Determining the viability for consumers of autogenerated Software Development Kits for Web APIs

Jean Alexandre de Leeuw

Technische Universiteit Delft

Delft University of Technology

Challenge the future

Determining the viability for consumers of autogenerated Software Development Kits for Web APIs

by

Jean Alexandre de Leeuw

in partial fulfillment of the requirements for the degree of

Master of Science

in Computer Science

at the Delft University of Technology,

to be defended publicly on Wednesday October 16, 2019 at 16:00.

Chair & academic supervisor:	Prof. Dr. Andy E. Zaidman	TU Delft
External supervisor:	Msc. Philip Leonard	Picnic Technologies
Committee member:	Dr. Lydia Y. Chen	TU Delft

TU Delft Delft University of Technology

Determining the viability for consumers of autogenerated Software Development Kits for Web APIs

by

Jean Alexandre de Leeuw
4251849
J.A.deLeeuw@student.tudelft.nl

Abstract

In this age of web APIs serving as the backbone of millions of services on the Internet, the developers aiming to make use of these existing services have to adapt to the developers providing these services. Whenever the services change, the users of the service have to change accordingly in order to keep using them. As the amount of third-party services used by an application grows, the more this process of adapting whenever something changes becomes more and more infeasible. One of the solutions to this problem is the usage of autogenerated Software Development Kits (SDKs). In this thesis we explore and determine the viability of these SDKs from the perspective of the consumer. We accomplish this by conducting an experiment that lets the participants solve tasks using a web API both with and without an SDKs. Their opinions were collected and manually analysed. Several of the participants were also invited for an in-depth interview regarding their opinions on the SDKs. We concluded that autogenerated SDKs are suitable for internal use in situations with low customization of settings. We also concluded that autogenerated SDKs are not suitable for third-party use.

Thesis Committee

Chair & academic supervisor:	Prof. Dr. Andy E. Zaidman	TU Delft
External supervisor:	Msc. Philip Leonard	Picnic Technologies
Committee member:	Dr. Lydia Y. Chen	TU Delft

TU Delft Delft University of Technology

Preface

I have started my journey at Delft studying Mechanical Engineering, only to discover after a year that my true passion lies in Computer Science. In contrast to the 3mE building, I always felt right at home in the EWI building, present on the cover of this thesis. Surrounded by friends, we have created many memories during our student life at EWI. Memories that I cherish greatly, and I hope you do too. With this thesis defense as the conclusion to my student life, it will be time to turn the page and start a new adventure. But before heading out towards the next challenge I would like to thank all my friends at Delft, for making these years unforgettable. It has been my pleasure to experience these years with you through both the highs and the lows, and without you I would likely not have been able to complete my studies the way I did.

Secondly, I would also like to thank my family, and in particular my parents for unconditionally supporting me both before, during and after my studies. You have taught (and proven!) to me me that a proactive stance and dedication can move mountains and you will always be an example for me.

In addition, I would also like to thank both my academic and my external supervisor for assisting me during this thesis. Andy for making sure I made the most out of this thesis and getting me back on track whenever I got lost, and Phil for providing an outside perspective and suitable advice. Furthermore I would also like to thank Dr. Lydia Chen for reading my thesis and being part of my thesis committee. My gratitude also extends to all the wonderful people at Picnic Technologies, who have provided me with all the resources I could possibly need to bring this thesis to a successful end. Moreover, I would also like to thank all the participants of my experiment, without which my thesis would hold no weight.

Finally, there is one person that deserves special mention for the amount of inspiration, motivation and patience that they have had for me during not only this thesis, but my entire studies. Willemijn, I would like to thank you from the bottom of my heart for supporting me through everything. There have been some difficult times during these years and you have made them bearable. I would not be where I am today without you.

Dear reader, you are the last person on this list to whom I would like to extend my gratitude. I would like to thank you for taking the time to read my thesis and hope you enjoy reading the fruits of several months of my labour.

Jean Alexandre de Leeuw
Delft, October 2019

Contents

List of Figures

List of Tables

1 INTRODUCTION

Using software built by third-party developers has become more and more popular over time. This increase in popularity is mainly due to open source becoming a phenomenon by the end of the 1990s [1], and for good reason. It allows the developers to focus their time on the core of the product that they are trying to build rather than on generic parts of the code. These third-party software building blocks normally came in the form of libraries that could be imported in the codebase.

While this code sharing makes development easier for the users of these third-party libraries, it requires producers of these third-party libraries to give a copy of their code to the consumers. This code sharing comes with two major downsides. The first downside is that the code may be sensitive from an intellectual property or security standpoint and needs to be defended against adversaries willing to exploit it. The second downside is that the producers hand control of the software over to the consumers, who could possibly share it further or tamper with it. While these problems are solvable with techniques such as watermarking, tamper-proofing and obfuscation [2], it would be more practical if the code did not need to be shared in the first place.

1.1. COMMUNICATION INSTEAD OF SHARING

Not sharing code while still providing consumers with the functionality of the software of the producers means that communication over a network needs to take place between the consumer and the producer. Communication can take place in many different ways, and as such different standards and technologies started to emerge. The Common Object Request Broker Architecture (CORBA) was an architecture that allowed communication through defining common objects. Remote Method Invocation (RMI) did the same but through methods instead of objects. However, these standards and technologies required implementation level details from both parties as the objects or methods needed to be exactly the same for both parties. Instead of sharing implementation level details however, it is also possible to define a messaging protocol through which to communicate, removing the necessity of sharing these implementation level details.

Standards such as the Simple Object Access Protocol (SOAP) are exactly these messaging protocols that aim to decouple the consumer and the producer. Applications began to use these common standards such as SOAP and started to communicate over the Internet, allowing them to be used by lots of different consumers at the same time essentially rendering them more into services. These services are what we now call web services [3]. Web services can be composed to create new more complex services and functionality, leading to the Service-oriented computing (SOC) paradigm. According to Papazoglou et al. [3], the “key to realizing this vision is the service-oriented architecture. SOA is a logical way of designing a software system to provide services either to end-user applications or other services distributed in a network, via published and discoverable interfaces”.

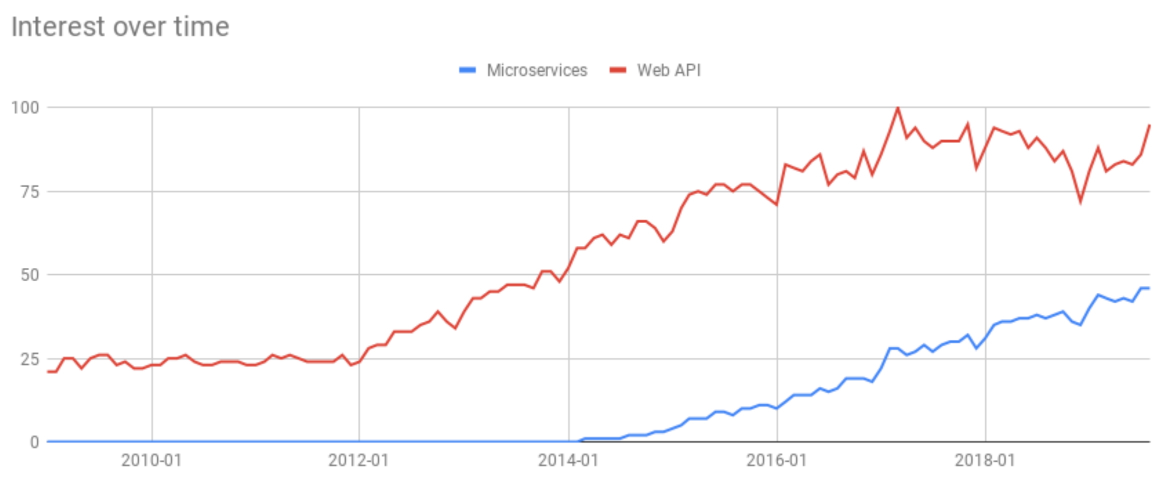


Figure 1.1: Worldwide interest in microservices and web APIs

1.2. THE RISE OF REST

While messaging protocols allowed both producers and consumers to stay in full control of their software, they still suffered from a major downside: brittleness. According to Muehlen et al. [4] SOAP suffered from tight coupling, as clients need to know the possible operations and their semantics beforehand. This is due to these messaging protocols which define the structure of the messages in great detail. While this removes all ambiguity over the messages, it also means that there is no room for flexibility from either side. Whenever either side wants to change the agreed upon structure of the messages, then both sides need to come together to discuss the potential change and then adjust their code base accordingly. This is especially difficult when a service is used by many (third-party) consumers. This led to an architectural style called Representational State Transfer (REST), which aims to reduce the brittleness and tight coupling compared to these protocols by "providing a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components ..." [5]. This simplification of the communication between producers and consumers is what really allowed these web services to become more and more common, starting a new trend the past decade: the rise of web APIs.

Before diving into the growth of this web API trend we will first define what a web API is. According to Wittern et al. [6] a web API is an "application programming interface invoked over network that relies on web technologies like HTTP as a transport protocol or XML and JSON as data formats. In practice, these APIs are often "REST-like", in that they adhere to some of the constraints imposed by the Representational State Transfer (REST) architecture style". To give an idea of who provides these web APIs, at the time of writing the top listed APIs on ProgrammableWeb[1] are for Google Maps, Twitter and YouTube, which were submitted in 2005, 2006 and 2006 respectively.

Figure 1.1 shows the interest in web APIs over the last decade, where the interest is based on the amount of Google searches worldwide including that term. The interest number is relative to the highest point in the given time frame. The data was retrieved from Google trends [2] for the time period between 01-01-2009 and 05-08-2019.

Unfortunately, RESTful web APIs are not the be-all end-all of producer-consumer communication. While exposing an API over the web through REST increases accessability and simplicity for the consumers, it however does not necessarily mean that the consumption is easy to do. Depending on the structure and complexity of the API the consumption of it can still be quite confusing. Furthermore, consumers need to be aware that communication with another service over the web is not instantaneous and requires consumers to consider asynchrony problems. The communication can fail at any time for many different reasons and their applications or services will have to be made resilient to those failures.

[1] https://www.programmableweb.com/category/all/apis
[2] https://trends.google.com/trends/explore?date=2009-01-01%202019-08-05&q=Microservices,Web%20API

1.3. SOFTWARE DEVELOPMENT KITS & MICROSERVICES

In order to ease the process of consuming an API some of the API producers started offering libraries that abstracted these API calls, giving the consumers the simplicity of a library while still allowing the producers to share their functionalities through a web API. During this dissertation, we will refer to these web API abstraction libraries as Software Development Kits (SDKs). While this SDK approach helps lift some of the burden of the consumers, not all problems are addressed. For example, asynchrony issues are still present, and might cause even more confusion for the consumer since the network call is hidden behind a layer of abstraction. Furthermore the API producers have to create and maintain the SDKs, which is extra effort purely for the benefit of the consumer. As a result, the amount of companies offering SDKs is more limited and consists mostly of the large companies with hugely popular APIs such as Google with Google Maps and Facebook with the Facebook API.

However, sometime around 2012 another trend started to emerge; Microservices. Sam Newman [7] roughly explains Microservices and their key benefits as follows: Microservices are small autonomous services that accomplish a task by communicating with each other, where each one specializes in one task of the functionality as a whole. Their autonomous nature allows them to be easily scalable by simply deploying more instances. It also allows them to maintain their own lifecycle and therefore makes them easy to deploy. Furthermore they make the system as a whole more resilient to failure, as only the problematic service will fail while the rest will continue working.

Microservices communicate with each other through web APIs, and therefore microservice communication suffers from the same problems as consumers of web APIs. The difference is that the developers creating a microservice-based architecture will have to deal with these problems for every single microservice they create and subsequently communicate with. This could significantly increase the burden on the developers, requiring them to familiarize themselves with every service they need to use. SDKs might alleviate some of the burden, but their major downside also increases: SDKs will have to be created and maintained for every microservice that can be consumed by others.

This is where autogeneration of the SDKs might prove to be useful. There are generators available such as Swagger Codegen[3] that transform a file describing the API into an SDK for a specified language, thereby diminishing the need for creating and maintaining an SDK for every microservice (for every supported language) into maintaining a generator (for every supported language). For companies striving towards a (polyglot) microservice-based architecture this could prove to be invaluable.

1.4. PICNIC TECHNOLOGIES

Picnic Technologies[4] is one of these companies that strives towards such a polyglot microservice-based architecture. For this reason, they are interested in the potential of these autogenerated SDKs and have agreed to aid in the research conducted in this thesis. For background purposes, we will shortly explain who they are and what they do. Picnic is a fast-growing online supermarket that delivers products straight to the customers' homes, thereby bypassing the need for a physical supermarket store. The delivery is free of charge, requiring Picnic to be extremely efficient in their process through constant innovation.

Now that we have introduced the background that led to this thesis, we would like to introduce two terms that are often used throughout this thesis before moving on any further.

1.5. TERMINOLOGY

We have already briefly mentioned some of the terms that will be commonly used in this thesis. The terminology for the concepts that we try to describe with these terms are not very well established, meaning that interpretation of the terms that we will use will vary from person to person. To avoid any such confusion throughout this thesis this section explains the concepts behind the terms that are used and briefly describes what the chosen definitions mean for the consumers.

[3] https://github.com/swagger-api/swagger-codegen
[4] https://picnic.app/nl/

1.5.1. NSDK

NSDK stands for No Software Development Kit, and it is the abbreviation that we use to signify the “traditional" way of communicating with a web API. This means that in order to communicate with a web API, the consumer has to manually make HTTP requests in their application to the web API using an HTTP client. Using Java Unirest [5] as an HTTP client such a request would look as follows:

```
SomeObject example = Unirest.get("http://example.com")
                .queryString("ParameterOne", 1)
                .queryString("ParameterTwo", 2)
                .asObject(SomeObject.class)
                .getBody();
```

This means that it is completely up to the consumer to figure out what the endpoints are, what parameters they accept, what responses they provide, what HTTP headers are required to communicate with the endpoints, and the list goes on. In other words, the amount of help provided by the producers to the consumers is limited to any resources published about the API, most commonly done in the form of online documentation.

1.5.2. SDK

As mentioned earlier, an SDK is a library that abstracts the calls to a web API. We consider two ways to create SDKs: manually written by developers and autogenerated by a generator. This thesis, mainly from Chapter 3 and onwards will use the term SDK to refer to autogenerated SDKs specifically. This means that some of the tasks that are necessary for the NSDK are no longer necessary for the SDK. For example, the available endpoints, their parameters and the return types are all described in the method signatures of the SDK. An example of an SDK call similar to the NSDK call in Section 1.5.1 would look like the following:

```
SomeObject example = exampleAPI.getSomeObject(1, 2)
            .block();
```

A comparison of the SDK and the NSDK call shows us that the NSDK call is much more verbose, requiring the consumer to specify all kinds of information as we already mentioned in Section 1.5.1. Furthermore, the SDK also diminishes the need for the consumer to resort to published documentation due to everything being described inside the method signatures of the SDK. This allows them to stay working inside their IDE, potentially increasing productivity.

1.6. COMPARING SDKS TO NSDKS

We have highlighted some of the potential of the (manually written) SDK compared to the NSDK at the start of this chapter, but in order to better demonstrate the impact that the SDKs could have compared to the NSDK we decided to list the advantages and disadvantages of the SDK compared to the NSDK, which can be seen in Table 1.1.

While the advantages can definitely make consumption of the API easier, the disadvantage is rather substantial. Especially when we consider the fact that the advantages are mostly for the consumers, while the disadvantage is purely for the producer. This diminishes the incentive for the producer to deliver such SDKs. This is where the autogeneration of SDKs comes in. Autogenerated SDKs have the potential to lower the burden of the creation and maintenance aspect of the SDK for the producer as well as being able to better distribute the advantages and disadvantages over the consumers and the producers, making it more likely for producers to provide such SDKs.

[5] `http://kong.github.io/unirest-java/Java`

Advantages	Disadvantages
✓ Includes data model objects and handles (de)serialization of these objects out of the box. ✓ Type safety out of the box due to these data model objects. ✓ Good IDE integration (code completion, discovery, etc.) because the SDK is a library. ✓ Can make intended web API usage more clear through the SDK API. ✓ Requests can already be checked on invalid parameters inside the SDK before sending the request to the server.	✗ SDK needs to be written and maintained for each supported web API in every language that needs to be supported.

Table 1.1: (Dis)advantages of (manually written) SDKs over NSDKs

Advantages	Disadvantages
✓ Single source of truth; SDKs written by different people could potentially have subtle but notable differences between them. ✓ API first design, forcing producers to properly think about their API before implementing it. ✓ Forces all business logic to reside in the backend. ✓ Server side also has the potential to be generated, which would leave only the business logic to be implemented by the producer. ✓ All data models will be generated and do not have to be made manually ✓ Parts of the configuration can be done automatically ✓ Only requires a generator to be created & maintained for each language, rather than an SDK for each supported project for each supported language.	✗ Generated code is generally considered to be less readable. ✗ Requires the producer to create a specification file describing the API. ✗ Does not allow producers to make intended usage of the web API clear through clear paths in the SDK API. They are simply an aggregation of endpoints.

Table 1.2: (Dis)advantages of autogenerated SDKs over manually written ones

1.7. THE AUTOGENERATION OF SDKS

Tools to generate SDKs are not new. For example, the Github repository of one of the well-known tools, Swagger Codegen[6], was created in 2011. The tool that we used during this dissertation is OpenAPI generator[7], chosen for its popularity and active development. The generator generates an SDK from a specification file describing all the endpoints and their parameters and settings. The specification file for the OpenAPI generator needs to be written in the OpenAPI specification format[8], and during this thesis we will assume that this specification file is written manually by the producer of the API. Although it is possible to generate this specification file for already existing web APIs using tools such as SpringFox[9] it remains to be seen how effective it is to generate a specification file based on an existing implementation and then use that generated specification file to generate an SDK. Furthermore it would also remove any potential advantages from performing API first design. To prevent any form of uncertainty regarding the input of our SDK generators and how that would effect them we decided to assume that the specification files are manually written.

We already mentioned that there are two ways to create an SDK, manually writing one or generating one from an SDK. The two different methods of SDK creation result in a few key differences, the comparison between the two can be found in Table 1.2.

In order to derive this list of (dis)advantages we manually created an OpenAPI specification file by taking a simple existing microservice provided by Picnic (see Section 1.4) as an example. We then generated a corresponding SDK using OpenAPI generator. This process demonstrated us a lot of the (dis)advantages and by

[6]https://github.com/swagger-api/swagger-codegen
[7]https://github.com/OpenAPITools/openapi-generator
[8]https://github.com/OAI/OpenAPI-Specification
[9]https://github.com/springfox/springfox

combining those with the arguments used during informal discussions on the subject with our colleagues we constructed this list of (dis)advantages.

As we mentioned earlier not all advantages and disadvantages are noticed by the same people, they are better spread out over the consumers and producers compared to the manually written SDK. To fully explore all the advantages and disadvantages for both groups would be very difficult in our time frame however, and therefore we decided to limit our research to the perspective of the consumer.

1.8. RESEARCH GOAL

Now that the background has been introduced and the potential advantages of autogenerated SDKs have been shown, we would like to explore whether or not the autogenerated SDKs provide viable alternatives for NSDKs, and if so, in what situations they would be warranted. We will do this by comparing the autogenerated SDK to the NSDK in the two scenarios where we believe the difference between the autogenerated SDK and the NSDK is the most notable for the consumer. The first scenario will focus on (de)serialization, which plays a large role in the initial implementation of the application for the consumer. The second scenario focusses on evolution; how well different types of breaking changes are handled, occurring as the underlying web API changes over time.

In Chapter 2 we will devise research questions that help us determine the viability of the autogenerated SDKs. In Chapter 3 we will discuss in-depth how we plan to answer the research questions through an experiment. In Chapters 4 to 7 we will go over and interpret the data collected during the experiment. Any threats to the validity of the experiment are discussed in Chapter 8 and the limitations of the experiment are discussed in Chapter 9. Finally we will conclude our research in Chapter 10.

2
RESEARCH

In Chapter 1 we compared the autogenerated SDKs to the NSDK and established that autogenerating SDKs can have many advantages compared to the NSDK in certain cases. It was also established that both the advantages and the disadvantages of the autogenerated SDK affect both the consumers as well as the producers of the API. Furthermore we decided to limit the scope of this experiment to the consumer due to time constraints.

In order to gain insights into the consumer's perspective of autogenerated SDKs we need to let consumers work with autogenerated SDKs in the areas where they differ from the NSDK to let them experience the difference between the two methods. If the autogenerated SDK is considered to be at least as good as the NSDK by the consumers then it could prove to be valuable to supply autogenerated SDKs to the consumers in those cases where the advantages outweigh the disadvantages. If the autogenerated SDK is not considered to be at least as good as the NSDK then the consumers are unlikely to use this method, regardless of any potential advantages for the producer.

In this chapter we will formulate the research questions that provide insight into the viability of using autogenerated SDKs from the perspective of the consumer. These research questions can then be used in Chapter 3 to design an experiment that will provide data on whether or not autogenerated SDKs are considered to be at least as good as the NSDK by the consumers.

2.1. METRICS

In order to establish whether or not the SDK is at least as good as the NSDK we first need to define what we consider to be "good". In other words, we need a metric which can measure the "goodness" of the two methods.

Even though the details of the experiment itself are not defined yet, we do know that there will be an experiment in which participants will get to experience both the autogenerated SDK and the NSDK in order to be able to perform comparisons between the two methods. As a result, we also know that the participants will have to communicate with an API.

2.1.1. OBJECTIVE VS SUBJECTIVE DATA

We would like to primarily use objective data, but if we take a look at what kinds of objective data we could collect we end up with the following:

1. Time taken per assignment per participant
2. Number of API calls per assignment per participant
3. The code created by the participants

This data however is unlikely to be directly related to how "good" the methods are. For example, while you could argue that the amount of time taken per assignment can give an indication for the ease of use for one of the methods, there are many other factors influencing that same property. It could be that the documentation for the SDK was better than the NSDK, causing the participants using the SDK to finish the assignments faster than the other participants. We decided that the best approach is to let the participants be the judge and ask them what parts of the SDK or NSDK they liked or disliked and for what reason.

Our primary data will therefore be subjective in nature, but we will use the listed objective data as a secondary source of data with the exception of the code created by the participants. We decided not to collect the code created by the participants for following two reasons:

1. It would introduce a lot of bias by the person grading the code, as what constitutes good code can differ from person to person.
2. We wanted the participant to focus on working with both methods rather than on unnecessarily optimizing their code because it would be graded in some way.

Details about the implementation of the measurements can be found in Section 3.1.

2.2. Fundamental differences between the SDK and the NSDK

The SDK and the NSDK have many advantages and disadvantages in common. Yet there are also several fundamental differences between the SDK and the NSDK which are the cause of the differences in (dis)advantages between the two.

