



Kober Franz Josef, BSc

# Using Git as distributed CAS Database for Vehicle Component Certification

## Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Engineering and Management

submitted to

**Graz University of Technology**

Supervisor

Ass.-Prof. Dipl.-Ing. Dr.techn. Christian Steger

Co-Supervisor

Dipl.-Ing. Dr.techn. Markus, Quaritsch

Institute of Technical Informatics

Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH. Kay Uwe Römer

St. Ruprecht an der Raab, September 2019

## **Affidavit**

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

---

Date

---

Signature

# Abstract

Die Europäische Kommission hat ein standardisiertes Zertifizierungsverfahren für schwere Nutzfahrzeuge und deren Fahrzeugkomponenten definiert um mittels einer Simulationssoftware den Kraftstoffverbrauch von schweren Nutzfahrzeugen festzustellen. Die Simulationssoftware VECTO (Vehicle Energy Consumption Calculation Tool) nutzt die im Zertifizierungsverfahren gewonnenen Messdaten, um den Energie- und Kraftstoffverbrauch sowie CO<sub>2</sub> Emissionen von schweren Nutzfahrzeugen zu berechnen.

Diese Masterarbeit beschreibt die Entwicklung eines Prototypen, zur Verwaltung der Daten welche während eines Zertifizierungsverfahren von Fahrzeugkomponenten entstehen. Zur Verwaltung der Daten wurde dabei nicht auf ein bestehendes Datenbanksystem zurückgegriffen, sondern auf das verteilte Versionsverwaltungssystem Git. Ausgehend von den gegebenen Anforderungen und Daten wurde in der Entwurfsphase ein Datenstrukturierungskonzept für Git entwickelt. Durch dieses Datenstrukturierungskonzept und den Einsatz der Tag Funktion von Git wurden die Metainformationen Herstellername, Modellname, Zertifizierungsnummer und Zertifizierungsdatum hinzugefügt.

Durch diese Metainformationen werden wieder Informationen über den Inhalt einer Datei zurückgewonnen, welche durch den Speicherprozess in das Content-Addressable Storage System von Git verloren wurde. Weiters ermöglicht die Datenstrukturierung ein einfacheres und effizientes Suchen der Daten innerhalb des Git Systems.

# Abstract

The European Commission has defined a standardised certification procedure for heavy duty vehicles and their vehicle components in order to determine the fuel consumption of heavy duty vehicles using a simulation software. The simulation software VECTO (Vehicle Energy Consumption Calculation Tool) uses the measurement data obtained in the certification process to calculate the energy and fuel consumption as well as CO<sub>2</sub> emissions of heavy commercial vehicles.

This master thesis describes the development of a prototype to manage the data generated during the certification process of vehicle components. Instead of using an existing database management system to manage the data, the distributed version control system Git was chosen. Based on the given requirements and data a data structuring concept for Git was developed in the design phase. Through this data structuring concept and the use of the tag function of Git, the meta information manufacturer name, model name, certification number and certification date were added.

This meta-informations recovers information about the content of a file that has been lost by the storage process into the Content-Addressable Storage of Git. Furthermore, the data structuring concept makes it easier and more efficient to search for data within the Git system.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	3
1.2 Thesis Goals . . . . .	3
1.3 Thesis Organization . . . . .	3
<b>2 Vehicle Energy Consumption Calculation Tool</b>	<b>4</b>
2.1 Regulation of the European Commission . . . . .	4
2.2 Possible options of HDV CO <sub>2</sub> certification . . . . .	5
2.3 What is VECTO . . . . .	6
2.4 Certification Process Overview with VECTO . . . . .	6
2.4.1 Measurement of vehicle components . . . . .	7
2.4.2 Evaluation Tools . . . . .	9
2.4.3 Data around VECTO . . . . .	9
2.4.4 VECTO Simulation Tool . . . . .	10
<b>3 Background</b>	<b>11</b>
3.1 Database . . . . .	11
3.1.1 Relational Databases . . . . .	11
3.1.2 NoSQL/Non-Relational Databases . . . . .	14
3.2 CAP Theorem . . . . .	16
3.2.1 ACID Consistency Model . . . . .	18
3.2.2 BASE Consistency Model . . . . .	19
3.3 Content-Addressed Storage . . . . .	20
<b>4 Version Control System</b>	<b>21</b>
4.1 What is a Version Control System? . . . . .	21

## Contents

4.2	Types of Version Control Systems . . . . .	22
4.2.1	Local Version Control System . . . . .	22
4.2.2	Centralized Version Control System . . . . .	22
4.2.3	Distributed Version Control System . . . . .	23
4.3	Git . . . . .	24
4.3.1	Git History . . . . .	24
4.3.2	Git Key Concepts . . . . .	25
<b>5</b>	<b>Design</b>	<b>35</b>
5.1	Use cases around VECTO . . . . .	35
5.2	Requirements for certification of component . . . . .	36
5.3	Design and Data Partitioning . . . . .	38
5.4	Search behaviour . . . . .	43
5.5	Transfer behaviour . . . . .	44
5.6	Structure of the certification number . . . . .	45
5.7	Use cases . . . . .	46
5.7.1	Component data of a new certificated component should be stored in the storage backend . . . . .	46
5.7.2	Certificate of a component shall be stored in the storage backend . . . . .	48
5.7.3	Measured data of a component shall be stored in the storage backend . . . . .	50
5.7.4	User-defined data for a component shall be stored in the storage backend . . . . .	53
5.7.5	Standard values for a component shall be stored in the storage backend . . . . .	54
5.7.6	All component-related data (component data, standard values, certificate, measured data, user-defined data) can be searched via the manufacturer name, model name, and date . . . . .	55
5.7.7	Component data, certificate, measured data can be searched via the certification number and the Git identifier of the component data . . . . .	56
5.8	GUI Design . . . . .	58
<b>6</b>	<b>Implementation</b>	<b>60</b>
6.1	Overview . . . . .	60

## Contents

6.2	Architecture . . . . .	61
6.2.1	Repository Management Implementation . . . . .	63
6.2.2	GUI . . . . .	64
<b>7</b>	<b>Conclusion</b>	<b>69</b>
	<b>Bibliography</b>	<b>70</b>

# List of Figures

1.1	VECTO Certification Process . . . . .	2
2.1	Certification Process Overview of VECTO . . . . .	7
3.1	Popularity division between Relation and Non-relation Databases	12
3.2	CAP Theorem . . . . .	16
4.1	Local version control system . . . . .	22
4.2	Centralized version control system . . . . .	23
4.3	Distributed version control system . . . . .	24
4.4	Example of Delta Storage System . . . . .	26
4.5	Example of Snapshot Storage System . . . . .	27
4.6	Git repository folder structure . . . . .	28
4.7	Git Object Model . . . . .	32
5.1	Organization and structuring of component data files in the Git repository . . . . .	42
5.2	GUI Design Main View . . . . .	58
5.3	GUI Design Save Component Data View . . . . .	58
5.4	GUI Design Search Component Data View . . . . .	59
6.1	VECTO GIT prototype architecture . . . . .	62
6.2	VECTO Git prototype save component data view . . . . .	66
6.3	VECTO Git prototype search component view . . . . .	66
6.4	VECTO Git prototype append component data view . . . . .	67
6.5	VECTO Git prototype detail component data view . . . . .	67
6.6	VECTO Git prototype export component data view . . . . .	68
6.7	VECTO Git prototype import component data view . . . . .	68



# 1 Introduction

In the year 2007, the European Union (EU) committed itself to a 20% reduction in greenhouse gas emissions based on 1990 greenhouse gas emissions, till the year 2020. In addition to its commitment to reduce greenhouse gas. The EU has also committed itself to increasing the share of renewable energy to up to 20% of total energy consumption, with a share of at least 10% in the transport sector and additionally to decrease the total energy consumption by 20% which should be achieved by the usage of energy-efficient technologies.

In order to meet these commitments to reduce the greenhouse gas emission by 20% until 2020, the EU created a two-fold legislative framework "EU 2020 Climate and Energy Package"[1] in 2009. Which contains the three key targets [1]

- *"20% cut in greenhouse gas emissions(from 1990)"*
- *"20% of EU energy from renewable energy sources"*
- *"20% improvement in energy efficiency efficiency"*

There are also further EU commitments to set greenhouse gas emission reduction goals after 2020, among others the long term goal to reduce transport emissions around 60% based on the value of 1990.

In order to achieve this long-term goal in the transport sector, a standardised measurement certification procedure for CO<sub>2</sub> emissions of Heavy Duty Vehicles (HDV) had to be defined. Due to the different applications of a heavy duty vehicle, where almost no two vehicles are the same, a simulation-based approach was chosen instead of the existing CO<sub>2</sub> certification procedure for passenger cars in order to keep effort and the certification costs within an acceptable range.

The development of the simulation based Vehicle Energy Consumption Calculation Tool (VECTO) began in 2009. The simulation model as well as

## 1 Introduction

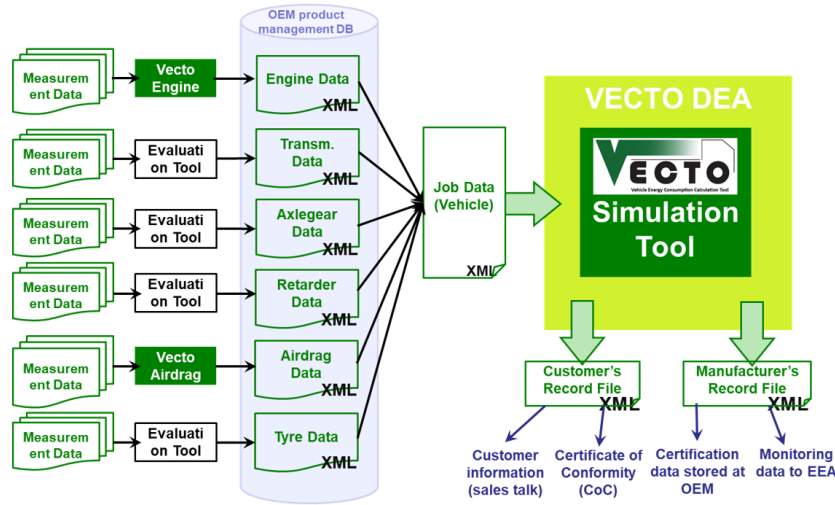


Figure 1.1: VECTO Certification Process taken from [2]

the software architecture is built as a component based model. The benefit of this architecture is to be as flexible as possible which enables to interchange as well as omit components if needed. The components Engine, Transmission, Axle Gear, Aerodynamic drag and Tyre listed in Figure 1.1 represents the vehicle parts which contributes to the vehicle's fuel consumption. The measurement data will be determined by standardised measurement methods. The measurement data which consists of several files represents the input data for the related VECTO component evaluation tool which generates the input file for the VECTO simulation tool. The above measurement data, component data and other VECTO simulation tool-related data such as certificate data and user-defined data should be collected and managed by a database that can be delivered together with the simulation tool.

## 1.1 Motivation

The motivation of this thesis is to repurpose the distributed version control system Git in way to use it as database tailored for component data as well as measurement data which can be used for the CO<sub>2</sub>-emission certification process of Heavy Duty Vehicles (HDV). This should enable to deliver a database in combination with the simulation tool. The database and its data should be easily accessible by the simulation software without any further configurations. This should avoid the need to set up an additional database management system to administrate the input data for the simulation.

## 1.2 Thesis Goals

The primary goal of this thesis is the development of a database prototype using the content addressable storage system of the distributed version control system Git as basis for the management of data used in the certification of heavy duty vehicles. The secondary goal is to use existing mechanisms and commands of Git to enable searchability of data within the database. And the third goal is the development of a transmission mechanism to transfer data from one database to another.

## 1.3 Thesis Organization

This thesis is organised into the following chapters. Chapter 2 gives an overview of the simulation tool VECTO and also an overview of the certification process of heavy duty vehicles and the resulting data. Chapter 3 provides background information about databases in general what types exists and what requirements they can meet. Chapter 4 provides a brief overview of the different types of version control systems and an insight into the distributed version control system Git. Chapter 5 deals with the design of the database prototype for storing the certification data. Chapter 6 captures the architecture and functionality of the implemented prototype.

## 2 Vehicle Energy Consumption Calculation Tool

This chapter gives a short overview of the simulation tool VECTO and describes which input data it requires and what it is used for. Also a short outlook onto the possible alternatives to measure CO<sub>2</sub>-emission of a Heavy Duty Vehicles (HDV) is given.

### 2.1 Regulation of the European Commission

In the year 2017, heavy duty vehicles, represented by trucks, buses and coaches, accounted for around 25% of total CO<sub>2</sub> emissions from road transport. Furthermore, an increase in CO<sub>2</sub> emissions in the coming years is expected.

Regarding the grim future of CO<sub>2</sub> emission increase the European Commission Regulation (EU) 2017/2400<sup>1</sup> concluded, which contains, among other things, the set target to reduce road transport CO<sub>2</sub> emissions by 60% until the year 2050.

---

<sup>1</sup>Regulation (EU) 2017/2400 <https://eur-lex.europa.eu/eli/reg/2017/2400/oj>

## 2.2 Possible options of HDV CO<sub>2</sub> certification

There are four possible concepts to determine the emission of a vehicle [3]:

- **Engine test**  
The engine test determines the CO<sub>2</sub> emission by the measurement on a test stand. Each engine will be operated on the basis of a standardised test cycle. This has the advantage that already standardised cycles such as the World Harmonized Transient Cycle (WHTC) can be used. The disadvantage is the lack of inclusion of the other consumers of the vehicle who also contribute indirectly to the increase in emissions.
- **Road load test and chassis dynamometer test**  
This option enables the possibility to measure the CO<sub>2</sub> emission of an entire vehicle. With the disadvantage that each vehicle should be tested separately for rolling resistance and air drag. In addition, this option is very cost-intensive due to the variety of possible configurations of a heavy duty vehicle.
- **On-road test**  
The on-road test options enables also to measure the CO<sub>2</sub> emission of an entire vehicle. With the disadvantage that measurements are difficult to compare because of the constantly changing traffic volume during each on-road test. In addition, this option is even more cost-intensive than the chassis dynamometer test option.
- **Component test plus vehicle simulation**  
The usage of a simulation tool resolves most of the disadvantages from the previously mentioned options. The tool enables to determine the CO<sub>2</sub> emission of an entire vehicle in any configuration and is at the same time more cost effective. With the disadvantage that the simulation tool has to be changed and updated regularly to cover all relevant technologies.

