TUM Department of Informatics
Technical University of Munich

Master's Thesis in Information Systems

A Process Model for Client Migration after Service Changes in Distributed Systems

Andre Weinkötz

TUM Department of Informatics
Technical University of Munich

Master's Thesis in Information Systems

A Process Model for Client Migration after Service Changes in Distributed Systems

Ein Prozessmodell für Anwendungsmigrationen nach Service Änderungen in verteilten Systemen

Author: Andre Weinkötz
Supervisor: Prof. Dr. Bernd Brügge
Advisors: Paul Schmiedmayer, M. Sc.
Submission Date: January 15, 2020

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, January 15, 2020

ANDRE WEINKÖTZ

Abstract

Modern web-based APIs offer a broad variety of services for numerous distributed client applications. Web APIs must evolve to remain functional and adapt to changes in requirements or their environment. Ideally, an API contract creates a consistent, shared understanding of its service between provider and consumer. The provider-side evolution of an API often introduces changes to this contract that consumers need to manually adapt to. Thus far, there is no automated process for migrating between different Web API versions. In this thesis, we propose a tool-supported workflow using a novel machine-readable migration guide in order to reduce the impact of Web API evolution on its consumers. The structure of the migration guide enables providers to specify breaking changes that they introduced in consecutive versions of their Web API. Different types of change are specified, each assigned to a change object and specific target. Furthermore, every type of change incorporates individual properties that facilitate their migration. Our proposed system uses the information stated in the migration guide and generates a client library with a stable public interface. The client library is integrated as an intermediary between a client application and a Web API. Our proposed system supports generating client libraries in multiple programming languages for various types of Web APIs. Therefore, components and subsystems concerned with the language-specific generation of libraries or the import of different types of IDL documents are designed with strict consideration of extensibility. We validate our approach by implementing Pallidor, an instantiation of our proposed system in the Swift programming language. Pallidor generates a Swift package based on an OpenAPI specification and automatically migrates it according to changes specified in a machine-readable migration guide.

Acknowledgements

Contents

Abstract i

Acknowledgements iii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Objectives . . . 4
- 1.3 Outline . . . 5

2 Related Work 7
- 2.1 API Evolution Patterns . . . 7
- 2.2 API Evolution Client Impact . . . 8
- 2.3 API Evolution Automation . . . 9

3 Requirements Elicitation 11
- 3.1 Requirements . . . 12
  - 3.1.1 Functional Requirements . . . 12
  - 3.1.2 Artifacts . . . 13
  - 3.1.3 Nonfunctional Requirements . . . 15
  - 3.1.4 Constraints . . . 16
- 3.2 Use Cases . . . 16
- 3.3 Scenarios . . . 22

4 System Design 25
- 4.1 Design Goals . . . 26
- 4.2 Subsystem Decomposition . . . 28
- 4.3 Boundary Conditions . . . 29
- 4.4 Hardware-Software Mapping . . . 30

5 Object Design 33
- 5.1 Reuse . . . 34
- 5.2 Interface Specification . . . 37
- 5.3 Restructuring . . . 43
- 5.4 Testing . . . 43

6 CaseStudy / Evaluation 49
- 6.1 Implementation . . . 50
  - 6.1.1 Pallidor Generator . . . 52
  - 6.1.2 Pallidor Migrator . . . 54
  - 6.1.3 Pallidor . . . 57

6.2 Evaluation 59
6.2.1 Automated Migration 59
6.2.2 Semi-Automated Migration 70
6.2.3 Manual Migration 74

7 Summary **77**
7.1 Conclusion 77
7.1.1 Realized Goals 77
7.1.2 Open Goals 77
7.2 Future Work 77

Appendix **87**

ANTLR ANother Tool for Language Recognition

API Application Programming Interface

AST Abstract Syntax Tree

CI/CD Continuous Integration, Continuous Delivery and Continuous Deploymenty

CLI Command-line interface

DSL Domain-specific Language

GoF Gang of Four

GPL General Purpose Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

IDL Interface Description Language

JSON JavaScript Object Notation

REST Representational State Transfer

RPC Remote Procedure Call

SDL Schema Definition Language

SOAP Simple Object Access Protocol

SPM Swift Package Manager

UID Unique Identifier

URI Uniform Resource Identifier

WSDL Web Service Description Language

W3C World Wide Web Consortium

XML Extensible Markup Language

YAML YAML Ain't Markup Language

1 Introduction

Reducing code complexity is a crucial measure in software projects to counteract rising costs for development and maintenance. Nowadays, many applications make use of external components instead of being built completely from scratch. Software reuse increases the productivity of the developers and improves the overall quality of the resulting product. These benefits encourage the popularity of third-party libraries which provide an interface to pretested features, created and maintained by a different team. This public interface of an integrated software component is refered to as its *Application Programming Interface (API)*. Consumers of an API simplify their development effort by taking advantage of the functionality provided without having to worry about its internal implementation, so that they can focus on their own requirements.

With increasing popularity of distributed systems and the web, APIs have evolved from local libraries to globally available services, providing language-independent access to actions and resources on remote machines which require central coordination or storage.The World Wide Web Consortium (W3C) coined the term *Web Service* for services using *Simple Object Access Protocol (SOAP)* messages that trigger *Remote Procedure Calls (RPCs)* and are typically transmitted using HTTP with an XML serialization[1]. Web based services are identified by *Uniform Resource Identifiers (URIs)* in either an action-based or resource-based format. Their functionality is described using an *Interface Description Language (IDL)* like *Web Service Description Language (WSDL)* or the OpenAPI Specification. In the remainder of this thesis, **Web APIs** denote modern, language-agnostic APIs, that are described by IDLs and can be accessed via HTTP and its standard ports by exchanging non-verbose messages in a human-readable or binary format. We define a verbose format as a messaging format that uses additional elements containing meta information to describe the message content instead of implicitly inferring this information by structure or other means. While, contrary to our definition, Web APIs may be provided on ports other than the standard HTTP ports 80 & 443, this configuration is predominantly found in productive systems.

According to ProgrammableWeb[2], one of the world's largest API directories, REST is the predominant architectural style for Web APIs with over 14,000 APIs listed, followed by RPC with about 1,700 entries. Their directory is currently recording an average monthly increase of 168 entries. With new technologies like GraphQL and gRPC emerging, this number will continue to rise. In addition to the consumers, the providers also benefit from supplying an API. They are able to in-

[1] https://www.w3.org/TR/ws-arch
[2] https://www.programmableweb.com/

crease their customer reach and create a new revenue stream by monetizing it [KFJA19, p. 243].

Despite their many advantages, using external components has one major drawback regarding their evolution. Software evolution denotes all changes to software systems after the initial release, which can also include architectural changes [EB18]. Eilertsen et. al also note that API evolution is closely related to software evolution research and is a fairly new field. Due to the decoupled provision and consumption of APIs, the authors refer to the independent evolution process as co-evolution.

Since providers can unilaterally introduce changes to API contracts, these can cause build or runtime errors in consuming applications, commonly referred to as breaking changes. The prevalent way for library consumers to handle them, was by not upgrading to the latest version. In contrast to statically linked libraries, web services can neither be rolled back nor can a specific version be used beyond the end of its support. Instead, consumers now have to adapt their applications by manually going through a cumbersome process which is associated with a lot of additional effort. For providers, the upgrading forced on their consumers is no desirable option either. Especially for public providers, who in this case might lose customers to alternative providers [LZP+19, p.3].

1.1 Motivation

Given the importance of dealing with API evolution, much research has been devoted into this area. Espinha et al. [EZG14] investigated the challenges for client developers regarding API changes and identified how large providers organize their evolutionary process. In [BVXH20], Brito et al. conducted a field study to determine the motivation of API providers to introduce breaking changes. For a better understanding of Web API evolution, Li et al. [LXLZ13] examined large popular Web APIs and defined common characteristics of changes. Given the recurring features of API changes, Lübke et al. [LZP+19] extracted a number of patterns for evolving Web APIs from best practices found in major public services as well as in literature.

Due to the fact that adapting a client application to the latest version of an API is cumbersome, various approaches to automate the migration process have been proposed. However, automation techniques for library evolution such as capturing and replaying refactoring steps [HD05] or using twinning to adapt to alternative APIs [NN10] have been rarely adopted in practice [LXLZ13, p. 300] and cannot be applied to remote Web APIs. Although tools were created to allow developers to parse IDLs to generate client libraries and server stubs in all modern programming languages, this generated code is a static view of an API and does not support migratory adaptions. Hence, every breaking change is directly reflected in the generated library and breaks the client application.

Regardless of whether an IDL generator is used to create a library, the work-

flow remains inconvenient for API consumers. Due to the independent release cycles, API providers update their code and documents without knowing how it might affect their consumers. Breaking changes are not necessarily limited to modifications to the public interface, but can also occur in the event of alterations in internal behavior such as changing standard parameters or return values. As shown in Figure 1.1, API consumers may not be notified of a new release and will only find out about it after their application has malfunctioned.

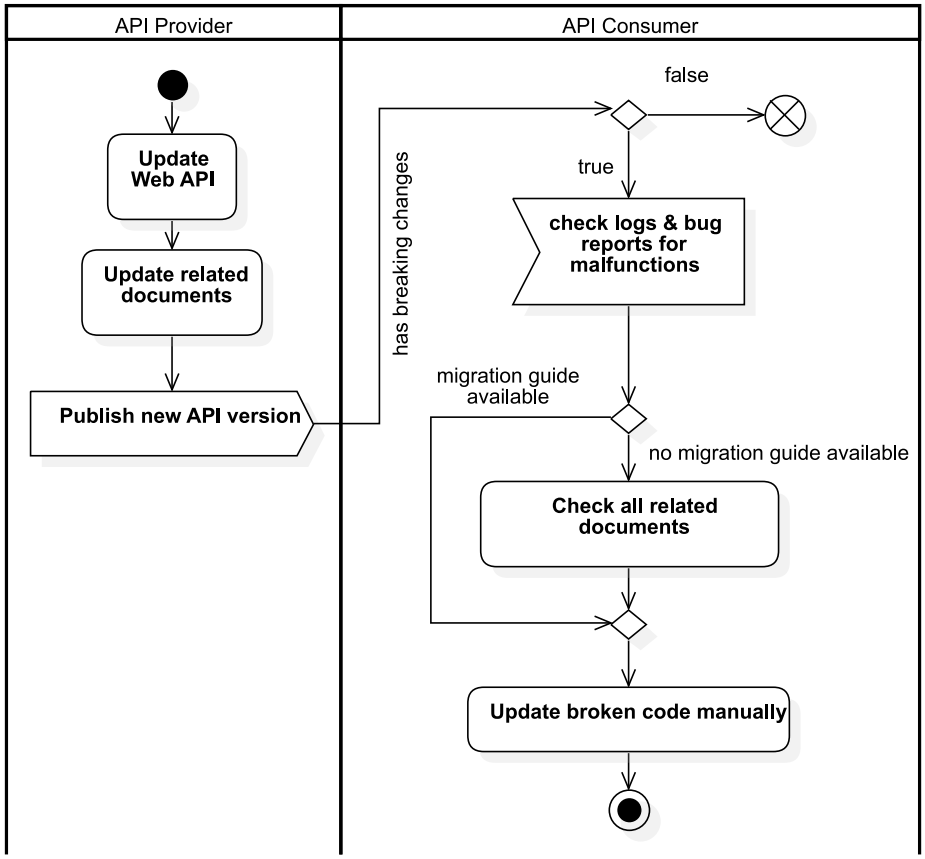

Figure 1.1: Cumbersome workflow after web service changes

In a field study, Brito et. al questioned API providers on how they plan to document their changes. They found out that 22% usually use release notes or changelogs, while only 11% aim to provide a migration guide. Without a detailed guide that includes all of the steps required to migrate between versions, API consumers have to go through various documents and apply the modifications themselves. Resolving breaking changes manually is error-prone and leads to rising maintenance costs. A tool-based workflow could significantly shorten the time from detection to the incorporation of changes and thus reduce maintenance costs and errors.

Recent research on API evolution focuses on classifying breaking changes and patterns for Web APIs or adapting client code after updating a local library. Currently, there is a research gap regarding tool-supported workflows for migrating client applications after a Web API introduced breaking changes. Research and

tools on Web API evolution have the potential to reduce development and maintenance time, improve quality and reliability of client code and prevent manually introduced bugs during the upgrade process. Automated code migration for API consumers would eliminate the need to manually inspect changes listed in change logs, newsletters, release notes or documentations and adapting the code accordingly.

1.2 Objectives

In this thesis, we propose a new tool-supported workflow in which client code gets automatically migrated after a web service was modified. We focus on lightweight Web API technologies which can be described by an IDL, such as REST, GraphQL and gRPC. Our main objective is to facilitate the workflow shown in Figure 1.1 by significantly reducing the effort for API consumers. Therefore, we introduce a state-of-the-art description language to specify a machine-readable migration guide. This guide enables API providers to describe all modifications of a Web API from a previous to its latest version. In addition, we propose a tool that parses our migration guide and generates library code that is persistent to changes to its public interface once it is run for the first time. It can be used in an existing *Continuous Integration, Continuous Delivery and Continuous Deploymenty (CI/CD)* pipeline or via the command line.

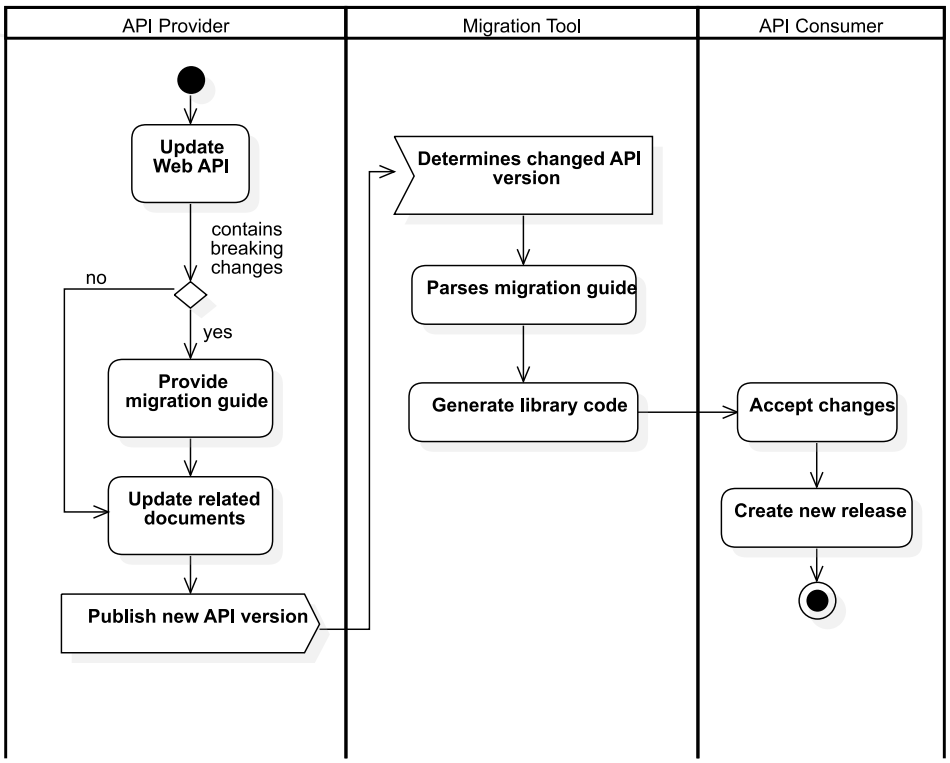

Figure 1.2: Workflow with tool support after web service changes

Figure 1.2 showcases the new workflow. In addition to their current tasks, Web API providers must state all introduced changes in a machine-readable migration guide and make it available to consumers. Web API consumers have to integrate our migration tool, which automates the incorporation of changes without breaking the client application. It determines the API version from the migration guide and creates a set of rules indicating how the library should be customized to avoid introducing breaking changes to the consumer's application code. After executing these rules, the public interface of the generated library code remains unchanged and the consumer's application continues to use the library's code which acts as a facade for the lower level API calls.

1.3 Outline

After the initial introduction into challenges of Web API evolution and our motivation to overcome them in Chapter 1, we examine the current state of related research topics in Chapter 2. Identifying the challenges in related research faciliates defining the key requirements for our proposed workflow and discovers several limitations to be aware of as Web APIs evolve. Chapter 3 comprises all requirements and constraints we identified from related work. In addition, we were able to derive multiple use cases from potential real-world scenarios. Taking into account all requirements and the existing related work, we create the design of our proposed system, which is described in Chapter 4. Our system is composed of multiple subsystems that each provide their own functionality. In Chapter 5 we select detailed solutions for individual subsystems by identifying either suitable third-party components or pattern templates. We instantiate our proposed system in PALLIDOR, our prototypical implementation in the Swift programming language. In Chapter 6, we use Pallidor to migrate various types of change for a sample Web API and evaluate our proposed system based on our findings. Chapter 7 concludes the thesis and discusses future work.

2 Related Work

Our work relates to current research of API evolution, which focusses on classification and patterns for Web API changes. Studies were also conducted to analyze thei impact on client applications for both local libraries and Web APIs. Furthermore, research has been devoted to automating migration of breaking changes in client code for evolving local libraries. Based on insights from previous work and considering the challenges associated with Web API evolution, our research concentrates on reducing the impact of breaking changes in Web APIs on consuming applications by automating the migration process for client developers.

2.1 API Evolution Patterns

Recently, in order to better understand Web API evolution, research has been devoted to classifying similar kinds of changes into patterns and creating guidelines for an efficient development of Web APIs. These studies provide useful hints for determining implementation details of the migration guide specification and creating rules for automating migration steps.

Li et al. [LXLZ13] examined large popular Web APIs and defined common characteristics of changes. Their study identifies 16 change patterns, 12 of them causing compile-time errors if a high-level library is used to wrap messages on HTTP level and four changes causing runtime errors. They found that on average more than 50% of the API was changed with 80% of that breaking changes being refactorings. In addition, they identified six new challenges that arise from Web API evolution compared to local libraries. As their results show, Web APIs are less likely to change at the HTTP level than it is on their corresponding library wrappers. Hence, they suggest providing a tool that automates migration tasks at the HTTP level. These results encourage to automate the migration process, since a large part of the changes are refactorings. Their well-defined patterns for changes help us to determine the requirements for the migration guide and rules.

Exploring the challenges Web API providers are facing, Lübke et al. [LZP+19] extracted eight patterns that can be applied when creating or adapting an evolution strategy for an API. These are derived from best practices found in literature and major public Web APIs [LZP+19]. The authors describe different alternatives for introducing incompatible changes and advocate for using API versioning and description techniques. While one pattern does not support incompatible changes, the others balance forces of stability, potential to innovate, and maintainability for API providers. Although client developers benefit from a well-structured evolu-

tion strategy, it does not reduce the effort required to migrate between versions, especially if the API is changed frequently. By providing a machine-readable migration guide, providers would simplify their customers' workflow while maintaining the innovation potential and maintainability of their API.

2.2 API Evolution Client Impact

Client developers must devote a large extent of their efforts to keep their application compatible with an API in use. Research related to API evolution therefore also includes investigations of the impact on API consumers. Although there are studies related to the impact of changes in Web APIs, most of the studies examine applications using local libraries.

In [BVXH20], Brito et al. conducted a field study to determine the motivation of API providers to introduce breaking changes. They discovered that 39% of these changes have a direct impact on consumers. The authors determined that although API providers state that the migration effort for their consumers is low (86%) or moderate (14%), 45% of API-related questions were asked by client developers who have difficulties overcoming breaking changes. To find out about them, client developers need to examine a variety of documentation options as API developers plan to note their changes in release notes, change logs, source code, website, a migration guide, samples, or in a README file [BVXH20]. In response to these challenges, API providers should focus on documenting changes in a single artifact to ease the effort of migrating for their consumers. Using a machine-readable migration manual ensures that all changes are documented in one artifact and can be easily found by consumers and tools.

Another large-scale study was conducted by Xavier et. al [XBHV17], analyzing the impact of API breaking changes in local libraries. The authors have seen the frequency of changes increase over time, which they associate with library development as the libraries become more complex. Their findings show that 27.99% of all API changes break backward compatibility. Although a median of only 2.54% of clients are potentially affected by these changes, they have identified several outliers which show that in some cases every client application is affected. Based on their findings, they suggest using an impact analysis tool for library developers before changing commonly used APIs. While their study focuses on local libraries, the results apply to modern Web APIs as well. Furthermore, breaking backward compatibility is a more serious problem for client developers because they cannot stick to an older version after a Web API has been updated.

Espinha et al. [EZG14] further investigated the challenges for client developers regarding Web API changes. They found that the impact of changes largely depends on the quality of client application architectures. Hence, the authors prioritize separation of concerns in order to encapsulate components that are subject to change due to API development. According to their findings, the churn percentage for updating these well-designed client applications is 50% lower than their average observations. This results in reduced efforts for maintaining an

API usage, which requires at least 50% of the initial setup effort or even exceeds it [EZG14]. In conclusion, an automated migration process must consider the principle of separation of concerns in order to minimize the effort of integrating non-breaking changes and migrating breaking changes.

