MIGRATING A LARGE JAVASCRIPT WEB UI TO TYPESCRIPT TO IMPROVE DEVELOPER EXPERIENCE

Lappeenranta–Lahti University of Technology LUT

Master's Programme in Software Engineering and Digital Transformation,

Master's thesis

2022

Maija Heiskanen

Examiners: Kari Smolander, Professor

Joonas Salminen, M.Sc. (Tech)

ABSTRACT

Lappeenranta–Lahti University of Technology LUT

LUT School of Engineering Science

Software Engineering

Maija Heiskanen

Migrating a large JavaScript web UI to TypeScript to improve Developer Experience

Master's thesis

2022

78 pages, 21 figures, 4 tables

Examiners: Professor Kari Smolander and Joonas Salminen, M.Sc. (Tech)

Keywords: Developer Experience, migration, type system, TypeScript

Popularity of TypeScript has been rising during the past six years. TypeScript is a programming language that extends JavaScript by adding static typing. Statically typed and dynamically typed programming languages have been compared in research and statically typed programming languages have many advantages over dynamically typed programming languages. Statically typed code bases have better documentation, tooling, and maintainability, and less bugs and greater development speed in large projects. These are also factors that improve Developer Experience. According to research, happy developers are more productive and produce higher quality than unhappy developers. A case company wanted to migrate the JavaScript code of the user interfaces of their SaaS product to TypeScript to improve Developer Experience. The web UI uses Inferno, and the mobile application uses React Native. TypeScript migration can be initialized by installing required type packages, by configuring TypeScript settings, by integrating TypeScript compiler within the build process, by configuring linter for TypeScript and by configuring development tools and environments. In addition, a generator can be used to create API types and automation tools can be searched to speed up the migration. The migration improved Developer Experience and the structure of the code, helped to keep the developers and the product up to date, and possibly decreased the number of bugs and optimized the code. The migration was easier to initialize for React Native than for Inferno, which has a smaller ecosystem. The migration improved Developer Experience by embedding documentation in the code, by improving the auto-complete feature of the IDE and forcing the code to follow good practices and have a consistent structure.

TIIVISTELMÄ

Lappeenrannan–Lahden teknillinen yliopisto LUT

LUT Teknis-luonnontieteellinen

Tietotekniikka

Maija Heiskanen

Suuren JavaScript-pohjaisen web-käyttöliittymän TypeScript-migraation toteuttaminen kehittäjäkokemuksen parantamiseksi

Tietotekniikan diplomityö

2022

78 sivua, 21 kuvaa, 4 taulukkoa

Tarkastajat: Professori Kari Smolander ja diplomi-insinööri Joonas Salminen

Avainsanat: kehittäjäkokemus, migraatio, tyyppijärjestelmä, TypeScript

TypeScriptin suosio on kasvanut viimeisen kuuden vuoden aikana. TypeScript on JavaScriptiä laajentava kieli, joka lisää JavaScriptiin staattisen tyypityksen. Tutkimuksissa on verrattu staattisesti tyypitettyjä ja dynaamisesti tyypitettyjä ohjelmointikieliä ja staattisesti tyypitetyissä on monia etuja verrattuna dynaamisesti tyypitettyihin. Staattisesti tyypitetyissä koodikannoissa on parempi dokumentaatio, työkalut ja ylläpidettävyys, vähemmän bugeja sekä suurempi kehitysnopeus isoissa projekteissa. Nämä ovat myös tekijöitä, jotka parantavat kehittäjäkokemusta. Tutkimusten mukaan onnelliset ohjelmistokehittäjät ovat tuottavampia ja tuottavat parempaa laatua kuin tyytymättömät ohjelmistokehittäjät. Case-yritys halusi vaihtaa SaaS-tuotteensa käyttöliittymien JavaScript-koodin TypeScript-koodiksi parantaakseen kehittäjäkokemusta. Web-käyttöliittymä käyttää Infernoa ja mobiilisovellus React Nativea. TypeScript-migraatio voidaan alustaa asentamalla tarvittavat tyyppipaketit, konfiguroimalla TypeScript-asetukset, integroimalla TypeScript-kääntäjän osaksi ohjelman rakennusvaihetta, konfiguroimalla lintterin TypeScriptille ja konfiguroimalla kehitystyökalut ja ympäristöt. Lisäksi voidaan luoda generaattori rajapintatyyppien luomiseksi, ja etsiä automaatiotyökaluja migraation nopeuttamiseksi. Migraatio paransi kehittäjäkokemusta ja koodin rakennetta, auttoi pitämään kehittäjät ja tuotteen ajan tasalla, ja mahdollisesti vähensi bugeja ja optimoi koodia. Migraatio oli helpompi toteuttaa React Nativelle, kuin Infernolle, jolla on pienempi ekosysteemi. Migraatio paransi kehittäjäkokemusta sisällyttämällä dokumentaatiota koodiin, parantamalla IDE:n automaattista tekstinsyöttöä, sekä pakottamalla koodin noudattamaan hyviä käytänteitä ja yhtenäisiä rakenteita.

ACKNOWLEDGEMENTS

There are many people I would like to thank.

I would like to thank my supervisor Kari Smolander and instructor Sampo Kivistö for their guidance. I would also like to thank all my colleagues.

I would like to thank my mom and dad for supporting me whenever I have needed it.

I would like to thank my friends for our years in the university. You made it awesome.

I would like to thank Niku for supporting me through this. You are the best.

ABBREVIATIONS

API	Application Programming Interface
CI/CD	Continuous Integration and Continuous Deployment
CLI	Command-line Interface
DSRM	Design Science Research Method
DX	Developer Experience
ESM	Ecmascript Modules
IDE	Integrated Development Environment
IS	Information System
JIT	Just In Time Compiler
JSX	JavaScript XML
LSP	Language Service Protocol
OOP	Object Oriented Programming
PSA	Professional services automation
PX	Programmer eXperience
TSC	TypeScript compiler
UX	User Experience

Table of contents

1. Introduction

It is known that developer's mood affects their performance (Graziotin et al., 2014a, 2017a; Lesiuk, 2005; Müller and Fritz, 2015). While some amount of negative emotions can be good for developer's performance, too much negative emotions and frustration tends to decrease developer's performance (Graziotin et al., 2017a) and on the other hand, positive mood tends to increase developer's performance (Graziotin et al., 2014a). One cost-effective way to promote positive mood is to decrease unhappiness by limiting factors that cause unhappiness (Graziotin et al., 2017a). Müller and Fritz (2015) conducted a study to find out which factors increase developer's mood and which factors decrease it. Their study found out that the most common reasons for decreased mood were "difficulty in understanding how parts of the code or API [(Application Programming Interface)] work" and "difficulty in locating relevant code". Most common reasons for increased mood were "locate relevant code" and "(better) understand parts of the code". These are struggles that occur especially in dynamically typed programming languages, such as JavaScript.

There have been discussions and attempts to add static typing in JavaScript for decades. Types were originally planned to be added in Ecmascript 4, but the version 4 was never released (Ecma International, 2006). After that, there has been multiple solutions for adding static typing in JavaScript by other actors: Google's Closure Compiler, Facebook's Flow and Microsoft's TypeScript, which is by far the most popular solution. In addition to them, some IDEs (Integrated Development Environment) can for example parse JSDoc strings for type checking. JavaScript is not the only dynamically typed programming language to which static typing has tried to be added. For example, Python version 3.5 added support for type hints (Python, 2022). Type hints let the developer use type syntax even though Python does not enforce types. Type hints can be used by third party type checkers, linters, and IDEs. It is clear, that the developer community needs more type features in dynamically typed programming languages.

TypeScript is a programming language that adds static typing in JavaScript (Microsoft, 2022a). TypeScript can be used anywhere JavaScript can be used because TypeScript is compiled to JavaScript (Microsoft, 2022a). Because it is compiled to JavaScript, it can run in any environment that JavaScript can run, for example browsers, Node, Deno, and command line terminals. TypeScript is said to embed documentation in the code in the form of typing, catch bugs in development and to provide better tooling for the developer in comparison to JavaScript ("TypeScript", 2022). During the past five years, TypeScript has been one of the most loved programming languages and technologies according to Stack Overflow surveys (Stack Overflow, 2022, 2021, 2020, 2019, 2018, 2017). In 2020 TypeScript was the second most loved language or technology, only behind Rust, which is a statically and strongly typed programming language (Stack Overflow, 2020). In 2022 73.46% of developers that used TypeScript loved the language, while only 61.46% of developers who used JavaScript loved it (Stack Overflow, 2022). TypeScript started to grow its popularity after the release of version 2.0 in 2016 ("Releases · microsoft/TypeScript", 2022). In 2022 40.08% of professional developers had used TypeScript in their work and TypeScript was more wanted technology than JavaScript by 17.03% of developers wanting to use TypeScript while only 12.98% of them wanted to use JavaScript (Stack Overflow, 2022). TypeScript is the third most wanted technology among all developers (Stack Overflow, 2022).

To cope with the challenges that dynamically typed JavaScript brings to a large software project, the case company wants to try out migrating their large JavaScript web user interface to TypeScript. While there are tens or even hundreds of blog posts and articles guiding how to migrate from JavaScript to TypeScript, they all cover only small projects and stay on too shallow level for migrating a large web user interface. TypeScript's own migration guide offers deeper and more technical guidance but is still lacking important parts of the process, such as handling JSX (JavaScript XML) syntax for other than the most popular frameworks and autogenerating types from API schemas. Based on the situation of the case company, the rising popularity of TypeScript and the lack of comprehensive migration guides, there seems to be a need for a practical migration guide for large web user interfaces.

1.1. Goals and delimitations

The goal of this thesis is to create a plan to migrate a large JavaScript web user interface to TypeScript, execute the migration plan to the level where the migration can be started, and evaluate how the migration affects the software, especially the Developer Experience. This thesis does not cover the whole migration process, but the initialization of the migration so that the project files could be converted to TypeScript and converting a small portion of the project to TypeScript, for example one feature, as a proof-of-concept. Alternative outcome of this thesis is to find that the migration cannot be executed in the context of the case company and reason why the migration cannot be done. Also, the profitability and needed resources are estimated. The research questions of this thesis are:

Q1: How to initialize a TypeScript migration for a large web user interface?

Q2: What are the benefits and drawbacks of executing a TypeScript migration?

Q3: How does a TypeScript migration affect Developer Experience?

1.2. Structure of the thesis

In chapter 1 the problem and its background are introduced. In chapter 2 the previous studies related to Developer Experience and type systems are presented. In chapter 3 the research method is introduced. In chapter 4 the objectives of the TypeScript migration are presented. In chapter 5 the TypeScript migration model is discussed. In chapter 6 the TypeScript migration model is demonstrated in the case company context. In chapter 7 the TypeScript migration model is evaluated. Finally, chapter 8 contains the conclusions of this study.

2. Developer experience and type systems

In this chapter, the concepts of Developer Experience and different type systems are introduced. Then the effects of different type systems to Developer Experience are discussed and finally, TypeScript is introduced. Different researchers use a different name for the person participating in the coding tasks. In this thesis the person is always referred to as a "developer" for consistency.

2.1. Developer Experience

In this chapter the link between the happiness of developers and their productivity and performance are introduced. Then, factors which can affect the affective states of developers are discussed. Finally, two models for modeling the affective states of developers are introduced.

2.1.1. Causes and effects of developer's affective state

Graziotin et al. (2018, 2017b, 2017a, 2015a, 2015b, 2015c, 2015d, 2014b, 2014a, 2013) have been studying effects of developer's emotions and moods for several years. They conducted a study (Graziotin et al., 2013) to research if a developer's affective state correlates with their productivity. The affective state of a person consists of their emotions, moods, and feelings. Affect is the person's experience of an emotion, mood or feeling (Abrams and Hogg, 1999). The study showed that there was a positive correlation between positive affective state and self-assessed productivity. Valence, which means the attractiveness of an event, and dominance, which is the sense of having appropriate skills for a task, were found to have positive correlation with productivity. Correlation between arousal, which means the sense of being awake and being able to react to stimuli, and productivity could not be found. The results implicated that happy developers are more productive than unhappy developers. Lesiuk (2005) had similar results when conducting a study on the effect of music on the productivity of developers. When developers could listen to music that they liked while

working, they were reported to have higher positive affective state and to be more productive. When they were forbidden from listening to music while working, their positive affective state decreased and so did their productivity. The results showed that developers with a high positive affective state were more productive compared to when they had a lower positive affective state. Not only did they finish their tasks faster, but they did so with higher quality. Graziotin et al. (2014b) also combined several studies studying the effects of affective states of developers and concluded that it seems like there is a positive correlation between positive affective state and productivity. Positive affective state positively affects the performance of developers in analytic problem solving (Graziotin et al., 2014a). Girardi et al. (2020) also found a correlation between the emotions and performance of developers. When developers had negative emotions, their performance decreased and when developers had positive emotions, their performance increased. Graziotin et al. (2015b) created the term "psychoempirical software engineering" and guidelines for conducting studies on affective states in software engineering. Later that year Graziotin et al. (2015c) published clarifications about the concept of affects and how to measure them, because it seemed that people often had misunderstood the topic and confused affects for example with job satisfaction and well-being, which are both attitudes, not affects.

Once it was known that happiness and productivity were linked together, Graziotin et al. (2017b) conducted a study to research the effects of unhappiness among developers. They conducted a survey and analyzed the 181 responses they got. They were able to identify 49 consequences of unhappiness, which were then divided into three categories: developer's own being, process, and artifact. The greatest consequences of unhappiness for developer's own being were low cognitive performance, mental unease or disorder, low motivation and even work withdrawal. The greatest consequences of unhappiness for the software development process were low productivity, delayed execution of process activities, lack of following the process practices, and breaking the development flow. The greatest consequence of unhappiness for the software artifact was low code quality, even to the point of discarding code that made the developer feel upset.

Graziotin et al. (2015d) conducted a study where they observed and interviewed two software developers who were running a real-word project in a work-like setting for several months for their bachelor's thesis. The study had similar results as previously; positive affects increase performance and move focus on the task, while negative affects did the opposite. The experienced affects could be from an event of the development work, for example fixing a bug or being stuck, or from personal life with no connections to the work. Regardless of the source of the affect, it had an impact on the development work: positive affects had a positive impact and negative affects had a negative impact. Some examples of positive affects are being able to fix a bug, making progress and finding useful features in a software tool. Some examples of negative affects are not making progress, loss of task purpose and vision, boredom because implementing only minor things, and lack of support from one's team.

Müller and Fritz (2015) found out in their study that the emotions of developers and their performance were linked together. They studied which factors were causing the changes in the emotions of developers and found out that locating the relevant code and understanding how the code works were the most effective factors both for increasing and decreasing developer's mood. When a developer found the relevant code and understood the code, their mood increased and when they were not successful with locating or understanding the code, their mood decreased. Writing code, whether it was correct or not, increased the mood of the developers. From participant interviews it was found that the participants would have liked to have more documentation and broader documentation as well as a documentation of the high-level architecture of the system and code examples.

Meyer et al., (2021) conducted a study about what makes a developer's day a good day and which factors of their day made them feel less happy. They sent a survey to 37 792 developers working at Microsoft and received 5 971 responses in total. Meyer et al. found out that developers see days with boring and repetitive tasks as bad days. Time pressure and approaching deadlines caused stress and therefore unhappiness for the developers. Developers get frustrated by disruptive activities, such as meetings, because they fragment their day and make progressing with the main coding tasks more difficult. However,

meetings were not always seen as a bad thing, but in order to have a positive impact, they need to be used during the specification, planning, or releasing phases of a software project. Otherwise, developers thought that the meetings did not have value for them and were only taking time from the actual tasks. Developers found a day to be a good day when they were able to focus on their main coding task for a long period of time without interruptions. Smaller offices, remote work and grouping meetings together could be ways of making a day good. Generally, developers seemed to have more good days when they had more control over their schedules and the contents of their working day and were able to make progress and create value on projects that they found meaningful.

