Refactoring API Endpoints, Operations and Messages

Interface Refactoring Catalog Slice Three

Mirko Stocker(✉), Stefan Kapferer, and Olaf Zimmermann

Eastern Switzerland University of Applied Sciences (OST), Oberseestrasse 10, 8640 Rapperswil, Switzerland
{mirko.stocker,stefan.kapferer,olaf.zimmermann}@ost.ch

Abstract. Application Programming Interfaces (APIs) enable communication in distributed systems of all kinds. As providers and clients evolve, APIs must adapt to changing requirements, scalability challenges, and operational insights. A lack of a structured approach to API evolution can lead to inconsistent interfaces, bloated endpoints and a poor developer experience. Refactoring has long been a staple in agile software development, focusing primarily on improving internal code quality. However, as distributed systems and remote APIs have become central to modern software architectures, the need for systematic API refactoring has grown. In this paper, we present a third slice of our Interface Refactoring Catalog (IRC), which now has 25 refactorings. Fifteen of these were published in our previous work; five more refactorings appear in this paper. API designers and maintainers that seek to improve certain design time and runtime API qualities form the primary target audience for IRC; API client developers wanting to use APIs effectively and efficiently and API provider developers wanting to minimize the impact of changes are targeted as well. Finally, API product managers concerned about API quality also belong to our target audience. The provided examples use the Microservice Domain-Specific Language (MDSL) and the Context Mapper Language (CML) to illustrate the refactorings in a technology-agnostic way.

Keywords: application programming interface · cloud computing · design patterns · enterprise application integration · interface definition languages · refactoring · software

1 Introduction

This work is about (re-)designing remote Application Programming Interfaces (APIs); many long-living such APIs exist in practice (e.g., public Web APIs based on HTTP and JSON). Requirements and technology change throughout API lifecycles; design adjustments are often needed. Code refactoring is a popular agile practice whose usage context can be extended. In our previous work we have shown that the concept of refactoring is also suited to share knowledge about

T. Plummer et al. (Eds.): EuroPLoP 2025, LNCS 16493, pp. 28–72, 2026.
https://doi.org/10.1007/978-3-032-19154-0_3

design change tactics (with some adjustments in the definition and the template structure) [29]. Note that we use the terms "API refactoring" and "interface refactoring" interchangeably and assume physical distribution requiring the use of remote APIs.

The content and contribution of this paper are five remote interface refactorings continuing the coverage of our Interface Refactoring Catalog (IRC) [30,31]: *Add Wish Template*, *Bundle Requests*, *Move Operation*, *Merge Endpoints*, *Rename Endpoint*. Two of these refactorings can be used to apply two "Patterns of API Design", Wish Template and Request Bundle [41]. The other three refactorings deal with structure and naming (*Move Operation*, *Merge Endpoints*, *Rename Endpoint*). See Fig. 1 for an overview of the refactorings, including their targeted API elements.

A sibling paper "API Refactoring and Reengineering: Distribution Patterns and Evolution Strategies" [36] presents three architectural refactorings that deal with client-server-cuts (i.e., distribution of logical components and layers onto physical tiers): *Distribute Application Frontend*, *Split Application Backend Logic*, and *Split Application Backend Persistence* and two refactorings with an organizational nature, dealing with API lifecycle management: *Tighten Evolution Strategy*, and *Relax Evolution Strategy*.

The remainder of the paper is structured in the following way. Section 2 introduces background information. Section 3 and its subsections feature the third slice of IRC in depth, following a common template. Section 4 discusses pros and cons of pattern-based API refactoring and covers related work in different fields; each refactoring also discusses related work individually in the "Related Content" subsection. Section 5 summarizes the paper.

2 Background Information

At a high level, API communication involves two participants: an API *provider* and one or more API *clients*. The provider exposes APIs through API *endpoints*, each with a unique address like a URL. Clients access these endpoints to use API *operations*. To invoke an operation, clients send a *request message* to the endpoint, which may respond with a corresponding *response message*. API operations are often mapped to code-level methods, functions or procedures, with request and response messages mapped to parameters and return values respectively. API messages have structured representations, usually consisting of multiple elements. An API contract governs the communication. These italicised abstractions and concepts, shown in Fig. 1, form a domain model for API design and development [41]. They can all be targets of refactorings.

An API can be implemented directly in executable code that defines the API contract. Alternatively, the API contract can first be defined in a protocol-specific interface definition language such as OpenAPI Specification for HTTP or in a technology-independent language such as Microservice Domain-Specific Language[1] (MDSL). MDSL defines endpoints, operations, parameters, and data

[1] https://microservice-api-patterns.github.io/MDSL-Specification.

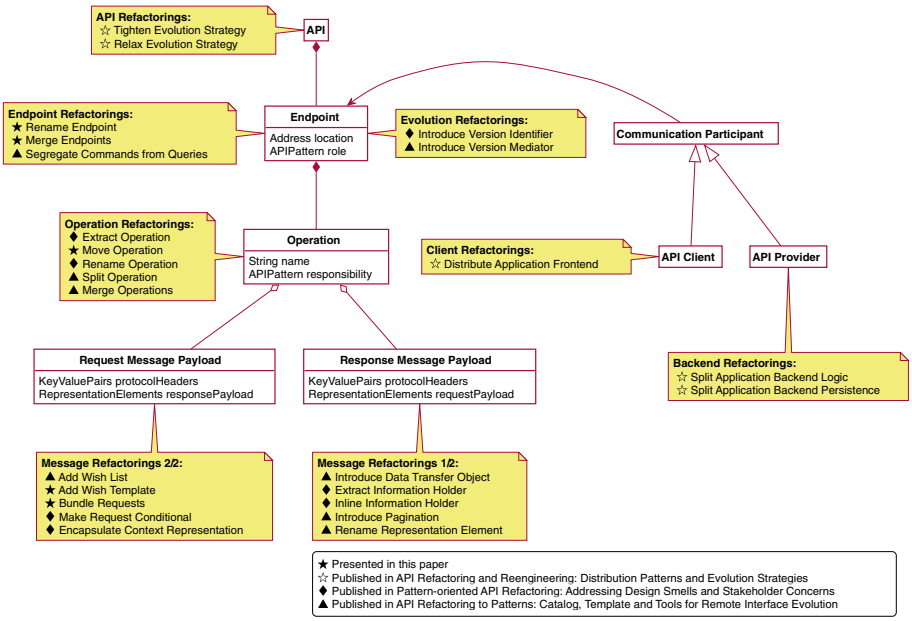

Fig. 1. Refactorings by targeted API element (as defined in API domain model [41])

types using abstractions that align with our API domain model. Protocol-specific specifications such as OpenAPI, Protocol Buffers or GraphQL can be generated from MDSL, as well as testable Java application prototypes. The following listing shows a sample MDSL specification of a `HelloWorldEndpoint` that exposes a `sayHello` operation (source: MDSL Website[2]):

```
API description HelloMDSLWorld

data type SampleDTO {ID, D<string>}

endpoint type HelloWorldEndpoint
exposes
  operation sayHello
    expecting payload "in": D<int>
    delivering payload SampleDTO

API provider HelloWorldAPIProvider
  offers HelloWorldEndpoint
  at endpoint location "http://localhost:8000"
  via protocol HTTP
    binding resource HomeResource at "/"
      operation sayHello to POST
```

[2] https://microservice-api-patterns.github.io/MDSL-Specification.

```
API client HelloWorldAPIClient
  consumes HelloWorldEndpoint
  from HelloWorldAPIProvider
  via protocol HTTP
```

This example also shows provider bindings and a client working with this API contract.

"Patterns for API Design" [41] provide guidance on how to design the different API elements; the book also contains the MDSL language reference. The patterns cover endpoint types and operation responsibilities, request and response message representation, refining the message design for quality, evolving APIs and documenting and communicating the overarching API contract between provider and client; Appendix A summarizes the API design patterns referenced in this paper. MDSL supports these patterns in its domain-specific language. For example, the above endpoint could be extended by an explicit endpoint role using `serves as`:

```
endpoint type HelloWorldEndpoint
  serves as PROCESSING_RESOURCE
```

The Processing Resource indicates that the endpoint exposes operations that bundle and wrap application-level activities or commands. Using patterns, the API description can be further enriched. Not only does this have documentary value, but MDSL tools can use this information for semantic validation, to apply transformations, in code generators or for refactoring. [24] provides an introduction to MDSL.

Our refactorings, some of which are supported by MDSL tools, focus on the provider side, specifically on endpoints and the operations they expose and the messages that are exchanged. *Refactoring* refers to the practice of improving a software system without changing its external/observable behavior. The purpose of refactoring can also be the alignment of the software with a design pattern [17] and to remove software design smells [32], which signal potential issues in the design or implementation of an API.

The target audience for our refactorings are all roles that develop, design, and maintain software systems exposing a remote API. We do not target any specific domain or technology; every system exposing an API is a potential candidate. Therefore, the refactoring activities are described in a technology-neutral way for the most part; their examples and implementation hints mainly stem from the design and evolution of HTTP resource APIs. All refactorings come with concrete examples, typically written in Java, using the Spring Framework; others use MDSL or the Context Mapper Language (CML). CML is a modeling language based on Domain-Driven Design, provided by the Context Mapper[3] project [15].

[3] https://contextmapper.org/.

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, the addressed design smells, and the affected stakeholder concerns in the form of quality attributes.

Refactoring	Design Smells Addressed	Stakeholder Concerns
Add Wish Template (Sect. 3.1) As an API client developer, I want to express the desired response structure and content with a mock object so that I can be precise and selective when expressing my wishes about structured response data that I am interested in.	Overfetching, structured artifact serialized and therefore mangled, underfetching	client information needs, maintainability, flexibility, performance
Bundle Requests (Sect. 3.2) As an API provider, I want to allow my clients to reduce the number of requests and responses to save bandwidth and network capacity and to avoid unnecessary message processing. To do so, I want to group multiple domain-level requests in a single API call.	Atomicity and consistency management issues, high latency/poor response time	developer-experience, performance
Move Operation (Sect. 3.3) As an API provider, I want to focus and consolidate the operations of an existing endpoint on a single role and purpose so that API clients serving a particular stakeholder group understand the API design intuitively and the provider has a single reason to change that endpoint. To achieve this goal, I want to move one or more operations to a different, already existing endpoint.	Data lifetime mismatches, endpoint implementation spaghetti, god endpoint, role and/or responsibility diffusion, too coarse-grained security or data privacy rules	maintainability, scalability and reliability, single responsibility principle and independent-deployability, understandability
Merge Endpoints (Sect. 3.4) As an API provider, I want to bring the operations from multiple endpoints together and consolidate them in a single endpoint so that they can be deployed, scaled, and evolved jointly.	API does not get to the POINT, cloud-native traits violated, endpoint implementation spaghetti, extreme decomposition, responsibility spread, synonyms for single concept	cohesion, coupling, understandability
Rename Endpoint (Sect. 3.5) As an API developer, I want to express the domain concepts exposed by an endpoint and the architectural role that the endpoint plays as accurately as possible in the published language and the API contract.	Cryptic, misleading or disrespectful name, REST principle(s) violated	learnability, understandability

3 API Endpoint, Operation and Message Refactorings

In this section, we present five Interface Refactorings (IRs) for API endpoints, operations, and messages not published previously:

- *Add Wish Template* and *Bundle Requests* help with the application of two API design patterns, Wish Template and Request Bundle [41].
- Two IRs operate on the API contract structure, adapting classical code refactorings for the API design domain [9]: *Move Operation* and *Merge Endpoints*.
- *Rename Endpoint* deals with naming and identifying API elements.

Table 1 lists the refactorings featured in this paper, along with the design smells and the stakeholder concerns they address. Note that more refactorings related to naming exist in the catalog, which were presented in our previous work [30,31].

