Master Thesis

Master of Science in Engineering

Semantic Versioned SDK Generation: Streamlining Cross-Language Development

Kristian Jørgensen
The Faculty of Engineering
The Maersk Mc-Kinney Moller Institute
University of Southern Denmark

Supervisor: Torben Worm - tow@mmmi.sdu.dk
Co-supervisor: Aslak Johansen - asjo@mmmi.sdu.dk

June, 2024

Abstract

This project aims to automate the conversion of Data Transferable Objects (DTOs) between different programming languages, eliminating a time-consuming task for developers and reducing errors that may occur in manual conversion. The approach involves creating a schema that details the structure of all DTOs, which can be converted into Software Development Kits (SDKs). This schema can be auto-generated by a Schema Provider, compared to previous versions by a Diff Checker, and converted into an SDK by a Schema Consumer.

The report presents a prototype demonstrating this approach's feasibility. While the prototype shows promising results, further work is required to adapt this solution to real-world scenarios. Overall, this project has the potential to significantly reduce development time and effort, and minimize bugs associated with manual DTO conversion.

Contents

Vocabulary

- **DTO**: Data Transferable Object
  The definition for a strongly typed object. Often referred to as a *Plain Old (Java / C#) Object* in each retrospective language.
  Synonyms: Data Structure, Model

- **SDK**: Software Development Kit
  A dependency other developers can include in their project, to gain access to code created by others. This could include DTOs and services to communicate with their API.

- **API**: Application Programming Interface
  A standardized way to communicate with an application.

- **Project**: A code repository, containing a single namespace.

- **Solution**: Like a *Project*, a solution is a code repository but contains multiple projects instead of just a single one.

1 Introduction

Modern software systems are often composed of different languages. This is especially true for a microservice architecture. To effectively communicate between these services, developers often use Data Transferable Objects (DTO), which describe exactly how the data structure looks. These DTOs can be used in a RESTful API, in a message bus, in socket communication or much more. This means they are widely used, and lay the standard for the data. When developing within the same language, a Software Development KIT (SDK) can be created and published, allowing other team members to have access to the same DTO, ensuring they have all properties defined in their code base, and can access it all. However, when switching between languages, this is no longer possible, as they cannot use an SDK for another language. To solve this, developers often undertake the task of duplicating the classes in multiple languages, having a strictly typed SDK for each language they use in development. This project aims to eliminate this manual conversion task.

1.1 How the DTOs are used in code spaces

A DTO is essentially a class structure, where properties are defined. They do not include any business logic, which is commonly associated with classes in Object-Oriented Programming (OOP). Instead, they focus solely on encapsulating data. Listing 1 shows an example of two DTOs, defined in C#

Listing 1: Example of a C# DTO

```
1  public class PersonDto
2  {
3      public string FirstName { get; set; }
4      public string LastName { get; set; }
5      public DateOnly Birthday { get; set; }
6      public string AddressDto { get; set; }
7  }
8
9  public class AddressDto
10 {
11     public string Road { get; set; }
12     public int Number { get; set; }
13     public int ZipCode { get; set; }
14 }
```

When a DTO has been defined, it can then be used in a project. Listings 2 and 3 show respectively how DTOs can be used in C# or TypeScript when calling a RESTful API. This is only one of the uses for these DTOs. In practice, they can be used everywhere data is exchanged, where the structure of the data is guaranteed.

Listing 2: DTO used in a GET request in C#

```
var user = await httpClient
    .CreateRequest($"localhost:3000/users/{userId}")
    .Get<UserDto>();
```

Listing 3: DTO used in a GET request in TypeScript

```
let user = await fetch(`localhost:3000/users/${userId}`)
    .then(res => res.json())
    .then((res: UserDto) => res);
```

1.2 Real-life use cases from stakeholders

For the project to succeed, it must be viable in a real software system. To test this, the solution is developed in collaboration with TicketBot and Digizuite, which both use ASP.NET in their backend development, and both need a TypeScript SDK to develop their frontend and publish to customers.

1.2.1 TicketBot

TicketBot is a small organization, with three developers, utilizing a microservice architecture. The backend is developed both in C# and TypeScript, with shared DTOs both for the REST API and message bus. This means there are a lot of duplicate classes in both C# and TypeScript. We would like to have a single source of truth that can be automatically converted into TypeScript when pushing an update. The project should therefore be able to convert a series of classes, not only limiting itself to the classes that are not exposed in the API. As a result, it should be able to create both an internal, which the developers can use internally, and a public SDK that customers can use. To do this, it should be able to differentiate between internal and public endpoints, preferably by checking against the required authorization. To sum up, the project must be able to generate multiple SDK from a single source. A public SDK, that includes all classes used in the public part of our API, an internal SDK, that

includes everything used by the internal API, and finally a message bus SDK, that contains all the classes used in the bus.

1.2.2 Digizuite

Digizuite is a large-scale enterprise, providing a Digital Asset Management (DAM) solution, used by large corporations. This means Digizuite has a large number of assets in its system. All of these assets have a set of metadata assigned. This metadata is assigned in a complex metadata structure, having text fields, number fields, tree structures, and much more. This results in an advanced DTO structure, which includes both polymorphism and generic models. Digizuite uses this API both internally and exposes it to clients, as many of them build on top with their own integrations. Currently, they provide a public TypeScript and C# SDK, however, these SDKs do not provide full coverage but are instead updated on a "we need to use this internally now" basis, resulting in a deficient SDK.

Due to this complex model structure, and the demand for a public SDK, Digizuite proposed this project, with a little twist. Instead of only generating the SDK, Digizuite would also like the ability to detect when a breaking change occurs in an update. The reason for this is that due to the complex structure, it is not always clear to developers what models are exposed, and therefore cannot be changed, and which models are internal, and allowed to be changed. As we would need to analyze the project to create the SDK, this could be the feature to make the project stand out.

1.3 The hypothesis

By understanding this issue, the following hypothesis is proposed:

By automating the conversion of data objects between languages, developers can both save time and reduce bugs, when developing an SDK.

To validate the hypothesis, the following sub-hypotheses are proposed:

- Automating the conversion of DTOs will greatly reduce the time spent developing multiple language SDKs.
- Automated conversion will remove all possibility of manual human error

The hypothesis will be evaluated based on an initial analysis, followed by testing a prototype of the proposed solution.

2 Related Work

OpenAPI

When describing an API, the de facto standard today is OpenAPI [1]. This is a standard, that can fully describe an API, including both type definitions, endpoint information, authentication, and more. However, while the standard covers everything, a lot of knowledge about the type structure is lost. It supports neither polymorphism, inheritance, or generics. This can result in duplicate code, as each type will be defined multiple times, with a few changes in fields. While having this disadvantage, OpenAPI is still one of the most commonly used to describe an API. This gives it some advantages, as many third-party tools build on top of this, e.g. Swagger [2] for documentation, automatic OpenAPI specification generation for multiple languages [3] [4] [5], and more.

Kiota

Kiota [6] generates multiple SDKs, supporting a variety of programming languages. This is done, by using the OpenAPI definition and creating a SDK based on that. As there is a lot of tools to convert a project to an OpenAPI specification [3] [4] [5], this solution should be able to convert most existing projects. The auto-generation Kiota performs is very similar to what this project aims to achieve, however, as discussed with OpenAPI, this does come with some limitations. In some projects, these limitations may be negligible, and in other applications, may be the reason the solution is not chosen.

ElastichSearch SDK generating

ElasticSearch has already developed a tool similar to the proposed tool, and presented it at NDC London 2021 [7]. However, instead of creating SDKs for multiple languages at once, they develop a language-specific SDK for C#. Their tool is not public, and can therefore not be studied further than the presentation, which focused on promoting a specific Microsoft tool. However, relevant parts were still included in the presentation. An example of this is how they describe their API and models. As a large-scale JSON document is deemed too complex to manage, they opted for a TypeScript solution, describing all their endpoints and datatypes. Furthermore, these are enhanced with comments, further describing the API. For example, they show *@since* and *@stability*, which is metadata about the class. The example talked about can be viewed in Listing 4. The code is automatically converted to a JSON scheme,

which can be ingested by other applications, such as they present with C#, or converted to other specifications, such as the OpenAPI standard.