2.1.2. Modeling Developer Experience

There are a few different models which are used to model which factors are causing the emotions, moods and feelings experienced by the developer. ISO 9241-210 (International Organization for Standardization, 2019) defines User Experience (UX) as "a person's perceptions and responses resulting from the use and/or anticipated use of a product, system or service". UX can be applied in any context, and it has been used as a one of the base models when defining Developer Experience and Programmer Experience (Fagerholm and Münch, 2012; Morales et al., 2019).

Fagerholm and Münch (2012) proposed the first definition for the concept of Developer Experience (DX) in 2012. They originally used the abbreviation DE^X, but it has then settled to be DX (Fagerholm, 2015). The concept is based on related models such as User Experience, Customer Experience, Brand Experience, and models of high-performing teams. In the context of DX every person engaged in development activities is considered to be a developer. Fagerholm and Münch divide experiences affecting the DX in three factors: cognition, affect and conation. Factors are shown in Figure 1. Fagerholm and Münch assume that a good DX leads to a greater performance of the developers and better software engineering outcomes, which is supported by the studies discussed in chapter 2.1.1. (Fagerholm, 2015; Fagerholm and Münch, 2012)

The cognition factor consists of a developer's experiences of the development infrastructure, for example, the development process and procedures, used development platform and technologies, and developer's technical skills. It includes the concrete interaction with the development tools, such as Integrated Development Environments (IDE) and programming languages and the software development process and process management tools. The affect factor consists of a developer's experiences of their work, for example their team, received respect, feelings of belonging and attachment, and the social aspects of their work environment. The conation factor consists of a developer's experiences of their contribution's value, for example whether they feel like the work they do is meaningful and aligned with their personal values, plans and goals. (Fagerholm and Münch, 2012)

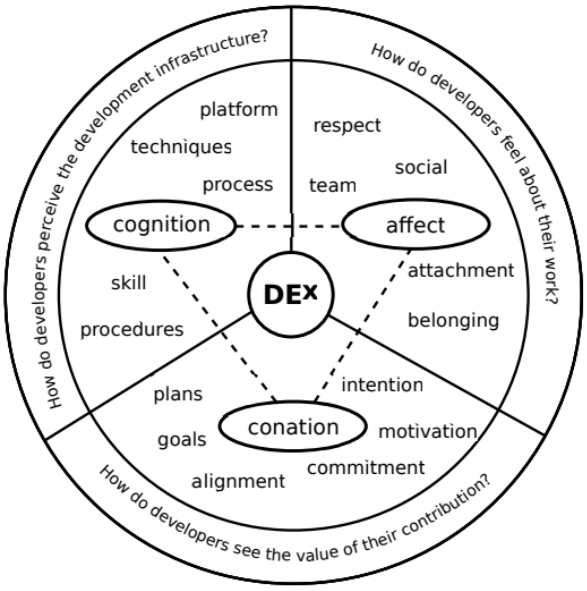

Figure 1. Developer Experience: Conceptual Framework. (Fagerholm and Münch, 2012)

Another model for understanding the mood of a developer is Programmer eXperience (PX) proposed by Morales et al. (2019). Morales et al. define PX as "the result of the intrinsic motivations and perceptions of programmers regarding the use of development artifacts." They established that both the intrinsic motivation of developers and the development tools that developers use influence the PX. In the context of PX, a developer is defined as a person who does programming and code maintenance, leaving other aspects of software engineering process outside of the concept of PX. PX is a particular case of UX, where the user is a

developer who is a user of programming environments and software artifacts. The main difference between PX and DX is that DX is a broader concept than PX, including all software engineering activities, processes, and artifacts, while PX includes only programming environments, design documents and programming codes (Fagerholm and Münch, 2012; Morales et al., 2019). PX has much in common with the cognition factor of DX but lacks the other two factors, affect and conation (Fagerholm and Münch, 2012; Morales et al., 2019). Still an explicit definition for PX is lacking, other than proposed by Morales et al. (2019), and the field of PX requires more research (Khan and Rana, 2021; Morales et al., 2020b).

Khan and Rana (2021) conducted a study where they gathered multiple studies about PX and classified those under four categories: programming languages, IDEs, Application Programming Interfaces (API), and developers. Programming languages and their tools can impact PX by either improving or decreasing the readability and understandability of the code as well as by either facilitating or not facilitating the tasks the developer need to do. Programming environments, such as IDEs, can also affect the PX. They can improve the PX by offering required tools and features in understandable form for the developer. However, they can also decrease the PX if there is too much in the programming environment making the developer overwhelmed. On the other hand, if the programming environment is too simple and therefore lacking functionalities, for example to easily edit and change code, it also decreases the PX. If an API is easy to use and there is sufficient documentation available, the developer would make less errors and have increased PX. If an API is difficult to use and the developer makes a lot of errors, it decreases the PX. The study (Khan and Rana, 2021) showed that more research is needed in the field of PX as well as a proper definition for PX.

There is not a precise model for evaluating PX, but Morales et al. (2020a) created a set of heuristics to evaluate programming environments, which are one part of PX, by applying known UX and usability attributes in programming environment context. The set of heuristics is shown in Table 1. A set of heuristics to evaluate a programming environment. (Morales et al., 2020a)

Table 1. A set of heuristics to evaluate a programming environment. (Morales et al., 2020a)

Heuristic	**Definition**
Visibility	The programming environment must keep the developer informed about its status. In addition, it must deliver timely information
Developer language	The programming environment must speak the developer's language without ambiguous terminology
Control	The programming environments must give control to the developer on their project, in a safe way
Consistency	The programming environments must be consistent in appearance and behavior
Error-Prevention	The programming environments should favor the prevention of errors over their reporting. If the system can prevent an error or enable to use a workaround to solve an error, then it should do it
Recognition	Programming environments should minimize the memory load of the developer, favoring recognition
Flexibility of use	The programming environments must allow different ways of use
Minimalist design	The programming environments must show only important information orderly and clear
Error handling and recovery	The programming environments should offer clear error messages and facilities to recovery
Help	The programming environments must provide aid for developers in relation to the way they should be used
Configurable interface	The programming environments must allow the developer to customize its interface
Automatic feedback	The programming environment should offer the developer automatic feedback that facilitates his/her work

2.2. Type systems

“At an abstract level, a type is a constraint which defines the set of valid values which conform to it.” (Zelkowitz, 2009). A type is most often seen as a form or as a shape of data. A type can be a primitive type, for example an integer or string, which describes the data structure in which the data is stored in a hardware. A type can also describe a class and its fields and methods, or a function and its parameters and return values. All programming languages have a type system. There are different type systems, which define how and when the programming language is using the types. (Zelkowitz, 2009)

Different type systems suit best for different usages and therefore it is important to know about the characteristic of different type systems when choosing a programming language for a specific task. Changing a programming language of a software later is usually difficult if not even impossible, so the chosen programming language should fit the requirements of the project from the very beginning until the end of the software’s life cycle.

2.2.1. Type system characteristics

Zelkowitz (2009) introduces seven language characteristics related to type systems:

1. Compile-time type checking
2. Run-time type checking
3. Safe typing
4. Implicit typing
5. Structural typing
6. Run-time type errors
7. Implicit type conversions

The first two characteristics, compile-time type checking and run-time type checking, define when the programming language executes type checking. Types can be checked in compile-time or at run-time, but for example in Java types are checked both in compile-time and run-

time. Generally, when types are enforced in compile-time, the programming language is considered to be statically typed, and when types are enforced in run-time, the programming language is considered to be dynamically typed. Popular (Stack Overflow, 2021) statically typed programming languages are, for example, Java (Oracle, 2022) and C# (Microsoft, 2022b). Popular (Stack Overflow, 2021) dynamically typed programming languages are, for example, JavaScript (MDN, 2022) and Python (Python, 2022). However, there is not one well-established definition for dynamically typed or statically typed programming languages, but a set of properties that are usually associated with them. (Zelkowitz, 2009)

The third characteristic, safe typing, defines whether the programming language permits a developer to completely overrule the used type system by casting a typed value to be treated as a value of another type. If a programming language permits arbitrary type castings, it is considered to be unsafely typed, and if it does not permit arbitrary type casting but has internal checks whether the casting is valid relative to the type system, it is considered to be safely typed. Being statically typed does not mean that the programming language is also safely typed and vice versa. A programming language can be statically typed and have unsafe typing, or dynamically typed and have safe typing. (Zelkowitz, 2009)

Zelkowitz (2009) states that safely typed is often referred as strongly typed and unsafely typed as weakly typed. However, there is no single commonly accepted definition for what is considered as weakly typed or strongly typed. Wolf (1989) defined the difference between weakly and strongly typed languages by the amount of type features they had. In the research Wolf compared two OOP (Object-Oriented Programming) programming languages, Flavors and C++, against each other. Flavors was said to be weakly typed, because it had limited number of types to use and there were practically very little ways to create more. C++ was seen as strongly typed, because it allowed user to define new types and had more type features. This contradicts with Zelkowitz' definition completely. Ray et al., (2014) defined the difference between weakly and strongly typed programming languages as whether the programming language silently admits type-confusion or detects type-confusion and reports it. Type confusions means that the language interprets a value to be a different type than it is. Wolf presented JavaScript's implicit type casting while adding together string and a

number as an example. Wolf's definition is more in line with Zelkowitz' definition, but still contradicts it, because Zelkowitz does not include implicit type casting within the division into weakly and strongly typed languages but sees it as an independent characteristic of a type system of a programming language.

The fourth and fifth characteristics, implicit typing and structural typing, are applicable only for statically typed programming languages because they define how the types declared by the developer are handled. If a programming language can infer the type of a variable and requires explicit type declaration only when it cannot infer the type automatically, it is said to be implicitly typed. If the type must always be explicitly declared, the language is said to be explicitly typed. For example, in Figure 2 the variable point1 is explicitly typed and the variable point2 is implicitly typed. Structural typing defines whether the programming language's mechanism of organizing types into hierarchies is structural typing or nominal typing. Nominal typing means, that the relations between types are explicitly declared, while structural typing means that the type system will automatically infer relations between types. For example, TypeScript is structurally typed programming language ("TypeScript," 2022), and therefore the function usePoint in Figure 2 can be called with both point1 and point2, even though only point1 is declared to be type Point and point2 has implicit type {x: number, y: number}. In structural typing all types that conform the constraints of a type are that type. In nominal typing usePoint could not be called with point2, because it has not been explicitly declared to be type Point. (Zelkowitz, 2009)

```
type Point = {
    x: number;
    y: number;
}

function usePoint(point: Point) {
    //...
}

const point1: Point = {
    x: 34,
    y: 56
}

const point2 = {
    x: 34,
    y: 56
}

usePoint(point1);
usePoint(point2);
```

Figure 2. Structural typing.

The sixth characteristic, run-time type errors, defines whether the programming language throws type errors in run-time. In statically typed languages type errors should not occur in run-time, but if the type system is unsafely typed, type errors can occur also in run-time. (Zelkowitz, 2009)

The seventh characteristic, implicit type conversions, defines whether the programming language will interpret a value with certain type to be other type when necessary (Zelkowitz, 2009). For example, JavaScript (MDN, 2022) is dynamically typed programming language, which does implicit type castings. If incompatible types would cause a run-time error, JavaScript usually casts types other than strings to strings and solves the error in run-time instead of throwing an error (MDN, 2022). Implicit type conversion is demonstrated in Figure 3. When a string and a number are added together, the result is string, because JavaScript detects the incompatible types and converts the number to string before proceeding.

```
>> "Type of number being added to type of string" + 1
<- "Type of number being added to type of string1"
>> 1 + "Type of string being added to type of number"
<- "1Type of string being added to type of number"
```

Figure 3. Implicit type conversion in JavaScript.

Python is also a dynamically typed programming language, which does not do implicit type casting, but throws an error when it detects a problem with typing (Python, 2022). The same example of a string and a number being added together behaves differently in Python than in JavaScript as seen in Figure 4. Instead of converting the types of the values to be compatible, Python throws a run-time type error which informs that an integer cannot be added to string.

```
>>> "Type of number being added to type of string" + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

Figure 4. Type error in Python.

To avoid the error, the developer must explicitly cast the number to be string. The example of this is seen in Figure 5. When the number is explicitly converted to string, Python does not throw an error.