2.3 API Evolution Automation

Due to the fact that adapting a client application to the latest version of an API is cumbersome, various approaches to automate the migration process have been proposed. However, automation techniques for library evolution have been rarely adopted in practice [LXLZ13, p. 300]. Li et. al divides the techniques into two categories. Operation-based tools record changes made by API providers and provide a replay function to refactor client code accordingly. The second technique analyzes changes by comparing source code and related documents of two different releases [LXLZ13, p. 306].

One example for an operation-based tool is *CatchUp!* [HD05] which creates a log of refactorings as API providers modify their code. Henkel et al. defined a trace file so that it can also be used for documentation purposes as it is in human-readable XML format and offers the option of creating an HTML document using style sheets. Their tool enables developers to replay this log to upgrade the client application by playing back the log of refactorings using its Eclipse plugin. Documenting changes in a human- and machine-readable format makes debugging easier and allows provider and consumer tools to be developed independently.

Dig et. al [DCMJ06] developed the Eclipse plugin *RefactoringCrawler* that automatically detects refactorings by analyzing the source code between two versions. To increase the accuracy, their algorithm analyzes the semantics of refactor candidates. For example, it detects renaming and changing of method signatures by analyzing the semantics of its call hierarchy [DCMJ06]. They are able to determine seven types of refactorings with an accuracy of 85%, focusing on rename and move modifications.

Current academic literature on tools to automate migration refactorings is devoted to local libraries. While there are industry tools that support generating libraries in multiple languages from service specifications (e.g. OpenAPI Generator[1]) and vice versa, they do not support migrating between different versions of an API. Furthermore, current automation tools do not cover new challenges introduced by Web APIs [LXLZ13, p. 306].

[1] https://openapi-generator.tech/

3 Requirements Elicitation

As described in Chapter 2, recent research has identified several drawbacks to modern Web APIs. Studies have been conducted, showing that their evolution has a large impact on their consumers ([BVXH20], [XBHV17], [EZG14]). As a consequence, client developers face an increased effort to keep their application code compatible, since current tools for the automatic migration of client code ([HD05], [DCMJ06]) are not applicable in a Web API context [LXLZ13]. In order to better understand Web API evolution, recent studies have introduced classifications of several recurring change patterns and evolution strategies that indicate the challenges providers and consumers are facing ([LXLZ13], [LZP+19]).

Addressing these challenges, we propose introducing a tool-supported workflow as shown in Figure 1.2 to significantly reduce the impact of Web API evolution on consumers. Therefore, a majority of migratory tasks needs to be automated according to a machine-readable migration guide which needs to be prepared in advance by providers. All changes are encapsulated in a separate library to be abstracted from client code. This library is added to client applications as a dependency. In addition, our system facilitates the workflow of API providers to release new versions by ensuring that changes to the interface do not affect consumer applications, hence providing perceived stability. The system's main objective is to combine the convenience of local libraries with the flexiblity of modern Web APIs. It aims to be seamlessly integrated into commonly used tools and workflows like continuous integration, version control and dependency management.

Given the current and proposed workflow, multiple functional and nonfunctional requirements can directly be derived. Web API consumers use the system to automatically migrate their application between two versions. Therefore, the system needs to import a machine-readable migration guide that is delivered by Web API providers. Our analysis identified all necessary system requirements which are listed in the following section in detail. Furthermore, use cases are identified for all actors and exemplary scenarios for possible future migration scenarios are presented. All requirements and constraints are listed in section 3.1 while use cases and scenarios are shown in section 3.2 and section 3.3, respectively.

3.1 Requirements

The system consists of two key components: a migration tool and a migration guide specification. This section provides an overview of the requirements and constraints for both of them. The system must meet all of them in order to be accepted. The requirements are split into functional and nonfunctional requirements. Bruegge et. al define functional requirements "[as] the interactions between the system and its environment independent of its implementation" [BD10]. The authors thereby describe users and external systems as part of a system's environment. Furthermore, Bruegge et al. define nonfunctional requirements "as aspects of the system that are not directly related to the functional behavior of the system" [BD10]. They further divide nonfunctional requirements into quality requirements and constraints.

3.1.1 Functional Requirements

In order to enable the implementation of the workflow shown in Figure 1.2, both, the functional requirements of the Web API consumers and providers must be taken into account.

FR1 **Development of a specification for a machine-readable migration guide:** The machine-readable migration guide needs to contain all necessary information regarding changes made to the Web API. Additionally, it must provide fields to specify important meta-data like versioning information. The format of the migration guide must allow the specification of simple and complex data structures. They are required to model hierarchies within the guide or hierarchical types provided by a Web API. Changes stated within the migration guide must be distinguishable from one another by their structure. Depending on its type, a change must include fields that contain information on how to adapt the client code to maintain compatibility with the Web API.

FR2 **Provide configuration option:** A user can configure the system by providing command-line parameters or a configuration file. If applicable, a default value is used in case of a missing configuration. For each API and project, an individual configuration can be applied.

FR3 **Generating a client library from service specification:** A user provides a Web API specification URI as input to generate a client library which can be used by the client application. The input can either be issued locally as a file or can be remotely retrieved from the web service.

FR4 **Provide a facade to maintain API consistency:** A user accesses the library's functionality via a generated facade to ensure a consistent view of the API. It mimics the original interface as it uses the same method signatures and exposes them to the client application.

FR5 **Self-adapting facade for API evolution:** The system automatically resolves

the current version of the web service and its corresponding migration guide. If the retrieved API version differs from the local API version, the system adapts the facade according to the migration guide. As a result, users automatically receive API-related changes in their application code.

FR6 **Integrate system in CI/CD:** The tool will be integrated into the CI pipeline of the client application to ensure the highest level of automation. Alternatively it can be used locally via its *Command-line interface (CLI)*.

FR7 **Integrate output via Git:** The modified library will be published via git pull request into the client application's repository. By publishing it, the library's version number will be incremented according to semantic versioning principles.

FR8 **Specify migration strategy:** A user can specify the migration strategy for non-migratable changes such as deletion of functionality without replacement. The selected strategy can be either provided via command-line parameters or using the configuration file.

FR9 **Specify output language:** A user can specify the programming language in which the target library is generated. The selected output language can be either provided via command-line parameters or using the configuration file.

3.1.2 Artifacts

To illustrate the artifacts from FR1 and FR4, Figure 3.1 shows our proposed structure for a machine-readable migration guide and Figure 3.2 demonstrates the architectural composition of the code generated by our system.

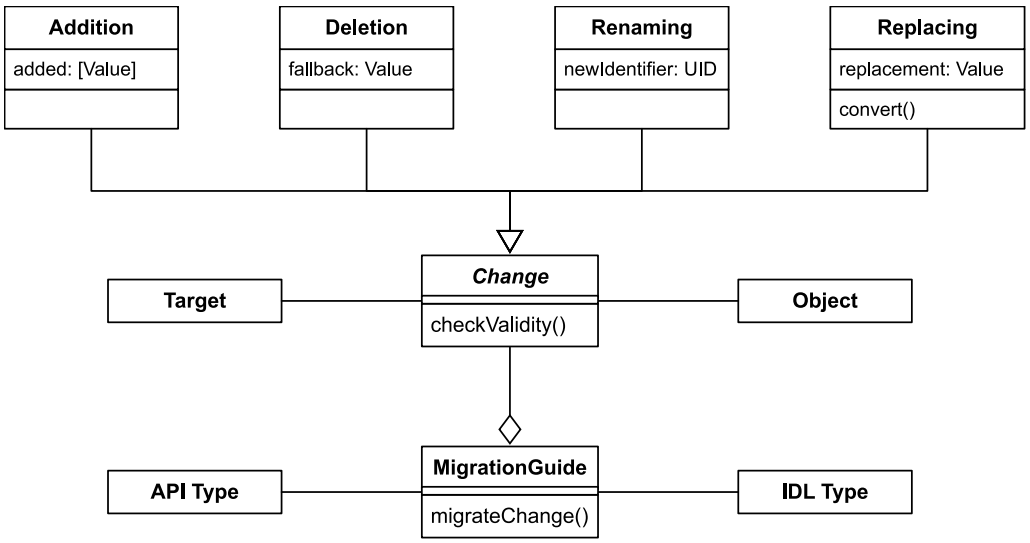

Figure 3.1: Proposed structure of a machine-readable migration guide

A migration guide is responsible for migrating different types of changes that might occur due to Web API evolution. Each change must contain an identifi-

cation of the object which is changed and the specific target of the change. For example, if a method parameter was changed, the method is the change object and the parameter is the change target.

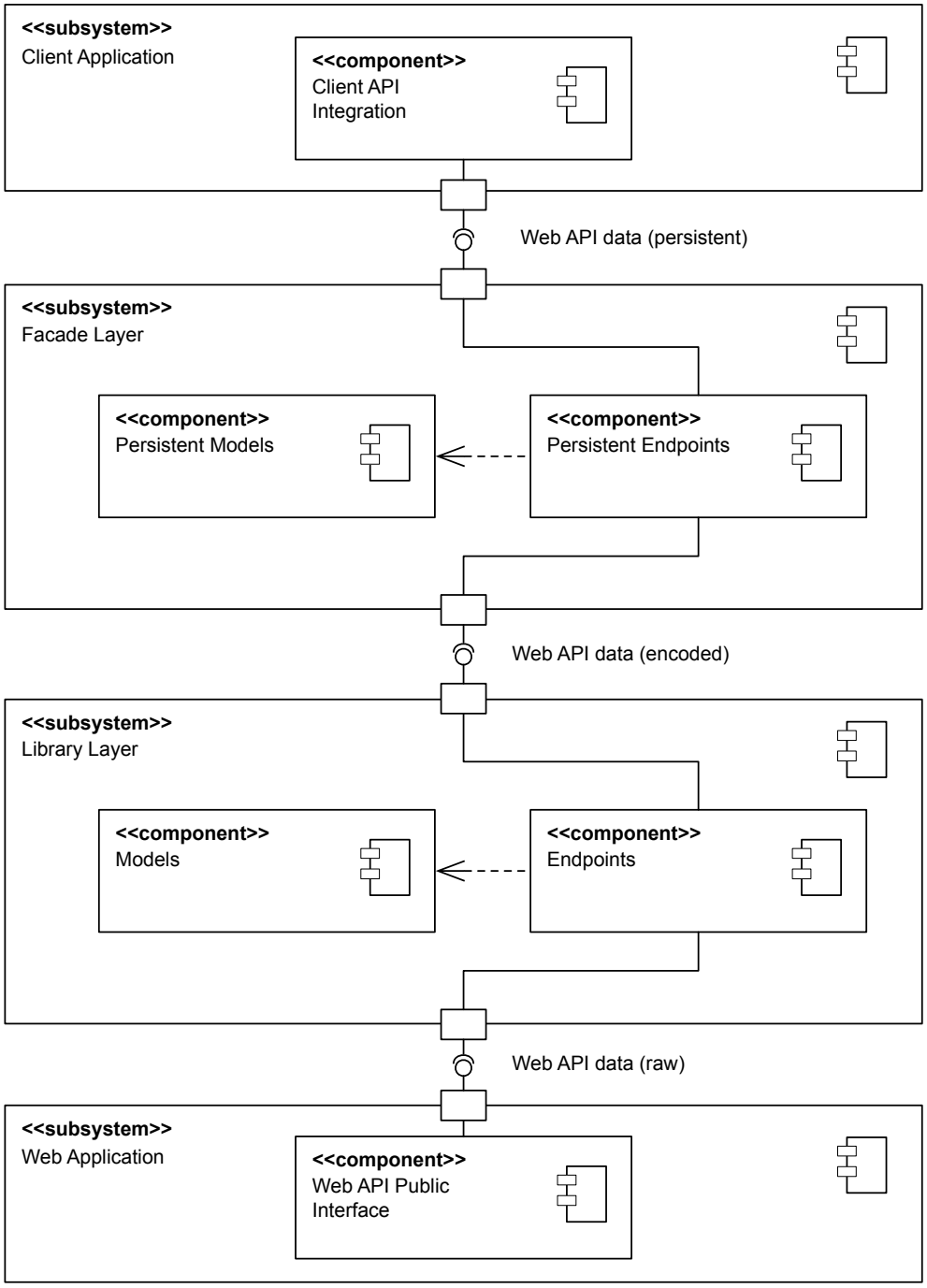

Figure 3.2: Proposed architecture of change-proof Web API integration

The output of our system is designed in a layered architectural style in order to incorporate changes made to the Web API without affecting the client application. Due to the fact that Web APIs are already integrated using a layered network architecture, introducing additional layers does not affect current development practices. The library subsystem is a local representation of the Web API and

provides its services only to the facade subsystem. The facade subsystem uses them to issue HTTP requests to the Web API while maintaining a stable interface to the client application. When changes are introduced to the Web API, the library subsystem gets updated accordingly and the API contract between the library and facade subsystems is broken. For fixing the API contract, all changes stated in the migration guide modify the internal functionality of the facade subsystem without altering any publicly exposed interfaces.

3.1.3 Nonfunctional Requirements

The following requirements were defined to ensure a high quality system. As described in [BD10], each requirement relates to one or more of the categories usability, reliability or supportability. According to the authors, supportability includes adaptability and maintainability.

NFR1 **Code Style:** All code generated by the system must follow a linting ruleset predefined in a configuration file to increase maintainability.

NFR2 **Documentation of facade:** To increase usability for client application developers, the facade's public interface will be documented using code comments. These comments explain the usage of the Web API and are derived from its IDL document.

NFR3 **Documentation of system:** To increase maintainability, the system will be documented using code comments. Documentation will be auto-generated and summarized in the `README.md` file.

NFR4 **Test coverage:** To ensure the system's robustness, unit tests must be provided for all supported migration types. Additionally, unit test must be provided with respect to different migration strategies. Special attention needs to be paid to edge cases.

NFR5 **Coloring of key messages:** Key output messages of the system will be highlighted using coloring schemes to increase usability.

NFR6 **Help messages:** The system provides detailed instructions on how to use its command-line interface. The help will be available via `help` command. This ensures that users are able to easily learn how to operate the system.

NFR7 **Code version:** The system is constrained to issuing language-specific library code in the language's current major version. This improves adaptability when client applications are using the latest version of a programming language.

NFR8 **Dependency Management:** Language-specific implementations are required to use a dependency manager to manage all third party libraries. Using a dependency manager increases maintainability significantly because dependency meta information provides a single source of truth.

NFR9 **Migration guide format:** The migration guide needs to be specified in a

human-readable format which web developers are familiar with. Using a domain-specific language increases acceptance and usability especially for debugging and manual inspection. Additionally, utilizing a format that is already widely used in the software development industry has the advantage that IDEs provide convenient features such as syntax highlighting or auto-completion.

NFR10 **Handling user input:** Missing or incorrect configuration parameters trigger a detailed error message so that the user can easily examine and correct the configuration.

3.1.4 Constraints

For the system to work properly, some constraints have to be fulfilled. A constraint is a nonfunctional requirement that has been specified in advance by external factors and must be reconciled with other affected design decisions [BCK13].

C1 **Legal requirements**: All dependencies must be available under MIT license regulations so that the system can also be published under the MIT license.

C2 **Operations requirements**: An IDL document describing the public interface of an API must be available in order to generate library code. Furthermore, both the API consumers and producers need to follow semantic versioning[1] principles.

C3 **Implementation requirements**: API consumers have to use GitHub as their version control system to use the system as a GitHub action in their CI/CD pipeline. Additionally, it is provided as a command-line program without a graphical user interface. This enables local execution or integration into other CI/CD tools.

3.2 Use Cases

After identifying requirements and constraints to enable an improved workflow for Web API consumers and providers, we derived use cases from their common interactions with our proposed system. Bruegge et. al define Use Cases as "general sequences of events that describe all the possible actions between an actor and the system for a given piece of functionality" [BD10].

For a better illustration we provide a use case diagram for both actors. Figure 3.3 shows the typical use cases of our system from a Web API consumer's perspective.

The migration tool is modeled as a subsystem of the client application due to its integration into the application's CI/CD pipeline. Integrating the tool requires

[1] see https://semver.org/

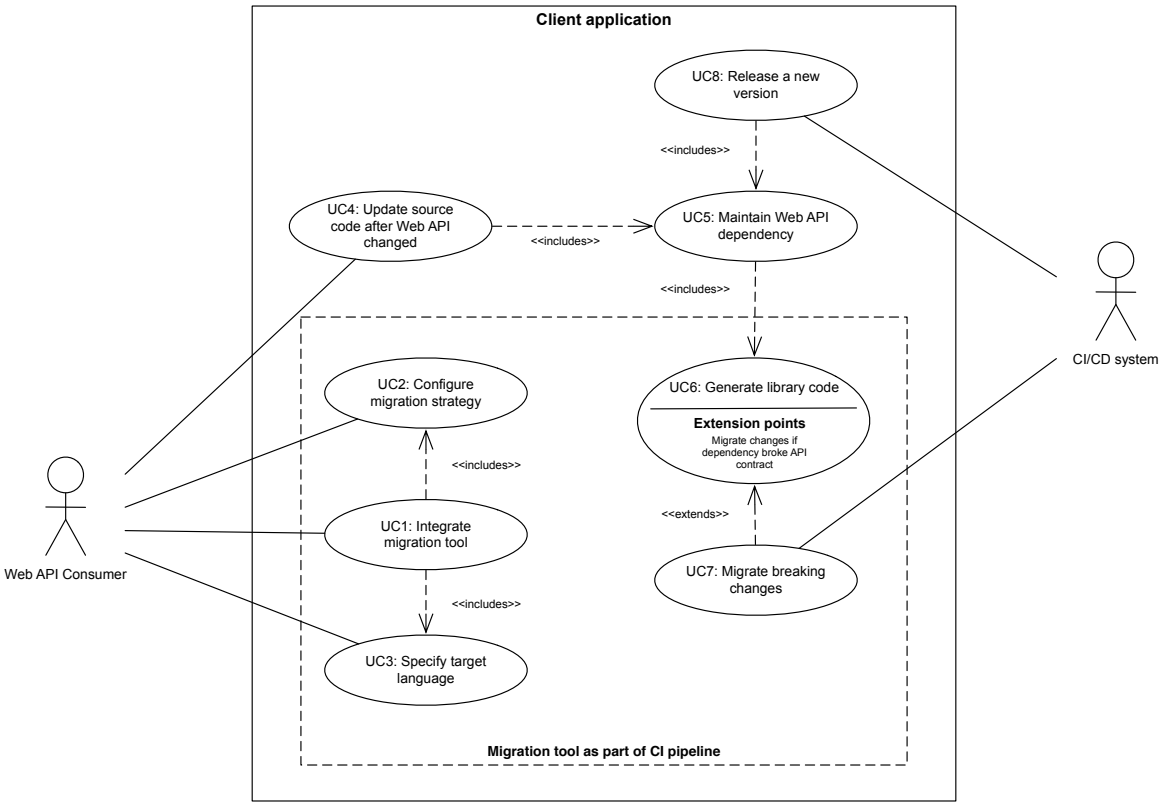

Figure 3.3: Use cases of Web API consumers

Web API consumers to specify the target programming language for the generated library code. Furthermore, a migration strategy can be configured to meet the specific needs of the client project. Although both activities are part of the integration use case, they are modeled explicitly for client projects that do not use a CI/CD pipeline.

Updating source code after a Web API dependency changed is the most important use case for Web API consumers. In addition to generating library code from a Web API's IDL document, our system migrates all changes as stated in its migration guide. This migration guide is created and published by Web API providers. The corresponding use case (`UC11`) is shown in figure 3.4.

The CI/CD system is modeled as a separate actor to illustrate the use cases that do not require manual interaction by client developers. Its primary task is to create a release of a new version of the client application without breaking changes that were introduced in the Web API. Therefore, it automatically migrates all of these changes by using our system.

For further explanation, the two most important use cases for Web API consumers are described in detail below.

Integrate Migration Tool

Source	Web API consumer
Stimulus	Client application needs to incorporate and persist a Web API.
Environment	**Preconditions:** The client application uses a CI/CD pipeline into which the migration tool can be integrated. The interface of the desired Web API is described using an IDL document. **Postcondition:** The resulting library must be added as a dependency to the client application.
Artefact	Migration tool
Response	1. The migration tool gets integrated as part of the CI/CD pipeline of the client application. Therefore, its configuration is either provided as a file via URI or via CLI parameters. Optionally, a code formatting ruleset can be specified via its configuration file URI. 2. The migration tool retrieves the IDL document from its URI. 3. The migration tool parses the information from the IDL document and generates library code encapsulating lower level HTTP calls. This code is for internal use only and cannot be accessed publicly. 4. The migration tool generates a public facade layer to persist the current view on the Web API by abstracting calls to the previously generated library code. 5. The migration tool generates related meta files to facilitate integrating the output via dependency management.
Response Measure	• Error: Execution fails if the IDL document is unavailable or incorrect. • Success: Output compiles and requests/responses to/from the Web API are processed without errors.