All refactorings presented in this section (and throughout our IR Catalog) start from a *context and motivation* and introduce several *stakeholder concerns*, including quality attributes and design forces. An *initial position sketch* shows which API parts or architectural elements are targets of the refactoring. In addition to the stakeholder concerns, each refactoring also lists *smells* that indicate problems with an existing solution (smells are "structures in the design that indicate violations of fundamental design principles and negatively impact design quality" [32]). Starting from the initial position, a series of *instructions* transforms the design into a *target solution sketch* that details how the refactoring can be applied and validated. Each refactoring comes with a concrete *example* that shows the refactoring in action. A discussion of *hints and pitfalls to avoid* follows. The coverage of each refactoring closes with a subsection about *related content*.[4]

Our layout conventions are as follows: Refactoring names like *Add Wish Template* are set in *Italics*. Pattern names, for example, Data Elements, appear in Small Caps (see Appendix A for an overview of API design patterns). We use #hash-tags for quality attributes, e.g., #performance, to discern them from smell names such as ***God endpoint***. When elements from code examples are referenced in the text, their names are set in a `monospaced` font.

3.1 Refactoring: Add Wish Template

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 the API provides.

[4] On the website supporting this paper, it is possible to navigate the refactorings by stakeholder concerns https://interface-refactoring.github.io/refactorings/by-stakeholder-concerns/ and by smells https://interface-refactoring.github.io/refactorings/by-smells-drivers/.

As an API client developer, I want to express the desired response structure and content with a mock object or a descriptive specification so that I can be selective and precise when expressing my wishes about the response data that I am interested in.

Stakeholder Concerns

#**client-information-needs** As motivated in *Add Wish List* [30, pages 7–10], APIs are often used by multiple clients with differing information needs, which makes it difficult for providers to offer one-size-fits-all solutions. Some clients may require overviews of information elements, for instance lists of identifiers; others may require a subset or all of the data fields (attributes) of these elements.

#**maintainability,** #**flexibility** Client information needs change over time, but an API and its implementation should not have to be updated every time a client does. Maintaining multiple variants of the same operation for different clients is possible, but increases maintenance cost and coordination effort, which in turn can limit the flexibility of the API provider.

#**performance** Response time and throughput are qualities that concern both API clients serving end users and API providers. Preparing, transporting, and processing data but not using it on the client side wastes resources and ought to be avoided when sustainability and "Datensparsamkeit" are required.

Initial Position Sketch. A client might request and receive nested and or flat response data via one or more operation calls, as shown in Fig. 2.

Any operation with response data structures that clients want to customize is a potential target. A Wish List might have been introduced to let clients explicitly mark the response data they are interested in (to reduce response message size); however, this list is considered insufficient and/or cumbersome to use. The Wish Template pattern extends Wish List by allowing clients to define a structured, possibly nested template for the data they want. This *Add Wish Template* refactoring can also be applied to improve performance if a Wish List has not been used yet, as shown in Fig. 2. Both patterns aim to give clients fine-grained control over API responses.

Design Smells

Overfetching Clients may throw away large parts of the received data because the API design follows a one-size-fits-all approach and includes all data that any present or future client might be interested in. Another phenomenon is "sell what is on the truck": implementation data is exposed just because it is there, without any client-side use case.

Structured artifact serialized and therefore mangled Some nested data representation is serialized into a custom string format because the API

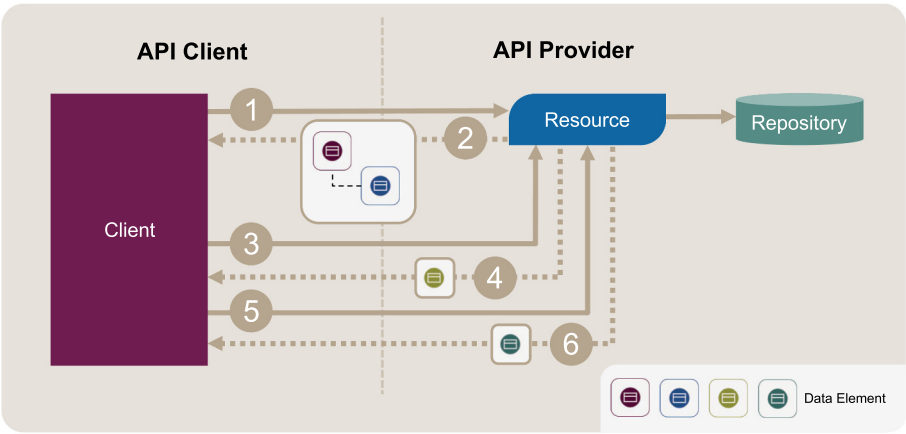

Fig. 2. Add Wish Template: Initial Position Sketch. The API provider responds to a request from a client (1) with a message (2) that contains Data Elements. The client then issues follow-up requests (3 and 5) to retrieve more Data Elements (4 and 6).

response message contract is not expressive enough to accommodate a complex data structure. This results in a string containing a lot of information, but is hard to parse and causes surprises in testing and operations. This smell often materializes during development. For instance, a remote API that is implemented as an object-oriented program might work with objects that contain other objects. These nested object clusters (e.g., aggregates with entities and value object in domain-driven design) often have to be mapped to and from text notations such as JSON. This process is also called serialization and deserialization. When the entire nested object cluster is squeezed into a plain string, the reserved characters of the notations (for instance, curly braces `{}` and double quotes `"` in JSON) have to be escaped during serialization, which can be tedious.

Underfetching Clients may have to call multiple API operations to obtain all required data because the interfaces of these operations do not offer any way to define the targeted representation elements (thus publishing parts or all of the entities of a domain model and their attributes).

Instructions. Introducing a Wish Template requires careful consideration, implementation, and enforcement:

1. Review the structure of the response message of the operation to define the scope of the wishes to be expressed; for instance, all elements must be optional. If necessary, wrap the structure in a Data Transfer Object (DTO) by applying the *Introduce Data Transfer Object* [30, pages 4–7] refactoring.

If the request message already contains a Wish List that is supposed to be upgraded and reified,[5] review its syntax and semantics.
2. Copy the schema definition of the eligible part of the response message to the response message schema to create the *mock DTO* template. Define which values mark inclusion and exclusion, respectively (for example, `"in"` and `"out"` for strings, `1` and `0` for integers, and lists with one or zero entries).
3. Optionally, change the data type of all atomic elements in the structure (such as strings and integers) to boolean. In this case, `true` marks inclusion and `false` marks exclusion. Apply these optional changes to the entire nested/hierarchical data structure recursively: in lists, turn all list elements into booleans; in Parameter Trees, process all child elements recursively.
4. Replace an existing flat, list-oriented Wish List (if any) with the mock DTO serving as Wish Template. If no such list already exists, add the Wish Template to the request message.
5. As for most, if not all, refactorings, apply the TELL steps that we introduced in Stocker and Zimmermann [30] according to the smell description in *Add Wish List* [30, pages 7–10]. Testing edge cases and combinations of input data is particularly important here. For instance, test empty wishes and a representative set of tree populations such as entire nested structure requested, only top-level information requested, and only leaves requested.

The revised API design is not necessarily backward-compatible. Making the template parameter optional helps preserve backward compatibility.

Target Solution Sketch (Evolution Outline). The hierarchical structure of the Wish Template matches the structure of the response data. It is shown in Fig. 3. A variant that boolean to indicate the desired response values is sketched in a UML class diagram in Fig. 4.

Example(s). To demonstrate the Wish Template in an API contract, we will start with an API operation that uses a simple Wish List. A Wish List for a customer response payload might look as the `"desiredElements":MD<string>*` in the request messages of `getCustomerMasterData` in the following example. Note that all response elements are optional, which is indicated by the question marks `?` (notation: MDSL):

```
operation getCustomerMasterData
 with responsibility RETRIEVAL_OPERATION
  expecting payload {
    "queryParameters",
    <<Wish_List>> "desiredElements":MD<string>*
  }
  delivering payload {
    "name":D<string>?,
```

[5] https://en.wikipedia.org/wiki/Reification_(computer_science).

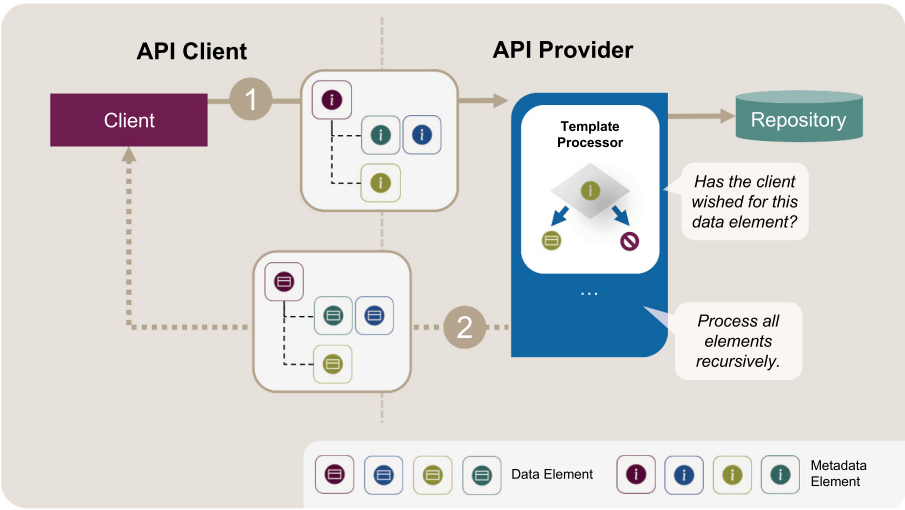

Fig. 3. Add Wish Template: Target Solution Sketch. In its request (1), the client sends a Wish Template. The provider uses a template processor to evaluate the wish Metadata Elements to decide whether some Data Element should be included in the response message (2).

```
    "address": {
        "street":D<string>?,
        "zip":D<int>?,
        "city":D<string>}?
  }
```

Listing fields or attributes of provider-side resources (in the broadest sense of the word) works fine and is used in many APIs, both public and community- or solution-internal ones. This works ok as long as the field names are unique and the resource structure does not cause complex navigation in nested, hierarchical name spaces. A downside of this approach is that it does not catch typos and other mistakes in field/attribute names (at least not on the client side).

