HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

MEMORY
Clinic

felix
platter

Test instrument for early diagnosis of Alzheimer's disease

Term Project

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2019

Author(s): Jan Frei, Cedric Wehli
Advisor: Prof. Dr. Markus Stolze
Project Partner: Dr. Sabine Krumm,
Memory Clinic, Basel

Table of Contents

1 Abstract 3
2 Lay-Summary 4
3 Management Summary 5
3.1 Initial situation 5
3.2 Approach / Technologies 5
3.3 Results 5
3.4 Outlook 5
4 Project plan 6
4.1 Project overview 6
4.1.1 Purpose 6
4.1.2 Deliveries 6
4.1.3 Assumptions and limitations 6
4.2 Project organization 6
4.2.1 Important Dates 6
4.3 Time Management 7
4.3.1 Timeline 7
4.3.2 Phases 7
4.3.3 Sprints 7
4.3.4 Milestones 7
4.4 Risk Management 9
4.5 Quality Management 12
4.6 Infrastructure 12
4.6.1 Development Tools (on our Clients) 12
4.7 Definition of Done 13
5 Engineering Documents 14
5.1 Requirements 14
5.1.1 Use Cases 14
5.1.2 Non-Functional Requirements 15
5.1.3 State Diagram and Screen Flow 16
5.2 Architecture & Design 19
5.2.1 System Context Diagram 19
5.2.2 Domain Model 19
5.2.3 Class Diagram/Data Schema 20
5.2.4 Sequence Diagram 21
5.2.5 Technology decisions 22
5.2.6 Development Infrastructure decisions 24
5.2.7 Virtualization / Deploying in a VM for extended lifetime 24
5.3 Results and Outlook 25
5.3.1 Results 25
5.3.2 Outlook 25
5.4 License report 26

1 Abstract

The team of the Memory Clinic located in the Felix Platter hospital for elderly people in Basel has developed a test instrument for early diagnosis of Alzheimer's disease. This diagnosis consists of three computer-assisted tests where the test persons answer memory related questions. Those were created on the basis of a proprietary platform.

To make this tool available for other clinics and experts the goal was to recreate these tests without the need to license an expensive software as well as improve the usability for the operator's needs in the clinic's daily business.

To create a software that runs the specified tests on all major desktop operating systems (Linux, Mac, Windows) in just three months with an intuitive user interface we decided to create an electron app on the basis of the following electron react boilerplate: https://electron-react-boilerplate.js.org

To comply with the legal requirements of FPS we split our application into a generic test execution shell that is able to read test specifications from JSON files and the specific configuration files with their Alzheimer's diagnosis tests. This allowed us to extract any external intellectual property into the configuration files which were handed to our client while the test execution shell could be published under the MIT open source license.
With this approach it is possible to expand the program or write similar experiments on the basis of the experiment execution shell by simply changing or creating JSON experiment configuration files.

2 Lay-Summary

The team of the Memory Clinic located in the Felix Platter hospital for elderly people in Basel has developed a test instrument for early diagnosis of Alzheimer's disease. This diagnosis consists of three computer-assisted tests where the test persons answers memory related questions. We developed a generic software that runs these experiments and allows to create new experiments in the future.

3 Management Summary

3.1 Initial situation

The team of the Memory Clinic located in the Felix Platter hospital for elderly people in Basel has developed a test instrument for early diagnosis of Alzheimer disease. This diagnosis consists of three computer-assisted tests where the test persons answers memory related questions. Those were created on the basis of a proprietary platform.

To make this tool available for other clinics and experts, the goal was to recreate these tests without the need to license an expensive software as well as improve the usability for the operator's needs in the clinic's daily business.

3.2 Approach / Technologies

To create a software that runs the specified tests on all major desktop operating systems (Linux, Mac, Windows) in just three months with an intuitive user interface we decided to create an electron app on the basis of the following react boilerplate: https://github.com/electron-react-boilerplate/electron-react-boilerplate

3.3 Results

We managed to develop a solution which meets all the requirements above. We split our application into a generic test execution shell that is able to read test specifications from JSON files and the specific configuration files with their Alzheimer diagnosis tests.

While some of the initially stated use cases didn’t make it into the delivered release the core requirements are fulfilled. This could be verified with on-site field tests and feedback from FPS.

At the time of writing, the software was not yet used in production.

3.4 Outlook

It is possible to expand the whole experiment program or write similar experiments with our execution shell by simply changing / adding JSON config files. This is not only true for FPS but for any user with a little technical knowledge.

FPS’s plan is to introduce the software in production/their daily operation.

Furthermore, they will also be able to make this tool available to other clinics without licensing issues since our software and all libraries are under unrestricted Open Source license.

The software itself could be further extended by providing the possibility to create own styles, use different export formats for the results or introduce new test types.

4 Project plan

4.1 Project overview

4.1.1 Purpose

Within this work, a software should be created that can replace the existing proprietary software and improves the usability for day-to-day operations.

4.1.2 Deliveries

The deliveries of this work include

- Working experiment execution shell
- Source code of the experiment execution shell
- Readme for further development of the experiment execution shell
- JSON Schema for experiment configuration files
- Documentation according to HSR guidelines
- Abstract
- Poster

Apart from the delivery to the HSR there is a separate delivery of results to the Memory Clinic of the FPS. This includes

- Configuration files to run tests for an early diagnosis of Alzheimer's disease

4.1.3 Assumptions and limitations

The project is limited to the requirements given in the problem definition and the specified effort and time frame which were made by the HSR.