Listing 4: ElasticSearch model definition

```
/**
 * @rest_spec_name search
 * @since 0.0.0
 * @stability stable
 */
export class Request extends RequestBase {
    path_parts: {
        index?: Indices
    }
    query_parameters: {
        allow_no_indices?: boolean
        ...
        size?: integer
        from?: integer
        sort?: string | string[]
    }
    body: {
    /** @aliases aggs */
    aggregations?: Dictornary<string, AggregationContainer>
    collapse?: FieldCollapse
    /**
    * If true, returns detailed information about score computation as part of the
        hit
    * @server_default false
    */
    explain?: boolean
    }
}
```

3 Analysis

3.1 Is there any need for the proposed solution?

In general, there are two types of code duplication; Code duplication within the same language, and code duplication across multiple languages. The latter is needed when developing a cross-language solution and needing DTOs for each language. Often a DTO is created in the native language for the API, and then copied to other languages that may use it. One of the main features of the proposed solution is to eliminate manual code duplication across languages. When converting code, there is a high chance of producing human errors [8], either by missing a property, making a typo, or other things. However, the chance of errors is not the only aspect the solution aims to reduce. Converting code is also a time-consuming task, that could be eliminated if an automatic tool is introduced.

By automating the conversion, and automatically generating the DTOs for other languages, the chance of errors can be removed, and it is estimated time to market is reduced for developers, as they no longer need to maintain multiple projects.

3.2 Time and error with manual conversion

A manual DTO conversion task is created to validate the solution's need. In this task, developers will convert three projects, and the results will be analyzed. The time it takes individuals to convert all the DTOs in all projects will be examined, and the number of errors that occur in the final result will be counted. Each individual that participates will receive the same DTOs, and be asked to convert them to either Java or TypeScript. Each of the three projects relates to a separate domain and complexity, and all contain several DTOs. Table 1 lists the projects, including how many files, classes, and properties there are.

Project	*Files*	*Classes*	*Properties*	*Properties pr class*
Person Example	4	4	12	3
ElasticSearch Example	4	10	54	5,4
Setting Example	5	16	93	5,8
Total	13	30	157	5,2

Table 1: Lines of code in example solution

By dividing the time total time it takes to convert all the DTOs with the amount of properties,

we can create an estimate of how long it takes to convert a single DTO. This representation does not portray every developer but will give a baseline, which will serve as the groundwork for this project. It is also important to note, that this is only an indication for creating new DTOs, and not updating existing ones. If a DTO evolves, the model will need to be updated in all projects. While this may not take a long time, it is a task that the developer may forget. The possibility of this oblivion is not showcased in the examples, but reaming extremely relevant.

When testing how long it takes to convert DTOs, three people, *Person A*, *Person B*, and *Person C*, have agreed to help. The procedure is to follow a pre-defined guide listed in the solution and record the entire conversion. The guide can be viewed in Appendix C.

	Person A	*Person B*	*Person C*
Target language	Java	Java	TypeScript
Time spent	49m 38s	1h 1m 07s	44m 12s
Person	8m 28s	12m 37s	9m 24s
ElasticSearch	18m 27s	20m 45s	13m 48s
Setting	22m 43s	27m 45s	21m 0s
Avg time per property	18,97s	23,55s	16,89s
Total errors	3	157	0
Invalid casing	1	0	0
Invalid syntax	2	0	0
Invalid nullability	0	157	0

Table 2: Manual DTO conversion results

Table 2 shows that the average time per property is 20,07 seconds, and errors are very likely to occur, even by experienced developers. To better understand the data, it is important to know the experience, domain knowledge, and post-mindset of each individual.

- **Person A**
  Experience: Has multiple years of experience converting DTOs.
  Domain knowlegde: An expert in the domain. This person created all the DTOs.
  Approach: Copying the DTOs into the IDE, and converting the syntax.
  Result summarized: Even with much experience performing the task, the developer made multiple errors. Many of the errors were cought when the developer validated the results, however, a few errors slipped through, and ended up in the final result.

Post Conversion Mindset: "DTO conversion is one of the most boring tasks to perform. To entertain myself while doing it, I am always watching a movie or similar while converting."

- **Person B**
  Experience: Has never converted a DTO before, but has some experience developing in the chosen target language.
  Domain knowledge: None, has never seen the DTOs or similar DTOs before.
  Approach: Utilizing an AI, in this case, Copilot Prompt to automatically convert the DTOs, by copying one class at a time, and asking it to convert to Java.
  Result summarized: As the AI did not understand the requirements, it failed all nullability checks. Furthermore, it imported additional packages to handle JSON, even though the project already specified a way to do this. So, while the classes itself has no errors, and is fully functional, this approach did come with other issues.
  Post Conversion Mindset: "The experience was alright, but I probably would not do it without Copilot."

- **Person C**
  Experience: Has multiple years of experience converting DTOs.
  Domain knowledge: Proficient, has not seen the DTOs beforehand, but has experience with familiar types.
  Approach: Utilizing an AI, in this case, GitHub Copilot, to automatically convert the DTOs. This was done by copying the class into the file, and letting Copilot suggest one property at a time. After all properties were converted, the invalid C# code was deleted.
  Result summarized: As the TypeScript project was new, the developer used some of the time to research a JSON mapper and figure out the approach. This was done when converting the first project *Person*. Afterward, the developer slowly converted one property at a time and validated the result Copilot gave. This resulted in no errors found when validating the final result.
  Post Conversion Mindset: "A very boring and time-consuming task, where errors can easily occur because one tries to get it done quickly. If you can avoid it, that would certainly be advantageous".

It becomes clear that the task of converting DTOs is a dull task, without much stimulation for the developer. The use of AI tools, in this case Copilot helps, but still leaves the task un-

pleasant. The developers all further claimed, that should they continue, the error rate would most likely start increasing, as the lack of stimulation decreases the effort provided over time.

To put the data into a real-life scenario, an examination of all TicketBots properties is performed. Based on this examination, it is estimated that TicketBot currently has 431 properties it exposes in its API.

$$431 \text{ properties} \times 20,07 \text{ seconds/properties} = 2h\ 24m\ 10s \tag{1}$$

Based on this, we can see in Equation 1, that it would take shy of 2,5 hours to convert all DTOs from TicketBot. This may not seem like a lot, however, this number is just for a one-time transformation and assumes there will be no errors that have to be later. As TicketBot is currently under rapid change, these DTOs often change, and would therefore have to be updated often. Furthermore, these are only the properties exposed in the API. TicketBot also uses DTOs for their message bus, which is not counted here. This is due to this part is still in development. It is estimated that these DTOs would reach the same number of properties, as those counted, resulting in double the amount of properties, and thereby shy of 5 hours needed to convert all properties. The same examination has not been performed with Digizuite, due to the complex structure of the system. When compared by an employee to TicketBot, it is estimated that there easily could be over 10x more properties, with the majority currently being unsupported by their public SDK.

3.3 Possible approaches to create a solution

While there are multiple solutions on the market today, which all explore the same options as the proposed solution, there is no direct match. This shows that there is a potential gap the solution can help fill. There are different ways to implement the solution. We will explore the following.

- **JSON definition**: Like OpenAPI, a definition could be created, that describes all the models. It could take inspiration from OpenAPI, and build on top on that.
- **Language specific definition**: Like ElastichSearch, a standard could be created in a specific language, and from that standard convert it into SDKs.
- **Convert code directly**: Taking the source code directly from the projects, and turning them into SDKs.