We came up with the following areas in which the functionalities of the SDK could provide extra help to the consumer compared to the NSDK:

- (De)serialization
- Evolution
- Security

We will discuss the cause and effect of the difference between the SDK and the NSDK for each of those areas in Sections 2.2.1, 2.2.2 and 2.2.3 respectively.

2.2.1. (De)serialization

We came up with four advantages that the SDK has compared to the NSDK with regards to (de)serialization that could have an impact on the experience of the consumers while consuming the API. But before heading into these advantages, we will briefly explain the concept of serialization and deserialization.

According to baeldung [8] “Serialization is the conversion of the state of an object into a byte stream; deserialization does the opposite. Stated differently, serialization is the conversion of a Java object into a static stream (sequence) of bytes which can then be saved to a database or transferred over a network.” While this definition is specifically for Java objects, (de)serialization of objects can be done for any type of data structure. The most popular data exchange format is the JavaScript Object Notation (JSON)[1] due to its simplicity and it being easily readable by both humans and machines [9]. Now that the concept of (de)serialization has been explained we will continue with the four advantages of the SDK compared to the NSDK.

The first advantage is that the SDK provides the consumers with object representations of all data communications (both requests and responses) with the API and automatically (de)serializes this data into these object representations. The NSDK requires the consumers to create these object representations manually, which can take a lot of time depending on the complexity of the objects. Automatic (de)serialization is often supported in HTTP clients and should therefore require little effort for the NSDK. The only difference between the SDK and the NSDK in that aspect would be the creation of the object representations for the data.

The second advantage is that this manual creation of the object representation also requires a thorough understanding of the responses given by the API. This means that the documentation of the API needs to be

[1] https://www.json.org/

Type of change	**SDK**	**NSDK**
Upstream location changes	A	M
Parameter addition or removal	C	M
Parameter type change	C	M
Return type change	C	M
HTTP verb change	A	M
HTTP header change	A/C	M
Endpoint replacement or removal	C	M

[1] A = automatically resolved
[2] C = compiler warnings/assistance
[3] M = manually resolved/no assistance

Table 2.1: Breaking change supported offered by the SDK and NSDK

consulted and studied in order to start working with the API. Due to the SDK already having these object representations allows the consumer to explore and learn the API from within the IDE through the signatures and documentation in the SDK itself.

The third advantage with respect to (de)serialization is that the SDK also allows the consumer to toy around with the API and try out out certain operations, receiving immediate feedback from the compiler about what is possible and what is not. This toying around is more difficult using the NSDK, as feedback is only given at runtime instead of at compile time.

The fourth and final advantage is that since the SDK can be generated at build time the object representations of the API data do not have to be included in the code base and therefore do not require active maintenance. Changes in the object representations used by the API are automatically updated when using the latest version of the SDK.

2.2.2. EVOLUTION

According to Espinha et al. [10], the evolution of a web API is inevitable, and at some point in time drastic changes will have to be made that are not compatible with the original way of using the API. Introducing these breaking changes means that consumers will be forced to change their code base to accommodate for these breaking changes or risk breaking (parts of) their application.

There are many different types of API changes that introduce breaking effects for the consumers. Some of these changes that would be breaking for the NSDK can be automatically solved by the SDK by simply updating to the new version. An example of such a change would be changing the remote location of an endpoint. The SDK contains the knowledge of where all the endpoint are located, and therefore updating the SDK would suffice to make the code work again without any changes to the implementation of the consumer. In the NSDK on the other hand the knowledge of the endpoint locations is defined inside the implementation of the consumer, which feeds this knowledge to the HTTP client. Therefore a change in the location of the endpoint would require the consumer to change the implementation.

Unfortunately not all types of breaking changes can be automatically solved. For example, if a data type used by the API changes, then the implementation of the consumer that uses this data type has to be modified to work with the new data type regardless of whether the SDK or the NSDK is used. The SDK does have the advantage however that it can warn about breaking changes through the compiler once the version has been upgraded. The NSDK offers no such compiler support. Table 2.1 shows all the breaking changes that we have identified (see also Section 3.8.2) and the support offered by the SDK and the NSDK to resolve the corresponding breaking change.

2.2.3. SECURITY

There are a lot of different standards and best practices for securing REST APIs from unauthorized consumers. Maleshkova et al. [11] analysed 222 web APIs registered on ProgrammableWeb [2] and found eight different standards of web API authentication not including the category “other”. Due to the vast amount of possibilities in this area and the limited time we have available we decided to exclude security from the scope of the thesis. Nonetheless, we would like briefly discuss the differences between the SDK and the NSDK on a security level.

Depending on your security requirements the SDK can automate or simplify some of the security requirements imposed by the API. If the security settings are standardized, then the SDK can save some significant effort by making the process as easy as possible. SDKs offered to third-parties can heavily benefit from this, as all consumers are likely have the same permission levels and therefore all use the same security settings.

For SDKs used internally in companies however there could be a lot of different permission settings depending on the authority of the consuming service. For example, it could be that certain low authority consumers are only allowed to read data from the API and not post new data to it, reserving this right only to consumers with a higher authority level. If the internal security permissions are standardized and therefore the amount of different security permissions and standards are limited, then the SDK can still provide some value. Otherwise the NSDK solution will likely take less effort because, the SDK will have to know about and properly support every single possible configuration.

2.3. RESEARCH QUESTIONS

The focus of this research is to establish in which settings (if any) the autogenerated SDK is considered to be at least as good as the NSDK by the consumers. If any such settings can be found then the autogeneration of SDKs can be considered as a serious alternative to the NSDK for the people who find themselves in that setting.

Now that we have explored the fundamental differences between the SDK and the NSDK and we have established the metrics which will be used to quantify the differences between the two methods we define will the questions that will be driving this research. We have divided the questions by the two areas established in Section 2.2: (de)serialization and evolution.

2.3.1. (DE)SERIALIZATION

Writing the (de)serialization code is often only done once: during the implementation phase. Whenever the API evolves there might be some small changes to the (de)serialization code, but most of it is only written during the implementation phase. As a result, there are two major factors that determine whether or not one of the methods (SDK or NSDK) has an advantage over the other: the amount of effort it takes to implement the solution and the quality of the code of the solution.

Code quality is an ambiguous term, as there is no widespread accepted definition. Rather, there are many different definitions that try to capture the concept and as many different models that try to make it measurable [12–14]. While we do not follow any such specific definition or measurement model we think it is safe to say that the amount of future maintenance necessary for a specific piece of code is related to the quality of said code. Therefore we think that the quality of the code resulting from the use of either the SDK or the NSDK be of interest. Futhermore, we also think that the amount of effort required to implement a solution using either the SDK or the NSDK is interesting, because the less effort spend on the implementation, the more effort can be spend on other tasks (e.g. writing documentation) or other projects. Therefore the two questions that we must ask ourselves regarding (de)serialization are:

Research question #1

Does one of the two methods take more effort compared to the other?

[2] `http://www.programmableweb.com`

Research question #2

Does one of the two methods result in higher quality/cleaner code compared to the other?

2.3.2. EVOLUTION

As we already mentioned in Section 2.2.2, breaking changes are inevitable. This means that the impact of any advantages provided by the SDK or NSDK could be very valuable, as these advantages present themselves every time the API evolves rather than just once during the implementation phase. This means that dealing with evolution is an import factor in establishing whether or not the SDK is at least as good as the NSDK.

First off, effort is also a factor in evolution. Even though breaking changes might be infrequent and often only affect certain parts of the code (rather than everything at once), they might still add up over time. Therefore any advantages gained in effort could still be considered significant over time. As we discussed in section 2.2.2,the SDK is able to automatically solve certain breaking changes and offers assistance in the other breaking changes, meaning that there could be a significant reduction in effort between the two methods. This leads us to the following question:

Research question #3

Does one of the two methods take more effort compared to the other?

Secondly, as we expect the quality of the code to be mostly determined by the initial implementation and less so by any breaking changes, we are interested in the testability of the two methods. According to Dogša et al. [15] having a thorough unit-test safety net, as introduced while performing Test Driven Development (TDD), improves maintainability. This means that proper testing can significantly reduce the amount of time spent on maintenance down the line, meaning that accessible testing provided by the methods could be very valuable. This leads us to the following question:

Research question #4

Does one of the two methods result in better testable code?

Lastly, the SDK is able to provide some extra resources (such as compiler warnings) and some extra assistance in certain breaking changes as we have mentioned in Section 2.2.2. To be able to determine the value of these extra resources and assistance we have to know what changes are considered to be difficult and what resources are considered to be useful. We ask ourselves the following two questions:

Research question #5

What types of breaking changes are considered to be difficult?

Research question #6

Which resources are considered to be useful during solving breaking changes?

2.3.3. SCIENTIFIC CONTRIBUTION

The answers to the research questions will provide us insight on the viability of the SDK compared to the NSDK. The goal and contribution of this thesis is to provide the readers with the knowledge that allows them to make informed decisions on whether their use case would benefit from autogenerated SDKs or if they should stick with NSDKs. Now that the research questions have been defined, we will gather the required knowledge through the experiment described in Chapter 3.

2.4. SUMMARY

In this chapter we have defined the goal of this thesis; to provide the readers with knowledge of when to use autogenerated SDKs. This allows them to make informed decisions about whether or not autogenerated SDKs are suitable for their specific use case. Furthermore, this chapter defined the research questions that will be used to design an experiment that will collect the data necessary to provide such knowledge.

3
Methodology

Now that the research questions have been defined in Chapter 2, we can start designing an experiment whose purpose is to gather data that is relevant to answer these questions. This chapter will describe and explain the processes, assignments and decisions that we made in order to gather the required data. The data acquisition process consists of two parts:

1. Acquiring information related to (de)serialization

2. Acquiring information related to evolution

Both measurable data as well as questionnaire data containing the opinions of the participants will be collected during these two parts. Even though the purpose of the questionnaires is to collect the opinions of the participants, we expect that participants are not likely to go in-depth in their answers when they have to perform multiple assignments and have to write everything down. As a result, we decided to also perform five 15-20 minutes interviews several days after the experiment with a select number of participants. The purpose of this interview is to be able to go more in-depth into the topic because the questions will be interactive rather than static, allowing us to dig deeper into the opinions of the participants wherever we deem it necessary.

3.1. Measurements

We start by defining the data we would like to measure for our research. The primary data we would like to have are the opinions of the participants on certain aspects of the SDK and the NSDK. These opinions are measured by letting the participants perform certain assignments and then asking them to share their preference for one method (SDK or NSDK) or the other. These preferences are measured using Likert scales with a range of five. The reasoning behind that preference is also asked and is shared with us through text. This questionnaire data is all saved for later analysis. All questionnaires can be found in Appendix A.

Secondly, as mentioned in Section 2.1.1, we decided to measure the amount of time each participant needs for each assignment. We suspect that the uses for this data is limited, but perhaps it could lead to some nice findings. Measuring these timings also has no impact on the participants and allows us to have some objective data.

Finally, we also measure every API call the participants make including the parameters. Because the API is deterministic, recording every API call allows us to replay all API interactions a participant has had. Advanced analytics could reveal some interesting patterns hidden in the data if the time constraints allow for it and otherwise it could be useful for debugging purposes in case something goes wrong.

3.2. Categorization of the participant responses

As we mentioned in Section 3.1 our primary data will consist of the opinions of the participants. Every participant has their own way of describing and explaining certain concepts or phenomenons. As a result, two participant who are trying to explain the same concept end up with two explanations that are close to each other in overall meaning, but might have slight differences due to word choices. Unfortunately this poses a problem for us, as we do not know if they mean the exact same thing, the differences being purely a consequence of word choice, or if they explicitly chose their words this way because they intended these slight differences to be the crucial part of their argument.

Because we are unable to differentiate between similar answers we often end up with the same amount of different answers as the number of participants who have answered that specific question, which in turn prevents us from performing any sort of comparative analysis on the responses. Therefore we need to reduce the amount of different answers and the way that we have performed this reduction is by categorizing the responses.

Grouping similar items into categories is a process where the result could potentially heavily depend on the people performing the categorization. This means that it is a process were potential biases can have a big impact on the final result. There is a technique called Card Sorting [16] which can identify potential categories by asking multiple subjects to group the items. The categories can either be predefined (closed card sorting) or can be designed by the participants themselves (open card sorting). This technique is often used in user experience design and information architecture.

We deem open card sorting to be unlikely to work, as there will be no natural categorization for the answers given by the participants, and therefore we will likely end up with the same number of different categories as there are people categorizing.

The problem with closed card sorting however is that we would have to provide categories to choose from upfront to the people categorizing, meaning that the categorization would still contain a bias introduced by us. Since open card sorting is unlikely to work we decided to use closed card sorting to categorize the responses. Furthermore as we cannot seem to avoid introducing this bias, in the interest of time we decided to perform the entire closed card sorting by ourselves.

3.3. Group division

In order to gain insight into participants opinions as to which method has their preference, they need to have experienced both methods. This means that we would use a within-subject experiment [17]. However, when a participants solves an assignment using one of the methods, and then solves the same assignment using the other method, the participant will already have gained experience in that assignment, and therefore solving the same assignment a second time should be easier compared to the first time, potentially skewing the results in favour of the method used during the second time. The effect of the participant learning from their previous experience (and therefore performing better the second time) is called a carry-over effect [17]. This problem could be solved by letting participants solve two different assignments, one for each method. This approach however causes unreliable results because any measured differences in the comparison of the methods could be caused by the difference in the assignments. Therefore the assignments have to stay the same.

In order to overcome this problem to the best of our abilities we will split the participants into two groups. One group that will solve all the assignments using the SDK first and then again but while using the NSDK. The other group will solve all assignments the other way around; using the NSDK the first time and using the SDK the second time. This is called counterbalancing, and according to Keren et al. [18] it is one of the ways to minimize these carry over effects.

This approach only works under the assumption that the carry-over effect for both groups is the same (i.e. the participants of one group do not learn faster than the participants of the other group). If the participants of one group are able to learn faster than the other due to the ordering of the iterations (SDK first or NSDK first) influencing the carry-over effect then the results will be skewed. Unfortunately we are not able to accommodate the experiment for this possibility due to time constraints, and therefore we will continue under the assumption that the learning factor experienced by the participants of both groups is the same.

Now we still have to divide the participants into two groups. In order to prevent skewing the result due to one group having more experienced developers than the other we must divide them based on their previous development experience. This requires knowledge of the previous experience of the developers and therefore we need another questionnaire that participants have to fill in before the start of the experiment. This questionnaire will contain questions about the previous development experience of the participants in general, but also in specific areas that might be relevant for the experiment (for example having a lot of experience with SDKs).

We asked the participants to share their amount of experience as a developer. Experience was defined in the questionnaire as:

> Anything where you consistently write and work with code for a significant amount of time during the week (e.g. university work, or a couple of hours every evening in your free time) counts as experience.

Due to time constraints, we will only take the general development experience of the participants into account while dividing them over the two groups. Information about the experience of participants in specific areas might become useful should we encounter any severe anomalies in the data. The previous experience questionnaire can be found in Appendix A.2.

We will look at the self-reported experience of each participant (question two of Appendix A.2), which will be given in ranges in order to preserve anonymity as much as possible. For experience ranges with an even number of participants we will place half of them in the first group and the other half in the second group. For experience ranges with an odd number of participants we will try to accommodate for any experience discrepancies in the group manually by placing participants with a higher amount of experience in the group with a higher number of lesser experienced participants. At all times we will try to keep the size of the two groups as equal as possible.

3.4. Participant experience expectations

In Section 1.4 we pointed out that we are currently performing this research in cooperation with Picnic Technologies. This means that most participants that will participate in this research are Picnic employees. This knowledge gives us some information regarding the previous experience of the participants, especially regarding tooling as they are likely to be familiar with the technologies used by Picnic.

Choosing the technologies used by Picnic would mean that we limit the amount of variance in the experience of participants. This variance reduction will make the results of the experience more reliable, as the difference in results are more likely to be caused by the variables we are testing for (SDK vs NSDK), rather than by variance in the amount of experience between the participants.

Not every participant is a Picnic employee however, as some of our non-Picnic colleagues will also be participating in the experiment. Since they are not guaranteed to have the same technology experiences as the Picnic employees there will be some differences in that regard. Since the amount of Picnic employees that will participate in the experiment will be significantly larger than the amount of non-Picnic employees, we have decided to choose the technologies to be used in the experiment with Picnic employees in mind. Even though this might put non-Picnic employees at a disadvantage in terms of technology experience, we feel that the benefit of minimizing the experience variance in the majority group outweighs this disadvantage.

3.5. Technology choices

Participants are not expected to build everything from scratch, as this would not be representative of actual development. It would also make the comparison between SDK and NSDK unfair and meaningless, because the extra work of building everything from scratch will be noticed in the generator and not in the SDKs that are generated from it. In this section we discuss which technologies the participants will come in contact with during the experiment.

Communication with a REST API means that HTTP calls have to be made, which requires us to select an HTTP client. Data will have to be exchanged by these HTTP calls in some serialization format, meaning that serialization and deserialization is also mandatory. Therefore a library capable of (de)serialization will have to be chosen. The technology choices we made for the HTTP client and the (de)serialization library will be discussed in Section 3.5.1 and 3.5.2 respectively.

3.5.1. HTTP CLIENT

Everyone has a personal preference in which HTTP client they prefer, mostly based on which HTTP clients they have previous experience with. Choosing an HTTP client that certain participants have experience with but others do not could cause skewed results because the inexperienced participants might experience the assignment to be more difficult than the experienced participants due to this lack of experience. Not to mention that there might be participants that have no experience with making HTTP calls at all, who will have to learn how to make HTTP calls while making the assignments.

The solution is to either choose an HTTP client that (almost) no one is familiar with, or one that (almost) everyone is familiar with. Unfortunately because we do not know the exact participants beforehand we can only make general predictions about the group of people we expect to participate as we discussed in Section 3.4.

Unfortunately it is impossible to completely avoid an advantage for the people with HTTP client experience compared to those without HTTP client experience. The best we can do in that regard is to use an HTTP client that is easy to use and understand in order to minimize the amount of time the inexperienced participants will spend on figuring out how it works. Furthermore we will include instructions on how to use the chosen HTTP client (see Section 3.6.2) and ask the participant for their previous experience using HTTP clients so that we can investigate if this is a relevant factor should we notice unusual data in the collected results.

SDK: WEBCLIENT

For the SDK we were limited in choice in terms of HTTP clients that were supported by our SDK generator. The list of supported HTTP clients is quite large, and finally we made the decision to use Spring 5 Webclient[1]. This choice was made because the majority of the participants will be Picnic developers and Webclient is becoming the standard in Picnic, meaning that most participants will be familiar with this technology.

While the participants should not notice the HTTP client in the SDK, as it will be abstracted behind methods, choosing Webclient did introduce a side effect that the participants do notice: all responses will be reactive. Spring Webclient uses Reactor[2], and causes all return types of the SDK to be reactive. Since reactive frameworks are part of the Picnic technology stack, most Picnic employees should already be familiar with this, but there might also be participants that are not familiar with reactive frameworks, introducing another potential cause for skewed results. The same precautions are taken here as we did to prevent skewed data based on HTTP client experience: instructions will be provided and the experience of participants with regards to reactive frameworks will be tracked.

NSDK: UNIREST

For the NSDK, the choice of HTTP client fell on Unirest [3]. The reason being that it is a relatively simple HTTP client so it is easy to learn when people are new to HTTP clients, as it makes certain aspects explicit instead of hiding it in configuration. The downside is that due to making certain aspects explicit the code can become a little verbose, but we felt like this tradeoff was worth making.

3.5.2. (DE)SERIALIZATION LIBRARY

Before deciding on the (de)serialization library we had to decide on the serialization format to use. We decided to go for the JSON, as it is the most commonly used data format for API communication [9]. It is also easily readable by both humans and machines and due to the popularity of the format there are quite some libraries available for it.

[1]https://www.baeldung.com/spring-5-webclient
[2]https://projectreactor.io/
[3]http://kong.github.io/unirest-java/

In terms of open source Java libraries that support JSON (de)serialization there are a number of choices such as Jackson[4] and Gson[5]. We went with Jackson, as it arguably the most common one and is used by both Webclient and Unirest, making it a natural choice.

Participant should only have limited interaction with Jackson, as both the SDK and Unirest should do most of the work for them. Nonetheless, there are several assumptions that Jackson makes about your data models which not all participants may be familiar with. The SDK provides data models and as a result the participants do not need to be aware of these assumptions. For the NSDK however the participants are expected to create the data models themselves, and as a result they do need to be aware of these assumptions. These assumptions have been clarified and explained in the Unirest documentation that we provided (see Section 3.6.2).

3.6. RESOURCES AVAILABLE TO THE PARTICIPANTS

In Sections 3.5.1 and 3.5.2 we already mentioned some of the measures we have taken in order to keep the differences in experience levels limited as much as possible. These measures consist of some documentation and examples on how to work with the technologies included in the experiment. Participants will need to be aware of more than just working with Reactor and Unirest however, and we have provided resources to them that should help them become aware of everything they need to now in a short time frame. In this section we will briefly go over all resources that we have provided to the participants.

3.6.1. REACTOR EXPLANATIONS

Reactive types have a lot of useful functionalities but can be quite difficult to grasp for developers who are inexperienced with them. Since educating the participants properly about reactive types can be quite time consuming we only provided a brief explanation about reactive types and some explanations about how to extract the actual value out of the reactive type, allowing them to proceed with the assignments with the value as a regular variable. The explanations contain examples for both reactive types in Reactor: `Mono` and `Flux` . We hope that these explanations allow participants that are inexperienced with Reactor to convert their reactive types to regular types with minimal time loss.

3.6.2. UNIREST EXPLANATIONS

There are four aspects of Unirest that need to be explained to the participants in order to mitigate the risk of them getting stuck in certain parts of the experiment:

1. Jackson requirements
2. Object and array deserialization
3. Object serialization (arrays are only serialized as part of an object, which Jackson handles automatically)
4. Polymorphic JSON types (required for one endpoint)

The first aspect is necessary because Unirest uses Jackson and Jackson has certain requirements for the data models that need to be (de)serialized. Failure to comply to the requirements causes the (de)serialization to fail. Therefore participants unfamiliar with Jackson need to be aware of these requirements.

The second and third aspects are examples on how object and array (de)serialization works in Unirest. Unirest is relatively straightforward, but certain headers need to be set which can take time to figure out. Giving the participants these examples prevents them from having to figure it out themselves.

The last aspect that needs explanation is how to use JSON polymorphism in the NSDK data models. One of the endpoints of the assignments returns different types depending on the input parameter, and therefore polymorphism is necessary. We provided the participants with a piece of code that could directly be used in their code base, as it can be tricky to get right.