```
>>> "Type of number being added to type of string" + str(1)
'Type of number being added to type of string1'
>>>
```

Figure 5. Explicit type conversion in Python.

2.2.2. Comparison between dynamic and static type systems

The debate between whether one should use a dynamically or statically typed programming language has been going on for decades (Okon and Hanenberg, 2016; Wolf, 1989). Back in 1989 Wolf compared two differently typed languages; one was very weakly typed and did not have many type features, while the other had more type features. After the comparison, Wolf concluded that stronger type system had more benefits than the weaker type system. In 2009 Zelkowitz compared type system characteristics of different programming languages. Zelkowitz promoted dynamic type systems over statical type system for their simplicity and flexibility. In 2016 Okon and Hanenberg questioned the previous studies comparing dynamically and statically typed languages because they might have been affected by the researcher's bias and assumption, that statically typed programming languages are more beneficial. Okon and Hanenberg designed an experiment, which favored dynamically typed programming languages over statically typed. Out of four tasks, only in two a benefit for dynamically typed programming language was measured. In the other two tasks, a benefit for statically typed programming language was measured. Type systems have been compared for decades by many researchers, but no rigorous conclusion has been made. However, subdivisions and characteristics of type systems can be derived from the studies and these characteristics can be compared.

Because types are checked at compile-time in statically typed programming languages, they have the ability to catch bugs before deploying to production and they have less type related bugs (Gao et al., 2017; Paulson, 2007; Ray et al., 2014). Static type system also reduces the effort needed to fix type related bugs (Hanenberg et al., 2014). However, static typing does not prevent semantic errors (Hanenberg et al., 2014). Gao et al., (2017) found that static typing would prevent 15% of bugs in JavaScript code. Ray et al., (2014) had similar results when they compared dozens of programming languages to find out, which ones are more prone to have bugs. While JavaScript itself had little less bugs than the languages in average, TypeScript had significantly less bugs in all categories compared to JavaScript. In addition to having generally more bugs, dynamically typed languages do not have the same level of performance in runtime as statically typed languages tend to have (Paulson, 2007). This is because a dynamically typed language must check types in run-time which slows down the

performance. On the other hand, the development work with statically typed programming languages can be slower due to longer compilation times (Pang et al., 2018; Paulson, 2007).

There are studies which have contradicting results about the effect of a type system on development speed. For example, Stuchlik and Hanenberg (2011) conducted a study using dynamically typed Groovy and statically typed Java. In three out of the five tasks, the participants using Groovy were faster and in the other two tasks there was no significant difference. Fischer and Hanenberg (2015) conducted a study to find out how the code completion features of an IDE and type systems affect development time. They created a project consisting of 1400 lines of code in 57 files divided into 5 folders. They defined two tasks that the participants completed. One group used dynamically typed JavaScript and the other statically typed TypeScript. In both tasks, the participants using TypeScript were able to complete the tasks faster and the code completion feature had greatest positive impact when used with TypeScript. Endrikat et al. (2014) conducted a similar study to find out how documentation and type systems affect development time. In the experiment the participants worked on one single larger task. Participants were divided into four groups of which two used dynamically typed programming language and the other two used a statically typed programming language, and two of the groups, one using statically typed language and the other dynamically typed, had documentation available while the other two groups did not. Participants who used statically typed programming language completed the task faster than participants who used dynamically typed programming language. Participants who had documentation available were faster than the participants who did not have access to documentation. The fastest participants used statically typed programming language and had access to documentation. Mayer et al. (2012) conducted a study to research how type systems affect when working with undocumented code. They found out that statically typed programming language was better choice when the code had many classes and the type annotations documented design decisions of the code. Dynamically typed programming language was better when the task was easy. Mayer et al. concluded that there is no single answer that which one is better, but that it is dependent of the context. Wolf (1989) said that development with dynamically typed programming languages can be faster at first, but then drastically slows down when the project size grows. Pang et al. (2018) had similar results when they interviewed developers about their experiences of type systems.

There is a difference in the amount and level of tooling available for dynamically and statically typed programming languages (Fischer and Hanenberg, 2015; Pang et al., 2018; Paulson, 2007). Paulson (2007) said that most of the dynamically typed programming languages do not have enough tooling to be beneficial for larger projects. While conducting the study about the effect of code completion feature and type system on development speed, Fischer and Hanenberg (2015) noticed that IDEs for dynamically typed languages do not have as sophisticated code completion features as IDEs for statically typed programming languages, because the IDEs for statically typed programming languages were able to suggest more accurate completion for the code. Petersen et al. (2014) found out that even when using a modern IDE to perform coding tasks, a statically typed language performed better. The set of available tools and support for a language affects developers choice of programming language and the used tools also affect the developers attitude towards the programming language (Pang et al., 2018). Statically typed languages seem to generally have better tooling and support than dynamically typed languages (Fischer and Hanenberg, 2015; Pang et al., 2018).

Hanenberg et al. (2014) conducted a study about type systems effect on software maintainability. In the experiment, two groups were given the same nine tasks to be completed in the given code base. Both groups did the same task both with Groovy and Java. One group did the tasks first with Java and the other started with Groovy. They found out that statical typing is an effective way to document code and that statical typing reduced the effort of fixing type errors. Hanenberg et al. tracked how many files the participants opened during the experiment and found out that the participants using Groovy opened a greater number of files before finding the correct one than participants using Java. Static typing seemed to help to find the correct files faster. Static typing helps to understand code written by other people and to create consistent structure (Pang et al., 2018).

While many studies seem to show that static type systems are more beneficial, dynamic type systems have their place. After concluding their comparison between strongly and weakly typed languages, Wolf (1989) did not state that one language was greater than the other, but that they suit for different purposes and their differences must be considered before deciding

which programming language to use in a project. Wolf said that programming languages with weaker type systems are more flexible and can suit well for example prototyping, while languages with stronger type system suit for larger projects. Pang et al. (2018) had similar results from their interviews: dynamically typed programming languages are great for small projects or for quick hacky things. Dynamically typed languages are not a great choice for mission critical applications or for rigid typed development like operations systems (Paulson, 2007). Developers do not find switching between programming languages with different type system problematic and using commonly known and liked programming languages might help to recruit competent employees (Pang et al., 2018).

2.2.3. Gradual typing

Siek and Taha (2007, 2006) introduced the idea of mixing static and dynamic type system within one program. They called the new type system that was capable to provide features of both dynamic and static type system as gradual typing. In 2006 they developed a gradual type system for a functional programming language and in 2007 for an object-oriented language. Once gradual typing had grown more popular, Siek et al. (2015) refined the criteria for gradual typing to clarify its purposes and to avoid misusing the term. The objective of gradual typing is to have the benefits of both type systems within one program by letting the developer control the degree of static and dynamic type checking they need to use in specific parts of the program. Gradual type system can have the early error detection of static type system and the flexibility of dynamic type system. Siek and Taha see also migration from dynamically typed programming language to statically typed programming language as a use case for gradual type system. Often the development of a new program is started with a dynamically typed programming language but then migrated into statically typed language once the program grows larger. Migration between the two type systems could ideally happen gradually, but without gradual typing it has not been possible. With gradual typing, the migration process can be done gradually while still using the same programming language but only with a different degree of typing. (Siek and Taha, 2007, 2006)

2.3. Type system's effect on Developer Experience

There is very little research about type system's effect on DX or PX. However, for example programming languages, IDEs and other tools, APIs and the inner motivations of developers affect the PX (Khan and Rana, 2021). In previous chapters statically typed programming languages are reported to have many features and characteristics, that are also reported to possibly improve the PX. A common source of frustration of a developer is lack of documentation (Müller and Fritz, 2015), and static typing has been found to function as a type of documentation for the code (Hanenberg et al., 2014; Mayer et al., 2012). Making progress and not getting stuck for too long time has been reported to increase DX (Graziotin et al., 2015d; Meyer et al., 2021), and type system seems to have an effect on development speed: static type system increases development speed for large projects and dynamic typing increases the speed for small projects (Endrikat et al., 2014; Fischer and Hanenberg, 2015). With statically typed programming language, a developer finds the relevant files (Hanenberg et al., 2014; Müller and Fritz, 2015) and understands the code faster (Müller and Fritz, 2015; Pang et al., 2018) than when using dynamically typed programming language, which both also increase the development speed and therefore have a positive impact on DX. On the other hand, static type checking can also take time and therefore slower the development speed (Paulson, 2007). Statically typed programming languages have generally better tooling available, which helps to increase development speed (Fischer and Hanenberg, 2015).

2.4. TypeScript

TypeScript is a programming language developed by Microsoft. TypeScript adds static typing to JavaScript. Depending on the compiler options, TypeScript can be seen as a statically typed programming language or a gradually typed programming language, because JavaScript and TypeScript can coexist within one project. (Microsoft, 2022a)

The first public version of TypeScript was version 0.8 which was released in October 2012 ("Announcing TypeScript 1.0," 2014). Version 1.0 of TypeScript was released in April 2014

("Announcing TypeScript 1.0," 2014) and TypeScript started to grow its popularity after the release of version 2.0 in 2016 ("Releases · microsoft/TypeScript", 2022, Stack Overflow, 2021, 2020, 2019, 2018, 2017, 2016, 2015). The growth of the popularity is shown in Figure 6.

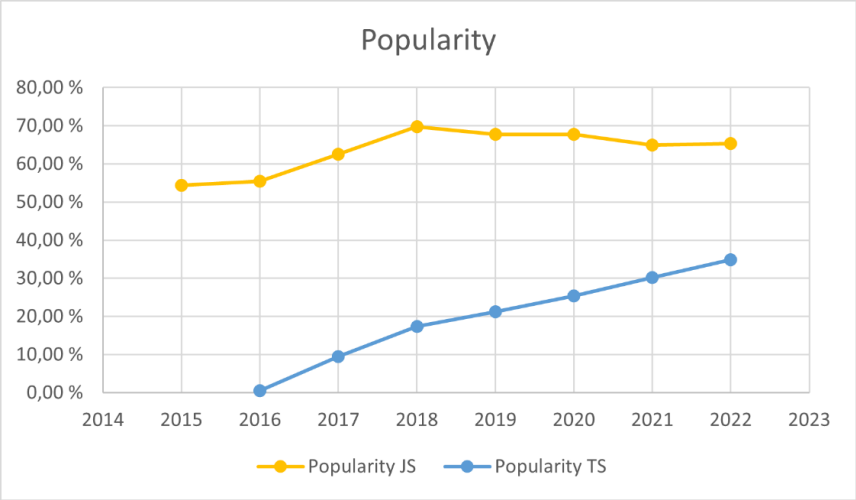

Figure 6. Popularity of JavaScript and TypeScript according to Stack Overflow surveys. (Stack Overflow, 2022, 2021, 2020, 2019, 2018, 2017, 2016, 2015)

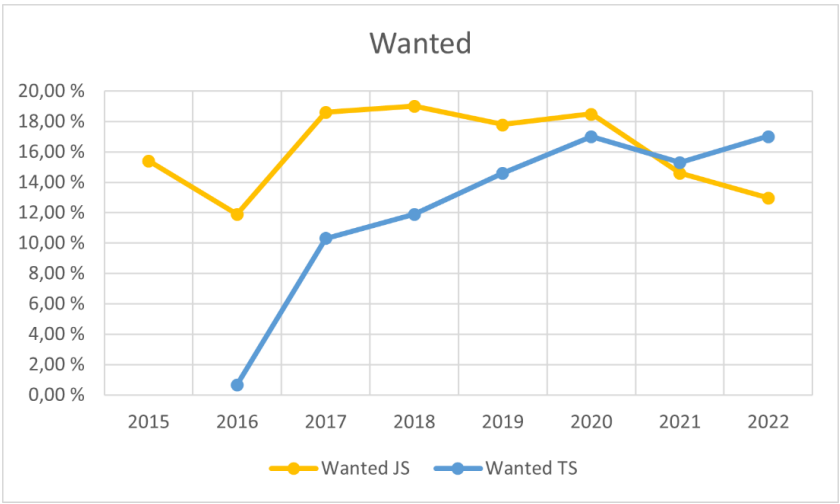

Figure 7. Amount of developers who would like to use JavaScript and TypeScript. (Stack Overflow, 2022, 2021, 2020, 2019, 2018, 2017, 2016, 2015)

TypeScript has become one of the most wanted technologies among developers according to Stack Overflow surveys. The amounts of developers wanting to use JavaScript and

TypeScript are shown in Figure 7. The figure shows that nowadays TypeScript is more wanted technology than JavaScript. However, TypeScript is used more among professional developers than among people who are just learning to code (Stack Overflow, 2022). Out of professional developers 40.08% had used TypeScript and 67.9% had used JavaScript over the past year for extensive development work, while out of people just learning to code only 15.05% had used TypeScript and 59.79% had used JavaScript for extensive development work (Stack Overflow, 2022).

TypeScript builds on top of JavaScript and all valid JavaScript is also valid TypeScript. TypeScript is compiled into JavaScript and therefore it can run in any environment that JavaScript can run, for example browsers, Node, Deno, and command line terminal. The type features introduced in chapter 2.2.1 of JavaScript and TypeScript are shown in Table 2. Type features of JavaScript and TypeScript. TypeScript is compiled to JavaScript and the static types are checked and removed during the compilation. Therefore, TypeScript has compile-time type checking and JavaScript does not. They both have run-time type checking because TypeScript cannot be ran as it is, but it needs to be compiled to JavaScript and JavaScript has run-time type checking. TypeScript is unsafely typed because it allows a developer to completely overrule a type to be treated as another type. JavaScript does not have a feature to cast types. TypeScript supports both implicit typing and explicit typing. TypeScript's type system is structural, not nominal. As both JavaScript and TypeScript have run-time type checking, they also have run-time type errors. The same applies for implicit type conversions too. (MDN, 2022; Microsoft, 2022a; Zelkowitz, 2009)

TypeScript Handbook (Microsoft, 2022a) lists all TypeScript features and how to use them. Most of them are related to the static type system, but in addition to it, TypeScript compiler comes with a built-in tool to compile to a specific Ecmascript version. Another feature that is not related to types is enums. Enums are a way of defining a set of values that can be used as a type and but also as a value. Unlike types, enums are a run-time feature and are not completely removed during the compilation, unless they are declared as constant enums. Constant enums are completely removed during compilation. Using constant enums instead of storing values in variables in JavaScript can offer code optimization. The TypeScript

Handbook also says that editors that support TypeScript can have better refactoring, navigation, and auto-complete features with TypeScript. As mentioned earlier, TypeScript decreases the amount of bugs in JavaScript (Gao et al., 2017; Ray et al., 2014).

Table 2. Type features of JavaScript and TypeScript.

Type feature	**JavaScript**	**TypeScript**
Compile-time type checking	No	Yes
Run-time type checking	Yes	Yes
Safe typing	-	No
Implicit typing	-	Yes
Structural typing	-	Yes
Run-time type errors	Yes	Yes
Implicit type conversions	Yes	Yes

2.5. Previous migrations from dynamically typed to statically typed