3.4 Advantages and disadvantages of each approach

JSON definition

There is a lot of existing documentation for this solution, as inspiration can be drawn from OpenAPI. Expanding on this foundation, it would not take long to complete the definition. However, the new definition would not be compatible with the original OpenAPI definition, as data models now should support polymorphism, and therefore lose inherited properties. Furthermore, managing a large-scale JSON definition manually can be cumbersome. To overcome this task when using OpenAPI, different tools are often used to automatically convert a solution to an OpenAPI definition [9].

Language specific definition

The approach utilized by ElasticSearch solves the issue of the unmanageable JSON document. With data structures sorted by folders and files, it matches the real repository. Furthermore, the users can make use of their IDE's code suggestions and validation, allowing for a quicker and more pleasant development experience. As this code repository would be responsible for being a single source of truth for multiple languages, specific annotations could be created, even if the annotation was designed for a specific language. This could be relevant for renaming a data structure different for a specific language, should the name be a reserved keyword. The major downside of this approach is that all models from existing codebases will need to be rewritten in the new repository. This can potentially prevent existing code spaces from adopting the solution.

Convert code directly

By looking directly at the code for a project, and directly converting it to SDK's, the time needed for creating an SDK is completely removed. This would be a fast way to create the SDK, as it is generated directly from the project. However, this method is not without fault either. By utilizing this approach, the SDK creation will only be available for projects written in a single language. If the solution should be expanded to other languages, the entire SDK generator would have to be rewritten in said languages.

Combining approaches

As stated, there is not one perfect solution. There are upsides and downsides to each approach. The most unlikely combination is it combines a language-specific definition with direct conversion of the code. The reason behind this is the fact that there is no point in having a language-specific definition, if the models still are created in the native language, and then converted. This would in practice just be a form for an SDK converter in itself. The other two approaches is to combine either *Language specific definition* or *Convert code directly* with the *JSON definition*. This gives the advantage of having a well-defined JSON schema, without the cumbersome task of manually managing it.

3.5 The semantic difference checker

Alongside converting DTOs, it is proposed to analyze the models, in order to further reduce the risk of errors when updating the code. The specific suggestion is to have a master schema, and with every update validate if there are any breaking changes between the newly generated version and the master version. This would help make the DTOs more robust, as it could help to reduce unintentional human errors.

3.6 Requirements for the solution

Based on the analysis, some requirements have been defined, which are defined in Table 3. Each requirement has a category, which is referred to as an iteration, with each sub-requirement being a task in that iteration. This is elaborated in section 4. The requirements are not ordered in any specific way.

#	*Title*	*Description*
1	Default Class	Being able to convert a plain DTO.
1.1	numeric	int32, int64
1.2	float	float, double
1.3	Other types	boolean, char, string, etc.
2	Generics	Add support for generic types.
2.1	Generic Class	A class should support a ¡T¿ generic type
2.2	Generic Property T	The class should be able to include property of type T, e.g. *public T {get;}*
2.3	Generic Property <T>	Generic properties should able to indicate the implementation of T, e.g. *public List<T> {get;}*
3	Collections	Being able to convert different kinds of collections.
3.1	Array	While not being directly a collection, arrays are included in this step
3.2	List	All implementations of list types
3.3	Set	All implementations of set types
3.4	Map	All implementations of list/ dictionary types
4	Polymorphism	Add polymorphism, and allow classes to be abstract and extend from another.
4.1	Abstract class	If the class is marked as abstract
4.2	Extend class	Which class the class extends
5	Inheritance	Add support for interfaces, and allow classes to implement interfaces
6	Settings	It should be possible for developers to configure the behavior to their liking.
6.1	Casing	As the product is only a prototype, the only setting that is showcased, is the option to select what casing should be used.
6.1.1	snake_case [10]	
6.1.2	camelCase [11]	
6.1.3	PascalCase [12]	
7	Diff Checker	There should be created a semantic version difference checker.
7.1	Added types	It should be able to see newly added types.
7.2	Removed types	It should be able to see removed types.
7.3	Modified types	It should be able to detect changes in a type.

Table 3: Requirements

4 Design

When reading both section 4 and section 5, it should be acknowledged that the outline of the report represents a linear approach. This is however not how neither the prototype nor the report was written. Instead of following a linear approach, an agile approach with several iterations was used. These iteration cycles follow an iteration of the requirements, as defined in Table 3. Each iteration includes its own minor design and implementation phase. Likewise, the iteration may change both content of the report and implementations, created in earlier iterations. The result of all these iterations leaves the final prototype.

Based on the analysis, a three-part prototype is proposed.

- The first part will consist of transforming DTOs into schemas, and therefore be referred to as a *Schema Provider*.
- The second part will consist of transforming schemas into SDKs, and be referred to as a *Schema Consumer*.
- The third is a version difference checker, and the called *Diff Checker*

The overall design of the proposed prototype can be viewed in Figure 1. Furthermore, the design of each part will be explored in the following sections.

4.1 Creating a schema definition

The backbone of the entire prototype is the schema. DTOs will be converted into schemas, and SDKs will created from schemas. The purpose of this schema is to define the structure of the DTO, and potentially act as the single source of truth. While iterating over the design, the schema has changed format multiple times. Each of these designs will be referred to as a phase. Before the final schema definition is explained, the phases leading to the final definition are explained.

- **Phase 1: File format**: The decision of the file format. The two proposed formats are YAML and JSON. As YAML is both more human readable, and a superset of JSON [13], this format was chosen, while still leaving the option for users to use JSON, if wanted.
- **Phase 2: List vs Map**: When defining the models, this phase went back and forth between whether the models should be in a list, with each model as an entry, or in a

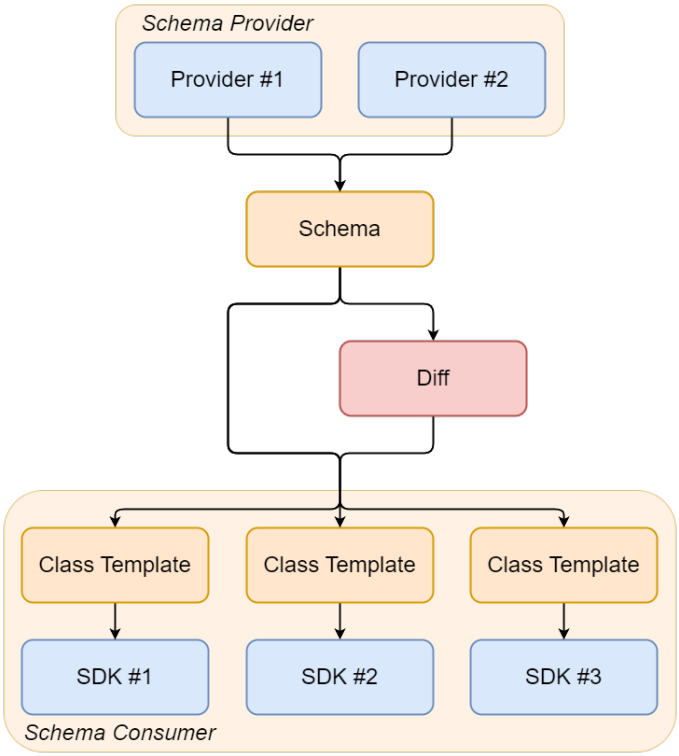

Figure 1: Flow of entire solution

key-value map, with the key being the unique identifier of the object. This key would often be $\{namespace\}.\{className\}$, however, this will not be a requirement. A key-value approach is chosen here, based on mainly two reasons; it is easier for humans to quickly see what type it is, and by taking inspiration from OpenAPI.

- **Phase 3: Additional properties**: When exploring hierarchy, it was identified the original definition was lacking properties necessary to implement this. To solve this, additional optional properties were added to the definition. If these properties are not needed, the schema can omit them.

- **Phase 4: Schema Type**: When exploring enums, it was identified that a single format for schemas is insufficient, as a class an enum is different, and thereby needs different properties. Therefore, a required property called *type* was introduced, specifying if it is a class or an enum schema, and a new enum schema was created.

