TECHNISCHE UNIVERSITÄT
CHEMNITZ

Department of Computer Science
Distributed and Self-organizing Systems Group

Master's Thesis

Rapid Web Migration Prototyping using WebAssembly

Valentin Siegert

Chemnitz, 8 February 2018

Examiner: Prof. Dr.-Ing. Martin Gaedke
Supervisor: Sebastian Heil M.Sc.

Siegert, Valentin
Rapid Web Migration Prototyping using WebAssembly
Master's Thesis, Department of Computer Science
Technische Universität Chemnitz, February 2018

Acknowledgement

I want to thank all, who supported or motivated me during the preparation of this master's thesis, either as a counterpart in a technical discussion or as emotional support. Namely I want to designate:

Sebastian Heil

Stefan Heyder

Luis Moreno

Frank Siegel

Marian Thull

Sebastian Wolf

Abstract

Changing user expectations impose new requirements on existing software systems. The initial hurdle to transform existing software into web-based software, however, is high. In particular, Small and Medium (SME)-sized enterprises are reluctant to initiate this process. Reasons for this can be summarized in two categories: doubts about feasibility and doubts about utility. In the context of agile development, rapid prototyping has proven a suitable tool for addressing similar concerns. Therefore, this thesis investigates the idea of rapid migration prototyping, i.e. the quick creation of running web versions of existing legacy desktop applications with only limited resources.

Recent technologies such as WebAssembly (WASM) allow execution of legacy code in web browsers. The objective of this thesis is to employ Web Assembly for rapid web migration prototyping. To achieve this, the process of rapid web migration has to be defined in terms of activities, actors and artifacts. Possibilities of supporting these activities have to be identified and suitable supporting infrastructure needs to be created. Re-use of existing legacy application logic is important for rapid web migration prototyping. Challenges arise from supporting migration engineers in particular in two areas: communication with the server-side backend and interaction with DOM-based HTML user interfaces. Existing APIs in JavaScript and WASM have to be extended by suitable functionality to allow for easy creation of web-based prototypes.

The objective of this master thesis is to find an approach or a combination of approaches to solve the previously mentioned problem in the context WebAssembly and web migration. This particularly includes the state of the art regarding web migration and rapid prototyping. The demonstration of feasibility with an implementation prototype of the concept is part of this thesis as well as a suitable evaluation with exemplary use cases.

Contents

List of Figures iii

List of Tables v

List of Listings vii

List of Abbreviations ix

1. Introduction 1
- 1.1. Current Situation . . . 1
- 1.2. Motivation . . . 4
- 1.3. Problem Analysis . . . 6
- 1.4. Scenario . . . 7
- 1.5. Objectives and Limits . . . 8
- 1.6. Outline . . . 11

2. State of the Art 13
- 2.1. Requirements . . . 13
  - 2.1.1. User Requirements . . . 14
  - 2.1.2. Manager Requirements . . . 15
  - 2.1.3. Developer Requirements . . . 16
  - 2.1.4. Process-related Requirements . . . 18
- 2.2. Web Migration . . . 20
  - 2.2.1. Replacement . . . 21
  - 2.2.2. Re-installation . . . 22
  - 2.2.3. Redevelopment . . . 22
  - 2.2.4. Relocation . . . 23
  - 2.2.5. Further Notes about Web Migration . . . 28
- 2.3. Rapid Prototyping . . . 29
- 2.4. Rapid Web Migration Prototyping . . . 31

3. Concept 35
- 3.1. ReWaMP leveraging WASM . . . 35
- 3.2. ReWaMP Architecture . . . 38
- 3.3. ReWaMP Process . . . 40

3.4. Challenges . . . 44
3.4.1. WASM related Serialization . . . 44
3.4.2. Client/Server Communication . . . 45
3.4.3. Access HTML DOM with WASM code . . . 46
3.4.4. Handling UI events in WASM code . . . 46
3.5. Additional Notes according ReWaMP . . . 47
3.5.1. Conceptual Decisions using WASM . . . 47
3.5.2. Server-side Application . . . 52
3.5.3. Reconstructing GUI . . . 52

4. Implementation **55**
4.1. WASM-T Architecture . . . 55
4.2. C++ Parser . . . 57
4.2.1. Classes . . . 57
4.2.2. Functions . . . 60
4.3. ReWaMP Processor . . . 62
4.3.1. ReWaMP Process Algorithm . . . 62
4.3.2. Assumptions on Legacy Code and Environment . . . 68
4.3.3. WASM-T's JSON Serialization . . . 70
4.3.4. Export Functions to JS . . . 71
4.3.5. Import Functions from JS . . . 72

5. Evaluation **75**
5.1. Procedure . . . 75
5.1.1. Guidelines . . . 77
5.1.2. Measurements during the ReWaMP process . . . 77
5.1.3. Design of the Questionnaire . . . 79
5.2. Results . . . 80
5.2.1. ReWaMP Measurements Results . . . 80
5.2.2. Questionnaire Results . . . 81
5.3. Discussion . . . 85
5.4. Summary . . . 91

6. Conclusion **93**

Bibliography **95**

A. Evaluation Appendices **xiii**
A.1. Questionnaire . . . xiii
A.2. Results . . . xviii

B. ReWaMP Paper **xxiii**

List of Figures

1.1. MyCal Windows 9

2.1. Information System Life Cycle [9] 21

2.2. Operation of AppCloak after the migration [56] 24

2.3. Migration Architecture with GUIDeliverer and GUIMaker [10] 25

2.4. Migrating a COBOL system to a Model-View-Controller (MVC) using web application [4] 26

2.5. Migrating a legacy using proxies [5] 28

2.6. The MockupDD iteration cycle [46] 33

3.1. ReWaMP overview leveraging WASM 37

3.2. ReWaMP Architecture 39

3.3. ReWaMP Process 41

3.4. Decision tree regarding adding external functionality to WebAssembly (WASM) 50

4.1. WASM-T architecture as Unified Modeling Language (UML) component diagram 56

4.2. C++ parser's UML class diagram 58

5.1. Average Processor Times 81

5.2. t Measurements 82

5.3. t_M Measurements 82

5.4. Average Effort Distribution 83

5.5. Functional Scope, Legacy and resulting Prototype 84

5.6. ReWaMP Process and WASM-T Ratings 85

5.7. Most difficult Task in the ReWaMP Process 85

A.1. t_T Measurements xviii

A.2. ReWaMP Process Total Effort xviii

A.3. Questionnaire Results 1 xix

A.4. Questionnaire Results 2 xix

A.5. Questionnaire Results 3 xix

A.6. Questionnaire Results 4 xx

A.7. Total Measurements xx

A.8. Detailed Measurements Results xxi

List of Tables

1.1. Conceptual Characteristics of the Legacy Software 9

1.2. Software Metrics of the Legacy Software 10

4.1. Assumptions 69

5.1. Statement Results regarding general Information 83

5.2. Statement Results regarding Usability 84

5.3. Requirements Satisfaction Summary 92

List of Listings

4.1. C++ parser functions 61
4.2. JSON syntax to server 71
4.3. JSON syntax from server 71
4.4. Embind Example 72
4.5. EM_ASM_INT() Example 73

List of Abbreviations

AJAX	Asynchronous JavaScript And XML
API	application programming interface
AST	abstract syntax tree
BPMN	Business Process Model and Notation
CSS	Cascading Style Sheet
CGI	Common Gateway Interface
DLL	dynamic-link library
DOM	Document Object Model
DTO	data transfer object
GUI	graphical user interface
HCD	human-centered design
HCI	Human Computer Interaction
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP address	Internet Protocol address
JS	JavaScript
JSON	JavaScript Object Notation
MDWE	Model-Driven Web Engineering
MFC	Microsoft Foundation Class Library

MockupDD	Mockup-Driven Development
MVC	Model-View-Controller
MVP	Minimum Viable Product
NCS	Network Computing System
OMG	Object Management Group
OS	operating system
RMI	Java's Remote Method Invocation
RPC	Remote Procedure Call
SaaS	Software as a Service
SME	small and medium-sized enterprise
SLOC	source lines of code
SOA	service-oriented architecture
SQL	Structured Query Language
SRS	System Requirements Specification
SSH	Secure Shell
SUI	structural user interface
UGUI	unified GUI
UI	user interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
VM	virtual machine
W3C	World Wide Web Consortium
WASM	WebAssembly
WWW	World Wide Web

X11 X Window System

XML Extensible Markup Language

1. Introduction

> It was really hard explaining the Web before people just got used to it because they didn't even have words like click and jump and page.
>
> *Tim Berners-Lee*

As Tim Berners-Lee assessed, explaining the Web was difficult in the beginning of web history. One can assume that he means nowadays this has changed for the better. This is substantiated by the fact that the web is now prevalent, and gets more and more inalienable. But the web is still young, and thus companies nowadays want often migrate their older legacy software to the web, be it to stay competitive or satisfy the new needs of their customers.

1.1. Current Situation

Every company producing and selling software faces at one point of time that their product has become a legacy. With regard to [61], software evolves to a legacy by time. Thus, it is longer in use than expected at the beginning of development, and aged as humans do [33]. The overall costs of the product increases by updating it over the years according to the changing circumstances [61]. And as [27] concludes, maintenance and enhancement are consuming much of the total resources within a software project.

Complexity and difficulty of maintaining such a legacy system increase over time, because the technical debt is increasing by maintaining the legacy. Hence, bugs will need to be fixed over the entire duration of the software. Every update will also bring up new bugs to fix, and new issues to solve. In addition, the fact that no software product is perfect, and rather any version of it, emphasizes the need of maintenance [29, 65]. Even bigger companies with enough resources for high-end test management cannot test all possibilities, and be free of bugs. Hence, small and medium-sized enterprises (SMEs) will not be able to do that at all.

Likewise, the software candidate for being a legacy is not only expensive as a whole, but has a big and important value for the whole company. Some definitions of a legacy software even describe it as business critical [61], which is running all day [10]. So the software is the business process or at least an important part of, and a failure could cause immense costs and delays in the business operations [61].

At such a point in time re-engineering the legacy is a chance to stay attractive and competitive. It can preserve the value, while lower the costs, decreasing the complexity of code and dismantle dependencies of old inconvenient third party software. Further the code can afterwards be re-factored for increased readability, performance, or just have less code doublings, whether syntactic or semantic.

Along with the aging of the software itself, another aging process sticked with many generational transitions is the developers team aging. At each transition some knowledge of the software will be lost, and has to be recovered by the new generation. Developers team aging is not the aging process of humans, which is significantly longer than software aging, but the aging of the team. Developer teams are governed by the typical management changes, as one member intends to retire, another one is going to work for a new project in the company, and another is leaving company, and so on and so forth. Thus, the software has already outlived many developers, but also administrators and users.

Beyond, several issues could come to matter, as describe by the criteria of a legacy system in [61]. Examples of this include: a backlog filled only with bug fixing tasks, a long compilation time, different versions around the user community, doublings in the source code and a bad or even not existing System Requirements Specification (SRS).

Nowadays such a legacy software is often a desktop application with graphical user interface (GUI) written in languages like C or C++, because in the near past software was developed for running on one machine without a communication to a second one. It was often even planned for a small group of expert users, which were not necessary the end-users, but mediators [2]. Hence, the software should work for a specific environment, shielded from outside.

Older applications were not planned for use on more than one machine. Due to the feasibility problem of traffic speed and web ubiquity at the deployment location appeared inevitable. Most of the times there were not even a need for end-users to have a software at several locations or different environments. Furthermore, the cost issues connected to the first Internet connections were for SMEs not even possible to realize.

Later, the World Wide Web (WWW) gained popularity, thus web applications. This was a consequence of additional improvement at the web ubiquity and the high

increment of traffic speed. Thereby most of the systems are now not a stand-alone version of one product, but a highly linked system, which communicates in different aspects with many interfaces. The growing market of mobile devices accelerates this trend. A web application is mobile and dynamic, and thus satisfies one of the most important needs for a modern company to stay competitive [2]. Moreover, the normal user of a system is increasingly the end-user. Thus, even the concept of the user is shifting [2].

Not only the popularity and the market are good reasons for web applications, web applications have also advantages regarding feasibility. The viability and the desirability of human-centered design (HCD) as it is described in [60] are already underpinned, the advantages regarding feasibility substantiate it even further, and thereby complete the HCD model.

The underlying client-server architecture of web applications can outsource calculation power and memory from client to server. Accordingly, hardware requirements are decreasing for the end-user's machine, traditionally calculating the client parts. As every web app is accessible over the world wide web, the software product itself is very mobile and accessible at every machine with a web-browser [25].

The web-browser initiates even further that the application is platform independent [25]. So no other software has to be installed on the client machine, and the browser takes the software apart from the underlying operating system (OS) or virtual machine (VM).

The system behind the Uniform Resource Locator (URL) does not even need to be only one web-server, and thus be a single-point of failure. It can be several servers with a backup strategy and distribution of the application itself. Thereby it still stays the same for the user as all acting, as the same server for the public.

Furthermore, the client software is delivered at every session. This also substantiates the mobility advantage and improves the update ability, since the software can be updated for every user without effort of the user himself. Additionally, the server-side is always staying under control of the developers by design. [25] sees these advantages as a consequence of the modular design of web applications.

Further, web applications have a better maintainability compared to desktop applications. This sticks to the fact that there is only a single-point of access and the improved update ability.

A web migration of legacy software is often the logical step forward for many SMEs using the several advantages of web applications to stay competitive. Moreover, this migration will start a re-engineering process, and thereby maintain the software.

But if a SME wants to bring their legacy to the web, normally some doubts arise. Those introduce a hurdle for the company, which I will call *the initial hurdle of web migrations*. This hurdle is normally created by doubts about feasibility and utility. Whereby issues like a high risk of failure, lack of domain knowledge about the legacy in the company and the absence of experience in web development in the team are faced by doubts about feasibility. Issues like unknown stakeholders, implicit needs and the actual yield are doubts about utility.

It is an issue that the software has many implicit requirements. Most of them are not evaluable due to the fact that they are unknown. It was and stays normal for a software product to have many stakeholders. But as the legacy is running some time a few of them may be forgotten or change at the point of migrating to the web. The different stakeholders generate many requirements for every application. Those requirements could remain the same, but mostly they will change by migrating to the web. Even if some of them do not change, most of the requirements are often forgotten after several years beyond the original development phase.

Another reasonable and inferring doubt is the high risk of failure implied by unknown requirements and stakeholders. Even further such a failure could also have unclear, or even unmanageable, negative consequences for the whole corporate business, as losing reputation or market shares. Most of the time, the lack of domain knowledge about the legacy after the several years and an absence of experience in web development promote this risk as a matter of fact.

Such an initial hurdle has to be overcome with good arguments and a straight plan with enough security for the management of the company. But initial hurdles are a well-known pattern at every start of developing a new software. In agile development one established paradigm to overcome such hurdles by saving time and money is rapid prototyping [19, 57]. By setting the design and research phases of a development in parallel, rapid prototyping generates viewable and usable prototypes of the final product in an early phase of development. It includes thereby screen displays, databases and major program modules, and assumes that a full understanding is the design result, not an input to the actual process [58]. Moreover, rapid prototyping will include all nontechnical stakeholders in the early design process and show a first version of the product as a base of discussion [63].

1.2. Motivation

It would be a dramatical change for all SMEs if they would be able to quickly create a running web-based version of their legacy system. Thereby a first rapid prototype can be the starting point of the migration, and help to overcome the initial hurdle

of web migration. Throughout this thesis I will refer to this as *rapid web migration prototype*.

The legacy software should thereby be recycled, and used for such a prototype. This ensures that most needs satisfied by the legacy are also complied by the prototype. As the architecture changes, the complete behavior of the product will change.

The prototype can be used to reveal unknown needs of stakeholders, but also unknown stakeholders by running it for a test phase and showing it to possible stakeholders. Several old needs will expire with the migration and many new ones will come up.

Such a rapid prototype brings also all advantages of web applications directly to the legacy software, and enables the possibility for a step by step migration over time. Each module can be reconsidered and redesigned to fit as good as possible in the new appearance of the legacy as a web application. And every step can be taken seriously without losing time having no web application of the legacy, because the prototype serves as the starting point of the migration.

All those facts lower the risk to fail the web migration, and thereby address the previously mentioned common doubts. This results in a dismantling procedure of the initial hurdle. I propose the hypothesis that such a rapid web migration prototype will help to overcome the initial hurdle of web migration as a rapid prototype helps to overcome initial hurdles at the starting of developing a new product.

To the contrary, such a prototype can also be used to demonstrate that a specific legacy software should not be migrated to the web. Thus, it does not only reduce the hurdle, but also can expose scenarios on which the product does not behave well when it is deployed to the WWW. For example, it could be the case that a migration to the web breaks in a specific operation or feature of the legacy some laws in one of the main countries the software is used. Or other requirements emerging from the web migration are not easy to solve, and may harm the company reputation.

The rapid web migration prototype will demonstrate those aspects in a very early point of the migration process, and has the potential to save a lot of development time and thereby money. Project abortions can also be the rapid feedback in the behalf of rapid prototyping within agile development [19].

SMEs often struggle at their own personnel, and thereby at the knowledge of migrating their legacy to the web. Hence even a rapid prototype seems impossible within days, because the many points of failure within a typical legacy ensure already a complex situation without a manageable overview. Non-existent SRSs, omitted documentation of the product and confusing code parts impede the understanding of the whole code, even for developers and experts in the programming language of the legacy. But not only parts of the legacy domain knowledge are typically missing,

also the know-how of programming web applications is often an absent commodity within a SME developing in the past especially desktop applications.

An automated process of the first rapid prototype would give such companies a huge benefit, and lessen the missing knowledge of the current personnel. There will still be the need for the company to recruit new employees with web knowledge or educate their elders further. But it would help them to give the migration a try, saving time and money. Thus, assuming a company does not have the needed experts for development of web applications and is missing the complete domain knowledge about their old legacy software, a first automated rapid web migration prototype would be beneficial.

1.3. Problem Analysis

A mostly automatically created rapid prototype is not a trivial task. Topics such as architecture of the software, decomposition in the three parts front-end, middleware and back-end, and separation of concerns regarding different programming languages are just the top view differences. Also challenges as accessibility of the source code, old dependencies to external third party libraries and ensuring the rapid prototyping characteristics need to be solved.

Whereas desktop applications are designed for the execution on one machine, web applications are designed with some kind of client/server architecture in mind. Hence, the process of building a rapid web migration prototype with the legacy as source needs considerations how to deal with this new architecture within the old product.

Furthermore, the current common web paradigms supposes that the application is strictly divided in at least three parts: front-end, middleware and back-end. The legacy system can also be divided in the three parts, but is not always easy to decompose [7]. Regarding web applications, the middleware, which constitutes the business logic of the application, is not always only on the server-side. So the decomposability of the legacy refers also further to the decomposability of the middleware.

Whereas the main language in a desktop application is also the language for most of the concerns, a web application uses in different concerns also disparate languages. A typical desktop application uses its main language, probably any Structured Query Language (SQL) dialect for the database, and a specific language to describe the GUI elements and their positioning, which typical is closely coupled with the main language. A web application on the contrary uses typically many different languages. Whereas a web application with regard to separation of concerns uses a different language per concern. Hypertext Markup Language (HTML) is used to describe

the user interface (UI) elements and their rough position, whereby a Cascading Style Sheet (CSS) is used to define colors, the exact position of each UI element and for behaviors of the GUI. JavaScript (JS) is then used to manipulate the Document Object Model (DOM) and to let the GUI communicate with the server. Finally, a programming language for the sever-side middleware and a SQL dialect for the database are chosen. The actual web server logic is additionally sometimes implemented again in another language.

Despite complete source code access, which is in this thesis assumed, it is not simple to reconfigure and rebuild the legacy as a web application. Besides internal linkages, dependencies to third party libraries, which misses typically the source code, there are further difficulties such as the execution of binary files, which is traditionally not possible in a web browser. This is based on the fact that all web applications are independent of their hardware. Since binary files such as dynamic-link librarys (DLLs) are not directly executable at a client, they need be migrated to a executable format at the client or only be executed on server-side.

Another difficulty arises at the conceptual stage by ensuring the rapid prototyping characteristics with this process' result. As the name terms, the prototype should be produced rapidly, preferably within days, not months. While rapid prototyping is an analysis-by-synthesis approach and brings thereby research and design to parallel tasks [58], it is important that the prototype can easily be expanded for further migration to a full web application of the legacy.

The resulting software is a prototype if the main features are already implemented and testable in the new environment. Therefore major program modules, a GUI and an according database should be included in the prototype. Because the company should be able to check with the prototype for new needs and circumstances to really overcome the initial hurdle of web migration, as rapid prototyping assumes to have as a result in forward engineering [58].

1.4. Scenario

Besides the described situation in section 1.1, a concrete scenario for the thesis is needed to have a background for the implementation and an experiment in the evaluation. The chosen legacy system is a sample application created by two other students. Its implementation is based on a real world legacy code base from an industrial partner enterprise. They abstracted characteristics out of the real world code base and created the sample application consequently.

This application is called *myCal* and provides a simple calendar for medical practices. While browsing the calendar, it is possible to list all appointments accounted for one specific day. Every appointment has a start and end date with an exact time, and is related to a patient, a doctor and a room. Patients and doctors are modeled by their full name as a string, and an id. All patients save in addition their date of birth, even if this is not visible in any view of the legacy. The rooms are described very similar, but instead of a name they have a description, which is mostly a room designator in string format.

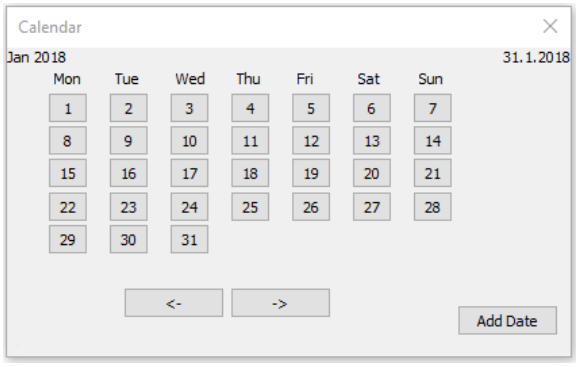
In addition, it is possible to arrange an appointment in the near future. The corresponding mask is providing a predefined set of data. So the user is enabled to specify, which patient and doctor should meet at which room. Further he must narrow the search for a new appointment with the earliest starting time, the appointment duration in minutes and by marking the possible business days for an appointment. All these data cannot be filled in by hand, but have to be chosen out of combo boxes or via check boxes.

Figure 1.1 shows all three described features with their corresponding views. At the startup of the application, the calender dialog and the appointments view are shown. While the calender is initialized with the current date, the appointments dialog starts as an empty window. If the user clicks on a button of any day in a month, the corresponding appointments will be listed in the appointments view. The dialog for accounting a new appointment will be opened by clicking on the *Add Date* button, and will be filled with the predefined set of data.

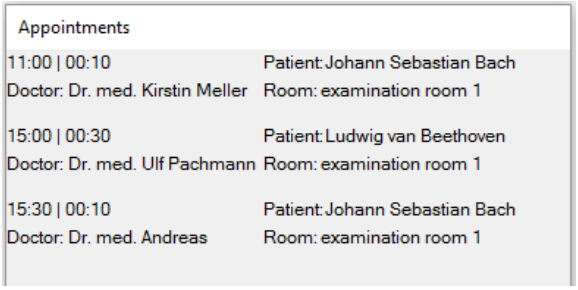
Further non-functional characteristics of the used legacy scenario are: Microsoft Foundation Class Library (MFC) as the used desktop UI library, Visual C++ as legacy language, data persistence via files and a database, and third-party dependencies via assembly loading of DLLs. Consequential to MFC also other non-functional characteristics arise, as the component communication, the UI description, and the UI message broker. In addition, the legacy code was inspected with cppDepend [13] to get also software metrics about it. While all characteristics are summarized in table 1.1, the software metrics of the legacy are shown in table 1.2.

1.5. Objectives and Limits

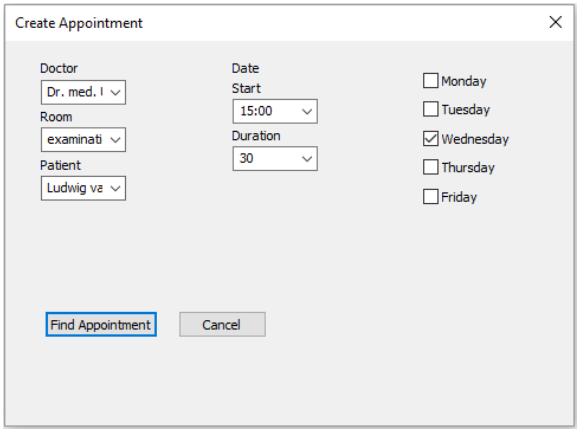
The purpose within this thesis is to build a rapid web migration prototype of a legacy system. As the title terms using WASM is preferred. To save resources, especially for SMEs, as few as possible human interaction within this process is an important factor. With focus on the middleware in this thesis, the only steps to be included are for demo purposes regarding the front-end and the back-end.



(a) Calendar Dialog



(b) Appointments View



(c) Accounting new Appointment Dialog

Figure 1.1.: MyCal Windows

Conceptual Characteristic	Representation in Scenario
Legacy Language	Visual C++
CRUD functionality	Creation and Readability of Appointments
Complex Business Logic	Account next possible Appointment
Third-Party dependencies	Assembly Loading of DLLs
Data Persistence	Files and MS SQL Server via ODBC
Desktop GUI	MFC
Component Communication	MFC SendMessage between Windows
UI Description	MFC Resource File
UI Message Broker	C++ Class for every Window

Table 1.1.: Conceptual Characteristics of the Legacy Software

Software Metric	Measured Data at Scenario
Lines of Code	937
Number of Types	35
Number of Methods	237
Number of Source Files	64
Number of Third-Party Elements	177
Technical Debt	6.34%
Average Cyclomatic Complexity for Methods	1.8 Paths
Average Cyclomatic Complexity for Types	16.36 Paths

Table 1.2.: Software Metrics of the Legacy Software

WASM is one possibility to bring old legacy code in a format usable in a web application. It is a stack machine designed binary format of code, which can be parsed and executed within the JS engine in all common browsers. Hence it „should be comparable to a big new JS feature, not a fundamental extension to the browser model “[64]. To this day WASM is only released with a first Minimum Viable Product (MVP), but is enjoying increasing popularity among different institutions. E.g. the unreal engine got a WASM support for demanding video games with its version 4.16 [34].

In the center of this thesis' attention is the development of a tool, which can compile a legacy code to WASM by outsourcing all necessary functions and calculations to the server-side and communicate with the new HTML GUI. This tool will hence separate the middleware for the client- and server-side and make the client middleware sensitive for a work with the new GUI.

The thesis will not contribute anything in detail how to achieve a separation of client- and server-side code out of a legacy nor anything about the decomposability of legacy systems. In fact, it is in this thesis assumed to have already the knowledge about the separation, and the decomposability of the legacy is assumed as *decomposable* [7].

The separation itself is given by an artifact, which is either a formatted file defining all parts of the server-side, or the knowledge of an expert. This artifact is quite important, and it needs a detailed consideration of the code to decide which features and functions should be at the server-side, and which at the client-side. Moreover, it needs to have the information where the interface is, which sends data from the middleware to front-end. But I will not contribute anything in detail to the question, how to come to such an artifact.

Moreover, and as already mentioned, this thesis will not contribute new things to the final front-end and back-end. I want in the context of this thesis to focus on the client-sided middleware part, even though the back-end is only the database, and thereby not the only server-side part. Hence the whole server-side part and how exactly the code will be deployed for execution on the server is not a part of this thesis. Technologies like Common Gateway Interface (CGI) [26] and fastcgi are well-known to execute code remotely on a web server, and should not be a part of this thesis. Furthermore, the GUI will not be in the corporate design of the company, but I will only use bootstrap [54] to give the GUI a look-and-feel. Thereby any animation stuff upon the standard bootstrap library will not be included in the final tool.

I will also not contribute anything new to WASM nor its compiler emscripten [68]. I am only using WASM as a technology for building the rapid web migration prototype by reusing the legacy. Also the topic web migration will not get anything new by this thesis. It is only the big objective behind the motivation of this thesis and the result after a successful rapid web migration prototype. I also do not contribute to rapid prototyping as a methodology or rather a paradigm of choice in software development. It is only the foundation of the hypothesis to overcome the initial hurdle of web migration.

1.6. Outline

Chapter 2 will establish related requirements about the first rapid web migration prototype and its process to create it. In addition, it will conduct the presentation about state-of-the-art web migration approaches, the rapid prototyping methodology and their synergy as in the thesis' context.

The overall concept to create a rapid web migration prototype will be presented in chapter 3 with focus on the leveraging of WASM. Hence, all conceptual knowledge about the resulting prototype and the process how to create the WASM files with its needed environment will get explained.

In chapter 4 the implemented tool, which was conceptual described in chapter 3, is set out with a higher level of details. Thereby its architecture, the implemented algorithm inferred from the underlying concept and used third-party tools are in the focus.

Next, chapter 5 is evaluating the presented concept by executing an experiment. It further analyses which defined requirements are fulfilled how good. The thesis will close then with a summary and ideas for future work in chapter 6.

2. State of the Art

This chapter will point out the requirements related to a rapid web migration prototype and go on with a consideration about state-of-the-art treatment and handling of the constituent parts. Those parts are the different topics, which come in this thesis together, and thus are web migration and rapid prototyping. Moreover, it will designate some works, which contexts may are interesting for the synergy. In addition, it discuses what this means for the specific context of this thesis or rather rapid web migration prototyping.

At first the requirements will be considered in section 2.1. Further the topics web migration and rapid prototyping will be discussed in the sections 2.2 and 2.3. The chapter will conclude within section 2.4 by some works closely to the crossover of web migration and rapid prototyping.

2.1. Requirements

The requirements of a rapid web migration prototype can be separated in two sections:

1. process-related requirements
2. result-related requirements

While process-related requirements are those the process needs to fulfill, result-related ones are in turn those different roles have in accordance to the result.

Hence, the process-related requirements will not be distinguished any further into subsections, but the result-related ones. I will use three different roles to distinguish the result-related requirements:

- developer
- manager
- user

These names are pretty common, but I still want to describe them further in the subsequent subsections to give them a face and prevent confusion.

To outline this section, it will start with the user requirements in subsection 2.1.1, and proceed with the manager requirements in subsection 2.1.2. The last subsection of result-related requirements is subsection 2.1.3 about the developer ones. Beyond the process-related requirements will close the section within subsection 2.1.4. It should be noted that all subsections about result-related requirements have in common to describe the related role at the beginning.

2.1.1. User Requirements

The *user* is a person who, as the name terms, uses the software. In context of this thesis, the group of users can be detailed further by persons, who already used the legacy software or will probably use the web application. Thus, they can be distinguished as long-time users, upcoming users or those who are somewhere in between, called normal users.

Many times, the user is determined as the role with the most importance due to the fact that his satisfaction essentially influences the amount of money the software will yield. This holds not completely in the context of this thesis. A first prototype must not even be shown to a user, but an internal audience. Still, there are some result-related requirements inferring of the potential a user is somewhere in the whole process of web migrating the legacy.

Functional Scope

The functional scope of the prototype compared to the legacy software should be the same as much as possible. To get a real feeling for the performance of the legacy as a web application, the same functional scope is an important requirement. Thereby the user wants to see that the web application is as great as the legacy was. Deleting functionality from the prototype could result in a wrong representation of the actual migration.

