
Software Testing 2020

*Assignment PROJECT: Heart Monitor
Test Plan*

Group: 11 - Korean Air

Members: Markus Funke (2644722)
Wouter Kok (2639768)
Sven Preng (2614131)
Pjotr Scholtze (2612369)

Master program: Computer Science - SEG

June 19, 2020

Contents

1	Introduction	3
1.1	Purpose	3
1.2	Project Overview	3
1.3	Audience	3
1.4	Scope	4
1.4.1	In Scope	4
1.4.2	Out of Scope	5
1.5	Schedule	5
2	Development Environment	6
2.1	Trunk-based Development Workflow	6
2.2	Agile process with GitHub	6
2.2.1	Projects	6
2.2.2	Issues	7
2.2.3	Actions	8
2.3	Code Style-guide	8
2.4	Test-Driven Development	8
3	Test Strategy	9
3.1	Code Review	9
3.2	Static Code Analysis - linting	10
3.3	Unit Testing	11
3.3.1	PyTest	11
3.3.2	PyTest Coverage	12
3.4	Mutation Testing	14
4	Data Flow Diagram	16
5	Integration Test	18
6	User Acceptance Test	20
7	Test Environment	22
7.1	Linux	22
7.2	MacOS	22
7.3	Windows	22
7.4	Windows	22
8	Appendix	23
8.1	Code Review Assignment	23
8.2	Test Coverage Data	24
8.3	Data Processing Module - BVA and EP	25

Change Log

Version	Date	Change
1	2020-04-22	First draft
2	2020-04-23	First draft - Review
3	2020-04-26	Final version
4	2020-04-26	Working version - Review
5	2020-04-30	End version (includes final product version)

Reference Documents

Version	Date	Document
1.0	2020-04-22	Assignment PROJECT: Heart Monitor — Software Requirements Specification (SRS)

1 Introduction

1.1 Purpose

This document explains a test plan which describes the testing approach and the overall testing mechanism that will drive the testing of the Heart Beat Monitor Simulation Software **HB-Sim2020**. The document consists of:

- Development Environment: explains the development environment how the system is developed and produced. It also describes the project management and its tools.
- Test Strategy: rules the test will be based on, including the project details (start / end dates, objectives, assumptions); description of the process to set up a valid test (test cases, specific tasks to perform, scheduling, data strategy).
- Execution Strategy: describes how the test will be performed and process to identify and report errors, and to fix and implement fixes.
- Test Management: process to handle the logistics of the test and all the events that come up during execution (communication, project management procedures, risk and mitigation)

1.2 Project Overview

The **HB-Sim2020** is to be developed as part of the PROJ assignment of the software testing course. It's main purpose is to perform as a piece of software to be tested after development, as to discuss the possible bugs for educational purposes. The software will run a simulation of a heartbeat monitor as used in a medical facility. The heartbeat monitor should be able to notify the medical team of any anomalies in the oxygen levels, heart beat and blood pressure as well as give a constant update on these current values via a command line interface. As this software should simulate a hear beat monitor, it is not connected to any actual sensors and thus will receive it's input through a test file.

1.3 Audience

- Project team members: The project team consists of four Master computer science students. Each member holds multiple roles.
- Developer: The developers consists of all four project team members.
- Code-based test-team: Code-based testing is conducted by the developer team.
- Specification-based test-team: Specification-based tests (i.e. black-box testing) are conducted by other students from another project team.
- others: e.g. teaching member and teaching assistants.

1.4 Scope

1.4.1 In Scope

All the features of **HB-Sim2020** which were defined in the Software Requirements Specification (SRS) are need to be tested.

Data input	CLI	CSV location can be specified by command-line argument at program start.
		CLI provides a <i>help</i> menu.
	CSV	CSV file simulates the sensor data.
		The CSV file contains all necessary sensors (blood pressure, heart rate, oxygen) and their possible circumstances (no data, corrupted data, etc.)
Data processing	validation	Input values will be validated against specific rules (i.e. numeric values, strings, etc.)
	error handling	Corrupted incoming values will be handled without terminating the program.
	blood pressure	Processing and evaluation.
	oxygen	Processing and evaluation.
	heart rate	Processing and evaluation.
	statistic	A statistic keeps track of all processed values.
Data output	CLI	All messages will be printed to the CLI.
		The messages will be formatted in a proper manner such that the messages can be read easily.
	status messages	Status messages contain the <i>status</i> , <i>value</i> , <i>unit</i> of oxygen, heart rate, and blood pressure.
	statistic	A statistic of all processed values will be displayed if the program ends or terminates.

Table 3: Function Groups

1.4.2 Out of Scope

These features are not be tested because they are out of the project scope and not included in the SRS.