- **Phase 5: Generics**: Like in phase 3, new properties were needed when adding generics,

however, this is not all that is required here. When defining a type, it is until now guaranteed to be a valid class. However, with the addition of generics, the type can now be either a generic T or a type with a generic type, e.g. *List<T>* or *List<string>*. For classes, an additional optional property was added. For class properties, the same syntax as both C#, Java, TypeScript, and other languages was used, meaning the value will be written the same as was just showcased. This means the Schema Consumer will be able to check if the type contains a <to indicate if it is a generic. Furthermore, it will be possible to detect if it is a class generic argument, or an implementation, by matching the type with the generic types that are defined in the class.

By undergoing all of these phases, the final design is ready. The definition itself can viewed in Table 4. For a better understanding of the definition, an example is further included in Listing 5. Furthermore, previous definitions are included in appendix B, to showcase the design journey.

Schema Structure		
Field	*Type*	*Description*
type	string	The type of the schema, either *object* or *enum*
namespace	string	Which namespace the class belongs to
name	string	The name of the class
Object Schema Structure : Schema Structure		
Field	*Type*	*Description*
abstract?	boolean	If the class is an abstract class
extends?	string	Which class the class extends
generics	string[]	A list of all the generic values the class contains
properties	Property[]	A list of all the properties the class contains
Enum Schema Structure : Schema Structure		
Field	*Type*	*Description*
values	string[]	A list of all the enum values
Property Structure		
Field	*Type*	*Description*
type	string	The type of the property, either a built-in type, or the key in the schema map.
name	string	The name of the property
nullable	boolean	If the property is marked as nullable

Table 4: Schema definiton

Listing 5: Schema definition

```yaml
schemas:
  "SchemaExtractor.Model.Example.Person":
    type: "object"
    namespace: "SchemaExtractor.Model.Example"
    name: "Person"
    generics:
    - "T"
    properties:
    - type: "string"
      name: "Name"
```

```
      nullable: false
    - type: "SchemaExtractor.Model.Example.Gender"
      name: "Gender"
      nullable: false
    - type: "T"
      name: "GenericProp"
      nullable: true
    - type: "List<SchemaExtractor.Model.Example.Person<T>>"
      name: "Parents"
      nullable: false
    - type: "Map<string, SchemaExtractor.Model.Example.Address>"
      name: "Address"
      nullable: false
  "SchemaExtractor.Model.Example.Gender":
    type: "enum"
    namespace: "SchemaExtractor.Model.Example"
    name: "Gender"
    values:
    - "Male"
    - "Female"
  "SchemaExtractor.Model.Example.Address":
    type: "object"
    namespace: "SchemaExtractor.Model.Example"
    name: "Address"
    properties:
    - type: "string"
      name: "Street"
      nullable: false
    - type: "int32"
      name: "Number"
      nullable: false
```

4.1.1 Adding support for native built-in types

By deep diving into Listing 5, it shows some types does not define a full path, such as string and int32. This is due to the schema having some "*built-in-types*". These are types that are not defined as schemas, but instead a native part, which a Schema Consumer should be able to consume. Table 5 lists all the built-in types. If a property uses a type that is not defined

here, it should be defined as its own schema.

Type	Description
boolean	A boolean input
int32	A 32-bit signed integer
int64	A 64-bit signed integer
float	A 32-bit floating point
double	A 64-bit floating point
char	Represent a single character
string	A collection of characters, as a single string.
object	A wildcard type. This type has not explicitly been stated, and everything should be accepted.
guid	A GUID / UUID
date	A date, that includes both date and time
dateOnly	A date, that only specifies the date and not the time
dateTimeOffset	A date, that includes both date, time, and time offset
List	A list of items.
Map	A key-value pair. Often referred to as either a map or dictionary in code.

Table 5: Built In Types

4.1.2 Adding support for generics

The adaptation of generics has a significant impact on the schema definition. The major decision is between should the schema be updated with new fields, or should the generics be added to the current schema, and have a custom parser. The later option is chosen, where a property type is defined the same was, as both *C#*, *Java* and *TypeScript*. To change e.g. a *string* type, to a generic string list, the type is updated to *List<string>*. When parsing a type, there should therefore be checked for <, and if found, the type should be parsed as a generic type. A pseudo implementation is suggested in Algorithm 1

4.2 Designing a schema provider

By having understood the schema, the Schema Provider can now be designed. The purpose of this is to take an existing solution and convert all DTOs defined here into schemas. This will enable the solution to act as a single source of truth for all DTOs. It is possible to create multiple schema providers if the code base is scattered across multiple languages.

Algorithm 1 Parse Type

```
1: function parseType(type)
2:     genericStart ← indexOf(type, "<")
3:     if genericStart == -1 then
4:         return type
5:     end if
6:     splitType ← split(type, genericStart)
7:     genericType ← splitType[0]
8:     genericImpl ← parseType(substring(splitType[1], 0, length(splitType[1]) - 1))
9:     return combine(genericType, genericImpl)
10: end function
```

The schema provider works by taking a single DTO, and from that output all necessary schemas. The reason why a single DTO can output multiple schemas is that both derived classes and classes used in properties should be defined. This is a recursive process, where the same steps are performed for all property types and derived types. An example of this is showcased in Figure 2, where a single DTO outputs five different schemas. These generated schemas would be:

- The own class itself
- The class it derives from, in this example called *ParentClass*
- All the classes used in the properties of the class, in this example *ClassOne*, *ClassTwo* and *ClassThree*

4.3 Designing a schema consumer

To create the final SDKs, the specified schemas should be consumed, an transformed into code. As showcased in Figure 1, a single schema can be turned into multiple templates, and thereby into a class in an SDK. The individual journey for creating a single SDK is illustrated in Figure 3, and will be looped for each target language.
Furthermore, the steps are explained in detail here.

- **Clone template**: The target language template is cloned to the output. What is included in this template, will be explored in Section 4.3.1.
- **Load template**: The class template for the target language is loaded and parsed, allowing for quick placeholder replacement.

Algorithm 2 Pseudo code for converting a DTO to a schema

```
allSchemas ← []
function convertClass(type)
    if type is in allSchemas then
        return
    end if
    allSchemas.add(generateSchema(type))
    if type.isDerived then
        convertClass(type.parentType)
    end if
    for each property in type.properties do
        convertClass(property.type)
    end for
end function
```

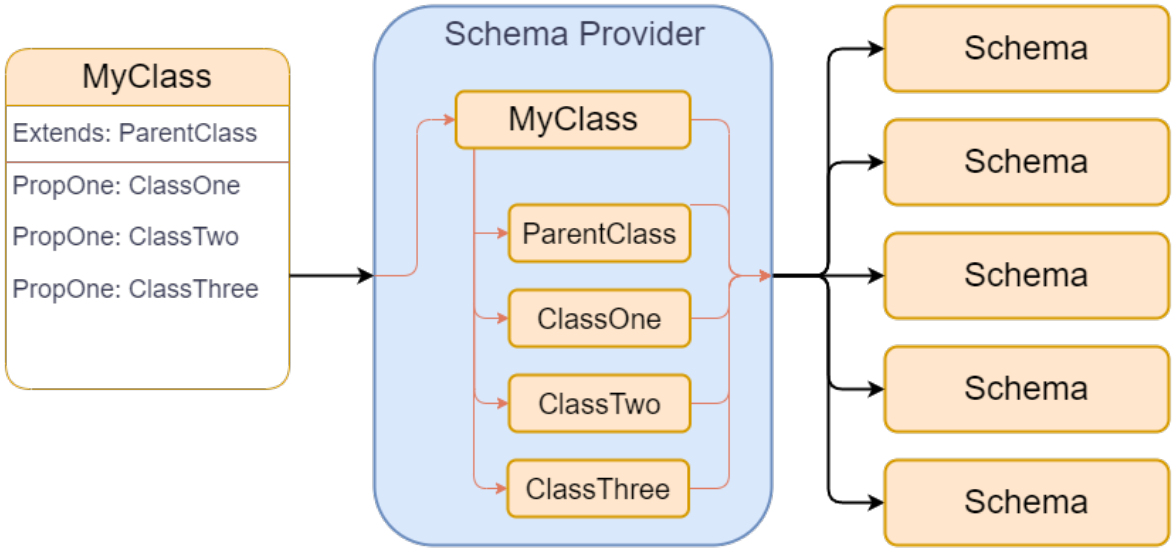

Figure 2: Schema Provider flow

- **Consume schemas**: The following steps will be executed for each schema that should be converted.
  - **Transform schema**: The schema is transformed to a language-specific data container, allowing for custom parsing of data, e.g. properties.
  - **Parse template**: The class template is parsed with the data, resulting in a complete class.