4.2 Project organization

Name	Role	Email
Markus Stolze	Supervisor	markus.stolze@hsr.ch
Sabine Krumm	Industry Partner	sabine.krumm@felixplatter.ch
Jan Frei	Team Member	jan.frei@hsr.ch
Cedric Wehli	Team Member	cedric.wehli@hsr.ch

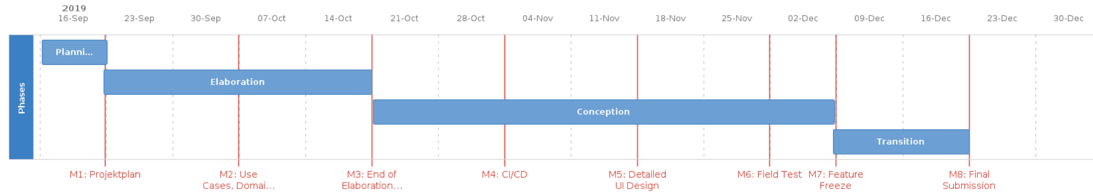
4.2.1 Important Dates

Title	Date
Kick-off	16 Sep 2019
Hand-in (abstract)	16 Dec 2019
Hand-in Document	20 Dec 2019

4.3 Time Management

In this chapter the single phases as well as the milestones of the project are described. The project takes place during 14 weeks beginning at the 16th of September and ending on the 23rd of December. The following image gives a rough overview.

4.3.1 Timeline



4.3.2 Phases

4.3.2.1 Planning

In the first week and at the beginning of the second week, the kickoff meeting with the supervisor and the industry partner is held. The infrastructure is evaluated and set up, since being required for creating the project documentation and the project management. The time management with its phases and milestones is defined and an overview of the general aspects of the project is worked out.

4.3.2.2 Elaboration

During the elaboration phase the missing milestones are specified, the problem domain is analyzed, the specification is worked out from the requirements, the use cases are documented and validated with the industry partner, different possible architectural solutions are evaluated and the deployment diagram is created. This will take approximately four weeks until completion and result in an architectural prototype.

4.3.2.3 Conception

In the seven weeks of the conception phase, the actual software is developed. Furthermore, some brief use cases can be further mapped out.

4.3.2.4 Transition

In the last three weeks of the project, the system is tested and evaluated, the project documentation is finalized and a manual for the software is written.

4.3.3 Sprints

Each phase is split into sprints of two weeks and sometimes a one week sprint to finish a phase. The concept of sprints and its usage is described in more detail in the project management chapter.

4.3.4 Milestones

Date	Milestone	Link(s):
25 Sep 2019	M1: Projectplan	this page
08 Oct 2019	M2: • Use Cases (all brief, some fully dressed) • Domain Model	

Date	Milestone	Link(s):
	• Requirements • Non-functional requirements	
22 Oct 2019	M3: End of Elaboration • GUI • (Software Architecture) • Architecture Prototype • Screen flows / Scribbles • State Diagram	
05 Nov 2019	M4: CI/CD	
19 Nov 2019	M5: Detailed UI Design	
03 Dec 2019	M6: Field Tests	
17 Dec 2019	M7: Feature Freeze	
22 Dec 2019	M8: Finalized Documentation and Report, Submission of Project	