- Hardware interfaces
- Software security
- Software performance
- Privacy

1.5 Schedule

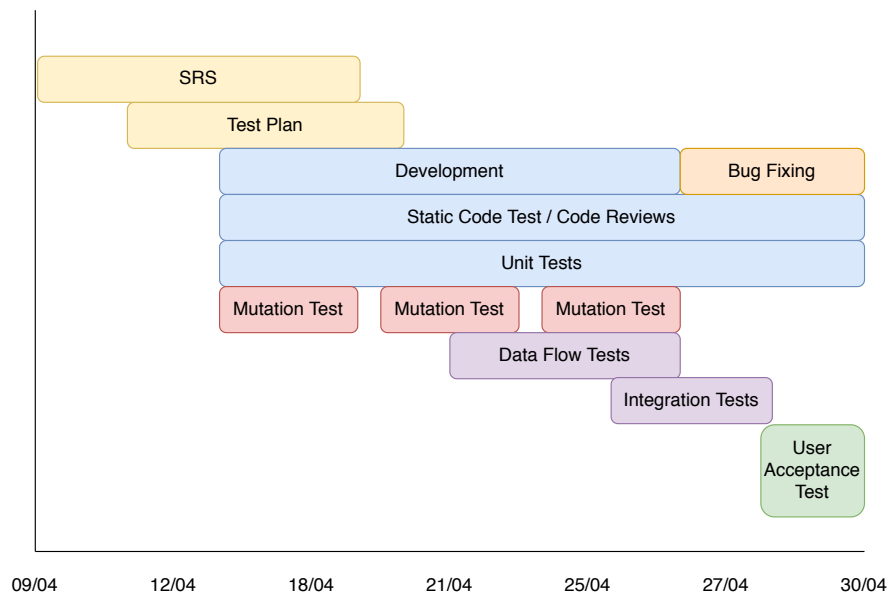


Figure 1: Schedule (approximately)

2 Development Environment

This section gives a brief overview and insights about the development environment. Its purpose is to examine the circumstances and the environment under which the system (i.e. the programming code) is developed to ensure a flawless process pipeline and to provide a background for the actual test environment.

2.1 Trunk-based Development Workflow

The development process is based on a mix between a Trunk-based and a Git(Hub) workflow. The combination of both creates a minimum of administration overhead and achieves a maximum of code quality.

The workflow consists of the **master** branch, which is open for every developer to push. But every feature, as well as every bugfix, is created in a separate **feature-branch**. After finishing the development of a specific branch, the branch is converted to a pull-request (PR). Another developer must review each PR. If and only if the reviewer marks the PR as approved, the PR is allowed to get merged into the master branch.

Through this code review strategy, i) a high **code quality** and ii) early **detection of bugs** is ensured, iii) **code-smells** are identified in an early stage, and iv) **knowledge sharing** among the developers is encouraged.

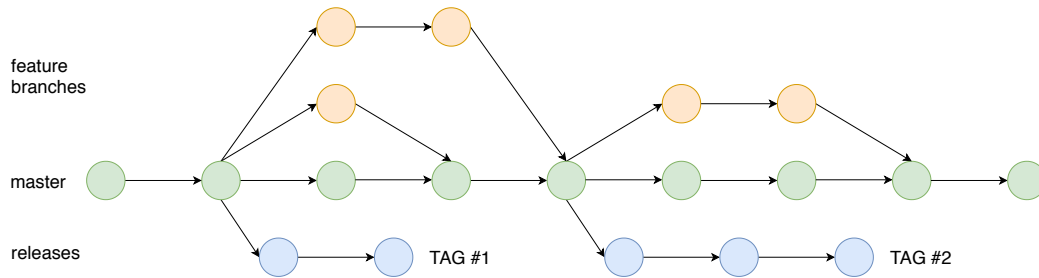


Figure 2: Trunk-based workflow

2.2 Agile process with GitHub

The project is structured in an iterative approach, focusing on continuous releases to gather feedback from every iteration. Since this project has no actual customer, the feedback comes from code reviews and continuous test iterations. One iteration phase contains 1) design, 2) development, 3) test, and 4) evaluation phase. To achieve agile project management, GitHub is used as distributed version control, source code management, and project management tool. The following subsections describe the various GitHub functions used.

2.2.1 Projects

As a central project management tool to organize and prioritize the different tasks, *GitHub Projects* is used with an automated Kanban board configuration. Every issue and task is listed. The board represents the used workflow and gives an overview of all tasks, their status, and the assigned person. Figure 3 shows an example of the configured Kanban board.

