TUM Department of Informatics
Technical University of Munich

Master's Thesis in Informatics

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 Informatics

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

- *What is the motivation of your thesis? Why is it interesting from a scientific point of view? Which main problem do you like to solve?*
- *What is the purpose of the document? What is the main content, the main contribution?*
- *What is your methodology? How do you proceed?*

Acknowledgements

Contents

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

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 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, in particular 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.

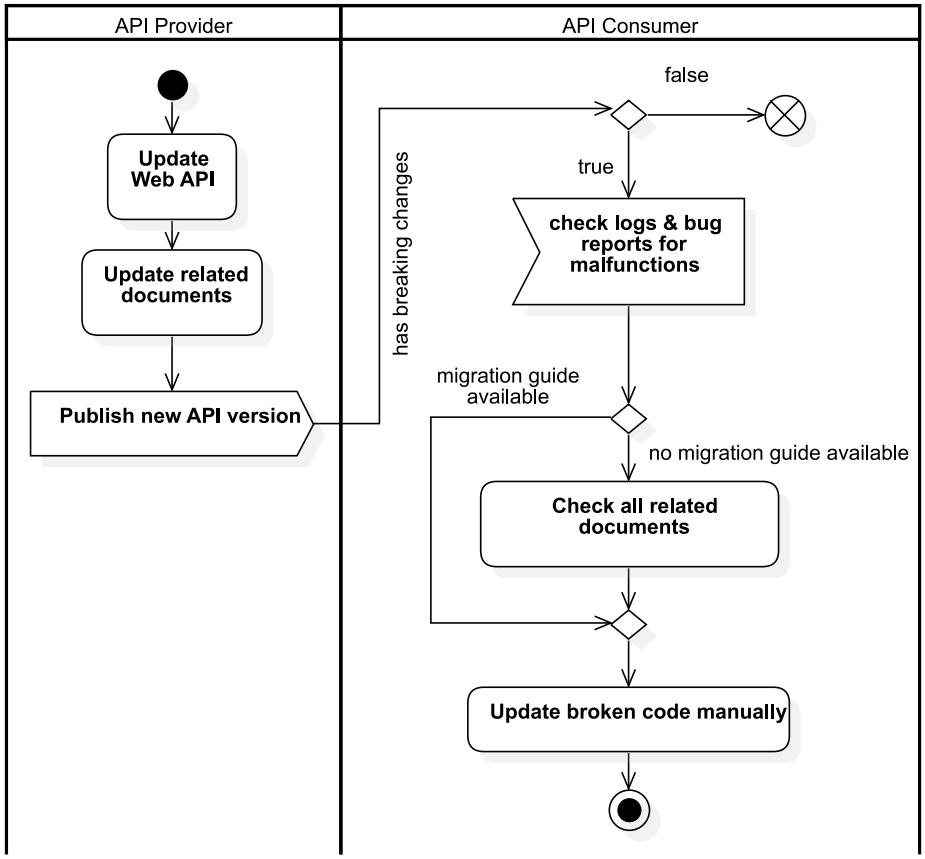

Figure 1.1: Cumbersome workflow after web service changes

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.

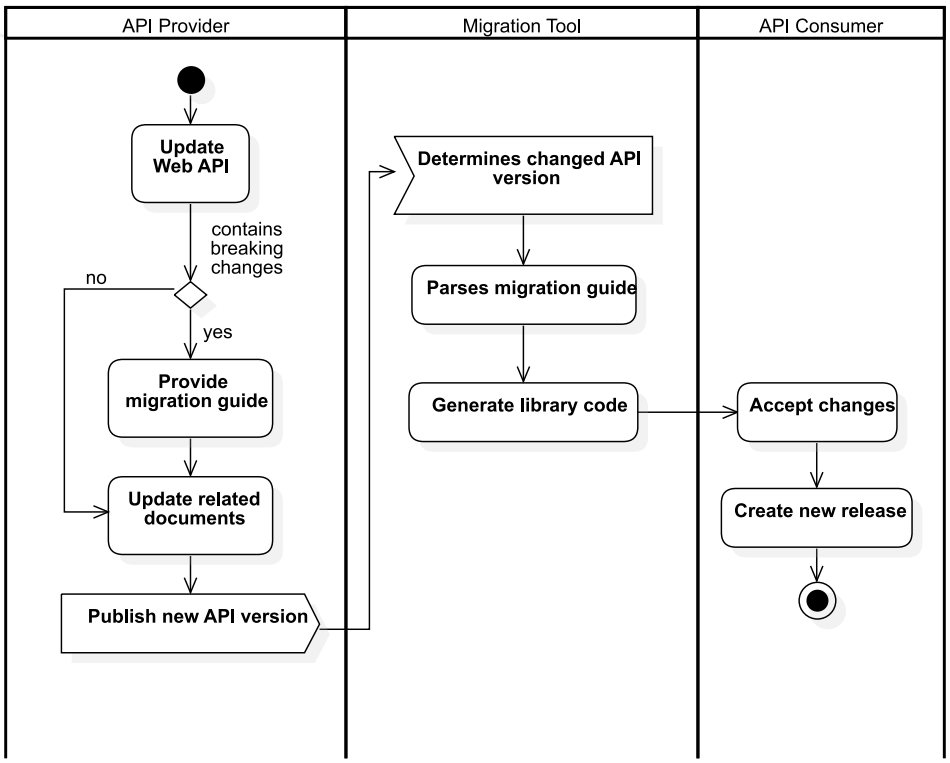

Figure 1.2: Workflow with tool support after web service changes

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.

// to be continued

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 classify 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 migratory 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 4 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 and well-defined patterns for changes help 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. They have seen the frequency of changes increase over time, which they associate with library development as they 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 backwards 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]. Hence, 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 puposes 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 non-functional 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, 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 **Developing 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 **Providing configuration:** 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 **Providing 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 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.

FR7 **Integrate output via Git:** The modified library will be published via merge request into the client application's git 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.

