UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Aghabayli, Aytaj

Software Runtime Data: Visualization and Integration with Development Data – A Case Study

Master's Thesis (30 ECTS)

Supervisors: Dietmar Pfahl, PhD
Silverio Martínez-Fernández, PhD

Tartu 2019

Software Runtime Data: Visualization and Integration with Development Data – A Case Study

Abstract:

Software quality is one of the key aspects of the software development process. Although software development and usage (runtime) processes produce a different type of data, there is little support for companies to obtain insightful and actionable information from data at the right time. Practitioners face a challenge in identifying software problems during the early software development stages. The goal of the master thesis was to provide actionable real-time information about runtime errors and crashes during the usage of software systems and explore its integration with development data. This work has been done within the project Q-Rapids at Fraunhofer IESE. The selected case is the internal smart village project - Digitale Dörfer (DD). The main contributions of the thesis are: a) collecting available runtime data from the DD the project; b) creating dashboards to make decisions during sprint planning; c) applying CRISP-DM method to the integration of software runtime and development data. The provided connectors and integration scripts are reusable. Reported challenges and lessons learned from the integration of software runtime and development data may be used for further research.

Keywords:

Software quality, software runtime data, external quality, software analytics

CERCS: P170, Computer science, numerical analysis, systems, control

Tarkvara käitusaja andmed: arendusteabe visualiseerimine ja integreerimine - juhtumiuuring

Lühikokkuvõte:

Tarkvara kvaliteet on tarkvaraarenduse protsessi üks peamisi aspekte. Kuigi tarkvaraarenduse ja kasutuse (käitusaja) protsessid toodavad erinevat tüüpi andmeid, on ettevõtetel vähe toetust, et saada õigel ajal andmete põhjal arusaadavat ja tegutsema panevat teavet. Praktikud seisavad silmitsi tarkvaraprobleemide kindlakstegemise väljakutsega varase tarkvaraarenduse etappide ajal. Magistritöö eesmärk oli pakkuda reaalajas tegutsevat teavet tarkvarasüsteemide kasutamise ajal esinevate käitusvigade ja krahhide kohta ning uurida selle integreerimist arendusteabega. See töö on tehtud projekti Q-Rapids raames Fraunhoferi Eksperimentaalse Tarkvaratehnika Instituudis (IESE). Valitud juhtum on sise-nutika küla projekt - Digitale Dörfer (DD). Uurimistöö peamisteks panusteks on: a) DD projektist saadaolevate käitusaja andmete kogumine; b) sprintide planeerimise käigus otsuste tegemiseks juhtpaneelide loomine; c) CRISP-DM meetodi rakendamine tarkvara käitusaja ja arendusteabe integreerimiseks. Pakutavad ühendused ja integratsiooni skriptid on korduvkasutatavad. Edasisteks uuringuteks võib kasutada kaudseid raskusi ja õppetunde, mis on saadud tarkvara käitusaja ja arendusteabe integreerimisest.

Võtmesõnad:

Tarkvara kvaliteet, tarkvara käitusaja andmed, väliskvaliteet, tarkvara analüüs

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

Software quality is an important factor in software engineering projects. Practitioners need to investigate software runtime quality problems during the early software development stages.

There is a lot of work done on analysis software internal and external factors separately. Nevertheless, there is a limitation in integrated analysis. This master thesis captures the topic of software runtime data to improve software quality. In the scope of this research, we study relevant cause dependencies between development time quality of software (referred to as *internal quality* [ISO 25000]) and runtime quality of software (referred to as *external quality* or *quality in-use* [ISO 25000])

This work has been done within the project Q-Rapids[1]. Q-Rapids is the tool to support software practitioners in managing software quality in the Agile development process.

The main research goal of the master thesis is **to provide actionable real-time information about runtime errors and crashes during the usage of software systems, and explore its integration with development data from repositories**.

Based on this research goal we constructed the following three questions:

- Q1. How can we gather runtime data to monitor external quality?
- Q2. How can we visualize quality problems to take actions?
- Q3. How can we integrate runtime data with development data?

In Q1 we explore, from the point of view of practitioners, what runtime data should be collected, which sources of runtime data are suitable, which data are suitable and have enough quality to make decisions on sprint planning, and how we can combine runtime data from different sources.

In Q2 we explore, again from the point of view of practitioners, how to make decisions on sprint planning based on collected runtime data. Since these kinds of decisions are not obvious [1], we study possibilities to visualize runtime data. Visualization is supposed to make it easier to detect quality problems in the software. This, in turn, will help prioritize tasks and thus help make decisions in the sprint planning process.

In Q3 we explore, from the point of view of researchers, the integration of runtime and development data. We wonder how useful it will be to integrate software runtime with development data in order to understand and predict external quality. In other words, what if problems occurred during the use of the software (external quality) are caused by problems during the development of the software (internal quality).

To answer the identified questions, we conducted a case study research. We chose a Fraunhofer IESE internal project, Digitale Dörfer (DD)[2], a platform providing several digital services (such as online shopping, news portal, car sharing services) in smart rural areas.

The main contributions of this thesis are:

[1] Q-Rapids: https://www.q-rapids.eu/
[2] Digitale Dörfer: https://www.digitale-doerfer.de/

- To answer Q1, we created connectors as an extension to a tool developed in the Q-Rapids project.
- To answer Q2, based on collected data, we created dashboards to improve the sprint planning process of software development. These dashboards were evaluated by members of the DD team.
- To answer Q3, we applied the Cross-Industry Standard Process for Data Mining (CRISP-DM) process for runtime data integration and provided the necessary Python scripts.

This thesis is structured as the following (see Figure 1):

Chapter 2. Literature Review reports the review results on the following topics: relevant (runtime) data and collection; visualizations to make a decision on sprint planning; and integration of runtime and development data. Based on identified gaps, we constructed the research described in Chapter 3. Methodology. Chapter 4. Runtime Data Visualization Tool captures the implementation of connectors and dashboards to answer Q1 and Q2. We report results from the evaluation of the implemented tool in Chapter 5. Tool Evaluation reports evaluation results. We describe, applied CRISP-based method to the integration process to answer Q3, in Chapter 6; and its implementation and analysis in Chapter 7. In Chapter 8 we discuss the outcomes, limitations, and lessons learned from the master thesis. Chapter 9 makes conclusion remarks and states future work.

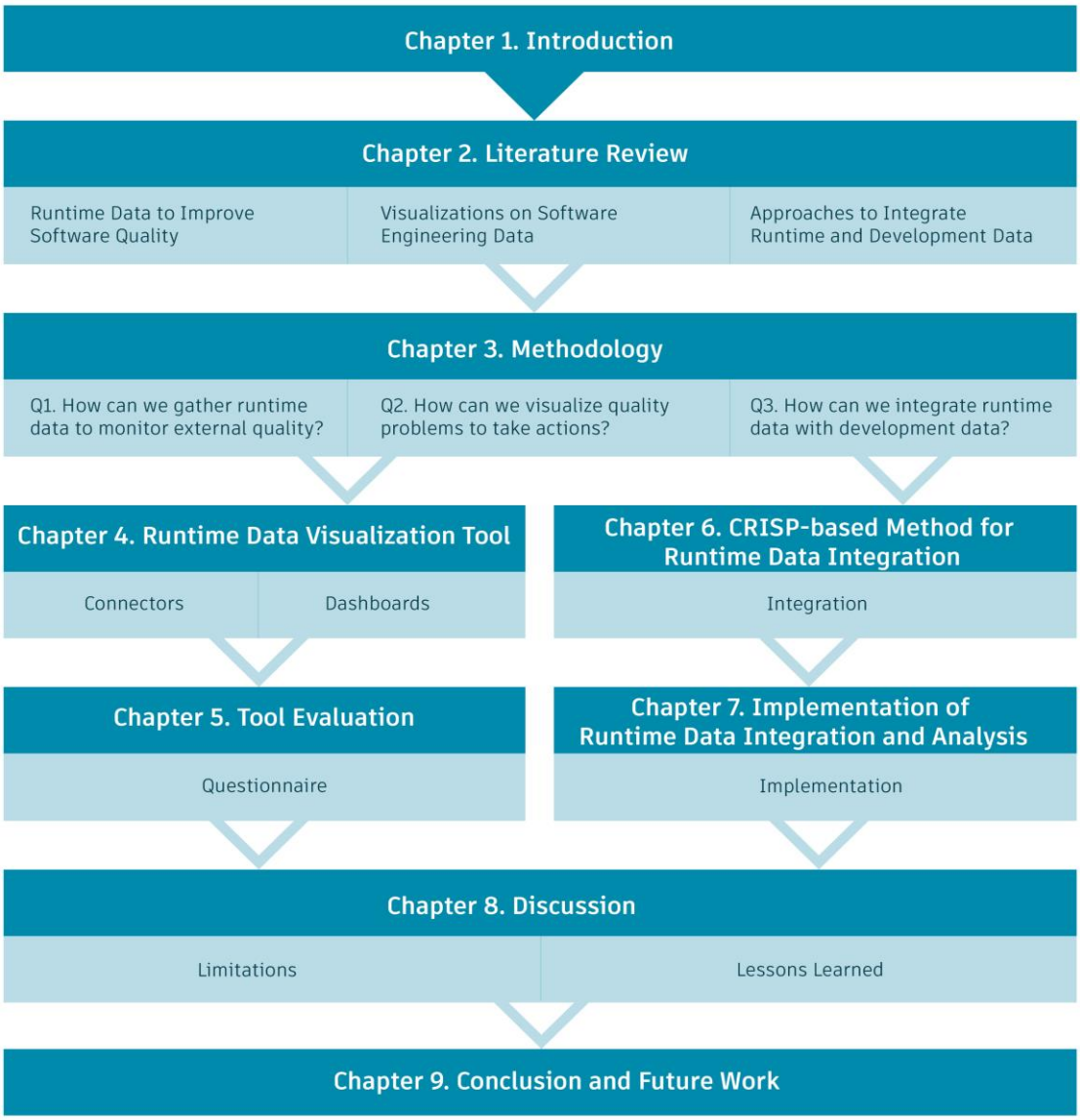


Figure 1. The master thesis structure.

2 Literature Review

In this section, we describe literature review results. We have done research on the following topics:

- runtime data to improve software quality
- visualizations to support decisions on agile software development
- analysis work on the integration of external and internal quality aspects

2.1 Runtime Data to Improve Software Quality

This section reviews the literature related to the usage of runtime data in improvements in software quality.

Many recent papers have focused on the problem of gathering runtime data. Previous studies have reported software analysis methods using runtime data such as error and activity logs. One of the challenges is real-time gathering runtime data. Several studies investigated the collection of server logs data during software run-time and their use in failure prediction [2]. Classification methods (such as support vector machine, rule-based classifier and the nearest neighbour methods) were applied to failure prediction on log data [2]. In this study, the main result was that the nearest neighbour method showed better results than others. A failure prediction clustering algorithm trained on failure and non-failure error log sequences was proposed in [3].

Metzger et al. investigated two approaches: prediction of external failures based on internal failures and prediction of internal failures before they were occurred [4].

Different methods of Machine Learning have been applied to system anomaly detection based on Log data, as described below.

Cao et al. showed anomaly detection by using Decision tree and HMM algorithms in Weblogs from real industry [5]. Du et al. demonstrate applying deep learning techniques into the online log anomaly detection system and comparison of PCA, IM, n-gram, DeepLog. In this paper, log data was analyzed as sentence sequences (NLP) [6].

Fraternali et al. applied log analysis in the tracking of the web path of users, to identify view, design problems, by analysing which path users take to get in the considering page (time of each path, etc.). Used input data: application server logs + the WebML runtime logs [7].

Karim et al. proposed SmartBot framework to runtime execution behaviour analysis with the six machine learning algorithms (BayesNet, SVM, multilayer perceptron (MLP), simple logistic regression, J48, and Random Forest) to analyze malware of mobile applications. Used input data was Trace and Log files [8].

2.2 Visualizations on Software Engineering Data

Nowadays, many companies use real-time visualizations of runtime data to make decisions on sprint planning. We analyzed some examples of the used tools: HockeyApp[3] and AWS CloudWatch[4].

[3] HockeyApp: https://hockeyapp.net/
[4] AWS CloudWatch https://aws.amazon.com/cloudwatch/

Figure 2. Visualization available in HockeyApp: “The number of application crashes per day”.

Figure 2 shows visualization available in HokceyApp tool. This graph shows the number of crashes occurred each day. This helps practitioners to see the overall picture of crashes, but it has limitations in showing information about crashed code class. What leads to not having enough information to take actions in sprint planning.

Figure 3. Visualization available in AWS CloudWatch.

AWS CloudWatch platform is commonly used for gathering log data. Figure 3 shows the example of visualization in AWS CloudWatch. From the figure we can see only time and number of errors, it is hard to distinguish for example: the most occurred log.