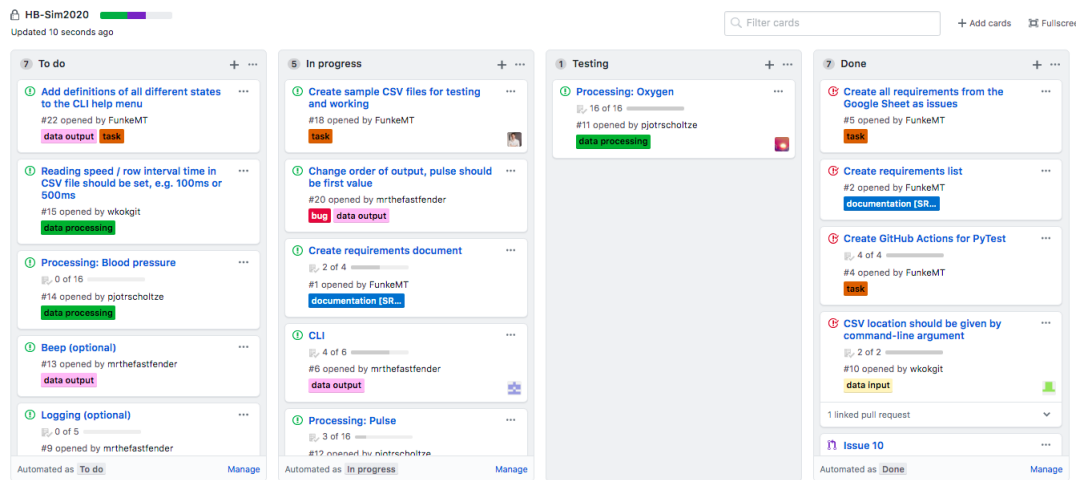


Figure 3: GitHub Projects

2.2.2 Issues

GitHub Issues is used as a centralized bug-tracker and task management tool. Through different labels, management beyond source code issues and tasks are likely. GitHub allows issue referencing to commits. This allows tracking the feature and its assigned developer. Together with pull request reviewers, one feature/issue can be traced through its whole lifetime—from issue creation, development, review, testing, and integration. Figure 4 shows an example issue list.

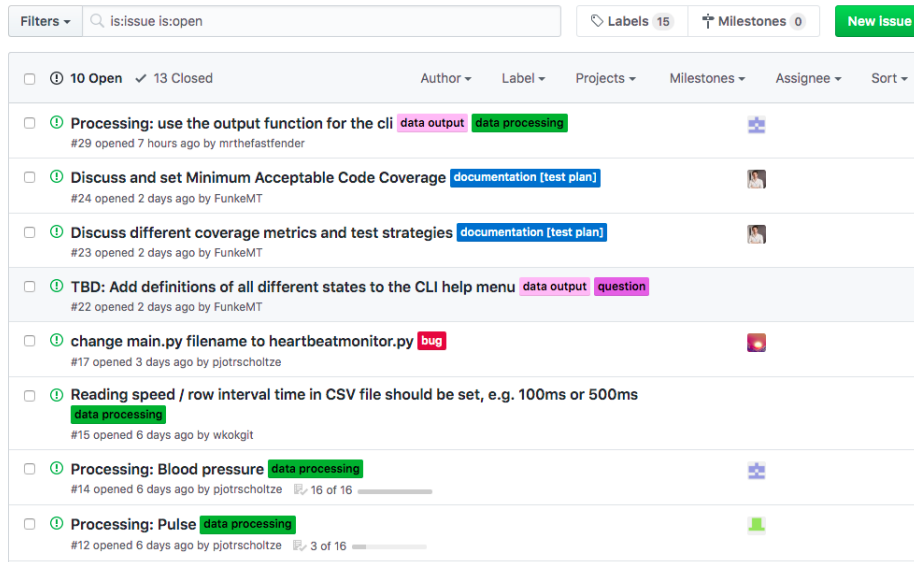


Figure 4: GitHub Issues

2.2.3 Actions

To automate several tasks, a *GitHub Action* pipeline is installed. Along with every commit or merge into the master branch, the pipeline is triggered. The pipeline executes, among other things, the following tasks:

1. Linter - run static code analysis
2. PyTest - run unit tests
3. PyTest Coverage - generate coverage report

Figure 5 shows an extract of a successful build and its different actions. See section 3 for more details about static code analysis and unit tests.