Migrations from dynamically typed programs to statically typed programs have been researched very little, probably due to the challenging nature of such research. Machiry et al. (2022) presented a process to migrate a program written in C to Checked C. Checked C is a programming language that extends C, but adds checked pointer types to reduce the number of type errors in C. The process consists of replacing the regular C pointers with the new typed pointer annotation and refactoring the code to satisfy the new typed pointers. The migration process was found to be impossible to fully automate, so human input was used as a part of it. They implemented a tool called 3C to assist with the migration process. 3C replaces the pointer annotations and in case of a problem, it traces the root problem of type unsafety and asks for a input from a developer, who then refactors the code from which the 3C then continues. Migrating a whole program at once was found to not be the optimal solution but migrating smaller pieces at a time was thought to be a possibly better experience.

Tobin-Hochstadt et al. (2017) researched migratory typing by explaining how Typed Racket, Racket's typed twin language, was implemented and how it has been used to add static typing in Racket programs. In migratory typing, the purpose is to migrate from dynamically typed code to statically typed code incrementally, minimally one module at a time, while still being able to run the program during the migration. Migratory typing differs from gradual typing by its goal and the minimum amount of code to be converted at once. Migratory typing aims to convert a code base from dynamically typed to statically typed. This is achieved by converting a module from dynamic typing to static typing, and one module is the minimum unit which can be converted independently. In gradual typing, the goal is to permit the mixed use of dynamic and static typing instead of enforcing the static typing (Siek and Taha, 2006). Gradual typing also enables adding types only for a portion of a module, instead of forcing the whole module to be typed (Siek and Taha, 2006). Tobin-Hochstadt et al. (2017) found out that usually the size of a module to be migrated was not a problem for developers, but that they would still prefer gradual typing so that they could add types only to the most relevant parts of the code instead of the whole module. Another observation they made was that a migration of functional code was relatively easy and low effort, but a migration of object-oriented code required more work and thought from the developers. While 5-10% of functional code had to be touched while migrating, 15-20% of object-oriented code needed changes during the migration. The migration effort should be kept low to avoid developers becoming reluctant to participate in the migration. Type soundness was found to have a negative impact on the programs performance, as some type checks were compiled into heavy run-time checks or to code that allocated a lot of memory. That raises a question whether performance be sacrificed for type soundness or not. (Tobin-Hochstadt et al., 2017)

3. Research method

Hevner et al. (2004) say that there are two research paradigms in information system (IS) research. One of them is the behavioral-science paradigm, where theories, for example laws and principles, about humans and organizations are built and then tested. The second paradigm is the design-science paradigm, which aims to solve problems by designing artifacts for specific problems and then evaluating the performance of the artifacts. IS design-science research often studies an information technology (IT) artifact, that was implement in a specific organizational context. Theories are then built to predict and explain how useful the used IT artifact is and how it affects the organization and its people. (Hevner et al., 2004)

This thesis uses the Design Science Research Methodology (DSRM) created by Peffers et al. (2007) as its research method. DSRM contains six activities (Peffers et al., 2007):

1. Identify problem and motivate
2. Define objectives of a solutions
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

The DSRM process model is shown in Figure 8. This research is done within a case company, which has five SaaS (Software as a Service) products and approximately 350 employees in total. The case product is one of the SaaS products with a research and design team of approximately 30 employees. The case product is described in chapter 6. TypeScript migration demonstration. The case software, which refers to the user interfaces of the case product, which are a web UI and a mobile application, is developed with an agile development process called Scrum (Schwaber and Sutherland, 2020). Scrum is an iterative

and agile framework for developing complex software. In the first activity of the DSRM, identify problem and motivate, the problem which is tried to be solved is defined and the value of solving the problem is shown. In this thesis the problem was identified and defined by the case company, so this thesis starts the DSRM process from activity 2, defining objectives for the solution. Defining the objectives for the solution consists of inferring the desired outcomes of using the artifact in the specific context from the problem and describing them. In this thesis, this is done in chapter 4. Activity 3, design and development, consist of the design and creation of the artifact. Activity 4, demonstration, means that the designed artifact is used to solve an instance of the problem described in activity 1. In this thesis, this is done by designing a model for a TypeScript migration and demonstrating it in the context of the case company. The migration model is described in chapter 5. Demonstration of the TypeScript migration is discussed in chapter 6.

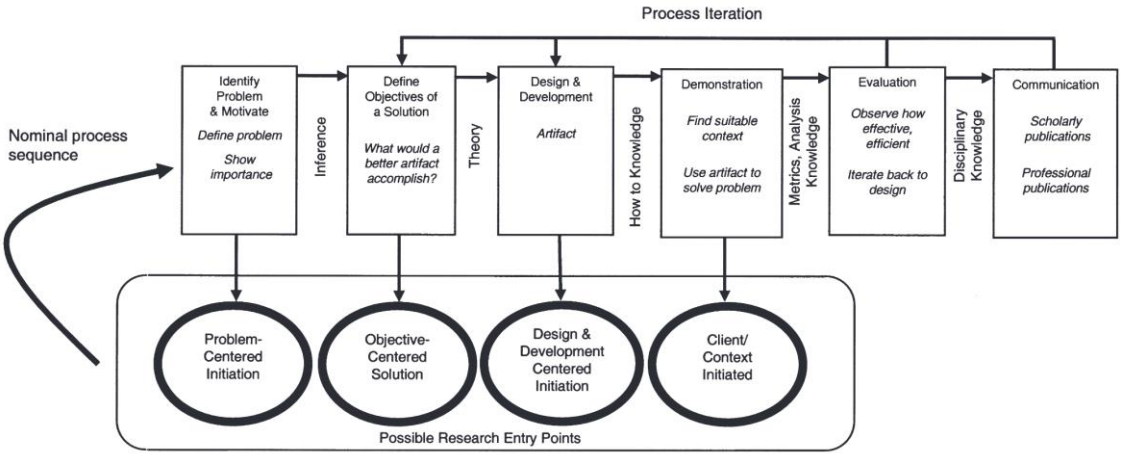

Figure 8. DSRM process model (Peffers et al., 2007).

Activity 5, evaluation, means that the usability of the artifact to solve the problem is observed and measured. The performance of the artifact is compared to the objectives that were set in activity 2. In this thesis, the evaluation is done by two activities: using the observations of the author who is also an organization insider, and interviewing the developers whose work and Developer Experience the artifact may affect. The observations of the author are used to evaluate the migration model and its sufficiency, while the interviews are used to evaluate how the migration possibly affects the Developer Experience. Therefore, the evaluation of

the artifact relies heavily on the observations made by the author. The author will execute each step of the migration model and will therefore have the best knowledge of the usability of the artifact. Activity 6, communication, consists of the activities done to communicate the research, its importance, the artifact, and results to researchers and practitioners. In this research, this thesis will be the way of communicating the DSRM process and its outcomes.

4. Objectives of a TypeScript migration

In this chapter the objectives of a migration from JavaScript to TypeScript are described. The migration is expected to improve developer experience, decrease the number of bugs in the code and possibly add code optimization. In addition to these, the migration is expected to help to keep both the developers and the software up to date with modern technologies. The objectives are shown in Table 3. Objectives of a TypeScript migration.

Table 3. Objectives of a TypeScript migration.

Objective	**Achieved by**
Improved DX	- Better documentation. - Better tooling. - More understandable code. - More consistent code structure.
Decreased number of bugs	- Decreasing the number of type related bugs with compile-time type checking.
Keeping up to date with modern technologies	- Enable usage of the newest Ecmascript features. - Keeping the software compatible with future technologies. - Keeping the skills of developers up to date with modern technologies.
Code optimization	- Less unnecessary conditionals. - Using constant enums which are replaced with inline values, which are optimized by JIT.

4.1. Improved Developer Experience

As discussed in the chapter 2.3, type systems probably have an impact on DX. Statically typed programming languages were reported to be better documented because the static typing itself acts as a documentation. Lack of documentation and difficulty to locate and understand code were reported to cause frustration and unhappiness among developers. Hall et al. (2007) conducted a study to find out which skills of a developer are most appreciated amongst other developers, and it turned out that the soft skills, such as communication, documenting their work and sharing knowledge, were rated higher than the technical skills of a developer. Static typing might help developers to communicate with each other. Therefore, the documentation created by adding static typing in the code might improve the DX by making the code more understandable, especially in large projects with multiple developers working with them. Nahum (2019) says in his presentation "How to inject TypeScript into existing projects" that the types help especially developers new to the project to understand the code. TypeScript can also help to keep a consistent structure in the code, which also helps to understand it.

In the chapter 2.2.2, static and dynamic type systems were compared, and static type systems were found to increase the development speed in big projects. Statically typed programming languages were also found to be having generally better tooling available than for dynamically typed programming languages, which could also mean that the development of a big software is more rapid with statically typed programming language. Being able to make progress and having a sense of accomplishment were reported to increase developers moods while slow development and being stuff caused frustration for them. Therefore, at least in big software projects the static type system may increase the development speed and thus increase the DX.

4.2. Decreased number of bugs

As it was discussed in chapter 2.2.2, one benefit of a static type system and compilation-time type checking is catching bugs before deploying to production. Statically typed

programming languages in general have less type related bugs than dynamically typed programming languages and software made with TypeScript has significantly less bugs than software made with JavaScript. Therefore, the TypeScript migration is expected to decrease the amount of type related bugs.

4.3. Keeping the software and developers up to date with modern technologies

TypeScript has a downleveling feature, which enables the usage of the newest Ecmascript features while also supporting older browsers by downleveling the modern syntax to older one during compilation. Also, often new technologies build on top of old ones. If the technologies of a software do not stay up to date with new technologies, they might end up in a situation where they cannot take some new technologies into use, because they are incompatible with the old technologies in use. The same can apply with TypeScript. Even though TypeScript is not yet a requirement for any crucial technology, it might be in the future, as its popularity is rising. In addition to keeping the software up to date, it is also good to keep the developers up to date with modern technologies. Offering developers opportunities to learn and use interesting new technologies might keep them happier and make them stay within the current product for a longer period, because their knowledge and skills will not get outdated. Therefore, one objective for the TypeScript migration is to keep the software product up to date and prevent the developers changing jobs for more modern technologies.

4.4. Code optimization

Using TypeScript's constant enums can decrease the size of the compiled JavaScript code and increase its performance in comparison to using JavaScript constants. Another way the migration may create code optimization is by helping to detect and remove unnecessary type related checks in the code. For example, checking if a variable is null or undefined is common in code, but the check is unnecessary if it is known that the type of the variable cannot be null or undefined. When this is applied in the code, unnecessary checks are possibly removed, and the size of the code decreases and the performance increases.

However, TypeScript compiler does not optimize code and it is not the goal of the framework. Because TypeScript is compiled into JavaScript, there will still be run-time type checking. Therefore, the potential optimization should not be the main goal of the migration but seen as a positive side effect.

5. TypeScript migration model

In this chapter the TypeScript migration model is presented. First, the two strategies to execute the migration are introduced and then the process and steps of the migration plan are discussed.

5.1. Strategies

In software modernization, a software is evolved to use new technologies and/or platforms. Migration is one of the software modernization strategies. Other modernization strategies include replacement strategy, wrapping and redevelopment. When using the migration strategy, the software is transformed to use a new technology while maximizing the reuse of the existing code. (Khadka, 2016) That makes it a good strategy for modernizing a JavaScript application by changing it to use TypeScript, because the existing JavaScript code can be reused by adding the static typing to it. The TypeScript Handbook (Microsoft, 2022a) does not state anything about the migration strategy. Rauschmayer (2020) and Vann (2019) list two higher level strategies to migrate to TypeScript: the migration can be done all at once or gradually in smaller pieces, for example converting one single file at a time. Both strategies have their strengths and drawbacks, and the strategy should be chosen based on the characteristics of the project to be migrated.

Migrating gradually gives an opportunity to test TypeScript before fully committing to it. The migrated project can still be developed as before while also converting some parts of it to TypeScript. Then, if TypeScript is found not to be suitable for the project, it is easy to convert the few TypeScript files back to JavaScript or rely on version control to rollback. In contrast, if the whole project would have been migrated at once, rollbacking with the version control would cause losing the development work done after the migration, and conversion by hand would probably require a great amount of work. Gradual migration also prevents developers from getting frustrated with the migration work, as it can be done in small pieces simultaneously with the regular development work. The weaknesses of gradual migration

are that it probably takes a long period of time and prevents using some TypeScript features. Depending on the project size and the migration speed, the migration can take weeks, months, or even years. There is also a risk, that some parts of the code are never converted to TypeScript. While mixing JavaScript and TypeScript within one project is possible, it prevents the full potential of TypeScript. For example, using fully typed function from a TypeScript file in a JavaScript file removes all the type safety created with TypeScript. This can even make the project more prone to type errors than before. Mixing JavaScript and TypeScript also often requires the usage of a type called “any”, which is a type that disables type checking for values having that type. Using the any type means that the strictest rules of TypeScript cannot be used, which again decreases the benefits achievable with TypeScript.

The migration can also be executed all at once. When all code is migrated at the same time and there is no JavaScript left, TypeScript can be used to its full potential. The strictest rules can be used from the beginning and all features of TypeScript can be used more easily, for example enums. Because all code is TypeScript, type safety is preserved even when importing from other files. The weaknesses of migrating all at once are that it requires a lot of work, either by hand or by creating automation tools, it is more difficult to revert, and it might stop the normal development activities. Because most of the migration work is manual and repetitive, there is usually no point in migrating a large project by hand. Instead, automation tools should be used to minimize the manual work. However, there is only limited amount of such tools available and there is no guarantee that those will work in every context. There is a risk that those tools need to be created for the specific context. Migrating all at once might also stop the normal development work for a while, because changes made for the JavaScript files possibly conflict with the changes occurred during the migration.