It should be noted that this requirement is not testable without manual effort. It is moreover only testable via an acceptance test of a user or an expert representing him. Such acceptance tests can never be a 100% proof, but can at least give a direction.

Usability

The usability [3, 20, 28] is a big topic in Human Computer Interaction (HCI), and it is also important for a rapid web migration prototype. In this context, I refer to the usability as a requirement, which especially concentrates on the handling of the software. It is possible to improve the usability during creation of the prototype or rather during web migration, but it should not decrease.

Moreover, the user should be able to recognize familiar structures from the legacy. For instance, a well-known button should be at the same place and nearby do the same. This is closely coupled to the requirement of functional scope, and the usability requirement is also only testable via an acceptance test.

2.1.2. Manager Requirements

As the company consists not only of developers, all non-technical persons involved in the process of creating a rapid web migration prototype are called further a *manager*. This person is always someone who is not caring about technical detail, but on money, project success at a deadline, user satisfaction, team's productivity and so on. It is a guy planning or designing the result. Maybe he has also technical knowledge, e.g. a project manager of the web migration, but can also be a guy from the economic division without further technical knowledge. A manager will always be somehow a boss, the developer has to convince for a web migration, and thus is a decision-maker.

Decision Support

With the purpose of overcoming the initial hurdle of web migration, it is deducible that the rapid web migration prototype should be a decision support. So it should help to decide whether a web migration of the legacy is desirable or not. Thereby the manager wants not only a preview how the result will look like, but also wants to prove feasibility. Yet viability and desirability should also be underpinned by the prototype.

This requirement is inferred from the user requirements and by the manager's motivation to earn money and reduce costs by migrating the legacy. So user feedback is needed to satisfy this requirement completely, and thereby it is again only testable via an acceptance test.

Rapid Creation

As the prototype is not a final product and time is for a manager closely coupled with spending money, he wants to have a first prototype as fast as possible. So building the first one should not take months, but days or weeks. Thereby the manager does not care about the legacy's complexity, but only on time.

The sooner a final decision has been made, the legacy can be migrated to the web and yield money. Also, if the prototype highlights that a web migration is not viable, desirable or feasible, an earlier decision is a benefit due to unbinding resources from an idea, which will not even be realized.

Cheap Production

Closely to a rapid creation, it is also important for the manager to have a cheap production. The main purpose behind is again to save money, because a prototype is not a final product, thus not usable for normal business.

Thereby the rapid creation plays a role, but also the number of employees which are needed for this process. In other words, this requirement is mainly bound to how many employees working how long to create the first web migration prototype. Hereby less is always better.

In addition, the material costs also play a role. While hardware costs is for the first prototype even less than for a final web application, material costs at software products may must also be spent for tools or third party libraries.

2.1.3. Developer Requirements

The last role involved in the process is the *developer*. As commonly known, he is responsible for actual implementing the software. Thereby he has a key role due to his technical domain knowledge. He can have, as mentioned in section 1.2, three different parts of domain knowledge in context of a web migration: legacy code, web application development and domain knowledge about the actual purpose of the legacy. Most times one developer does not have all three parts, but at least one of them.

The legacy code knowledge is specified by knowing much about the programming languages and the architecture of the legacy. Web application development knowledge connects the expertise of typical web programming languages and web architectures.

Knowing much about the actual domain where the legacy is used in and its derived technical aspects is summarized under the domain knowledge about the actual purpose of the legacy.

Low Effort

As the developer has as close deadline for finishing the first prototype by the managers, he is interested in a low effort approach. Thus, this requirement depends on the rapid creation and cheap production requirements of the manager.

The developer wants to save time as much as possible and be not required to manually change a lot. Furthermore, an effort can also be to learn many new skills before being able to perform the prototype creation.

Looking forward to the evaluation, this can be evaluated through a comparison between different approaches how to create the first rapid prototype. If no other approach is available, the comparison can also be done via changing some aspects at the creation process of the desired prototype.

Rapid Prototyping Methodology

Since the rapid web migration prototype should not only be a decision support, but also a starting point for further web migration, it is important for the developer that requirements of the rapid prototyping methodology are fulfilled. According to [1, 19, 45, 57, 58, 63], the three main requirements thereby are:

1. Simple Expandability
2. Rapid Creation
3. Containing all necessary Modules

Thereby simple expandability means that there should be more than one prototype, which is underpinned additionally by the idea of a step by step migration. A rapid creation is important for the methodology by deriving from nearby the same backgrounds as the rapid creation requirement of the manger role. And by containing all necessary modules within a first prototype, the forward engineering wants to assure that the user feedback is relevant enough to influence the design. This is also deducible by the manager requirement of having a decision support.

2.1.4. Process-related Requirements

The process-related requirements have more technical details due to their definition of being process-related, and the context of this thesis. Thus, the following requirements are often more detailed upon the technologies or challenges which have to be used or solved in the concept at chapter 3.

Process Automation

Holding the effort as low as possible for the developer derives the process to be automated as much as possible. In the context of this thesis, thereby especially the compiling to WASM and JS, the integration of the new client/server architecture and the establishing of an application programming interface (API) from the WASM code to the new GUI is important.

Since there is often already a lack of domain knowledge at the company, it would be even more onerous for the firm to know many details about WASM or its integration in an HTML page. Hence, such an automation will support the step by step migration idea after the first prototype by saving time at the realization of the next iteration.

If this automation would not be needed, the whole issue of this thesis would get trivial. Because if a process automation is not required, the effort for a developer would be low, and thus the prototype would be created rapidly.

Legacy Code as a Source

By using the legacy code as a source it is possible to represent the functional scope and the usability of the old legacy code within a new web application. This supports also being a decision support. Further it decreases manual effort of the developers, but uses still the expert knowledge to generate a prototype.

Additionally, it avoids some fails related to the already named requirements, from which this requirement is derived. Hence, either the code will be recompiled regarding the separation of concerns or it should be at least been used for semantic analysis to reproduce a real image of the legacy.

Using WebAssembly

As the name of this thesis terms, it is a requirement to use WASM within the approach to realize a rapid web migration prototype. This limitation leads to a consideration of one specific technology, and thus shows how it performs in the context of rapid web migration prototyping.

Open Web Standards

Using only open web standards is required to have finally a web application, which is sustainable and future-proofed. So only technologies which are open web standards defined by World Wide Web Consortium (W3C), Institute of Electrical and Electronics Engineers (IEEE) or at least Object Management Group (OMG), are acceptable. Thereby it is not only sustainable, but also feasible for SMEs and performing better due to no library conflicts of two third party libraries within the approach. Technologies like Flash or ActiveX are thereby excluded from the concept, as well as any emulation of some old habits regarding the legacy's environment.

Client/Server Architecture

As the introducing of the client/server architecture is a challenge, it also needs to be solved by the approach. In more details, this refers to the WASM code, which should be able to communicate with the server. Thus, this requirement is closely coupled with the usage of WASM. To assure the same scope of functionality the application should not ignore the code at the server, but call it to get the related results.

This is comparable to technologies like Remote Procedure Call (RPC) [32, 55], improved RPC within a Network Computing System (NCS) [11] and Java's Remote Method Invocation (RMI) [62]. The server-side is then maybe also considerable as a service within a service-oriented architecture (SOA) [35, 69] or the Software as a Service (SaaS) model [59], where software parts can be ordered dynamically to solve tasks.

Communication with HTML UI

By migrating in the web and only using open web standards, the legacy's GUI system needs to be transformed in a new HTML variant. Thereby the WASM code must be able to communicate with this new UI to assure the legacy's functionality in

the prototype. Otherwise, the UI would be encapsulated and the web migration prototype would not be complete as required from the rapid prototyping methodology.

2.2. Web Migration

On a general level, migrating legacy software to the web is a modernization of the system. By considering the life cycle of software in general, every software comes to the need of modernization or replacement at a some point in time. Yet a modernization is not possible or cost effective at every point of a maintenance stagnation [9].

The life cycle of every software product is abstractly described in figure 2.1. So all starts at a development phase, where the software is built. After a certain time of maintenance, updates are not anymore able to satisfy all needs, they rather disimprove the legacy.

In figure 2.1 this is marked with stagnated growth at the end of first maintenance section. Beyond the figure shows that after a modernization and a second maintenance time the system will be replaced by a new system, which will then start at the beginning of the cycle. Further it should be noted, that the business needs are always changing, which is illustrated in the figure with a constant growth. The business needs are thereby an accumulation of all needs relating to the software.

The thesis' contextual situation is to classify at the end of a maintenance section, but before the decision if a modernization or a replacement should take place. As mentioned in the current situation section, the legacy replacement would cost too much money and time. So the decision makers or rather the mangers would like to hear that it is possible to modernize the system fast and step by step with a cycle of earning and reinvesting money.

Furthermore, [9] distinguishes every modernization in two categories: black-box and white-box. Whereby black-box modernization approaches only observing the in- and outputs of the legacy, and white-box ones are reverse engineering all details to have a full understanding of every legacy part. With regard to the approaches I will present in the following subsections that this classification can only be applied approximately. Because some are neither completely a black-box nor entirely a white-box migration, and thus reverse engineering only parts of the legacy.

The idea to migrate existing software to the web came implicitly with the start of using the web. Thus, there are several ideas and approaches available how to achieve that. To have a further detailed distinction between the approaches, I will organize

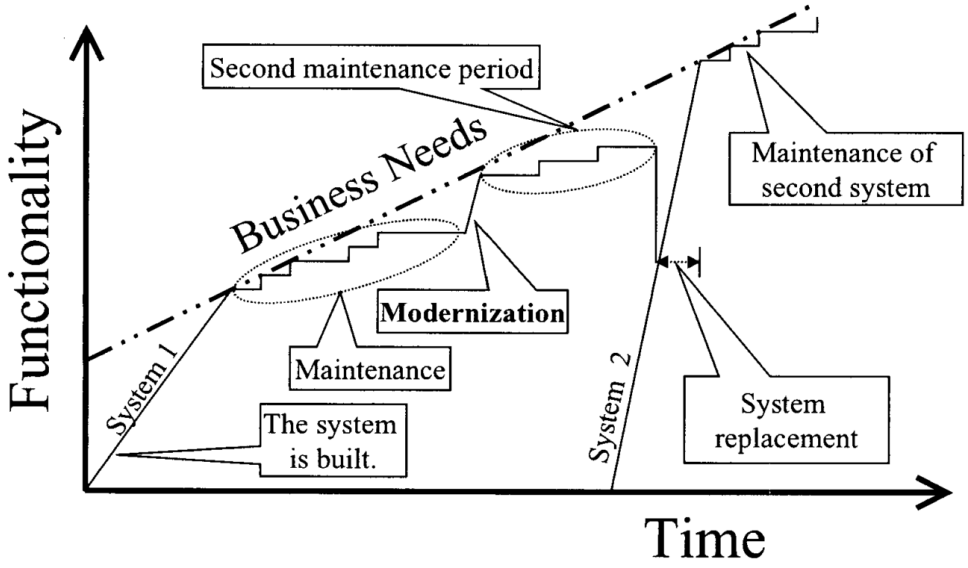


Figure 2.1.: Information System Life Cycle [9]

the rest of this section closely to the distinction in [56]. Even if they are cloud migrating a web application, the categories can also be translated to web migration. Hence, this section will cover: Replacement, Re-installation, Redevelopment and Relocation. The classification in black-box and white-box migrations would be not detailed enough, apart from the approximate applicability. Conclusively further notes about web migration will close this section.

2.2.1. Replacement

One kind of migrating existing software to the web is to replace it with a new developed web application. It can be implemented by the company or even bought from a third party. May it be not really a migration, due to the fact that nothing of the old software will be used in the new application. But it solves somehow the issue, and delivers a web application. Indeed, much domain knowledge of the legacy will get banned by just replacing it.

As this is the other way round of the thesis' motivation, replacing the legacy software with a new product should not be the solution. Further it has too many possibilities of failure for the company to do that with a real legacy of a business critical value. Issues like loosing much knowledge established over the years, all gathered data or even the competitive race by investing again years for a web application are all not desirable results. Additionally, it may be the case that the company is then even

investing in a software project, which results in a worse behavior than the legacy with lower satisfaction of all needs.

2.2.2. Re-installation

Another idea at [56] is to reinstall the application at the new environment. In their context they would take the server-side software and reinstall it at the cloud operator. To solve this, all dependencies of the server-side part need to be reassured at the new environment, and many modules need to be exchanged with newer versions to run at the new environment. Thus, they connect this approach with much manual work for the developers.

In context of web migration re-installation will not lead to a real web application. As nothing will be changed upon functionality, the legacy will still be a desktop application with no distributed design or rather architecture. Hence, it will only run a desktop application on a server without real access for many users. To circuit this a bit, it is possible to transfer the screen of the server to several clients. But this only leads to distribution of one session to many users. Thus, only one user could really interact with the application, while the rest could just watch.

2.2.3. Redevelopment

Close to replacing the software is redeveloping it. The difference hereby is that the result is clearly not bought and uses re-engineering to gain as much knowledge as possible from the domain the legacy was supporting. This approach is very time-consuming and thus not applicable for a rapid web migration prototype. [2] states that even any long-term migration will need to be redeveloped, because this is only realizable due to a complete re-engineering.

Even if the long-term goal is real web application with open web standards, the first prototype cannot be created via redevelopment.

If just looking at the long-term goal, it could lead to the assumption that migrating software to the web is only possible via thoroughly understanding the legacy system and redeveloping it as a web application [21]. This leads to a complete white-box migration with fully re-engineering the whole legacy.

2.2.4. Relocation

The last idea to perform a migration is to relocate the software. Thereby all needed files will be copied to the new environment. At cloud migration as in [56] all files of the server will be copied to the cloud operator. But within web migration such approaches will copy all or nearby all files to a server and then build a structure to create the new client/server architecture.

It is possible to distinguish the relocation approaches even further. Hence, this subsection will present approaches how to migrate via a relocation legacy to the web, and sort them into two different categories. Those are namely:

1. Relocation and Environment Disguise
2. App+OS Enclosure and Reconfiguration

In the context of cloud migration it is also possible to name a third category: Relocation and Reconfiguration [56]. But this is not possible in the context of web migration, since the needed architecture transformation. At cloud migration it can be possible to just take all files of the server, copy them in the cloud and then update all configurations until everything works well. Such a performing is quite close to the idea of re-installing in the cloud server-side components. But in contrast, the re-installing assumes a big change in the environment and thus changes at the required software packages. Relocation and Reconfiguration in contrast assumes that such a big intervention is not needed and only configuration files need an update.

Relocation and Environment Disguise

In terms of AppCloak [56], relocation and environment disguise means to copy all related files to the new environment and manipulate the OS to hide any changes. Hence, the new environment will be disguised and the legacy will not become aware of any changes.

AppCloak is introducing with its migration a new layer between the application and all system libraries. In figure 2.2 an example of such an intermediate layer is given. Thereby an application, which has been migrated via the approach of AppCloak to the cloud, wants to communicate with a peer component. Thus, it needs to establish a connection with a specific machine. They assume that the application gets the Internet Protocol address (IP address) of its configuration files. As the file system is accessible as before, the AppCloak layer does not need to interfere at this point. After the application has invoked the connection command, the AppCloak layer needs to exchange the IP address with a new one related to the new environment. The saved

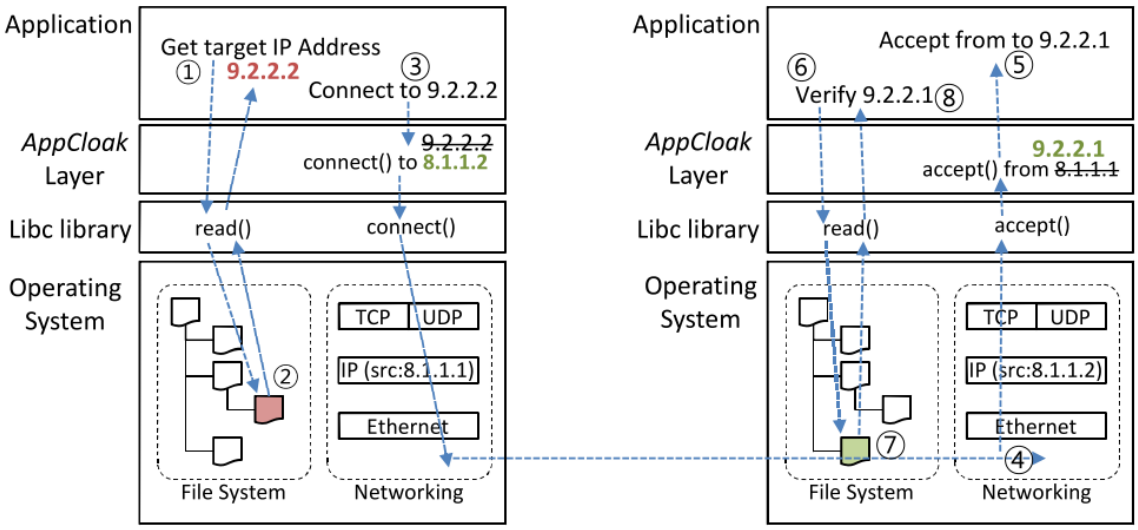


Figure 2.2.: Operation of AppCloak after the migration [56]

IP address in the file system is wrong, since all configuration files were not edited at the migration. Later on at the peer component, the AppCloak layer needs to exchange again the IP address. This time it is made correspondingly for the sending application part, such that the peer component can verify if it is valid to accept this connection.

As the context of [56] is still cloud and not web migration, the environment disguise is different in the context of web migration. Hereby a disguise needs to hide the architecture change from desktop to client/server, and build a bridge to the new HTML UI. Most ideas with environment disguise in mind therefore wrap the legacy code [2, 4, 10]. The UI is in these approaches always detached from the legacy and sometimes transformed as in [31]. So the UI is mostly not wrapped but only the rest of the legacy.

The idea of relocation and environment disguise is thus a mix of white-box and black-box migration. It is not only a black-box migration due to the re-engineering or transformation of the UI, whereby more habits than just the in- and output have been analyzed. As the middle-ware and back-end are not re-engineered, but wrapped, those are only analyzed at their in- and outputs, thus are black-box migrated.

One possibility is to detach the UI from the legacy and then completely re-engineer it [2]. Thereby the new UI has no further issues in contrast of serving it as a web application, but the web server needs a wrapper to enable communication between the new UI and the wrapped legacy. So if needed the UI triggers a method of the legacy via a wrapper and then shows the result to the user. Its communication is

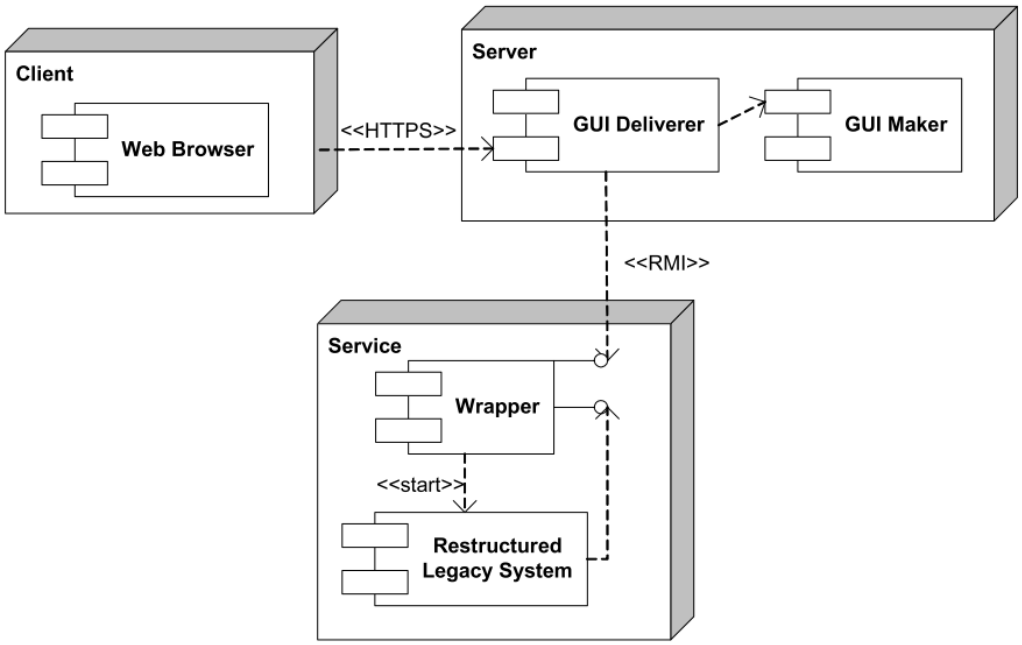


Figure 2.3.: Migration Architecture with GUIDeliverer and GUIMaker [10]

hereby solved via strings. To separate between the information given within the string special characters are used. Moreover, the approach in [2] uses more than just one wrapper, and distinguish in two object wrappers. One for the routing software and utilities and one for the lower connection layer. Thus, one for the business logic and one for the back-end.

In contrast, [10] does not re-engineer the UI completely. They wrap the legacy software without separating the UI. Two more components named GUI deliverer and GUI maker are working instead dynamically with the wrapper as to see in figure 2.3. So the wrapper has two interfaces, one for the legacy and one for the GUI deliverer, which is the access point for the client and decides which page should be displayed. The GUI deliverer further communicates with the wrapper to execute the needed methods or rather commands, and with the GUI maker to get an HTML page for the client according to the legacy's UI. To solve this dynamic transformation, all user inputs and the whole UI are described in Extensible Markup Language (XML). Furthermore, figure 2.3 shows that the legacy is not on the same machine as the server, but acts as a service.

Again different is the approach of [4]. Their purpose is to have finally a web application using the MVC pattern. After analyzing the source code statically, it will be decomposed in the three parts front-end, back-end and middleware. Then an object model will be abstracted, and afterwards the system is ready to migrate. The UI

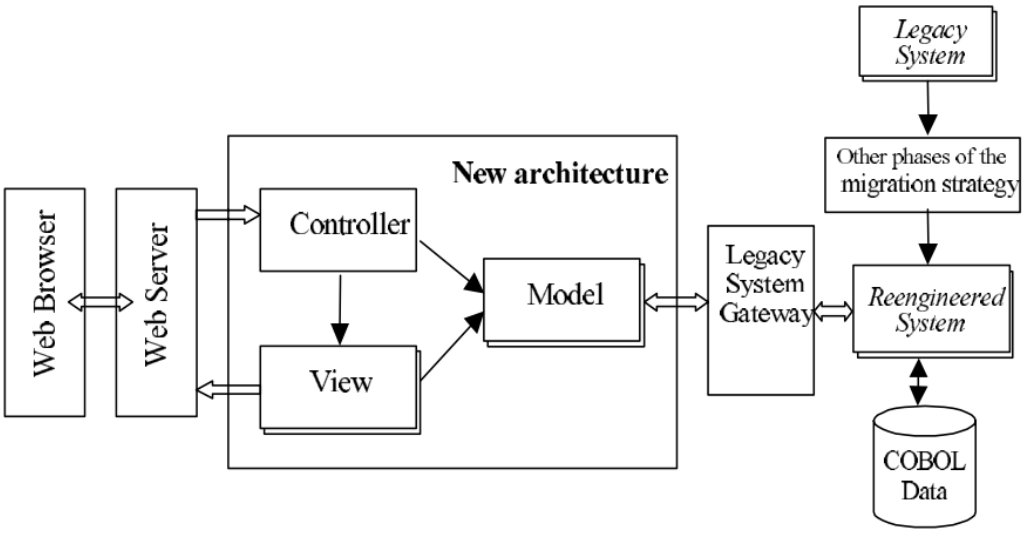


Figure 2.4.: Migrating a COBOL system to a MVC using web application [4]

will again be re-engineered and all the identified objects will be wrapped separately. Consequently, the system is transformed from a COBOL legacy into a MVC using web application. All instructions and data of the legacy are thereby within the Model part. Figure 2.4 shows how the result architecture will look like. The big plus behind many legacy wrappers is that it can be transformed within iterations to a web app without holding any old code line in the final web application.

App+OS Enclosure and Reconfiguration

The second category of relocation packs the application with its OS in one big package. To create such one and be able to run it, often VMs are the tool of choice. In the context of cloud migration such procedure seems legit to reduce manual effort to nearby zero. Further a VM is copyable, and thus dynamically and multiple applicable in the cloud to offer the service as often as needed. But there are still problems with using VMs directly in a cloud application, and VMs are generating a performance overhead due to their large tail of OS features, which are not necessary needed anymore within the cloud [56].

The purpose of a web migration distinguishes thereof by having only one server with the application in mind and not several nodes of the same server-side application. In other words, it is enough to have one server with the application, answering all the requests. As a VM image of the old desktop application can still not enhance it by relocating it on a server to a web application, this category is formed differently compared to cloud migration. Close to having a VM is also to black-box the whole

application as it was and copy it on a server, which then uses it for distribution. The migrations are then fully a black-box migration by only using the in- and outputs.

So the legacy will be wrapped as one big component and copied to the server, as well as the UI. To bring a HTML-based UI finally to the client, the server needs to emulate the UI in the browser. Apart from the following approaches this can also be done by using a remote access windowing system and an according browser plug-in. E.g. by using Flash and connect it with the UI at the server via Secure Shell (SSH) [67] and X Window System (X11).

In the following two different possibilities will be presented to enclose the legacy and reconfigure it for a web application. One possibility is to build a wrapper for the legacy system, which can interact with the legacy UI as a user agent [5, 6, 8, 44]. The other close one is to create an intermediate translation in another language, e.g. XML, and enable therewith the legacy to be kind of service for the web server [51, 52].

Having a kind of user agent in the wrapper to interact with the legacy, [5] uses therefore proxies. One proxy is for the UI and called screen proxy. Another one is named database proxy and used for saving old legacy data in a relational database. Whereby the screen proxy will be needed also for more modern systems than in [5], database proxies may not be needed, since most applications today using a database to access their data.

To give an overview about the usage of those proxies figure 2.5 shows the resulting architecture. The numbers above the arrows are giving an order of the work flow. When the web server gets a request from a client, it will start the legacy system if not already done, and hand over all necessary information of the request to the screen model. This saves the UI state and creates the HTML output for the web server. Moreover, it is able to deal with client's inputs to send them to the screen proxy. This gets all in- and output requests of the legacy's screen, and thus is able to communicate with the legacy. The database proxy whereas is used for data, which was transformed from flat file into a database. As [5] uses COBOL legacies, it is often the case that there was no database included. Thus, the legacy expects flat files of its data and is not able to communicate with a database. The figure shows also that it could still be used a flat file system for data accessibility besides the database proxy. This similarity is explained due to may two different sets of data, whereby one is transmitted into a database and the other is let untouched.

Close to the proxies in [5] are [6, 8] with one single wrapper for the legacy. Both exclude any database proxy, and only access the legacy from top-level via the UI. Inside that wrapper is always something like the screen proxy and the screen model

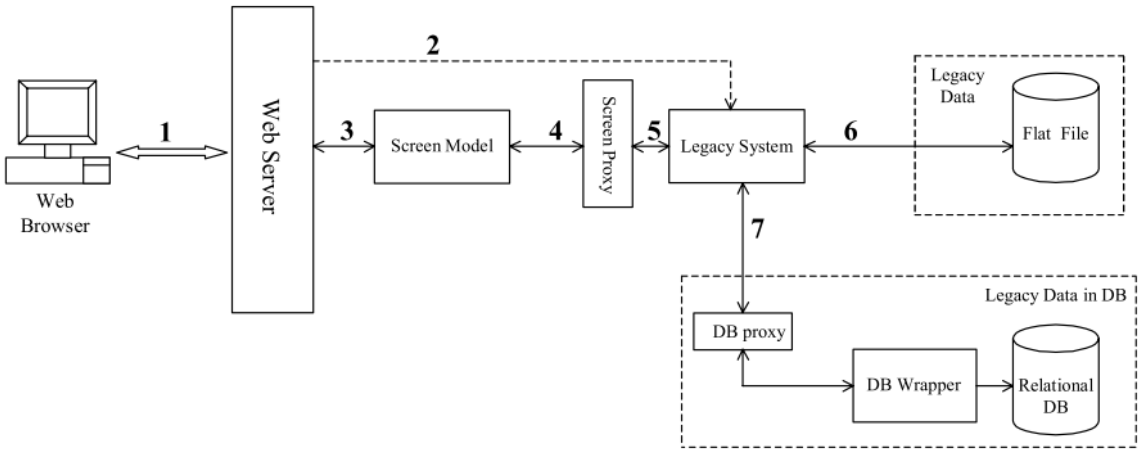


Figure 2.5.: Migrating a legacy using proxies [5]

of [5]. In [6] even XML is used inside the wrapper to exchanged information between the components of the wrapper.

XML is also used in [51, 52] by Sneed to wrap legacy code and provide it as a service in a SOA. [8] also sees the migrated legacy system a service for web applications. But Sneed goes even further, and talks about getting a legacy only migrated as a service within the web. He describes further that all legacy system will get an interface for XML-based in- and output and finally uses many legacy systems within one new web application.

Most of today's applications are not using just textual UIs, but GUIs. Migrating those legacies to the web black-boxed brings the approach to more challenges. One solving this is presented in [44]. With the OS, a GUI proxy and a unified GUI (UGUI) library loaded the legacy is fully copied to the server. The GUI proxy is working as already known and the unified UGUI is working as the screen model in [5]. Finally, all screens of the legacy are send via the UGUI to the client as a HTML page. The big issue within this approach is that all events trigger a page reload, also mouse events. This breaks the good usability. [44] proposes to shift some functionality of the GUI back to the client as in normal web applications. This would then lead also to another migration style.

2.2.5. Further Notes about Web Migration

All those presented approaches show that they need to redesign legacy's UI. If it is dynamically done as in [5, 6, 8, 10, 44, 51, 52], it can save resources at the migration process. But according to the big topic usability, the different usage of the legacy

within its new environment can also bring up new needs. A dynamically generated UI could miss the target of usability. Thus, a supporting tool set for a re-engineering UI project can help, as presented in [31]. Even if it is only for textual UIs and not for GUIs.

As the decomposability of the legacy system is an important factor for the web migration, the working approach for a specific one is different. As less the system is decomposable, as fewer options are working. Furthermore, the minds are divided in the context of non-decomposable legacies. Some say those should be directly re-engineered [2], and others work on a migration strategy for them without the need of re-engineering [5].

Research has also been made close to web migration without a direct new idea how to migrate to the web. For example, [50] shows that nearby every layer of software can be encapsulated and wrapped. This leads to the assumption, that if the legacy is decomposable enough it can be really migrated step by step exchanging one wrapped part after the other. And getting finally a web application without technical debt of the legacy.

As the MVC architecture for web applications was not developed at the beginning of web application development some older web applications are not using, just like some results of the presented web migration approaches. Therefore, [36] presents an idea how to translate web applications to the MVC standard.

Yet web migration is not only from desktop to web application. Migrating partial interfaces on-demand to another device is also a topic in research [14, 15, 16, 66]. Within the modern society it is normal to change the device often, e.g. leaving the house to meet a friend, and thus changing from computer to smart-phone. In [14] this is solved with an extra migration server, which migrates the needed parts of the interface for another device on-demand.

2.3. Rapid Prototyping

Rapid prototyping is not limited to software development. Its origin is rather at production processes or mechanical engineering. But rapid prototyping is also nothing new in the software developers' community. The benefits of a first rapid prototype at the beginning of a development project supplanted the possible drawbacks and failures of rapid prototyping [58]. Nowadays, rapid prototyping is well-known and commonly used at the start of a new software project, whereby the requirements from the stakeholders upon resulting software are gathered [1, 63]. The out-coming

of this starting phase is a guide to fulfill with the development as much as possible needs and requirements according to the software.

