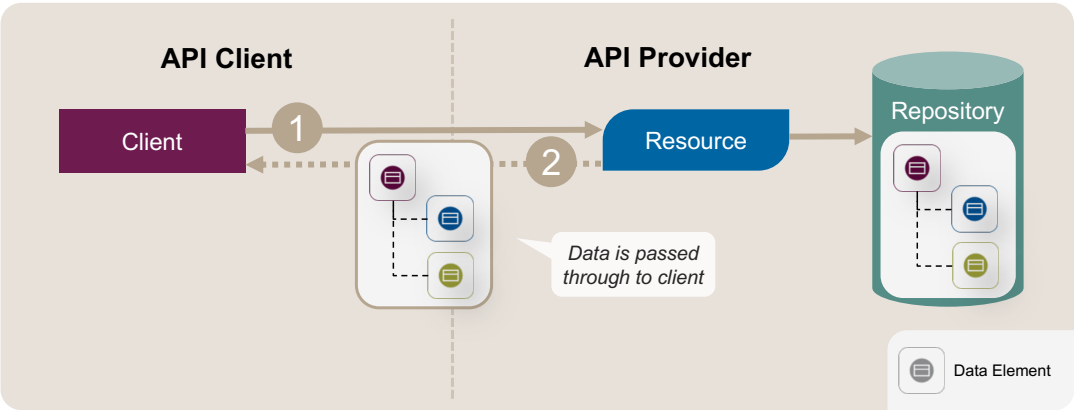

Figure 3: Introduce Data Transfer Object: Initial Position Sketch. The API provider responds to a client request (1) with a message (2) that contains some data elements.

3.1.6 Target Solution Sketch (Evolution Outline). When the refactoring has been applied, a mapping step takes place that copies the data to/from the DTO structure. The mapping can preserve the structure or adjust it, depending on the information needs of the message recipient. Figure 4 shows a refactored response message.

Now that the internal implementation has been decoupled from the response entity, the DTO can transfer additional data, such as Embedded Entities, Link Elements, or Metadata Elements. Such richer messages help against confetti design.

If the internal implementation type evolves, the DTO can implement more complex mapping logic to maintain backward compatibility. Additional metadata, such as a Version Identifier, can also be added to the DTO.

3.1.7 Example(s). The following excerpt from a Java Spring Boot controller shows an implementation of an operation `getMyEntity` that fetches an entity from a database repository and directly returns it in its response:

```
@GetMapping(value = "/{id}")
public ResponseEntity<MyEntity> getMyEntity(
        @ApiParam(value = "the entity's unique id")
            @PathVariable MyEntityId id) {

    MyEntity myEntity = myEntityRepository.getMyEntity(id);
    if (myEntity == null) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Entity Not Found");
    }
    return ResponseEntity.ok(myEntity);
}
```

The Spring `GetMapping` annotation turns the `getMyEntity` method into an API operation (HTTP GET) that receives an `id` parameter and returns a `ResponseEntity`. The `ResponseEntity` class has helper methods – such as `ok` – to generate HTTP messages (`200 OK` in this case).

The `MyEntity` class is also used in the object-relational mapping in this example. More specifically, the Java Persistence API (JPA) is used:

```
@Entity
@Table(name = "my_entities")
public class MyEntity {
  String attribute;
  ...
}
```

A sample response could look like this:

```
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
{
  "attribute" : "1c184cf1-a51a-433f-979b-24e8f085a189"
}
```

When the refactoring has been applied, a `MyEntityDto` is returned (instead of returning `MyEntity` directly):

```
@GetMapping(value = "/{id}")
public ResponseEntity<MyEntityDto> getMyEntity(
        @ApiParam(value = "the entity's unique id")
            @PathVariable MyEntityId id) {

    MyEntity myEntity = myEntityRepository.getMyEntity(id);
    if (myEntity == null) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Entity Not Found");
    }
    MyEntityDto myEntityDto = MyEntityDto.toDto(myEntity);
    return ResponseEntity.ok(myEntityDto);
}
```

The `MyEntityDto` DTO is implemented as follows:

```
public class MyEntityDto {
  // Attributes that mirror those in MyEntity
  String attribute;
  ...

  static MyEntityDto toDto(MyEntity myEntity) {
    // Copy attributes from myEntity to new DTO instance
  }
}
```

Because the DTO mirrors the attributes of `MyEntity`, the resulting HTTP response remains unchanged.

Another example comes from the rapid prototyping framework JHipster. The application generator provides the option to generate the Spring Boot code with DTOs. Enabling this option changes the signature of the service class (the `+` and `-` stand for added and removed lines, respectively):

```
-    public Optional<Customer> findOne(Long id) {
+    public Optional<CustomerDTO> findOne(Long id) {
         log.debug("Request to get Customer : {}", id);
         return customerRepository
-            .findById(id);
+            .findById(id).map(customerMapper::toDto);
     }
```

The `CustomerDTO` that replaces `Customer` as the response type in this example is a simple Java bean with attributes, getters, and setters. For the mapping from entity to DTO and vice-versa, JHipster uses MapStruct, an annotation processor that frees the developer from writing trivial mapping code:

```
import org.mapstruct.Mapper;
@Mapper(componentModel = "spring", uses = {})
public interface CustomerMapper
    extends EntityMapper<CustomerDTO, Customer> {}
```

Note that the refactoring can also be applied to request messages.

3.1.8 Hints and Pitfalls to Avoid. Do not over-eagerly apply this refactoring to all API operations, but use it only when its value is higher than its cost (note: this is general advice that makes sense in most cases). A good reason might be that the implementation data structures change often and these changes should not be reflected in the structure of the API-level response messages.

DTO classes and mappings are often straightforward to create, so various libraries and code-generation tools exist to automate this task. For example, Lombok is an alternative to MapStruct in the Java ecosystem. The recently introduced Java Records also address this topic.

When receiving data, be a Tolerant Reader by making "minimum assumptions about the structure" and only consuming the data you need. This approach has the advantage that recipient code will not be affected if unused parts of the DTO change.

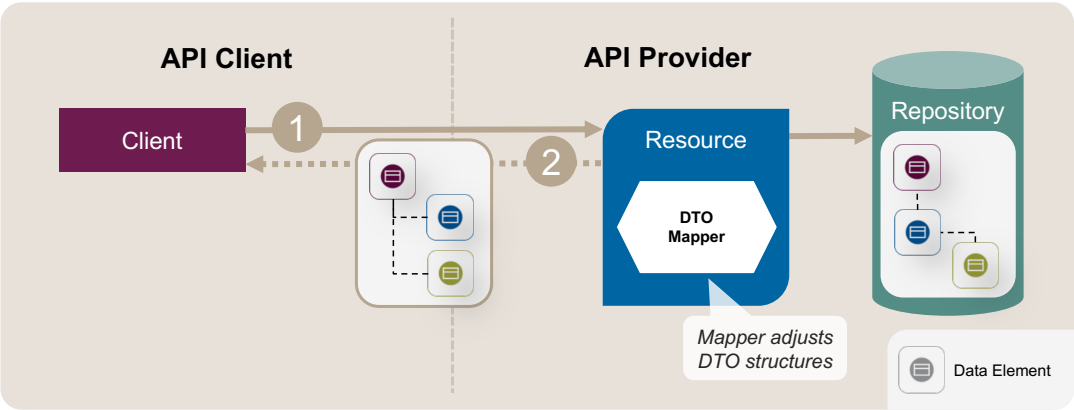

Figure 4: Introduce Data Transfer Object: Target Solution Sketch. Instead of passing through the data for the client's request (1), an additional DTO mapper transforms the data elements before they are returned (2). The implementation types can be changed without affecting the API.

Another motivation for the refactoring can be that additional (meta-)data has to be returned, for example, when applying the *Introduce Pagination* refactoring.

There is a potential risk of introducing memory leaks in API implementations when developers allocate and release memory manually. Marshaling and unmarshaling of request and response data is often handled by frameworks (for example, JSON to Java and Java to JSON in Jackson when using Spring); caching might occur. Unit tests usually will not catch memory bugs; this requires dedicated reliability tests.

DTOs are meant for communicating with external clients and should not be used internally in the API implementation. If the API implementation needs to pass around data internally, it should use the existing data structures directly. See the article about Internal Data Transfer Objects by Phil Calçado for reasons why DTOs should not be used internally.

3.1.9 Related Content. The DTOs and related mapping logic are usually placed in an implementation-level Service Layer [11].

Chapter 8, "Evolve APIs", in [48] discusses evolution strategies for APIs. Refactorings such as *Introduce Version Identifier*, *Introduce Version Mediator*, *Relax Evolution Strategy*, and *Tighten Evolution Strategy* provide guidance on refactoring an API towards those patterns.

Domain Driven Design (DDD) offers additional techniques and patterns to structure domain classes. A brief introduction to Tactic DDD can be found in the Design Practice Repository (DPR) on GitHub and the corresponding DPR eBook [45].

Many Enterprise Integration Patterns [18] are related. For instance, Content Enricher and Content Filter can be used to wrap and map implementation-internal data.

Step 4 of the Stepwise Service Design in DPR [45] advises to "foresee a Remote Facade that exposes Data Transfer Objects (DTOs) in the request and response messages of its API operations to decouple the (published) languages of frontends and backends and to optimize the message exchange over the network w.r.t exchange frequency and message size." The Remote Facade that is mentioned in the quote helps to "translate coarse-grained methods onto the underlying fine-grained objects." This means that the DTO can be used to restructure the data so that clients can easily interact with it while using the network efficiently.

3.2 Refactoring: Add Wish List

also known as: Introduce Payload Feature Toggle, Support Response Shaping and Expansion

3.2.1 Context and Motivation. An API offers one or more endpoints exposing operations to retrieve structured data. Not all API clients are interested in the same Data Elements that can be retrieved, and both clients and providers want to reduce unnecessarily transmitted and processed data.

> *As an API client, I want exact control over response message content so that I receive just the data I require to realize an application feature.*

Ideally, a single call should be enough the retrieve all required data. So one solution would be to offer custom operations for different clients, but this requires much knowledge about the clients on the provider side, which might not be easy to obtain and change dynamically. It would also increase maintenance effort and coordination needs.

3.2.2 Stakeholder Concerns (including Quality Attributes).

#client-information-needs APIs are often used by multiple clients with different information needs, which makes it difficult for their providers to offer one-size-fits-all solutions.

#evolvability, #flexibility The information needs of clients and users vary over time, and an API and its provider-side implementation should not have to be adjusted every time something changes in its clients. Maintaining several variations of the same operation for different clients is possible but increases the maintenance effort and coordination needs, which in turn might constrain the flexibility of the API provider.

#sustainability, #performance, #data-parsimony Overall response time, throughput, client-side and server-side processing time are qualities that concern API clients and their providers. Unused data that is prepared, transported, and processed wastes resources and ought to be avoided.

3.2.3 Initial Position Sketch. The initial response shown in Figure 5 comprises several, possibly nested, Data Elements, also known as attributes.

The targets of this refactoring are API operations with their request and response messages. The refactoring applies in different situations: the provider might return data the client is not interested in, as shown in Figure 5, where the client wants the ability to *filter* elements to avoid overfetching. The refactoring is also applicable when the provider omits data the client must retrieve through a second request from a different API operation. Clients can use the Wish List to *expand* the message content to avoid underfetching in that case.

3.2.4 Design Smells.

Underfetching Clients have to call multiple API operations to get all data they require because these operations do not offer any way to define the targeted Data Elements.

Overfetching Clients throw away large parts of the received data because the API design follows a one-size-fits-all approach, and the provider includes all data in responses that any present or future client might be interested in. For example, in an e-commerce API, product procurement information might only interest a few clients, while most want to learn about current prices and items in stock. Another phenomenon is "sell what is on the truck": implementation data is exposed just because it is there, without any client-side use case.

API does not get to the POINT According to the POINT principles for API design, any operation should have a *purpose*. It should also be *T-shaped* (both broad and deep, that is). Underfetching and overfetching indicate that these two principles are violated or only partially met.

3.2.5 Instructions (Steps). Adding a Wish List to request messages lets clients select what Data Elements they want the response to contain. Applying this pattern reduces the mismatch between client expectations and provider capabilities:

1. Analyze the current response message structure with respect to nesting, optionality, and data usage profiles. All its elements to become selectable must be optional; otherwise, not wishing for them cannot have any effect.
2. Apply *Introduce Data Transfer Object* (DTO) or adjust existing DTOs as needed.
3. Add a set-valued request parameter that allows clients to enumerate the desired Data Elements. The name of the parameter could, for example, be called `wishList`, `select`, or `expand`. Decide on a list separator that is easy to transport over the given networking protocol (for instance, slashes "/" might not be a good choice in HTTP APIs, but commas "," work well).
4. Populate this parameter when preparing requests on the client side.
5. Process this parameter when responding to requests on the provider side. Evaluate which Data Elements to return and prepare the response message accordingly. Avoid retrieving data from the data store only to discard it afterward. For example, customize SQL queries to only fetch the required attributes and avoid unnecessary joins.

Complete the refactoring with the following steps that apply to most/all IRC entries (also see Section 4 of the paper on TELL):

- Test the changes to ensure that the new design works and that the end-to-end capabilities of the API remain unchanged. Use this opportunity to learn about its effectiveness; establish success criteria derived from the stakeholder concerns. For example, performance might be an important metric that can be tracked by a benchmark.
- Enhance the external API Description and document the rationale for the new design.
- Upgrade the version number (indicating a backward-compatible feature enhancement in this case) and inform the clients when the new version has been released.
- Analyze whether the refactoring application has been successful according to the success criteria.

Several approaches to evolution exist. To preserve the original behavior for clients that omit the Wish List and ensure backward compatibility, the API should make the new parameter optional and keep the current response message structure and content. This approach does not break the clients, but they might miss the opportunity to learn about the Wish List and will continue to request the complete response structure. A different, possibly incompatible approach is only to return a minimal response message by default so that clients are forced to state their wishes.

3.2.6 Target Solution Sketch (Evolution Outline). The Wish List in the operation signature lets clients specify certain Data Elements they are interested in. The provider then tailors the response to the client's wishes. This solution is sketched in Figure 6. Chapter 7 of [48] shows a solution sketch with a dedicated List Evaluator component to handle the client's wish.