Rauschmayer (2020) and Vann (2019) both bring up a migration method, where the whole large project is migrated at once and the caused type errors are handled with snapshot testing. When the project is converted to TypeScript, there will be probably thousands of type errors. From them, the initial snapshot is created. Then, the code can be edited, and its type errors can be compared to the snapshot. If type errors are fixed, the snapshot can be recreated from

the current situation and if there has been regression, it can be noticed from the previous snapshot.

5.2. Process and steps

Regardless of the chosen migration strategy, the process of initializing the migration will be similar for all migrations. The steps of the process are presented as an ordered list, but especially if there is uncertainty in the migration, it is best to start with the most uncertain part of it to avoid wasting time on other activities if a later step blocks the whole migration. This model does not provide an extensive guide on how to achieve each step for each technology and tool existing, but a structure to follow while initializing the migration for a web UI. The process contains the following steps:

1. Choose a migration strategy
2. Check library support and add type packages if needed
3. Configure TypeScript
4. Integrate TypeScript compilation within build process
5. Configure linter
6. Configure development tools and environments
7. (Optional) Generate API types
8. (Optional) Search for migration tools
9. Start migrating by following the chosen strategy

The first step includes choosing a suitable migration strategy based on the characteristics of the project to be migrated. Both strategies are described in the previous chapter. The second step includes checking if packages used in the project support TypeScript as they are or if they require a separate type package or if they do not have type definitions available. Whether a package supports TypeScript can be checked from the npm (npm Inc., 2022) page of the package or from TypeScript's own search tool (Microsoft, 2022c). Npm is world's

largest software registry, software package manager and installer (npm Inc., 2022). The form of type definitions is indicated with an icon. Figure 9 shows a package that has bundled type definitions within the package, Figure 10 shows a package that has a separate type package. If there is neither of the icons, there are no type definitions available for the package. Figure 11 shows the TypeScript's own package search tool which shows also how to install the required packages.

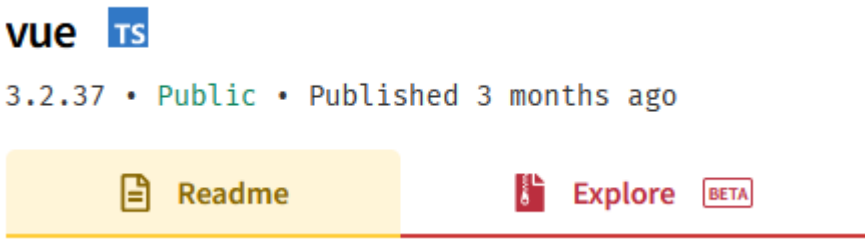

Figure 9. A npm package that has a TS icon next to its name, which means that it has bundled type definitions within the package.

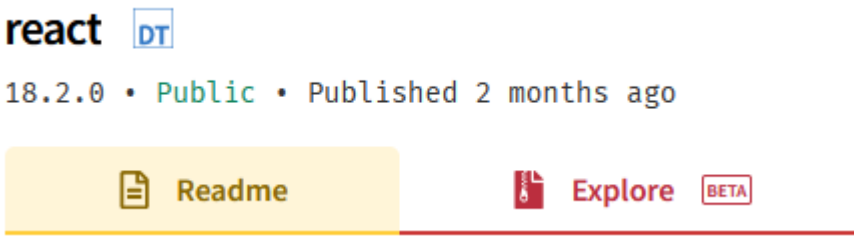

Figure 10. A npm package that has a DT icon next to its name, which means that it has a separate type package under Definitely Typed.

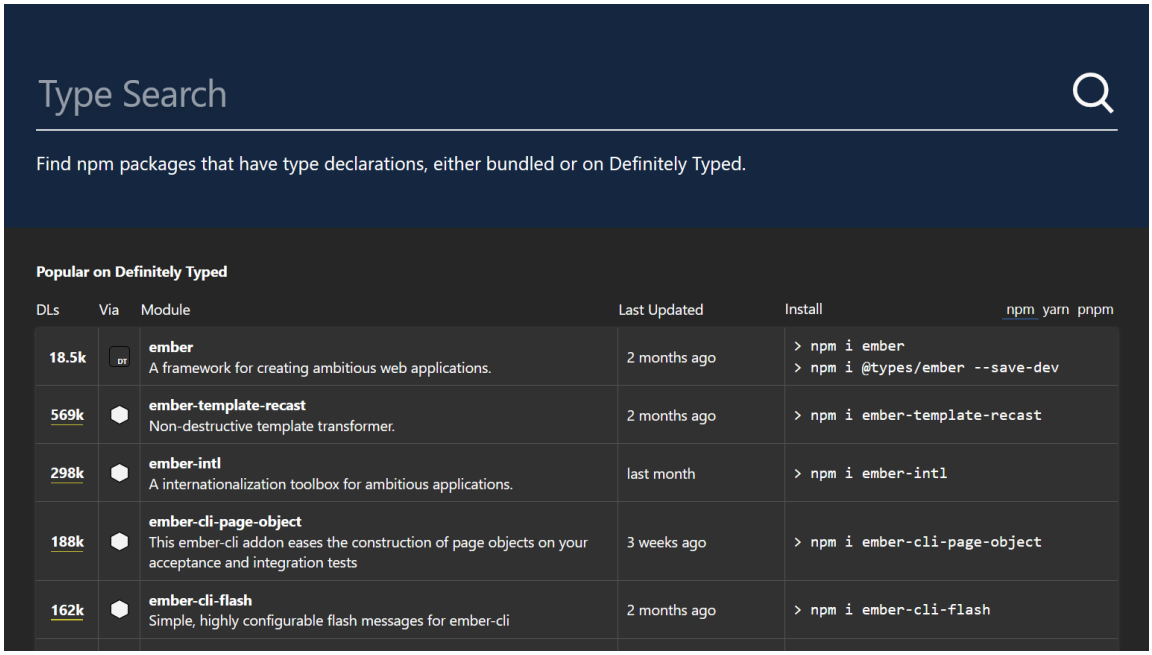

Figure 11. Type package search provided by Microsoft (Microsoft, 2022c).

If every used package has bundled type definitions, there is no need to do anything else in the second step. If some packages have type definitions in a separate type package, those packages need to be installed. If some packages do not have bundled type definitions or a separate type package, it must be considered if the type definitions should be created before continuing with the migration. If the package is not widely used in the project, it might be tolerable that it does not have type definitions. However, if the package is widely used or one of the frameworks used, then it should have type definitions. There are tools to generate type definitions from JavaScript code (“TypeScript,” 2022). If type definitions cannot be created for a key package, it should be carefully considered whether the migration should be continued or not. It is also good to note, that especially when using a separate type package, the types might not match perfectly with the implementation.

The third step is to create a TypeScript configuration file and add a way to compile the TypeScript code to JavaScript. There are multiple settings that can be used, and the configuration depends heavily on the project. Also, the chosen migration strategy affects how strict settings can be enabled. In gradual migration the any type cannot be avoided but if the migration is done all at once, it is possible to not use the any type. The configuration

file for TypeScript is “tsconfig”. There are over hundred settings that can be set in tsconfig, but they are not required. An example of a minimalistic tsconfig is shown in Figure 12.

Figure 12. An example of a TypeScript configuration.

The fourth step is to integrate TypeScript compilation to the build process of the project. Usually, the build process of a web UI can use other tools than TSC (TypeScript compiler), for example transpilers, such as Babel, and bundlers such as Webpack and Rollup. There are two main ways for integrating TypeScript compilation with them. Either the TSC can be added as a part of the used bundler and it can be used for compiling and type checking, or the transpilation from TypeScript to JavaScript can be done with another transpiler, such as Babel, and TSC can be used only for checking types. Both ways have their benefits and drawbacks. If the TSC is integrated with the bundler, type checking is done at same time with the build. Also, using TSC for compilation means that all TSC features can be used, for example downleveling to target Ecmascript version. If another compiler than TSC is used to compile TypeScript, it depends on the used compiler which kind of features are available. If the used tools do not support type-checking during compilation, type checking must be done separately or rely on only the errors shown in IDEs. However, the building can be faster without type checking and other TSC features.

For example, for Webpack there is a third-party TypeScript loader, that can be used to compile TypeScript code to JavaScript with TSC as part of the bundling. TSC can handle normal Ecmascript syntax and features, but many web UI frameworks use also JSX syntax. TypeScript has a built-in JSX support for React and React Native. For other frameworks, for example Preact, the JSX transpilation process can be customized by using the jsxFactory, jsxFragmentFactory and jsxImportSource settings in the TypeScript configuration. The settings change how the JSX syntax is transpiled from JSX to function calls. If the jsxFactory, jsxFragmentFactory and jsxImportSource settings do not offer enough customization for the framework, there is also a possibility to use a custom transformer with TSC, which can then transpile the JSX in a way that works for the framework. A custom transformer takes a syntax tree formed by the TSC as an input and can manipulate the nodes of the tree, for example, by replacing the nodes which have JSX syntax with the corresponding function calls of the framework. Custom transformers are still rather experimental because they are not directly supported in tsconfig. Custom transformers can be used only with another third-party package, for example Webpack or Rollup.

The fifth step is to configure a linter for TypeScript. The most popular linter for JavaScript is ESLint and it has a separate TypeScript parser to be used with TypeScript. It has the same linting rules as the default parser, but in addition it has rules for linting TypeScript and its features. Even though linter is not obligatory, it is highly recommended as it helps to keep the code style unified among the code base and decreases the cognitive payload of a developer. When defining the linter rules, it is important to avoid too big changes in the code as it might increase the migration effort.

The sixth step is to configure the development tools and environments, such as code editors and IDEs, to be usable with TypeScript. Depending on the tool, they might support TypeScript out of the box, or they may need an extension or plugin for it. For example, WebStorm IDE from JetBrains supports developing, running, and debugging TypeScript as it is. Visual Studio Code from Microsoft may need extensions to get the most benefit out of using TypeScript.

The seventh step is to generate types for any APIs the project is using. This step is optional. Schürmann et al. (2020) address the problem of HTTP-based APIs making the client sides prone to errors, because the data is transferred as plain text. Any changes in the API can have unpredictable effects on the client side. Generating the API types from the server-side code and using them in the client can help to control the API changes. Also validating the payloads in run-time can help to notice unpredictable behavior. This way any breaking changes in the API would be noticed during the development. The need for generating types from API depends on what kinds of APIs the project uses and should be considered for each migration separately.

The eight step is to investigate if there are suitable migration tools available which would automate the migration process and decrease the migration effort. This step is also optional. Migrating a large project might be overwhelming if the whole migration is done by hand. Conversions are often very repetitive, and therefore at least some of them could be automated. For example, Airbnb created a migration tool (Airbnb, 2022) to automate their migration process and later published it under MIT license for others to use (Shaw, 2021). In the case of Airbnb, without the automation tool, the whole migration was estimated to take five years, but after creating the migration tool and taking it into use, the migration took only few weeks.

The ninth step is to start the migration with the chosen strategy. If the migration was chosen to be done gradually, it is often best to start migrating the code which from other code depends on. That would be, for example, utility functions, helpers, and base classes. Code is easier to migrate when it depends only on other TypeScript code, not JavaScript. (Nahum, 2019)

6. TypeScript migration demonstration

The case company has a SaaS product which has two UIs, a web UI and a mobile application, which both use the same backend. The web UI shares code with the mobile application of the product as they both use Flux pattern ("Flux," 2022) as their architecture. The migration was originally planned to be executed first for the web UI only, but it was then expanded to also cover the shared code and therefore also the mobile application, as the migration was not practical without migrating the shared code also. The package diagram is shown in Figure 13. The web UI is 7 years old and was created to replace the old version of the web UI of the case product. There have been eight developers in total working with the web UI code base all time and three developers who have worked with the mobile application. Now there are four developers, three of whom have been working with the web UI since the beginning of the project and three developers working with the mobile application. The web UI consists of 4 800 files which have a total of 340 000 lines of JavaScript code, out of which 1 500 files and 60 000 lines of code are shared with the mobile application. The mobile application has 450 own files which have a total of 30 000 lines of code in addition to the shared code. When the new web UI was planned and designed, TypeScript was considered as a one option, but back then it was not mature enough to be used, so the web UI was decided to be developed with JavaScript instead.

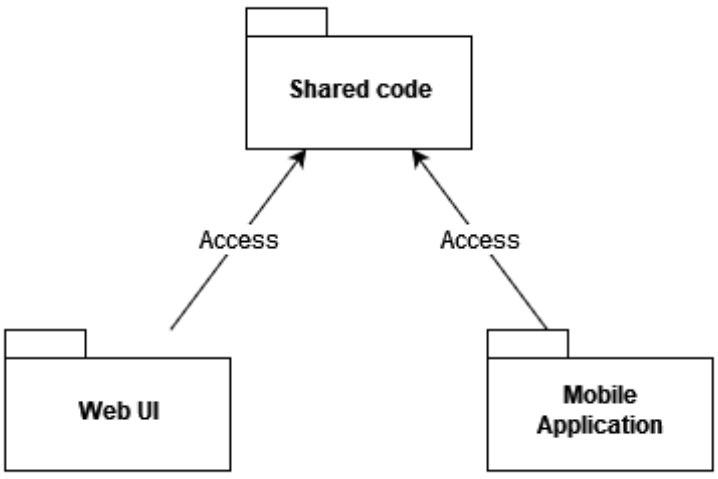

Figure 13. Case software package diagram.

6.1. Architecture of the case software

The web UI uses Inferno (“Inferno,” 2022b) for building the UI and the mobile application uses React Native. Inferno is a library for building user interfaces with JavaScript. It has a React-like API, but is faster than React (“Inferno,” 2022b; “UI Benchmark,” 2022). The web UI and the mobile application share the Flux architecture. Flux pattern includes stores, actions, views, and a one dispatcher. A store contains the application state and logic. A view represents a component that renders the UI with the data from a store. Views can communicate with stores via actions, which are dispatched to stores by the dispatcher. The web UI has two types of actions: server actions and view actions. A server action is triggered when the backend returns data for the client. A view action is usually triggered by the user, for example, by clicking a button in the UI. The Flux flow of the case software is shown in Figure 14. The code is managed in a monorepo, which contains all the code for both the web UI and the mobile application.

The web UI uses two bundlers: Webpack and Rollup. Webpack is used for development and Rollup for production build. In addition to them, the web UI also uses Babel for transpiling the JSX-syntax of Inferno and to downgrade newer Ecmascript syntax down to ES5 and ES6 to support older browsers. The web UI uses REST API to communicate with the backend.

The web UI has comprehensive automated unit testing. The unit testing is done with Jasmine and Karma, which means that the tests are ran in a real browser. The shared code is tested with Jasmine. The code coverage is approximately 97 %.

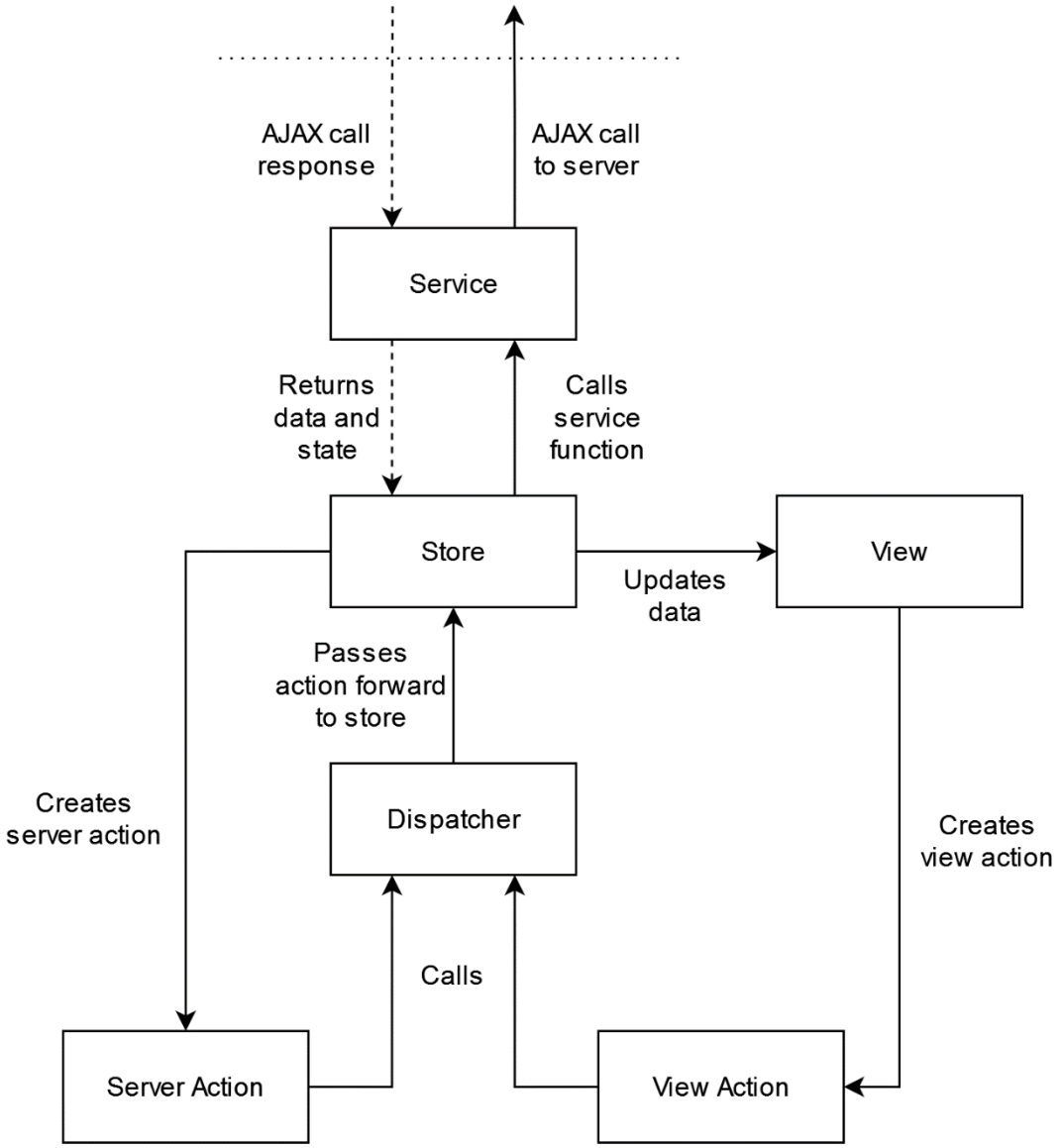

Figure 14. Flux architecture of the case software.

6.2. Migration plan

Following the migration model described in chapter 5, the migration plan for the case software was created. The migration was decided to be done gradually, because the architecture of the software and the Inferno framework caused uncertainty whether the migration would be beneficial or even possible. The planned steps and activities are:

1. Create a TSX transpiler for Inferno. This will determine if the migration is possible or not.
2. Configure TypeScript for monorepo.
3. Integrate TypeScript compilation within build and testing processes.
4. Check library support and add type packages if needed.
5. Configure linter for TypeScript.
6. Configure WebStorm for TypeScript.
7. Generate API types from OpenAPI Schema.
8. Search for migration tools.
9. Start the gradual migration.

6.3. Observations from the implementation

Following the migration plan described in chapter 6.2, the migration process was started by creating a custom TSX transformer for Inferno. Inferno has a React-like API, but in Inferno, the JSX can be transpiled into multiple different function calls, while in React the JSX is transpiled into only one function call. Because of this, the jsxFactory, jsxFragmentFactory and jsxImportSource settings in the TypeScript configuration could not solve the problem for Inferno. It was decided to go with the custom transformer instead of using the existing Babel plugin also for TypeScript, because then once the migration is finished, Babel could be removed from the build completely. An open-source implementation of a such transformer was found, but it needed to be updated and enhanced to be used in the web UI. The transformer uses TypeScript's inner API, which lacks documentation, which slowed down the work with the transformer. The owner of the original transformed did not respond when tried to be contacted, so the updated transformer was released as a new npm package called "ts-plugin-inferno" (Inferno, 2022a) under Inferno. It was also noticed, that the TSX transformer could possibly automatically optimize the Inferno code by using type information and adding the correct flags for elements, but this was not yet investigated any further. Updating the TSX transformer took approximately 120 hours of work and was the

most time-consuming part of the initialization of the migration. The custom TSX transformer will require time in the future as well since it must be kept up to date with the continuously evolving TypeScript.