[4]https://github.com/FasterXML/jackson
[5]https://github.com/google/gson

3.6.3. AUTHENTICATION

As explained in Section 3.1 we save the responses and some other measurements of the participants. In order to do this the participants need to authenticate themselves to the API using an identification key. This means that the SDK and Unirest need to be aware of this identification key. The authentication key is explicitly mentioned in all the Unirest examples and the code will fail to compile if simply copy pasted. For the SDK it is explicitly mentioned in the code which will also fail to compile without it. This failure to compile is intentional in order to alert the participant that they need to provide their identification key.

3.6.4. API ENDPOINT DOCUMENTATION

The last resource we provided was a swagger page containing documentation for all endpoints for each assignment. All data models could also be found on this page. Furthermore, this documentation contained simple example responses for each endpoint and is also interactive, allowing for manual querying of the endpoints through the swagger page to get an idea on how they work.

3.7. (DE)SERIALIZATION ASSIGNMENTS

With regards to (de)serialization we are interested in:

1. Whether participants think that implementing a (de)serialization task using one method (SDK or NSDK) takes more effort than the other
2. Whether participants think that the resulting code of the (de)serialization task is of higher quality for one method compared to the other

The easiest way to gather this information is to create a (de)serialization assignment and letting the participants do the assignment twice, once while using the SDK and once while using the NSDK (order depending on their group). After completing both iterations we ask them their opinion using a questionnaire. But the usefulness of the SDK or NSDK can also heavily depend on the complexity of the task at hand. For example, an SDK already handles a lot of the (de)serialization work by providing the data models and automatically serializing requests and deserializing responses into those data models. This advantage could be quite significant while (de)serializing heavily nested objects, but is likely to be minimal when working with primitives.

While fully representing the entire spectrum of (de)serialization difficulties is impossible given the time frame of the research, we do want to minimize the chance of skewed results due to only introducing participants to one side of the spectrum. As a result, we tried to represent the spectrum to the best of our abilities by separating three different cases:

- Primitives
- Heavily nested objects
- Domain translation

For each of these cases an assignment was made and was included in the experiment. The details of each assignment will be discussed in Section 3.7.2, 3.7.3 and 3.7.4 respectively.

3.7.1. DOMAIN

Before going into the details of each assignments we will briefly discuss the domain that we have chosen for the assignment to prevent any confusion during the explanations of the assignments themselves.

The domain for the (de)serialization assignment is that of a computer store. Participants will be asked to fetch computer parts based on certain requirements, construct computers from computer parts, process order requests and sales, etc. They will have to do this by requesting computer parts, order requests and other necessary components for the task at hand from the API. Then they have to perform some operations on the data they received, and submit the resulting object to a specific endpoint.

So in order to both cover serialization as well as deserialization the (de)serailization assignments all follow the same format:

1. The participant has to request some data for the assignment
2. The participant has to process this data and use it to create a certain object. This object will require additional data that the participant will have to fetch (and possibly filter) from various endpoints.
3. After the processing is done the created object has to be submitted to a specific endpoint which verifies if the created object is correct.

3.7.2. PRIMITIVES

The most simple types to (de)serialize are primitives. This is also why they were chosen to be the first assignment in the list, so that people could slowly get introduced to both the concept of the assignments as well as any technologies that are used in the project (see Section 3.5) that the participants might be unfamiliar with.

3.7.3. COMPLEX NESTED OBJECTS

The second assignment is about the (de)serialization of complex nested objects. The difficulty here is that the matter of complexity or nesting heavily influences the test results. For example, if there is only a nesting of one layer, the benefit of the SDK providing the data models out of the box is less than if there are 10 layers with different data objects that all need to be manually made for the NSDK.

There is no natural value to default to, so at first we decided upon 4 layers deep, each of them only incorporating one or two different data models except for one layer, the PC object, which incorporated six different data models, namely all kinds of different parts that together form a PC.

The first pilot run showed however that it took a lot of time for the participant to complete this assignment, and as a result we reduced the amount of layers by one. After a second pilot run however, the assignment was still too time consuming so we simplified the complexity of the PC object by reducing the amount of objects the PC object consists of from five to two.

3.7.4. DOMAIN TRANSLATION

The final assignment regarding (de)serialization is about domain translation. We define domain translation as the process of changing an object that might have a certain definition in one domain into another domain where it might have another definition. APIs likely work in a (slightly) different domain than the consumer, and as a result the consumer will have to adjust the responses of the API to fit their domain.

For example, a sale might just be a product id, a quantity, a price and a customer from the perspective of the administration department. But from the perspective of the warehouse department it is just a reduction in stock for a certain product id. This means that in order for this translation of domain to happen additional information might be needed or needs to be presented differently.

In a way, the complex nested objects assignment could also be considered domain translation, because the requests that were received needed to be transformed into another object. The difference between the two assignments however is that the focus in the complex nested objects task was more on creating certain data models and nesting them, keeping the computational tasks to a minimum. This assignment however has more focus on the computational tasks, keeping the data models themselves very simple.

During the pilot runs we found that the experiment as a whole was very time consuming, and certain parts needed to be removed. We decided that the effect of the SDK and the NSDK with regards to domain translation is likely to be minimal as the focus is more on the computational tasks, which both the SDK and the NSDK provide no help with. Rather than completely removing the assignment, we decided to move it to the end of the experiment as an optional assignment for those participants that finished the other assignments and still have some time left.

3.8. EVOLUTION ASSIGNMENTS

As mentioned in Section 2.2.2, evolution is inevitable, and as such we would like to investigate the following aspects of the breaking changes introduced by the evolution:

1. What type of breaking changes are considered to be difficult to resolve by the participants
2. Which resources the participants found to be useful for resolving breaking changes, as well as the relative importance of each resource
3. Whether participants think that solving breaking changes using one method (SDK or NSDK) takes more effort than the other
4. Whether participants anticipate that the code resulting from one method will be easier to test compared to the other

We will discuss all the aspects briefly one by one in Sections 3.8.2, 3.8.3 and 3.8.4, but first we will clarify the domain we used for the evolution assignment.

3.8.1. DOMAIN

The domain of the assignment is finding the shortest path between two train stations in central Tokyo. The API provides endpoints that allow consumers to query the distance of certain routes, the connections between certain stations, and everything else that is needed in order to determine the shortest path. The participants do not need to implement their own solution, solutions are already provided for both the SDK and the NSDK. The provided solutions however are based upon the previous version of the API, and a breaking change of each type has happened (see Section 5.5), rendering the provided solutions unusable. It is up to the participants to update the provided solution so that it works again. The participants will also be provided with a changelog containing a brief description of all the changes that have happened between the two versions and the full specification of the new version. The participants are asked to share their opinions in the form of questionnaires both after implementing the changes in their first iteration (using the SDK for group one, NSDK for group two) and a second questionnaire after also implementing the changes in their second iteration (using the NSDK for group one, SDK for group two).

3.8.2. DIFFICULT CHANGES

The SDK is capable of solving some changes automatically which normally would be breaking, with just a version upgrade. Therefore we would like to know which type of changes are considered to be difficult by the participants to see if these changes affect both the SDK and the NSDK equally. We came up with the following list of possible breaking changes that could happen to an API, with the type of the change on te left and a description of the actual change in the API on the right. Several of these breaking changes were also classified by Li et al. [19].

Upstream location changes	Prepend all endpoints with /api/v2
Parameter addition or removal	The origin and destination fields in Task objects have been renamed to start and finish respectively
Parameter type change	Route parameter changed to a query parameter
Return type change	Get a list of station objects instead of a list of station IDs
HTTP verb change	Changed POST to PUT
HTTP header change	Header changed from x-id-key to x-idKey
Endpoint replacement or removal	Route objects have to have an ID field created through a new endpoint
Other	Other

We asked the participants to share what type of breaking changes they found to be difficult after they changed the implementation for their first method (SDK for group one, NSDK for group two), but before they implemented the changes for their second method. We did this to get a better idea of the difference in which changes are considered to be difficult per method. If this was only asked after participants changed the implementation of both methods their answer would reflect more on the difficulty of changes across both methods, so only the changes that were difficult in both versions might be marked as difficult rather than on a per method basis.

3.8.3. USEFUL RESOURCES

The second aspect allows us to gain insight in which resources are considered to be useful by participants, and if there is a difference between which resources are considered useful for breaking changes in the SDK versus breaking changes in the NSDK. We defined the following four different resources that could be used by consumers in solving breaking changes:

Specification The documentation of the API: a list of all the endpoints and data models

Error messages at runtime The messages received from the API, the SDK or Unirest whenever something goes wrong during code execution

Changelog The list of breaking changes that happened between the old version and the new version

Compiler In case of the SDK, the compiler is able to detect any signatures that have changed and will not compile unless the signatures between the consumer implementation and the SDK match again, thereby giving the consumer information about what needs to be changed

The participants were asked to share which resources they found useful, and the relative importance of each resource by dragging them in order of usefulness. Just like the question about which changes were considered to be difficult (Section 5.5), this question was only asked after the participants implemented their changes for the first method, but before implementing them for the second method. The reason for this is also the same; to highlight the difference between which resources are considered useful for each method, rather than which ones are considered useful in both methods.

3.8.4. EFFORT & TESTABILITY

The third aspect gives us insight into whether one of the methods takes more effort than the other. This could be very interesting as maintenance is responsible for a significant part of the cost of software [20][21]. This means that any reduction in maintenance effort could result in significant (time) savings.

The fourth and final aspect is interesting for the same reasons as the third; the easier a piece of code is to test, the more likely it is that it is properly tested which could result in less maintenance effort that is needed down the line. Unfortunately the experiment was already getting quite sizeable and therefore we are unable to properly test whether or not one method is indeed easier to test compared to the other. So instead of actually testing it we asked the participants to make a judgement call, meaning that the data might not be as reliable as an actual test, but it will still give us an indication which method is likely to be easier to test.

3.9. PARTICIPANT INTERVIEWS

During the experiment, participants are expected to completely type out all of their answers. We suspect that participants are more likely to give shallower answers when having to explain their arguments through text rather than verbally. Therefore we decided to interview five of the participants to get more in-depth answers about the topics that they think are worth discussing. This means that the participant can choose which topics related to the experiment and everything around it they want to bring up because they deem it important. Furthermore, the face-to-face aspect of the interviews enable us to ask them to expand more on the answers they give wherever we feel that a more in-depth explanation is necessary. In other words, the interview is semi-structured. The dialogue nature of such a semi-structured interview allows us to really get to the core of their argument. As a result of this flexibility, each interview is also unique and does not cover a fixed amount of questions and answers. The interview will consist of two parts:

The first part will be about asking the participants about the experiment itself, did they feel that either side was not well represented or that something else was relevant towards the SDK versus NSDK discussion that was not present in the experiment? This will be asked to discover any important aspects of the discussion that we might have overlooked which could influence the conclusions we will draw from the experiment.

The second part will be about whether or not (autogenerated) SDKs should be used within Picnic, which parts in Picnic they could influence, and what problems they could solve or cause. This part of the interview is much more flexible, letting the answers of the participants steer the conservation.

3.10. SUMMARY

The experiment will consist of two assignments, one on (de)serialization and one on evolution. The (de)serialization assignment has the goal to gather insight into the existence of and reasoning behind any preference of the participants for one of the methods regarding effort and code quality for (de)serialization tasks in API consumption.

The evolution assignment gathers insight into handling breaking changes in API consumption, but has the same goal but for effort and testability instead. It also gathers insight on any differences between the two methods with regards to resource utility and the difficulty of different types of breaking changes.

Finally, we are also going to perform an interview with a select number of participants to gather more in-depth knowledge about the reasons behind the preference of the participants for one method or the other.

A flowchart representing the entire experiment as experienced by the participants can be found in Figure 3.1 below. Furthermore, the questions in the questionnaires used during the experiment can be found in Appendix A.

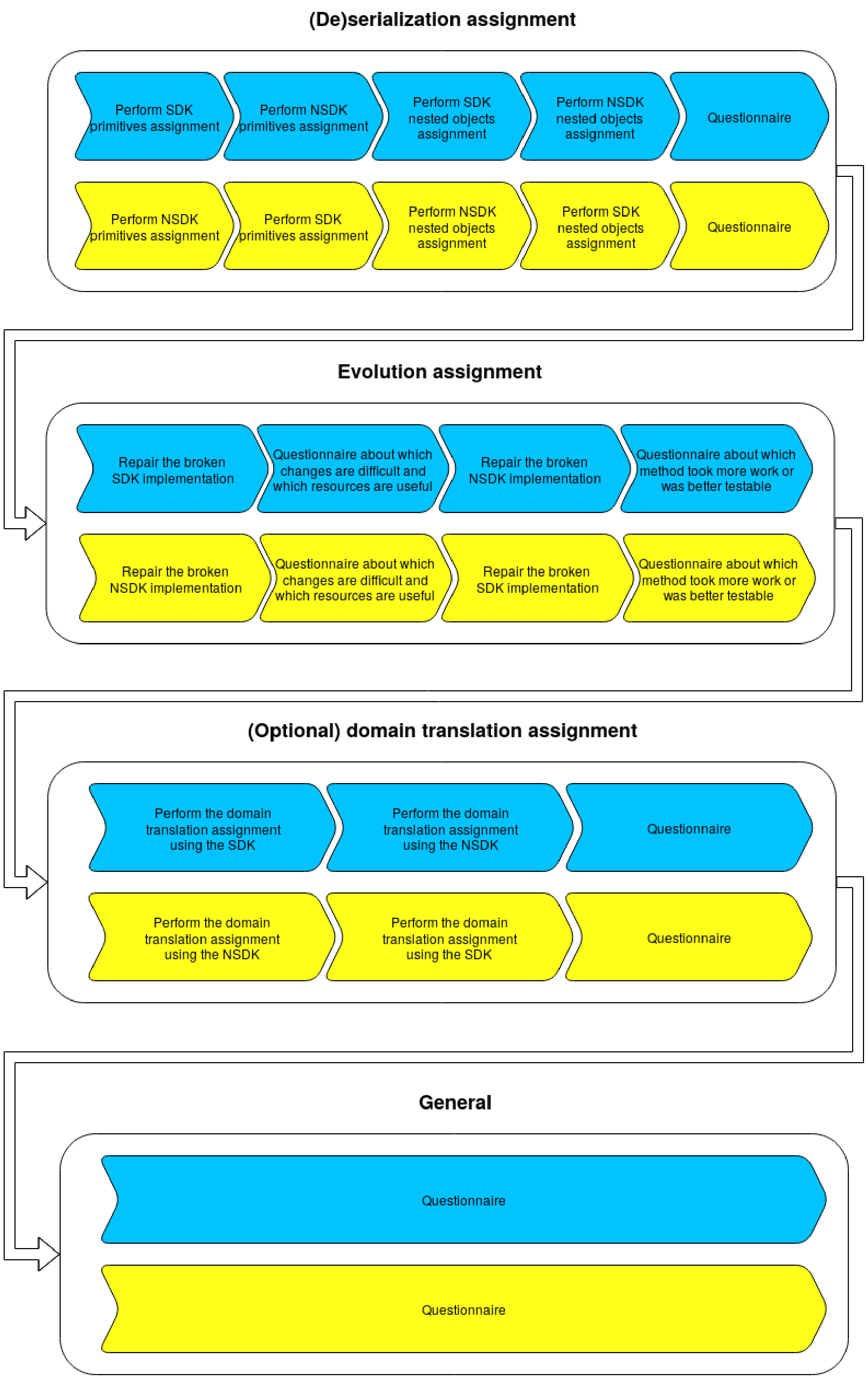

(De)serialization assignment
Perform SDK primitives assignment
Perform NSDK primitives assignment
Perform SDK nested objects assignment
Perform NSDK nested objects assignment
Questionnaire
Perform NSDK primitives assignment
Perform SDK primitives assignment
Perform NSDK nested objects assignment
Perform SDK nested objects assignment
Questionnaire
Evolution assignment
Repair the broken SDK implementation
Questionnaire about which changes are difficult and which resources are useful
Repair the broken NSDK implementation
Questionnaire about which method took more work or was better testable
Repair the broken NSDK implementation
Questionnaire about which changes are difficult and which resources are useful
Repair the broken SDK implementation
Questionnaire about which method took more work or was better testable
(Optional) domain translation assignment
Perform the domain translation assignment using the SDK
Perform the domain translation assignment using the NSDK
Questionnaire
Perform the domain translation assignment using the NSDK
Perform the domain translation assignment using the SDK
Questionnaire
General
Questionnaire
Questionnaire

Figure 3.1: Flowchart representing the entire experiment

4

(DE)SERIALIZATION: DATA ANALYSIS & EVALUATION

In this chapter we will show all the data that we collected about (de)serialization. We will look at our research questions related to (de)serialization (Section 2.3.1), and we will go over both the objective measurements as well as the collected opinions. The goal of this chapter is to discuss how the collected data affects the corresponding research questions and what conclusions we can draw from them.

It must be kept in mind that, as explained in Section 3.5, there are some differences between the SDK and the NSDK with regards to tooling. Normally, developers would be able to choose their own tools, and therefore choose tools that they are already familiar or proficient with. As this was not the case during the experiment, the reasons that are listed in Figures 4.1b and 4.2b with regards to tooling should be ignored.

4.1. VISUALIZING PARTICIPANT ARGUMENTATIONS

Throughout this dissertation we will make use of Sankey diagrams to visualize the reasons given by the participants for the preferences that they hold. They have multiple dimensions each with a different meaning and as such will explain how we created them and how they should be read.

In order to gain a visual overview of the reasons given by the participants, we must first categorize the answers given by the participants. This categorization is not easy given the wide range of different possible answers and the different ways of explaining the same concept. We already highlighted this problem in Section 3.2, and we use the card sorting technique described there. We opted for a Sankey diagram which allows us to represent both the categorizations that we have established through card sorting as well as the answers given by the participants. The colours of a flow in the Sankey diagram is a linear interpolation between the two colours we used for the groups: blue for group one and yellow for group two.

This means that if people from both groups gave the same reason, the color is a mix between blue and yellow, which is linear to the amount of people from both groups. For example, if a given reason was mentioned by both groups, but considerably more by participants from group one than from group two, the resulting color will be a green that is close to blue. We considered taking the difference in group size into account, but decided against it in the end because the group size are relatively close to each other (7-9), and it would have made all graph endings with a support of only one or two to also become mostly green, making the graph less useful, as everything would practically become a shade of green.

4.2. RQ #1: (DE)SERIALIZATION EFFORT

Research question #1

Does one of the two methods take more effort compared to the other?

In order to be able to answer this question we should first see the answers the participants gave to this question in the questionnaire. We plotted the answers of the participants in Figure 4.1a. The first thing to note is that not all possible answers are present in the graph. This is due to the fact that none of the participants chose those missing answers.

4.2.1. DOES ONE OF THE METHOD TAKES MORE EFFORT COMPARED TO THE OTHER?

If we look at the numbers that are actually present in the graph we can deduce that 81.25% of the participants were of the opinion that completing the serialization assignment using the NSDK took significantly more work compared to completing the same assignment using the SDK. The other 18.75% of the participants agreed that the NSDK method took more time, but only slightly. Interestingly enough this distribution is also roughly equal among both groups, so the order in which you did the assignments (SDK or NSDK first) has no influence on your opinion.

So now we know that one of the methods, namely the NSDK, did indeed take more effort compared to the SDK. But just knowing this fact is not very helpful without knowing the reason behind it, and therefore we decided to look at the reasons given by the participants for their preference.

4.2.2. VISUALIZATION OF THE REASONS GIVEN BY THE PARTICIPANTS

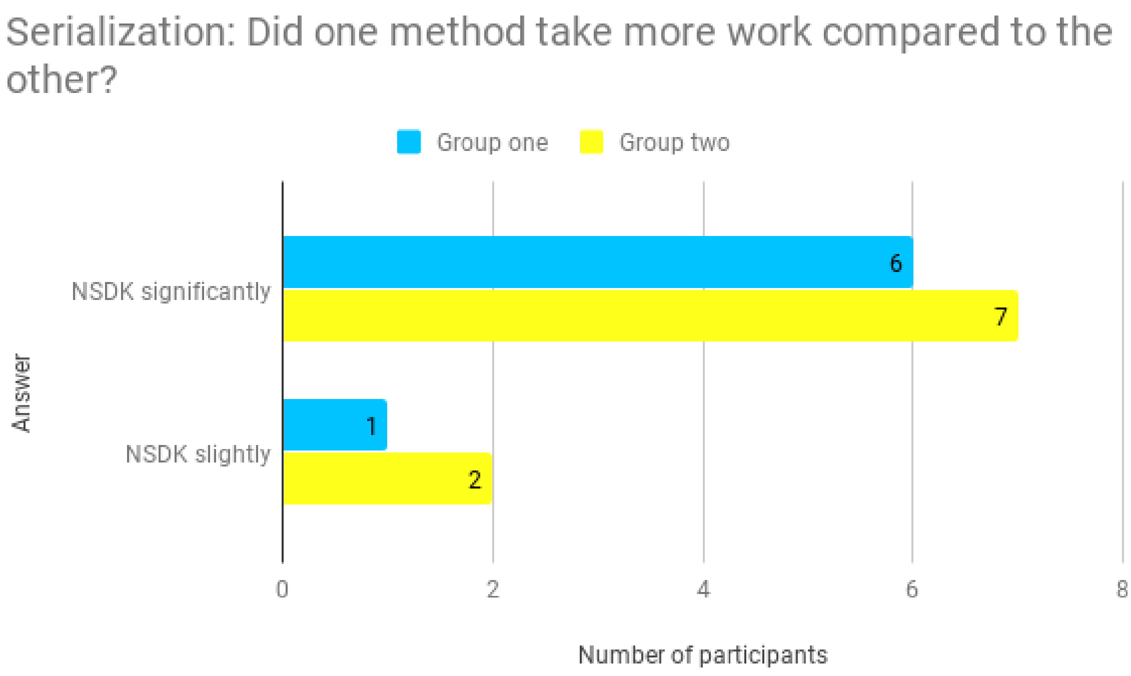
The reasons given by participants are short pieces of text with their argumentation behind their preference. It can be difficult to see trends in multiple pieces of text, and therefore we will visualize the reasons given by the participants. We do this as described in Section 4.1, by categorizing the answers given by the participants as described in Section 3.2, and the resulting Sankey diagram is displayed in Figure 4.1b.