Type safety and fine-tuning of the wishes can be achieved in the example by upgrading the flat string list to a `mockCustomer` Wish Template that mirrors the customer structure appearing in the response:

```
operation getCustomerMasterData
 with responsibility RETRIEVAL_OPERATION
  expecting payload {
    "queryParameters",
      <<Wish_Template>> "mockCustomer": {
        "mockName":D<string>?,
        "mockAddress":{"mockStreet":D<string>}?,
        "mockZip":D<int>?, "mockCity":D<string>}?}
  delivering payload {
```

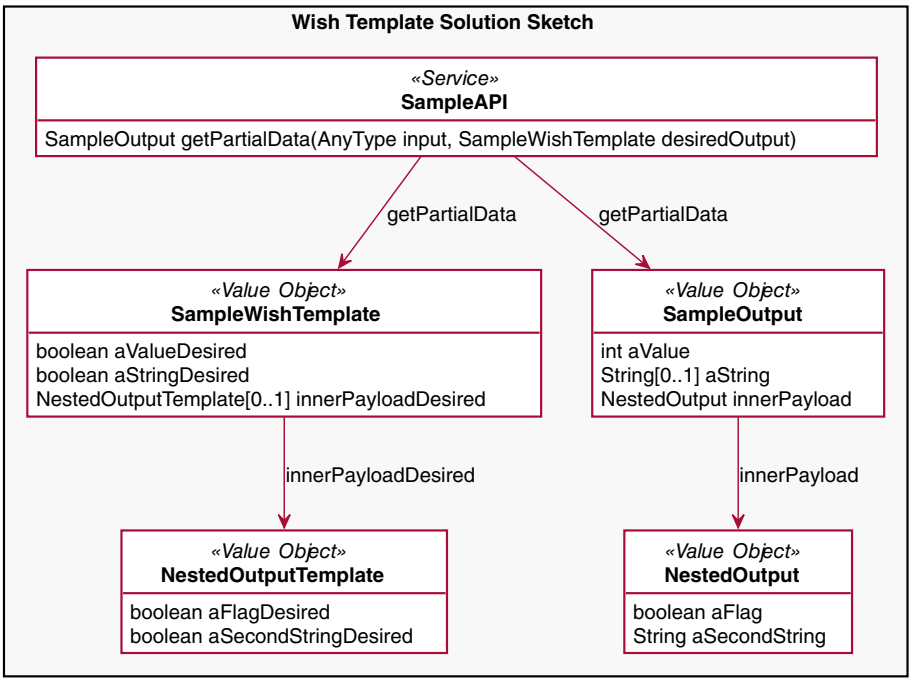

Fig. 4. Variant of Wish Template using booleans to indicate desired values. The template mirrors the output DTO on two nesting levels.

```
    "name":D<string>?,
    "address": {
        "street":D<string>?,
        "zip":D<int>?, "city":D<string>}?
}
```

If such copy of the response structure is used, marker values for the basic types have to be defined in the API contract (for instance, `-1` for `int` values and `""` empty strings and lists). Alternatively, all basic types can be made optional or changed to boolean:

```
<<Wish_Template>> "inclusionMarkers": {
    "includeName":D<bool>?,
    "includeAddress":{
        "includeStreet":D<bool>?,
        "includeCity":D<bool>
    }?
}
```

The API Patterns website[6] provides another example of a Wish Template.

[6] https://api-patterns.org/patterns/quality/dataTransferParsimony/WishTemplate#sec:WishTemplate:Example.

A large integration interface to core banking systems that applied the pattern is described in Zimmermann et al. [37].

Hints and Pitfalls to Avoid. The design of a Wish Template requires more effort than that of a plain Wish List:

- Explicitly define a schema for the template and the corresponding response message elements. Use the same level of detail as for the elements in the response messages themselves, e.g. `NULL` values, optionality, etc.
- Have the template design be reviewed by different client developers before implementing it. Following a variation of the 80-20 rule,[7] also known as the Pareto Principle, 80% of the required features (here: Wish Template design elements) often can be elicited from, and tested by, 20% of the client developers.
- Focus on cardinalities and n to m relations in nested/linked data when designing, implementing, and testing the template. Complex requests with deep nesting can put a strain on your infrastructure. For instance, is it possible to include entire subtrees when specifying inclusion of nodes that are neither roots nor leaves? In a study on the "Semantics and Complexity of GraphQL" conducted by Hartig and Pérez [11], the performance of the GitHub GraphQL API was examined, and an "exponential increase in result sizes" was observed as the query level depth was increased.
- Avoid the ***god endpoint*** API design smell: having one endpoint that does everything. See the *Move Operation* refactoring for an explanation of this smell.
- Be pragmatic and driven by client/user needs in the detailed design. Resist the temptation to optimize the wish declaration syntax just because it is technically challenging and fun to experiment with; focus on the business and domain logic using it instead. In other words, do not become a middleware provider/vendor but use existing languages and engines such as GraphQL for advanced scenarios.
- Do not forget to update security policies so that clients cannot suddenly access more data than they should.

Keep an eye on server side performance. The Wish Template can lead to many combinations of wishes, which can result in a combinatorial explosion of the data that has to be processed and sent to the client. This can lead to performance problems on the server side, especially if the template is not designed carefully.

Developer experience (DX) can suffer if the Wish Template is too complex or not well documented. Clients may struggle to understand how to use it effectively, leading to frustration and potential misuse. Further discussion of the benefits and liabilities of the Wish Template can be found in Zimmermann et al. [41].

Wish Lists and Wish Templates may be applied to several operations called in sequence. When improving API performance and rightsizing one or

[7] https://en.wikipedia.org/wiki/Pareto_principle.

more messages, also consider applying API design patterns such as Embedded Entity and Linked Information Holder.

Related Content. An application of *Add Wish List* [30, pages 7–10] may provide the starting point for this refactoring. A Wish Template can also be introduced if no Wish List existed before; the steps dealing with the existing list are simply skipped in this case.

An alternative to introducing a Wish Template is to simply offer API clients two or more distinct operations. This simpler solution can be sufficient if the number of clients is small and their information needs are well understood.

GraphQL engines can be seen as a tool- and middleware-supported application of this refactoring, providing a single endpoint for flexible queries and mutations. The MDSL Tools[8] contain an implementation of this refactoring.

3.2 Refactoring: Bundle Requests

Context and Motivation. API clients make many small requests (in terms of request message size) in quick succession, and individual responses are returned for one or more of these requests. These high-velocity request-response sequences have a negative impact on scalability and throughput.

> *As an API provider, I want to reduce the number of requests and responses that API clients have to send and receive so that bandwidth and network capacity are saved and unnecessary message processing is avoided. To do so, I want to allow clients to group multiple domain-level requests in a single, structured API call.*

Stakeholder Concerns

#**developer-experience** Bulk/batch uploads might be prepared by iterating through a data set and then calling API operations multiple times, once for each entry in the set. It might be easier for client developers to prepare a single technical request message (for instance, a JSON array) that contains several business-, domain-, or application-level requests (each represented as a JSON object). Such design frees the clients from having to manage the state (running, succeeded, failed) of all individual requests.

#**performance** Transferring many small requests may stress network and communication endpoints preparing requests and consuming them (for instance, on the HTTP or the TCP/IP level).

Initial Position Sketch. The API client currently sends multiple requests to a provider. Figure 5 sketches this initial position.

[8] https://microservice-api-patterns.github.io/MDSL-Specification/tools.

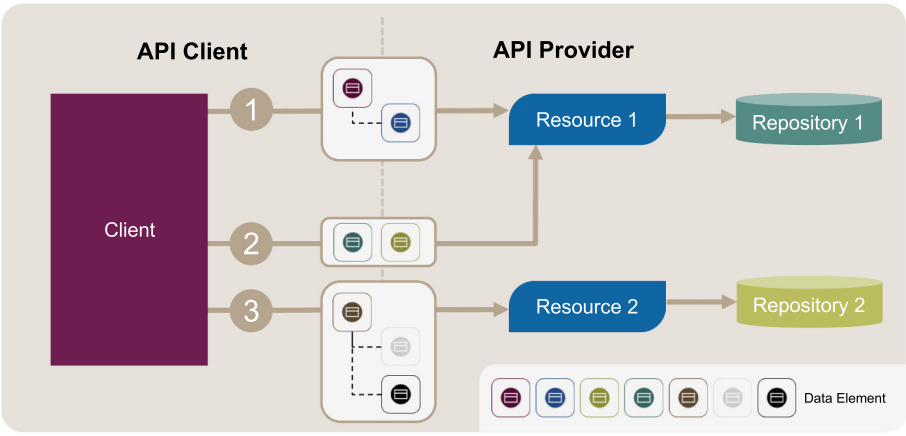

Fig. 5. Bundle Requests: Initial Position Sketch. A client sends three requests to the API provider. Each request comprises several Data Elements and can have its own structure. Note that the responses are not shown for the sake of clarity.

The targeted design elements are one or more API operations and the request and response messages of these operations.

Design Smells

Atomicity and consistency management issues Some clients might want to make sure that either all or none of the requests and corresponding responses complete successfully. This can be much harder if they arrive one by one; most remote APIs do not offer global system transactions (for instance, distributed two-phase commits) – for good reasons: operations overhead, testing effort, and technical risk over time. "Software Architecture: The Hard Parts" covers the consistency design concern, its relationship with other concerns (communication, coordination) as well as related tradeoffs [8].

High latency/poor response time Responses take a long time to be returned to clients because many small requests cause high workload for the communications infrastructure and protocol endpoint(s) on the provider side.

Instructions. The refactoring can be applied to an existing operation. It causes a breaking change:

1. Apply the Request Bundle Pattern to an existing operation: "Define a Request Bundle as a data container that assembles multiple independent requests in a single request message. Add metadata such as identifiers of individual requests (bundle elements) and a bundle element counter." (source: "Patterns for API Design" [41])

2. Adjust the API Description, tests, and all other supporting/supplemental artifacts accordingly.

Alternatively, it can be offered as/in a new operation in the same endpoint:

1. Copy an existing operation.
2. Apply the Request Bundle Pattern to the new operation, as already described above.
3. Document and test the new operation.

Optionally, the bundles can be annotated with additional Metadata Elements. A bundle-level Error Report may be included in the response.

Special emphasis should be put on the monitoring to assess the positive or negative effect of the refactoring. Average response times and message sizes should be logged, and the performance of the initial API design be compared with that of the refactored one.

Target Solution Sketch (Evolution Outline). After the refactoring, both the request message and, optionally, the response message of the refactored operation contain collections of individual requests. See Fig. 6 for this solution sketch.

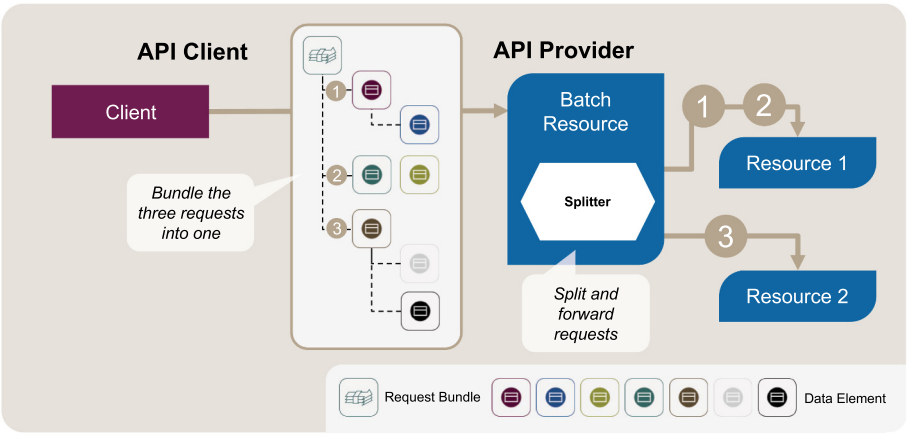

Fig. 6. Bundle Requests: Target Solution Sketch. The client bundles the three individual requests into a Request Bundle message. The provider uses a Splitter component to disassemble the bundle and forward the requests (1, 2, 3) to their final destination. Note that only the request bundling is shown in the figure; the corresponding responses are left out. The repositories supporting the resources are not shown either, but continue to exist.

A variant of the Request Bundle pattern is shown in the HTTP resource Customer Information Holder[9] in Lakeside Mutual, a sample microservice application we built to demonstrate API design and patterns.