The next step was to configure TypeScript for a monorepo. Originally the plan was to first use TypeScript only in the web UI, but because the web UI and the mobile application share such a big amount of essential code, it was found to be difficult to limit the migration only to the web UI. Therefore, the migration was initialized for both the web UI and for the mobile application. The mobile application uses React Native and the initialization for it took only few hours and was significantly easier to do than for the web UI. One tsconfig was created at the root of the monorepo and every other tsconfig that was created extended from it. Project-wide rules and settings were set in the root tsconfig, for example how strict type checking rules to use etcetera. Tsconfigs for the web UI, the mobile application and the packages under shared code, which all extended from the root tsconfig, were created. The tsconfigs created for the different parts of the monorepo had settings specific to them, for example how to handle JSX and how to reference to other packages inside the monorepo. This step was revisited a few times during the other steps when something needed to be adjusted in some of the tsconfigs. For example, at first the test folder containing all the test files did not have a separate tsconfig, but all the test files were included in the main tsconfig of the web UI. This caused slowness with webpack, as after every change it had to go through also every test file. The slowness issue was then resolved by excluding the test files from the main tsconfig of the web UI and a separate tsconfig was created for the test folder. After some time, it was discovered, that WebStorm could not use type information in JavaScript files, if they were not included in a tsconfig. To solve this, a separate tsconfig called tsconfig.build.json were created, which extended the existing tsconfig.json file, but included only files with “ts” and “tsx” file extensions. The existing tsconfigs were changed to include also files with “js” file extension. This removed the need for a separate tsconfig for tests, as the test files were now also included in the tsconfig.json. The hierarchy of the TypeScript configurations is shown in Figure 15. TypeScript configuration hierarchy.. It was difficult to separate this step and the next step, so the time used for them is combined.

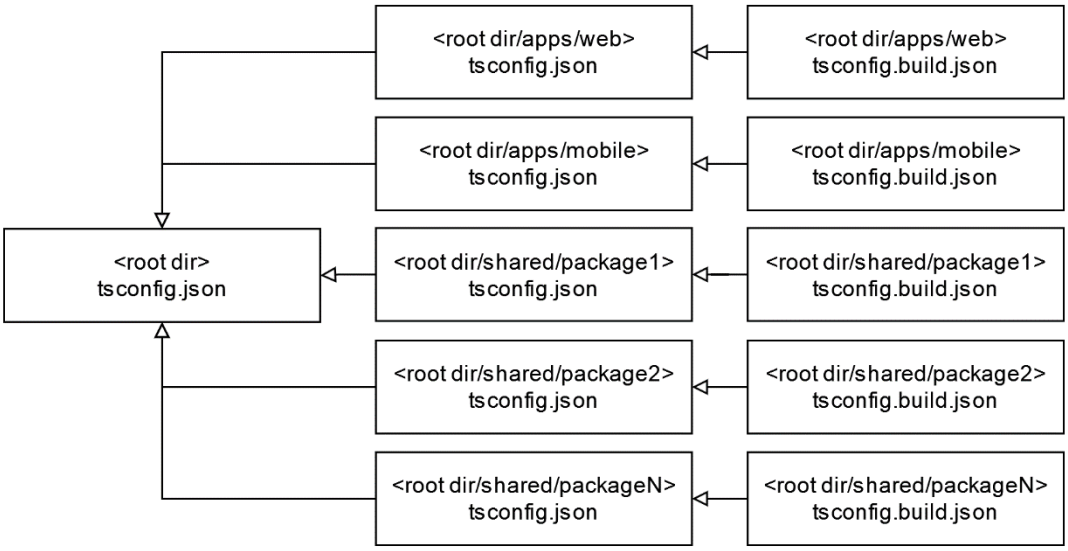

Figure 15. TypeScript configuration hierarchy.

The third step was to integrate TypeScript compilation with build and testing processes. Because React Native had a built-in support for TypeScript, there was no need to add any other additional configuration for the mobile application than a Babel plugin named "@babel/plugin-transform-typescript" to transpile TypeScript to JavaScript and a command to run type checking. The web UI could have been configured similarly, but there the goal was to eventually remove Babel completely and have one dependency less. In the web UI, the TSC needed to be added to multiple different places. The web UI uses Webpack for development, Rollup for production build, Jasmine and Karma for running tests for the web UI and Jasmine for running tests for the shared code. All these places needed to be configurated to handle TypeScript correctly. For Webpack a npm package called "ts-loader" needed to be installed and used to load TypeScript files with Webpack. Also, the custom TSX transformer created at the beginning of migration process was taken into use. For Rollup configuration, the process was similar, but instead of ts-loader, an npm package named "rollup-plugin-typescript2" was installed and used with the TSX transformer. For web UI tests, the same configuration as for Webpack was needed to do as Karma uses also Webpack. For the tests of the shared code, the same Babel plugin as used in the mobile application was installed. The shared code does not contain any JSX, so there was no need for TSX transformer. The configurations did not take that much time, but the challenge was to ensure that they worked in the expected way and would not cause any unexpected problems in the production. Configuring TypeScript and integrating compilation with build

tools took approximately 60 hours in total. That number includes all the work done after the initialization to enhance the configurations as problems were discovered.

The web UI did not have file extensions in the import statements, and all files, even the files which contained JSX, had a file extension “js”. This was designed to decrease the amount of reads from the disk that Webpack would do. During the migration, there would be a mix of at least two file extensions, “js” and “ts”, and possibly also “tsx”. Because of this, it was decided to add file extensions to import statements in the whole code base. WebStorm was set to use file extensions in JavaScript code. Soon after this was done, some drawbacks of the file extensions were discovered. WebStorm has implemented the feature so, that it uses always the “js” file extension in import statements, even when the file has a different file extension (JetBrains, 2022). This means that the file extensions needed to be corrected by hand, which was found to be annoying by the developers. Another thing was, that now converting a file from JavaScript to TypeScript caused the pull request to grow to be thousands of files, because in every file that depended on the converted file, the import statement was changed. This made the pull requests almost impossible to review, even if the author of the pull request wrote down the places which had actual changes. GitHub turned very slow with thousands of changed files and the reviewer was forced to trust the author of the pull request about what all had changed. A process to ease out the review process was created. The new process for converting a file from JavaScript to TypeScript consist of two separate parts: first, the file to be converted is renamed and at the beginning of the renamed file is added a “@ts-nocheck” comment. Then, a first pull request is made from these changes. After the pull request containing only the file rename is approved and merged, the “@ts-nocheck” comment is removed, and types are added to the converted file. Then another pull request is made with the actual changes. The second pull request should now have only relevant changes and no changed imports statements. This process was found to be slower and to require more work. It was also found to be error prone, as the rename change did not affect on branches, which were created before the rename change was merged to master branch and caused the new imports to have an incorrect file extension in imports, which then caused the software not to work. The CI/CD (Continuous Integration and Continuous Deployment) pipeline was then enhanced to have a separate type checking step before deployment to avoid deploying a broken product. At the same time with the migration, also

a need to migrate from the used module format, CommonJS, to the new modern format, ESM (Ecmascript Modules), was identified, as some of the used packages could not be updated if the project was using CommonJS module format. Once the project is migrated to ESM, all the imports will have “js” file extension, which will then decrease the migration effort, as the pull requests would not grow with every renamed file and is also the behavior that WebStorm supports. It was noticed that the used TypeScript plugin for Rollup did not support ESM and blocked the ESM migration. The struggle with the imports and file extensions showed, that there are two sufficient options: either have no file extensions in the imports statements or have “js” file extensions in every import statement, and which one to use depends on which module system the project is using. The migration could have been done without changing the file extensions, so the time used for that is excluded when measuring how much time the TypeScript migration takes.

The fourth step was to check which packages support TypeScript, which need to have a separate type package installed and which do not have type definitions available at all. In the web UI most of the packages came with built-in type definitions and few packages needed a separate type package to be installed. There were no packages that would have completely lacked type definitions. This step took two hours to complete.

The fifth step was to setup a linter for TypeScript. The web UI already used ESLint for linting. ESLint can be used to lint also TypeScript by changing it to use @typescript-eslint/parser as its parser. The ESLint configuration of the code base was changed by overriding the rules for files with “ts”, “tsx” and “d.ts” extension. These files were set to be parsed with a parser called “@typescript-eslint/parser”. The project was set up for the parser so, that the parser can use type information. There was a huge amount of new linting rules for TypeScript. It took a relatively long time to go through all the rules and investigate, whether they would be beneficial in the code base of the case software, as the linting rules are partly shared with both web UI and mobile application. The rules could be separated into two classes: pure style rules and other rules. Style rules would only affect code style things, such as usage of white spaces, delimiters, and empty lines. Other rules could have an impact on the code and its semantics, for example preferring one method over the other. It was

decided to take into use most of the recommended rules and rules that were automatically fixable were favored as they decrease the migration effort. Style rules were used to keep the TypeScript code style consistent with the JavaScript code and other rules to enforce usage of modern syntax and TypeScript features. There were also rules to order methods and properties in classes, which has so far been done by hand in the code base. It was discussed to be nice to have, but dropped out of the scope of this thesis, as it would require more investigation work and would possibly increase the migration effort. Some rules could not be used, because of the gradual migration would have caused them to give errors that could not have been fixed. Going through all the possible linting rules and figuring out which to use took approximately 50 hours of work. An option to decrease the time used for this would have been to use a preselected collection of rules instead of adjusting every single rule by hand. But because the project already had a comprehensive amount of linting rules, the new ones needed to be adjusted to fit with the existing ones.

The sixth step was to configure WebStorm for TypeScript. This was already done gradually during the previous steps. WebStorm has code style settings and those were synchronized to be same for both the JavaScript for TypeScript code. WebStorm needed a LSP (Language Service Protocol) plugin to be installed for its auto-import tool to work correctly between JavaScript and TypeScript files. The color schema of the editor was found to not highlight types or interfaces in any way. The color scheme was edited so, that the types and interfaces can be easily recognized from other code. WebStorm assumed that the project was using React, and some settings needed to be adjusted to get WebStorm offer auto-import option for Inferno components. This step took only few hours, as most of the work was done simultaneously with the previous steps.

The seventh step was to generate API types from OpenAPI Schema. The backend of the case product produces an API documentation in the form of OpenAPI Schema. There are several third-party open source type generator packages available, which will parse the OpenAPI Schema and output TypeScript types based on it. Multiple packages were tested, but none of them seemed to fit perfectly with the case software. Most of the packages outputted a singular file, which would cause any changes in the models in the future to be difficult to

see. Having each model in a separate file became a requirement for the generator. Also, it was thought that the best suitable outcome for the case software, which uses the Flux architecture which has a very clear data flow, would be gained by implementing an own generator. That however requires time and other resources, so it was not done within the scope of this thesis. Another question that was left open is that how to integrate the type generation process with the agile development process. Now both the backend and frontend could be developed individually or simultaneously. If the frontend becomes dependent on the generated types, it might force the backend to be developed first and the frontend only after the backend is ready. Besides of the open questions, it was seen that the generated types would clearly benefit the case software, as they would provide an easier way of checking the data structures and to have compile-time errors if the code does not comply with any model. Approximately 10 hours were used in this step before deciding to move on without a type generator, as it was clearly going to need a significant amount of time to be created.

The eighth step was to search for migration tools. Only one migration tool that could work was found and it was a package called “ts-migrate” made by Airbnb. It migrates React codebases from JavaScript to TypeScript and is open source. It seemed to remove some manual work but did not completely suit for the case software. The web UI, as mentioned before, used only “js” extension in its files, even when the file contained JSX. A common practice is to use “jsx” file extension for files that contain JSX, and this was a constraint for the ts-migrate. The ts-migrate tool did not migrate the JSX of the web UI correctly, because it did not expect there to be JSX. This decreased the usability of the tool for the web UI. A tool made specifically for the case software could decrease the migration effort, because there is a good amount of manual work when converting files. Approximately 20 hours were used to search for and test migration tools before deciding that they did not suit for the case software.

The ninth step was to start the gradual migration. Few files were converted during the previous steps to validate that the steps were completed properly, for example that the build process worked with TypeScript files. The converted files were small utility files. The migration was started from the shared code, as a great part of the other code was dependent

on the shared code. The code of the case software is based heavily on classes and inheritance. For example, the code base has a base service class, from which every other service class is derived. The derived service classes have a more specific purpose and can override the methods of the base class. While migrating these kinds of classes, it was discovered that the code did not follow the convention of keeping the method signature and its types unchanged when overriding a method of a parent class. TypeScript prevents changing the number and type of parameters and the return type of a method when overriding it. It was found that the code of the case software needed refactor to be able to utilize TypeScript. The wrongly overridden methods need to be renamed, so that they no longer override the method of the parent class, when the method signature changes. This applies for all the structures using classes.

There was also found to be another reason for refactoring the code. The service classes were originally implemented as singletons, but then changed to be classes with only static methods, because the IDE in use did not understand the singleton pattern and lost some code editing features. Now with TypeScript, the static methods prevented using generic types efficiently, so it was decided to refactor the services classes to be singletons again instead of using static methods. With TypeScript the IDE do not lose any features while using the singleton pattern. There were approximately 500 service classes, so it was decided to create a new base class called “Api” which would use a singleton pattern, and then while migrating files to TypeScript, simultaneously migrate every other service to extend from the new Api class instead of the base service class. The need to refactor the code increases the resources and time needed to execute the migration, but it also simultaneously enhances the code quality.

Otherwise, the code seemed to be able to be migrated into TypeScript. Most of the code did not need any changes other than adding types, but in some cases the code needed to be changed to provide better understanding for the TSC and to prevent losing type information. In some cases, this meant adding more variables and condition statement, which was the opposite of expected optimization by removing unnecessary checks. On the other hand, using constant enums instead of storing strings in objects can provide optimization with the

JIT (Just In Time Compiler). Constant enums are replaced with inline values when TypeScript is compiled to JavaScript, and JIT can then replace functions called with inline values with the return value and reduce the amount of function calls. This is shown in Figure 16, Figure 17 and Figure 18. Finally the function call mapToString(1) is replaced with string "second" by JIT. However, it might not be a trivial task to change all places to use constant enums and might require a significant amount of refactoring.