In order to support certification of an entire heavy-duty vehicle, which can consist of a large number of possible vehicle component combinations, and to keep the certification costs within an acceptable range, the "component test plus vehicle simulation" approach was chosen.

## 2.3 What is VECTO

The Vehicle Energy Consumption Calculation **TO**ol (VECTO) is a simulation software tool which calculates the fuel consumption that directly correlates to CO<sub>2</sub> emission (depending on the fuel type) of an entire heavy duty vehicle.

The development of VECTO started in course of the Regulation (EC) No 595/2009<sup>2</sup>. VECTO represents the basis of the EU Regulation 2017/2400 and serves as a calculation tool for fuel consumption and CO<sub>2</sub> emissions for most heavy duty vehicles sold from 1.1.2019 onwards.

## 2.4 Certification Process Overview with VECTO

The certification process of VECTO shown in the overview Figure 2.1 contains four essential parts that are necessary for the certification of a heavy duty vehicle, these are [3]:

- The measurement procedure of the components.
- The evaluation of the measurement results.
- The simulation itself with the VECTO simulation tool.
- Reporting of declared CO<sub>2</sub> figures to the European Environment Agency (EEA) and local authorities.

The vehicle certification process as shown in Figure 2.1 passes through various stages, which are described in detail below.

The certification process starts with the measurement process of the vehicle component under the control of the technical service from the Type Approval Authority (TAA). Depending on the component type, the resulting measurement data is used by the evaluation tool, the VECTO Engine Tool or the VECTO Air Drag Tool to generate the component data. A certain time after the component certification process, the TAA issues a certificate for the component and hands it over to the component manufacturer.

---

<sup>2</sup>Regulation (EC) No 595/2009 <http://data.europa.eu/eli/reg/2009/595/oj>

## 2.4 Certification Process Overview with VECTO

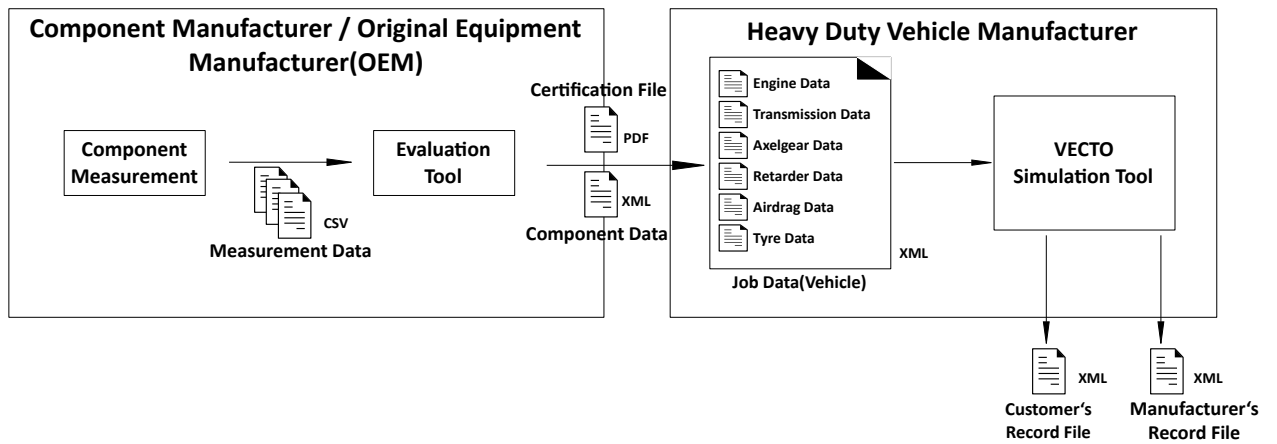


Figure 2.1: Certification Process Overview of VECTO

If a vehicle manufacturer buys a vehicle component, it receives the certificate file and the component file in addition to the component itself.

In order to certify the entire vehicle, the vehicle manufacturer selects each component data file of the components used in the vehicle and place them in the job data file.

With the job data file as input, the fuel consumption of the entire vehicle will be calculated with the VECTO simulation tool. After a successful simulation two XML files will be created the customer record file and the manufacturer record file. The customer record file is available to the buyer of the vehicle. The manufacturer record file must be stored by the vehicle manufacturer for at least 20 years and has to be available to the TAA and the European Commission upon request.

### 2.4.1 Measurement of vehicle components

For the main vehicle components which have an influence on fuel consumption a related measurement procedure was defined. These fuel consumption relevant vehicle components are [4, 3]:

## 2 Vehicle Energy Consumption Calculation Tool

- **Engine**

This measurement procedure determines the maximum performance, the necessary torque that is needed to drag the engine at a certain rotation speed and the fuel consumption of the engine when running at defined operation points. The engine measurement takes place on the test stand.

- **Transmission (plus torque converter in the case of an automatic transmission)**

The measurement procedure determines the torque loss for each gear at the given transmission input speed and at the given input torque.

- **Axles and other torque transferring components**

The measurement procedure determines the loss of torque by the given input-torque given to the axle. Measurements can be omitted and standard values as defined in Regulation 2017/2400 are used instead.

- **Aerodynamic drag**

This measurement procedure determines the air resistance which will be applied to the front area of the vehicle. The procedure takes place on a test track with a standard body and/or trailer. During the test drive on track the torque of the wheels, the vehicle velocity, the actual air flow velocity and the air flow direction are measured.

- **Tires**

The measurement procedure determine the losses caused by tires with a drum test according to the Regulation (EC) No 1222/2009<sup>3</sup>.

- **Vehicle auxiliaries**

The category auxiliaries includes engine cooling fan, steering pump, electric system, pneumatic system, HVAC system, power take off (PTO). The required power of the auxiliaries depends on the technology used in the vehicle and the simulated driving cycle in VECTO.

The results of the component measurement will be stored in more than one CSV file.

---

<sup>3</sup>Regulation (EC) No 1222/2009 <http://data.europa.eu/eli/reg/2009/1222/oj>



### 2.4.2 Evaluation Tools

The evaluation tool checks whether the measurement data comply with the specified regulations of the component and performs the evaluation steps as outlined in Regulation EU 2017/2400 to generate the component data [4].

### 2.4.3 Data around VECTO

As shown in Figure 2.1 various files types occur during the certification process of heavy duty vehicles.

1. **Measurement Data**

The measurement data represent the files that are created during the measurement of a vehicle component.

2. **Component Data**

The component data file is the output file of the related evaluation tool which takes the measurement data as input.

3. **Certificate**

The certificate is the proof that the component or group of similar components has been successfully certified as part of the specified regulations to the corresponding component type. The certificate will be issued as hardcopy document or as PDF-file.

4. **Job Data**

The job data file contains the complete simulation configuration as well as the component configuration of the vehicle to be certified.

5. **Manufacturers Record File**

The manufacturers record file is the output file of VECTO after the simulation for the manufacturer. This file contains all the information needed to validate that the correct component data for a given vehicle was used for declaring the CO<sub>2</sub> figures with the simulation tool. This file must be kept by the manufacturer for at least 20 years.

6. **Customers Record File**

The customers record file is the output file of VECTO after the simulation for the customer. It contains the CO<sub>2</sub> emissions and fuel consumption and the applicable mission profiles.

### 2.4.4 VECTO Simulation Tool

To simulate an entire vehicle with the VECTO simulation tool two different simulation modes are available [3]:

- **Engineering mode**  
The engineering mode allows it to set all simulation parameters freely in order to test certain constellations of vehicle components and familiarize users with the simulation tool itself.
- **Declaration mode**  
The declaration mode allows only model parameters which are conform with the Commission Regulation (EU) 2017/2400, only certain model parameters such as payload, gearshift parameters etc., can be chosen generically.

#### **Certification mode**

The certification simulation mode will be used to certify a heavy duty vehicle. This mode is the same as the declaration mode with the only difference that the input for the simulation models will be done by the job data file. This XML file contains the entire configuration (each certify component data file) of the vehicle to be simulated [3].

## 3 Background

This chapter provides background information about databases in general, what types exist and what requirements they can meet. Some of the background information can be used later in the design and development phase of the Git database prototype.

### 3.1 Database

A database can in general be described as an organization system for a collection of digital data. To provide accessibility to these data collection usually a database management system (DBMS) will be used. The DBMS enables it to retrieve, change, delete and update the containing data within the database.

Today's database management systems can be roughly divided into two major groups, the traditionally relational database system and the Non-relational database system. The relational databases systems are still the most popular used database type, but the market share of Non-relational database which emerged in the last decade is still growing as shown in Figure 3.1.

#### 3.1.1 Relational Databases

In the 70's the term relational database was first introduced by Edgar Frank Codd in his research paper "A Relational Model of Data for Large Shared Data Banks" [6], what could be considered as origin point of relational databases. The first attempts of relational databases were developed in the following years.

### 3 Background

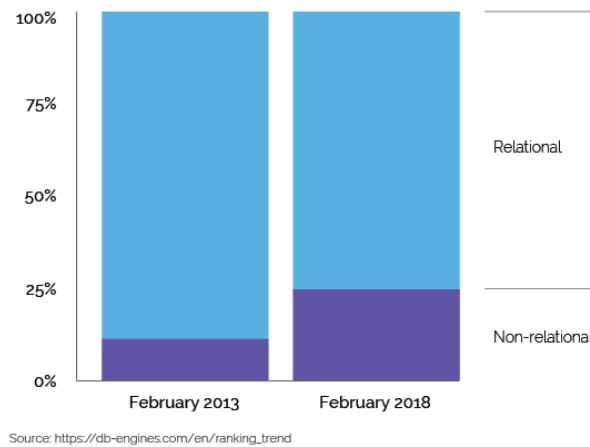


Figure 3.1: Popularity division between Relation and Non-relation Databases taken from source [5]

A relational database is characterized by its structured data scheme which organise the data in form of tables. A table consists at least one field that represents one column of the table, each of these columns being assigned a fixed data type such as string, integer, and so on. Besides the structured data scheme organised in table form, relational databases are further characterized by the relation between them.

A relation between tables can be one of the following types:

- **One-to-one relation**  
A row entry of one table can only be linked to one row entry of another table.
- **One-to-many relation**  
A row entry of one table can be linked to multiple row entries of another table.
- **Many-to-many relation**  
Multiple row entries of one table can be linked to multiple row entries of another table.

To reduce data redundancy and to increase the data integrity, the tables and their dependencies are normalized. A normalization can be applied through formal rules that must meet different criteria depending on the type of normalization.

To enable read, create, update and delete functionality on tables a respective relation database management system provides the domain-specific language Structured Query Language (SQL). Relational databases have the following advantages and disadvantages [5]:

### **Advantages**

- Based on the long time of usage, relational database management systems are a mature technology that is well documented and tested.
- The domain specific language SQL has established itself as the standard for relational databases management systems.
- All relational database management systems are ACID conformal and meet the requirements of Atomicity, Consistency, Isolation, and Durability.

### **Disadvantages**

- Relational database management systems cannot handle unstructured or semi-structured data, because of the strict table schema and the associated data type restrictions.
- Mapping between objects of an application and tables dose not always work.
- Migration between different relational database management system is difficult to perform, if the destination table differs from the source table structure.
- Relational database management systems tend to degrade the performance if data volume increase.

### 3.1.2 NoSQL/Non-Relational Databases

Not only SQL (NoSQL) or non-relational databases are a collection of different database types which use a different data model compared to the strict relational model. With the increasing complexity of web applications over the last two decades, the demands on data management and storage have changed. This results in the development of non-relational databases to overcome two key problems of relational databases. First, the scalability issue, which is reflected in rapid performance degradation as the data volume increases, and second, the inability to manage and store unstructured and semi-structured data [7].

The NoSQL database types can be assigned to one of the following five groups [7]:

- 1. Key-Value Store Databases**

Key-value store databases are one of the simplistic database models. A data entry within the key-value store is composed as a key-value pair, whereby the key represented a string and value representing the data the key refers to. Key-value stores databases can be used e.g. for the management of user sessions.

- 2. Column-Oriented Databases**

Column-oriented databases store the data tables column by column, which is the opposite of the relational database that store the data tables row by row. This meets the requirements of data mining and analytical applications which have to perform calculations on an accumulation of data such as aggregations.

- 3. Document Store Databases**

Document store databases store their data in form of documents, which are in the standard formats like XML, PDF, JSON, Word and so on. Document stores share similarities with relational databases as well as with key-value stores. The stored documents resemble a table of a relational database and each stored document is addressed with a unique key, which could be a string related to a URI or path, similar to the key of a key-value store entry. Document store databases enable the creation of complex data structures and the retrieval of documents according to specific properties. In addition, it enables the creation

of complex data structures because, unlike relational databases, no mapping of structures to tables is required.

#### 4. Graph Databases

Graph databases arrange the data in a graph structure, with the data representing a node and the edges represents their relationship between the data. Furthermore, additional properties can be assigned to each node. The execution of queries expressed as traverses is faster than in relational databases. The application areas of graph databases are social networking applications, recommendation software, security and access control, content management, network and cloud management and so on.