A first rapid prototype should therefore simplify the complex situation, and start the learning process of the developers about the needs linking to the software and how to satisfy them. So rapid prototyping with several iterations is an analysis-by-synthesis model, which serves as design as well as research phase. Furthermore, it lays the foundation of communication between developers and nontechnical stakeholders, and for the whole project itself. [58]

So it should cover all essential modules, features or parts of the software and give everybody a common base line for further discussions about the project and its purpose [1, 53, 58]. But it also does not need to be a full implementation of the essential parts. It should only cover them such that everybody gets a feeling for the probable result [63]. And is thereby using a „under the hood“ philosophy. So it needs to be evident what happens, e.g. when the user clicks a specific button. If the modules behind this are to complex, a predefined answer without running the real procedures can also be enough. Rapid prototypes thus need not be a software, but can only be paper mockups.

The early feedback in rapid prototyping is not even limited on what a stakeholder says, but how he uses the prototype or what he is doing with it [63]. Thereby also several unconscious requirements can be detected by a developer, and refine the requirement descriptions. It is still possible that not all needs will be revealed, and an iteration phase with many iterations can even lead to a design-by-repair handling [58]. This would even decrease personal initiative of all developers. But the prototyping iteration cycle also gives them the possibility to experiment a lot, and thus increases in turn their creativity [63].

The other big plus is the user enthusiasm, which is often generated through a rapid prototype. Continuously, all non-technicals are motivated to help with the requirements and notice that they are important for the design process and want to use their influence [1]. Errors within the prototype are getting fixed and needs get clarified, and with such a motivation often even faster [58]. As enthusiasm can also topple in disappointment due to unrealistic expectations [1], it is important to be aware that a prototype can always be oversold. So the enthusiasm of non-technicals need to be managed, also to not terminate after the first prototypes [1, 58]. Thereby high-level requirements could be named, but still be unclear in detail for the developers.

By contrast, if rapid prototyping is not used within a project, several issues can obstruct or even impair the completion. For instance, stakeholders' needs and requirements can change during the whole process without any feedback, because no common base line for mutual communication is given. And often non-technicals do

not even know what they really want to have in relation to the software. Another big minus is also that the requirements' analysis is harder for the developers due to no given requirements teacher or rather no communication with other stakeholders about more than musings. Thereby the requirements analysis will take more time and have more faults. [58, 63]

Conclusively, prototyping is boosting the overall project performance, especially when all involved know what the rapid prototyping is for. So everybody needs to know that the prototype is just a prototype, and that design and functional behavior can change in further proceeding. The rapid prototyping methodology can then save time and money, and prevent bigger failures. Therefore, the developers should not be overstrained or frustrated to have many changes of the requirements within each prototype iteration. [1]

Additionally, the prototyping should be subject to rules such that the prototyping is not leaping too great forwards, and run therewith in the extremes and the drawbacks. If this is given, rapid prototyping will clarify needs and requirements, which were unclear or ambiguous at first. Furthermore, the developer's experimenting and learning will especially help at using new tools or approaches, even if the developers are unfamiliar with a certain technology. [1]

2.4. Rapid Web Migration Prototyping

Rapid web migration prototyping is until now not a term in research. Thus, no references can be named nor any approaches how to deal with this coalescence. But there are fragments in some approaches, which I want to emphasize. This section will thereby present those, and infers what this means for rapid web migration prototyping.

A goal within web developing is to find the right balance between client-sided and server-sided code to exploit the full potential of web applications. If this would also be reflected within the first prototype, the actual perceived overall picture of the future web application would be closer to the reality and the probable result of the migration. But all presented web migration approaches bring at least the legacy's main part on the server. Thereby the client is often simple and naive, thus the GUI. But modern web applications do not calculate everything on the server. For a better usability and a more dynamic and stable version, several features are delivered to the client in JS format. By using WASM this could lead even to a solution the other way round with more calculations at the client-side and less functionality at the server-side.

Furthermore, a web migration is more durable and future-proof when the legacy is not only pressed in a web application, but migrated with at least some re-engineering. Thus, a web application without nearby any old legacy code should be the long-term goal [2, 4]. The iteration cycle and further improvement with received feedback of the prototype substantiates this goal. The claim that a first migration prototype should present the essential functionality, but be not the whole project's result, supports that even further [1, 58, 63]. Some only see this long-term goal achieved by a complete new development [2, 21], but others are already close to the iteration cycle of rapid prototyping and want to migrate the legacy step by step [4] to a web application without any old code dependency.

This shows also that satisfying the rapid prototyping methodology requirements is important. And especially it is underpinning simple expandability, which ensures iterations and step by step enrichment. Furthermore, it shows that a step by step migration was also in minds without using rapid prototyping, and is not a complete new idea nor untouched.

One context close idea to rapid web migration prototyping is pursued by Rivero et al [45, 46, 47, 48]. They want to use the UI mockups from rapid prototyping for modeling the features and generating Model-Driven Web Engineering (MDWE) meta models They call this process Mockup-Driven Development (MockupDD) [46]. To clarify what they are doing their process model or rather iteration cycle is shown in figure 2.6. So after a first phase with paper mockups they generate a structural user interface (SUI) out of those, and then bring the user stories captured in mockups and SUI models a semantic via tags. Thereby the requirements will be defined with these tags. It is possible to run a demo with the tagged SUI and ensure stakeholder satisfaction. Moreover, it is possible to generate MDWE meta models, as UML-based Web Engineering [45] and WebML [48], out of those tag enriched SUIs.

The most interesting part for rapid web migration prototyping is step 4.a in figure 2.6, where Rivero et al. used an approach called MockAPI [47] to generate a first server-side API. With such an approach rapid web migration gets easier, since the server-side gets accessible according to the used GUI. MockAPI is generating a RESTful [22] API mockup out of the enriched SUI. Therewith it is possible to connect the early UI mockups with a server-side. The approach MockAPI focuses on basic CRUD functionality and uses XML to define the related access points.

In accordance to rapid web migration prototyping Rivero et al. show further that rapid prototyping is an enrichment for developing web applications. Web migration is not explicitly a development, but is closely coupled due to the fact that in the end a new web application with the functional scope of the legacy should be created. Therefore, at least some parts should be re-engineered, as stated before, and from

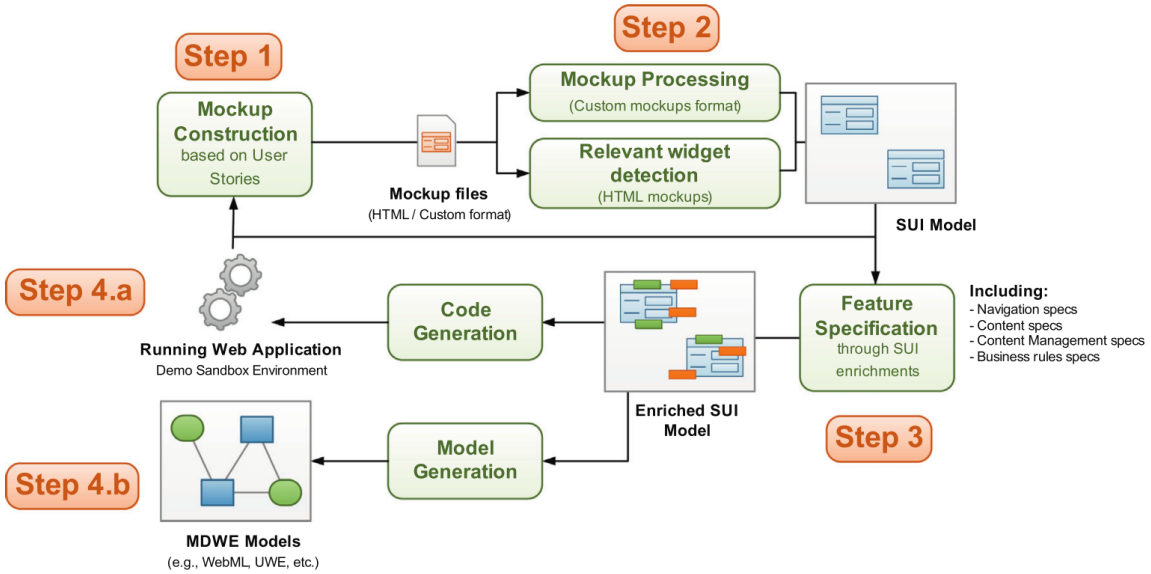


Figure 2.6.: The MockupDD iteration cycle [46]

those some will be re-implemented. Thus, web migration with future-proof interest is in parts also a development.

Another context close idea is AppCloak [56] as presented in subsection 2.2.4. AppCloak is as MockupDD not directly in the context of web migration, but wants to create a rapid cloud migration to save resources of the company. They do not mention any prototyping, but show how to rapidly migrate a legacy. With regard to other relocation approaches in web migration AppCloak shows that a rapid migration can only be performed by wrapping the legacy and refine specific important points. In the context of AppCloak this is made visible by the example of interfering at the connection to a peer component, but not at a file system access. Back to rapid web migration, it is observable that only at important points of the runtime an interference is necessary. For rapidly create a first prototype this will also hold true.

3. Concept

This chapter presents a concept to for creating a rapid web migration prototype, called ReWaMP. Further it describes ReWaMP process, which was developed in the course of this thesis. This chapter starts with section 3.1 about the ReWaMP approach leveraging WASM. The ReWaMP overview presents the actions and artifacts needed of the whole process. Further on, in section 3.2 the client-sided architecture of prototypes using ReWaMP is presented. Section 3.3 is then describing the ReWaMP process itself. Next, section 3.4 discusses the challenges of ReWaMP and how they are solved. Closing this chapter, additional notes according ReWaMP are described in section 3.5.

3.1. ReWaMP leveraging WASM

As shown in chapter 1, a rapid web migration prototype seems to be a suitable concept, to overcome the initial hurdle of web migration. Taking the legacy system as the source and trying to minimize additional needed artifacts, following three are used to get a first prototype:

1. legacy code
2. UI mockups
3. annotations

The legacy code is the legacy's source code. I assume that the SME has access to it, according to the situation described in section 1.1. For reminding, legacy software was described there as a software with active maintenance and business critical value, thus it is too important to lose it.

The UI mockups are drafts visualizing the user interface. Such drafts are often used in agile software development [19], and can be either paper- or computer-based [49]. In agile development they are part of the early feedback and an output for the stress in design. They are the very first possibility in forward engineering to create a base for discussions between developers and managers, previously defined as non-technical stakeholders. As those drafts are visual sketches, developers can extract technical

details from them and managers can understand them without detailed technical knowledge. Besides creating such drafts as in a new project manually and from scratch, a first version of UI mockups in context of web migration can be screenshots from the legacy's GUI.

The annotations are then the UI mockups' tags as Rivero et al. [46, 47] described. These annotations are specifying the syntactical structure and how different UI elements are related to each other. For instance, they describe the exact role of an UI element within the UI mockup or the relation of different UI elements. HTML is a comparable format, but not as comprehensive as those tags are.

All three artifacts are either part of the legacy or can be created in accordance to it. Thus, the legacy is used as a source and no documentation nor any other additional artifact is needed to start. The legacy code is, as it is the technical description of the system, a direct part of the legacy system. The UI mockups can be generated directly out of the legacy via screenshots, and the tags can again be created via these generated mockups.

The ReWaMP overview is visualized in figure 3.1. With the starting point of the legacy code in the upper left corner, at first all three necessary artifacts need be created or extracted. With all input artifacts in hand, three different actions have to be taken. These can be run in parallel, and each produce another part of the first prototype. Moreover, they are numbered in figure 3.1 from 1 to 3 and can be named as:

1. ReWaMP process
2. GUI re-engineering
3. using MockAPI to generate a RESTful API

The ReWaMP process is producing the ReWaMP runtime, which depicts the legacy's view behavior to the browser. It uses therefor only the legacy code as input and brings the related code parts as WASM in the browser. To bring the code not only syntactically, but also semantically to the browser, the ReWaMP runtime is needed to enable the resulted WASM file also semantically. As this process is the core of this thesis, it will be further described in section 3.3.

To gain a web-version of the legacy's GUI, the UI mockups are used to create according HTML and CSS files. This step is currently done manually by the migration engineer without further support. In the future this step could be supported with a tool, generating a first web version out of the mockups. This could be done by transforming an absolute layout into a grid-based layout. Absolute layouts are typical for desktop GUIs, while grid-based layouts are used by common style libraries such

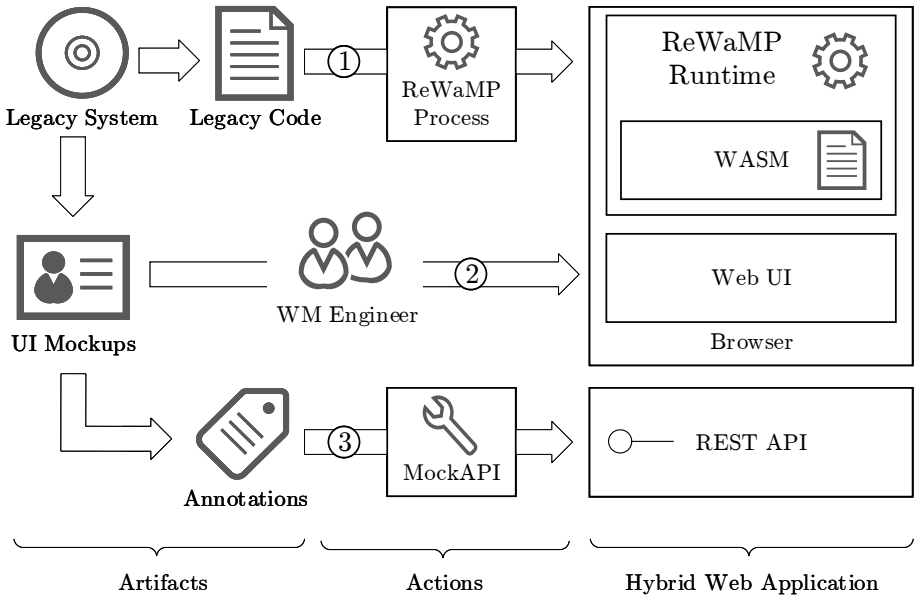


Figure 3.1.: ReWaMP overview leveraging WASM

as bootstrap [54] for web UIs. To achieve this transformation, optimization algorithms could try to get the best out of the mockups by referring to defined similarity constraints [18].

To have a server-side within the first prototype and enable thereby a first client/server architecture, the MockAPI [47] can be used to generate a RESTful API. This can then process requests from the client and answer accordingly. The created API is planned as a support for the first front-end creating iterations in forward web development. With respect to rapid prototyping, this comes in hand with the idea of a first rapid web migration prototype.

The resulting architecture is superficially described at the very right of figure 3.1. This shows that the result is a hybrid web application since it uses old legacy code in WASM format with a newly developed web UI and newly created REST API. The hybrid web application can be improved by further iterations to create a full-stack web application without any old legacy code. Furthermore, this is the sought prototype to overcome the initial hurdle of web migration.

Noticeably, the ReWaMP overview leveraging WASM is closely coupled to the ReWaMP overview within the paper in appendix B. This is not only inferred by the naming, but also by the figure itself. The concept within this thesis distinguish also at the resulting client-side, because it is using WASM by design. It is further very close to an UI-first software development approach, where impact of the UI design is present [12]. Thus, the first prototype implements the basic functionality, but is

only a simple mockup to make the UI executable. In context of web applications, this means that the client has already a visual web UI with the expected features, but the business logic is may not implemented at this state, and the data are may not be stored consistently.

3.2. ReWaMP Architecture

According to a detailed consideration of the ReWaMP process, it is important to have closer look upon the detailed architecture at the client side. This section will therefore present the detailed ReWaMP architecture and give an example, detailing how the common workflow will proceed within this architecture.

It is desirable to reflect the common web architecture usage within the first rapid web migration prototype. Most native web applications use the client side only for purposes of the actual UI logic and its behavior. Thus, only parts of the legacy code base will be located at the client-side. Otherwise, it could lead to an unintentional perception of the legacy's first web version. For instance, possible longer page loading time at the first visit due to a bigger WASM file, and faster processing of business related tasks, where no data access is needed, could occur.

Thereof it is deducible that every view will get its own WASM code to support the old legacy behavior of the UI. So only the view layer code describing the UI's behavior needs be compiled to WASM, i.e. the code making a GUI performing dynamically. And as a typical web application distinguishes its user interface in separate HTML files, each view of the legacy's desktop can be compiled to one WASM file and delivered with the related HTML view.

The respective architecture during a loaded session at the client, consists as shown in figure 3.2 of the following four elements:

1. HTML/CSS
2. WASM
3. JS glue code
4. ReWaMP JS code

The HTML/CSS element describes the new UI and is created in figure 3.1 at action 2. To act like the old GUI, the client is required to handle the same tasks as the legacy view behavior code. Therefore, the HTML/CSS element is accessed by the ReWaMP JS code via the DOM API of JS.

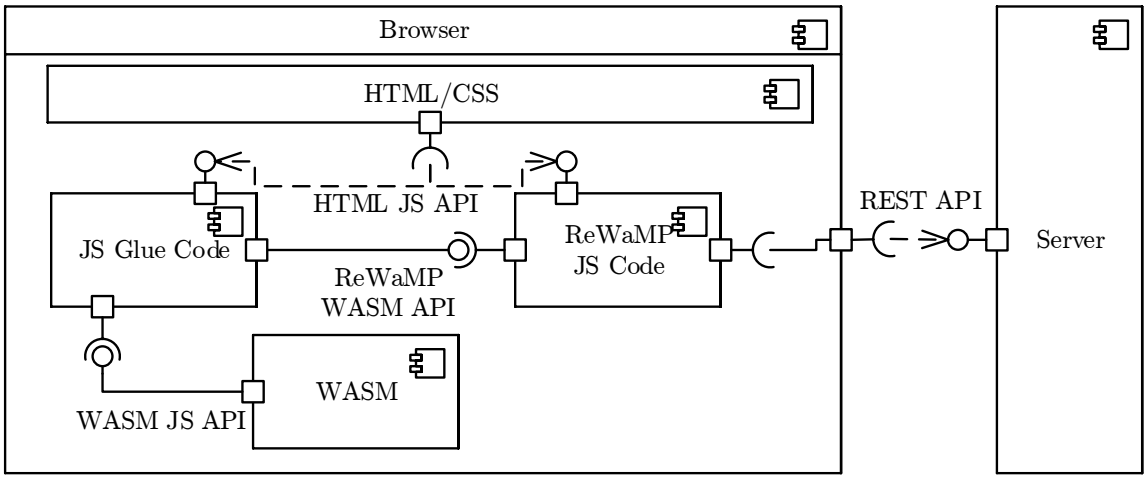


Figure 3.2.: ReWaMP Architecture

As it is desirable that the web UI behaves as the legacy, the legacy's view behavior code is compiled to WASM. Consequently, the ReWaMP runtime is built by the three client-side elements, called WASM, JS glue code and ReWaMP JS. Thereby the runtime's core is settled down in the WASM file, which creates the actual behavior. This includes the whole UI functionality and all UI semantics. To enable communication with other components it is bound to the JS glue code.

The JS glue code is created by emscripten [68] at the compilation of code to WASM. It provides the JS API of WASM with many standard functionalities like WASM memory management and serialization of strings for message exchange. The glue code can be configured for the exact view context via specific emscripten expressions within the legacy code. This special syntax will then be used by emscripten to create a JS API with context sensitivity. So to speak, it is the bridge for the ReWaMP JS code to communicate with WASM.

The ReWaMP JS code enables the communication of WASM with the server, and ensures abstracted access to the new HTML UI. Therefore, it provides standard functionality like DOM value exchange, server requests via Asynchronous JavaScript And XML (AJAX), and the initialization method to start the view as it was started at the desktop. Thereby the start values of the view need be set and the event listeners must be bind accordingly. For instance a method, which should be triggered after clicking on a button, will get bound at this initialization method. Accordingly, it is another API extending the JS API of WASM. This is necessary to enable WASM as the core functionality within the view.

In conclusion, the client uses basically its normal HTML/CSS/JS behavior to create a GUI for the user. To perform as the old UI, the toolchain of WASM with ReWaMP amendments is used.

The server creates the REST API to receive requests from the client and to answer them accordingly. The server is hereby implemented in a very simple level of detail. Its only job is to support the intended client/server communication and to enable the first prototype executable with web application standards.

To explain the architecture with an example, suppose that the user could press on an OK button of a specific view. As this event is triggered at the HTML/CSS layer, the event will be caught by the standard functionality of the JS VM. Since the ReWaMP JS code initialized the session in relation to the context of the specific view, the JS VM knows which function should be called at this event. The method called by the JS VM is calling a function of the JS glue code. This in turn calls the method in the WASM file, which was original called when the related button was pressed at the legacy's desktop version. The function in the WASM file performs the same calculations.

As the user pressed the OK button it is assumable that the triggered function will call the specific ReWaMP function to send all gathered information to the server with a status flag, which is in this scenario then 'OK'. Thus, the WASM file calls a function in the JS glue code to send something to the server. In turn the JS glue code method will call a function in the ReWaMP JS code. This function will then execute the request via AJAX, thus send the information to the server. If the answer is a simple redirect to another view, the function will directly execute it. Otherwise, the method will send the answer back to the WASM file via its return statement.

3.3. ReWaMP Process

As the purpose of the ReWaMP process is to compile the view behavior files per view in one WASM file, the related files need to be extracted from the legacy code, transformed in a WASM-compilable version and compiled to WASM. The compilation of C++ code to WASM is easy due to emscripten, but the transformation in contrast not. This differs depending on the used desktop UI library, but is always a time-consuming task for the migration engineer. Thereby it is important to delete all dependencies to unrelated files after the extraction. Furthermore, all now unknown functions, types, classes, variables, etc. need be declared again with either standard types or re-engineered versions. In other words, after extracting the files all arising errors need be fixed by re-engineering all related files of one view.

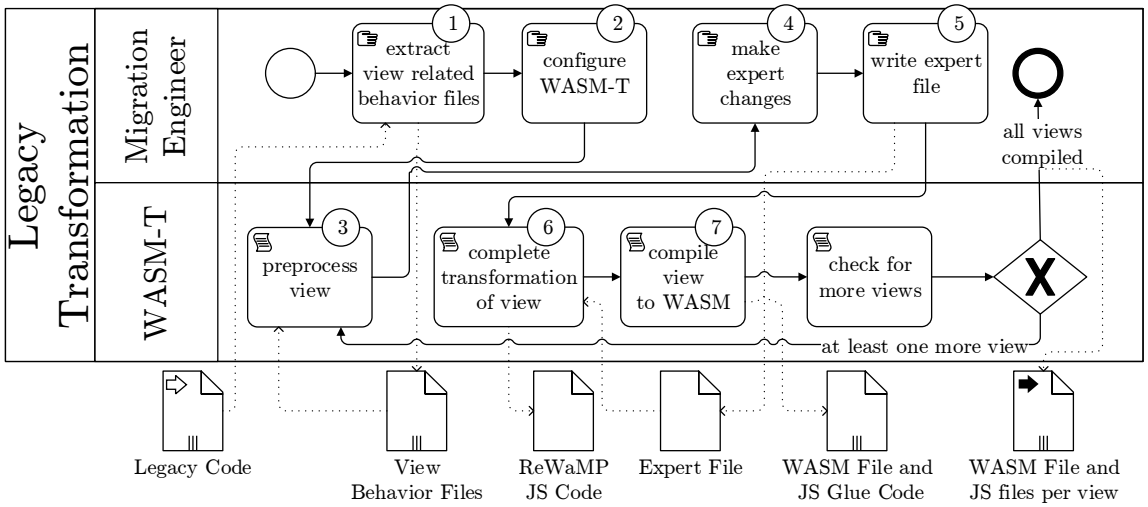


Figure 3.3.: ReWaMP Process

To support the rapid creation of WASM files for all views, I modeled a WASM transformation tool called WASM-T, and implemented it prototypical. While the abstract process will be presented within this section, the implementation will be described in chapter 4.

The overall workflow to get from the legacy code to one WASM file per view is shown in figure 3.3, where also the WASM transformation tool is included accordingly. Figure 3.3 is a Business Process Model and Notation (BPMN) diagram [17], with one pool for creating the WASM file per view, and two lanes inside. The expert or rather migration engineer is portrayed in one lane, and the supporting WASM-T in the other one.

At first, the expert must prepare the workstation for WASM-T. Therefore, he must separate all view behavior files from the legacy code at action 1. Hereby all code files, which directly access the UI, are important make changes to perform a UI behavior or combining these two. According to our scenario, described in section 1.4, those files are only files with C++ code in it, but not UI description files as resource files, also called RC files. Further accordingly to the scenario, the extracted files are implementing one main class, which inherits from a typical MFC class as for example *CDialog*, or is closely coupled to such one.

With these in hand, he can configure WASM-T at action 2, which means to edit the configuration of the tool respectively to the extracted files. At this time in the process, WASM-T needs know where all files are located, and which of them are directly linked to one view. Thus, which one are implementing an inherited class of a typical MFC class or rather the main class. I call those directly linked files *main files*

due to the purpose that they are serving as a starting point for one WASM file, as the main file in a common C++ program. All further related files, which are important for the translation of one view to WASM, are files used within a main file excluding standard and UI libraries. Nearby all of them are also clearly coupled to the view layer and representing a view or view element within the code. Thus, in MFC most of those related files are MFC dialog items like menus or toolbars. Another example could also be a class only communicating directly with one main file. Such a class could easily be assigned to the client-side, and thus should also be one of the related files, which got extracted.

Beyond, the script will run for each view, translate the code to make it compilable, and compile it to WASM. Thereby the transformation tool is not able to re-engineer every possible code fragment, because most dependencies are missing due to the extraction, and thus information about non C++ standard types. Furthermore, complex fragments are not easy to understand semantically for a simple parser. For this task, WASM-T will refer to the expert knowledge of the migration engineer.

WASM-T consequently transforms at first the view related files' parts at action 3. These files are comprehendable, and thus it provides a preprocessing of one view or rather one main file. It deletes or rewrites well-known parts of the code. For instance, it exchanges MFC types as CString to std::string, deletes call expressions to MFC classes or rewrites well-known functions of the UI library. A good example in context of MFC dialogs is the function DoDataExchange(). This method is written with standardized method calls to describe how the GUI and the class should exchange data, so which input field value should for instance be written in which class variable and the other way round.

Besides the first transformational tasks, the preprocessing step is also extracting some knowledge for action 6 to work correct according to the original semantic. Preprocessing is mainly concerned with deletion of some code parts, but then they would get lost as a source of information. Thus, relevant information of parts, which can be deleted within the preprocessing action, need be saved to preserve them. A good example in our scenario are the message maps of MFC. They describe which method should be called at which specific event. So if a user clicks on a button, the message map depicts which method should be called in accordance. This knowledge is important to correctly link the new GUI with the WASM file at action 6, not directly but via the ReWaMP runtime.

Next the expert must re-engineer code parts WASM-T was not able to, and provide corresponding changes. He does that either via expert changes in the code at action 4 or within the expert file at action 5. Expert changes in the code are mostly semantic to set aside complex constructs involving missing dependencies. The expert file entails

thereby container for missing declarations and definitions of classes, variables and functions.

It could be the case that the migration engineer is for instance finding a if-clause, which depends on a variable, which has an unknown type for the set of related files. So he has to write a re-engineered class or type definition in the expert file. To this end, he searches the related files of the view on occurrences of this type and gathers knowledge about it. Then he can formally describe this type in the expert file. After that he can also come back to the if-clause and redesign it. To achieve that there will only be few changes necessary. But as he re-engineered the type it could be that some accesses change, for example pointer versus object access in C++.

In the supportive way of the ReWaMP process architecture, the expert does not need to know much about the legacy nor read the legacy code entirely. All required information to write the expert file or to make expert changes are within the related files. Thereby it is not a need for him to rebuild for example missing functions entirely. He has only to re-engineer the function header and can leave the function body empty. Thus, the only needed information about a function written by the migration engineer in the expert file are its name, its return type and its parameters with their types. The bodies will then be filled by the transformation completion of WASM-T in action 6. Furthermore, added classes are closer to be a data transfer object (DTO) than a real class and need only fields and functions, which are also used within the related files.

Another support of WASM-T for the migration engineer are the ReWaMP flags. Those are comments with a special syntax, and tagging therewith the code location for a specific handling. Consequently, they are working as precompiler statements in C++ do. They are marking for example a function in the expert file which body should not be changed by WASM-T in action 6, because the body was already provided by the engineer. This is for example the case at simple getter and setter methods of a specific class field. Another possibility is to mark lines, which should be reformatted according to a specific style. Such lines are often using a well-known C++ syntax, but need be changed as a consequence of other changes. For instance, the C++ standard function *strcpy* was used in a line, but its parameters are now of type *std::string* and not anymore of type *CString*.

WASM-T completes in the next action 6 the transformation. It starts this step with adding the code of the empty expert functions. Next it adds JavaScript Object Notation (JSON) serialization methods for all involved classes to enable them for transmitting. After that it processes the ReWaMP flags from the expert. WASM-T completes this task with configuring the API of WASM to JS. Thereby especially all methods, which should be exported from the WASM file, and thus accessible by JS, getting marked for the emscripten compiler.

Finally, WASM-T compiles one view to one WASM file in action 7, so the main file and its other related files. It does this not on its own, but uses emscripten to do so. With all needed JavaScript files, namely the ReWaMP JS code and the JS Glue Code, the view is then in combination with the WASM file and the new web UI of step 2 in the top-level concept ready for usage.

3.4. Challenges

By using WASM at the ReWaMP approach, several challenges arose, which this section will discuss in more depth. These challenges are not in the focus of WASM's developers. Their motivation for WASM is to make old code written in other languages than JS accessible in the browser by concentrating on C and C++. Using WASM at web migration is thus not a part of their vision or a bullet point at their guideline for further development. But yet the solving of these challenges within WASM's design would also enhance it, and make it more common to use in migration prototyping. The relevant challenges are mainly:

1. WASM related Serialization
2. Client/Server Communication
3. Access HTML DOM with WASM code
4. Handling UI events in WASM code

Accordingly this section is divided in subsections, whereby each subsection is presenting one challenge and how it is solved. The ordering of the subsections is the same sequence as the numbering.

3.4.1. WASM related Serialization

Serialization is always important if variables or objects are passed via an API to another component. WASM itself only knows integers and floats, both in 32- and 64-bit format. To preserve the expressive power of common programming languages, handling other constructs is possible via loading and saving objects in the WASM memory. Since straight forward passing of variables or objects need a concrete handling, passing arrays, strings or custom objects from or to WASM is not possible directly via return statements and parameter exchange. To work around this issue, the WASM memory need be accessed via pointers of the allocated object. Therewith it is possible to read or write an object to or from the WASM memory. The generated

glue code delivers therefor the required functionality. It even provides some code to read or write strings formatted in different UTF versions. Hence, there will be no need for the ReWaMP process to work with bit transformations or direct memory access.

The chain of exchanging strings between WASM and JS is due to those UTFx methods achieved easily. Having the other challenges in mind, especially the client/server communication, the idea was to use the string exchange for every exchange that is not able to depict understandable in numbers. Thus, only numbers and booleans are exchanged via integers or floats, all other types will be serialized to strings. To give the string-based communication a common syntax, I decided to use JSON strings. Thus, all objects and arrays will be serialized as a JSON string, but strings remain without being JSONified.