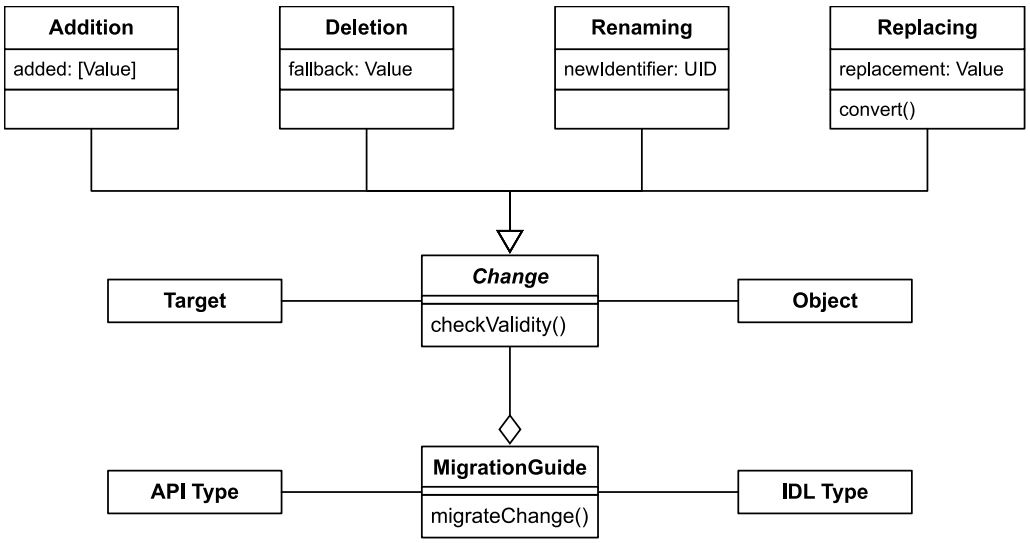

Figure 3.1: Proposed structure of a machine-readable migration guide

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

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.

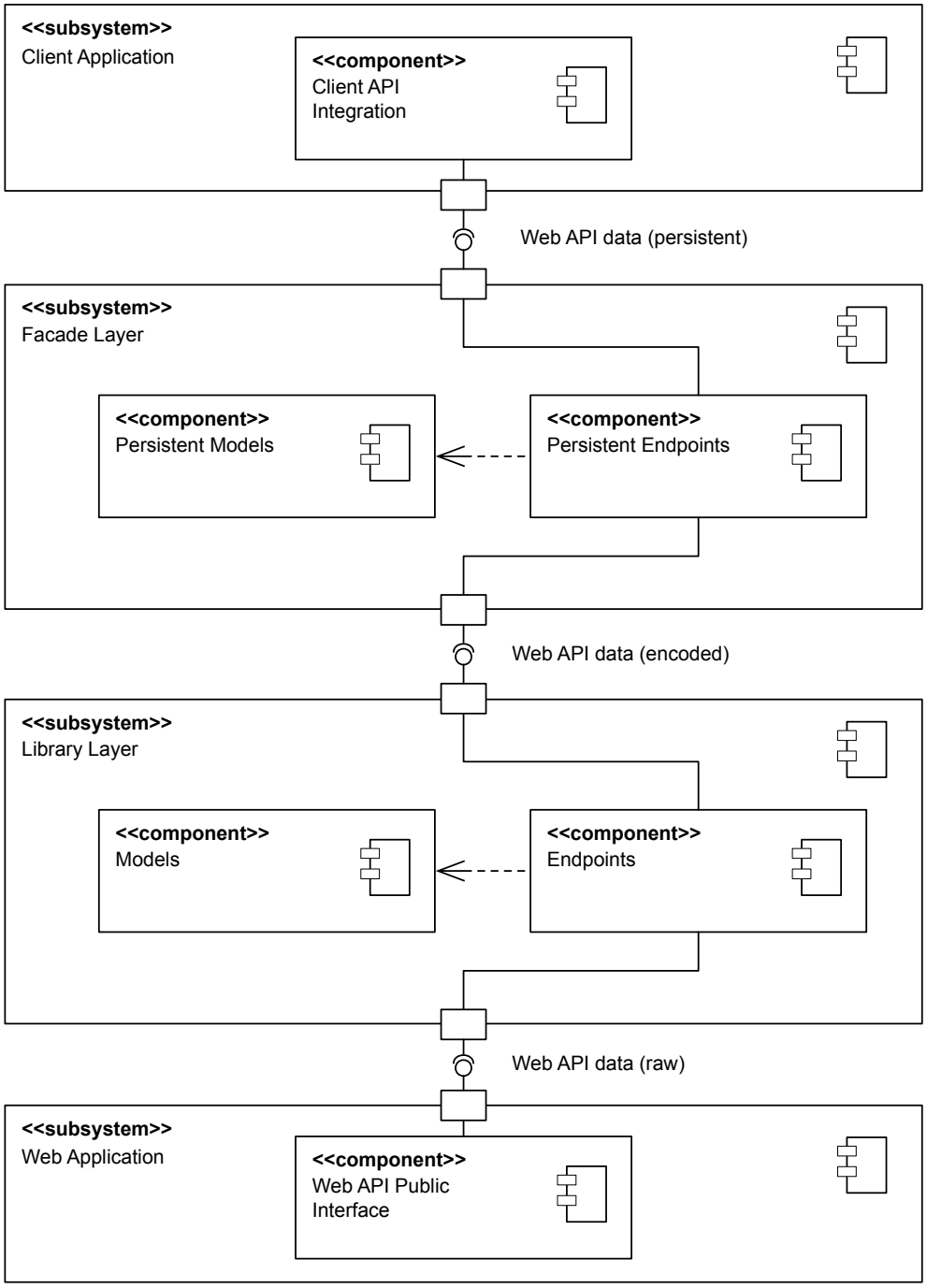

Figure 3.2: Proposed architecture of change-proof Web API integration

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 `README.md`.