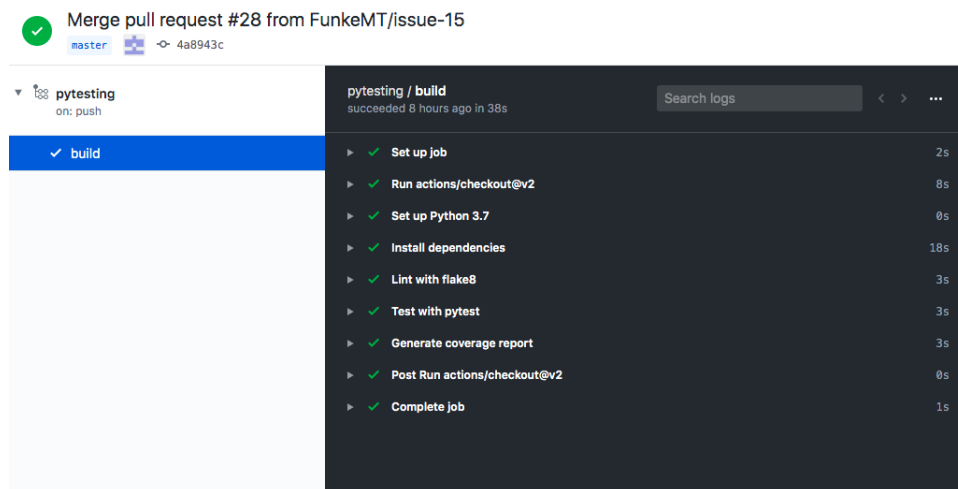


Figure 5: GitHub Actions

2.3 Code Style-guide

The code development adhere to the "PEP 8 – Style Guide for Python Code"¹ and is fulfilled by using the Python code formatter *Black*² (version 19.10b0).

2.4 Test-Driven Development

Together with agile project management (see section 2.2) we tried to approach the development in a test-driven method (TDD). The developers are obliged to write test functions consequently before the actual component is implemented. This ensures that only proven code is added which meets the specified requirements in the SRS. For more information about unit tests see section 3.3.

¹<https://www.python.org/dev/peps/pep-0008/>

²<https://github.com/psf/black>

3 Test Strategy

3.1 Code Review

Purpose

The usage of a random assignment to review code segments removes personal preferences between the developers, removes human bias, and helps to focus on technical facts.

Scope

Code reviews should be applied on all system modules.

Method

To achieve a random reviewer assignment to each developer, a small R-script is used (the R-script and its result after execution can be found in Appendix 8.1). The final assignment after the script execution is showed in table 4.

Testers

Developer	Gets Review From
Sven Preng	Wouter Kok
Pjotr Scholtze	Markus Funke
Wouter Kok	Pjotr Scholtze
Markus Funke	Sven Preng

Table 4: Review Assignment

Test Acceptance

The test strategy is a continuous strategy which will happen during the whole development phase. As in section 2.1 explained, each pull request has to pass a review strategy and is conducted by another developer. Every review is following these principles, which are based on³.

- Is the code well structured?
- Is the functionality as in the SRS specified?
- Is the code as less complex as possible?
- Are the unit tests appropriate and designed well?
- Is the naming (methods, variables, etc.) well chosen?
- Do the comments explain *why* instead of *what* and are they clear formulated?
- Does the code follows the style guide?

³<https://google.github.io/eng-practices/review/reviewer/>

3.2 Static Code Analysis - linting

Purpose

As static code analysis the continuous development (CD) process should contain a static code analysis tool which is executed automatically and at code-compile time. The execution should be performed *before* the unit tests.

Scope

Static code analysis should be applied on all system modules.

Method

As in section 2.2.3 already mentioned, the linting execution is performed by a GitHub Action through the Python library *flake8*⁴ (version 3.7.9).

Testers

All developers should keep an eye on the automatic reports generated by flake8 and integrate the results.

Test Acceptance

The test strategy is a continuous strategy which will happen during the whole development phase.

⁴<https://flake8.pycqa.org/en/latest/>

3.3 Unit Testing

To follow the TDD approach as described in section 2.4, unit tests should be written and executed as in the workflow diagram (Figure 6) described.

While we initially tried to follow the principles from TDD, we ended up using them sparsely. For the most part, the code was written before the tests were written.



Figure 6: Unit Test Workflow

3.3.1 PyTest

Purpose

Unit testing is carried out throughout the entire implementation phase and starts from day one. Since TDD is chosen as a development process (see 2.4), test cases should be written before the actual unit implementation. Omitting test cases leads to higher technical dept. Therefore test cases should be available for as many units as possible and as early as possible to save time in the actual integration phase and uncover bugs in an early stage.

Scope

Unit testing should be applied to all system modules. Since the **data processing module** is the most important module (priority *high*) it should get the most attention.

Method

To write and execute unit tests, the Python extension *PyTest*⁵ is used. The extension should be used on a local machine at development time. In addition, a GitHub Action (see section 5) is executed to trigger PyTest automatically. To achieve an efficient test-case selection, such that all possible test instances are covered, **Equivalence Partitioning** (EP) and **Boundary Value Analysis** (BVA) are used.

As already mentioned, the data processing module is prioritized as *high*. Hence, only for this module a dedicated spreadsheet with test-cases is prescribed. This spreadsheet has to be used by the developer to write the test cases for this modules. More test cases are acceptable, while less test cases are not acceptable. The other modules follows EP and BVA as well, but the developer has to decide which test-cases should be written to achieve the test acceptance. For those modules, the written test-cases itself serve as documentation. The spreadsheet for the data processing module can be found in Appendix 8.3.

Testers

All developers.

⁵<https://docs.pytest.org/en/latest/>

Test Acceptance

see PyTest Coverage acceptance in 3.3.2.