Update Source Code After Service Change

Source	Web API consumer
Stimulus	The client application crashes after a modified Web API is published.
Environment	**Preconditions:** • The client application integrates a Web API via library code generated by our migration tool. • Execution is either triggered manually or as part of a CI/CD pipeline of the client application. • The latest release of a Web API introduced breaking changes either to its public interface or behavior. • The Web API provider published a machine-readable migration guide along with an updated IDL document. **Postcondition:** The client application's dependency must be updated to use the latest version of the generated library code.
Artefact	Migration tool
Response	1. The migration tool retrieves the updated IDL document and the migration guide from their URIs. 2. The migration tool checks the changes specified in the migration guide for consistency and correctness. If this check fails, execution is aborted and an error is raised. 3. The migration tool parses the information from the IDL document and generates library code encapsulating lower level HTTP calls. This code is for internal use only and cannot be accessed publicly. 4. The migration tool uses the information from the migration guide to adapt the behavior of the previously generated facade layer without modifying the public interface of the facade. 5. The migration tool generates related meta files to facilitate integrating the output via dependency management.

Response Measure	• Error: Execution fails if the IDL document is unavailable or incorrect. • Error: Execution fails if changes stated in the migration guide are contradicting or incompatible. • Success: Output compiles and requests/responses to/from the Web API are processed without errors.

Web API providers are the second group of actors that participate in our proposed workflow. Figure 3.4 shows the typical use cases of our system from their perspective.

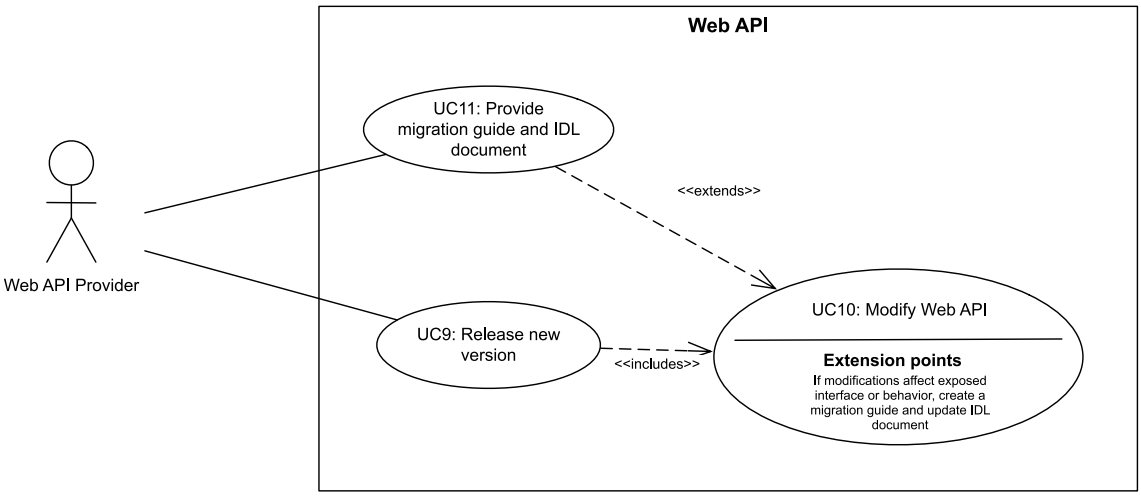

Figure 3.4: Use cases of Web API providers

Web API providers need to decide whether to introduce breaking changes to the publicly available interface or behavior. When breaking changes cannot be avoided, providers must create a machine-readable migration guide that specifies them. Furthermore, they need to provide an updated IDL document describing the current state of the Web API. Since our proposed workflow is only slightly different from their current workflow, a detailed description of a provider's use case focuses on providing a migration guide and an IDL document.

Provide Migration Guide and IDL document

Source	Web API provider
Stimulus	New functionality was added to the Web API or existing behavior was changed. Therefore, a new version of the Web API is released.

Environment	**Preconditions:** The provider introduced breaking changes to the exposed interface or behavior when modifying the Web API. **Postcondition:** The output needs to be published on a publicly accessible server and its URI has to be provided to Web API consumers.
Artefact	Migration guide, IDL document
Response	1. The IDL document must describe the publicly exposed interface and behavior. The Web API provider can either manually create this document or generate it using tools. 2. The migration guide must contain the type of IDL used to describe the Web API. 3. The migration guide must contain the type of Web API. 4. The migration guide must contain both the previous and the current version number according to the semantic versioning principles. 5. The migration guide can contain a short textual summary to provide an overview of the changes introduced in the current version. 6. The migration guide must contain a list of all breaking changes introduced in the current version.
Response Measure	• Success: The migration guide can be retrieved and parsed by migration tool. • Success: The IDL document can be retrieved and parsed by migration tool.

Currently, Web API providers have to manually create a migration guide. We propose that future research should explore automating this task to reduce the effort for providers. Existing tools for automatic generation of IDL documents can serve as a reference here.

3.3 Scenarios

According to Bruegge et. al, scenarios are instances of use cases describing a concrete set of actions [BD10]. In the following section we present three example scenarios for possible uses of our system. They are intended to demonstrate how Web API providers and consumers can use our tool to simplify their respective workflows and meet the challenges of the independent evolution processes.

Migrating an iOS application

Simplified migration process in an iOS application that uses a REST API backend for storing and retrieving notes.

The iOS application "MyNotesApp" published by NoteMe Ltd. uses a REST API backend to store and retrieve the notes of its users. The backend utilizes the OpenAPI specification to describe its publicly exposed interface and behavior. Both systems are developed by individual teams that incorporate our migration tool into their workflow. Both code bases are located in separate repositories on GitHub[2]. After Alice, a developer from the backend team, published a new version of the REST API, she provides two URIs located on the same web server pointing to the updated OpenAPI specification and a migration guide. Since new business demands required her to modify the application's data models, breaking changes were introduced that she specified in the migration guide. Bob, a member of the frontend developer team is aware of these breaking changes, but he cannot apply all changes manually because the frontend team faces staffing shortage. However, our migration tool is part of the CI pipeline as a GitHub action, which automatically migrates all changes and generates updated library code that is submitted via git pull request to its own repository on GitHub. Since the iOS application uses the Swift Package Manager to maintain its dependencies, the updated library code is automatically incorporated after the frontend team accepted the pull request. A new release is created by the CD system and Claire, the tech leader of "MyNotesApp" publishes the app to the Apple AppStore.

Migrating a web application using a third party service

Simplified migration process in a web application that uses a public gRPC-based API provided by Google to create 3D animated GIFs.

The social media platform "Yearbook" is a web application written in TypeScript, that enables users to share the most important milestones of their year. It differs from other platforms in that the number of posts per user is limited to 12 posts per year. SocialViz Inc., the company behind Yearbook, identified that its customers want to use animated GIFs to celebrate their birthdays with their friends and families with an eye-catching post. Creating a GIF is an inconvenient process for their users. Fortunately, Google LLC provides "gifinator", a public service that allows

[2]https://www.github.com/

client applications to create animated 3D GIFs by sending gRPC-based messages with parameters specifying the style of these animations. Lucas, a developer on the Yearbook frontend team, is responsible for integrating this interface. Since Google released the API in a beta version, it is subject to frequent, unannounced changes, namely renaming of public operations. Lucas spends a large share of his working hours adjusting the interface to adapt to these changes while his other tasks remain unfinished. To meet these challenges, he integrates our tool into the CI pipeline of Yearbook. Thereby, all changes are automatically incorporated and abstracted, generating a stable interface that client code can rely on. To make this possible, Google offers a machine-readable migration guide and an updated `.proto` file, that are fetched by our system. Since the generated output is built into the Yearbook as a JavaScript library, updating this dependency can easily be done by Lucas, who publishes it on a web server.

Automating migration documentation for Web API providers

Simplified evolution process for the provider of a GraphQL-based Web API by generating a machine-readable migration guide that can be used by the consumer's tools.

"Unsensored" is an application of the startup Ingenuitees. It collects and analyzes sensor information from industrial machines to predict the need for maintenance activities. After they built a solid customer base and sufficient industry knowledge, they want to expand their services to new potential customers. Therefore, they decided to create a monetized Web API that allows to query analytical information using the GraphQL query language and incorporate the responses in client applications. As they work closely with their customers, Amanda, Ingenuitees 'senior developer, presents their plan to Drew, a developer responsible for integrating third-party services at OnTrack Ltd., one of Ingenuitees' key customers. Although Drew likes this idea, he points out that OnTrack Ltd. has strict requirements for the stability of API contracts. These requirements are at odds with Ingenuitee's entrepreneurial spirit and agile development processes. Therefore, Amanda wants to ensure that new features can be added frequently while deprecated features can be changed without breaking client applications. Drew advises her that he has included our proposed system in their application's CI pipeline to automatically migrate other third-party Web APIs. For this to work, Amanda would have to state every change in a machine-readable migration guide and provide it along with the new version and IDL document. They agree to develop a proof of concept, whereupon Amanda begins to gather information regarding the creation of a migration guide. She finds that by comparing the differences in the schema definitions of two versions of their API, she can develop an algorithm to generate a machine-readable migration guide. She integrates this algorithm as a GitHub action into the CI pipeline of the Web API and publishes the result along with the updated schema definition on a staging web server. Drew uses our system to integrate the Web API of "Unsensored" in their test environment. After verifying that the migrations are done correctly, Amanda publishes the Web API for production.

4 System Design

After analyzing functional and nonfunctional requirements, identifying use cases and possible usage scenarios, this chapter focusses on creating a system design model derived from the specifications of the analysis model. The resulting system design artifact forms the foundation for the subsequent object design analysis.

In order to achieve the best possible quality of our proposed system and its output, the criteria identified from Section 3.1 must be evaluated and balanced with regard to their applicability. Demands that have not been considered up to now as well as contradicting requirements must be incorporated or resolved. Therefore, the design goals of our system are defined in Section 4.1 and prioritized to maximizie the possible benefit for all stakeholders.

Considering all functionality and quality criteria defined during requirements analysis results in a complex system that is difficult to develop and maintain as a whole. In order to distribute the workload among development teams and to facilitate training of new team members, subsystems need to be identified that support a clear separation of concerns and can be maintained independently. We identified a composition of subsystems in which each subsystem provides well-defined interfaces to its encapsulated internal functionality. Section 4.2 contains the details of the interactions between subsystems and gives an overview of the modular architecture of our proposed system.

Our system is supposed to be executed as an additional step during the build stage in an existing CI/CD pipeline. While it is designed to run automatically, various boundary conditions require manual intervention of the user in order to resolve errors in the configuration or during runtime. Integrating the system into an existing CI/CD pipeline and troubleshooting for occuring errors is described in detail in Section 4.3.

In Section 4.4 an exemplatory execution environment is used to demonstrate the flexibility of the subsystem decomposition. Therefore, its abstract components are mapped to representational subsystems that could be applied in this execution environment. In addition, all modified services and resulting artifacts are described in detail.

4.1 Design Goals

The main purpose of our system is to facilitate the evolution process of Web APIs. Therefore, it must meet several requirements in order to be of real benefit to all actors involved. Some of these requirements are not complementary but contradict each other. In order to achieve the best possible result for our users, we prioritize our design decisions in the following section.

Dependability Criteria Decisions about the dependability of the system concern the effort to be invested in minimizing system crashes and their effects on users [BD10]. Integrating our system into the CI/CD pipeline of an application means that subsequent process steps rely on its error-free execution. Emphasis is therefore on Test Coverage (NFR4) which implies creating extensive tests of the migration logic, especially with regard to edge cases. In addition, incorrect user input must be handled and detailed error messages must be provided so that users can identify and correct errors themselves. By prioritizing robustness and fault tolerance, the delivery time for new functionality in our system, such as additional language or API type support, is significantly extended.

Furthermore, client applications rely on an error-free execution of the code emitted by our system. Therefore, the resulting library code needs to handle internal errors appropriately and forward any response emitted by the Web API. As it is designed to mirror the behavior of the Web API, also any errors that are issued by the Web API are returned to the requesting method within client code. Hence, client developers are responsible for handling errors appropriately and designing their application for failures.

Performance Criteria Achieving high performance measures is not an essential requirement of our system as it runs as a background process in an application's CI/CD pipeline or in a user's local terminal. While users do not depend on fast response times or a high throughput, the memory usage of our system increases with the size of the IDL document or migration manual. Therefore, when designing the system architecture, care should be taken that memory-intensive components are encapsulated as this provides the possibility to replace them with more efficient algorithms if necessary. In terms of the library code generated, our design choices Providing a facade to maintain API consistency (FR4) and Self-adapting facade for API evolution (FR5) result in decreased performance for client applications as API requests and responses must be processed across multiple layers.

Maintenance Criteria According to Bruegge et. al, maintenance design decisions determine the difficulty of changing a system after its initial deployment [BD10]. Well-defined subsystems that enable a clear separation of concerns ensure that our system maintains a high degree of extensibility and modifiability.

In particular, components concerned with the language-specific generation of libraries or the import of different types of IDLs must be designed with strict consideration of extensibility. This is due to the fact that supporting more languages and types of APIs is critical to increasing our system's adoption and adding additional value to its users. The encapsulation of language or API-specific functionality also improves the modifiability of our system by limiting changes due to external influences to the corresponding component.

Focussing on expandability and modifiability involves a trade-off in terms of portability and adaptability. Adapting the system to another application domain, e.g. managing API evolution for providers is not intended. Furthermore, the support of different types of language-specific library generation for several Web API styles increases the complexity of the system, which means that the effort for new maintainers to understand the system is increased. Porting it to other platforms which implies using a different host language becomes more complicated.

Although we do not consider any cost criteria in this thesis, it is important to note that our maintenance criteria together with an extensive testing result in longer delivery times with significantly higher costs for development and maintenance. Using the output of our system, on the other hand, reduces development costs and delivery times of client applications, as switching between different API types and programming languages is frictionless, an according implementation of our system implied. In order to be able to use the generated code for a different application domain, programming language or API type, our system just requires the corresponding parameters and does not need any adaptation of its functionality.

End User Criteria Reducing the manual overhead of evolution for Web API consumers is the main goal of our proposed system. Automating the workflow for migrating client applications adds an additional task for Web API providers. By requiring them to specify all changes in a migration guide for each release, the effort of consumers is shifted to the provider side. This design decision was made based on the assumption that future research can concentrate on automating the creation of a migration guide for Web API providers. Related research as shown in section 2.3 demonstrates how capturing and replaying or comparing techniques can be used to realize possible scenarios like scenario 3.3. In addition, the number of consumers of public Web APIs is exceeding their providers, resulting in an overall reduction in maintenance effort.

To maximize the positive impact of the system, special attention is paid to its usability for Web API consumers. Nonfunctional requirements were specified for coloring of key output messages (NFR5), documentation of generated library code (NFR2) and provision of help messages (NFR6) to ensure a high degree of usability. All of the code emitted by our system is documented by code comments that explain how to use the Web API thus facilitating learning for users.

4.2 Subsystem Decomposition

By dividing our system into smaller subsystems, the workload can be distributed among development teams and components that are already available to developers can be reused. The subsystems were identified by analyzing the flow of events shown in Figure 3.3 and 3.4. Each subsystem has precisely defined interfaces that are provided to its consumers and thus enable a modular architecture in which subsystems can easily be replaced by more efficient alternatives. Furthermore, our design enables implementations that support importing multiple IDL types and generating output for multiple programming languages at the same time.

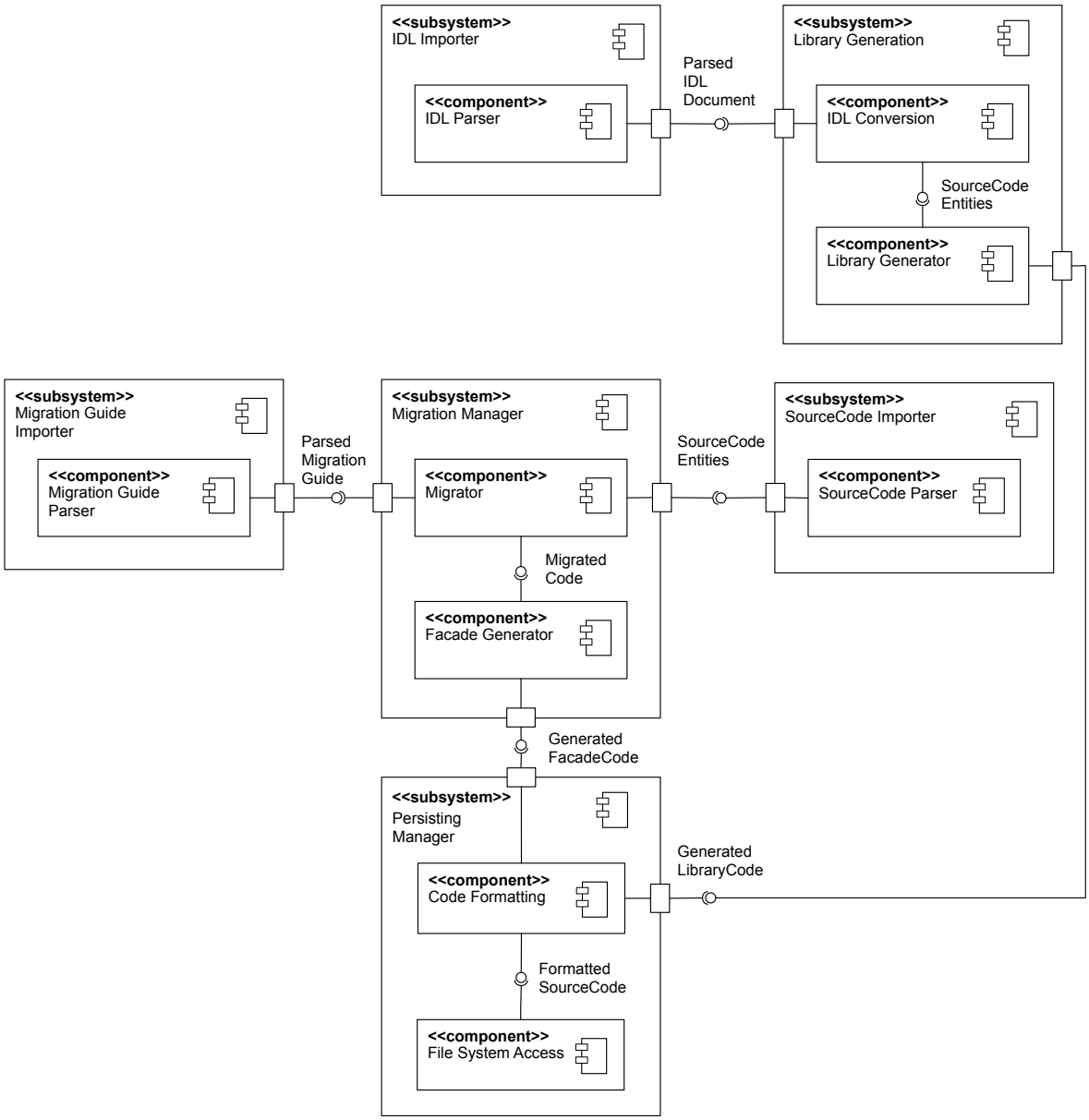

Figure 4.1: Subsystem decomposition

For creating the library and persistent facade subsystem as shown in Figure 3.2,

various tasks need to be executed in order to generate them from IDL documents, previous versions of them and a machine-readable migration guide.

In order to generate the library subsystem an IDL document must be imported and converted into the source code of a specific programming language. Therefore, a dedicated `IDL Importer` subsystem for reading and parsing of an IDL document supplies its encoded representation to the `Library Generation` subsystem. This converts the encoded IDL document into a textual representation of source code of a selected programming language.

Creating the persistent facade subsystem requires the interaction of several resources that have to be imported first. Our migration guide as shown in Figure 3.1 includes any changes made to the Web API between the previous and current versions. The corresponding `Migration Guide Importer` subsystem handles all of the tasks related to fetching, parsing, and validating a migration guide. It provides the parsed representation of the document to the `Migration` subsystem. For some types of changes, missing information is too cumbersome to be defined in a migration guide and cannot be implicitly derived from an IDL document. Therefore, the source code of the previous version of the facade subsystem must be analyzed to obtain this information. The `SourceCode Importer` subsystem reads and parses all files of the previous facade and library to provide the encoded result to the `Migration` subsystem. The `Migrator` component uses the received information to adapt the previous version of the facade using the changes as stated in the migration guide. Its output is used by the `Facade Generator` component to generate a textual representation of source code in the desired programming language.

The emitted source code from both subsystems, `Migration` and `Library Generation`, is used by the `Persisting Manager` subsystem which formats the code according to a user-defined linting rule set. The formatted source code is written to file using the `File System Access` component.

4.3 Boundary Conditions

When using our proposed system, various boundary conditions regarding integration and execution must be taken into account. It is designed to be integrated into an existing CI/CD pipeline of a client application but can also be run locally via a CLI. When it is used as a step in an existing CI/CD pipeline, it must be part of a stage prior to releasing in order to ensure that the most recent version of the Web API is used. Client developers must specify multiple configuration parameters to setup the system. While configuring a migration strategy and specifying a target programming language are mandatory tasks, users can optionally define a linting rule set which determines the code formatting of the emitted code.

On its initial run, our proposed system does not require a migration guide as the latest version of the Web API is used as a starting point for generating the persistent facade. Hence, only the URI of the IDL document needs to be present. After

that, each run requires specifying the URIs of both, the IDL document and the migration guide in order to adapt the facade and update the library subsystems. Omitting mandatory parameters results in an error message during runtime that highlights the missing configuration. Optional parameters do not trigger error messages as a default configuration is used instead. Detecting missing configuration parameters and specifying error messages are tasks of the `Executable` subsystem.