Overall, there are limitations in visualizing data coming from heterogeneous data sources to support taking decisions during agile activities (e.g., sprint planning).

2.3 Approaches to Integrate Runtime and Development Data

There have been few empiric investigations into the integration of software runtime and system and process data. So, far there was not found a single set of correlated metrics for all projects [9]. Nagappan et al. reported the prediction of component failure based on mining metrics. Five different software systems of Microsoft were analyzed. Spearman correlation method was applied to identify a set of complexity metrics which are correlated with post-release defects [10].

F. Lautenschlager et al. studied the cloud software system. Authors produced the root cause identification of runtime data collected from different parts of the software and argue that it is important to be data integrated into one tool [11]. In analysis tools such as Zipkin, Prometheus, Grafana, fluentid, Elasticsearch[5] and Kibana[6] (exploring log files) were used. There were indicated that it is not easy to integrate runtime software data collected from different sources, as there is a need to have a common component (e.g. nam-

[5] Elasticsearch: https://www.elastic.co/
[6] Kibana: https://www.elastic.co/products/kibana/

ing, log structure or sharing timeline). As a result, they have created a chatbot, where all runtime data collection, storing and analysing tools were combined.

Most of the previous studies have focused on the usage of software quality data either on runtime or development time. Mostly, researchers produced an analysis of non-on-going projects, which makes the data preparation process easier. Therefore, there is a need for further research on the integration of real project runtime and development data and its analysis.

3 Methodology

In this chapter, we describe problem, goal and constructed research design to answer the defined questions.

3.1 Context

The software development process produces various type of data such as source code, bug reports, check-in histories, and test cases. The data sets not only include millions of data points produced per second about the **usage of the software** (e.g., Facebook or eBay ecosystems), but also data from the **development** (e.g., GitHub with over 14 million projects) [12].

3.2 Problem

As discussed in Sections 2.2 and 2.3, there is little support for companies to obtain insightful and actionable information from data at the right time (e.g., to anticipate quality problems before they occur).

To take actions on external quality aspects, practitioners need tool support during sprint planning. For instance, how can practitioners use runtime data to improve the external quality of their software systems? The difficulties are that the software quality data are collected from different tools and it is heterogeneous. To identify the dependency of external quality aspect with runtime data, there is a need to integrate them.

3.3 Goal

The main goal of this master thesis is the following:

To provide actionable real-time information about runtime errors and crashes during the usage of software systems, and explore its integration with development data from repositories.

This research goal is divided into three questions:

- Q1. How can we gather runtime data to monitor external quality?
- Q2. How can we visualize quality problems to take actions?
- Q3. How can we integrate runtime data with development data?

3.4 Design

To answer the questions above, we envisaged a case study methodology [13]. This approach was chosen to gain a detailed understanding of software quality data, its usefulness, and integration possibilities. Initially, we selected a case (i.e., project) where we learned the different data sources and conducted preliminary data analysis work. However, the quality of the data in this project from a runtime perspective was insufficient.

Due to the inability to gather the data in an external project, we decided to select a project in-house where we could ingest the runtime data in the required format and suitable quality. The project is called Digitale Dörfer (Digital Villages). Digitale Dörfer is a technical platform, which provides solutions and services for smart rural areas in Germany [14].

The major advantages of the selected case were: (a) having access to runtime data sources of the software; and, (b) possibility to easily communicate with team members working on the project.

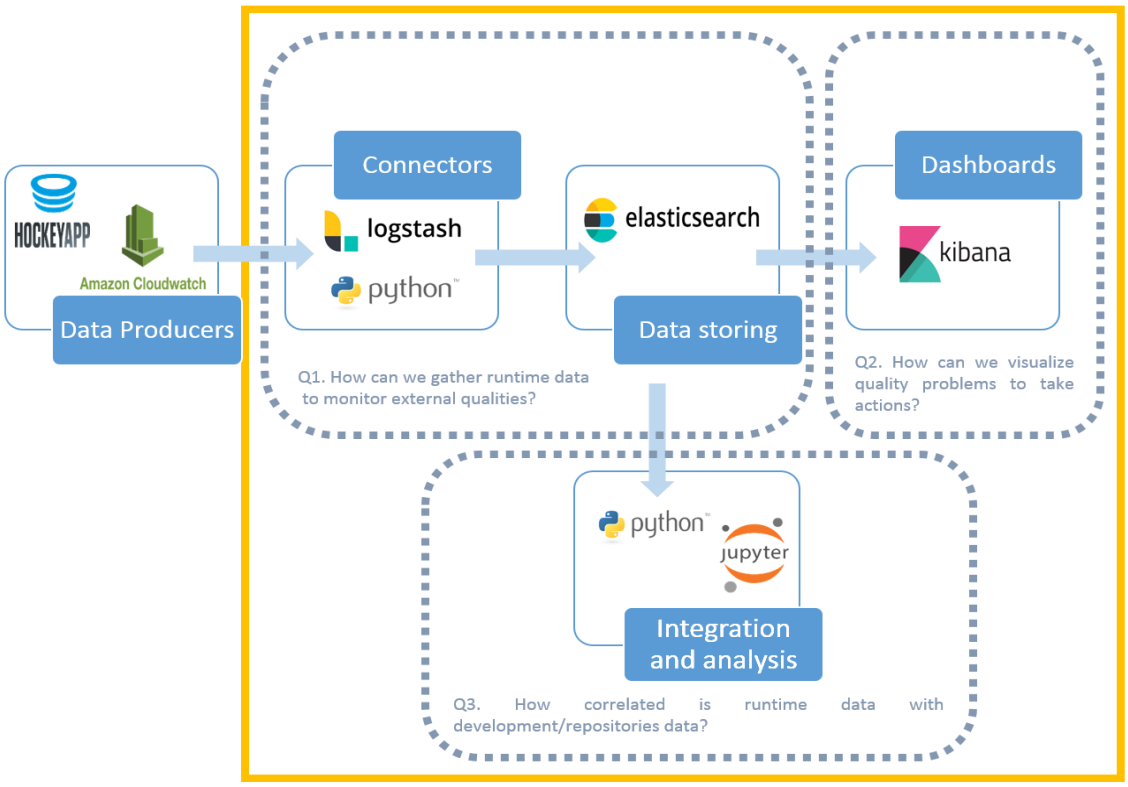


Figure 4. Approach overview.

After the selection of the case, we identified two main actions to address our aforementioned questions (Figure 4):

1. Gathering runtime data, creating dashboards based on the collected data and evaluating the tool (Q1 and Q2).
    a. Runtime Data Visualization Tool
    b. Tool Evaluation
2. Applying CRISP-DM methodology on runtime data integration and analysis (Q3).
    a. CRISP-based Method for Runtime Data Integration
    b. Implementation of Runtime Data Integration and Analysis

The first action captures answers to questions the first two and intended to be used by practitioners. When the second action is focused on the third question and convenient to be used by researchers. This is explained in the following subsections.

3.4.1 Runtime Data Visualization Tool

In this section, we describe the structure of implemented work to create the runtime data visualization tool.

We had an initial meeting with the architect of the DD project, to investigate data collection possibilities and currently used tools in analysing runtime data. There were investigated two types of collected runtime data: application crashes and backend log data. Applica-

tion crashes are stored in HockeyApp, backend log data in Amazon CloudWatch platform. As each data type was kept in different tools, we decided to implement two connectors for gathering each type of runtime data.

First, we gathered data in the Elasticsearch storage, which is a real-time scalable search platform. In this platform, we can easily store our data, filter, search and visualize on its plugin. The visualization plugin of Elasticsearch is called Kibana, which is an open source service. Then, we created four dashboards, based on the DD project architect interests, on Kibana. One of the dashboards presents application crashes data, the other three – HTTP log data. The overall architecture of the implemented work is depicted in Figure 5

The connectors 1 and 2 were implemented in Python 3. For the implementation of the connector 2, we additionally used the third component of Elastic Stack[7]- Logstash[8]. This plugin makes easier continuous streaming of the logs to the Elasticsearch.

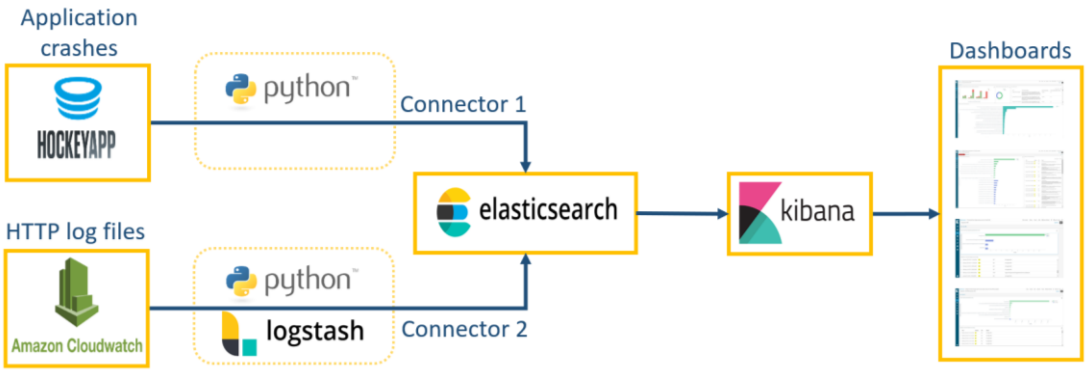


Figure 5. The architecture of the runtime data visualization tool implementation.

To sum up, the purpose of implementing connectors, which send runtime data from initial storages to Elasticsearch, was to get an answer to the Q1: "How can we gather runtime data to monitor external qualities?". The dashboards, created based on collected runtime data, demonstrate answers to the Q2: "How can we visualize quality problems to take actions?". Details on this activity are reported in Chapter 4.

3.4.2 Tool Evaluation

To evaluate the dashboards, we conducted a meeting with the relevant team members of the project. We selected four candidates by the advice of the DD team leader. Participants had more than 2 years' experience and carried roles of a software architect, backend developer, mobile developer and organizing a mobile team. Table 1 shows roles and experience in years of the participants.

The benefit of questioning a small number of people was the familiarity of the participants with the data sets. Therefore, they could provide precise feedback. On the other hand, it is hard to generalize the results of the questionnaire and open discussion based on the opinions of a small group of practitioners.

[7] Elastic Stack: https://www.elastic.co/products/
[8] Logstash: https://www.elastic.co/products/logstash/

Table 1. Questionnaire participants role and experience.

Participants	Role	Experience (Years)
P1	Software architect	4-6
P2	Backend developer	2-4
P3	Mobile developer	2-4
P4	Organizing mobile team	2-4

First, the dashboards were presented to the team members. During the presentation, we had exploring session where participants could get familiar with the tool and found out answers to the given small tasks.

After the presentation, we asked members to fill out the online questionnaire. The questionnaire was constructed based on Technology Acceptance Model 3 (TAM3) [15].

To evaluate the tool and presented data we selected the following constructs:

1. Perceived Usefulness (PU)
2. Perceived Ease of Use (PEOU)
3. Relevance to Job (REL)
4. Output quality (OUT)
5. Behavioral Intention (BI)

By **output quality**, we have identified the correctness of visualized data. **Job relevance** helped us to understand how the tool is relevant to the job of participants. **Perceived usefulness** was chosen to evaluate to what extent the dashboards are useful in the improvement of software quality. In what degree the dashboards user-friendly and interactive was evaluated by **perceived ease of use**. By item **behavioral intention** we have clarified how practitioners are interested in using the created dashboards. Each statement was evaluated in 7 degrees (1 - strongly disagree, 2 - mostly disagree, 3 - slightly disagree, 4 – neither disagree nor agree, 5 – slightly agree, 6 – mostly agree, 7 - strongly agree). The questionnaire itself can be found in Appendix II.

After filling the questionnaire, we had an open discussion session, where participants shared their opinions about the strengths and weaknesses of the tool.

Details on the outcomes of this activity are reported in Chapter 5.

3.4.3 CRISP-based Method for Runtime Data Integration

To integrate runtime data with development data, we followed the CRISP-DM process method (Figure 6). CRISP-DM is the cross-industry process for data mining [16]. The CRISP methodology is widely used in data analysis problems. It provides a well-defined structure for planning data-driven projects.