3.3.2 PyTest Coverage

Purpose

The purpose of test coverage is to find untested code segments and functions. It helps also to identify unnecessary test cases and uncovers paths in the application and code which are never be executed and meaningless. Together with the SRS, coverage reports identifies gaps in the requirements. The developer should not attempt to artificially increase test coverage, but should use the reports as information and identify vulnerabilities.

Scope

Test coverage should be applied to all system modules. For modules with priority *medium* or *low* (e.g. UI), a lower test coverage is acceptable.

Method

As major code coverage method **Statement Coverage** is used. The aim is that all executable statements in the code are executed at least once.

The coverage report is created automatically by GitHub Actions (see section 2.2.3) and is based on the Python PyTest plugin *PyTest-Cov*⁶ (version 2.8.1). It is also possible to create the coverage report locally.

Testers

All developers.

Test Acceptance

Since the overarching system contains also of an user interface (printing to the CLI) which is prioritized in the SRS as *Medium* and for which it is difficult to create reliable test cases, the test coverage acceptance is set to **96%** (indicated by the blue dashed line in Figure 7). To keep track of the coverage, Figure 7 examines the growth of the code statements in relation to the coverage in percentage (the diagram data are in appendix 8.2).

⁶<https://pypi.org/project/pytest-cov/>

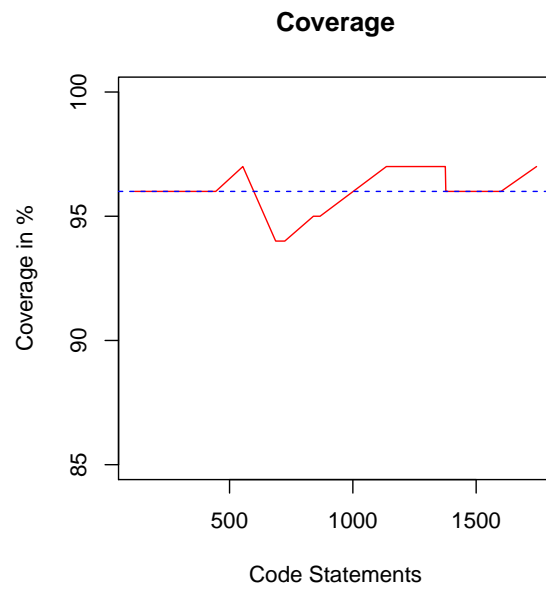


Figure 7: Statement Coverage Report, growth of code statement

3.4 Mutation Testing

Purpose

Data processing includes the analysis of the incoming measurements (via CSV) and the classification of those values. This classifications (also known as status - see SRS for more information) could be life threatening in the worst case. To ensure a faultless behaviour and detect weaknesses in the tests and code, mutation tests are executed and analysed.

Scope

Since the most important module is the **data processing module**, mutation tests should be executed only for this module.

Method

To automate mutation testing with Python, *mutmut*⁷ (version 2.0.0) is used. The workflow which should be stuck to for analyzing the processing module is explained in Figure 8.



Figure 8: Mutation Testing Workflow with *mutmut*

Testers

Developers who are involved in the data process implementation.

Test Acceptance

The Mutation Score (MS) is calculated as $MS = \frac{\#killedmutants}{total} \cdot 100[\%]$ and should be at least 70% or higher in the final release. The focus should be rather on the report analyzing process and find weaknesses than increasing the MS artificially. A protocol for the different milestones and test-phases are provided in table 5. Since the execution of the mutation tests itself and the analyzing process costs a lot of time, mutation tests should be not automated (i.e. GitHub Actions - see section 2.2.3).

⁷<https://mutmut.readthedocs.io/en/latest/>

Version	Date	Developer	MS [%]	<i>mutmut</i> Result
v0.5	2020-04-19	Markus Funke	61 %	<ul style="list-style-type: none"> • killed: 33 • timeout: 0 • suspicious: 0 • survived: 21 • skipped: 0
v0.6	2020-04-21	Markus Funke	64 %	<ul style="list-style-type: none"> • killed: 55 • timeout: 0 • suspicious: 0 • survived: 31 • skipped: 0
v0.9	2020-04-25	Pjotr Scholtze	100 %	<ul style="list-style-type: none"> • killed: 87 • timeout: 0 • suspicious: 0 • survived: 0 • skipped: 0

Table 5: Mutation Test Protocol

4 Data Flow Diagram

Purpose

The Data Flow visualizes how information flows throughout our system. Thus brings a better view of the system and makes it easier to see data changes and which data should be tested for validity.

Scope

The scope of the data flow is mostly within the *Data processing* module in which it processes user input data. After processing the data can be used to output.

Method

A Data Flow is created which is shown in Figure 9.