NFR4 **Test coverage:** To ensure 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 in 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/

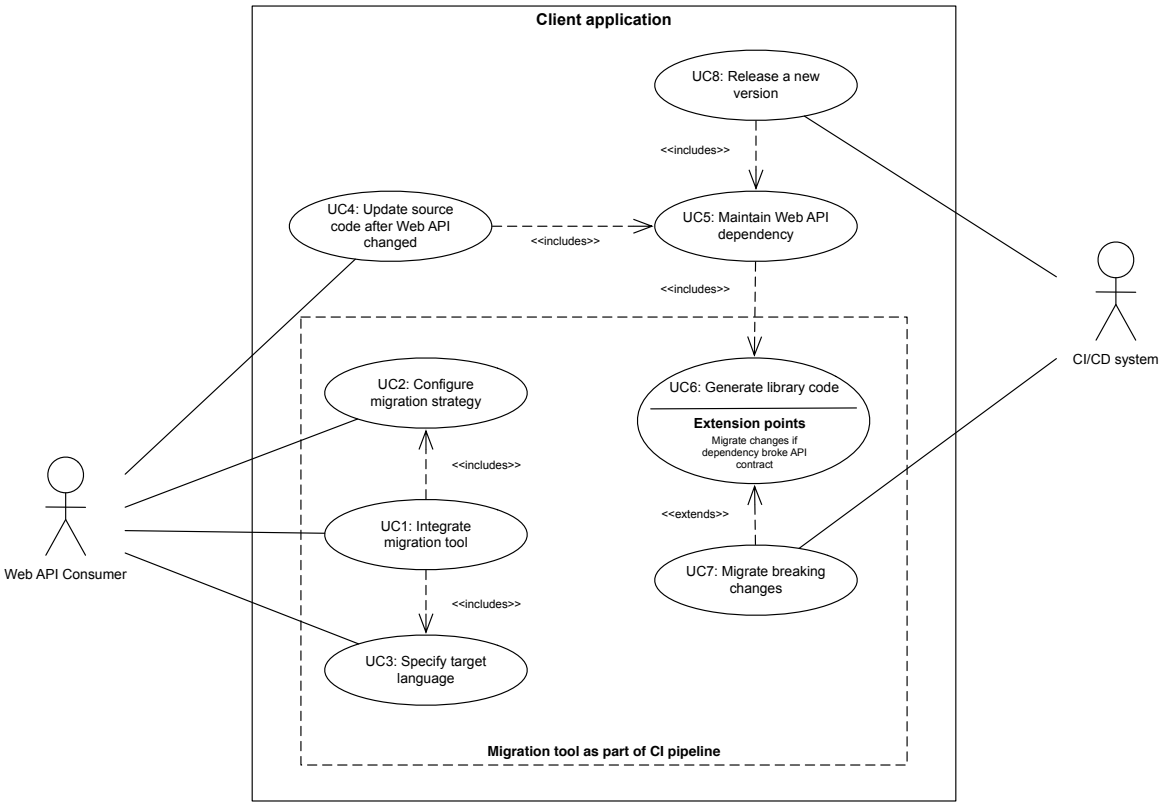

Figure 3.3: Use cases of Web API consumers

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 Command-line interface (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:** 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.

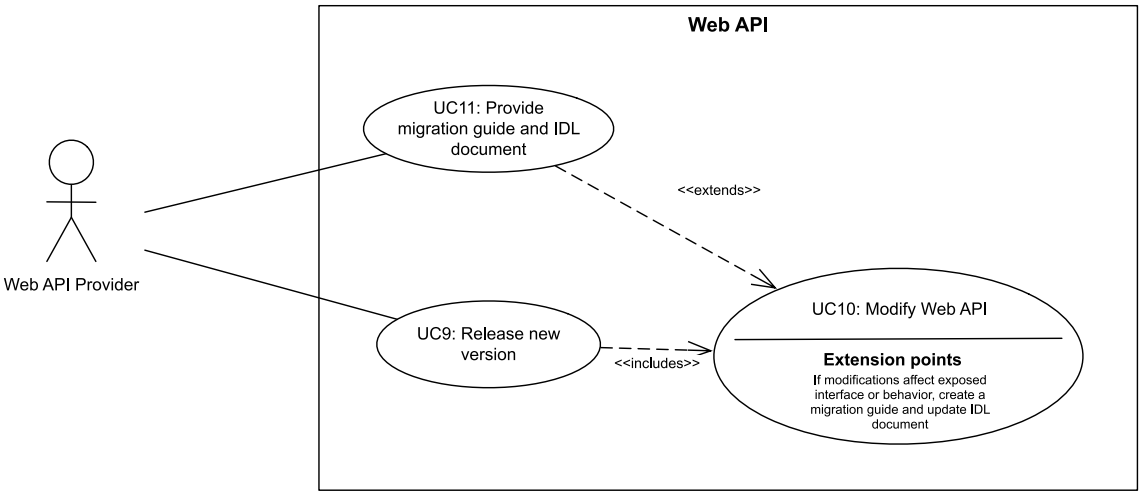

Figure 3.4: Use cases of Web API providers

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:** Provider introduced breaking changes to the exposed interface or behavior when modifying the Web API. **Postcondition:** 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. 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. Migration guide must contain the type of IDL used to describe the Web API. 3. Migration guide must contain the type of Web API. 4. Migration guide must contain both the previous and the current version number according to the semantic versioning principles. 5. Migration guide can contain a short textual summary to provide an overview of the changes introduced in the current version. 6. Migration guide must contain a list of all breaking changes introduced in the current version.
Response Measure	• Success: Migration guide can be retrieved and parsed by migration tool. • Success: 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 updated 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, with which users 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 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.

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

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 non-functional 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 4.1 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 interaction 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 in order 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 con-

sideration 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 like for example 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 and porting it to other platforms like 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 systems, 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 diffing 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.

In order to maximize the positive impact of the system, special attention is paid to its usability for Web API consumers. Non-functional 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 usabilitys. 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.