In addition to configuration parameters, unmet preconditions also raise errors. If the system fails to fetch either the IDL document or the migration guide, no output can be generated and execution will be aborted. The specific reason for the failure is given in the CI/CD system's log or on the command line. Furthermore, if changes specified in the migration guide are incompatible or inconsistent, the facade adaptation will fail and an error message will be logged. Removing a parameter from a non-parameterized method or adding and removing the same parameter to and from a method at the same time are examples of incompatible or inconsistent changes, respectively. Errors related to fetching and parsing of an IDL document or migration guide are managed by the `IDL Importer` subsystem and `Migration Guide Importer` subsystem. Checking for incompatibilities and inconsistencies within the migration guide is done by the `Migrator` component.

4.4 Hardware-Software Mapping

For better understanding, in this section the abstract components as shown in Figure 4.1 are mapped to concrete components in an exemplatory execution environment. Therefore, the target client environment is defined as the Apple ecosystem, particularly the development of iOS, iPadOS or macOS-based client applications. A common use case for this environment is the integration of resource-based Web APIs in the REST architectural style that are described by the OpenAPI IDL. Swift is used as the target programming language in which to output the result and to import previous versions of the facade code. By restricting to these criteria, the subsystem decomposition now contains more specific components as shown in Figure 4.2.

The Web API is built using the Representational State Transfer (REST) architectural style. It is served by a `nodeJS` web server hosted on a virtual machine maintained by the Web API provider. Its functionality is described using the OpenAPI IDL. This document is published on a separate URI of the Web API. Additionally, a `Migration Management` subsystem is used to track all changes that have been introduced to the Web API. It notes all changes in a machine-readable migration guide and publishes it on a separate URI. Both document can be retrieved using `HTTPS`.

To import the specification document, the OpenAPI Importer subsystem must be able to parse specifications using both, the *JavaScript Object Notation (JSON)* and *YAML Ain't Markup Language (YAML)* documentation styles. While the system

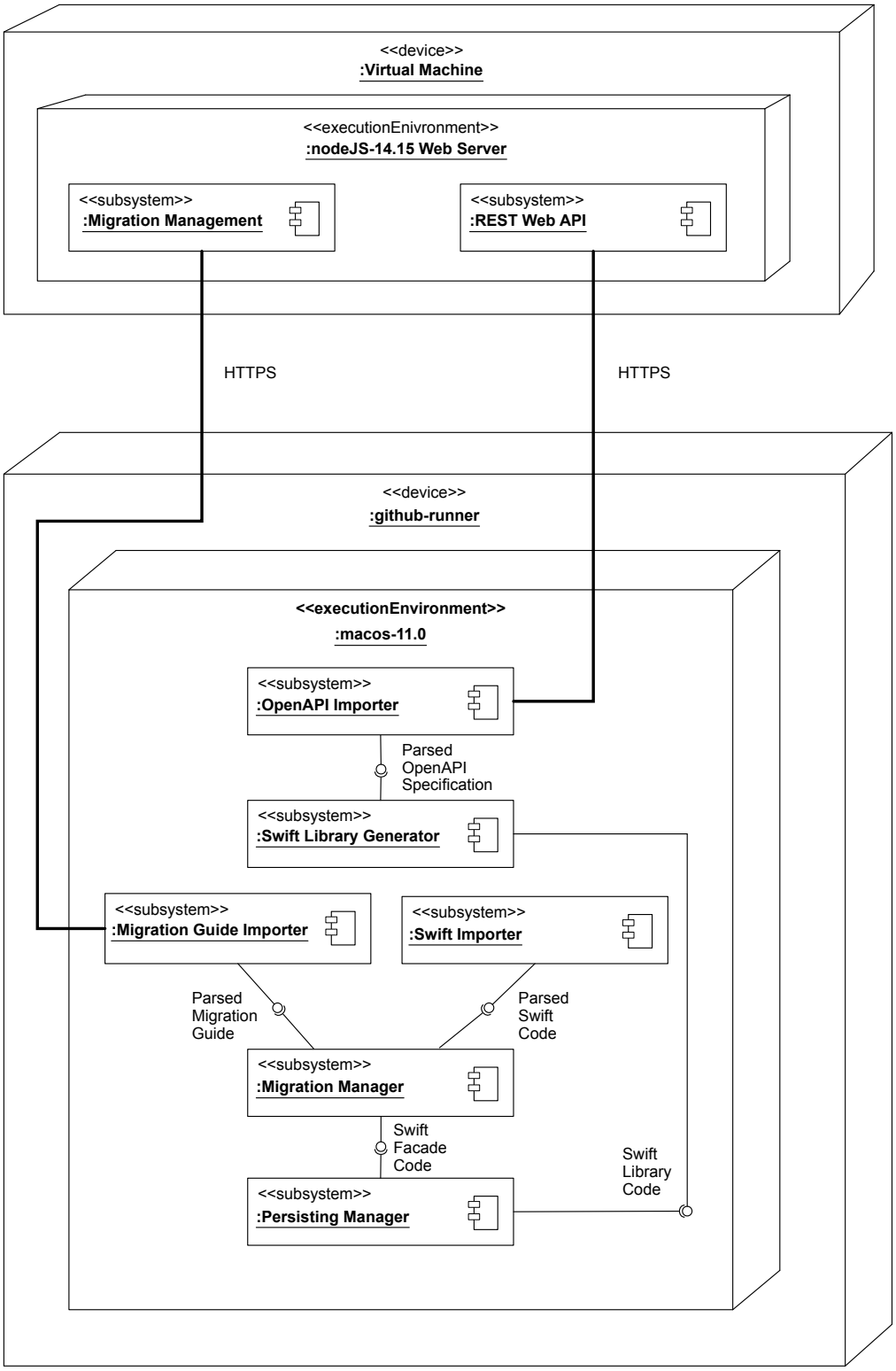

Figure 4.2: Subsystem decomposition in an Apple ecosystem

itself can be implemented using an arbitrary programming language, every subsystem that is concerned with importing or generating source code must support Swift. The `Swift Library Generation` subsystem uses the parsed specifica-

tion to create Swift files containing model and endpoint definitions that enable the interaction with the Web API. Appropriate tooling to generate a library from an OpenAPI specification is available for all major programming languages. The OpenAPI Generator[1], implemented using Java, is one of them, supporting 65 programming languages and dialects. It combines the functionality of the `OpenAPI Importer` and `Swift Library Generation` subsystems.

Parsing a migration guide requires a custom implementation of the `Migration Guide Importer` as no reusable components exist yet. The subsystem responsible for importing the previous facade must support parsing Swift code. The internal behavior of it is modified according to the changes as stated in the encoded migration guide. After that, the adapted facade is transformed into the textual representation of Swift-based source code by the `Swift Facade Generator`. There are multiple tools available that support generating Swift code using Stencil templates. The most popular ones are SwiftGen[2] and Sourcery[3]. They provide similar features that enable users to generate Swift code based on several template types by executing the compiled application via CLI. Sourcery additionally offers a well-maintained framework to enable an integration into existing Swift applications.

The `Persisting Manager` subsystem uses a linting ruleset to format the library and facade code according to its configuration, before adding meta-files and creating a Swift package which can be imported by client applications using the *Swift Package Manager (SPM)*. Formatting code based on a linting ruleset is one feature of SwiftLint[4], the most popular linter for Swift code. While it can be installed and used via CLI, it also supports an integration as a Swift package that enables developers to lint and format Swift code stored in its textual representation.

[1] https://openapi-generator.tech/
[2] https://github.com/SwiftGen/SwiftGen
[3] https://github.com/krzysztofzablocki/Sourcery
[4] https://github.com/realm/SwiftLint

5 Object Design

During object design, we refine the system design artifact of Chapter 4 to detail the specifications for each subsystem. Therefore, we define the subsystem interfaces and identify off-the-shelf components that can be reused to provide the functionality of a subsystem. Furthermore, design patterns are selected to solve common problems and ensure future extensibility of the subsystems in which they are used. The subsystem decomposition model in Figure 4.1 is restructured to reflect the changes we introduced in defining the interfaces of the subsystems to improve the maintainability of our system. Finally, we detail our test strategy, which ensures that all functional and nonfunctional requirements are met. For that, we describe the inputs and expected outputs for each subsystem and type of test to be performed.

In Section 5.1, we identify and describe off-the-shelf components and design patterns, that can be used to realize our subsystems. Since our system does not prescribe using a specific programming language, we consider multiple off-the-shelf components for each subsystem. Section 5.2 describes the specification of our subsystems' interfaces by specifying their operations and important properties. Furthermore, we define the structure of the machine-readable migration guide in detail. The class diagrams reflect the design patterns used in each case. By specifying the interfaces, potential improvements of our system model are revealed. In Section 5.3 all modifications are explained and the updated diagram of our subsystem decomposition is provided. Section 5.4 outlines our test strategy and details the tests to be performed as well as their corresponding inputs and expected outcomes. While our unit tests are designed to detect faults of individual subsystems and components, our integration tests define an appropriate interaction of multiple components. Our system tests are derived from the functional requirements and are used to validate the functionality of the overall system.

The resulting object design model can be divided into modules that can be implemented by individual developers. It provides the foundation for an instantiation of our system, which is described and used in Chapter 6 to evaluate our approach.

5.1 Reuse

Software reuse increases the productivity of development teams and improves the overall quality of the product. Therefore, our proposed system defines several components that can be reused from existing solutions. All external components must be open-sourced under the MIT license in order to fulfill our legal requirements (C1). Furthermore, custom components of subsystems can be implemented by reusing pattern solutions as defined by the Gang of Four (GoF).

Tooling to parse an IDL specification and generate library code for various programming languages is available for many modern Web APIs. For REST-based Web APIs, the OpenAPI Generator[1] is an open-source tool that can be integrated in existing Java projects via Maven/Gradle plugins, or used via CLI and an online service[2]. It supports parsing of OpenAPI IDLs regardless of whether they were encoded to YAML or JSON. Additionally, it generates library code in 65 programming languages and dialects. Google's gRPC[3] technology uses protocol buffers as an IDL and as a format for exchanging messages. Parsing protocol buffers and generating library code is currently available for 16 programming languages and dialects, with support for additional languages upcoming. Emerging GraphQL[4] APIs are described by the Schema Definition Language (SDL) for which different tools exist that are able to generate library code from it. On the one hand, the GraphQL code generator[5] supports five programming languages and offers several configuration options to further customize the emitted code. Quicktype[6], on the other hand, supports 19 programming languages and similar configuration options. One limitation of all tools above is their Integratability as a framework. They are designed to generate library code persisted in files, which prevents developers from accessing intermediate artifacts like the decoded IDL document. Generating source code in a programming language that is not supported by those tools requires forking the codebase to extend their functionality. However, for some IDLs and programming languages new frameworks are published that can be integrated into existing projects to get access to intermediate artifacts. For example, the OpenAPIKit[7] is an open-source project that offers a Swift library to en-/decode OpenAPI IDL documents. By separating the parsing of an IDL document and the subsequent library generation, the extensibility for new programming languages and IDL types is considerably increased.

Supporting multiple programming languages is also a challenge for the `Source-Code Importer` subsystem. It must support parsing source code of a programming language into its Abstract Syntax Tree (AST). Tools are available for most programming languages as many features of Integrated Development Environments (IDEs), such as syntax highlighting or auto-completion rely on them. How-

[1] https://openapi-generator.tech/
[2] http://api.openapi-generator.tech/index.html
[3] https://grpc.io/
[4] https://graphql.org/
[5] https://graphql-code-generator.com/
[6] https://quicktype.io/
[7] https://github.com/mattpolzin/OpenAPIKit

ever, most of these tools are commercial products and are not available under an open-source licence. One example is the JavaParser[8] which is available as a Maven and Gradle plugin for parsing, analyzing and generating Java code. Other frameworks were created through the active contribution of open-source communities. Acorn[9] is a parser for JavaScript code that supports plugins for JavaScript dialects and frameworks like React JSX. Sourcery[10] is a Swift library that provides an API that offers parsing and generating Swift code based on Stencil templates. All of these frameworks are limited to parsing source code of a single programming language. To support many different languages simultaneously, custom lexers and parsers must be specified and implemented for each language. While this is a cumbersome manual effort, tools like ANother Tool for Language Recognition (ANTLR)[11] enable developers to generate lexers and parsers in Java, C#, C++, JavaScript, Python, Swift and Go. Therefore, users have to provide a grammar of the language that describes its structure. This grammar is defined using the Antlr4 grammar syntax. The application's repository contains over 200 predefined grammars that cover many modern programming languages.

Our system uses a machine-readable migration guide to collect all changes that were introduced between subsequent versions of a Web API. In order to fulfill NFR9 (Migration Guide Format), it should be using a Domain-specific Language (DSL) that is in a human-readable format which is prevalent in the context of web development. While an implementation in a custom language using the auxiliary features of ANTLR would be conceivable, we advocate the use of JSON as an external, compositional DSL. This decision is based on the fact that JSON is not just easy to understand and learn, but is also an established standard that is widely adopted by web developers. Another key advantage comes from existing utility libraries for parsing JSON documents in many programming languages. As they are either part of core libraries or can be integrated using third-party open-source frameworks, the need to use a custom built lexer and parser is eliminated. Listing 5.1 shows an example JSON-based migration guide that contains one change and specifies some meta-information.

```
{
  "summary" :"This is an examplatory migration guide.",
  "api-spec": "OpenAPI",
  "from-version" :"0.0.1b",
  "to-version" :"0.0.2",
  "changes" :[
    {
      "reason": "Default value was added.",
      "object" :{
        "operation-id" :"findPetsByStatus",
        "defined-in" :"/pet"
      },
```

[8] https://javaparser.org/
[9] https://github.com/acornjs/acorn
[10] https://github.com/krzysztofzablocki/Sourcery
[11] https://www.antlr.org/

```
      "target" :"Content-Body",
      "added" :[
        {
          "name" :"_",
          "type" :"Pet",
          "default-value" :"{ 'type' : 'Cat', age: 2 }"
        }
      ]
    }
  ]
}
```

Listing 5.1: Examplatory migration guide in JSON format

The decoded representation of a migration guide is used by the `Migration Manager` subsystem to adapt previous facade code to the changes it contains. Although there are no off-the-shelf components that can be reused, patterns have been identified that are applicaple here. The behavior of the `Migration` component depends solely on the changes in the migration guide. Therefore, an alternative computational model is to be preferred in contrast to the common imperative models, in which code is organized in an object-oriented manner [FP11]. Fowler et. al coined the term *Semantic Model* to define models that are populated by a DSL. The authors further define an *Adaptive Model* as a specific variation of a Semantic Model in that they take the primary behavioral role in a system. According to the authors, Adaptive Models enable developers to shift the execution context from compile time to runtime and thus change behavior of a running system without a recompile. As the defined behavior is implicit, their implementation is difficult to understand and maintain [FP11]. The ability to alter the migration behavior during runtime by specifying changes in an external migration guide outweighs the negative aspects of using this pattern to model the `Migration` component.

Before the generated source code emitted by the `Library Generation` and `Facade Generation` subsystems is persisted, users are able to specify formatting rules in order to fulfill NFR1 (Code Style). Therefore, multiple open-source tools have been created for every modern programming language. SwiftFormat[12] is an open-source tool that formats Swift code using a predefined ruleset. It can be used either via CLI or by integrating its framework into an existing Swift application. Prettier[13] is a code formatter for multiple languages and dialects used in web development. It supports formatting of JavaScript, JSX, and TypeScript. Additionally, its active community provides plugins for even more languages like Java, Swift, and Ruby. Prettier can be integrated in IDEs or CI environments and provides an API to be used within JavaScript applications.

All subsystem that deal with importing or generating different types of programming languages or IDL documents are required to adapt their behavior based on the needs of the user. User preferences are provided by configuration options

[12]https://github.com/apple/swift-format
[13]https://prettier.io/

and need to be applied during runtime. The *Strategy* pattern [Gam95] is a behavioral design pattern that enables selecting these preferences without the need to recompile the application [BD10]. In our proposed system we identified four subsystems that benefit from applying this pattern.

As shown in Figure 3.2, our system emits library code that distinguishes between different layers that serve their own purpose. The library layer issues requests and receives responses directly from the Web API. It offers an interface to the facade layer that adapts to changes within the library layer without altering its own public interface. Client applications are unaware of the internal behavior of the facade layer and the interfaces of the library layer. The *Facade* pattern [Gam95] used here is a software design pattern that enables masking of internal behavior and any changes made to it.

5.2 Interface Specification

Reusable components and patterns used to create custom components offer their functionality via public-facing interfaces. A detailed specification of these interfaces is required to enable an integration of all subsystems. Additionally, each subsystem defines artifacts that they produce for and consume from other components. The following section demonstrates the internal structure of custom components while for reused third-party subsystems their integration and emitted artifacts are illustrated.

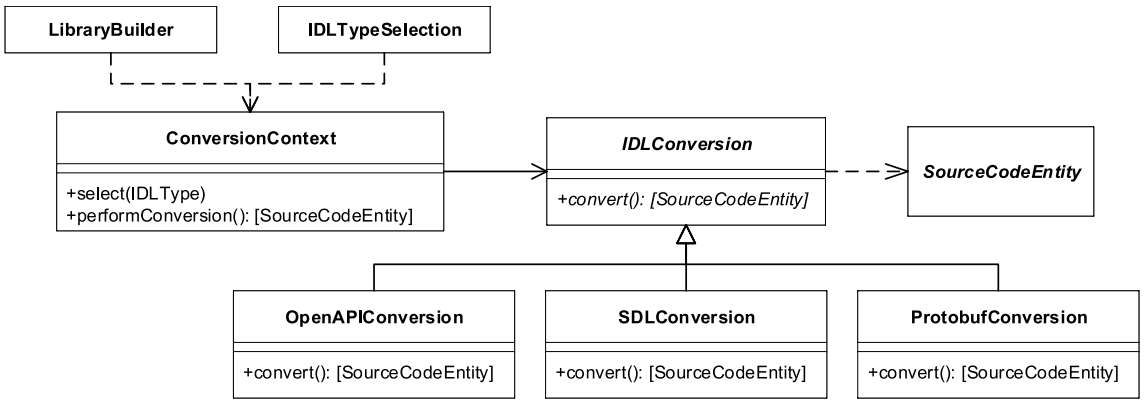

Figure 5.1: Class diagram of IDL importer subsystem

Importing an IDL for a specific type of Web API needs to flexibly switch between different implementations for various kinds of IDLs. Therefore, the strategy pattern is used to abstract the internal implementation from the consuming client code. The `LibraryBuilder` is the client that uses the `performConversion` method. Depending on the type of IDL which was set by the `IDLTypeSelection` for the `ConversionContext`, the appropriate implementation of `IDLConversion` performs the conversion and returns a list of `SourceCodeEntity` objects.

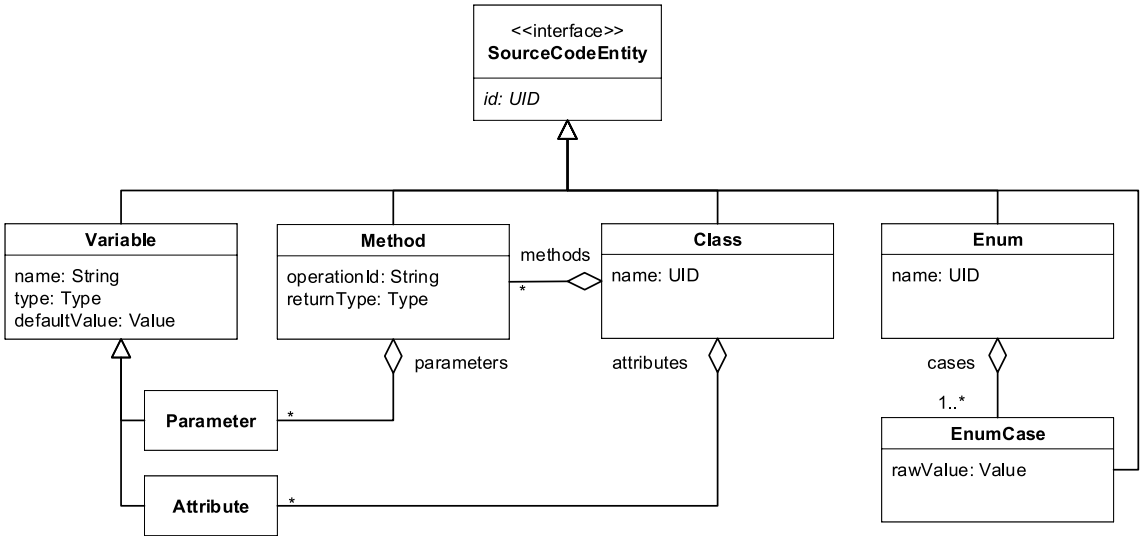

Figure 5.2: Class diagram of source code entity artifact

`SourceCodeEntity` objects are used by multiple components. They represent the basic syntactical elements that are common in all modern programming languages. Figure 5.2 shows their composition and their most important attributes and relations. `Class`, `Method` and `Enum` entities represent structual elements in which other elements can be nested. A `Class` element contains `Attributes` and `Methods`, whereas `Enums` specify at least one `EnumCase`. Each `SourceCodeEntity` can be identified by a Unique Identifier (UID). For `Class` and `Enum` elements, their name itself is unique, while for `Variables`, `Methods` and `EnumCases` their context needs to be taken into account.