3.4.2. Client/Server Communication

Since the prototype is created for a web migration, it is important to enable communication from the prototype to the server. Thus, satisfying the requirement of a client/server architecture, defined at section 2.1.4, is only fulfilled if the WASM file is enabled to communicate with the server instance, because the WASM is the core of the client-side code, as already stated in section 3.2. But this communication has to be solved in a way that WASM or rather the old legacy code part does not recognize the outsourcing of some code parts within the workflow. So the code must be transformed in a way that it performs as before but with integrated client/server communication.

The chosen solution is the deposit of JS code within the C++ code, which gets compiled to WASM. With thanks to emscripten this is easily possible. So there is no need of implementing a direct communication from the WASM file to the server, which also supports the requirements of only using open standards and the requirement about the rapid prototyping methodology. Thereby the open standards requirement is supported, because the basic part of communicating with the server is entrusted to JS, as common in the most web applications. The rapid prototyping methodology requirement is underpinned through the supported simpleness of expandability. Since the WASM file may not stay in a future-proofed web version of the legacy, the communication with the server needs not be transformed into JS in a future iteration cycle.

So after an initialization of the WASM file, all requests to the server are triggered by WASM. As it causes those requests via its JS API, the ReWaMP process provides a standard method within the ReWaMP JS code to execute a method at the server.

Since the WASM file uses a main file of a view to handle UI behavior, the WASM file needs only to outsource function executions to the server. Everything else needs be defined sufficiently to even compile the code to WASM. The called JS function sends the transmitted string to the server's REST API via AJAX. To preserve thereby the execution order, this request must be synchronous. After receiving the answer, the WASM file decodes the serialization string and instantiates variables or objects respectively.

3.4.3. Access HTML DOM with WASM code

As representing the client-sided core, the WASM file needs access to the HTML DOM. To have a smoother or more pleasant GUI, WASM should be able to change it dynamical. But a page reload for every alteration in the GUI's depiction would break with the common style of a web application. Another important point of DOM access is the retrieval of information, as for example via an text input field, where the user was motivated to fill in according information. So the DOM access is needed to either manipulate the displayed items, their style or the layout, or to retrieve values.

The access is generated by a similar construct as the client/server communication. So the C++ code, which should be compiled to WASM, was edited in order to access the new UI with the same semantic as the old desktop UI. This is depicted in a method, which accesses JS with the emscripten construct of writing JS code in C++. This code is then calling the related function in the ReWaMP JS code, which is then finally accessing the DOM accordingly.

3.4.4. Handling UI events in WASM code

In all GUIs a user can interact with the elements, and thus those interactions need to cause an underlying functionality. Because it is important to produce the desired result or rather behave as expected from the user. In the thesis' context this means that the new web UI needs to know what function to call when the user clicks on a specific button.

The solution of this challenge is depicted via the export possibility of WASM. Because as WASM is able to include functions, what was used in the DOM access and the client/server communication, it can also export functionality to JS. The solution in the ReWaMP runtime is that the WASM file exports the desired functionality. Further an initialization method within the ReWaMP JS code is then configuring the HTML to transmit the user interactions correctly.

There are a few possibilities how to export functionality from WASM to JS, and finally I decided to use one, where the object orientation of typical C++ code is included. The exported functions are in the thesis' scenario used desktop UI library always bound to the class, which is described in the main file, and was responsible for the interaction with the view.

3.5. Additional Notes according ReWaMP

Besides the presentation of the top-level approach and the ReWaMP process, a few other topics belong also to this chapter, but are rather additions. The point of view in this section is from the thesis' core context, and thus from the ReWaMP process and the tool selection of using WASM at rapid web migration prototyping.

At first the next subsection will describe the conceptual decisions by using WASM in the approach. Next, some additional notes about the server-side in accordance to the ReWaMP architecture will be named and presented at subsection 3.5.2. Thereby will come up also some ideas for pursuing the first rapid web migration prototype in further iteration cycles. The section will then be closed with additions about the expectations of the ReWaMP process in accordance to the new web UI at subsection 3.5.3.

3.5.1. Conceptual Decisions using WASM

Since one integral part of this thesis is how to use WASM at rapid web migration prototyping, this subsection presents the conceptual decisions upon the questions raised at the concept designing. Their answering was leading to fundamental decision regarding the concept. These questions are as follows:

- How to use WASM in a web migration prototype?
- What need be considered by using WASM at web migration prototyping?
- How to add external functionality to a WASM file program sequence?

The questions hold also a chronological sequence within their order. Whereby the first two questions raised directly at the very beginning of thinking about the topic of this thesis, the third question in turn raised later and as a deduction by answering especially the first question. To answer those questions, this subsection is further divided in three parts.

Using WASM in a rapid web migration prototype

The decision where WASM will be used within the architecture of a rapid web migration prototype is self-explanatory, and clearly settled down at the client-side. Because WASM's main purpose is to bring code written in other languages as C or C++ in the browser with similar performance as in their normal execution environment.

In accordance how to use WASM, it is questionable what functionality WASM should take care of. As known from the ReWaMP process, WASM is thereby used as substitute for the view behavior code, which is in common web applications written in JS. The idea behind is that this information is already written in the legacy code, and thus can be reused after a transformation into a WASM-compilable format.

This decision is a clear step in direction to rapid prototyping and its usage in agile development, because a functioning prototype is important for the early feedback [19]. As most stakeholders, who should be enhanced to give feedback, are non-technicals or rather managers, the first prototype needs to focus on a well performing GUI rather than technical descriptions as back-end code or architecture and sequence flow diagrams [12].

In addition, the decision of legacy code at the client-side introduces something new to web migration. Because also if some approaches presented in section 2.2 divide the legacy code in UI part and the legacy's rest with business logic and data access, they are not dividing the code even further between client- and server-side. This is possibly the fact due to the focus on web migration and rather creating a rapid prototype. Even the closest approaches with separating the UI part from the remaining legacy code re-engineered the whole GUI, and thus the view behavior code.

Considerations using WASM in a rapid web migration prototype

An approach using WASM should always consider that there could still be things changing in the design. This could happen because WASM is relatively new and still under active development. But the change of basic design decisions is quite improbable as WASM has already released its first MVP, and evolved by the purpose to create an open web standard. Furthermore, it is created first via the emscripten compiler, which was prior the compiler for asm.js.

Furthermore, the compilation itself can be problematic as WASM will be used at a rapid prototype, and thus it needs to be rapidly changeable. Changing code fragments in the code of any WASM file in an iteration cycle means also to compile the file again. But this is solved at ReWaMP, as the to WASM compiled code is in the ReWaMP

process only a small part for each file. Thus, the compilation should be insignificantly time-consuming.

Another aspect of using WASM is the compilability of the code, which should be compiled into a WASM file. As already mentioned in the ReWaMP process in section 3.3 the code parts in the related files for one WASM file need to be transformed from the legacy format to a WASM-compilable format. Consequently, this transformation needs to be a part of the approach as it is in the ReWaMP process. But as the transformation results are compiled, no fragments are allowed to be undefined in the code. Thus, there is except of the client/server architecture and communication a need of adding functionality to the WASM file. This functionality is not in the WASM file, but in an external context. Thus, it is implemented in JS or even at the server.

Adding external functionality to WASM

The last two questions clearly stated, that the WASM file necessarily add external functionality. So functionality, which is defined in an outer scope of the WASM file. Therefore, a few strategies are possible. As the functionality adding is clearly only restricted to functions, figure 3.4 is showing the decision tree that arose by designing the ReWaMP process.

The functionality adding can be limited to functions, because all other constructs as classes, arrays, types, variables and etc. can be re-engineered detailed enough for compilation and processing. This refers further only to empty functions in the expert file, since their definition is at the point of view from the related files unknown. Thus, their functionality would be missed at the WASM file, or they even kill the process by resultant exceptions.

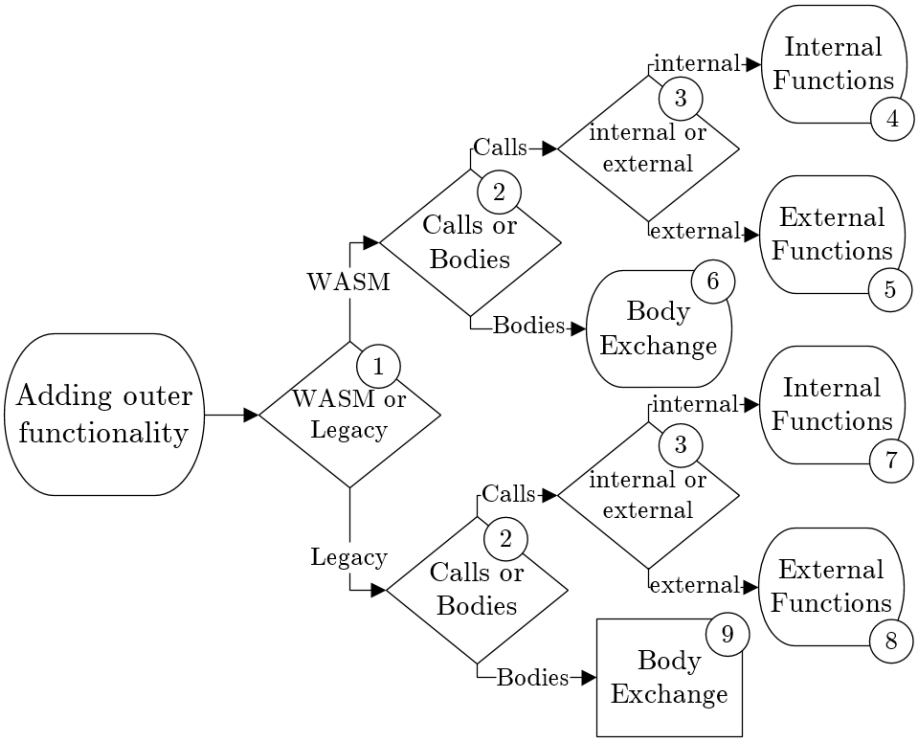


Figure 3.4.: Decision tree regarding adding external functionality to WASM

The decision tree is depicted as a decision flow diagram. The root is thereby the start and the leaves are depicted as end nodes. One leaf is not a typical start or end marker, but a task. Hereby the strategy, which was taken for the ReWaMP process is marked. The symbolic of a task assumes, that there will be further changes according to this leaf, whereby the other leaves are an end point as they will not be realized.

At first, it needed to be decided whether the changes at the approach should be made at the legacy code or after the compilation to WASM. This decision is tagged with the marker 1. The editing in the legacy code is imaginable as it is a typical source code and thus human-readable, but WASM is in a binary format and thus the parsing and changing could be very difficult in this format. Yet WASM should also be read- and writable for humans by design, and thus also a text format of WASM was developed, which is describing the binary format via S-expressions. If making changes at the text format, correspondingly the text format needs be compiled again into WASM.

Next, it is necessary to decide at marker 2 if the calls to the unknown functions or the bodies of those should be exchanged. Thereby it is irrelevant if the choice before was WASM or the legacy code, because both know the construct of functions and their calls. Exchanging the calls means thereby to parse the whole code and exchange

calls of unknown functions with a call to a special function. This special function can then send the original call to the outer scope and return the value respectively. Hereby a special function for each used return type of unknown functions in the related files is needed. Exchanging the bodies on the other side means that every unknown function needs be declared, but its body will just call another function, which will get the desired results from outer scope.

If the second decision is to exchange the calls, it is further to decide at marker 3 if the new called function should be internal or external. The internal choice would mean, that the special function for every return type will be defined within the inner scope. Respectively, the external choice would mean that all calls would get exchanged, but the functions will get imported. C/C++ as well as WASM can import functions.

Deciding to use the WASM path in the decision tree would mean to write S-expressions or rather the text format of WASM. Thereby a knowledge base of the whole code need to be generated by reading or parsing the text format. At least the knowledge about global type definitions of WASM, and where the unknown functions are, is necessary. The global type definitions represent also the types of a function, which are in WASM not only defined via their return type. They are defined as a combo of the function's return type and its parameter types. The understanding of the global types is important as they are all getting a numerical index, and the numbering is not standardized and can vary at compilations of different sources.

If the calls should be exchanged as an internal at marker 4 or if the bodies should be exchanged at marker 6, all special functions need to be written in the text format of WASM. This would mean to parse and source to source transform the text format. Such an approach needs a tool, which is nearby as powerful as emscripten is. Leaving this task completely the migration engineer would add more work, respectively more time and a higher qualification level.

On the other side if it is an external at marker 5, the function will be respectively imported by the WASM file. But still the calls need to be exchanged, and to write the new one again the global type knowledge is needed, and how to write such imports in WASM. Thus, again a source to source transformation of the WASM text format would be needed.

Another general counterargument of editing the text format of WASM is that the back compilation after editing the text format infers a second compilation circle a second time, as WASM uses as already mentioned indexes. So after editing, it needed to be compiled to the binary format, back to the text format, some indexes need to be adjusted and then it can be compiled back to WASM, because the indexes at the top space will be optimized when compiling it the first time back to WASM. But

then the indexes used within a function body mostly get wrong as they will not get changed by the compilation. Thus, the WASM file would not work as expected.

So adding external functionality seems overall more suitable by adding it at the legacy code. Further the transformation of the legacy code parts to a WASM-compilable format requires already editing at the related files. The final choice of mine was to use body exchange at the related files with marker 9. But also at editing the legacy code it would be possible to exchange the calls, and again with an internal or external function, which are marked with 7 and 8. Thereby the external function imports of 8 would lead in the result the same imports in WASM as done at marker 5.

The body exchange in the legacy is thereby the only good possibility to add further customization to the unknown functions. For example a function in the legacy code does not need to change only global variables or one variable via its return. It is also possible to change the parameters, which may also be accessible before and after the function processing as they could be allocated at the stack.

3.5.2. Server-side Application

The REST API of the top-level approach is built by using the MockAPI as shown in action 3 of figure 3.1. The correct controller behind one specific URL address is covered by the MockAPI, but it further needs to understand the syntax of the JSON strings used at the ReWaMP runtime. These are used in the ReWaMP JS code and in the serialization at the WASM file. Only then it is able to process correctly as requested and also answer in the same syntax as assumed by the ReWaMP runtime.

In beyond, the legacy code can be further reused to perform the server-side actions of business logic and data access. The server therefor can enhance the dynamic loading of the legacy code for instance with CGI or use another method like presented in section 2.2. The therefor needed wrapping and enabling of the legacy code at a server is clearly described in many related works at subsection 2.2.4. Further there are also approaches under 'Relocation and Environment Disguise', which are also dividing the legacy code between the view layer and the rest. Any of such an approach with reengineering at least the GUI could also be used within ReWaMP to enhance the prototype in further iteration cycles.

3.5.3. Reconstructing GUI

At constructing and building the new web UI, the ReWaMP process assumes also to have a resulting GUI, which is quite similar to the legacy's. So it assumes that

the ids of the old GUI are reused and grant correct access. Thereby the structural redesign should not change much. A button should for example stay a button for the very first web migration prototype.

To weaken this hard assumption a bit, ReWaMP also tries to approach in some examples. If for instance in the legacy code a variable is filled with a string received from the GUI, the ReWaMP JS code will get the string value either by accessing the value attribute or the inner text. The validation how to access the requested value is made via the accessed HTML element's tag. While a input field should be accessed via its value attribute, a div should rather be accessed by reading the inner text.

In addition, the UI is also enabled by the top-level concept to enhance in a future iteration cycle. It is possible to re-engineer the UI further in a modern web style and edit older UI constructions. This would break with the assumed access to the HTML of the ReWaMP runtime, but therefor only the ReWaMP JS code needs be edited in the related iteration. Thereby all HTML accesses, who change, need to be mapped accordingly to the new element. The old ids can change, but if also need be mapped.

4. Implementation

In section 3.3 the ReWaMP process was presented. Thereby WASM-T, a transformation tool supporting the migration engineer, was introduced, and its tasks were described on a conceptual level. Its prototypical version is developed with a close relation to the scenario presented at section 1.4. Thus, it expects the MFC library as used desktop UI library and a typical dialog based UI structure, which was defined via RC files.

This chapter will present WASM-T with more technical details and how the prototypical version was implemented. It is therefore divided in four parts. At first section 4.1 will give an overview about the architecture of WASM-T and its components. Next, the two main components, the C++ parser and the ReWaMP processor, will be further described in section 4.2 and section 4.3.

4.1. WASM-T Architecture

WASM-T is designed with its architecture to work as expected within the ReWaMP process. Python is the used programming language for the prototypical implementation. One useful advantage therewith are the later used python bindings for libclang, but this is explained with more detail in section 4.2.

The architecture of WASM-T consists as shown in figure 4.1 of following three internal components:

- ReWaMP Processor
- C++ Parser
- Configuration

Thereby the ReWaMP processor and the C++ parser are the main components, which are also responsible for the program flow. The configuration is thereby only a side component for contextual correct processing. It is used by the migration engineer at action 2 in figure 3.3 to configure WASM-T correctly. Consequently, it includes the path to all files, which were extracted at action 1 of the ReWaMP process. Further

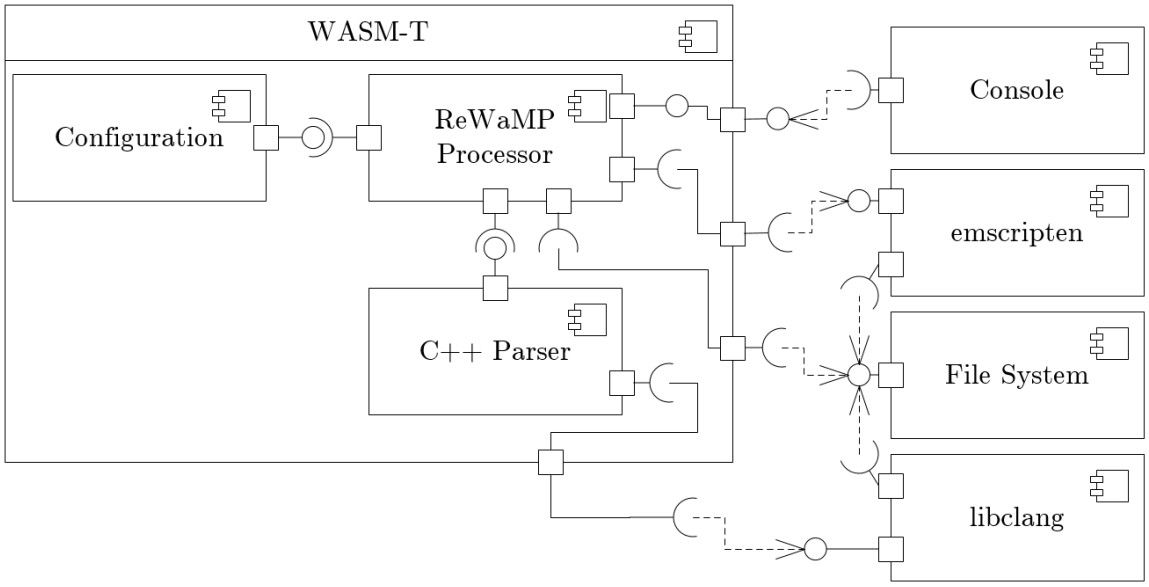


Figure 4.1.: WASM-T architecture as UML component diagram

it contains the list of all main files, which are the access point of one WASM file, and the name of the expert file. Besides that, it also contains the paths for the used tools, libclang and emscripten, and the information about which C++ includes should be added to each main file.

The C++ parser component is responsible for parsing the C++ files with abstracting semantic knowledge. Therefore, it provides classes and functions to save this knowledge in a proper structure, and enable further work with them. The parser is accessed at runtime by the ReWaMP processor, and thus is a custom API of libclang and its python bindings for the ReWaMP process. Because the actual parsing is not done within the C++ parser, but by libclang. Detailed information about the parser and its implementation will be provided in the next section.

The ReWaMP processor is the core component of WASM-T and implements the algorithm of the code transformation. As WASM-T is a script and the ReWaMP processor is implementing the main algorithm, it is also the starting point of the program flow, and thus has an interface in figure 4.1 to the console. Further it uses the emscripten compiler to compile the related files of one view to WASM, which infers from action 7 in figure 3.3. It further accesses within the ReWaMP process algorithm the configuration, the file system, and the C++ parser. The detailed information about the ReWaMP processor will be described in section 4.3.

Besides the internal components of WASM-T, figure 4.1 also shows four outer components. These are either third party tools as emscripten and libclang, which are

called by WASM-T, or are visualizations of the underlying operating system. Since the python script is started via console, the console is mirrored as a component. For the file system it is quite similar as this will provide data used by WASM-T, libclang or emscripten. So for instance it provides the extracted files of action 1 in the ReWaMP process, the according subset of these files for one specific view and the preimplemented files needed by the ReWaMP processor.

4.2. C++ Parser

Compiling extracted files of the legacy code to WASM needs a transformation as stated in section 3.3. To edit the code correctly, input files must be parsed to recognize fragments, which need to be transformed. Parsing code is often done using a naive approach with regular expressions. In cause of the C pre-processor, C++ code is often sticked with pre-processing macros, and thus parsing C/C++ code can be difficult by just using regular expressions or even lead to wrong assumptions.

In order that the parsing of WASM-T is not clearly lacking by possible wrong interpreted entities, the C++ parser component was designed and implemented. To circumvent the mentioned problem with possible wrong interpretations, libclang is used as parsing technology. The C++ parser is then only putting on top of the python bindings of libclang some abstracted functions and classes to provide an easier to use API.

libclang [39] is the C API of clang [37] to parse source code into an abstract syntax tree (AST), and traverse this AST accordingly. It further also is intended to link elements of one AST with the specific source code location and vice versa. clang itself is part of the LLVM compiler project and creates a C-based language front-end for programming languages of the LLVM project, thus also C and C++. Advantageously the clang project also implemented python bindings [38] to use libclang via python, thus the C API is also accessible for WASM-T.

This section has the purpose to present the C++ parser component on a detailed level. Therefore, the section continues with presenting the classes, which are provided by the C++ parser component in subsection 4.2.1. Next and section closing, the API functionality with the according functions will be presented in subsection 4.2.2.

4.2.1. Classes

The classes of the C++ parser were introduced to give the exported information a better human-readable form and thus a more abstracted handling of them within the

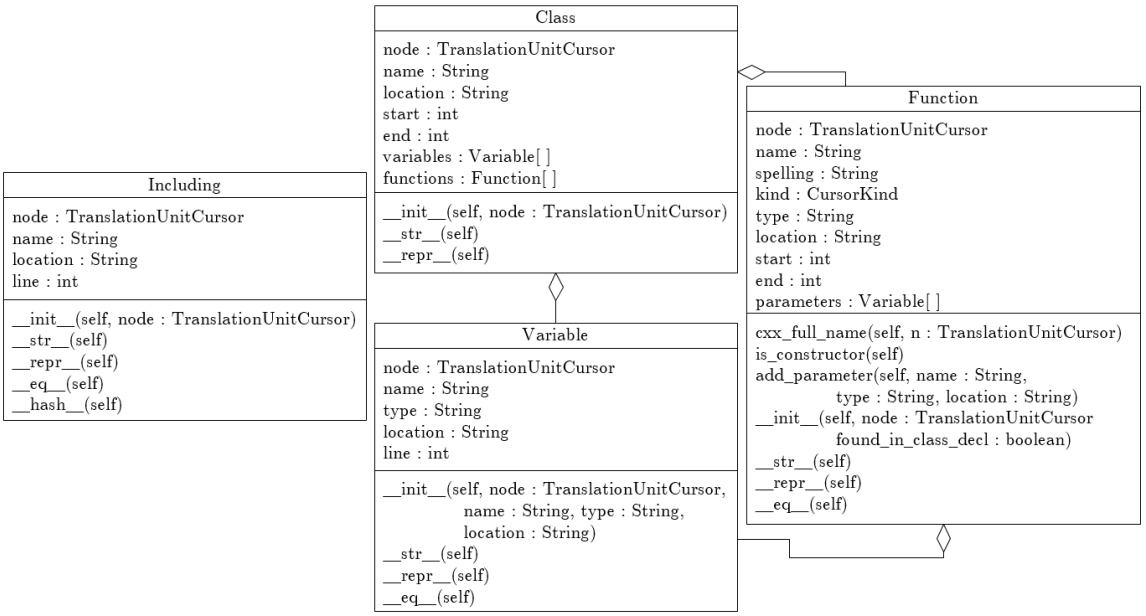


Figure 4.2.: C++ parser's UML class diagram

ReWaMP processor. The needed classes are mostly representing the typical object-orientated pattern of programming, yet at the current prototypical version in a naive form. Additional, the includings of C/C++ also got an own class. Namely the classes are:

- Including
- Variable
- Function
- Class

For a better overview, they are also illustrated as a UML class diagram in figure 4.2. The visible diagram is pythonic, as the visibility of attributes and methods is ignored and every method includes a non further specified parameter called *self*. Python does not need a visibility within classes as *public* or *private* in C++. The *self* argument whereas is a reference to the class, which is implementing the function, and by convention named *self*. The aggregations further are only showing that an object has a reference to objects of another class, which in turn has not. As python is also not in need of any list size at the initialization, the class diagram also forgo cardinally numbers at the aggregation ends.

All classes in the diagram have in common, that one attribute is called *node*, which has the type *TranslationUnitCursor*. In addition, they all have the python typical methods __str__() and __repr__() for a string representation. The mentioned attribute is hereby holding the reference to the corresponding AST. As the type's name terms, it is a cursor of the translation unit. The translation unit is thereby the root of the AST, which was the result of parsing one view's main file via libclang. The actual *node* saves the pointer, which points in the semantic of libclang's python bindings to an element in the AST. This element is further modeled by the specific object of any class in the C++ parser, and thus saved to hold the reference.

The class *Including* is modeling C/C++ typical includes of other files. Beyond the already mentioned attribute *node*, this class is saving in addition the *name* of the include, the *location* and the *line*. The *location* is hereby the full path of the file, where the include is written. Accordingly, the *line* is the line number in the *location*. Further the class has beyond the string representation methods, a constructor method, which is called __init__() in python, and the methods __eq__() and __hash__(). The constructor fills thereby the attributes by accessing the node parameter, which will become the *node* attribute. The other two methods are overwriting python standard functions and enable thereby comparison of *Including* objects.

The *Variable* class is modeling variables, attributes of classes and parameters of functions, because all of them have in common to have a name, a type, a location and a line, equally are the class' attributes named. While the most attributes were already mentioned at the *Including* class, the *type* attribute is just the type of the variable in string format. Beyond the class has the same methods as the *Including* class, but without the hash function. It is additionally possible to call the constructor with the typical node process or via the direct three parameters *name*, *type* and *location*, which will fill the eponymous attributes.

Further on, the class *Function* is modeling C++ functions of any libclang type. libclang is distinguishing between constructors, deconstructors, class methods and functions without a class or name space subscription. The attributes list is overlapping in many elements with the other two already described classes, but some are also changing. Thus, the line attribute is here divided in the *start* and *end* attributes as a function is assumed to be written in more than one line. The *spelling* attribute is also the name as the *name* attribute, but the *name* will be the full name of the function with all C++ name spaces. So for example, a function *hello* is part of the class *test* in a C++ code. The *Function* object of *hello* will then have the *name* TEST::HELLO, and the *spelling* HELLO. It has further also the variables *parameters* and *kind*. While *parameters* is a list of the functions' parameters as *Variable*, *kind* is the string of the function type parsed by libclang. The methods behave as in the

other classes, generating the full name as it will be saved in *name*, checking if the function is a constructor, or adding a parameter.

Lastly, the class *Class* in the diagram is modeling C++ classes. It has beside the already mentioned attributes *variables* and *functions*. These are lists of *Variable* or *Function* objects and model the attributes and methods, who are declared in the *Class*. The functions of the class are the contextual same as in the previous described classes.

4.2.2. Functions

The functions of the C++ parser are fulfilling a need of information at the ReWaMP processor by processing one view in the context of the ReWaMP process. But talking about all functions of the C++ parser would be out of the thesis' scope. To understand how the parser's classes are used, and how the parser itself should be used as an API, the parser's function design will be clarified in this section. Therefore, two functions will be explained completely. These two functions are also shown in listing 4.1. The other functions are mostly only variants of the presented GET_FUNCTION_DEFINITIONS() with a different context, but a closely same design.

The function GET_MAIN_CURSOR() is hereby the function to receive a translation unit cursor. This is not any cursor of the file, but the root cursor. So the method will parse the given file, create the respective AST and return the cursor, who is pointing to the root of the resulting AST. The method is therefore called with one or two parameters. One is the file's path and the other one is a boolean to decide between specific options at the parsing process. As libclang's python bindings are used, the method starts to initialize a so called index in line 2, to be able to parse a file. Line 3 is then deciding whether the options of parsing the detailed processing record should be activated or not. Accordingly to this decision, line 4 or 6 will parse the file, and create the AST. In line 7 is then finally the root cursor returned.

This method is used to parse in a view's main file as the compiler will do in later steps of the ReWaMP process to compile it to WASM. Thus, the AST is of the same structure as emscripten will process it later. The root cursor is hereby the standard access point for the AST, as it is possible to iterate over the whole AST by accessing this root cursor. The named option is modifying the parsing action, because libclang is processing the pre-compiler statements and creates afterwards the resulting AST. So to access also the pre-compiler statements by iterating over the AST, libclang needs to know that these statements should also occur in the resulting AST. This is especially used to also parse in the *include* statements, and add or delete some in accordance to the ReWaMP process.

```
1  def get_main_cursor(file_path, pre=False):
2    index = clang.cindex.Index.create()
3    if pre:
4      translation_unit = index.parse(file_path, args=['--std=c++11'],
           options=clang.cindex.TranslationUnit.
           PARSE_DETAILED_PROCESSING_RECORD)
5    else:
6      translation_unit = index.parse(file_path, args=['--std=c++11'])
7    return translation_unit.cursor
8
9  def get_function_definitions(cursor):
10   return_list = []
11   for n in cursor.walk_preorder():
12     if (n.kind == CursorKind_FUNCTION_DECL or n.kind ==
           CursorKind_CXX_METHOD or n.kind == CursorKind_CONSTRUCTOR or
           n.kind == CursorKind_DESTRUCTOR) and n.is_definition() and
           ntpath.dirname(n.location.file.name) == ntpath.dirname(cursor
           .displayname):
13       return_list.append(Function(n))
14   return return_list
```

Listing 4.1: C++ parser functions

The second function GET_FUNCTION_DEFINITIONS() is starting in the listing at line 9. Its purpose is to find all function definitions, which are within the cursor's AST. To know which cursor is meant, it is given as a parameter. To return all definitions, the corresponding element cursor within the iteration over the cursor's AST is saved as a Function object and added to a list. The list of all function definitions is in the end returned.

In line 11 the common phrase of libclang's python bindings for iterating over all child elements of the cursor. There is also a method called get_children() at every cursor, but this method is only returning a list of all direct children. Further on, in line 12 is the if-clause written for deciding if the current element should get in the return list. At first, it is thereby compared if the element is one of the python bindings' specific function types. Next, it is checked if the element is a definition via a libclang included method. Lastly of the if-clause, it is compared if the current element is within a file in the same directory as the given cursor. This helps to exclude function definitions of C++ standard libraries, because as they are included they will also be part of the AST. But with regard to the ReWaMP process standard functions are not of any interest, thus they are excluded with this comparison.