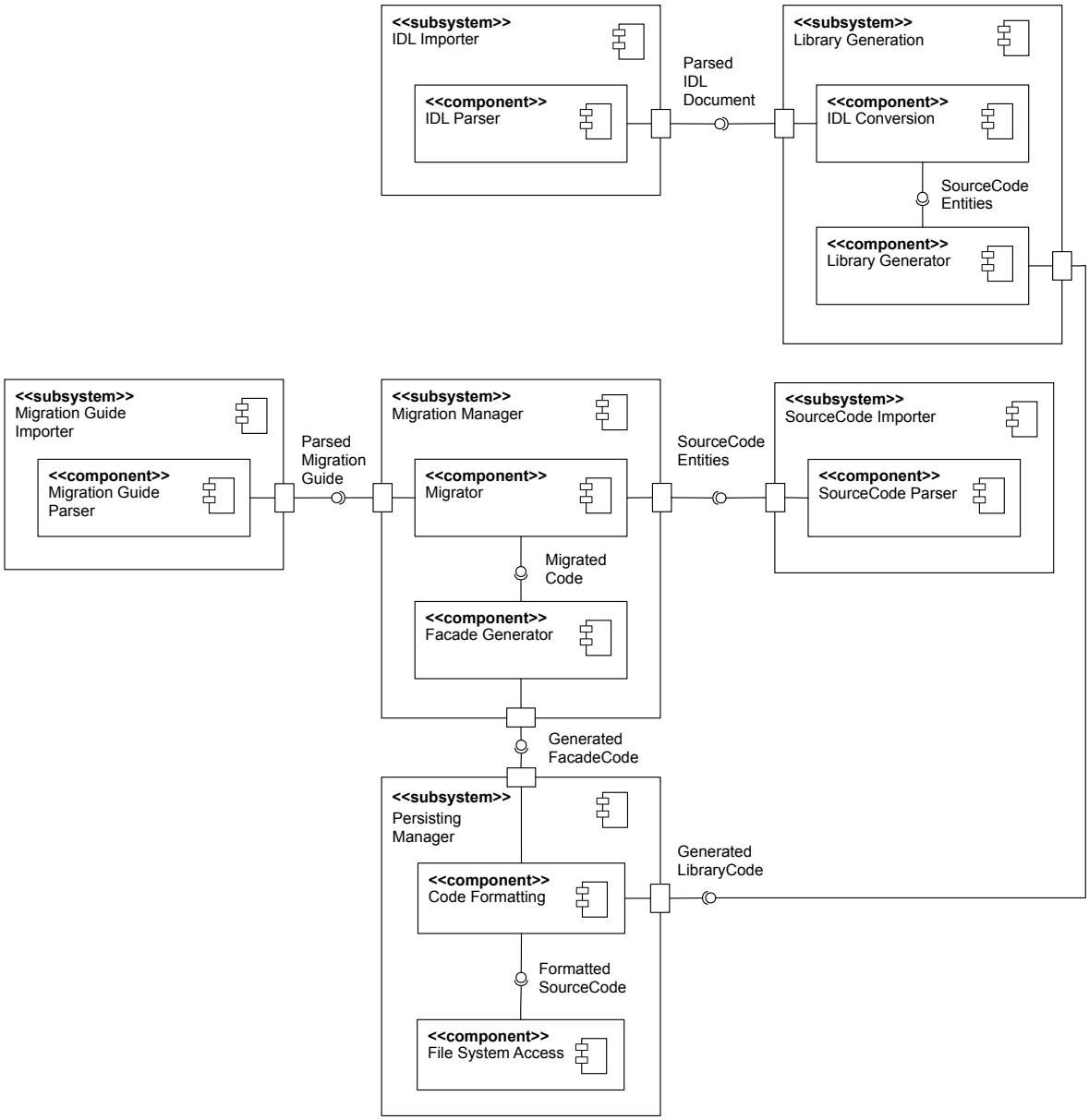

Figure 4.1: Subsystem decomposition

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. 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. Therefor, 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`.

In order 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.