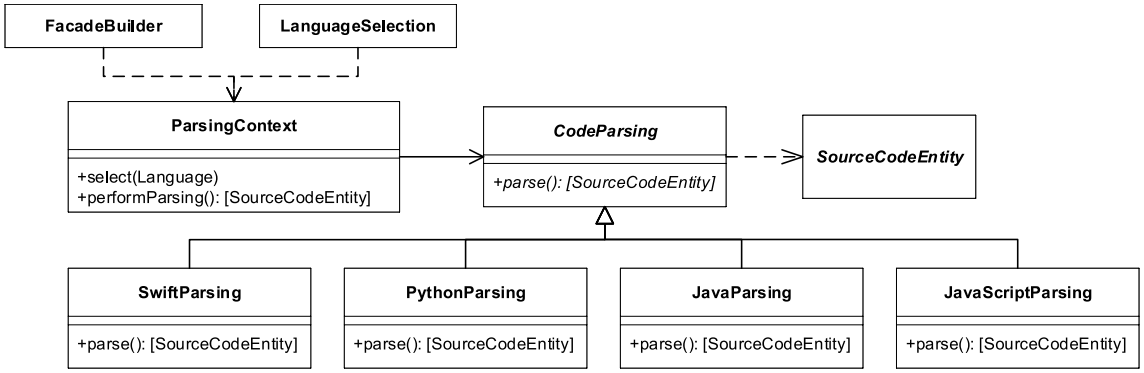

Figure 5.3: Class diagram of code importer subsystem

Adapting a previous facade to recent changes of a Web API requires importing its source code. Therefore, the `SourceCode Importer` subsystem must be able to parse source code of multiple programming languages. The decision which programming language needs to be parsed is made during runtime by the user. Since the requirements are the same as for importing different types of IDLs, the strategy design pattern is also used here. The selection of the programming language to parse is done by the `LanguageSelection` component while the `FacadeBuilder` requests the parsed `SourceCodeEntity` objects.

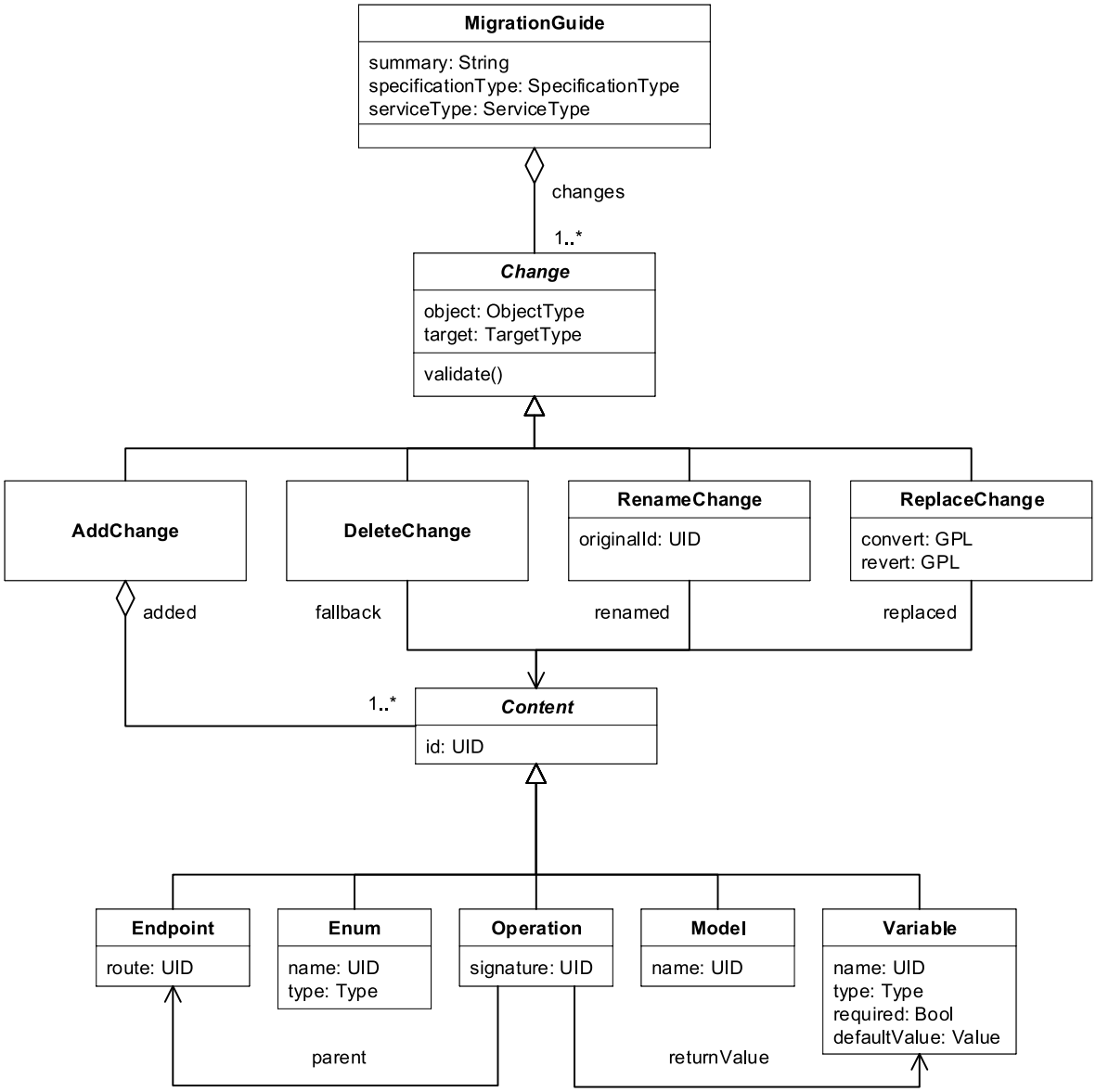

Figure 5.4: Class diagram of migration guide artifact

In addition to the decoded representation of the previous facade code, the migration guide needs to be parsed, too. Based on our decision to use JSON for defining a migration guide, it can be parsed using a language's core libraries or third-party components. The decoded migration guide structure is shown in Figure 5.4.

A migration guide contains the type of IDL specification in which the Web API is described as well as its service type. Additionally, it specifies all changes that were introduced by the latest version of the Web API. Every change is associated with one of four change types. `AddChanges` denote, that one or more elements of a Web API were added to it. This includes for example adding attributes to a model, parameters to an operation or operations to an endpoint. `DeleteChanges` indicate that an element was removed from the Web API. For migratable deletions such as removing an attribute of a model, a fallback value can be

specified which is used instead of the removed value. `RenameChanges` are used to automate refactorings. Since the functionality of a renamed element remains unchanged, only the previous and new identifications have to be specified here.

Elements that were renamed and whose functionality also changed are specified with `ReplaceChanges`. They are the most complex type of change as they need to contain not only the replacement element, but additionally instructions on how to revert them to their previous version and convert their previous to their current version. Code stated in a `ReplaceChange` needs to be converted into statements of the programming language specified by the user. Instead of introducing another DSL or extending the migration guide DSL, we decided to use a General Purpose Language (GPL) that is common to web developers. We chose JavaScript for that purpose, as it is one of the most used programming language by web developers and it complements JSON well. Furthermore, JavaScript code can be directly executed by an interpreter without previously compiling it. Various programming languages support interpreting JavaScript code. For example, Microsoft provides Windows Script Engines[14] for C# and Apple released the framework JavaScriptCore[15] for Swift and Objective-C in order to parse and interpret JavaScript at runtime .

Each type of change is applied to one object of the Web API and targets one property of it. For example, adding a parameter to an operation results in an `AddChange` for a `Operation` object type with a `Parameter` target type while renaming a model results in a `RenameChange` for a `Model` object type with a `Signature` target type. All changes reference `Content`, i.e. the unique identification of an element of the Web API including additional information that supports the migration process. Invalid changes are indicated by an incorrect combination of their internal information. They are identified by their validation method which reports any errors to the user.

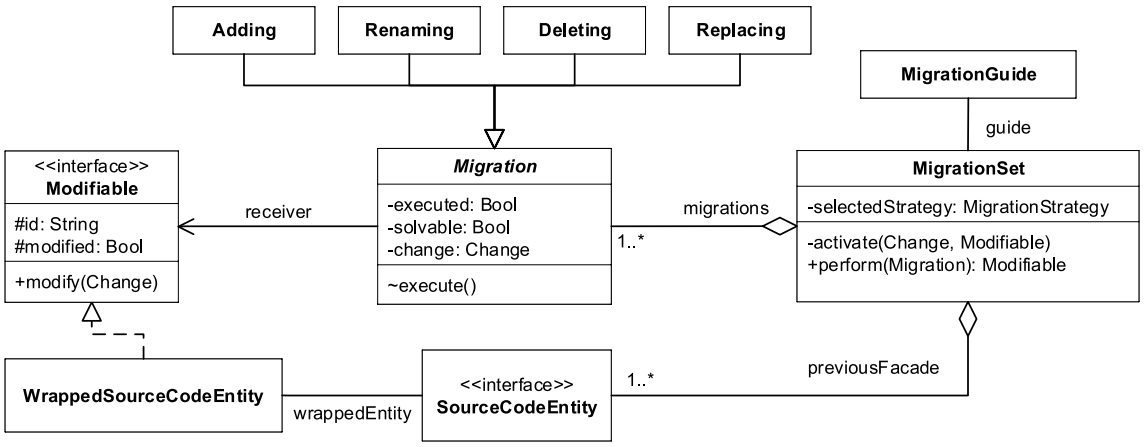

Figure 5.5: Class diagram of migration manager subsystem

For performing the migration steps, the `Migration Manager` subsystem in-

[14]https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/windows-scripting/windows-script-engines
[15]https://developer.apple.com/documentation/javascriptcore

corporates the imported migration guide and decoded previous facade. Analogous to the types of changes, corresponding types of migration are defined. `Migrations` contain the change that must be migrated as well as information that indicates whether they are solvable and have been executed. Furthermore, they reference a `Modifiable` object on which the migration is performed. The `Modifiable` interface is implemented by a wrapper component that acts as an adapter for modifying a `SourceCodeEntity` of the previous facade. By executing a `Migration`, the `modify` method of a `Modifiable` is triggered which adapts the `SourceCodeEntity` according to the type of migration. The `modified` property of a `Modifiable` denotes that this object has already been migrated. All migrations as well as the previous facade are composed in the `MigrationSet`. By analyzing the changes stated in the migration guide and their target entities, it creates the corresponding migrations by invoking the `activate` method. Unsupported or incorrect combinations result in an unsolvable migration. In addition, the user can select a migration strategy to control the migration under certain conditions. The selected migration strategy thereby alters the adaption of source code entities or prevents migrating certain types of changes. Breaking changes that are not migrated automatically must be manually adapted by client developers to fix their application. The `MigrationSet` is the public interface of the subsystem by providing the `perform` method to perform a migration and return the adapted facade element.

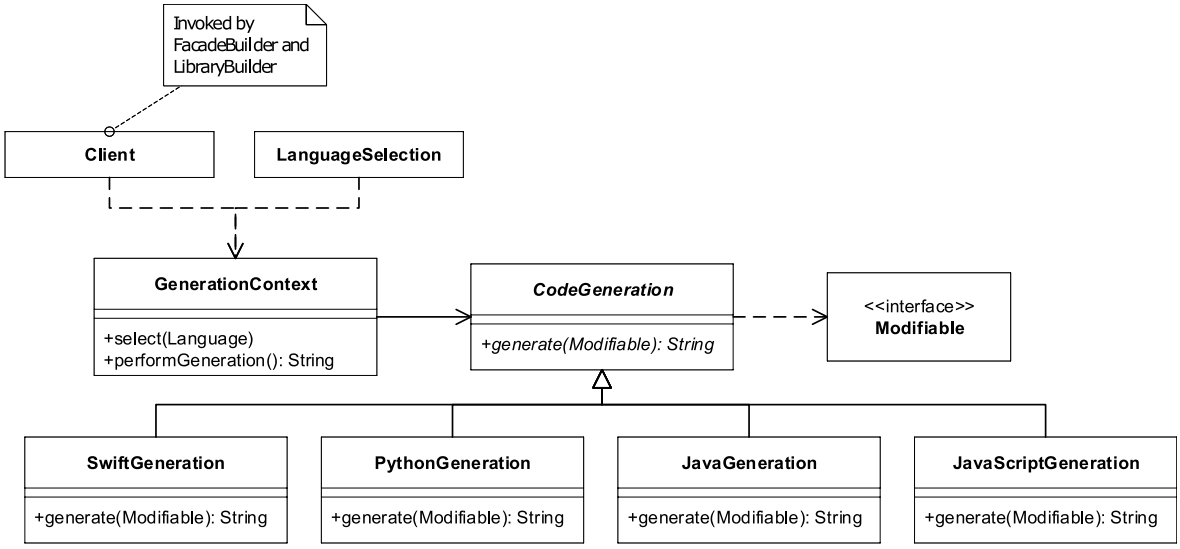

Figure 5.6: Class diagram of code generator subsystem

Generating source code from decoded entities for various programming languages requires the `Code Generator` subsystem to use the strategy pattern. Thereby, it can select the correct syntax elements during runtime. In contrast to the `Source-Code Importer` subsystem, it takes `Modifiable` objects, i.e. wrapped `Source-CodeEntity` objects, as input and generates a string of compilable source code in the programming language the user specified. Its service is used by the subsystems concerned with generating the library code for an IDL and generating the migrated facade code.

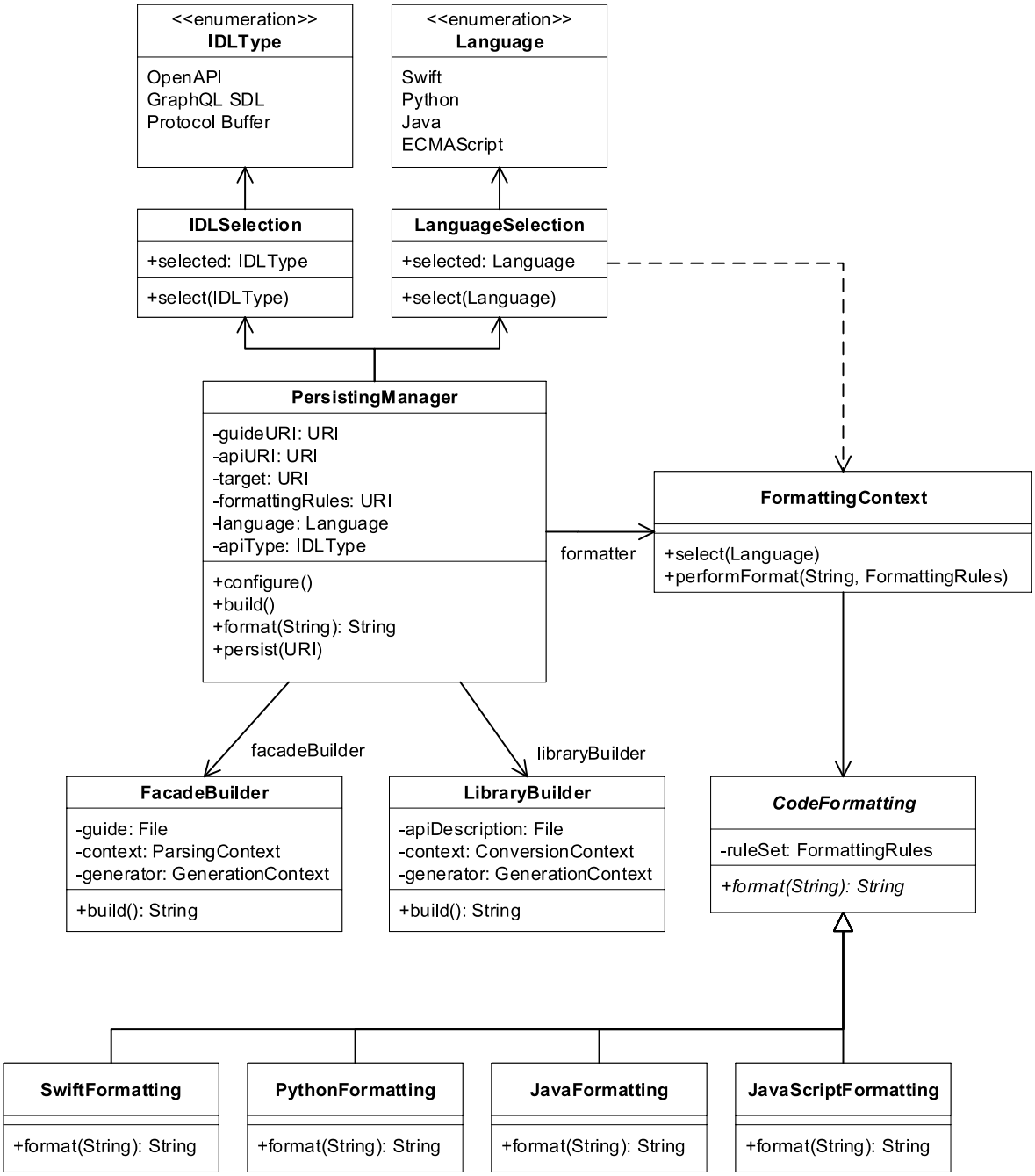

Figure 5.7: Class diagram of persisting manager subsystem

The process is orchestrated by the `Persisting Manager` subsystem. It takes the user input and selects the desired type of IDL and programming language. The selected IDL type results in the initialization of the corresponding concrete strategy of the `IDL Importer` subsystem. The user's choice of programming language determines the concrete strategy of the `SourceCode Importer`, `Code Generation` and `Code Formatting` subsystems. Additionally, the `Persisting Manager` contains all configuration options of the user that are necessary to retrieve the Web API's IDL and migration guide. Furthermore, it contains the ruleset specifying the formatting style of the source code. After configuring the sys-

tem according to the user's input, the `build` method starts the importing, adapting and generating process. Therefore, the `LibraryBuilder` and `FacadeBuilder` objects are instantiated and their `build` method is executed. Both objects hold references to their subsystems that are responsible for generating library code from an imported IDL and generating source code of an adapted facade, respectively. Once it receives the generated source code strings, the `PersistingManager` formats the code using a `CodeFormatting` component instantiated according to the currently selected programming language. As this is done at runtime, the strategy pattern is also applied here. Formatting source code is available by using third-party components for many programming languages, e.g. SwiftFormat[16] for Swift and Prettier[17] for JavaScript und Java. The formatted source code is then persisted by invoking the `persist` method. Although the internal structure of the emitted library is defined by our system, the user is able to specify its identifier and its target location.

5.3 Restructuring

After specifying the interfaces of all components, the subsystem decomposition needs to be restructured to reflect all changes. Regarding the generation of code in various programming languages, we identified that the `Migration Manager` and `Library Generation` subsystems require the same functionality. As a result, a `Code Generation` subsystem was extracted that provides its service to the `Migrator` component and the `IDL Conversion` subsystem. By that no redundant functionality has to be implemented.

Additionally, we decided to use JavaScript for handling convert and revert operations defined in `ReplaceChanges`. Therefore, the `SourceCode Importer` subsystem must support parsing it. The restructured subsystem decomposition is illustrated in Figure 5.8.

5.4 Testing

Testing ensures that all requirements defined in Section 3.1 and use cases detailed in Section 3.2 are fulfilled. Therefore, unit tests are specified that test all components of our subsystems in isolation. Furthermore, our integration tests demonstrate the faultless interaction between our subsystems. System tests are defined to ensure that the fully integrated application produces the correct output for various inputs. The testing strategy of our proposed system is subsequently described in a structured manner according to these types of tests.