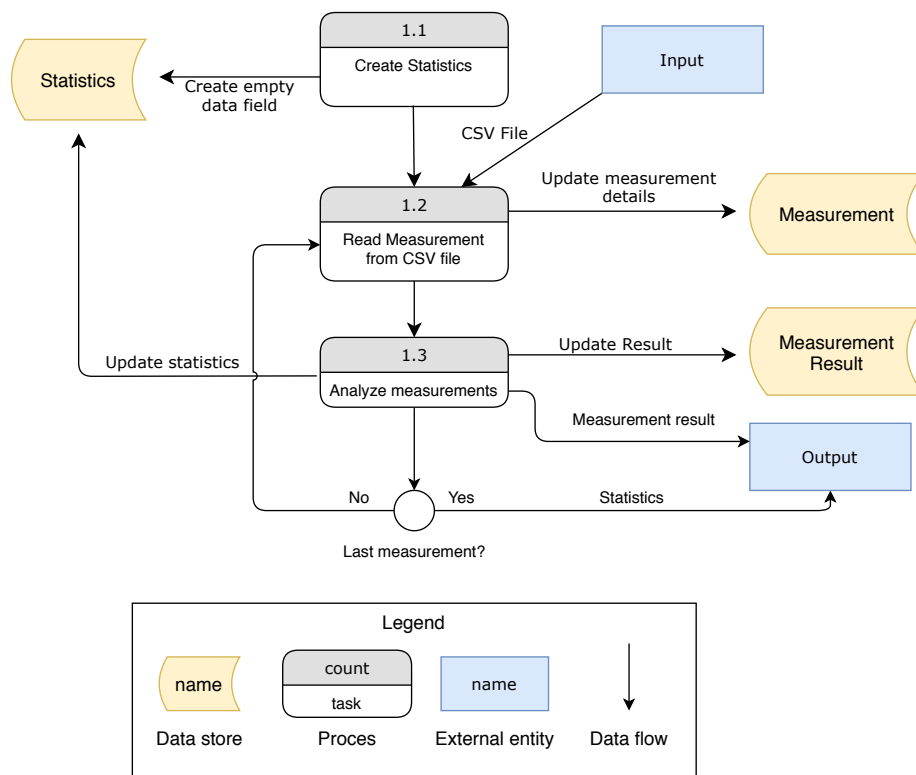


Figure 9: Data Flow Diagram

Testers

Developers who are involved in the data process implementation.

Test Acceptance

Only if every data flow path in Figure 9 is tested with all CSV files, the test may be accepted.

We used various CSV values with injected issues to test the program. We covered these issues:

- Missing headers
- Malformed headers
- Too many headers
- String input instead of numeric input
- Numeric input outside boundary
- Integer overflow input
- Missing values in a recording row
- With only headers and no values

5 Integration Test

Purpose

Integrate all different software modules into one logically group to test it as a single system to identify and uncover interaction problems between these modules.

Scope

All different modules as in the SRS and in 3 defined. Module i) data input, ii) data processing, and iii) data output will be grouped into the final system *HB-Sim2020*.

Method

To integrate all modules into the actual HB-Sim2020 system, a top-down approach should be used. In particular, first (1) combine module *data input* and *data processing*. In the second step (2) the module *data output* is integrated. This approach ensures that data input and data output works correctly together—independently from the data output layer.

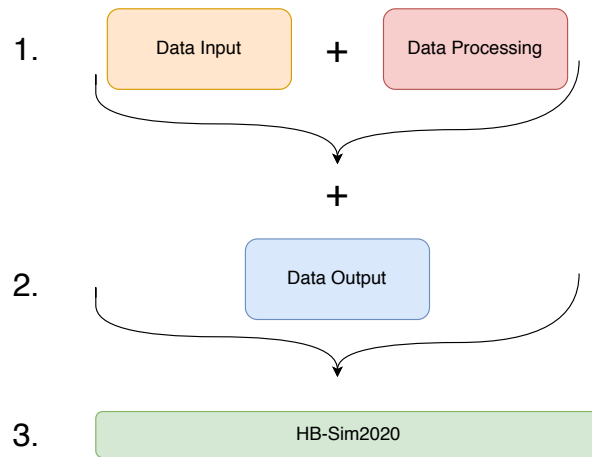


Figure 10: Module Integration

Testers

All developers.

Test Acceptance

Test should be performed according to the SRS. The following table gives an overview of the requirements which have to be **tested** and have to be **fulfilled** according to each integration phase.

Integration Phase	Modules	Requirements
1	Data Input & Data Processing	<ul style="list-style-type: none"> • SR-#4.2.1: Input recording location • SR-#4.2.2: Comma-separated values (CSV) file • SR-#4.2.3: Oxygen measurements • SR-#4.2.4: Pulse measurements • SR-#4.2.5: Blood pressure measurements
2	Data Output	<ul style="list-style-type: none"> • Requirements from phase #1 plus: • SR-#4.2.6: Data output - statistic • SR-#4.2.7: User interface • SR-#4.2.8: Logging
3	HB-Sim2020	<ul style="list-style-type: none"> • Requirements from phase #1 plus • Requirements from phase #2.
General	General	<p>For all test executions, all testing CSV files have to be used and executes before the test will be accepted and the next integration phase can be started. Repository paths to CSV testing files:</p> <ul style="list-style-type: none"> • heartmonitor/input/normal_100.csv • heartmonitor/input/normal_with_errors_100.csv • heartmonitor/input/random_100.csv • heartmonitor/input/simulation.csv

Table 6: Integration Test Strategy

6 User Acceptance Test

Purpose

After fully developing the software of this project, a user acceptance test is done as a final step towards rolling out the application. This test is done after various levels of testing are already completed and aimed at usefulness and usability instead of cohering to requirements. The goal is to see if the customer's needs are solved.