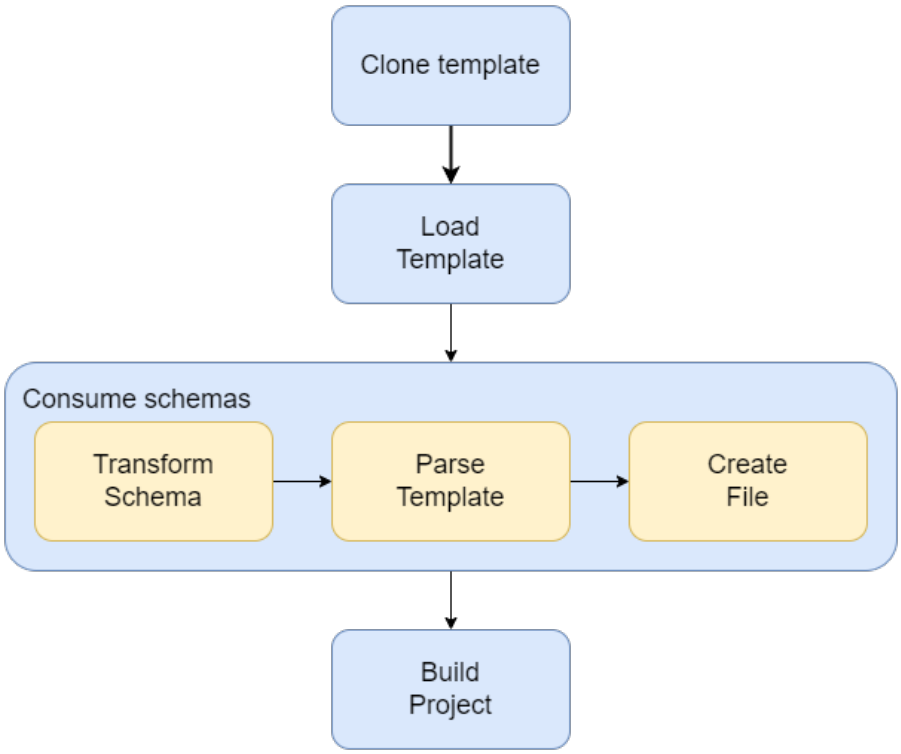

Figure 3: Schema Consumer flow

  - **Create file**: The class is written to a file, with a file type associated with the language.

- **Build project**: The project is built, using the language-specific build tools.

4.3.1 Adding default template projects

As specified, it should be possible to clone a default project, as each language has a file structure that needs to be followed. The alternative is to programmatically create files and populate these files with data, however, since the default files remain the same, they are simply cloned instead. This means that the developer can create a template project via the language's own tools, and build the SDK on top of this. If multiple versions of the same language should be added, the developer can then simply create two templates, one with each version. Furthermore, different package managers, default dependencies, and similar things can be included in this template. This may result in a large number of default projects, but that is by design, as it can cater to everyone's specific needs.

4.4 Designing a semantic difference checker

The job of the Diff Checker is to find all semantic differences between a list of schemas. This can be done by taking all the schemes and removing the shared schemas, as shown in equation 2.

$$\Delta = A \bigcup B - A \bigcap B \tag{2}$$

An example of this is shown in Figure 4, which shows two lists, and old list A, and a new list B. The Diff Checker will compare the figures semantically, meaning the order of properties

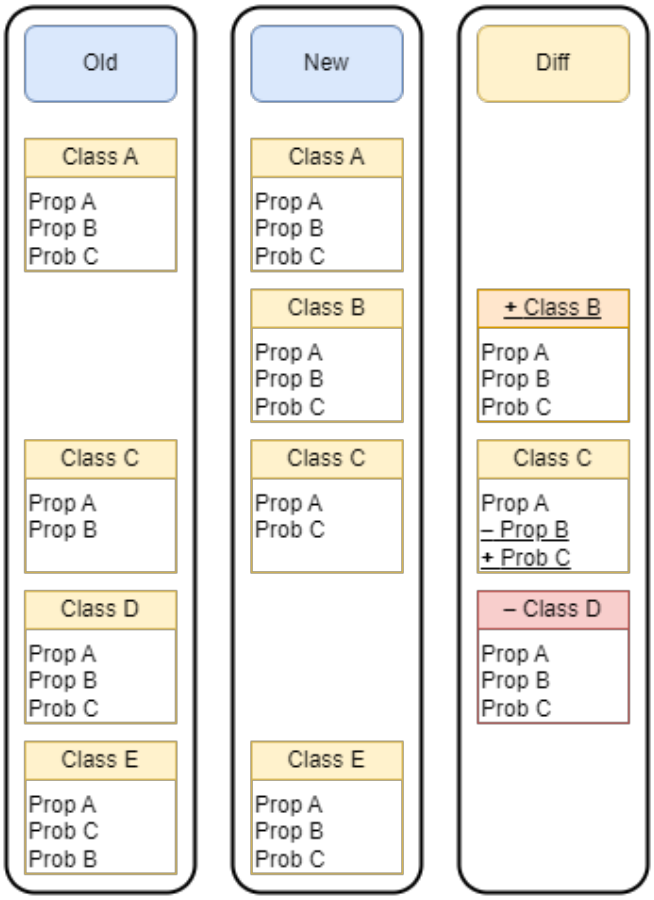

Figure 4: Diff checker result

does not matter, as long as all properties are included.

- **Class A**: Two identical classes, and therefore there is no difference.

- **Class B**: A new class that has been added
- **Class C**: A class where a property has been removed, and replaced with another
- **Class D**: A class that has been removed
- **Class E**: A class where the order of properties has been switched, but remains semantically identical, which results in there being no difference.

In this case, both *Class C* and *Class D* should give a warning, as these classes have changed. As *Class B* is a new addition, there is no need for a warning, and it can be added. From here, a developer should determine if the changes the Diff Checker warns about are acceptable or not. An acceptable change could be a planned change, where they are aware it is a breaking change. Are the changes accepted, the schemas can be transformed to an SDK, and the schemas the Diff Checker checks against can be updated with the new content.

5 Implementation

5.1 Schema Provider

For this prototype, only a single Schema Provider has been implemented, which is a C# Schema Provider. This will follow the principles defined in Section 4.2. One of the main features is the mapping of built-in types. As the definition does not specify a built-in type for all the types provided in C#, multiple types have been mapped to the same built-in type. An example of this is *byte*, *short* and *int* all being mapped to a *int32*. A full example of how types are mapped can be viewed in Listing 6, where the mapping to *int32* is highlighted.