[16] https://github.com/apple/swift-format
[17] https://prettier.io/

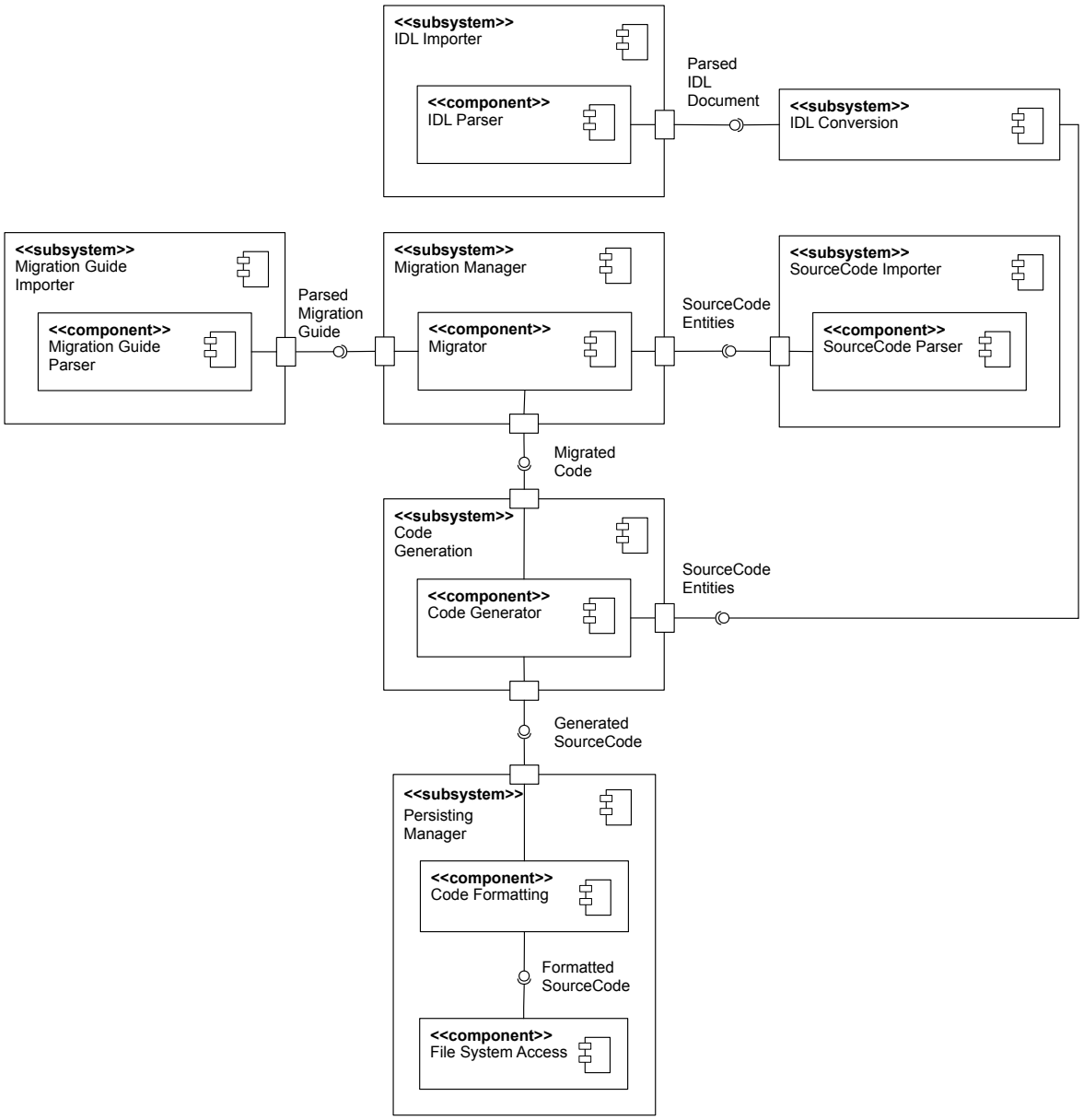

Figure 5.8: Restructured subsystem decomposition - The `Code Generation` subsystem was extracted from the `Migration Manager` and `Library Generation` subsystems to provide a uniform interface to both subsystems

Unit testing Unit tests are used to find faults in subsystems with respect to use cases from use case models [BD10]. Any subsystem for which a custom implementation is preferred over using a third-party component requires extensive testing.

Testing `UC1`, all subsystems affected by integrating our system are tested in isolation. Various IDL documents must be used as an input for the `IDL Importer` subsystem which determines their validity and provides the parsed document. While correct IDL specifications are processed without interruption, invalid or

incomplete IDL documents raise an error message in the component which is forwarded to the user. Test cases need to be defined for each type of IDL that is supported by the system. For testing the `IDL Conversion` subsystem, examplatory parsed entites of IDL documents are used to find faults in the conversion algorithm. Its expected output is manually defined and validated based on the emitted source code entities. When the system is set up for the first time, a new facade must be created without migrated changes. Hence, the `SourceCode Importer` targets the generated library code in order to create source code entities. Testing this process requires examplatory source code files in various programming languages that are validated based on previously defined source code entities. Empty or invalid files raise a parsing exception. Producing the textual representation of source code needs to be validated by testing the algorithm of the `Code Generation` subsystem for each supported programming language. Therefore, example collections of source code entities are used as an input that are transformed into strings of source code in the respective language. The unformatted source code strings are formatted by the `Persisting Manager` subsystem which is tested by validating the formatted source code based on predefined examples.

Subsystem/ Component	**Input**	**Output**
IDL Importer	Invalid or incomplete IDL document	Error message
IDL Importer	Valid IDL document	Parsed IDL document
IDL Conversion	Parsed IDL document	Source code entities
SourceCode Importer	Compilable source code files	Source code entities
SourceCode Importer	Uncompilable source code files	Error message
SourceCode Importer	Source code entities	Unformatted, compilable source code strings
Code Formatting	Unformatted, compilable source code strings	Formatted source code strings

Table 5.1: Input and output of subsystem unit tests for UC1

Testing the automated migration is required to detect faults during the upgrade of client source code after the Web API changed (`UC4`). In addition to the ones defined for `UC1`, test cases need to be specified that cover importing a machine-readable migration guide and the adaption of the facade to the changes it con-

tains. Since the migration guide is structured using JSON, unit tests for the `Migration Guide Importer` subsystem are performed by the developers of the third-party component. Migrating a previous facade must focus all types of migrations. Additionally, test cases need to be defined for combining different types of changes targeting the same object. Testing this subsystem requires providing example migration guides and manually specified source code entities of a previous facade as an input. Validation is performed by matching the migrated source code entities to predefined end-results. By creating unsolvable `Migrations`, edge cases can be simulated that result in error messages which need to be forwarded to the user. For example, adding a new parameter to a model of the Web API is unsupported and raises an error message. The message contains information about the type of change, the target object, and a reason for the fault. Incomplete `Migrations`, i.e. `Migrations` that do not contain a change, must not be created and result in an unsolvable `Migration`.

Subsystem/ Component	**Input**	**Output**
Migration Manager	Parsed migration guide with changes of multiple target objects Source code entities	Migrated source code entities
Migration Manager	Parsed migration guide with multiple changes of a single target object Source code entities	Migrated source code entities
Migration Manager	Parsed migration guide with invalid changes Source code entities	Unsolvable Migration object

Table 5.2: Additional input and output of subsystem unit tests for UC4

Integration testing The subsequent tests are used to find faults by testing individual components in combination [BD10]. Therefore, the public interfaces of the individual components are used to provide the input for the consuming components. Integration tests must be specified for all subsystems, including subsystems that use third-party components.

Identifying faults in the process of generating library code from an IDL document requires testing the `IDL Importer`, `Library Generation` and `Code Generation` subsystems. Various IDL documents are provided as an input. The

results are validated by comparing the generated source code with predefined test examples in multiple programming languages. IDL documents of different types that describe the same Web API must produce the same source code, depending on the programming language.

For testing the migration process, the interaction of the `Migration Guide Importer`, `Migration Manager`, `SourceCode Importer` and `Code Generation` subsystems must be analyzed. Source code files of an examplatory facade and a corresponding migration guide file are mandatory inputs. For identifying any faults, predefined examples for multiple programming languages are compared against the generated output. By combining different changes, some of them targeting the same object, errors caused by contradicting or incompatible information can be detected.

Subsystem/ Component	**Input**	**Output**
IDL Importer *Library Generation* *Code Generation*	Valid IDL document	Unformatted, compilable source code strings
Migration Guide Importer *Migration Manager* *SourceCode Importer* *Code Generation*	Compilable source code files Migration guide with multiple changes of multiple target objects	Unformatted, compilable source code strings

Table 5.3: Input and output of subsystem integration tests

System testing System testing tests all the components together, seen as a single system to identify faults regarding functionality, performance issues and user related problems [BD10].

For testing the systems functionality, its command-line interface is used to set different configuration options and produce a library for accessing a Web API. The system works as defined if the emitted code compiles and is able to communicate with the Web API. Error messages issued by the Web API, e.g. in the event of unauthorized access, do not represent a system failure. When integrated into the CI/CD pipeline of a client application, the system uses a preset configuration to produce the library code every time its stage is executed.

Performance tests ensure, that all nonfunctional requirements and design goals are fulfilled [BD10]. Missing mandatory configuration parameters must result in error messages displayed by the CLI. The system provides a parameter to list all parameters and other helpful advice. Important messages must be colored to clearly emphasize their information. The generated library code must be formatted according to the user's ruleset and contains documentation derived from the IDL document. All code emitted in a specific programming language must use the syntax elements of its latest major version.

6 CaseStudy / Evaluation

Evaluating our proposed system demonstrates its usefulness in a potential target environment. For the concrete instantiation, the Swift programming language was selected to implement custom subsystems. Required functionality that was already available is integrated using third-party Swift packages. All used packages are published under MIT license. Since it is a prototypical implementation, it supports importing IDLs only for REST-based Web APIs that are described using the OpenAPI specification. Furthermore, its support for generating client libraries is limited to the Swift programming language.

Section 6.1 addresses the details of the implementation of PALLIDOR which is composed of three libraries that are developed and maintained as standalone Swift packages. Each package is described in detail in its respective section, which provides an overview of the package's control flow and public interface. Furthermore, each section details how the package can be integrated into existing applications and shows its current limitations.

For evaluating our proposed system, we perform a migration of two consecutive Web API versions using Pallidor. Therefore, we manually manipulate the OpenAPI specification of the sample `Pet Store`[1] Web API so that it reflects all change types we identified. Furthermore, we manually create a machine-readable migration guide for each version of the OpenAPI specification to document all occuring changes.

The evaluation in Section 6.2 is structured according to the degree of automation to which the respective type of change can be migrated. Changes of the first category, AUTOMATED MIGRATION, can be automatically migrated by using Pallidor with our novel machine-readable migration guide. The second category, SEMI-AUTOMATED MIGRATION, contains changes that can be migrated by Pallidor, but which also require manual intervention by client developers to overcome them. Changes that require new approaches to automating their migration are addressed in the third category, MANUAL MIGRATION. Each category describes the problems that are caused by the types of change as well as our approach to solving these problems including its benefits and consequences. Each category is concluded by providing our evaluation process and results.

[1] https://petstore3.swagger.io/

6.1 Implementation

For evaluating the proposed system, we created *Pallidor* a prototypical implementation using the Swift programming language. It is composed of multiple subsystems that are developed and maintained as standalone Swift packages. Each package provides the functionality of a subsystem that is defined in Section 4. `PallidorGenerator` implements the `IDL Conversion` and `IDL Importer` subsystems. Migrating a previously generated facade is performed by the `PallidorMigrator` package that provides the services of the `Migration Manager`,`Migration Guide Importer` and `SourceCode Importer` subsystems.

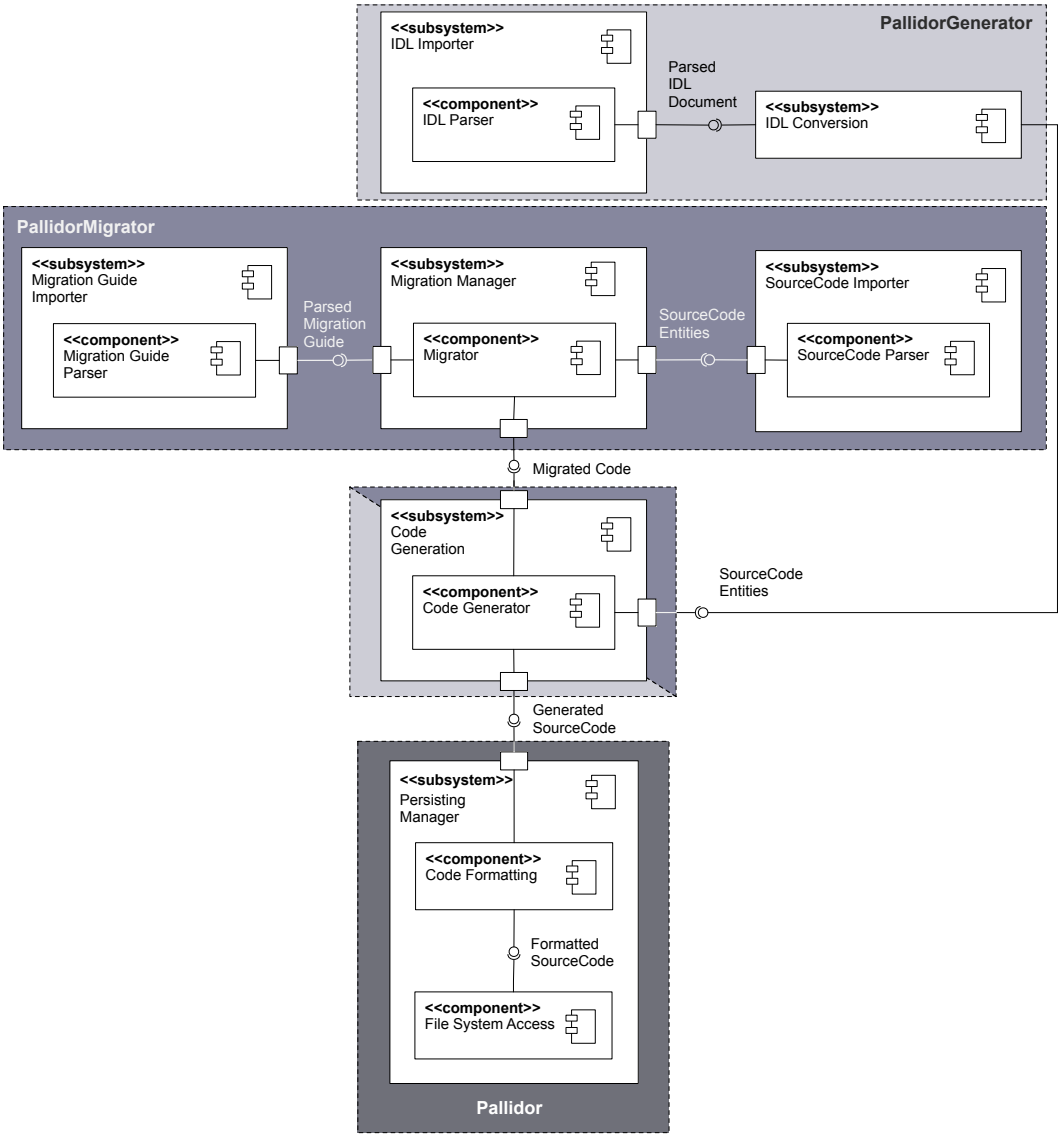

Figure 6.1: Implementation of subsystems with Pallidor Swift packages

Both packages implement the functionality of the `Code Generation` subsystem and persist their respective results in files. The `Code Generation` subsystem is not separately implemented because our prototype only supports the Swift programming language. The PALLIDOR package combines both subpackages in an

executable that provides a command-line user interface. Furthermore, it formats the source code and persists it afterwards. Figure 6.1 illustrates which package implements the respective subsystem.

In addition to our own implementations, third-party open-source components that provide the required functionality are integrated using the Swift Package Manager. Key components and their respective implemented subsystems are listed in Table 6.1. Since Pallidor is a prototype, its functionality is limited to generating a persistent Swift package from an OpenAPI specification. It can be integrated in client applications using Swift in version 5.2 or later. Since our generated Swift package uses Apple's `Combine`[2] framework, the client application's execution environment must be at least iOS 13 or MacOS 10.15.

Subsystem/ Component	**Swift Package Name**	**Swift Package Version**	**Swift Package Author / Publisher**
IDL Importer	OpenAPIKit	2.2.0 (Dec. 20th, 2020)	Mathew Polzin
Source Code Importer	Sourcery	1.0.2 (Nov. 30th, 2020)	Krzysztof Zabłocki
Code Formatter	swift-format	0.50300.0 (Sep. 19th, 2020)	Apple, Inc.

Table 6.1: Third-party Swift packages used for subsystems in Pallidor

The `OpenAPIKit` is a library that enables encoding to- and decoding Swift types from OpenAPI documents. It supports importing documents structured in JSON and YAML formats and provides Swift types for most types of the OpenAPI specification in (3.0.2). In its latest version, the `OpenAPIKit` does not support `Link Objects` and specification extensions for `OpenAPI.XML` and `OpenAPI.Oauth-Flows`. As opposed to the OpenAPI specification, this library specifies the `required` property of schema elements on the element instead of the parent object. OpenAPI documents must be provided as `Data` or `String` types, as the `OpenAPIKit` does not support reading files. Furthermore, all reference elements (`$ref`) must be internal references and target the same file in order to be resolved.

`Sourcery` is a code generator for Swift, built on top of Apple's `SourceKit`. It extends its functionality and types to facilitate generating Swift code based on Stencil templates. Although `Sourcery` is designed as a command-line tool to be integrated in a custom build phase, it provides a framework that can be incorporated using the Swift Package Manager. In addition to generating Swift code, it provides support for parsing Swift strings in corresponding source code entities.

[2]https://developer.apple.com/documentation/combine

Formatting the generated Swift code is performed by `swift-format`. It can be used as a command-line tool or integrated into applications using the Swift Package Manager. It provides an API to format Swift code according to a style configuration. If no set of styling rules is specified, a default style is used. The package is compatible with Swift 5.1 and higher and requires selecting the appropriate branch that corresponds to the version of Swift used in the integrating application.

6.1.1 Pallidor Generator

The `PallidorGenerator` Swift package is concerned with retrieving and parsing of OpenAPI specifications to generate the library layer of the Swift package for the client application. Therefore, OpenAPI specifications must be parsed and converted to source code entities. After that, they are transformed into source code strings in the Swift programming language and are persisted in files. The package integrates the `OpenAPIKit` which decodes an OpenAPI specification in its `OpenAPI.Document` type. This type is composed of elements that represent the attributes of an OpenAPI specification. Furthermore, it provides a locally dereferenced representation that facilitates the traversal of its structure by replacing all `$ref` elements with their referenced entities. The `OpenAPI.Document` needs to be converted to source code entities before Swift code can be generated. This conversion is performed during the traversal of the document by using various `Resolver` objects. Depending on the OpenAPI specification element that needs to be converted, a corresponding `Resolver` initializes a source code entity model using the information of the OpenAPI element. Resolving an endpoint struct from the `ResolvedRoute` type of the `OpenAPIKit` is shown in Listing 6.1.

```
1  /// Resolves an endpoint struct
2  /// - Parameters:
3  /// - path: path in OpenAPI document
4  /// - route: Resolved route from OpenAPI document
5  /// - Returns: resolved EndpointModel
6  static func resolve(path: OpenAPI.Path, route: ResolvedRoute)
       -> EndpointModel {
7    let endpoint = EndpointModel(
8       name: path.components[0].upperFirst(),
9       operations: [],
10      detail: route.summary)
11   endpoint.operations = route.endpoints.map(
12      { OperationModel.resolve(endpoint: $0) })
13   return endpoint
14 }
```

Listing 6.1: Resolving an operation

All source code entity models of the `PallidorGenerator` package are extended with the `CustomStringConvertible` protocol that adds the `description` computed property. It is used to specify string templates for generating the Swift

code of the library layer. They are dynamically composed of all string templates of nested models such as methods, parameters or properties. Furthermore, static templates for meta files such as the `Package.swift` file are provided as Markdown (`.md`) files. Although minor adjustments are made, for example specifying the name of the Swift package, their functionality remains the same between different Web APIs.

```
/// Extension of the Endpoint Model
/// Used for generating client library code in Swift
extension EndpointModel : CustomStringConvertible {
  var description: String {
    """
    import Foundation
    import Combine

    struct _\(name.upperFirst())API {
      static let decoder ...
      \(operations
      .sorted(by: {$0.operationId < $1.operationId})
      .map({$0.description})
      .joined())
    }
    """
  }
}
```

Listing 6.2: Source code string template for an endpoint

The public interface of the `PallidorGenerator` Swift package is provided by the file of the same name. Its initialization requires specifying the URL of the OpenAPI specification to be decoded. Depending on its format, a `JSONDecoder` or `YAMLDecoder` is created that is used to decode the OpenAPI specification. Initializating the `PallidorGenerator` fails, if the OpenAPI document cannot be decoded or resolved due to an invalid URL or malformed structure. Generating the Swift code of the library layer and persisting it in files is performed by invoking its `generate` method. It receives a `path` parameter and the name of the client library as specified by the user. After generating all files, a list of their URLs is returned to the method's caller. The method throws an error if writing the files in the target directory fails. Several OpenAPI specifications of various Web APIs are used for testing the `PallidorGenerator` Swift package. They are specified in Markdown files and are located in the `Resources` subfolder of the test folder. The generated files are validated against predefined results specified in Markdown files that are located in the `Results` subfolder.

Integrating the `PallidorGenerator` package in a Swift project, the URL of its GitHub repository must be added to the dependencies in the `Package.swift` file. Furthermore, the `develop` branch must be selected for using its latest version.

```
.package(
   url: "https://github.com/Apodini/PallidorGenerator.git",
   .branch("develop")
)
```

Listing 6.3: Integrating PallidorGenerator in SPM

With regard to the implementation of the `PallidorGenerator` Swift package, some restrictions must be observed. Instead of defining inline objects in nested elements of an OpenAPI specification, schema components must be used to specify custom data types. They must be referenced at their designated target element. This limitation is based on the problem that nested elements do not provide an identification, which is required for building the library layer. The OpenAPI specification supports defining primitive data types as schema components using a custom name. These type aliases are resolved and their underlying primitive types are used in the generated client library. Furthermore, all endpoint methods in an OpenAPI specification must specify the `operationId` property. Analogous to schema components, this identifier is used by PALLIDOR to recognize changes and to generate the client library. Although it is not enforced by the OpenAPI specification, all operations must specify at least one response that is returned after the request was processed successfully. Otherwise the return value type of an operation cannot be inferred from the OpenAPI document. Unlike successful responses, error responses can be of various types.