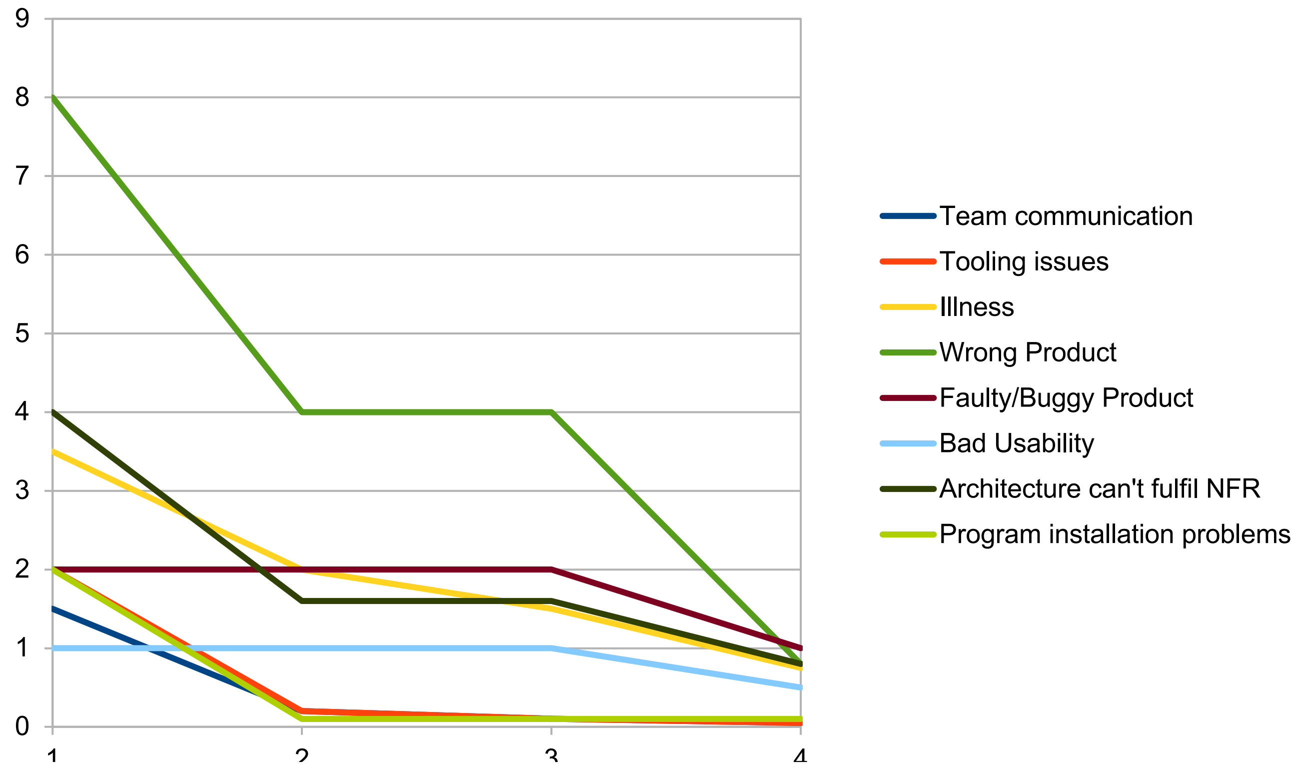
4.4 Risk Management

Nr	Title	Description	max. Damage [h]01 Oct 2019	Probability of occurrence01 Oct 2019	Risk (Damage * Probability) [h] 01 Oct 2019	Preventive Measures	Measures if described scenario occurs
1	Team communication issues	Communication and organization problems due to inefficient team communication.	30	0.05	1.5	Define scheduled Sprint reviews / plannings and other meetings.	More meetings, improved planning of Sprint (Review / Planning / etc)
2	Tooling issues	Some parts of our infrastructure don't fully work (bugs / misconfiguration / etc).	20	0.1	2	Research and evaluate all tools and set them up early. Use technologies which are well known / documented.	Resolve the issues or at worst, migrate to another tool.
3	Illnesses	Due to illness, deadlines need to be postponed.	70	0.05	3.5	Risk we have to live with. It is not possible to plan this much time reserves.	Improvise.
4	Wrong Product	Product does not do what the customer expected it to do	80	0.1	8	Early screen flows, usability tests, validation with customer	Reimplement / change parts of the product for it to fulfill the customer's requirements
5	Faulty / Buggy Product	Product has faults which lead to errors (i.e. crashes / errors in tests)	20	0.1	2	PR (four-eye-principle), Quality management	Fix bugs
6	Bad Usability	The usability of the product is not good enough, either	10	0.1	1	Acceptance of screen-flows, Acceptance of UI, Field test with Memory clinic	Improve bad usability

Nr	Title	Description	max. Damage [h]01 Oct 2019	Probability of occurrence01 Oct 2019	Risk (Damage * Probability) [h] 01 Oct 2019	Preventive Measures	Measures if described scenario occurs
		for the supervisor or for the patient (or both).					
7	Architecture can't fulfill NFR	Our chosen architecture / framework can't fulfill one or multiple non-functional requirements	80	0.05	4	Define NFR early and research whether architecture fulfills all NFR	Migrate architecture / framework
8	Program installation problems	The installation is faulty or doesn't work at all.	10	0.2	2	Use build pipeline on all required OS. Deliver prototype with installer early so any issues can be resolved as early as possible.	Fix installer issues

Risk analysis over time

Nr	Title	01.10.19			06.11.19			20.11.19			04.12.19		
		max damage	probability	risk	max damage	probability	risk	max damage	probability	risk	max damage	probability	risk
1	Team communication	30	0.05	1.5	20	0.01	0.2	10	0.01	0.1	5	0.01	0.05
2	Tooling issues	20	0.1	2	20	0.01	0.2	10	0.01	0.1	5	0.01	0.05
3	Illness	70	0.05	3.5	40	0.05	2	30	0.05	1.5	15	0.05	0.75
4	Wrong Product	80	0.1	8	80	0.05	4	80	0.05	4	80	0.01	0.8
5	Faulty/Buggy Product	20	0.1	2	20	0.1	2	20	0.1	2	20	0.05	1
6	Bad Usability	10	0.1	1	10	0.1	1	10	0.1	1	10	0.05	0.5
7	Architecture can't fulfil NFR	80	0.05	4	80	0.02	1.6	80	0.02	1.6	80	0.01	0.8
8	Program installation problems	10	0.2	2	10	0.01	0.1	10	0.01	0.1	10	0.01	0.1



4.5 Quality Management

The quality management is a dynamic process. It consists of different categories/steps.

Measurement	Goal	Time of Execution
Review of evaluations and specifications	• Four-eyes principle	Definition of Done
Pull requests and code reviews	• Increased code quality • Increased readability and understandability of code • Four-eyes principle	On pull request
Unit Test Coverage > 80%	• High maintainability • High flexibility • ligh confidence	From M2: end of elaboration on always
At least one integration test for each use case	• Integration problems are detected early on	From M2: end of elaboration on always
Automated build & test execution	• Product/software/program is ‘always buildable’	From M3: CI/CD on always
Evaluation of quality management execution	• Assure that the defined measures are respected	From the start, at each iteration
Field test	• Assure	

4.6 Infrastructure

Topic	Tool	Cloud / Self-hosted
Documentation	Confluence Cloud https://ced0.atlassian.net/wiki/spaces/SAMC	Cloud
Project Management	Jira Cloud https://ced0.atlassian.net/browse/SAMC	Cloud
Source Code Management	GitHub https://github.com/mr-perseus/memory-clinic	Cloud
CI/CD	Travis CI https://travis-ci.com/mr-perseus/memory-clinic	Cloud

4.6.1 Development Tools (on our Clients)

- Webstorm
- Git Client
- Nodejs with NPM libraries (Electron, React, etc)

4.7 Definition of Done

Definition of Done for stories / issues (checklist)

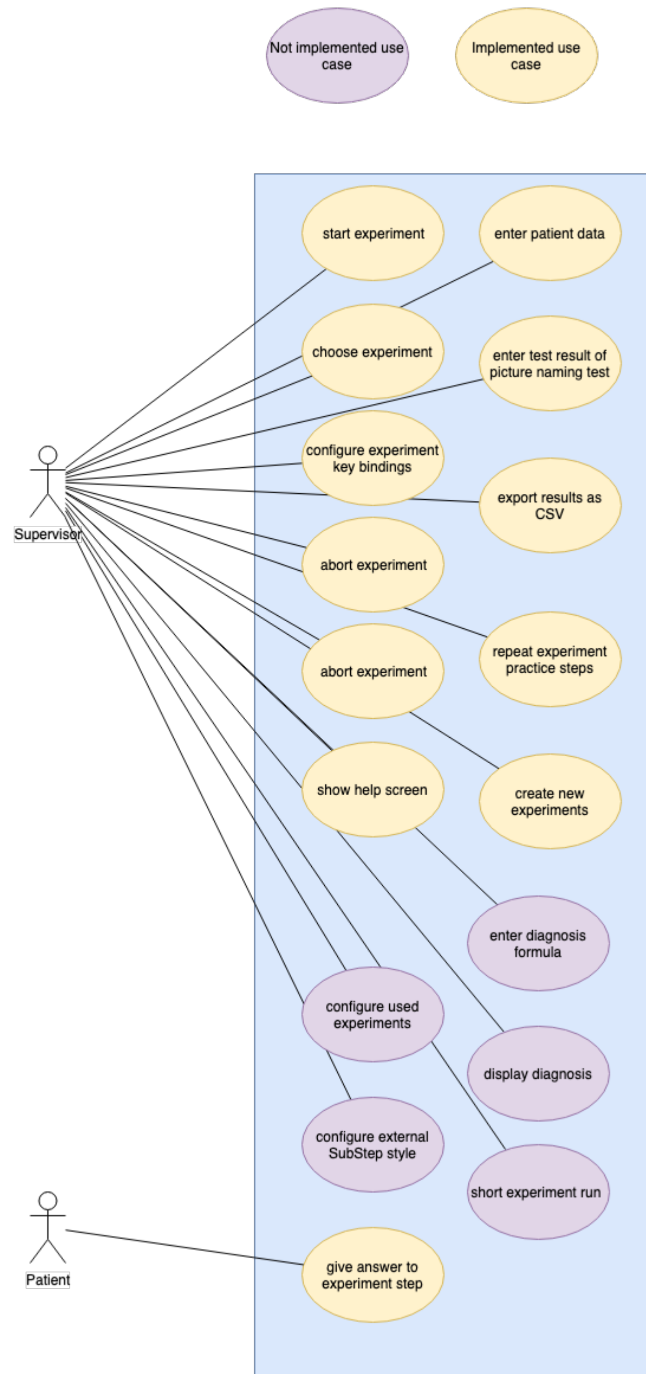
- Feature described in story is fully implemented without affecting other app parts negatively
- Travis build completed with no errors (green), this means:
    - Build completed without errors
    - Flow / Lint checks without errors
    - Unit / E2E Test without errors
- Pull request for the feature is reviewed and approved by Jan Frei / Cedric Wehli and merged on master.