4.2.3. INTERPRETING THE VISUALIZATION

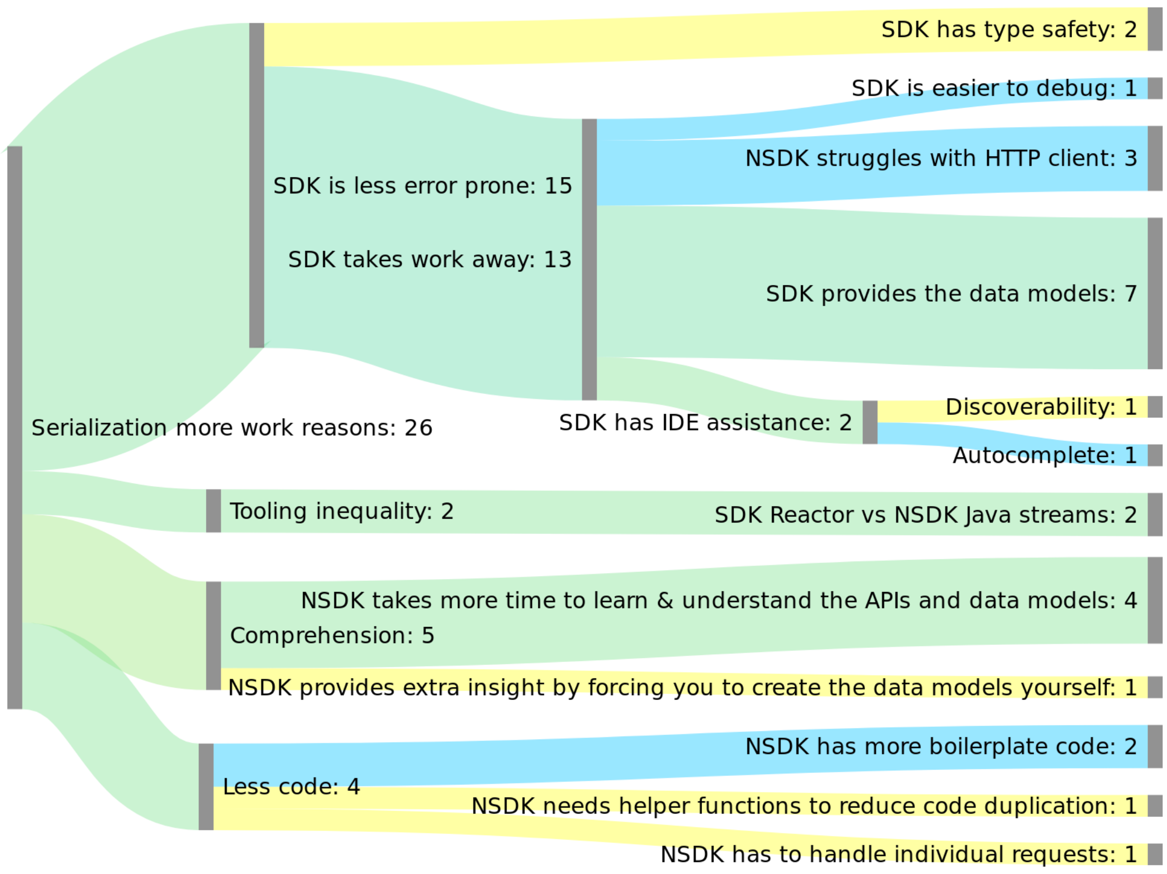
If we take a high level look at the given reasons, we can see that by far the largest category (57.7%) of given reasons is that the SDK is less error prone, which in turn is mostly caused by the SDK taking work away from the participant, removing the amount of things that could go wrong. Secondary reasons are that the SDK makes it easier to understand the API and data models, and that the solution requires less code. If we take a more detailed look at the given reasons, we find three noticeable details that we will discuss one by one.

The first detail is that the two reasons that were most often given by participants: `The SDK provides the data models` and `The NSDK took more time to learn & understand` have roughly equal support from both groups. This means that the largest contributors that determine the preference for the participants is group insensitive, and is therefore not a side effect of the setup of the experiment itself. This means that these reasons are very useful in determining the (dis)advantages of SDKs and NSDKs.

The second detail is that all participants that mentioned they had difficulties with using the HTTP client (needed in the NSDK) are all from group one. We suspect that the reason that this obstacle was more noticeable for the participants from group one is because their working solution from the SDK could not simply be converted to the NSDK variant; they had to figure out how to work with Unirest first. Participants from group two had to learn how to use Unirest while working on the solution to the problem at the same time. This means that unless the participant was unfamiliar with reactive frameworks there was nothing new that they needed to learn while converting their NSDK solution to the SDK solution,making the conversion feel simple while in reality the participants went through the same difficulty of learning Unirest as the other group, just at a different time. In a non-experiment setting this problem would likely not exist, as the developers would likely use a HTTP client they are already familiar with and thus would not have to learn one while creating solutions, and thus this reason should be disregarded.



(a) Survey answers given by the participants


Serialization more work reasons: 26
SDK is less error prone: 15
SDK takes work away: 13
SDK has type safety: 2
SDK is easier to debug: 1
NSDK struggles with HTTP client: 3
SDK provides the data models: 7
SDK has IDE assistance: 2
Discoverability: 1
Autocomplete: 1
Tooling inequality: 2
SDK Reactor vs NSDK Java streams: 2
Comprehension: 5
NSDK takes more time to learn & understand the APIs and data models: 4
NSDK provides extra insight by forcing you to create the data models yourself: 1
Less code: 4
NSDK has more boilerplate code: 2
NSDK needs helper functions to reduce code duplication: 1
NSDK has to handle individual requests: 1

(b) Reasons given by the participants

Figure 4.1: (De)serialization visualizations w.r.t. effort

The third and final detail is that type safety is only mentioned by participants from group two. Participants from group one only had to create data models that are identical to the ones that are given by the SDK to create a solution that would work. Participant from group two however had to closely look at the documentation in order to understand how they should communicate with the APIs; what information should be supplied and in which form. Doing this incorrectly would lead to problems with (de)serialization which would only be found at runtime because the compiler has no way of knowing what the API will return. This could lead to frustration for the participants of group two, making it extra memorable for them that the SDK did not suffer from such issues.

4.2.4. Conclusion: Does one of the two methods take more effort compared to the other?

All in all it seems that in terms of effort the SDK is considered to be preferable for everyone, the driving reason being that the SDK provides data models out of the box. The SDK also makes it easier for the participant to understand where and how to use them compared to the NSDK in for which the participants had to search and navigate through the documentation.

4.3. RQ #2: (De)serialization code quality

Research question #2

Does one of the two methods result in higher quality/cleaner code compared to the other?

Now that we have established that in terms of amount of work the SDK is preferable, we want to know if this SDK preference also exists when looking at code quality. We will use the same approach as before. We will first visualize the preferences given by the participants in order to gain some initial insights into the question. Then we will visualize the reasons given by the participants for their preference, and finally we will interpret this visualization, which will lead us to more insights that will aid in answering the research question at hand.

4.3.1. Does one of the methods result in higher quality code compared to the other?

We start by plotting the answers given by the participants, which is show in Figure 4.2a. This plot still shows a general preference (81.3%) towards the SDK, but this is already less decisive compared to figure 4.1a where everyone had a preference for the SDK.

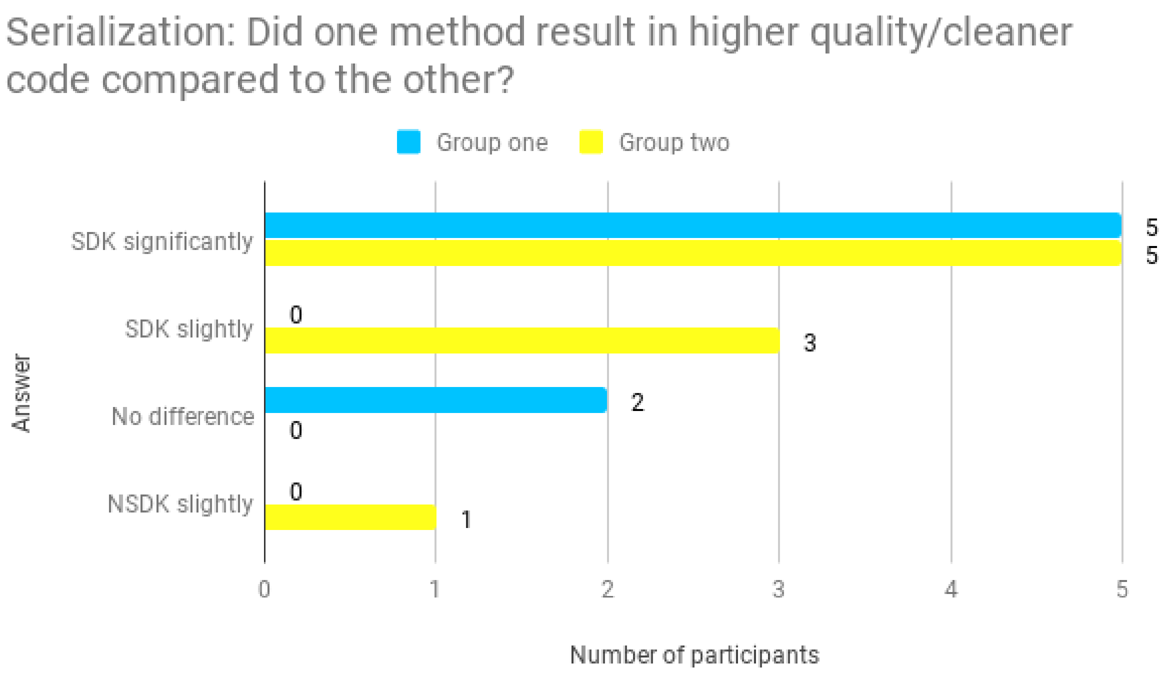
Now that the preference has been made clear, it is time to visualize the reasons behind the preferences again.

4.3.2. Visualization & interpretation of the reasons given by the participants

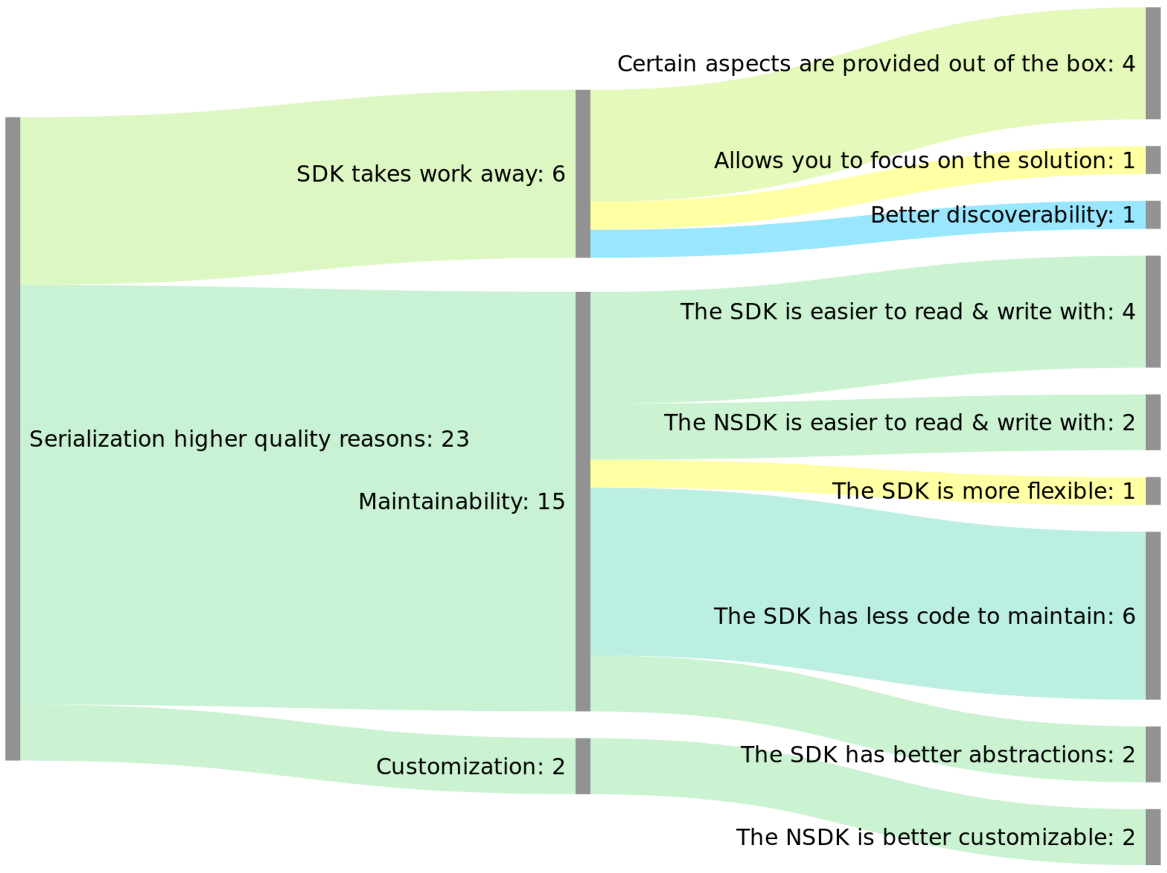
The Sankey diagram representing the visualization of the reasons given by the participants for their preference regarding code quality is shown in Figure 4.2b.

Now that the Sankey diagram has been created we can start interpreting it. Taking a high level look at this figure shows three main categories for why participants chose for one method over the other; `maintainability` , `SDK takes work away` and `customization`.

According to the diagram maintainability is the largest category and therefore is the key factor in determining whether or not code can be considered of high quality, accounting for 65.2% of the reasons given by the participants. The second largest category, `SDK takes work away`, is the same reason as why participants preferred the SDK in terms of amount of work. The SDK taking work away by providing certain aspects such as (de)serialization or settings headers out of the box means that developers do not have to concern themselves with these aspects and do not have to include and maintain them in the code base. The final category that is mentioned is customization. One of the big advantages of the SDK (as mentioned before) is that it is a out-of-the-box solution, but the downside is that there is little to no configuration possible. Meaning that if the



(a) Survey answers given by the participants


Certain aspects are provided out of the box: 4
SDK takes work away: 6
Allows you to focus on the solution: 1
Better discoverability: 1
The SDK is easier to read & write with: 4
The NSDK is easier to read & write with: 2
Serialization higher quality reasons: 23
Maintainability: 15
The SDK is more flexible: 1
The SDK has less code to maintain: 6
The SDK has better abstractions: 2
Customization: 2
The NSDK is better customizable: 2

(b) Reasons given by the participants

Figure 4.2: (De)serialization visualizations w.r.t. code quality

out-of-the-box solution is lacking for your use case then there is little you can do, and you might be forced to opt for the NSDK, where you have practically unlimited customization as you fully own the NSDK and can make changes as you please.

If we now take a closer look we can see that there are three interesting observations to be made. The first observation is that both the SDK and the NSDK are considered to be easier to read and write with. The SDK version is preferred by more participants, but it seems to come down to personal preference. Some people do not mind having auto generated code in exchange for being able to make REST calls as if it were a local library, while others are not willing to make this tradeoff.

The second observation is that group two seems to mention the out-of-the-box functionalities more often than group one, resulting in that category becoming relatively more yellow than blue in the diagram. This indicates that the order in which you performed the assignment could influence how much you value these out-of-the-box aspects in terms of code quality. Since both groups have to write code for both methods, it seems like the order in which you perform them should have little to no effect on your preference for them quality wise. We can only speculate if there is an actual reason for it or if it is merely coincidental.

The third and final observation is that group one has slightly more participants who mentioned that the SDK has less code to maintain compared to the participants from group two, causing the category to become relatively more blue than yellow. We suspect that this is due to the fact that for the participants from group two, creating the data models is simply part of the initial assignment, and while transforming their NSDK solution to an SDK solution they are unlikely to notice that the data models that they created have become obsolete. Participants from group one however needed to explicitly create all the required data models while trying to transform their SDK solution to an NSDK solution in order to make it work, making this requirement for the NSDK (or the lack of it for the SDK) very noticeable.

4.3.3. CONCLUSION: DOES ONE OF THE METHODS RESULT IN HIGHER QUALITY CODE COMPARED TO THE OTHER?

It seems that in terms of code quality the opinions are more divided, but the participants still seems to favour the SDK compared to the NSDK. The largest reason being that the SDK simply has less code that is part of the code base and therefore less code needs to be maintained.

5
EVOLUTION: DATA ANALYSIS & EVALUATION

In Chapter 4 we plotted all the collected data regarding (de)serialization and looked for any trends that would help in answering our research questions. In this chapter we will do the same but for evolution instead. This chapter will therefore also share the same setup, where we go over research questions regarding evolution one by one in Sections 5.2 to 5.5.

Before we can do this however, there is one anomaly that needs to be addressed: group discrepancy.

5.1. GROUP DISCREPANCY

We start by looking at Figure 5.1. From Figure 5.1a it immediately becomes clear that the group sizes have changed drastically. Where as for the (de)serialization assignment we still had 16 participants, for the evolution assignment we only had 7 participants. Even worse is that the relative group sizes have significantly changed as well. Whereas we have a group split of 7 - 9 for group one and group two respectively for the (de)serialization, for the evolution we have a group split of 1 - 6. This means that it is now impossible to make any comparisons between the two groups, since one group consists of only one data point. Furthermore, from 5.1b we can see that the only data point we have for group one did not give any explanations for their preference. It should therefore be kept in mind that it is impossible to determine whether a given reason by the participants is an actual difference between the two methods, or if it is simply a side effect of the experiment caused by performing the assignments in a certain order.

These effects are likely caused by the fact that the experiment was too long, and therefore not everyone was able to complete all the assignments. For some reason the participants from group one were affected more than the participants from group two, we cover this group discrepancy and the potential reasons behind it in-depth in Section 9.2.

5.2. RQ #3: EVOLUTION EFFORT

Research question #3

Does one of the two methods take more effort compared to the other?

This research question is the same as the research question in Section 4.2 but for the evolution assignment instead of the (de)serialization assignment. As such, we will follow the same approach to answer this question.

The plot representing the answers given by the participants is shown in Figure 5.1a. According to this plot there is still a consensus that the NSDK takes more work (57.1%), but this consensus is a lot smaller compared to the consensus in the (de)serialization task described in Section 4.2.1, where all participants considered the

NSDK to take more effort. We think that this is to be expected since the evolution assignment is less about writing new implementations and more about adjusting existing implementations. This makes the benefits of the SDK, such as providing data models out of the box, mostly irrelevant given that they are already present in the existing implementation.

5.2.1. VISUALIZATION & INTERPRETATION OF THE REASONS GIVEN BY THE PARTICIPANTS

The visualization of the reasons given by the participants were created the same way as in Chapter 4. The Sankey diagram is now entirely yellow due to there only being one participant left from group one and that participant did not state any reasons for his preferences, meaning that all reasons given in the Sankey diagram are from participants of group two. The Sankey diagram can be found in Figure 5.1b.

The relative reduction in consensus among the participants about which method takes more work when compared with the consensus of the same question in the (de)serialization task is not only noticeable in the preferences of the participants but also in the reasons for that preference given by the participants. By looking at the Sankey diagram it can be seen that the reasons given by the participants are heavily divided between those in favour of the SDK and those in favour of the NSDK. Even though the listed reasons are self explanatory, we want to note that unclarity of the SDK error messages and the debugging difficulty of Reactor can be seen as both realistic and as a threat to the validity of the experiment. This will be discussed in more detail in Chapter 8. Furthermore, the SDK also taking work away regarding the evolution task was to be expected, because even though the amount of work the SDK can provide for you out of the box is diminished while modifying existing implementations, they can still be of help depending on the changes that have to be made to the SDK. For example, HTTP header name changes are automatically changed in the SDK, requiring no additional changes for the developer.

5.2.2. CONCLUSION: DOES ONE OF THE TWO METHODS TAKE MORE EFFORT COMPARED TO THE OTHER?

The data indicates a minimal majority concluding that the NSDK takes more effort for the evolution assignment compared to the SDK. The given reasons arguing against the benefit of the SDK however point out factors that are not necessarily intrinsic of the SDK. These factors, the unclarity of the SDK error messages and the debugging difficulty of Reactor, can both be considered realistic as well as limitations of the experiment. In case these factors are considered to be realistic then nothing changes and the minimal majority concluding that the NSDK takes more effort still stand. If the factors are considered to be limitations of the experiment however then the minimal majority could potentially grow in a larger majority.

5.3. RQ #4: EVOLUTION TESTABILITY

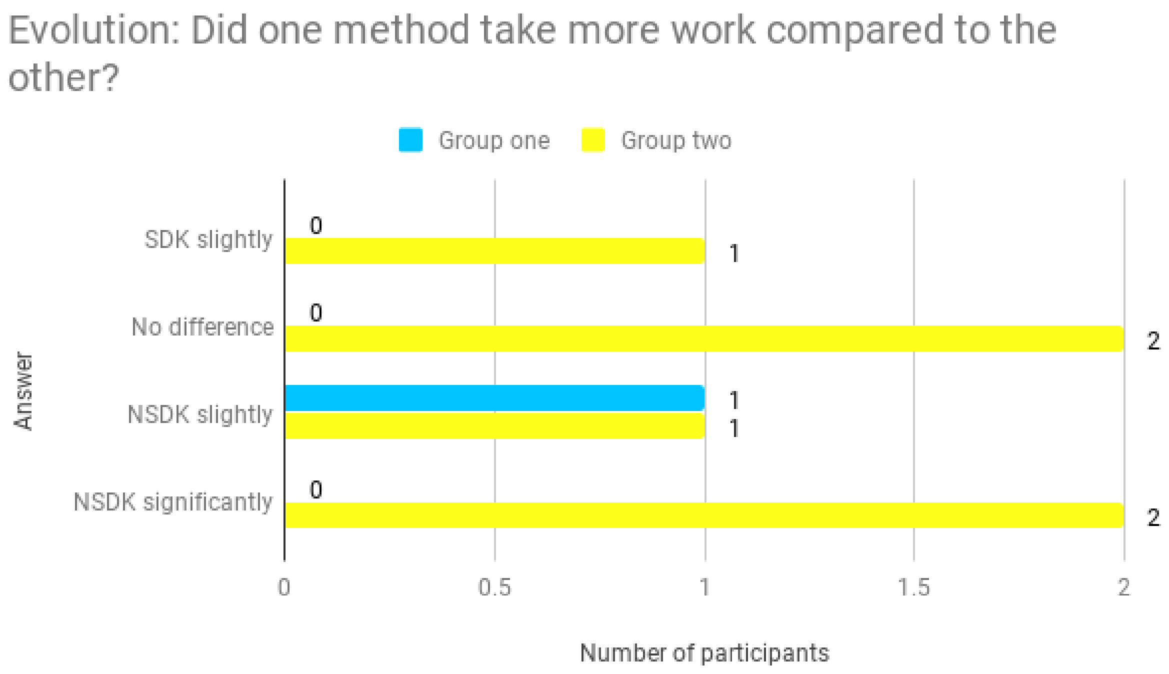
Research question #4

Does one of the two methods result in better testable code?