4.3. ReWaMP Processor

As already mentioned, the ReWaMP processor is implementing the actual ReWaMP process. By using different other components, the processor is therewith the core of WASM-T. This section is further divided in four subsections to present the processor and how it is performing. Subsection 4.3.1 starts with presenting the implemented algorithm, which is reflecting the tasks of WASM-T within the ReWaMP process as described in section 3.3. Next, the assumptions of the current ReWaMP processor on the legacy code are described in subsection 4.3.2, because the current implemented version would may execute wrong or stop by an exception if these are not fulfilled. Further on, subsection 4.3.3 presents the serialization of all exchanging information. Then the section closes, with the two subsections about exporting and importing functions to or from JS in subsection 4.3.4 and subsection 4.3.5.

4.3.1. ReWaMP Process Algorithm

The ReWaMP process algorithm needs to cover all tasks of WASM-T. Therefore, the ReWaMP processor implements the steps according to algorithm 1. This algorithm is executing the whole ReWaMP process with the point of view from WASM-T, thus the algorithm starts at action 3 of figure 3.3 and ends with action 7. As the migration engineer needs to perform action 4 and 5 for every view between the processing of 3 and 6, the algorithm waits also for a signal from the migration engineer between those tasks. This ensures that the actions 4 and 5 are finished, so WASM-T can continue.

The algorithm has as input the configuration component and the extracted view behavior files. All view behavior files should be copied in one directory, and the configuration component should be edited context related in the actions 1 and 2 of the ReWaMP process. The output is also already known, as the result of the ReWaMP process should be the ReWaMP runtime for each view, which consists of the ReWaMP JS code, the JS glue code and a WASM file.

The function itself starts at line 1 without any parameters, as the files will be loaded within the function, and the configuration is imported via python imports. After reading all files within the working directory in line 2, the actual process is able to start. Therefor a for-each-loop in line 3 is started, which covers the whole rest of the function. This loop is hereby running for each main file, which was defined in section 3.3 as the access point for each view. Thus, this for-each-loop is implementing the loop for every view.

Algorithm 1: ReWaMP Process Algorithm

Input: Configuration C, extracted view behavior files
Output: ReWaMP runtime for each view

```
1  function ReWaMPProcess()
2      files = readAllFilesInDir(C.workingDir)
3      foreach file_main in C.mainFiles do
4          cursor_now = getMainCursor(file_main in C.workingDir, True)
5          files_rel = getRelatedFiles(cursor_now, files)
6          files_rel = editIncludings(cursor_now, files_rel, C.headers)
7          files_rel, maps = minorChanges(files_rel)
8          dir_rel = createRelatedDir(file_main)
9          cursor_now = saveRelatedFilesAndGetMainCursor(file_main in dir_rel,
             files_rel)
10         files_rel = deleteUnusedParams(cursor_now, files_rel)
11         files_rel, bodies, funcs = editMFCFunctions(cursor_now, files_rel)
12         saveRelatedFiles(files_rel)
13         printHints(cursor_now, files_rel)
14         waitForME()
15         files_rel = loadRelatedFiles()
16         cursor_now = getMainCursor(file_main in dir_rel)
17         bodies = buildExpertFunctions(cursor_now, files_rel, bodies)
18         files_rel = addJSONMethods(cursor_now, files_rel)
19         files_rel = processFlags(files_rel)
20         cursor_now = saveRelatedFilesAndGetMainCursor(file_main in dir_rel,
             files_rel)
21         embinds, funcs = collectEmbinds(cursor_now, files_rel, maps, funcs)
22         cursor_now = saveRelatedFilesAndGetMainCursor(file_main in dir_rel,
             files_rel)
23         files_rel = addFunctions(cursor_now, files_rel, funcs)
24         cursor_now = saveRelatedFilesAndGetMainCursor(file_main in dir_rel,
             files_rel)
25         files_rel = writeNewBodies(cursor_now, files_rel, bodies)
26         files_rel = addSendStatusFunction(files_rel)
27         files_rel = addEmbinds(embinds, files_rel)
28         saveRelatedFiles(files_rel)
29         copyLibraryFiles(dir_rel)
30         addJSInitFunction(embinds, maps)
31         compileFilesToWasm(files_rel)
```

The lines 4 to 12 are implementing the first task of WASM-T, hence the view preprocessing. Line 13 and 14 are then implementing the pause, where the migration engineer has to perform actions 4 and 5 of the ReWaMP process. Continuing lines 15 to 30 implementing the action 6 of the ReWaMP process. Line 31 finishes then the for-each loop with the compilation of the view to WASM, thus represents action 7.

View Preprocessing

At first the main cursor, as explained in subsection 4.2.2, needs be received of the C++ parser component in line 4. With this in hand, it is afterwards possible to collect all related files of the current main file or rather view in one dictionary $files_{rel}$. It consists of the key value pair from the file name and its content. So every related file, which is accessed within the AST of the main cursor, is now parsed via file-stream and added to $files_{rel}$ behind its name.

Next, the C++ includings of all related files will be edited at line 6. The cursor was therefor created in line 4 with the pre option, such that the includings will be visible in the corresponding AST. All unknown includings in the related files will be deleted and several in the configuration defined headers will be added. These headers are mostly standard libraries needed after the transformation, headers of emscripten or ReWaMP defined headers. Additionally, it is possible, that the migration engineer even added more headers at action 2 to the configuration. Then he knew before starting WASM-T that the transformation will need some specific standard headers in cause of a type change or any other transformation step, and has not to include it later manually at action 4 or 5.

In line 7, it will be iterated once over all related files and thereby minor changes will be performed. These minor changes are the biggest part of the view preprocessing, as they change the most lines of code in action 3. These changes are called minor, as they do not need any semantic knowledge of the code. This is also noticeable since the only parameter is the related files dictionary. Within this method, many lines of code will be edited or even removed. Hereby also the MFC typical message maps are deleted and saved in the dictionary *maps*, where the key is again the file name and the value are the lines originally defined in the message map.

Further on, line 8 and 9 are saving the related files on the hard drive to reinitialize the main cursor. Line 8 is hereby adding a subdirectory in the working directory with the name of the main file. Subsequent it is possible to save the changes made until now at the related files to reinitialize the AST or rather the main cursor in line 9. This will further sometimes happen in the algorithm, as libclang needs access via the file system to parse in an AST, and this need an update to mirror the current

version of the related files. The new directory is only for better versioning, and due a possible doubling of some file in different views. Because then a file related to two views could be processed differently, thus the extracted version is needed at each iteration start of the for-each-loop.

Ensuing, the parameters of all functions are checked if they are really used in the function at line 10. Especially at MFC functions it is possible, that some parameters stay unused in the implementation, but are prescribed by the definition in MFC. Thus they need also to be defined at the parameter list in the legacy code. But after extracting the files such unused parameters are not anymore needed, and are in the most cases even more work for the migration engineer at his manual changes. To prevent this overhead of manual work, these parameters are removed in the definition, and if available, also in the declaration of the function, e.g. within a class.

After that, all well-known MFC methods are edited. Thereby they are all transformed from their typical standard behavior within MFC into a form such that they are performing with the same semantic as before, but without any MFC patterns. The bodies are thereby not always changed directly, because if the changes will add emscripten defined code fragments, libclang would not be able to correctly parse the file anymore. Thus, these bodies are saved with the changes in a new dictionary called *bodies*. Furthermore, all later to JS exported functions are added as headers to the dictionary $funcs$ to add their declarations later as public to their the class.

To give an example, the DoDataExchange() method is using mostly calls to other standard MFC functions. These calls are implementing which variable in the dialog class should be bound to a specific UI element, and thus exchange their data. As it is decided in MFC via a boolean parameter which direction of the exchange should be performed, the function is transformed to a big if-clause. This clause is then deciding with the boolean in which direction the exchange should be performed. And each old call is transformed in an exchange with the new UI for both directions.

Next, the files are saved in line 12 in the related directory for pausing the algorithm. Line 13 is printing some hints for the migration engineer to the console, before the pause starts. Therewith the migration engineer has a possible starting point of making changes. These hints are describing currently only declarations of variables, which types are unknown for the parser. Thus, they must be missing in the related files.

Transformation Completion

After the migration engineer continued the algorithm, the related files will be loaded in line 15 from the hard drive to bring in the engineer's changes. Next, the main

cursor will also be updated with the new files in line 16. Then the algorithm can continue with the action 6 of completing the transformation.

The algorithm builds then in line 17 the expert functions. These are the functions, which are written in the added expert file. As they are mostly without a body, these functions will be filled with the code to send the function call to the server and retrieve the possible result back to the WASM instance. If one expert function is flagged with the ReWaMP flag *real_func*, the function will be correspondingly skipped. By adding the transmission of the call to the server, the serialization of the data to JSON and the afterwards initialization of the results are also added to the body. Hence, the complete function body will be created. The bodies are again not filled directly in the related files, but in the dictionary *bodies*, because there will again be emscripten code fragments in them.

Line 18 will add the JSON methods needed to perform the serialization at each call from WASM to JS. This means that all declared classes are getting one TO_JSON() and one FROM_JSON() method. Further details upon the serialization will be presented in subsection 4.3.3.

After adding the JSON methods, the ReWaMP flags will be processed in line 19. As a reminder, the ReWaMP flags are comments with a special syntax, which are introduced to support the migration engineer even further. He is therewith enabled to prevent or execute specific processes of WASM-T at a specific code fragment. These flags always start with *//RWMPFLAG:* and then continue with the actual flag name. In the current version of WASM-T is only the flag *strcpy2string* parsed in this flag process method. But future versions could also add more cases of typical code, and thus add new flags. A second already implemented flag is the *real_func* flag, but this is used at building the expert functions in line 17. Thus, it is not processed at the method in line 19.

The *strcpy2string* flag is hereby exchanging the well-known method STRCPY() with a assignment via the '=' operator. This step is often needed, when a char* was copied into a CString. Because the CString is transformed at the minor changes in line 7 to a std::string. And the std::string is not allowed to be a parameter of STRCPY().

Next, the related files have again to be saved, as the main cursor has to be refreshed after the last changes. Subsequently, the emscripten bindings are collected in line 21. The collected information is saved in the dictionary *embinds*, and is later used in line 27 to write the bindings. The emsripten bindings are responsible for the export of functions, classes or variables of the resulting WASM file. Thus, they describe which parts of the code should be accessible by JS, when the WASM file was loaded and initialized as a module in a browser session. These bindings are again not written directly to the main file, as they would again break the parsing of libclang.

In beyond, the cursor has again to be refreshed with saving the related files before, as the gathering of the emscripten bindings is deleting function declarations at the class definition. This is needed, as all exported functions need be public accessible. These methods need to be added again, and thus the dictionary $funcs$ will be updated accordingly in line 21.

In line 23 all in $funcs$ saved function declarations are getting finally added to their classes. Thereby it is ensured, that all of them are from now really public accessible. As hereby also changes occur in the related files, they have again to be saved and the main cursor also again refreshed, but for the last time.

After that, all new bodies, which where saved in *bodies*, are written in line 24 to their functions. Thereby the main cursor will be used the last time in the for-each-loop, and thus the last time in the algorithm. It is mainly used to get the correct indexes of the functions, where new bodies should be parsed in. After adding them, the cursor will also not be able to parse in the corresponding AST, because as already described libclang will break its parsing at emscripten code fragments.

Before saving the related files for the last time in this algorithm, the emscripten bindings need to be added in line 27 to the end of the main file, and one specific function need to be added in line 26. The specific function is called SendStatusToServer() and is necessary for closing the current view. Because in the legacy, or rather in MFC, the view would be closed by specific actions of the user. Thereby often data of the dialog are sent to the business logic or also to another dialog. For preserving the old behavior, the function SendStatusToServer() is introduced by the ReWaMP processor. This sends a status in form of a string and the serialized object of the dialog class to the server.

Next, the library files can be copied to the related directory in line 29. In the current version, the library files are *rwmp.h*, *rwmp.js* and *json.hpp*. *rwmp.h* is hereby the ReWaMP C++ library, which is adding some standard functions to C++ for the access of the resulting WASM instance to the new web UI. For example, GetFocus() which returns the id of the current focused element at the GUI. *rwmp.js* is the already known ReWaMP JS code file with the JS functions, which will be accessed by the WASM file if it accesses the JS API. And *json.hpp* is the library, which is used for adding the JSON serialization to the C++ code.

The library file *rwmp.js* is lacking in this state still of the initialization method. This will be added respectively in line 30. It will be called by the HTML file to initialize the new web UI correctly with the WASM file, and bind the correct events to the appropriate UI elements. As this function is context related to the emscripten bindings and the message mappings of the specific dialog class, the function needs to be written for each view.

Finally, the for-each loop is closing with the compiling of the transformed files to WASM in line 31. After finishing also the for-each loop all view behaviors are available in the WASM format, and can be used in a first rapid web migration prototype.

4.3.2. Assumptions on Legacy Code and Environment

This section is presenting with Table 4.1 the current assumptions on the legacy code and the environment of the ReWaMP runtime. These have been made actively by implementing WASM-T. As the C++ parser or rather libclang is not gathering all possible information about a code fragment, it is not able to return all needed information for the ReWaMP process. Thus, some assumptions on the legacy code must apply to process correctly. As the current version of WASM-T is only prototypical, the assumptions can shrink in future versions. On the other side it is also possible that some assumptions on the code were made implicit by missing knowledge or missing examples of possible C++ code fragments.

Detailing all current assumptions would get out of the chapter's scope. But for example the first assumption *Function Body* is needed for parsing in function bodies and especially for writing lines in a body. Or for example the limiting assumption *No long long* is needed, as JS is not able to represent their full spectrum. And the *::GetFocus* assumption is only assumed for the execution of the current minorChanges() method in line 7 of algorithm 1.

Assumption	Description
Function Body	All functions start with one line as the header. Next, one line is possible starting with ':', but then the body starts. The first line of the body is only '{' and the last only '}'
1 Return	All functions have only one return value. This infers that no changed parameter is needed after function's execution.
UI IDs	The web UI uses the old MFC ids. Not only the string versions, but also the integers in *data-rwmpId*.
File Size	All extracted files are relatively small and the number of extracted files is also manageable.
Main Space	To find the name space of the view, the main file's name can be used. Searching all class names if including this name will match only at one class.
Class Methods	All class functions are only declared in the class definition, but not defined.
1 Name/Class	All class methods have a unique name inside of one class. Overriding functions by parameter lists is not present.
No long long	No variables of the type long long or long double need to be exchanged with JS.
Collections	Any collection as vector or map does not contain pointers of non-fundamental types and need be exchanged with JS.
Pointers & Vectors	If a vector is needs to be transferred from or to JS, there is only one pointer in the type. Types as 'vector<int*>*' thus are not needed at exchange.
Public Class Fields	All variables declared in a class are public accessible without getter or setter.
1 DoDataExchange	There is only one method DoDataExchange() in one view, and it is located in the main file. Further the only parameter is always called as in the MFC definition 'pDX'.
::GetFocus	In legacy code '::GetFocus()==' or '::GetFocus()!=' is only used in if-clauses.
.Format	'.Format' only appears at CString variables with '%d'.

Table 4.1.: Assumptions

4.3.3. WASM-T's JSON Serialization

The serialization of data was already mentioned as a solved challenge in section 3.4. Thereby it was explained, that all non-fundamental types, which will be exchanged with JS will be dumped in a string, which is to be more precise a JSON string. Further these JSONified strings are only needed at sending a request to the server, as the needed data of a HTML are always fundamental types.

In subsection 4.3.1 it was also described that every class within the related files are getting methods to create corresponding JSON objects. These methods are implemented regarding the used JSON C++ library of Niels Lohmann [30]. Therewith it is possible to create a JSON object out of all classes used in the related files and vice versa. The used JSON library also provides a method to dump a JSON object into a string and to parse in a JSONified string into a JSON object.

Furthermore, it must be sent one big string to the server at one request, because one Hypertext Transfer Protocol (HTTP) request has only one body. Each request is thereby generating its own JSON object, because the server needs be able to interpret the request. As already mentioned, the WASM file will only send function calls to the server. Thus, the built JSON syntax for passing strings between the client and the object can be limited to functions.

The JSON syntax need consequently more information than only a list of all parameters in the function. The current used JSON syntax is quite simple and represent the function based syntax. To give an example for the syntax of sending a request to sever and one for receiving an answer from it, the examples are listed in listing 4.2 and listing 4.3.

So when the WASM file wants to execute the example expert function void SaveBox(Box box), the call will be sent to the server. Thereby, the JSON with the syntax of listing 4.2 is created to serialize the function call. Line 2 saves the called function name, as the server needs to know which function was called. The rest of JSON string is filled with one big JSON object referenced by *arguments* in line 3. Next, the parameter *box* is described from line 4 to 16. As it was a non-fundamental type and rather an object than a variable, it is described via an own JSON object. The syntax hereby is again started with the name, which is in this case the type's name. Under *variables* all variables are listed, and if there would be again a non-fundamental type it would be again a respective JSON object.

Listing 4.3 is further showing an answer, but not on the same function. It is showing the server answer on the call of Box GetBox(). For example purposes it returns the same Box object as sent before. Thereby, the main difference is easier to notice. The return value is returned under the tag name *return* in a JSON object called

```
{
  "name": "SaveBox",
  "arguments": {
    "box": {
      "name": "Box",
      "variables": {
        "length": 10,
        "width": 7.0,
        "height": 42,
        "title": "Testbox",
        "tags": [
          "test",
          "box"
        ]
      }
    }
  }
}
```

Listing 4.2: JSON syntax to server

```
{
  "name": "GetBox",
  "returns": {
    "return": {
      "name": "Box",
      "variables": {
        "length": 10,
        "width": 7.0,
        "height": 42,
        "title": "Testbox",
        "tags": [
          "test",
          "box"
        ]
      }
    }
  }
}
```

Listing 4.3: JSON syntax from server

returns. At first this might seem overpowered with the assumption of having only one return value at a function. But this is already the preparation to eliminate this assumption, because any other return value can be serialized to JSON and saved in *returns*. The client can then assign after receiving the answer also more variables, by iterating over *returns*.

4.3.4. Export Functions to JS

Subsection 4.3.1 already talked about the emscripten bindings, and explained that they are used to exporting functions from the view behavior files or rather the WASM file to JS. So these bindings define which legacy functions are accessible for JS, and thus solve the challenge of handling UI events in WASM of subsection 3.4.4. This is for example needed, if a user presses a button, and it is assumed that this will change something in the UI. Because then the click event in JS must call a function of the view behavior code. This is at the created ReWaMP runtime respectively in a WASM file. Thus, the JS VM must be able to call the according function in the WASM file.

JS needs to construct this object of the view class, and call later the respective functions with referencing the constructed object. Because all main files consist in the scenario with MFC of a context-related dialog class object, which is also the defining the view's object. This object also saves often data from the GUI, as JS

```
1 EMSCRIPTEN_BINDINGS(rwmp_binds){
2   using namespace emscripten;
3   class_<Cal01_View>("Cal01_View")
4     .constructor()
5     .function("OnOK", &Cal01_View::OnOK)
6     .function("OnBnClickedButtonAdddate", &Cal01_View::
          OnBnClickedButtonAdddate)
7     ;
8 }
```

Listing 4.4: Embind Example

does in the context of common HTML and JS usages. So all exported functions are non-static class methods.

Besides the standard API functions of the JS glue code, emscripten offers different possibilities to export specific functions to JS. The object-orientated access reduces this to two possibilities, either Embind [41] or WebIDL Binder [42]. Both tools operate at different levels, but none of them is better at performance, and both are used successfully in a number of projects [43].

The used tool in the ReWaMP process is Embind. It uses the EMSCRIPTEN_BINDINGS() block at the end of the C++ code to define the according exports. An example for this block is shown in listing 4.4, which was written at the end of one main file at processing the scenario. To shorten this example and eliminate redundancies, the listing shows an excerpt of the full block.

In the listing lines 1 and 2 are initialization steps with giving the block a name, here *rwmp_binds*. Line 3 then starts to describe the class *Cal01_View*, which also gets a name in the parentheses. For homogeneity, it is the same name as in the legacy code. Next, line 4 is defining, that the class has an constructor without any parameters. This line is automatically binding to the constructor without parameters of the class. If it was only a constructor with parameters defined in the class, this line would respectively raise an error at compilation. The other lines 5 and 6 are adding two functions, who should also be exported. Thereby they get a name, which identifies them in JS, and the bound function.

4.3.5. Import Functions from JS

The ReWaMP architecture earmarks that the WASM file will be able to call functions written in JS. This also solves the two challenges of client/server communication and the HTML DOM access, which were defined in subsection 3.4.2 and subsection 3.4.3.

```
1 std::string rs = (char*)EM_ASM_INT({
2   var data_string = UTF8ToString($0);
3   var return_string = send_server_function(data_string);
4   var lengthBytes = lengthBytesUTF8(return_string)+1;
5   var stringOnWasmHeap = _malloc(lengthBytes);
6   stringToUTF8(return_string, stringOnWasmHeap,
        lengthBytes+1);
7   return stringOnWasmHeap;
8 },cha);
```

Listing 4.5: EM_ASM_INT() Example

To import functions from JS into WASM, it is also possible to call it in the legacy code via an emscripten construct. There are again several possibilities, but this is the most promising one, as it does not need a context-related configuration for the API creation. With this in mind, two possibilities of writing the JS code are left, emscripten_run_script() or the *inline javascript* method of EM_ASM(). As EM_ASM() is the faster one of them [40], it was chosen for the ReWaMP process.

EM_ASM() enforces emscripten to bring the in itself written JS code as a new function into the JS glue code and let the WASM file call this new function and therefore import it. As it is a method, it also can have a return type and parameters. Currently there are the variants: EM_ASM() for no return and no parameter, EM_ASM_() for no return with parameters, EM_ASM_INT() for integer returns with parameters and EM_ASM_DOUBLE() for double returns with parameters. As the integer and double variants have no variant without parameters, there must but at least one parameter defined. But this can be a standard as 0 and without usage if no parameter is needed.

Listing 4.5 shows a short example of the usage in an expert function after transformation completion. Hereby the integer variant is used and the example also shows, how to return a std::string by only using these four variants. For C++ the whole listing is thereby one line, but for a better understanding this was formatted also with the syntax of JS in mind.

The first and the last line are C++, while the lines in between, so 2 to 7, are JS. In the last line, it is also visible, that the construct takes at this example one argument, which is called *cha*. This variable is declared as a char array, and thus has the type char*. Line 2 shows then how to access parameters within an EM_ASM(), as it transfers the char array as a string into one JS variable. So all parameters are getting a number behind the dollar sign.

Next, the actual function is called in and the return is saved in a variable in line 3. Lines 4 to 6 are assuming, that the returned value is a string, and thus can and need be allocated at the memory of the WASM file. After allocating and value assigning

the string, the pointer to it in the memory is returned at line 7. Line 1 indicates that this is strictly conversed into a variable with the type char* and is thus saved in the variable rs.

5. Evaluation

This chapter will evaluate the ReWaMP process and the first prototype related to the defined requirements in section 2.1. The chapter is therefore structured in four sections. At first, the procedure of the experiment will be explained with focus on what was done in the experiment, what was measured, how it was measured and who was participating. It also presents the guidelines for the executing researcher, how to deal with various situations at one evaluation round. Next, section 5.2 will present the results of the experiment without any inferring or assumption. These will be made in section 5.3 with accordance to the defined requirements. Thus, every requirement will be discussed and how much it is fulfilled. Summarizing all results of the discussion and the experiment, section 5.4 will then close this chapter.

5.1. Procedure

The evaluation's procedure is simulating the situation that a company wants to know whether it should migrate the presented scenario legacy of section 1.4 to the web. The company's management has commissioned a migration engineer to create a first rapid web migration prototype of the legacy. The migration engineer is hereby creating the prototype as designed in ReWaMP. Thus, he comes to the point of performing the ReWaMP process.

The conducted procedure is designed by orientating on the validation of the paper in appendix B. So in one evaluation round are again two persons involved. One is the executing researcher, who is leading the conduction, and one is the test subject, who is acting as the migration engineer during the ReWaMP process.

Before and during the whole evaluation the executing researcher did always prepare the next step for the test subject. So he opened programs, and copied backup files of each partial result during the ReWaMP process. With these preparations, he was always leading the test subject from task to task and thereby ensured the correct measurement procedure.

At first the executing researcher introduced the test subject by explaining the simulated situation and what the test subject has to do during the experiment. After explaining the scenario situation, the test subject is in the experiment, the execut-

ing researcher described the ReWaMP process by using Figure 3.1 and Figure 3.3. Thereby not the whole process was described with all details to prevent excessive demands at the migration engineer. So the actions 1 and 2 of the ReWaMP process were described with a high level of detail, but the actions 4 and 5 were only explained with information the migration engineer need to know by performing 1 and 2. So for example, the details how a function can be outsourced to the server or which exact functions are in the ReWaMP C++ library was not explained at the beginning, but before the first time of action 4.

Next, the executing researcher presented the legacy to the test subject and the test subject got a maximum of five minutes to use a bit the legacy and gather knowledge of what happens where. If the test subject has then not directly any questions, the ReWaMP process was ready to start.

To limit the time of one evaluation round the ReWaMP process was limited to the two main views of the legacy. So when action 1 was finished by the test subject, the executing researcher deleted all extracted source files of the third view, thus the according main file and those which had only a relation to this main file.

Another intervention in the ReWaMP process was established after action 7, because the code of the test subject could not compile at action 7. If this happened, the executing researcher supported the migration engineer at understanding the errors of emscripten.

After the ReWaMP process finished with the two views, a prepared demo of the possible resulting prototype was shown to the test subject. To get a feeling of the resulting prototype, another five minutes were provided to the test subject.

The evaluation procedure ended then with a questionnaire about the test subject, the functional scope and usability comparison of legacy and prototype, and about the ReWaMP process and WASM-T. Further details this are described in subsection 5.1.3.

For the evaluation always the same computer was used. It is an Intel Core i7 CPU 930 @ 2,8 GHz with 14 GB RAM on Windows 10 Education N x64. The evaluation environment consisted of four different tools. For inspecting C++ code during the ReWaMP process at actions 1, 4 and 5 CLion [23] was used to support the test subject. WASM-T was started and configured in PyCharm [24], and the prepared demo of a resulting prototype and the Google form of the questionnaire were opened in two tabs in Firefox Nightly. In addition to copy and access the correct files the windows explorer was opened with two windows, one in the working directory of the evaluation run and the other one in the legacy source code directory for copying the files in the working directory at action 1 of the ReWaMP process.

5.1.1. Guidelines

To behave in all evaluation rounds equally, the procedure earmarked some guidelines for the execution researcher. These ensured the similar procedure and are the following four:

1. Answer all questions of the test subject, but stay as neutral and objective as possible. The answer should not influence the test subject.
2. Point out all apparent syntax errors a test subject made. The note can be given to the test subject as soon as it continues in another line.
3. Evince the test subject if it is apparently on the wrong way or performs useless actions. So for example if the test subject reads in details about the data persistence implementation in action 1 instead of concentrating upon the task.
4. Help the test subject at missing programming experience or other programming issues. Thereby it does not play a role if it is related to C++, MFC or web development. Only point out common structures, the support should not influence the test subject.

5.1.2. Measurements during the ReWaMP process

During the ReWaMP process all tasks of it were measured regarding two aspects, time t and effort e. As the process has two roles involved, the measurements of the needed time and effort are determined in t_M and e_M for the time and effort of the migration engineer, and t_T and e_T for the time and effort of WASM-T.

These variables were measured for every task, and thus e.g. $t_M = t_M^1 + t_M^2 + t_M^4 + t_M^5$, where t_M^1 is the time the migration engineer or rather the test subject needed for action 1 of the ReWaMP process. All actions from 3 to 6 are in the for-each loop to process every view, thus they will be executed more than once in one evaluation round. As the process was limited to two views for reducing the overall needed time for one round, the time and effort values of actions 3 to 6 are divided further in e.g. $t_M^{4^1}$ and $t_M^{4^2}$.

While t_T was measured via the python time module, t_M was measured by the executing researcher with a stopwatch. In front of every task, it was accounted to start the timer and with the start the test subject started working. The timer was only paused, if the test subject was disturbed by anything, e.g. a ringing smart-phone, the desire of visiting a toilet or other unexpected environment disturbances. The timer was then stopped, when the subject mentioned to finished the task and the

researcher saw that nothing is left. After the first mentioning of a finished task, the researcher must also be convinced that the task was finished. Most times the researcher was able to decide this directly. Only at action 4 it happened some times that the researcher scrolled through the source code to assess if the subject finished the task or not.

The effort e_M and e_T are measured both by the executing researcher and counting the changed source lines of code (SLOC). While e_M was evaluated directly at the process, e_T was counted after the evaluation round. To count e_T after each round, all intermediate results of the source files were saved between different tasks. The measurement of e_T was supported by CLion [23], which helped the researcher to compare the different source file versions. As action 1 and 2 are not changing any SLOC, the effort measurements are only assessed at action 3 to 6.

In addition, the efforts were further separated in three categories: *create*, *update* and *delete*. A line counts as create if it was added to the existing code. The defined subsections of effort are saved in the effort variables with an index at the tool or engineer specifier, e.g. it applies the equation $e_M^{4^1} = e_{M_C}^{4^1} + e_{M_U}^{4^1} + e_{M_D}^{4^1}$. If a line was deleted, the line counted as *delete*. If a line was created, the line counted as *create*. The updated lines are either edited lines, which existed also before, or a summary of one deleted and one created line. So if a line was created, one was deleted, and these two lines are in the same context space, the overall process counted as one *update*. Two lines are in one context, if they for example are in the same method, and semantically closely coupled. It could be the case that two lines with MFC semantic are exchanged with one line of calling a function in *rwmp.h*. The line with the function will be created and the other two lines will be deleted, but the effort measurement is counting one *update* and one *delete*.

In summary, the following equations are defined for describing the effort and duration measured during the evaluation:

$$t = t_M + t_T \tag{5.1}$$

$$t_X = \sum_{n \in Y(X)} t_X^n, X \in \{`M`, `T`\} \wedge Y(X) = \begin{cases} \{`1`, `2`, `4^1`, `5^1`, `4^2`, `5^2`\} \\ \{`3^1`, `6^1`, `7^1`, `3^2`, `6^2`, `7^2`\} \end{cases} \tag{5.2}$$

$$e = e_M + e_T \tag{5.3}$$

$$e_X = \sum_{n \in Y} e_X^n \begin{cases} X = `M` & Y = \{`4^1`, `5^1`, `4^2`, `5^2`\} \\ X = `T` & Y = \{`3^1`, `6^1`, `3^2`, `6^2`\} \end{cases} \tag{5.4}$$

$$e_X^Y = e_{X_C}^Y + e_{X_U}^Y + e_{X_D}^Y, (X,Y) \in \left\{ \begin{array}{l} (`M`, `4^1`), (`M`, `5^1`), \\ (`M`, `4^2`), (`M`, `5^2`), \\ (`T`, `3^1`), (`T`, `6^1`), \\ (`T`, `3^2`), (`T`, `6^2`) \end{array} \right\} \tag{5.5}$$

As a last measurement during the ReWaMP process WASM-T's performance was monitored. Therefore, the performance monitor of windows ran during the calculations of WASM-T and the processor time was logged. To classify the performance monitoring, in front of the evaluation round the executing researcher minuted the number of processes running before starting the evaluation environment. Before WASM-T was started or continued, the logging was activated, and when it paused or finished the logging was paused. After every pausing of the performance monitoring, the current average of the processor time was read and written down by the executing researcher. To ensure the correct monitoring the researcher was handling WASM-T. The test subject could still imagine how easy the usage of WASM-T was, since it was always able to watch.