5 Engineering Documents

5.1 Requirements

5.1.1 Use Cases

The documented use cases are based on an initial set of requirements. Not all of them were implemented during the development. The missing ones are highlighted with another background color.



5.1.2 Non-Functional Requirements

5.1.2.1 General Considerations

The NFRs are grouped by using the FURPS methodology. Each NFR is specified by its "contstraint key" Cx.x and might be referenced by it through the document. Optional requirements or requirements with low priority are marked with COx.x.

The NFRs are represented in the "Who, What, Why" format. The requirements are only written for one actor, but often hold true for other perspectives as well. Each fulfilled requirement serving the user (IT administrator, supervisor, patient) does serve the publisher, too.

NFRs that are out of scope are mentioned at the end of this chapter.

5.1.2.2 Functionality

C1.1: As a customer, I need the program to be able to run offline because not all PCs are connected to the internet.

T1.1: Use a computer that is not connected to the internet, execute e2e tests.

C1.2: As a supervisor, I want the exercises to behave exactly as/as similar as possible to the 'Testinstrument zur Fruehdiagnostik von Alzheimererkrankungen' already implemented in e-prime because the test instrument is the result of scientific studies and the results are only valid for the exact test setup.

T1.2: Methods to test and verify this requirement and the verification itself is to be done by Felix Platter, Memory Clinic/Sabine Krumm.

5.1.2.3 Usability

C2.1: As a supervisor, I want to be able to perform the intended action without unnecessary extra steps, so I can focus on the test and the patient instead of navigating through the different screens.

T2.1: Field Tests: Question supervisors and patient if they were happy.

5.1.2.4 Reliability

…

5.1.2.5 Performance

C4.1: As a user (and supervisor), I want to have fast response times for input and output for the test result of the patient not to be affected by any delay.

T4.1: Field Tests: Question about any observed delay.

C4.2: As an IT administrator, I want to have moderate hardware requirements because my institution cannot afford new hardware.

T4.2: Execute program on system with minimum Windows 7 hardware requirements (VM).

5.1.2.6 Supportability

5.1.2.6.1 Testability

C5.1.1: As a developer, I want the program to be testable, so when I deliver the program to my customer I can be sure that it works as expected.

T5.1.1: There are unit, component and e2e tests!

5.1.2.6.2 Extendability

C5.2.1: As the publisher of the software, I want to be able to add translations of the program, so that the tool can be used around the world without the language restricting it.

T5.2.1: All user facing texts that are part of the program are referenced by a language key and not *hardcoded* into the source code.

5.1.2.6.3 Installability

C5.3.1: As a user, I want the program to be installable (have an installation routine), so I don't have to perform manual steps that might need additional knowledge.

T5.3.1: Install the program on the operating systems described in C5.3.2.

C5.3.2: As an IT administrator, I want the program to run on Windows >= 7, MacOS > 10, Linux (Ubuntu > 12.04), so I don't have to worry about my existing operating system.

T5.3.2: Automated Travis build pipeline builds and end-to-end tests on all required operating systems. The travis setup will be included in the published repository (as code) .

C5.3.3: As a developer, I want to have a development environment setup instruction, so I can start working on the project without the need for personal introduction.

T5.3.3: A README.md that is part of the repository and has a step-by-step instructional text.

5.1.2.6.4 Serviceable

C5.4: As the publisher, I don't want to be restricted by any licenses of libraries. All used libraries must be permissive. This means no libraries with copylefted licenses can be included and all libraries must allow linking from code with a different license.

T5.4: The tool "license-checker" allows analyzing and comparing licenses of in the project used libraries. See License report

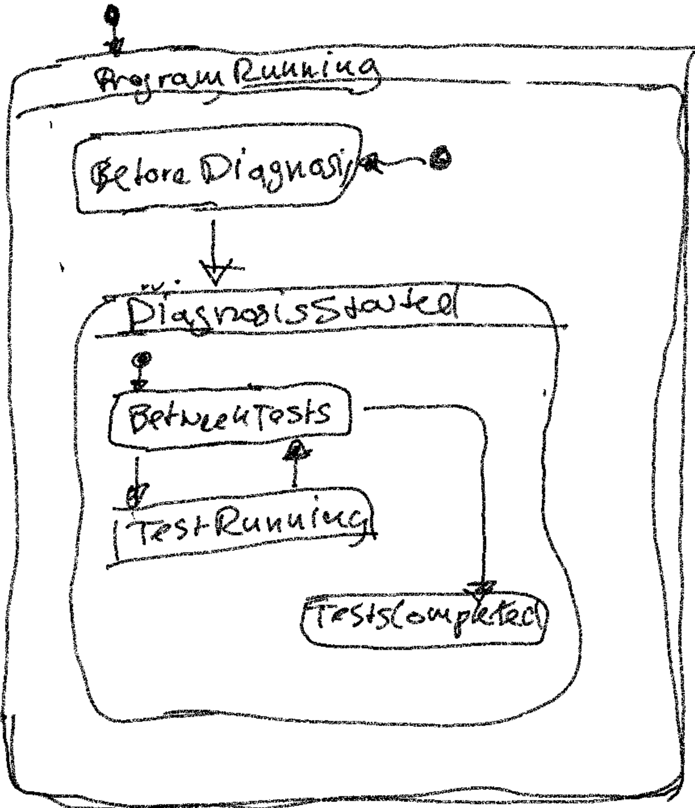
5.1.2.7 Out of scope NFRs

- Installation manual for the software product:
  The software is provided with an installation routine. Details about this are dependent on the used operating system.