Listing 6: C# type schema mapping

```
dtoDirectory
    .AddMapping<bool>("boolean")
    .AddMapping<char>("char")
    .AddMapping<string>("string")
     // ------------
    .AddMapping<byte>("int32")
    .AddMapping<sbyte>("int32")
    .AddMapping<short>("int32")
    .AddMapping<ushort>("int32")
    .AddMapping<decimal>("int32")
    .AddMapping<int>("int32")
    .AddMapping<uint>("int32")
    .AddMapping<nint>("int32")
    .AddMapping<nuint>("int32")
     // ------------
    .AddMapping<long>("int64")
    .AddMapping<ulong>("int64")
    .AddMapping<float>("float")
    .AddMapping<double>("double")
    .AddMapping<Guid>("guid")
    .AddMapping<DateOnly>("dateOnly")
    .AddMapping<DateTime>("date")
    .AddMapping<DateTimeOffset>("dateTimeOffset")
    .AddMapping(typeof(List<>), "List")
    .AddMapping(typeof(HashSet<>), "List")
    .AddMapping(typeof(Dictionary<object, object>), "Map");
```

Other than the mapping, the implementation does not have anything special, that is not discussed in section 4.2. The program converts on class to a schema at a time and adds it to an internal storage. Afterwards, all associated classes are converted the same way. All the mapping is done according to the specification of the schema, with properties either being a built-in type, as mapped in Listing 6, or another schema. Once all classes have been mapped, the list of all schemes can be extracted from the storage, in serialized.

5.2 Schema Consumer

The schema consumer is designed to allow adding multiple languages with minimum code. By limiting the unique amount of work required for each language to be implemented, the support is easily This is done by handling as much as the transformation behind the scenes, and only having a few interfaces that would need to be implemented to add support for a new language.

5.2.1 Built in types

To accommodate built-in types having different formats in different programming languages, it is possible to register custom converters. Listing 7 shows how all built-in types are registered.

Listing 7: Schema types to C# mapping

```
dtoDirectory
    .Register<object>("object")
    .Register<bool>("boolean")
    .Register<char>("char")
    .Register<string>("string")
    .Register<int>("int32")
    .Register<long>("int64")
    .Register<float>("float")
    .Register<double>("double")
    .Register<Guid>("guid")
    .Register<DateOnly>("dateOnly")
    .Register<DateTime>("date")
    .Register<DateTimeOffset>("dateTimeOffset")
    .Register(typeof(List<>), "List")
    .Register(typeof(Dictionary<object, object>), "Map");
```

All registered types require a custom Type Converter, which is shown in Listing 8.

Listing 8: ITypeConvertor interface

```
public interface ITypeConvertor<T> : ITypeConvertor
{
    Type ITypeConvertor.TypeToConvert() => typeof(T);
    public string ConvertProperty(SchemaProperty context);
}
```

The implementation of the type converter, tells the system how to format a property for the target language. An example of the need for this interface is when e.g. converting an *int32*. In Java, the property type would be *int*, while it would be *number* in TypeScript.

5.2.2 Class Converter

Another one of the main features of the solution is the implementations of the *IClassConverter* interface, which can be seen in Listing 9. This class serves as the main element to convert a schema to a complete SDK.

Listing 9: IClassConvertor interface

```
public interface IConvertor
{
    public string TargetLanguage { get; }
}

public interface IClassConvertor : IConvertor
{
    /// <summary>
    /// Defines the template from which a class template is created from.
    /// Placeholders, conditions and more are supported via <see
        href="https://github.com/scriban/scriban">Scriban</see>.
    /// </summary>
    /// <returns>A complete class template.</returns>
    public string GetClassTemplate();

    /// <summary>
    /// Formats a class property, to be ready to add into the class.
    /// The output will be available when parsing the class template.
    /// </summary>
    /// <param name="classProperty">The details for the property.</param>
```

```
20      /// <returns>A formatted property.</returns>
21      public string FormatProperty(ClassProperty classProperty);
22
23      /// <summary>
24      /// An optional way to further transform a class template, after the system has
            performed the initial transformation.
25      /// </summary>
26      /// <param name="classTemplate">The pre parsed template</param>
27      /// <returns>An updated template</returns>
28      public ClassTemplate PostTransform(ClassTemplate classTemplate)
29      {
30          return classTemplate;
31      }
32  }
```

To better understand how this interface is used, an in-depth explanation, with examples, is provided. The main functionality lies in the following three methods.

- **GetClassTemplate**: This provides a class template, that when populated will create an entire class for the specific language. This is the one of the core functions that drives the entire solution. The code in Listing 10 shows an example of a Java class.

Listing 10: Implementation of a Java class template

```
1       package {{namespace}};
2       {% for import in imports %}
3       import {{import}};{% endfor %}
4
5       public class {{class}} {
6           {% for property in properties %}
7           {{property}};{% endfor %}
8
9           {% for method in data.methods %}
10          {{method}};{% endfor %}
11
12      }
```

- **FormatProperty**: Instead of having the class template containing checks properties, a separate method is instead used to parse these properties. This should return a correctly

formatted property, which can be directly inserted into the class template. Listing 11 shows an example on how Java properties are formatted.

Listing 11: How a Java property is defined

```
$"{(classProperty.Nullable ? "@Nullable" : "@NotNull")}\n\t" +
$"private {classProperty.Type} {classProperty.Name};";
```

- **PostTransform**: The last method is an optional way to transform the Class Template further than is done behind the scenes, and create custom data for the specific language. This is e.g. used in Java to create getters, which is not needed in either TypeScript or C#.

5.3 Diff Checker

When implementing the Difference Checker, a Diff Schema is needed. The purpose of this schema is to return all changes to the user. The checker will iterate over all schemes, returning a list of all changes. It will differentiate objects by the key of the schema. This means, that if a key name has been modified, the system will treat it as if the old schema has been removed, and a new one has been added. A full example of how the diff schema is defined can be viewed in Listing 12.

Listing 12: Diff schema example

```
- type: "Removed"
  namespace: "Example.Model"
  name: "TestClass01"
- type: "Added"
  namespace: "Example.Model"
  name: "TestClass02"
- type: "Modified"
  namespace: "Example.Model"
  name: "TestClass03"
  differences:
    - path: "abstract"
      oldValue: "false"
      newValue: "true"
    - path: "generics"
      oldValue: "[TKey, TValue]"
```

```
      newValue: "[TKey, TVValue, TDefault]"
  properties:
    - type: "Removed"
      name: "MyProp1"
    - type: "Added"
      name: "MyProp2"
    - type: "Modified"
      name: "MyProp3"
      differences:
        - path: "type"
          oldValue: "string"
          newValue: "int32"
        - path: "nullable"
          oldValue: "true"
          newValue: "false"
```

6 Evalutaion

6.1 An overview of the DTOs used to evaluate the prototype

To properly evaluate the prototype, an advanced DTO structure has been created. It involves a complex and nested structure. The example DTOs consist of 486 properties, across 93 different classes. Based on the results found in the analysis, it can be estimated that a manual conversion would take approximately 2 hours and 42 minutes, as seen in Equation 3.

$$486 \text{ properties} \times 20,07 \text{ seconds/properties} = 2h\ 42m\ 34s \tag{3}$$

The example DTOs will be used in the data used in the entire evaluation and can be viewed in the source code, available from Appendix A

6.2 Performance Evaluation

From the analysis, manual DTO conversion was shown to take an average of 20.07 seconds per property, as documented in Table 2. This process is labor-intensive and prone to human errors, such as incorrect casing, syntax issues, and nullability errors. In contrast, the automated solution provided by the prototype performed these conversions almost instantaneously. Table 6 shows the time it takes for each service to perform its respective task. The time showed, is without any overhead, such as saving the schema as a YAML file or loading schemas from a YAML file.

Service	Time taken
Schema Provider	21ms
Schema Consumer	150ms
Diff Checker	10ms

Table 6: Prototype time evaluation

The total time for running the Schema Provider, validating the schemas with the Diff Checker, and creating a new SDK is *181 ms*, which is insignificant, when compared to the time a manual conversion would take, as shown in Equation 3. The task that takes the longest time, is consuming the schemas, and creating an SDK. This is mainly due to the I/O operations, where a file is created and written to, for each class generated. In scenarios where the Diff Checker checks all schemas, and validates there are no changes, this step could be omitted.