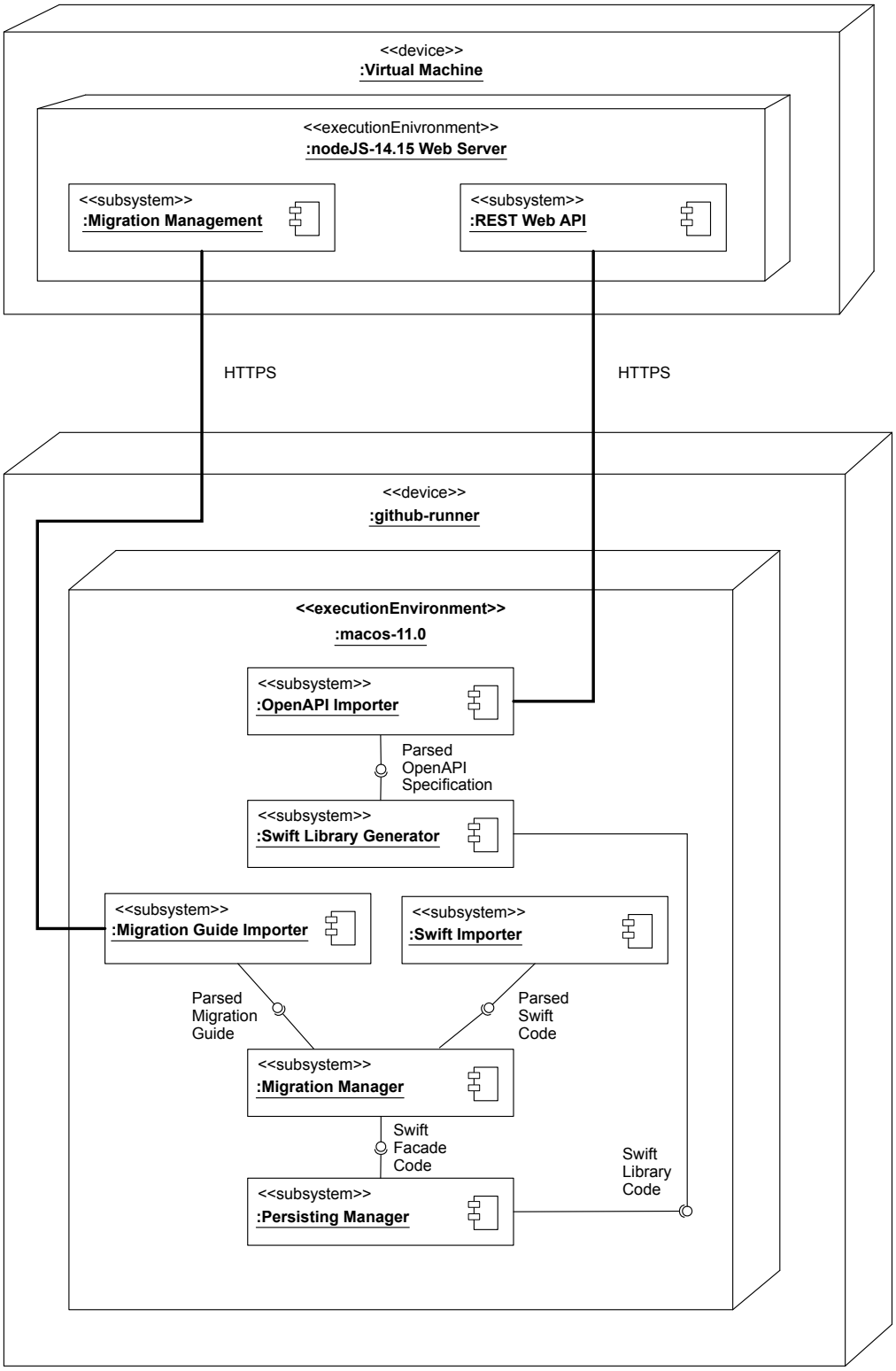

Figure 4.2: Subsystem decomposition in an Apple ecosystem

While the system itself can be impemented 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 specification 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

tbd..

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 non-functional requirement 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 integrability 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. This restricts developers from generating code in a programming language that is not supported by those tools or formatting the source code to suit their needs. 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) like syntax highlighting or auto-completion rely on them. However,

[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

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, TypeScript and more. 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 adapt 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 pattern 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.

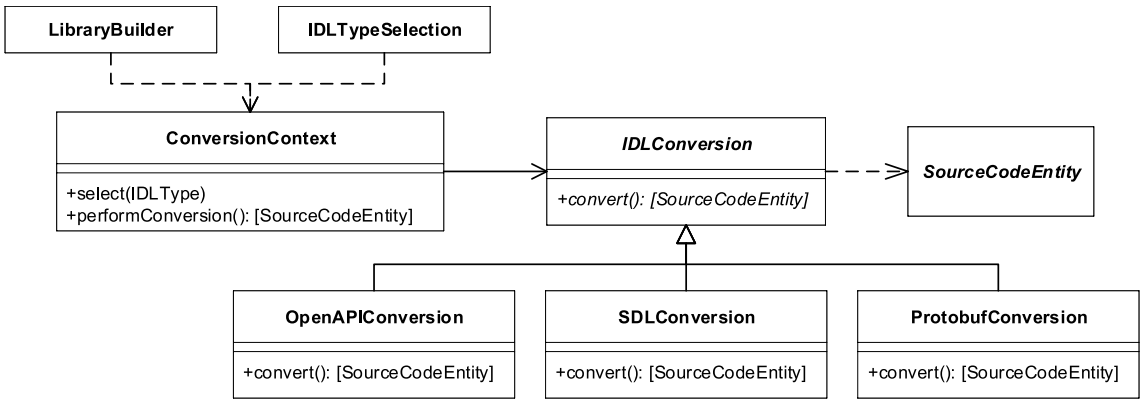

Figure 5.1: Class diagram of IDL importer subsystem

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.

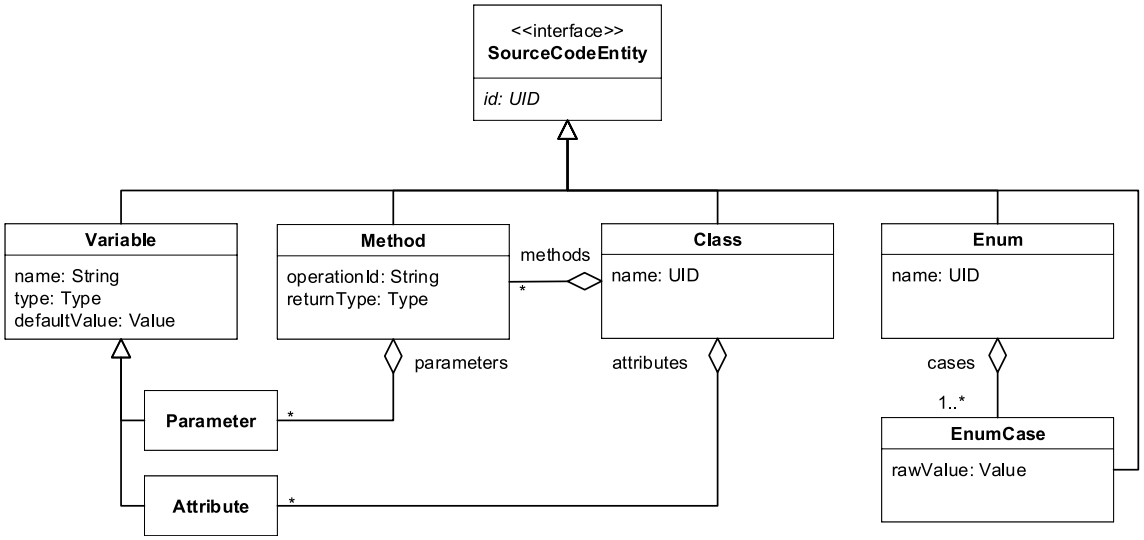

Figure 5.2: Class diagram of source code entity artifact

`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 `EnumCases`. 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.

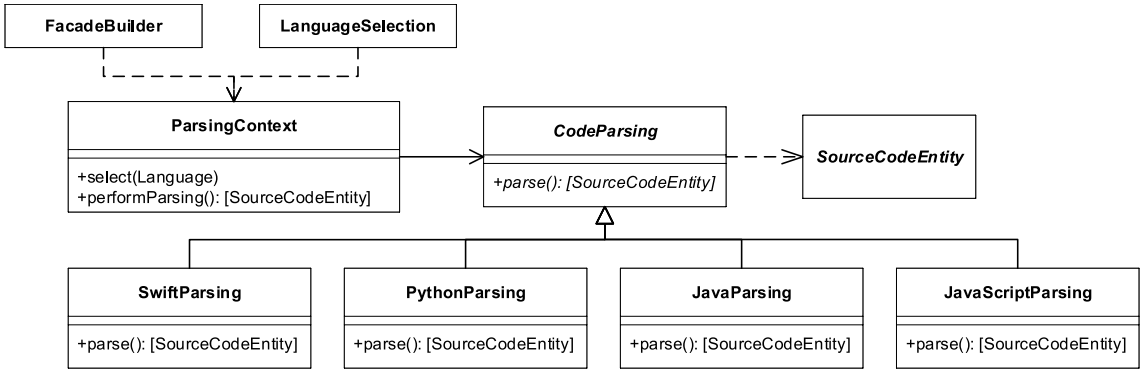

Figure 5.3: Class diagram of code importer subsystem

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 those 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.