3.2.7 Example(s). The following example from our Lakeside Mutual sample application shows a response of the Policy Management backend microservice before refactoring. Note that no customer master data is included, only a `customerId`:

```
curl http://localhost/policies/fvo5pkqerr
{
  "policyId" : "fvo5pkqerr",
  "customerId" : "rgpp0wkpec",
  "creationDate" : "2021-07-07T13:40:52.201+00:00",
  "policyPeriod" : {
    "startDate" : "2018-02-04T23:00:00.000+00:00",
    "endDate" : "2018-02-09T23:00:00.000+00:00"
  },
  ...
}
```

A second request to the respective endpoint could then be performed to retrieve the customer data. However, the API description specifies that the GET operation may take an optional `expand` parameter:

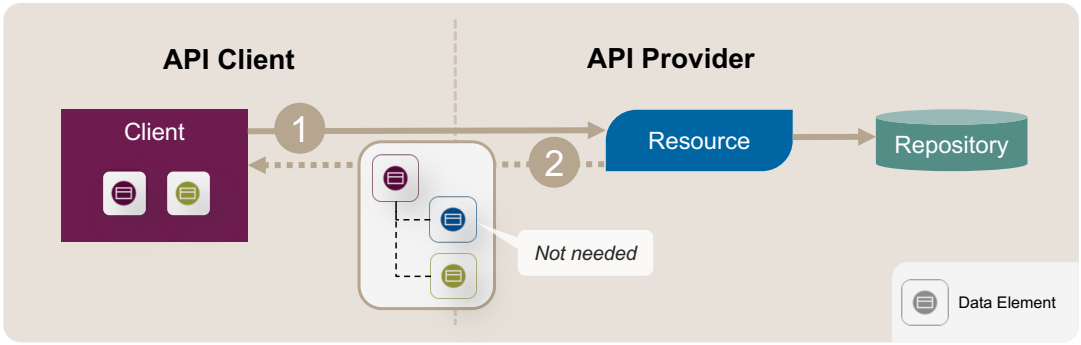

Figure 5: Add Wish List: Initial Position Sketch. The API provider responds to a request from a client (1) with a message (2) that contains Data Elements. The client does not require all the received data.

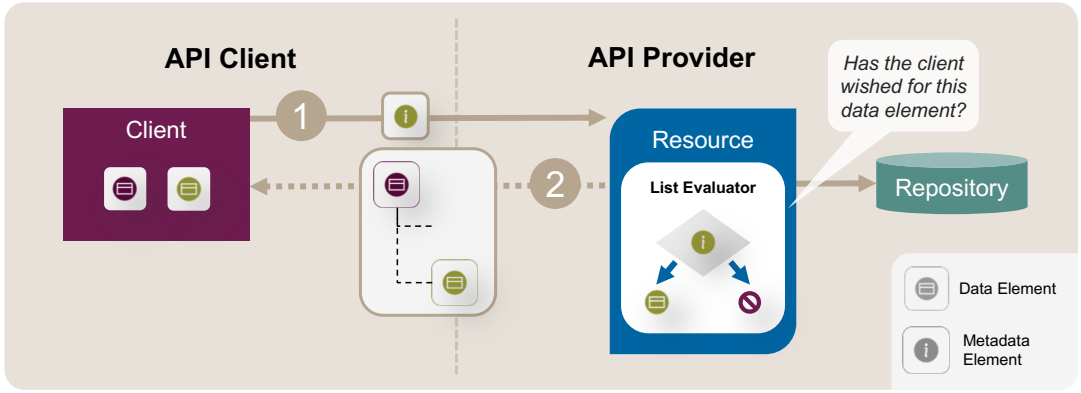

Figure 6: Add Wish List: Target Solution Sketch. In its request (1), the client also sends a Wish List. The provider uses this wish to decide whether some Data Element should be included in the response message (2). The provider implementation might also delegate this filtering to the repository component to avoid retrieving unneeded data from the data store.

```
'/policies/{customerIdDto}':
  get:
    tags:
      - customer-information-holder
    summary: Get a customer's policies.
    operationId: getPoliciesUsingGET
    produces:
      - '*/*'
    parameters:
      - name: customerIdDto
        in: path
        description: the customer's unique id
        required: true
        type: string
      - name: expand
        in: query
        description: a comma-separated list of the fields
          that should be expanded in the response
        required: false
        type: string
    ...
```

Adding `expand=customer` to the query string results in the following response, which now includes customer master data:

```
curl http://localhost/policies/fvo5pkqerr?expand=customer
{
  "policyId" : "fvo5pkqerr",
  "customer" : {
    "customerId" : "rgpp0wkpec",
    "firstname" : "Max",
    "lastname" : "Mustermann",
    "birthday" : "1989-12-31T23:00:00.000+00:00",
    "streetAddress" : "Oberseestrasse 10",
    "postalCode" : "8640",
    "city" : "Rapperswil",
    "email" : "admin@example.com",
    "phoneNumber" : "055 222 4111",
    "moveHistory" : [ ],
    ...
  },
  "creationDate" : "2021-07-07T13:40:52.201+00:00",
  "policyPeriod" : {
    "startDate" : "2018-02-04T23:00:00.000+00:00",
    "endDate" : "2018-02-09T23:00:00.000+00:00"
  },
  ...
}
```

The client does not have to issue a second request for the data when using the Wish List.

3.2.8 Hints and Pitfalls to Avoid. If some of the data that can be referred to in the wishes resides in another API endpoint, the API implementation is now depending on that other endpoint, which could not be permitted or desired according to the overall architecture that has been decided for.

Remember that security measures might have to be adjusted because customers might retrieve more data in a single call than before the refactoring. Measures could include checking for authorization and other quality management concerns, like accounting for Rate Limits.

If providing a single, general-purpose operation for different clients does not feel right, there is nothing wrong with having specialized operations, endpoints, or even entire APIs for clients. See the Backend for Frontend pattern for a deeper discussion of the pros and cons of this approach.

Too many optional parameters can lead to difficult-to-use APIs, impeding learnability and resulting in a complex implementation logic.

3.2.9 Related Content. Step 2 of this refactoring applies *Introduce Data Transfer Object*. The MDSL Tools contain an implementation of this refactoring.

Split Operation could be used to undo this refactoring: if the Wish List is small (for example, there might only be a single data element that can be selected), it might be better to offer two separate operations instead of one with an optional Wish List parameter.

In the example above, we saw that the entire customer entity was included in the response. If only parts of that data are used, the Wish Template pattern provides a mock object-based approach to further tailor the response to the client's wishes. A Wish List can be evolved and graduated into such a Wish Template with the *Add Wish Template* refactoring.

The Known Uses[2] section of the Wish List pattern explains variants and implementation options. For example, Atlassian JIRA has a concept of *resource expansion*.

A Wish List that is introduced with this refactoring can also provide flexibility regarding the content of request messages. For example, when partially updating server-side data, instead of offering many distinct operations that each update a specific field, a single operation can be provided. The request parameters of a Wish List are not used to tailor the response but instruct the endpoint on what data to update. "Practical API Design at Netflix, Part 2: Protobuf FieldMask for Mutation Operations" in the Netflix technology blog shows this in the context of gRPC using Field Masks, the gRPC and Protocol Buffers pendant to our Wish List pattern.

3.3 Refactoring: Introduce Pagination

also known as: Paginate Responses, Slice Response Message

3.3.1 Context and Motivation. An API operation returns a large sequence of Data Elements. For example, such a sequence may enumerate posts in a social media site or list products in an e-commerce shop. The API clients are interested in all Data Elements in the sequence but have reported that processing a large amount of data at once is challenging for them.

> *As the API provider, I want to return data sets in manageable chunks so that clients are not overwhelmed by a huge amount of data arriving at once.*

3.3.2 Stakeholder Concerns (including Quality Attributes and Design Forces).

#performance, #resource-utilization Transferring all Data Elements at once can lead to huge response messages that burden receiving clients and the underlying infrastructure (i.e., network and application frameworks as well as databases) with a high workload. For instance, single-page applications that receive several megabytes of JSON might freeze until all contained JSON objects have been decoded.

#data-access-characteristics In principle, the client wants to access all data elements, but not all have to be received at once or every time. For example, older posts to a social media site might be less relevant than recent ones and can be retrieved separately.

3.3.3 Initial Position Sketch. The API provider currently returns an extensive sequence of Data Elements in the response messages of the operation. Figure 7 shows this initial position sketch.

This refactoring targets an API Retrieval Operation and its request and response messages.

3.3.4 Design Smells.

High latency/poor response time Responses take a long time to arrive at the client because a lot of data has to be assembled and transmitted. This might be evident in a provider-side log file analysis or client-side performance metrics.

Overfetching A client may not need all data (at once or at all) and truncate an overly large dataset. Since this truncation happens on the client side, data was unnecessarily processed and transmitted.

Spike load Regular requests for large amounts of data cause Periodic Workload [8] for CPU and memory, for instance, when a large JSON object has to be constructed (on the provider side) and read (on the client side). For example, the "Time-Bound Report" variant of a Retrieval Operation might lead to relatively large responses, depending on the time interval size chosen.

3.3.5 Instructions (Steps). Decide on a variant of Pagination that best fits your API: Page-Based, Offset-Based, Cursor-Based, or Time-Based Pagination. Clients request the data differently in these variants; see the Pagination pattern description [48] for details on the variants and their pros and cons.

1. All variants involve certain metadata, so if the current response message directly returns the underlying domain model elements, possibly contained in a list, wrap the structure in a Data Transfer Object (DTO) first by applying the *Introduce Data Transfer Object* refactoring.
2. Add additional response attributes to the DTO to hold the metadata required for Pagination (for instance, page size, page number, and the total number of pages for the Page-Based pattern variant).

[2] https://www.api-patterns.org/patterns/quality/dataTransferParsimony/WishList#sec:WishList:KnownUses

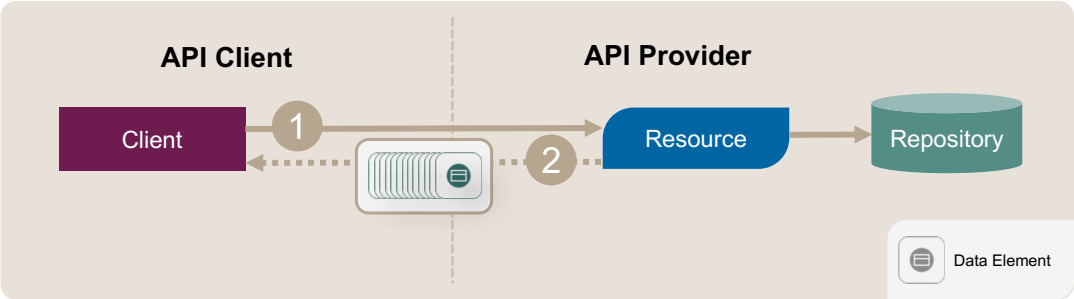

Figure 7: Introduce Pagination: Initial Position Sketch. The client's request (1) is met by a large sequence of data elements (2).

3. Adjust the expected parameters in the request message to give the client control over the number of results returned. Provide default values so that existing clients will continue to work.
4. Enhance the unit and integration tests to include and check for these additional attributes. Test with different chunk sizes. Include complete versus partial retrievals and changes to data while being paginated to the test suite.
5. Update API Description, sample code, tutorials, etc., with information about the Pagination options (for instance, variant, metadata syntax, semantics, and session management concerns).
6. Increase the version number as suggested under Semantic Versioning. The refactoring typically results in a major update, but a minor update might be sufficient if the API provider implements the change in a backward-compatible way.

When already following the API best practice of consistently returning an object as a top-level data structure, it is straightforward to implement Pagination in a backward-compatible manner, returning all results as a single page if no control metadata appears in incoming requests. While this approach is backward-compatible, it does not remove any of the above smells.

3.3.6 Target Solution Sketch (Evolution Outline). After the refactoring, the client indicates the desired amount and position in the sequence of data in their request messages (depending on the Pagination variant). In Figure 8, the number of elements, offset (desired first element, that is), and so on is represented by Metadata Elements.

More but smaller messages are exchanged after the refactoring has been applied.

3.3.7 Example(s). In this example, we will add Offset-Based Pagination to the Customer Core service of the Lakeside Mutual sample application. The `customers` endpoint in this service returns a list of customer representations:

```
$ curl http://localhost/customers
[ {
  "customerId" : "bunlo9vk5f",
  "firstname" : "Ado",
  "lastname" : "Kinnett",
  ...
}, {
  "customerId" : "bd91pwfepl",
  "firstname" : "Bel",
  "lastname" : "Pifford",
  ...
} ]
```

Note that the response is a JSON array of objects. To transmit the Pagination metadata, we first wrap the response in a JSON object (this wrapping is usually done by introducing a DTO that encapsulates the Data Elements), with a `customers` property to hold the entities:

```
$ curl http://localhost/customers
{
  "customers" : [ {
    "customerId" : "bunlo9vk5f",
    "firstname" : "Ado",
    "lastname" : "Kinnett",
    ...
  }, {
    "customerId" : "bd91pwfepl",
    "firstname" : "Bel",
    "lastname" : "Pifford",
    ...
  } ]
}
```

Unfortunately, this makes the response backward incompatible. Initially, the array was returned at the top level of the response, but now it is nested inside a `customers` object. Enabling such future extensibility is why API guidelines (e.g., from Zalando) recommend always returning an object as the top-level data structure in the first place.

With the basic structure in place, we can now add HTTP query parameters (`limit`, `offset`) and return the Pagination metadata (`limit`, `offset`, `size`) in our response. Here is a request for the next chunk of elements (including the JSON response to it):

```
$ curl http://localhost/customers?limit=2&offset=2
{
  "limit" : 2,
  "offset" : 2,
  "size" : 50,
  "customers" : [ {
    "customerId" : "qpa66qpilt",
    "firstname" : "Devlin",
    "lastname" : "Daly",
    ...
  }, {
```

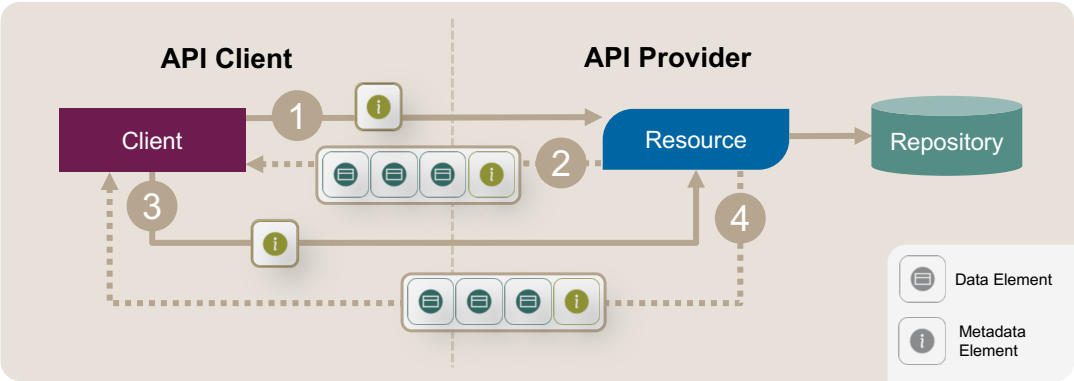

Figure 8: Introduce Pagination: Target Solution Sketch. After the refactoring, the client includes metadata in the request (1) that tells the provider which elements to return. In its response (2), the provider returns the desired elements, along with more metadata (for instance, the total number of elements available). This exchange can then be repeated with follow-up requests (3) and responses (4), where the client can specify the next page (or offset, cursor, depending on the chosen pattern variant).

```
    "customerId" : "en2fzxutxm",
    "firstname" : "Dietrich",
    "lastname" : "Cordes",
    ...
  } ]
}
```

See the Lakeside Mutual repository for the full Spring Boot implementation, including HATEOAS links and filtering.

3.3.8 Hints and Pitfalls to Avoid. The Data Elements the operation returns typically have an identical structure, as in our example above. Still, Pagination can also be used if the structures of the individual Data Elements differ from each other, as long as there is a sequence of elements that can be split up. If the structure of the response does not comprise a sequence of elements that can be split into pages, the *Extract Information Holder* refactoring offers an alternative solution to reduce the amount of data transferred.