Scope

The scope is the whole final application.

Method

There are User Acceptance Testing Test Cases created to show the steps a tester has to do to fulfill the user's needs.

User Story	As a doctor I want to be able to easily start-up the system
Test Scenario Steps	<ol style="list-style-type: none">1. Open a command-line interface2. Make sure HB-Sim2020.exe and simulations.csv are in the same folder3. Go to the folder holding both files4. Run: python HB-Sim2020.exe -path simulation.csv

Table 7: UAT TC1: Start-up the system

User Story	As a doctor I want to be able to see my patients oxygen, pulse and blood pressure levels.
Test Scenario Steps	<ol style="list-style-type: none">1. Make sure the system is started2. Look at the incoming measurement results coming in every second in the CLI3. Execute actions upon those results

Table 8: UAT TC2: Use heartbeat monitor as intended

User Story	As a doctor I want to be able to see the statistics of what happened during the last run.
Test Scenario Steps	<ol style="list-style-type: none"> 1. Make sure the system is started 2. Exit the program by using CTRL-C 3. Look at the statistics given from the system's run

Table 9: UAT TC3: See statistics

Testers

For the reason that we're doing this project for the Software Testing course, we're unable to let real doctors, which would be the end-users/client, test our project. Therefore we have chosen that we ourselves do this test, keeping in mind that it is not the right way to do it.

Test Acceptance

The tests are executed and if there are any faults, they are being noted and fixed by the development team.

7 Test Environment

If no specific test method is specified in section Test Strategy 3 (e.g. GitHub Actions), the tests have to be done at a local development environment. The following different hard-/software combinations are available and admitted.

7.1 Linux

- OS Version: Ubuntu 20.04 LTS
- Terminal: XFCE 4 Terminal with ZSH

7.2 MacOS

- OS Version: MacOS 10.13.6
- Terminal: iTerm2

7.3 Windows

- OS Version: Windows 10.0.18363 Build 18363
- Terminal: CMD

7.4 Windows

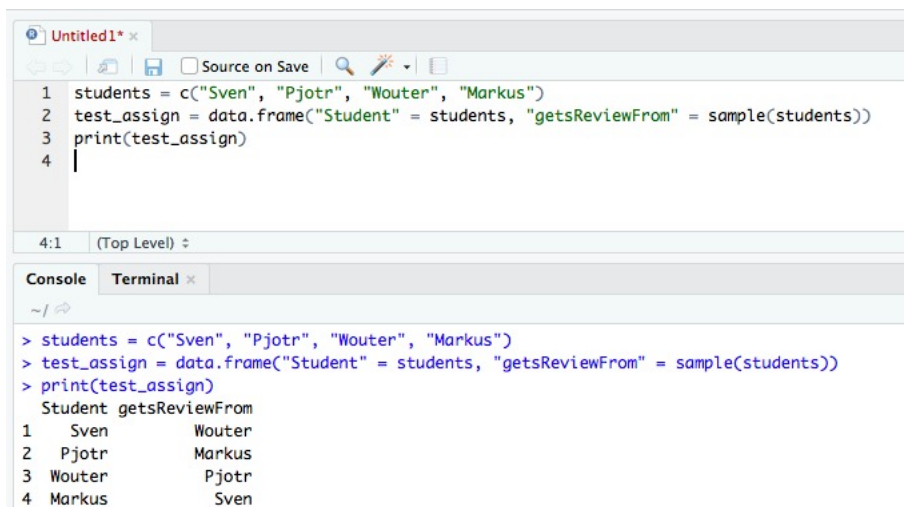
- OS Version: WSL subsystem version 4.4.0-18362-Microsoft x86_64
- Terminal: bash

8 Appendix

8.1 Code Review Assignment

```
students = c("Sven", "Pjotr", "Wouter", "Markus")
test_assign = data.frame(
  "Student" = students,
  "getsReviewFrom" = sample(students)
)
print(test_assign)
```

Listing 1: Random Mapping - R Code



The screenshot shows an R IDE window titled 'Untitled1*' with a toolbar at the top. The editor contains the following R code:

```
1 students = c("Sven", "Pjotr", "Wouter", "Markus")
2 test_assign = data.frame("Student" = students, "getsReviewFrom" = sample(students))
3 print(test_assign)
4 |
```