5.1.3 State Diagram and Screen Flow

The following state diagrams/screen flows were used as a basis for later development but are not implemented 1:1.

5.1.3.1 State Diagram



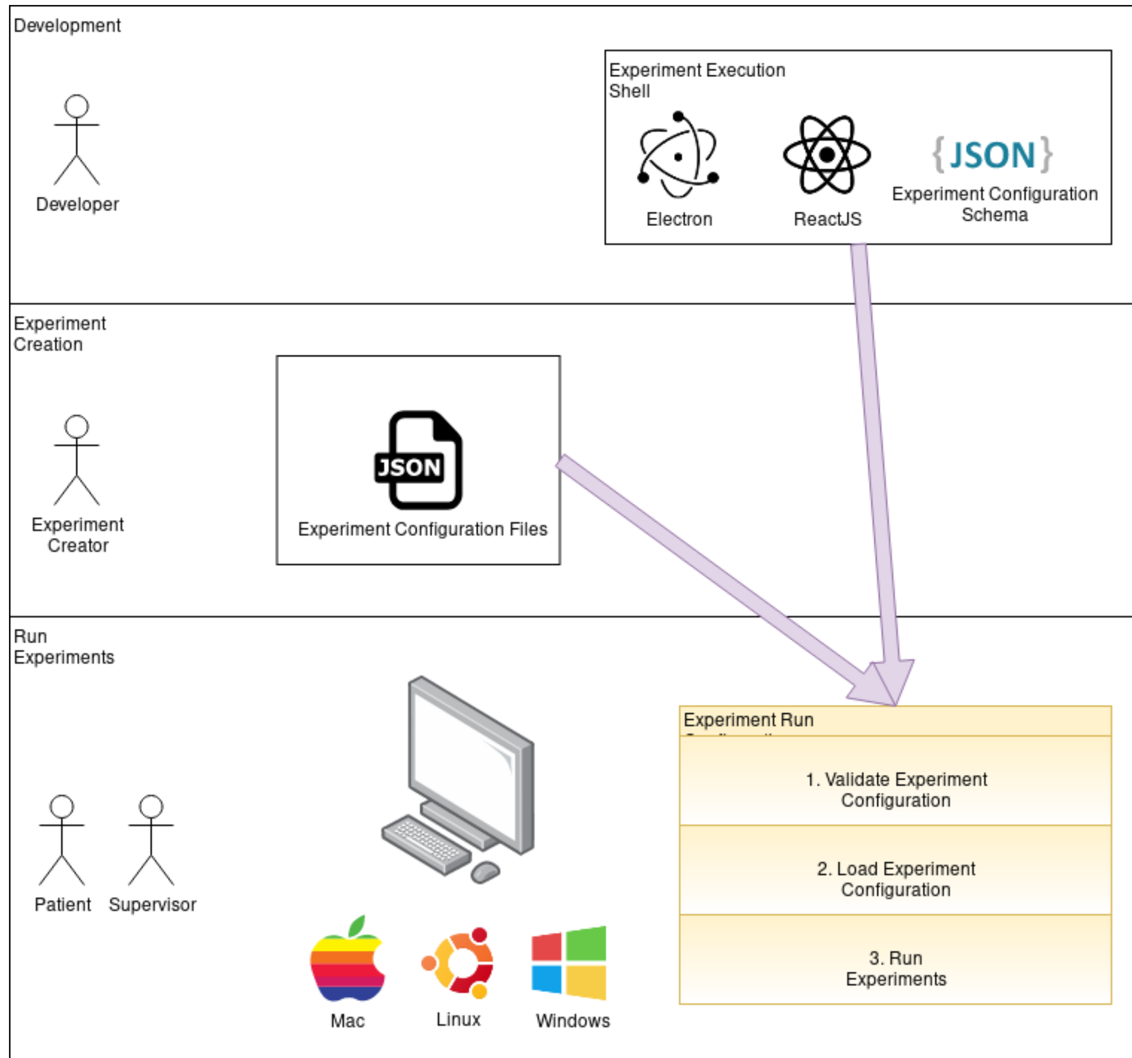
5.1.3.2 Screen Flow

- Solid lines/arrows indicate a link that is visible on the page
- The ‘Back’ transition indicates a back button on the page
- The ‘Escape’ transition from Goodbye to Completion overrides the invocation of the Escape page
- Depending on the previous page in the escape screen a ‘Back’ button is rendered
- Invoking the ‘Escape’ transition from the ‘Test’ state cancels the test

- 'Cancel' from the escape page when diagnosis started aborts the actual diagnosis and transitions to 'Before' diagnosis