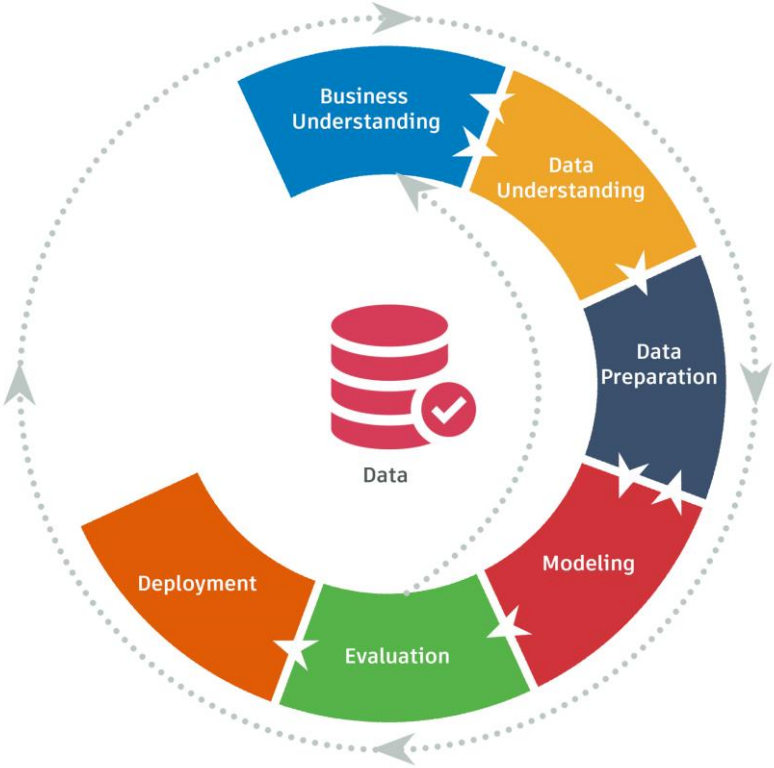


Figure 6. CRISP-DM Process Method.

CRISP-DM process includes the following steps:

1. Business Understanding
2. Data Understanding
3. Data Preparation
4. Modeling
5. Evaluation
6. Deployment

In the scope of the master thesis, we covered from data understanding (step 2) to modeling/analysis (step 4) the CRISP-DM.

Details on this activity are reported in Chapter 6.

3.4.4 Implementation of Runtime Data Integration and Analysis

In this thesis work, we report the implementation of only successfully integrated data sets.

All three steps of CRISP-DM were implemented in Python 3 and demonstrated on Jupyter notebook (Appendix III). The notebook contains the following sections:

- Data Preparation
- Data Integration
- Data Analysis

The default format of the data in the Elasticsearch tool is JSON. Due to that, we gathered the data in JSON format, then by additional script formatted to CSV (see Appendix I). In the notebook, we used data in CSV format.

Details on this activity are reported in Chapter 7.

4 Runtime Data Visualization Tool

In this chapter, we report implementation and description of the dashboards. Section 4.2 describes the collected software runtime data. Section 4.2 shows the architecture of connectors. The purpose of connectors is to collect runtime data from sources and send to the needed destination. Section 4.3 depicts created dashboards.

4.1 Runtime Data

There were investigated two types of DD project runtime data: application crashes and backend log data. Application crashes were stored in HockeyApp, server log data in Amazon CloudWatch platform. Server log data contains access and error logs. Due to the lack of server error logs in the production environment of the product, we do not report work done on it in this section.

4.2 Connectors

Implemented connectors are the extensions to the Q-Rapids project. The purpose is to gather runtime data of the project in the unit platform. As each data type was kept in different tools, we required in creating two separate connectors. The design of the implementation is shown in Section 3.4.1.

The connectors are implemented in Python 3. For parsing application crashes, we used the HockeyApp API [17]. Code is reported in Appendix I.

For parsing HTTP access logs, we used Amazon CloudWatch API to collect data and the third component of Elastic Stack – Logstash to parse to the Elasticsearch [18]. This plugin makes easier continuous streaming of the logs to the Elasticsearch. Code is reported in Appendix I.

4.3 Dashboards

The main purpose of the dashboards is to improve the decisions making process in software development sprint planning meetings.

We created dashboards in Kibana, which is an open source visualization plugin of Elasticsearch. The tool provides an interactive and user-friendly interface and helps easily to track data.

We built up the following four dashboards based on the discussion with Digitale Dörfer stakeholders:

1. Prioritized crash reasons per version
2. Prioritized 4xx HTTP access errors
3. Prioritized 5xx HTTP access errors
4. Prioritized requests with success status code 2xx

The first dashboard presents application crashes data, the other three – HTTP access logs.

Due to confidentiality reason, we covered some parts of the dashboards.

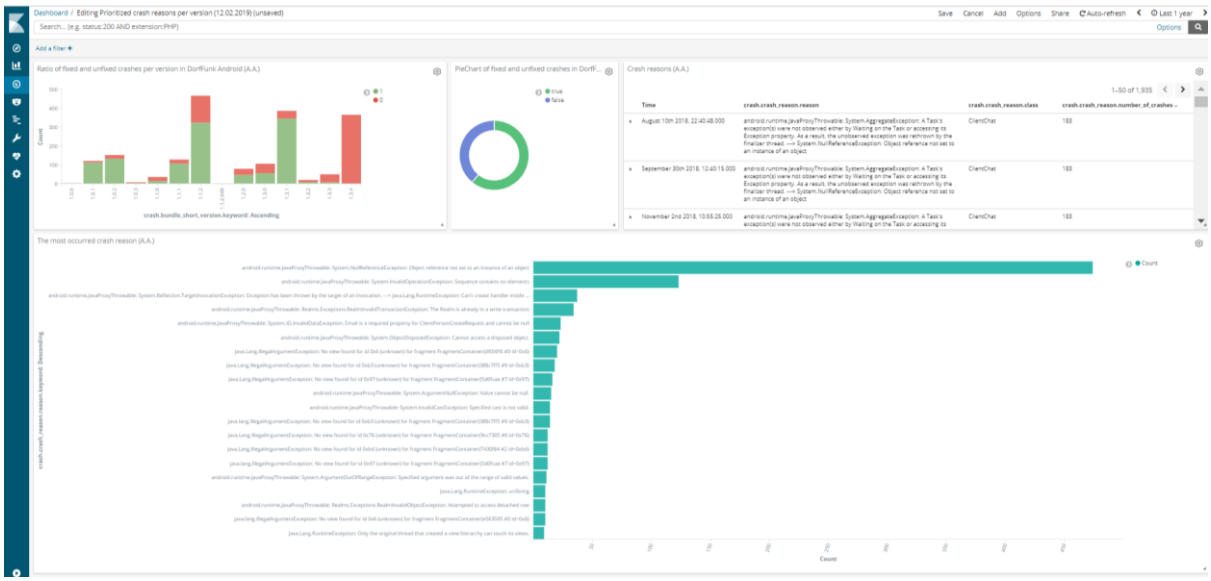


Figure 7. Dashboard 1: “Prioritized crash reasons per version”.

Figure 7 presents a dashboard of the most occurred crashes during the selected time period. The dashboard consists of four parts.

The visualization on the upper left corner shows how many crashes had each version of the software. Additionally, it depicts the status of the crashes (solved or unsolved). The pie chart in the middle shows the ratio of solved and unsolved crashes. Filtering of the data can be done by selecting elements from graphs. For example, John is a project manager. John sees that in the version 1.3.4 number of crashes is noticeably higher than in version 1.3.3. John is interested to know why the previous version was more stable. He selects the version 1.3.4 and status ‘unsolved’ and sees on the bar chart on the bottom the most occurred crash. John can find on the right corner of the dashboard entire crash reasons and timestamp when the crash appeared. He identifies that the crashes reasons indicate mostly the newly added feature. John decides to spend more time on the development of this feature during the next sprint.

The dashboard (Figure 7) can help practitioners to understand the progress of solving application crashes and decide the needed time to spend on it during the next sprint.

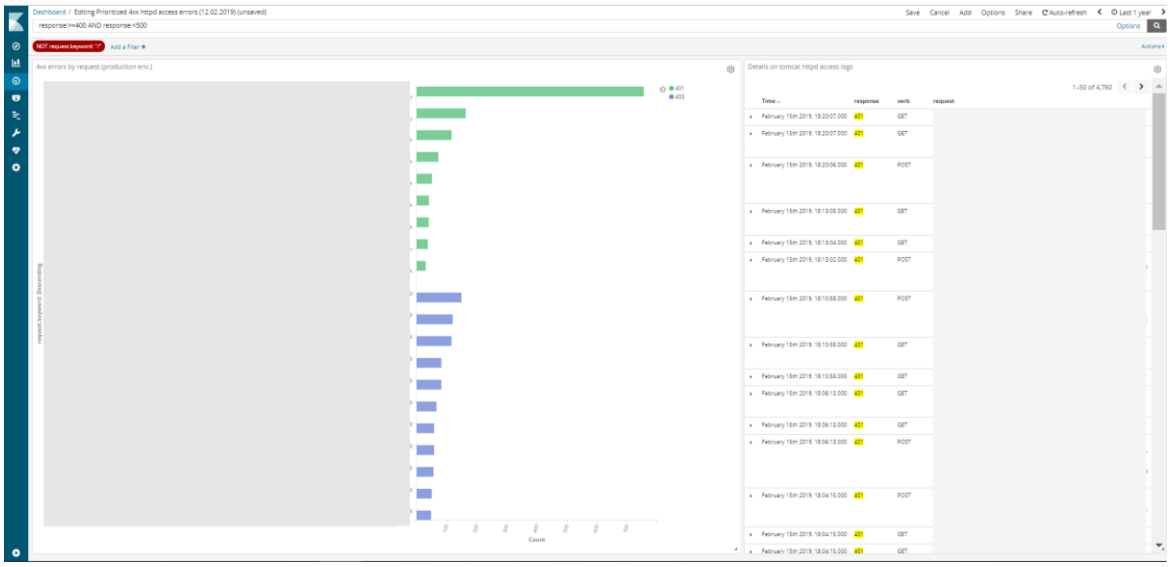


Figure 8. Dashboard 2: “Prioritized 4xx HTTP access errors”.

Figure 8 shows the dashboard of prioritized 4xx errors, which help to identify occurred problems in the software requests. The dashboard shows the number of the most occurred 20 requests with 4xx responses and their error codes. In our case, the most appeared errors had codes ‘401’ (Unauthorized error) and ‘403’ (Forbidden error). ‘403’ has a more significant role, as the reason for Forbidden errors can be misleading in the software implementation.

By using this dashboard project members can identify problems in the software and prioritize improvements during agile meetings.

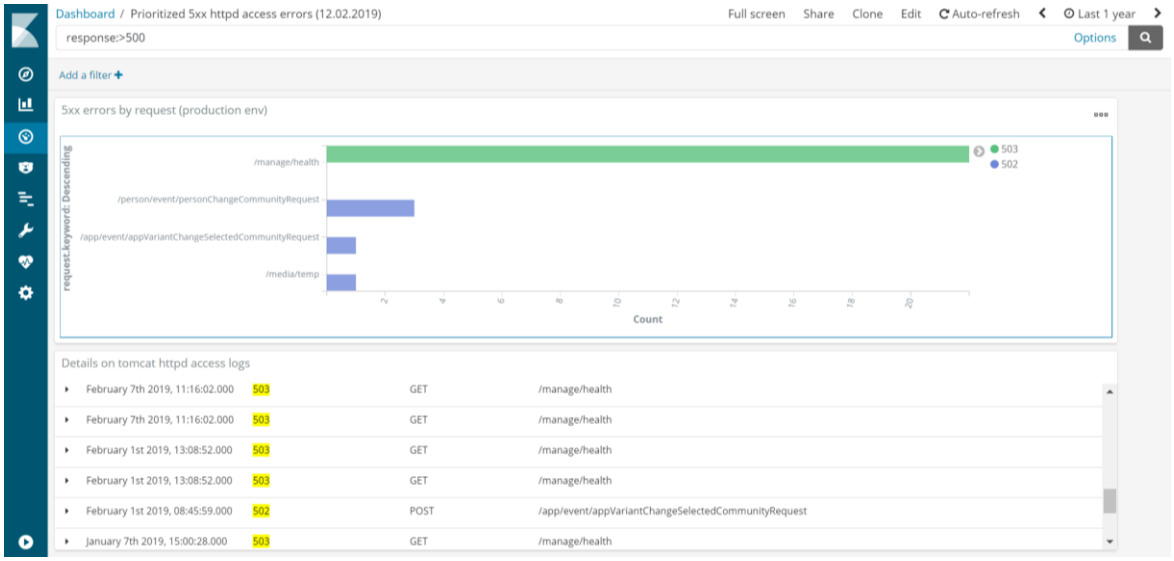


Figure 9. Dashboard 3: “Prioritized 5xx HTTP access errors”.

Figure 9 depicts prioritized requests with 5xx server errors as a response.

From the vertical bar chart (upper) we can see the most occurred requests and their error code. In our case, there were two types of errors ‘502’ and ‘503’. The most occurred error was ‘503’ and all from the request ‘manage/health’. The table (bottom) shows the method (GET/POST) and timestamp of the requests. The dashboard helps to identify server problems faster and intended to be an indicator to pay attention to this part of the software in the next sprint planning.