Example(s). In the following example, a Request Bundle is added to a Customer Relationship Management endpoint (notation: MDSL):

```
API description RequestBundleExample

data type YesOrNo D<bool>
data type QueryExpression D<string> // detailed query syntax not shown
data type CustomerDTO {"name":D<string>, "contacts":D<string>*}

endpoint type CustomerRelationshipManagement
 serves as MASTER_DATA_HOLDER
exposes
  operation createCustomer
   with responsibility STATE_CREATION_OPERATION
    expecting payload CustomerDTO
    delivering payload "id":ID<int>

  operation updateCustomer
    expecting payload CustomerDTO
    delivering payload "ok":YesOrNo

  operation lookupCustomers
   with responsibility RETRIEVAL_OPERATION
    expecting payload "criteria":QueryExpression
    delivering payload "results":CustomerDTO*
```

Note that the response of the new operation `sendBulkRequest` contains a tree of objects (just like the request):

```
API description RequestBundleExample

data type YesOrNo D<bool>
data type QueryExpression D<string> // detailed query syntax not shown
data type CustomerDTO {"name":D<string>, "contacts":D<string>*}

data type CustomerOperation {
     "createCustomer":CustomerDTO
    |"updateCustomer":CustomerDTO
    |"lookupCustomers":QueryExpression
}
```

[9] https://github.com/Microservice-API-Patterns/LakesideMutual/blob/master/customer-core/src/main/java/com/lakesidemutual/customercore/interfaces/CustomerInformationHolder.java.

```
data type CustomerOperationResponse {
     "id":ID<int>
    |"ok":YesOrNo
    |"results":CustomerDTO*
}

endpoint type CustomerRelationshipManagement
 serves as MASTER_DATA_HOLDER
exposes
  operation sendBulkRequest
    expecting payload <<Request_Bundle>>
      "operations":CustomerOperation+
    delivering payload
      "results":CustomerOperationResponse+
```

The MDSL Tools[10] prototype contains an implementation of this refactoring.

Hints and Pitfalls to Avoid. Consider the following:

- Apply the Splitter[11] pattern from "Enterprise Integration Patterns" to unbundle the incoming request message.
- Leverage parallel programming concepts in the provider implementation. An advanced option to do so is the Map-Reduce[12] pattern.
- Decide whether the responses should also be bundled (see solution and discussion of the consequences of the Request Bundle pattern for detailed design information).
- Measure and compare: Do the performance gains justify the extra effort and complexity? Be willing to undo if the new design turns out to be difficult to teach and maintain and the performance gains do not outweigh these negative consequences.
- When implemented by a new endpoint, make sure that this endpoint is covered by the same security measures as the original endpoint.
- Do not prescribe a minimum bundle size, but allow clients to decide how many requests they want to bundle together. Otherwise, they might never be able to send their requests. Client developers can prepare bundles incrementally, for example, by accumulating requests while processing other tasks. Common scenarios include processing files in batches (such as monthly sales figures in Backend Integration) or collecting user input selections (like checked boxes in a UI form) before sending them as a bundle.

Carefully think about error handling when applying the Request Bundle pattern:

[10] https://microservice-api-patterns.github.io/MDSL-Specification/soad.
[11] https://www.enterpriseintegrationpatterns.com/patterns/messaging/Sequencer.html.
[12] https://www.cloudcomputingpatterns.org/map_reduce/.

- If one of the requests in the bundle fails, how should the provider respond? Should an error be returned for the entire bundle or just for the failed request? Consider using an Error Report to communicate errors in a structured way.
- What happens if one of the bundled requests fails and the client simply retries the entire bundle? This can lead to unintended consequences, such as duplicate processing of requests that were already successful. An idempotent design of the request processing can help mitigate this risk. See the Idempotent Receiver pattern by Hohpe and Woolf [13] for more information.

A deeper discussion of the pros and cons of the Request Bundle pattern is available on the "Patterns for API Design" website and in [41].

Related Content. A blog post on API design by James Higginbotham titled "API Design Guidance: Bulk and Batch Import"[13] clarifies the difference between batch and bulk processing and gives related advice.

Merge Operations [30, pages 15–18] can be used in preparation of *Bundle Requests* if the requests that should be bundled currently target different operations. Merging these into a single operation enables the introduction of a *Request Bundle*.

Introduce Pagination [30, pages 10–12] is an alternative to improve performance, focussing on responses rather than requests.

As an alternative solution to expose batch processing in an API, you may want to consider using a job scheduler rather than bundled API operations.

Note that we do not feature an "Unbundle Requests" refactoring in our catalog.

3.3 Refactoring: Move Operation

Context and Motivation. One or more API endpoints (for instance, HTTP resources) have been developed, tested, and deployed. One endpoint contains multiple operations (for instance, HTTP PUT and POST methods). The externally exposed responsibilities of these operations differ with regards to stakeholder groups and their concerns; some of them work with different domain concept(s) in the underlying API implementation. For instance, some operations are process-oriented activities while others are data-oriented and have information management responsibilities; the quality characteristics, including data protection needs, of these operations differ. As a consequence, the endpoint serves multiple roles in the API architecture. There is a risk that these roles drift further apart during API evolution.

> *As an API provider, I want to focus and consolidate the operations of an existing endpoint on a single role and purpose so that API client developers serving a particular stakeholder group understand the API design intuitively and the provider has a single reason to change that endpoint. To achieve this goal, I want to move one or more operations to a different, already existing endpoint.*

[13] https://tyk.io/api-design-guidance-bulk-and-batch-import/.

Stakeholder Concerns

#**maintainability** There are many drivers for changing an API [29]. APIs should be changed as much as required and as little as possible, and ripple effects be avoided. One of the first steps in maintenance tasks is to determine which parts of a system have to be changed. Separation of concerns tends to make changes smaller, but also adds to the amount of dependencies between system parts.

#**scalability and** #**reliability** When co-located in a single endpoint and deployed jointly, operations might influence each other. For instance, if one of these operations is long-running or causes a provider-internal error, its siblings might suffer from quality-of-service degradation too.

#**single-responsibility-principle and** #**independent-deployability** Principles such as single responsibility and independent deployability can guide the selection of refactorings. Some principles are affected positively, others negatively. With respect to the POINT[14] principles for API design (purposeful, style-oriented, isolated, channel-neutral, and T-shaped), moving an operation to another endpoint can improve **P**, **O**, and **I** (but might harm **T** when looking at a single endpoint and not an entire API). Note that some principles might be defining an architectural style; for instance, statelessness of interactions in REST. Such prominent role makes it even more important to adhere to a principle during design, and stick to its evolution.

#**understandability (a.k.a. explainability)** Endpoints with clearly stated, properly separated concerns and distinguished roles help developers to quickly understand and navigate the API. More documentation has to be studied, but its parts are easier to grasp.

Initial Position Sketch. The following platform- and technology-independent API design, specified in MDSL,[15] serves as the starting point for this refactoring:

```
endpoint type SourceEndpoint
exposes
  operation operation1
    expecting payload "RequestMessage1"
    delivering payload "ResponseMessage1"
  operation operation2
    expecting payload "RequestMessage2"
    delivering payload "ResponseMessage2"

endpoint type TargetEndpoint
    // zero or more already existing operations
```

See Fig. 7 for a graphical representation of this initial position sketch. Note that the sketch does not show any particular application/domain semantics or desired qualities that are violated at present; see the following example for such design smells.

[14] https://medium.com/olzzio/apis-should-get-to-the-point-c79113efa31c.
[15] https://microservice-api-patterns.github.io/MDSL-Specification/.

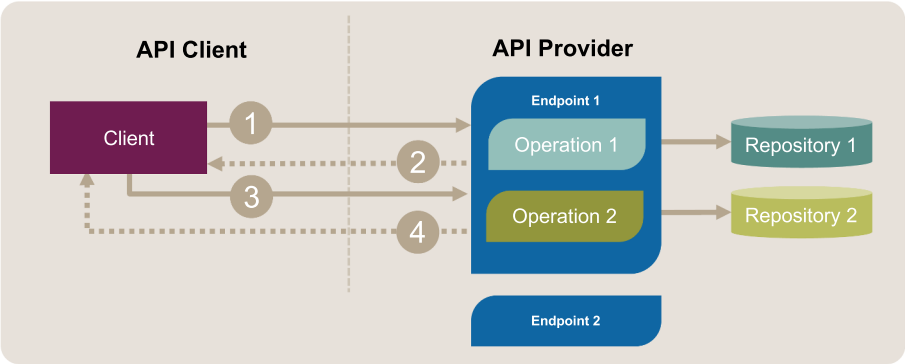

Fig. 7. Move Operation: Initial Position Sketch. A client uses two different operations (message exchanges 1–2 and 3–4) in an API endpoint.

This refactoring targets API endpoints that have several operations, one of which will be moved to an already existing second endpoint (if no such endpoint exists, a new one can be created; this design change is described in the *Extract Operation* [31, pages 10–12] refactoring).

Design Smells

Data lifetime mismatches Two operations in an endpoint deal with data that changes differently, both on the data definition and on the data manipulation level. For instance, one operation may expose master data[16] and the other one may expose operational data, also known as transaction data.[17] This causes undesired constraints on endpoint evolution. Different lifetimes make caching, especially cache invalidation, harder to manage.

Endpoint implementation spaghetti There are several n to m relations between API endpoints and their implementation parts. At least one endpoint works with many implementation parts that evolve independently.

God endpoint One endpoint contains a large number of operations that do not serve related purposes. Derived from the term "god class" [21] in object-oriented design, the term describes a class or object that controls many other parts of a system. Many such dependencies on external systems and data make the API implementation difficult to operate and evolve.

Role and/or responsibility diffusion The endpoint is both an Information Holder Resource and a Processing Resource. It is hard to explain what it does coherently. For instance, some of the exposed data lives long and changes rarely; other data goes through many create, update, delete operations in a short time span.

[16] https://en.wikipedia.org/wiki/Master_data.
[17] https://en.wikipedia.org/wiki/Transaction_data.

Too coarse-grained security or data privacy rules Some operations in an endpoint have more advanced quality requirements than others. For instance, some of them may work with sensitive personal information that has to be protected while others merely operate on public data. Access control policies for operations and shared data may vary as well. Other quality requirements mismatches might exist, for instance regarding availability and scalability.

Instructions. Follow these steps to apply this refactoring:

1. Remove the operation from the API Description of the endpoint that currently exposes it, or mark it as deprecated and soon to be retired.
2. Refactor at the code level (for instance, Spring REST controllers when implementing the API in Java). Optionally, implement a new stub that merely redirects clients to the new endpoint operation (in HTTP, this can be achieved with URL redirection and status code `301` [7]).
3. Review the required and provided security policies including the access rights management (both in source and in target endpoint) and adjust as needed.
4. Add the operation to the target API Description. Update all documentation accordingly.
5. Test whether the source and the target endpoint still meet their contracts (both functional and non-functional characteristics).
6. Evaluate whether smells are removed (or at least some desired qualities have improved); record any remaining or new technical debt and identify additional follow-on refactorings.
7. Inform all API clients about the completed change (in which version will it be introduced?) and provide migration information (or support the transition on a technical level, for instance with an HTTP redirect).

Depending on the backwards-compatibility measures taken by the provider, clients may need to update their request URLs. Alternatively, if URL redirection is used, clients may automatically pick up the new location. See the *Rename Endpoint* refactoring example for details on how to implement this using the Spring Boot Web framework.