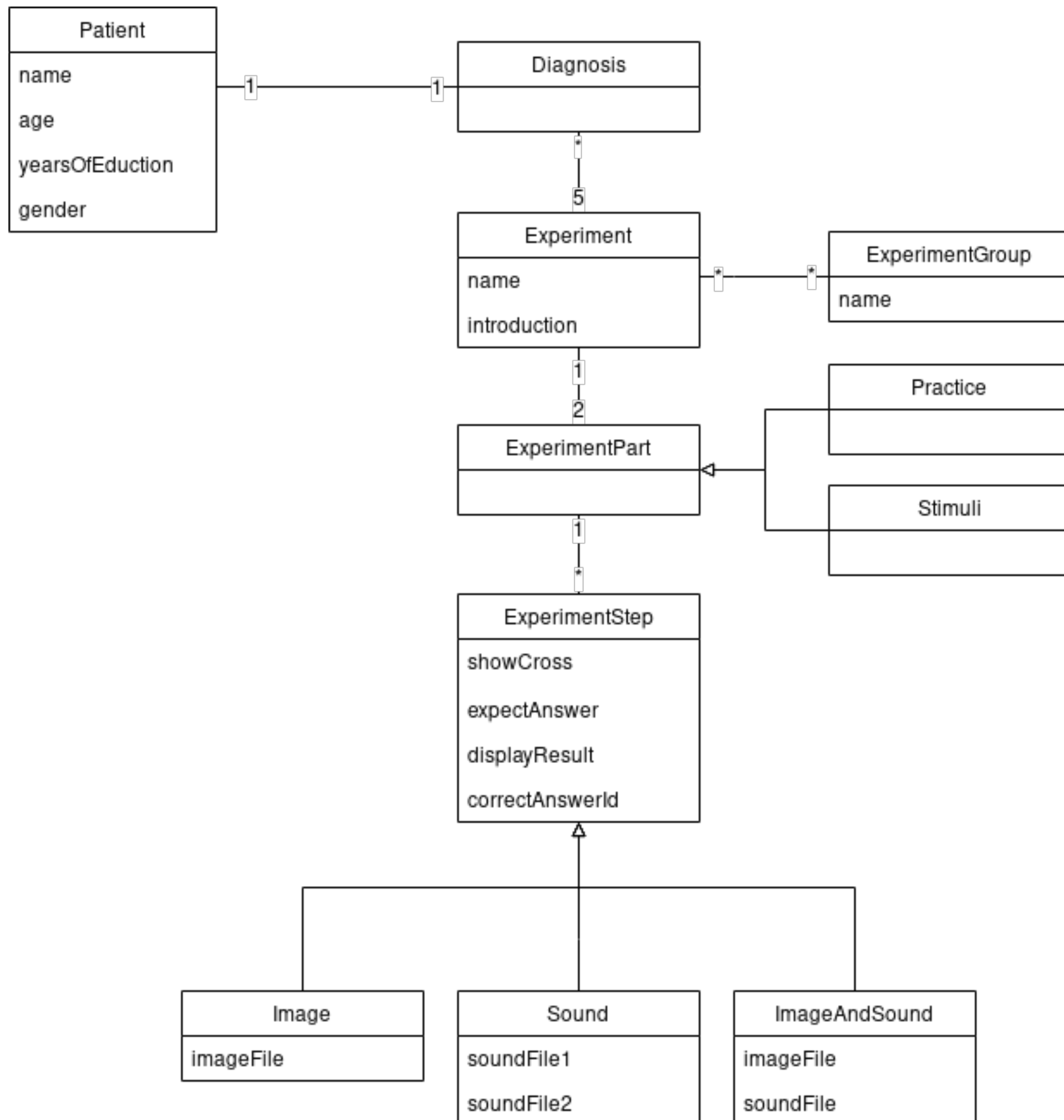
5.2 Architecture & Design

5.2.1 System Context Diagram



5.2.2 Domain Model

This diagram allows a brief overview over the problem domain. It was created on the basis of the specific requirements of the FPS.

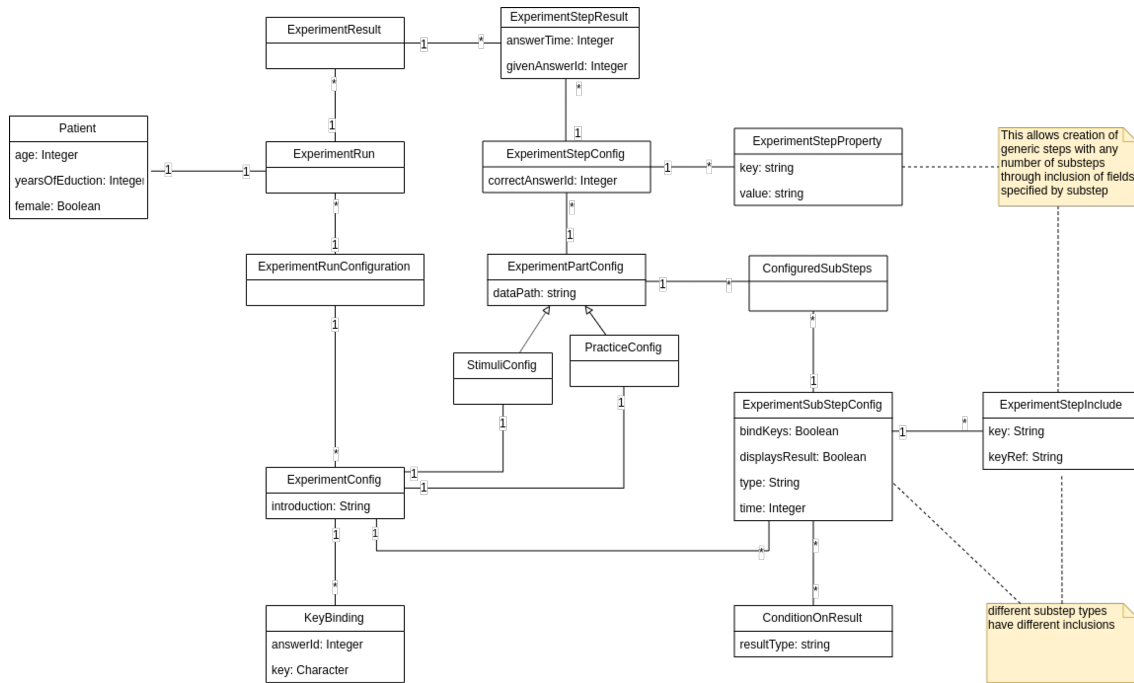


5.2.3 Class Diagram/Data Schema

The class diagram shows two parts of the program.

One part corresponds with the JSON configuration files that are loaded into the program on start. The relationships are established over properties that specify file names and references based on string keys when reading the files and creating Javascript objects*.

The other part describes objects created during run time. These are the results and the patient data.