The API implementation should ensure that the order of elements is consistent when implementing Pagination. For example, the API provider can specify an explicit `order by` when querying a relational database. Otherwise, clients might receive inconsistent or duplicate results across multiple pages.

Keep in mind that not all API clients are part of end user applications. Backend services can also be API clients that may want to paginate the data they receive.

If the API deployment infrastructure involves load balancers and failover/standby configurations, keep the following in mind:[3]

- The request for a follow-up page (Step 3 of Figure 8) could go to a different API service provider instance than the first initial request. In that case, that (second) instance would perform another database request to retrieve the second page. However, the data on that second page could have changed in the repository between the two page requests. So this only works for static data that does not change often.
- Data consistency/transaction mechanism: Assuming we are dealing with highly dynamic repository data (e.g., the backend database is constantly changing), we need to either make sure that all page requests reach the same service instance that initially retrieved the data from the database (effectively making the service stateful), or develop a caching mechanism in the repository so that data changes between page requests are not causing data inconsistencies in the client.
- If the service instance fails between the two page requests (assuming the service is now stateful, and we have a routing rule to reach the same instance with each page request), the provider has to notify the client that Pagination has failed entirely, and the client then must retrieve the first page again.

Instead of adding Pagination metadata to the body of the response message, it can be transmitted in HTTP headers, as in the GitHub API. This can be an alternative implementation approach if the body of the response message cannot be adjusted for backward compatibility.

3.3.9 Related Content. The *Introduce Data Transfer Object* refactoring prepares request and response messages to introduce the Pagination metadata.

"Patterns for API Design" [48] describes Pagination and its variants in detail and points at additional information.

3.4 Refactoring: Split Operation

also known as: Decompose API Call

3.4.1 Context and Motivation. An API with endpoints and operations has been defined and implemented. Some operations acquire multiple responsibilities because new capabilities are added to the API. For example, an operation might accept several types of request messages that lead to different parts of execution logic in the API implementation.

> *As an API provider, I want to offer endpoints that expose operations with distinct responsibilities so that the API is*

[3] Thanks to Andrei Furda for suggesting this advice.

easy to understand and use for client developers and can be modified rapidly and flexibly by provider developers.

3.4.2 Stakeholder Concerns (including Quality Attributes and Design Forces).

#single-responsibility-principle and #understandability An API that offers endpoints with operations that follow the *single responsibility principle* is easier to understand for clients because operations are focused and do not offer extra capabilities that not all clients use. The API provider team also benefits from a lower complexity of the implementation.

#maintainability, #evolvability Operations that follow the *single responsibility principle* are easier to maintain and evolve, for instance, because their API provider only has a single reason to change their request and response message structure and content when evolving the API. Changing an operation that does many things is unnecessarily complex if the change only affects certain aspects of the operation; unexpected side effects may occur. For example, deprecating the operation is more complicated if it has multiple stakeholders.

#testability Test cases have to cover all, and all combinations of, the many things the operation is in charge of. Testing such an operation is more complex than testing a single-responsibility operation.

#security Access restrictions of an API can be implemented on different levels: whole API, per endpoint, individual operations, or even depending on the executed control flow paths in the implementation of operations or the data accessed. Securing an operation that does many things is complex. The access control list must include the superset of all involved participants; changing it has a significant impact.

3.4.3 Initial Position Sketch. The implementation logic chosen by the operation depends on data sent by the client (represented by the Metadata Element). This branching parameter could take the form of a boolean parameter, an enumeration, or any other value obtained from the request message or the resource state. For example, the request body could contain the data of a resource representation, and depending on whether that resource is already known to the API, it is either updated or created. Figure 9 visualizes this initial position.

This refactoring targets an endpoint implementation and its request and response messages.

3.4.4 Design Smells.

Role and/or responsibility diffusion* and *low cohesion An operation does (too) many things. Clients have to understand all these things to use the operation correctly. Its request message is rather deeply structured and may contain optional, generic, or variable parts to express diverse input options. This complexity may lead to errors and a degraded developer experience. The internal cohesion of the operation is low.

Combinatorial explosion of input options Boolean parameters or other flags that determine the execution path lead to a combinatorial explosion of possibilities. Explaining these options bloats the API Description and is problematic for the client, who has to understand this complex option space to prepare valid requests, and the provider, who has to validate and process the parameter handling. API testing on the client and provider side is also complicated.

Change log jitter* or *commit chaos The operation has been, and continues to be, modified frequently, according to the commit logs kept by the version control system. Frequent changes may indicate that the operation has too many responsibilities and is not focused enough.

3.4.5 Instructions (Steps). To split an operation, the following steps apply:

1. Copy the operation implementation, remove the branching metadata parameter (in the copy), and delete the part of the implementation that is no longer needed for the new operation.
2. (Optional) If there is common code in the two operations, extract it into a new method and call it from both operations. This step is optional because the shared code might be small enough to be inlined in both operations (consider the Rule of Three of refactoring[4] by Fowler [12]).
3. Expose the new operation to clients. Depending on the underlying technology, this can be non-trivial. When using HTTP, choosing a different (previously unused) verb might be appropriate for the new operation. See Singjai et al. [32] for a collection of patterns on designing API operations.
4. Copy the tests covering the implementation parts that now reside in the copy. Remove the obsolete branching metadata parameter from the tests, adjust the test data and assertions, and ensure the tests pass.
5. Include the newly created operation in the API Description. Inform clients that a new operation now covers the previous behavior. The decision logic previously encoded on the provider side must now be implemented on the client side instead of just passing a branching parameter.
6. If access to the original operation was restricted to specific clients, apply the same security rules to the new operation.
7. If necessary, to avoid breaking changes, mark the branching metadata parameter as deprecated or immediately remove it, along with any unused code in the original operation implementation. Note that this decision depends on the lifecycle guarantees given to clients, as documented as one of the Evolution Patterns.

For backward compatibility, a Content-Based Router [18] can forward requests to the correct operation.

3.4.6 Target Solution Sketch (Evolution Outline). After the refactoring, the behavior of the original operation is cut into two parts, which are implemented by two distinct operations (“Op1” and “Op2”).

Splitting the operation into two (or more) distinct operations makes each one easier to use and maintain. No provider-side dispatching or branching logic is required anymore. As a downside, the client implementation might become more complex because it has

[4] Not to be confused with the Rule of Three of the patterns community: Only call it a pattern if there are at least three known uses (https://wiki.c2.com/?RuleOfThree).

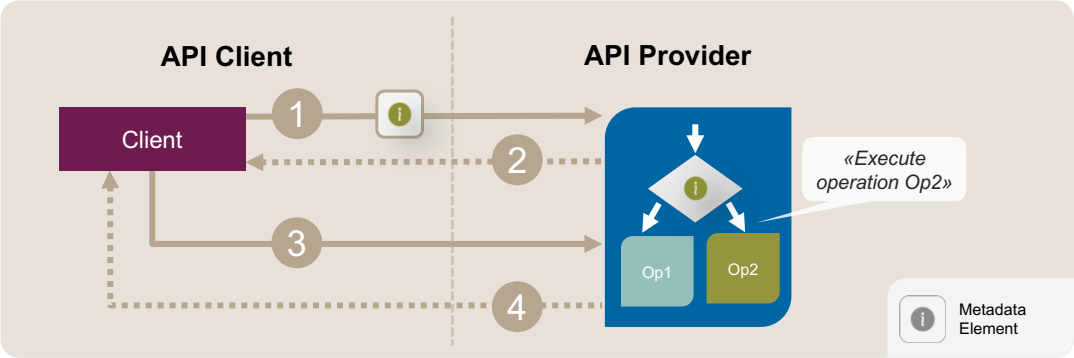

Figure 9: Split Operation: Initial Position Sketch. A client sends a request (1) that includes metadata, such as an enumeration value or boolean flag. The provider uses this information to steer the control flow, choosing between operations (Op1, Op2) to execute and compose a response message (2). This metadata might be optional, as shown in the second exchange (Request 3, Response 4). The provider might choose a default operation or even return an error message in that case.

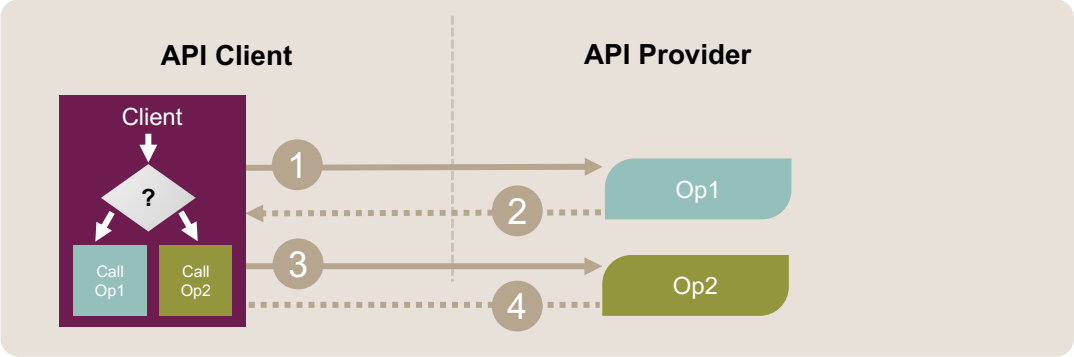

Figure 10: Split Operation: Target Solution Sketch. Instead of using metadata parameters to steer the provider behavior, two distinct operations (Op1, Op2) are offered. Clients can invoke one (Request 1, Response 2) or the other (Request 3, Response 4).

to decide which operation to call. The amount of request-response message exchanges usually stays the same, though.

3.4.7 Example(s). The following example from a construction management API shows a Spring Boot implementation of an endpoint that offers an `updateConstruction` operation to modify the data of a particular building site, specified by the `id` parameter:

```
@PutMapping("/constructions/{id}")
public ResponseEntity<Construction> updateConstruction(
    @PathVariable(value = "id") Long id,
    @PathVariable(value = "partial-update") Boolean partial,
    @NotNull @RequestBody Construction construction) {

    if (!constructionRepository.existsById(id)) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST, "Entity not found");
    }

    Construction result;
    if (partial) {
        result = constructionRepository
            .findById(construction.getId())
            .map(existingConstruction -> {
                if (construction.getName() != null) {
                    existingConstruction.setName(
                        construction.getName());
                }
                // repeat this for all attributes
                return existingConstruction;
            })
            .map(constructionRepository::save).get();
    } else {
        result = constructionRepository.save(construction);
    }

    return ResponseEntity.ok().body(result);
}
```

The operation takes a boolean `partialUpdate` parameter. If it is set to `true`, the attributes that the client provides in the request body are overwritten. If `partialUpdate` is `false`, the entire entity is replaced, as shown in the `else` block.

HTTP provides the `PATCH` verb to represent partial updates (whereas `PUT` methods are supposed to replace the entire resource). So we can move the "patch" parts of the operation to a new operation:

```
@PatchMapping("/constructions/{id}")
public ResponseEntity<Construction> updatePartially(
    @PathVariable(value = "id") Long id,
    @NotNull @RequestBody Construction construction) {

    if (!constructionRepository.existsById(id)) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST, "Entity not found");
    }

    Construction result = constructionRepository
        .findById(construction.getId())
        .map(existingConstruction -> {
            if (construction.getName() != null) {
                existingConstruction.setName(
                    construction.getName());
            }
            // repeat this for all attributes
            return existingConstruction;
        })
        .map(constructionRepository::save).get();

    return ResponseEntity.ok().body(result);
}
```

Note the `PatchMapping` annotation on the `updatePartially` operation that was added. The old `updateConstruction` operation has become much simpler now (the + and − stand for added and removed lines, respectively):

```
 @PutMapping("/constructions/{id}")
 public ResponseEntity<Construction> updateConstruction(
     @PathVariable(value = "id") Long id,
-    @PathVariable(value = "partial-update") Boolean partial,
     @NotNull @RequestBody Construction construction) {
    if (!constructionRepository.existsById(id)) {
         throw new ResponseStatusException(
             HttpStatus.BAD_REQUEST, "Entity not found");
    }
+   Construction result =
+       constructionRepository.save(construction);
-   if (partial) {
-       result = constructionRepository
-           .findById(construction.getId())
-           .map(existingConstruction -> {
-               if (construction.getName() != null) {
-                   existingConstruction.setName(
-                       construction.getName());
-               }
-               // repeat this for all attributes
-               return existingConstruction;
-           })
-           .map(constructionRepository::save).get();
-   } else {
-       result = constructionRepository.save(construction);
-   }
    return ResponseEntity.ok().body(result);
 }
```

In this example, no operation in the endpoint had a `PatchMapping` so far. If this had been the case, we would have had to introduce a new endpoint for the split-off operation and apply *Move Operation* to move either operation to the new endpoint.

Even though the method name `updateConstruction` is an implementation detail and not exposed to API clients, it could also be renamed. For example, `replaceConstruction` would fit better with the new responsibility of the method.

3.4.8 Hints and Pitfalls to Avoid. When using HTTP, follow the conventions of the protocol. For example, a PUT request should be idempotent, meaning that the result of sending such a request is the same whether it has been sent exactly once or multiple times. In contrast, a POST request is not necessarily idempotent. Sending it multiple times might lead to incorrect provider-side state, such as duplicated data. Not following such conventions confuses clients and may cause API usage bugs.

HTTP redirections provide a technical solution for informing clients about the new operation [9]. This approach only works if the URI changes. Using redirects to tell clients to use another HTTP verb is not possible.

With respect to security concerns, the split-off operation should probably be accessible to the same clients as the original operation if authentication and authorization are required.

Possible impacts on Rate Limits, monitoring, caching, and other aspects of the API should be considered as well.

3.4.9 Related Content. *Merge Operations* is the inverse refactoring. *Add Wish List* can also combine two Retrieval Operations that return related data.

Once an operation has been split into two, one can also be moved to a different endpoint with the *Move Operation* refactoring.

When refactoring the API implementation, the Extract Method [12] refactoring is eligible.

The correct use of HTTP verbs and many other REST implementation hints are explained in the RESTful Web Services Cookbook [1].

3.5 Refactoring: Merge Operations

also known as: Colocate Responsibilities, Consolidate Operations

3.5.1 Context and Motivation. An API endpoint contains two operations with similar, possibly overlapping responsibilities. Typically, this was not the intention of the original API designers, but the result of an API evolution process. For instance, the API provider might have added a new operation to the endpoint instead of adjusting an existing one.

> *As an API designer, I want to remove an API operation from an endpoint and let another operation in that endpoint take over its responsibilities so that there are fewer operations to maintain and evolve and the inner cohesion of the endpoint improves.*

3.5.2 Stakeholder Concerns (including Quality Attributes and Design Forces).

#understandability, #explainability, #learnability There are many reasons to change an API (not just refactorings) [35]. APIs should be changed as much as required and as little as possible, and ripple effects be avoided. One of the first steps in related maintenance tasks is to determine which parts of the system should be changed. An API whose

endpoints have clearly identified roles helps developers to quickly understand the API.