The intervention in the ReWaMP process after the compiling to WASM, was included in the actions 4 and 5. So the needed corrections were measured in the way that they were able to add them to the time and effort measurements of the related actions.

5.1.3. Design of the Questionnaire

As already mentioned, the questionnaire was designed to get further information about the test subject, the functional scope and the usability of the resulting prototype, and about the ReWaMP process and WASM-T. To achieve appropriate results, the questionnaire was divided in four different parts. In addition, the complete questionnaire can be found in appendix A.1.

At first the test subject was asked to fill in information about itself. Hereby the years of programming and the experience in web development, programming in C++ and using MFC were covered.

Next, the functional scope of the resulting prototype was compared with the legacy's by two multiple choice grids about different features, one grid for the legacy and one for the resulting prototype. Some features are existing in the legacy and the prototype, and some of them not. As the legacy is not that big, the feature list can be fully covered with three points. Two others were not implemented, but included to the grid. They were just there to confuse the subject a bit and demanding its focus.

In the third part, the usability of the legacy was compared with the resulting prototypes' one. Therefore, three statements about parts of usability were made about the legacy and the prototype. So at large six statements were made in this part. The comparison was made via statements about the usability parts: compatibility, minimal action and learnability. [28]

The last part was about the ReWaMP process and WASM-T. Thus, the purpose there was to get feedback about the ReWaMP process and the current version of WASM-T. The first two statements in this part were about how easy it was to understand, what the test subject has to do in the ReWaMP process, and how easy the handling of WASM-T was. The next five statements were about the time and effort in the ReWaMP process and how well WASM-T supported the test subject.

Further on, the three statements were made about the WASM-T support via the hints after action 3, the ReWaMP flags and the expert file supporting by writing all function bodies, which should be executed at the server. The last statement concluded the questionnaire with the question about the most difficult task.

5.2. Results

The procedure was conducted with 6 different test subjects. They were all between 24 and 31, and 5 of them were male, thus 1 female. I was in all 6 rounds acting as the executing researcher.

As the design of the procedure already differentiates between two kinds of data, this section will accordingly have two subsections. First subsection 5.2.1 will present the results from the measurements during the ReWaMP process. Subsequently, subsection 5.2.2 will outline the results of the questionnaire.

5.2.1. ReWaMP Measurements Results

During the ReWaMP process, WASM-T was monitored according its performance. The results of this monitoring are depicted in figure 5.1. A 2-dimensional diagram shows there within a scatter chart the average processor times of all evaluation rounds. Additionally, a linear trendline of the average processor time overall regarding the other prototypes is drawn, which was calculated by minimizing the sum of squared distances from every point. The exact calculated equation for this trendline is:

$$y = 0.01 \times x - 0.729 \tag{5.6}$$

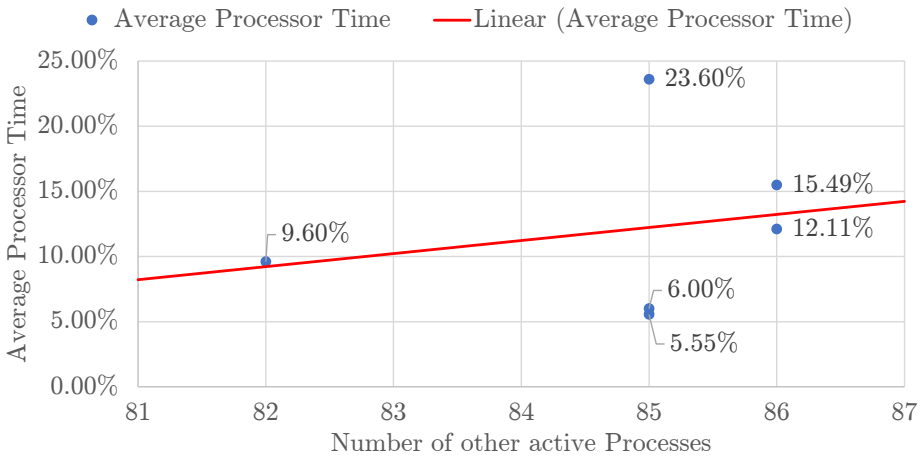


Figure 5.1.: Average Processor Times

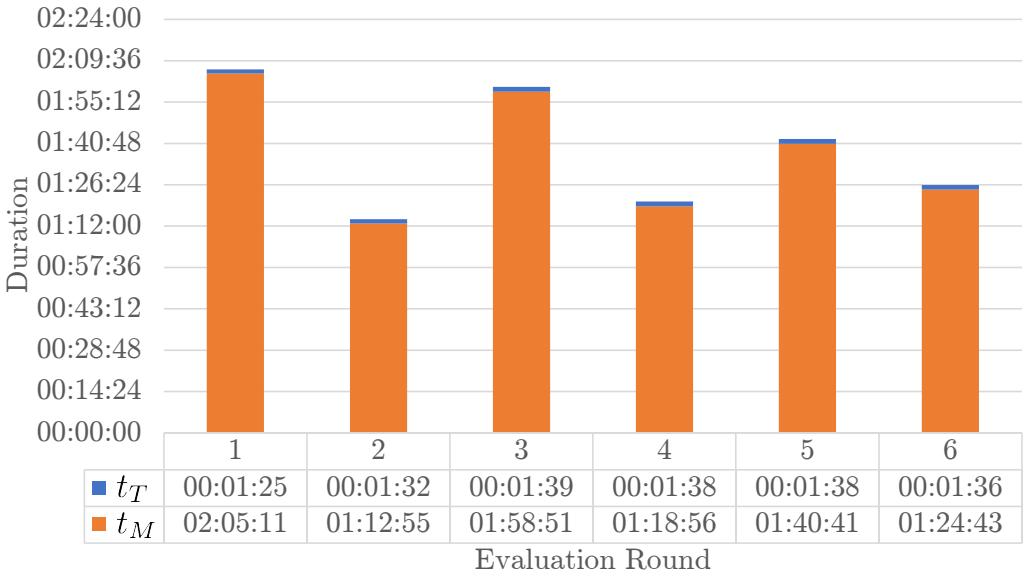
Every ReWaMP process needed in the evaluations at least 1 hour and 14 minutes. The longest duration in turn was about 2 hours and 5 minutes. All durations are figured in figure 5.2, thus t_M and t_T of all evaluation rounds. It also shows, that the needed time is as conceivable depending on the test subject, regarding the comparison of t_M and t_T, and regarding the comparison of all test subjects.

Figure 5.3 shows the detailed durations of t_M for every evaluation round. It divides the durations further in the subdurations of every task the test subject had to perform during the ReWaMP process. Moreover, it is noticeable that the actions 1 and 4 are consuming the most time. In average t_M^4 is a bit longer than t_M^1. At the most subjects $t_M^{4^2}$ is faster than $t_M^{4^1}$, while $t_M^{5^1}$ is faster than $t_M^{5^2}$.

Last, the effort distributions are shown in figure 5.4. The average distribution of e shows in subfigure 5.4a that WASM-T is already changing more SLOC than the migration engineer. In addition, subfigure 5.4b sketches the average e_M distribution, and shows that action 4 is nearby $\frac{3}{4}$ of e_M, as $e_M^{4^1} + e_M^{4^2} = 73\%$ of e_M. Further the subfigures 5.4c and 5.4d show the distribution of e_M and e_T according to their *create*, *update* and *delete* of SLOC. While the main action of the test subjects was *delete*, WASM-T mainly created new SLOC.

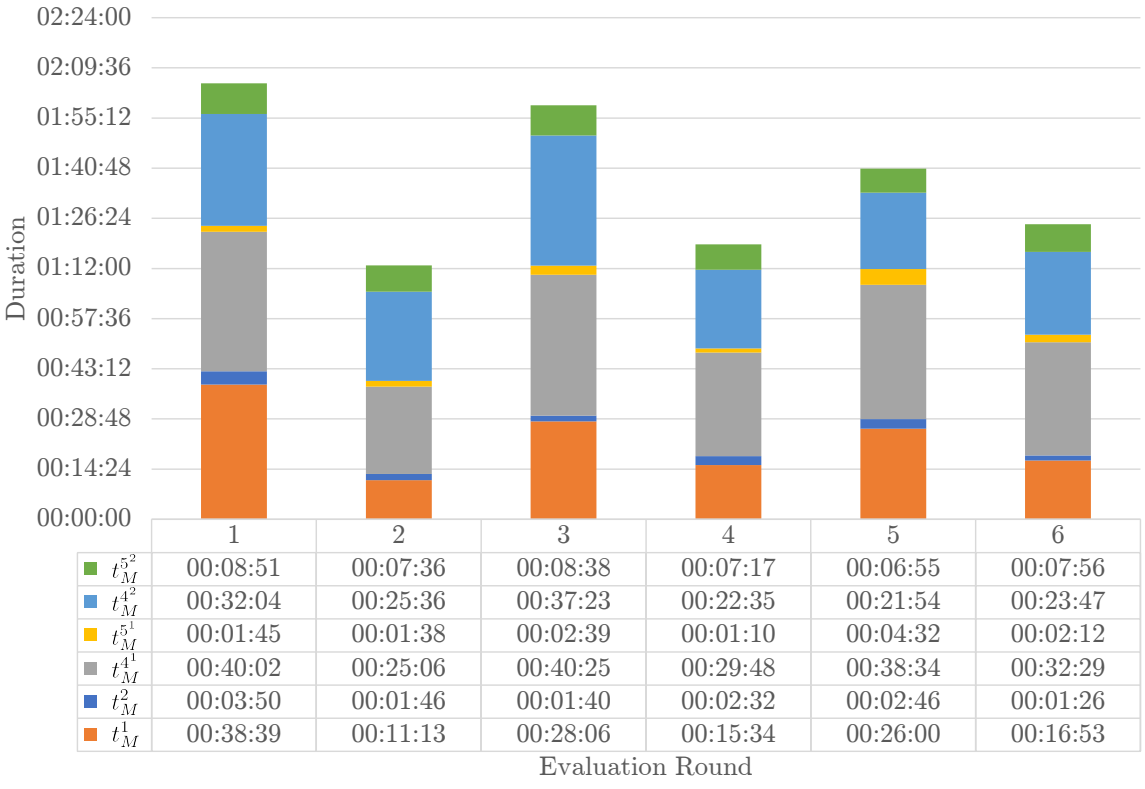
5.2.2. Questionnaire Results

The questionnaire started with the part about general information regarding the programming skills of the test subject. The results of this part are listed in table 5.1. It is noticeable, that either the subjects had no full year of experience in programming or more than 5 if not 10. Furthermore, most of the test subjects had moderate



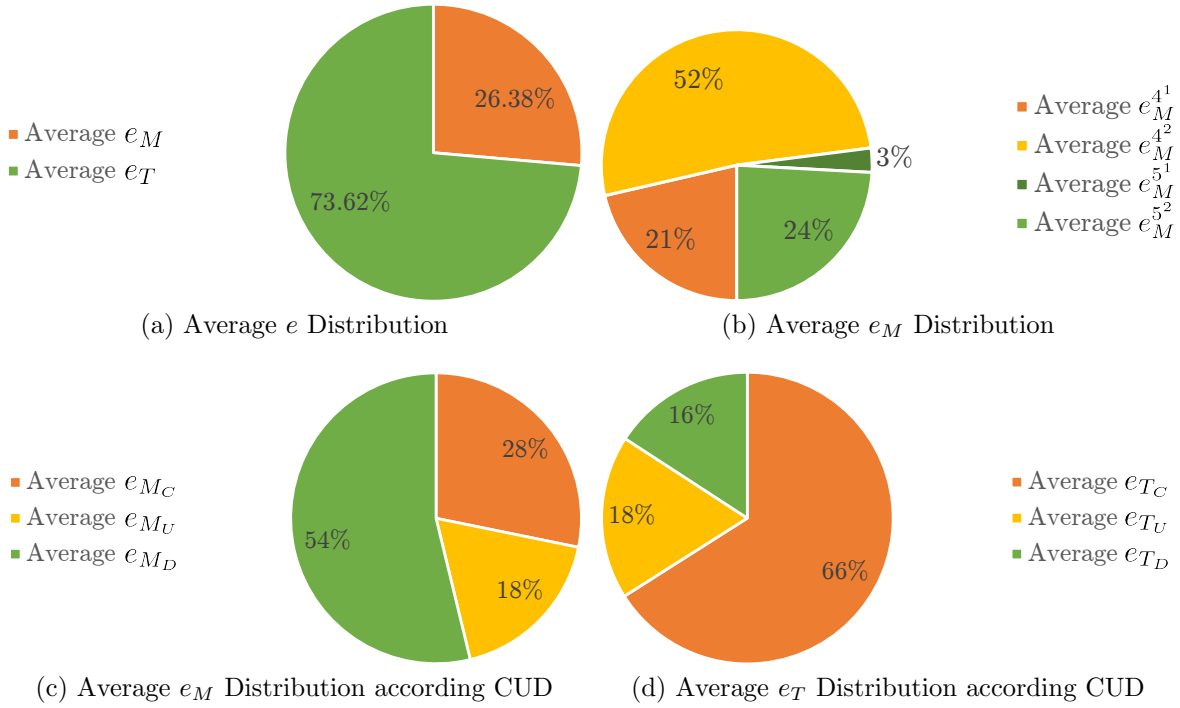
	1	2	3	4	5	6
t_T	00:01:25	00:01:32	00:01:39	00:01:38	00:01:38	00:01:36
t_M	02:05:11	01:12:55	01:58:51	01:18:56	01:40:41	01:24:43

Figure 5.2.: t Measurements



	1	2	3	4	5	6
$t_M^{5^2}$	00:08:51	00:07:36	00:08:38	00:07:17	00:06:55	00:07:56
$t_M^{4^2}$	00:32:04	00:25:36	00:37:23	00:22:35	00:21:54	00:23:47
$t_M^{5^1}$	00:01:45	00:01:38	00:02:39	00:01:10	00:04:32	00:02:12
$t_M^{4^1}$	00:40:02	00:25:06	00:40:25	00:29:48	00:38:34	00:32:29
t_M^2	00:03:50	00:01:46	00:01:40	00:02:32	00:02:46	00:01:26
t_M^1	00:38:39	00:11:13	00:28:06	00:15:34	00:26:00	00:16:53

Figure 5.3.: t_M Measurements



(a) Average e Distribution

(b) Average e_M Distribution

(c) Average e_M Distribution according CUD

(d) Average e_T Distribution according CUD

Figure 5.4.: Average Effort Distribution

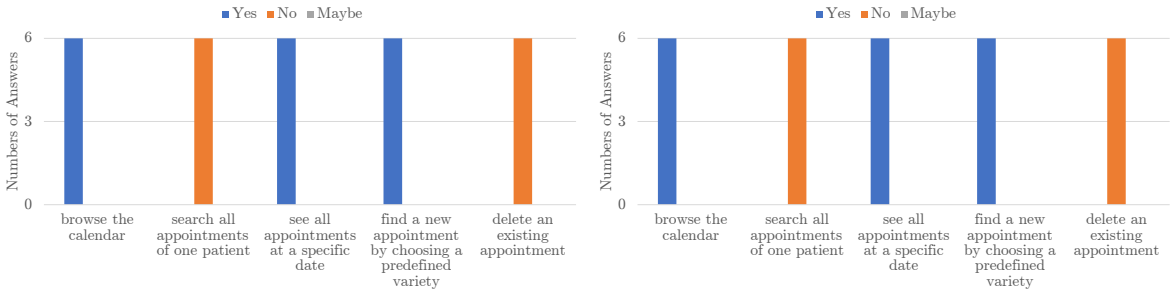
experience with C++, some had much experience in web development, and most nearby no experience using MFC.

The second part about the functionality, has unequivocal one statement. The listed features in the multiple choice grids are implemented in the resulting prototype just like in the legacy. The results can also be seen in figure 5.5.

In the questionnaire's third part, the usability comparison between the legacy and the resulting prototype was in the focus. The results are listed in table 5.2, and are quite different regarding the test subjects. On the contrary, the comparison of

	Results					
How many years of experience in programming do you have?	0	10	0	6	9	9
I have very much experience in web development.	1	5	2	4	1	4
I have very much experience in programming C++.	2	3	2	3	3	2
I have very much experience in using MFC for creating a GUI.	1	3	1	1	1	1

Table 5.1.: Statement Results regarding general Information



(a) In the Legacy I am able to...

(b) In the resulting Prototype I am able to...

Figure 5.5.: Functional Scope, Legacy and resulting Prototype

	Results					
In my opinion the legacy is performing just like I expected initially.	4	2	3	5	3	2
In my opinion the resulting prototype is performing just like I expected initially.	4	3	4	5	3	2
In my opinion the legacy is easy to use.	4	2	2	4	4	2
In my opinion the resulting prototype is easy to use.	5	2	4	4	4	3
In my opinion it is easy to learn how to use the legacy.	5	3	2	5	5	4
In my opinion it is easy to learn how to use the resulting prototype.	5	4	4	5	5	4

Table 5.2.: Statement Results regarding Usability

the legacy with the resulting prototype is in turn closely positive for the resulting prototype, since all line pairs are equal or higher at the prototype. In other words, every tested usability option is divided in two statements, one regards the legacy and the other regards the resulting prototype. The results of these pairs show all that the rating about the prototype is equal or even higher than the rating about the legacy.

The last part of the questionnaire was about the ReWaMP process and WASM-T. All statements' results except the last one were averaged by sum up all results and divided the sum by the amount of test subjects. The average results can be seen as ratings of the statements, and are shown in figure 5.6 as a bar diagram. The last question in the questionnaire asked what the most difficult task was. Its result is unequivocal action 4 making the expert changes, as to see in the pie chart at figure 5.7.

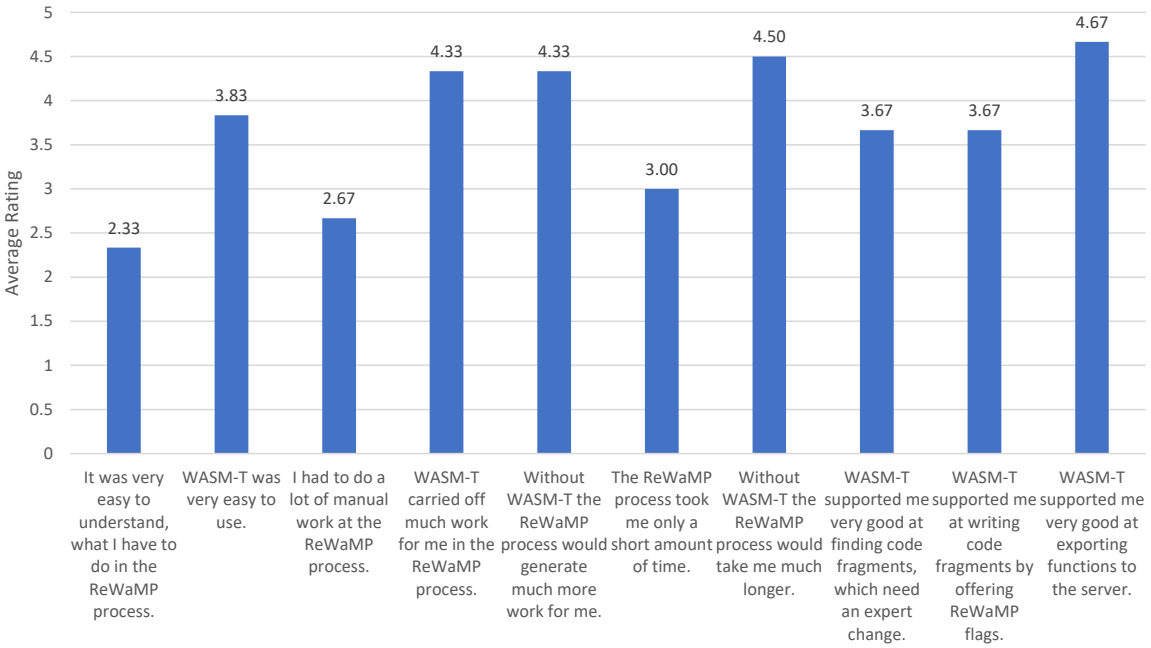


Figure 5.6.: ReWaMP Process and WASM-T Ratings

5.3. Discussion

After the evaluation experiment and the presentation of its results, the core of this chapter will be addressed in this section. Several requirements were defined in section 2.1. They were classified as result- and process-related requirements, and the result-related ones were even further divided in user, manager or developer requirements. All defined requirements will be discussed and analyzed regarding their satis-

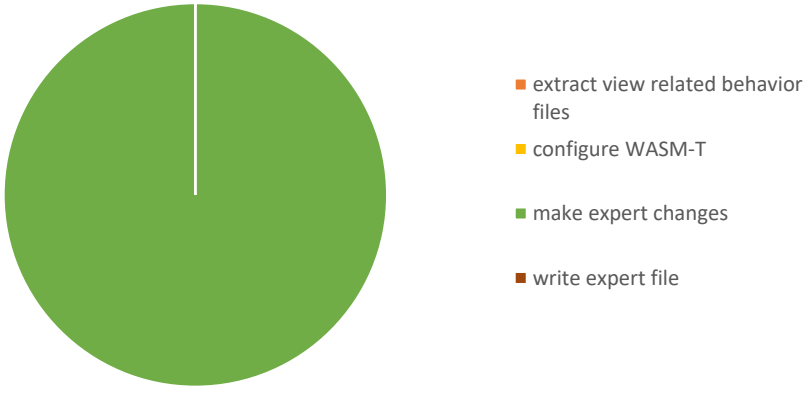


Figure 5.7.: Most difficult Task in the ReWaMP Process

faction in this section. The evaluation results as well as the conceptual design itself, will be used as root for deciding how good the results are satisfied.

Functional Scope

The functional scope was a user requirement, which demands the same functionality in the resulting rapid web migration prototype as the legacy had. As it is a first prototype, the functionality is may not be given in its full spectrum, but the user should at least be able to do the same as before. Thus, the front functionality must be provided, but not the complete business logic in the back-end.

The functional scope is completely provided in the given scenario. Initially the purpose of enabling a user-orientated prototype is given by the design of ReWaMP, and further the questionnaire's results show with figure 5.5 that also the test subjects were satisfied. The functional scope is therewith unequivocally given in the scenario.

As the scenario's prototype was thus comply the functional scope, it is assumable that this requirement will also be fulfilled in other scenarios.

Usability

The usability of UIs can be divided in many parts. In the third part of the questionnaire, the usability was investigated regarding three parts of usability: compatibility, minimal action and learnability. [28] These are not all, but they are good indicators for the usability.

All procedure results are positive. Because the comparison between the legacy and the resulting prototype were at all test subjects positive, since the prototype statements of each pair was equal or higher than its legacy partner. The concrete level of the three parts are not relevant, as the requirement is defined to fulfill the same level of usability with the prototype as the legacy did. This is unexceptional the case for all test subjects.

The usability can thus be seen as fulfilled in the manner of the defined requirement, but not completely. As the three considered parts are only some, they only give an indication, but not a clear evidence.

Decision Support

The decision support requirement is the first manager requirement. It demands that the rapid web migration prototype should be a decision support to overcome the initial hurdle of web migration.

To be such a prototype, the users must be convinced or rather happy about the new type of application. Further the prototype must show a web version of the legacy, which has common web application attributes. Such web application attributes are a client/server architecture, the related communication between the client and the server, a data serialization for the communication, and a working HTML-based GUI.

With the satisfaction of functional scope and usability, the user is at least convinced. As migration is change, the user is maybe not happy by design, but this is unprovable with the given scenario. Furthermore, the named challenges of section 3.4 cover the named common web application attributes. As all of these challenges are solved with ReWaMP, they further indicate that ReWaMP's result is a decision support.

In summary, a rapid web migration prototype of ReWaMP is also a decision support. It is possibly, not covering all doubts with a clear evidence, but it is a decision support that can give a first version of the legacy in the web.

Rapid Creation

Another manager requirement is the rapid creation, as the legacy is often of business critical value, and thus future decisions about it are important for the whole company. So the management will often want a fast decision to yield money. This applies for every resulting decision after the creation of the prototype, no matter whether the migration will take place or the company focuses on the desktop version.

To create a complete prototype, also the other both steps of the ReWaMP overview are needed. The evaluation procedure still focused as the thesis on the ReWaMP process. Checking its results, the measured durations are completely dependent on the migration engineer and his programming skills. If he is experienced, the ReWaMP process will take not that much time. But if he is quite inexperienced, the measured durations show, that it will take almost twice as much. This difference could also get higher, as the participated test subjects have still some weaknesses and maximal 10 years of coding experience in total.

The measured durations are still quite good. WASM-T is performing under 2 minutes and the test subjects finished one view in less than 45 minutes, some even in less than 30 minutes. Furthermore, the test subjects rated that the ReWaMP process

took them some time. So it is not just a finger snapping and it is done. At least they are pretty common about the fact that WASM-T shortened their time already with the current version, as the related statement got an average rating of 4.5.

The rapid creation prototype can be seen as partially complied, because within the scenario the rapid creation is supported by the ReWaMP process and WASM-T. It yet needs some more improvement for perfect rapidness.

Cheap Production

The cheap production, infers from the main motivation of the managers to yield money and rather earn more. As the first prototype is not a final product, the production should not cost much, because the prototype cannot be taken for making business as normal.

The cheap production is ensured by reducing the personnel costs with needing less people at creating the prototype, and by low expenses for tools and materials. As the company worked also before with the legacy, it can be assumed that they have a development environment for their developers. Hence, there will be no costs regarding a computer or work station and an installed integrated development environment to read the legacy code in action 1 and perform changes at actions 4 and 5.

Further, the ReWaMP process is assuming to need only one migration engineer. More personnel would cost more, but the creation could also be even faster, as yet there are still some manual tasks to perform. This is not only the case for the ReWaMP process, but also for the actions 2 and 3 of the ReWaMP overview.

As the used tools of WASM-T are available for free, and WASM-T is just a python script, the whole production of the first rapid web migration prototype can be seen as cheap process.

Low Effort

Every developer is interested in an approach with effort as low as possible. This is meant by doing as less as possible, but also not need to learn many new skills.

The rating of the ReWaMP process or rather how easy it was to understand the task, the test subjects had to do, was rather bad with an average of 2.33. The quite high effort regarding the 2.67 average rating of the to complete manual work is also bad, as it is indicating that the effort was quite high in the opinion of the test subjects.

On the other side, the measurements and questionnaire also show, that WASM-T is in its current state already supporting the migration engineer quite much, and carrying a lot of work for the migration engineer. As figure 5.4a shows WASM-T is already taking off nearby $\frac{3}{4}$ of the incidental work. Further, the test subjects rated with an average of 4.33 that WASM-T carried off much work and that it would be much more work for them if WASM-T would not be there to support.

In summary, the low effort can be seen as nearby completely complied. It is already very low as the measurements show, and the only other possibility to compare it with is performing the ReWaMP process by hand without WASM-T. This will then in turn take even more effort. To reduce the effort of the migration engineer significantly more, WASM-T must be improved to understand code semantically with all aspects. This is particularly the case since the most difficult task was unequivocal action 4 with making the expert changes.

Rapid Prototyping Methodology

Sticking to the rapid prototyping methodology is important for the developer to extract knowledge, experience or even advantages out of the first rapid prototype. Thereby three aspects were defined within the requirement: rapid creation, simple expandability, and containing of all necessary modules. The rapid creation was hereby discussed already in another section, thus it is fulfilled as analyzed there.

The simple expandability is granted. Subsections 3.5.2 and 3.5.3 already pointed out how to expand the first rapid web migration prototype of ReWaMP at two points, server-side and GUI. Furthermore, the WASM file can be reduced in size and functionality by each iteration, as more functions can be directly edited in JS and only called by WASM.

The containing of all necessary modules is also given, because all necessary modules are included. So all typical web application modules are in the design of ReWaMP, and it also contains all necessary modules of the legacy, since it reflects the same functional scope.

As all three aspects are complied, the requirement is also fulfilled.

Process Automation

The process automation requirement is defined to take as much as possible effort and knowledge of the migration engineer into a tool.

By using the ReWaMP process and WASM-T several challenges are solved without effort of the migration engineer. All of those named in section 3.4 are solved by WASM-T without any manual influence.

Further the measurements show, that already $\frac{3}{4}$ of the effort are carried of by WASM-T, cf. figure 5.4a. WASM-T even guides the migration engineer from the first action 3 to the desired results.

At last the process automation requirement is satisfied on a very good level. There are still possibilities for improving the automation, but the current level is automating many aspects.

Legacy Code as a Source

The legacy should be used as a source for obtaining the whole semantic implemented in the legacy. This is the case by the design of ReWaMP, as shown in figure 3.1.

Using WebAssembly

In the course of this thesis, WASM should be used, as it is becoming a new open web standard and enables C/C++ code to run in the browser with the same speed and semantics as before. Since the ReWaMP design and the ReWaMP process are using or rather generating WASM files, this requirement is complied.

Open Web Standards

For future-proofed usage and feasibility of all SMEs the open web standards requirements defined that the rapid web migration prototype should only use open standards. The ReWaMP architecture shows in figure 3.2 that this is the case. It uses only HTML, CSS, JS and WASM within the resulting prototype and all of them are open web standards or will be soon one, as WASM will.

Client/Server Architecture

A client/server architecture is one of the main artifacts a common web application has, and thus the resulting prototype should have. The ReWaMP architecture solved this as presented in section 3.2 and section 3.4 even the serialization for exchanging data between all architecture components was solved.

Communication with HTML UI

Since the new web environment infers a new HTML UI the legacy code must also be able to communicate with this new UI to ensure the complete functionality. This requirement is also complied as the requirement of the client/server architecture within the design of the ReWaMP architecture and the solved challenges of the ReWaMP process.

5.4. Summary

The last section discussed if the defined requirements are defined. Further it pointed out also the aspects, which are not perfect and how they influence the complying result of the requirement. Since the analysis section was quite long, this section will provide a short summary of the requirement analysis.

To bring all requirements in one overview table 5.3 is summarizing all and how good they are fulfilled. Since every requirement is complied according to the scenario, evaluation procedure and the ReWaMP concept and process, the assessment scale in the table is defined as follows:

- ++ the requirement is fulfilled unequivocally
- + the requirement is complied, but a few aspects are limiting it to be fulfilled completely

	Assessment
Functional Scope	++
Usability	+
Decision Support	+
Rapid Creation	+
Cheap Production	++
Low Effort	+
Rapid Prototyping Methodology	++
Process Automation	+
Legacy Code as a Source	++
Open Web Standards	++
Client/Server Architecture	++
Communication with HTML UI	++

Table 5.3.: Requirements Satisfaction Summary

6. Conclusion

This thesis was motivated by overcoming the initial hurdle of web migration. Several aspects of the legacy and the SME were discussed, and as solution a rapid web migration prototype was proposed. This prototype could support the decision as rapid prototyping does in forward agile development. It was further defined that this thesis will conduct how to use WASM in a rapid web migration prototype. According to this technology usage, the thesis focus was set, and components in the prototype besides WASM were rather a black-box in the thesis' context, thus should only be considered as much as needed for an executable prototype.