#### 5. Object Databases

An object database stores the data in form of objects similar to an object used in object-oriented programming. In addition to the management of stored objects, object databases support all common features of object-oriented programming such as data encapsulation, polymorphism and inheritance. Each stored object is assigned with a unique identifier within the object database, which can be used to access every single object. Due to the close integration between object databases and object-oriented programming languages, the retrieval of objects is faster than with relational databases, since an assignment of objects to tables is no longer necessary.

NoSQL databases have the following advantages and disadvantages [5]:

#### Advantages

- NoSQL databases can handle large volumes of unstructured and semi-structured data.
- The schema-free data models are easier to manage and more flexible than the relational data model.
- NoSQL databases can be used for big data, because of the faster transaction rates.
- NoSQL provide easier methods to scaling horizontal the database due to the possibility of distribution.

### 3 Background

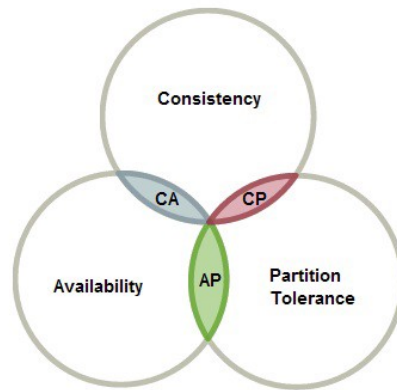


Figure 3.2: CAP Theorem taken from source [9]

#### Disadvantages

- NoSQL are less mature than relation database management systems.
- Most NoSQL databases are only BASE conformal and meet the requirements of Basic Availability, Soft State, Eventual Consistency.
- Missing standardized query language, which can be used in the majority of NoSQL databases.
- Every NoSQL database type requires specific expertise.

All relational database management systems are ACID conformal and meet the requirements of Atomicity, Consistency, Isolation, and Durability.

## 3.2 CAP Theorem

The CAP theorem [8] or also known as Brewer's theorem named after Eric Brewer was defined in the year 2000 and describes a special characteristic of every distributed database system. The theorem stated out that only two of the three attributes **C**onsistency, **A**vailability and **P**artition Tolerance can be fulfilled simultaneously [9].



- **Consistency**  
Consistency describes the overall consistency of the data in a distributed database environment. Consistency is given if it is ensured that the manipulated data record and all its replicas are updated after a transaction has been finished.
- **Availability**  
Availability describes the response time after a request to a distributed database environment. Availability is given if every request gets a response in an acceptable amount of time, but the response does not have to guarantee that it is the latest version of the requested data.
- **Partition Tolerance**  
Partition Tolerance describes the failure tolerance of a distributed database environment. Partition tolerance is given if the entire system is still functioning, if some nodes of the distributed database environment are down.

### CAP Theorem Structure and Mode Of Operation

The CAP assumption by Eric Brewer could be proven by an axiomatic proof by Seth Gilbert and Nancy Lynch[10] in the year 2002. As figure 3.2 shows, the theorem can be represented as three circles, while the intersection of two circles yields the possible combinations of two attributes [11].

- **Availability and Partition Tolerance (AP)**  
The Domain Name System (DNS), which maps domain names to numeric IP addresses, can be assigned to this category, it provides high availability and the downtime of certain nodes does not affect the distributed domain name system. The missing consistency of the domain name system can lead to outdated responses.
- **Consistency and Availability (CA)**  
Relational Database Management Systems (RDMS) can be assigned to this category. The data of an RDMS must always be stored consistently and a high availability must be guaranteed, depending on the area of application of the database. The lack of partition tolerance means that a RDMS cannot be used as a distributed database.

## 3 Background

- **Consistency and Partition Tolerance (CP)**

Applications in the financial sector can be assigned to this category. High consistency must be ensured, especially when working with money, and a high partition tolerance must also be ensured, failures of some nodes should not affect the rest of the financial system. In case of network failures, the service should not be available instead of processing incorrect transactions, which is the reason for the lack of availability in this operation mode.

### 3.2.1 ACID Consistency Model

ACID defines four characteristics of a database transaction which must be fulfilled to guarantee its reliability. These four characteristics must also be met when faults such as power failures, connection problems, etc. occur. All relational database systems fulfil these four characteristics and are therefore ACID conform. The four parts of the ACID model are described in more detail in the following list [12]:

- **Atomicity**

A single transaction is fulfilled atomically if the transaction is either completely successful or fails completely, e.g. if an error occurs during the transaction. This ensures that the database remains unchanged in the event of an error.

- **Consistency**

A single transaction is fulfilled consistently when a transaction is only executed if the transaction does not violate any defined rules, such as restrictions, cascades, and so on, that violate the data consistency of the database. A valid transaction transfers the database from one valid state to another.

- **Isolation**

A database system fulfills the isolation criteria when concurrently executed transactions do not interfere with each other. Each of the concurrent transactions should be executed in isolation on the database, and faulty transactions should not be detected by other concurrent transactions.

- **Durability**

Durability in the context of transactions means, if a transaction has been committed to the database it remains committed even in the event of system failures. To ensure transaction durability in the event of system failures, the database system uses database backups and transaction logs to enable the recovery of already committed transaction.

### 3.2.2 BASE Consistency Model

The BASE consistency model is used in case the ACID consistency model is not applicable, to provide a more scalable and affordable data architecture. The BASE model is used by most NoSQL databases. The three parts of the BASE model are described in more detail in the following list [12]:

- **Basic Availability**

Basic availability means to focus on availability over the correctness of the data. High availability will be achieved by the usage of a distributed database approach, which tolerates replication and temporary inconsistency of the data.

- **Soft State**

The soft state transfers the responsibility for data consistency from the database system to the developer. Soft state represents the opposite of the ACID model's consistency property.

- **Eventual Consistency**

Eventual consistency defines that inconsistent data will be consistent at a later point in time. However, there are no guarantees as to when this conversion from inconsistent to consistent data will take place.

### 3.3 Content-Addressed Storage

Content-Addressed Storage (CAS) is a type of a data management technology on hard disks, where the information gets stored and searched by its content. This type of data management technology has been primarily developed for data which rarely or never changes over time.

To store the data according to its content, CAS-systems use a cryptographic hash function whose task it is to calculate an identifier from the content. A cryptographic hash function takes an arbitrary length of data and generates a fixed length hash value. Each change, even if it is only 1 bit in the input data, leads to a generation of a different output by the cryptographic hash function.

With this generated identifier, the CAS-system linked it to the storage location of the data. If new data is added to store, first the identifier is generated and compared with all existing identifiers. If it is a non-matching identifier in the link list of the CAS-system, the file is stored, otherwise it is ignored because the file is already stored. This also reveals the most obvious advantage of a content-addressed storage, it prevents the storage of the same data [13].

#### Advantage

- Content-addressed storage prevents the storage of duplicated data.
- The search by the given data content is very fast and provides the certainty that the document found matches the specified data.

#### Disadvantage

- Content-addressed storage can only be used efficiently for data which is rarely or never changed.
- Search and save behaviour is only as good as the underlying cryptographic hash function.

# 4 Version Control System

## 4.1 What is a Version Control System?

A Version Control System (VCS) tracks and saves the changes which are applied to a file over time in a way so that these file can be returned to the status before the respective change happened.

The simplest approach of such a system is to create a separate copy each time after a change happened to the file. This can be done with almost every file type, however the big downside of this simple approach is the rapid rise of administration and storage effort after a certain amount of changes and by using this approach for more than one file. The main application area of a version control system is tracking changes of text files, especially source code files.

The first type of a version control systems were published in the year 1972 with the release of the Source Code Control System (SCCS). This version control system was the first which enabled to track the history of changes on individual files such as configuration files or documentation. In the 1980's another early days version of a version control system was released, called Revision Control System (RCS), which simply stores the different versions of a file similar to the previously mentioned simple approach [14].

## 4 Version Control System

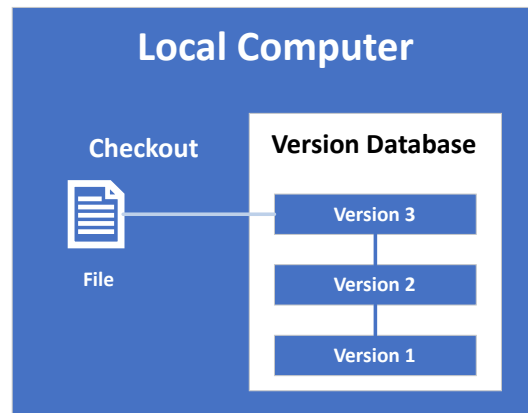


Figure 4.1: Local version control system (based on [15])

## 4.2 Types of Version Control Systems

### 4.2.1 Local Version Control System

The local Version Control System (VCS) solves the two problems of the previously mentioned simple approach. The administration effort issue will be remedied by the usage of a database and the data effort issue can be reduced through saving file modifications only of files which one time already fully saved within the VCS. The conceptual construction of a VCS is shown in Figure 4.1. The best known example of a local VCS are the Revision Control System (RCS) [15].

### 4.2.2 Centralized Version Control System

To enable project collaboration between users, Centralized Version Control System (CVCS) was developed. With this system a central server holds one or more repositories containing all the data in its various versions that have been added to the system over the time. To pick up a file the client connects to this central server and retrieves the file in the respective version. This type of version control system has the advantage that the whole data and its history is collected on a central location which minimizes the administration

## 4.2 Types of Version Control Systems

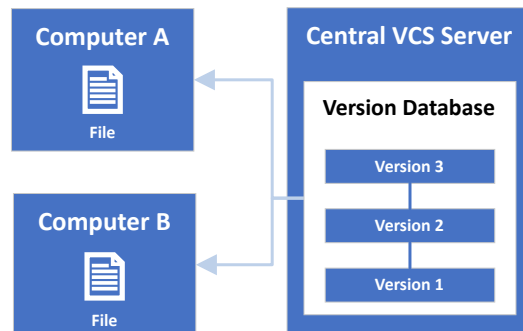


Figure 4.2: Centralized version control system (based on [15])

effort. However, the advantage is also the biggest disadvantage: if the central server is corrupted the whole data and its history can be lost. And every downtime of the central server prevents collaboration between users, which is the main intention of this type of version control system. The conceptual construction of a centralized VCS is shown in Figure 4.2. The best known examples of centralized VCS are the Concurrent Versions System (CVS) and the Apache Subversion, both are open source [15].

### 4.2.3 Distributed Version Control System

The Distributed Version Control System (DVCS) is used to deviate from the centralization of data on a single server. This decentralization distributes the error concentration to several locations. The data as well as its corresponding history will be mirrored over to the used repositories within the DVCS. By mirroring of the data at every checkout, each repository represents a potential backup if any of the data gets corrupted. The conceptual construction of a distributed VCS is shown in Figure 4.3. The best known example of distributed VCS are Git and Mercurial, both are open source [15].

## 4 Version Control System

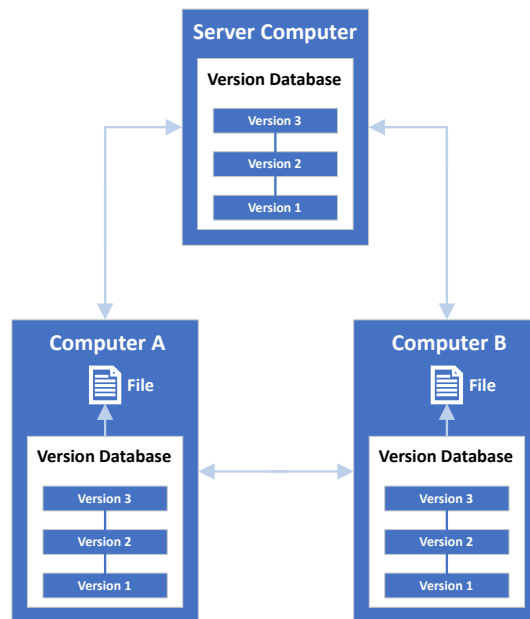


Figure 4.3: Distributed version control system (based on [15])

### 4.3 Git

Git represents the base of the development of the Content Addressable Storage (CAS) database of this thesis. Therefore, this section takes a closer look at Git and how it is built up and how it works.

#### 4.3.1 Git History

Git is an open source Distributed Version Control System(DVCS) and was initiated by Linus Torvalds. In April of the year 2005 Linus Torvald started with the development of Git, which was later the replacement of the used DVCS BitKeeper<sup>1</sup>. BitKeeper was free for open source projects and was used between 2002 and 2005 as the source code version control system for the Linux kernel project.

<sup>1</sup>BitKeeper <https://www.bitkeeper.org/>



In the year 2005 the BitKeeper creator refuses the free of charge status for the Linux Kernel project. The reason of the refusion was the violation of the noncompete clause of BitKeeper's license agreement by Andrew Tridgell. In order to access a BitKeeper repository, a provided closed code client was used. Andrew Tridgell wrote an open source tool that provided the same functionality as the client, which lead to the violation of the competition clause and finally to the rejection of free of charge status from the Linux kernel project[16].

This rejection lead Linus Torvald to the decision to develop his own source code management system system for the Linux kernel development.

With the lessons learned during the usage of BitKeeper and the given requirements of kernel development the new system should fulfil the following points [15]:

- **Speed**
- **Simple design**
- **Strong support for non-linear development (thousands of parallel branches)**
- **Fully distributed**
- **Able to handle large projects like the Linux kernel efficiently (speed and data size)**

The development of Git started at the beginning of April 2005 and a first version was released in July.