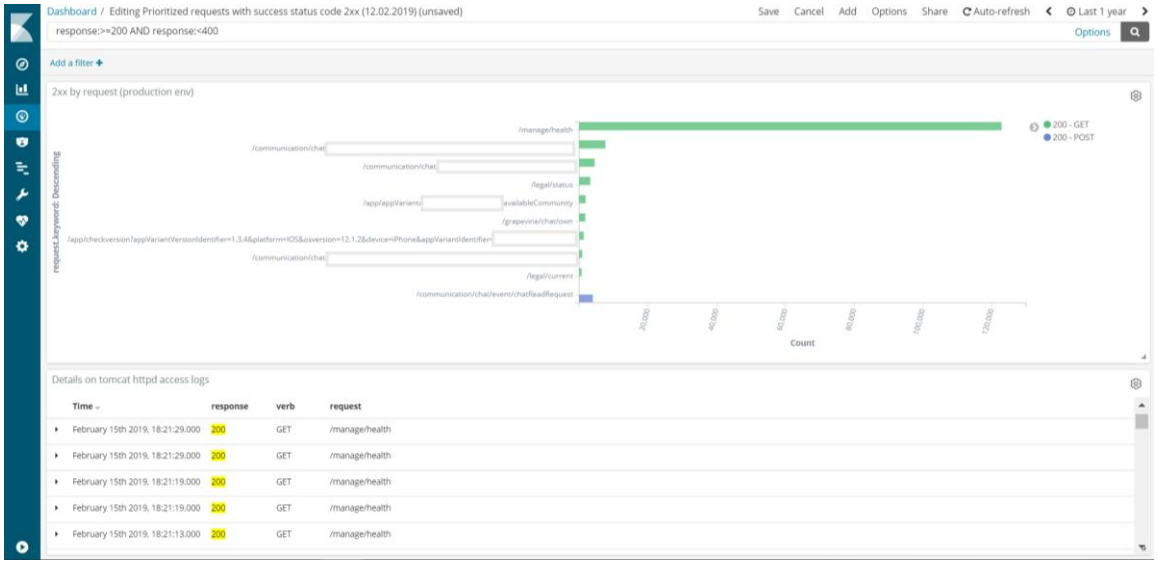


Figure 10. Dashboard 4: “Prioritized requests with success status code 2xx”.

Figure 10 represents the dashboard of prioritized requests with status code 2xx. The bar chart (upper) shows the most frequent called GET and POST requests with a success response code - ‘200’.

This dashboard was created by the request of the project application architect. By using the dashboard team members can identify the most used part of the software. It will help in feature prioritization of the product during sprint planning.

5 Tool Evaluation

In this chapter, we show and summarize the tool questionnaire results (Section 5.1) and open discussion outcomes (Section 5.2).

The following shows selected five TAM3 constructs and questions [19]:

1. Perceived Usefulness (PU)
   1.1. Using the system would improve my performance in my job.
   1.2. Using the system in my job would increase my productivity.
   1.3. Using the system would enhance my effectiveness in my job.
   1.4. I find the system to be useful in my job.
2. Perceived Ease of Use (PEOU)
   2.1. My interaction with the system is clear and understandable.
   2.2. Interacting with the system does not require a lot of my mental effort.
   2.3. I find the system to be easy to use.
   2.4. I find it easy to get the system to do what I want it to do.
3. Relevance to Job (REL)
   3.1. In my job, usage of the system is important.
   3.2. In my job, usage of the system is relevant.
   3.3. The use of the system is pertinent to my various job-related tasks.
4. Output quality (OUT)
   4.1. The quality of the output I get from the system is high.
   4.2. I have no problem with the quality of the system's output.
   4.3. I rate the results from the system to be excellent.
5. Behavior Intention to Use (BI)
   5.1. Assuming I had access to the system, I intend to use it.
   5.2. Given that I had access to the system, I predict that I would use it.
   5.3. I plan to use the system in the next <n> months.

The questionnaire itself you can find in Appendix II.

5.1 Questionnaire Results

Initial questionnaire answers were scaled from **one** to **seven**. To analyze results easier, we moved to scale from **minus three** to **three** (-3, -2, -1, 0, 1, 2, 3).

Table 2 summarizes the questionnaire results. Each row corresponds to one of the five constructs of the TAM. The columns labeled 'P1' to 'P4' show the means of the answers of each participant with regards to each construct. 'N' indicates the number of questions in the construct. The following three columns display statistics, such as minimum ('Min'), maximum ('Max') and rounded average ('AVG') of the 'P1'-'P4' per each construct.

Table 2. Statistics on the questionnaire results.

	P1	P2	P3	P4	N	Min	Max	AVG
Perceived Usefulness	1.25	2	-0.75	0.75	4	-0.75	2	0.81

(PU)								
Perceived Ease of Use (PEOU)	0.25	-0.75	-0.75	2	4	-0.75	2	0.19
Relevance to Job (REL)	2	2	1.67	1	3	1	2	1.67
Output Quality (OUT)	2.33	1.67	-0.33	1.33	3	-0.33	2.33	1.25
Behavioral Intention (BI)	3	2	2	2	3	2	3	2.25

To see an overall picture we visualized results on the box plot (Figure 11). The graph includes minimum (bottom line), maximum (upper line), average (middle dash line) and median (middle straight line) values of the constructs with regards to averaged answers of each participant.

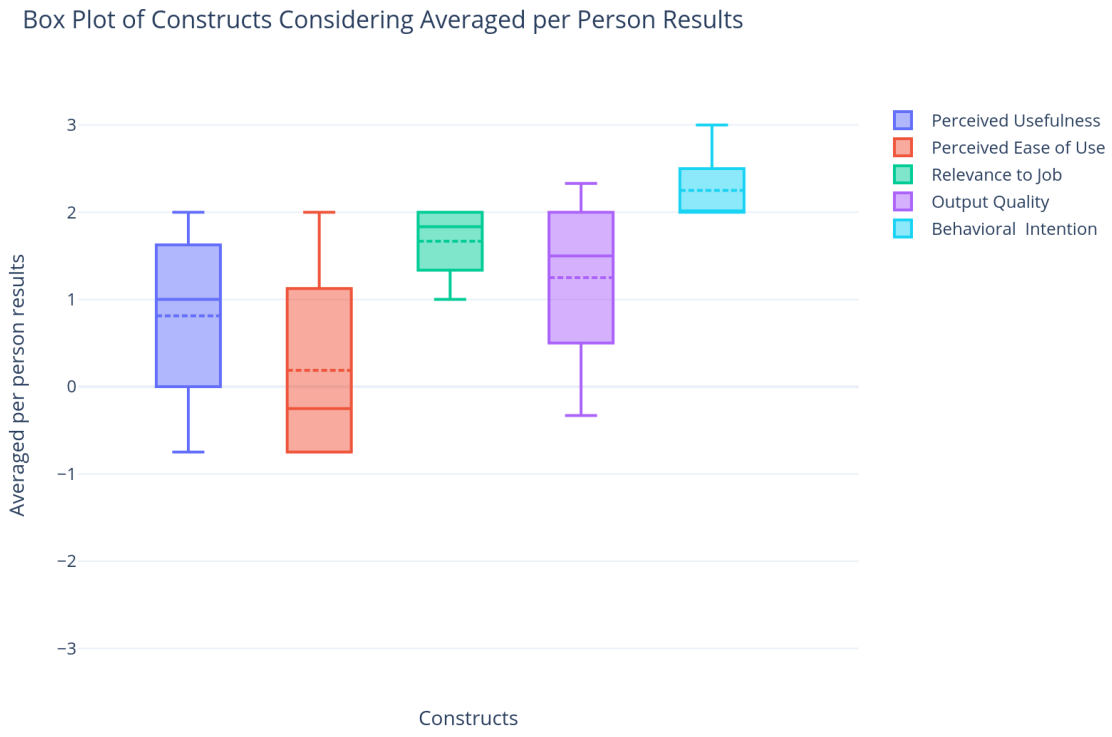


Figure 11. Box plot of constructs with regards to averaged answers of each participant (-3 - strongly disagree, -2 - mostly disagree, -1 - slightly disagree, 0 – neither disagree nor agree, 1 – slightly agree, 2 – mostly agree, 3 - strongly agree).

From the results, we can see that averages per all constructs are above zero (positively evaluated).

The opinions of the participants about the usefulness of the dashboards were slightly different from each other. Overall average of the Perceived Usefulness responses was positive (0.81).

The results of Perceived Ease of Use show that participants found the tool neither easy nor hard to use. The average of the construct responses was close to zero (0.19).

All participants considered the dashboards to be related to their job. The average of the construct Relevance to Job was close to ‘mostly agree’ (1.67).

The average of the Output Quality construct was between ‘slightly agree’ and ‘mostly agree’ (1.25).

The construct Behavioral Intention is the most positively voted construct, which has an average is equaled to 2.25. What can be interpreted as “All participants show high interest to use the tool in future”.

During the evaluation of the results, there were noticed some unclear questions. For example, when answering to item ‘I plan to use the system in the next <n> months’ of construct ‘Behavioral Intention’, some participants answered ‘yes’ or ‘no’ instead of providing a number for variable n.

5.2 Open Discussion

During the open discussion session, we have discussed the strengths and weaknesses of the tool. In Figure 12 strengths and weaknesses are depicted respectively in green and orange cards.

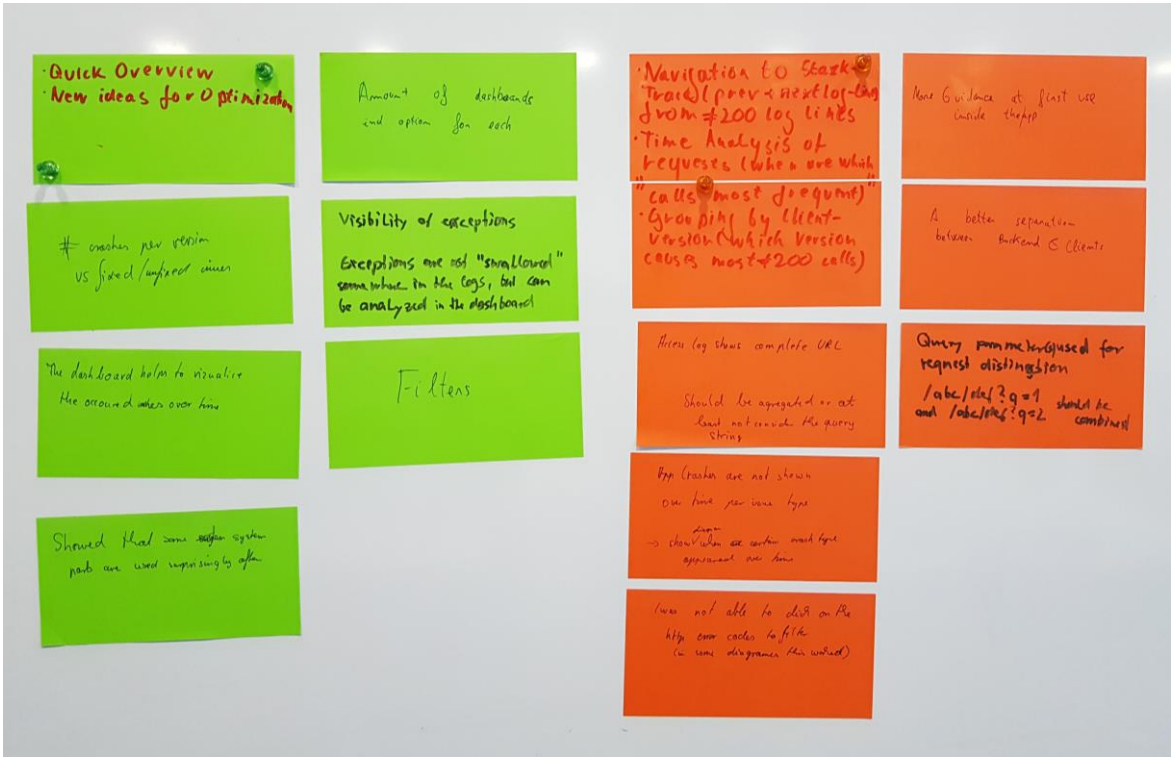


Figure 12. Open discussion results.

We made categorization and grouped open discussion responses.

The identified strengths of the dashboards were the following:

- Ability to show an overview of the system problems occurring during runtime;
- Ability to analyze exceptions;
- Ability to make the crashes that occurred over time visible.

There were two types of weaknesses of the dashboards identified:

- Problems related to existing features that need improvement;
- Missing features.

Suggestions were: to include more hyperlinks that relate one dashboard with the other, add the possibility to customize the dashboards easily and aggregation of requests with defined "warning" threshold.

The defined weakness of the dashboards was including query parameters when differentiating between request. The suggestion was aggregating access logs based on queries.

Overall, the evaluation showed satisfactoriness about the dashboards and considered the tool to be useful in problem identifying.

6 CRISP-based Method for Runtime Data Integration

This chapter describes the three intermediate steps of CRISP-DM (Figure 6) followed in the runtime data integration process [16]. We applied this approach to have a clear overview of the software quality data of the DD project, understand data sets and identify possibilities to integrate fields of the internal and external quality measures.