Below the editor is a console window showing the execution of the code:

```
> students = c("Sven", "Pjotr", "Wouter", "Markus")
> test_assign = data.frame("Student" = students, "getsReviewFrom" = sample(students))
> print(test_assign)
  Student getsReviewFrom
1   Sven         Wouter
2  Pjotr         Markus
3 Wouter         Pjotr
4 Markus          Sven
```

Figure 11: Random Mapping - Result

8.2 Test Coverage Data

```
statements , miss , coverage
115 , 6 , 96
227 , 9 , 96
251 , 9 , 96
322 , 12 , 96
324 , 13 , 96
444 , 13 , 96
554 , 16 , 97
687 , 44 , 94
690 , 44 , 94
716 , 44 , 94
724 , 44 , 94
840 , 43 , 95
867 , 43 , 95
1136 , 39 , 97
1153 , 40 , 97
1375 , 48 , 97
1377 , 50 , 96
1601 , 63 , 96
1745 , 57 , 97
```

Listing 2: Test coverage data

8.3 Data Processing Module - BVA and EP

Test overview for Equivalence Partitioning and Boundary Value Analysis testing techniques. Intervals are given as mathematical intervals (see ⁸).

SR-#4.2.3: Oxygen measurements				
OK				
invalid	invalid	invalid	valid	invalid
[0, 60]	[61, 89]	[90, 94]	[95, 100]	[101, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5
MINOR				
invalid	invalid	valid	invalid	invalid
[0, 60]	[61, 89]	[90, 94]	[95, 100]	[101, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5
MAJOR				
invalid	valid	invalid	invalid	invalid
[0, 60]	[61, 89]	[90, 94]	[95, 100]	[101, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5
LIFE THREATENING				
valid	invalid	invalid	invalid	invalid
[0, 60]	[61, 89]	[90, 94]	[95, 100]	[101, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5

SR-#4.2.4: Pulse measurements							
OK							
invalid	invalid	invalid	valid	invalid	invalid	invalid	invalid
[1, 39]	[40, 49]	[50, 59]	[60, 100]	[101, 120]	[121, 160]	[161, 230]	[231, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7	partition 8
MINOR							
invalid	invalid	invalid	valid	invalid	invalid	invalid	invalid
[1, 39]	[40, 49]	[50, 59]	[60, 100]	[101, 120]	[121, 160]	[161, 230]	[231, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7	partition 8
MAJOR							
invalid	invalid	invalid	valid	invalid	invalid	invalid	invalid
[1, 39]	[40, 49]	[50, 59]	[60, 100]	[101, 120]	[121, 160]	[161, 230]	[231, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7	partition 8
LIFE THREATENING							
invalid	invalid	invalid	valid	invalid	invalid	invalid	invalid
[1, 39]	[40, 49]	[50, 59]	[60, 100]	[101, 120]	[121, 160]	[161, 230]	[231, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7	partition 8

⁸[https://en.wikipedia.org/wiki/Interval_\(mathematics\)](https://en.wikipedia.org/wiki/Interval_(mathematics))

SR-#4.2.5: Blood Pressure measurements - Diastolic FR-#1

OK

invalid	invalid	invalid	valid	invalid	invalid	invalid
[1, 40]	[41, 50]	[51, 59]	[60, 79]	[80, 89]	[90, 119]	[120, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7

MINOR

invalid	invalid	valid	invalid	valid	invalid	invalid
[1, 40]	[41, 50]	[51, 59]	[60, 79]	[80, 89]	[90, 119]	[120, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7

MAJOR

invalid	valid	invalid	invalid	invalid	valid	invalid
[1, 40]	[41, 50]	[51, 59]	[60, 79]	[80, 89]	[90, 119]	[120, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7

LIFE_THREATENING

valid	invalid	invalid	invalid	invalid	invalid	valid
[1, 40]	[41, 50]	[51, 59]	[60, 79]	[80, 89]	[90, 119]	[120, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7

SR-#4.2.5: Blood Pressure measurements - Systolic FR-#2

OK

invalid	invalid	invalid	valid	invalid	invalid	invalid
[1, 39]	[40, 59]	[60, 89]	[90, 129]	[130, 139]	[140, 179]	[180, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7

MINOR

invalid	invalid	valid	invalid	valid	invalid	invalid
[1, 39]	[40, 59]	[60, 89]	[90, 129]	[130, 139]	[140, 179]	[180, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7

MAJOR

invalid	valid	invalid	invalid	invalid	valid	invalid
[1, 39]	[40, 59]	[60, 89]	[90, 129]	[130, 139]	[140, 179]	[180, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7

LIFE_THREATENING

valid	invalid	invalid	invalid	invalid	invalid	valid
[1, 39]	[40, 59]	[60, 89]	[90, 129]	[130, 139]	[140, 179]	[180, +maxInt]
partition 1	partition 2	partition 3	partition 4	partition 5	partition 6	partition 7