In the second chapter the thesis was continued with a requirement specification or rather definition. Those were divided in result- and process-related requirements, and the result-related ones even further by the three roles user, manager and developer. The rest of the chapter was then presenting state-of-the-art approaches according web migration and rapid prototyping. The synergy of these two topics is rather new, thus no state-of-the-art approaches exist. To combine them still on a abstract level, the chapter closed with emphasizing aspects of some approaches or references according web migration or rapid prototyping, and what that mean for rapid web migration prototyping.

Next ReWaMP was presented as the solving concept to create a first rapid web migration prototype. Besides the new web UI and a server-side, WASM is used in the ReWaMP runtime to create a web application fitting version of the presentation logic. ReWaMP was further defined in this thesis only with leveraging WASM, whereby the appended paper at Appendix B is also conducting other technologies within the ReWaMP runtime. As the thesis' focus is the WASM usage, action 1 of ReWaMP was explained with more details upon the architecture and the ReWaMP process. After describing also the solved challenges, additional notes were pointed out according to the conceptual decisions using WASM and the two black boxes of GUI and server-side.

After presenting the implementation of the supporting transformation tool in the ReWaMP process called WASM-T at chapter 4, the concept was evaluated. In the evaluation an experiment was conducted, which simulated a web migration undertaking of the scenario presented at section 1.4. According to this different measurements were made during the ReWaMP process of each evaluation round and in the end of

each a questionnaire was filled by the test subject. With all these results the defined requirements were analyzed and discussed if they are complied. In summary all requirements were fulfilled by ReWaMP.

Further WASM-T was rated as a very good supporter in its current state, but still some aspects can be improved. Referring to the bar chart of average ratings in figure 5.6, WASM-T holds a good position in the middle ranks at the supporting of finding obsolete code fragments, offering ReWaMP flags and how easy it was to use in general. It still does not want to be missed since it carried off much work and saved a lot of time. The expert functions are filled automatically, and the test subjects also honored that with a good rating. So this feature is already matured.

The ReWaMP process was in turn not rated that good, since the average rating how easy it was to understand what to do when, was only at 2.33. It further was rated that it took rather a longer time than a short, but on the other side the manual effort was rather low than high.

For future work, the ReWaMP process should as a main goal be improved by eliminating the expert changes in action 4, since this was the most difficult task with an agreement of 100%. If this would be the case, also action 5 of writing the expert file could easily be included in WASM-T. To eliminate this action there is still research required, since a semantically understanding of C++ code and the comparison of MFC constructs to the typical browser environment with WASM is needed.

A first approach reducing the difficulty of the expert changes can be the improvement of the hints given by WASM-T after the preprocessing of each view. As a next step it could also be a good idea to create an addon for an integrated development environment to support the expert changes with effective understanding of the code and marking of possible compile errors. This could also be a first step in the direction of completely eliminating action 4 and 5 for the migration engineer.

The ReWaMP process could on the other side also be improved at action 1 of extracting the related view behavior files. An automatism of this step could further rapid the complete process and also discharge the migration engineer.

As WASM-T is in its current state a prototypical version, this could also be improved to a stable one with supporting more MFC fragments. The assumptions on the legacy code should also be dismantled to improve the overall support. In the distant future it should then also support more desktop libraries than just MFC.

Bibliography

[1] Alavi, Maryam: *An assessment of the prototyping approach to information systems development.* Communications of the ACM, 27(6):556–563, 1984.

[2] Aversano, Lerina, Gerardo Canfora, Aniello Cimitile and Andrea De Lucia: *Migrating legacy systems to the web: an experience report.* In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 148–157. IEEE, 2001.

[3] Barnum, Carol M.: *Usability Testing and Research.* Allyn & Bacon, Inc., Needham Heights, MA, USA, 1st edition, 2001.

[4] Bodhuin, Thierry, Enrico Guardabascio and Maria Tortorella: *Migrating COBOL systems to the Web by using the MVC design pattern.* In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 329–338. IEEE, 2002.

[5] Bodhuin, Thierry, Enrico Guardabascio and Maria Tortorella: *Migration of non-decomposable software systems to the web using screen proxies.* In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 165–174. IEEE, 2003.

[6] Bovenzi, Diego, Gerardo Canfora and Anna Rita Fasolino: *Enabling legacy system accessibility by web heterogeneous clients.* In *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, pages 73–81. IEEE, 2003.

[7] Brodie, Michael L. and Michael Stonebraker: *Legacy Information Systems Migration: Gateways, Interfaces, and the Incremental Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[8] Canfora, Gerardo, Anna Rita Fasolino, Gianni Frattolillo and Porfirio Tramontana: *Migrating interactive legacy systems to web services.* In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10–pp. IEEE, 2006.

[9] Comella-Dorda, Santiago, Kurt C Wallnau, Robert C Seacord and John E Robert: *A Survey of Black-Box Modernization Approaches for Infor-*

mation Systems. In *icsm*, pages 173–183, 2000.

[10] De Lucia, Andrea, Rita Francese, Giuseppe Scanniello, Genoveffa Tortora and Nicola Vitiello: *A strategy and an Eclipse based environment for the migration of legacy systems to multi-tier web-based architectures.* In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 438–447. IEEE, 2006.

[11] Dineen, Terence H, Paul J Leach, Nathaniel Mishkin, Joseph N Pato and Geoffrey L Wyant: *The Network Computing Architecture and System: An Environment for Developing Distributed Applications.* In *COMPCON*, pages 296–299, 1988.

[12] Ferreira, Jennifer, James Noble and Robert Biddle: *Agile development iterations and UI design.* In *Agile Conference (AGILE), 2007*, pages 50–58. IEEE, 2007.

[13] Gears, Coder: *CppDepend*, 2018. `https://www.cppdepend.com/`, Last checked: 16.01.2018.

[14] Ghiani, Giuseppe, Fabio Paternò and Carmen Santoro: *On-demand cross-device interface components migration.* In *Proceedings of the 12th international conference on Human computer interaction with mobile devices and services*, pages 299–308. ACM, 2010.

[15] Ghiani, Giuseppe, Fabio Paternò and Carmen Santoro: *On-demand cross-device interface components migration.* In *Proceedings of the 12th international conference on Human computer interaction with mobile devices and services*, pages 299–308. ACM, 2010.

[16] Ghiani, Giuseppe, Fabio Paternò and Carmen Santoro: *Push and pull of web user interfaces in multi-device environments.* In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 10–17. ACM, 2012.

[17] Group, Object Management et al.: *Business Process Model and Notation (BPMN) Version 2.0.* Object ManagementGroup, 2011.

[18] Heil, Sebastian, Maxim Bakaev and Martin Gaedke: *Measuring and Ensuring Similarity of User Interfaces: The Impact of Web Layout.* In *International Conference on Web Information Systems Engineering*, pages 252–260. Springer, 2016.

[19] Highsmith, Jim and Alistair Cockburn: *Agile software development: The*

business of innovation. Computer, 34(9):120–127, 2001.

[20] Holzinger, Andreas: *Usability Engineering Methods for Software Developers*. Commun. ACM, 48(1):71–74, January 2005.

[21] Horowitz, Ellis: *Migrating software to the world wide web*. IEEE software, 15(3):18, 1998.

[22] Jakl, Michael: *Rest representational state transfer*. 2008.

[23] JetBrains: *CLion*, 2018. `https://www.jetbrains.com/clion/`, Last checked: 16.01.2018.

[24] JetBrains: *PyCharm*, 2018. `https://www.jetbrains.com/pycharm/`, Last checked: 16.01.2018.

[25] Law, Ken CK, Horace Ho-Shing Ip and Fang Wei: *Web-enabling legacy applications*. In *Parallel and Distributed Systems, 1998. Proceedings. 1998 International Conference on*, pages 218–225. IEEE, 1998.

[26] Lennox, Jonathan: *Common gateway interface for SIP*. 2001.

[27] Lientz, Bennet P, E. Burton Swanson and Gail E Tompkins: *Characteristics of application software maintenance*. Communications of the ACM, 21(6):466–471, 1978.

[28] Lin, Han X, Yee-Yin Choong and Gavriel Salvendy: *A proposed index of usability: a method for comparing the relative usability of different software systems*. Behaviour & information technology, 16(4-5):267–277, 1997.

[29] Littlewood, Bev and Lorenzo Strigini: *The risks of software*. Scientific American, 267(5):62–75, 1992.

[30] Lohmann, Niels: *JSON for modern C++*, 2018. `https://github.com/nlohmann/json`, Last checked: 16.01.2018.

[31] Moore, Melody M and Lilia Moshkina: *Migrating legacy user interfaces to the internet: Shifting dialogue initiative*. In *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, pages 52–58. IEEE, 2000.

[32] Nelson, Bruce Jay: *Remote procedure call*. 1981.

[33] Parnas, David Lorge: *Software aging*. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[34] Paschall, Alexander: *Unreal Engine 4.16 Released!*, 2017. https://www.unrealengine.com/en-US/blog/unreal-engine-4-16-released, Last checked: 29.10.2017.

[35] Perrey, Randall and Mark Lycett: *Service-oriented architecture*. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[36] Ping, Yu, Kostas Kontogiannis and Terence C Lau: *Transforming legacy Web applications to the MVC architecture*. In *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on*, pages 133–142. IEEE, 2003.

[37] Project, LLVM: *clang: a C language family frontend for LLVM*, 2018. https://clang.llvm.org/, Last checked: 16.01.2018.

[38] Project, LLVM: *Clang Python Bindings*, 2018. https://github.com/llvm-mirror/clang/tree/master/bindings/python, Last checked: 16.01.2018.

[39] Project, LLVM: *libclang: C Interface to Clang*, 2018. https://clang.llvm.org/doxygen/group__CINDEX.html, Last checked: 16.01.2018.

[40] Project emscripten: *Calling JavaScript from C/C++*, 2018. https://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#calling-javascript-from-c-c, Last checked: 16.01.2018.

[41] Project emscripten: *Embind*, 2018. https://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/embind.html, Last checked: 16.01.2018.

[42] Project emscripten: *WebIDL Binder*, 2018. https://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/WebIDL-Binder.html, Last checked: 16.01.2018.

[43] Project emscripten: *WebIDL Binder and Embind*, 2018. https://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-binding-cpp, Last checked: 16.01.2018.

[44] Puder, Arno: *Extending desktop applications to the web*. In *Proceedings of the 2004 international symposium on Information and communication technologies*, pages 8–13. Trinity College Dublin, 2004.

[45] Rivero, José Matías, Julián Grigera, Gustavo Rossi, Esteban Robles Luna and Nora Koch: *Towards agile model-driven web engineering.* In *Forum at the Conference on Advanced Information Systems Engineering (CAiSE)*, pages 142–155. Springer, 2011.

[46] Rivero, José Matías, Julián Grigera, Gustavo Rossi, Esteban Robles Luna, Francisco Montero and Martin Gaedke: *Mockup-driven development: providing agile support for model-driven web engineering.* Information and Software Technology, 56(6):670–687, 2014.

[47] Rivero, José Matías, Sebastian Heil, Julián Grigera, Martin Gaedke and Gustavo Rossi: *MockAPI: an agile approach supporting API-first web application development.* In *International Conference on Web Engineering*, pages 7–21. Springer, 2013.

[48] Rivero, José Matías, Gustavo Rossi, Julián Grigera, Juan Burella, Esteban Robles Luna and Silvia Gordillo: *From mockups to user interface models: an extensible model driven approach.* In *International Conference on Web Engineering*, pages 13–24. Springer, 2010.

[49] Sefelin, Reinhard, Manfred Tscheligi and Verena Giller: *Paper prototyping-what is it good for?: a comparison of paper-and computer-based low-fidelity prototyping.* In *CHI'03 extended abstracts on Human factors in computing systems*, pages 778–779. ACM, 2003.

[50] Sneed, Harry M: *Encapsulating legacy software for use in client/server systems.* In *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pages 104–119. IEEE, 1996.

[51] Sneed, Harry M: *Wrapping legacy COBOL programs behind an XML-interface.* In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 189–197. IEEE, 2001.

[52] Sneed, Harry M: *Integrating legacy software into a service oriented architecture.* In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 11–pp. IEEE, 2006.

[53] Sprague Jr, Ralph H and Eric D Carlson: *Building effective decision support systems.* Prentice Hall Professional Technical Reference, 1982.

[54] Spurlock, Jake: *Bootstrap: Responsive Web development.* O'Reilly Media, Inc., 2013.

[55] Srinivasan, Raj: *RPC: Remote procedure call protocol specification version 2.*

1995.

[56] Tak, Byung Chul and Chunqiang Tang: *Appcloak: Rapid migration of legacy applications into cloud.* In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 810–817. IEEE, 2014.

[57] Thomke, Stefan and Donald Reinertsen: *Agile product development: Managing development flexibility in uncertain environments.* California management review, 41(1):8–30, 1998.

[58] Tripp, Steven D. and Barbara Bichelmeyer: *Rapid prototyping: An alternative instructional design strategy.* Educational Technology Research and Development, 38(1):31–44, 1990.

[59] Turner, Mark, David Budgen and Pearl Brereton: *Turning software into a service.* Computer, 36(10):38–44, 2003.

[60] Veling, Louise: *Human-Centred Design for Digital Transformation.* IVI White Paper Series, 2014.

[61] Wagner, Christian: *Model-driven software migration: A methodology: Reengineering, recovery and modernization of legacy systems*, volume 9783658052706. Springer Science & Business Media, 2014.

[62] Waldo, Jim: *Remote procedure calls and java remote method invocation.* IEEE concurrency, 6(3):5–7, 1998.

[63] Wasserman, Anthony I and David T Shewmake: *Rapid prototyping of interactive information systems.* In *ACM SIGSOFT Software Engineering Notes*, volume 7, pages 171–180. ACM, 1982.

[64] WebAssembly.org: *FAQ, Why create a new standard when there is already asm.js?*, 2017. `http://webassembly.org/docs/faq/#why-create-a-new-standard-when-there-is-already-asmjs`, Last checked: 29.10.2017.

[65] Weinberg, Gerald M: *Perfect software: And other Illusions about testing.* Dorset House Publishing Co., Inc., 2008.

[66] Yang, Jishuo and Daniel Wigdor: *Panelrama: enabling easy specification of cross-device web applications.* In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2783–2792. ACM, 2014.

[67] Ylonen, Tatu and Chris Lonvick: *The secure shell (SSH) protocol architecture.* 2006.

[68] Zakai, Alon: *Emscripten: an LLVM-to-JavaScript compiler*. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

[69] Zhang, Liang-Jie, Jia Zhang and Hong Cai: *Service-oriented architecture*. Services Computing, pages 89–113, 2007.

Appendix A.

Evaluation Appendices

In this chapter all appendices of the evaluation experiment are submitted. To give them a structure, they are divided into different sections.

A.1. Questionnaire

This section will show on the next four pages the questionnaire every test subject had to fill out at the end of one evaluation round.

ReWaMP Process Evaluation

Thank you for participating in the ReWaMP process evaluation.

Your feedback is needed to improve the experiment data.
Please fill this quick survey.
Your answers will be anonymous.

*Required

1. **How many years of experience in programming do you have?** *
   Please fill in the integer number of years.

2. **I have very much experience in web development.** *
   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

3. **I have very much experience in programming C++.** *
   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

4. **I have very much experience in using MFC for creating a GUI.** *
   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

Questions regarding the legacy and the resulting prototype - part 1

5. **In the legacy I am able to ...** *
   Mark only one oval per row.

	Yes	No	Maybe
browse the calendar	◯	◯	◯
search all appointments of one patient	◯	◯	◯
see all appointments at a specific date	◯	◯	◯
find a new appointment by choosing a predefined variety	◯	◯	◯
delete an existing appointment	◯	◯	◯

6. **In the resulting prototype I am able to ...** *
Mark only one oval per row.

	Yes	No	Maybe
browse the calendar			
search all appointments of one patient			
see all appointments at a specific date			
find a new appointment by choosing a predefined variety			
delete an existing appointment			

Questions regarding the legacy and the resulting prototype - part 2

7. **In my opinion the legacy is performing just like I expected initially.** *
Mark only one oval.

	1	2	3	4	5	
Not at all						Exactly

8. **In my opinion the resulting prototype is performing just like I expected initially.** *
Mark only one oval.

	1	2	3	4	5	
Not at all						Exactly

9. **In my opinion the legacy is easy to use.** *
Mark only one oval.

	1	2	3	4	5	
Not at all						Exactly

10. **In my opinion the resulting prototype is easy to use.** *
Mark only one oval.

	1	2	3	4	5	
Not at all						Exactly

11. **In my opinion it is easy to learn how to use the legacy.** *
Mark only one oval.

	1	2	3	4	5	
Not at all						Exactly

12. **In my opinion it is easy to learn how to use the resulting prototype.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

Questions regarding WASM-T

13. **It was very easy to understand, what I have to do in the ReWaMP process.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

14. **WASM-T was very easy to use.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

15. **I had to do a lot of manual work at the ReWaMP process.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

16. **WASM-T carried off much work for me in the ReWaMP process.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

17. **Without WASM-T the ReWaMP process would generate much more work for me.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

18. **The ReWaMP process took me only a short amount of time.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

19. **Without WASM-T the ReWaMP process would take me much longer.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

20. **WASM-T supported me very good at finding code fragments, which need an expert change.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

21. **WASM-T supported me at writing code fragments by offering ReWaMP flags.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

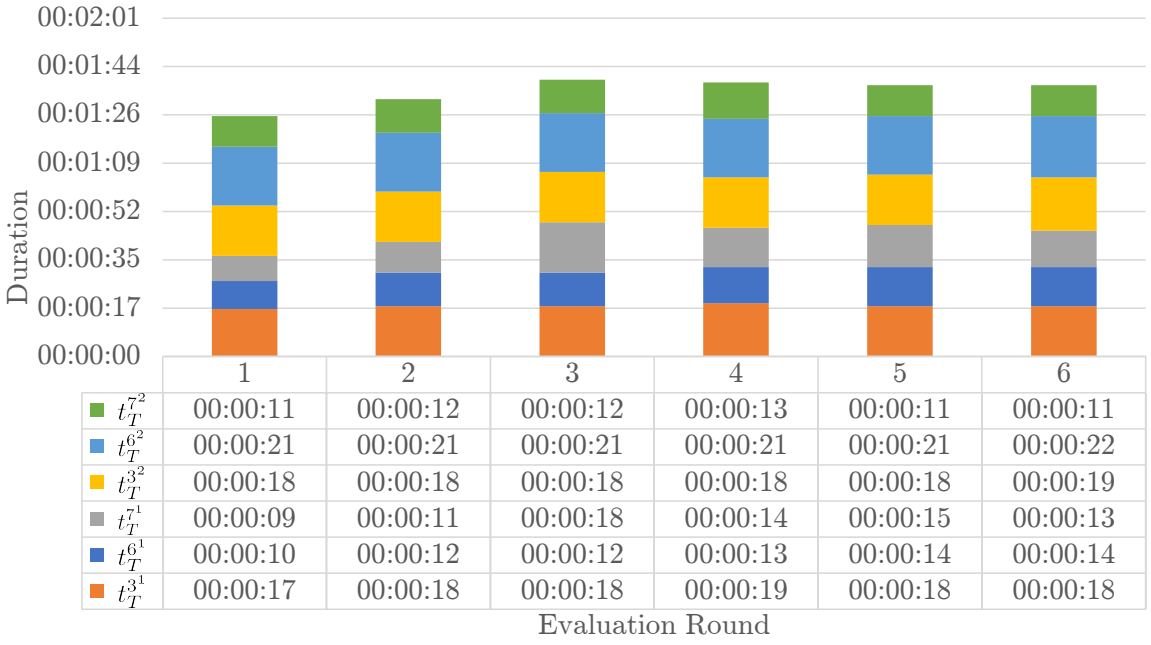
22. **WASM-T supported me very good at exporting functions to the server.** *
Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Exactly

23. **The most difficult task was for me...** *
Mark only one oval.

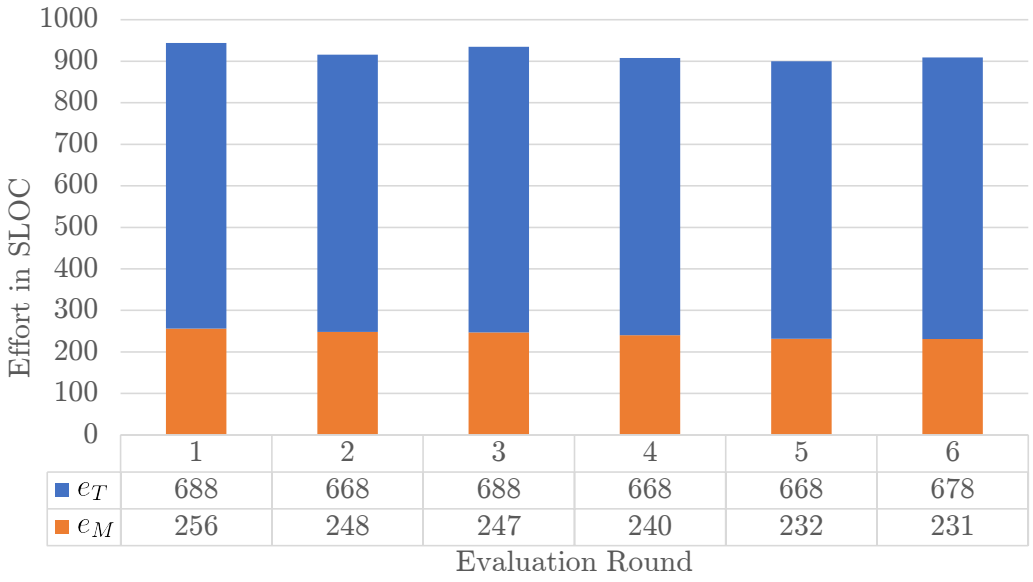
- ◯ Extract view related behavior files
- ◯ configure WASM-T
- ◯ make expert changes
- ◯ write expert file

A.2. Results



	1	2	3	4	5	6
$t_T^{7^2}$	00:00:11	00:00:12	00:00:12	00:00:13	00:00:11	00:00:11
$t_T^{6^2}$	00:00:21	00:00:21	00:00:21	00:00:21	00:00:21	00:00:22
$t_T^{3^2}$	00:00:18	00:00:18	00:00:18	00:00:18	00:00:18	00:00:19
$t_T^{7^1}$	00:00:09	00:00:11	00:00:18	00:00:14	00:00:15	00:00:13
$t_T^{6^1}$	00:00:10	00:00:12	00:00:12	00:00:13	00:00:14	00:00:14
$t_T^{3^1}$	00:00:17	00:00:18	00:00:18	00:00:19	00:00:18	00:00:18

Figure A.1.: t_T Measurements



	1	2	3	4	5	6
e_T	688	668	688	668	668	678
e_M	256	248	247	240	232	231

Figure A.2.: ReWaMP Process Total Effort

1	How many years of experience in programming do you have?	I have very much experience in web development.	I have very much experience in programming C++.	I have very much experience in using MFC for creating a GUI.
2	0	1	2	1
3	10	5	3	3
4	0	2	1	1
5	6	4	3	1
6	9	1	3	1
7	9	4	2	1

Figure A.3.: Questionnaire Results 1

1	In the legacy I am able to ...					In the resulting prototype I am able to ...				
2	browse the calendar	search all appointments of one patient	see all appointments at a specific date	find a new appointment by choosing a predefined variety	delete an existing appointment	browse the calendar	search all appointments of one patient	see all appointments at a specific date	find a new appointment by choosing a predefined variety	delete an existing appointment
3	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	No
4	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	No
5	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	No
6	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	No
7	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	No
8	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	No

Figure A.4.: Questionnaire Results 2

1	In my opinion the resulting prototype is performing just like I expected initially.	In my opinion the legacy is easy to use.	In my opinion the resulting prototype is easy to use.	In my opinion it is easy to learn how to use the legacy.	In my opinion it is easy to learn how to use the resulting prototype.
2	4	4	5	5	5
3	3	2	2	3	4
4	4	2	4	2	4
5	5	4	4	5	5
6	3	4	4	5	5
7	2	2	3	4	4

Figure A.5.: Questionnaire Results 3

1	It was very easy to understand, what I have to do in the ReWaMP process.	WASM-T was very easy to use.	I had to do a lot of manual work at the ReWaMP process.	WASM-T carried off much work for me in the ReWaMP process.	Without WASM-T the ReWaMP process would generate much more work for me.	The ReWaMP process took me only a short amount of time.	Without WASM-T the ReWaMP process would take me much longer.	WASM-T supported me very good at finding code fragments, which need an expert change.	WASM-T supported me at writing code fragments by offering ReWaMP flags.	WASM-T supported me very good at exporting functions to the server.	The most difficult task was for me...
2	2	5	3	5	4	3	5	5	4	5	make expert changes
3	2	3	2	4	4	4	4	3	3	4	make expert changes
4	1	4	3	4	4	2	4	4	4	4	make expert changes
5	4	3	3	4	5	2	5	5	4	5	make expert changes
6	3	5	2	5	5	3	5	4	3	5	make expert changes
7	2	3	3	4	4	4	4	1	4	5	make expert changes

Figure A.6.: Questionnaire Results 4

1	Average Processor Time	Other Processes	CPU-Time 1	CPU-Time 2	CPU-Time 3	Total Duration	t_M	t_T	e_M	e_T
2	5.55%	85	5.61%	5.46%	5.59%	02:06:36	02:05:11	00:01:25	256	688
3	12.11%	86	10.70%	13.55%	12.09%	01:14:27	01:12:55	00:01:32	248	668
4	6.00%	85	4.90%	4.71%	8.40%	02:00:30	01:58:51	00:01:39	247	688
5	15.49%	86	12.09%	16.73%	17.65%	01:20:34	01:18:56	00:01:38	240	668
6	23.60%	85	24.81%	22.30%	23.68%	01:42:19	01:40:41	00:01:38	232	668
7	9.60%	82	8.63%	10.54%	9.64%	01:26:19	01:24:43	00:01:36	231	678

Figure A.7.: Total Measurements

	Time Action 1	Time Action 2	Time Action 3	Time Action 4	Time Action 5	Time Action 6	Time Action 7	Time Action 3	Time Action 4	Time Action 5	Time Action 6	Time Action 7
1												
2	00:38:39	00:03:50	00:00:17	00:40:02	00:01:45	00:00:10	00:00:09	00:00:18	00:32:04	00:08:51	00:00:21	00:00:11
3	00:11:13	00:01:46	00:00:18	00:25:06	00:01:38	00:00:12	00:00:11	00:00:18	00:25:36	00:07:36	00:00:21	00:00:12
4	00:28:06	00:01:40	00:00:18	00:40:25	00:02:39	00:00:12	00:00:18	00:00:18	00:37:23	00:08:38	00:00:21	00:00:12
5	00:15:34	00:02:32	00:00:19	00:29:48	00:01:10	00:00:13	00:00:14	00:00:18	00:22:35	00:07:17	00:00:21	00:00:13
6	00:26:00	00:02:46	00:00:18	00:38:34	00:04:32	00:00:14	00:00:15	00:00:18	00:21:54	00:06:55	00:00:21	00:00:11
7	00:16:53	00:01:26	00:00:18	00:32:29	00:02:12	00:00:14	00:00:13	00:00:19	00:23:47	00:07:56	00:00:22	00:00:11

(a) Duration Measurements

	e_M4	e_M5	e_M4	e_M5	e Create 4	e Update 4	e Delete 4	e Create 5	e Update 5	e Delete 5	e Create 4	e Update 4	e Delete 4	e Create 5	e Update 5	e Delete 5
1																
2	59	9	128	60	3	11	45	9	0	0	0	1	33	94	60	0
3	57	6	128	57	4	10	43	6	0	0	0	2	30	96	57	0
4	49	9	129	60	2	10	37	9	0	0	0	0	36	93	60	0
5	52	6	125	57	1	14	37	6	0	0	0	0	36	89	57	0
6	51	6	118	57	2	14	35	6	0	0	0	0	30	88	57	0
7	43	6	122	60	2	6	35	6	0	0	0	0	32	90	60	0

(b) e_M Measurements

	e_T3	e_T6	e_T3	e_T6	e Create 3	e Update 3	e Delete 3	e Create 6	e Update 6	e Delete 6	e Create 3	e Update 3	e Delete 3	e Create 6	e Update 6	e Delete 6
1																
2	150	274	87	177	9	81	60	273	1	0	12	32	43	164	9	4
3	150	264	87	167	9	81	60	263	1	0	12	32	43	154	9	4
4	150	274	87	177	9	81	60	273	1	0	12	32	43	164	9	4
5	150	264	87	167	9	81	60	263	1	0	12	32	43	154	9	4
6	150	264	87	167	9	81	60	263	1	0	12	32	43	154	9	4
7	150	264	87	177	9	81	60	263	1	0	12	32	43	164	9	4

(c) e_T Measurements

Figure A.8.: Detailed Measurements Results

Appendix B.

ReWaMP Paper

The next 14 pages will append a paper written during the writing phase of this thesis. It was further submitted to the 18TH INTERNATIONAL CONFERENCE ON WEB ENGINEERING.

ReWaMP: Rapid Web Migration Prototyping leveraging WebAssembly

Sebastian Heil, Valentin Siegert, and Martin Gaedke

Technische Universität Chemnitz, Germany
{sebastian.heil,valentin.siegert,martin.gaedke}@informatik.tu-chemnitz.de

Abstract. Web Migration is a challenge, in particular for Small and Medium-sized Enterprises (SMEs). In previous collaborations with SMEs we noticed an initial resistance to migrate legacy desktop applications to the web, due to concerns about the risk and lack of developers with web expertise. This initial hurdle can be mitigated by the ability to rapidly create running web prototypes based on the existing desktop codebase and expertise of the developers. Therefore, we outline a rapid prototyping approach for Web Migration, assess technologies which allow execution of legacy code in the browser and present a solution architecture, process and supporting infrastructure based on WebAssembly. We describe challenges and report on an experiment applying WebAssembly on a scenario desktop application derived from real-world industrial code.

Keywords: Web Migration, Prototyping, WebAssembly, Software Reuse

1 Introduction

Migration of legacy systems (LS) to the Web is an important challenge for software developing companies. On the one hand, user expectations, advantages of platform-independent deployment [2] and lack of long term support for obsolete technologies provide a rationale for renewing legacy software as web applications. Many companies are aware of these reasons for web migration (WM). On the other hand, in particular Small and Medium-sized Enterprises (SMEs) find it difficult to commence a WM [6]. In several previous projects in collaboration with SMEs, we noticed initial resistance to migrate legacy desktop applications to the web. We refer to this resistance as *initial hurdle*.

Using LFA[1] problem trees, we identified two main problems that contribute to the notion of risk which keeps SMEs from commencing a WM: *doubts about the desirability* and *doubts about the feasibility*. Desirability concerns are rooted in a lack of knowledge of current web technologies and thus possibilities and advantages. Feasibility concerns are mainly due to a lack of employees with web development expertise in companies which have focused on the development of desktop applications. Solving this problem means to lower the initial hurdle of WM for SMEs by demonstrating desirability and feasibility to these companies.