#cohesion Cohesive design elements (here: API operations in an endpoint) belong together naturally because they share certain properties. In an API design context, the security rules that apply are an example of such a property. Cohesion is desirable because it makes the system easier to understand and maintain. ISO/IEC/IEEE 24765 [10] defines cohesion as "manner and degree to which the tasks performed by a single software module are related to one another."

#coupling In our context, coupling is a measure of how closely connected and dependent on each other endpoints and their operations are. The coupling may concern the data exchanged and/or the operation call sequencing. See Wikipedia entry for Coupling and Loose Coupling pattern for general explanations.

3.5.3 Initial Position Sketch. The refactoring is applicable if the current API exposes an endpoint with at least two operations, as shown by the following MDSL snippet:

```
endpoint type Endpoint1BeforeMerge
exposes
  operation operation1
    expecting payload "RequestMessage1"
    delivering payload "ResponseMessage1"

   operation operation2
    expecting payload "RequestMessage2"
    delivering payload "ResponseMessage2"
```

As the snippet shows, this refactoring targets a single endpoint and two of its operations. Figure 11 visualizes this initial position. The API offers two operations, which clients may or may not call in any particular order.

3.5.4 Design Smells.

Responsibility spread Endpoint roles and/or operation responsibilities are rather diffuse; the Single Responsibility Principle is violated. For instance, API clients serving a particular stakeholder have to call multiple operations to satisfy their information needs. Another example would be that a choreographed or orchestrated business process implementation has to consult too many distributed operations to fulfill its job.

High coupling Two or more operations perform narrowly focused, rather low-level activities. Clients have to understand and combine all of these activities to achieve higher goals, leading to a degraded developer experience and coordination needs. This causes these operations to be coupled with each other implicitly.

REST principle(s) violated The "Uniform interface" is an important design constraint imposed by the REST style that many HTTP APIs employ. REST mandates using the standard HTTP verbs (POST, GET, PUT, PATCH, DELETE, etc.), which are associated with additional constraints. For instance, GET and PUT requests must be idempotent to be cachable [1]. Sometimes, mismatches between the API semantics and the REST constraints can be observed; sometimes, the REST constraints limit extensibility (for instance, when a resource identified by a single URI runs out of verbs) [30].

3.5.5 Instructions (Steps). This refactoring requires careful planning and execution:

1. Merge the data structures used in the two request messages if they differ. A straightforward approach is to combine and wrap the original message contents in a new DTO (see *Introduce Data Transfer Object*) in the consolidated message.
2. (Optional) Add a boolean flag to the request message to distinguish the merged operations, for instance, to dispatch the request to the proper API implementation logic. This step is optional because it might be possible to inspect the merged request message to select the implementation logic (see the example later).
3. Merge the response message data structures as well. A DTO can be used to do so.
4. Consolidate the implementation code, deciding how to route incoming requests and how to prepare the consolidated response.
5. Add at least two tests if the boolean flag introduced in Step 2 is present: one sets it to `false`, and the other sets it to `true`. Test the new API and compare old and new behavior.
6. Update supporting artifacts such as API Description and usage examples. Show how to use the boolean flag (if introduced) and explain how to migrate from the old to the new API.
7. Inform the API user community about the change and its rationale. Make the news item self-contained or provide direct links to the updated API Description and usage examples; avoid general statements such as "We have changed our API in an incompatible way. Please consult the documentation to learn how."

The changes introduced in Steps 1 to 4 are not backward-compatible per se. Steps 5 to 7 apply to most refactorings in our catalog; we refer to them as TELL (Test, Explain, Let Know, and Learn).

3.5.6 Target Solution Sketch (Evolution Outline). The API contract from the Initial Position Sketch above still contains one endpoint. But only one operation is present now (note that `{ , }` is the MDSL notation for Parameter Trees [47], an abstraction of JSON objects):

```
data type ConsolidatedRequestMessage {
    "RequestMessage1", "RequestMessage2"
}
data type ConsolidatedResponseMessage {
    "ResponseMessage1",  "ResponseMessage2"
}

endpoint type Endpoint1AfterMerge
exposes
  operation operation1and2Merged
    expecting payload
        "RequestMessage12":ConsolidatedRequestMessage
    delivering payload
        "ResponseMessage12":ConsolidatedResponseMessage
```

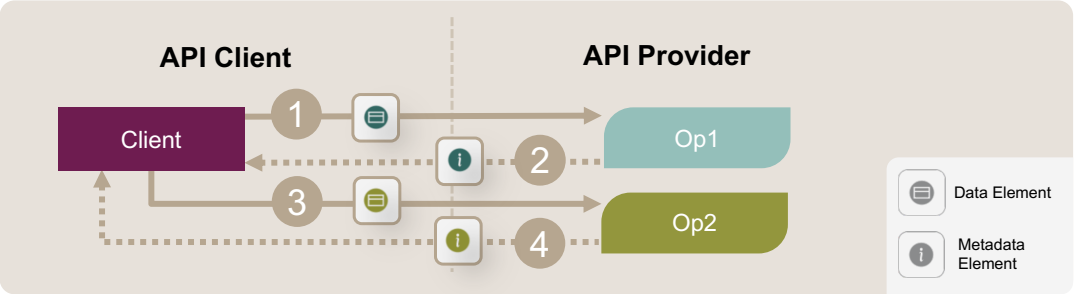

Figure 11: Merge Operations: Initial Position Sketch. The API provider offers two distinct operations (Op1, Op2). The client can invoke one (Request 1, Response 2) or the other (Request 3, Response 4). Some metadata and other data elements are exchanged.

Figure 12 visualizes the resulting API design that uses a Content-Based Router to select the operation to execute.

3.5.7 Example(s). In the following example of the user administration endpoint of an API implemented in Spring Boot, there are two operations to change the e-mail address and username, respectively. Both use the same endpoint `/users/{id}`, but the developer decided to use different HTTP verbs (POST and PATCH) to implement the two operations:

```
@PostMapping("/users/{id}")
public ResponseEntity<User> changeEmail(
    @RequestBody ChangeEmailDTO changeEmailDTO)  {
    log.debug("REST request to change email : {}",
        changeEmailDTO);
    ...
}

@PatchMapping("/users/{id}")
public ResponseEntity<User> changeUsername(
    @RequestBody ChangeUsernameDTO changeUsernameDTO) {
    log.debug("REST request to change username : {}",
        changeUsernameDTO);
    ...
}
```

Keep in mind that the API client does not see these method names but the `POST` and `PATCH` verbs only. Using different HTTP verbs simply to distinguish between two operations violates REST principles. `POST` is meant for creating resources, not updating them, as done in `changeEmail`. These endpoints can then be used as follows:

```
curl -X POST  api/users/123 -d '{ . . . }'
curl -X PATCH api/users/123 -d '{ . . . }'
```

The HTTP verb used is the only difference from the perspective of the API client; the fixed amount of available HTTP verbs limits future extensibility given (maybe passwords should also be changeable?). Hence, it is decided to merge the two operations and create a composite request message DTO:

```
class ChangeUserDetailsDTO {
    ChangeEmailDTO changeEmail;
    ChangeUsernameDTO changeUsername;
}

@PatchMapping("/users/{id}")
public ResponseEntity<User> changeUserDetails(
    @RequestBody ChangeUserDetailsDTO changeUserDetailsDTO) {
    if (changeUserDetailsDTO.changeEmail != null) {
        log.debug("REST request to change email : {}",
            changeUserDetailsDTO.changeEmail);
        ...
    }
    if (changeUserDetailsDTO.changeUsername != null) {
        log.debug("REST request to change username : {}",
            changeUserDetailsDTO.changeUsername);
        ...
    }
    ...
}
```

Further operations changing other properties of the user can now be implemented by extending the DTO without introducing new operations or even endpoints. The DTO content determines the nature of the change; no boolean parameter was needed in this example. Clients can now also initiate several changes in a single request.

3.5.8 Hints and Pitfalls to Avoid. Merging operations is more challenging than merging endpoints (which usually merely group operations under a unique address such as a parent URI):

- The operations to be merged must appear in the same endpoint. Apply *Move Operation* first if needed.
- Do not break HTTP verb semantics when merging (in HTTP resource APIs). For instance, idempotence might get lost if a replacing PUT and an updating PATCH are merged.
- When merging the request and response messages, decide where and how the merged messages embed the original message elements. Some data exchange formats have first-class concepts for choices. If this is not the case, the optionality of list items combined with feature flags/toggles can be used.
- The complexity of the implementation logic and the tests increases when merging operations: The implementation logic must distinguish between the merged operations. The tests must cover all possible combinations of the merged operations.

Implementing this refactoring in a backward-compatible way is not trivial because of the changes imposed on request and response messages. One tactic could be to provide a new operation for the

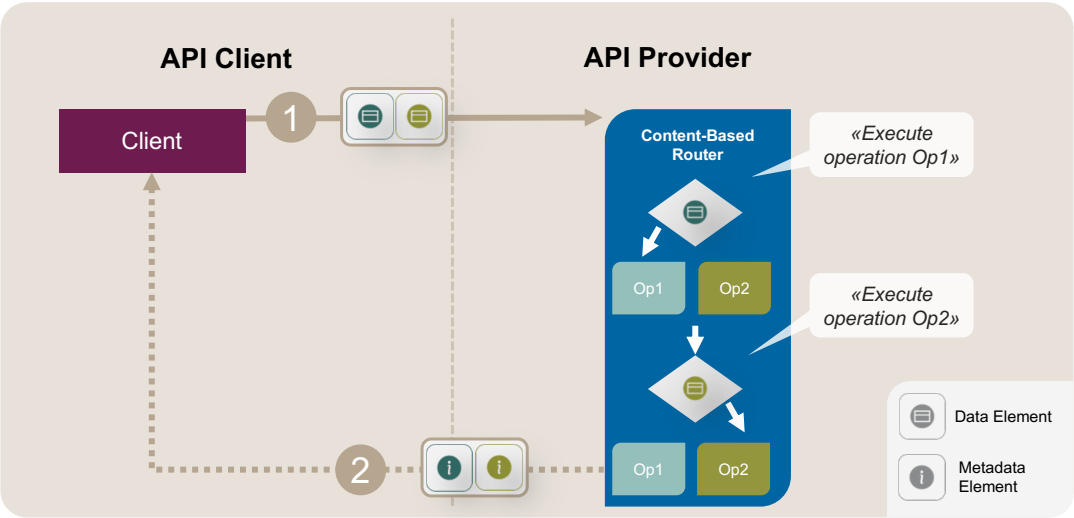

Figure 12: Merge Operations: Target Solution Sketch. The client sends a request (1) that includes one or more data elements. This figure shows two data elements, but they might not all be mandatory. The provider uses a Content-Based Router to execute operations (Op1, Op2) depending on the content of the message and returns a response, for example, some metadata (2).

merged functionality and keep the original ones in place. The original ones can then forward incoming requests to the new operation, wrapping and un-wrapping request and response messages.

See the *Merge Endpoints* refactoring for Confidentiality, Integrity, and Availability (CIA) considerations; inconsistent or inappropriate CIA settings (authentication, authorization) are less likely to result from this refactoring (depending on how the API endpoint and operations have been identified) but are still worth considering.

3.5.9 Related Content. This refactoring reverts *Split Operation*. If the two operations do not reside in the same endpoint, an upfront *Move Operation* refactoring can prepare its application.

The Service Cutter tool and method suggest sixteen coupling criteria [14]. These criteria primarily apply when merging endpoints but are also worth considering when merging operations.

3.6 Refactoring: Rename Representation Element

also known as: Rename Parameter, Rename Payload Part

3.6.1 Context and Motivation. An API endpoint operation expects data from clients and/or delivers data to them. The data representations are structured, and certain parts of the structural elements are named (for instance, keys in key-value pairs or type definitions).

> *As a data modeler and/or Data Transfer Object (DTO) designer, I want to use expressive and domain-specific terminology so that data representations and their elements become self-explanatory to API developers on the client as well as on the provider side.*

3.6.2 Stakeholder Concerns (including Quality Attributes and Design Forces).

#understandability, #explainability, #learnability A well-chosen name expresses what a representation element (such as a JSON object with its properties) has to offer, which helps clients decide whether and how to use it. The named element has a single meaning and purpose.

#maintainability The more expressive and meaningful a name (for instance, the key in a key-value pair) is, the easier it is to search and navigate the code base, for instance, when fixing bugs and adding features during API evolution. *Split Operation* covers this quality in more detail.

3.6.3 Initial Position Sketch. Let us assume the following simple endpoint design (notation: abstract MDSL):

```
endpoint type Endpoint1
exposes
  operation operation1
    expecting payload {
      "v1":D<string>,
      "v2":D<int>}
    // one-way exchange in this example,
    // so no response message
```

As the sketch shows, the refactoring targets a single endpoint and one of its operations. Presumably, the v in v1 and v2 stands for "value", but this does not get clear from the terse one-letter, one-number acronyms v1 and v2.

3.6.4 Design Smells.

Cryptic or misleading name The chosen element name is difficult to understand for stakeholders unfamiliar with the API implementation. For instance, it might not be part of the agreed-upon domain vocabulary or unveil implementation details such as column names in database tables. It might

also be ambiguous and overloaded with different meanings (in the same context).

Security by obscurity Sometimes, it is argued that unlabeled, undocumented data is harder to tamper with. But such a tactic alone does not qualify as a sound security solution. It harms maintainability because it increases the risk of introducing bugs because of a lack of clarity for maintainers and auditors, not just attackers.

Change log jitter* or *commit chaos The name has been, and continues to be, frequently changed according to the logs kept by the version control system. Frequent changes indicate that the domain language is not yet stable or has not yet been defined, communicated, and agreed upon sufficiently.

Leaky encapsulation* and *high coupling Program-internal names or identifiers might accidentally have leaked into the API. For example, the initial API could have been generated from internal classes. For API clients, such internal names might be hard to understand.

3.6.5 Instructions (Steps). This refactoring is rather straightforward to apply:

1. Discuss alternative new names and decide on one. Have this decision reviewed and agreed upon.
2. Apply code-level rename refactorings; do not forget to update code-level comments when doing so. If the code is generated from a specification, update the API design specification and then re-generate the code.
3. Update API tests and API Description.[5]

Several strategies for backward compatibility exist. The implementation could continue to accept the old names as well. However, this approach can quickly become unwieldy if many such changes accumulate over time. By applying the *Introduce Version Mediator* refactoring, clients can continue using the old names which will then get translated by the version mediator to the new names.

3.6.6 Target Solution Sketch (Evolution Outline). The element names `v1` and `v2` in the initial position have been replaced by `customerName` and `customerID` in this MDSL sketch:

```
endpoint type Endpoint1
exposes
  operation operation1
    expecting payload {
      "customerName":Data<string>,
      "customerID":Data<int>
    }
```

Arguably, `customerName` is more expressive than `v1` unless the representation element is a simple loop counter (which does not seem to be the case). This renaming makes the API specification easier to understand, assuming that a common understanding of the term "customer" has been reached within a context and community. `operation1` remains generic in the above sketch, which makes it a candidate target for *Rename Operation*.