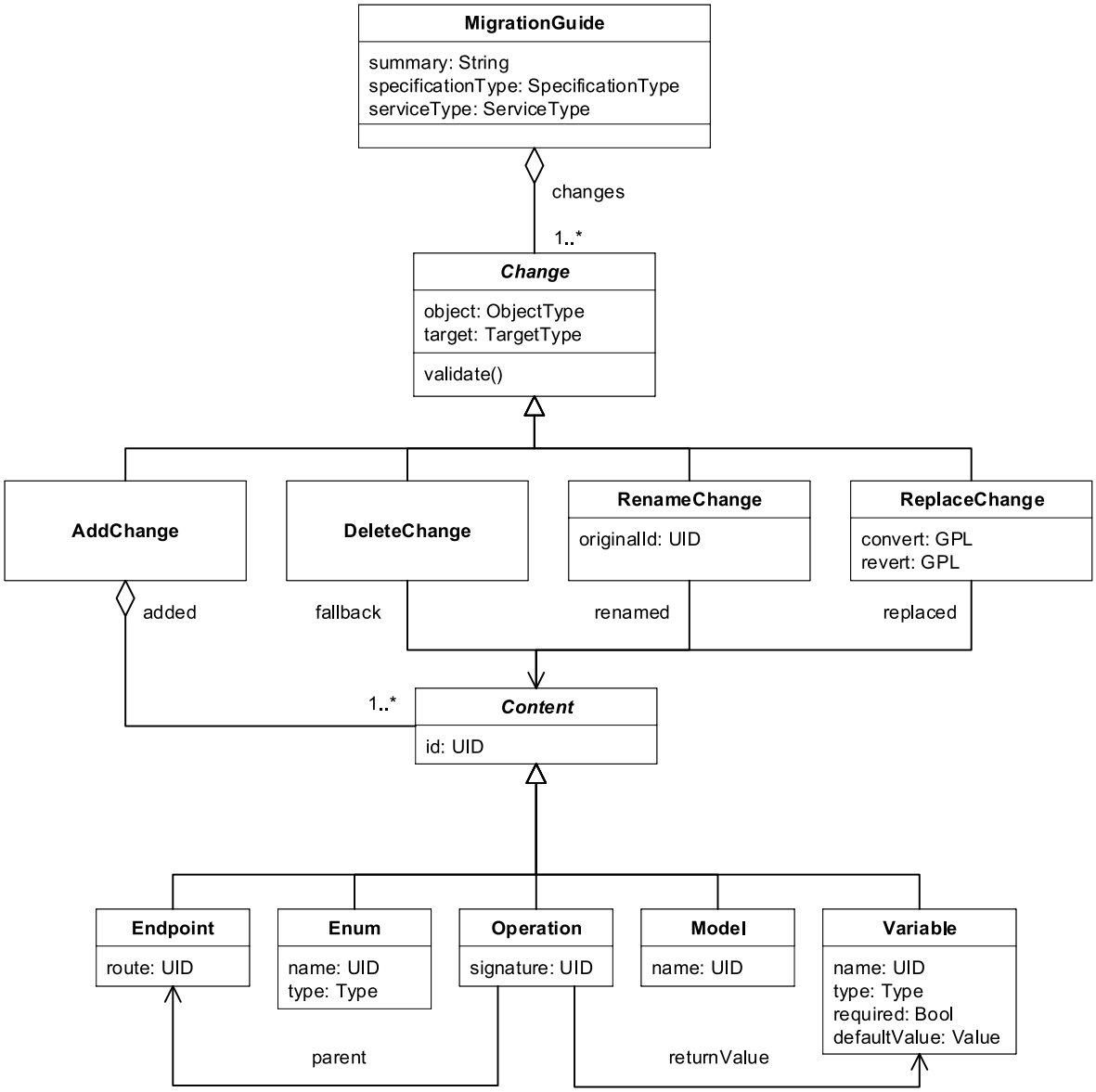

Figure 5.4: Class diagram of migration guide artifact

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 a method or methods 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 a method results in an `AddChange` for a `Method` 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.

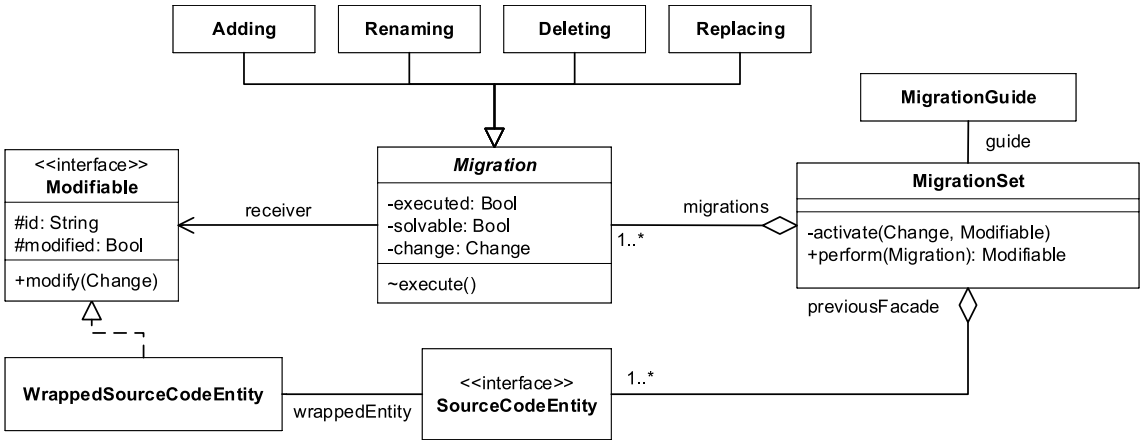

Figure 5.5: Class diagram of migration manager subsystem

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. The `MigrationSet` is the public interface of the subsystem by providing the `perform` method to perform a migration and return the adapted facade element.