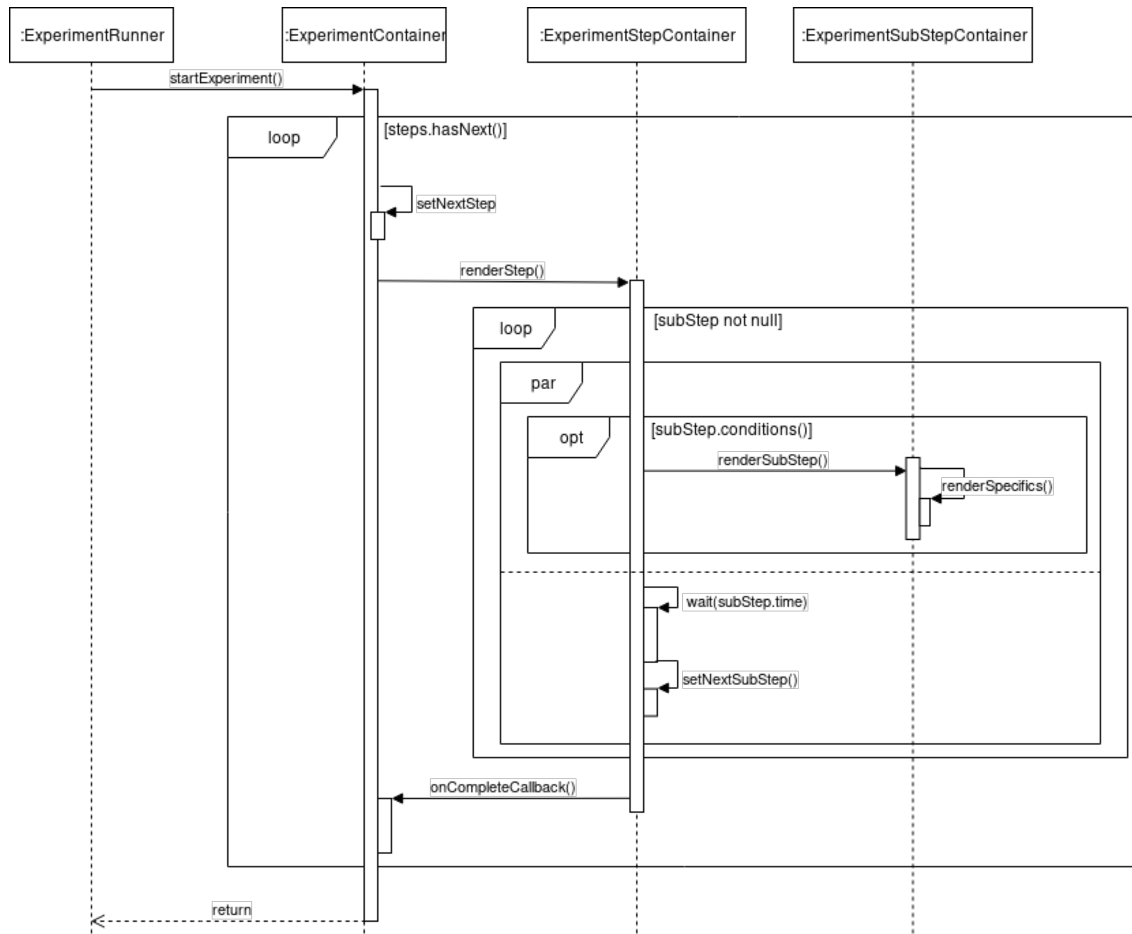


(* See the JSON schema for a complete specification)

5.2.4 Sequence Diagram

The following sequence diagram explains the logic used for iterating through a single experiment. In contrast to the class diagram where the ‘config objects’ represent the state, the here modeled entities represent the React components and its control flow.

For improved clarity some abstractions/simplifications were made.*



(* Creation and disposal of components are managed by React and are, thus, omitted. There is no separation of container and page components, some container components are completely left out (responsible for key bindings, result dispatch). Furthermore, in reality React is responsible for rendering and re-rendering the components on state change, but in this diagram components are explicitly invoked (renderStep(), renderSubStep()). External signals are not considered either.)

5.2.5 Technology decisions

5.2.5.1 Electron

5.2.5.1.1 Advantages

- Open source
- Cross-platform
- NodeJs
- Uses web standards we are already familiar with (JS, HTML, CSS)
- Good documentations, big community
- Provides functionalities such as installer creator, system specific features, etc out of the box without us needing to worry about it
- Chromium engine (for rendering) => same dev tools as in Chromium / Chrome

5.2.5.1.2 Disadvantages

- Code is open source: since it is JS, there is no compiling / etc

    - We solve the problem of this disadvantage by splitting our app into two parts: Experiment Execution Shell as an open source application and Experiment Configuration Files which contains the sensitive data.

- Slower than other technologies since it runs a web server and browser
    - This disadvantage is neglectable since our software doesn't do any resource intensive operations.

5.2.5.2 Boilerplate

We decided to use the following Electron React Boilerplate: https://electron-react-boilerplate.js.org/ GitHub: https://github.com/electron-react-boilerplate/electron-react-boilerplate

It is recommended by the Electron foundation on https://electronjs.org/docs/tutorial/boilerplates-and-clis#electron-react-boilerplate

5.2.5.3 Libraries

Most of the libraries we used were already included in the boilerplate mentioned above. It had ReactJS and Electron included and also other libraries, i.e.:

- Building / Packaging: Yarn, Babel, CSS-Loader, Electron-Builder, Webpack
- Code style checker / Types: Flow, ESLint, Prettier, Stylelint
- Testing: Jest, TestCafe

We added four libraries to the boilerplate.

5.2.5.3.1 Mousetrap

Homepage: https://github.com/ccampbell/mousetrap

Easily allows binding and unbinding of key presses. We needed this library so patients can participate in the interactive experiments and give answers.

5.2.5.3.2 I18next

Homepage: https://www.i18next.com/

We used I18next for the internationalization of our software. I18next is one of the biggest internationalization-frameworks for JavaScript. It also provides a react library with a react-hook (useTranslation): https://github.com/i18next/react-i18next

An alternative solution would have been FormatJS (https://formatjs.io/) and the React library React Intl (https://github.com/formatjs/react-intl). We decided against it since I18next was much easier to use for our software. I.e. with FormatJS we would need to write the translation loading strategy ourselves and there is no language detection either. I18next does both for you.