3.6.7 Example(s). In the Policy Management backend of the Lakeside Mutual sample application, we find a Data Transfer Object (DTO) called `InsuranceQuoteDto`, which is exposed in the Web API that is provided by a RESTful HTTP controller:

```
public class InsuranceQuoteDto {
    @Valid
    @NotNull
    private Date expirationDate;

    @Valid
    @NotNull
    private MoneyAmountDto insurancePremium;

    @Valid
    @NotNull
    private MoneyAmountDto policyLimit;

    ...
}
```

The name `InsuranceQuoteDto` as well as `expirationDate`, `insurancePremium`, and `policyLimit` are all expressive names; names such as `inputData`, `date`, and `amount` would be less domain-specific and could be renamed with this refactoring.

One could argue that abbreviations and technical terms should be avoided in API naming. Removing `Dto` from the name in a refactoring would make the name shorter and cleaner but also hide the architectural role played by the class and go against the "architecturally-evident coding style" recommended by Fairbanks [7]. In this particular case, the acronym is explained in a comment in the source code.

3.6.8 Hints and Pitfalls to Avoid. Coming up with good names is challenging, and some authors consider it one of the hardest problems in software engineering [3]. When choosing names, keep in mind that good names should:

- Precisely reveal the purpose/intent of the named element.
- Be intelligible, so they do not need to be deciphered first.
- Be pronounceable so that they can be talked about.
- Be as simple and short as possible, but not shorter.
- Adhere to the conventions of the interface specification and/or programming language in use.

When applying this refactoring, one should promote a controlled amount of domain vocabulary into the published language of the API. Optionality and value ranges should be explained in the API documentation. Comments in machine-readable specifications and specification documents targeting humans are valid locations for this information. The reuse of already existing data structures and standard vocabularies may be considered if that is permitted, for instance, microformats or schema.org[6].

Mapping implementation-level data structures one-to-one introduces undesired tight coupling between API and implementation and should therefore be avoided. Implementation-level data structures should be wrapped or mapped (see *Introduce Data Transfer Object*) to hide implementation details.

It makes sense to avoid technical jargon (in particular jargon that might go out of fashion soon or has a different meaning in other communities already). As already touched upon in the example, abbreviations should be avoided in names unless they are

[5] Step 3 applies to all refactorings and is part of the general Test, Explain, Let Know and Learn (TELL) activities to enact and evaluate refactorings (see Section 4 for more information).

[6] Example: how should we name and structure the attribute to contain a user address? Easy, use the already existing definition https://schema.org/PostalAddress.

widespread in the API stakeholder community. Humor is a good thing, of course, but technical specifications are not necessarily a good place to (try to) be funny; ethical codes of conduct must not be violated when choosing names (e.g., [CoC:ACM]).

3.6.9 Related Content. Representation elements can be grouped by applying *Introduce Data Transfer Object*. *Rename Operation* and *Rename Endpoint* are available to rename API parts with a larger scope. The ID ELEMENT pattern [PatternsForAPIDesign:2022] makes naming suggestions (for instance, regarding uniqueness) and provides pointers to additional information. DATA ELEMENT is the general pattern for any kind of representation element.

The book "The Programmer's Brain: What every programmer needs to know about cognition" by Hermans [16] has a chapter on "How to get better at naming things" that explains the cognitive processes involved in reading names in code and provides practical advice.

Fowler [12] covers various rename refactorings, such as "Rename Variable" and "Rename Field." And "The Art of Readable Code" [5] has helpful hints on naming program elements that also apply to APIs. Benner [3] provides rich advice as well.

"API Handyman" Arnaud Lauret provides many examples and counterexamples in his book [21] and the online API Stylebook. Applying this refactoring might be the result of an API review, and a new name then has to be decided upon. "A Checklist for API Design Review" is available online.

The blog post "A Definition of Done for Architectural Decisions" presents five criteria to assess whether a decision such as a name change has been elaborated upon sufficiently: evidence, criteria, agreement, documentation, and realization/review.

3.7 Refactoring: Segregate Commands from Queries

also known as: Introduce Command Query Responsibility Segregation (CQRS)

3.7.1 Context and Motivation. An endpoint cohesively bundles all operations dealing with a particular domain concept. Some of these operations modify the application state on the API provider side, while others only retrieve data. Some but not all read operations (following the RETRIEVAL OPERATION pattern [48]) offer declarative query parameters and return rich, multi-valued response structures, causing provider-side workload.

> *As an API provider, I want to serve queries and process commands separately so that I can optimize the respective read and write model designs independently.*

This distinction between commands and queries is known as the Command Query Separation (CQS) principle by Meyer [24]. CQS states that every method in an object-oriented program should either be a command that performs an action and thus changes state or a query that returns data to the caller, but not both.

3.7.2 Stakeholder Concerns (including Quality Attributes and Design Forces).

#performance and #scalability Computationally expensive workloads such as loading data from data stores, filtering, and formatting it, and high data volumes may make certain operations expensive. Such complex query operations should not slow down cheaper operations exposed by the same endpoint (for example, atomic updates of single attribute values).

#agility and #development-velocity Read operations and write operations may evolve at different speeds. For example, data analytics queries may often change, driven by client demand and insights just gained, while commands to modify master data might only change with major releases, if at all.

#flexibility to change the API vs. #simplicity Keeping the read and write operations of an endpoint together is easy to understand and brings functional endpoint cohesion. A separation of these types of operations increases the ability to change them independently from each other; this can then happen more flexibly and more frequently.

#security, #data-privacy Read and write operations might have different protection needs. Few user roles, for instance, are usually authorized to update master data; many or all user roles may read it. If there are two separate endpoints for read and write access, it might be easier to fine-tune the Confidentiality, Integrity, and Availability (CIA) rules and related compliance controls. See the OWASP API Security Top 10 for risks and related advice on API security.

3.7.3 Initial Position Sketch. The operations an API endpoint offers can be sorted into four categories, depending on whether they read/write state. Each target quadrant is represented by a "Pattern for API Design" [48], as shown in Figure 13.

- COMPUTATION FUNCTIONS derive a result solely from the client input, neither reading nor writing server-side state.
- STATE CREATION OPERATIONS initialize some new state at the API endpoint (for instance, by creating an implementation resource such as a customer record). A minimal amount of state can be read, for example, to ensure the uniqueness of identifiers.
- RETRIEVAL OPERATIONS are read-only queries that clients use to fetch data.
- STATE TRANSITION OPERATIONS update the server-side state, including full or partial replacement and deleting of state.

These operations are often implemented as CRUD (create, read, update, delete) resources, shown in Figure 14.

The target of the refactoring is an endpoint, such as an INFORMATION HOLDER RESOURCE [43] that offers both state-writing and state-reading operations. These operations can be realized by HTTP verbs/methods such as POST, GET, PUT, PATCH, and DELETE, supported by an HTTP resource that a URI identifies.

3.7.4 Design Smells.

High latency/poor response time Poor performance may be caused by too tight operation coupling. Expensive queries slow down the execution of write operations (for instance, operations performing state creation or transition). Transactional isolation is insufficient.

Feature/release inertia a.k.a. stale roadmap An endpoint provides both read and write operations; there might be many read, but only few write operation calls. These types of operations evolve at different speeds; possibly, different

		State Write: No	State Write: Yes
State Read	No	Computation Function	State Creation Operation (create)
	Yes	Retrieval Operation (read)	State Transition Operation (update, delete)

Figure 13: The combination of reading and writing state leads to four different operation responsibilities.

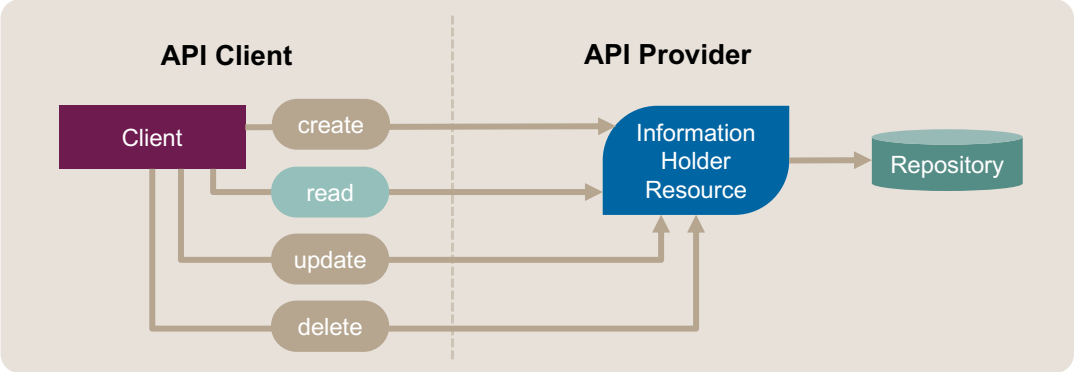

Figure 14: Segregate Commands from Queries: Initial Position Sketch. Commands and Queries in Same Endpoint

development teams are responsible for them. For instance, new query options in a customer relationship management application may be introduced in every two-week iteration in response to frequently arriving customer inquiries and client insights. In contrast, commands evolve with a frequency imposed by a master data management or Enterprise Resource Planning (ERP) package in the backend. The operations also differ in the amount of design and test work required; write operations change state and, therefore may have nontrivial "given" preconditions and "then" postconditions and require consistency management. The conceptual integrity of the endpoint and all of its read and write operations has to be preserved during each evolution step. As a result, it takes longer than desired to introduce new features, new queries in particular.

Too coarse-grained security or data privacy rules The security and data protection requirements of commands and queries differ. They are specified on the endpoint rather than the operation level. Hence, generalization has to take place that bears risks such as under-specification and over-engineering.

3.7.5 Instructions (Steps). Command Query Responsibility Segregation (CQRS) is an architectural pattern that increases flexibility but adds complexity. It can be introduced in the following steps:

1. Classify and group endpoint operations by their purpose and impact on provider-side state: read-only, write-only, read-write, neither-read-nor-write.
2. Apply the *Extract Endpoint* refactoring to move the read-only operations to a new endpoint, the Read Model API.
3. Adjust the API implementation to match the outcome of Steps 1 and 2. Consciously decide for a data store serving both endpoints, the new Read Model API and the already existing endpoint that has become a Write Model API.
4. (Optional) Distribute the data store. When distributing data stores, choose suited data replication and consistency management solutions (for example, how current/fresh should the replicated data be?). Include all data stores in the backup and recovery strategy [25].
5. Test "sunny day scenario" as well as "edge" cases and error situations such as slow and temporarily failing network and replication conflicts.
6. Update the API Description, including the technical API contract and supporting documentation.
7. Provide teaching material that covers migration from the old domain concept-oriented API to the new command-query API: What has to be changed in the API client? How do the Service Level Agreements change?

The operation responsibility Computation Function neither reads nor writes provider-side application state.[7] Such operations may appear in command endpoints as well as query endpoints; they might also go to separate stateless endpoints, yielding a "Command Computation Responsibility Segregation" variant of CQRS.

The messages and operations stay the same when applying this refactoring. However, the resource address might change. An intermediary such as an API Gateway [27], Service Registry, or Version Mediator can be used to preserve backward compatibility by mapping or forwarding messages.

3.7.6 Target Solution Sketch (Evolution Outline). After applying the refactoring, shown in Figure 15, two distinct endpoints/resources implement the API operations. One is a Processing Resource handling the commands, and the other is an Information Holder Resource handling the queries.

3.7.7 Example(s). The following example shows an introduction of CQRS (notation: Context Mapper DSL (CML)):

```
Service PublicationManagementFacade {
    // a state creation/state transition operation:
    @PaperId add(@PublicationEntryDTO newEntry);

    // retrieval operations:
    @PublicationArchive dumpPublicationArchive();
    Set<@PublicationEntryDTO>
        lookupPublicationsFromAuthor(String writer);
    String renderAsBibtex(@PaperId paperId);

    // computation operations (stateless):
    String convertToBibtex(@PublicationEntryDTO entry);
```

This single publication management endpoint can be separated into two in this API design:

```
Service PublicationManagementCommandFacade {
    // a state creation/state transition operation:
    @PaperId add(@PublicationEntryDTO newEntry);

    // computation operations (stateless):
    String convertToBibtex(@PublicationEntryDTO entry);
}

Service PublicationManagementQueryFacade {
    // retrieval operations:
    @PublicationArchive dumpPublicationArchive();
    Set<@PublicationEntryDTO>
        lookupPublicationsFromAuthor(String writer);
    String renderAsBibtex(@PaperId paperId);
}
```

This API design achieves command-query segregation at the expense of distributing the two operations related to BibTeX to two different endpoints. Consequently, the two endpoints are coupled from a domain design standpoint (to some extent).

3.7.8 Hints and Pitfalls to Avoid. When deciding to separate commands from queries by introducing the CQRS pattern:

- Replicate data as needed. Decide between strict and eventual consistency consciously.
- Be aware of the implications of the *Backup Availability Consistency (BAC)* theorem [25]. The BAC theorem states that it is not possible to backup and restore across services consistently without degrading availability.
- Acknowledge that read models and event messages sent as Data Transfer Objects (DTOs) over APIs increase the data coupling between clients and providers. If multiple clients use the same DTOs, they might indirectly also be coupled consequently.[8]
- Consider asynchronous, queue-based messaging to update the read model after a change to the write/command model caused by an API command or a backend activity. This integration style supports throttling and is able to guarantee message delivery (depending on the quality-of-service properties chosen for a particular queue).
- Consider applying Event Sourcing as one of several options when segregating commands from queries. An event source stores a series of state changes in chronological order but does not store the resulting final/current state. In such designs, it often makes sense to take snapshots of the current state periodically or upon client request; such snapshots can then be stored separately from the events and provided to clients via additional calls to API operations.

3.7.9 Related Content. This pattern refines *Extract Endpoint* in the context of CQRS. Hence, *Merge Endpoints* reverts it. *Introduce Pagination* and *Add Wish List* might be alternative options to improve query performance. The *Introduce Data Transfer Object* refactoring explains DTO usage.

Information Holder Resources of various types are related patterns that may benefit from command-query segregation. In "Patterns for API Design" [48], queries are represented as Retrieval Operations; commands are State Creation Operations or State Transition Operations.

Michael Ploed provides a comprehensive introduction to CQRS and event sourcing on slideshare. A presentation video by Michael Ploed is available as well. Also see an online article by Udi Dahan for examples and a discussion of pros and cons. The Context Mapper website provides a tutorial, "Event Sourcing and CQRS Modeling in Context Mapper."

3.8 Refactoring: Introduce Version Mediator

also known as: Add Compatibility Gateway, Add Tolerant Proxy, Support Virtual Version

3.8.1 Context and Motivation. An API runs in production. One of its supported versions will be retired soon, but existing clients still use it. One or more breaking changes have been introduced in subsequent, active versions.[9]

[7] unlike State Creation Operation, Retrieval Operation, and State Transition Operation

[8] This cannot be avoided entirely in any Published Language [6] in an API; the coupling still exists but becomes less obvious when commands and queries are separated (as they still work on the same domain concepts). If the two endpoints evolve autonomously (independently of each other, that is), the models will eventually deviate further and further (which to some extent is desired). Over time, this may cause technical debt and hidden dependencies that counter the original motivation of the pattern and the refactoring. If this happens, the inverse *Merge Endpoints* refactoring may be applied.

[9] which should, but cannot always be avoided; sometimes, it is better to break an existing client and cause work for its developers/maintainers rather than pretending that nothing has changed and letting some hard-to-catch bugs creep in

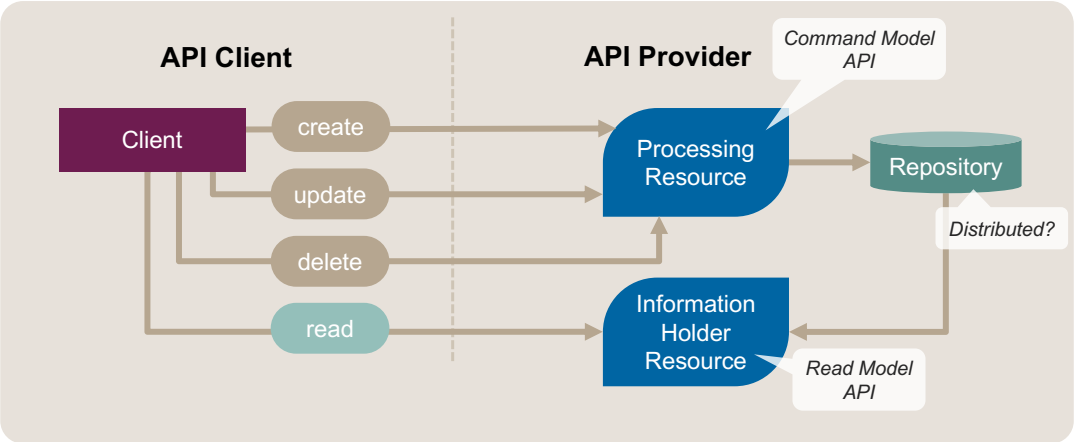

Figure 15: Segregate Commands from Queries: Target Solution Sketch. Commands and Queries in Separate Endpoints

> *As an API client, I want to continue to call a deprecated API for some time, and I expect the provider to support me with a temporary solution for doing so. The behavior of this solution should be identical to those of the API that I have been using so far.*

We will call the API version that will go out of service the "old" API and its successor "new" API.

3.8.2 Stakeholder Concerns (including Quality Attributes and Design Forces).

#flexibility and #evolvability To evolve an API, providers must have the flexibility to refactor, redesign, and adapt an API over time. Ideally, this happens without breaking compatibility, but this is not always realistic.

#developer-experience Breaking changes in APIs and pressure to upgrade may frustrate client developers, especially if they must migrate to a newer version on short notice. For instance, cloud application developers sometimes have to react rather quickly to changes introduced by their public providers.

#maintainability Fewer components and/or code paths that handle deprecated behavior make an API and its implementation easier to maintain for the provider.

See "Interface Evolution Patterns — Balancing Compatibility and Extensibility across Service Life Cycles" [22] for a general discussion of desired qualities, their conflicts, and related trade-offs.

3.8.3 Initial Position Sketch. This architectural refactoring affects the following API elements:

- An endpoint and at least one of its operations (with their roles and responsibilities)
- Representation elements in request and response messages of these operations with their names, roles, and types (including information about value ranges, optionality, and cardinality)
- API clients and the remote communication proxies they use

See Figure 16 for a visualization of the initial position.

3.8.4 Design Smells.

Evolution strategy does not meet client expectations The API provider has decided to commit a short lifetime of an API version only, or has announced to retire one or more active versions soon. This has caused a negative reaction in the API client community (the related refactorings *Tighten Evolution Strategy* and *Relax Evolution Strategy* explain this smell further).

Resistance to change caused by uncertainty One or more breaking change of the API have happened, or the lifetime guarantee has been softened.[10] However, clients are unwilling or unable to migrate to the latest version immediately. They might fear the effort and risk of the migration or they might lack confidence and trust in the new version.

Large and/or partially unknown user base API providers are not in control of their users and lack information about them. The less information and control a provider has, the higher the risk of impacting clients negatively (or losing them) when making breaking changes in upgrades.

3.8.5 Instructions (Steps). Add a new endpoint to the API for the "new" API version. Derive its contract from the "old" one and adjust the "old" endpoint to *mediate* "old" operations and their representation elements to "new" ones with mapping rules.

When establishing the mapping rules for the operations whose "old" and "new" versions are incompatible, start with the request messages:

1. Identify and mark the fields that remain unchanged and do not require a mapping.
2. Define a mapping rule for fields that are *renamed* only and can be mapped one-to-one (pass-through).
3. Find the fields that change their type, and define a mapping rule to implement the type change:
   - If a previously *optional* field becomes mandatory, define a mapping rule that leverages a Content Enricher a.k.a. Data Enricher to define a default value (filler). No mapping

[10] while this should generally be avoided, this is not always possible; the provider might have good reasons to do it

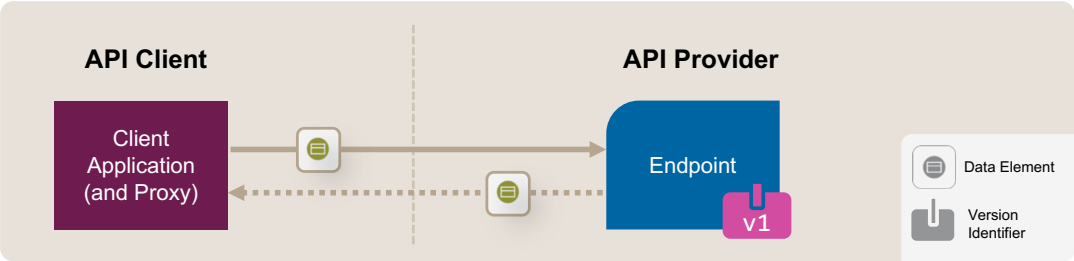

Figure 16: Introduce Version Mediator: Initial Position Sketch. Client communicates with Version 1 of an API Endpoint

action is required for the opposite case (mandatory fields becoming optional).
- Mark the fields that change *cardinality*, for instance from atomic to list/set (and vice versa).
- Do the same for other basic *type changes*, such as from numeric to strings.

4. Find the fields that are *added* in the new version. Add a Content Enricher to support "old" clients. Define a default value or make the newcomer optional (to preserve compatibility).
5. Mark the fields that *disappear* in the new version. Add a Content Filter to mediate requests from "old" clients (and log the fact that some data is no longer processed, having double-checked that this makes sense).
6. Find places where two (or more) "old" fields map to one "new" field. For each such place, define a mapping rule realizing an Aggregation strategy. Define a Splitter rule for the opposite case, one "old" field mapping to two or more "new" fields. These two cases can also occur in combination (which actually can be seen as the default/catch case); a Scatter-Gather rule can handle them.

Continue with the response messages and define similar rules for them, including error cases. If a representation element in a response used to be a single Atomic Parameter but now is set- or list-valued, a Content Filter can select a single element to return. However, the "old" client and the mediator might suffer from information loss through this filtering. Additional patterns such as Context Representation might be able to provide additional (meta-)information in such situations.[11]

Include any custom request and response headers in the mapping for requests and responses.

Secure the mediator endpoint exactly as the updated main endpoint.

3.8.6 Target Solution Sketch (Evolution Outline). A rule-based Compatibility Mediator implements the compatibility mappings, either as plain code or declaratively, and acts as a gateway between "old" clients and the "new" provider. Figure 17 shows this solution.

The mediator should be a transitional, interim solution preserving a good client developer experience and giving clients more time to adopt the "new" API.

3.8.7 Example(s). The fictitious insurance firm Lakeside Mutual could expose the following Customer Core microservice (notation: MDSL):

```
API description LakesideMutual version "v1.0.0"

data type CustomerDTO1 {"name":D<string>}

endpoint type CustomerCoreOriginalContract
exposes
  operation createCustomerMasterData
    expecting payload "customerData": CustomerDTO1
    delivering payload "customerId": ID<int>
  operation readCustomerMasterData
    expecting payload "customerId": ID<int>
    delivering payload "customerData": CustomerDTO1

API provider LakesideMutualAPI
  offers CustomerCoreOriginalContract
  at endpoint location "http://some.original.address"
  via protocol HTTP binding resource Home

API client CustomerRelationshipManagementApplication
  consumes CustomerCoreOriginalContract
  via protocol HTTP
```

Lakeside Mutual could then update its Customer Core interface (for instance, after a merger with another company):

```
API description LakesideMutual version "v2.0.0"

data type CustomerDTO1 {
  "name":D<string>, "zipString":D<string>,
  "toBeSunset":MD<bool>}

data type CustomerDTO2 {
  "firstName":D<string>, "lastName":D<string>,
  "zipCode": D<int>, "newKey":ID<int>}
// not featuring all deviations here (as defined
// in Steps 1 to 6 in "Instructions")

endpoint type CustomerCoreRevisedContract
exposes
  operation createCustomerMasterData
    expecting payload "customerData": CustomerDTO2
    delivering payload "customerId": ID<int>
  operation readCustomerMasterData
    expecting payload "customerId": ID<int>
    delivering payload "customerData": CustomerDTO2

API client CustomerRelationshipManagementApplication
```

[11] Such cases should be avoided if at all possible, for instance, by providing the "old" and the "new" version of the operation in parallel in the "new" API.

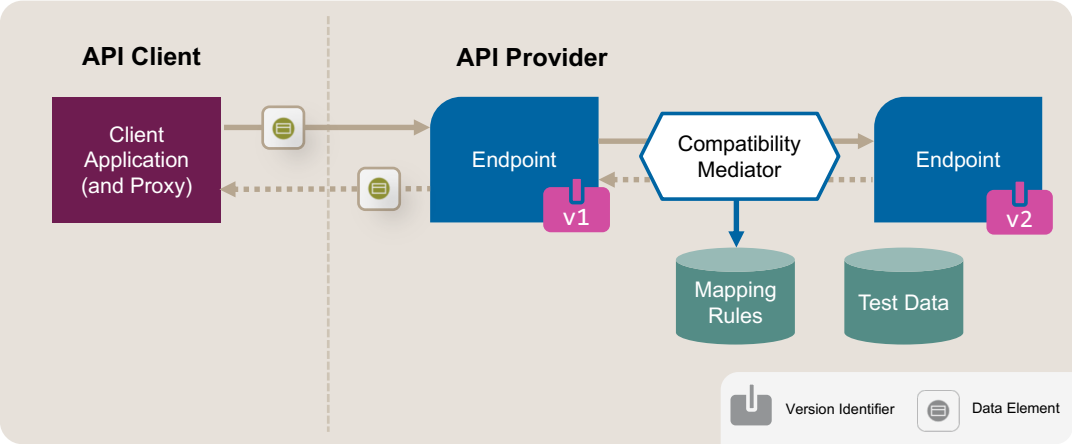

Figure 17: Introduce Version Mediator: Target Solution Sketch. Client communication is mapped to a new endpoint via a Compatibility Mediator.

```
// will no longer work:
consumes CustomerCoreOriginalContract
via protocol HTTP
```

Support for the "old" contract can be modeled as a mediation gateway in MDSL (which does not imply that a visible API gateway is deployed; the mediation can happen in the background):

```
API gateway Version1ToVersion2Mediator
  offers CustomerCoreOriginalContract
    at endpoint location "http://some.new.address"
    via protocol HTTP binding resource Home
  consumes CustomerCoreRevisedContract
    from LakesideMutualAPI
    via protocol HTTP
  mediates from CustomerDTO1 to CustomerDTO2
    element zipString to zipCode
    // not featuring element-level mapping rules here
```

3.8.8 Hints and Pitfalls to Avoid. The user story and the smells motivating this refactoring mention scenarios in which it may be applied; it is also important to know when not to use a particular design.

To decide when not to apply this refactoring, analyze whether the roles of an endpoint, responsibilities of an operation, and/or semantics of a Data Element change. Such changes require more than a rule-based mediator; in such situations, this refactoring is less suited.

As a general rule for any communication party (API client and provider), apply Postel's Law and be liberal when consuming messages and conservative when producing them.

Having decided to apply this refactoring, make sure to:

- Catch and handle mapping errors, both at specification time and at runtime.
- Test all combinations of the "old" versus "new" clients and provider that appear in the refactored system landscape; up to four (two times two) cases might occur. Add test data for all steps/situations from the step descriptions further up that may occur. Include mapping errors in the tests (for instance, set-valued responses that the introduced Content Filters realizing the Version Mediator are not prepared to process).
- Monitor the performance of the Compatibility Mediator (in particular, when it is realized as a mapping rule engine) and end-to-end latency (as the number of request/response messages is doubled).
- Do not prolong the lifetime of the intermediary/the temporary mediation endpoint; for instance, do not place a second gateway in front of the gateway to cope with a future change of a different kind.

A domain-specific language, either embedded in a general-purpose language or explicit, might be an appropriate choice for expressing the mapping rules. Many application integration tools provide such languages (often proprietary). MDSL is a technology-independent contract language that supports mediation rules.

An API Gateway may play the role of a Version Mediator. An example of such gateway usage can be found in the blog post "8 Common API Gateway Request Transformation Policies". LinkedIn also uses an API Gateway in their new Marketing APIs that support request mapping to mediate between API versions.

3.8.9 Related Content. An application of *Tighten Evolution Strategy* may trigger this refactoring. And *Introduce Version Identifier* might have to be applied before this one so that clients can learn about versions and their (in-)compatibilities.

The Enterprise Integration Patterns category Message Transformation provides partial solutions; the patterns Content Enricher and Content Filter are used to realize the Compatibility Mediator (which effectively is a special-purpose Content-Based Router).

For an example of Enterprise Service Bus product capabilities and integration services, refer to Scenario 4/Figure 7 in "Enterprise Service Bus" by Jürgen Kress et al. This technical article, available on a vendor site, positions the pattern, presents usage scenarios, and suggests selection criteria.

In object-oriented programming, the *Adapter* design pattern [13] provides a different view on the interface of a class so that it can be used by clients that cannot work with the original interface easily.

Note that there is also a *Mediator* pattern, whose goal is different (decouple objects from each other).

Also related are:

- Data mapping and Enterprise Application Integration (EAI) tools such as Apache Camel, possibly complemented with a data mapper such as Nomin as featured in "An integration job engine for Apache Camel" as well as expression languages such as the Spring Expression Language (SPeL) or XPath and its JSON pendants
- An API Versioning/Evolution DSL described in "Continuous API Evolution in Heterogeneous Enterprise Software Systems" [20]
- Extract-Transform-Load (ETL) tools in the Data Warehouse and Information Management communities

Smart proxies in service middleware operating on change-aware contracts are emerging [20].

4 REFACTORING EXECUTION AND TOOL SUPPORT

In this section, we discuss how to apply the refactorings from our catalog and how to find suited ones. Tool support is available, also demonstrating their validity.