The structure of this chapter is the following:

1. Data Understanding
2. Data Preparation
3. Modeling and Analysis

6.1 Data Understanding

In this section, we describe relevant data sets of Digitale Dörfer project (0), features needed to integrate (6.1.2) and data quality (6.1.3).

6.1.1 Relevant Data Sets

In this subsection, we report available data sets of the DD project potentially relevant to integrate. Data sets cannot be made public due to confidentiality reason.

Table 3 shows relevant runtime and development data (the second column) and their origins (the third column).

Table 3. Relevant runtime and development data sets of the DD project and their origins.

	Relevant Data Sets	Origin
Runtime (external quality)	Access logs	HTTP access logs (CloudWatch)
	Error logs	HTTP error logs (CloudWatch)
	Crashes	Application crashes (HockeyApp)
	Sprint issues	Issue tracking system (JIRA)
Development (internal quality)	Code quality measures	Static code analysis tool (SonarQube)
	Quality rue violations	Static code analysis tool (SonarQube)
	Commits	Version control system (Git)

In our case, **runtime data sets** include access and error logs; crashes; and sprint issues:

1. **Access logs** data set contains all HTTP server access requests of the project. These data were collected from the Amazon CloudWatch platform.
2. **Error logs** data set contains all HTTP server errors of the project. These data were collected from the Amazon CloudWatch platform.

3. **Crashes** data set contains android application crashes occurred runtime. These data were collected from the HockeyApp platform.
4. **Sprint issues** data set contains all reported bugs, tasks and change requests. These data were collected from the tool Jira[9].

Development data sets include code quality measures; quality rule violations; and commits:

1. **Code quality measures** data set contains metric evaluations for either each file, directory, or module. These data were collected from the SonarQube[10] tool.
2. **Quality rule violations** data set contains each rule violation for either each file, directory, or module. These data were collected from the SonarQube tool.
3. **Commits** data set contains source code changes in files. These data were collected from the version control system (Git[11]).

6.1.2 Relevant Features

This subchapter describes relevant features from runtime and development data sets to integrate. Identifying potentially relevant features for quality modeling: (1) identifiers required for data integration and (2) features capturing potentially relevant quality aspects.

Here, we report the minimum features needed to integrate the data sets.

Runtime data:

Table 4 shows relevant runtime data sets and features.

We extracted from each data set ‘timestamp’, as this is a key artifact for the runtime data.

From access logs, we extracted ‘request’ (HTTP request) and ‘response code’. As people from DD needed to know how the software was accessed. This can be distinguished by request name and response code. The ‘request’ is a key factor in possible integrations. The initial idea was to integrate access logs with development data sets by ‘request’.

From the data set error logs, we selected ‘error type’, this field is the only granularity factor in the data set. This field has a description of the occurred errors.

From data set crashes ‘crash reason’, ‘status’ and ‘class’ were considered as a relevant feature. The ‘crash reason’ filed shows the full description of the classes and reason for the failed crashes. We consider this feature important to be able to connect the data set with development data.

Relevant features of data set sprint issues are ‘issueid’ and ‘issuetype’. We extracted the field ‘issueid’, as it is a key factor in the integration of this data set with commits. We needed the feature ‘issuetype’ to be able to distinguish bugs from stories, tasks, and change requests.

[9] Jira: https://www.atlassian.com/software/jira
[10] SonarQube: https://www.sonarqube.org/
[11] Git: https://git-scm.com/

Table 4. Relevant features in runtime data sets.

	Relevant Features
Access logs	timestamp, request, response code
Error logs	timestamp, error type
Crashes	timestamp, crash reason, status, class (if exists)
Sprint issues	timestamp, issueid, issuetype

Development data:

Table 5 shows the relevant development data sets and features.

From each development data set we have selected 'timestamp' and 'path' (path to the file or class), as these factors present granularity of the data sets.

In data set code quality measures, the relevant features are 'metric' and 'value'. 'Metric' indicates quality measure type, 'value' - the quantitative measure of the metric.

Extracted feature from the quality rule violations data set was 'rule'. This field gives us information about rule violation distinguished by the static code analysis tool.

From commits data set, we extracted the field 'issues', which is the list of the solved issues by each code change. This factor is needed to be able to integrate the data set with sprint issues.

Table 5. Relevant features in development data sets.

	Relevant Features
Code quality measures	timestamp, path, metric, value
Quality rule violations	timestamp, path, rule
Commits	timestamp, path, issues

6.1.3 Quality of data

In this subsection, we report the quality of the integrated data sets and issues made integration not possible.

The number of samples (rows) in the integrated data sets is the following:

- Sprint issues – 9,925
- Code quality measures – 2,245,945
- Quality rule violations – 156,548
- Commits - 22,907

More details about the data quality of integrated data sets can be found in Appendix III.

In the collected data, we had the following quality issues, which impede some integration among data sets:

- **Incomplete information** - data sets contain missing data. For example, some crashes have class name others do not.
- **Data inconsistency** - in different data sets the field 'path' is constructed differently.
- **Inaccurate data** - not all data sets correspond to the same software version and part.

We report solving incomplete information and data inconsistency problems in Chapter 6.2.

Due to data inconsistency and inaccurate data quality problems, we could not integrate access logs, error logs and crashes with development data (external quality) (Figure 13).

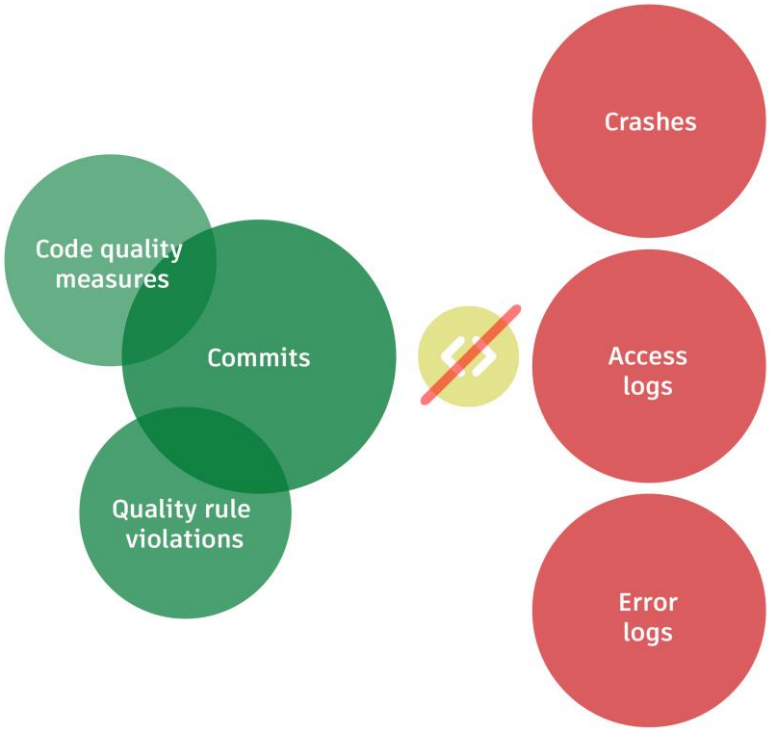


Figure 13. Failed integration approaches.

We failed in the integration of crashes with development data, as available commits data set did not contain changes to the application source code repositories.

Access and error log data sets do not have a direct link with the development data granularity factors. This fact made the integration inapplicable.

6.2 Data Preparation

This section gives an overview of the data cleaning approach and feature engineering for individual data sets, independently from one another. And, describes how the data-set-specific quality problems identified during the data understanding step were handled.

6.2.1 Data Cleaning

This subsection describes the cleaning process of the relevant data sets.

First, we filtered data sets from non-used features and kept only relevant ones. Then, we cleaned data sets from missing values and duplications. We removed missing values to solve incomplete information issue.

We extracted the following rows:

- From sprint issues data set - reported bugs
- From commits data set - commits, which solved an issue

The common ‘timestamp’ period was taken for all data sets.

After data cleaning, we produced the feature engineering process described in the following subsection.

6.2.2 Feature Engineering

These subsection reports feature engineering produced to integrate data sets.

To solve data inconsistency quality problem, we constructed new identical ‘path’ structure for all data sets.

Code quality measures were constructed in the following way: first, for each metric, we created a new column and calculated the average of the metric values per each file during the identified period.

Code violation rules were grouped by common ‘path’ and calculated new field - ‘number of violations’ in each path.

In the commits data set, we reconstructed the field ‘issues’ and put each issue in a separate row.

In sprint issues, we have done feature engineering after integration with commits (described in Section 6.3.1). We created a new field ‘number of bugs’ and per each file, we calculated an overall number of occurred bugs. Integrated files which had no bugs were considered as having ‘zero’ bugs.

6.3 Modeling

In this section, we describe the process of data integration and analysis approach.

6.3.1 Integration

Figure 14 shows the structure of the integration approach.

We could easily integrate bugs with commits by issue id. What we can not say for other development data sets. Due to the absence of file (class) path in sprint issues, direct integration of it with code quality measures and quality rule violations was impossible. We used an indirect approach to integrate bugs with development data.

After integration of commits with bugs, we have integrated code quality measures and quality rule violations by a file (class) path with the commits. In that way, we indirectly integrate bugs with static code quality measures and rule violations.

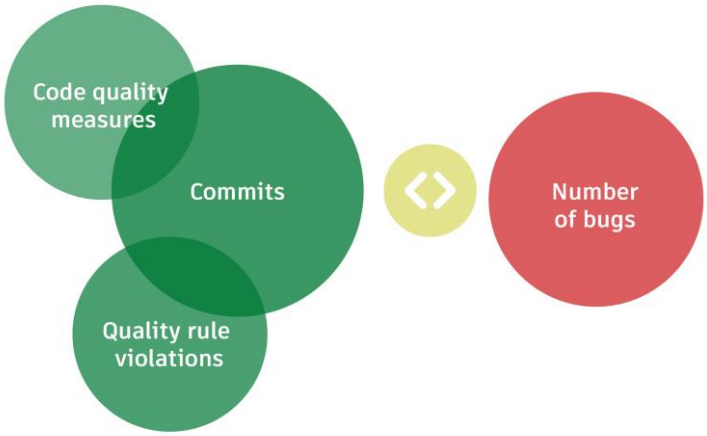


Figure 14. Integration of the number of bugs with software development quality factors.

As a conclusion, we integrated bugs directly with commits and indirectly with static code quality measures and rule violations.

6.3.2 Analysis

After the integration of the bugs with code quality measures, quality rule violations and commits, we performed analysis on the integrated data. The purpose was to analyze to what extent we can use development data in the identification of software runtime quality problems.

The constructed hypotheses were the following:

1. The number of bugs occurred in the file is positively correlated with the code quality measures of the file.
2. The number of bugs occurred in the file is positively correlated with the number of violation rules founded in this file.

Considering data quality aspects to identify dependencies between data sets we decided to apply correlation analysis. We calculated the Spearman correlation coefficients for integrated data sets. Results are reported in Section 7.3.

7 Implementation of Runtime Data Integration and Analysis

This chapter shows the implemented work based on the CRISP-DM methodology and analysis. The implementation process was divided into two sections: Data Preparation (Section 7.1) and Data Integration (Section 7.2). The work is implemented in Python 3 and reported on Jupyter notebook (see Appendix III).

7.1 Data Preparation

In this section, we provide important parts of the code.

First, we have loaded relevant data sets from the respective repositories. Data were collected in JSON format, then parsed to CSV using the script in Appendix I. Then, we extracted relevant features, produced cleaning and data engineering processes.

To follow the instruction, check the comments.

Sprint issues

```
#List of relevant feature names
relevant_features = ['created', 'issuekey', 'issuetype']

#Extracting relevant features
jira_bugs = jira_bugs[relevant_features]

#Filtering only issues which are 'Bug's
jira_bugs = jira_bugs[(jira_bugs.issuetype == 'Bug')]
```

Commits data

```
#List of relevant feature names
relevant_features = ['date', 'filename', 'issues']

#Extracting only changes which solved an issue and needed features
git_issue_solved = git_data[git_data.issues != '[]'][relevant_features]

#Changing format of column 'issues' from 'str' to 'list'
git_issue_solved = git_issue_solved.assign(issues = git_issue_solved.issues.apply
(lambda x: eval(x)))

#Splitting commits which solved several issues
git_issue_solved = git_issue_solved.issues.apply(pd.Series).merge(git_issue_solve
d, left_index=True, right_index=True).drop('issues', axis = 1).melt(id_vars = ['d
ate', 'filename'], value_name = 'issue').dropna().drop('variable', axis = 1)
```

Code quality measures

```
#Creating field which contains only date
sq_m_data = sonarqube_m_data.assign(date = sq_m_data['snapshotDate'].apply(lambda
 x: pd.to_datetime(x).date()))

#Filtering only measures per "file"s and averaging values per each metric
sq_metric = pd.DataFrame(sq_m_data[(sqe_m_data.qualifier == 'FIL')].groupby(['key
', 'metric']).floatvalue.mean().unstack().reset_index())
```

```
6.
7.  #Structuring filename feature
8.  sq_metric = sq_metric.assign(filename = sq_metric.key.apply(lambda x: (re.sub('.*
    :', '', x)))).drop('key', axis=1)
```

Code quality rule violations

```
1.  #Grouping by file and calculating the number of rule violations per each
2.  sq_issues = pd.DataFrame(sq_issue_data.groupby(['component']).rule.count().reset_
    index())
3.
4.  #Structuring filename feature
5.  sq_issues = sq_issues.assign(filename = sq_issues.key.apply(lambda x: (re.sub('.*
    :', '', x)))).drop('key', axis=1)
```

7.2 Data Integration

In this section, we report the part of the code related to the integration.