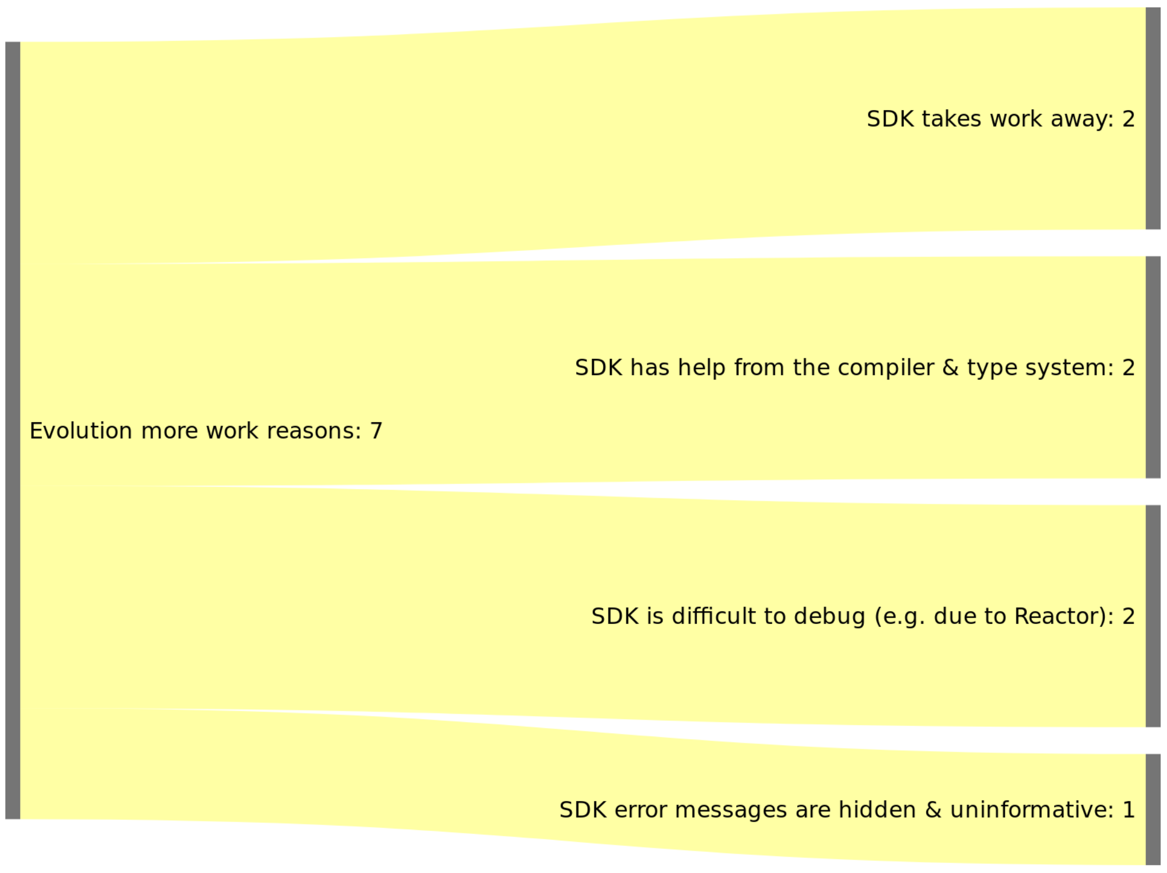
The same approach is used for this research question as for all the previous research questions, where we visualize the answers and reasons given by the participants, and then draw conclusion from those visualizations.

As we mentioned in Section 2.3.2, testability is a large factor in maintainability and can save a lot of effort down the line. Unfortunately due to the experiment already being quite sizeable we were unable to add some assignments that would thoroughly measure this. Therefore we simply asked the participants for a judgement call regarding the ease of testing for the evolution assignment based on the work they performed with both methods. The resulting plot can be found in Figure 5.2a.

This plot shows that there is a clear preference for the SDK regarding ease of testing. Only one participant did not have a preference, everyone else preferred the SDK either slightly or significantly. The next step is to look at the reasons given by the participants for their preference and that visualization can be found in Figure 5.2b.



(a) Survey answers given by the participants



(b) Reasons given by the participants

Figure 5.1: Evolution visualizations w.r.t. effort

This Sankey diagram shows that the given reasons can be divided into two main categories: tooling and code generation. We will go over both these categories and interpret them in Sections 5.3.1 and 5.3.2 respectively.

5.3.1. TOOLING

In terms of tooling, the participants seemed to judge that the SDK network calls are easier to mock than Unirest calls. At a first glance, this definitely seems plausible since the SDK calls are nicely abstracted behind generated methods inside the SDK, whereas the Unirest calls are part of the code base. If the participants were given more time and were not instructed to focus on completing the assignments rather than on code quality, then they could have made such abstractions themselves and then the SDK would not have had an advantage in terms of mocking. However, they would still have to fully test the Unirest calls, and test whether all (de)serialization works as expected. For the SDK, this is not necessary as the SDK is not part of the code base, but this means that all trust regarding the SDK fully behaving as expected needs to be placed in the code generator generating the SDK, which is difficult to do if you are not the owner of the generator or the SDK. Which brings us to the next issue: trust. We will discuss the trust issue in Section 5.3.3 after discussing the code generation category, as it contains reasons that are also trust based.

One of the participants mentioned that the SDK might be difficult to test since it uses Reactor, and Reactor can be hard to test for if you are not familiar with it. But since developers can choose what HTTP client they would like to use for their SDK (and thus can choose one without Reactor) we decided not to take this reason into consideration.

5.3.2. CODE GENERATION

The participants stated that in the NSDK version, you will have to fully test the Unirest calls, and all the (de)serialization that comes with it, which is not necessarily required for the SDK. Whether or not you have to do this for the SDK entirely depends on trust, which we discuss in Section 5.3.3.

The SDK having less versioning issues than the NSDK is likely to be correct, but it depends on how the API evolves. If the API only applies relatively non-impactful changes, such as changing HTTP headers, upstream location changes or HTTP verb changes then the SDK can automatically solve them simply by incrementing the version of the SDK to the new version (assuming that the SDK is deployed as a artifact somewhere). For other types of changes however, the SDK will, just like the NSDK, require changes in the implementation.

5.3.3. TRUST

As mentioned in the previous sections, whether or not the SDK needs to be tested entirely depends on trust. We assume that the SDKs are generated from a generator, but the same holds for manually written SDKs as well. We separate two different cases:

1. The SDK is created and used internally. This means that the company using it has full control over every aspect of it.
2. The SDK is from a third party. This means that companies using it have limited to no control over the SDK.

In the first case, trusting the generator and the SDKs that are generated by them should not be an issue, since they are completely in the hands of the company, and can be fully tweaked according to their needs. It might take quite some effort to fully test the generator and the results it produces, but this effort only has to be done once, and then testing the SDKs it produces can be omitted from the projects using the SDK.

In the second case it is a lot harder to trust the generator and the SDKs that are generated from it since you have no control over it. As a result, if you want to be completely sure that the SDK behaves as expected, you are forced to fully test the SDKs that you use at least once, even if they come from the same generator.

So if you use the SDKs internally, then the SDK approach could reduce the amount of effort in terms of testing down the line. If the SDK you use comes from a third party however, then it is recommended to fully test the SDK regardless, negating this advantage of the SDK. It could even be argued that the SDK might be at a disadvantage in that scenario, as the SDK might encompass many different endpoints and functionalities that you do not use. This forces the consumer of the SDK to either only test the endpoints and functionalities that they use, not knowing exactly what happens under the hood of the rest of the SDK, or try and fully understand the entire SDK order to be able to write suitable tests, in which case it might be less effort to simply write and test a small SDK yourself which covers only the endpoints and functionalities that you actually need.

5.3.4. CONCLUSION: DOES ONE OF THE TWO METHODS RESULT IN BETTER TESTABLE CODE?

While almost all participants agreed that the SDK is easier to test compared to the NSDK, it must be kept in mind that this was only a judgement call. Some of the participants argue that the NSDK requires complete testing of all (de)serialization and endpoints, which is not needed in case of the SDK if you trust either the SDK to be properly tested or the generator generating the SDK to be properly tested. While this assumption seems plausible if the SDK is for internal use, the assumption becomes dangerous to make if the SDK is provided by a third-party. Therefore we think that it heavily depends on the situation whether or not the SDK is easier to test compared to the NSDK.

5.4. RQ #5: TYPES OF BREAKING CHANGES AND THEIR DIFFICULTY

Research question #5

What types of breaking changes are considered to be difficult?

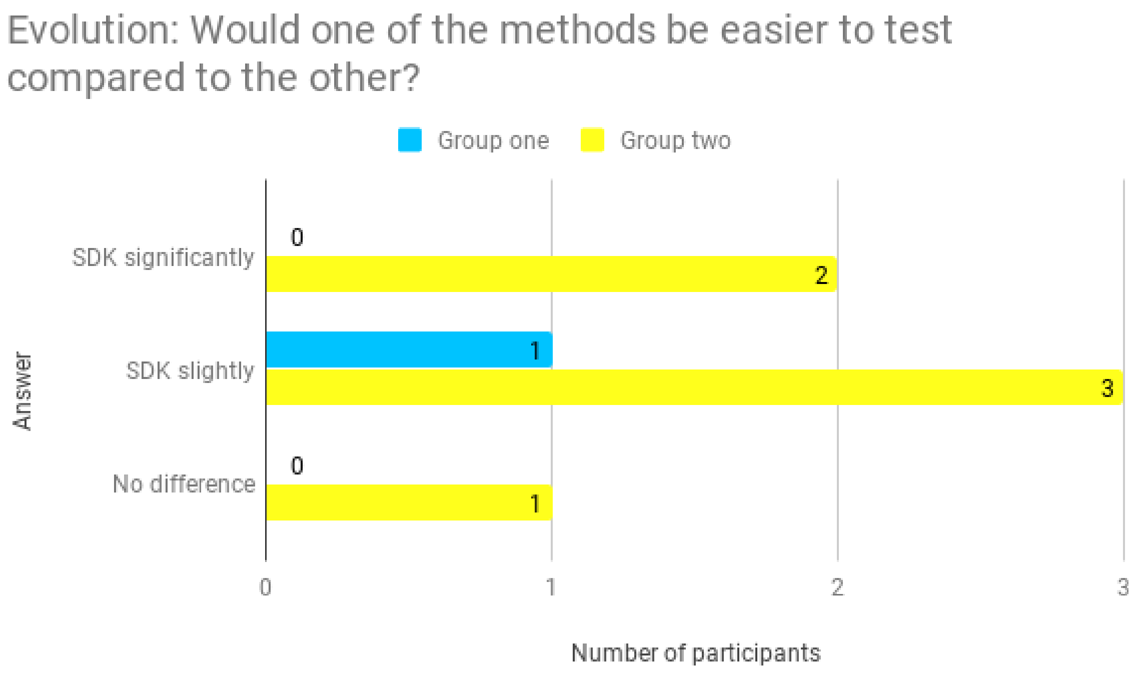
In order to gain insight into this research question, the participants were asked to mark all changes they considered to be difficult to implement and to clarify why they found those changes to be difficult in particular. The list consisted of all possible changes that we could think of that would require changes on the consumer's side as explained in Section 3.8.2. As a courtesy to the reader the list can be found below.

Upstream location changes	Prepend all endpoints with /api/v2
Parameter addition or removal	The origin and destination fields in Task objects have been renamed to start and finish respectively
Parameter type change	Route parameter changed to a query parameter
Return type change	Get a list of station objects instead of a list of station IDs
HTTP verb change	Changed POST to PUT
HTTP header change	Header changed from x-id-key to x-idKey
Endpoint replacement or removal	Route objects have to have an ID field created through a new endpoint
Other	Other

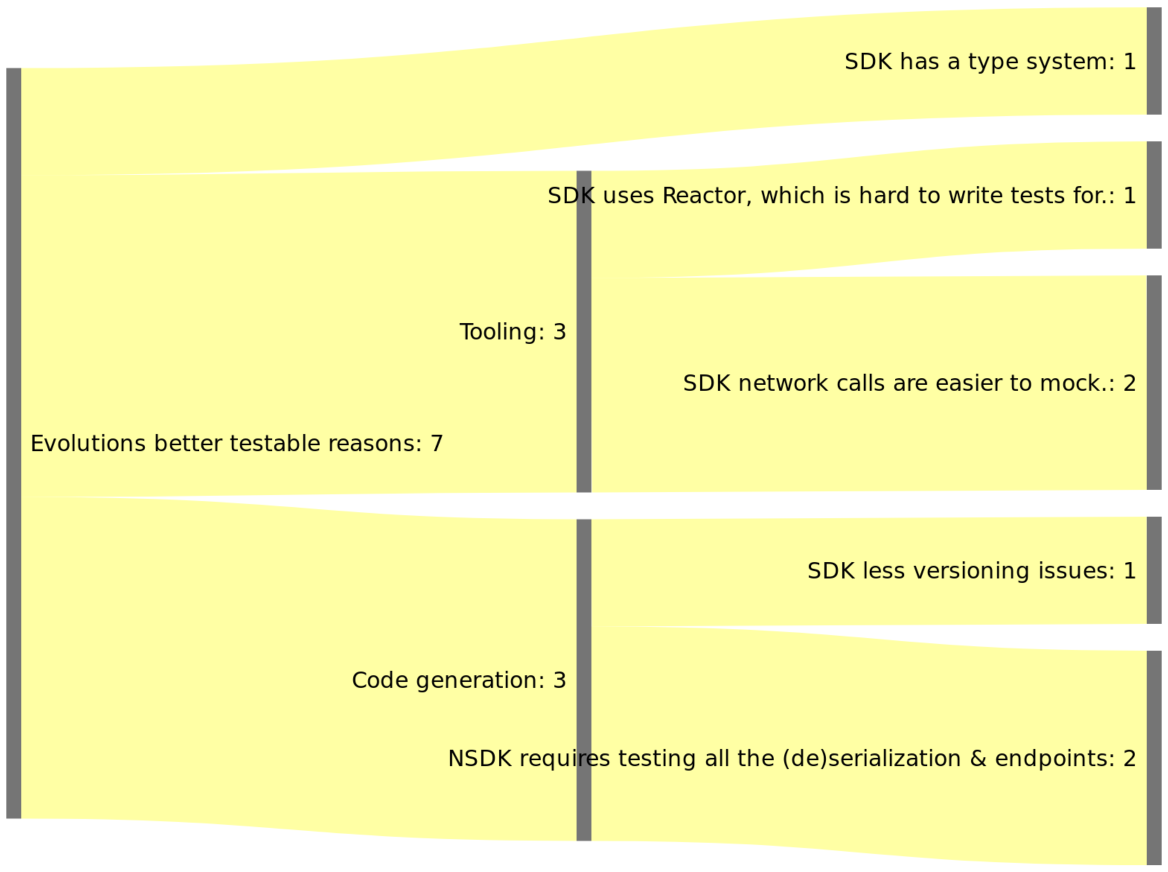
We counted the number of participants who considered a particular change to be difficult and plotted it in a bar graph visible in Figure 5.3a. It goes without saying that participants could select multiple types of breaking changes and mark them as difficult. It seems that majority of the participants had difficulties with the endpoint replacement or removal, while the other changes were only considered to be difficult by one or two participants. We will go over the listed changes and try to figure out why they are considered to be difficult in Sections 5.4.1 and 5.4.2.

5.4.1. ENDPOINT REPLACEMENT OR REMOVAL

Intuitively it makes sense that participants found the replacement or removal of an endpoint to be the difficult, because it is likely to cause relatively large changes in the implementation of the consumer, depending on the what exactly is removed and how it is replaced. If we now visualize the reasons given by the participants like we have done before in Chapter 4, we get the resulting Sankey diagram displayed Figure 5.3b. One aspect immediately stands out in this diagram: while the reason behind our intuition is listed in the diagram, it is only a small part of it. Most people that found the endpoint replacement or removal to be difficult apparently found it to be difficult due to the documentation or the task not being clear enough. Only two



(a) Survey answers given by the participants


Evolutions better testable reasons: 7
SDK has a type system: 1
Tooling: 3
SDK uses Reactor, which is hard to write tests for.: 1
SDK network calls are easier to mock.: 2
Code generation: 3
SDK less versioning issues: 1
NSDK requires testing all the (de)serialization & endpoints: 2

(b) Reasons given by the participants

Figure 5.2: Evolution visualizations w.r.t. testability

participants explicitly mentioned that the reason the endpoint replacement or removal was difficult was due to it requiring a deeper knowledge of the API, which was what we expected the main reason to be.

It seems that the reason that most participants had difficulties with endpoint replacement or removal is due to the new endpoint (route ID) simply being unclear. It was unclear where and how the endpoint needed to be used, and for one participant it was not even clear that there was a new endpoint. Unfortunately we can only speculate as to what the results would be if the documentation was not the limiting factor for these participants. Our intuition says that the results would likely be the same, with the endpoint replacement or removal still being considered as the most difficult change. We speculate this because the change is still a difficult one compared to the other changes. However, it could also be that the results would have been completely different, showing that another change would be considered the most difficult change to implement rather than the endpoint replacement or removal.

5.4.2. OTHER BREAKING CHANGE TYPES

The other types of breaking changes that had to be implemented were only considered to be difficult by a single participant. The exception being the return type change, which was considered to be difficult by two participants. Almost all of these difficulties can be attributed to one person missing the changelog, and therefore having more trouble determining the changes that needed to be made.

5.4.3. POINT OF CONCERN

Please keep in mind that, as explained in Section 3.8.2, the questions regarding the difficulty of the changes were asked after the participants implemented the changes using the first method, but before they implemented the changes a second time using the second method. Due to the group discrepancy this means that for the vast majority of participants (all but one) the questions were asked after using the NSDK method. This means that results of this experiment may be different when group one would be included.

5.4.4. CONCLUSION: WHAT TYPES OF BREAKING CHANGES ARE CONSIDERED TO BE DIFFICULT?

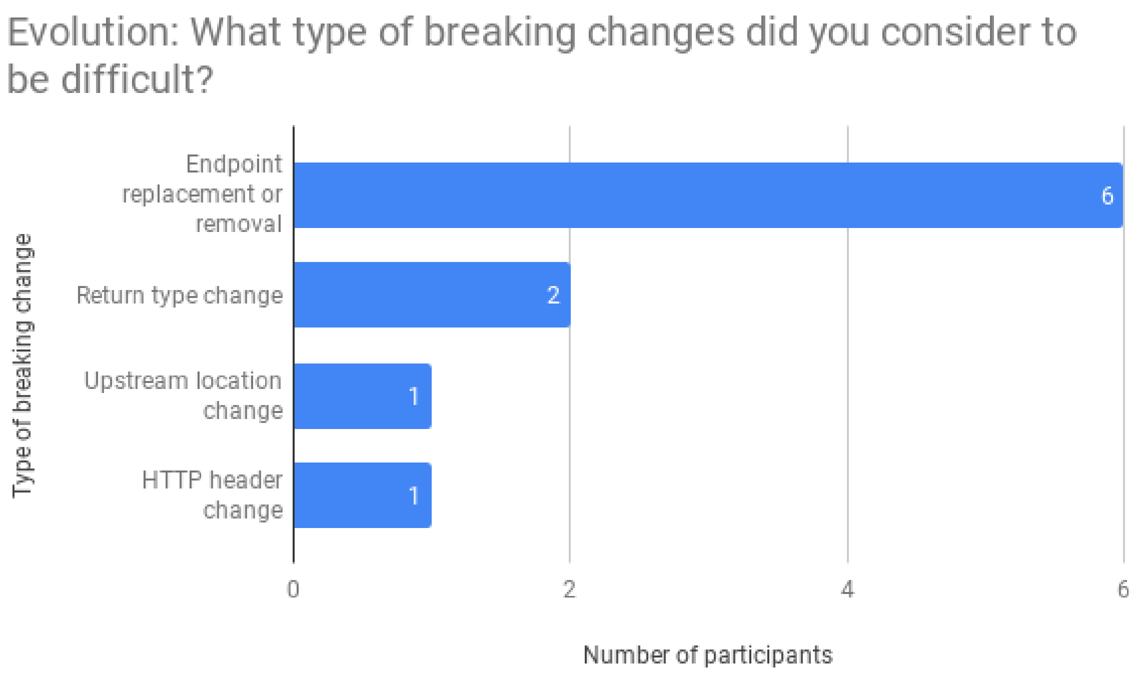
The vast majority of the participants found the endpoint replacement or removal to be the most difficult type of breaking change. However this seems mostly due to the documentation not being clear enough on these changes. Therefore it is entirely possible that this majority diminishes if the documentation was more clear in that area. Therefore we cannot conclude with certainty which type of breaking change is the most difficult for consumers.

5.5. RQ #6: USEFULNESS OF THE RESOURCES

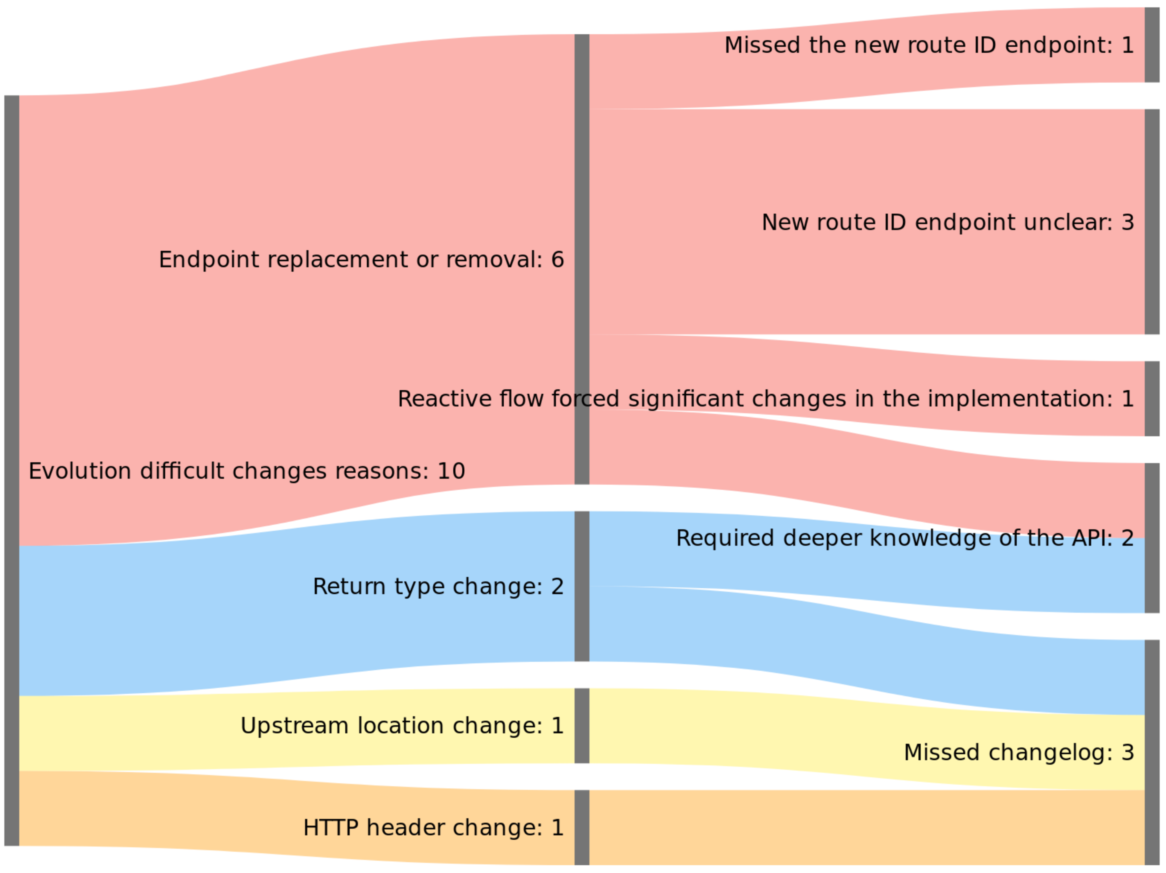
Research question #6

Which resources are considered to be useful during solving breaking changes?

We asked the participants about which resources they found to be useful while accommodating for the changes that needed to be made during the assignment. We specified four resources: `Changelog`, `Specification`, `Errors at runtime` and `Compiler`. The participants were also instructed to drag these resources in their order of usefulness, so the most useful resource would be in first place, and the least useful resource would be in fourth place. When a resource was not marked as useful by the participant we ignored the position of that resource, regardless of which position it was given. We put all of this information in Table 5.1. This table consists of multiple parts. We will briefly discuss each part in Sections 5.5.1, 5.5.2 and 5.5.3.



(a) Number of participants that considered the change difficult



(b) Reasons given by the participants

Figure 5.3: Evolution visualizations w.r.t. the difficulty of changes

Resource	**Useful** (%)	**Points**	**First** (#)	**Second** (#)	**Third** (#)	**Fourth** (#)
Changelog	62.5	17	3	1	1	0
Specification	62.5	15	2	1	2	0
Errors at runtime	62.5	16	1	4	0	0
Compiler	62.5	14	2	1	1	1

Table 5.1: The usefulness of each resource

5.5.1. USEFULNESS

The `Useful` column indicates for each resource by how many participants it was considered to be useful. Each resource was considered to be useful by 5 participants (out of the 8 participants that performed this part of the experiment), resulting in a 62.5% usefulness rating for each resource.

5.5.2. ORDERING

The `First`, `Second`, `Third` and `Fourth` columns indicate how often each resource was placed at that rank. For example, `Changelog` was ranked first (so considered to be the most useful) by 3 participants, second by one participants, third by one participant, and fourth by no one. These rankings were used to calculate a usefulness ranking score in Section 5.5.3.

5.5.3. USEFULNESS RANKING SCORE

In order to compare the usefulness of each resource, we came up with a point system which translates the rankings given by the participants into a score. The idea is that if two resources (A and B) are both considered to be useful, but resource A is ranked first by most participants while resource B is mostly ranked second, then even though both resource A and B are useful, resource A can be considered to be more useful than resource B. The point system puts this in numerical values, and work as follows:

- First place is worth 4 points
- Second place is worth 3 points
- Third place is worth 2 points
- Fourth place is worth 1 point

Each ranking position for each resource can then by multiplied by a point value and these can then by added together to arrive at a final point score which can be seen in the table. For example, the `Changelog` was ranked first by 3 participants, which is worth $3 * 4 = 12$ points. It was ranked second by 1 participant worth $1*3 = 3$ points, and finally it was ranked third by 1 participant which is worth $1*2 = 2$ points (and it was ranked fourth zero times, so no points are awarded for that). This results in a total points score of $12+3+2 = 17$ for the `Changelog`.

5.5.4. INTERPRETATIONS

The next step is to see what we can learn from these ranking scores. All resources come in relatively close to each other, meaning that even though some resources are considered to be more useful than others, the difference in usefulness is relatively small.

Intuitively, it is unsurprising that the `Changelog` is considered to be the most useful resource, as it lists all the changes that were made and thus is a great starting point to determine which parts of the implementation need to change.

Errors at runtime are considered to be the second most useful resource, which was unexpected. We expected errors at runtime to be the least useful resource, only being used whenever all the other resources were insufficient. But after reconsideration it makes sense that the errors at runtime are considered to be more useful than the specification because people will likely only look at the specification whenever either:

1. A change in the API requires drastic changes in the implementation and makes use of new endpoints or old endpoints have significantly changed.
2. The participants encounter an error at runtime because the implemented changes were incorrect.

The first case only happens for a couple of the changes made in the API, the other changes in the API can likely be solved by only looking at the changelog and the code. The second case can occur for each change made in the implementation, and is more likely to occur in changes that require drastic implementation changes (the first case). Therefore participants are more likely to encounter errors at runtime (and reading them as a result) than they are to consult the specification.

The least useful resource is considered to be the compiler, which contradicted our expectations. The compiler can aid in showing where breaking changes affect the current implementation, which we considered to be extremely useful, perhaps being second only to the changelog. However, just like the questions about the difficulty of the changes (in Section 5.5), these questions were only asked before the participants were able to experience both methods. This means that due to the group discrepancy that we had during the experiment the vast majority of the participants only experienced the NSDK version, meaning that the compiler was not aware of any changes that have happened on the API side as it is with the SDK variant where the specification that generates the SDK is updated and the types of the data models and the signatures of the endpoints change. This means that the compiler could only have been useful for the only participant of group one that was able to do this assignment.

This raises the question as to why two participants were then of the opinion that the compiler was so useful. It turns out that not all participants have read the instructions thoroughly and missed the part where they had to order the resources according to their perceived usefulness. Unfortunately, we have no way of knowing which or how many participants have missed this so the only thing we can do is to be mindful of this and take the data with a grain of salt.

5.5.5. Conclusion: Which resources are considered to be useful during solving breaking changes?

All resources seems to be considered useful, the differences in usefulness being marginal. The changelog is considered to be the most useful resource, as it contains all the breaking changes and therefore is a prime location to figure out what exactly changed. These findings must be taken lightly however, since not all participants followed the instructions properly causing the findings to be more unreliable.

5.6. Summary

It seems that even for changing existing code participants still prefer the SDK. There are two main reasons for this. The first reason for this being that the SDK is able to solve some required changes automatically. It must be noted however that this preference for the SDK is lower than in the (de)serialization task of Chapter 4, probably because the amount of effort the SDK is able to take away is also less than for (de)serialization task. The second reason is that participants anticipated the SDK to be easier to test, but this anticipation relies on the assumption that the SDK does not need to be tested, which likely only holds for internal SDKs and not for third-party SDKs.

6 Measurements: data analysis & evaluation

In Chapter 3 we mentioned that we were going to measure the amount of time each participant needed to complete each assignment as well as every API call made by the participants. In this chapter we will visualize and evaluate these measurements in order to gain extra insight into the topic of this research. Unfortunately however we have not been able to perform any meaningful analysis on the API calls made by the participants due to time constraints.

First, we will explain how the visualizations in this chapter are made and how reliable the measurements are in Section 6.1. Then we will go over the visualizations one by one and try to analyze them in Section 6.2. Finally we will conclude with some remarks on our findings in Section 6.3.

6.1. Reliability & visualizations

While looking at the data to analysis we noticed one issue regarding the reliability of the measurements. Some of the data points are heavy outliers compared to the others. For example, occasionally participants have accidentally closed the web browser tab in which the assignment was running and timing the experiment. This caused the timing to be off. As a result, the average amount of time each participant took will not be reliable.

To compensate for this problem all visualization in this chapter are based of the median of the measurements instead of the mean. This will minimize the effect of outliers on the visualization. To calculate the median we use the self-explanatory "mean of middle two" method.

6.2. Interpreting the visualizations

The first visualization we are interested in is the average amount of time each participant needed to complete one assignment. This visualization can be found in Figure 6.1.

The only point of observation here is that group one needed significantly more time to complete the primitives assignment compared to group two. While it is to be expected that group one spend more time in the (de)serialization assignment (primitives & nested objects) compared to group two due to the group discrepancy we discovered in Section 5.1, it is strange that all this extra time is spent in the primitives assignment as group one was faster than group two in the nested objects assignment.

To dive deeper into this observation we proceed by visualizing the median time duration of each group per assignment. This visualization can be found in Figure 6.2. In this visualization we can clearly see the carry-over effect we described in Section 3.3. The participants of a group always perform worse on their first iteration of the assignment compared to their second iteration. So group one always performs better in the NSDK assignments and group two always performs better in the SDK assignments. There is one notable exception

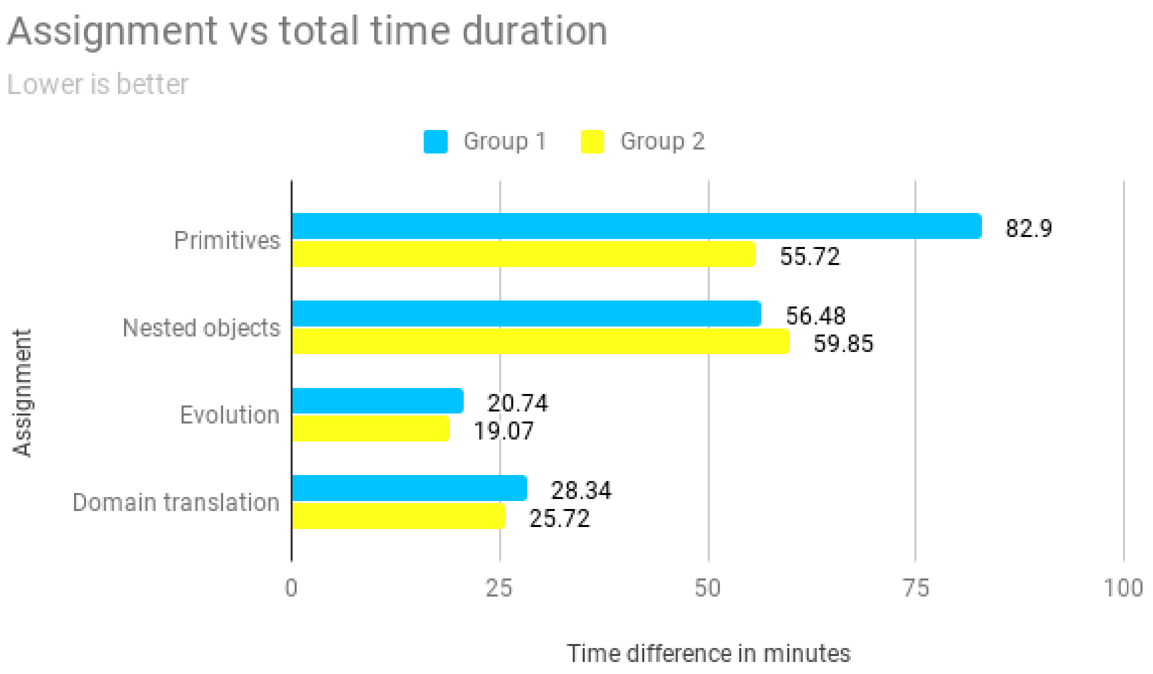


Figure 6.1: Total time duration per assignment

however, and that is the primitives assignment done by group one. Due to the carry-over effect we expect the second iteration to be faster, however in this case it was slower, which is surprising considering that for group two the second iteration was significantly faster.

This difference is made more clear in Figure 6.3, were we plotted the median time difference between the two iterations. A negative number means that the second iteration was slower than the first. In this plot we clearly see the significant difference between group one and group two for the primitives assignment. We suspect this difference exists due to the following reason:

The NSDK version takes more effort to figure out how it works compared to the SDK version. This can also be seen in the NSDK primitives time duration of group two, where group two needed roughly twenty minutes more to complete the assignment compared to the SDK time duration of group one. Since the SDK version is relatively straightforward this means that the second iteration for group two would be very fast compared to the first iteration, resulting in this massive time difference between the first and second iteration. Group one however had to figure out how the NSDK version works in their second iteration, causing them to slow down compared to their first iteration, making the gap between the speedup of the group seem extra significant, even though this gap is quite logical in retrospect.

6.3. CONCLUSION & SUMMARY

The visualizations have shown some observations that seemed interesting at first glance, but are to be expected when reasoning about the experiment. This means that, as we expected in Section 3.1, the objective measurements we performed did not provide much useful insight for achieving the goal of this research.

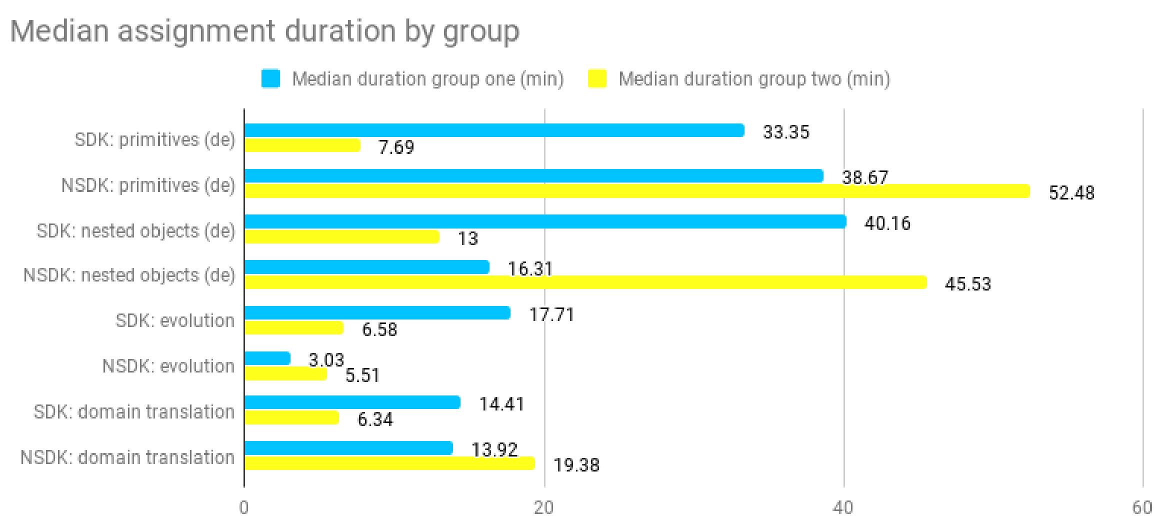


Figure 6.2: Median assignment duration by group

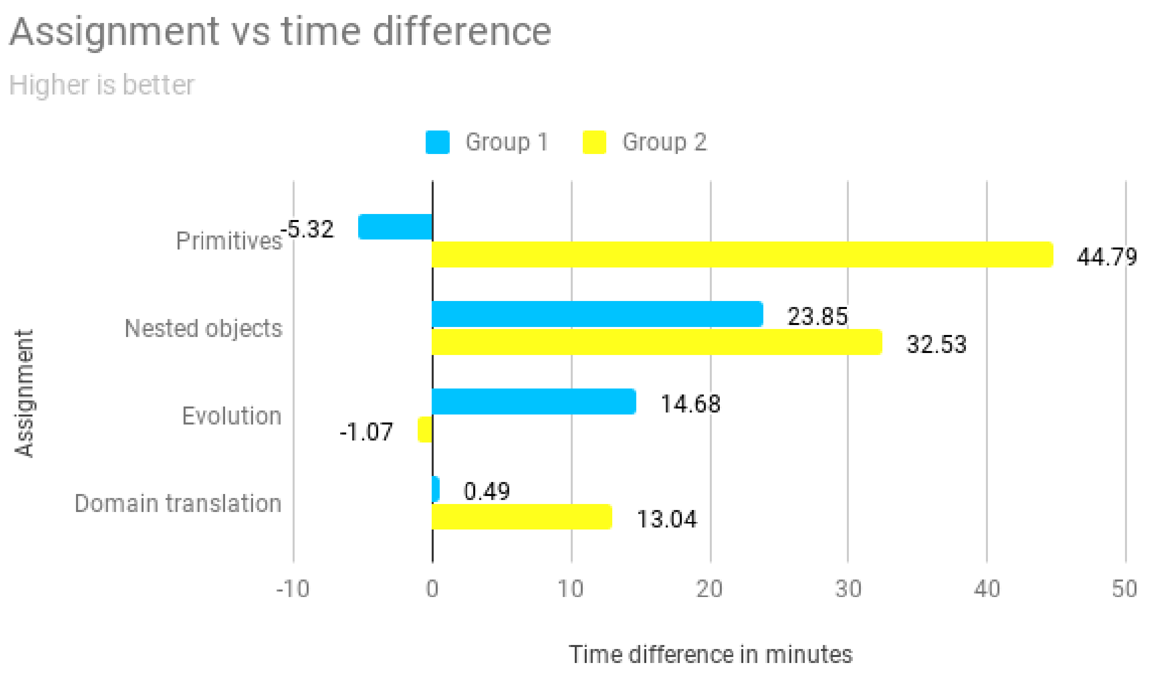


Figure 6.3: Participant speedup between first and second iteration

7
Participant interviews

While the questionnaires presented in the experiment allowed participants to share their opinions on both methods, it is unlikely that participants will share every detail of their opinion and will rather briefly highlight the reasons behind their answers. As a result most reasons mentioned here can also be found in Chapters 4 and 5, but in this chapter we hope to cover them more in-depth. In order to gain a bit more of this in-depth reasoning, we performed 15-20 minute interviews with five participants of the experiment. The participants have been carefully chosen for a variety of reasons. We will label the chosen participants and explain the reason why they have been chosen.

In this chapter we will first explain how the participants have been chosen in Section 7.1 and the setup of the interviews in Section 7.2. Then we will discuss the arguments that were brought up during these interviews in Sections 7.3 to 7.10 in order to provide more insight on any arguments that we might have missed or have not considered during the interpretation of the collected data from the experiment.

7.1. Chosen participants

We have chosen five participants to interview. Most of them are chosen for their experience and knowledge that they could share, others because they offered arguments that were not mentioned by others. This allows us to cover the topic as broadly and in-depth as possible. All the interview participants are Picnic employees. The participants will be labeled P1 to P5, and they have been chosen for the following reasons:

P1 P1 is chosen for P1's expertise and practical perspective as P1 is a core part of a large part of the codebase at Picnic, especially of the parts that would be impacted by the introduction of autogenerated SDKs.

P2 P2 is also chosen for P2's expertise, especially regarding P2's architectural knowledge inside of Picnic.

P3 P3 is chosen because P3 is a relatively inexperienced developer with no experience with SDKs or NSDKs. This makes P3 the perfect candidate to determine any difference in beginner friendliness of the SDK and the NSDK.

P4 P4 is chosen because P4 was the first participants to complete all the assignments.

P5 P5 is chosen because P5 is known to be critical of SDKs, and could therefore potentially provide argumentation regarding why SDK's should not be used.

With the exception of P4, all of the knowledge on which we based our choices of participants were known before the experiment. In other words, the responses they gave during the experiment were not considered during the interview selection process. This is because the interviews needed to be done quickly after the experiment while it was still fresh in the memory of the participants, giving us not enough time to sort and properly analyse the responses in the experiment before choosing the participants to be interviewed.

7.2. Experiment setup

As discussed in Section 3.9 the interview consists of two parts: one about the experiment themselves and one about the potential usage of (autogenerated) SDKs in Picnic. We will discuss both parts individually and more in-depth in Sections 7.2.1 and 7.2.2 respectively.

7.2.1. The experiment itself

The goal of the first part interview is to find any flaws or misrepresentations in the experiment itself. Was the SDK or the NSDK misrepresented? Or are there certain parts to the discussion that are relevant but were not represented in the experiment? This information allows us to account for these flaws, misrepresentations or missing parts of the discussion in any conclusions we will draw from this experiment.

The only question in this part of the experiment was the following:

> After having experienced the experiment, did you feel that any consideration for either the SDK or the NSDK was not represented or misrepresented?

7.2.2. Use of autogenerated SDKs at Picnic

The goal of the second part of the interview is to dive deeper in the opinion of the participant by bringing the subject closer to them. We do this by asking them their opinion on what direction Picnic should take. Should Picnic as a whole strive to autogenerate SDKs for every service or should they continue doing what they are currently doing and not let this SDK vs NSDK discussion affect them at all?

As discussed in Section 3.9 this second part of the interview is much more flexible, and thus the topics covered in the interview depend heavily on the participant and the topics that they value within this discussion. We start off the second part of the experiment with the following question:

> For Picnic, do you think that we should move towards an SDK for every service? Why or why not?

This will spark a discussion, and depending on the direction of the discussion and the liveliness of it we might also ask the following question to spark the discussion again:

> Autogenerated SDKs have many advantages, but they also have disadvantages. Do you think that these advantages outweigh the disadvantages?

We try to challenge the participant into sharing his personal opinion on these matters, resulting in this flexible structure. However, as a result of this structure the questions might be posed slightly differently depending on the content of the conversations and how it is going. But these questions should give the reader an idea about how the discussions are started and how we challenge the participants.

The participants have brought up several topics, for a variety of reasons. We will discuss these topics one by one in Sections 7.3 to 7.10, listing for each topic both who the participants was that we had this discussion with and what their argumentation was. It must be noted however that each participant has their own way of explaining things, making it hard for us to determine what the core part of their argumentation was, which is the same problem we encountered in Section 3.2. We decided to discuss the topics one by one for the sake of readability and context. Each topic consists of both the arguments provided by the participants as well as some background context provided by us wherever needed. We included their arguments to the best of our abilities, but please keep in mind that the possibility exists that there are some variations in how the participants meant their arguments and how we interpreted it. The opinions voiced by the participants are explicitly noted as such.

7.3. API first design

P1 argued that by generating an SDK from a specification file the producers are forced to think about how their API will be consumed while creating it. This disincentives simply writing the easiest solution for the producer to expose their API. This line of reasoning only applies to new endpoints however, since for already existing endpoints the specification file will just mirror the already existing endpoints.

Another point that P1 mentioned is that unless the server side (so the producer side) of the API is also generated (meaning that only business logic needs to be implemented), writing a specification file means extra work for the producers, and makes it likely that they will still write the server side first, and then simply mirror their server side in the specification file, thereby bypassing the advantage of API first design.

P4 argued something else. P4 was of the opinion that designing with the consumer in mind is more the responsibility of the developer itself rather than on the tools; Developers should care about the quality and usability of their code regardless.