4.1 For Any Refactoring: Test, Explain, Let Know and Learn (TELL)

When refactoring the API boundaries of a system, no matter whether these are local or remote APIs, the clients of the API must also be considered. Most API refactorings are not finished once the API definition or implementation is adjusted; operations or parameters may need to be deprecated rather than removed immediately. We propose a *Test*, *Explain*, *Let Know*, and *Learn* (TELL) approach to all API refactoring. Having applied a refactoring, always apply the four TELL steps:

- *T*est the updates locally and end-to-end to ensure the API works as expected. Do not just consider functional tests but also non-functional ones, such as performance tests, to ensure that the refactoring did not introduce a performance regression.
- *E*xplain the new design in an Architectural Decision Record (ADR)[12] and have it reviewed as needed. Adjust the API description (including Version Identifier) accordingly. Make sure that your conceptual, platform-independent design, its technology- and platform-specific refinement, the implementation, the tests, and the documentation of all of these artifacts stay current and consistent.
- *L*et API clients and other stakeholders know about the change (even if a change might be syntactically and semantically backward-compatible, the refactored API might offer qualitatively different features now, for example, a Wish List that clients could use to optimize their conversations). Decide whether the API version that has been refactored should continue to be supported for some time. One or more lifecycle management strategies can be applied, for instance, Version Identifier or Two in Production evolution patterns.
- *L*earn about the effectiveness of the refactoring (as well as potentially negative side effects) with logging and other observation instruments.

The TELL steps help to ensure that the refactoring improves the API quality and that the API clients are informed about the changes and can evolve without disruption. See Chapters 3 and 8 of "Patterns for API Design" [48] for more information about options, criteria, and consequences of applying different evolution patterns. The refactorings also outline possible evolutions: e.g., the section "Hints and Pitfalls to Avoid" contains related hints.

4.2 Catalog Navigation

The IRC Website offers two tools that help developers discover applicable refactorings: a Smell Browser (Figure 18) and a Stakeholder Concerns View (Figure 19) to enter the catalog.

4.3 IDE Extensions and MDSL Web Tool

Automated refactoring tools that are part of integrated development tools (IDEs) are commonly used by developers to quickly and safely refactor program code, ensuring that software remains maintainable and well-structured. Originating in the Smalltalk Refactoring Browser [28], automated refactoring tools have been adopted by all major IDEs and for various programming languages, both dynamically typed ones (e.g., Ruby [4]) as well as statically typed ones (e.g., Scala [34]).

The MDSL Tools implement many/most of the refactorings in our catalog as quick fixes, including those presented in this paper. See the documentation on "Transformations Related to Patterns and Refactorings" and the blog post "Refactorings implemented in MDSL Tools" for details.

The refactorings are also available through *MDSL Web Tool*, provided as an open source Web application implemented in JavaScript (frontend) and Java (backend, using Spring Boot). Both IDE and Web versions of the MDSL tools are able to transform API specifications written in MDSL and generate Open API and other interface descriptions from them (see Figure 20).

5 SUMMARY

We motivated the need for API refactoring and introduced 22 interface refactorings collected so far. We presented eight of these refactorings in this paper (*Introduce Data Transfer Object*, *Add Wish List*, *Introduce Pagination*, *Split Operation*, *Merge Operations*, *Rename Representation Element*, *Segregate Commands from Queries*, *Introduce Version Mediator*). To do so, we used a common template (see Appendix B). We also discussed emerging tool support and refactoring usage, supported by four TELL principles: Test, Explain, Let Know and Learn.

Future work concerns more details on existing refactorings and documenting additional ones. We plan to mine software repositories for refactoring applications and to validate the refactorings in practice. Such future research will help us answer questions such as which refactorings are most frequently applied, which refactorings are applied together, and what are typical API design patterns

[12] To learn more about creating ADRs, we recommend the following article: https://medium.com/olzzio/how-to-create-architectural-decision-records-adrs-and-how-not-to-93b5b4b33080

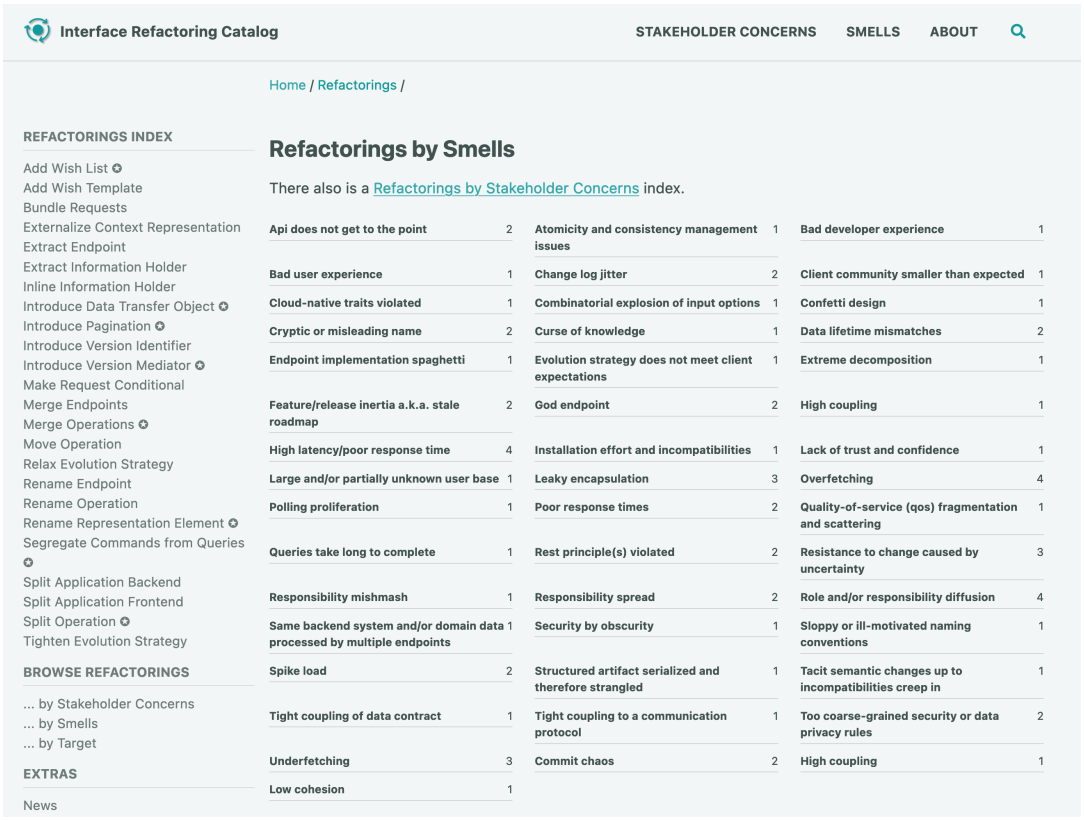

Figure 18: Catalog Navigation 1: Refactorings by Smells

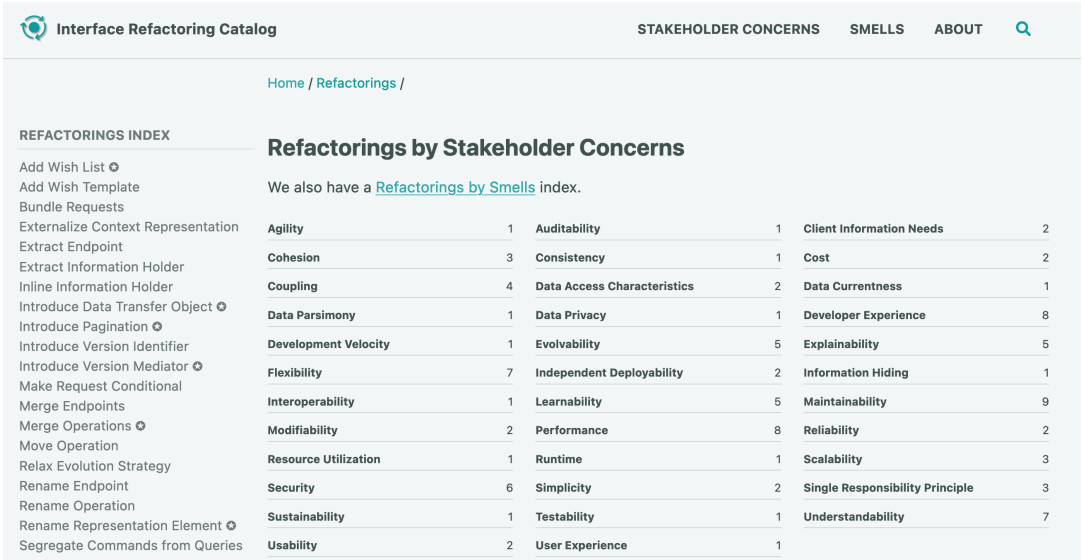

Figure 19: Catalog Navigation 2: Refactorings by Stakeholder Concerns

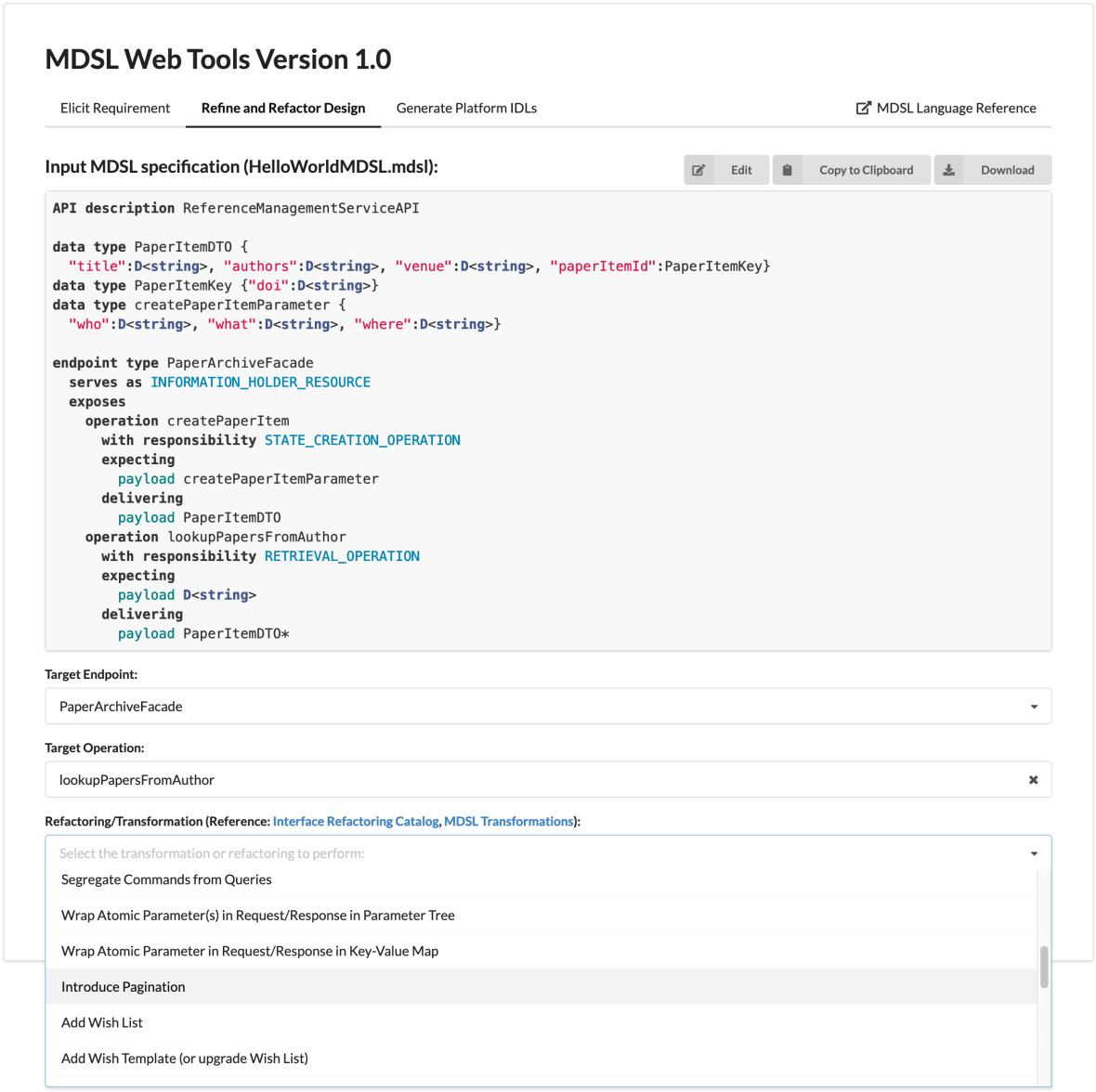

Figure 20: Screenshot of MDSL Web Tool

that developers refactored to or even away from in case of pattern alternatives such as Embedded Entity and Linked Information Holder. Our knowledge engineering backlog comprises candidate refactorings such as Split Application Frontend and Split Application Backend.

Finally, a promising direction for future work is to combine architectural refactoring with green software engineering. Additional information in refactoring catalogs and tools may make software applications and their developers more aware of the direct and indirect resource consumption of current and future application deployments. The impact of changes on the resource consumption can then be taken into account when making architectural decisions that aim at improving the carbon footprint of the application and its deployments.

ACKNOWLEDGMENTS

We would like to thank the contributors to the interface refactoring catalog. We also want to thank the participants of the EuroPLoP 2023 Writers' Workshop Carlos Albuquerque, Filipe F. Correia, Daniel Lübke, Cesare Pautasso, Souhaila Serbout, and Uwe Zdun for their valuable feedback. Andrei Furda has reviewed drafts of selected refactorings. A grant from the Hasler Foundation partially supported the research presented in this paper.

REFERENCES

[1] Subbu Allamaraju. 2010. *RESTful Web Services Cookbook.* O'Reilly.

[2] Scott W. Ambler and Pramodkumar J. Sadalage. 2006. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley.

[3] Tom Benner. 2023. *Naming Things: The Hardest Problem in Software Engineering.* Independently published.

[4] Thomas Corbat, Lukas Felber, and Mirko Stocker. 2007. Refactoring support for the ruby development tools.. In *Software Engineering (Workshops) (LNI, Vol. P-106)*, Wolf-Gideon Bleek, Henning Schwentner, and Heinz Züllighoven (Eds.). GI, 313–315. http://dblp.uni-trier.de/db/conf/se/se2007w.html#CorbatFS07

[5] Trevor Foucher Dustin Boswell. 2011. *The Art of Readable Code.* O'Reilly Media, Inc.

[6] Eric Evans. 2003. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Addison-Wesley.

[7] G. Fairbanks. 2010. *Just Enough Software Architecture: A Risk-driven Approach.* Marshall & Brainerd.

[8] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer. https://doi.org/10.1007/978-3-7091-1568-8

[9] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. https://doi.org/10.17487/RFC7231

[10] International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical, and Electronics Engineers. [n. d.]. *ISO/IEC/IEEE 24765: 2017(E): ISO/IEC/IEEE International Standard - Systems and Software Engineering–Vocabulary.* IEEE.

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

[12] Martin Fowler. 2018. *Refactoring* (2 ed.). Addison-Wesley, Boston, MA.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley.

[14] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. In *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9846)*, Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski (Eds.). Springer, 185–200. https://link.springer.com/chapter/10.1007/978-3-319-44482-6_12

[15] Neil B. Harrison. 2003. Advanced Pattern Writing Patterns for Experienced Pattern Authors. In *Proc. Eighth European Conference on Pattern Languages of Programs (EuroPLoP).* 1–20.