6.1.2 Pallidor Migrator

The `PallidorMigrator` package integrates the functionality of the `Migration Manager`, `Migration Guide Importer` and `SourceCode Importer` subsystems. It uses the information stated in the parsed migration guide to adapt the source code entities of a previously generated facade layer. Therefore, a set of migrations is initialized that performs the necessary migratory steps on the changed target properties of the source code entity. After migrating all changes, the source code entities are converted into Swift code that gets persisted in files. The machine-readable migration guide is parsed by the builtin `JSONDecoder` of Swift's `Foundation` framework. The `PallidorMigrator` package provides a model for mapping the migration guide to its decoded representation. Depending on the type of change that is stated in the migration guide, the `MigrationSet` initializes a corresponding `Migration` object. Each `Migration` is checked for plausiblity on its initialization. Thereby, invalid or inconsistent changes are detected and its `solvable` property is set to false. An unsolvable `Migration` results in an error message that is propagated to the user. Every `Migration` targets one source code entity of a previously generated facade. The `Sourcery` framework is used for importing its source code. It provides source code entities for all syntactical elements of Swift code such as classes, structs and methods. These entities cannot be extended within the `PallidorMigrator` package, because they are declared as `final`.

```
/// Wraps struct types of Sourcery
class WrappedStruct: Modifiable {
  ...
  convenience init(from: SourceryRuntime.Struct) {
    self.init(
      localName: from.localName.removePrefix,
      variables: from.variables
        .map({ WrappedVariable(from: $0) }),
      methods: from.methods
        .map({ WrappedMethod(from: $0) }))
  }
}
```

Listing 6.4: Extending a wrapped Sourcery struct

In order to adapt them to changes, wrapper classes are used to extend their functionality with the `Modifiable` protocol. This protocol introduces a `modify` method that receives a `Change` object as a parameter. A modifiable endpoint struct that wraps the `SourceryRuntime.Struct` type of the `Sourcery` framework is shown in Listing 6.4. Depending on the type of change that the `modify` method receives, the corresponding handling routine is performed.

```
/// Wrapped method of SourceryMethod
class WrappedMethod: Modifiable {
  ...
  func modify(change: Change) {
    self.modified = true
    switch change.changeType {
      case .add:
        handleAddChange(change: change as! AddChange)
        break
      case .rename:
        handleRenameChange(change: change as! RenameChange)
        break
      case .replace:
        handleReplaceChange(change: change as! ReplaceChange)
        break
      case .delete:
        handleDeleteChange(change: change as! DeleteChange)
        break
    }
  }
}
```

Listing 6.5: Modification of a changed method entity

Their implementations vary depending on the type of source code entity that is modified. They adapt an entity's associated properties according to the details of the migration. In Listing 6.5, the implementation of the `modify` method highlights the various handling routines that are used to migrate the properties of a

`WrappedMethod`. Similar to the source code entities of the `PallidorGenerator` package, the wrapped source code entities provide templates that are used to convert them into Swift code. These source code strings are persisted in files.

The public interface of the `PallidorMigrator` Swift package is provided by the file of the same name. Its initialization requires specifying the URL of the migration guide and the URL of the directory in which the previously generated client library is located. If the migration guide cannot be retrieved or decoded, an error message is thrown and forwarded to the user. Furthermore, on initialization of the `PallidorMigrator` struct a set of migrations is created. If their plausibility check fails, an error message is shown to the user. Migrating the changes as stated in the migration guide and persisting the adapted facade layer in files are started by invoking the `buildFacade` method. After generating all files, it returns a list of their URLs. An error is thrown if writing the files of the adapted facade layer fails. The `PallidorMigrator` Swift package is extensively tested to discover erroneousness migrations. Therefore, migration guides are created that contain various types of changes with their respective targets. The unit tests migrate prepared source code entities according to these migration guides and compare the outcome to predefined results. Various integration tests are performed that combine different changes on a single target to ensure that they do not affect each other. The input documents of previously generated facade layer code and their predefined adapted results are specified in Markdown files located in the `Resources` subfolder of the test folder.

Integrating the `PallidorMigrator` package in a Swift project, the URL of its GitHub repository must be added to the dependencies in the `Package.swift` file. Furthermore, the `develop` branch must be selected for using its latest version.

```
1  .package(
2    url: "https://github.com/Apodini/PallidorMigrator.git",
3    .branch("develop")
4  )
```

Listing 6.6: Integrating PallidorMigrator in SPM

The implementation of the `PallidorMigrator` Swift package deviates from the design of our proposed system. In addition to the original design, the generated library layer must also be imported. No migration guide or previously generated facade can be used during the initial integration of PALLIDOR as the current version of the Web API is persisted. Therefore, the facade layer is generated based on the public interface of the library layer. Furthermore, no migration strategy can be selected so that all changes of a WebAPI are migrated. A limitation of `Sourcery` concerns the automatic documentation of the public facade layer of the client library because it does not support parsing comment annotations. Therefore, the documentation for the client library is located at the library layer and must be consulted manually. Additionally, some types of migration must be performed by combining multiple migratory steps. Replacing an endpoint is achieved by replacing all of its methods and then renaming it. Renaming and replacing enum

cases and inherited schemas is accomplished by adding their replacements before removing them. Return values of endpoint methods can only be replaced because they do not specify a name that could be renamed. Furthermore, all operations must provide a return value type, which means that deleting them or adding them later is not applicable.

6.1.3 Pallidor

The PALLIDOR package contains the command-line user interface and the `Code Formatting` component. Furthermore, it stores all configuration settings of the user. By invoking PALLIDOR via its commandline interface and providing all required configuration parameters, the client library is generated. The `swift-argument-parser`[3] package is used to supply detailed error and help messages and it facilitates type-safe argument parsing. The URLs of the OpenAPI specification and migration guide are used to initialize the `PallidorGenerator` and `PallidorMigrator` Swift packages. Additionally, the name and location of the client library are used by the packages to both, import previously generated source code and persist their results after execution. After both packages successfully completed their tasks, the `swift-format` package is used to format the code according to the ruleset that the user provided. The formatted source code strings overwrite the previously generated Swift files.

Several parameters are used to configure PALLIDOR. While the URLs of the migration guide, OpenAPI specification and location of the package as well as its name are mandatory parameters, specifying a ruleset for formatting the source code strings is optional. PALLIDOR is limited to emitting the client library as a Swift package, hence selecting a programming language currently defaults to Swift. A migration strategy can be set to exclude certain types of migration. This configuration parameter defaults to the implemented strategy of migrating all types. The `help` argument is provided by the `swift-argument-parser` and lists all available parameters of PALLIDOR. It also supports the user with helpful error messages that highlight the correct usage.

```
Error: Missing expected argument
    '--target-directory <target-directory>'

USAGE: pallidor
--openapi-specification-url <openapi-specification-url>
--migration-guide-url <migration-guide-url>
--target-directory <target-directory>
--package-name <package-name>
[--language <language>]
[--strategy <strategy>]
[--custom-formatting-rule-path <custom-formatting-rule-path>]

OPTIONS:
```

[3] https://github.com/apple/swift-argument-parser

```
14 -c, --custom-formatting-rule-path <custom-formatting-rule-path>
15 If you want to use your own code formatting rules, specify path
       here
16 -o, --openapi-specification-url <openapi-specification-url>
17 URL of OpenAPI specification of the package to be generated
18 -m, --migration-guide-url <migration-guide-url>
19 URL of migration guide of the package to be generated
20 -t, --target-directory <target-directory>
21 Output path of the package generated
22 -p, --package-name <package-name>
23 Name of the package generated
24 -l, --language <language>
25 Programming language that the client library should be
       generated in (default: Swift)
26 -s, --strategy <strategy>
27 Migration strategy that excludes certain types of change from
       being migrated (default: all)
28 -h, --help
29 Show help information.
```

Listing 6.7: Command-line arguments of Pallidor to support user

PALLIDOR is designed as a command-line tool that can be integrated into an existing CI/CD system. Therefore, it needs to be compiled and the binaries must be deployed. The configuration parameters can be predefined in a script file that is integrated in the build phase of the CI pipeline. The script is automatically executed as a step phase and performs the migration to generate the Swift package that is integrated in the client application. The CI/CD system must be configured to publish the generated Swift package in a separate respository that is referenced from the `Package.swift` file of the client application. After that it automatically creates a new release of the client application.

6.2 Evaluation

For evaluating our proposed system, we use IDL documents of two consecutive Web API versions and a corresponding migration guide. The IDL documents must be manually manipulated as they only describe the latest version of a Web API. Since a machine-readable migration manual is neither publicly available nor can it be generated automatically, it must be created manually in order to document all occuring changes. Most changes cannot be viewed in isolation, as changes to higher-level elements affect changed sub-elements. For a thorough assessment of our proposed system's capability to migrate the various changes and their permutations, we decided to perform the evaluation by manually incorporating them in the sample `Pet Store`[4] Web API. Therefore, its OpenAPI specification is modified to reflect the change patterns that were discovered by Li et. al [LXLZ13]. Additionally, migration guides are created for each type of migratable change and permutation that provide instructions on how to overcome the breaking change.

The documentation of our results is based on the structure of the *Deductive Mini-Pattern* template, that focuses the outcomes of the described solution including the respective benefits and consequences [BMI+98]. The different types of changes are separated into the categories AUTOMATED MIGRATION, SEMI-AUTOMATED MIGRATION, and MANUAL MIGRATION depending on their degree of automation. Furthermore, they share commonalities in the problems they cause for client applications and their developers. These problems are presented in detail for each category to clarify the necessity of their solution. In the description of the respective solution, we explain the different approaches that support client developers in migrating their application. Each solution offers individual benefits, but also has other consequences. By describing the benefits of a solution, we focus on the positive effects of our system on the reduction of the effort required for the manual migration for client developers. The consequences described serve as support for users of our system, since observing them helps preventing undesired side effects in the client application. Finally, we describe the testing procedure and the required documents used in this process.

6.2.1 Automated Migration

Breaking changes that belong to this category result in compile-time errors because they affect syntactical elements of the client library. They are always reflected in a Web APIs IDL document and currently require manual refactoring activities by client developers. In the course of this, client developers must consult all related documents in order to integrate the changes into their application accordingly. By using PALLIDOR with our novel machine-readable migration guide, this process can be fully automated.

[4] https://petstore3.swagger.io/

Category: AUTOMATED MIGRATION

Automating the migration is based on two steps. The first step is taking advantage of the separation of concerns principle through the encapsulated design of the client library. Due to its architecture, changes at the HTTP level and the addition of new elements can be automatically integrated. The second step is to automatically adapt the client library to the changes stated in the migration guide by modifying the internals of its facade layer.

Problem: All of the changes in this category require refactoring the client application. Depending on whether the client application uses a wrapper library or accesses the Web API directly via HTTP request, these changes cause an error at compile time or runtime. Breaking changes of this category are listed below:

- *Adding changes* of models, attributes, optional parameters, methods or endpoints
- *Renaming changes* of models, attributes, parameters, methods or endpoints
- *Replacing changes* of models, attributes, parameters and methods
- *Adding required parameters*
- *Combining and splitting methods*
- *Combining and splitting parameters*
- *Changing default value of parameters*
- *Changing the underlying HTTP method*
- *Changing the server's URL or ports*
- *Permutations of changes*

Solution: For migrating the changes of the underlying HTTP method and the server's URL or ports, we designed the architecture of the client library to automatically incorporate these changes. Therefore, the `NetworkManager` struct in the library layer encapsulates lower-level HTTP calls and contains a list of server URLs that is extracted from the OpenAPI specification. By default, the first server in the list is selected to state requests to the Web API. The library layer specifies the HTTP method that is used to query the Web APIs endpoints. The facade layer only invokes the higher-level methods of the library layer, hence the client application remains its functionality regardless of these changes in the library layer. New services are automatically integrated in the client library and do not require an entry in the migration guide. All other changes cannot be incorporated by design and must be migrated. Therefore, PALLIDOR uses the information of the machine-readable migration guide to instantiate a `MigrationSet` that contains all `Migrations` that need to be performed. Migrating a change is executed by adapting the target syntax elements of the facade layer to incorporate the changes of the library layer.

Benefits: By automating the refactoring process, the effort for client developers is significantly reduced. By concentrating all evolution-related information in a machine-readable migration guide the necessity of manually examining related documents of the Web API is eliminated. Client developers are able to integrate PALLIDOR into their CI workflow which ensures that their application always incorporates the latest changes of a Web API dependency.

Consequences: Using our solution enables client developers to operate their application on an outdated syntax of the Web API. Some changes require fallback or default values in order to be migratable and they may not contain the specific values required by the client developer. Migrations of *Replacing* changes need to execute foreign code which reduces the applications performance and raises security concerns. By migrating the combination or splitting of methods instead of using the implementation of the Web API, client applications cannot benefit from potential increases in performance and efficiency.

Tested By: All change types of this category are tested by unit tests or executable script files. Each change type is tested on a target in isolation and also in combination with other change types if this applies for the target type. In the remainder of this section, the test cases for all change types are listed in their respective table except for the *Permutation* change types which can be found in the Appendix 1 and 2. All tables in this section list the performed tests on each target type. The tests are either given as unit tests or they refer to a script that can be locally executed from the root directory of the PALLIDOR Swift package. They require to specify the target directory of the generated Swift files using the `-t` parameter. Each script generates separate Swift packages for the unchanged and updated version of the Web API to facilitate the comparison of the changed elements. Script files are highlighted in *italic* while unit tests are prefixed with the `test` keyword.

New elements of a Web API are incorporated either by the design of the generated Swift package or by automatically adding them during the migration process.

Adding new attributes or required parameters must be migrated as they break the client application. Therefore, Web API providers must specify a default value that can be used in case no value is passed by the client developer.

Target	Test	Identifier	Description
Model	*addModel.sh*	-	The `ApiResponse` model is integrated in the Swift package. No migration guide is required as this is a non-breaking change. However, replacements using the new type must be documented.

Target	Test	Identifier	Description
Attribute	testAdded-Property	`city`	A new attribute is added to the `Address` model. It specifies a default value to prevent en-/decoding errors
Endpoint	*addEndpoint.sh*	-	The `Store` endpoint gets automatically integrated in the Swift package. No migration guide is required as this is a non-breaking change.
Method	*addMethod.sh*	-	New methods are automatically integrated in the respective endpoint of the facade layer.
Parameter	testAdded-Parameter	`status`	A new required parameter `status` is added to `updatePet`.
Return Value	-	-	Due to our predefined constraints, return values must be available for each method. Adding a new return value is therefore not valid.

Table 6.2: Adding migrations for all target types

Adding models, methods and endpoints does not break client applications as they introduce new functionality without modifying existing services. However, these additions should be made available to client developers to extend their applications. Therefore, they are automatically integrated in the library and facade layer of the Swift package. The test cases of both adding change types are listed in Table 6.2. Methods must specify a return value as described in Section 6.1.1, hence return values cannot be added after publishing the Web API.

Renaming changes alter the identifier of the syntax elements without modifying other related properties or their behavior. They require client developers to adapt their application to use the new identifiers. Although modern IDEs provide support for automatically renaming all occurrences of a syntax element, a client developer must manually start this process. Renaming migration tests are listed in Table 6.3. Depending on whether a wrapper library is used to access the Web API or the Web API is queried directly, renaming elements either leads to compiler errors or error messages returned from the Web API. Refactoring is performed by modifying the internals of the facade layer to use the new identifiers in the library layer. However, the public interface of the facade layer is not changed. Return values have no identifier, hence renaming them is infeasible. Changes in models affect other elements of the Web API as they are used as types by parameters, return values or attributes. In order to rename them, the Web API provider must replace the types of the affected elements accordingly.

Target	Test	Identifier	Description
Model	testRenamed-Model	`Address`	Renamed the `Address` model to `NewAddress`. Changing the identifier of a model requires to replace the types of all referencing elements. Therefore, also a replace change of the `address` attribute of the `Customer` model is migrated.
Attribute	testRenamed-Property	`tags & name`	The attributes of `Pet` and `Category` are renamed to `tagsi` and `namenew`.
Endpoint	testEndpoint-Renamed	`/pet`	The route of the Pet endpoint is changed to `/pets`. This results in a renaming change from `PetAPI` to `PetsAPI`.
Method	testRenamed-Method	`addPet`	The method is renamed to `addMyPet`.
Parameter	testRenamed-Parameter	`status`	The parameter of `findPetsByStatus` is renamed to `petStatus`
Return Value	-	-	Return values do not have an identifier and hence cannot be renamed.

Table 6.3: Rename migrations for all target types

Replacing changes are reflected in the IDL document by the removal of the element. In contrast to a *Removing* change listed in Table 6.7, a replacement for the removed element is provided, which can be specified in other related documents. Client developers must manually inspect these documents in order to find the replacement that can be used in their application. This examination is cumbersome for Web API consumers because according to Brito et. al, providers do not use a single artifact for documenting changes [BVXH20]. Automating the migration of a *Replace* change requires Web API providers to specify the replaced element and its replacement in the machine-readable migration guide. Furthermore, they must provide algorithms for converting and reverting types of a replaced model, attribute or a methods parameters and return values using the JavaScript programming language. All tests that concern replacements of all target types are listed in Table 6.4.

Target	Test	Identifier	Description
Model	testReplaced-Model	`Order`	Replaced the `Order` model by `NewOrder`. In contrast to renaming a model, algorithms for converting and reverting must be specified. Replacing a model requires to use the `Signature` target value of changes.
Attribute	testReplaced-Property	`address`	The attribute was renamed to `addresses` and also its type is changed to `[NewAddress]`.
Endpoint	-	-	Endpoints can not be directly replaced using our migration guide. Replacing an endpoint is performed by replacing all of its methods before renaming the endpoint.
Method	testReplaced-Method	`updatePet`	The method is replaced by `updateMyPet` in the `User` endpoint. Replacing a method requires to use the `Signature` target value of changes.
	testReplaced-Method-InSame-Endpoint	`updatePet`	The method is replaced by `updatePetWithForm` in the `Pet` endpoint. Replacing a method requires to use the `Signature` target value of changes.
Parameter	testReplaced-Parameter	`petId`	The parameter `petId` of `updatePetWithForm` is replaced by `betterId`. The type of the parameter was changed from `Int` to `Double`.
Return Value	testReplaced-ReturnValue	`addPet`	The type of the return value was changed from `Pet` to `Int32`.
	testReplaced-ReturnValue	`updatePet`	The type of the return value was changed from `Pet` to `ApiResponse`.

Table 6.4: Replace migrations for all target types

In addition to renaming their identifiers, replacing Web API elements also changes

their types. In order to maintain a stable public interface of the facade layer, the previously used types must be converted to their replacement types and the replacement types must be reverted to their original types. Therefore, Web API providers must specify the respective algorithm in the migration guide using the JavaScript programming language. When a method is replaced, the conversion algorithm is used to convert the parameters of the original method to an object of the parameters of its replacement. The reverting algorithm is used to convert the replacements return type back to the original return type. Replacing an endpoint cannot be specified as a dedicated change type in our migration guide. However, to replace an endpoint, its identifier can be renamed after all of its methods were replaced.

Permutations of changes are not described by Li et. al [LXLZ13] as they are not a distinct type of change but they introduce multiple side effects that need to be considered. For example when replacing a parameter of a method is followed by a *Renaming* change of its signature, the latter must be migrated before replacing the parameter.

Renamed Targets	**Order of changes**	**Additional notes**
Endpoint	1. Rename change of method 2. Rename change of endpoint	In order to identify the renamed method in the renamed endpoint, the endpoints new identifier must be used in the `defined-in` property of the method object.
	1. Replace change of method (replaced method and replacement in same endpoint) 2. Rename change of endpoint	In order to identify the replaced method, the endpoints previous identifier must be used in the `defined-in` property of the `replaced` object.
	1. Rename change of endpoint 2. Remove change of method	In order to assign the removed method to the renamed endpoint, the endpoints new identifier must be used in the `defined-in` property of the method object.
Method	1. Rename change of method 2. Add change of parameter	In order to identify the renamed method, its new identifier must be used in the `operation-id` property of the method object.

Renamed Targets	Order of changes	Additional notes
	1. Replace, rename or remove change of parameter 2. Replace change of return value 3. Rename change of method	In order to identify the renamed method, its new identifier must be used in the `operation-id` property of the method object. Changes targeted at parameters and return values can be arranged arbitrarily.
Endpoint & Method	1. Replace change of return value 2. Add, rename, replace or remove change of parameter 3. Rename change of method 4. Rename change of endpoint	In order to identify the renamed method, its new identifier must be used in the `operation-id` property of the method object. Furthermore, to locate it in the renamed endpoint, the endpoints new identifier must be used in the `defined-in` property of the method object.
Model	1. Add, replace, rename or remove change of attribute 2. Rename change of model	In order to identify the target of the attribute change, the models new identifier must be used in the `name` property of the model object.