Target Solution Sketch (Evolution Outline). The following simple and abstract MDSL sketch shows the API design after the *Move Operation* steps have been performed:

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

endpoint type TargetEndpoint
```

```
exposes
  // already existing operations (if any) still there
  operation operation2
    expecting payload "RequestMessage2"
    delivering payload "ResponseMessage2"
```

A visual target solution sketch is shown in Fig. 8.

Example(s). In a publication management system such as JabRef,[18] the remote service layer exposing a SOLUTION-INTERNAL API for FRONTEND INTEGRATION in a Web application might look like Fig. 9. There are two endpoints that offer operations to query and manage publications, represented by the `PublicationManagementCommands` and the `PublicationManagementQueries` services. The `PublicationManagementQueries` service suffers from role and/or responsibility diffusion: The `bulkImportFromBibtex` operation is not a query method and should be moved to `PublicationManagementCommands`. Applying the refactoring, we can clean up this mismatch.

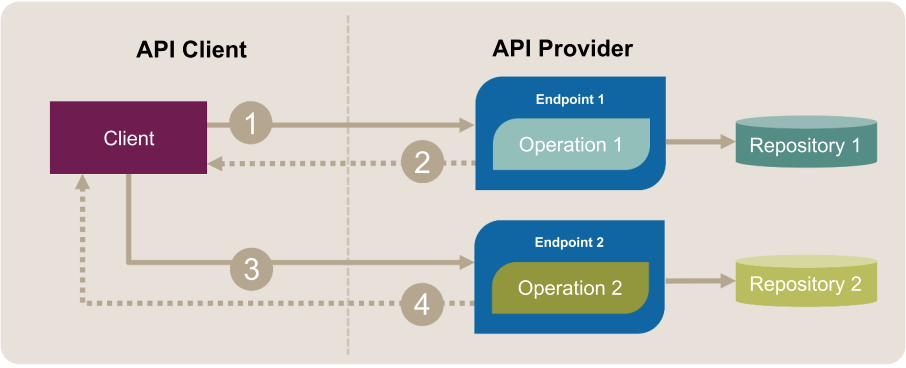

Fig. 8. Move Operation: Target Position Sketch. The client's API conversations (message exchanges 1–2 and 3–4) now go to two different API endpoints.

Hints and Pitfalls to Avoid. It is worth checking the following concerns:

- Is the operation name still adequate in the new endpoint? If not, *Rename Operation* [31, pages 12–14] can be applied.
- Is the technology mapping of the request and response messages still ok? For instance, the MIME types in RESTful HTTP as well as the platform-specific authentication and authorization settings might have to be changed because the target endpoint receives an additional (sub-)responsibility.

[18] https://www.jabref.org/.

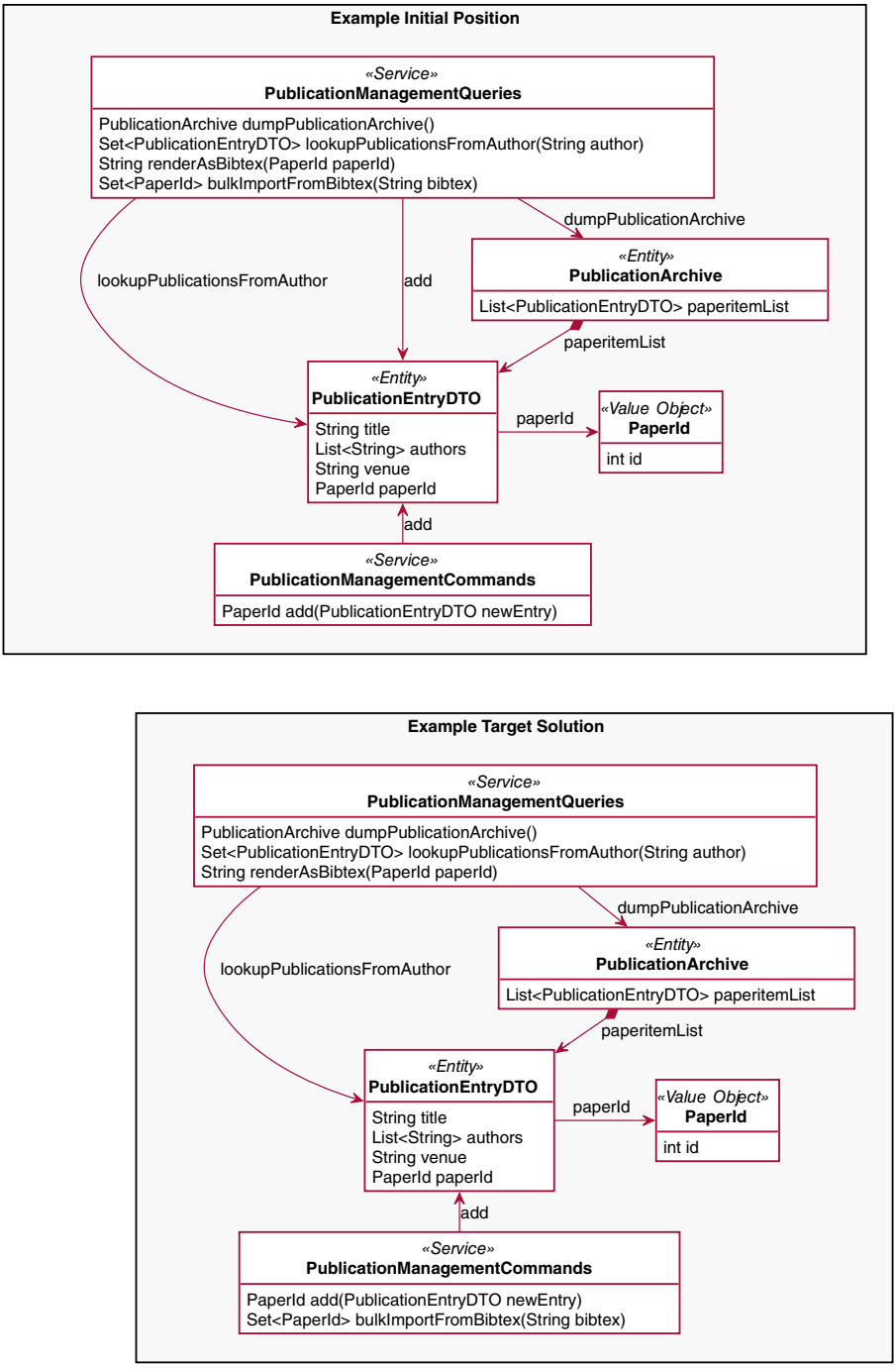

Fig. 9. Example of Refactoring Move Operation

- Are additional test cases and test data required in the API endpoint that the operation is moved to?
- Has the deprecation been communicated to the API clients? If the operation is not deprecated, has the API client been informed about the change?

The effect of the new API design on the compatibility with the original one should be communicated. If needed, migration instructions and simple migration tools (such as scripts) may be provided.

Related Content. The refactoring *Move Operation* reverts itself; it can be undone by applying it again (and move the operation in the reverse direction). In code refactoring, *Move Method* [9] accomplishes a similar goal.

Consider *Split Operation* [30, pages 12–15] if the operation has several responsibilities and only some of them should be moved. *Rename Operation* [31, pages 12–14] is related. *Extract Operation* [31, pages 10–12] is eligible when an operation should be moved but no suitable target endpoint exists.

The Design Practice Reference/Repository (DPR) [38] has a stepwise service design activity, which works with Candidate Endpoint Lists[19] and Refined Endpoint Lists.[20]

The API evolution patterns by Zimmermann et al. [41] (also available as "Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles" by Lübke et al. [18]) cover versioning. For instance, the patterns Semantic Versioning and Two in Production might be applicable.

3.4 Refactoring: Merge Endpoints

Context and Motivation. An API consists of multiple endpoints that expose one or more operations. Two or more of these operations are strongly related to each other. For instance, one can only be executed when the other one has succeeded, or one prepares the data that the other one works with. Because of their mutual dependencies, they have the same reasons to change and usually do so at the same time. This creates an undesired, rather tight coupling between the endpoints.

> *As an API provider, I want to bring the operations from multiple endpoints together and consolidate them in a single endpoint so that they can be deployed, scaled, and evolved jointly.*

[19] https://socadk.github.io/design-practice-repository/artifact-templates/SDPR-CandidateEndpointList.html.

[20] https://socadk.github.io/design-practice-repository/artifact-templates/SDPR-RefinedEndpointList.html.

Stakeholder Concerns

#**cohesion** Cohesion refers to "the degree to which the elements inside a module belong together" according to Wikipedia.[21] In the context of this refactoring, the "elements" are the API operations, and the "modules" are the API endpoints. API designers, and software engineers in general, strive for high cohesion, meaning that operations that belong together are offered by the same endpoint. Stevens et al. [27] define a scale of cohesiveness that ranges from coincidental, logical, temporal, communicational, sequential to functional cohesiveness. In the highest form, functional cohesiveness, "all of the elements [of a module] are related to the performance of a single function." The Systems Engineering Body of Knowledge[22] also provides an explanation of this general term.

#**coupling** According to the Wikipedia entry[23] for this term, coupling is the degree of interdependence between software modules, a measure of how closely connected two modules or API endpoints are. "Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules" [27]. The pattern summary of Loose Coupling[24] suggests time, platform, reference, and format as key autonomy dimensions. "The Many Facets of Coupling"[25] and other "ramblings" by Gregor Hohpe suggest a "nuanced discussion of coupling".

#**team-organization** Both well-established [4] and newer literature [26] emphasizes the impact that team structures and organization charts have on software structure and software quality. "Team Topologies"[26] suggests four types of teams: stream-aligned team, platform team, enabling team, and complicated subsystem team. Assuming that these teams provide and consume APIs, changes to the team organization may cause a need to refactor their APIs.

#**understandability (a.k.a. #explainability)** Fewer endpoints might be easier to understand and maintain—if they are cohesive and do not contain too many operations. The refactorings *Move Operation* and *Rename Operation* [31, pages 12–14] explain the quality attributes maintainability and understandability further.

Initial Position Sketch. The sketch in Fig. 10 shows that the refactoring targets two endpoints and their operations.

The client must be aware of and understand both endpoints. According to the context, they are semantically coupled through their operations, so the current API design does not meet the desire for high cohesion.

[21] https://en.wikipedia.org/wiki/Cohesion_(computer_science).
[22] https://www.sebokwiki.org/wiki/Cohesion_(glossary).
[23] https://en.wikipedia.org/wiki/Coupling_(computer_programming).
[24] https://www.cloudcomputingpatterns.org/loose_coupling/.
[25] https://www.enterpriseintegrationpatterns.com/ramblings/coupling_facets.html.
[26] https://teamtopologies.com/key-concepts.

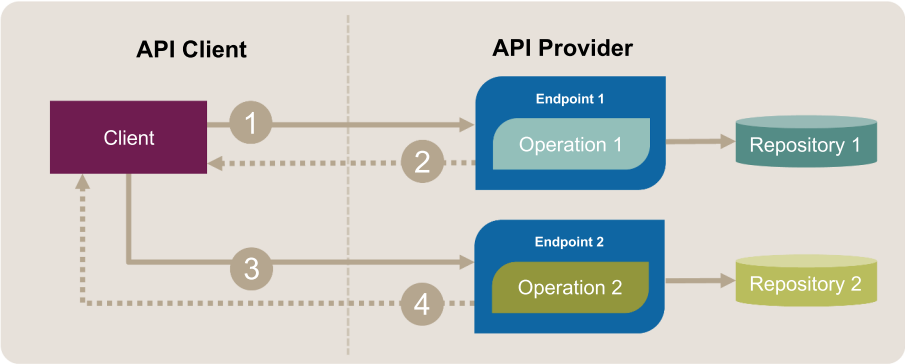

Fig. 10. Merge Endpoints: Initial Position Sketch. An API client uses two different operations (message exchanges 1–2 and 3–4) in two API endpoints for its communication with the API.

Design Smells

API does not get to the POINT According to the **I** in the POINT principles, which stands for *Isolation*, API operations should be free of unexpected side effects; they should not interfere with calls to other operations in the same or other APIs. The blog post "APIs should get to the POINT"[27] explains the isolation principle (and the other four POINTs) in depth.

Cloud-native traits violated Cloud-Native Applications (CNAs) should be modular with each deployment unit having an adequate size. If two API endpoints are coupled and their operations depend on each other, this rule might be violated. The blog post "What is a Cloud-Native Application Anyway? 10 SUPER-IDEAL Application Properties and 7 Cloud-Native Traits"[28] provides a summary of modularization and other CNA traits.

Endpoint implementation spaghetti The same backend system and/or domain data is processed by multiple endpoints. The endpoints are not self-contained and autonomous, but depend on each other indirectly because they access one or more shared implementation resources (such as another system or a data store). Such "implementation spaghetti" violates several of the defining principles of service-oriented architectures and microservices,[29] for instance *independent deployability*.

Extreme decomposition A strong desire to decompose an API and its implementation into independently deployable units can be observed, for instance caused by external forces such as the popularity of microservices. This desire has led to an intense decomposition that cannot be justified by requirements

[27] https://medium.com/olzzio/apis-should-get-to-the-point-c79113efa31c.
[28] https://medium.com/olzzio/what-is-a-cloud-native-application-anyway-part-2-f0e88c3caacb.
[29] https://ozimmer.ch/patterns/2020/07/06/MicroservicePositions.html.

and context; numerous endpoints expose narrowly-scoped operations. These operations call each other or have to be orchestrated on the client side.

Responsibility spread Having two highly coupled operations in different modules (here: endpoints) is an indication that the single responsibility principle[30] is violated. As another example, consider a stakeholder group that has to work with multiple endpoints to satisfy its information needs.

Synonyms for single concept ("aliasitis") Several endpoints have different names but actually expose the same domain concept and are therefore coupled tightly.[31] This smell is related to *responsibility spread*. Using different names for the same concept to distinguish multiple endpoints could indicate that things are separated that actually belong together. A domain-driven approach such as Context Mapping[32] help decide whether endpoints truly belong to different Bounded Contexts or should be merged.

Instructions. Before applying this refactoring, decide for an evolution strategy[33] for the new merged endpoint that balances the different forces and stakeholder concerns according to the needs of both the provider and the clients of the previously existing endpoints. The following steps can then be used to apply the refactoring:

1. Move all operations from the source endpoint to the target endpoint. If an intermediate mapping/configuration from endpoints to classes/methods exists in the Web frameworks that are used to implement the API, adjust both the configuration and the code.[34]
2. Depending on the chosen evolution strategy, implement a new stub to redirect clients to the new endpoint (in HTTP, this can be achieved with URL redirection and status code `301` [7]).
3. Delete the now orphaned/empty source endpoint from the code base.
4. Update API test cases. Run them and compare the test results with those of the previous version to ensure that clients are served in the same way as before, both from a functional and from a non-functional point of view.
5. Update both API Descriptions (source and target) as well as the related Service Level Agreements. The API description of the target solution must reflect the new design; the one of the source should report the change and, if implemented, the redirect.
6. Also update all API directories (i.e., service registries and repositories) and/or portals (i.e., gateways, cluster managers, service meshes) that refer to source and target endpoints.
7. Communicate these changes to the affected clients early and consistently.

[30] https://en.wikipedia.org/wiki/Single-responsibility_principle.
[31] https://www.enterpriseintegrationpatterns.com/ramblings/coupling_facets.html.
[32] https://www.infoq.com/articles/ddd-contextmapping/.
[33] https://api-patterns.org/patterns/evolution/.
[34] An example of such framework is Play https://www.playframework.com/. Other frameworks work with annotations; Spring and JAX-RS/JAW-WS containers are examples of this distributed configuration approach.

Target Solution Sketch (Evolution Outline). After merging the endpoints, the operations are co-located in a single endpoint. Figure 11 visualizes the new endpoint design.

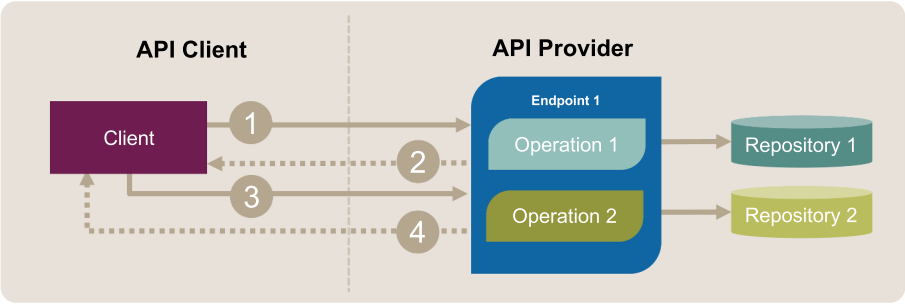

Fig. 11. Merge Endpoints: Target Solution Sketch. After the refactoring, the operations (message exchanges 1–2 and 3–4) are co-located in a single API endpoint.

Example(s). Reverting the target example to the initial/source in *Extract Operation* [31, pages 10–12] qualifies as an example of this refactoring. After Extract Operation has been applied, the publication management example may provide the following command and query endpoints:[35]

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

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

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

[35] The notation in this example is CML, the tactic domain-driven design language supported by Context Mapper https://contextmapper.org/docs/language-reference/.

These two endpoints can be merged into one, leading to the following API design:

```
Aggregate PublicationEndpoint {
    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);
    }
}
```

Hints and Pitfalls to Avoid. Before and when applying this refactoring, make sure to:

- Watch out for naming conflicts; if a conflict does occur, apply the *Rename Operation* [31, pages 12–14] refactoring before the merge.
- In HTTP resource APIs, make sure that the merge does not cause any conflicts in the operation-to-verb mappings. Each resource, uniquely identified with a URI, is able to support one GET, one POST and one PUT operation (and so forth) only. Update the URI/resource naming scheme if required, for instance via sub-resources.
- Do not take this refactoring to the extreme; creating one endpoint per stakeholder group or per application usually does not make much sense (end user requirements and API-level user/job stories will help to unveil the right API granularity.

The source and the target endpoints may have different security requirements. For instance, some operations may require rather fine-grained, attribute-based authorization while others use role-based authorization (or none). Another case might be one endpoint using basic authentication and API Keys and the other one using OpenID Connect and OAuth. Such situations can either be (viewed as) opportunities to improve the weaker security solution, or as threats because they could cause unexpected/undesired complexity. It is important to identify such mismatches before applying the refactoring and make a conscious architectural decision[36] whether to merge or not.

[36] https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html.

Related Content. This refactoring undoes *Extract Operation* [31, pages 10–12]. A simpler form of it is *Move Operation*. Viewed at the code level, Inline Class[37] is a related refactoring that can be used to merge the implementation classes of the endpoints.

There are different reasons why tightly-coupled endpoints exist. If the endpoints are responsible for data retrieval, they might provide access to the same underlying data, but for clients with different information needs. In that case, the *Add Wish List* [30, pages 7–10] or *Add Wish Template* refactorings might be suited as well.

The general topic of API granularity is discussed in "What is the Right Service Granularity in APIs?"[38]). The Service Cutter[39] tool and method suggest sixteen coupling criteria such as "Semantic Proximity", "Structural Volatility", and "Security Contextuality" [10]. These criteria are worth considering when merging endpoints and merging operations.

3.5 Refactoring: Rename Endpoint

Context and Motivation. An API endpoint runs in production and is used by clients. Its name does not match its domain semantics well; this name is a build-time concept that might also be visible at runtime, for instance in the address of the endpoint on the protocol level.

> *As an API developer, I want to express the domain concepts exposed by an endpoint and the architectural role that the endpoint plays as accurately as possible in the published language that the API exposes, in the API contract in particular.*

Stakeholder Concerns

#learnability (#readability in particular) Using expressive and meaningful names makes navigating a code base easier, for instance when searching for endpoints while fixing bugs and adding features. The *Move Operation* refactoring has more explanations of such learnability and readability qualities.

#understandability (a.k.a. #explainability) A good name expresses what an endpoint has to offer, which helps clients decide whether and how to use it. The *Rename Operation* [31, pages 12–14] refactoring provides additional explanations of the understandability and explainability qualities.

Initial Position Sketch. Let's assume that an API endpoint type called `OldCrypticName` exists (note that this name is deliberately poor, to make the change and the need for it clear):

[37] https://refactoring.guru/inline-class.
[38] https://www.informit.com/articles/article.aspx?p=3153211.
[39] https://github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria.

```
endpoint type OldCrypticName
exposes
  operation operation1
    expecting payload "SomeRequestMessage"
```

The refactoring targets a single endpoint that contains one or more operations (here: just one).

Design Smells

Cryptic, misleading or disrespectful name The given endpoint name is not straightforward to understand and/or is not part of the domain terminology. It may raise wrong, unexpected or undesired associations or simply be outdated. It might even be unethical/irresponsible and offend or insult certain parts of the API community. For example, terms like "Blacklist/Whitelist" (alternative: "Blocklist/Allowlist") or "Master/Slave" (alternative: "Primary/Secondary" or "Leader/Follower") have been replaced in favor of more neutral and often also more accurate alternatives. Similarly, pattern names such as the "Strangler" have evolved to "Strangler Fig" to reduce negative connotations [34], though even these may still be problematic. When selecting names, consider ethical implications and inclusivity. See "Nonviolent Communication" by Rosenberg [23] and papers on ethics in technology [14,16].

REST principle(s) violated Abstract API endpoints correspond to resources identified by URIs in RESTful HTTP APIs. Their names should convey the meaning and role of the resources. For instance, using verbs as relative paths might be considered an antipattern; resources should be named with nouns that can be traced back to domain entities (note that these entities and their resource representations can be Processing Resources and/or data-oriented Information Holder Resources both in RESTful HTTP APIs and in APIs leveraging other integration styles and protocols).

Instructions. This refactoring is rather straightforward to apply.[40] The main task is to find a suited name that balances the desired qualities and stakeholder concerns:

1. Discuss alternative new names and decide for one. Have this decision reviewed and agreed upon.
2. Apply code-level rename refactorings, for instance for the controller class realizing the endpoint. If the code is generated from a specification, update the specification and regenerate the code.

[40] assuming that the used tools (including IDEs, API contract editors, and test tools) support consistent find-and-replace across files, and do so conveniently and reliably.

3. For backward compatibility, consider implementing a redirect, either at the protocol level (for instance with an HTTP redirect [7]) or at the code level by keeping the old method and calling the new one from it. If the name change is noticeable to API clients, mark the old endpoint name as deprecated in the API DESCRIPTION (for instance, its OpenAPI Specification, OAS) and announce that it will be removed in future versions of the API. Plan this cleanup work for later.
4. Update security rules, tests and documentation as well as any address resolution services (gateways, portals, registries, repositories, service meshes). Check that the old name has been updated everywhere, e.g., OAS and examples; use a global search in editor or command line for that.
5. Notify all known API clients of the change, ideally with detailed migration instructions.

Target Solution Sketch (Evolution Outline). The endpoint type is called `NewExpressiveNameFromDomainVocabulary` now (for demonstration purposes; when applying the refactoring, an actual domain term would be used):

```
endpoint type NewExpressiveNameFromDomainVocabulary
exposes
  operation operation1
    expecting payload "SomeRequestMessage"
```

Example(s). In a publication management example, an application of the refactoring might change the following interface (notation: CML):

```
Aggregate APIEndpoint {
    Service JabRefDatabaseFacade {
        Set<@PublicationEntryDTO>
          lookupPublicationsFromAuthor(String writer);
        ...
    }
}
```

to this one:

```
Aggregate PublicationManagementEndpoint {
    Service PublicationManagementFacade {
        Set<@PublicationEntryDTO>
          lookupPublicationsFromAuthor(String writer);
        ...
    }
}
```

The specific name of an open source tool, JabRef, was replaced by a more general, domain-specific name. This reduces the prerequisite knowledge that stakeholders of the name (such as API client developers, but also API implementation maintainers and API product managers and testers) must have.

In this second example, we will look at a concrete implementation in Java Web framework Spring Boot. Endpoints are defined as classes, with Java methods implementing the API operations. Annotations, such as `@GetMapping` and `@PathVariable` are used to instruct Spring how to map the URL paths and query parameters to Java code. The `CustomerInformationHolder` class has two methods, one for retrieving and another one for updating a customer (note that the code is simplified, the full implementation can be found on GitHub[41]).