[16] F. Hermans. 2021. *The Programmer's Brain: What every programmer needs to know about cognition.* Manning.

[17] Gregor Hohpe, Ipek Ozkaya, Uwe Zdun, and Olaf Zimmermann. 2016. The Software Architect's Role in the Digital Age. *IEEE Software* 33, 6 (2016), 30–39. https://doi.org/10.1109/MS.2016.137

[18] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley.

[19] Joshua Kerievsky. 2004. *Refactoring to Patterns.* Pearson Higher Education.

[20] Holger Knoche and Wilhelm Hasselbring. 2021. Continuous API Evolution in Heterogenous Enterprise Software Systems. In *18th IEEE International Conference on Software Architecture, ICSA 2021, Stuttgart, Germany, March 22-26, 2021.* IEEE, 58–68. https://doi.org/10.1109/ICSA51549.2021.00014

[21] Arnaud Lauret. 2019. *The Design of Web APIs.* Manning.

[22] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. 2019. Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLop '19).* Association for Computing Machinery, New York, NY, USA, Article 15, 24 pages. https://doi.org/10.1145/3361149.3361164

[23] Gerard Meszaros and Jim Doble. 1997. A Pattern Language for Pattern Writing. *Pattern Languages of Program Design* 3 (1997), 529–574.

[24] Bertrand Meyer. 1997. *Object-Oriented Software Construction (2nd Ed.).* Prentice-Hall, Inc., USA.

[25] Guy Pardon, Cesare Pautasso, and Olaf Zimmermann. 2018. Consistent Disaster Recovery for Microservices: the BAC Theorem. *IEEE Cloud Computing* 5, 1 (12 2018), 49–59. https://doi.org/10.1109/MCC.2018.011791714

[26] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (dec 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[27] Chris Richardson. 2018. *Microservices Patterns.* Manning.

[28] Don Roberts, John Brant, and Ralph E. Johnson. 1997. A Refactoring Tool for Smalltalk. *Theory Pract. Object Syst.* 3 (1997), 253–263.

[29] Mahsa H. Sadi and Eric Yu. 2023. WEBAPIK: a body of structured knowledge on designing web APIs. *Requirements Engineering* (2023). https://doi.org/10.1007/s00766-023-00401-2

[30] Souhaila Serbout, Cesare Pautasso, Uwe Zdun, and Olaf Zimmermann. 2022. From OpenAPI Fragments to API Pattern Primitives and Design Smells. In *26th European Conference on Pattern Languages of Programs* (Graz, Austria) *(EuroPLoP'21).* Association for Computing Machinery, New York, NY, USA, Article 21, 35 pages. https://doi.org/10.1145/3489449.3489998

[31] Apitchaka Singjai, Uwe Zdun, and Olaf Zimmermann. 2021. Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory. In *18th IEEE International Conference On Software Architecture (ICSA 2021).* https://doi.org/10.5281/zenodo.4493865

[32] Apitchaka Singjai, Uwe Zdun, Olaf Zimmermann, Mirko Stocker, and Cesare Pautasso. 2021. Patterns on Deriving APIs and their Endpoints from Domain Models. In *28th Conference on Pattern Languages of Programs (PLoP'21).* ACM, ACM, Virtual.

[33] Michael Stal. 2013. *Agile Software Architecture (Chapter 3 in "Aligning Agile Processes and Software Architectures").* Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-407772-0.00003-4

[34] Mirko Stocker. 2010. *Scala Refactoring.* Master's thesis. HSR Hochschule für Technik Rapperswil, http://eprints.ost.ch/id/eprint/286.

[35] Mirko Stocker and Olaf Zimmermann. 2021. From Code Refactoring to API Refactoring: Agile Service Design and Evolution. In *Service-Oriented Computing*, Johanna Barzen (Ed.). Springer International Publishing, Cham, 174–193.

[36] Mirko Stocker, Olaf Zimmermann, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 2018. Interface Quality Patterns – Communicating and Improving the Quality of Microservices APIs. In *23rd European Conference on Pattern Languages of Programs 2018.* https://doi.org/10.1145/3282308.3282319

[37] Jeffrey Stylos, Benjamin Graf, Daniela K. Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. 2008. A Case Study of API Redesign for Improved Usability. https://doi.org/10.1109/VLHCC.2008.4639083

[38] Jeffrey Stylos and Brad Myers. 2007. Mapping the Space of API Design Decisions. 50–60. https://doi.org/10.1109/VLHCC.2007.44

[39] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for Software Design Smells: Managing Technical Debt* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[40] Joseph W. Yoder and Paulo Merson. 2022. Strangler Patterns. In *Proceedings of the 27th Conference on Pattern Languages of Programs* (Virtual Event) *(PLoP '20).* The Hillside Group, USA, Article 8, 25 pages.

[41] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. 2018. Guiding Architectural Decision Making on Quality Aspects in Microservice APIs. In *Service-Oriented Computing*, Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu (Eds.). Springer International Publishing, Cham, 73–89.

[42] Olaf Zimmermann. 2015. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software* 32, 2 (2015), 26–29. https://doi.org/10.1109/MS.2015.37

[43] Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. 2020. Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. In *Proc. of the European Conference on Pattern Languages of Programs* (Online) *(EuroPLoP '20).*

[44] Olaf Zimmermann, Daniel Pautasso, Cesare Lübke, Uwe Zdun, , and Mirko Stocker. 2020. Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. In *Proc. of the European Conference on Pattern Languages of Programs* (Online) *(EuroPLoP '20).*

[45] Olaf Zimmermann and Mirko Stocker. 2021. *Design Practice Reference - Guides and Templates to Craft Quality Software in Style.* LeanPub. https://leanpub.com/dpr

[46] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. 2020. Introduction to Microservice API Patterns (MAP). In *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019) (OpenAccess Series in Informatics (OASIcs), Vol. 78)*, Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:17. https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4

[47] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns - Crafting and Consuming Message-Based Remote APIs. In *22nd European Conference on Pattern Languages of Programs (EuroPLoP 2017).* 1–36. https://doi.org/10.1145/3147704.3147734

[48] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 2022. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges.* Addison-Wesley Professional.

A API PATTERNS OVERVIEW

The following patterns from Zimmermann et al. [48] are used in this paper. The problem and solutions are reproduced here for reference, the API Patterns website and book have more details.

Pattern Name	Pattern Summary (Problem and Solution)
API Description 	*Problem:* Which knowledge should be shared between an API provider and its clients? How should this knowledge be documented? *Solution:* Create an API Description that defines request and response message structures, error reporting, and other relevant parts of the technical knowledge to be shared between provider and client. In addition to static and structural information, also cover dynamic or behavioral aspects, including invocation sequences, pre- and postconditions, and invariants. Complement the syntactical interface description with quality management policies as well as semantic specifications and organizational information.
Atomic Parameter 	*Problem:* How can simple, unstructured data (such as a number, a string, a Boolean value, or a block of binary data) be exchanged between API client and API provider? *Solution:* Define a single parameter or body element. Pick a basic type from the type system of the chosen message exchange format for it. If justified by receiver-side usage, identify this Atomic Parameter with a name. Document name (if present), type, cardinality, and optionality in the API Description.
Computation Function 	*Problem:* How can a client invoke side-effect-free remote processing on the provider side to have a result calculated from its input? *Solution:* Introduce an API operation `cf` with `cf: in -> out` to the API endpoint, which often is a Processing Resource. Let this Computation Function validate the received request message, perform the desired function cf, and return its result in the response.
Context Representation 	*Problem:* How can API consumers and providers exchange context information without relying on any particular remoting protocols? How can identity information and quality properties in a request be made visible to related subsequent ones in conversations? *Solution:* Combine and group all Metadata Elements that carry the desired information into a custom representation element in request and/or response messages. Do not transport this single Context Representation in protocol headers, but place it in the message payload. Separate global from local context in a conversation by structuring the Context Representation accordingly. Position and mark the consolidated Context Representation element so that it is easy to find and distinguish from other Data Elements.
Data Element 	*Problem:* How can domain/application-level information be exchanged between API clients and API providers without exposing provider-internal data definitions in the API? How can API client and API provider be decoupled from a data management point of view? *Solution:* Define a dedicated vocabulary of Data Elements for request and response messages that wraps and/or maps the relevant parts of the data in the business logic of an API implementation.
Embedded Entity 	*Problem:* How can one avoid sending multiple messages when their receivers require insights about multiple related information elements? *Solution:* For any data relationship that the client wants to follow, embed a Data Element in the request or response message that contains the data of the target end of the relationship. Place this Embedded Entity inside the representation of the source of the relationship.
Id Element 	*Problem:* How can API elements be distinguished from each other at design time and at runtime? When applying domain-driven design, how can elements of the Published Language be identified? *Solution:* Introduce a special type of Data Element, a unique Id Element, to identify API endpoints, operations, and message representation elements that have to be distinguished from each other. Use these Id Elements consistently throughout API Description and implementation.
Information Holder Resource 	*Problem:* How can domain data be exposed in an API, but its implementation still be hidden? How can an API expose data entities so that API clients can access and/or modify these entities concurrently without compromising data integrity and quality? *Solution:* Add an Information Holder Resource endpoint to the API, representing a data-oriented entity. Expose create, read, update, delete, and search operations in this endpoint to access and manipulate this entity. In the API implementation, coordinate calls to these operations to protect the data entity.
Link Element 	*Problem:* How can API endpoints and operations be referenced in request and response message payloads so that they can be called remotely? *Solution:* Include a special type of Id Element, a Link Element, to request or response messages. Let these Link Elements act as human- and machine-readable, network-accessible pointers to other endpoints and operations. Optionally, let additional Metadata Elements annotate and explain the nature of the relationship.

Pattern Name	Pattern Summary (Problem and Solution)
Linked Information Holder 	*Problem:* How can messages be kept small even when an API deals with multiple information elements that reference each other? *Solution:* Add a Link Element to messages that pertain to multiple related information elements. Let this Link Element reference another API endpoint that represents the linked element.
Metadata Element 	*Problem:* How can messages be enriched with additional information so that receivers can interpret the message content correctly, without having to hardcode assumptions about the data semantics? *Solution:* Introduce one or more Metadata Elements to explain and enhance the other representation elements that appear in request and response messages. Populate the values of the Metadata Elements thoroughly and consistently; process them as to steer interoperable, efficient message consumption and processing.
Pagination 	*Problem:* How can an API provider deliver large sequences of structured data without overwhelming clients? *Solution:* Divide large response data sets into manageable and easy-to-transmit chunks (also known as pages). Send one chunk of partial results per response message, and inform the client about the total and/or remaining number of chunks. Provide optional filtering capabilities to allow clients to request a particular selection of results. For extra convenience, include a reference to the next chunk/page from the current one.
Processing Resource 	*Problem:* How can an API provider allow its clients to trigger an action in it? *Solution:* Add a Processing Resource endpoint to the API exposing operations that bundle and wrap application-level activities or commands.
Retrieval Operation 	*Problem:* How can information available from a remote party (the API provider, that is) be retrieved to satisfy an information need of an end user or to allow further client-side processing? *Solution:* Add a read-only operation `ro: (in,S) -> out` to an API endpoint, which often is an Information Holder Resource, to request a result report that contains a machine-readable representation of the requested information. Add search, filter, and formatting capabilities to the operation signature.
Semantic Versioning 	*Problem:* How can stakeholders compare API versions to detect immediately whether they are compatible? *Solution:* Introduce a hierarchical three-number versioning scheme `x.y.z`, which allows API providers to denote different levels of changes in a compound identifier. The three numbers are usually called major, minor, and patch versions.
State Creation Operation 	*Problem:* How can an API provider allow its clients to report that something has happened that the provider needs to know about, for instance, to trigger instant or later processing? *Solution:* Add a State Creation Operation `sco: in -> (out,S')` that has a write-only nature to the API endpoint, which may be a Processing Resource or an Information Holder Resource.
State Transition Operation 	*Problem:* How can a client initiate a processing action that causes the provider-side application state to change? *Solution:* Introduce an operation in an API endpoint that combines client input and current state to trigger a provider-side state change `sto: (in,S) -> (out,S')`. Model the valid state transitions within the endpoint, which may be a Processing Resource or an Information Holder Resource, and check the validity of incoming change requests and business activity requests at runtime.
Two in Production 	*Problem:* How can a provider gradually update an API without breaking existing clients but also without having to maintain a large number of API versions in production? *Solution:* Deploy and support two versions of an API endpoint and its operations that provide variations of the same functionality but do not have to be compatible with each other. Update and decommission the versions in a rolling, overlapping fashion.
Wish List 	*Problem:* How can an API client inform the API provider at runtime about the data it is interested in? *Solution:* As an API client, provide a Wish List in the request that enumerates all desired data elements of the requested resource. As an API provider, deliver only those data elements in the response message that are enumerated in the Wish List ("response shaping").
Wish Template 	*Problem:* How can an API client inform the API provider about nested data that it is interested in? How can such preferences be expressed flexibly and dynamically? *Solution:* Add one or more additional parameters to the request message that mirror the hierarchical structure of the parameters in the corresponding response message. Make these parameters optional or use Boolean as their types so that their values indicate whether or not a parameter should be included.

B REFACTORING TEMPLATE

The template to describe the refactorings is structured as follows:

Refactoring: *Name*

also known as: *Alternative Names*

Context and Motivation. Where and under which circumstances is this refactoring eligible? And why? Are there preconditions for this refactoring? The motivation for the refactoring is stated as a goal-statement in the form of a User Story [45]:

> *As a …, I want to … so that … .*

Stakeholder Concerns (including Quality Attributes and Design Forces). Which non-functional requirements and constraints are impacted by this refactoring?

#quality-attribute This is the explanation of the quality attribute.

Note that we use *#hash-tags* for quality attributes to discern them from *Smell Names* shown below.

Initial Position Sketch. Which API parts or architectural elements have to be changed and are targets of this refactoring (for example, component, endpoint, or message)? Which design problems pertain to this refactoring, and which design options are currently chosen to resolve them?

Design Smells. Smells are "structures in the design that indicate violations of fundamental design principles and negatively impact design quality" [39]. This section identifies smells that indicate a problem with an architecture or API that the refactoring may solve.

Smell Name NN This is the definition of the specific smell.

Instructions (Steps). How can the refactoring be applied and validated? Contains a series of small and simple steps to follow. More complex steps, for instance, TELL, are described elsewhere and may be referenced and included here.

Target Solution Sketch (Evolution Outline). Which design options should be chosen to address the concerns and remove the smells? What does the target solution look like? In particular, what impact on API clients, and what can be done to mitigate it? Do variants of the refactoring exist, e.g., one that preserves backward compatibility?

Example(s). A concrete instance of the refactoring in action, in an API implementation (RESTful HTTP, gRPC) or at the interface definition and specification level (MDSL, UML, OpenAPI).

Hints and Pitfalls to Avoid. Things to watch out for and consider when applying the refactoring. When refactoring to a pattern, we do not repeat all consequences of the pattern application but keep the focus on the most relevant ones in the refactoring context.

Related Content. Points to reverse refactorings, (external) code-level refactorings, etc.

This template is also applicable to other types of refactorings. Our refactorings are licensed under the Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0) license. Contributions to the refactoring catalog are very welcome.