[1] Logical Framework Approach, cf. http://ec.europa.eu/europeaid/aid-delivery-methods-project-cycle-management-guidelines-vol-1_en

To address similar concerns in (forward) web engineering, prototyping techniques are common. Agile methodologies which found wide adoption in the software industry during the last years have promoted prototype-first development. Influenced by Design Thinking methods like in the HCD toolkit [7], *rapid prototyping* and iterative development are used to involve stakeholders and gather early feedback. The prototypes allow to explore the solution space in the two dimensions of desirability and feasibility [7]. The running software prototype serves as a proof-of-concept. To demonstrate technical feasibility of the solution idea, required technologies and frameworks are explored in the prototype. On the other hand, the prototype demonstrates core features of the solution idea. Thus, software prototypes produced at the beginning of a web engineering project are used as a means of communication, presenting the main characteristics of the proposed solution to the stakeholders and allowing to interactively gather feedback. In this paper, we explore how the rapid prototyping paradigm can be transfered from forward web engineering to the field of web migration.

One of the key characteristics of rapid prototyping is the limited time and effort which is used to create the prototypes. To achieve this in the field of web migration prototyping, a suitable process and tools are necessary. Existing prototyping methods from forward web engineering like [13] can be re-used to support parts of this process which require development from scratch because of the paradigm shift involved when turning an application to a web application. To meet the effort and time requirements of rapid prototyping, re-use of as much of the legacy code as possible is crucial. Technologies like WebAssembly[2], Google Native Client[3] or emscripten[4] allow to run non-JavaScript programs in the browser by compilation into JavaScript or by providing a runtime environment.

This makes them interesting for rapid WM prototyping, since it would allow to execute legacy source code in the browser. However, the potential of these technologies for web migration has not yet been assessed. In contrast to a full WM, creating a WM prototype consisting of slightly adapted pieces of legacy code and newly implemented components in HTML/CSS/Javascript requires a lower amount of Web Engineering expertise. Instead, it can be primarily performed with existing staff experienced in the legacy technology base.

We describe our approach in section 2, present requirements and the assessment of technologies in section 3, detail our WebAssembly-based method in section 4, report on our findings in a validation experiment in section 5, provide an overview on related work in section 6 and conclude with a roadmap in section 7.

2 Approach

To address the initial hurdle described in section 1, we propose the *Rapid Web Migration Prototyping (ReWaMP)* approach for enabling web engineers to quickly

[2] `http://webassembly.org/`
[3] `https://developer.chrome.com/native-client`
[4] `https://github.com/kripken/emscripten`

create running web application prototypes from legacy code. As shown in fig. 1, ReWaMP requires three **artifacts**: legacy code, UI mockups and annotations.

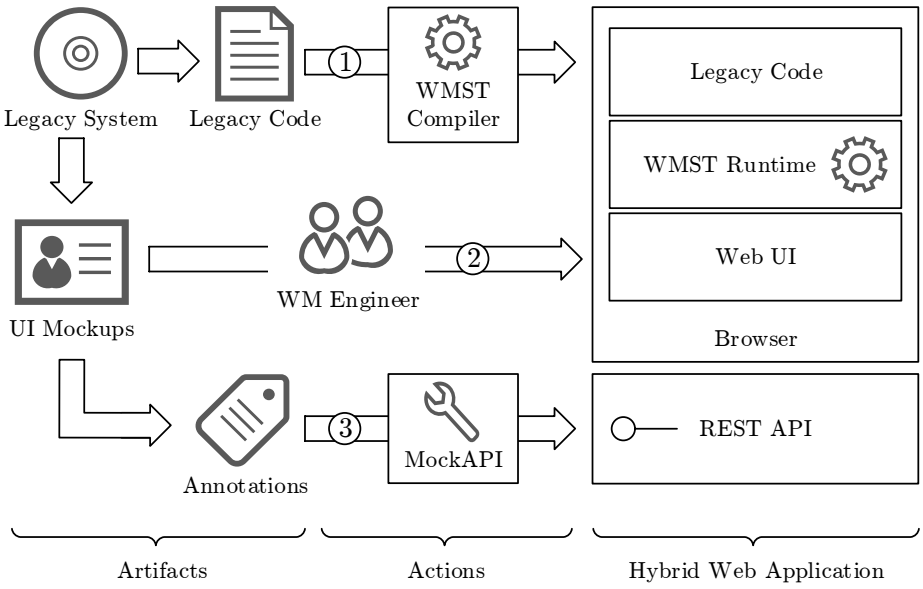


Fig. 1. ReWaMP Overview

The **legacy code** is the source code of the LS which should be transformed into a web application. ReWaMP assumes that the legacy source code is available, which typically is the case in enterprise software development contexts, where legacy systems are still actively operated, maintained and even developed.

UI mockups are visual sketches of the user interface. They are a common artifact in agile development, where they are used to represent the user interface of the application to be developed at a very early stage, even before a running software prototype is created. Thus, they allow to get early feedback from the target users on the understandability and information structuring of the user interface and can help to identify problems in the process or the conceptual abstractions of the underlying domain model. Mockups are traditionally created manually, as hand-drawn rough sketches. Addtionally, there are many software tools for mockup creation like Baslamiq[5]. Since the LS is still a running software, UI mockups can be created from screenshots of the legacy UI with minimal effort in the first iteration of our approach. Over time, the screenshot mockups can be replaced and incrementally improved towards a modern web UI, taking into account similarity aspects in order to smoothly transition from the legacy layout to a potentially different web UI layout [5].

[5] `https://balsamiq.com/`

Based on the initial artifacts legacy code and UI mockups, in ReWaMP, the migration engineer creates the web application prototype by three **actions**:

1. Leveraging a suitable web migration support technology (WMST), he creates an adapted version of the legacy code artifact which can be run in the browser.
2. Based on the UI mockups, he implements a web UI using HTML, CSS & JS.
3. By annotating the UI mockups, he generates a REST API backend (cf. [13])

This paper focuses on action 1. In section 3 we assess potential WMST for action 1 and provide details on this action in section 4. For action 2 we are currently investigating automatic transformations of legacy UIs to web-based UIs, in particular from absolute layouts to grid-based layouts under similarity constraints [5] using optimization algorithms. Action 3 integrates our previous work [13, 12] from forward web engineering contexts into this migration prototyping scenario. For this, the UI mockups are annotated with formal specifications used to automatically generate a prototypical RESTful API implementation. This API is intended to help building the first iterations of web frontends with limited time and effort, which suits well to the migration prototyping usage. The annotation is a stereotyping strategy, using different predefined tag types: *Data tags* specify the different types of objects that are managed by the API and represent business entities. *Constraints* define business rules which must be satisfied by the API regarding the data. *Actions* are tags describing the execution of heterogeneous or complex business logic, manipulating the data managed by the API.

The resulting prototype is a hybrid web application since it combines legacy code with a generated REST API and a newly developed web UI. It can be used to demonstrate feasibility and desirability of a WM with limited effort. From this interim stage, the web application can be incrementally transformed into a real web application by migrating the business logic contained in the legacy code towards client- or server-side web programming platforms. Therefore, ReWaMP is a rapid prototyping approach for WM.

3 ReWaMP Web Migration Support Technologies

This section provides an overview on technologies supporting the process described in section 2. They are assessed according to the following requirements:

R1 Limitations
R2 Platform and OS support
R3 Currentness of the technology
R4 Prevalence and supporting partners

Limitations refers to aspects that cannot be achieved with the given WMST. Platform and OS support determines whether a technology is compatible with many platforms and operating systems. Currentness of the technology considers whether the technology is still actively maintained. Prevalence and supporting partners assesses how widespread a technology is and which partners (institutions/companies) support it.

Today's legacy systems are desktop applications developed between the mid 1980s and late 1990s, i.e., before web applications became widespread. In this time frame, C and C++ were the two leading programming languages[6]. Therefore, ReWaMP focuses on C/C++ legacy code. As shown in table 1, we consider seven WMST which allow executing legacy code in the browser. They can be grouped into compilers producing JavaScript, runtimes, browser plugins and assembly languages. In the following, we briefly introduce each technology and report on

Table 1. Web Migration Support Technology Groups and Implementations

Group	Web Migration Support Technology
Compilers producing JavaScript	emscripten
	cheerp
Runtimes	Google Native Client
Browser Plugin APIs	NPAPI
	PPAPI
	js-ctypes
Assembly Languages	WebAssembly

their assessment according to the requirements.

Emscripten. Emscripten [17] is an LLVM-based compiler which allows compilation of C/C++ Code into JavaScript. Using Clang, the C/C++ code is compiled into LLVM bitcode and then compiled to JavaScript by Fastcomp. The resulting code is in asm.js, a subset of JavaScript whith statical typing for all operations. Any other language compilable to LLVM bitcode can be used as initial input or the runtime can be compiled to allow execution of interpreted languages. Dependencies have to be available as source code for compilation with emscripten. The resulting program is not platform-specific or OS-specific and can be executed by any browser supporting JavaScript.

Cheerp. Cheerp[7] is another LLVM-based C++ to JavaScript compiler. In addition, it provides a simple API for DOM access and manipulation. Emscripten and cheerp differ in the way how to expose a C/C++ API. Cheerp is more restricted, lacking support for virtual methods or destructors and handling exports based on classes. While being conceptually comparable to emscripten, cheerp requires more code adaptions due to these limitations.

Google Native Client. Google Native Client (NaCl) allows to execute natively compiled source code in the browser. It is implemented as browser plugin communicating with the browser via PPAPI. In addition to NaCl, which uses OS-independent architecture-specific executables, Portable Native Client (PNaCl) uses OS- and architecture-independent executables. Both are compiled using the LLVM Compiler. NaCl programs are directly run at runtime, PNaCl programs are compiled from the portable into a host-specific executable prior

[6] cf. `http://www.tiobe.com/tiobe-index/`
[7] `http://www.leaningtech.com/cheerp/`

to first execution. For security reasons, (P)NaCl programs are run in a sandbox which restricts interactions with the host system using Software Fault Isolation. This requires compilation using the NaCl SDK. Thus, dependency source code has to be available. Google Chrome supports NaCl for Windows, Linux, OS X and Chrome OS, mobile versions and other browsers do not support NaCl.

NPAPI-based Browser Plugins. Netscape Plug-in Application Programming Interface (NPAPI) allows browser plugins to access OS resources and manipulate the DOM. They can be implemented in C/C++. The scripting feature allows interaction with JavaScript. A ReWaMP prototype could be created by means of an NPAPI-based C/C++ browser plugin that interacts with the web UI. Frameworks like FireBreath[8] allow for platform-independent development plugins. With FireBreath, dependencies like MFC[10] cannot be linked at compile-time. However, it is possible to load them at runtime as DLL. In 03/2017, support for NPAPI was dropped by major browsers due to security implications.

PPAPI-based Browser Plugins. The Pepper Plugin API (PPAPI) is an API similar to NPAPI aiming at better security and higher portability. Like NPAPI, PPAPI allows using OS resources, DOM manipulation and interaction with JavaScript. As of 01/2018, PPAPI is still actively maintained. However, browser support is limited to chromium-based browsers. Also, dependencies have to be available as source code at compile time.

Js-ctypes-based Browser Plugins. Js-ctypes[9] is a library that allows Firefox extensions to call native C functions from within JavaScript. The C functions called are contained in either the shared standard libraries of the OS or in custom library files such as DLLs. Functions from shared libraries can be loaded by their path. Custom functions have to be shipped in custom library files together with the extension. Declared and loaded through the js-ctypes API, C functions are available as JavaScript functions. C++ functions are supported through vtable pointers. Legacy C/C++ functionality as well as dependencies could be loaded and used from JavaScript. However, access to the js-ctypes API is restricted to extensions. Moreover, it is only supported by Firefox and derivatives.

WebAssembly. WebAssembly (WASM) is an open standard currently designed by the W3C WebAssembly Community Group of a format for compilation to the web aiming at portability and size- and load-time-efficiency. The group includes all major browser providers i.e., Google, Mozilla, Apple and Microsoft. WASM combines experience from previous work on emscripten and NaCl. It defines binary format optimized for fast execution and a textual format for pretty-printing for human source readers. Conversion between the two is possible using the WebAssembly Binary Toolkit. Using parts of the emscripten toolchain, C/C++ code can be compiled to WASM. Dynamic linking is supported by WASM. As of 01/2018, wasm has reached cross-browser consensus with preview implementations in all major browsers.

To conclude the assessment of ReWaMP technologies, WASM has the fewest limitations, is compatible with major platforms, is the most current and actively

[8] `http://firebreath.org/`
[9] `https://developer.mozilla.org/en-US/docs/Mozilla/js-ctypes`

developed technology and sees wide support from major companies due to being an open standard. Therefore, we showcase our prototypical implementation of a ReWaMP toolchain based on wasm in the following section.

4 WebAssembly-based Rapid Web Migration Prototyping

In this section, we detail the WASM-based ReWaMP approach for action 1 of section 2 For this, we first describe the architecture of the resulting web migration prototype, detail the ReWaMP process and report on the main challenges.

4.1 Architecture of ReWaMP Prototypes

Detailing the client side of fig. 1, the architecture of ReWaMP prototypes as shown in fig. 2 consists of 4 components: HTML/CSS, WASM, JS Glue Code and ReWaMP JS Code.

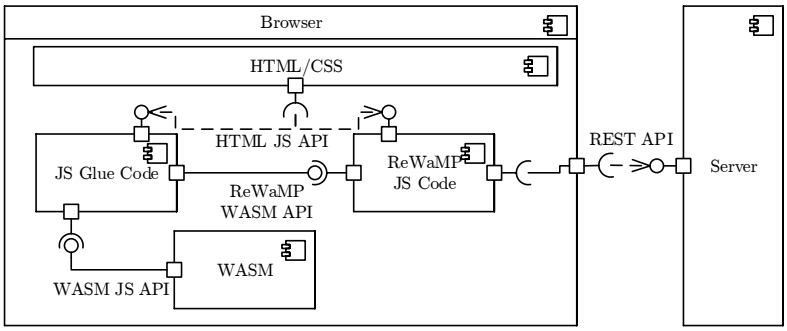


Fig. 2. Architecture of ReWaMP Prototypes

HTML/CSS. The HTML/CSS component describes the layout and content structure of the Web UI as created in action 2 of fig. 1. The behavior of the Web UI is handled the generated WASM Code of the prototype. Access to the Web UI is achieved through ReWaMP JS Code via the DOM API of JS.

WASM. The WASM Code component is derived by compilation of the legacy view behavior code to WASM. For each user interface, the corresponding WASM file includes all UI functionality and semantics. To enable communication with other components, it is bound to the JS glue code.

JS Glue Code. The JS glue code is created at the compilation of legacy code to WASM by the emscripten compiler. It provides the JS API of WASM, and thereby standard functionalities like WASM memory management and serialization of strings for message exchange. The glue code can be configured for the exact view context via specific compilation parameters. It functions as a bridge for communication between the ReWaMP JS code and WASM Code.

ReWaMP JS Code. The ReWaMP JS code provides an infrastructure for the communication with the server side and for abstracted access to the generated Web UI. For example, it allows to get values of HTML form elements and to send information to the server, handling serialization etc. It therefore provides web migration prototyping-specific API extensions for the JS API of WASM required to control the behavior from legacy code compiled to WASM.

This architecture allows re-use of legacy view behavior code through WASM. The web-based frontend of the migration prototype interacts with the server side through a RESTful API resulting from action 3 for data access and advanced business logic. Re-use of legacy code on the server side has been described in related work [10].

4.2 ReWaMP Process

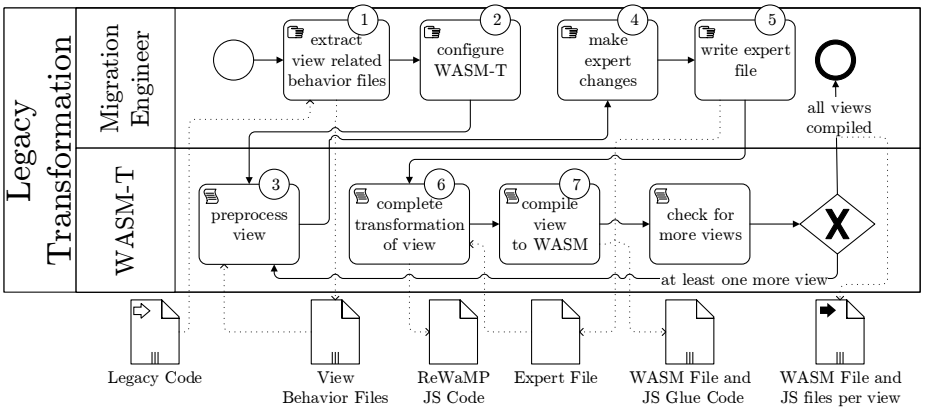


Fig. 3. ReWaMP Process

The ReWaMP process (cf. fig. 3) defines the necessary steps for creating a web migration prototype according to the architecture introduced above. It creates the WASM Code, JS Glue Code and ReWaMP JS Code for each view of the legacy system. This is achieved through extraction of relevant information from the legacy code, transformation in a WASM-compilable version and compilation to the WASM target. While the compilation of C++ code to WASM is supported by emscripten, the transformation is not trivial. It is required to remove all unresolvable dependencies after the extraction. This task depends on the desktop UI library used by the legacy code and has to be performed manually by the migration engineer. To support the rapid creation of WASM files, we developed a prototypical WASM Transformator (WASM-T) for legacy C++ code bases using

the MFC framework[10]. The responsibilities of the migration engineer and the WASM-T are represented as lanes in fig. 3.

To start the process, the migration engineer (1) extracts the source files describing view behavior from the legacy code and (2) configures WASM-T, specifying file location and the main file per view. Comparable to main functions in C++, main files are those which directly interact with a dialog or a window, containing event handlers for UI elements. They serve as starting point for the creation of the WASM file. Additionally, related files are required. They are other files referenced from the main files, which are also part of the view layer. In MFC, most of the related files are MFC dialog items like menus or toolbars.

To make it compilable to WASM, the WASM-T transforms the extracted legacy code. Since automated semantic analysis of source code is complex and the code can reference dependencies which are not available (eg. binary assemblies), a semi-automated process is required. In (3), the WASM-T performs pre-processing by deleting or rewriting GUI framework-specific parts of the code. It replaces MFC types as `CString` with `std::string`, deletes calls to MFC classes and rewrites well known functions of the UI library like `DoDataExchange` in MFC dialogs. Also, it extracts information required for the correct coupling of the Web UI with the legacy code, as found in message maps in MFC.

Following the pre-processing, the migration engineer re-engineers code parts WASM-T was not able to transform and provides required changes. He does that either via expert changes in the code (4) or within a separate expert file (5). Expert changes replace or remove complex constructs to resolve missing dependencies. The expert file contains missing declarations and definitions of classes, variables and functions. For providing this information, not much knowledge about the legacy code is required, since it can be found in the related files. The engineer copies signatures of missing functions with their parameters, parameter types and return type. Function bodies are later filled in (6). As common in prototyping, mocks can be employed for abstracting complex functionality. Mock classes in the expert files provide a limited abstraction, comparable to a DTO (data transfer object), of their legacy counterparts, including only the fields and functions required by the view. Full automation of these tasks is out of scope for this exploration of WASM in web migration prototyping.

To control the transformation process, migration engineers can use *ReWaMP flags*. These flags are comments with a special syntax similar to precompile statements and are markers for specific code transformations such as replacement of method calls due to differing types.

The WASM-T completes the transformation (6) by generating the bodies of empty functions in the expert files, calling the required functionality on the server-side and handling the responses. It also adds stringify methods for all classes to support serialization for data transmission. Transformation is influenced by the ReWaMP flags. Code is dynamically generated, allowing the C++ Classes to be accessible from JavaScript. Then, the code is compiled to WASM (7).

[10] `https://docs.microsoft.com/de-de/cpp/mfc/user-interface-elements-mfc`

4.3 Challenges

As WebAssembly is not explicitely targeted at migration prototyping, we faced several challenges in our experiment with WASM, which we describe in the following. They also indicate potential enhancements of WASM towards WM.

Serialization. Data has to flow between WASM and JavaScript in both directions. WASM only supports passing 32 and 64 bit integers and floats. Thus, transferring arrays, strings or objects from or to WASM is not straightforward. Thus, to transfer arrays or objects that are instances of classes or structs, we serialize them to JSON Strings. Transferring strings via pointers to their addresses is supported by the generated glue code.

Client/Server Communication. Client/Server communication is an important characteristic of a web application, also represented in ReWaMP prototypes. After initialization of the WASM file, all requests are triggered by WASM. The requests are bridged through the ReWaMP JS Code. Thus, the legacy code is transformed by WASM-T to use the JS API to call a ReWaMP JS function which sends the transmitted string to the server's REST API via AJAX. All required information is included by the transformed code into one JSON string. The response is relayed back to the WASM Code where C++ variables and objects are instantiated accordingly.

DOM Access in WASM Code. As the UI can change dynamically or receive user input, the WASM Code needs to access the DOM. This includes manipulation of UI structure, elements, and layout and retrieving input values. These DOM accesses are mapped to calls of ReWaMP JS functions by WASM-T.

Handling UI Events in WASM Code. In the legacy UIs, user interaction trigger events which are handled by functions in the legacy code. To react on the events in the web UI, they have to be mapped to the corresponding functions in WASM Code. ReWaMP JS Code initializes the HTML page so that UI events are forwarded to the correct functions in WASM Code.

5 Validation

In this section, we describe the experimental validation of our approach. By experimentation with ReWaMP, we are interested in answering the following questions: How much time and effort are required? What expertise is required by the migration engineer? What are the longest/most difficult activities in ReWaMP? How does WASM perform in the context of WM?

Based on a real world legacy codebase by our industrial project partner, we abstracted characteristics and created a sample application to answer these questions. Table 2 provides an overview of the abstracted LS characteristics and their concrete instantiation in our scenario. The scenario is a medical appointment management application implemented in VC++, that allows users to manage appointments (basic CRUD functionality). Appointments have a related patient, doctor and resource (e.g., a room). As business logic beyond CRUD, the system can determine the next time slot available for an appointment with a given

combination of doctor and resource. The following non-functional characteristics are represented: GUI is implemented using a desktop GUI framework, MFC in our scenario. Third party module dependencies are loaded at runtime, in the sample application as DLLs via assembly loading. The application consists of several components which exchange messages via MFC SendMessage. Files are used to store configuration data e.g., opening hours, and appointments are stored in an MS SQL database via ODBC. The scenario codebase comprises 3373 LOC, including 187 functions and 31 classes.

Table 2. Abstract Characteristics of Legacy Software and Scenario Instantiation

Abstract Characteristics	Instantiation in scenario
Legacy language	Visual C++
CRUD functionality	CRUD for medical appointments
Complex Business Logic	Find next available time slot
Desktop GUI	Microsoft Foundation Class (MFC)
Component Communication	Window-to-Window via MFC SendMessage
Third-party dependencies	DLLs via assembly loading
File and Data Base Persistence	Files and MS SQL Server via ODBC

5.1 Procedure

The scenario codebase was created by two researchers. Two other researchers conducted the validation experiment according to the following procedure. One researcher defined the experiment, measurements and observed, while the other researcher was the test subject taking the role of the migration engineer.

The test subject followed the steps described in fig. 3. First he extracted the view behaviour files, then configured WASM-T accordingly and finally made the expert change for each view. To extract the files and perform the expert changes, Visual Studio was used to navigate the code base.

Since rapid prototyping requires a quick and easy creation of prototypes, we measure time and effort in the experiment. **Time** has two main components, $t_M = t_M^{12} + t_M^{45}$ the time required by the migration engineer to perform actions 1&2 and 3&4, and t_T the time required by the supporting infrastructure to analyze and transform the legacy code. Measurement of t_M was achieved by stopwatch, t_T was measured using python's time library. Time measurement for the migration engineer was started at the start of manual tasks after preparation of the tools, interrupted if WASM-T started or resumed processing and ended by finishing the last expert changes. t_T is dependent on the system running the transformations. We used Intel Core i7 CPU 930 @ 2,8 GHz with 14 GB RAM on Windows 10 Education N x64. During measurement 87 other processes were active and an average load of 13,793% was determined with Window's performance monitor.
Effort evaluates the amount and extent of changes to the existing legacy code required by ReWaMP. We measure the effort in terms of e_T - the lines of code (LOC)

added/changed/deleted by WASM-T and e_M - the LOC added/changed/deleted manually by the migration engineer. e_M was measured by observing the test subject's actions, and e_T by comparing the inputs and results of one continious task set after the migration.

5.2 Results

The resulting ReWaMP prototype is available online[11]. Table 3 shows the measurements. We were surprised to see that the related files extraction did not take that much time (t_M^{12}). Here, expertise in the legacy UI framework is advantageous. We expected a lower ratio of e_M to the overall effort $e_M + e_T$, which is at 29%. This was caused by low decomposability of one view in the scenario, comprising DLL loading and debug code. Removal of debug code and transfer of DLL loading to the server side impacted many lines and therefore significantly increased e_M.

The experiment showed that C++ experience and knowledge of the legacy UI framework are important. Basic web engineering expertise is helpful but not required. The most difficult task were the expert changes due to required legacy code comprehension. An interactive web prototype based on the legacy code could be created with reasonable time and effort through WASM-enabled code re-use. While a similar prototype can be created by an experienced web engineer in comparable time, the contribution of ReWaMP is to allow developers without web engineering expertise from the existing SME staff to achieve similar results.

Table 3. Time (in hh:mm:ss) and Effort

t_M	t_M^{12}	t_M^{45}	t_T	e_M	e_T
01:17:42	00:21:31	00:56:11	00:00:72	253LOC	597LOC

6 Related Work

The large body of research in the web migration field [2, 9, 8] focuses on program decomposition, language transformations, legacy to SOA and legacy to Cloud migrations. Only few approaches consider usage of a WMST. The *ReWeb3D* framework [4] focuses on the migration of legacy 3D desktop software towards WebGL using the emscripten compiler. *Doppio* [15] is a framework providing services like file system access, heap, multithreading and reflection for supporting execution of non-JavaScript programs in the browser. A use case is presented which extends emscripten with Doppio for executing a C++ game in the browser. *O2WebCL* [3] supports migration of computationally intense applications (physics engines, video processing etc.) through transformation from OpenCL to WebCL

[11] `https://vsr.informatik.tu-chemnitz.de/demos/ReWaMP`

based on emscripten. These approaches target only specific domains. Technologies other than emscripten are not considered.
Overall, consideration of *emscripten* [17] and *Native Client* [16] in scientific publications is sparse. Emscripten is mainly considered with regard to compiler features and optimization. The majority of papers about Native Client address security, a few address performance. We could not find any systematic consideration of these technologies in the context of web migration. With WebAssembly still being relatively new and a draft standard under development, there are no relevant publications available.

Prototyping approaches have seen a long consideration in the development of information systems. They provide a means to create an early version of an information system to be developed, which exhibits its essential features and can be continuously evolved into the production system. [1]

For software engineering, rapid prototyping has been acknowledged to increase the efficiency and effectiveness of software design [14]. By building and using a model of the system, representing the persistence, major business logic and user interface, it allows to gather feedback from users and to explore requirements. The availability of tool support for software prototyping plays a crucial role for its success [14]. In our work, we therefore seek to support the rapid web migration process by a WASM-based infrastructure. Appropriateness of rapid prototyping particularly includes new situations with limited experience to draw from [14]. For ReWaMP, the new situation is formed by the new technological environment of the web application, which significantly differs from the desktop application environment of the existing legacy system and bears uncertainties regarding feasibility and desirability.

It has to be noted that rapid prototyping as a methodology has often seen transfer into new domains. As a concept originating in the hardware development domain, it has been transfered into software engineering, especially in the context of information systems [1], UI Design [11] and even seen adoption for instructional design [14]. Likewise, in this paper we explore the application of rapid prototyping in the context of web migration, enabled by technologies like WebAssembly.

7 Conclusions and Roadmap

In this paper, we introduced ReWaMP, a novel rapid prototyping approach addressing the initial hurdle of web migration. ReWaMP enables migration engineers to quickly demonstrate desirability and feasibility. We outlined requirements and assessed technologies for supporting the approach, selecting WebAssembly as the most promising. Based on WASM, we presented a prototype architecture, creation process and supporting infrastructure for web migration prototyping, outlined challenges when applying WASM for WM prototyping and reported on the results of a validation experiment. Our next steps are investigation of automatic transformations of legacy UIs to web-based UIs under similarity constraints using optimization algorithms, the enhancement of automation of the supporting infrastructure and evaluation of the method in industrial context.

Acknowledgment. The authors would like to thank Thomas Blasek and Tobias Lang for their valuable contributions. This research was supported by the eHealth Research Laboratory funded by medatixx GmbH & Co. KG.

References

1. Alavi, M.: An assessment of the prototyping approach to information systems development. Communications of the ACM 27(6), 556–563 (June 1984)
2. Aversano, L., et al.: Migrating legacy systems to the Web: an experience report. In: Proc. of CSMR'01. pp. 148–157. IEEE Comput. Soc (2001)
3. Cho, M., et al.: O2WebCL : An automatic OpenCL-to-WebCL Translator for High Performance Web Computing. The Journal of Supercomputing pp. 2050–2065 (2015)
4. Glander, T., et al.: ReWeb3D - Enabling Desktop 3D Applications to Run in the Web. In: Proc. of Web3D'13. pp. 147–155. ACM Press (2013)
5. Heil, S., Bakaev, M., Gaedke, M.: Measuring and ensuring similarity of user interfaces: The impact of web layout. In: Proc. of WISE 2016. pp. 252–260. Springer International Publishing (2016)
6. Heil, S., Gaedke, M.: Web Migration - A Survey Considering the SME Perspective. In: Proc. of ENASE'17. pp. 255–262. SCITEPRESS (2017)
7. IDEO.org: The Field Guide to Human-Centered Design. IDEO (2015)
8. Jamshidi, P., Ahmad, A., Pahl, C.: Cloud Migration Research: A Systematic Review. IEEE Transactions on Cloud Computing 1(2), 142–157 (2013)
9. Khadka, R., et al.: Legacy to SOA Evolution: A Systematic Literature Review. In: Migrating Legacy Applications, chap. 3, pp. 40–71. IGI Global (2013)
10. Lucia, A., et al.: A Strategy and an Eclipse Based Environment for the Migration of Legacy Systems to Multi-tier Web-based Architectures. In: Proc. of ICSM'06. pp. 438–447. IEEE (September 2006)
11. Nebeling, M., Leone, S., Norrie, M.C.: Crowdsourced Web Engineering and Design. In: Web Engineering, pp. 31–45. Springer Berlin Heidelberg (2012)
12. Rivero, J.M., et al.: An Extensible, Model-Driven and End-User Centric Approach for API Building. In: Web Engineering. vol. 8541, pp. 494–497. Springer, Cham (2014)
13. Rivero, J., et al.: MockAPI: An Agile Approach Supporting API-first Web Application Development. In: Web Engineering. pp. 7–21. Springer Berlin Heidelberg (2013)
14. Tripp, S.D., Bichelmeyer, B.: Rapid prototyping: An alternative instructional design strategy. Educational Technology Research and Development 38(1), 31–44 (March 1990)
15. Vilk, J., Berger, E.D.: DOPPIO : Breaking the Browser Language Barrier. ACM SIGPLAN Notices 49(6), 508–518 (2014)
16. Yee, B., et al.: Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 79–93. IEEE (May 2009)
17. Zakai, A.: Emscripten: an LLVM-to-JavaScript compiler. In: Proc. of OOPSLA '11. pp. 301–312 (2011)

TECHNISCHE UNIVERSITÄT CHEMNITZ

Studentenservice – Zentrales Prüfungsamt

Selbstständigkeitserklärung

Name: Siegert Vorname: Valentin geb. am: 29.03.1993 Matr.-Nr.: 426161	Bitte beachten: 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: 08.02.2018 Unterschrift: ……

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.