```ts
// values.ts
export const enum Values {
    Option1,
    Option2,
    Option3
}

// mapToString.ts
export function mapToString(value: Values) {
    if (value === Values.Option1) {
        return 'first';
    } else if (value === Values.Option2) {
        return 'second';
    }
    return 'rest';
}

// otherfile.ts
mapToString(Values.Option2);
```

Figure 16. Example of const enum usage.

```ts
// mapToString.ts
export function mapToString(value) {
    if (value === 0 /* Values.Option1 */) {
        return 'first';
    }
    else if (value === 1 /* Values.Option2 */) {
        return 'second';
    }
    return 'rest';
}
// otherfile.ts
mapToString(1 /* Values.Option2 */);
```

Figure 17. Example of const enum usage compiled to JavaScript.

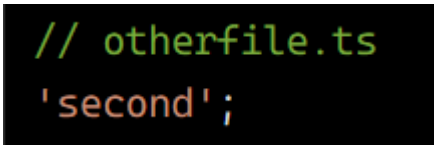

Figure 18. Example of const enum usage after JIT optimization.

In some cases, the migration revealed a possibility of bugs, when the code lacked necessary checks that, for example, an object is not null before trying to read its property. When converting the code from JavaScript to TypeScript, TSC found a method of a class that could have caused a run-time error, if it was called when the state of the class was undefined. However, it was decided to be a design decision, that the method should not be called at other than correct times, and if method is called incorrectly, it should result in run-time error. There were also other possible bugs revealed by TSC, which were then refactored, so the number of bugs was possibly decreased.

Having types in the code can help to understand the code, because the types will make it clear, which kind of objects, arrays or function can be given as a parameter to a function. Previously it was sometimes quite challenging to know when a parameter was named as "obj" and there was no additional information about what the "obj" can be. So, types help to document the code and its usage. Because most of the developers working with the case software has been working with it since the beginning of its development, they have already a good general knowledge how the code works, but, for example, for a new recruited employee the code can be difficult to work with.

The used IDE, WebStorm, had both improvements and impairments after the case software was started to be migrated to TypeScript. When working with only TypeScript files, the auto-complete feature of the IDE was improved. For example, previously WebStorm could not suggest the properties of an object, but with TypeScript it was able to do so. For example, in case software the actions are generated with a utility function. Before the migration was started, the IDE could not suggest anything relevant when accessing the properties of an action object. In this example, the exampleActionTypes is an object, which has properties

containing strings. The strings are used to identify actions. The auto-complete feature while using only JavaScript is shown in Figure 19.

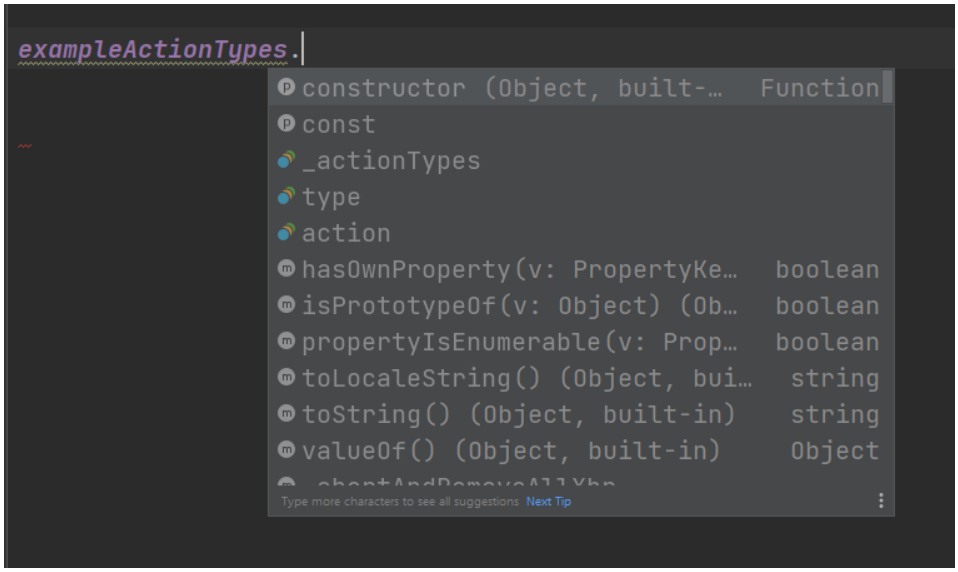

Figure 19. The auto-complete feature while using only JavaScript.

Just converting the function used to generate action types to TypeScript but leaving rest of the code to still be JavaScript gives already improvements. Now the IDE can suggest the fields, but not yet optimally. This is shown in Figure 20. When also the file where the exampleActionTypes object is created is converted to TypeScript, the auto-complete feature can order the properties correctly. This is shown in Figure 21.

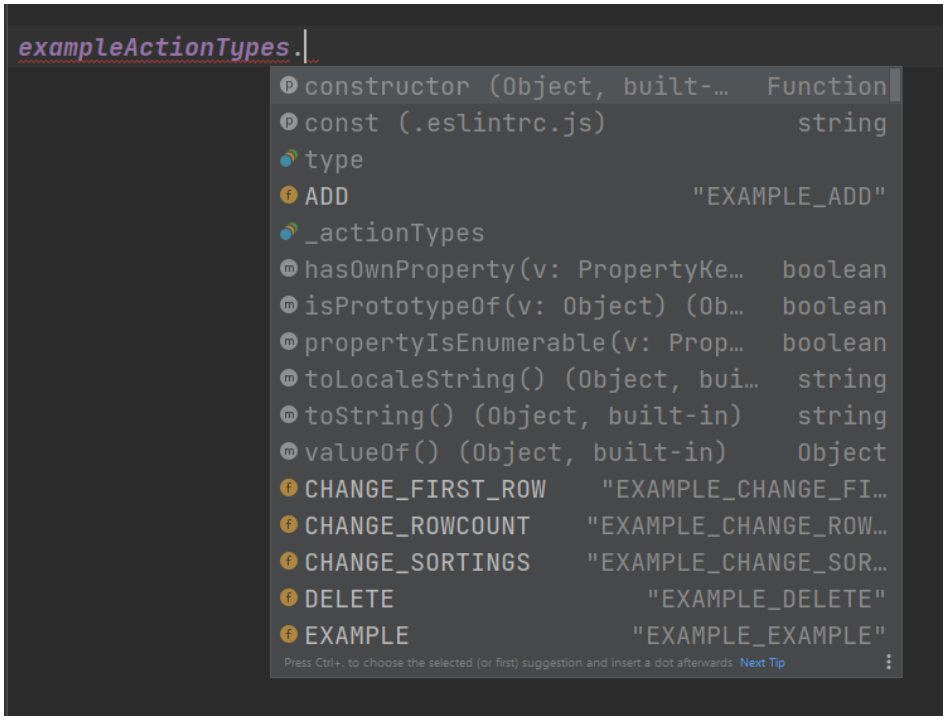

Figure 20. The auto-complete feature while using TypeScript function in JavaScript.

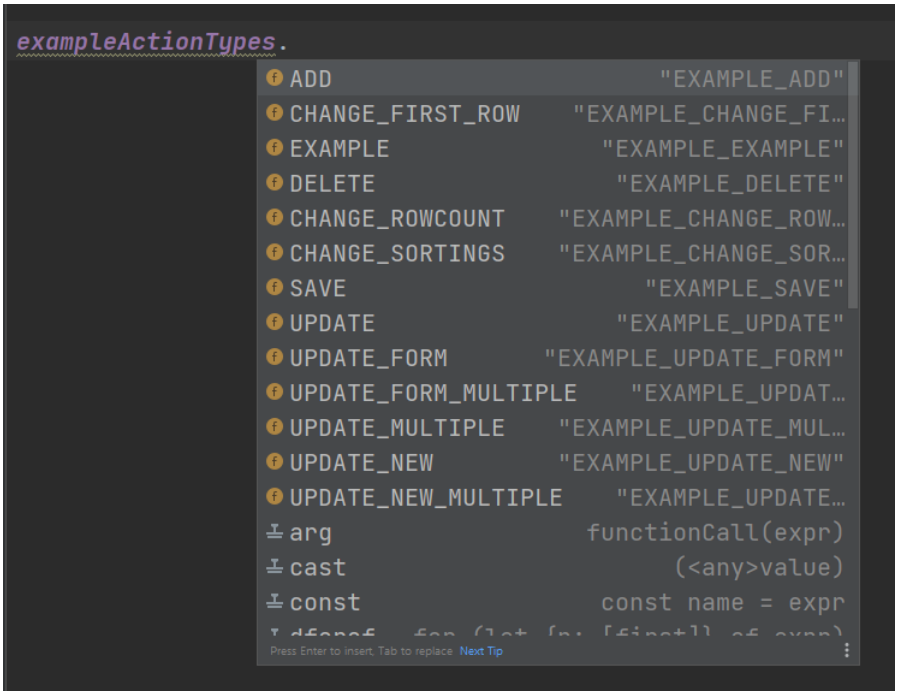

Figure 21. The auto-complete feature while using only TypeScript.

Another improvement in the IDE was that when it auto-completed a JSX component, it previously did not auto-complete any props to be given. Now, in a TypeScript file, the IDE auto-completes all the required props, which was found to be a greatly liked feature among the developers, because it reduced the amount of code the developer needed to write manually.

When using TypeScript, the IDE also shows the TSC errors which can help to notice problems with the code and identify possible bugs more easily. However, there were some situations, where the IDE did not show every TSC error that using TSC CLI (Command-line Interface) gave. This means, that the IDE alone is not enough, but at least before making a pull request, the types need to be checked with the command line command.

The gradual migration was started from code that a lot of code was dependent on, for example utility functions, such as isNullOrUndefined, and the Flux framework. At first, the migration was done so that TypeScript files would not import JavaScript code, but to only use other TypeScript code. This method worked very well. However, it restricted where TypeScript could be used and tested. One developer then tested to create a new feature with TypeScript. It was possible, but there were multiple problems. Some parts of the code had to rely on JavaScript code, which means that types were lacking and therefore the benefits of type checking were lost for that part of the code. Also, when a JSX component was imported from JavaScript file to be used in TypeScript file, TSC interpret that every prop needs to be given, even though some of the props were meant to be optional.

For the mobile application, initializing the migration was significantly easier than for the web UI. The main difference between them is that the mobile application uses React Native, which is rather popular and widely used framework, while the web UI uses Inferno, which is not widely used and therefore lacking an active ecosystem around it, at least compared to React Native or React. The web UI required a custom TSX transformer to be created, and the settings of an IDE to be changed for the IDE to work. TypeScript includes in its source code a TSX transformer for React, but not for any other framework. For every other framework using TypeScript is more difficult. Also, the used IDE, WebStorm, assumed that

React is used, and it required some work to figure out how to disable React related assumptions and get WebStorm to work with Inferno.

The distribution of time used on different steps to initialize the migration is shown in Table 4. Distribution of time usage. Then in addition to the time used to initialize the migration, a significant amount of time was spent to execute the migration and convert files so that it could be evaluated how the migration affects the DX.

Table 4. Distribution of time usage.

Step	**Time spent (h)**	**Time spent out of total time (%)**
Create a TSX transpiler for Inferno	120	46 %
Configure TypeScript for monorepo and integrate TypeScript compilation with build and testing processes	60	23 %
Configure linter for TypeScript	50	19 %
Search for migration tools	20	8 %
Generate API types from OpenAPI Schema	10	4 %
Check library support and add type packages if needed	2	1 %
Configure WebStorm for TypeScript	2	1 %

7. Discussion

In this chapter the observations made during the migration are discussed. First, the results are compared against the objectives described in chapter 4, then the workload, benefits and drawbacks of the migration is estimated, and finally it is discussed, how the migration model could be improved in the future.

7.1. Comparing objectives to results

The migration had four main objectives, which were described in chapter 4. The first objective was to improve DX. It was assumed to be achieved with better documentation, better tooling, enhancing the understandability of the code and having more consistent code structure. Because this thesis does not cover the whole migration process, but only the initialization and a proof-of-concept migration, it is difficult to evaluate how it affected the overall DX. However, the individual factors listed above can be evaluated separately. As mentioned in the chapter 6.3, having types in the code acted as a form of documentation, and helped to understand the code and how it is supposed to be used, so it could be said that TypeScript enhanced the documentation of the code. Also, the features of the used IDE were improved, as shown in chapter 6.3, so it could be said that there will be better tooling when using TypeScript. The tools were improved also for JavaScript files when another part of the code was migrated to TypeScript. As mentioned in chapter 6.3, changing to use TypeScript requires some refactoring to be done, which can help to have a more consistent code structure, when the code is following good practices everywhere. Evaluating how using TypeScript affects the understandability of the code is a more difficult task, because even though the types document the code, it depends on the developer that how well they can read the types, especially more complex ones. With complex types and unexperienced developer, having the types might even make it more difficult to understand the code. Using TypeScript also increases the amount of code, which can reduce its readability. On the other hand, simple types describe what kind of data is stored in a variable with a generic name. With all these four factors combined, it seems that the overall effect is rather positive than negative, so it could be said that migrating to TypeScript improves the DX. However, it was also noted that

at the beginning of the migration the DX was probably slightly decreased due to the temporary problems that occurred during the initialization, for example the build time increasing significantly at first, which annoyed the developers, but which was then decreased back to tolerable level after the configurations were fixed. The migration encourages the code to be refactored to follow good practices and have a consistent structure, which helps to understand the code and therefore improves DX.

The second objective was to decrease the number of bugs. It was assumed to be achieved by being able to detect type related bugs in the compilation time. The proof-of-concept portion of the migration did not reveal a great amount of type related bugs, but it showed that calling methods of a class could result in error if the call was done at the wrong state of the class. On the other hand, using JavaScript and TypeScript at the same time makes the code more prone to type errors. At this state of the migration, it cannot be said if it decreases the number of bugs in the case software or not.

The third objective was to keep both the software and the developers up to date with modern technologies. This was assumed to be achieved by enabling the use of the newest Ecmascript features, keeping the software compatible with future technologies, and keeping the skills of developers up to date with modern technologies. The migration did enable the use of most of the newest Ecmascript features, but some features cannot be used if the compilation target is too low. The case software can now be migrated completely to TypeScript, and if TypeScript is ever a requirement for any new technology, the case software can use it. The migration also, depending on the perspective of a developer, either offered an opportunity or forced to learn a new modern technology. All developers have started to learn TypeScript and get familiar with it, so it could be said that both the software and the developers are kept up to date with modern technologies. On the other hand, when TypeScript was added to the case software, it required multiple third-party packages to be added. Having more packages means that there are more dependencies that need to be kept up to date.

The fourth objective was to optimize code. This was assumed to be achieved by having less unnecessary conditionals and using constant enums. In the proof-of-concept migration there

was not found any unnecessary conditionals, that could have been removed, so it does not offer any optimization at this point of the migration. Using constant enums instead looked very promising. The only question is that how much the case software can utilize constant enums without a great refactor. So, it could be said that the migration has potential to optimize the code, but it needs to be researched more.

7.2. Estimated workload

The workload of migrating a large software to TypeScript is big. However, because the case software consists of two UIs, the web UI and the mobile application, the migrations of those two were compared in chapter 6.3. The migration was significantly more time consuming to initialize for the web UI than it was for the mobile application. There are multiple possible reasons for why. It seems that the migration is easier to initialize for a project using React or another popular framework, than for a project using a less popular framework. For a popular framework there is a greater chance that many tools already exist, while for a less popular framework everything must be created from the scratch.

Other factors that seemed to increase the workload were also identified. The case software had relatively little number of third-party packages in use, which also meant that the Flux framework was custom made for the case software, and the source code was part of the case software and therefore plain JavaScript. Most of npm packages have type definitions available, but in this case the Flux framework did not, which meant that also it had to be migrated.

The web UI had significantly more complex tools for building and testing the application than the mobile application had. The web UI used Babel and both Webpack and Rollup, while the mobile application used only Babel. This meant that for the web UI, TypeScript needed to be integrated with multiple different tools, while for mobile adding one plugin to babel was enough. The mobile application is newer and smaller than the web UI, which might also explain the difference.

The source code of the case software was mostly compatible with TypeScript, but few structures were identified that needed to be refactored. The need for refactoring increased the migration effort and prevented automating the migration. If all structures would have followed the best practices already, the workload would have been smaller. For example, one best practice that was not being followed was that the method signature of a method of a class should not be changed when the method is overridden. Also, even though both the web UI and the mobile application both used already Babel, they did not use the most modern syntax everywhere. Refactoring older syntax to be newer and more readable does not increase the workload as much other factors, but still has an effect.

Factors which possibly decrease the migration workload were also identified. The opposite of many factors that increase the workload would probably decrease it: usage of typed third-party packages, modern syntax, and popular frameworks. Other factors that were identified to decrease the workload were having a comprehensive automated testing and a robust CI/CD pipeline, which both help to ensure, that no regression is caused by migrating a part of the code to TypeScript. Otherwise, a lot more manual testing would be needed, and manual testing is known to be both slow and error prone.

It seems like the workload depends heavily on the characteristics of the project to be migrated. The initialization of the migration for the web UI took approximately 274 hours, while for the mobile application it took only few hours.

7.3. Estimated benefits and drawbacks of the migration

Based on the demonstration, it seems that the migration would have both benefits and drawbacks. The benefits include the previously mentioned improved DX, which includes better documentation and better tooling, and the software possibly having less bugs, and options to optimize the code. These benefits were partially gained even when only a small part of the source code was migrated. So even a partial migration has benefits and more value is created from each migrated file. Having a better documentation might also help to

onboard new employees, because it might make it easier to understand the code and how to use it. The migration encourages the code to be refactored to follow good coding practices, which improves the code structure and quality.

There were also drawbacks from the migration. One drawback is the already mentioned possible need for refactoring, which takes time. The migration can require a lot of resources, especially the time of the developers. When developers need to use time for the migration, it means that there is less time for other activities, for example creating new functionality. The DX might also decrease at the beginning, if the initialization of the migration causes problems, or if the developers resist the change. Also, because TypeScript is compiled to JavaScript and the types are checked on build-time, the built time increases, which can affect the development and publishing speed.

7.4. Improvement suggestions based on demonstration

The migration model described in chapter 5 seemed to work relatively well for the case software, but a few improvements were identified. The steps to initialize the migration that were described in chapter 5.2 could be renamed as tasks or something, which would not have an order. For example, in the demonstration the order of the steps was drastically modified, but it did not affect the results of the initialization. The steps can also be executed simultaneously, which also suggest that they should not have an order.