### 4.3.2 Git Key Concepts

#### Delta Storage

Delta storage is a file-by-file based storage model and is used by most source management systems. This storage model observes the changes made to a file regardless of the changes made to other files. Which means every newly added file adds a newly created object to the repository of the source management system. If an already added file gets modified only the difference or also known as delta, between the original file and the modified

## 4 Version Control System

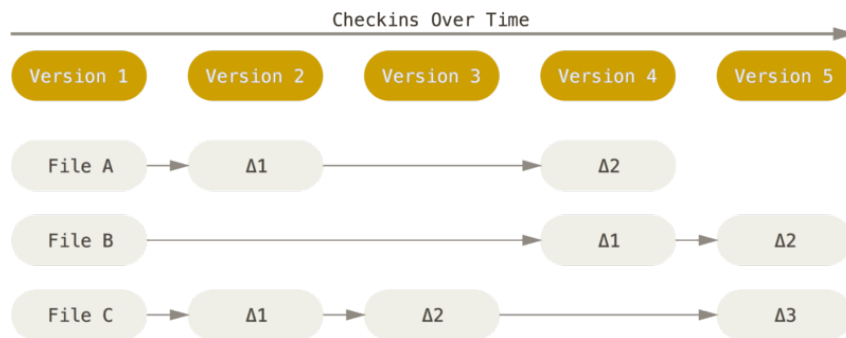


Figure 4.4: Example of Delta Storage System taken from [15]

file gets saved and linked as the file's next revision.

To preserve the file content at a certain point in time within the tracking history, all deltas that were created before that point will be added to the original file [17]. An example of a delta storage system is shown in Figure 4.4 only a newly added file (Version 1) creates an object, each change to an existing file will create a delta file at each check in.

### Snapshot Storage

Unlike the delta storage model, Git takes a completely different approach to track changes done to a file over time, known as snapshot storage. Git does not monitor the changes that happen to each individual file in the workspace. Instead, it saves changes as separate snapshots across the entire workspace. A snapshot gets generated by the usage of the Git command `commit`.

The snapshot contains the content of all involved files and directories within the workspace at that time. Every commit which represents the snapshot is linked to the previous commit that was taken before the current commit, unless it is the very first commit. The connection between commits allows to switch between a snapshot taken at one time and a snapshot taken at another time.

Every modified file between two snapshots generates a new object. This object contains all data of the modified file, any other file that does not change between two snapshots does not create an additional object in the

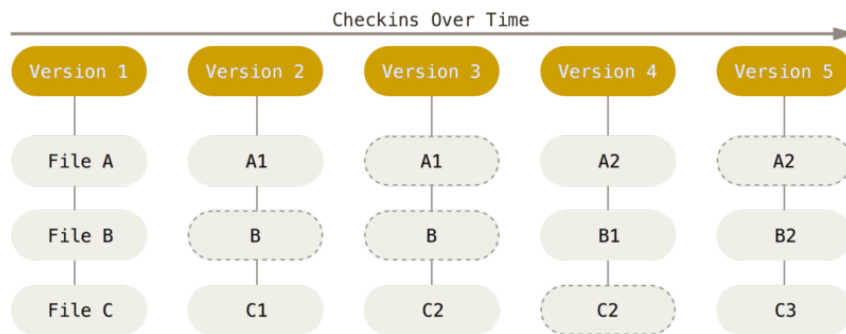


Figure 4.5: Example of Snapshot Storage System taken from [15]

repository.

Saving each file and the modified file as a new object allows it to switch very quickly from one snapshot to another. It also allows to switch very quickly from one file status in a particular snapshot to a different file status in another snapshot [17].

An example of a snapshot system is shown in Figure 4.5, each change on a file will create a new object at each check in.

### Git Repository

In order to version a project, the first step is to create the repository skeleton in the project folder. The repository will be created by the usage of the Git command `git init`. This creates a hidden subdirectory within the project folder named `.git`. This repository folder contains the following files and folder as shown in Figure 4.6.

The task of the folders and files listed in the figure 4.6 will be briefly described in the following listing [15]:

- **hooks**

By default this directory contains some example scripts with shell-code. The task of this folder is to provide a place where self-defined programs can be executed at certain points during the Git execution phase.

## 4 Version Control System

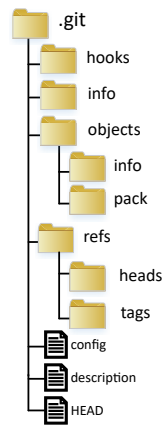


Figure 4.6: Git repository folder structure

- **info**

The info directory contains the exclude file similar to the ".gitignore file" (specifies files which should be ignore by Git) with the only difference that the exclude file will not be shared between repositories.

- **objects**

The objects directory contains all newly created objects. The objects are organized in over 256 subdirectories, each subdirectory name consists of the two initial letters of the SHA-1 identifier of the respective object. These additional subdirectories facilitate the administration of the objects.

- **info**

The info directory contains further information about the object store.

- **pack**

The pack directory contains the pack files, which are single files containing several similar objects that are delta compressed.

- **refs**

The refs directory stores all references such as branches and tags. If a remote repository exists, its references are also created in the subdirectory.

  - **heads**

The directory heads contains all local branch references.
  - **tags**

The directory tags contains all local tag references.
- **config**

Is the repository specific configuration file.
- **description**

The description file can contain the description of the repository.
- **HEAD**

The HEAD file contains the reference(top commit SHA-1 hash) of the active branch within the repository.

## Git Objects

Git is a content addressable storage similar to the description in section 3.3. Each version-controlled file is stored as an object within the Git repository. Each created object will be identified by a 40 digit SHA-1 hash generated from the content of the object.

Git uses four different object types to organize the content within the content addressable storage, these are described in the following list [15]:

### Blob Object

The blob (abbreviation of Binary Large Object) object contains the content of a file that has been added for versioning. The content of the blob object will be compressed using the gzip<sup>2</sup> algorithm before the blob object will be created. Each blob object can only contain the content of a single file.

### Tree Object

The tree object will be used to group files together. A tree object can contain

---

<sup>2</sup>Gzip <https://www.gzip.org/>

## 4 Version Control System

references to existing blobs as well as to other existing trees. One tree entry consists of four entries the file permission code, the object type, the SHA-1 identifier and the file name of the associated file. A example of two tree entries are shown in the following listing:

```
040000 tree 1bd86bd18e2389f9edd2c10da4d4eaca527cdb76 Files
100644 blob 5dcd581f2ef55612fbb9e69b5821eod0562b5f57 main.cs
```

### Commit Object

The commit object is responsible for storing metadata of a snapshot. The metadata consists of the following parts: the tree reference, the parent commit reference, the author signature, the committer signature and the commit message. Each commit refers to one parent commit except the initial commit which cannot refer to any commit. In case of a commit merge, a commit can refer to multiple parents. An example of a commit is shown in the following listing:

```
tree 5d21358b192c1a699boae1eobf4f6d65d8e9df15
parent 49cf244225cd2406c1e58dcca6c76fod9b00286f
author John Doe <john@doe.org> 1567950964 +0200
committer John Doe <john@doe.org> 1567950964 +0200

Second commit
```

### Tag Object

The tag object is similar to a commit object. It also adds additional meta information to the object it points at. The tag object consists of the following parts:

- The reference of the tagged object.
- The object type of the tagged object.
- The name of the tag (the name must be unique within the repository).
- The creator signature of the tag ("tagger").
- The tag message of the tag.

An example of a tag is shown in the following listing:

```
object 7c74cdc71aa4c80fb84820cbae30aef5bod64c49
type commit
tag tag name V.1
tagger John Doe <john@doe.org> 1567950964 +0200

Some tag message
```

A tag is able to point at any git object type, but commonly only commit objects will be tagged.

## Branch

Branches are the central data organization feature of Git. A branch is a symbolic reference that has a unique name and points to single commit. The default branch of a newly created Git repository is the "master" branch. A branch will be created in the "heads/refs" folder of the repository and is a simple file containing a commit SHA-1 hash and the branch name as filename.

In comparison to a tag that also points to a commit, a branch is moveable. A tag can only reference a commit for what it was created for. Whereas the reference of branch, which is pointing to a commit, can be changed to any other commit [17].

## Git Object Model

Git organises the data in a simplified form of the UNIX file system. The folder structure as well as the containing data will be mapped by the usage of the three main Git objects types blob, tree and commit. An example mapping of a directory structure is shown in the Figure 4.7.

Each folder will be represented by a tree, the very first tree represents the root tree of the directory. Each file creates a new blob in the repository and will be referenced by the tree above, which also stores the filename of the related file. The commit points to the tree root of the mapped directory,

## 4 Version Control System

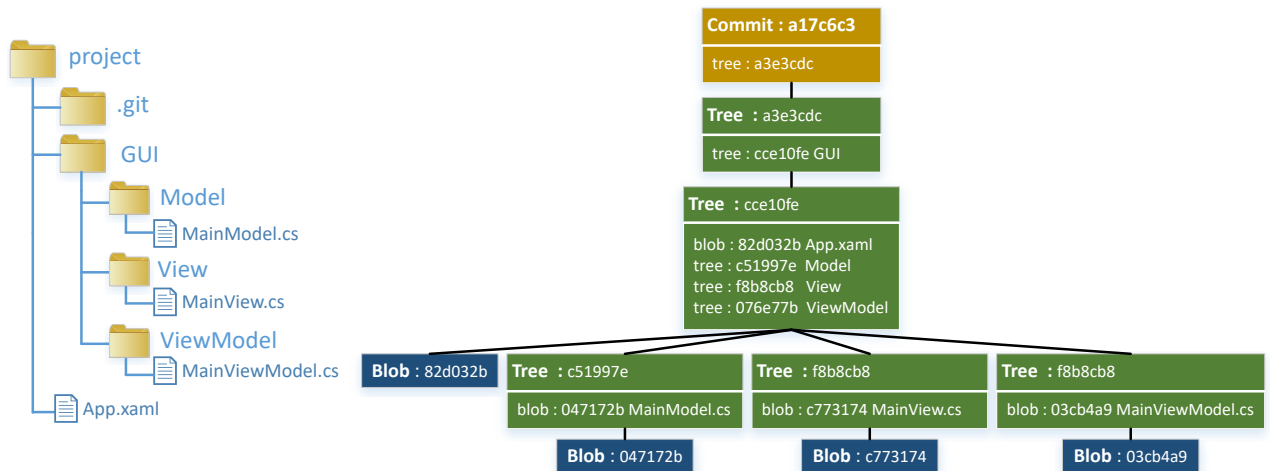


Figure 4.7: Git Object Model

which will later be used to track when it was committed and as selection point to switch between commits.

### Git Data Integrity

To ensure data integrity, Git calculates a checksum for each file before storing it in the repository. In addition, this checksum will be used as filename for the stored objects in the object database. This allows Git to recognize immediately changes in any file content or directory.

The checksum will be calculated using the SHA-1 algorithm which belongs to the group of cryptographic hash functions. This means that the algorithm is suitable for cryptographic hash functions such as digital signatures, Message Authentication Codes (MACs), etc.

A hash function, also called a one-way function, is characterized by the fact that it always generates a message of fixed length, independently of the length of the given input data. Additionally, each change of the input data leads to a change of the calculate hash. A secure hash function has also to fulfil the following conditions:



- **Preimage resistance**  
Preimage resistance property of a hash function means that from a given output of a hash function it is very hard to calculate the input message of the hash function.
- **Second preimage resistance**  
Second preimage resistance property of a hash function means that it is hard to find another message that leads to the same hash.
- **Collision resistance**  
Collision resistance property of a hash function means that it is hard to find any at least two messages that lead to the same hash.

In February 23 of the year 2017 Google official announced the first collision of the SHA-1 hash function. Until then, there were only theoretical assumption that a collision could be found. The computation of the collision took approximately 6500 single CPU years and 100 single GPU years which is still more than 100000 times faster than the brute force attempt [18].

Because of the wide use of Git and the several complex problems which will be described in the next listing it was not yet possible to switch to another hash algorithm.

There are several complex problems that need to be solved to allow a change to another hash algorithm these are [19]:

1. **Variable declarations**  
There are a lot of variable declaration within Git in the following form *unsigned char sha1[20]* that need to be adjusted for the new hash algorithm.
2. **Existing repositories**  
The new hash algorithm must be introduced without breaking the existing ones. To support existing repositories, a new repository type could be introduced in Git that can manage two different blob types, the existing blob type and a new blob type that uses the new-hash type. As soon as only new hash object types are used within the new repository, it is not allowed to add old object types. With the sole exception that new-hash commits can refer to old commits to preserve the repository history.

#### 4 Version Control System

The disadvantage of this approach is that duplications can occur when the content of a file is stored in both the new and old blob type.

Another approach to use existing repositories is the usage of some kind of mapping where the old objects will be mapped to the new-hash object types. A disadvantage of this approach is the increasing complexity and also the increase in storage space, especially for large-scale projects.

# 5 Design

This chapter presents the design document that forms the basis for the later developed prototype of a distributed CAS database for vehicle component certification data using Git.

## 5.1 Use cases around VECTO

As shown in chapter 2 there are different types of data around VECTO. These data can basically be assigned to one of the following groups. Furthermore, each group represents an overall use case of VECTO, in which data arises that should be stored in a structured and retrievable manner.

- **Certification of component**  
Represents the use case of a single component certification process by a component manufacturer together with an independent technical service.
- **Declaration of vehicle**  
Represents the use case of a vehicle certification process by an Original Equipment Manufacturer (OEM).
- **Vehicle Test Procedure(VTP)**  
Represents the use case when an independent laboratory selects an already certified vehicle from the OEM to perform the CO<sub>2</sub> emission measurement under real conditions. Purpose is to make sure the declared values match with the vehicle's CO<sub>2</sub> emissions.  
This use case represents an archiving purpose, where the emerged

## 5 Design

measuring data recorded during the measurement under real conditions, are to be stored in relation to the already existing measurement data.