7.4. CUSTOMIZABILITY

By generating the SDK from a source file using a generator you limit the amount of customization possible to the SDK to what the generator can provide. Assuming that companies will not write generators from scratch but would rather extend existing generators, customization is limited, as can be seen by looking at OpenAPI generator [1], which is one of the bigger generators and is actively maintained at the time of writing.

P2 argued that you could extend the existing generators to incorporate customization options as needed, however it is hard to build a SDK that can cater to everyone while not becoming so generic that is becomes useless. Especially because there are so many things that a consumer could want to tweak, such as authentication, retry policies, and many more. Furthermore, P5 argued that you should let the consumers decide on how to consume your API.

Nevertheless, P1 argued that if you have a lot of default cases limited customization is less of an issue and it might still be worth it to provide such an SDK. People that absolutely need certain customizations done can always use the NSDK approach, which will be more work but can be fully customized.

7.5. SDK SIZE & DEPENDENCIES

P4 argued that SDKs incorporate a set amount of endpoints that they cater to. They can also have certain dependencies that they rely on. If you only use a small subset of this set of endpoints, then it might be a big burden to your project to accommodate for this entire SDK. Especially for third-party SDKs where you have limited to no control it might be a big risk to rely on their SDK, as we already discussed in Section 5.3.3.

If consumers have access to the specification file, then the fixed size downside of the SDK can be mitigated, as it allows the consumers to only include the endpoints that are relevant for them. The fixed dependencies downside unfortunately remains.

All in all, if you only use a small amount of endpoints or would like to avoid certain dependencies that are included in the SDK it might be wise to opt for the NSDK method over the SDK method.

7.6. SINGLE SOURCE OF TRUTH AND EDIT

According to P4, whenever you use an SDK, both the consumer and the producer have an idea of what a certain endpoint is supposed to do and how it is delivered. This means that there is the possibility for a mismatch in expectations, even with documentation in place. Introducing standards decreases the risk of expectation mismatches but does not fully eliminate it, as developers can (just like with the documentation) have differing opinions on what the standard entails or how they work.

Using a generator to generate SDKs means that there is now a single source of truth. The generator is deterministic, meaning that it will always output the same code if given the same input. This means that standards can now be defined or configured inside the generator and the generator will always behave the same way, eliminating the possibility of expectation mismatches.

Furthermore, according to P1,whenever a certain standard changes, this change in standard only has to be reflected in the generator and now all generated SDKs adhere to the new standard, regardless of when they were made. For example, imagine if your SDKs use reactive code and you previously used RxJava, and you

[1] `https://github.com/OpenAPITools/openapi-generator/blob/master/docs/customization.md`

would like to move everything to Reactor. Without a generator this switch to Reactor would be slow, having to refactor every existing SDK using RxJava, but if you use SDK generation, only the generator has to be adjusted and all SDKs only have to be regenerated for them to adhere to the new Reactor standard.

7.7. More control over requests and responses

The SDK is only able to form requests and responses in a certain way, for example in the form of domain model objects that are defined by the SDK. According to P5, this means that if the consumer already has domain model objects that are different from the ones used in the API, the SDK prevents them from simply writing a (de)serializer that only listens for the relevant parts of the response and deserializes those parts accordingly. They will be forced to write some sort of adapter between their own data models and the SDK provided data models. The same applies for the serialization part of the requests.

P5 mentions that this is also a problem when there is a disconnect between the SDK and the API. Whenever the SDK expects a certain object from the API, but receives a different object, it will try to deserialize that object and fail. The NSDK approach allows the consumers to build fallback mechanisms more easily. They simply offer more fine grained control to the consumers.

7.8. Insight into consumer usage

An argument could be made that the NSDK offers more insights to producers into which endpoints are used by the consumers of their API, making it easier for the producers to change their API if necessary. For example, if no one uses part of your API, it could be deprecated.

On the other hand, according to both P2 and P5 consumers are unlikely to notify the producers that they are using their API, as it is counterintuitive. The purpose of exposing an API is to provide a contract to anyone willing (and allowed) to use your API that you, as a producer, deliver certain data if they adhere to the API. The only moment when consumers will notify producers is whenever that contract is broken, meaning that unless the producer will continuously monitor the usages of their API they cannot easily make such API changing decisions anyway.

It could even be argued that the SDK might be preferable for the producers, as it gives them some control on how their API is used.

7.9. Source of bugs

P1 mentions that using the NSDK method means that it will be part of your code base, meaning that it is a potential source for bugs and needs to be maintained. It is generally very boilerplate heavy code and therefore not really engaging to write either. Generating this from an SDK eliminates these type of bugs entirely assuming that the generator is not faulty.

7.10. Setup of the SDK generation

P4 fears that initial setup of the SDK generation is likely to be quite difficult. Finding suitable generator(s) and modifying it to comply with your standards might take quite some time, especially if you have a lot of customization that needs to be applied. Furthermore, distribution of the SDKs or the SDK API specifications need to be thought out and facilitated. Therefore, using and maintaining generator(s) might only be worth it if applied at a certain scale to offset the initial setup.

8
Threats to validity

In this chapter we discuss all threats that potentially influence the validity of the research. We will explain the threat at hand, the reason why we consider it to be a threat, and the impact it has on the research itself.

We have divided the threats to the research in three different parts: internal, external and construct [22]. We will go over each of these one by one in Sections 8.1, 8.2 and 8.3 respectively.

8.1. Internal

In Section 5.2.1 we mentioned that the unclarity of the SDK error messages and the debugging difficulty of Reactor could be seen as a threat to the validity of this research. Reactor was not a technology that was necessary for the SDK, and by using a reactive framework for the SDK but not for the NSDK means that we have introduced another variable that could potentially be responsible for any discrepancies between the two methods rather than the fundamental differences between the SDK and the NSDK. This means that this threat is an unnecessary one introduced by a poor experimental setup.

The threat from this mistake to our experiment is likely limited to Section 5.2, where we try to determine any differences between the SDK and the NSDK in terms of effort with regards to evolution. We think the threat is limited to this part as this is the only part where a significant number of participants mentioned the unclarity of the SDK error messages and the debugging difficulty of Reactor as a reason for their preferential choice between the SDK and the NSDK (in terms of effort with regards to evolution).

8.2. External

The fact that our participants mostly consisted of Picnic employees could be considered an external threat to the validity of this research, as the results of this research might not hold up when performed with a more diverse group of participants. We have mitigated this threat by reducing the amount of variance in technology experience between the participants by using technologies that most Picnic employees are familiar with as described in Section 3.4.

8.3. Construct

We have two potential threats to the construct of this research. In this section we will discuss them both one by one.

8.3.1. Unfair setup between the SDK and the NSDK

The experiment was set up in such a way that the assignments could immediately be started for both the SDK and the NSDK, meaning that all initial setup was already completed. However, as discussed in Section 1.6, the

SDK however comes with a lot of extra benefits out of the box such as data model objects. We consider this to be one of the benefits of the SDK and therefore did not provide the NSDK with such data model objects, forcing the participants to create them themselves. Some of the participants argued however that they felt that this was unfair towards the NSDK, saying that the NSDK should have been provided data models as well, eliminating much of the manual work that needed to be done for the NSDK.

While we agree that this would significantly change the results for the (de)serialization effort that we discussed and analysed in Section 4.2, we argue that normally the data models used internally by the consumers would likely be different than the ones used by the web API, forcing the consumers to create these data models for the NSDK, as was the case in the experiment. As discussed above, we consider having the data model objects out of the box to be an advantage that is part of the SDK, and therefore do not think that this is poses a threat to the validity of this research.

8.3.2. CARRY-OVER AFFECT IN COUNTER-BALANCING

As we described in Section 3.3, the group division that we performed is only effective when the carry-over effect between the two groups is constant. If one of the groups is able to have a larger carry-over effect, i.e. learns faster than the other group, then the comparisons made in this thesis might not be valid because if the groups would be switched then conclusions might have been entirely different due to one group being able to adapt to the experiment faster than the other.

While we definitely consider this to be a threat to this research, we do not see any reason as to why this possibility has a significant likelihood of happing. We have mitigated this threat by dividing the participants according to their previous development experience as explained in Section 3.3.

9
Research limitations

In this chapter we discuss the limitations of the experiment. We will go over these limitations one by one in Sections 9.1, 9.4 and 9.5 and explain for each of them what the limitation is, how it came to be and what effects it might have on the research in this thesis.

9.1. Categorization

As described in Section 3.2 we explained that we used card sorting in order to group together similar responses given by the participants to reduce the number of different responses, in turn allowing us to perform some comparative analysis.

While grouping similar responses together does reduce the number of different responses, it merely moves the problem. The similarity of the answers is entirely decided by the people performing the card sorting, meaning that any bias or errors on their part will be reflected in the results, more so since we decided to perform the categorization ourselves as also described in Section 3.2. We acknowledge the possibility of such bias and errors and consider it to be a limitation of the research.

9.2. Group discrepancy

In Chapter 5 we established that the two participant groups were not evenly divided anymore. To get a better overview of the group imbalance per assignment, we counted the number of responses on each of the questionnaires, resulting in Table 9.1. This table makes it obvious that the people that were unable to complete any assignment past the serialization assignment primarily came from group one, rather than evenly from both groups.

As can be seen from the table, 7/9 participants of group two were able to make it past the serialization assignment, while only 1/7 participants of group one was able to accomplish te same. This means that 3 times the amount of people (2 for group one and 6 for group two) for group one got stuck in the serialization assignment compared to group two, resulting in a large group discrepancy. This group discrepancy is strange considering that most participants (of both groups) seemed to prefer the SDK while making the assignments.

The fact that this group discrepancy happened forces us to change how we approach the analysis of the collected data and as a result how we will formulate answers for the hypotheses. Sections 9.3 to 9.7 will go over the possible causes of this discrepancy as it could contain information which might be relevant. Then we will determine how this discrepancy influences our research in Section 9.7.

9.3. Determining the cause

We came up with the following possible explanations as to why this discrepancy happened, and will go through them one by one in Sections 9.4, 9.5 and 9.6 respectively.

assignment questionnaire	**group 1 (#)**	**group 2 (#)**
serialization	7	9
first evolution	1	7
second evolution	1	6
domain translation (optional)	1	2
post experiment	1	4

Table 9.1: Amount of questionnaire responses per group per assignment

years (#)	**participants (#)**	**participants (%)**
Less than 1	0	0
Between 1 & 3	3	18.8
Between 3 & 5	1	6.3
Between 5 & 10	11	68.8
More than 10	1	6.3

Table 9.2: Developer experience responses

- The developers of group two were more experienced in some way compared to the developers of group one.
- There were one or several roadblocks present that participants from group one had to deal with which were not (or to a lesser extent) present for participants of group two.
- The discrepancy is purely coincidental.

9.4. EXPERIENCE DIFFERENCES

Before the experiment we asked the participants to share their amount of experience as a developer and used this data to divide them into two groups as described in Section 3.3.

The ranges and their distribution can be found in Table 9.2. According to the table, 68.8% of the participants fall into the same range. This is an indication that chosen ranges are suboptimal, especially because one of the ranges is unused. This could have been prevented by making better estimations about the participants joining the experiment. For example, due to the way the definition of experience was formulated, combined with the fact that the participants were either software engineers or computer science students it was to be expected that there would be no participants with less than one year of experience. Spreading the ranges to better conform the participants could have resulted in higher accuracy regarding the developer experience of the participants without losing too much anonymity. It must be noted however that while it is probable that more narrow experience ranges result in a better group division, we think that innate differences between participants largely negate the effect of smaller experience ranges. Different people learn at different speeds so a certain amount of experience for one person affects their skills and abilities differently than the same amount of experience for another person. These innate differences could be taken partly into account through more extensive participant selection but we considered such an extensive selection to be out of the scope of this thesis.

There are five different reasons regarding experience where a significant difference in said experience between participants could result in the observed group discrepancy. We will go over each of them in Sections 9.4.1 to 9.4.5.

9.4.1. EXPERIENCE RANGES TOO BROAD

The first reason is that given the high concentration of participants with development experience between 5 and 10 years, and the fact that participants were divided solely on their self reported development experience, it is possible that one group has significantly more participants with an experience level on the higher side

(8-9 years) of that range compared to the other group, where the participants would be on the lower side (5-6 years) of that range. As discussed earlier in Section 9.4, we do not believe that the ranges of experience being to broad could cause the observed group discrepancy and we would sooner believe the innate difference between the participants to be the cause.

9.4.2. JAVA VS OTHER LANGUAGES

The second reason is that the participants were only asked to share their development experience in general, unrelated to a specific programming language. The experiment was entirely in Java, so a difference in Java experience will affect the speed at which a participant is able to complete the assignments. Even though all participants did have some Java experience, some of them use the language on a daily basis while others did not use the language in years and are currently working with other languages that might not resemble Java.

9.4.3. DAY-TO-DAY ACTIVITIES

The third reason is that while certain participants may have acquired a lot of development experience, not all of them are necessarily still developing on a daily basis. They might have changed from a developer role to a more managerial role. It could then be the case that they might be slower than a more junior developer, simply because remembering the exact syntax might take a while, causing them to be slower and therefore have more difficulty in completing the assignments. Therefore the day-to-day activities of the participants could influence their performance in the experiment.

9.4.4. REST API EXPERIENCE

The fourth reason is that not all participants may have the same amount of previous experience regarding REST APIs. Some of them might work with REST APIs on a daily basis, while for others it might be their first time using a REST API. We suspected beforehand that this might become a factor and therefore we asked the participants to share their experience regarding APIs before performing the experiment. The amount of REST API experience per group can be found in Table 9.3. Please keep in mind three things while looking at this table:

- The group sizes were not ideally distributed.
- Due to the relatively low number of participants, each participant is responsible for a large percentage of the total. So shifting a participant from one group to the other not only has a large impact percentage wise in the respective category, but across all categories because the group sizes would change.
- Shifting people from one group to another not only has an effect on the distribution of REST API experience, but also on all other distributions (developer experience, reactive experience, etc).

Keeping those points in mind, the distribution of REST API experience is pretty even across the groups, except for the “Every few weeks" category, where it could be better distributed (both for this category as for total group size) if one of the participants of group two would move to group one. This was originally the case but due to one participant using the wrong group number we unfortunately ended up with the current distribution.

Based on the tables, we think that distribution of the REST API experience wil not have had a large impact on the speed discrepancy between the groups, although we do not dare to exclude it either.

9.4.5. TOOLING EXPERIENCE

The final reason is that not all participants may have the same amount of previous experience regarding the tooling that is used during the experiment. As described in Section 3.5, the experiment makes use of Reactor and Unirest. Just like we suspected beforehand that a difference in REST API experience might become a factor, we also suspected that Reactor and Unirest experience might also become a factor, and as such we asked the participants to share their experience regarding these two tools as well. The amount of experience with reactive frameworks and Unirest per group can be found in Tables 9.4 and 9.5 respectively.

REST API experience	**group 1 (#)**	**group 2 (#)**	**group 1 (%)**	**group 2 (%)**
None	1	0	14.3	0
Every few months	1	2	14.3	22.2
Every few weeks	2	4	28.6	44.4
Weekly	1	1	14.3	11.1
Daily	2	2	28.6	22.2

Table 9.3: REST API experience per group

Reactive framework experience	**group 1 (#)**	**group 2 (#)**	**group 1 (%)**	**group 2 (%)**
None	2	2	28.6	22.2
Every few months	1	3	14.3	33.3
Every few weeks	2	1	28.6	11.1
Weekly	0	0	0.0	0.0
Daily	2	3	28.6	33.3

Table 9.4: Reactive framework experience per group

Keeping in mind the same points as for the REST API experience, it seems that the distribution in terms of reactive experience is roughly equal, although it is hard to tell. Group two has sightly more daily reactive users, but the majority of their participants use reactive frameworks only once every few months. Most participants in group one on the other hand use reactive frameworks once every few weeks, but relatively speaking they also have more users that have never used a reactive framework before. The distribution in terms of Unirest experience is nearly optimal, which is no surprise since most people never used Unirest before, making this a hurdle that all participants had to overcome. Therefore we can conclude that a discrepancy of Unirest experience between the two groups is not a factor for group two being faster than group one.

9.4.6. EXPERIENCE DIFFERENCES: CONCLUSION

We think that Unirest experience has not played a factor in the speed discrepancy between the two groups, and we think that both reactive and REST API experience will only have had minimal impact, although it is hard to say with certainty. Unfortunately the data from the experiment is insufficient to say anything about the likelihood of the other three possible experience related explanations. For convenience purposes we listed these three remaining experience related explanations below.

- The innate differences between participants
- The questionnaire asking for development experience in general and not for Java experience while the experiment could only be performed in Java
- The difference in day-to-day activities of the participants

Unirest experience	**group 1 (#)**	**group 2 (#)**	**group 1 (%)**	**group 2 (%)**
None	6	9	85.7	100.0
Every few months	1	0	14.3	0.0
Every few weeks	0	0	0.0	0.0
Weekly	0	0	0.0	0.0
Daily	0	0	0.0	0.0

Table 9.5: Unirest experience per group

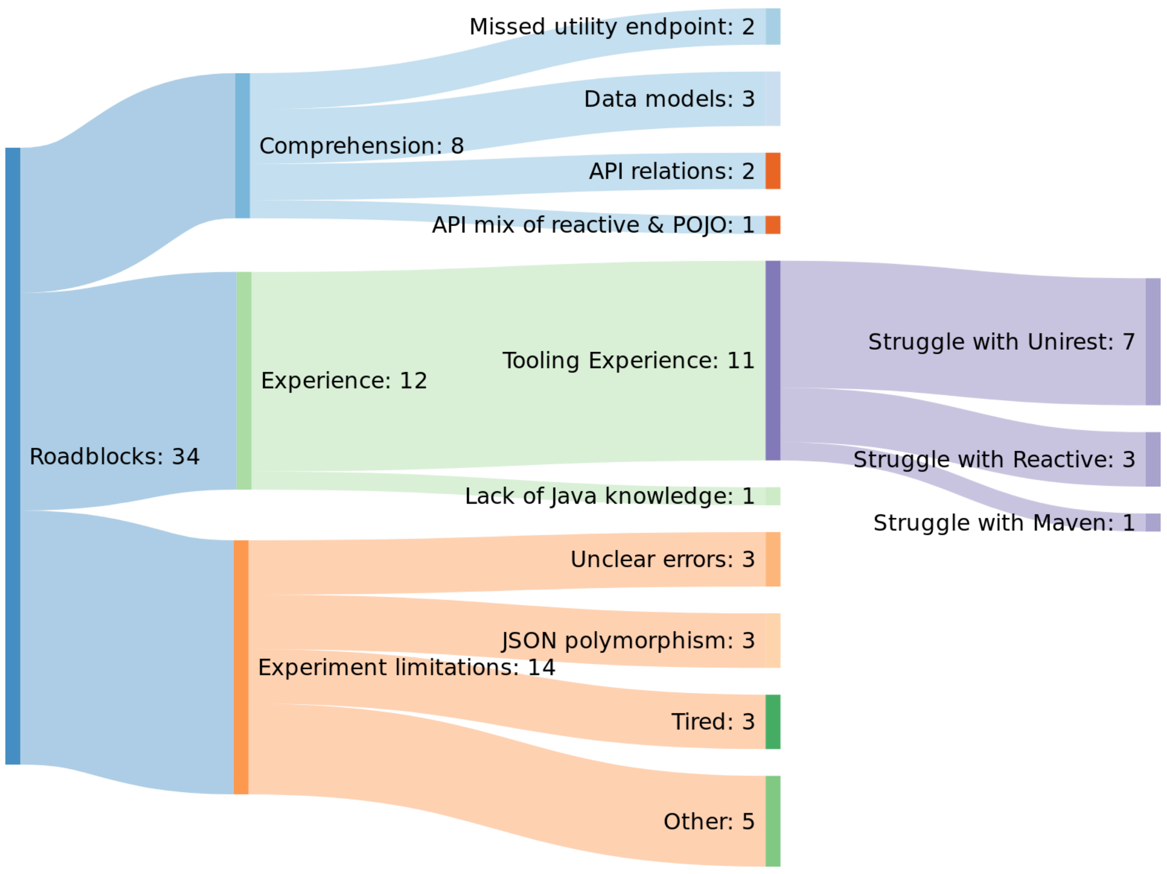


Figure 9.1: Roadblocks experienced by participants from group one who were unable to finish

9.5. ROADBLOCK DIFFERENCES

We will now explore the second possible explanations for the group discrepancy: the presence of roadblocks. It is possible that the participants of group one experienced some form of difficulties while participating in the experiment that the participants of group two did not (or only to a lesser extent) have to endure. In order to get an idea of what these difficulties might be we contacted the participants of group one that were unable to finish and inquired about the difficulties they encountered during the experiment: the parts they got stuck on or the parts that took them a lot of time. We categorized their responses through card sorting (see also Section 3.2), resulting in Figure 9.1. From this figure we can distinguish that the roadblocks can be separated into three large groups. The three groups are, in descending order based on the number of responses, as follows: `Experiment limitations`, `Tooling experience` and `Comprehension`. We will discuss each group separately in Sections 9.5.1, 9.5.2 and 9.5.3 respectively.

9.5.1. EXPERIMENT LIMITATIONS

The first category of roadblocks seems to be due to the limitations of the experiment. These roadblocks are mainly due to limitations from our side, where we either made a mistake or did not take the proper precautions due to time limitations. We have placed these in three different subcategories, and `Other` for the answers that were only mentioned once or twice:

Unclear errors Participants have indicated that the error messages from the API were not always clear. They were considered not to be descriptive enough or lacking correlation to what actually was going wrong. We tried to make all errors thrown by the server as descriptive as possible, but it seems that the (de)serialization errors thrown by Jackson (the (de)serialization library used by both the SDK and NSDK) proved to be difficult to handle. Server errors caught in the reactive streams of the SDK were also considered to be difficult, because debugging reactive code can be quite tricky. These unclear errors should affect both groups equally however, with a potential exception for server errors in the reactive code, which should affect group one more than group two. This is because group two already figured out the correct solution in the NSDK, so they only had to translate their solution to reactive code, drastically reducing the potential for errors coming from the server which would then need to be debugged in the reactive streams. We are undecided whether or not the insufficiently descriptive error messages should be considered as a limitation of the experiment or not, seeing as though the descriptiveness of API error messages from non-experiment APIs can differ from API to API meaning that it could be considered as realistic as well.