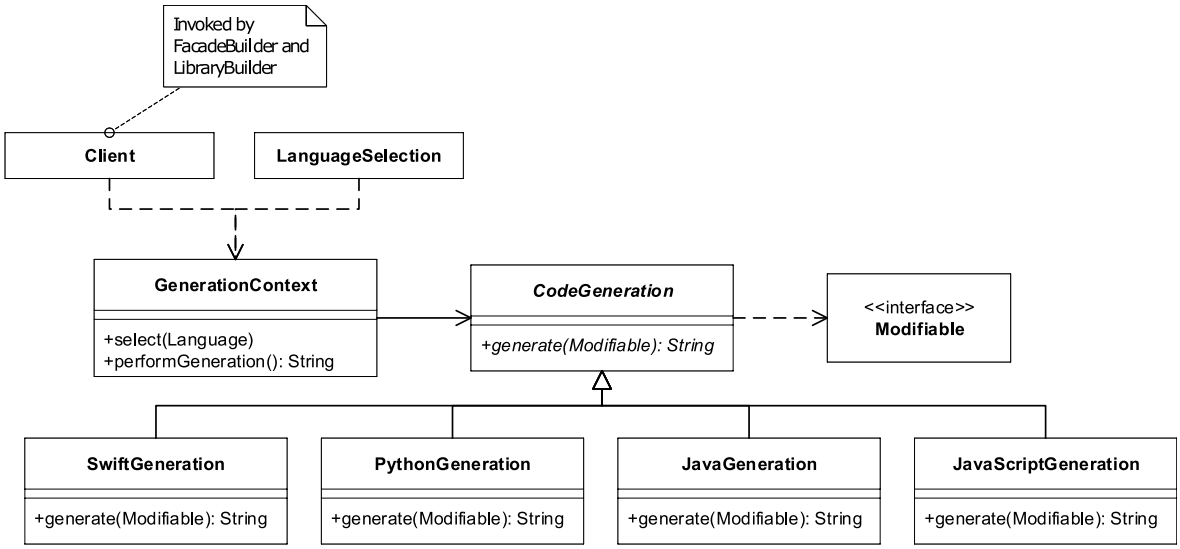

Figure 5.6: Class diagram of code generator subsystem

Generating source code from decoded entities for various programming languages requires the `Code Generator` subsystem to use the strategy pattern. Thereby, the subsystem can select the correct syntax elements during runtime. In contrast to the `SourceCode Importer` subsystem, it takes wrapped `SourceCodeEntity` 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.

The whole process is orchestrated by the `Persisting Manager` subsystem. The `Persisting Manager` takes the user input and selects the desired type of IDL and programming language. Additionally, it contains all configuration options of the user that are necessary to retrieve the Web API's IDL and migration guide.

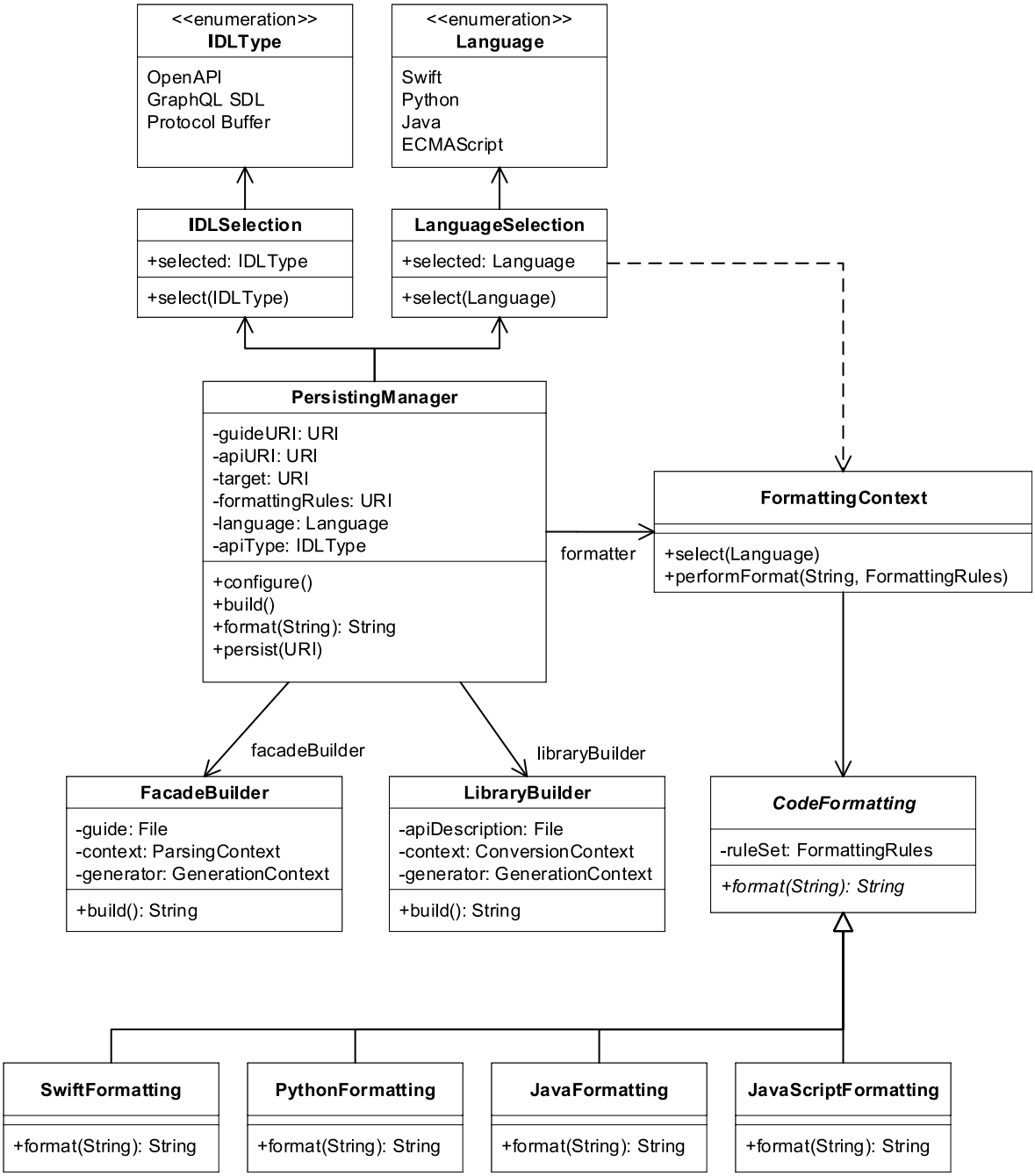

Figure 5.7: Class diagram of persisting manager subsystem

Furthermore, it contains the ruleset specifying the formatting style of the source code. After configuring the system 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 pro-

gramming 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.