The other major element of the prototype is to reduce the number of errors in the final SDK. The number of errors is tightly coupled with the implementation of the interfaces. If there are any errors in the supplied interfaces, the error will persist in every class. However, with a correct implementation, all errors are completely eradicated. For the prototype, the one error that remains is the system uses an invalid casing, This id due to the casing is extracted from the C# code, and directly used in the templates. An implementation of a casing converter, as described by the requirements, would solve this issue.

6.3 Strengths and Limitations

The are both strengths and limitations to the developed prototype.

Strengths

- **Efficiency**: As proven by table 6, the prototype improves the efficiency of a developer, allowing them to create multiple SDKs at once, and freeing up time to focus on more complex and higher-value tasks, than a time-consuming conversion.
- **Accuracy**: The prototype eliminates all possibility of manual errors. Furthermore, it enforces the same conversions in the entire SDK, as the same rules are followed for every file.
- **Scalability**: The prototype can easily handle complex schemas and large-scale projects, making the prototype suitable for large-scale enterprise solutions.

Limitations

- **Initial Setup**: While a lot of time and errors can be saved, it may be a time-consuming task to set up the prototype for a project. This may prevent small projects from utilizing the prototype, as the time it takes to implement, may be greater than the time a manual conversion will take.
- **Customization**: The prototype has no configuration at all, and is therefor not suitable for a real-life scenario.
- **Evaluation Scope**: The prototype has not been validated and tested by stakeholders, or in any real-life scenario, but is instead developed from a theoretical aspect, and tested with theoretical cases.

6.4 Fulfillment of requirements

The prototype has not implemented all requirements, as defined in the requirements in Table 3. Instead, only the bare-bone required requirements has been implemented. The status of all implemented requirements can be viewed in Table 7.

#	*Fulfilled*	*Description*
1	Yes	Requirement 1, and all sub-requirements have been fulfilled.
2	Yes	Requirement 2, and all sub-requirements have been fulfilled.
3	Partly	Requirement 3.1 has not been fulfilled. All other sub-requirements have been fulfilled.
4	Yes	Requirement 4, and all sub-requirements have been fulfilled.
5	No	Requirement 5 has not been implemented.
6	No	Requirement 6 has not been implemented.
7	Yes	Requirement 7, and all sub-requirements have been fulfilled.

Table 7: Fufillment of the requirements

7 Future Work

As described earlier, the implemented solution is a primitive prototype, that acts as a proof of concept. This prototype would not be sufficient in a production environment, due to limited implementation and lack of features. This section will try to go through the steps needed, for the proposed solution to be effective in a real-life scenario.

7.1 Validation with costumers

As the project is not fully complete, it has not been verified with customers on their solutions. That said, the prototype is fully functional and ready to be presented to the stakeholders, but the final product is not ready for production, as several key elements that take it from a prototype to an integrated ready product are missing. Several of these missing parts will be discussed in this section.

7.2 Implementation of additional features

Adding configuration

One of the major features missing is the ability to configure the solution. It is impossible to create a single solution, that would please all developers out of the box. It is therefore important to be able to configure the solution. An analysis of which setting should be in a production-ready environment has not been made. However, a few settings are suggested. Some settings are relevant, no matter what the target language is.

- **Property Naming Attribute**: Should the system add an explicit attribute to properties, stating the name of the property in a JSON scheme?
- **Property Naming Policy**: If the user does not provide a property name attribute, what naming policy should the system default to?
  - **Camel case**: Name the properties as *camelCase*.
  - **Snake case**: Name the properties as *snake_case*.
  - **Pascal case**: Name the properties as *PascalCase*.
  - **None**: Do not modify the property name in any way.
- **Output directory**: Where should the outputted SDK be created?

On top of the general options, some options are specific for each individual language.

- **Java: Root Package**: In Java, namespaces often has a root, often a domain the user owns. This setting indicates what, if anything, should be prepended to the namespace.
- **Property Naming Policy**: If the user does not provide a property name attribute, what naming policy should the system default to?
- **Output directory**: Where should the outputted SDK be created?

Add options to change used 3rd party libraries

It should be possible to customize which libraries the generated SDK includes. This is potentially already possible, as it is possible to create multiple default projects, that are cloned. A new version could be created, where other default libraries are installed, resulting in the final SDK having those included. For the default projects that are included with the solution, they must include as few as possible libraries, to reduce both the size of the generated SDK and the libraries it will provide when added to a project.

Schema naming convention

At the moment, there is not defined any standard for how a class or property should be named. What is input in the schema, will be what is created in each SDK. However, languages often have different naming conventions they follow. An example of this is Java using *camelCase*, C# using *PascalCase* and Python using *snake_case*. Therefore, there should be defined a naming convention, and each language map to follow the language-specific style.

Add support for custom serilizers

When creating the Schema Provider, it should be possible to create custom sterilizer for data. It could be that the internal code handles a type as numbers, but the API exposes them as strings, or having a class being on encapsulation if a property, where the API only provides the encapsulated property. This should be implemented similarly to the ITypeConverter defined in section 5.2.1, with a different name, possible *ITypeSerilizer*

7.3 Expanding with an auto-generated HTTP client

As one of the major use cases of DTOs is HTTP communication, a suggested addition to the solution is an auto-generated HTTP client. When analysing a code-space in the Schema

Provider, it could further scan the same codespace if any HTTP endpoints have been defined, and include this information in the schema. The inclusion of an HTTP client would further benefit the use of the SDK, as neither the developers or users would need to manually built this on top of the generated SDK. This also refers back to the example showcased in both Listing 2 and Listing 3, where the need for automated SDKs were introduced. To implement this HTTP client an update to the schemas is required, to allow for endpoints to be defined. An early design of this is shown in Listing 13.

Listing 13: Example of adding endpoints to the schema definition

```
endpoints:
  # Base URL
  "localhost:3000":
    - path: "api/users/{userId}"
      method: "GET"
      parameters:
        userId: "string"
      response: "SchemaExtractor.Model.Example.Person"
    - path: "api/users"
      method: "POST"
      body: SchemaExtractor.Model.Example.CreatePersonDto
      response: "SchemaExtractor.Model.Example.Person"
schemas: {...}
```

Furthermore, the Diff Checker should be included in this updated schema. By both validating schemas and endpoints, there is further protection against unintentional changes.

7.4 Continuous Integration

To maximize the benefits of the Semantic Versioned SDK Generation prototype, it is necessary to integrate it into a continuous integration (CI) pipeline. Having the prototype included within CI processes has several notable advantages:

- **Automated SDK Generation** Incorporating this prototype in a CI pipeline would automatically generate SDKs whenever there are changes. This means that they will always be up-to-date with the latest coding changes, which eliminates the possibility of mismatch between source code and SDKs.
- **Automated Testing and Validation** Integration of CI automates validation for pro-

duced SDKs. This makes sure that errors or inconsistencies are not introduced when changing DTOs or schema. Any breaking changes can be instantly found. Therefore, this continuous process ensures high-quality code

8 Conclusion

This paper set out to validate the hypothesis, that an automated SDK generation could save both time and reduce errors. To validate the hypothesis, the time it takes a developer, and the amount of bugs is created in the process are compared with the automated prototype.

- **Time Savings**: The prototype shows a clear indication that the potential time saving is quite significant. It goes from approximately 2 hours and 40 minutes to shy of 200 microseconds to create the same SDK.
- **Error Reduction**: The reduction of errors is not as proven, as the time savings, This is mainly due to the limitations of the SDK. As described in Section 6.2, the prototype is shipped with known casing bugs. However, with further development, the rate of bugs can be greatly reduced. Furthermore, once a bug is addressed in the prototype, it will be permanently fixed, and not occur again.

As the sub-hypothesis has been validated, the main hypothesis is also validated.

References

[1] OpenAPI Initiative. *OpenAPI*. URL: https://www.openapis.org/ *(visited on: 31/3/2024)*.

[2] SmartBear Software. *Swagger*. URL: https://swagger.io/ *(visited on: 31/3/2024)*.