JSON polymorphism The documentation of a couple of the endpoints stated that it returned objects of a specific (abstract) class: `PCPart`. In reality this endpoint returned instances of the subclasses of `PCPart`: `PCCase`, `CPU`, `GPU`, `PSU`. `PCPart` contained some fields that were relevant to all subclasses, and each subclass defined some extra fields that were only relevant to that class. Because the concrete instance type that would be returned was dependent upon the request, the endpoint could only specify that it would return one of the instances of `PCPart`. In order to make this possible, one of the fields of `PCPart` was called `category` and specified the concrete type of the instance, which would allow clients to simply look at that field and deserialize the JSON object as an object of the specified type. We used JSON Jackson's subtype annotations to convey this information, but we did not realise that this would send a second field `category` in the JSON objects, resulting in each object having two JSON fields with the same name, which is not allowed according to JSON specification. If participants used the same subtype annotations in their solutions (as explained in the documentation for the people who were unfamiliar API consumption) then everything went as expected. But if people tried to parse the JSON manually, it would fail because of the responses from the server which were not valid JSON would confuse those participants. This is an error from our side and unfortunately impacted some of the participants. But this error should theoretically affect the participants from group two more than the participants from group one. The participants from group one are likely to adapt their SDK solution to the NSDK variant to prevent extra effort. The easiest way of doing so is recreating the data models used by the SDK, in which case they should not encounter the incorrect JSON problems. The participants of group two however did not yet have a solution to adapt, so whether they chose data model deserialization through Jackson over manual JSON parsing is a developer preference, making it more likely for them to pick the manual approach compared to the participants from group one.

Tiredness The experiment was rather long, so a lot of participants would not be able to participate during the day due to work. As a result, the experiment was conducted in the evening to maximize the number of participants that would be able to participate. This unfortunately caused people to be tired during the experiment, especially as the evening progressed. This roadblock should theoretically affect people of both groups equally.

Other This category contains all roadblocks that were minimally or not shared by others and therefore do not disproportionally affect the members of one group compared to the other.

9.5.2. TOOLING EXPERIENCE

The second category of roadblocks is tooling experience. This means that the inexperience of certain participants with some of the tools used in the experiment slowed them down in completing the assignments. As explained in Section 3.5 we predicted that this might become an issue, but as we concluded in Section 9.4.5 the tooling experience division was quite even overall, and should therefore affect both groups equally. The exception being reactive experience, due to the unclear errors as stated in section 9.5.1. It must be noted however that participants had access to instruction on how to work with reactive and how to minimize the use of reactive in their solution if they were not comfortable with reactive code. However, the possibility of them not finding or reading this explanation remains.

9.5.3. COMPREHENSION

The last category of roadblocks is comprehension. It means that something was unclear or confusing for the participant. Most roadblocks (`Data models & API relations`) here are due to participants not fully understanding how the data models or the API endpoints relate to each other. Furthermore some of the participants missed the fact that there was an endpoint that heavily simplified the work that needed to be done. Without this utility endpoint the task could still be completed, but without knowing the visitor programming pattern[1] it would take a considerable amount of effort to do so. The task at hand (and the endpoint to achieve it) should have been better explained or should have been made more obvious in some way. However these roadblock reasons should affect group two more than group one, since group one has the IDE to their advantage while finding a solution. The IDE allows the participants to explore the data models (that are given by the SDK) more easily, and gives a better idea of what endpoints can do and how they relate to each other through code completion. But it was group one and not group two that ended up having more issues finishing the assignments, so this contradicts the group discrepancy that has taken place.

9.5.4. ROADBLOCK DIFFERENCES: CONCLUSION

From the responses we have gotten from the participants from group one that were unable to finish the experiment we have been able to deduct all kinds of factors that could have played a role in creating the group discrepancy. It seems that most of these roadblocks should have affected both groups equally, with the exception of the following three:

1. The reactive code of the SDK solution might have slowed participants down due to the difficulty of debugging reactive code
2. The participants from group two might be more likely to run into the invalid JSON problem than the participants from group one
3. The IDE might have helped participants of group one to get a better understanding of the APIs and the data models, therefore making it more likely for group one to have discovered certain utility endpoints

The first roadblock is likely to have affected the participants of group one more than those of group two and is therefore in line with the group discrepancy that has occurred. The second and third roadblock however are the opposite, they would likely have affected participants of group two more than those of group one, and therefore contradict the observed group discrepancy.

The impact that the roadblocks have had is difficult to assess. The fact that we have discovered roadblocks that contradict the explanation makes the assessment even more difficult. It could be that without these contradictory roadblocks that the group discrepancy might be even larger, but it could also be that the impact of the contradictory roadblocks have been minimal. Unfortunately the best we can do is speculate.

9.6. COINCIDENCE

The final possibility and third possible explanation for the group discrepancy is that it is merely coincidental. The fact that we were unable to discover large discrepancies in the experience of the participants in terms of REST APIs and tooling, and the fact that some of the possible roadblocks contradict the observed group discrepancy make this explanation more likely. However due to the sheer size of the discrepancy combined with the fact that we can only speculate on the impact of the other three experience differences (the innate differences between participants, the questionnaire asking for development experience in general instead of Java experience and the difference in day-to-day activities of the participants) makes us believe that the observed group discrepancy is not coincidental.

9.7. CONSEQUENCES

The group discrepancy forced us to change how we approach the analysis of the collected data, rendering us unable to perform any comparative analysis between the two methods for every aspect with the exception of (de)serialization. While group two did gather data on both methods, due to the lack of data from group one, the data might not be representative, as the order of the experiment might have had an influence on the data.

[1] `https://sourcemaking.com/design_patterns/visitor`

Since we were unable to determine with certainty a cause for the discrepancy after exploring the three possible differences in participants that might have caused it we cannot assess whether the conclusions drawn on the data of those aspects is still valid after successful replication of the experiment without group discrepancy, and therefore the observed group discrepancy is a limitation of the research.

10
Conclusions & future work

During this thesis we have gone over two fundamental differences between the SDK and the NSDK; (de)serialization and evolution. For both of these differences we have established some research questions to gain insight into the viability of using autogenerated SDKs from the perspective of the consumer. In Section 10.1 we aim to provide knowledge to the reader about the existence of this viability and in which circumstances (if any) the autogenerated SDKs is a viable option. Furthermore, we will go over the future of autogenerated SDKs and some of the questions that are unanswered by this research in Section 10.2, leaving them open for future researchers.

10.1. Conclusions

In this section we will go over both of the fundamental differences between the autogenerated SDK and the NSDK that we have researched into this dissertation and provide some our final remarks on them.

10.1.1. (De)serialization

The first fundamental difference between autogenerated SDKs and NSDKs is in the (de)serialization of requests and responses between the consumer and the web API. This process is mostly a one-time effort during the implementation phase of the application built by the consumer. We posed ourselves two questions in that regard:

1. Does one of the two methods take more effort compared to the other (with regards to (de)serialization)?
2. Does one of the two methods result in higher quality/cleaner code compared to the other?

We discovered that the autogenerated SDK definitely has an advantage over the NSDK there. Participants found the SDK to take significantly less effort compared to the NSDK due to providing data models out of the box and the SDK API making it easier for them to understand how it should be used. They also found the SDK to result in higher quality code due to all the data models not being part of the SDK, resulting in the SDK simply having less code to maintain.

10.1.2. Evolution

The second fundamental difference between the autogenerated SDK and the NSDK is the ability to adapt to breaking changes, which we called evolution throughout this dissertation. This adaptability is a continuous effort throughout the lifetime of the application as long as both the application and the web API it uses are being maintained. We expect this to take more effort overall compared to the initial implementation. This expectation is confirmed by some of the developers interviewed by Espinha et al. [10], who, when asked how the effort of initial integration compares to maintaining this integration over time, stated that "it takes them far more time maintaining the integration than it does integrating with a web API in the beginning" or "at least

50% into each task, with possibly even more time going into maintaining the integration". This expectation is why we placed more emphasis on this aspect compared to the (de)serialization, resulting in four questions that we posed ourselves:

1. Does one of the two methods take more effort compared to the other (with regards to evolution)?
2. Does one of the two methods result in better testable code?
3. What types of breaking changes are considered to be difficult?
4. Which resources are considered to be useful during solving breaking changes?

Here we discovered that the effort required to accommodate breaking changes is slightly in favour of the SDK. The true benefit of the SDK however is in the testability of the code, due to not having to test the networking code and the (de)serialization process, given that the generator can be trusted. Depending on who is responsible for the generator and the SDK in general (internal vs third party) this assumption might be dangerous to make.

In terms of the difficulty of the breaking changes the replacement or removal of an endpoint was widely considered to be the most difficult. The reason for its difficulty however was due to unclarity in the documentation, so it could very well be that this difficulty might only exist in this experiment. Lastly all resources were considered to be useful, with the changelog which describes all changes being the most useful.

10.1.3. PARTICIPANT INTERVIEWS

We also talked with some of the participants about the topic to gain more in-depth views on their opinions which varied from participant to participant. It seems that most participants are open to the idea of the autogenerated SDK and see the potential advantages, but they can be hesitant about the practicality of it. The lack of customization of the SDK requires common default settings to make its existence useful. On the other side however the SDK makes future changes in the standard settings and tools used in web API communication very easily changeable, applying the changes across the board by simply adjusting the generator accordingly.

10.1.4. VIABILITY OF THE AUTOGENERATED SDK

All in all, we think that the autogenerated SDK is definitely a viable alternative compared to the NSDK in certain cases. While the advantages in (de)serialization are definitely a welcome bonus, we do not think that this is enough to make the autogenerated SDK a good choice. As some of the interviewed participants also noted we agree that common default settings are a necessity to make the autogenerated SDK useful. If these are not present then configuration will often need to be done on top of maintaining an SDK generator, which likely outweighs the benefits provided by the autogenerated SDK.

Furthermore due to the issues regarding trust which we discussed in Section 5.3.3 we believe that autogenerated SDKs are not suitable for third-party use. We would rather see these autogenerated SDKs only being used internally where both the consumer and the producer have influence in its development.

10.2. FUTURE WORK

While we explored some of the aspects of the viability of the autogenerated SDK during this thesis there are still questions left unanswered.

We did not consider the security of web APIs during this thesis and how the existance of many different standards might affect the practicality of the autogenerated SDKs. Furthermore the group discrepancy that occurred during the research has affected the reliability of the results with regards to API evolution. Replication of the experiment with more participants could restore this reliability and potentially further improve the findings of this thesis.

BIBLIOGRAPHY

[1] S. Weber, *The Success of Open Source* (Harvard University Press, 2004).

[2] C. S. Collberg and C. Thomborson, *Watermarking, tamper-proofing, and obfuscation - tools for software protection,* IEEE Transactions on Software Engineering **28**, 735 (2002).

[3] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, *Service-oriented computing: State of the art and research challenges,* Computer **40**, 38 (2007).

[4] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, *Developing web services choreography standards—the case of rest vs. soap,* Decision Support Systems **40**, 9 (2005), web services and process management.

[5] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. thesis (2000), aAI9980887.

[6] E. Wittern, A. Ying, Y. Zheng, J. A. Laredo, J. Dolby, C. C. Young, and A. A. Slominski, *Opportunities in software engineering research for web api consumption,* in *Proceedings of the 1st International Workshop on API Usage and Evolution*, WAPI '17 (IEEE Press, Piscataway, NJ, USA, 2017) pp. 7–10.

[7] S. Newman, *Building microservices: designing fine-grained systems* (O'Reilly Media, Inc., 2015).

[8] baeldung, *Introduction to java serialization,* (2019).

[9] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč, *Json: Data model, query languages and schema specification,* in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17 (ACM, New York, NY, USA, 2017) pp. 123–135.

[10] T. Espinha, A. Zaidman, and H. 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)* (2014) pp. 84–93.

[11] M. Maleshkova, C. Pedrinaci, and J. Domingue, *Investigating web apis on the world wide web,* in *2010 Eighth IEEE European Conference on Web Services* (2010) pp. 107–114.

[12] R. Plöesch, S. Schuerz, and C. Koerner, *On the validity of the it-cisq quality model for automatic measurement of maintainability,* in *2015 IEEE 39th Annual Computer Software and Applications Conference*, Vol. 2 (2015) pp. 326–334.

[13] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, *Software quality models: Purposes, usage scenarios and requirements,* in *2009 ICSE Workshop on Software Quality* (2009) pp. 9–14.

[14] S. Wagner, K. Lochmann, S. Winter, A. Goeb, and M. Klaes, *Quality models in practice: A preliminary analysis,* in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09 (IEEE Computer Society, Washington, DC, USA, 2009) pp. 464–467.

[15] T. Dogša and D. Batič, *The effectiveness of test-driven development: an industrial case study,* Software Quality Journal **19**, 643 (2011).

[16] J. R. Wood and L. E. Wood, *Card sorting: Current practices and beyond,* J. Usability Studies **4**, 1 (2008).

[17] G. Charness, U. Gneezy, and M. A. Kuhn, *Experimental methods: Between-subject and within-subject design,* Journal of Economic Behavior & Organization **81**, 1 (2012).

[18] G. Keren and C. Lewis, *A Handbook for Data Analysis in the Behaviorial Sciences*, 1st ed. (New York: Psychology Press, 1993).

[19] J. Li, Y. Xiong, X. Liu, and L. Zhang, *How does web service api evolution affect clients?* in *2013 IEEE 20th International Conference on Web Services* (2013) pp. 300–307.

[20] V. Nguyen, *Improved size and effort estimation models for software maintenance,* in *2010 IEEE International Conference on Software Maintenance* (2010) pp. 1–2.

[21] S. S. Yau and J. S. Collofello, *Some stability measures for software maintenance,* IEEE Transactions on Software Engineering **SE-6**, 545 (1980).

[22] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering* (Springer, 2012).

A
QUESTIONNAIRES

A.1. PREFACE

The questions marked with an asterisk (*) symbol are questions that slightly change depending on the answer to the previous question. For example, if the previous question was:

> Do you think that the resulting code to complete the serialization task is cleaner/of higher quality using one of the methods (SDK vs NSDK) compared to the other?

And the participant answered `The solution using the SDK is significantly cleaner`, then the question marked with an asterisk will become:

> Please briefly describe why you think that the code of the solution using the SDK is of higher quality compared to the solution using the NSDK.

Whereas if the participant answered `There is no difference in cleanliness between the SDK and the NSDK`, then the question marked with an asterisk will become:

> Please briefly describe why you think that the code of the solution using either method (SDK or NSDK) are of equal quality.

A.2. PREVIOUS EXPERIENCE QUESTIONNAIRE

1. What is your (nick)name? (This question was purely asked to be able to link this questionnaire to results of the experiment)
2. How many years of experience do you have as a developer? Anything where you consistently write and work with code for a significant amount of time during the week (e.g. university work, or a couple of hours every evening in your free time) counts as experience.

   o I have less than one year of development experience.

   o I have between one and three years of development experience.

   o I have between three and five years of development experience.

   o I have between five and ten years of development experience.

   o I have more than ten years of development experience.
3. How many months have you worked for Picnic?

   o I do not work for Picnic

   o I have worked for Picnic for less than three months

   o I have worked for Picnic for three to six months o I have worked for Picnic for six to twelve months

   o I have worked for Picnic for more than twelve months

4. How many different companies have you worked for (excluding Picnic) as a developer? Something counts as a company if you have worked for that instance (as a developer) for more than roughly 20 hours per week for at least a couple of months. This includes universities and having your own business.

   o I do not have any work experience at a (different) company yet

   o I have worked for one different company before

   o I have worked for two or three different companies before

   o I have worked for four or five different companies before

   o I have worked for more than five different companies before

5. How often do you work with/consume REST APIs?

   o I have never used a REST API before

   o I work with REST APIs once or multiple times every few months

   o I work with REST APIs once or multiple times every few weeks

   o I work with REST APIs once or multiple times per week

   o I work with REST APIs on a daily basis

6. How much experience do you have with using NSDKs?

   o I have never used a NSDK before

   o I use a NSDK once or multiple times every few months

   o I use a NSDK once or multiple times every few weeks

   o I use a NSDK once or multiple times per week

   o I use a NSDK on a daily basis

7. How much experience do you have with using SDKs?

   o I have never used a SDK before

   o I use a SDK once or multiple times every few months

   o I use a SDK once or multiple times every few weeks

   o I use a SDK once or multiple times per week

   o I use a SDK on a daily basis

8. How much experience do you have with reactive frameworks (e.g RxJava, Spring Reactor or something similar)?

   o I have never used such frameworks before

   o I use such frameworks once or multiple times every few months

   o I use such frameworks once or multiple times every few weeks

   o I use such frameworks once or multiple times per week

   o I use such frameworks on a daily basis

9. How much experience do you have with using unirest? (unirest is a http client library)

   o I have never used unirest before

   o I use unirest once or multiple times every few months

   o I use unirest once or multiple times every few weeks o I use unirest once or multiple times per week

o I use unirest on a daily basis

A.3. (DE)SERIALIZATION QUESTIONNAIRE

1. Do you think that completing the serialization task took more work using one of the methods (SDK or NSDK) compared to the other?

    o The SDK took significantly more work.

    o The SDK took slightly more work.

    o The SDK and NSDK took an equal amount of work.

    o The NSDK took slightly more work.

    o The NSDK took significantly more work.

2. Please briefly describe why you think that the SDK requires more effort to complete the serialization task compared to the NSDK. *

3. Do you think that the resulting code to complete the serialization task is cleaner/of higher quality using one of the methods (SDK vs NSDK) compared to the other?

    o The solution using the SDK is significantly cleaner.

    o The solution using the SDK is slightly cleaner.

    o There is no difference in cleanliness between the SDK and the NSDK.

    o The solution using the NSDK is slightly cleaner.

    o The solution using the NSDK is significantly cleaner.

4. Please briefly describe why you think that the code of the solution using the SDK is of higher quality compared to the solution using the NSDK. *

5. Do you have a preference for one of the methods (SDK or NSDK) regarding the serialization & deserialization of primitives (task A1 and A2)?

    o Significant preference for the SDK.

    o Slight preference for the SDK.

    o No preference for either the SDK or the NSDK.

    o Slight preference for the NSDK.

    o Significant preference for the NSDK.

6. Do you have a preference for one of the methods (SDK or NSDK) regarding the serialization & deserialization of nested objects (task B)?

    o Significant preference for the SDK.

    o Slight preference for the SDK.

    o No preference for either the SDK or the NSDK.

    o Slight preference for the NSDK.

    o Significant preference for the NSDK.

7. Please briefly describe the deciding factor(s) behind your preference (or lack thereof) with regards to the (de)serialization.

A.4. FIRST EVOLUTION QUESTIONNAIRE

1. What changes did you find difficult to incorporate in the solution? Please mark all reasons that apply.
   - ☐ Prepend all endpoints with /api/v2 (Changelog #1)
   - ☐ The origin and destination fields in Task objects have been renamed to start and finish respectively (Changelog #7)
   - ☐ Route parameter changed to a query parameter(Changelog #4)
   - ☐ Get a list of station objects instead of a list of station IDs (Changelog #2)
   - ☐ Changed POST to PUT (Changelog #5)
   - ☐ Header changed from x-id-key to x-idKey (Changelog #3)
   - ☐ Route objects have to have an ID field created through a new endpoint (Changelog #6)
   - ☐ Other
2. (Only shown when one of the answers to the previous question was "Other") Please briefly describe the other reason(s).
3. Please briefly describe the reasons why you found those changes in particular difficult.
4. Which resources did you find useful while adjusting the code to the new API version? Please mark resources that you found useful and drag them in order where the most used resource is the element at the top of the list, and the least used element is the element at the bottom of the list.
   - ☐ Specification
   - ☐ Error messages at runtime
   - ☐ Changelog
   - ☐ Compiler

A.5. SECOND EVOLUTION QUESTIONNAIRE

1. Do you think that completing the evolution task took more work using one of the methods (SDK or NSDK) compared to the other?
   - o The SDK took significantly more work.
   - o The SDK took slightly more work.
   - o The SDK and NSDK took an equal amount of work.
   - o The NSDK took slightly more work.
   - o The NSDK took significantly more work.
2. Please briefly describe why you think that the NSDK requires more effort to complete the evolution task compared to the SDK. *
3. Do you think that the resulting code using one of the methods (SDK vs NSDK) will be easier to test compared to the other?
   - o The solution using the SDK will be significantly easier to test.
   - o The solution using the SDK will be slightly easier to test.
   - o There will be no difference in ease of testing between the SDK and the NSDK.
   - o The solution using the NSDK will be slightly easier to test.
   - o The solution using the NSDK will be significantly easier to test.

4. Please briefly describe why you think that the solution using the SDK will be easier to test compared to the NSDK. *

A.6. DOMAIN TRANSLATION QUESTIONNAIRE (OPTIONAL)

1. Do you have a preference for one of the methods (SDK or NSDK) regarding the transformation of domain models from one domain to another? (the previous assignment)

   o Significant preference for the SDK.

   o Slight preference for the SDK.

   o No preference for either the SDK or the NSDK.

   o Slight preference for the NSDK.

   o Significant preference for the NSDK.

2. Did the previous assignment change your opinion on which method (SDK or NSDK) took more work to complete the tasks at hand compared to the other method?

   o Yes

   o No

3. (Only shown when the answer to the second question was "Yes") What is your new opinion?

   o The SDK took significantly more work.

   o The SDK took slightly more work.

   o The SDK and NSDK took an equal amount of work.

   o The NSDK took slightly more work.

   o The NSDK took significantly more work.

4. (Only shown when the answer to the second question was "Yes") Please briefly describe the deciding factor(s) that changed your opinion on which method takes more work.

5. Did the previous assignment change your opinion on which method (SDK or NSDK) is cleaner/of higher quality compared to the other method?

   o Yes

   o No

6. (Only shown when the answer to the fifth question was "Yes") What is your new opinion?

   o The solution using the SDK is significantly cleaner.

   o The solution using the SDK is slightly cleaner.

   o There is no difference in cleanliness between the SDK and the NSDK.

   o The solution using the NSDK is slightly cleaner.

   o The solution using the NSDK is significantly cleaner.

7. (Only shown when the answer to the fifth question was "Yes") Please briefly describe the deciding factor(s) that changed your opinion on which method is cleaner/of higher quality.

A.7. GENERAL QUESTIONS REGARDING SDK VS NSDK

1. I can see myself using an SDK in a production environment (stable application that has to run 24/7)

   o Heavily disagree

   o Slightly disagree

   o Neither agree nor disagree

   o Slightly agree

   o Heavily agree

2. I can see myself using an NSDK in a production environment (stable application that has to run 24/7)

   o Heavily disagree

   o Slightly disagree

   o Neither agree nor disagree

   o Slightly agree

   o Heavily agree

3. If given a choice between using a SDK and a NSDK while performing a task, which one would you choose?

   o I would choose the SDK

   o I would choose the NSDK

   o My choice would depend on the task at hand

   o I have no preference at all

4. Please briefly describe the main reason(s) why you prefer SDKs over NSDKs. *

5. If you have had any experience with manually written SDKs, please describe any differences you have noticed between manually written SDKs and the generated SDKs used in these assignments.