Table 6.5: Order of permutation changes in migration guide

Permutations significantly increase the effort involved in manually migrating a client application. They can occur for all types of changes listed in Table 6.2, 6.3 and 6.4. with the exception of *Removing* changes, because removing the parent element has the same effect on all child elements. Furthermore, since the subsequent addition of an element has no effect on existing elements and the replacement of a higher-level element takes into account all effects on its subordinate elements, we focus on permutations in which the higher-level elements are renamed. The test cases for all types of change permutations are listed in the appendix of this thesis. Table 1 contains permutations that are affected by the renaming of an endpoint or method and by the renaming of methods in a renamed endpoint. Table 2 lists the test cases with changes of attributes in a renamed model.

Although our proposed system is unopinionated about the order of changes stated in the machine-readable migration guide, PALLIDOR requires a specific order to migrate permutation change types. The orders that must be observed for a specific type of permutation change are listed in Table 6.5. Change types that are not specified in the migration guide such as adding a method do not affect the order of other changes.

Further test of the remaining change types that can be automatically migrated are shown in Table 6.6. These types add additional challenges to the previous types of change or affect lower level services.

Requiring a previously optional parameter changes the conditions of invoking a method and must be migrated. Therefore, Web API providers must specify a default value that can be used to call the method in case no value is passed by the client developer. However, if a required parameter is made optional, then it is a non-breaking change which does not need to be considered.

Combining and splitting methods extends the problem of a single replacement for a removed method. By combining the functionality of multiple methods in a single method, the parameters of the replaced methods must be combined before invoking the replacement. Its return value must be split to match the return values of the replaced methods. By splitting a single method into multiple methods, its parameters are split and the return values of all replacements must be combined before returning the result. This type of change is currently not supported by PALLIDOR but it can be extended to add this feature. However, example test cases are specified in the `MethodMNReplaceTest.md` file that describe their migration processes including required inputs and proposed results. These test cases can be used to validate PALLIDOR after it has been extended to include their migration functionality.

Change	Test	Description
Requiring parameter	testRequiring-Parameter	The optional parameter `status` of the `findPetsByStatus` method is now required. A default value is provided that prevents client applications from breaking.
Combining and splitting methods	-	Combining and splitting methods is not yet implemented by PALLIDOR. Testing this type of change is described in the `MethodMNReplaceTest.md` document in the proposed tests folder.

Change	Test	Description
Combining and splitting parameters	testReplacedMN-ParametersOfMethod	Three parameters of `updatePetWithForm` are replaced by two parameters.
	testReplacedM1-ParametersOfMethod	All primitive parameters of `updatePetWithForm` are replaced by one complex parameter.
	testReplaced1N-ParametersOfMethod	One complex `Pet` parameter of `updatePet` is replaced by multiple primitive parameters.
Changing default value of parameters	testDefaultParameter-PetEndpoint	The default value of the `status` parameter of `findPetsByStatus` is changed from `available` to `pending`.
Changing the underlying HTTP methods	*changeHTTPmethod.sh*	The underlying HTTP methods of `updatePet` and `addPet` are changed to POST and PUT.
Changing the server's URL or ports	*changeServerURL.sh*	The default server URL is changed and port 8080 instead of 443 was set. This change modifies the server list of the `NetworkManager` struct.

Table 6.6: Further tests of various change types

Combining and splitting parameters are an extension of single replacements for parameters. Combining multiple independent parameters of a primitive type to a complex type can be migrated by synthesizing the new type from the previous types. This may be complemented by a change in the underlying HTTP method that allows transmitting a content body. For splitting a complex type into multiple primitive types, it must be decomposed before invoking the method.

Changing the default value of parameters does not prevent the client application from compiling as default values are only applicable to optional parameters. However, when client developers use a parameters default value, changing it can result in unexpected behavior. For example, if a parameter specifies the number of elements returned from the Web API, changing the default number affects the number of displayed elements in the UI of a client application. Therefore, unlike the other types of changes in this category, changing the default value of a parameter affects not only the syntax of the Web API, but also its semantics.

Changing the underlying HTTP method results in an error message returned by the server. Depending on the architecture of an application, this change can

affect multiple files that must be modified by client developers. By encapsulating lower-level HTTP calls, this type of change only affects one file in the library layer of the generated Swift package. Since this change is not exposed to the client application, changes to the underlying HTTP method are automatically incorporated and do not need to be specified in the migration guide.

Changing the server's URL or ports results in an error message of the client application. As with changing the underlying HTTP method, the client applications architecture has a decisive influence on the effects of this change. Impacts on security systems such as firewalls are ignored in our evaluation. Migrating a change of the server's URL or ports is only considered fully automatable if the client developers access the server list of the `NetworkManager` struct in a secure manner. For example, since at least one server must be specified in order to access the Web API, using the first entry in the list is always considered secure access.

6.2.2 Semi-Automated Migration

Changes categorized as semi-automatable can be migrated by our proposed system, but the adapted client library is incapable of reproduce the previous state exactly. Nonetheless, our proposed system provides additional support for client developers to manually migrate their application. Changes in this category are reflected in the IDL document of the Web APIs either in modifications to their elements or in their absence.

Category: SEMI-AUTOMATED MIGRATION
Changes of this type can be further divided into changes that can be migrated but generate an error at runtime if the client library is used incorrectly and changes that can be migrated but do not restore the original functionality. New approaches must be explored for the automatic migration of the latter.

Problem: Changes in this category require client developers to manually migrate their application in addition to the automated refactoring performed by our proposed system. Without manual adjustments, these changes lead to runtime errors. A full list of these changes is shown below:

- *Removing changes*
- *Parameter boundaries*
- *Requiring authentication*
- *Multi-version migrations*

Solution: The problems are reflected in the IDL document, which enables a semi-automated migration. In addition, removing changes can be stated in the machine-readable migration guide, whereby fallback values can be specified for certain targets that enable their fully automated migration. Targets that do not allow specifying a fallback value are provided with the `@available` annotation, which returns an error message stating the reason for the removal at compile time. Changes to the parameter boundaries are automatically adjusted at the library level, so that client developers receive an error message if they are exceeded. All methods generated by PALLIDOR enable the provision of authentication information which can be stored centrally in the `NetworkManager` struct. In the event that a method's authentication requirement changes, this information is automatically used, implied that it has been specified once by the client developers. For migrating the client library between multiple versions of a Web API, future research should be conducted to automatically generate a migration guide on demand for a particular version.

Benefits: Although the changes cannot be automatically migrated, our proposed system provides helpful support for client developers. Errors caused by changes in this category can only be identified at runtime, when a Web API

is manually integrated. By using the generated client library, which directly reflects these changes, the detection of these errors is shifted to compile time. In addition, by migrating removed elements, we integrate detailed error messages that are displayed as compilation errors by Xcode. Web API providers can use them to explain their clients why the affected element was removed and, if possible, what further steps can be taken. Compliance with parameter boundaries is ensured by preconditions, which are automatically adjusted whenever the boundaries change. Unit tests should be defined by client developers so that they are informed about these changes and can make appropriate adjustments before they publish a new version of their application. The design of the client library allows its users to avoid the problem of authentication requirements that are added after the initial integration of the Web API.

Consequences: Providing error messages for removed elements does not restore the Web API dependency to its original state. Replacing the removed functionality must be done manually by client developers. Since unit tests have to be created manually by client developers, a change in the parameter boundaries remains unnoticed if they are omitted. The use of central authentication information is only possible if it is already used elsewhere in the client application. If the Web API did not previously require any authentication or if no such method was used, there is no information that could be automatically added. Until they can be generated, providers must manually create and maintain migration guides for each version of their Web API.

Tested By: While unit tests are used to validate the migration of removed elements, changes in authentication requirements and parameter boundaries are tested by scripts that can be locally executed from the root directory of the PALLIDOR Swift package. They require to specify the target directory of the generated Swift files using the `-t` parameter. Each script generates separate Swift packages for the unchanged and updated version of the Web API to facilitate the comparison of the changed elements. Script files are highlighted in *italic* while unit tests are prefixed with the `test` keyword. For multi-version migrations, example test cases are specified in the `.md` file that describes the process of how migration guides from several versions can be merged into one migration guide for a specific version step. These test cases can be used by future research to integrate them in the automatic generation of migration guides.

Removing elements of a Web API without providing a replacement cannot be migrated to restore their original state. In order to support client developers, PALLIDOR annotates removed elements with Swift's `@available` attribute. So whenever a removed item is accessed, a compiler error is displayed explaining the reason for the removal. All tests for migrating removed elements are listed in Table 6.7.

Target	Test	Identifier	Description
Model	testDeleted-Model	`ApiResponse`	The model `ApiResponse` is removed from the schema components of the Web API.
Attribute	testDeleted-Property	`weight`	The attribute is removed from the `Pet` model. A fallback value is provided to prevent the client application from breaking.
Endpoint	testDeleted	`/pet`	The endpoint `Pet` is removed from the Web API. This includes removing all of its methods.
Method	testDeleted-Method	`addPet`	The method is removed from the `Pet` endpoint.
Parameter	testDeleted-Parameter	`username`	The parameter is removed from the `updateUser` method of the `User` endpoint. A fallback value is provided to prevent the client application from breaking.
Return Value	-	-	Methods must specify a return value. Therefore, removing changes cannot target return values.

Table 6.7: Remove migrations for all target types

Web API providers can specify fallback values for `Parameters` and `Attributes` that are incorporated by PALLIDOR to prevent the client application from breaking. Each method must specify a type that is returned upon successful completion of the request. Hence, removing changes cannot target return values.

The remaining change types in this category modify the semantics of the Web API. Our proposed system provides a means for client developers to automatically migrate these changes, provided that certain requirements are met. Tests for these change types are listed in Table 6.8.

Changing the boundaries of parameters affect the preconditions of invoking a method of a Web API. Supplying a value that falls below the lower or exceeds the upper boundary, results in an error message issued by the Web API. Extending boundaries does not break client applications as the existing implementation is still valid. However, narrowing boundaries can lead to the situation that the previously valid values are no longer within the limits.

Requiring authentication of previously unauthorized functionality forces client developers to authenticate before requesting services. If no authorization information is transmitted, the request fails and an error message is issued by the Web API.

Multi-version migrations are currently not supported by PALLIDOR. It supports migrating client libraries between two versions of a Web API. Therefore, their providers must specify and maintain a migration guide for every version ever released.

Change	Test	Description
Parameter boundaries	*changeBoundaries.sh*	The parameter `orderId` of the `getOrderById` method changed its boundaries from accepting values in the range of 1-9999 to values in the range of 1000-4999.
Requiring authentication	*changeAuth.sh*	The method `placeOrder` now requires users to be authenticated while the method `updatePet` is publicly accessible without authenticating. Their `authorization` parameters changed from optional to non-optional and vice versa.
Multi-version migration	-	Currently unsupported. Further investigation is required to develop an algorithm to generate a machine-readable migration guide for a specific version step.

Table 6.8: Further tests of semi-automatable change types

Changing the boundaries of parameters and requesting authentication of a previously unauthenticated method are tested by generating a Swift library for each version of the Web API. In order to support the development of multi-version migrations in the future, we provide an abstract approach on merging multiple migration guides in `MultiVersionMigration.md`. It provides a brief overview of an examplatory approach to combine multiple migration guides of various versions of a method into a single migration guide for migrating the method from its original version to the latest version available.

6.2.3 Manual Migration

There are changes that may occur during the development of a Web API which require manual intervention by client developers. These changes are assigned to the MANUAL MIGRATION category. They cannot be migrated by our proposed system and they are not reflected in the IDL document of a Web API. These changes potentially result in a fundamental redesign of the client application so that it can use an alternative service.

Category: MANUAL MIGRATION
Changes of this category must be manually migrated by client developers. Further research must be conducted to develop new approaches to automate their migration.

Problem: Client developers need to manually migrate their application to restore its functionality. Depending on the extent of the change, migration may not be possible at all. More research is required to create new approaches that provide support for client developers. A list of the changes is shown below:

- *Change Web API architecture*
- *Manual authorization process*
- *Web API discontinuation*
- *Request frequency limitation*

Solution: These changes cannot be migrated by our proposed system. Client developers need to manually adapt their application to the changed environmental conditions. Adapting the client application may be infeasible. Therefore, client developers need to identify alternative services that can substitute the functionality of the originally used Web API.

Benefits: Although our proposed system cannot migrate changes in this category, their classification provides useful insights for related work. Future research can focus on exploring new approaches that will facilitate their migration.

Consequences: Client developers cannot solely rely on an automation of the migration process for their application. They need to monitor the evolution of the Web API in order to react immediately to these types of changes. As the number of published Web APIs increases, alternative services are often offered that can be prepared to replace the functionality of an unavailable Web API.

Tested By: Since our proposed system focuses on automating the migration of changes that affect the syntactic elements of a Web API, manual migrations of semantic changes are not tested.

Changing the Web API's architecture results in a fundamental restructuring of the service. During this process, the type of access and the elementary logic of the

Web API can change. A well-known example of this type of change is the GitHub API[5], the latest version of which is implemented as a GraphQL API instead of the former REST style. The main reasons expressed by its operators are the increased scalability and flexibility of GraphQL, which better fit the expectations of the Web API. Constantly changing requirements result in emerging technologies that will eventually replace previous approaches. Our proposed system can be expanded to generate a client library for the new architectural style. However, neither the system nor the migration guide support migrating a Web API between different technologies.

A **manual authorization process** often requires client developers to go through additional procedures in which documents are manually reviewed by the Web API provider. This enables providers to verify that the client application uses their Web API in an approved manner. In addition, they can use this process to implement existing laws and regulations. Li et. al noticed this change pattern when analyzing the Sina Weibo Web API [LXLZ13]. As some methods provide access to sensitive personal information such as acquiring a user's private messages, client developers who want to access them, are required to apply for an authorization of their application [LXLZ13]. With the increasing awareness of the correct handling of sensitive information, it can be safely assumed that this type of change will occur more frequently in the future. In addition to developing new technical approaches, a legislative basis must be created to automate their migration.

Web API discontinuation renders the entire service unavailable and requires new approaches for its migration. Client developers identify a replacement for the discontinued Web API dependency by comparing its functionality with related services. Once they have decided on a suitable alternative, it can be integrated with our proposed system. Automated migration of the discovery process requires a detailed semantic description of available Web APIs to identify related services and recommend the best fit. At the moment there is no recommendation service that provides this functionality.

Limiting the frequency of requests requires client applications to reduce their number of request accordingly. Unlike parameter boundaries, rate limits are not reflected in the IDL document of a Web API. The rate limits are enforced in the backend of a Web API by throttling or disconnecting client applications that exceed them. Distributed client applications that access this Web API must be dynamically adapted to the remaining quota which must be retrievable from the Web API. For example, all APIs provided by Facebook are subject to rate limits[6] that are included in the headers of most API responses. However, these headers are not standardized and vary between the different Web API providers. A standardization of these headers enables their incorporation into our client library, so that requests can be automatically reduced or postponed as soon as the limit is reached.

[5] https://github.blog/2016-09-14-the-github-graphql-api/
[6] https://developers.facebook.com/docs/graph-api/overview/rate-limiting/

7 Summary

- *This chapter includes the status of your thesis, a conclusion, and an outlook about future work.*

7.1 Conclusion

- *Describe honestly the achieved goals (e.g., the well implemented and tested use cases) and the open goals here.*
- *If you only have achieved goals, you did something wrong in your analysis.*

7.1.1 Realized Goals

- *Summarize the achieved goals by repeating the realized requirements or use cases stating how you realized them.*

7.1.2 Open Goals

- *Summarize the open goals by repeating the open requirements or use cases and explaining why you were not able to achieve them.*
- *It might be suspicious if you do not have open goals, this usually indicates that you did not thoroughly analyze your problems.*

7.2 Future Work

- *Tell us the next steps, that you would do if you have more time. Be creative, visionary, and open-minded here.*

List of Figures

1.1 Cumbersome workflow after web service changes 3
1.2 Workflow with tool support after web service changes 4

3.1 Proposed structure of a machine-readable migration guide 13
3.2 Proposed architecture of change-proof Web API integration 14
3.3 Use cases of Web API consumers 17
3.4 Use cases of Web API providers 20

4.1 Subsystem decomposition 28
4.2 Subsystem decomposition in an Apple ecosystem 31

5.1 Class diagram of IDL importer subsystem 37
5.2 Class diagram of source code entity artifact 38
5.3 Class diagram of code importer subsystem 38
5.4 Class diagram of migration guide artifact 39
5.5 Class diagram of migration manager subsystem 40
5.6 Class diagram of code generator subsystem 41
5.7 Class diagram of persisting manager subsystem 42
5.8 Restructured subsystem decomposition 44

6.1 Implementation of subsystems with Pallidor Swift packages 50

List of Tables

5.1 Input and output of subsystem unit tests for UC1 45
5.2 Additional input and output of subsystem unit tests for UC4 46
5.3 Input and output of subsystem integration tests 47

6.1 Third-party Swift packages used for subsystems in Pallidor 51
6.2 Adding migrations for all target types 62
6.3 Rename migrations for all target types 63
6.4 Replace migrations for all target types 64
6.5 Order of permutation changes in migration guide 66
6.6 Further tests of various change types 68
6.7 Remove migrations for all target types 72
6.8 Further tests of semi-automatable change types 73

1 Tests for all permutation types for renaming endpoints and methods 88
2 Tests for all permutation types for renaming models 89

Listings

5.1 Examplatory migration guide in JSON format 35

6.1 Resolving an operation 52
6.2 Source code string template for an endpoint 53
6.3 Integrating PallidorGenerator in SPM 54
6.4 Extending a wrapped Sourcery struct 55
6.5 Modification of a changed method entity 55
6.6 Integrating PallidorMigrator in SPM 56
6.7 Command-line arguments of Pallidor to support user 57

Bibliography

[BCK13] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, Upper Saddle River, NJ, 3rd ed edition, 2013.

[BD10] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns, and Java*. Prentice Hall, Boston, 3rd ed edition, 2010.

[BMI+98] William J Brown, Raphael C Malveau, Hays W McCormick Iii, Thomas J Mowbray, John Wiley, Robert Ipsen, and Theresa Hudson. Refactoring Software, Architectures, and Projects in Crisis. page 157, 1998.

[BVXH20] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25(2):1458–1492, March 2020.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated Detection of Refactorings in Evolving Components. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Dave Thomas, editors, *ECOOP 2006 – Object-Oriented Programming*, volume 4067, pages 404–428. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[EB18] Anna Maria Eilertsen and Anya Helene Bagge. Exploring API: Client co-evolution. In *Proceedings of the 2nd International Workshop on API Usage and Evolution - WAPI '18*, pages 10–13, Gothenburg, Sweden, 2018. ACM Press.

[EZG14] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API growing pains: Stories from client developers and their code. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 84–93, Antwerp, Belgium, February 2014. IEEE.

[FP11] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ, 2011.

[Gam95] Erich Gamma, editor. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series.

Addison-Wesley, Reading, Mass, 1995.

[HD05] Johannes Henkel and Amer Diwan. CatchUp! capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 274–283, St. Louis, MO, USA, May 2005. Association for Computing Machinery.

[KFJA19] Rediana Koci, Xavier Franch, Petar Jovanovic, and Alberto Abello. Classification of Changes in API Evolution. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 243–249, Paris, France, October 2019. IEEE.

[LXLZ13] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How Does Web Service API Evolution Affect Clients? In *2013 IEEE 20th International Conference on Web Services*, pages 300–307, Santa Clara, CA, USA, June 2013. IEEE.

[LZP+19] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs - EuroPLop '19*, pages 1–24, Irsee, Germany, 2019. ACM Press.

[NN10] Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 205–214, Cape Town, South Africa, May 2010. Association for Computing Machinery.

[XBHV17] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147, Klagenfurt, Austria, February 2017. IEEE.

Appendix

Rename Target	**Method**				**Parameter**				**Return Value**
	Add	*Rename*	*Replace*	*Remove*	*Add*	*Rename*	*Replace*	*Remove*	*Replace*
Endpoint	test-Renamed-AndAdd-Method	test-Renamed-Endpoint-And-Renamed-Method	test-Renamed-Endpoint-And-Replace-Method	test-Renamed-Endpoint-And-Deleted-Method	test-Renamed-Endpoint-And-Rename-Method-And-AddAnd-Delete-Parameter-Change	test-Renamed-Endpoint-And-Rename-Method-And-Changed-Parameters-And-Replace-Return-Value-Change	test-Renamed-Endpoint-And-Rename-Method-And-Replace-And-Delete-Parameter-Change	test-Renamed-Endpoint-And-Rename-Method-And-AddAnd-Delete-Parameter-Change	testRenamed-Endpoint-AndRename-Method-AndReplace-AndDelete-Parameter-AndReplace-ReturnValue-Change
Method	-	-	-	-	test-Rename-Method-And-Added-Parameter	test-Rename-Method-And-Rename-Parameter	test-Rename-Method-And-Replace-Parameter	test-Rename-Method-And-Remove-Parameter	testRenamed-MethodAnd-Replaced-ReturnValue

Table 1: Tests for all permutation types for renaming endpoints and methods

Renamed Target	**Attribute**			
	Add	*Rename*	*Replace*	*Remove*
Model	testRenamed-ModelAnd-AddProperty	testRenamed-ModelAnd-RenamedProperty	testRenamed-ModelAnd-ReplacedProperty	testRenamed-ModelAnd-DeletedProperty

Table 2: Tests for all permutation types for renaming models