[3] Christoph Nienaber and Rico Suter. *ASP.NET Core web API documentation with Swagger / OpenAPI*. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-8.0 *(visited on: 25/4/2024)*.

[4] baeldung. *Documenting a Spring REST API Using OpenAPI 3.0*. URL: https://www.baeldung.com/spring-rest-openapi-documentation%22 *(visited on: 25/4/2024)*.

[5] François Wouts. *Auto-generating OpenAPI documents with TypeScript interfaces*. URL: https://www.highlight.io/blog/auto-generating-open-api-documents-with-type-script-interfaces *(visited on: 25/4/2024)*.

[6] Microsoft. *Kiota*. URL: https://github.com/microsoft/kiota *(visited on: 31/3/2024)*.

[7] Steve Gordon. *Writing Code with Code: Getting Started with the Roslyn APIs*. 2021. URL: https://www.youtube.com/watch?v=2AtNjxnwxZk *(visited on: 4/4/2024)*.

[8] Woochang Shin. "A Study on the Method of Removing Code Duplication Using Code Template". In: (2019).

[9] OpenAPI Tools. *OpenAPI Tools*. URL: https://github.com/OpenAPITools *(visited on: 14/4/2024)*.

[10] Joshua Chen, "wbamberg", and Hamish Willee. *Camel case*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Camel_case *(visited on: 3/5/2024)*.

[11] Joshua Chen, "wbamberg", and Hamish Willee. *Snake case*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Snake_case *(visited on: 3/5/2024)*.

[12] Cameron McKenzie. *Pascal case*. URL: https://www.theserverside.com/definition/Pascal-case *(visited on: 3/5/2024)*.

[13] YAML Language Development Team. *YAML Specifications*. URL: https://yaml.org/spec/1.2.2/ *(visited on: 14/5/2024)*.

Appendices

A Soruce Code

The source code can be found on the following link https://github.com/Arkobat/SdkGenerator

B Schema definitions

Schema Structure extends Schema Structure		
Field	*Type*	*Description*
type	string	The type of the schema, either *object* or *enum*
namespace	string	Which namespace the class belongs to
name	string	The name of the class
Object Schema Structure extends Schema Structure		
Field	*Type*	*Description*
abstract?	boolean	If the class is an abstract class
extends?	string	Which class the class extends
properties	Property[]	A list of all the properties the class contains
Enum Schema Structure extends Schema Structure		
Field	*Type*	*Description*
values	string[]	A list of all the enum values
Property Structure		
Field	*Type*	*Description*
type	string	The full name of the type, both namespace and class combined
name	string	The name of the property
nullable	boolean	If the property is marked as nullable

Table 8: Thrid iteration of the schema

Listing 14: Thrid iteration of the schema

```
schemas:
  - type: "object"
    namespace: "SdkGenerator.Model.Example"
    name: "LivingEntity"
    abstract: true
    properties:
      - type: "int32"
        name: "age"
        nullable: false
```

```
  - type: "object"
    namespace: "SdkGenerator.Model.Example"
    name: "Person"
    extends: "SdkGenerator.Model.Example.LivingEntity"
    properties:
      - type: "string"
        name: "firstname"
        nullable: false
  - type: "enum"
    namespace: "SdkGenerator.Model.Example"
    name: "MyEnum"
    values:
      - "ValueOne"
      - "ValueTwo"
      - "ValueThree"
```

Schema Structure		
Field	*Type*	*Description*
namespace	string	Which namespace the class belongs to
name	string	The name of the class
abstract?	boolean	If the class is an abstract class
extends?	string	Which class the class extends
properties	Property[]	A list of all the properties the class contains
Property Structure		
Field	*Type*	*Description*
type	string	The full name of the type, both namespace and class combined
name	string	The name of the property
nullable	boolean	If the property is marked as nullable

Table 9: Second iteration of the schema

Listing 15: Second iteration of the schema

```
schemas:
  - namespace: "SdkGenerator.Model.Example"
    name: "LivingEntity"
    abstract: true
```

```
    properties:
      - type: "int32"
        name: "age"
        nullable: false
  - namespace: "SdkGenerator.Model.Example"
    name: "Person"
    extends: "SdkGenerator.Model.Example.LivingEntity"
    properties:
      - type: "string"
        name: "firstname"
        nullable: false
```

Schema Structure		
Field	*Type*	*Description*
namespace	string	Which namespace the class belongs to
name	string	The name of the class
properties	Property[]	A list of all the properties the class contains
Property Structure		
Field	*Type*	*Description*
type	string	The full name of the type, both namespace and class combined
name	string	The name of the property

Table 10: First iteration of the schema

Listing 16: First iteration of the schema

```
schemas:
  - namespace: "SdkGenerator.Model.Example"
    name: "LivingEntity"
    properties:
      - type: "int32"
        name: "age"
  - namespace: "SdkGenerator.Model.Example"
    name: "Person"
    properties:
      - type: "string"
        name: "firstname"
```

C DTO Conversion Guide

C.1 Convert guide for Java

Listing 17: DTO conversion guide for Java

````
# How to create DTOs with Java

## Prerequisites
You must have Java and Maven installed.
To validate this, run ‘mvn -v‘ and ‘java -version‘

## Step by step guide
1) Check out the project ‘git clone git@github.com:Arkobat/SdkGenerator.git‘
2) Navigate to this folder ‘cd .\SdkGenerator\Templates\Java\‘
3) Create a new branch for your changes ‘git swtich -c <branch_name>‘. Replace
    ‘<branch_name>‘ with something unique, e.g. your GitHub username.
4) Install the dependencies ‘mvn install‘
5) Open the project in your desired IDE, and create the files needed
   - You are free to create both folders and files as pleases you
   - This is an example DTO, you can take inspiration from.
```typescript
package sdk;

import com.google.gson.annotations.SerializedName;
import org.jetbrains.annotations.NotNull;
import org.jetbrains.annotations.Nullable;

public class ExampleDto {

   @NotNull
   @SerializedName("name")
   private String name;

   @SerializedName("version")
   private int version;

   @Nullable
   @SerializedName("childNode")
````
````

```
33    private String childNode;
34
35    @Nullable
36    @SerializedName("parentNode")
37    private String parentNode;
38
39
40    public @NotNull String getName() {
41       return name;
42    }
43
44    public int getVersion() {
45       return version;
46    }
47
48    public @Nullable String getChildNode() {
49       return childNode;
50    }
51
52    public @Nullable String getParentNode() {
53       return parentNode;
54    }
55
56 }
57 ```
58 6) Build the project with `mvn package`
59 7) Commit and push the files `git commit -am "Creates DTOs" && git push`
60 8) Now you are done
```

C.2 Convert guide for TypeScript

Listing 18: DTO conversion guide for TypeScript

```
1  # How to create DTOs with TypeScript
2
3  ## Prerequisites
4  You must have NPM installed.
5  To validate this, run `npm -v`
```

````
6
7  ## Step by step guide
8  1) Check out the project ‘git clone git@github.com:Arkobat/SdkGenerator.git‘
9  2) Navigate to this folder ‘cd .\SdkGenerator\Templates\TypeScript\‘
10 3) Create a new branch for your changes ‘git swtich -c <branch_name>‘. Replace
       ‘<branch_name>‘ with something unique, e.g. your GitHub username.
11 4) Install the dependencies ‘npm install‘
12 5) Open the project in your desired IDE, and create the files needed
13    - You are free to create both folders and files as pleases you
14    - This is an example DTO, you can take inspiration from.
15 ‘‘‘typescript
16 export interface ExampleDto {
17     name: string;
18     version: number;
19     childNode?: ExampleDto;
20     parentNode: ExampleDto | undefined;
21 }
22 ‘‘‘
23 6) Once every file has been generated, run ‘npm run generate-barrels‘ to generate
       SDK.
24 7) Commit and push the files ‘git commit -am "Creates DTOs" && git push‘
25 8) Now you are done
````