```
@RestController
@RequestMapping("/getCustomers")
public class CustomerInformationHolder {
  ...
  @GetMapping(value = "/{ids}")
  public ResponseEntity getCustomer(@PathVariable String ids) {
    ...
  }
  @PutMapping(value = "/{id}")
  public ResponseEntity updateCustomer(
    ...
```

API clients can then retrieve the details of a customer by their id `bunlo9vk5f`:

```
curl  http://localhost:8110/getCustomers/bunlo9vk5f
{
  "customers" : [ {
    "customerId" : "bunlo9vk5f",
    "firstname" : "Ado",
    "lastname" : "Kinnett",
    "birthday" : "1975-06-13T23:00:00.000+00:00",
    "streetAddress" : "2 Autumn Leaf Lane",
    "postalCode" : "6500",
...
```

If we look closely at the path (`getCustomers`), we see that it violates the principle that no verbs should be used. Also, the endpoint offers operations to update customers, so the 'get' is also misleading and makes it difficult to understand. So let us rename it to simply `customers` by changing the request mapping:

```
@RestController
--- @RequestMapping("/getCustomers")
+++ @RequestMapping("/customers")
```

Unfortunately, this will break existing clients, which will now get an Error Report:

```
{
  "timestamp" : "2025-02-14T09:01:47.779+00:00",
```

[41] https://github.com/Microservice-API-Patterns/LakesideMutual.

```
  "status" : 404,
  "error" : "Not Found",
  "path" : "/getCustomers/bunlo9vk5f"
}
```

To inform clients about the new name, we can implement another controller class, which informs clients about the new location using HTTP status 301 (`HttpStatus.MOVED_PERMANENTLY`) and the `Location` header:

```
@RestController
@RequestMapping(path = "getCustomers")
public class OldCustomerInformationHolder {
  ...
  @GetMapping(value = "/{ids}")
  @ResponseBody
  public ResponseEntity getCustomer(@PathVariable String ids) {
    URI movedTo = linkTo(
      methodOn(CustomerInformationHolder.class).getCustomer(ids))
      .toUri();
    return ResponseEntity
      .status(HttpStatus.MOVED_PERMANENTLY)
      .location(movedTo)
      .build();
    }
}
```

Clients will then be redirected to the new location. We can see this by enabling the `--verbose` option. The `--location` instructs curl to follow redirects (greater-than > signs indicate requests, less-than < show responses).

```
curl --verbose --location http://localhost:8110/getCustomers/bunlo9vk5f
> GET /getCustomers/bunlo9vk5f HTTP/1.1
< HTTP/1.1 301
< Location: http://localhost:8110/customers/bunlo9vk5f
> GET /customers/bunlo9vk5f HTTP/1.1
< HTTP/1.1 200
{
  "customers" : [ {
    "customerId" : "bunlo9vk5f",
    "firstname" : "Ado",
    "lastname" : "Kinnett",
    "birthday" : "1975-06-13T23:00:00.000+00:00",
    "streetAddress" : "2 Autumn Leaf Lane",
    "postalCode" : "6500",
...
```

This approach is fully backwards compatible for clients that implemented the HTTP location header. To support clients that do not, we can also include the response in the redirect response:

```
return ResponseEntity
  .status(HttpStatus.MOVED_PERMANENTLY)
  .location(movedTo)
  .body(customerInformationHolder.getCustomer(ids).getBody());
```

As a next step, not shown here, we will update security rules to cover the old and new endpoint, tests and documentation.

Hints and Pitfalls to Avoid. Here is some advice regarding naming:

- Organization-, unit-, or project-wide naming conventions should be considered. It often makes sense to A/B test naming options with members of the target audience (here: API client developers).
- Domain-Driven Design[42] with its concepts of Ubiquitous Language and Published Language is well-suited to contribute naming suggestions. Singjai et al. [25] recommend "that names and abstractions in the API follow the terms defined in the ubiquitous language which is formally specified by the domain model."
- Naming collisions with other endpoints should be avoided (even if they are technically possible and permitted, which depends on the chosen protocol technology).
- URI templates `{id}` in HTTP resource APIs also require attention (and these names may appear in multiple places in API specifications and implementations).
- Consistency of name across an API and/or multiple related APIs maintained by the same organization is an important meta-quality; it contributes to conceptual integrity [2].

More hints can be found in the *Rename Operation* [31, pages 12–14] and *Rename Representation Element* [30, pages 18–20] refactorings.

Related Content. *Rename Operation* [31, pages 12–14] and *Rename Representation Element* [30, pages 18–20] also address readability, but have other parts of an API as targets.

The *Rename* refactoring, as described by Fowler [9], can be applied to any named program element, not just endpoints. It is also often implemented in integrated development environments for various programming languages [5,22,28].

Refactoring.Guru calls for Clean Code[43] and emphasizes that poor names make code difficult to grasp.

"The Art of Readable Code" [6] contains many helpful hints on naming program elements that also apply to APIs.

[42] https://socadk.github.io/design-practice-repository/activities/DPR-StrategicDDD.html.

[43] https://refactoring.guru/refactoring/what-is-refactoring.

The book "Clean Code" [19] by Robert C. Martin devotes an entire chapter to meaningful names, for instance recommending that we should *use intention-revealing names*, *avoid disinformation*, *use pronounceable names*. The advice given applies not only to code, but also to many other artifacts such as APIs and their parts.

4 Discussion

In this section, we discuss the pros and cons of our API refactorings (and their presentation in catalog form, with many entries targeting existing patterns) as well as related work.

4.1 Pros and Cons

Like design patterns, refactorings offer structured guidance to teams designing, developing and maintaining software (here: APIs). We presented each refactoring with context, motivation, example, and actionable instructions. Refactoring names and smell labels provide a shared vocabulary for API designers, provider and client developers and maintainers, and product owners. The stakeholder concerns help teams decide when a refactoring is appropriate.

Several refactorings not only clean up design smells but also align API structure with proven design patterns (e.g., Request Bundle). This improves the conceptual integrity of the API and eases understandability, an important stakeholder concern. Some patterns also add complexity, so they should not be introduced just for the sake of it, as this can lead to bloated interfaces or unnecessary abstractions. For example, *Add Wish Template* adds complexity and overhead to message structure and API contracts, which can confuse clients. In such cases, *Add Wish List* [30, pages 7–10] offers a simpler alternative. Separate and more focused operations for different client needs can be introduced using *Split Operation* [30, pages 12–15].

Granularity and backwards compatibility also differ among refactorings: Small, localized changes affecting a single operation (e.g., Wish Template) avoid ripple effects and reduce risk during API evolution. Other refactorings might be riskier if applied without caution. Applying *Move Operation* or *Rename Endpoint* must be accompanied by appropriate communication to stakeholders or be implemented backwards-compatible (e.g., by redirecting API clients). In Stocker and Zimmermann [30], we introduced the Test, Explain, Let Know, and Learn (TELL) approach to communicate API changes early, clearly, and continuously on/for the architectural level so that clients are not surprised.

Many refactorings (e.g., *Merge Endpoints*, *Move Operation*) are reversible. Some explicitly undo each other, such as *Extract Information Holder* [31, pages 3–6] and *Inline Information Holder* [31, pages 6–10]. This supports experimentation and agile evolution strategies.

Overall, the catalog provides flexible, pattern-oriented guidance for evolving APIs in a structured and maintainable way. Several open questions justify further

research; for instance, it is not trivial how to evaluate or measure whether the application of a refactoring was successful and had the desired impact. It is one thing to validate and demonstrate that the refactored API continues to work as expected; it is more challenging to provide evidence of a positive impact on client developer experience and other stakeholder concerns. Initial evidence that the usage of patterns improves understandability can be found in Bogner et al. [1]. Architecture and code-level metrics should be established; suggestions for such metrics from a practitioner perspective can be found in "Metrics for Architectural Synthesis and Evaluation – Requirements and Compilation by Viewpoint" [35] and in "Software Architecture Metrics" [3]. Monitoring and observability are discussed in "Observability Engineering: Achieving Production Excellence" [19].

4.2 Related Work

On the World-Wide Web, the system context boundaries become blurred; single software applications are replaced with inter-connected ecosystems of services forming a remote *API economy*. "No longer having control over all system components makes it difficult for teams to freeze designs, rendering design flexibility a top architectural quality" [12]. API governance and API management principles and practices have been established in industry, supported by open source tools, software products, and managed cloud services. API evolution and maintenance, which may involve API refactoring as defined in our papers and online catalog, has to fit into such initiatives.

We covered related work on architectural refactoring and remote API refactoring in this context rather extensively in our EuroPloP 2023 paper "API Refactoring to Patterns – Catalog, Template and Tools for Remote Interface Evolution" [30]; the EuroPLoP 2024 paper "Pattern-oriented API Refactoring: Addressing Design Smells and Stakeholder Concerns" [31] added pattern summaries from other pattern languages that serve as refactoring targets in IRC. This section provides an update on related works about these and other topics, both peer-reviewed and gray literature (including online resources).

Suryanarayana et al. [33] discuss the relationship between design smells and their removal on the one side and design processes and metrics on the other hand. The refactorings in this paper also highlight design smells; applying them systematically can become part of design and evolution processes.

"A Conceptual Framework for API Refactoring in Enterprise Application Architectures" presents API pattern implementations and refactorings for them that are scored according to their Efficiency, Maintainability, and Isolation (EMI). The service-oriented programming language Jolie is used to illustrate the design options [20].

We have been mining and documenting "Patterns for API Design" (formerly known as "Microservices API Patterns" [18] and "Interface Representation Patterns" [40]) since 2016. Our IRC leveraging many of these patterns as targets has been under construction since 2021 [30,31]. The paper "API Refactoring and Reengineering: Distribution Patterns and Evolution Strategies" [36] covers

related work in the areas of serverless cloud computing and software architecture and comes with a larger implementation example.

5 Summary

We presented a third slice from our Interface Refactoring Catalog (IRC) in this paper. It comprises five refactorings: two refactorings to patterns (*Add Wish Template*, *Bundle Requests*), two structural (*Move Operation*, *Merge Endpoints*) refactorings and one about naming (*Rename Endpoint*). A sibling paper [36] features three architectural refactorings (*Distribute Application Frontend*, *Split Application Backend Logic*, *Split Application Backend Persistence*) and two organizational ones (*Tighten Evolution Strategy*, *Relax Evolution Strategy*). We also discussed pros and cons of IRC usage.

We see a connection between API refactoring and sustainability. Refactorings like *Add Wish Template* and *Bundle Requests* can reduce message size, avoid unnecessary data transfer, and lower computational overhead. These effects align with green API design goals by improving resource efficiency and contributing to more sustainable software systems [14]. Future work concerns value tactics [39] in the context of API design and (tooling for) reusable architectural decision guidance models for API design.

We also plan to measure how effective our refactoring are, for instance with regard to API understandability [1]. Finally, we consider continuing our work on API design reviews,[44] which possibly may lead to the application of refactorings from IRC.

We welcome feedback from practitioners using the refactorings in real-world projects. Insights from API designers and developers will help assess practical relevance, identify gaps in the catalog, and guide future enhancements.

Acknowledgments. We would like to thank all the contributors to the interface refactoring catalog. We also thank our EuroPLoP 2025 shepherd, Fikret Basic, for his constructive and insightful feedback. We also want to thank the EuroPLoP 2025 Writers' Workshop A participants Andreas Fiesser, Christopher Preschern, Dennis Dubbert, Klaus Marquardt, and Simon Furrer for their valuable feedback.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

A Summary of Patterns for API Design

In the refactorings presented in this paper, we refer to patterns from Zimmermann et al. [41]. The pattern name, icon, problem and solution summary statements are listed in Table 2 to provide a quick reference. For more details, we refer the reader to the API Patterns website and book.

[44] https://medium.com/nerd-for-tech/a-checklist-for-api-design-reviews-5f7db45b0cb3.

Table 2. Patterns from Zimmermann et al. [41] mentioned in this paper, with their problem and solution summaries.

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.
API Key 	*Problem:* How can an API provider identify and authenticate clients and their requests? *Solution:* As an API provider, assign each client a unique token the API Key that the client can present to the API endpoint for identification purposes.
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.
Error Report 	*Problem:* How can an API provider inform its clients about communication and processing faults? How can this information be made independent of the underlying communication technologies and platforms (for example, protocol-level headers representing status codes)? *Solution:* Reply with error codes in response messages that indicate and classify the faults in a simple, machine-readable way. In addition, add textual descriptions of the errors for the API client stakeholders, including developers and/or end users such as administrators.

(*continued*)

Table 2. (*continued*)

Pattern Name	Pattern Summary (Problem and Solution)
Frontend Integration 	*Problem:* How can client-side end-user interfaces that are physically separated from server-side business logic and data storage be populated and updated with computing results, result sets from searches in data sources, and detailed information about data entities? How can application frontends invoke activities in a backend or upload data to it? *Solution:* Let the backend of a distributed application expose its services to one or more application frontends via a message-based remote Frontend Integration API.
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.
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.
Parameter Tree 	*Problem:* How can containment relationships be expressed when defining complex representation elements and exchanging such related elements at runtime? *Solution:* Define a Parameter Tree as a hierarchical structure with a dedicated root node that has one or more child nodes. Each child node may be a single Atomic Parameter, an Atomic Parameter List, or another Parameter Tree, identified locally by a name and/or by position. Each node might have an exactly-one cardinality, but also a zero-or-one cardinality, an at-least-one cardinality, or a zero-or-more cardinality.

(*continued*)

Table 2. (*continued*)

Pattern Name	Pattern Summary (Problem and Solution)
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.
Request Bundle 	*Problem:* How can the number of requests and responses be reduced to increase communication efficiency? *Solution:* Define a Request Bundle as a data container that assembles multiple independent requests in a single request message. Add metadata such as identifiers of individual requests and bundle element counter.
Service Level Agreement 	*Problem:* How can an API client learn about the specific quality-of-service characteristics of an API and its endpoint operations? How can these characteristics, and the consequences of not meeting them, be defined and communicated in a measurable way? *Solution:* As an API product owner, establish a structured, quality-oriented Service Level Agreement that defines testable service-level objectives.
Solution-Internal API 	*Problem:* How can access to and usage of an API be limited to an application, for instance, components in the same or another logical layer and/or physical tier? *Solution:* Decompose the application logically into components. Let these components expose local or remote APIs. Offer these APIs only to system-internal communication partners such as other services in the application backend.
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.

References

1. Bogner, J., Wójcik, P., Zimmermann, O.: How do microservice API patterns impact understandability? A controlled experiment. In: 21st IEEE International Conference on Software Architecture, ICSA 2024, Hyderabad, India, 4–8 June 2024, pp. 158–169, IEEE (2024), https://doi.org/10.1109/ICSA59870.2024.00023, URL https://doi.org/10.1109/ICSA59870.2024.00023
2. Brooks, F.P.: The Mythical Man-Month: Essays on Software-Engineering. Addison-Wesley Longman Publishing Co., Inc, USA, 1st edn. (1978), ISBN 0201006502
3. Ciceri, C., et al.: Software Architecture Metrics. O'Reilly Media (2022), ISBN 9781098112202
4. Conway, M.E.: How do committees invent? Datamation, April 1968, http://www.melconway.com/research/committees.html
5. Corbat, T., Felber, L., Stocker, M.: Refactoring support for the ruby development tools. In: Bleek, W.G., Schwentner, H., Züllighoven, H. (eds.) Software Engineering (Workshops), LNI, vol. P-106, pp. 313–315, GI (2007), ISBN 978-3-88579-200-0, URL http://dblp.uni-trier.de/db/conf/se/se2007w.html#CorbatFS07
6. Dustin Boswell, T.F.: The Art of Readable Code. Inc, O'Reilly Media (2011)
7. Fielding, R.T., Reschke, J.: Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, June 2014, https://doi.org/10.17487/RFC7231, URL https://www.rfc-editor.org/info/rfc7231
8. Ford, N., Richards, M., Sadalage, P., Dehghani, Z.: Software Architecture: The Hard Parts. O'Reilly Media (2021), ISBN 9781492086840
9. Fowler, M.: Refactoring. Addison-Wesley Signature Series (Fowler), Addison-Wesley, Boston, MA, 2 edn. (2018), ISBN 978-0-13-475759-9
10. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: a systematic approach to service decomposition. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, 5–7 September 2016, Proceedings, LNCS, vol. 9846, pp. 185–200, Springer (2016), https://link.springer.com/chapter/10.1007/978-3-319-44482-6_12
11. Hartig, O., Pérez, J.: Semantics and complexity of graphql. In: Proceedings of the World Wide Web Conference (WWW), pp. 1155–1164 (2018), ISBN 978-1-4503-5639-8, https://doi.org/10.1145/3178876.3186014
12. Hohpe, G., Ozkaya, I., Zdun, U., Zimmermann, O.: The software architect's role in the digital age. IEEE Softw. **33**(6), 30–39 (2016). https://doi.org/10.1109/MS.2016.137
13. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley (2003)
14. Kapferer, S., Stocker, M., Zimmermann, O.: Towards responsible software engineering: combining value-based processes, agile practices, and green metering. In: 2024 IEEE International Symposium on Technology and Society (ISTAS), pp. 1–4 (2024), https://doi.org/10.1109/ISTAS61960.2024.10732097
15. Kapferer, S., Zimmermann, O.: Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling, pp. 299–306, January 2020, https://doi.org/10.5220/0008910502990306
16. Kapferer, S., Zimmermann, O., Stocker, M.: Value-driven analysis and design: applying domain-driven practices in ethical software engineering. In: Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices, EuroPLoP 2024, ACM, New York, NY, USA (2024), ISBN 9798400716836, https://doi.org/10.1145/3698322.3698332

17. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004), ISBN 0321213351
18. Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., Stocker, M.: Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In: Proceedings of the 24th European Conference on Pattern Languages of Programs, EuroPLop 2019, ACM, New York, NY, USA (2019), ISBN 9781450362061, https://doi.org/10.1145/3361149.3361164, URL https://doi.org/10.1145/3361149.3361164
19. Majors, C., Fong-Jones, L., Miranda, G.: Observability Engineering. O'Reilly Media (2022), ISBN 9781492076414
20. Montesi, F., Peressotti, M., Picotti, V., Zimmermann, O.: A Conceptual Framework for API Refactoring in Enterprise Application Architectures. In: European Conference on Service-Oriented and Cloud Computing (ESOCC), ESOCC 2025, Springer LNCS (2025)
21. Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, Reading, MA (1996), ISBN 978-0-201-63385-6
22. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for smalltalk. Theory Pract. Object Syst. **3**, 253–263 (1997)
23. Rosenberg, M.: Nonviolent Communication: A Language of Life. BusinessPro Collection, Independent Publishers Group (2003), ISBN 9781892005038
24. Serbout, S., Pautasso, C., Zdun, U., Zimmermann, O.: From OpenAPI fragments to API pattern primitives and design smells. In: 26th European Conference on Pattern Languages of Programs, EuroPLoP 2021, ACM, New York, NY, USA (2022), ISBN 9781450389976, https://doi.org/10.1145/3489449.3489998, URL https://doi.org/10.1145/3489449.3489998
25. Singjai, A., Zdun, U., Zimmermann, O., Pautasso, C.: Patterns on deriving APIs and their endpoints from domain models. In: Proceedings of the 26th European Conference on Pattern Languages of Programs, EuroPLoP 2021, ACM, New York, NY, USA (2022), ISBN 9781450389976, https://doi.org/10.1145/3489449.3489976, URL https://doi.org/10.1145/3489449.3489976
26. Skelton, M., Pais, M., Malan, R.: Team Topologies: Organizing Business and Technology Teams for Fast Flow. IT Revolution (2019), ISBN 9781942788829
27. Stevens, W.P., Myers, G.J., Constantine, L.L.: Structured design. IBM Syst. J. **13**(2), 115–139 (1974), ISSN 0018–8670, https://doi.org/10.1147/sj.132.0115, URL https://doi.org/10.1147/sj.132.0115
28. Stocker, M.: Scala Refactoring. Master's thesis, HSR Hochschule für Technik Rapperswil, http://eprints.ost.ch/id/eprint/286 (2010)
29. Stocker, M., Zimmermann, O.: from code refactoring to API refactoring: agile service design and evolution. In: Barzen, J. (ed.) Service-Oriented Computing, pp. 174–193, Springe, Cham (2021), ISBN 978-3-030-87568-8, https://doi.org/10.1007/978-3-030-87568-8_11
30. Stocker, M., Zimmermann, O.: API Refactorings to patterns: catalog, template and tools for remote interface evolution. In: Proceedings of the 28th European Conference on Pattern Languages of Programs, EuroPLop 2023, ACM, New York, NY, USA (2023), ISBN 979-8-4007-0040-8, https://doi.org/10.1145/3628034.3628073, URL https://doi.org/10.1145/3628034.3628073
31. Stocker, M., Zimmermann, O., Kapferer, S.: Pattern-oriented api refactoring: addressing design smells and stakeholder concerns. In: Proceedings of the 29th European Conference on Pattern Languages of Programs, EuroPLop 2024, ACM, New York, NY, USA (2024), https://doi.org/10.1145/3698322.3698334, URL https://doi.org/10.1145/3698322.3698334

32. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2014), ISBN 0128013974
33. Suryanarayana, G., Sharma, T., Samarthyam, G.: Software process versus design quality: tug of war? IEEE Softw. **32**(4), 7–11 (2015). https://doi.org/10.1109/MS.2015.87
34. Yoder, J.W., Merson, P.: Strangler patterns. In: Proceedings of the 27th Conference on Pattern Languages of Programs, PLoP 2020, The Hillside Group, USA (2022), ISBN 9781941652169
35. Zimmermann, O.: Metrics for architectural synthesis and evaluation - requirements and compilation by viewpoint. an industrial experience report. In: Ozkaya, I., Nord, R.L., Koziolek, H., Avgeriou, P. (eds.) 2nd IEEE/ACM International Workshop on Software Architecture and Metrics, SAM 2015, Florence, Italy, 16 May 2015, pp. 8–14, IEEE Computer Society (2015), https://doi.org/10.1109/SAM.2015.9, URL https://doi.org/10.1109/SAM.2015.9
36. Zimmermann, O., Kapferer, S., Stocker, M.: API refactoring and reengineering: distribution patterns and evolution strategies (2025), submitted to EuroPLoP '25
37. Zimmermann, O., Milinski, S., Craes, M., Oellermann, F.: Second generation web services-oriented architecture in production in the finance industry. In: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2004, pp. 283–289, ACM, New York, NY, USA (2004), ISBN 1581138334, https://doi.org/10.1145/1028664.1028772, URL https://doi.org/10.1145/1028664.1028772
38. Zimmermann, O., Stocker, M.: Design Practice Reference - Guides and Templates to Craft Quality Software in Style. LeanPub (2021), URL https://leanpub.com/dpr
39. Zimmermann, O., Stocker, M., Kapferer, S.: Bringing ethical values into agile software engineering. In: Smart Ethics in the Digital World: Proceedings of the ETHICOMP 2024. 21th International Conference on the Ethical and Social Impacts of ICT, pp. 90–93, Universidad de La Rioja (2024)
40. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U.: interface representation patterns - crafting and consuming message-based remote APIs. In: 22nd European Conference on Pattern Languages of Programs (EuroPLoP 2017), pp. 1–36, July 2017, https://doi.org/10.1145/3147704.3147734, URL http://eprints.cs.univie.ac.at/5161/
41. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C.: Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley Signature Series (Vernon), Addison-Wesley Professional (2022), ISBN 9780137670109