With the current information available, it can be difficult to evaluate if a software can be migrated to TypeScript, how much work does it require and what kind of benefits it will offer. The official documentation does not offer any list of factors that might prevent the migration, while in reality there are multiple situations where the migration might not be profitable or even possible. For example, if the migration requires most of the source code to be refactored by hand, the migration might not be profitable, or if some used technology or framework is not compatible with TypeScript, the migration is not possible. Research to identify situations where the migration should not be executed could be done. Also, the

migration model presented in this thesis could be tested in other contexts and based on the result be improved and generalized.

8. Conclusions

The objective of this thesis was to investigate, how a large JavaScript web UI can be migrated to TypeScript, what benefits and drawbacks the migration has and how it affects the Developer Experience of the developers working with the migrated software. Previous research has compared dynamically and statically typed programming languages. Statically typed programming languages have been found to be better in many ways compared to dynamically typed programming languages. Statically typed programming languages have better documentation, tooling, and maintainability, and also greater development speed in large projects, and less bugs. Previous research shows that happy developers are more productive and produce higher quality than unhappy developers. Many of the advantages of statically typed languages are found to increase happiness of developers, for example better documentation and IDE tooling.

The migration model created in this thesis answer to the first research question about how a TypeScript migration can be initialized for a large web UI. The migration model lists various tasks that need to be done before the migration can be started. The second research question was about which are the benefits and drawbacks of a TypeScript migration. The main benefits of the migration that were observed were improved DX, improved code structure, keeping the software and developers up to date, possibility of having less bugs and options to optimize the code. The main drawbacks that were observed were that the migration requires time, it requires to add third-party dependencies and increasing build time. The characteristics of the software and frameworks used in the software seem to affect highly on how time-consuming the process will be.

The third research question was how a TypeScript migration affects Developer Experience. The TypeScript migration seems to improve DX by embedding documentation in the code, which helps to understand the code and how to use it easier. TypeScript seems also to enhance the features of an IDE by improving its auto-complete capabilities. The migration

also forces the code to follow good practices and have a consistent structure, which helps to understand the code and improves DX.

The migration model can be improved by testing it in various contexts, and based on the results, enhancing the migration model. There is also a need to research possible obstacles that would prevent migrating a software to TypeScript.

References

Abrams, D., Hogg, M., 1999. Social Identity and Social Cognition.

Airbnb, 2022. ts-migrate [WWW Document]. URL https://github.com/airbnb/ts-migrate (accessed 8.24.22).

Announcing TypeScript 1.0 [WWW Document], 2014. . TypeScript. URL https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/ (accessed 6.13.22).

Ecma International, 2006. Ecmascript 4 Language Specification.

Endrikat, S., Hanenberg, S., Robbes, R., Stefik, A., 2014. How do API documentation and static typing affect API usability?, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014. Association for Computing Machinery, New York, NY, USA, pp. 632–642. https://doi.org/10.1145/2568225.2568299

Fagerholm, F., 2015. Software Developer Experience : Case Studies in Lean-Agile and Open Source Environments. Kehittäjäkokemus : Tapaustutkimuksia virtaviivaisessa ja ketterässä sekä avoimen lähdekoodin ohjelmistokehityksessä.

Fagerholm, F., Münch, J., 2012. Developer experience: Concept and definition. Presented at the 2012 International Conference on Software and System Process, ICSSP 2012 - Proceedings, pp. 73–77. https://doi.org/10.1109/ICSSP.2012.6225984

Fischer, L., Hanenberg, S., 2015. An empirical investigation of the effects of type systems and code completion on API usability using TypeScript and JavaScript in MS visual studio, in: Proceedings of the 11th Symposium on Dynamic Languages. Presented at the SPLASH '15: Conference on Systems, Programming, Languages, and Applications: Software for Humanity, ACM, Pittsburgh PA USA, pp. 154–167. https://doi.org/10.1145/2816707.2816720

Flux [WWW Document], 2022. URL https://facebook.github.io/flux/ (accessed 8.29.22).

Gao, Z., Bird, C., Barr, E.T., 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Presented at the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 758–769. https://doi.org/10.1109/ICSE.2017.75

Girardi, D., Novielli, N., Fucci, D., Lanubile, F., 2020. Recognizing developers' emotions while programming, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20. Association for Computing Machinery, New York, NY, USA, pp. 666–677. https://doi.org/10.1145/3377811.3380374

Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P., 2018. What happens when software developers are (un)happy. Journal of Systems and Software 140, 32–47. https://doi.org/10.1016/j.jss.2018.02.041

Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P., 2017a. Unhappy Developers: Bad for Themselves, Bad for Process, and Bad for Software Product, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). Presented at the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 362–364. https://doi.org/10.1109/ICSE-C.2017.104

Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P., 2017b. Consequences of Unhappiness while Developing Software, in: 2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion). Presented at the 2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion), pp. 42–47. https://doi.org/10.1109/SEmotion.2017.5

Graziotin, D., Wang, X., Abrahamsson, P., 2015a. Do feelings matter? On the correlation of affects and the self-assessed productivity in software engineering. Journal of Software: Evolution and Process 27, 467–487. https://doi.org/10.1002/smr.1673

Graziotin, D., Wang, X., Abrahamsson, P., 2015b. Understanding the affect of developers: theoretical background and guidelines for psychoempirical software engineering, in: Proceedings of the 7th International Workshop on Social Software Engineering, SSE 2015. Association for Computing Machinery, New York, NY, USA, pp. 25–32. https://doi.org/10.1145/2804381.2804386

Graziotin, D., Wang, X., Abrahamsson, P., 2015c. The Affect of Software Developers: Common Misconceptions and Measurements, in: 2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering. Presented at the 2015 IEEE/ACM 8th International Workshop on

Cooperative and Human Aspects of Software Engineering, pp. 123–124. https://doi.org/10.1109/CHASE.2015.23

Graziotin, D., Wang, X., Abrahamsson, P., 2015d. How do you feel, developer? An explanatory theory of the impact of affects on programming performance. PeerJ Comput. Sci. 1, e18. https://doi.org/10.7717/peerj-cs.18

Graziotin, D., Wang, X., Abrahamsson, P., 2014a. Happy software developers solve problems better: psychological measurements in empirical software engineering. PeerJ 2, e289. https://doi.org/10.7717/peerj.289

Graziotin, D., Wang, X., Abrahamsson, P., 2014b. Software Developers, Moods, Emotions, and Performance. IEEE Software 31, 24–27. https://doi.org/10.1109/MS.2014.94

Graziotin, D., Wang, X., Abrahamsson, P., 2013. Are Happy Developers More Productive?, in: Heidrich, J., Oivo, M., Jedlitschka, A., Baldassarre, M.T. (Eds.), Product-Focused Software Process Improvement, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 50–64. https://doi.org/10.1007/978-3-642-39259-7_7

Hall, T., Jagielska, D., Baddoo, N., 2007. Motivating developer performance to improve project outcomes in a high maturity organization. Software Qual J 15, 365–381. https://doi.org/10.1007/s11219-007-9028-1

Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, É., Stefik, A., 2014. An empirical study on the impact of static typing on software maintainability. Empir Software Eng 19, 1335–1382. https://doi.org/10.1007/s10664-013-9289-1

Hevner, A.R., March, S.T., Park, J., Ram, S., 2004. Design Science in Information Systems Research. MIS Quarterly 28, 75–105. https://doi.org/10.2307/25148625

Inferno, 2022a. ts-plugin-inferno [WWW Document]. npm. URL https://www.npmjs.com/package/ts-plugin-inferno (accessed 10.21.22).

Inferno [WWW Document], 2022b. . Inferno.js. URL https://www.infernojs.org/ (accessed 8.29.22).

International Organization for Standardization, 2019. ISO 9241-210:2019.

JetBrains, 2022. Auto import from TS file to JS file with Use file extension as “Always” adds wrong file extension to the import [WWW Document]. URL https://youtrack.jetbrains.com/issue/WEB-57147 (accessed 10.2.22).

Khadka, R., 2016. Revisiting Legacy Software System Modernization [WWW Document]. URL https://dspace.library.uu.nl/handle/1874/330662 (accessed 8.16.22).

Khan, U.F., Rana, T., 2021. Programmer Experience: – An HCI Prespective, in: 2021 International Conference on Communication Technologies (ComTech). Presented at the 2021 International Conference on Communication Technologies (ComTech), pp. 44–48. https://doi.org/10.1109/ComTech52583.2021.9616689

Lesiuk, T., 2005. The effect of music listening on work performance. Psychology of Music 33, 173–191. https://doi.org/10.1177/0305735605050650

Machiry, A., Kastner, J., McCutchen, M., Eline, A., Headley, K., Hicks, M., 2022. C to checked C by 3c. Proc. ACM Program. Lang. 6, 1–29. https://doi.org/10.1145/3527322

Mayer, C., Hanenberg, S., Robbes, R., Tanter, É., Stefik, A., 2012. An empirical study of the influence of static type systems on the usability of undocumented software. Presented at the Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, pp. 683–702. https://doi.org/10.1145/2384616.2384666

MDN, 2022. JavaScript [WWW Document]. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript (accessed 3.9.22).

Meyer, A.N., Barr, E.T., Bird, C., Zimmermann, T., 2021. Today Was a Good Day: The Daily Life of Software Developers. IEEE Transactions on Software Engineering 47, 863–880. https://doi.org/10.1109/TSE.2019.2904957

Microsoft, 2022a. TypeScript Handbook [WWW Document]. URL https://www.typescriptlang.org/assets/typescript-handbook.pdf (accessed 6.13.22).

Microsoft, 2022b. C# documentation [WWW Document]. URL https://docs.microsoft.com/en-us/dotnet/csharp/ (accessed 3.9.22).

Microsoft, 2022c. Search for typed packages [WWW Document]. URL https://www.typescriptlang.org/dt/search?search= (accessed 8.22.22).

Morales, J., Rusu, C., Botella, F., Quiñones, D., 2020a. Programmer eXperience: A Set of Heuristics for Programming Environments, in: Meiselwitz, G. (Ed.), Social Computing and Social Media. Participation, User Experience, Consumer Experience, and Applications of Social Computing, Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 205–216. https://doi.org/10.1007/978-3-030-49576-3_15

Morales, J., Rusu, C., Botella, F., Quinones, D., 2019. Programmer eXperience: A Systematic Literature Review. IEEE Access 7, 71079–71094. https://doi.org/10.1109/ACCESS.2019.2920124

Morales, J., Rusu, C., Quinones, D., 2020b. Programmer Experience: A Systematic Mapping. IEEE Latin America Transactions 18, 1111–1118. https://doi.org/10.1109/TLA.2020.9099749

Müller, S.C., Fritz, T., 2015. Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Presented at the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 688–699. https://doi.org/10.1109/ICSE.2015.334

Nahum, C., 2019. How to Inject TypeScript Into Existing Projects [WWW Document]. URL https://www.youtube.com/watch?v=-htA_n4P7gQ (accessed 8.17.22).

npm Inc., 2022. npm [WWW Document]. URL https://www.npmjs.com/ (accessed 8.22.22).

Okon, S., Hanenberg, S., 2016. Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? A controlled experiment, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). Presented at the 2016 IEEE 24th International Conference on Program Comprehension (ICPC), pp. 1–10. https://doi.org/10.1109/ICPC.2016.7503719

Oracle, 2022. Java Documentation [WWW Document]. Oracle Help Center. URL https://docs.oracle.com/en/java/ (accessed 3.9.22).

Pang, A., Anslow, C., Noble, J., 2018. What Programming Languages Do Developers Use? A Theory of Static vs Dynamic Language Choice, in: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Presented at the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 239–247. https://doi.org/10.1109/VLHCC.2018.8506534

Paulson, L.D., 2007. Developers shift to dynamic programming languages. Computer 40, 12–15. https://doi.org/10.1109/MC.2007.53

Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S., 2007. A Design Science Research Methodology for Information Systems Research. Journal of Management Information Systems 24, 45–77. https://doi.org/10.2753/MIS0742-1222240302

Petersen, P., Hanenberg, S., Robbes, R., 2014. An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. groovy with eclipse, in: Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014. Presented at the the 22nd International Conference, ACM Press, Hyderabad, India, pp. 212–222. https://doi.org/10.1145/2597008.2597152

Python, 2022. 3.10.2 Documentation [WWW Document]. URL https://docs.python.org/3/ (accessed 3.9.22).

Rauschmayer, A., 2020. Tackling TypeScript: Upgrading from JavaScript.

Ray, B., Posnett, D., Filkov, V., Devanbu, P., 2014. A large scale study of programming languages and code quality in github, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014. Association for Computing Machinery, New York, NY, USA, pp. 155–165. https://doi.org/10.1145/2635868.2635922

Releases · microsoft/TypeScript [WWW Document], 2022. . GitHub. URL https://github.com/microsoft/TypeScript/releases (accessed 4.26.22).

Schürmann, J., Tegeler, T., Steffen, B., 2020. Guaranteeing Type Consistency in Collective Adaptive Systems, in: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles, Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 311–328. https://doi.org/10.1007/978-3-030-61470-6_19

Schwaber, K., Sutherland, J., 2020. The Scrum Guide.

Shaw, E., 2021. Automatically Migrating to TypeScript with ts-migrate [WWW Document]. URL https://www.youtube.com/watch?v=y7WUsi6NeH8 (accessed 8.24.22).

Siek, J., Taha, W., 2007. Gradual Typing for Objects, in: Ernst, E. (Ed.), ECOOP 2007 – Object-Oriented Programming, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Siek, J.G., Taha, W., 2006. Gradual Typing for Functional Languages 12.

Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T., 2015. Refined Criteria for Gradual Typing 20 pages. https://doi.org/10.4230/LIPICS.SNAPL.2015.274

Stack Overflow, 2022. Stack Overflow Developer Survey 2022 [WWW Document]. Stack Overflow. URL https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022 (accessed 7.5.22).

Stack Overflow, 2021. Stack Overflow Developer Survey 2021 [WWW Document]. Stack Overflow. URL https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021 (accessed 2.23.22).

Stack Overflow, 2020. Stack Overflow Developer Survey 2020 [WWW Document]. Stack Overflow. URL https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020 (accessed 2.23.22).

Stack Overflow, 2019. Stack Overflow Developer Survey 2019 [WWW Document]. Stack Overflow. URL https://insights.stackoverflow.com/survey/2019/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2019 (accessed 2.23.22).

Stack Overflow, 2018. Stack Overflow Developer Survey 2018 [WWW Document]. Stack Overflow. URL https://insights.stackoverflow.com/survey/2018/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2018&utm_content=social-share (accessed 2.23.22).

Stack Overflow, 2017. Stack Overflow Developer Survey 2017 [WWW Document]. Stack Overflow. URL https://insights.stackoverflow.com/survey/2017/?utm_source=so-owned&utm_medium=social&utm_campaign=dev-survey-2017&utm_content=social-share (accessed 2.23.22).

Stack Overflow, 2016. Stack Overflow Developer Survey 2016 [WWW Document]. Stack Overflow. URL https://insights.stackoverflow.com/survey/2016 (accessed 2.23.22).

Stack Overflow, 2015. Stack Overflow Developer Survey 2015 [WWW Document]. Stack Overflow. URL https://insights.stackoverflow.com/survey/2015 (accessed 2.23.22).

Stuchlik, A., Hanenberg, S., 2011. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time, in: Proceedings of the 7th Symposium on Dynamic Languages, DLS '11. Association for Computing Machinery, New York, NY, USA, pp. 97–106. https://doi.org/10.1145/2047849.2047861

Tobin-Hochstadt, S., Felleisen, M., Findler, R.B., Flatt, M., Greenman, B., Kent, A.M., St-Amour, V., Strickland, T.S., Takikawa, A., 2017. Migratory typing: Ten years later. Presented at the Leibniz International Proceedings in Informatics, LIPIcs. https://doi.org/10.4230/LIPIcs.SNAPL.2017.17

TypeScript [WWW Document], 2022. URL https://www.typescriptlang.org/ (accessed 4.26.22).

UI Benchmark [WWW Document], 2022. URL https://localvoid.github.io/uibench/ (accessed 8.29.22).

Vann, D., 2019. How to Incrementally Migrate 100k Lines of Code to Typescript. URL https://dylanvann.com/incrementally-migrating-to-typescript (accessed 8.16.22).

Wolf, W., 1989. A practical comparison of two object-oriented languages. IEEE Software 6, 61–68. https://doi.org/10.1109/52.35590

Zelkowitz, M.V., 2009. Advances in computers. Academic, London.