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.

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

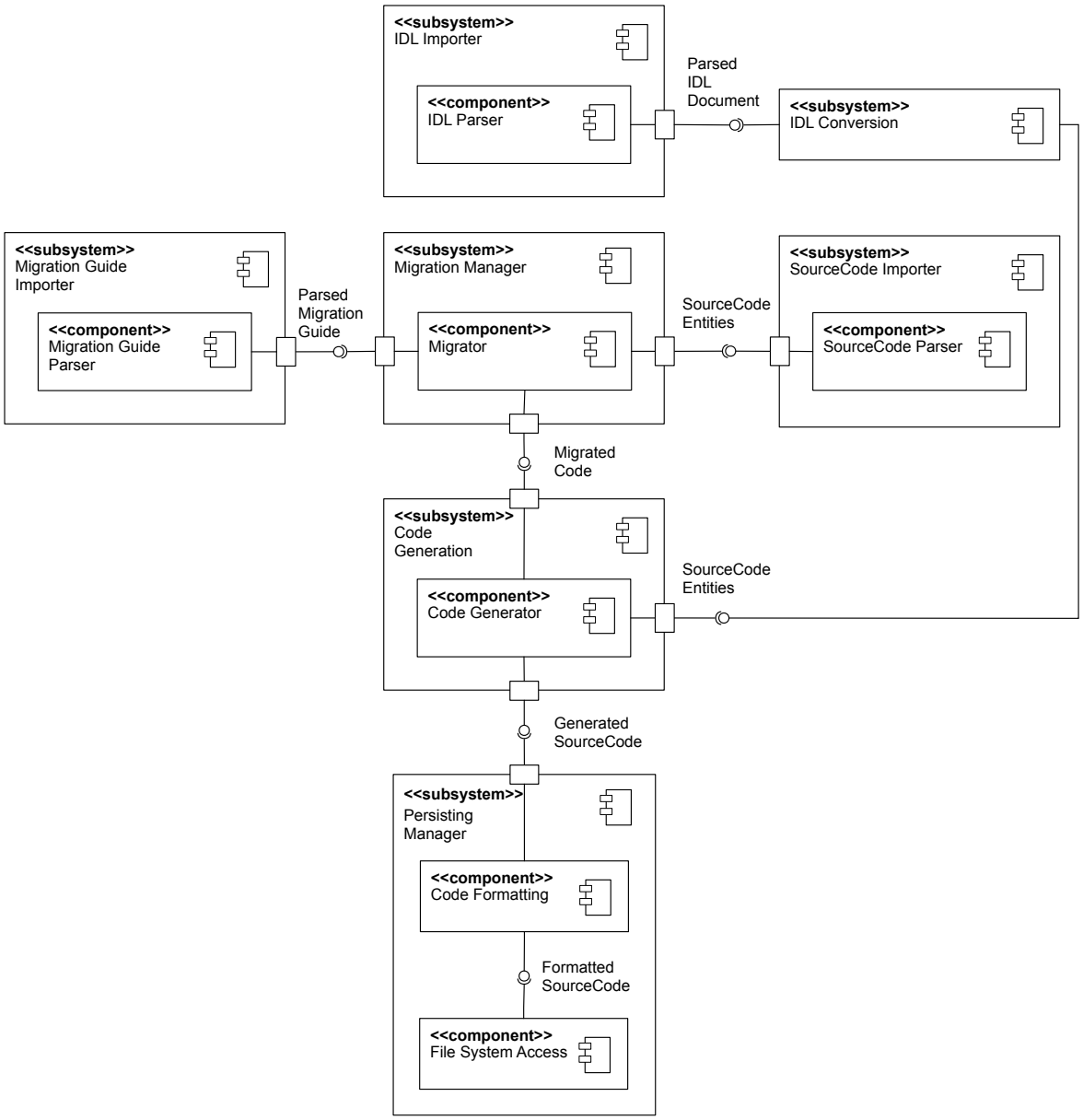

Figure 5.8: Restructured subsystem decomposition - The `Code Generation` subsystem was extracted from the `Migration Manager` and `Library Generation` subsystems to provide a uniform interface to both subsystems

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 `Source-Code 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 contains. 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.

/// part about evaluation

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. In addition to our own implementations, third-party components are integrated using the Swift Package Manager. All packages are published under an open-source license. Their source code is publicly available in their respective repositories on Github. Pallidor is composed of three Swift packages that are combined in an executable that provides a commandline user interface. The *PallidorGenerator* package implements the subsystems `IDL Importer` and `IDL Conversion`. The *PallidorMigrator* package integrates the functionality of the `Migration Manager`, `Migration Guide Importer` and `SourceCode Importer` subsystems. The *Pallidor* package contains the commandline user interface and the `Persisting Manager` subsystem. For the individual components, we identified several open-source Swift packages that provide the required functionality.

Subsystem/ Component	**Swift Package Name**	**Swift Package Version**	**Swift Package Author / Publisher**
IDL Importer	OpenAPIKit	2.0.0 (Sep. 28th, 2020)	Mathew Polzin
Source Code Importer	Sourcery	1.0.0 (Aug. 14th, 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

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`[1] framework, the client application's execution environment must be at least iOS 13 or MacOS 10.15.

6.2 Evaluation

How we evaluated it. using 3 web apis and perform migration. all hand-made

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

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

List of Tables

Listings

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.

[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 Example