Integration of Bugs with Commits

Here, we integrate the number of bugs with commits.

```
1.  #Merging by issue code
2.  merged_git_and_jira = pd.merge(git_issue_solved,jira_bugs, left_on='issue', right
    _on='issuekey')
3.
4.  #Grouping bugs
5.  merged_git_and_jira_grouped = merged_git_and_jira.groupby(["filename"]).agg({'iss
    ue': 'count'}).reset_index()
```

Integration of Bugs with Code Quality Measures

Here, we integrate the number of bugs with code quality measures.

```
1.  #Merging by file name, considering files where metric values exist
2.  integrated_bugs_metrics =
3.  pd.merge(merged_git_and_jira_grouped, s_metric, left_on='filename',
4.  right_on = 'filename', how = 'right').fillna({'number_of_bugs':0}).dropna()
```

Integration of Bugs with Code Quality Rule Violations

Here, we integrate the number of bugs with code quality rule violations.

```
1.  #Merging by file name
2.  integrated_bugs_violations =
3.  pd.merge(merged_git_and_jira_grouped, sq_violations, left_on='filename',
4.  right_on = 'filename', how = 'right').fillna({'number_of_bugs':0}).dropna()
```

7.3 Analysis

To check the hypotheses, we calculated Spearman correlation coefficients based on integrated data sets.

Figure 15 represents the correlation coefficients between number of occurred bugs and code quality measures of the files.

The row shows the number of bugs occurred in the file; each column corresponds to the one code quality metric (see an explanation of metrics in Appendix IV).

The results show only weak correlations between the items. We can see that the number of bugs has a weak positive correlation with the number of functions, lines, and code lines in the files. There are very weak correlations with other metrics.

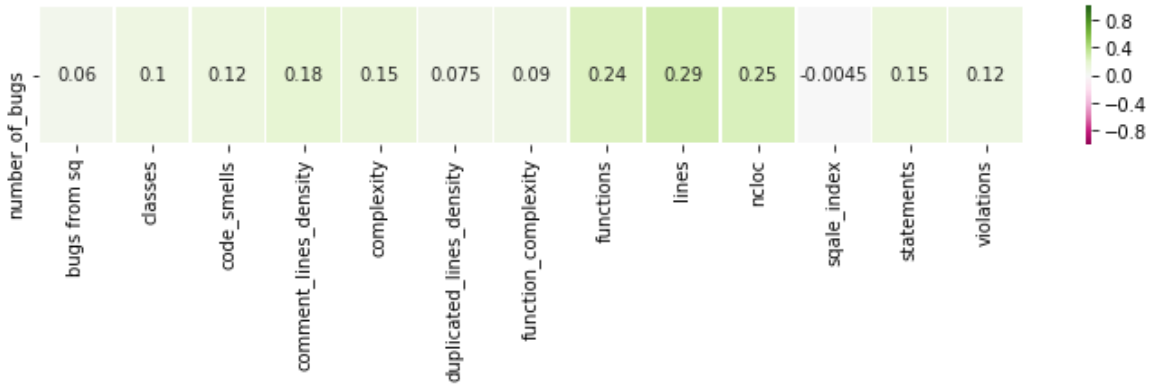


Figure 15. Spearman correlation coefficients between the number of bugs and code quality measures.

Figure 16 represents the correlation coefficients between number of occurred bugs and number of violations. The row shows the number of bugs occurred in the file; the column corresponds to the number of violations.

Figure 16. Spearman correlation between number of bugs and number of violations.

Surprisingly, the result shows the moderate negative correlation between the data sets, which is contradictory to our hypothesis. Further research is needed to identify the reason.

8 Discussion

This chapter reports the limitations and lessons learned from the research.

As a result of this master thesis, we fully answered constructed Q1 and Q2:

- Q1. How can we gather runtime data to monitor external quality?
- Q2. How can we visualize quality problems to take actions?

We have created an extension to Q-Rapids connectors to gather software runtime data into one tool. Based on the collected data we have created the dashboards to help to make a decision on sprint planning. The created dashboards were evaluated by the DD team.

We conducted a tool evaluation session with the DD project members. The majority of respondents felt that the tool is useful for their work and would help to investigate problems in the software. There were some limitations on easiness in using the tool, the reasons for what can be lack of provided tutorials and a short period of time to explore the tool. In our opinion, this approach could be used by practitioners in identifying problems of software.

We partially answer Q3:

- How can we integrate runtime data with development data?

Based on the integration with the development data sets (commits, code quality measures, violation rules), we have grouped relevant runtime data sets into the following three groups (see Figure 17):

- **Integrated** – data sets which are integrated with this thesis work;
- **Possible** – data sets which are realistic to integrate, after solving some issues;
- **Not possible** – data sets which currently we do not see to be possible to integrate.

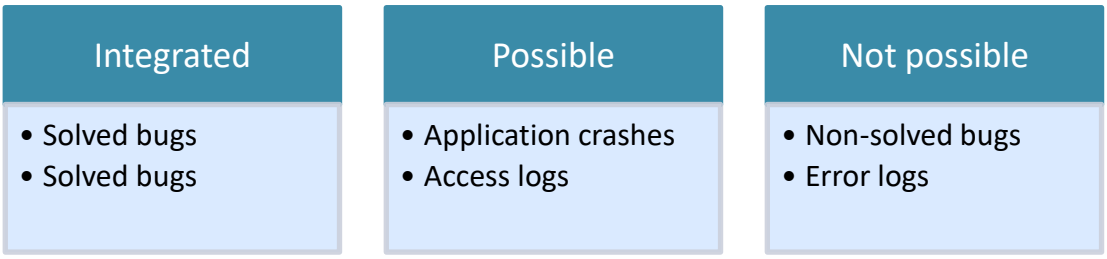


Figure 17. Three groups of relevant runtime data sets

In this thesis, we reported the integration of solved bugs with development data sets (internal quality aspects). One of the challenges was that first, we integrated bugs with commits, where we lost non-solved bugs. Then we could integrate the data set with code quality measures and violation rules. Currently, we do not see possibilities in the integration of non-solved bugs.

In the integration of application crashes with development data, we faced the problem of having data from different parts of the software. The code quality measures and rule violations of the application source code were not available yet. We believe by solving this issue we can integrate these data sets.

Access logs data set has entire request queries, which possibly can be integrated with internal quality aspects, by considering the class handled this request in the source code. We did not have access to a particular part of the source code.

Error logs contained only a general description of the issues. We could not find any direct or indirect connection with development data. In our case, there were no clear integration possibilities.

Only limited data sets of the available data were possible to integrate. We have integrated reported bugs with development data. Based on the integration we produced correlation analysis to test two hypotheses.

The constructed hypotheses were the following:

1. The number of bugs occurred in the file is positively correlated with the code quality measures of the file.
2. The number of bugs occurred in the file is positively correlated with the number of violation rules founded in this file.

Unfortunately, the results of the analysis were not encouraged. For hypothesis 1 we could identify a weak positive correlation of the number of bugs, with the number of lines, code lines, and functions. For hypothesis 2 we obtained the contradictory results, that the number of bugs has a moderate negative correlation with the number of violation rules. The reason for that needs to be investigated in further research.

Based on the research on Q3, we report the following challenges and lessons learned to consider in future research:

- Data access problems. As an example, during the research, we did not have access to the DD project android application source code.
- Data quality problems:
    - Data inconsistency: In different data sets, the artifacts had a different structure. Data preparation process would be easier if the data sets were constructed in a similar way. For example, the path to the files had a different pattern in each data set.
    - Data correctness: We faced the problem of having data sets corresponded to different time periods. We spent a lot of time on finding the correct data sets from the static code analysis tool. It is important to collect the data sets in the correct way beforehand.
    - Incomplete information: For example, only a few crashes contained code class name field, which would be useful in the integration.

Although the performance was not ideal, we believe that the lessons learned and implemented work can be reused in future research.

9 Conclusions and Future Work

This chapter reports a summary of the main contributions and future research.

The main contributions of the thesis were:

1. Created an extension to Q-Rapids connectors to collect available runtime data from the DD the project;
2. Creating dashboards to make decisions during sprint planning;
3. Applying the CRISP-DM method to the integration of software runtime and development data.

The provided connectors and integration scripts are reusable. However, the results of the integration were not very encouraging, we reported challenges from the integration of software runtime and development data can be used for further research.

Future work will be based on solving challenges in the integration of application crashes and access logs with development data sets (internal quality). We will conduct research on improving the quality of the data sets.

10 References

[1] S. Martinez-Fernandez, A. Jedlitschka, L. Guzman and A. M. Vollmer, "A Quality Model for Actionable Analytics in Rapid Software Development," *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, no. 732253, pp. 370-377, 2018.

[2] Y. Liang, Y. Zhang, H. Xiong and R. Sahoo, "Failure prediction in IBM BlueGene/L event logs," *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 583-588, 2007.

[3] F. Salfner and S. Tschirpke, Error Log Processing for Accurate Failure Prediction, 2008, p. .

[4] A. Metzger, E. Schmieders, O. Sammodi and K. Pohl, "Verification and testing at run-time for online quality prediction," *2012 1st International Workshop on European Software Services and Systems Research - Results and Challenges, S-Cube 2012 - Proceedings*, pp. 49-50, 2012.

[5] Q. Cao, Y. Qiao and Z. Lyu, "Machine learning to detect anomalies in web log analysis," *2017 3rd IEEE International Conference on Computer and Communications, ICCC 2017*, Vols. 2018-Janua, pp. 519-523, 2018.

[6] M. Du, F. Li, G. Zheng and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," *ACM SIGSAC Ccs'17*, pp. 1285-1298, 2017.

[7] P. Fraternali, M. Matera and A. Maurino, "Conceptual-level log analysis for the evaluation of Web application quality," *Proceedings - 1st Latin American Web Congress: Empowering our Web, LA-WEB 2003*, pp. 46-57, 2003.

[8] A. Karim, R. Salleh and M. K. Khan, "SMARTbot: A behavioral analysis framework augmented with machine learning to identify mobile botnet applications," *PLoS ONE*, vol. 11, no. 3, pp. 1-35, 2016.

[9] C. Couto, C. Silva, M. T. Valente, R. Bigonha and N. Anquetil, "Uncovering Causal Relationships between Software Metrics and Bugs," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012.

[10] N. Nagappan, T. Ball and A. Zeller, Mining metrics to predict component failures, vol. 2006, 2006, pp. 452-461.

[11] F. Lautenschlager and M. Ciolkowski, "Making Runtime Data Useful for Incident Diagnosis: An Experience Report: 19th International Conference, PROFES 2018, Wolfsburg, Germany, November 28–30, 2018, Proceedings," 2018, pp. 422-430.

[12] S. Martínez-Fernández, P. Jovanovic, X. Franch and A. Jedlitschka, Towards Automated Data Integration in Software Analytics, 2018, p. .

[13] M. Runeson Per and Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 12 2008.

[14] F. Elberzhager, M. Koch and B. Weitzel, "Towards a Digital Ecosystem for Rural Areas: Experiences from Three Years of Development: 19th International Conference, PROFES 2018, Wolfsburg, Germany, November 28–30, 2018, Proceedings," 2018, pp. 98-105.

[15] A. Mockus and D. Weiss, “Interval Quality: Relating Customer-Perceived Quality to Process Quality,” in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008.

[16] C. Shearer, The CRISP-DM model: the new blueprint for data mining, vol. 5, 2000, pp. 13-22.

[17] “HockeyApp API,” [Online]. Available: https://support.hockeyapp.net/kb/api/api-crashes.

[18] “Amazon Cloudwatch API,” [Online]. Available: https://www.logicmonitor.com/monitoring/amazon-cloudwatch/.

[19] V. Venkatesh and H. Bala, Technology Acceptance Model 3 and a Research Agenda on Interventions, vol. 39, 2008, pp. 273-315.

[20] K. Petersen, S. Vakkalanka and L. Kuzniarz, “Guidelines for conducting systematic mapping studies in software engineering: An update,” *Information and Software Technology*, vol. 64, pp. 1-18, 2015.

[21] X. Franch, C. Ayala, L. Lopez, S. Martínez-Fernández, P. Rodríguez, C. Gómez, A. Jedlitschka, M. Oivo, J. Partanen, T. Räty and V. Rytivaara, Data-Driven Requirements Engineering in Agile Projects: The Q-Rapids Approach, 2017, p. .

[22] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou and S. Pasupathy, “SherLog: error diagnosis by connecting clues from run-time logs,” *International Conference on Architectural Support for Programming Languages and Operating Systems (APLO)*, pp. 143-154, 2010.

[23] M. Wen, R. Wu and S.-C. Cheung, “Locus: locating bugs from software changes,” *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pp. 262-273, 2016.

[24] S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit and A. Trendowicz, “Operationalised product quality models and assessment: The Quamoco approach,” *Information and Software Technology*, vol. 62, no. 1, pp. 101-123, 2015.

[25] P. E. Strandberg, W. Afzal and D. Sundmark, “Decision Making and Visualizations Based on Test Results,” in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2018.

[26] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams and A. E. Hassan, “Understanding the Impact of Code and Process Metrics on Post-release Defects: A Case Study on the Eclipse Project,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2010.

[27] B. Russo, G. Succi and W. Pedrycz, “Mining system logs to learn error predictors: a case study of a telemetry system,” *Empirical Software Engineering*, vol. 20, no. 4, pp. 879-927, 2015.

[28] Meiliana, S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman and B. Soewito, “Software metrics for fault prediction using machine learning approaches: A literature review with PROMISE repository dataset,” in *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, 2017.

[29] L. Mariani, “Behavior capture and test for verifying evolving component-based systems,” *Proceedings. 26th International Conference on Software Engineering*, no. October, pp. 78-80, 2004.

[30] C. Ioannou, A. Burattin and B. Weber, “Advanced Information Systems

Engineering," vol. 3084, 2004.

[31] H. Huijgens, D. Spadini, D. Stevens, N. Visser and A. van Deursen, "Software Analytics in Continuous Delivery: A Case Study on Success Factors," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2018.

[32] S. He, J. Zhu, P. He and M. R. Lyu, "Experience Report: System Log Analysis for Anomaly Detection," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 207-218, 2016.

[33] J. Ge, "Comparative Study on Defect Prediction Algorithms of Supervised Learning Software Based on Imbalanced Classification Data Sets," *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 399-406, 2018.

[34] D. Gadler, M. Mairegger, A. Janes and B. Russo, "Mining Logs to Model the Use of a System," *International Symposium on Empirical Software Engineering and Measurement*, Vols. 2017-Novem, pp. 334-343, 2017.

[35] A. Filieri, C. Ghezzi and G. Tamburrelli, "Run-time efficient probabilistic model checking," *Proceedings of the 33rd International Conference on Software Engineering*, no. May, pp. 341-350, 2011.

[36] T. Dey and A. Mockus, "Modeling Relationship Between Post-Release Faults and Usage in Mobile Software," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, New York, NY, USA, 2018.

[37] A. R. Contreras and K. Mahbub, "MORPED: Monitor rules for proactive error detection based on run-time and historical data," *5th International Conference on the Applications of Digital Information and Web Technologies, ICADIWT 2014*, pp. 28-35, 2014.

[38] J. Cito, "Developer targeted analytics: supporting software development decisions with runtime information," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016.

[39] J. Cito, F. Oliveira, P. Leitner, P. Nagpurkar and H. C. Gall, "Context-based Analytics: Establishing Explicit Links Between Runtime Traces and Source Code," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, Piscataway, NJ, USA, 2017.

Appendix

I. Source Code

Code to parse from JSON to CSV format

```
#This code parse from json format to csv
########################################
#Import libraries
import json
import pandas as pd
from os import listdir

#Assign pathes to directories
input_dir = "../data/JSON/DD_last/"
output_dir = "../data/CSV/DD_last/"


def json_to_csv(input_path, output_path):
    data = []
    with open(input_path, 'r', encoding="utf-8") as input_file:
        for line in input_file:
            da-
ta.append(json.loads(line.encode('ascii', 'ignore'), )['_source'])
        data_df = pd.DataFrame(data)

    with open(output_path, 'w') as output_file:
        data_df.to_csv(output_file, index=False)
    return data_df


if __name__ == "__main__":
    filenames = listdir(input_dir)

    for file in filenames:
        if file.replace("json", "csv") not in listdir(output_dir):
            print(file + " - started")
            json_to_csv(input_dir + file, output_dir + file.replace("json", "csv"
))
            print("Done")
```

Connector to collect logs from Amazon CloudWatch

```
import os
import subprocess
import re
import io

a = os.system('aws logs describe-log-groups')
result = subprocess.check_output('aws logs describe-log-groups', shell=True)
log_group_names = re.findall(r'(?<=log-group:)(.*)(?=:)', result)
log_stream_names = dict()
for group in log_group_names:
    try:
        result = subprocess.check_output('aws logs describe-log-streams --log-
group-name ' + group, shell=True)
        log_stream_names[group] = list(set(re.findall(r'i-\w+', result)))
    except e as Exception:
        print(e)

```

```
17.
18. for group in log_group_names:
19.     dir_name = 'log_data' + re.sub("/", "_.", group)
20.     if not os.path.exists(dir_name):
21.         os.mkdir(dir_name)
22.     for stream in log_stream_names[group]:
23.         try:
24.             result = subprocess.check_output('aws logs get-log-events --log-
    group-name {0} --log-stream-name {1}'.format(group, stream), shell=True)
25.             #print(result)
26.             with open(dir_name + '/' + stream + '.log', 'a+') as out_file:
27.                 out_file.write(result)
28.                 print stream
29.         except Exception as e:
30.             print e
```

Connector to collect crashes from HockeyApp

```
1.  #Import libraries
2.  import os
3.  import subprocess
4.  import re
5.  import json
6.  import requests
7.  from elasticsearch import Elasticsearch
8.  import time
9.
10. #Configure the elasticsearch host ip and port
11. try:
12.     es = Elasticsearch([{'host': 'ip', 'port': port}])
13. except Exception as e:
14.     print(e)
15.
16. crash_groups_id_list=list()
17. crash_reasons = json.loads(subprocess.check_output('curl -H "X-
    HockeyAppToken: key
    " https://rink.hockeyapp.net/api/2/apps/applicationkey/crash_reasons', shell=True
    ))
18. for i in range(1,crash_reasons['total_pages']+1):
19.     crash_reasons = json.loads(subprocess.check_output('curl -H "X-
    HockeyAppToken: key
    " https://rink.hockeyapp.net/api/2/apps/applicationkey/crash_reasons?page={0}'.fo
    rmat(i), shell=True))
20.     crash_groups_id_list = crash_groups_id_list + [crash_group['id'] for crash_gr
    oup in crash_reasons['crash_reasons']]
21. crash_list = list()
22.
23.
24. i = 0
25. for _id in crash_groups_id_list:
26.     try:
27.         for crashes in json.loads(subprocess.check_output('curl -H "X-
    HockeyAppToken: key " https://rink.hockeyapp.net/api/2/apps/ applicationkey
    /crash_reasons/{0}?per_page=100'.format(_id), shell=True))['crashes']:
28.             try:
29.                 es.index(index='hockeyapp1', doc_type='log', id=i, body=json.load
    s(subprocess.check_output('curl -H "X-HockeyAppToken: key of the project
    " https://rink.hockeyapp.net/api/2/apps/applicationkey/crashes/{0}?format=json'.f
    ormat(crashes['id']), shell=True)))
30.                 i+=1
31.                 time.sleep(3)
32.             except Exception as e:
33.                 print(e)
```

```
    except Exception as e:
        print(e)
```

Logstash configuration file

```
input {
    file{
            path => ["pathtologfiles/*.log" ]
        start_position => "beginning"
            }
}

filter {
    grok{
        match => { "message" => "%{COMBINEDAPACHELOG}"}
            }

    date{

        match => ["timestamp", "dd/MMM/yyyy:HH:mm:ss Z"]
            }
}

output {
    elasticsearch {hosts => ["host"]
    index => "name"
    }

stdout { codec => rubydebug}
}
```

II. Questionnaire

Actionable dashboards on the software runtime data.

Evaluation of the dashboards created based on Digitale Dörfer project runtime data.

* Required

1. **Using the system would improve my performance in my job.** *
   Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

2. **Using the system in my job would increase my productivity.** *
   Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

3. **Using the system would enhance my effectiveness in my job.** *
   Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

4. **I find the system to be useful in my job.** *
   Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

5. **My interaction with the system is clear and understandable.** *
   Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

6. **Interacting with the system does not require a lot of my mental effort.** *
   Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

7. **I find the system to be easy to use.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

8. **I find it easy to get the system to do what I want it to do.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

9. **In my job, usage of the system is important.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

10. **In my job, usage of the system is relevant.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

11. **The use of the system is pertinent to my various job-related tasks.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

12. **The quality of the output I get from the system is high.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

13. **I have no problem with the quality of the system's output.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

14. **I rate the results from the system to be excellent.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

15. **Assuming I had access to the system, I intend to use it.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

16. **Given that I had access to the system, I predict that I would use it.** *
Mark only one oval.

	1	2	3	4	5	6	7	
Strongly disagree	◯	◯	◯	◯	◯	◯	◯	Strongly agree

17. **I plan to use the system in the next <n> months.** *

18. **Which tools are used currently to analyse software runtime data?**

19. **How can dashboards be improved?**

20. **Comments**

21. **Your current role in the project.**

III. Notebook

Integration of Bugs with Code Quality Measures

Importing Libraries

In [1]:

```
import pandas as pd
import numpy as np
import re
import matplotlib.pyplot as plt
import seaborn as sns
from termcolor import colored
from IPython.display import Markdown, display
import datetime
```

Loading Relevant Data Sets

In [19]:

```
#Loading Jira Sprint issue
jira_data = pd.read_csv('                    /jira_DD.csv')
#Loading Sonarqube measures
sonarqube_m_data = pd.read_csv('                    /sonarqube_mes_DD.csv')
#Loading Git
git_data = pd.read_csv('                    /git_DD.csv')
#Loading Sonarqube issues
#sonarqube_issue_data = pd.read_csv('                    /sonarqube.issues.dd.csv')
```

Extracting Relevant Features and Cleaning

Objective of this section is identification of specific data that are potentially relevant for integration and preparing it for possible integration.

BUGS

In [20]:

```
#Cleaning from duplicates
jira_bugs = jira_data.drop_duplicates().drop(['assignee', 'creator', 'reporter'], axis=1)

#Printing columns from data set JIRA
print(colored("\nExisting fields in JIRA data set:\t\t\t  ", "blue") + colored("Example:", "blue"))

#Print example
print(jira_bugs.iloc[0,:])

print(colored("\nNumber of data points:", "green"))
print(len(jira_data))

#List of relevant feature names
relevant_features = ['created', 'issuekey', 'issuetype']

print(colored("\nRelevant features:", "blue"))
print(colored(relevant_features, color= "red"))



#Extracting relevant features
jira_bugs = jira_bugs[relevant_features]

#Cleaning from issues remained for several sprints
jira_bugs = jira_bugs.drop_duplicates()
```

```
#Filtering only issues which are 'Bug's
jira_bugs = jira_bugs[(jira_bugs.issuetype == 'Bug')]

print(colored("\nNumber of bugs:", "green"))
print(len(jira_bugs))



#Creating field which contains only date
jira_bugs = jira_bugs.assign(date = jira_bugs['created'].apply(lambda x: pd.to_datetime(x).date()))

#Print date range
print(colored("\nDate range of bugs:", "green"))
print(jira_bugs.date.describe())
print(jira_bugs.date.min())
print(jira_bugs.date.max())
```

```
Existing fields in JIRA data set:     Example:
aggregatetimeestimate                                                NaN
aggregatetimespent                                                   NaN
category                                                             NaN
component                                                            [redacted]
created                                          2017-07-06T17:12:41.000+0200
description              As an[Author|https://[redacted]...
duedate                                                              NaN
issueid                                                            17851
issuekey                                                          DD-608
issuetype                                                          Story
jiraIssueApi               https://[redacted]
jiraUrl                                        https://[redacted]/jira
lastViewed                                                           NaN
platform                                                             NaN
priority                                                          Medium
projectKey                                                            DD
projectName                                                Digitale Drfer
resolution                                                         Done
resolutiondate                                   2017-08-02T10:21:26.000+0200
sprintid                                                             NaN
sprintname                                                           NaN
sprinttotal                                                          NaN
status                                                            Done
subtask                                                              NaN
summary                                                Publish news article
timeestimate                                                         NaN
timespent                                                            NaN
updated                                          2017-08-02T10:21:26.000+0200
Name: 0, dtype: object

Number of data points:
5213

Relevant features:
['created', 'issuekey', 'issuetype']

Number of bugs:
1172

Date range of bugs:
count           1172
unique           289
top       2017-11-03
freq              30
Name: date, dtype: object
2017-03-22
2019-01-25
```

Quality of the data set:

- Date granularity: day and time, when the issue is reported
- Date range: 09.03.2017 – 01.25.2019
- Artifact granularity: issue key and sprint name
- Completeness of data:
  - as data does not contain filename information, in case of integrating it with Git data, we will lose not yet solved

issues

- 'sprintname' field has missing values

COMMITS

In [21]:

```
#Printing columns from data set GIT
print(colored("Existing fields in GIT data set:\t\t\t  ", "blue") + colored("Example:", "blue"))


#Print example
print(git_data.iloc[0,:].drop(['author', 'hash']))

print(colored("\nNumber of commits:", "green"))
print(len(git_data))

#List of relevant feature names
relevant_features = ['date', 'filename', 'issues']

print(colored("\nRelevant features:", "blue"))
print(colored(relevant_features, color= "red"))


#Extracting only changes which solved an issue and needed features
git_issue_solved = git_data[git_data.issues != '[]'][relevant_features]

#Deleting duplicates
git_issue_solved = git_issue_solved.drop_duplicates()

#Changing format of column 'issues' from 'str' to 'list'
git_issue_solved = git_issue_solved.assign(issues = git_issue_solved.issues.apply(lambda x: eval(x)
))

#Splitting commits which solved several issues
git_issue_solved = git_issue_solved.issues.apply(pd.Series).merge(git_issue_solved, left_index=True
, right_index=True).drop('issues', axis = 1).melt(id_vars = ['date', 'filename'], value_name = 'iss
ue').dropna().drop('variable', axis = 1)


print(colored("\nNumber of commits which solved issues:", "green"))
print(len(git_issue_solved))



#Creating field which contains only date
git_issue_solved = git_issue_solved.assign(date1 = git_issue_solved['date'].apply(lambda x: pd.to_d
atetime(x).date()))

#Print date range
print(colored("\nDate range of solved issues:", "green"))
print(git_issue_solved.date1.describe())
print(git_issue_solved.date1.min())
print(git_issue_solved.date1.max())
```

```
Existing fields in GIT data set:     Example:
added                                                1
date                                 2019-01-23T10:46:16.000Z
deleted                                              0
filename    shared-api/src/main/java/[redacted]..
issues                                          ['DD-[redacted]']
message     DD-[redacted] Geo Area Type available for clients\r\...
Name: 0, dtype: object

Number of commits:
31413

Relevant features:
['date', 'filename', 'issues']

Number of commits which solved issues:
9447

Date range of solved issues:
```

```
count           9447
unique           262
top       2017-05-26
freq            1776
Name: date1, dtype: object
2017-04-05
2019-01-23
```

Quality of the data set:

- Amount of data: 22,907 hits
- Date granularity: date and time when changes were committed
- Date range: 24.08.2015 – 01.14.2019
- Artifact granularity: each changed file
- Completeness of data: we consider only changes, which are issue fixing, in that case, the number of remained hits is equal to 5,170

CODE QUALITY MEASURES

In [23]:

```
#Print all quality measures
print(colored("Metrics: ", "red") + str(set(sonarqube_m_data.metric)))

print(colored("\nInitial Sonarqube measures data:\t\t\t  ", "blue") + colored("Example:", "blue"))
#Printing an example
print(sonarqube_m_data.iloc[6].drop(['bcId', 'Id']))


#Creating field which contains only date
sonarqube_m_data = sonarqube_m_data.assign(date = sonarqube_m_data['snapshotDate'].apply(lambda x:
pd.to_datetime(x).date()))

#Print date range
print(colored("\nDate range of sonarqube measures:", "green"))
print(sonarqube_m_data.date.describe())
print(sonarqube_m_data.date.min())
print(sonarqube_m_data.date.max())



#Filtering only measures per "file"s and averaging values per each metric
sonarqube_metric = pd.DataFrame(sonarqube_m_data[(sonarqube_m_data.qualifier == 'FIL')].groupby(['
key', 'metric']).floatvalue.mean().unstack().reset_index())



#Structuring filename feature
sonarqube_metric = sonarqube_metric.assign(filename = sonarqube_metric.key.apply(lambda x: (re.sub(
'.*:', '', x)))).drop('key', axis=1)

print(colored("\nSonarqube measures data set after cleaning:\t\t\t  ", "red") + colored("Example:"
, "red"))
#Printing an example
print(sonarqube_metric.iloc[6])


print(colored("\nNumber of files which have code quality measures:", "green"))
print(len((sonarqube_metric)))
```

```
Metrics: {'directories', 'statements', 'bugs', 'duplicated_lines_density',
'comment_lines_density', 'code_smells', 'lines', 'classes', 'files', 'sqale_index', 'violations',
'function_complexity', 'complexity', 'ncloc', 'functions'}

Initial Sonarqube measures data:     Example:
bcKey                                [redacted]:platform-parent
bcName                                          [_parent] _platform
bcQualifier                                                    TRK
floatvalue                                                       0
key                                  [redacted]:shared-service:...
```

```
language                                                    java
metric                                                     bugs
name                                  LoginDataInvalidException.java
path                                                /service/...
qualifier                                                   FIL
snapshotDate                                         2018-01-28
sonarUrl                                                    
value                                                       0
date                                                 2018-01-28
Name: 6, dtype: object

Date range of sonarqube measures:
count        2245970
unique           112
top       2019-01-24
freq           23056
Name: date, dtype: object
2018-01-28
2019-02-10

Sonarqube measures data set after cleaning:     Example:
metric
bugs                                                          0
classes                                                       1
code_smells                                                   1
comment_lines_density                                       2.9
complexity                                                    4
duplicated_lines_density                                   29.1
files                                                         1
function_complexity                                         1.3
functions                                                     3
lines                                                        55
ncloc                                                        34
sqale_index                                                  20
statements                                                    8
violations                                                    1
filename                                          /api/admi...
Name: 6, dtype: object

Number of files which have code quality measures:
2507
```

Quality of the data set:

- Amount of data: 2,245,970 hits
- Date granularity: snapshot date – every day, once
- Date range: 26.01.2018 – 10.02.2019
- Artifact granularity: per file, directory, unit test, and module
- Completeness of data:
  - missing data between 21.02.2018 – 02.03.2018
  - missing 'values' in metric measures

Data Integration

To integrate files, which contained causes of bugs, with code quality measures of this file we have to integrate: Jira sprint issues, Sonarqube measures, and Git data sets. As we do not have the filename in Jira issues, Jira data can be integrated with Git data by considering the common issue id.

As a second step joined data (Jira and Git) can be integrated with the Sonarqube measures data set by 'filename' from Git and 'key' from Sonarqube measures.

Integration of Bugs with Commits

In [24]:

```
print("Number of bugs: {0}".format(len(jira_bugs)))
print("Number of commits to solve an issue: {0}".format(len(git_issue_solved)))

merged_git_and_jira = pd.merge(git_issue_solved,jira_bugs, left_on='issue', right_on='issuekey')

#Grouping bugs
```

```
#Grouping bugs
merged_git_and_jira_grouped = merged_git_and_jira.groupby(["filename"]).agg({'issue': 'count'}).re
set_index()

#Changing column names
merged_git_and_jira_grouped.columns = ['filename', 'number_of_bugs']

print("Number of files with bugs: {0}".format(len(merged_git_and_jira_grouped)))

#Structuring filename format to correlate
merged_git_and_jira_grouped_hyp1 = merged_git_and_jira_grouped.assign(filename =
merged_git_and_jira_grouped.filename.apply(lambda x: x[x.find('/')+1:]))
```

```
Number of bugs: 1172
Number of commits to solve an issue: 9447
Number of files with bugs: 473
```

Integration of Bugs with Sonarqube measures

In [25]:

```
#Merging by file name, considering files where metric values exist
integrated = pd.merge(merged_git_and_jira_grouped_hyp1, sonarqube_metric, left_on='filename',
right_on = 'filename', how = 'right').fillna({'number_of_bugs':0}).dropna()

#Removing file name from data frame
integrated = integrated.drop(['filename', 'files'], axis=1).dropna()
```

Correlation analysis

Correlation visualization function

In [26]:

```
def plot_fig(data, title, method):
    fig = plt.figure(figsize=[10,4])
    ax = fig.add_subplot(111)
    print(colored("Features and correlation coefficients with number of bugs:", "red"))
    print(data.corr(method = method).iloc[0,1:])
    sns.heatmap(data.corr(method = method).iloc[0:1,1:], annot=True, vmin=-1, vmax=1, linewidths=.5
, ax=ax,cmap="PiYG")
    ax.set_title(title, pad=50)
    ticks = np.arange(0,len(data.columns)-1,1)
    #ax.set_xticks(ticks)
    plt.xticks(rotation=90)
    #ax.set_yticks(ticks)
    ax.set_xticklabels(['bugs from sq', 'classes', 'code_smells',
       'comment_lines_density', 'complexity', 'duplicated_lines_density',
       'function_complexity', 'functions', 'lines', 'ncloc', 'sqale_index',
       'statements', 'violations'])
    #ax.set_yticklabels(data.columns)

    plt.tight_layout()
    plt.show()
```

Hypothesis -

The number of bugs occurred in the file is positively correlated with the code quality measures of the file.

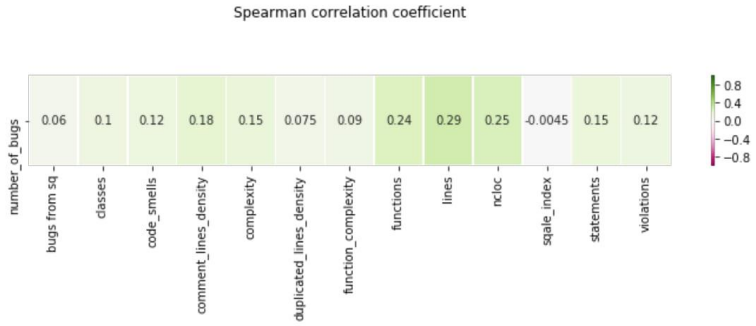
To test the hypothesis we will use Spearman correlation coefficient. It show monotonic dependencies between variables.

In [27]:

```
plot_fig(integrated, 'Spearman correlation coefficient', 'spearman')
```

```
Features and correlation coefficients with number of bugs:
```

```
bugs                          0.059846
classes                       0.104260
code_smells                   0.119507
comment_lines_density         0.180530
complexity                    0.150164
duplicated_lines_density      0.075047
function_complexity           0.090137
functions                     0.237360
lines                         0.285409
ncloc                         0.254865
sqale_index                  -0.004452
statements                    0.146721
violations                    0.115061
Name: number_of_bugs, dtype: float64
```



The strength of the correlation:

.00-.19 “very weak”

.20-.39 “weak”

.40-.59 “moderate”

.60-.79 “strong”

.80-1.0 “very strong”

http://www.statstutor.ac.uk/resources/uploaded/spearmans.pdf

Results:

Number of bugs has weak positive correlation with number of lines, number of code lines, number of functions of the file.

Number of bugs has very weak positive correlation with comment_lines_density, complexity, number of statements, number of violations, number of classes and code_smells.

Number of bugs has correlation close to 0 with bugs from sonarqube, duplicated lines density, function_complexity and sqale_index.

IV. Code Quality Metrics Measures

Metric name	Explanation
classes	number of classes (including nested classes, interfaces, enums and annotations)
ncloc	number of physical lines which contains at least one character (no comment)
lines	number of all physical lines
functions	number of functions
comment_lines	number of lines containing either comment or commented-out code
comment_lines_density	comment_lines / (ncloc + comment_lines) * 100
duplicated_lines_density	duplicated_lines / lines * 100
duplicated_lines	number of lines involved in duplications
complexity	cyclomatic complexity calculate based on the number of paths through the code
function_complexity	complexity average by function
violations	total number of issues

Source: *https://docs.sonarqube.org/7.4/user-guide/metric-definitions/*

V. Acknowledgments

This research was made partially possible by Erasmus+ traineeship grant.

I would like to express my deepest appreciation to my supervisors Silverio Martínez-Fernández and Dietmar Pfahl for their help, time, ideas and encourage.

I am thankful to Andreas Jedlitschka to making possible carrying my master thesis at the Fraunhofer IESE Data Engineering department and his support. Special thanks to Adam Trendowicz for his competent guidance to my master thesis and Axel Wickenkamp for help in gathering data for my research.

I would like to express my thanks to Balthasar Weitzel and all Digitale Dörfer team for providing me access to their data, active participation in the evaluation process and useful comments.

I am extremely grateful to my family for their understanding, love and endless support.

VI. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Aytaj Aghabayli

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Software Runtime Data: Visualization and Integration with Development Data – A Case Study,

(*title of thesis*)

supervised by Dr. Dietmar Pfahl and Dr. Silverio Martínez-Fernández.

(*supervisor's name*)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Aytaj Aghabayli

20/05/2019