- **COP Conformity of production**

Represents the use case when a component is selected again for measurement after a series (approximately 3000) produced components, in order to check the wear out during production. Furthermore, to control the variance between the received component data and the emerged measurement component data of the selected component.

This use case represents an archiving purpose, where the measured component data should be stored in relation to the already stored component data.

In order to be able to manage and store the data in the above-mentioned use cases, Git is selected as the storage backend. The following design and requirement listings covers only the first use case "Certification of component" fully, this was necessary due to complexity of the prototype. The remaining use cases will be considered during the development of the design but are not included separately in the design.

### 5.2 Requirements for certification of component

During the certification process of components like Engine, Gearbox, etc., with an independent technical service the following data can occur:

- **Measurement data**

The measurement data emerges during measurement process of a component on the test stand and will be used as input for the the related VECTO component tool.

- **Component data**

The component data represents the output file of respective component tool such as VectoAirdrag, VectoEngine etc., which uses the measurement data files as input.

## 5.2 Requirements for certification of component

- **Standard value**

The standard value file is a replacement file for the component data file that is created when no certification of the component was performed. The standard value file has the same structure as the component data, but contains only default values, that are generally worse than the values of a certificated component and its respective component data file.

- **User defined data**

The user defined data can be defined by the user to add additional information to the associated component.

Each of the above mentioned data types must be stored into the Git database.

In order to distinguish between the data types mentioned above an appropriate organization is required.

In addition to organizing the data, the prototype should be able to search for data already stored in the Git database. Depending on the data type different search criteria should be supported.

- **Certification number**

The certification number will be issued during the certification process of a component. A certification number can be assigned to a component or a group of similar components.

- **Manufacturer**

The manufacturer name under which the vehicle component was manufactured.

- **Model**

The unique model name of the component.

- **Date**

The date when the certification of the vehicle component happened. In case of the standard value file, the creation date of the file is specified in the date field.

## 5 Design

- **Git Id of the component data**

The Git Id of the component data, represents the unique identifier (SHA-1 hash) generated by Git. This unique identifier can be used later to fetch related component data file.

All data types listed in this section should be searchable by the search terms listed above except **Standard values files** and **User-defined data files**, which should only be searchable using the terms Manufacturer, Model and Data.

In addition to the search function, a export as well as an import function should be available for previously selected component data files.

### 5.3 Design and Data Partitioning

In order to fulfil the given requirements in section 5.2, the following structure which is illustrated in Figure 5.1 within a Git repository was designed. The most important points and their conditions is described in more detail in the following.

- To simplify the exchange process between Git repositories, the data from the certification process of a component is split up into two orphan branches. Orphan branches are a special kind of a branch, which is independent from the log history of other existing branches within the repository.
- The branches will be created with the following two naming conventions:
  - public/<Manufacturer>/COMPONENT/<Year>/<Model>/<Abbreviated XML Hash>
  - private/<Manufacturer>/COMPONENT/<Year>/<Model>

Every standard value file as well as the component data file contains a XML-hash, which will be further added (abbreviated form 12 characters long) as part of the public branch name, in order to make them unique within the repository.

- The public branch can contain only the following file data types:
  - **Standard values**
  - **Component data**
  - **Certificate**
- The private branch can contain only the following file data types:
  - **Measurement data**
  - **User-defined data**
- The data within a public branch may be sent to some other party (e.g. vehicle manufacturer), while the data contained in a private branch should remain in-house.
- The public branch will be built up in the following two ways:
  1. a) The first commit of the public branch contains a tree node pointing to component data and the commit message contains the certification date read from the component file.
  - b) The second commit of the public branch holds a tree node, which points to the component data file and the certificate data file.

In case of a re-certification of a vehicle component the public branch will be built up in the following way:

2. a) The first commit of the public branch holds the tree node, which points to standard values file and the commit message contains the certification date read from the standard value file.
  - b) The second commit holds a tree node, which points to the standard value file and the component data file.
  - c) The third commit holds a tree node, which points to the standard value file, the component data file and the certificate file.
- If no re-certification takes places, only the first mentioned commit sequence will be executed.

## 5 Design

Additional commits can be added to the public branch with additional data after one of the above defined sequences has been performed.

- After the creation of the first commit on the public branch, the commit will be tagged with two tags. The first tag with the certification number as tag name and the second tag with the Git Id as tag name. The certification number as well as the component data Git Id will be split up in smaller pieces as described in section 5.6 and indicated in the following listing. The certification number tag contains the same abbreviated XML-hash of the branch name that was read in the associated component data file.
  - /CERT\_NR/<section 1&2&3>/<section 4&5>/<Abbreviated XML Hash>
  - /GIT/<First 2 GIT ID characters>/<remaining 38 GIT ID characters>
- When a certificate is added to an existing public branch by a new commit, all public branches tagged with the same certificate number are automatically extended by the certificate commit if the certificate commit does not already exist.
- The private branch is divided into two directories (which are represented in Git as two tree nodes), one for measurement data files and one for user-defined data files.
- User-defined data can only be stored to an already existing private branch within the repository.
- The first commit of a private branch will contain the certification date as commit message.
- Files stored within a private branch can keep the current file name. If the file name is already used within the private branch and the new file content differs from the stored one, the user should be able to choose another one.



- To identify the different data types within a public branch some naming conventions are required. Every data file name within a tree node will be replaced by one of the following terms:

- **COMPONENT\_DATA**
- **CERTIFICATE\_DATA**
- **STANDARD\_VALUES**

- All messages which will be added to tags or commits must be in the following JSON-Form:

The message content of a certification tag and Git Id tag:

```
{
  'BranchType' : 'Private',
  'Manufacturer': 'MAN',
  'Year': '2017',
  'Model': 'T-13',
  'XmlHash': 'af2452tzrysf'
}
```

The message content for the first component data commit of a public branch:

```
{
  'CertificationDate' : '28.12.2017'
}
```

The message content for the first standard value file commit of a public branch:

```
{
  'CommitDate' : '28.12.2017'
}
```

- Before a new files are added to an existing branch, the Git Id of the new file will be precalculated. The calculated Git Id will be used to compare it with the existing stored files within the branch. Only if the precalculated Git Id differs from all stored files within the branch a new commit can be added.

## 5 Design

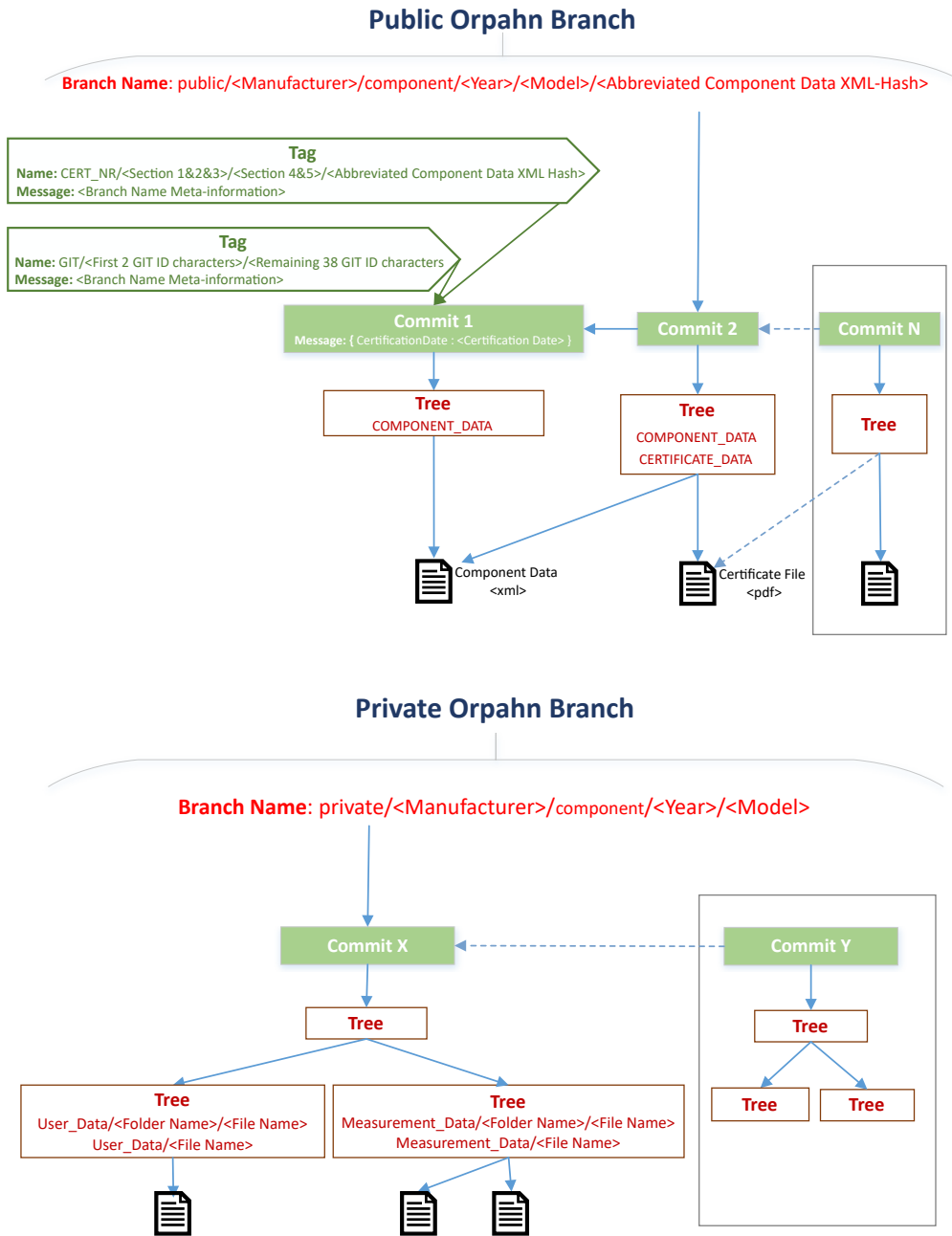


Figure 5.1: Organization and structuring of component data files in the Git repository

Disallowed character	Replacement character
\	(
/	)
:	;
*	#
?	\$
"	,
<	]
>	[
	+
""	-
.	%

Table 5.1: Replacement characters for windows systems.

## 5.4 Search behaviour

As already mentioned in section 5.2 different searches must be supported depending on the data type. Each given search term except the date, will be verified if the given term contains any disallowed characters as mentioned in table 5.1 and will be replaced before the search starts.

All search terms which are given by the user will be connected by the logical AND operator, which means only components will be displayed where all given search terms hold true.

The search starts with the public branch names if nothing could be found, the search continues with the private branch names. In case of a given Date as search term, the year will be used in the first place to narrow down the branch name list. Afterwards every first commit of the public branches will be investigated whether it contains the searched date or not.

The certification number and component Git Id will be searched under the existing tags of the repository, each tag contains the public branch name in JSON-format where it points at, as message, to easily find the related public branch.

## 5.5 Transfer behaviour

As already mentioned in section 5.2, a transfer functionality between repositories must be supported for both private data (measurement data, user defined data) and public data (standard values file, component data, certificate file).

To transfer from one repository to another a additional bare repository will be used as transfer medium. The user should be able to pre-select both private data and public data for transmission if the corresponding data is present within the component. After the preselection, the user can create the transfer package which contains the repository with the branches. If the transfer package are selected for import, the following import behaviour should be supported:

### **Import behaviour in the case of public data**

- The source repository will be scanned to determine whether the public branch of the transfer repository already exists in the source repository. If not, the branch from the transfer repository and its associated data are added to the source repository.
- If the public branch already exists in the source repository, the tree entries of the two top commits of the corresponding branches are compared. The files which are missing in the source branch compared to the transfer branch will be added. As well as the tags that refer to the component data commit, if they do not already exist.

### **Import behaviour in the case of private data**

- The source repository will be scanned to determine whether the private branch of the transfer repository already exists in the source repository. If not, the branch from the transfer repository and its associated data are added to the source repository.
- If the private branch already exists in the source repository, the tree entries of the two top commits of the corresponding branches are compared. If a file exists in the source tree with the same filename but different content as in the transfer tree, the user must decide whether the file from the transfer tree should be added with a different

## 5.6 Structure of the certification number

filename or if the file should be ignored. All other files from the transfer tree with a different filename than the files in the source tree are automatically added.

### 5.6 Structure of the certification number

A certification number has the following structure:

`eX*YYY/YYYY*ZZZ/ZZZZ*X*0000*00`

The certification number can be divided into five sections. The meaning of every single section is described in the following listing:

<b>eX</b>	Indication of which country issuing the certificate
<b>YYY/YYYY</b>	CO <sub>2</sub> certification act
<b>ZZZ/ZZZZ</b>	Latest amending act
<b>X</b>	Additional digit for section 3 to classify the component type which can be:  <b>T</b> for transmission <b>O</b> for other torque transferring components <b>K</b> for angle drive <b>M</b> for torque converter <b>E</b> for engine <b>L</b> for axle <b>P</b> for air-drag <b>T</b> for tyre
<b>0000</b>	Base certification number
<b>00</b>	Extension

For the tagging purpose the certification number which is used for the naming will split up into the following two parts:

- `eX*YYY/YYYY*ZZZ/ZZZZ*X`
- `0000*00`

This split should prevent that all certificate tags will be saved within one folder, which can lead to performance problems. This method is similar to object management within Git.

## 5.7 Use cases

The following use cases are split up into two main sections, which are the user and the VECTO-GIT perspective. The VECTO-GIT perspective lists and describes the single steps, in particular the individual GIT commands, which are needed to fulfil the respective user interaction. This exact description should outline the later implementation of VECTO-GIT with LibGit2Sharp<sup>1</sup> and should also make it easier to prevent possible errors during the implementation.

### 5.7.1 Component data of a new certificated component should be stored in the storage backend

The component data is the result data of the respective component tool, which uses the measurement data as input data.

#### User perspective

- The user selects "Save Component Data" in "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Save Data" view Figure 5.3.
- The user chooses the component data file for upload.
- The chosen file will be checked per XSD for validity.
- If the XML file is invalid, the user will be informed of the reason for the invalidity and the invalid file will be highlighted in red in the selection list.

---

<sup>1</sup>LibGit2Sharp <https://github.com/libgit2/libgit2sharp>

- By pressing on the button "Save" the listed data, will be stored into the VECTO-GIT repository and the user will be informed whether the save operation was successful or not.

### **VECTO-GIT perspective**

- The file chosen for uploading will be examined by the use of the XSD file to verify the validity.
- After a successful validity check the model name, certification number, manufacturer name, XML-hash and the certification date will be read from the specified component data file.
- The disallowed characters of the certificate number will be exchanged by the alternative characters listed in the table 5.1.
- The repository will be scanned by the use of the certification number, model name and manufacturer name, to ensure that the specified component data is not already stored within the repository, otherwise the user will be notified.
- A new public branch with the specified naming convention in the section 5.3, will be created and the selected component data will be committed with the certification date as commit message, as specified in section 5.3.
- A tag with the certification number and a tag with component data Git Id will be created and linked to the first commit of the new branch, with the related branch name as tag message in JSON-Format, as specified in section 5.3.

To set up the branch the following Git commands are needed in the following order:

1. `git checkout --orphan <Branch Name>`  
Create a new orphan branch with the given name.
2. `git hash-object -w <File Path>`  
Write the Git object in the repository from the given file.
3. `git update-index --add --cacheinfo 100644 <Blob GIT ID> <File Name>`  
Adds the new Git object to the index.

## 5 Design

4. `git write-tree`  
Create a new tree node from the current index.
5. `git commit-tree <Tree GIT ID> -m <Message>`  
Create a new commit with the newly created tree node.
6. `git update-ref refs/heads/<Branch Name> <Commit GIT ID>`  
Set the branch head to the newly created commit.

### 5.7.2 Certificate of a component shall be stored in the storage backend

The issued certificate will be handed out some time later after the certification proceed. This certificate must also be linked to the corresponding component data within the public branch.

#### User perspective

- The user selects "Append Component Data" from the "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Search Data" view Figure 5.4.
- The user enters at least one search term and press the "Search" button.
- If the right component was found, the user presses "Add Data" button in the result listing and will be redirected to the "Save Data" view Figure 5.3.
- The user chooses the certificate file for upload.
- The selected certificate file will be listed in the "Selected Data" listing and the user has to choose CERTIFICATE\_DATA as component data type from the combobox.
- By pressing the "Save" button the certificate file will be added to all components (tagged with the same certificate number) for which the certificate was issued.
- The user will be informed whether the append operation was successful or not.



- On success the user will be redirected to "VECTO-GIT-Main" view Figure 5.2.
- In the event of an error, the user remains on the "Save Data" view Figure 5.3 and will be informed of the cause of the error.

#### VECTO-GIT perspective

- After pressing "Append" in the search result view Figure 5.4 the user will be redirected to the "Save Data" view Figure 5.3.
- After the redirection to the "Save Data" view Figure 5.3, the certificate can be added to the selected component.
- After the selection of the certificate file and the component data type CERTIFICATE\_DATA the selected component and all components with the same certificate number will be scanned, whether the certificate file has already been added.
- If a certificate file has been already added to the component the user gets informed and remains in the "Save Data" view Figure 5.3.
- If the user press save the certificate file will be assigned to the selected component as well as to any component which is tagged with the same certificate number.

To append the certificate file to the corresponding component data the following git commands are necessary, in the following order:

1. `git log -n 1 --pretty=format:"%T " <Branch Name>`  
Determine the tree node Git Id from top commit of the branch.
2. `git read-tree <Tree Node GIT ID>`  
Read the tree node entries into the index.
3. `git hash-object -w <File Path>`  
Write the Git object in the repository of the given file.
4. `git update-index --add --cacheinfo 100644 <Blob GIT ID> <File Name>`  
Add the new Git object to the index.
5. `git write-tree`  
Create a new tree node from the current index.

## 5 Design

6. `git log -n 1 --pretty=format:"%H " <Branch Name>`  
Determine the top commit Git Id of the branch.
7. `git commit-tree <Tree GIT ID> -p <Parent Commit GIT ID>`  
Create a new commit with the newly created tree node and the actual top commit as parent.
8. `git update-ref refs/heads/<Branch Name> <Commit GIT ID>`  
Set the branch head to the newly created commit.

### 5.7.3 Measured data of a component shall be stored in the storage backend

The measurement data consists of more than one file, depending on the certificated component. The measurement data files will be used for the corresponding VECTO-Component-Tool as input to generate the component file. These files should be saved only for archive purposes and are not intended to transfer them to other parties.

#### User perspective

- In case of an already existing private branch the user perspective course of events are as follows:
  - The user selects "Append Component Data" from the "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Search Data" view Figure 5.4.
  - The user enters at least one search term and press the "Search" button.
  - If the right component was found, the user presses "Add Data" button in the result listing and will be redirected to the "Save Data" view Figure 5.3.
  - The user chooses at least one measurement data file for upload.
  - The selected files will be listed in the "Selected Data" table and the user has to choose MEASUREMENT\_DATA as component data type from the combobox.

- In case of an already used file name, the user will be informed and has to choose another one.
- By pressing of the button "Save" the listed data, will be stored into the existing private branch, the user will be informed if the save process was successful or not.
- In case of new private branch for measurement the user perspective course of events are as follows:
  - The user selects "Save Component Data" in "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Save Data" view Figure 5.3.
  - The user chooses at least one measurement data file and the associated component data file for upload.
  - The selected files will be listed in the selected data table and the user must select MEASUREMENT\_DATA and COMPONENT\_DATA as component data type for the associated files from the combobox.
  - The selected component data file will be checked per XSD for validity and the repository will be scanned to verify that the manufacturer name and model name have not already been used within a branch name.
  - On invalidity (invalid XML-file or manufacturer name and model name combination already used), the user will be informed for the invalidity reason.
  - By pressing the "Save" button the listed data, will be stored into the new private branch, the user will be informed if the save process was successful or not.

#### **VECTO-GIT perspective**

- During the XSD validation process the model name and manufacturer name will be extracted from the selected component data file.
- If the redirection to "Save Data" view Figure 5.3 occurs by the "Component Data Search" view Figure 5.4 the following checks are necessary.

## 5 Design

- The file name as well as the content must differ from the already saved files within the private branch.
- If the redirection to “Save Data” view Figure 5.3 occurs by “VECTO-GIT-Main” view Figure 5.2 the following checks are necessary.
  - The given component data file must be validated by the XSD file.
  - The composition of the extracted model name and manufacturer name should be used to determine if the same named private branch already exists within the repository.
- In case of an already existing private branch the selected measurement data will be added if the data differs from the data already stored otherwise the user gets informed.
- Otherwise a new private branch will be created with the selected measurement data.

To create a new private branch and add the selected measurement data files the following Git commands are necessary, in the following order:

1. `git checkout --orphan <Branch Name>`  
Create a new branch with the given branch name.
2. `git hash-object -w <File Path>`  
Write the Git object in the repository of the given file.
3. `git update-index --add --cacheinfo 100644 <Blob GIT ID> MEASUREMENT_DATA/<File Name>`  
Add the new Git object into the folder MEASUREMENT\_DATA to the index.
4. `git write-tree`  
Create a new tree node from the current index.
5. `git commit-tree <Tree GIT ID> -m ""`  
Create a new commit with the newly created tree node.
6. `git update-ref refs/heads/<Branch Name> <Commit GIT ID>`  
Set the branch head to the newly created commit.

#### 5.7.4 User-defined data for a component shall be stored in the storage backend

User-defined data is the opportunity for a user to define and add additional information for a component. The given user-defined data can be files of any kind, and will be saved in a separated folder within the private branch of the selected component. User-defined data can only be stored if at least the corresponding private branch already exists as defined in section 5.

##### User perspective

- The user selects "Append Data To Component" in the "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Search Data" view Figure 5.4.
- The user enters at least one search term and press the "Search" button.
- If the right component was found, the user presses "Add Data" button in the result listing and will be redirected to the "Save Data" view Figure 5.3.
- In the view "Save Data" the user opens a file dialog by clicking on "Select File", after the selection the file is displayed in the table of the view "Save Data" view Figure 5.3.
- The user have to select the component data type "USER.DEFINED" from the combobox, in the corresponding table cell.
- After the mentioned steps above the user must press "Save" button to add the user defined data to the corresponding private branch.

##### VECTO-GIT perspective

- The user-defined files selected for upload must differ form the previously saved one in its content and file name of the respective data type.
- If the same file name of the respective data type is already stored and the content distinguish between the stored one, the user will be informed and can choose another file name.

## 5 Design

- If the user clicks the save button, the user-defined data will be added to the existing USER\_DATA folder if available, otherwise a new USER\_DATA folder will be created within the private branch.
1. `git hash-object -w <File Path>`  
Write the Git object in the repository from the given file.
  2. `git update-index --add --cacheinfo 100644 <Blob GIT ID> USER_DATA/<File Name>`  
Add the new Git object into the folder USER\_DATA to the index.
  3. `git write-tree`  
Create a new tree node from the current index.
  4. `git commit-tree <Tree GIT ID> -m ""`  
Create a new commit with the newly created tree node.
  5. `git update-ref refs/heads/<Branch Name> <Commit GIT ID>`  
Set the branch head to the newly created commit.

### 5.7.5 Standard values for a component shall be stored in the storage backend

The standard values file is equal to the component data file and contains the calculated values of a non-certificated component and thus has no certification number. This type of data will be also saved within a public branch. In case of a re-certification the component data will be added to the existing branch and the newly created commit with the component data will be tagged with the tags specified in section 5.3.

#### User perspective

- The user selects "Save Component Data" in "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Save Data" view Figure 5.3.
- In the view "Save Data" the user opens a file dialog by clicking on "Select File", after the selection the file is displayed in the table of the view "Save Data" 5.3.
- The user has to select the component data type "STANDARD.DATA" from the combobox, in the corresponding table cell.

- The chosen file will be checked per XSD for validity.
- If the file is not valid, the user gets informed for the invalid reason and the given file can not be saved.
- By pressing of the button "Save" the listed data, will be saved into VECTO-GIT repository and the user will be informed whether the save operation was successful or not.

### VECTO-GIT perspective

- During the XSD validation process the manufacturer name, the model name and XML-hash will be extracted from the selected standard values file.
- On invalidity (invalid XML-file or the Manufacturer name, the Model name and abbreviated the XML-hash combination already used), the user will be informed for the invalidity and the invalid file will be highlighted in red in the selection list.
- The disallowed characters of the model name, the manufacturer name and the XML-hash will be exchanged by an alternative character listed in the table 5.1.
- A new public branch with the listed naming convention in section 5.3, will be created with the selected standard value file as first commit and the extracted date as commit message.

The setup of the public branch is similar to Git command listing 5.7.1 with the only difference that no tags will be created.

### **5.7.6 All component-related data (component data, standard values, certificate, measured data, user-defined data) can be searched via the manufacturer name, model name, and date**

To improve the search possibility within the Git repository some key points must be fulfilled during the storing step of the component data these are

## 5 Design

already explained in more detail within section 5. To sum it up the crucial points are:

- Each component data type of the component can be assigned either only the private branch or the public branch.
- The composition of the every branch name must be follow the specified naming convention.
- The data commits within every branch(public and private) follows also a specific order.

### User perspective

- The user selects "Search Component Data" in "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Component Data Search" view Figure 5.4.
- The user enters atleast one search term in the present text fields.
- After clicking the search button, the search result will be displayed in the same view in a separate table.
- By pressing on the details button of the founded components the user gets redirected to the "Detail Component Data" view which lists all related files of a component. The detail view allows also to mark data of the component for the transmission packet.

### VECTO-GIT perspective

Before the search starts any given search term must be checked for disallowed characters which are listed in the table 5.1 and must be replaced if required. The given search terms will be performed over all branch names as specified in section 5.4.

### **5.7.7 Component data, certificate, measured data can be searched via the certification number and the Git identifier of the component data**

To make component data, certificate and measured data searchable via certification number and the Git identifier of the the component data the



following crucial points are necessary:

- The composition of the every branch name must be follow the specified naming convention in section 5.3.
- Every commit which contains the component data file must be tagged with two tags with the specified naming convention in section 5.3.

### **User perspective**

- The user selects "Search Component Data" in "VECTO-GIT-Main" view Figure 5.2 and gets redirected to the "Component Data Search" view Figure 5.4.
- The user enters the certification number and/or Git identifier of the component data.
- After clicking the search button, the search result will be displayed in the same view in a separate table.
- By pressing on the Details button of the founded components the user gets redirected to the "Detail Component Data" view which lists all related files of a component. The detail view allows also to mark data of the component for the transmission packet.

### **VECTO-GIT perspective**

Depending on the given search term, the term will be search within the certificate tag list or the Git identifier tag list.

## 5.8 GUI Design

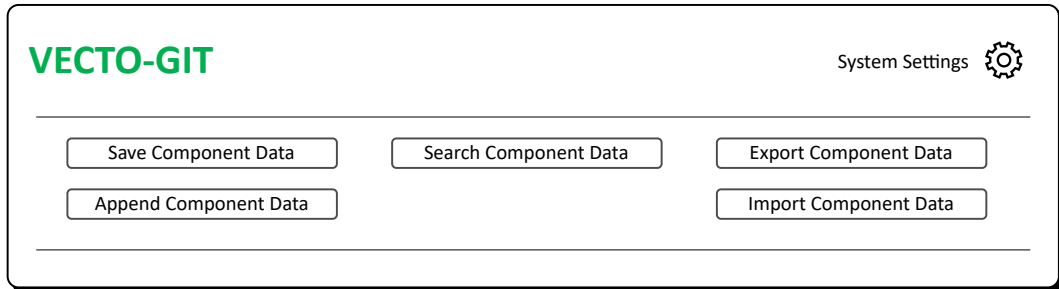


Figure 5.2: GUI Design Main View

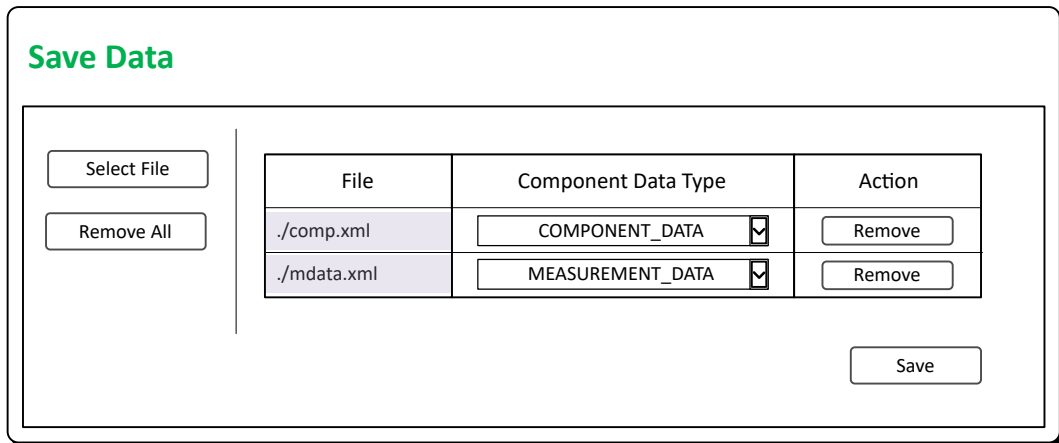


Figure 5.3: GUI Design Save Component Data View


### Component Data Search


**Manufacturer Name:**

**Model Name:**

**GIT Identifier:**

**Certification Number:**

**Certification Date:**  

 Search

#### Search Result

Manufactuer	Model	Certification Date	Component Details	Append Data
Scania	40t Truck Engine	12.10.2006	<input type="button" value="Details"/>	<input type="button" value="Append"/>
Scania	30t Truck Engine	05.05.2003	<input type="button" value="Details"/>	<input type="button" value="Append"/>
Scania	20t Truck Engine	02.04.2002	<input type="button" value="Details"/>	<input type="button" value="Append"/>

Figure 5.4: GUI Design Search Component Data View

# 6 Implementation

This chapter gives an overview of the actual implementation status of the designed prototype for the management of the component certification data.

## 6.1 Overview

For the development of the prototype the programming language C-Sharp (C#) in the .NET Version 4.5.2 was used. The programming language was chosen to facilitate the integration of the prototype into the VECTO simulation tool if required.

In case of an integration, the prototype works like a portable database which contains all relevant vehicle component files for a simulation. To fetch data directly from the prototype database the interface which is currently connected to the Graphical User Interface (GUI) can be adapted for this case.

For the development of the GUI the Windows Presentation Foundation (WPF) was used with Model View ViewModel (MVVM) design pattern. To execute Git commands and to administrate the Git repository of the prototype the .Net library LibGit2Sharp<sup>1</sup> was used. The LibGit2Sharp library is C# wrapper for the libgit2<sup>2</sup> library which is a pure C implementation of the Git core methods. Both libgit2 and LibGit2Sharp are still under development, which is the reason that not all Git commands are supported yet. The architecture and functioning of the prototype is described in more detail in the following sections.

---

<sup>1</sup>LibGit2Sharp <https://github.com/libgit2/libgit2sharp>

<sup>2</sup>libgit2 <https://libgit2.org/>

## 6.2 Architecture

The software architecture as shown in Figure 6.1 of the prototype VECTO Git can essentially be split up into two main parts: the GUI part and the repository management part. The ComponentDataManagement class implements the interface IComponentDataManagement which is shown in the listing 6.1 and represents the connection class between these main parts. This interface defines the required functions which were defined in the requirement list 5.2 of the design chapter.

```

1  ComponentDataCommit VerifySaveComponentData(
2      string compDataFilePath , Branch stdValuesBranch ,
3      string stdValuesFilePath , bool validateXml );
4
5  StandardDataCommit VerifySaveStandardValueFile(
6      string filePath , bool validateXml );
7
8  CertificateDataCommit VerifySaveCertificateByComponentFile(
9      string certFilePath , string compFilePath ,
10     Branch publicBranch );
11
12 MeasurementDataCommit VerifySaveMeasurementData(
13     ComponentToSave componentToSave );
14
15 UserDataCommit VerifySaveUserData(
16     ComponentToSave componentToSave );
17
18 IEnumerable<VectoComponent> SearchVectoComponentsByTerms (
19     VectoSearchTerms searchTerms );
20
21 IEnumerable<VectoComponent> SearchAllVectoComponents ( );

```

Listing 6.1: IComponentDataManagement interface

## 6 Implementation

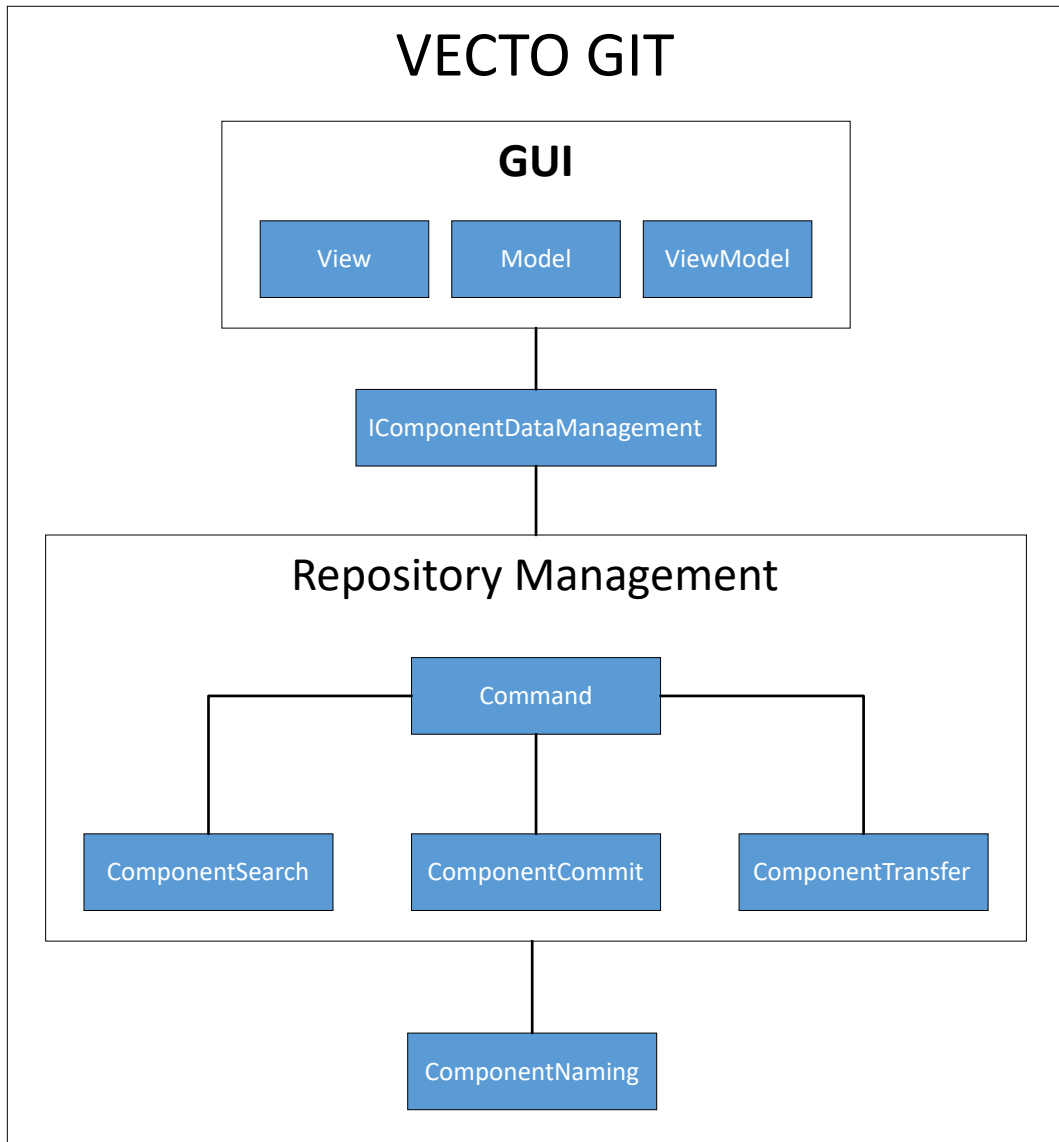


Figure 6.1: VECTO GIT prototype architecture

### 6.2.1 Repository Management Implementation

The repository management part of the prototype architecture as shown in Figure 6.1 consists of four overall parts in which all classes with similar tasks are grouped together. These four groups and their containing classes and tasks are described in more detail in the following listing.

- **Command**

The command group contains all classes responsible for executing Git commands via the LibGit2Sharp library, each class in these group covers one git command. Furthermore, each class is derived from the abstract Command class as shown in the listing 6.2. This base class ensures that the respective command object has access to the currently selected repository and that each command class implements the respective Git command within the execution method.

- **ComponentCommit**

The component commit group uses the implemented command classes to execute the commit of the respective component data type in its structured form, as defined in the design and data partitioning section 5.3.

- **ComponentTransfer**

The component transfer group contains the TransferHandler class, which are responsible for file handling to enable transport between repositories.

- **ComponentSearch**

The component search group contains the search classes, which are responsible for the search behaviour within the repository, as defined in the search behaviour section 5.4.

## 6 Implementation

```
1 public abstract class Command
2 {
3     protected Repository repository;
4
5     protected Command( Repository repository )
6     {
7         this.repository = repository.CheckValidRepository();
8     }
9     protected abstract void Execute();
10 }
```

Listing 6.2: IComponentDataManagement interface

### 6.2.2 GUI

For the implementation of the Graphical User Interface (GUI) in WPF the Model View ViewModel (MVVM) pattern was used. The MVVM consist of the following three component types:

- **Model**  
Represents the data access layer of the content that is displayed to the user by the usage of the view model.
- **View**  
Represents the view for the user where the data and all User Interface (UI) elements will be shown. Events (click, select , etc. ) of GUI elements as well as data changes of properties will be forwarded to the ModelView by the use of the data binding mechanism of WPF.
- **ViewModel**  
Represents the interaction class between model and view. The view model provides data by public properties which will be bound by data binding to the view. On change of the data, it will be forwarded to the respective model. The view model also implements the ICommand interface to enable an interaction with the UI elements (Button, Checkbox, etc.) in the view.



The implemented GUI of the VECTO Git prototype consists of six different views which fulfil different purposes. The intended use of these are explained in more detail in the following enumeration:

1. **Save component data view**

The save component data view shown in Figure 6.2 allows to save a new component with the respective vehicle component file. To save the vehicle component file the respective data type must be selected from the combobox. The vehicle component file can be stored in combination with other files like for example the related measurement files.

2. **Search component view**

The search component view shown in Figure 6.3 allows to search over the existing components within the repository. The entered search terms will be correlated by a logical AND. The available search terms are the manufacturer name, the model name, the Git Id, the certification number and the certification date.

3. **Append component data view**

The append component data view shown in Figure 6.4 allows to add new files to an existing component. It also allows to view the existing files within the component.

4. **Detail component data view**

The detail component data view shown in Figure 6.5 allows to view the existing files within the component. It also allows to preselect existing component files for transfer to other repositories.

5. **Export component data view**

The component data export view shown in Figure 6.6 contains the list of preselected component files that can be exported to an archive file, which will be used for the transfer to another repository.

6. **Import component data view**

The import component data view shown in Figure 6.7 enables to import data into the repository from an archive file which was created by the export function.

## 6 Implementation

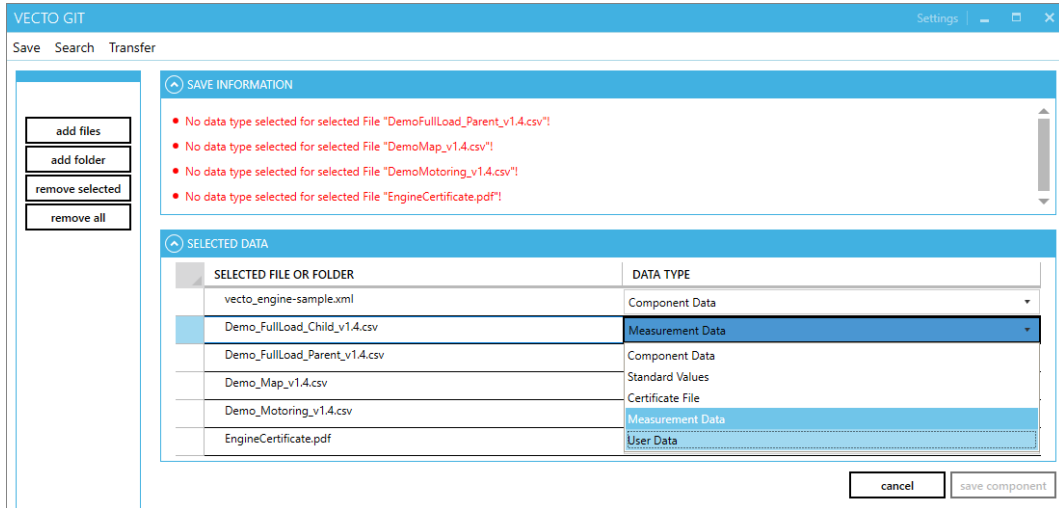


Figure 6.2: VECTO Git prototype save component data view

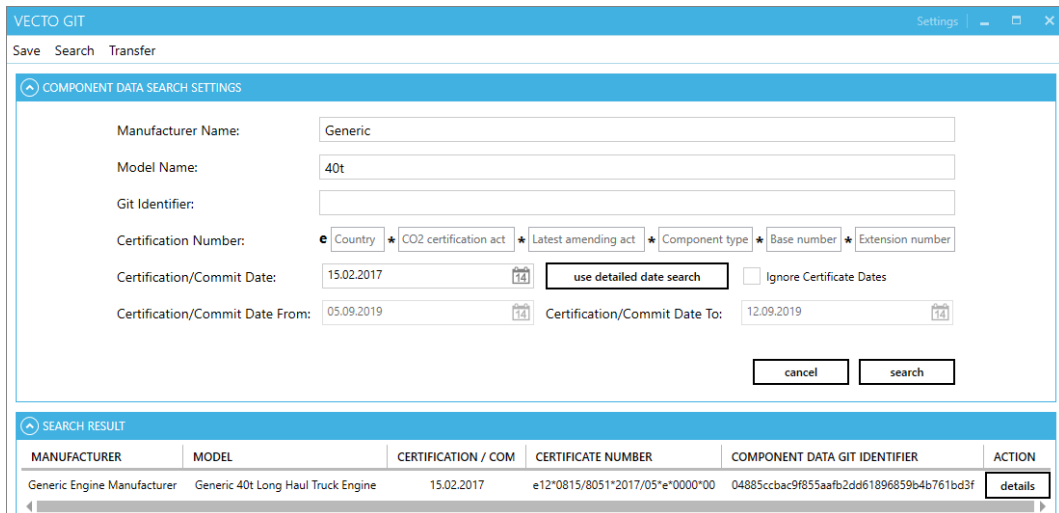


Figure 6.3: VECTO Git prototype search component view

6.2 Architecture

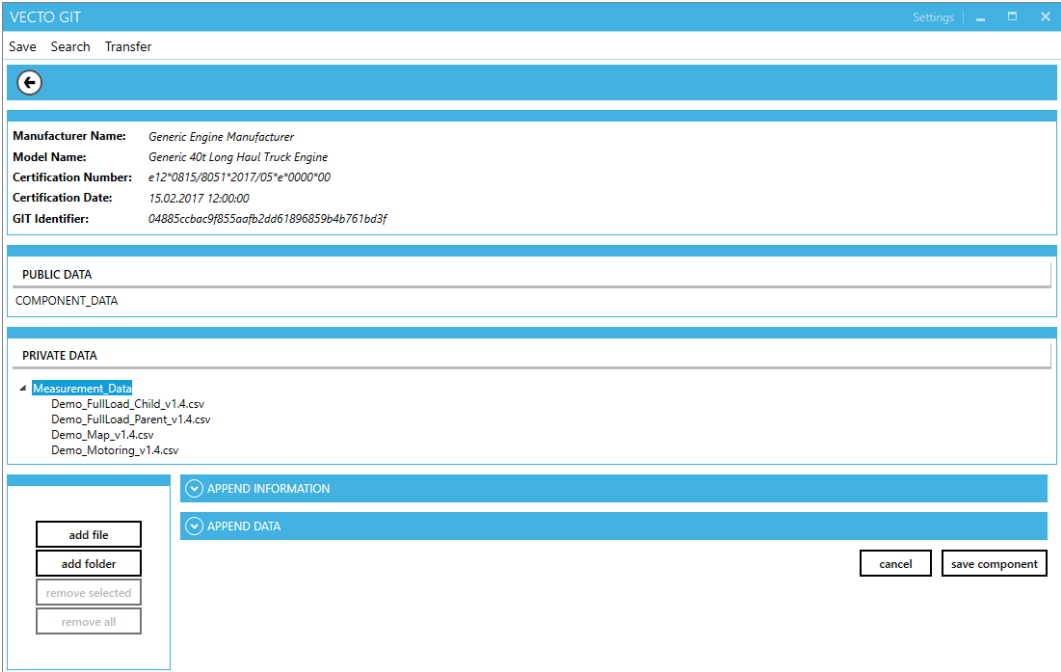


Figure 6.4: VECTO Git prototype append component data view

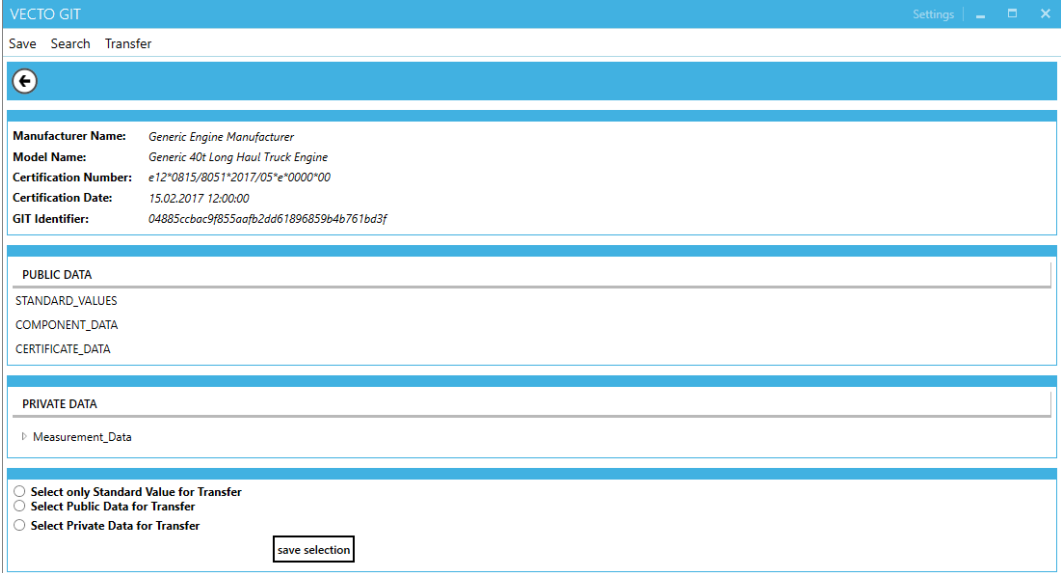


Figure 6.5: VECTO Git prototype detail component data view

## 6 Implementation

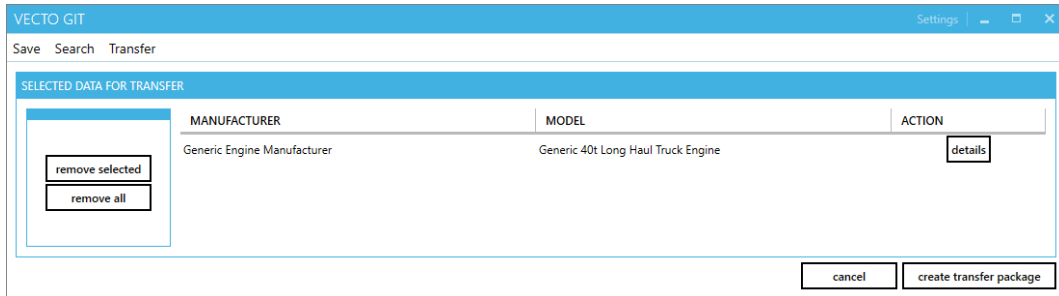


Figure 6.6: VECTO Git prototype export component data view

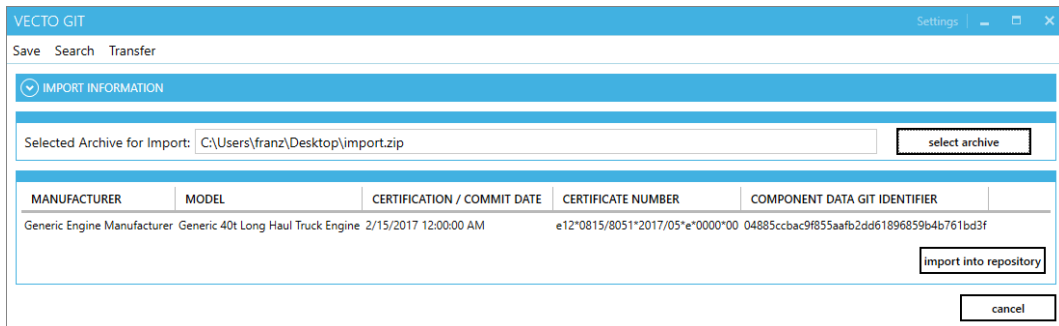


Figure 6.7: VECTO Git prototype import component data view

## 7 Conclusion

This thesis introduced a way how the distributed version control system Git can be adapted to use it as a database for vehicle certification data. The key to this adaptation was to find a common structure which can be linked with the data to be stored and can also be realized with the existing Git functionalities.

The common structure adds additional meta information to the stored data which gets lost due to the storage into the content addressed storage system of Git. If a suitable structure can be found depends on the data and the search requirements which should be supported. It has to be considered if the additional meta information is not too much overhead and a database would be the better solution.

# Bibliography

- [1] EU Comission. *2020 climate & energy package*. URL: [https://ec.europa.eu/clima/policies/strategies/2020\\_en](https://ec.europa.eu/clima/policies/strategies/2020_en) (visited on 09/02/2019) (cit. on p. 1).
- [2] EU Comission. *Vehicle Energy Consumption calculation TOol - VECTO*. URL: [https://ec.europa.eu/clima/policies/transport/vehicles/vecto\\_en#tab-0-1](https://ec.europa.eu/clima/policies/transport/vehicles/vecto_en#tab-0-1) (visited on 09/02/2019) (cit. on p. 2).
- [3] EU Comission. *VECTO Workshop - Overview*. URL: [https://ec.europa.eu/clima/sites/clima/files/transport/vehicles/vecto/201811\\_overview\\_en.pdf](https://ec.europa.eu/clima/sites/clima/files/transport/vehicles/vecto/201811_overview_en.pdf) (visited on 09/02/2019) (cit. on pp. 5–7, 10).
- [4] EU Comission. *VECTO tool development: Completion of methodology to simulate Heavy Duty Vehicles' fuel consumption and CO2 emissions*. URL: [https://ec.europa.eu/clima/sites/clima/files/transport/vehicles/docs/sr7\\_lot4\\_final\\_report\\_en.pdf](https://ec.europa.eu/clima/sites/clima/files/transport/vehicles/docs/sr7_lot4_final_report_en.pdf) (visited on 09/02/2019) (cit. on pp. 7, 9).
- [5] John Hammink. *The Types of Modern Databases*. Aug. 2019. URL: <https://www.alooma.com/blog/types-of-modern-databases> (visited on 08/10/2019) (cit. on pp. 12, 13, 15).
- [6] Edgar Frank Codd. "A relational model of data for large shared data banks." In: *Communications of the ACM, Volume 13, Issue 10*, pp. 377–387. URL: <https://dl.acm.org/citation.cfm?id=362685> (cit. on p. 11).
- [7] Ameya Nayak, Anil Poriya, and Dikshay Poojary. "Article: Type of NOSQL Databases and its Comparison with Relational Databases." In: *International Journal of Applied Information Systems* 5.4 (Mar. 2013). Published by Foundation of Computer Science, New York, USA, pp. 16–19 (cit. on p. 14).

- [8] Eric Brewer. "A Certain Freedom: Thoughts on the CAP Theorem." In: *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. PODC '10. Zurich, Switzerland: ACM, 2010, pp. 335–335. ISBN: 978-1-60558-888-9. DOI: 10.1145/1835698.1835701. URL: <http://doi.acm.org/10.1145/1835698.1835701> (cit. on p. 16).
- [9] Syed Sadat Nazrul. *CAP Theorem and Distributed Database Management Systems*. 2019. URL: <https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e> (cit. on p. 16).
- [10] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services." In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: <http://doi.acm.org/10.1145/564585.564601> (cit. on p. 17).
- [11] *CAP-Theorem*. URL: <https://de.wikipedia.org/wiki/CAP-Theorem> (visited on 09/02/2019) (cit. on p. 17).
- [12] Mike Chapple. *Abandoning ACID in Favor of BASE in Database Engineering*. URL: <https://www.lifewire.com/abandoning-acid-in-favor-of-base-1019674> (visited on 09/02/2019) (cit. on pp. 18, 19).
- [13] *Content-addressable storage*. URL: [https://en.wikipedia.org/wiki/Content-addressable\\_storage](https://en.wikipedia.org/wiki/Content-addressable_storage) (visited on 09/02/2019) (cit. on p. 20).
- [14] Dipl Medieninformatiker BA Daniel Kuhn. *Distributed Version Control Systems*. 2010 (cit. on p. 21).
- [15] Ben Straub Scott Chacon. *Pro Git*. second edition. Apress, Berkeley, CA, 1993 (cit. on pp. 22–27, 29).
- [16] Neil McAllister. *Linus Torvalds' BitKeeper blunder*. URL: <https://www.infoworld.com/article/2670360/linus-torvalds--bitkeeper-blunder.html> (visited on 09/02/2019) (cit. on p. 25).
- [17] Brent Laster. *Professional Git*. John Wiley & Sons, Inc., 2017 (cit. on pp. 26, 27, 31).
- [18] Marc Stevens et al. "The first collision for full SHA-1." In: *Annual International Cryptology Conference*. Springer. 2017, pp. 570–596 (cit. on p. 33).

## Bibliography

- [19] Jonathan Corbet. *Moving Git past SHA-1*. URL: <https://lwn.net/Articles/715716/> (visited on 09/02/2019) (cit. on p. 33).