5.2.5.3.3 Jsonschema

Homepage: https://github.com/tdegrunt/jsonschema

We split our software into two parts: the Experiment Execution Shell and the Experiment Configuration Files which are JSON files. For those files, we added Jsonschema to validate them which prevents undefined behavior by malformed JSON config files.

5.2.5.3.4 Await-timeout

Homepage: https://github.com/vitalets/await-timeout

Await-timeout is a simple library used for waiting time with promises. It basically wraps “setTimeout” and “clearTimeout” so you can use them as promises. We used this library for the required waiting times in experiments and to make the code more readable (compared to directly use setTimeout / clearTimeout).

5.2.6 Development Infrastructure decisions

5.2.6.1 Serverless infrastructure

We decided to use Cloud solutions wherever it is possible. The main advantage compared to self-hosting is that we don't need to spend time on installing / updating / maintaining servers. Setting up your own virtual servers is very time consuming, especially if you need to worry about updating / securing / maintaining the whole stack (OS, application). The disadvantage of Cloud Solutions is often the price. However, the Atlassian stack and GitHub are not that expensive (GitHub is free of charge for students). The costs are covered by the developers.

5.2.6.2 Infrastructure decisions

5.2.6.2.1 Source code management

The three main players for Cloud source code management are Bitbucket, GitHub and GitLab.

	Gitlab	**Github**	**Bitbucket**
Best integration with Jira (Atlassian stack)	-	-	+
CI/CD Integration	Gitlab runner	Travis (free for educational use)	Pipelines
SCM freely available	+	+	-
CICD freely available	- (Minor cost for self-hosted gitlab runners)	+ (Travis for educational use)	- (10$ / month for 500 build minutes on Pipelines, additional costs for code analysis)
Directly integrated boilerplate	-	+	-

We decided to use GitHub since our Electron template app came out of the box with setup CICD (Travis config file and automatic release deploy) and we both have some experience with GitHub (i.e. Jan Frei worked on his EPJ with GitHub). The only downside of GitHub is the worse Jira integration compared to Bitbucket, but the GitHub integration provides enough functionality for us.

5.2.6.2.2 Project Management / Documentation

Since we are familiar with Jira and Confluence from our business projects, we decided to use them for our SA as well. This saves us a lot of time we would otherwise need to spend on getting to grips with other tools. Other advantages are:

- Great integration between Confluence / Jira.
- Collaborative teamwork on Confluence in a well-structured space.
- Runs in a browser (no additional app installations required).
- Jira fully supports Scrum projects compared to other players, see here.

5.2.7 Virtualization / Deploying in a VM for extended lifetime

The versions of Node, Electron and other libraries are fixed to the version we used when we deploy the final version of our app. This causes two problems:

5.2.7.1 Security aspect

In case of security fixes in any of our used libraries, they won't be applied to our app automatically and thus leave vulnerabilities in our app. This is not a problem for us since we develop an offline app which doesn't communicate to any other service. All libraries / frameworks are deployed with our app and not used by any other tool on the respective machine.

5.2.7.2 Incompatibilities

Our app uses libraries (Node, Electron) which are only compatible with specific versions of OS. Currently, the supported Windows versions are 7, 8(.1) and 10.

However, our app might be incompatible to future releases of operating systems and would require updating / migrating its libraries. In this case, the customer would need to use an old version of his operating system. In our estimation, the usage period of our app on a supported / maintained OS version is at least 10 years.

5.2.7.3 Virtualization

Deploying our app in a virtual machine would seem to improve the incompatibility issue. We would deploy a VM image with our app installed which will always run (since the OS never changes). However, this causes three problems:

- Size: Even the smallest Linux image would be close to 1 GB
- Performance: Running it in a VM results in a performance loss
- Limited lifetime: Virtual images are bound to a Virtualization software and to a version range of the respective software. This means that our virtual image has the same lifetime as the highest version of the Virtualization software supporting our image. This version might be incompatible to future releases of operating systems. Virtualization leads to the same issue as non-virtualization while the only advantage is that only a new VM image instead of a new app version needs to be created.

5.2.7.4 Conclusion

We decided against deploying our app in a virtual machine due to the disadvantages mentioned above.

5.3 Results and Outlook

5.3.1 Results

At end of this work a functional software was delivered to FPS. Some use cases didn’t find their way into the current release. The main missing use case in that regard is the ability to enter a diagnosis formula and display the diagnosis for a patient.

This is partly due to the change of legal situation during the development. While at the beginning the free distribution of the software at other clinics was planned, the legal department of FPS Basel required us to remove any of their intellectual property from the software. Implementing the two use cases from above was not easily possible anymore.

The product was adapted to comply with the new requirements and now consists of two components. One component, the experiment execution shell, is a generic test runner that is published under the MIT license. It can be used, altered and sold without further legal complications. It is capable to run various tests that can be created in the form of JSON configuration files.

The second part, the specific tool for early diagnosis of Alzheimer’s disease, is delivered directly to FPS in the form of configuration files. With these they will be able to replace the existing proprietary software.

5.3.2 Outlook

In the near future, FPS and the team of the Memory Clinic want to distribute the software to other clinics and experts.

To implement the diagnosis related use cases, a different approach would be necessary. This could be to load native modules that protect the source code.

Some parts of the experiment execution shell are not fully generic and might be a little bit too specific for broad use. Further development could be:

- styling and layout of tests
- additional types of tests
- customizable workflow
- different export options

5.4 License report

We will publish the Experiment Execution Shell under the MIT license to allow FPS and other institutes to use it commercially. Furthermore, all of our used libraries need to be permissive and not copylefted. License check of 18 Dec 2019

```
$ license-checker --production --summary
├─ MIT: 70
├─ BSD-3-Clause: 6
├─ ISC: 3
├─ (CC-BY-4.0 AND OFL-1.1 AND MIT): 1
├─ Apache-2.0: 1
├─ BSD-2-Clause: 1
├─ MIT*: 1
├─ Apache-2.0 WITH LLVM-exception: 1
└─ (MIT AND Zlib): 1
```

All of the licenses in our dependencies are permissive, see https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses.