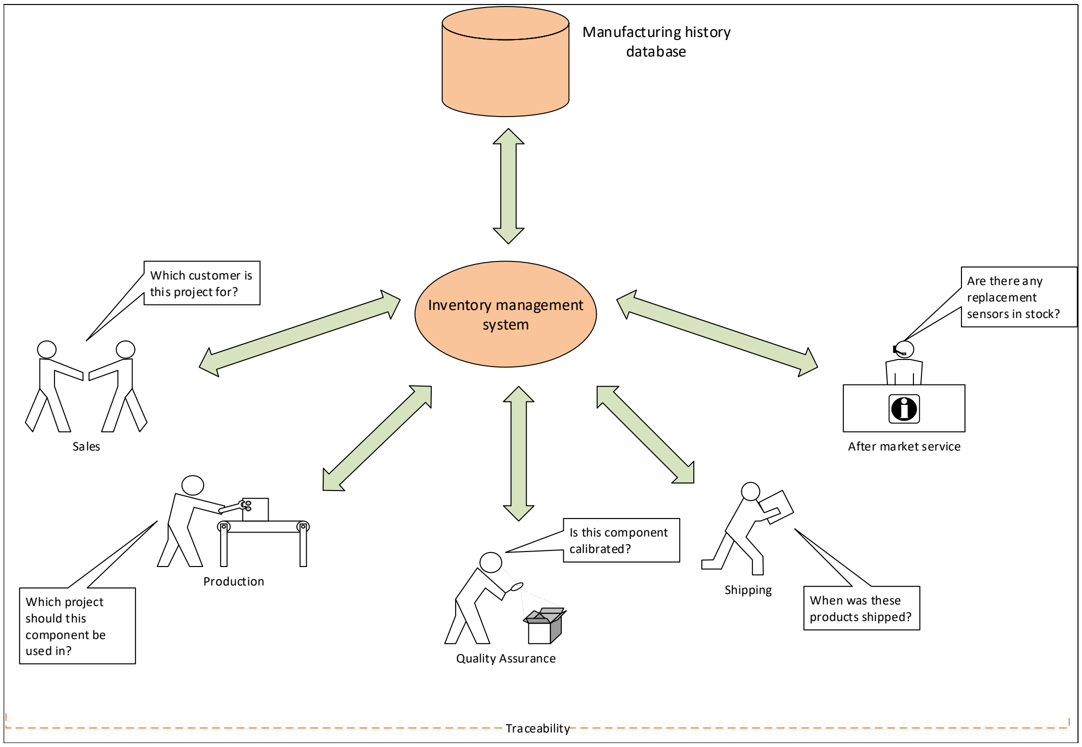
University of South-Eastern Norway

www.usn.no

PRH612-1 19V Bachelor's thesis

# Project and inventory manager - creating an extensible application in C++ and a future-oriented database with PostgreSQL


Manufacturing history database
Inventory management system
Which customer is this project for?
Sales
Are there any replacement sensors in stock?
After market service
Production
Which project should this component be used in?
Is this component calibrated?
Quality Assurance
Shipping
When was these products shipped?
Traceability

IA6-10-19

Faculty of Technology, Natural Sciences and Maritime Sciences
Campus Porsgrunn

University of
South-Eastern Norway

www.usn.no

**Course**: SCE4006 Project, 2019/PRH612-1 19V Bachelor's thesis

**Title**: Project and inventory manager – creating an extensible application in C++ and a future oriented database with PostgreSQL.

This report forms part of the basis for assessing the student's performance in the course.

**Project group:** IA6-10-19 **Availability:** Open

**Group participants:** Martin Holm
Espen Buø
Sindre Eiken

**Supervisor:** Hans-Petter Halvorsen

**Project partner:** Scanjet Ariston AS

**Approved for archiving:** ________________________________________

**Summary:**

Scanjet Ariston AS is a company that delivers and develops a universal and flexible solution for tank monitoring and control. To ensure traceability of their products, a database is used to store information about each product. Today Scanjet Ariston AS uses a database based on Clarion, a file-based proprietary solution. The company evaluates Clarion to have an uncertain future and concludes that it is a considerable risk to continue using the current system. Scanjet wants to develop a new solution which is extensible, future-oriented and created in a well-known development environment.

The main objectives in this project consists of creating a new database, an associated application and migration of all existing data to the new solution. Additional tasks include creating a user manual for the application and test documentation.

This project has chosen an agile development methodology called Scrum as a guiding principle during the development process.

A new database is developed and documented during the project. Using Devart for Excel and Python, methods for migrating the existing data to the new database has been developed. A guide has been written to assist Scanjet Ariston AS with the migration after this project.

A desktop application prototype is developed for accessing this database. This application remains unfinished but consists of three semi-independent software layers that can be developed further or reused in a different setting. The application incorporates flexible solutions such as a general SQL interface for PostgreSQL, and a platform independent configuration layer.

*The University of South-Eastern Norway takes no responsibility for the results and conclusions in this student report.*

University of
South-Eastern Norway

www.usn.no

**Emne**: SCE4006 Project, 2019/PRH612-1 19V Bacheloroppgave

**Tittel**: Project and inventory manager – creating an extensible application in C++ and a future oriented database with PostgreSQL.

Denne rapporten utgjør en del av vurderingsgrunnlaget i emnet.

**Prosjektgruppe:** IA6-10-19 **Tilgjengelighet:** Åpen

**Gruppedeltakere:** Martin Holm
Espen Buø
Sindre Eiken

**Veileder:** Hans-Petter Halvorsen

**Prosjektpartner:** Scanjet Ariston AS

**Godkjent for arkivering:** ______________________________

**Sammendrag:**

Scanjet Ariston AS er et firma som leverer og utvikler en universell og fleksibel løsning for overvåkning og styring av tanker om bord på skip. For å sikre sporbarhet for produktene deres, brukes det en database til å lagre informasjon om hvert produkt. I dag bruker Scanjet Ariston AS en database basert på Clarion, som er en filbasert proprietær løsning. Scanjet vurderer Clarion til å ha en usikker fremtid, og konkluderer dermed at det er en risiko å fortsette med løsningen de har i dag. Firmaet vil derfor utvikle en ny løsning som er fleksibel, fremtidsrettet og utviklet i et kjent rammeverk.

Oppgaven går ut på å lage en ny database, en tilhørende applikasjon og migrering av eksiterende data fra det gamle til det nye systemet. I tillegg skal det utarbeides en brukermanual og testdokumentasjon.

Under dette prosjektet er det valgt å bruke en fleksibel utviklingsmetode kalt Scrum som et ledende prinsipp gjennom utviklingsprosessen.

En ny database er utviklet og dokumentert under prosjektet. Ved hjelp av Devart for Excel og Python er metoder for migrering av eksisterende data utarbeidet. En veiledning er skrevet for å hjelpe Scanjet Ariston AS med migreringen etter dette prosjektet.

En prototype av applikasjonen er utviklet for databasen. Applikasjonen er utferdig, men består av tre delvis uavhengige programvarelag som kan utvikles videre eller gjenbrukes i andre omstendigheter. Programmet inneholder fleksible løsninger som et generelt SQL-grensesnitt for PostgreSQL, og en plattformuavhengig konfigurasjon.

*The University of South-Eastern Norway takes no responsibility for the results and conclusions in this student report.*

# Preface

This bachelor thesis is written by three students at the University of South-Eastern Norway, attending the educational program for information and automation technology in the 6. Semester. The report is a result of a project done as a bachelor thesis in cooperation with Scanjet Ariston AS. At the end of the project, all contents of this report and developed software will be handed over to the project partners. The faculty will receive this report and all appendixes which are not confidential.

The picture used on the front page is made by the group using MS Visio.

The tools used during the project includes the following: MS Word, MS Excel, MS Visio, MS Visual Studio, MS Visual Studio Code, Toad Data Modeler, PGadmin, RAD Studio, InnoSetup, JetBrains PyCharm Community Edition, pyinstaller, draw.io, Oracle VirtualBox, SoftVelocity Database Scanner, Maple and Windows 10.

All program code is stored on a memory stick which is delivered to Scanjet Ariston AS.

To understand the full context of this thesis, basic knowledge about programming of applications and databases is an advantage.

The bachelor group would like to express their gratitude and thank the project partners for their guidance and cooperation during the project, especially Kai Ebersten (Scanjet Ariston AS) and Erik Syvertsen (Scanjet Ariston AS).

Porsgrunn, 14.05.19

# Nomenclature

| | | |
|---|---|---|
| Attribute | - | A database field containing a value of a column. (Database) |
| Attribute | - | A key-value pair written within an element tag. (XML) |
| CMD | - | Windows command shell |
| Cursor | - | SQL object that describes the values of an entry in a database |
| CSV file | - | Text file with comma-separated values |
| DOM | - | Document Object Model. |
| Entity | - | A table in a database. |
| ER | - | Entity Relation. |
| GUI | - | Graphical User Interface. |
| IDE | - | Integrated Development Environment. |
| PO | - | Purchase Order. |
| PgDAC | - | PostgreSQL Data Access Components, plugin for Rad Studio. |
| VCL | - | Visual Component Library |
| XML | - | Extensible Markup Language, with the associated file extension .xml. |

# Contents

# 1 Introduction

## 1.1 Background and problem description

Scanjet Ariston AS is a company that delivers and develops a universal and flexible solution for tank monitoring and control. To ensure traceability of their products, a database is used to store information about each product. Today Scanjet Ariston AS uses a database based on Clarion, which is a file-based proprietary solution. Clarion is a programming language developed by SoftVelocity and is a business-oriented software, which means its focused less on the technology and more on the business of software. [1]

Scanjet Ariston AS evaluates Clarion to have an uncertain future and concludes that it is a considerable risk to continue using the existing solution. Additionally, expanding the current solution can be complicated due to Scanjet Ariston's lack of experience and knowledge about Clarion as a programming language. As Scanjet Ariston AS has departments localized in several countries such as China, Korea, Indonesia, Norway and Sweden there is now also a need for a client/server-based solution. The company wants to develop a new solution which is extensible, future-oriented and created in a well-known development environment.

## 1.2 Project objectives

The main objectives in this project consists of developing a new database with an associated desktop application, as well as migration of data from the existing system. The intermediate objectives include creating requirements based on the existing system and customer needs, a user manual for the application and test documentation. Additional tasks consist of creating a web application, implement functions to program Scanjet Ariston's components and generation of configuration files for the customers products. *Figure 1-1* illustrates an overview of which tasks the new system should perform.

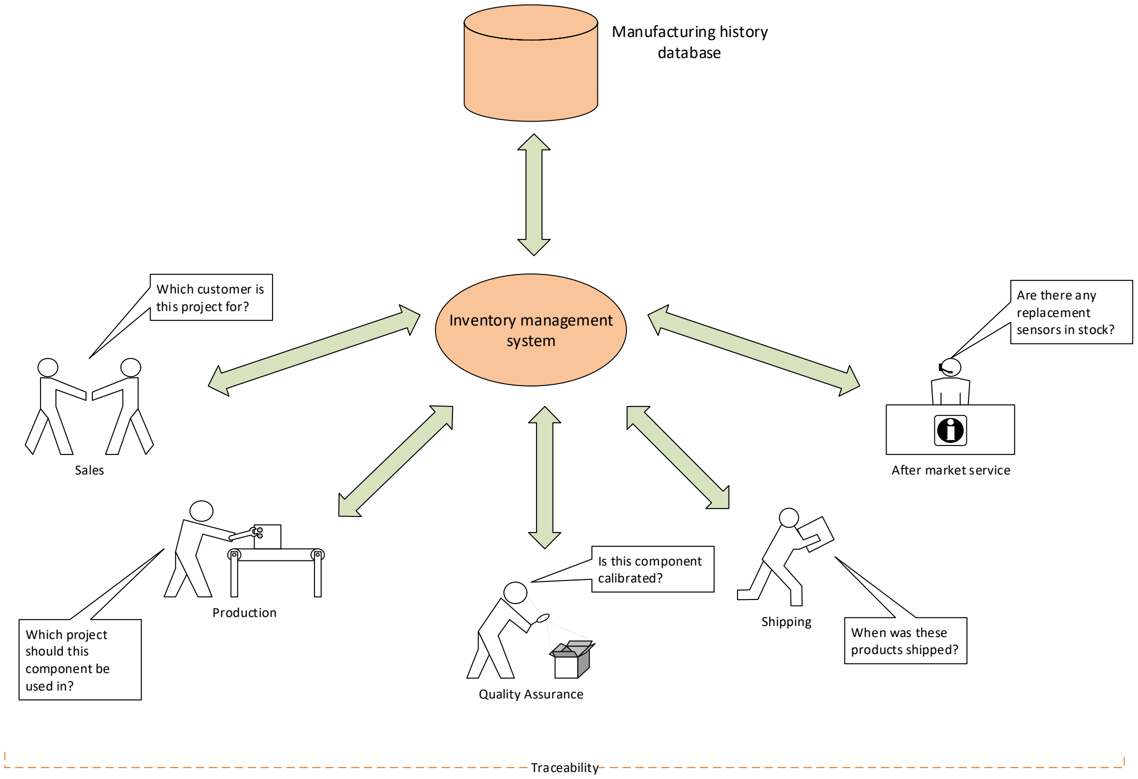

Manufacturing history database
Inventory management system
Which customer is this project for?
Sales
Are there any replacement sensors in stock?
After market service
Production
Which project should this component be used in?
Is this component calibrated?
Quality Assurance
Shipping
When was these products shipped?
Traceability

*Figure 1-1: System overview*

## 1.3 Methods

This project has chosen Scrum as a guiding principle rather than methodology as recommended by the project supervisor. Scrum is an agile methodology, meaning it prefers face-to-face communication and improving on working software, among other things. This project's take on Scrum includes a total of three release iterations with similar time constraints, weekly meetings with customer and project supervisor, and the use of living documentation. Most importantly are workloads broken down into tasks of no more than a few hours. [2]

## 1.4 Scope

As the task described and discussed with Scanjet Ariston is large, constraints are needed. The task is limited to include development of a new database, an associated application and migration of data from the existing system. The task includes development of user authentication control, a user manual and test documentation. Creating a web application, programming of components and Scanjet's configuration file generation are excluded from the project scope.

## 1.5 Report structure

The introduction chapter gives information about the background of the customer Scanjet, and the bachelor group. It then describes the project objectives, work methodology and the scope of the project.

Chapter two describes the existing system and the planned solution.

Chapter three describes the requirements and design for the new database.

Chapter four describes the requirements for migrating data and the designed solution.

Chapter five describes the requirements and design for the new application.

Chapter six describes the status of prototypes and solutions for the new system.

Chapter seven is a discussion regarding problems, methods and solutions in the project.

Chapter eight is a summary of the project.

# 2 Existing and planned solution

Today, Scanjet Ariston AS uses a software based on Clarion to store information about projects and products. However, there is now a need for a better system which ensures greater traceability of their products. Due to Scanjet's lack of experience and knowledge about Clarion as a programming language, expanding the existing solution can be complicated. Scanjet Ariston AS evaluates Clarion to have an uncertain future and want to develop a new system in a well-known development environment. This chapter describes the solution used today and the philosophy and ambitions for the new software.

## 2.1 Existing solution

Scanjet Ariston AS uses a solution today which consists of a database and an associated application. This subchapter gives the reader an overview of how the existing solution operates today.

### 2.1.1 Existing database

This subchapter will give a picture of how the old database is built and how data is related. The existing database is created by an earlier employee of Scanjet without much education in database modeling. It was created in Clarion, which is an IDE for Clarion (language) databases. It's created as a relational database but does not seem to enforce foreign keys. There are relations between all tables but two, these tables contain very little data and doesn't appear to be in use. *Figure 2-1* shows an overview of the current database, here the tables are generalized for an easier overview, and will be expanded upon throughout the subchapter.

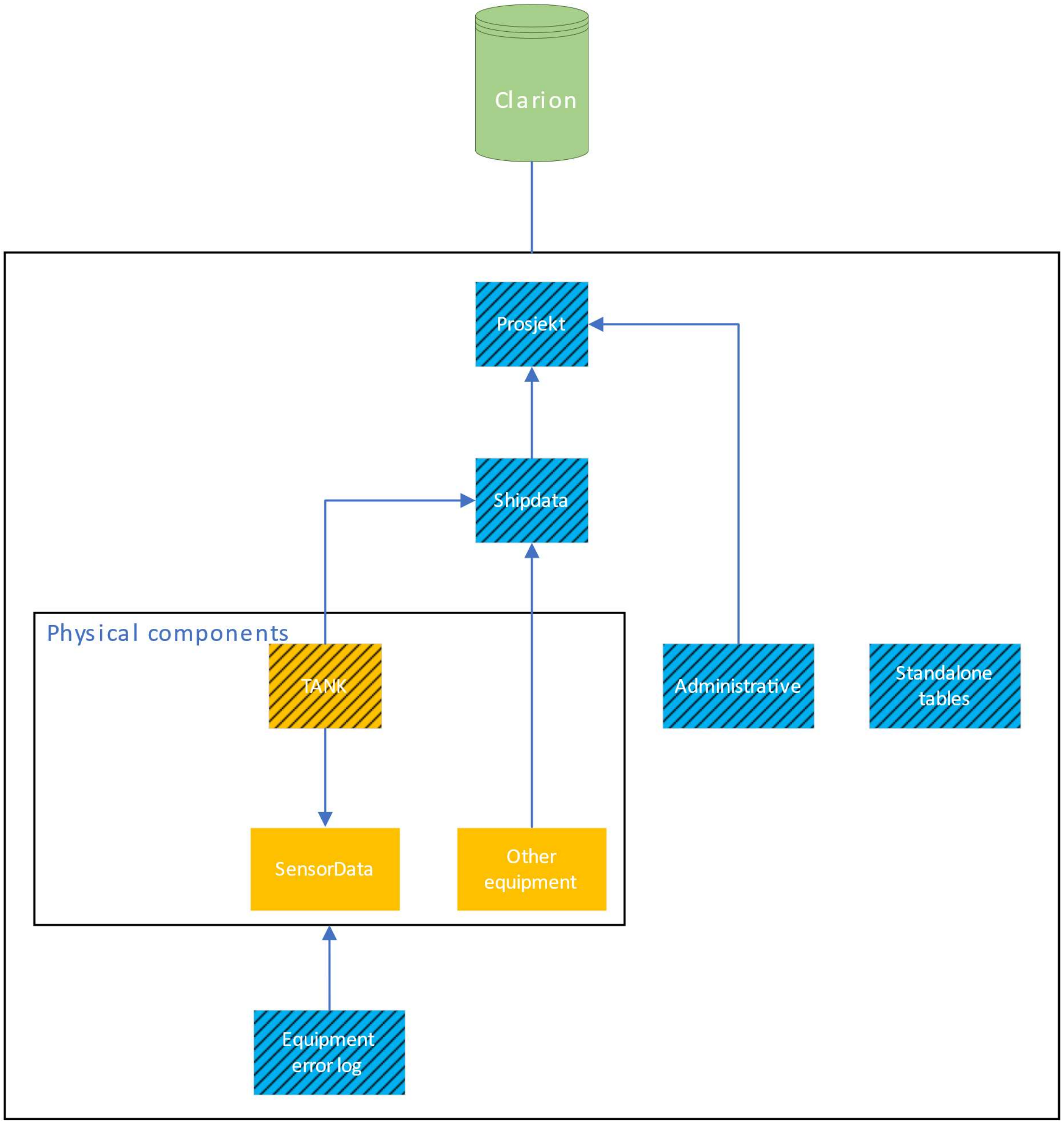


*Figure 2-1 High level overview of the existing database*

The model will be split up to give a full description of the database. In the coming figures a table will have its description as in *Figure 2-2*. It will contain a table name and may contain none to several foreign key column names, while pointing to the tables where these keys are found. In *Figure 2-2*, to the left (not hatched), a table describing a physical unit, will be referred to as a component table. Centre (hatched with lighter color), a physical unit, but not regarded as a component throughout the report. Right (hatched with darker color), other tables will be described where they’re used.

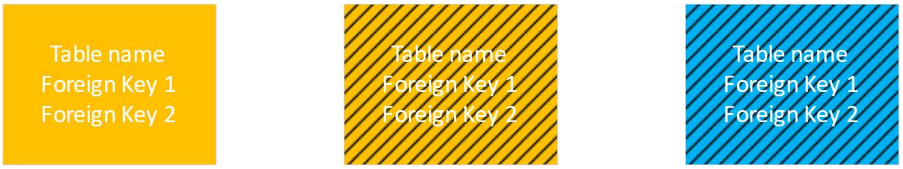


*Figure 2-2 Description of tables and how to separate them.*

Using *Figure 2-3* as an example, Tank uses a value from *Shipdata*, ship number (*ShipNo*) to tell which project it's connected to. It also uses serial numbers (*serienummer*) from *Sensordata* to tell which sensors are connected to a given tank.

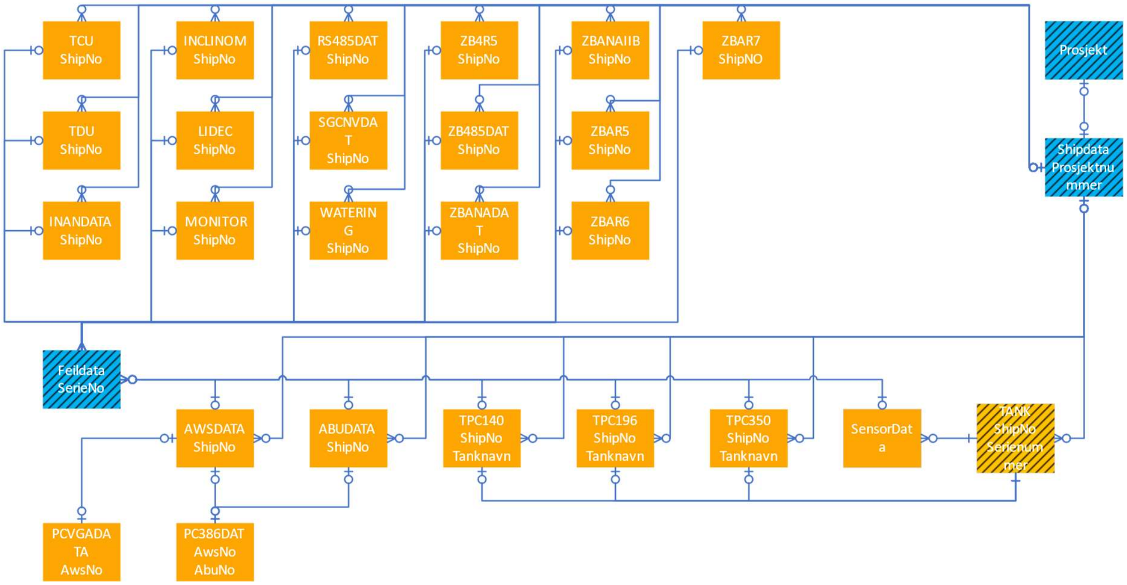


*Figure 2-3 Tables of physical components and how they are connected to a project.*

The tables considered *other tables,* are *Shipdata, Feildata* (error data), and *prosjekt* (project). *Shipdata* gives description of which ship components are installed in, what project a ship is connected to, delivery and warranty dates, and software information. *Prosjekt* (Project) gives a name and number to a project. *Feildata* (error data) contains records of detected faults in components.

Below in *Figure 2-4* the administrative tables are expanded on from *Figure 2-1*. It describes communication logs, contact information, drawings and offers. These are all connected to a project through *journal*.

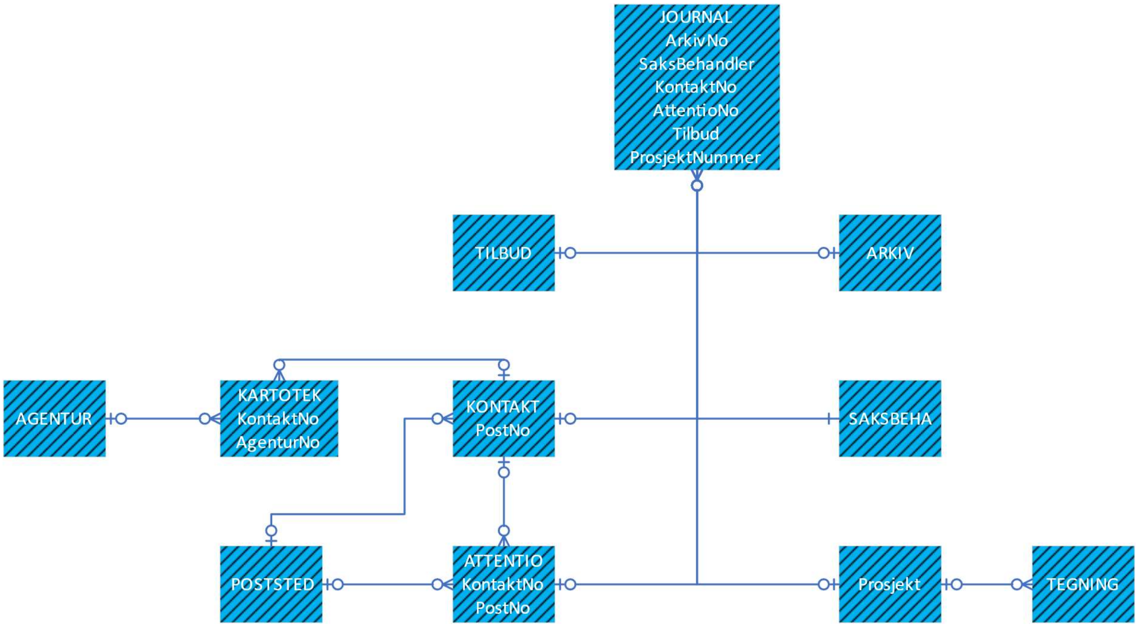


*Figure 2-4 Administrative tables of the Clarion database*

*Figure 2-5* shows the remaining tables of the Clarion database. They contain information about updates to components (*Updateda*) and some non-context data about components (*Reportte*).

*Figure 2-5 Tables in Clarion database which is not related to other tables*

### 2.1.2 Existing application

To manipulate records in the database, Scanjet Ariston AS uses an application called *Shipdata*. The application contains different features to add, view, edit and delete information about the company's projects and products. *Figure 2-6* shows the start page of the application containing the main menu.

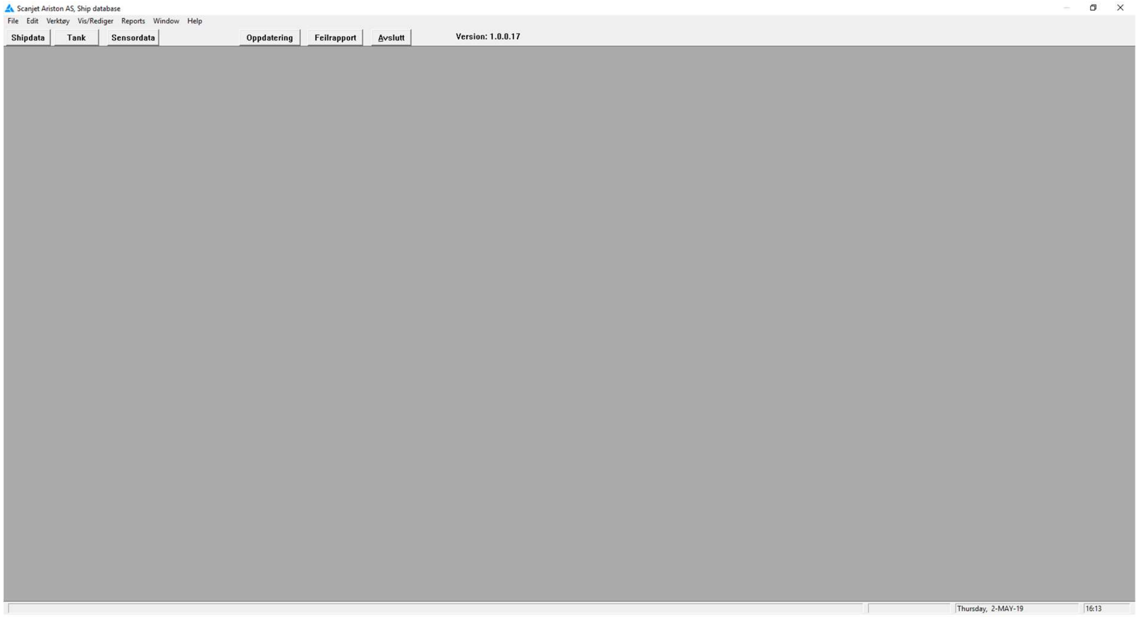


*Figure 2-6: Shipdata: Start page.*

The main menu consists of five main functions shown in *Figure 2-7*, which are *shipdata*, *tank*, *sensordata*, *oppdatering* (Updates) and *feilrapport* (error reports*)*. The function *Shipdata* leads to an overview of all central information regarding project records in the database. The next function *Tank* leads to a list of tanks connected to a selected project. Most sensor sold by Scanjet Ariston is stored in the database and accessed through the function *sensordata*, other may be found under *vis/rediger* which is described in the next section. The fourth function *oppdatering* contains information about each update to components or articles. To trace any errors of products, the last main function *feilrapport* leads to list of all errors registered and functions to register new errors.

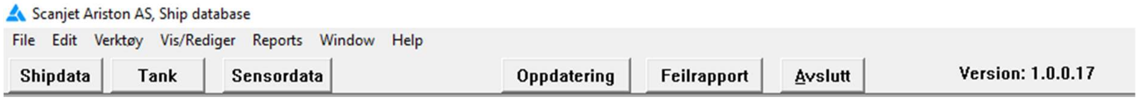

Scanjet Ariston AS, Ship database
File Edit Verktøy Vis/Rediger Reports Window Help
Shipdata
Tank
Sensordata
Oppdatering
Feilrapport
Avslutt
Version: 1.0.0.17

*Figure 2-7: Shipdata: Main menu.*

*Shipdata* has a menu bar containing additional functions. There are three important sub menus shown in *Figure 2-8*, which are *vis/rediger* (1), *verktøy* (2) and *reports* (3). *Vis/rediger* (1) contains functionality to view records in the database and edit them. The second function *verktøy* (2), which stands for tools, is where the user can add new sensors and sensor data, generate configuration files, search for sensors, copy tank lists and adjust communication settings. *Reports* (3) contains functions to generate reports based on records in the database e.g. a list of sensors for one system.

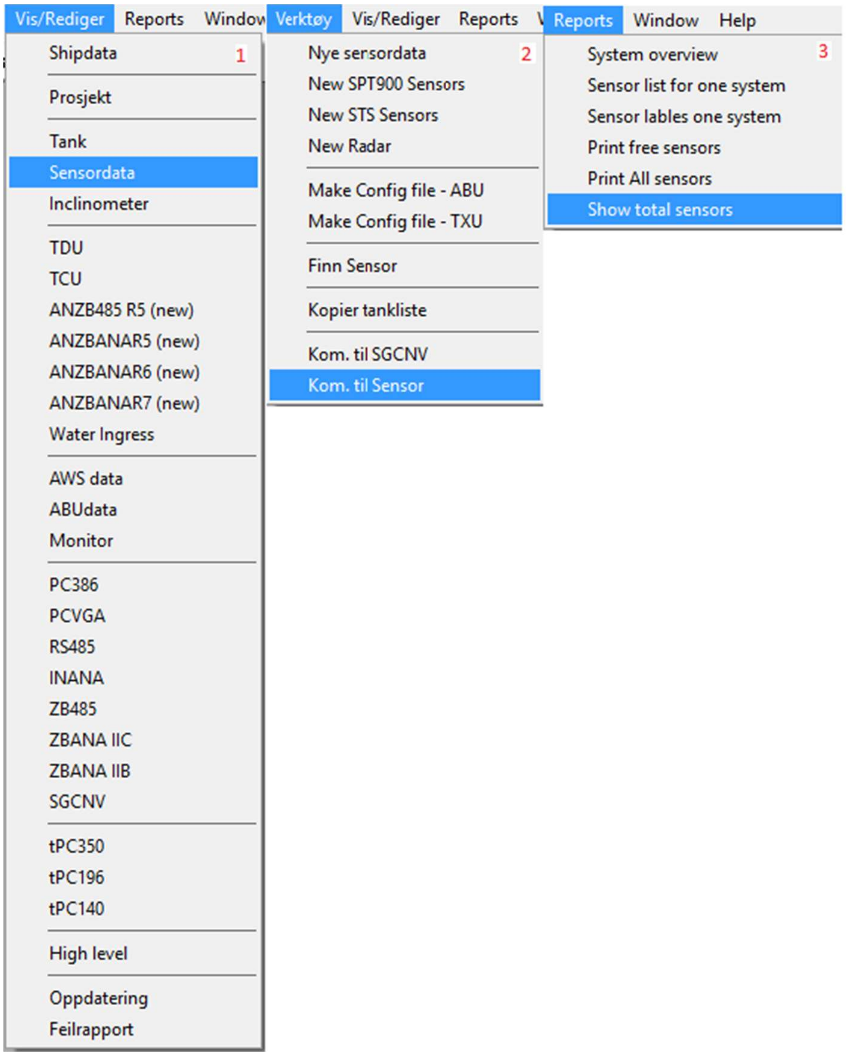
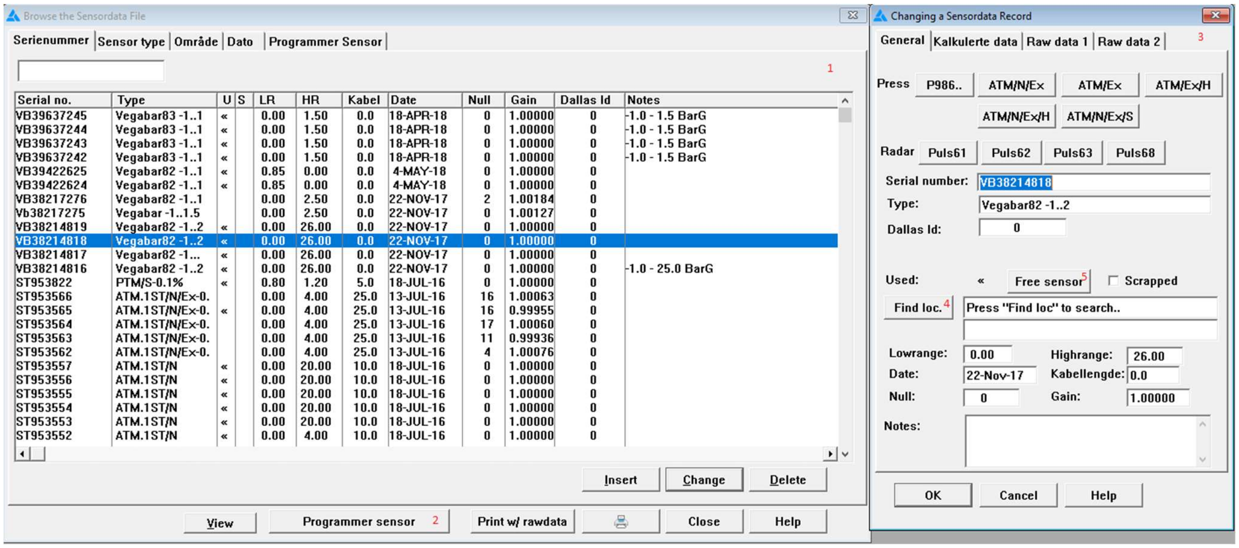


*Figure 2-8: Shipdata: menu bar functions.*

One of the most used features of the application is *sensordata* (1) shown in *Figure 2-9*. When a new order is being assembled, each sensor is stored through *sensordata* or the sub menu *vis/rediger*. Users may also change or delete records. An important function within *sensordata* is *programmer sensor* (2) which is used to program Scanjet Ariston's sensors.

By clicking an item in the list shown in *Figure 2-9*, a window for changing the record is presented (3). The window shows details about a sensor record and enables the user to change or add data to the record. As well as editing data, the users may also find where the sensor is located on a ship by using the function *find loc* (4). In some cases, its needed to disconnect sensors from the associated project, which is done by clicking *free sensor (5)*.



| Serial no. | Type | U | S | LR | HR | Kabel | Date | Null | Gain | Dallas Id | Notes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| VB39637245 | Vegabar83 -1..1 | « | | 0.00 | 1.50 | 0.0 | 18-APR-18 | 0 | 1.00000 | 0 | -1.0 - 1.5 BarG |
| VB39637244 | Vegabar83 -1..1 | « | | 0.00 | 1.50 | 0.0 | 18-APR-18 | 0 | 1.00000 | 0 | -1.0 - 1.5 BarG |
| VB39637243 | Vegabar83 -1..1 | « | | 0.00 | 1.50 | 0.0 | 18-APR-18 | 0 | 1.00000 | 0 | -1.0 - 1.5 BarG |
| VB39637242 | Vegabar83 -1..1 | « | | 0.00 | 1.50 | 0.0 | 18-APR-18 | 0 | 1.00000 | 0 | -1.0 - 1.5 BarG |
| VB39422625 | Vegabar82 -1..1 | « | | 0.85 | 0.00 | 0.0 | 4-MAY-18 | 0 | 1.00000 | 0 | |
| VB39422624 | Vegabar82 -1..1 | « | | 0.85 | 0.00 | 0.0 | 4-MAY-18 | 0 | 1.00000 | 0 | |
| VB38217276 | Vegabar82 -1..1 | | | 0.00 | 2.50 | 0.0 | 22-NOV-17 | 2 | 1.00184 | 0 | |
| Vb38217275 | Vegabar -1..1.5 | | | 0.00 | 2.50 | 0.0 | 22-NOV-17 | 0 | 1.00127 | 0 | |
| VB38214819 | Vegabar82 -1..2 | « | | 0.00 | 26.00 | 0.0 | 22-NOV-17 | 0 | 1.00000 | 0 | |
| VB38214818 | Vegabar82 -1..2 | « | | 0.00 | 26.00 | 0.0 | 22-NOV-17 | 0 | 1.00000 | 0 | |
| VB38214817 | Vegabar82 -1... | « | | 0.00 | 26.00 | 0.0 | 22-NOV-17 | 0 | 1.00000 | 0 | |
| VB38214816 | Vegabar82 -1..2 | « | | 0.00 | 26.00 | 0.0 | 22-NOV-17 | 0 | 1.00000 | 0 | -1.0 - 25.0 BarG |
| ST953822 | PTM/S-0.1% | « | | 0.80 | 1.20 | 5.0 | 18-JUL-16 | 0 | 1.00000 | 0 | |
| ST953566 | ATM.1ST/N/Ex-0. | | | 0.00 | 4.00 | 25.0 | 13-JUL-16 | 16 | 1.00063 | 0 | |
| ST953565 | ATM.1ST/N/Ex-0. | « | | 0.00 | 4.00 | 25.0 | 13-JUL-16 | 16 | 0.99955 | 0 | |
| ST953564 | ATM.1ST/N/Ex-0. | | | 0.00 | 4.00 | 25.0 | 13-JUL-16 | 17 | 1.00060 | 0 | |
| ST953563 | ATM.1ST/N/Ex-0. | | | 0.00 | 4.00 | 25.0 | 13-JUL-16 | 11 | 0.99936 | 0 | |
| ST953562 | ATM.1ST/N/Ex-0. | | | 0.00 | 4.00 | 25.0 | 13-JUL-16 | 4 | 1.00076 | 0 | |
| ST953557 | ATM.1ST/N | « | | 0.00 | 20.00 | 10.0 | 18-JUL-16 | 0 | 1.00000 | 0 | |
| ST953556 | ATM.1ST/N | « | | 0.00 | 20.00 | 10.0 | 18-JUL-16 | 0 | 1.00000 | 0 | |
| ST953555 | ATM.1ST/N | « | | 0.00 | 20.00 | 10.0 | 18-JUL-16 | 0 | 1.00000 | 0 | |
| ST953554 | ATM.1ST/N | « | | 0.00 | 20.00 | 10.0 | 18-JUL-16 | 0 | 1.00000 | 0 | |
| ST953553 | ATM.1ST/N | « | | 0.00 | 20.00 | 10.0 | 18-JUL-16 | 0 | 1.00000 | 0 | |
| ST953552 | ATM.1ST/N | « | | 0.00 | 4.00 | 10.0 | 18-JUL-16 | 0 | 1.00000 | 0 | |

Serial number: VB38214818
Type: Vegabar82 -1..2
Dallas Id: 0
Used: «
☐ Scrapped
Press "Find loc" to search..
Lowrange: 0.00
Highrange: 26.00
Date: 22-Nov-17
Kabellengde: 0.0
Null: 0
Gain: 1.00000
Notes:

*Figure 2-9: Shipdata: Sensordata and changing a record*

In general, the existing application is quite user-friendly as it is and include several great functions. Data is usually presented in lists which can give a good overview. However, the lists can get quite long and there are no filter functions to reduce the number of records shown.

Today, the application supports several good functions regarding sensors. When adding sensors of types STS, SPT or radar it is possible to add multiple records at a time. The application also supports importing sensor data from excel files received from subcontractors. The current application supports functions for programming Scanjet's equipment. These functions are crucial for their production.

Scanjet Ariston AS delivers embedded systems for tank control and monitoring, which require a configuration file describing which components are installed on the ship. The existing application include crucial function for generating these files which works great today.

## 2.2 Planned solution

This subchapter gives the reader an overview of the planned solution and the group's ambitions for the new software. As the existing solution is hard to make changes to, the group has decided to focus on extensibility and generalization of solutions in the new software. The overall philosophy is to make it future-oriented and extensible without compromising functionality.

### 2.2.1 Planned database

To improve something, it requires change. In contrast to the old database, the new solution will focus on extensibility. To accomplish this, the new database will generalize tables and account for completely new entry types in the future.

To make the new database general there is a drawback, datatype integrity. This means that some attributes in the new database will be variable, and not hold a specific datatype.

The requirements and design for the new database is explained in chapter 3.

### 2.2.2 Planned application

An application made over 20 years ago, with GUI elements and plenty of quirks gathered from its specific area of use, is bound to require a serious update now. The new application aims primarily to cover the functionality of the old, in a user-friendly and efficient way, all the while taking measures to be flexible enough to handle significant database changes.

As the new database will sacrifice data type integrity for the sake of generalization, the new application will have to deal with it to avoid deviation from intended data types. From many different sources, problems like this might pop up if the intended result is a flexible solution based on variables where there once were constants. To tackle this issue, another application layer is planned to serve as a joint between database, GUI and general configuration variables.

The key element to make this software future oriented and extensible is a resource document which contains, among other things, information about the structural design of the database. *Figure 2-10* illustrates an overview of the planned solution with a resource file.

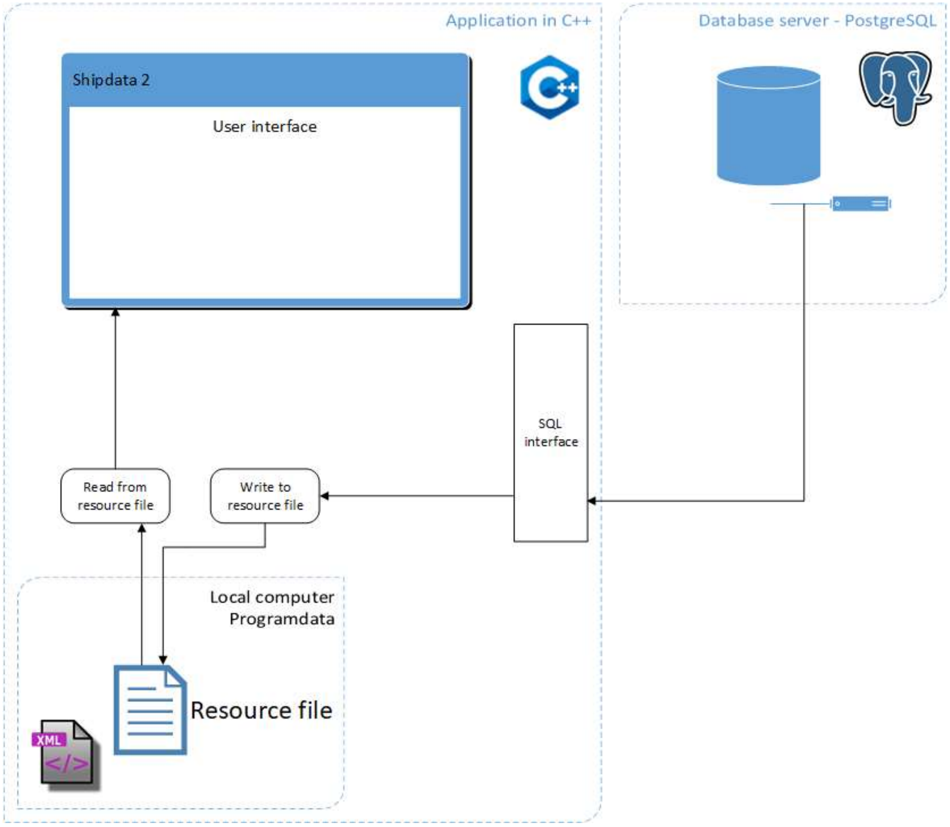


*Figure 2-10: Planned system with resource file.*

# 3 Database – Requirements and design

This chapter describes the requirements and needs discussed with the customer regarding the database and user authentication, as well as suggested design.

## 3.1 Requirements

This subchapter gives the reader an overview of the customer needs and requirements regarding the database and user access. The requirements described are needed to develop a functional, secure and user-friendly database. Details regarding requirements can be seen in Appendix B.

### 3.1.1 Database requirements

The database should be developed based on an entity relationship model, made as a PostgreSQL database and fulfill normalization standards 3NF, preferably BCNF. The new model should be extensible, future-oriented and shaped such that all existing data can be transferred. Independent tables that contain significant data should also include simple data logging to ensure record traceability. [3]

### 3.1.2 User access requirements

Users of the database should be able to retrieve, add or modify information limited by a user access groups controlled by the database server. Unauthorized attempts to retrieve data should fail. *Table 3-1* describes the required user access groups that should be implemented. *Table 3-2* describes which privileges each user group should have.

*Table 3-1: Required user access groups*

| User access group | Description |
|---|---|
| Sysop | System operators. |
| Administrator | Administrators for production in Norway. |
| User | Normal users for production in Norway. |
| KRadmin | Administrators for Korean production facility. |
| KRuser | Normal users for Korean production facility. |

*Table 3-2: User group privileges*

| User | KRuser * | KRadmin * | User | Administrator | Sysop |
|---|---|---|---|---|---|
| View records | √ | √ | √ | √ | √ |
| Edit records | √ | √ | √ | √ | √ |
| Add new records | √ | √ | √ | √ | √ |
| Delete records | X | √ | X | √ | √ |
| | | | | | |
| Config file generation | √ | √ | √ | √ | √ |
| User access control | X | X | X | X | √ |
| Superuser | X | X | X | X | √ |

* Limited to tables component, component_detail, component_error_data, tank, article and project. Read only privileges in tables article and project.

## 3.2 Entity-relationship model

This subchapter describes the design of the database in this project represented as an entity-relationship model. An ER-model is a systematic analysis to define and describe each table and their interaction with each other in a database. In this subchapter the ER-model is split into several pieces and described separately. To view the complete model, see Appendix C and the ER-model guide in Appendix D.

### 3.2.1 Relationship between main tables

The main tables in the database consists of the six tables: *purchase_order*, *project*, *article*, *component*, *tank* and *company*. *Figure 3-1* illustrates the main tables and their relationships. The table project contains information about all Scanjet Ariston's projects, where each project has a unique attribute, the primary key *project_id*. In practice, a project number is used to identify each record, but can have duplicates. To ensure a unique primary key, a project id is designed for the table, and the project number as an attribute. Each project can have one or many purchase orders connected, where a *one-to-many* relationship is designed between the tables.

As part of Scanjet's product, instrumentation for tanks are supplied. Each project has a relationship to one or more tanks. The table *tank* in *Figure 3-1* contains information about each tank in the project. *Tank_id* is the primary key of the table and *project_id* is a foreign key to connect each tank to a project.

The table *purchase_order* shown in *Figure 3-1* contains a primary key *po_number*, and the foreign key *project_id* where a project can be connected to the order. Each purchase order has one customer and one production company. To accomplish this connection, two *one-to-many* relationships from the table *company* to *purchase_order* are designed.

Every PO made by a customer can be for several components for their projects. The table *component* in *Figure 3-1* contains information about every component and can be connected by the foreign key *po_number* to POs. Each PO can be connected to one or more components. To be able to know which components are connected to which tanks, a *one-to-many* relationship is designed from tank to components. Each component can then be connected to a tank through the foreign key *tank_id*.

The last main table is *article* in *Figure 3-1* which contains general information about a group of products. Each record in the table *component* can be connected to an *article_id,* where an article can be connected to one or many components. Each record in *article* has a vendor which has produced or sold the items. To be able to trace this, a relationship between *company* and *article* is designed. A company can be connected to one or more articles by the foreign key *company_id* in the table *article*. Over time articles can be obsolete and be replaced by another article. This requires a unary relationship from *article* to itself and a foreign key *replaced_by*.

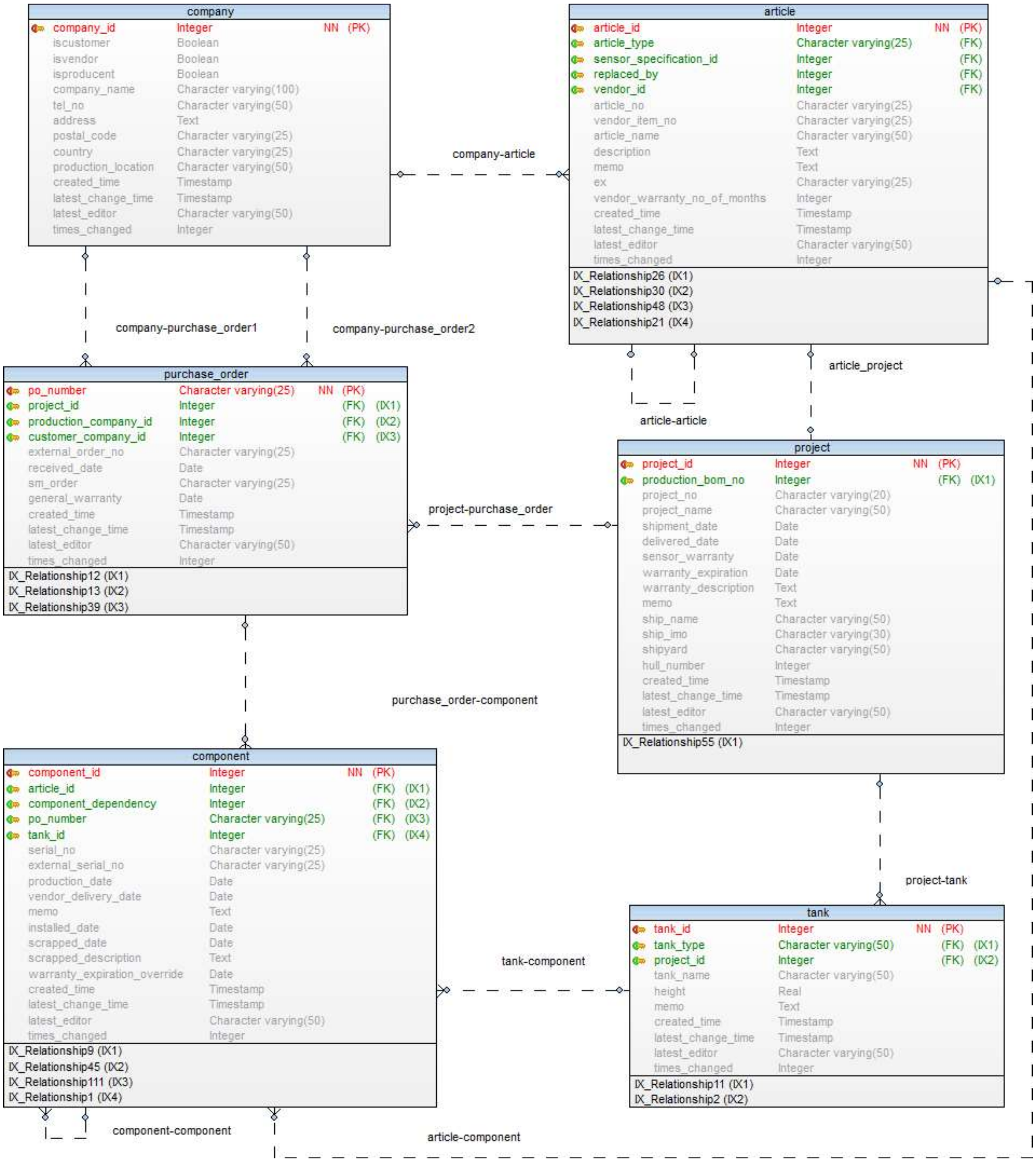

company
company_id Integer NN (PK)
iscustomer Boolean
isvendor Boolean
isproducent Boolean
company_name Character varying(100)
tel_no Character varying(50)
address Text
postal_code Character varying(25)
country Character varying(25)
production_location Character varying(50)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
article
article_id Integer NN (PK)
article_type Character varying(25) (FK)
sensor_specification_id Integer (FK)
replaced_by Integer (FK)
vendor_id Integer (FK)
article_no Character varying(25)
vendor_item_no Character varying(25)
article_name Character varying(50)
description Text
memo Text
ex Character varying(25)
vendor_warranty_no_of_months Integer
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
company-article
company-purchase_order1
company-purchase_order2
article_project
article-article
purchase_order
po_number Character varying(25) NN (PK)
project_id Integer (FK) (IX1)
production_company_id Integer (FK) (IX2)
customer_company_id Integer (FK) (IX3)
external_order_no Character varying(25)
received_date Date
sm_order Character varying(25)
general_warranty Date
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship12 (IX1)
IX_Relationship13 (IX2)
IX_Relationship39 (IX3)
project-purchase_order
project
project_id Integer NN (PK)
production_bom_no Integer (FK) (IX1)
project_no Character varying(20)
project_name Character varying(50)
shipment_date Date
delivered_date Date
sensor_warranty Date
warranty_expiration Date
warranty_description Text
memo Text
ship_name Character varying(50)
ship_imo Character varying(30)
shipyard Character varying(50)
hull_number Integer
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship55 (IX1)
purchase_order-component
component
component_id Integer NN (PK)
article_id Integer (FK) (IX1)
component_dependency Integer (FK) (IX2)
po_number Character varying(25) (FK) (IX3)
tank_id Integer (FK) (IX4)
serial_no Character varying(25)
external_serial_no Character varying(25)
production_date Date
vendor_delivery_date Date
memo Text
installed_date Date
scrapped_date Date
scrapped_description Text
warranty_expiration_override Date
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)
project-tank
tank-component
tank
tank_id Integer NN (PK)
tank_type Character varying(50) (FK) (IX1)
project_id Integer (FK) (IX2)
tank_name Character varying(50)
height Real
memo Text
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship11 (IX1)
IX_Relationship2 (IX2)
component-component
article-component

*Figure 3-1: The relationship between main tables.*

### 3.2.2 Project and tank tables

As described in chapter 3.2.1 each project is connected to one or more tanks. *Figure 3-2* describes the tables *project*, *tank*, *tank_type* and their relationships. Each tank has a type description which in this model is a separate table. Each *tank_type* can be connected to one or more tanks. This design is done to prevent users to accidently type in the wrong tank type for a tank. *Tank_type* is a foreign key in the table *tank* and requires a correct input corresponding with a record in the table *tank_type*.

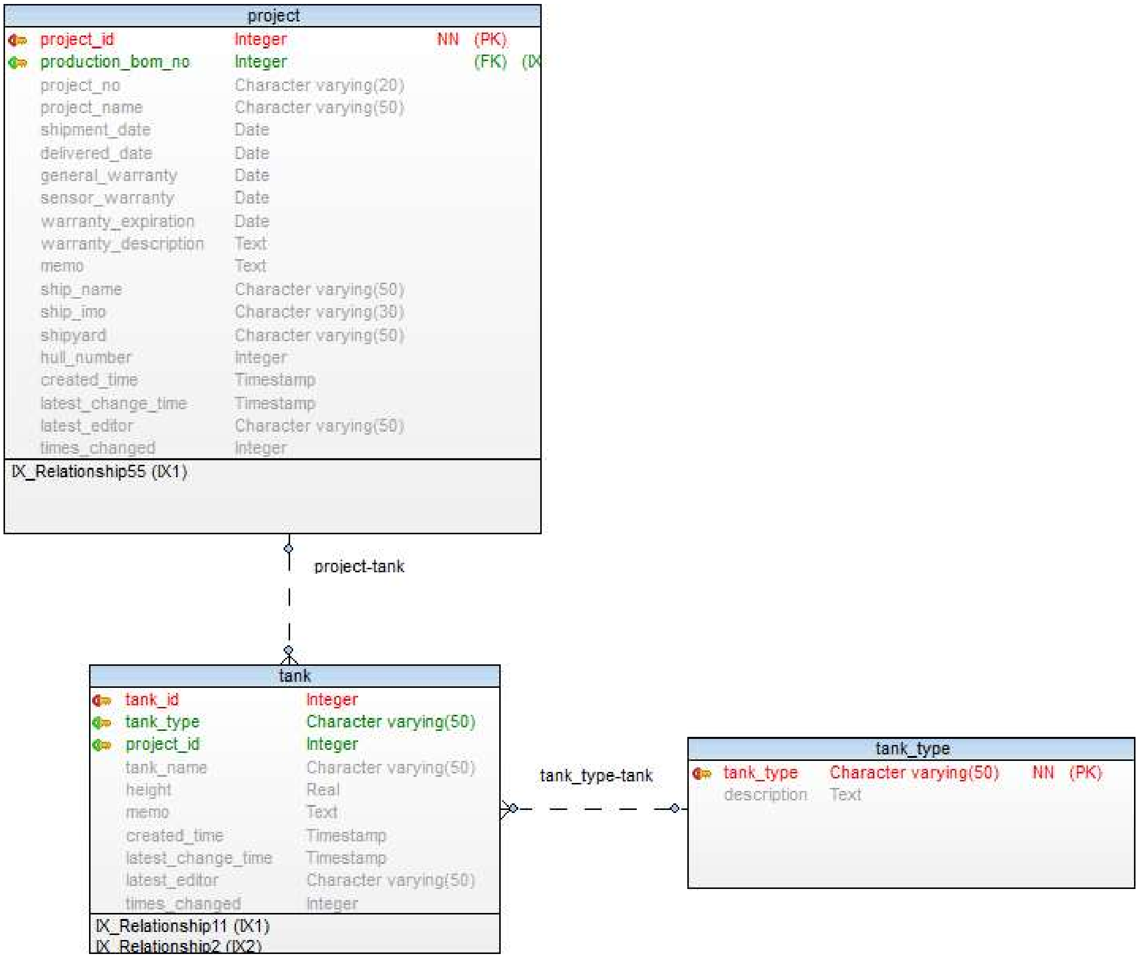
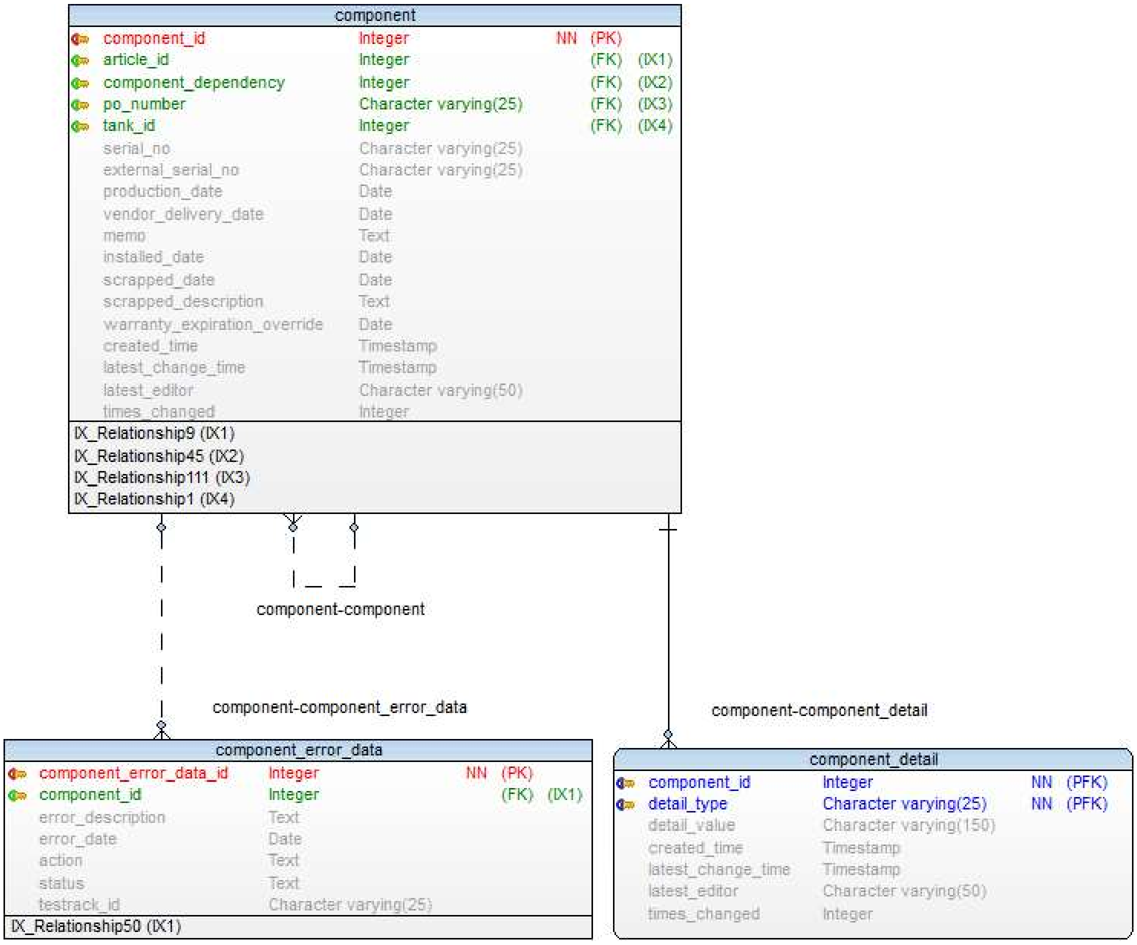

project
project_id Integer NN (PK)
production_bom_no Integer (FK) (IX
project_no Character varying(20)
project_name Character varying(50)
shipment_date Date
delivered_date Date
general_warranty Date
sensor_warranty Date
warranty_expiration Date
warranty_description Text
memo Text
ship_name Character varying(50)
ship_imo Character varying(30)
shipyard Character varying(50)
hull_number Integer
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship55 (IX1)
project-tank
tank
tank_id Integer
tank_type Character varying(50)
project_id Integer
tank_name Character varying(50)
height Real
memo Text
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship11 (IX1)
IX_Relationship2 (IX2)
tank_type-tank
tank_type
tank_type Character varying(50) NN (PK)
description Text

*Figure 3-2: Relationship between tank and project*

### 3.2.3 Component and tank tables

*Figure 3-3* illustrates the following tables: *component*, *component_error_data*, *component_detail*, *tank* and their relationships. The table *component* contains information for each unique component. To ensure a general and extensible design, a table *component_detail* is designed. The table can hold records of additional properties for records in *component.* This design also enables new types of components with another set of properties to be created in the future, without having to alter the database.

Error data for components can be stored in the table *component_error_data.* Each record in *component* can have one or many records in *component_error_data.* The last relationship from the table *component* is a *one-to-many* relation with itself. Products like *radar* consists of several components. To ensure its possible to trace which are connected, the relationship is designed.



| component | | | | |
|---|---|---|---|---|
| component_id | Integer | NN | (PK) | |
| article_id | Integer | | (FK) | (IX1) |
| component_dependency | Integer | | (FK) | (IX2) |
| po_number | Character varying(25) | | (FK) | (IX3) |
| tank_id | Integer | | (FK) | (IX4) |
| serial_no | Character varying(25) | | | |
| external_serial_no | Character varying(25) | | | |
| production_date | Date | | | |
| vendor_delivery_date | Date | | | |
| memo | Text | | | |
| installed_date | Date | | | |
| scrapped_date | Date | | | |
| scrapped_description | Text | | | |
| warranty_expiration_override | Date | | | |
| created_time | Timestamp | | | |
| latest_change_time | Timestamp | | | |
| latest_editor | Character varying(50) | | | |
| times_changed | Integer | | | |
| IX_Relationship9 (IX1) | | | | |
| IX_Relationship45 (IX2) | | | | |
| IX_Relationship111 (IX3) | | | | |
| IX_Relationship1 (IX4) | | | | |

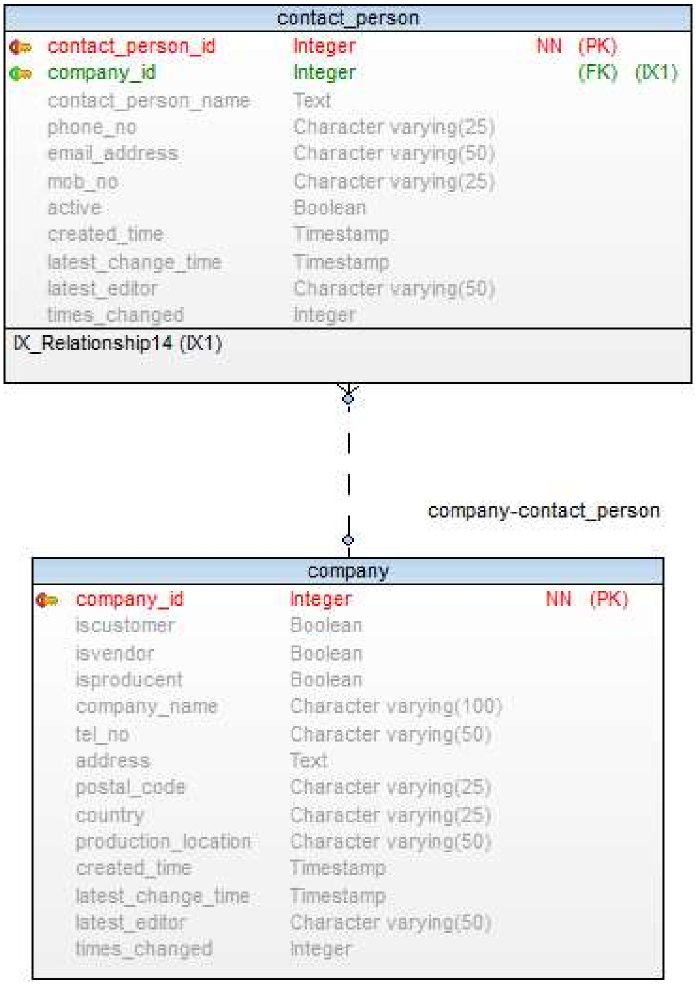
| component_error_data | | | | |
|---|---|---|---|---|
| component_error_data_id | Integer | NN | (PK) | |
| component_id | Integer | | (FK) | (IX1) |
| error_description | Text | | | |
| error_date | Date | | | |
| action | Text | | | |
| status | Text | | | |
| testrack_id | Character varying(25) | | | |
| IX_Relationship50 (IX1) | | | | |

| component_detail | | | |
|---|---|---|---|
| component_id | Integer | NN | (PFK) |
| detail_type | Character varying(25) | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

*Figure 3-3: Relationship between component and tank*

### 3.2.4 Company and contact person tables

*Figure 3-4* illustrates the tables *company*, *contact_person* and their relationship. The table *company* contains information about customer, production and vendor companies. To separate what each company is, boolean attributes are used to identify whether it is a customer, a production or a vendor company. Each record of a company can be connected to one or more records in *contact_person* with a relationship between the two tables.



| contact_person | | | | |
|---|---|---|---|---|
| contact_person_id | Integer | NN | (PK) | |
| company_id | Integer | | (FK) | (IX1) |
| contact_person_name | Text | | | |
| phone_no | Character varying(25) | | | |
| email_address | Character varying(50) | | | |
| mob_no | Character varying(25) | | | |
| active | Boolean | | | |
| created_time | Timestamp | | | |
| latest_change_time | Timestamp | | | |
| latest_editor | Character varying(50) | | | |
| times_changed | Integer | | | |
| IX_Relationship14 (IX1) | | | | |

| company | | | |
|---|---|---|---|
| company_id | Integer | NN | (PK) |
| iscustomer | Boolean | | |
| isvendor | Boolean | | |
| isproducent | Boolean | | |
| company_name | Character varying(100) | | |
| tel_no | Character varying(50) | | |
| address | Text | | |
| postal_code | Character varying(25) | | |
| country | Character varying(25) | | |
| production_location | Character varying(50) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

*Figure 3-4: Relationship between purchase order, project and company*

### 3.2.5 Article, company and attachment tables

Each component in the database is connected to an article which describes the general information for a group of components. In practice, an article number, which is an attribute in *article*, is used instead of the primary key *article_id*. The design is chosen because duplicates of article numbers may occur.

Articles can have e.g. drawings or certificates connected to an article number. *Figure 3-5* shows the tables regarding attachments and articles and their relationships. Information about each attachment is stored as records in the table *attachment*, which has a *many-to-many* relationship with articles. Each article can have many attachments, but also an attachment can be connected to many articles. If additional attributes are needed for a record, the table *attachment_detail* is designed with a *one-to-many* relationship from attachment. A record in *attachment* also contains a predefined type description as a foreign key from the table *attachment_type*. This solution is designed to maintain consistent type descriptions and prevent typos.

For each attachment it’s vital to know which revision it has. A table *attachment_revision* in *Figure 3-5* is designed to contain records with revision details. A record in *attachment* can have one or more records in *attachment_revision*. To ensure it is possible to add new attributes in the future, the table *attachment_revision_detail* is designed to hold records of additional attributes for revisions.

In *Figure 3-5* a *one-to-many* relationship from *company* to *article* is designed to be able to connect vendor companies to articles which they have produced. Each record in *article* can only be connected to one record in *company*.

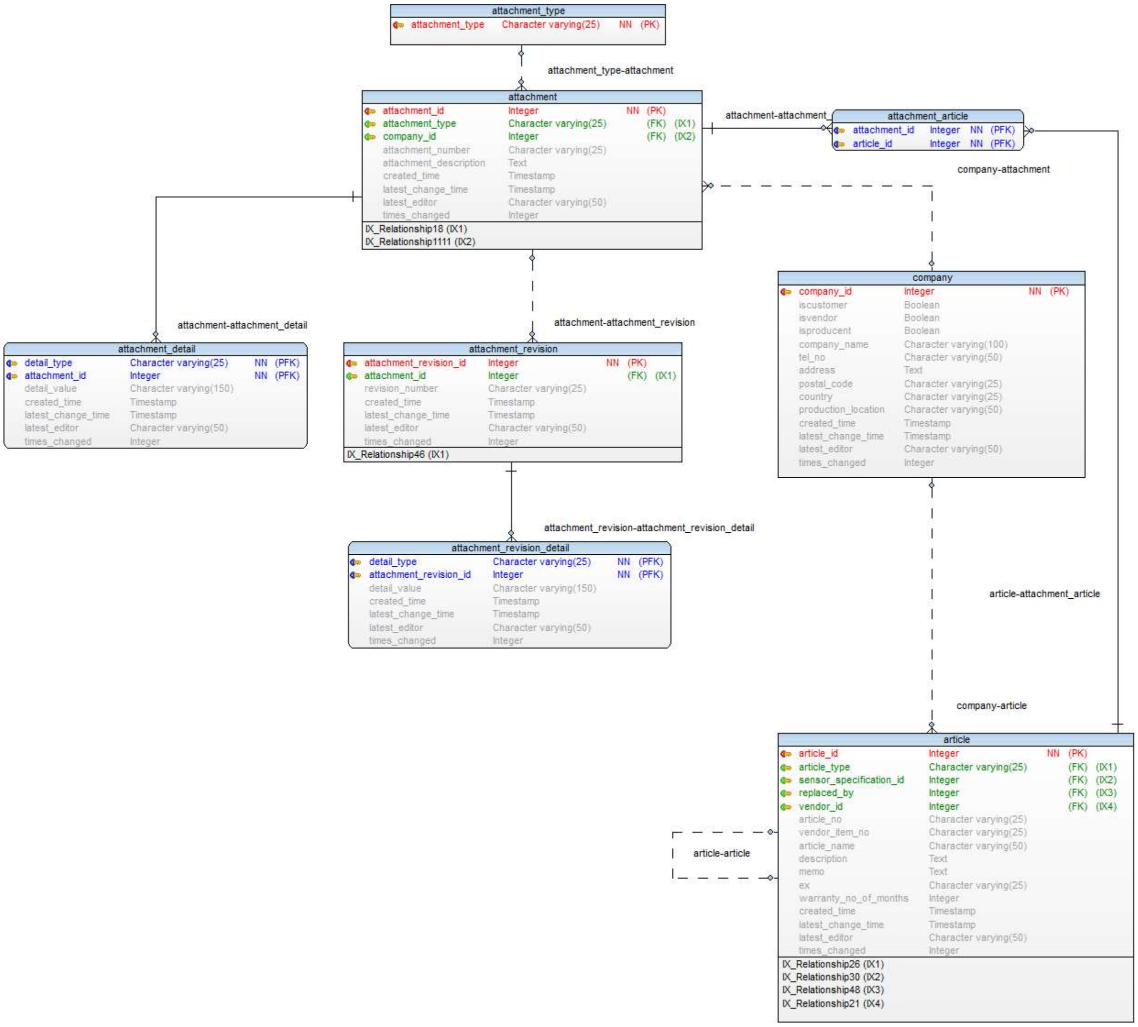

attachment_type
attachment_type Character varying(25) NN (PK)
attachment_type-attachment
attachment
attachment_id Integer NN (PK)
attachment_type Character varying(25) (FK) (IX1)
company_id Integer (FK) (IX2)
attachment_number Character varying(25)
attachment_description Text
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship18 (IX1)
IX_Relationship1111 (IX2)
attachment-attachment
attachment_article
attachment_id Integer NN (PFK)
article_id Integer NN (PFK)
company-attachment
attachment-attachment_detail
attachment-attachment_revision
attachment_detail
detail_type Character varying(25) NN (PFK)
attachment_id Integer NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
attachment_revision
attachment_revision_id Integer NN (PK)
attachment_id Integer (FK) (IX1)
revision_number Character varying(25)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship46 (IX1)
company
company_id Integer NN (PK)
iscustomer Boolean
isvendor Boolean
isproducent Boolean
company_name Character varying(100)
tel_no Character varying(50)
address Text
postal_code Character varying(25)
country Character varying(25)
production_location Character varying(50)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
attachment_revision-attachment_revision_detail
attachment_revision_detail
detail_type Character varying(25) NN (PFK)
attachment_revision_id Integer NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
article-attachment_article
company-article
article
article_id Integer NN (PK)
article_type Character varying(25) (FK) (IX1)
sensor_specification_id Integer (FK) (IX2)
replaced_by Integer (FK) (IX3)
vendor_id Integer (FK) (IX4)
article_no Character varying(25)
vendor_item_no Character varying(25)
article_name Character varying(50)
description Text
memo Text
ex Character varying(25)
warranty_no_of_months Integer
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
article-article

*Figure 3-5: Relationships between attachments, article and company*

### 3.2.6 Article and sensor specification tables

A large part of Scanjet's components is sensors. While other types of products split their data between *component* and *article*, sensors are special in that they also have data on another, more general level. This level is named *sensor_specification*. Data such as the sensor's accuracy is stored here. One *sensor_specification* record may be associated with many different article numbers. In the real world, this distinction only occurs to account for different wire lengths.

Merging *sensor_specification* with *article* data would have resulted in a more intuitive database design, but also a lot of duplicate data. In addition, this violates the understanding Scanjet employees have built on how sensors are defined. The ER model cut-out shown in *Figure 3-6* shows the final design with *article* and *sensor_specification* as separate tables.

*article_type* and *sensor_category* work in similar ways. They define all possible occurrences of different types. E.g. "amplifier" or "sensor" in *article_type*, and "pressure" or "temperature" in *sensor_category*. This distinction is critical for when using detail tables as explained below, as well as in chapter 4.2.3.

Would-be columns under *article* that are not common for all article types, are instead stored in *article_detail*. Sensors, for example, have the *wire_length* property, but amplifiers don't. Similarly, this also applies to *sensor_specification*, where pressure and temperature sensors naturally have different sets of properties.

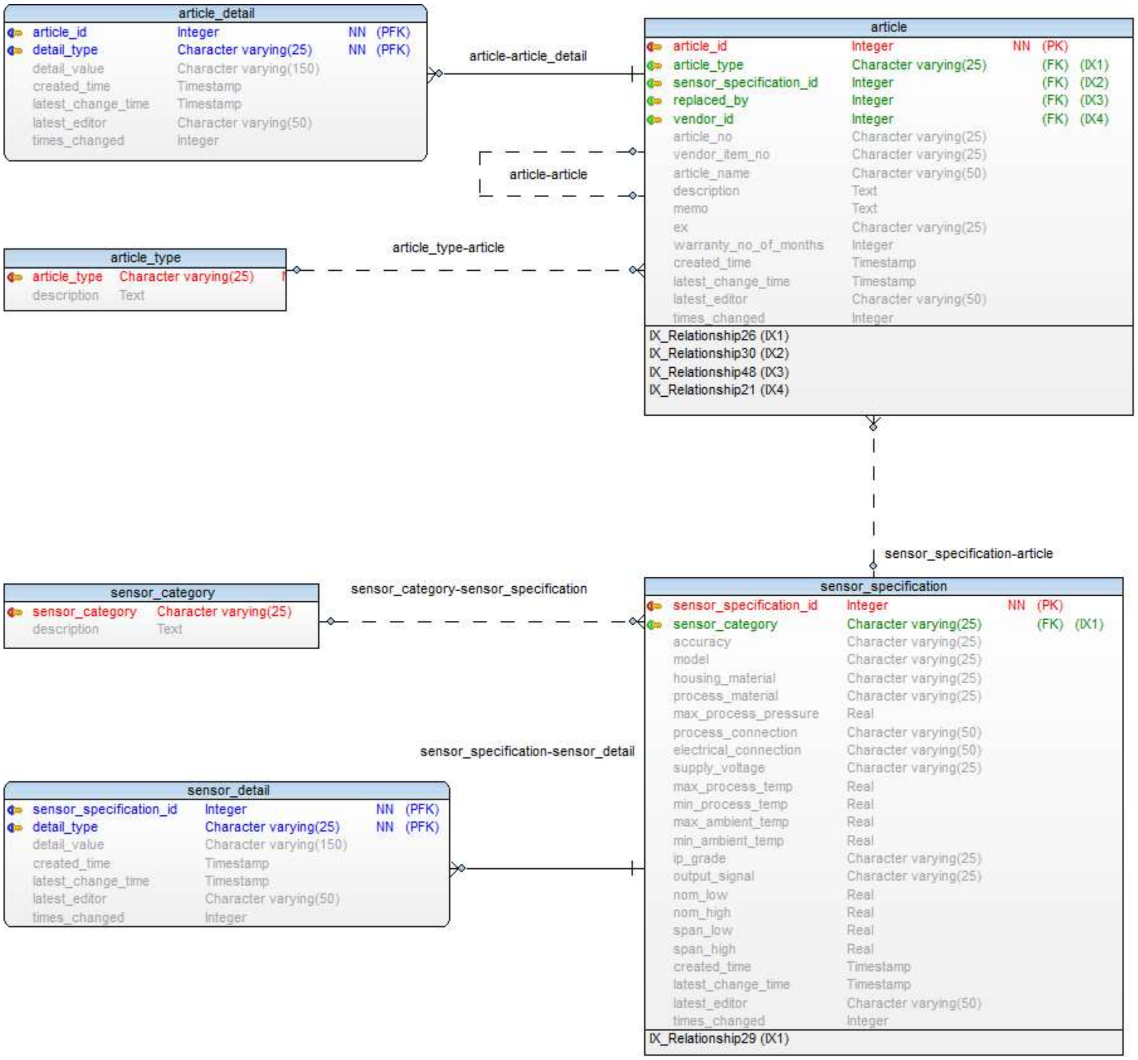
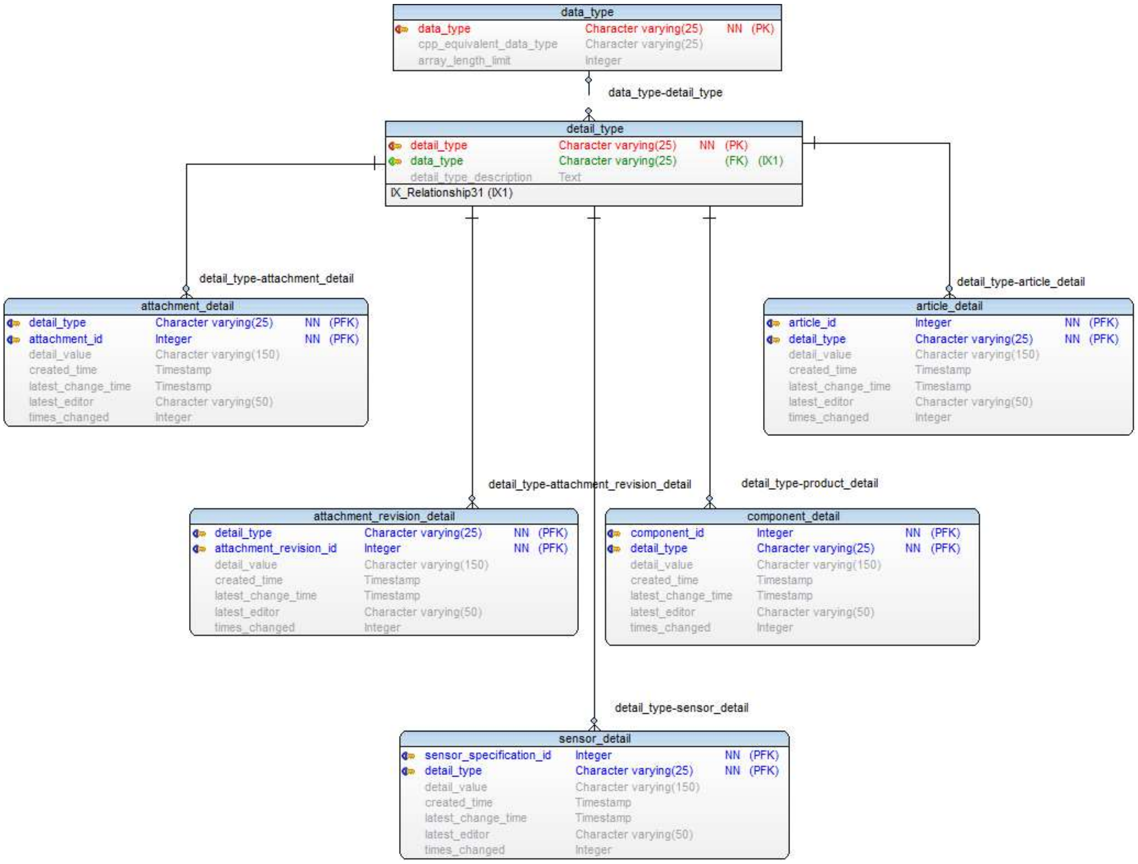

article_detail
article_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
article-article_detail
article-article
article_type
article_type Character varying(25)
description Text
article_type-article
article
article_id Integer NN (PK)
article_type Character varying(25) (FK) (IX1)
sensor_specification_id Integer (FK) (IX2)
replaced_by Integer (FK) (IX3)
vendor_id Integer (FK) (IX4)
article_no Character varying(25)
vendor_item_no Character varying(25)
article_name Character varying(50)
description Text
memo Text
ex Character varying(25)
warranty_no_of_months Integer
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
sensor_specification-article
sensor_category
sensor_category Character varying(25)
description Text
sensor_category-sensor_specification
sensor_specification-sensor_detail
sensor_detail
sensor_specification_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
sensor_specification
sensor_specification_id Integer NN (PK)
sensor_category Character varying(25) (FK) (IX1)
accuracy Character varying(25)
model Character varying(25)
housing_material Character varying(25)
process_material Character varying(25)
max_process_pressure Real
process_connection Character varying(50)
electrical_connection Character varying(50)
supply_voltage Character varying(25)
max_process_temp Real
min_process_temp Real
max_ambient_temp Real
min_ambient_temp Real
ip_grade Character varying(25)
output_signal Character varying(25)
nom_low Real
nom_high Real
span_low Real
span_high Real
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship29 (IX1)

*Figure 3-6: Relationships between article and sensor specification*

### 3.2.7 Detail and detail type tables

To develop an extensible database, the solutions for table details is designed. This solution is more general and does not require changes to the database if a new attribute is needed in the future. *Figure 3-7* illustrates the relationships between each detail table and *detail_type*. The table *detail_type* contains information about detail types, which is connected to every detail table with a *one-to-many* relationship. *Data_type* is a table which contains information about the datatype of a specific detail type. Each record in *data_type* can be connected to one or more *detail_type* records.


data_type-detail_type
detail_type-attachment_detail
detail_type-article_detail
detail_type-attachment_revision_detail
detail_type-product_detail
detail_type-sensor_detail
IX_Relationship31 (IX1)

| data_type | | | |
|---|---|---|---|
| data_type | Character varying(25) | NN | (PK) |
| cpp_equivalent_data_type | Character varying(25) | | |
| array_length_limit | Integer | | |

| detail_type | | | |
|---|---|---|---|
| detail_type | Character varying(25) | NN | (PK) |
| data_type | Character varying(25) | | (FK) (IX1) |
| detail_type_description | Text | | |

| attachment_detail | | | |
|---|---|---|---|
| detail_type | Character varying(25) | NN | (PFK) |
| attachment_id | Integer | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

| article_detail | | | |
|---|---|---|---|
| article_id | Integer | NN | (PFK) |
| detail_type | Character varying(25) | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

| attachment_revision_detail | | | |
|---|---|---|---|
| detail_type | Character varying(25) | NN | (PFK) |
| attachment_revision_id | Integer | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

| component_detail | | | |
|---|---|---|---|
| component_id | Integer | NN | (PFK) |
| detail_type | Character varying(25) | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

| sensor_detail | | | |
|---|---|---|---|
| sensor_specification_id | Integer | NN | (PFK) |
| detail_type | Character varying(25) | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

*Figure 3-7: Relationships between details and detail type.*

# 4 Migration of data – Requirements and design

When a new database is created in the attempt to improve an existing solution it is often required to preserve the old data. For the new database to be an improvement, it must also be different from the current design. This creates the issue of how to preserve the current data. In this chapter, problems and solutions to various migration topics will be discussed regarding the migration between the existing Clarion database and the new PostgreSQL database.

It is recommended to read chapter *2.1.1 Existing database* and *3.2 Entity-relationship model* to be acquainted with the two databases.

## 4.1 Requirements and restrictions

The original task description of data migration does not specify which data to be transferred. Requirements and restrictions for migration of data are made in cooperation with Scanjet. The migration will take place after this project by the company itself. The migration must account for future data. Ideally, executing the migration should not require programming knowledge.

Generally, all component and project related data should be transferred without losing its relations. No contact information or data about customers should be transferred by the group. Selected data will be procured by Scanjet and should be delivered to the group. *Figure 4-1* shows which tables should be transferred to PostgreSQL, and data which should not be migrated.  All *details* should go to *component_details*, although ideally, they should be distributed amongst several detail tables. The exception is *attachment_details.* Further requirements and restrictions can be read in Appendix B.

It is important to note that not all fields in the new database will be filled out during the migration. Old data will not be as rich in information as new entries.

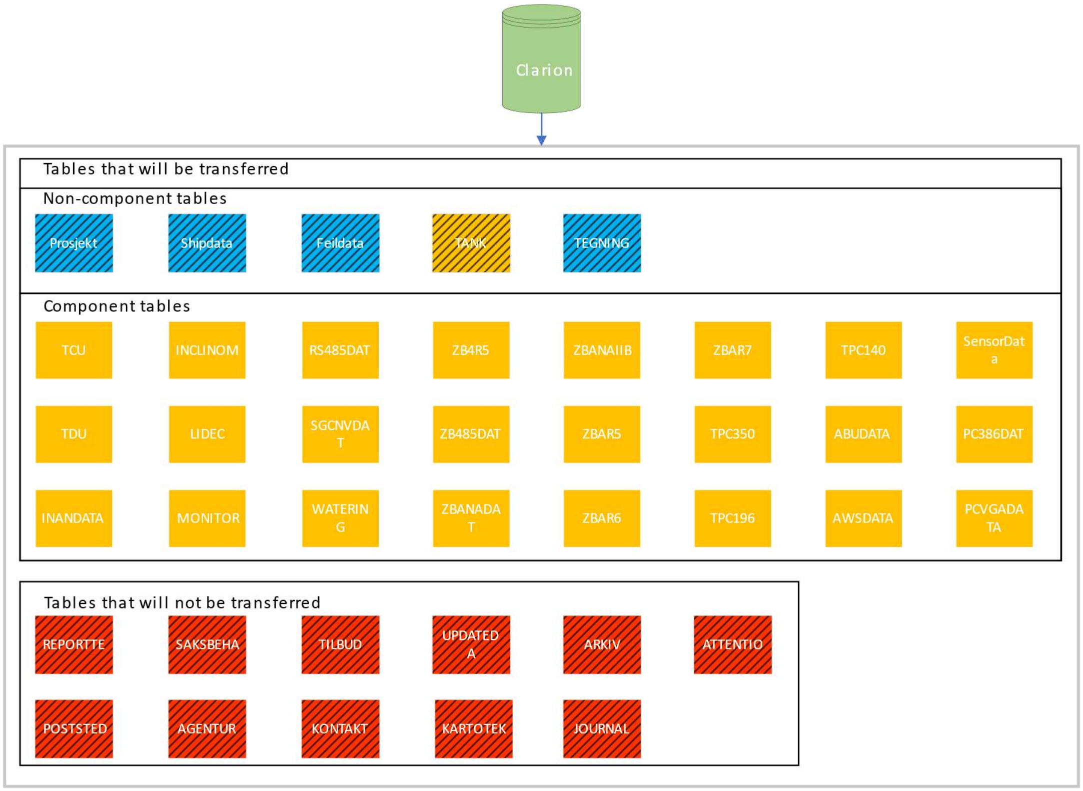


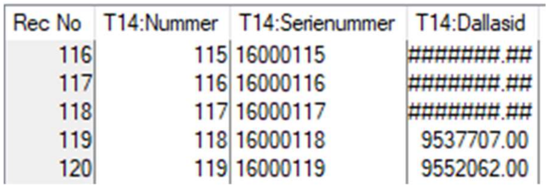
*Figure 4-1 Clarion tables which should be transferred*

## 4.2 Migration issues

The main functional difference between the Clarion and PostgreSQL database is the structure of the tables. This subchapter will try to explain the differences, thus giving an impression of what the challenges are. Central questions are: where should data be put in the new database? How are the data relations preserved? Are there data that don't fit the new model?

### 4.2.1 Clarion, and differences from Excel

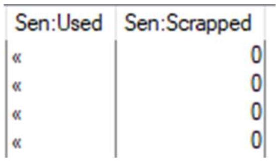
In Clarion, the column *Dallasid,* and some others which is found in several tables, is unreadable in its current format as can be seen in *Figure 4-2*. Details of which tables and columns are concerned can be seen in Appendix G.



| Rec No | T14:Nummer | T14:Serienummer | T14:Dallasid |
|---|---|---|---|
| 116 | 115 | 16000115 | ########.## |
| 117 | 116 | 16000116 | ########.## |
| 118 | 117 | 16000117 | ########.## |
| 119 | 118 | 16000118 | 9537707.00 |
| 120 | 119 | 16000119 | 9552062.00 |

*Figure 4-2 In Clarion, Dallasid is unreadable in several occasions*

Several columns which are used as the *Boolean* data type are of *String* data type in Clarion, these are expected to become true *Boolean* values. *Figure 4-3* shows an example of this, there are several occasions of similar use of Boolean values.



| Sen:Used | Sen:Scrapped |
|---|---|
| « | 0 |
| « | 0 |
| « | 0 |
| « | 0 |

*Figure 4-3 Sensordata's Used and Scrapped columns. Both are used as boolean values, but uses varying ways to indicate true and false*

Additionally, the Clarion date data type counts days from 28-12-1800 as zero, while Excel counts days from 00-01-1900 also as zero. Which means the date will be wrong by approximately 100 years if not dealt with.

## 4.2.2 Project related data

In PostgreSQL project data is more centralized, where one table stores the data of the previous *prosjekt* (project) and *Shipdata* tables. There has however been a change in the structure between components and *project*. For increased traceability, a component is now connected to a PO. To keep the relations from the existing database, dummy POs must be created for every existing project. Component tables are, as in chapter 2.1.1, considered tables not hatched in figures depicting the old database. These tables have been created for each article type, except for *Lidec* and *Sensordata* which contain many article types. The changes between the databases and generally how the data must be directed regarding project data can be seen in *Figure 4-4*.

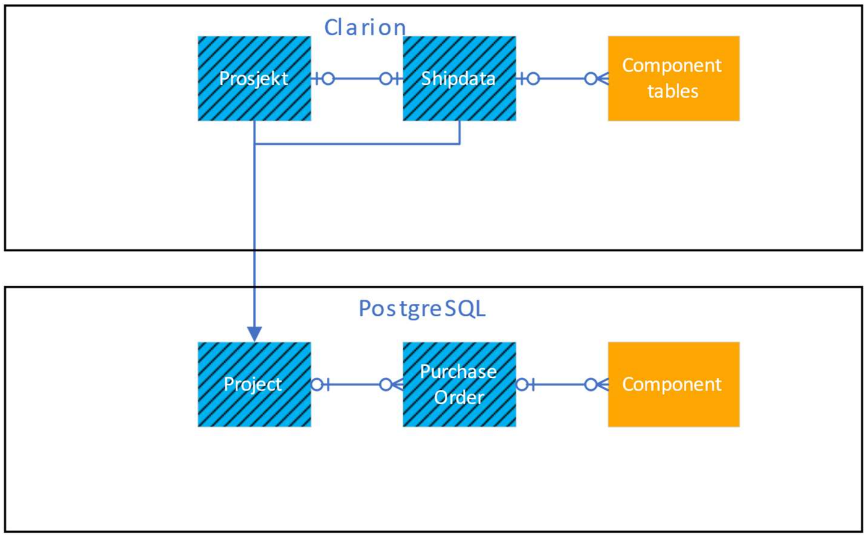


*Figure 4-4 Indicates the structural changes of how project data is stored*

### 4.2.3 Component data

In the PostgreSQL database there are general tables for components, articles and sensors which are used to describe an item in layers. As mentioned in chapter 4.1, data will be transferred to *component* and *component_details.* The old tables must be split up into two tables for either *component* and *component_details.* The change in details require significant attention as it restructures the data. The general distribution of components can be seen in *Figure 4-5*.

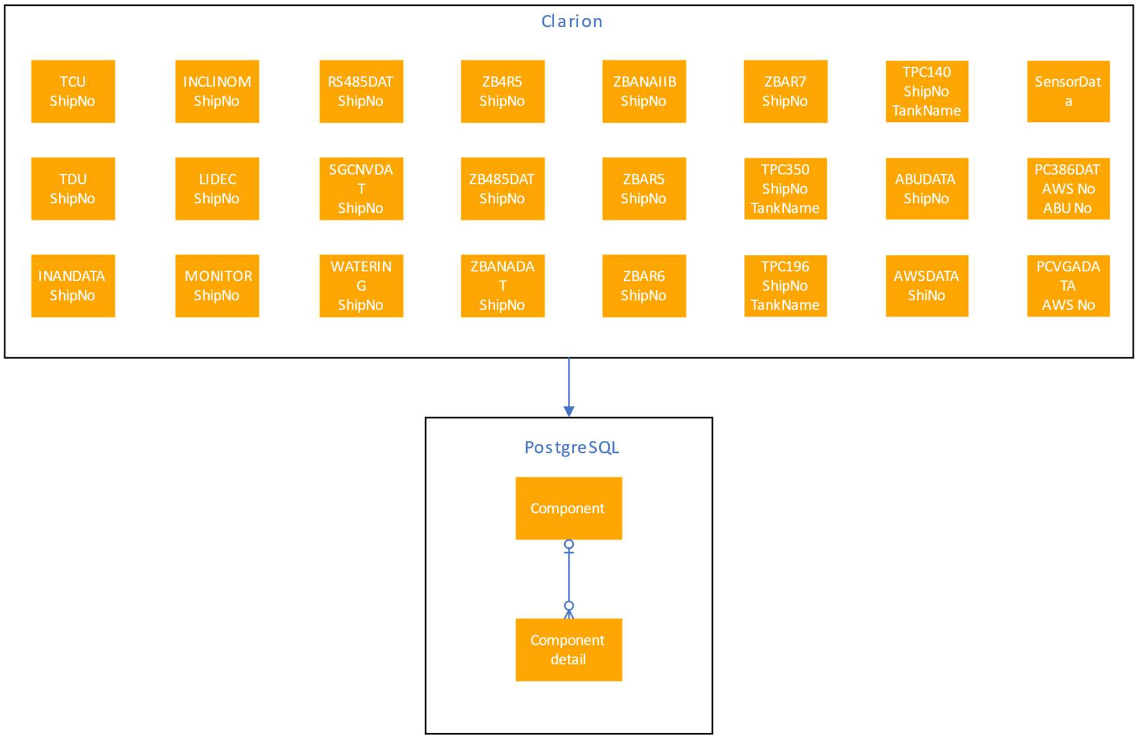
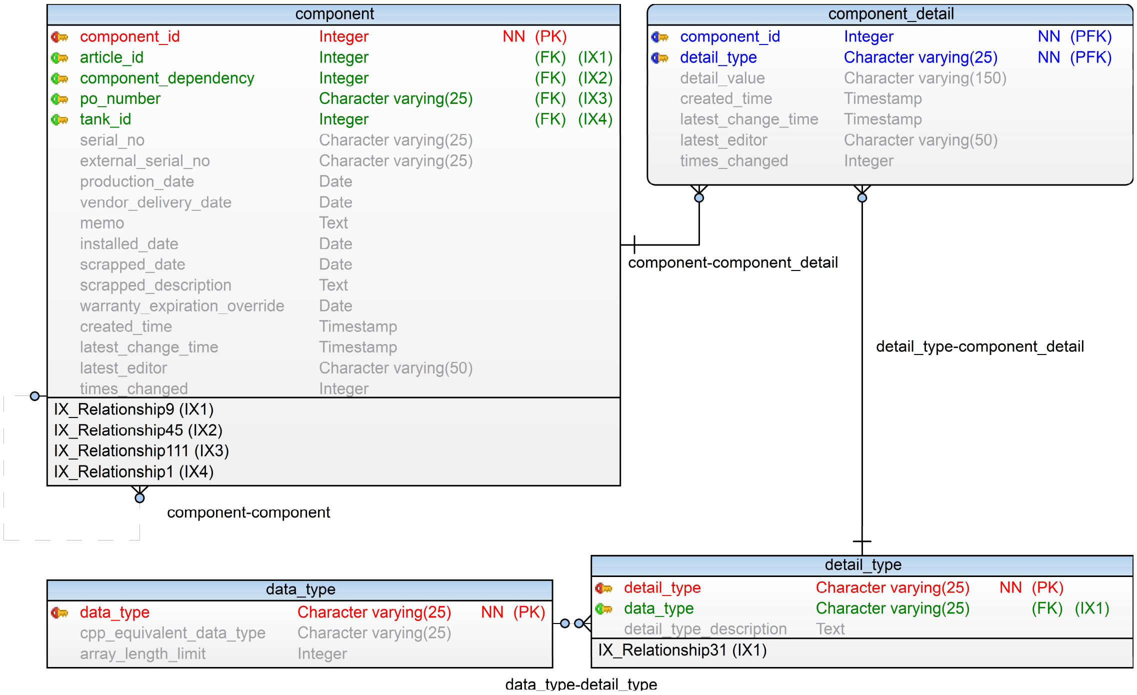


*Figure 4-5 General relocation of data between Clarion and PostgreSQL*

In Clarion, all tables have specific columns for varying components. Now that a general solution is chosen, it's necessary to show how non-general data is stored in the PostgreSQL database which can be seen in *Figure 4-6*. All general information will be stored in component, and all other data will be stored in a detail table. The same will apply to attachment during migration. After the new system is deployed, this principle will apply to *article* and *sensor_specificaiton.*



component

| | | | |
|---|---|---|---|
| component_id | Integer | NN | (PK) |
| article_id | Integer | | (FK) (IX1) |
| component_dependency | Integer | | (FK) (IX2) |
| po_number | Character varying(25) | | (FK) (IX3) |
| tank_id | Integer | | (FK) (IX4) |
| serial_no | Character varying(25) | | |
| external_serial_no | Character varying(25) | | |
| production_date | Date | | |
| vendor_delivery_date | Date | | |
| memo | Text | | |
| installed_date | Date | | |
| scrapped_date | Date | | |
| scrapped_description | Text | | |
| warranty_expiration_override | Date | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)

component_detail

| | | | |
|---|---|---|---|
| component_id | Integer | NN | (PFK) |
| detail_type | Character varying(25) | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

data_type

| | | | |
|---|---|---|---|
| data_type | Character varying(25) | NN | (PK) |
| cpp_equivalent_data_type | Character varying(25) | | |
| array_length_limit | Integer | | |

detail_type

| | | | |
|---|---|---|---|
| detail_type | Character varying(25) | NN | (PK) |
| data_type | Character varying(25) | | (FK) (IX1) |
| detail_type_description | Text | | |

IX_Relationship31 (IX1)

*Figure 4-6 Component table with belonging detail table*

### 4.2.4 Tank data

The other big deriving issue involves the *tank* table of the Clarion database. This table mostly has the same data as before, but sensor installation is no longer in the tank table, but on the sensor. *Figure 4-7* shows an overview of where data from *Sensordata* and *Tank* tables are placed in the PostgreSQL database.

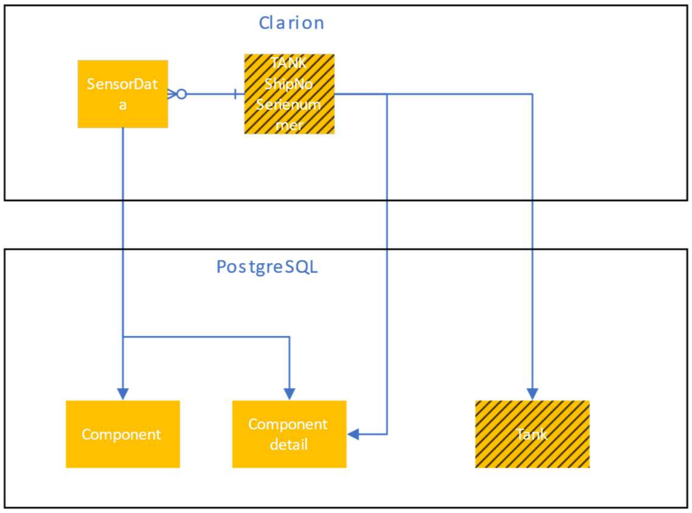


*Figure 4-7 General flow of data from the existing database's tank and sensordata tables to PostgreSQL's component, component detail and tank tables*

### 4.2.5 Attachment data

The new database will be able to store more data regarding certificates and drawings. In the Clarion database, only drawings were possible to store in *Tegning* (Drawing). These will now be stored in *attachment* with a detail table and a type table as can be seen in *Figure 4-8*.

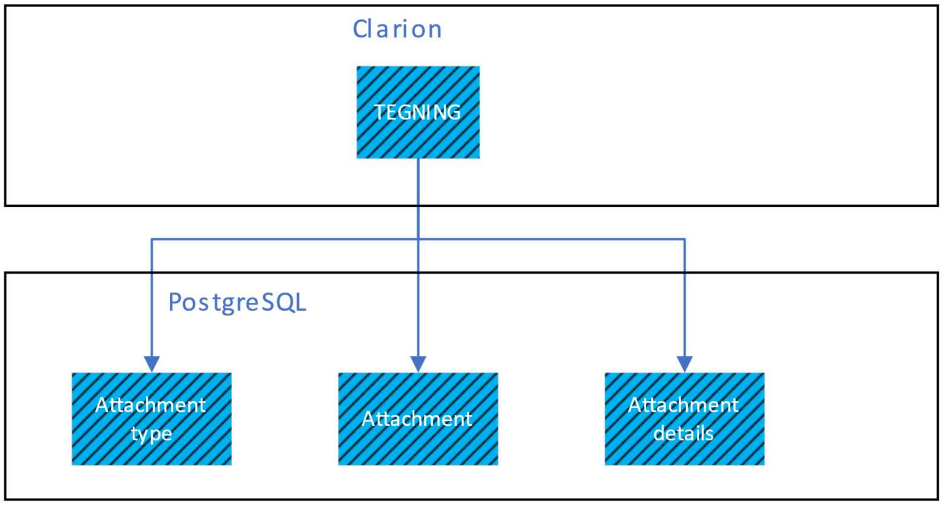


*Figure 4-8 Data flow between Tegning from Clarion to PostgreSQL tables*

## 4.3 Choice of tools

There are close to unlimited ways to migrate data, each with its own advantages and disadvantages. This subchapter will investigate a few possibilities.

### 4.3.1 Export formats

SoftVelocity Database Scanner is a program used to read tables in Clarion databases, as well as exporting tables. The following four export formats are supported: tab-, character-, ASCII value-delimited text files or as JSON arrays.

### 4.3.2 Required functionality

The following functionalities are required for the choice of tools:

- Conversion of datatypes.
- Calculations.
- Correcting typos.

Transformation and formatting are done in more than one step and it is therefore helpful to store this data temporarily.

### 4.3.3 PgAdmin with PostgreSQL

As the goal of the migration is to have the data on a PostgreSQL server, one option should be to read a text file or JSON array to SQL and transform the data there. Two databases must be created, one to store the original data of Clarion and the production database. This will require an additional ER-model. There appears no obvious way to have access to both databases at the same time, but temporary data may be stored in a JSON dump file and accessed on the receiving database. The approach of using SQL will be a relative fast execution but may require significant amount of planning, experimentation and testing.

### 4.3.4 Devart and Excel

Scanjet suggests using Devart for Excel. This is a plugin for Excel, which can connect directly to several types of databases. Excel can read all the exportable data formats from Clarion and Devart can replicate a PostgreSQL database. This can be a visual interface between Clarion and PostgreSQL. This allows for both databases to be accessible simultaneously. The downside of using Excel and Devart is that the uploading is slow.

### 4.3.5 Rad Studio with C++ or PyCharm with Python

The Shipdata application in Chapter 5 uses Rad Studio as an IDE. This also makes for an obvious choice. Both Rad Studio and PyCharm can read text and JSON files, a program may load several files and read both the Clarion tables and PostgreSQL tables. They both have several tools and plugins which can ease the workload. They offer all functionality as the other options except for visualization. As suggested in 4.3.3, this also requires a lot of planning and testing.

### 4.3.6 Picking tools

All choices seem plausible, less for PostgreSQL which doesn’t have a direct interface between the two databases. It was decided to use Devart for Excel as it is visual, and data can easily be checked for faults, pre- and post-change. It was noted well into the design that some PostgreSQL tables required significant changes, which proved hard in Excel. Python was chosen for this task as it is a high-level, easy to use programming language with many plugins for known tasks.

## 4.4 Design

Answering the questions in chapter 4.2 is best done by looking at cases which involves several of the questions asked.

### 4.4.1 Project and purchase order migration

Migrating project information is trivial, from *prosjekt* (Clarion) data is copied to Project and Project IDs are generated. When moving data from Shipdata, data must be verified that it belongs to the correct project number as this is the commonality. *Figure 4-9* shows the general flow of data.

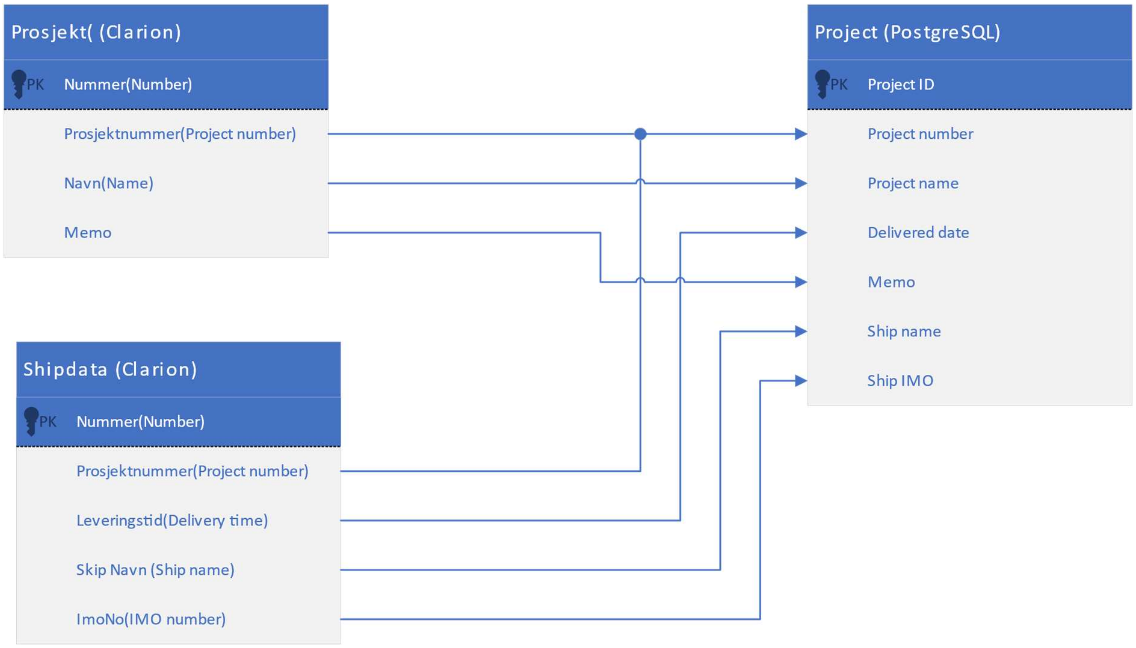


*Figure 4-9 Relocation of data regarding project table of PostgreSQL*

Purchase orders are not a part of the Clarion database, records must be created. These are for the migrated data a concatenated text string of PO: and project number. For new entries proper purchase orders will be entered.

### 4.4.2 Finding project IDs

All Clarion tables shown in *Figure 4-5* should use the route in *Figure 4-10* to find which project and purchase order they belong to. There are exceptions to this, among them are *sensordata, PCVGAdata and PC386data* which have another joint before getting *ShipNo*. These will be discussed separately.

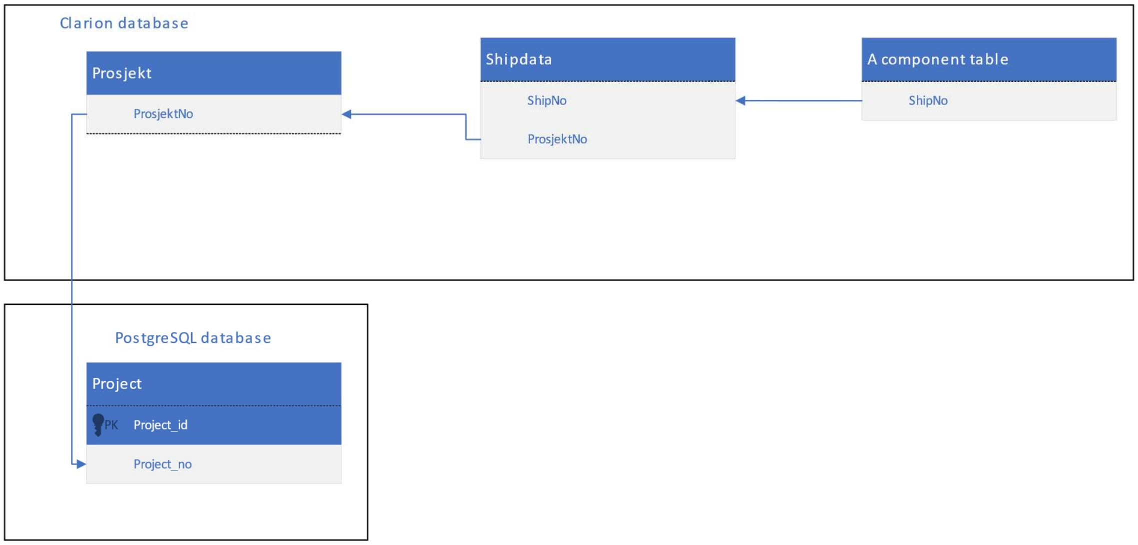


*Figure 4-10 How project IDs are derived for components*

A component typically has a *ShipNo*, this can be used to find which project number it belongs to. The project number will be the same in both the Clarion database and the PostgreSQL database. From this project number, a project ID has been generated for the PostgreSQL database.

Later, this process will be referred to as Find project IDs.

### 4.4.3 Component migration

A Clarion component must be split up and its data relocated to two tables in the PostgreSQL database. In *Figure 4-11* a general component table is used to show how most component tables will be transformed. Tables which are not included are *Sensordata, PCVGAdata* and *PC386data*, these will be covered in their own sections. A component must go through the Find project IDs process to find a purchase order. For the migrated data, purchase order is simply "PO:project number". A component ID must be generated for each entry and the serial number inserted with correct PO.

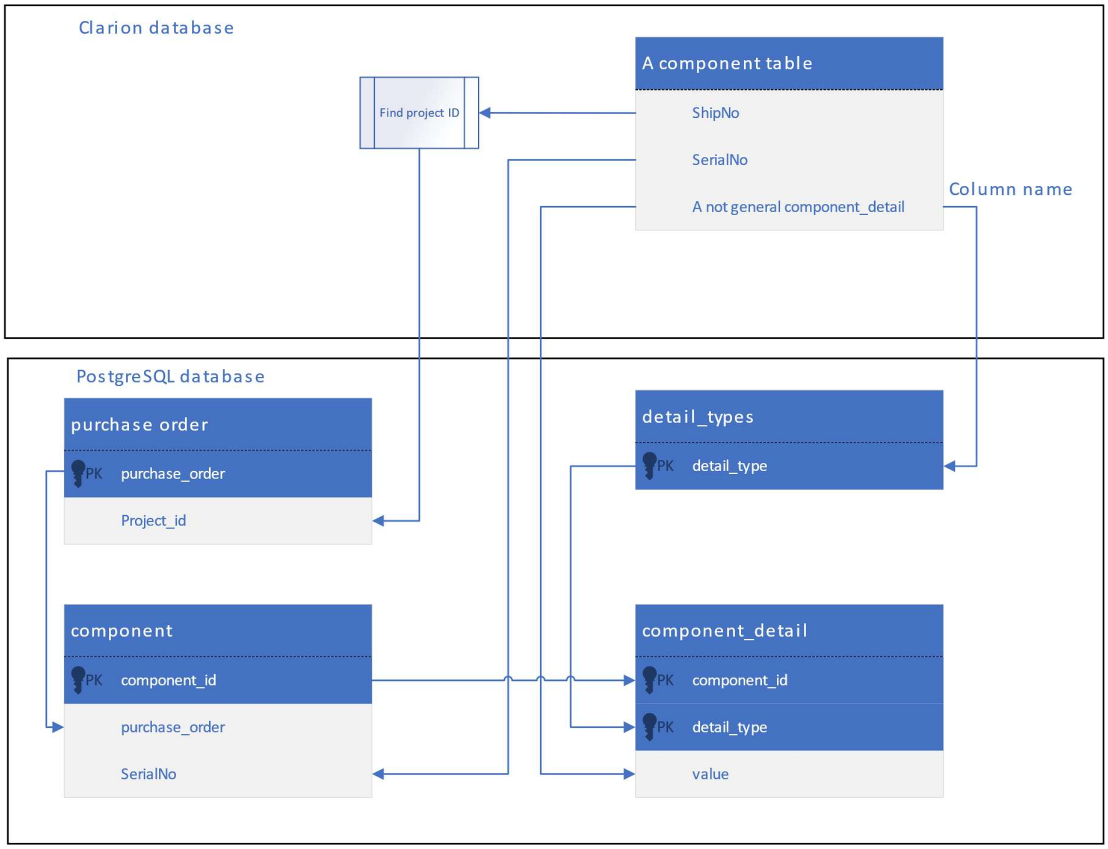


*Figure 4-11 Clarion to PostgreSQL transformation of a component table*

A component in the Clarion database may have several columns that doesn't fit the component table of the new database. These columns must be transformed to fit *detail_type* and *component_detail,* and be stored in the correct table. The transformation is described in chapter 4.4.4.

Column names from a component in the existing database are given to *detail type* which will store all detail names. The detail type is then distributed to any of the detail tables. The detail tables have a column for the detail value, this is where the column value from the table in the Clarion database is stored. A complete list of *detail_type* with new and old names can be seen in Appendix E.

*PC386data* and *PCVGAdata* have an extra step in finding the purchase order, as shown in *Figure 4-12* and *Figure 4-13*. *Sensordata* will be covered in *Chapter 4.4.6 Tank and sensor data migration* as they are closely related.

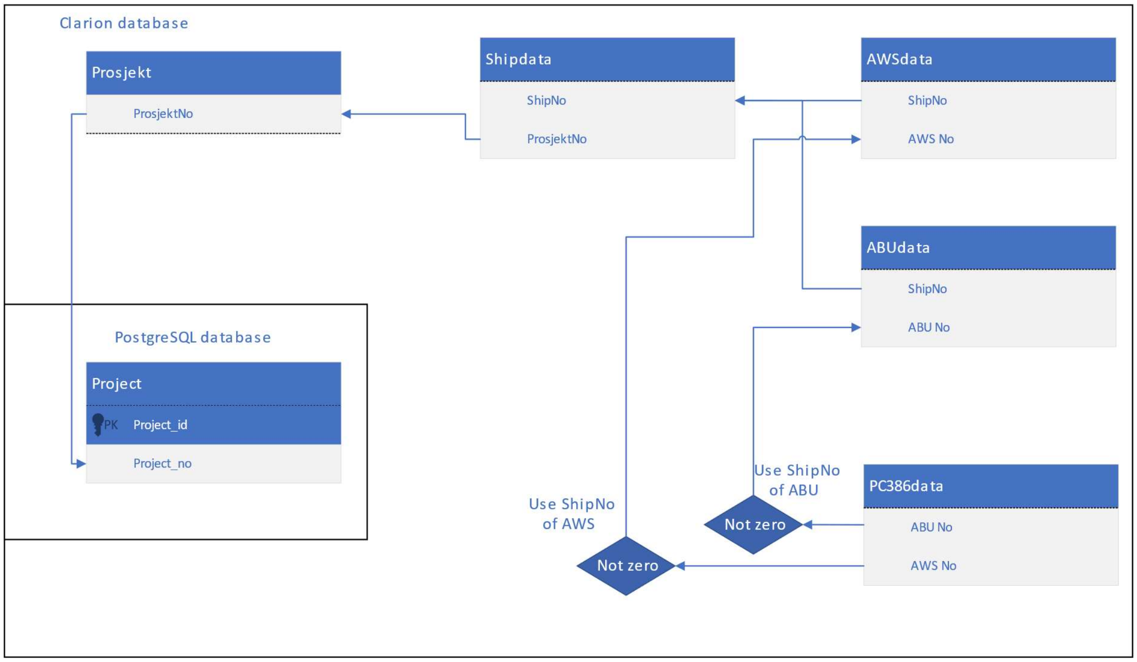


*Figure 4-12 Finding project ID for PC386Data*

A *PC386data* entry may be installed on either an AWS or and ABU and must therefore in every case be decided which is true and then a PO may be found.

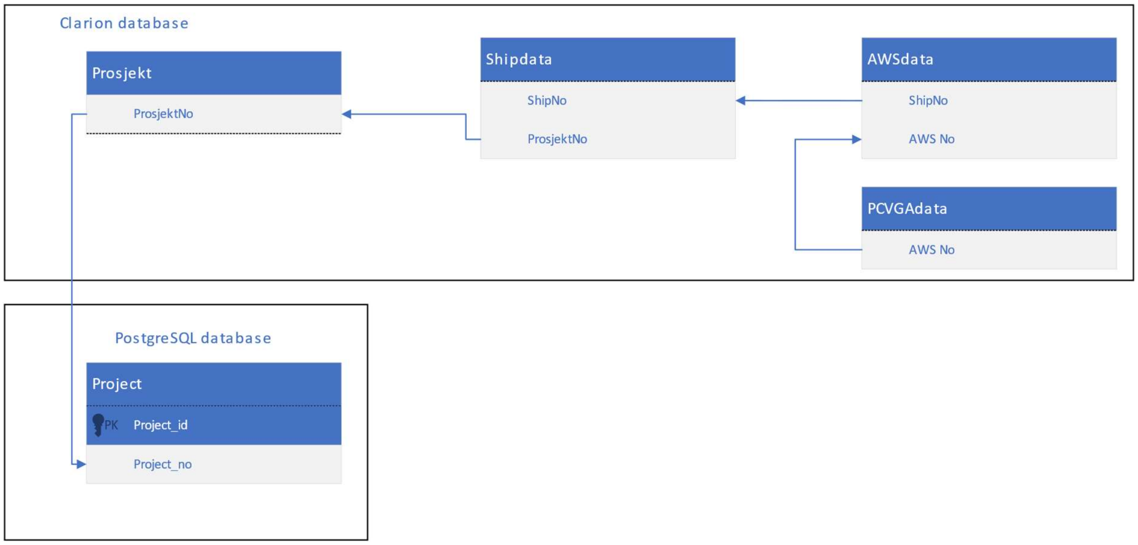


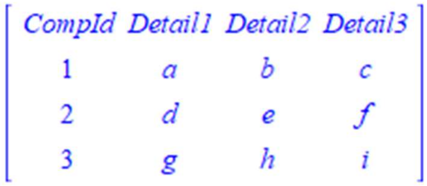
*Figure 4-13 Finding project ID for PCVGAdata*

*PCVGAdata* needs only one more joint to find the PO.

A component has an article ID which corresponds to an article number. This article number describes many components on an abstract level. Article numbers don’t exist in the Clarion database and must be given by Scanjet Ariston AS. There may however be many article numbers for a given type of component. Depending on the table there are columns which separates one article from another. *Sensordata* will not be accounted for regarding article IDs. Appendix F shows how to separate article numbers.

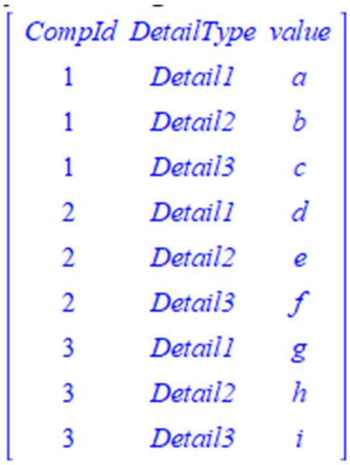
### 4.4.4 Python script

The transformation described in chapter 4.4.3 regarding *component_details* are not so easily executed in Excel. Therefore, a Python script will be developed to work around this issue. The script must be able to restructure matrices of data as shown in *Figure 4-14* and *Figure 4-15*.



| CompId | Detail1 | Detail2 | Detail3 |
|---|---|---|---|
| 1 | a | b | c |
| 2 | d | e | f |
| 3 | g | h | i |

*Figure 4-14 Data structure of columns from the existing database which doesn’t fit in the component table of PostgreSQL*



| CompId | DetailType | value |
|---|---|---|
| 1 | Detail1 | a |
| 1 | Detail2 | b |
| 1 | Detail3 | c |
| 2 | Detail1 | d |
| 2 | Detail2 | e |
| 2 | Detail3 | f |
| 3 | Detail1 | g |
| 3 | Detail2 | h |
| 3 | Detail3 | i |

*Figure 4-15 Data structure post Python script, now fitting the PostgreSQL database*

### 4.4.5 Check match

In this subchapter a new subprocess will be introduced. This process will use serial numbers from S*ensordata* and check if they are found in one of four columns. If found, it will give a value to the new *detail_type* called *Position* in *component_detail*. If not found it will give a null value. A flowchart of the process is seen in *Figure 4-16*.

This process is used in *Figure 4-19*.

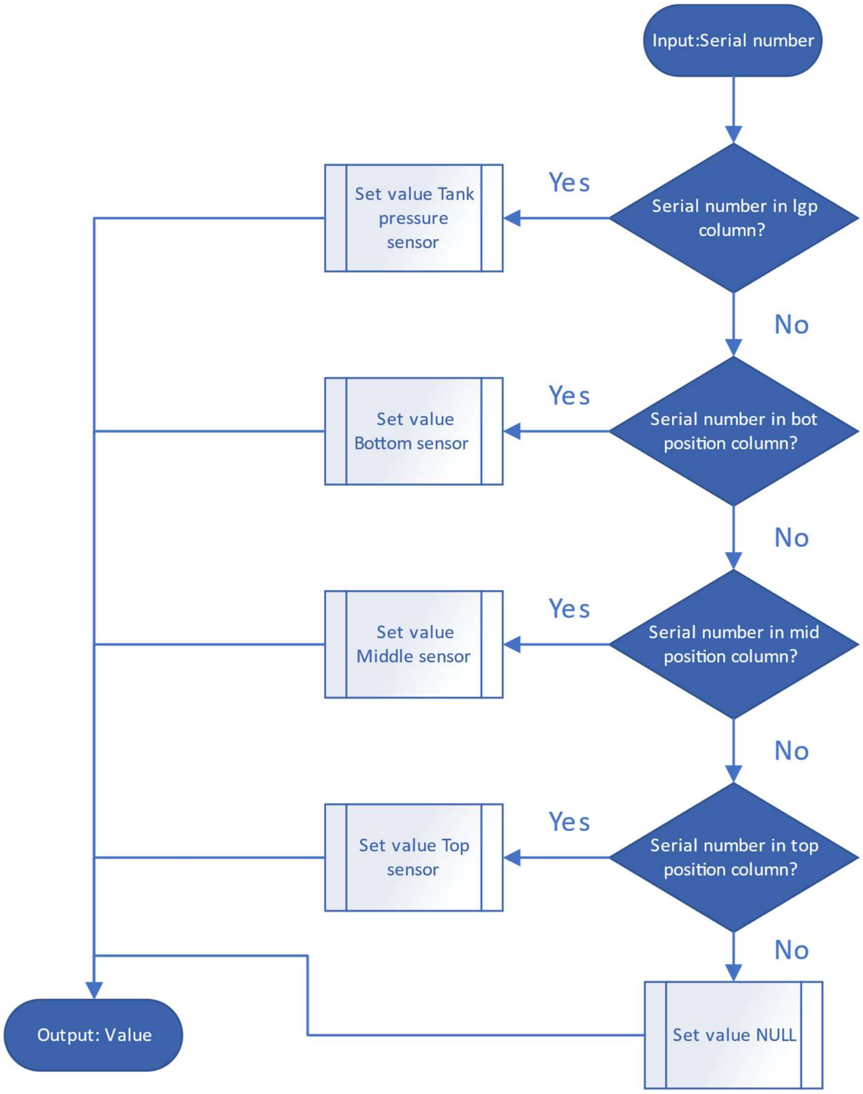

Input:Serial number
Serial number in lgp column?
Yes
Set value Tank pressure sensor
No
Serial number in bot position column?
Yes
Set value Bottom sensor
No
Serial number in mid position column?
Yes
Set value Middle sensor
No
Serial number in top position column?
Yes
Set value Top sensor
No
Set value NULL
Output: Value

*Figure 4-16 Check match subprocess to find sensordata position on tanks*

### 4.4.6 Tank and sensor data migration

The *tank* table from the existing database holds data concerning tank specifications, but also about which sensors are mounted and which position it is mounted in. The question regarding how to transfer existing database's *tank* table to the new model is a trivial one and is explained in *Figure 4-17*.

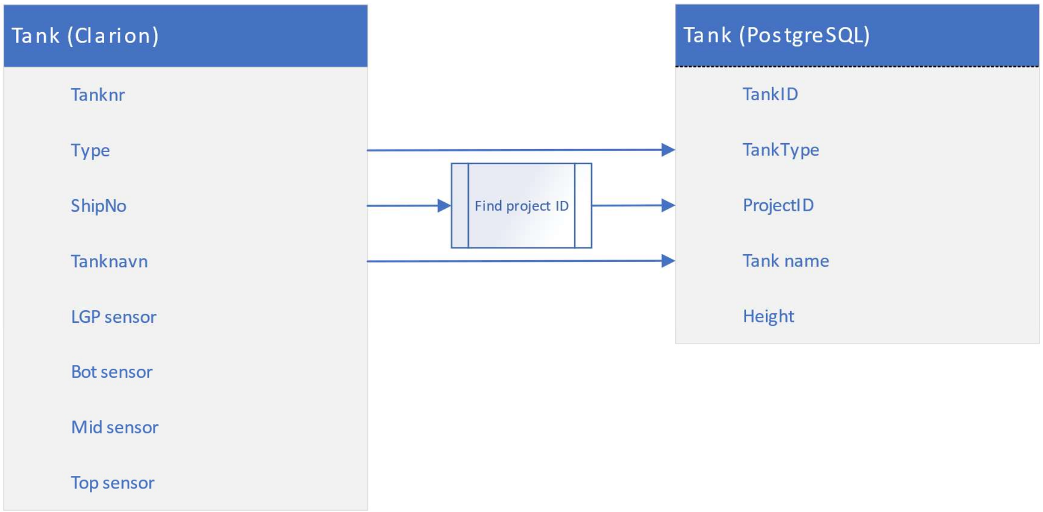


*Figure 4-17 Finding project id for tanks through the process described in Chapter 4.4.2*

In the new database the relation between tank and component are different. Previously a tank could have up to four sensors installed, and the installed sensors were found in the tank table. In the new database, a component may have a *tank_id* to indicate which tank it's installed on. As tank names are not unique this may prove tedious as each sensor must find a matching tank name and correct *project_id* to verify the *tank_id*.

The flow of this is shown in *Figure 4-18* starting in *sensordata* where a serial number is given. It then checks which entry it's installed on, this gives a tank name (*tanknavn)* and a ship number *(ship_no)*. From the ship number a *project_id* can be derived and must be found in *Tank* in the new database. Additionally, the tank name must match. This will indicate which *tank_id* the component is installed on.

Because most tank data are mirrored with a new primary key, the previous task may be simplified by using an additional sheet to find the correct *tank_id.* This Excel sheet will have the *tank* data from the Clarion database, and additionally the *tank_id* from the new database as this ID is an incrementing value for each tank.

Regarding the PO this procedure is the same as in earlier components described in chapter 4.4.3. Lastly, the serial number is copied to the appropriate *component* record.

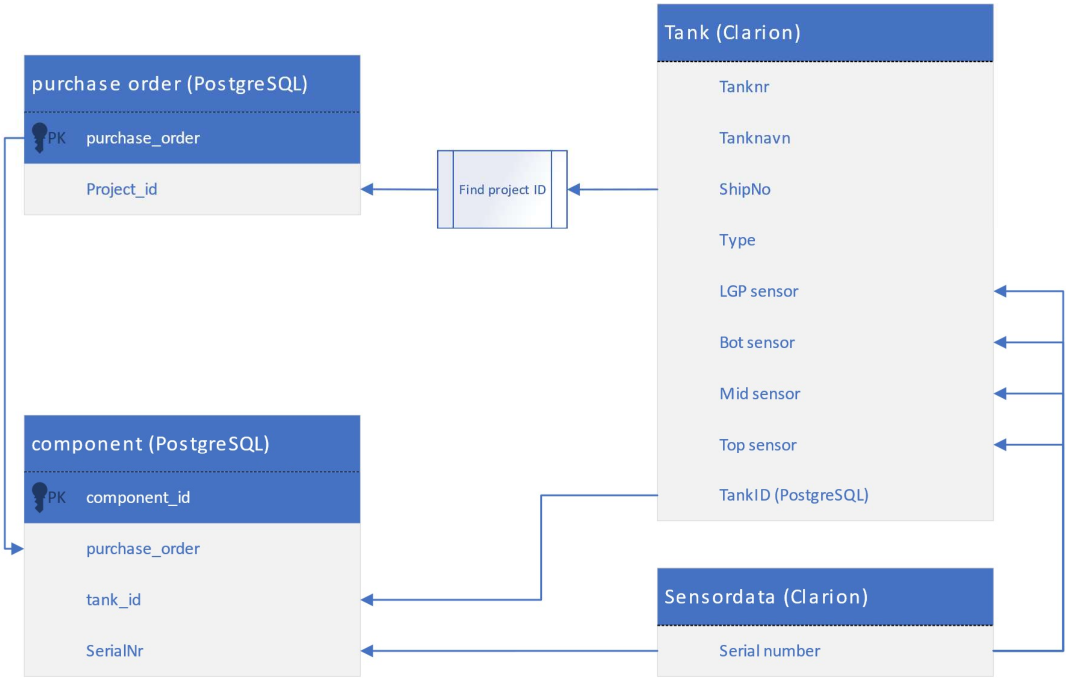


*Figure 4-18 Deriving component data for components in sensordata*

The sensor position on the tank will in the new database be stored per sensor. As this does not fit the general *component* table, it will be stored in *component_detail* table with the *detail_type* "Position". Each sensor must then look up which column the serial number is found in the Clarion database's *tank* table, and have the appropriate position written as a detail on the component. The process is shown in *Figure 4-19*.

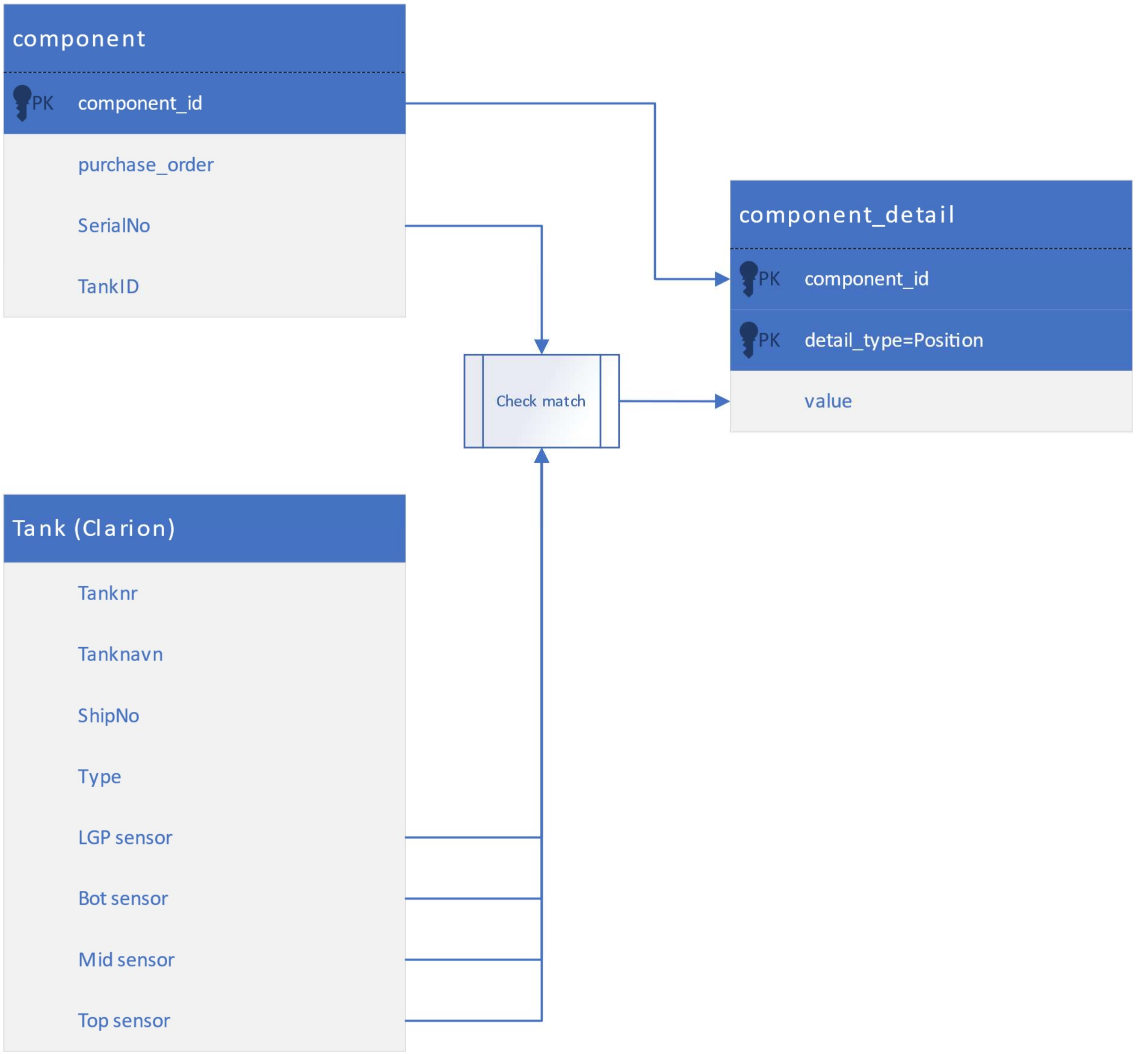


*Figure 4-19 Finding sensordata mounting position and which tank it is mounted on and how it now is stored.*

*Sensordata* in the Clarion database keeps track of calibration results on all sensors. There are three types of calibration[1] details in the Clarion database, one for the general sensors and two for a special sensor P906 which is a temperature compensating pressure sensor.

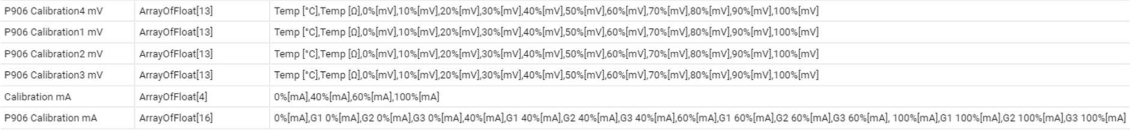
The first calibration detail is composed of 4 measurements at 0%, 40%, 60% and 100% of any given range of a sensor.

The second calibration type measures pressure as millivolts on 11 points at a given temperature. For the P906 sensor, four of these are needed at varying temperatures.

[1] Calibration is a measurement of an instrument comparing to a standard, allowing the calibrator to adjust the instrument to measure correctly.

The third uses the previously mentioned calibration type and creates a polynomial for each series. From each of the polynomials it should find milliampere equivalents at four points (0%, 40%, 60% and 100%). This gives 16 values which are used as an interpolated calibration characteristic for varying temperature.

To fit the new model of *detail_type* (text), *value* (text) and keeping the data in context, it was decided to store one calibration measurement series as one text. The detail types are shown in *Figure 4-20*. As one detail only can belong to one component, four P906 calibrations are created.



| | | |
|---|---|---|
| P906 Calibration4 mV | ArrayOfFloat[13] | Temp [°C],Temp [Ω],0%[mV],10%[mV],20%[mV],30%[mV],40%[mV],50%[mV],60%[mV],70%[mV],80%[mV],90%[mV],100%[mV] |
| P906 Calibration1 mV | ArrayOfFloat[13] | Temp [°C],Temp [Ω],0%[mV],10%[mV],20%[mV],30%[mV],40%[mV],50%[mV],60%[mV],70%[mV],80%[mV],90%[mV],100%[mV] |
| P906 Calibration2 mV | ArrayOfFloat[13] | Temp [°C],Temp [Ω],0%[mV],10%[mV],20%[mV],30%[mV],40%[mV],50%[mV],60%[mV],70%[mV],80%[mV],90%[mV],100%[mV] |
| P906 Calibration3 mV | ArrayOfFloat[13] | Temp [°C],Temp [Ω],0%[mV],10%[mV],20%[mV],30%[mV],40%[mV],50%[mV],60%[mV],70%[mV],80%[mV],90%[mV],100%[mV] |
| Calibration mA | ArrayOfFloat[4] | 0%[mA],40%[mA],60%[mA],100%[mA] |
| P906 Calibration mA | ArrayOfFloat[16] | 0%[mA],G1 0%[mA],G2 0%[mA],G3 0%[mA],40%[mA],G1 40%[mA],G2 40%[mA],G3 40%[mA],60%[mA],G1 60%[mA],G2 60%[mA],G3 60%[mA], 100%[mA],G1 100%[mA],G2 100%[mA],G3 100%[mA] |

*Figure 4-20 Calibration details and its format.*

### 4.4.7 Attachment

The task of migrating attachments is trivial, there are only one table involved from the Clarion database, *Tegning* (Drawing). Most columns will go to one table, *attachment*, with one detail to the *attachment_detail* table and unique values of type will go to *attachment_type*. *Figure 4-21* illustrates the flow of data during migration to new database.

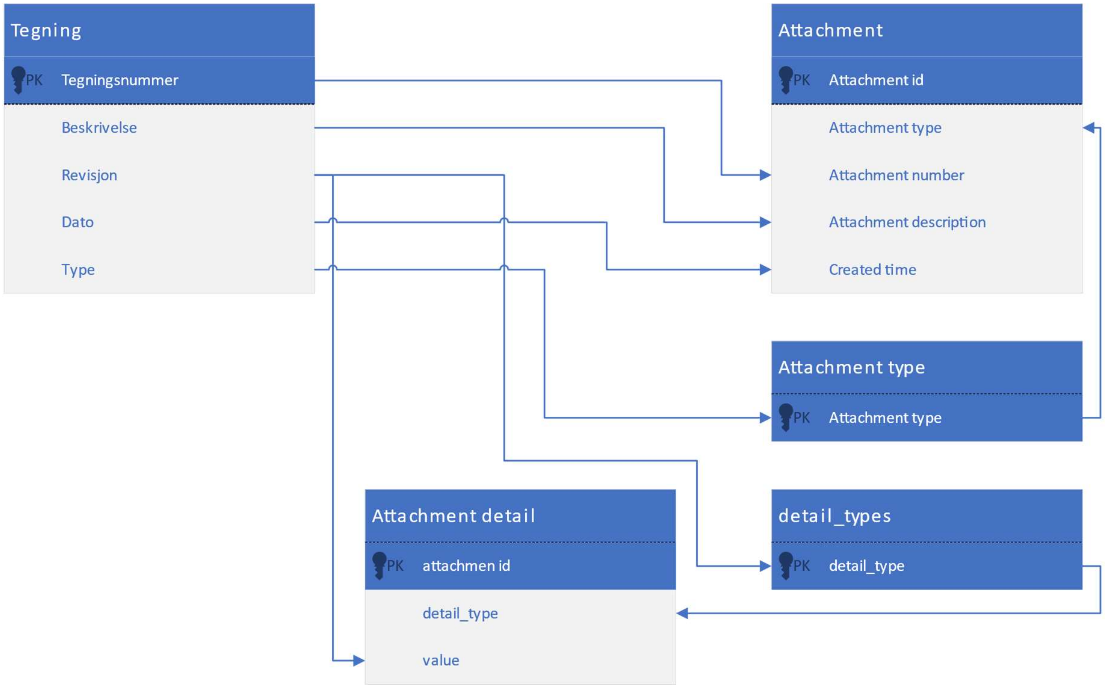


*Figure 4-21 Flow of data when migrating Tegning table*

# 5 Application – Requirements and design

To utilize the new database in a convenient and efficient way, Scanjet has requested a desktop application to be made. The following chapter outlines the design of this application, as well as list the interpreted requirements in the opening subchapter.

## 5.1 Requirements

This subchapter describes requirements and needs discussed with the customer regarding the desktop application.

A general requirement for the application is that it should be written in C++ using RAD Studio as an integrated development environment.

As a guiding principle, all parts of the application should be implemented in a way that allows changes to be made in the future. Either through an easy-to-understand process of extension, flexible code layers, or simply well commented code. Of course, it is desirable that more than one of these apply at any given point.

Because access restrictions are imposed server-side based on user roles (see chapter 3.1.2), the application itself doesn't have to take security too much into consideration. However, it is important that the connection with the server remains strongly encrypted, and no authentication data is stored locally. To handle this connection, one of the software libraries integrated for RAD Studio (TPgConnection from PgDAC) has been utilized. Other than that, the application does *some* access filtering based on the same user roles as before. More on that later in this chapter. Whether or not this is a secure approach is discussed in chapter 7.

Below are some of the most critical functional requirements for the desktop application. A more detailed specification can be read in Appendix B.

### 5.1.1 Adding data – New records

The application should include functionality for adding data and details for each table in the database. It should also consider the relationships between tables when adding data, e.g. every purchase order should be linked to a company.

### 5.1.2 Modify data – Update existing records

The application should allow users to search for desired data through a specialized search function. When matching data is found, users should be able to update the record.

### 5.1.3 Grid view – Browse data

The application should allow users search for several data records within a grid view. A filter function should be included to narrow down the search.

### 5.1.4 Configuration files and programming of sensors

Scanjet's systems rely on a configuration file to function correctly. The application should include a function to generated it by collecting data from the database. The application should also include a function to program sensors. These functions should be implemented by Scanjet's engineers themselves. However, the application should facilitate room for the functions to be implemented later.

### 5.1.5 Log in function

The application should have a log in functions where users must enter a username and password before being able to access any data from the server. Authentication should be done by the database server and not the application.

### 5.1.6 Print reports

The application should allow users to print reports based on stored data. The following reports should be included: a system overview, lists of sensors or labels for one system, sensors which is not connected to any projects or tanks, lists of all sensors and a total amount of sensors.

## 5.2 Graphical user interface

The graphical user interface (or simply GUI) was the first part of the entire application to get its design specified. The reader should keep this in mind, as the illustrations in this subchapter mostly served as tools in directing developers and Scanjet Ariston toward a mutual goal. By reading through this subchapter, the intended result of the product should become more apparent.

### 5.2.1 Start page

When starting up the application the user is prompted with the start page shown in *Figure 5-1*. Other menus and function should not be accessible before a user has logged inn successfully, e.g. *New* and *Browse.*

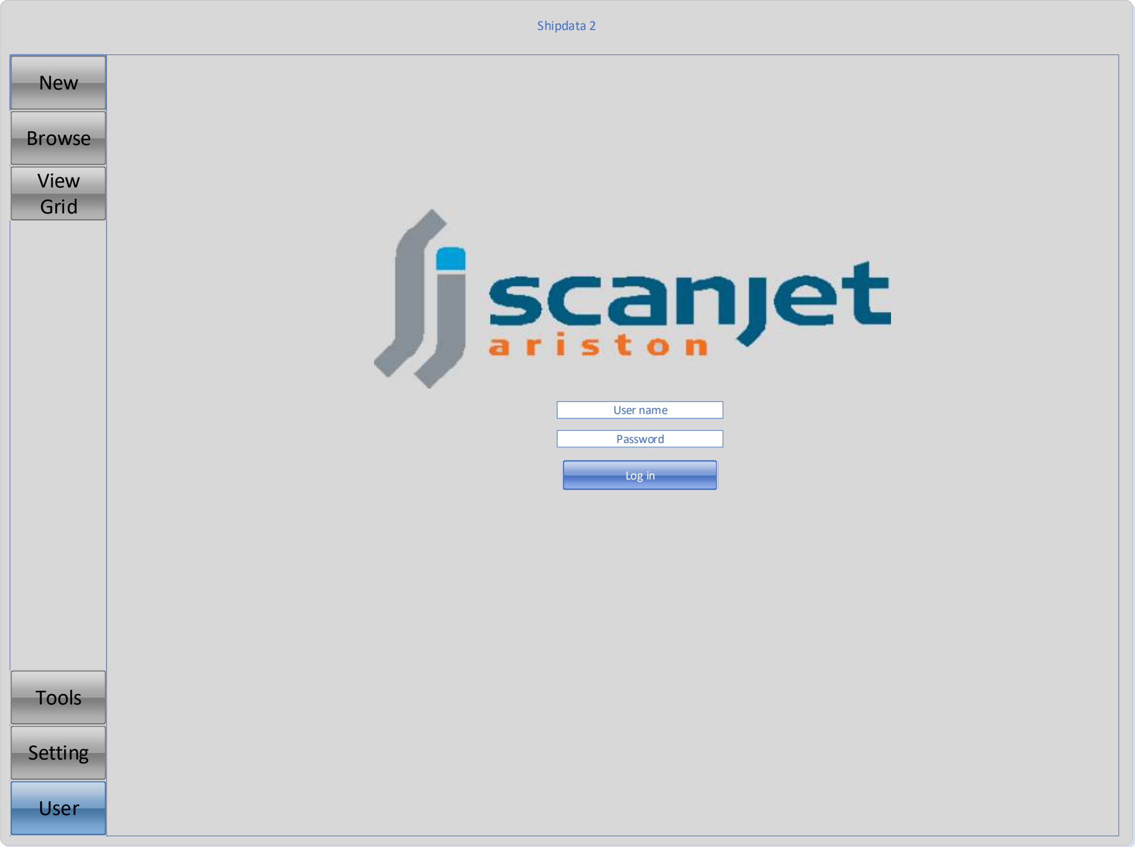


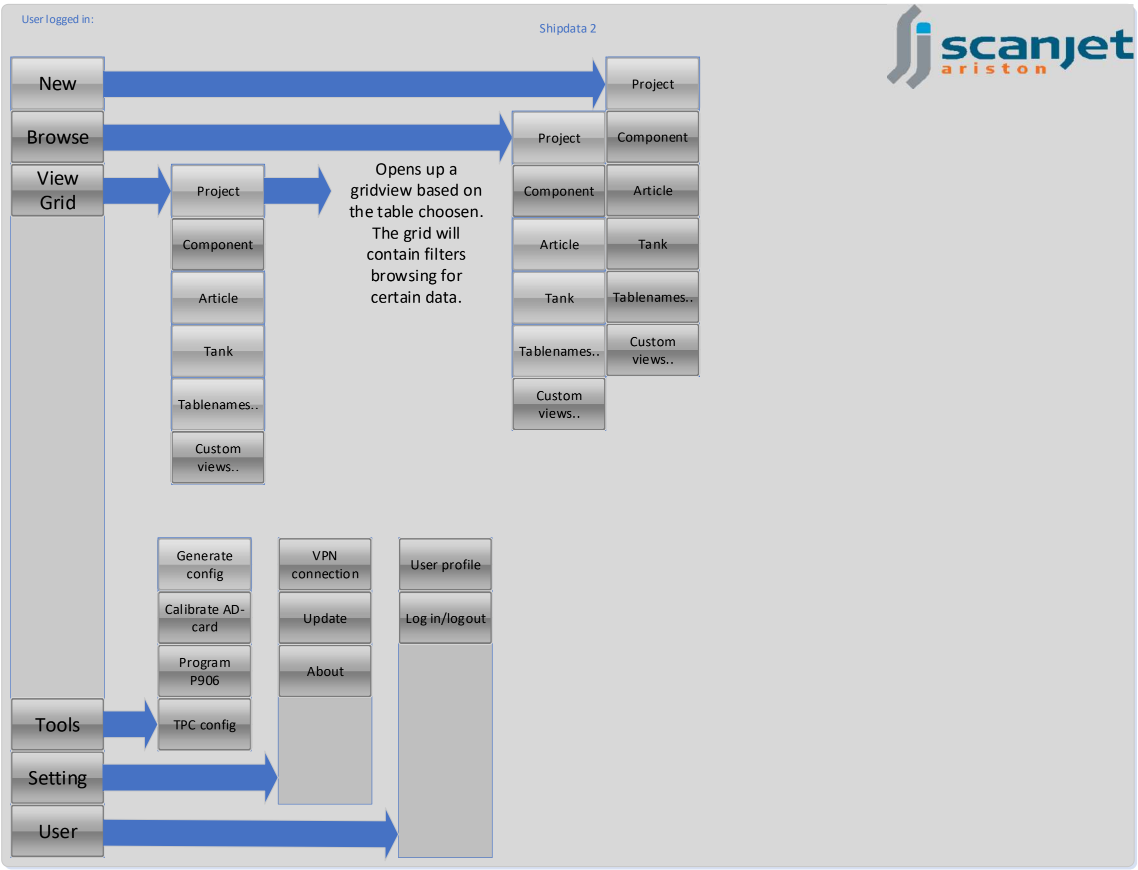
*Figure 5-1: Design concept of start page with log in function.*

### 5.2.2 Menu

*Figure 5-2* shows a complete design of the menu structure for the desktop application. The first menu layer includes six main functions. The functions N*ew*, B*rowse* and *View Grid* opens a second menu layer containing buttons for each table name in the database. A second layer button will open a corresponding panel with information from the selected table.

The *Tools* button shown in *Figure 5-2* opens a second menu layer containing buttons for generating configuration files, calibration and programing of sensors. These functions will be implemented by Scanjet Ariston’s engineers.

The two last main functions, *Settings* and U*ser*, leads to different options for authentication, user profiles, connection parameters etc.

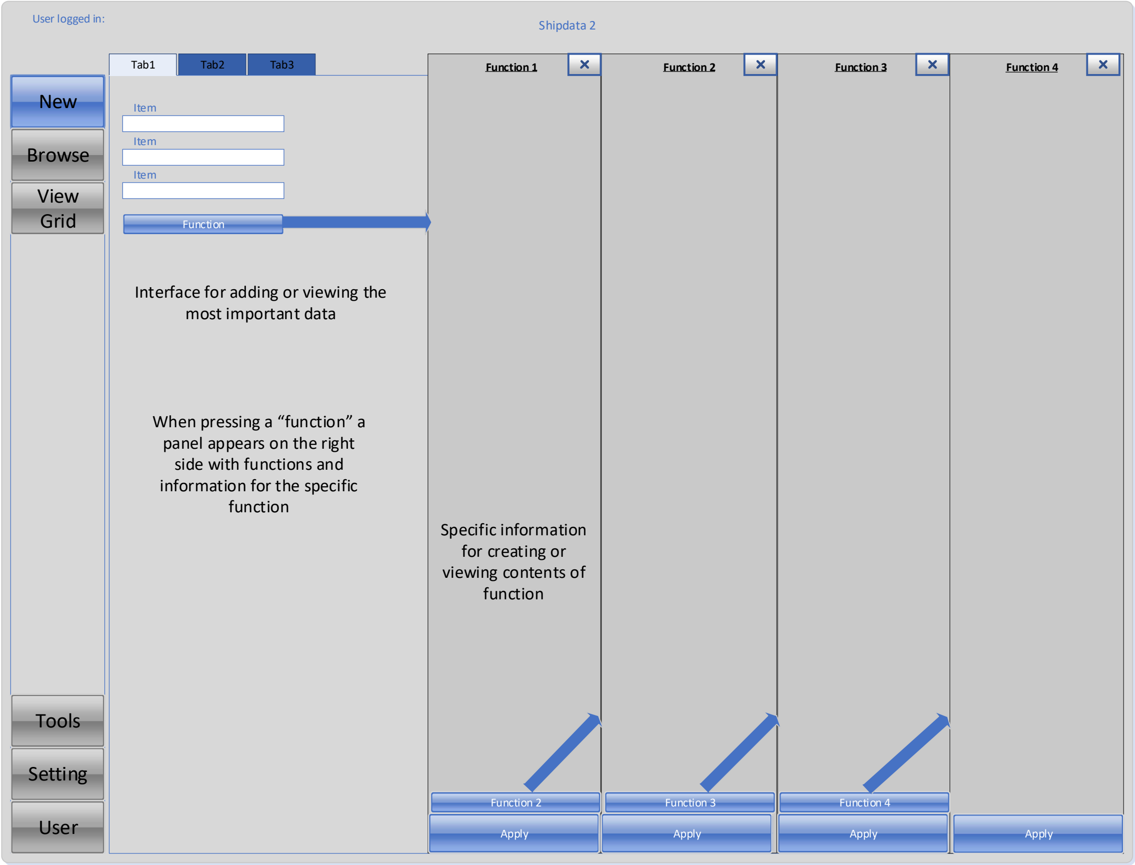


Opens up a gridview based on the table choosen. The grid will contain filters browsing for certain data.

*Figure 5-2: Design of menu structures.*

## 5.2.3 Panels

*Figure 5-3* illustrates a general design of a tab for browsing from or adding data to the database. The idea is, when a menu layer two item within main functions *New* and *Browse* is pressed, a panel with the most important information is shown. If there is a need to add or view further information, a button is generated. When pressed, a new panel will be shown on the right side of the screen containing data for the function. This concept can be done multiple times where more information can be added or viewed within subfunctions. E.g. when adding a new project, it is possible to view which tanks are added to the project. The first panel shows a list of tanks associated with the project and a function button to add new tanks to the project. This opens a new panel with the correct objects to add information regarding a new tank. When applied, the new tank is added to the list of tanks.



Interface for adding or viewing the most important data

When pressing a “function” a panel appears on the right side with functions and information for the specific function

Specific information for creating or viewing contents of function

*Figure 5-3 A general design of panels*

## 5.2.4 Grid view

The third main function *View Grid* opens a second menu layer to select which table is desired to view in the grid. Initially this will show all data in the selected table. This enables the user to do an advanced search for specific data in the database without knowing exact search parameters. The design is based on a grid with the possibility to add predefined or user defined filters to search for data. *Figure 5-4* illustrates the grid view panel. The users select which table to view in the grid by clicking one of the buttons in the second menu layer. Filter options are presented on the right side of the panel. It is possible to filter out columns from the view by clicking the checkboxes or apply user defined filters.

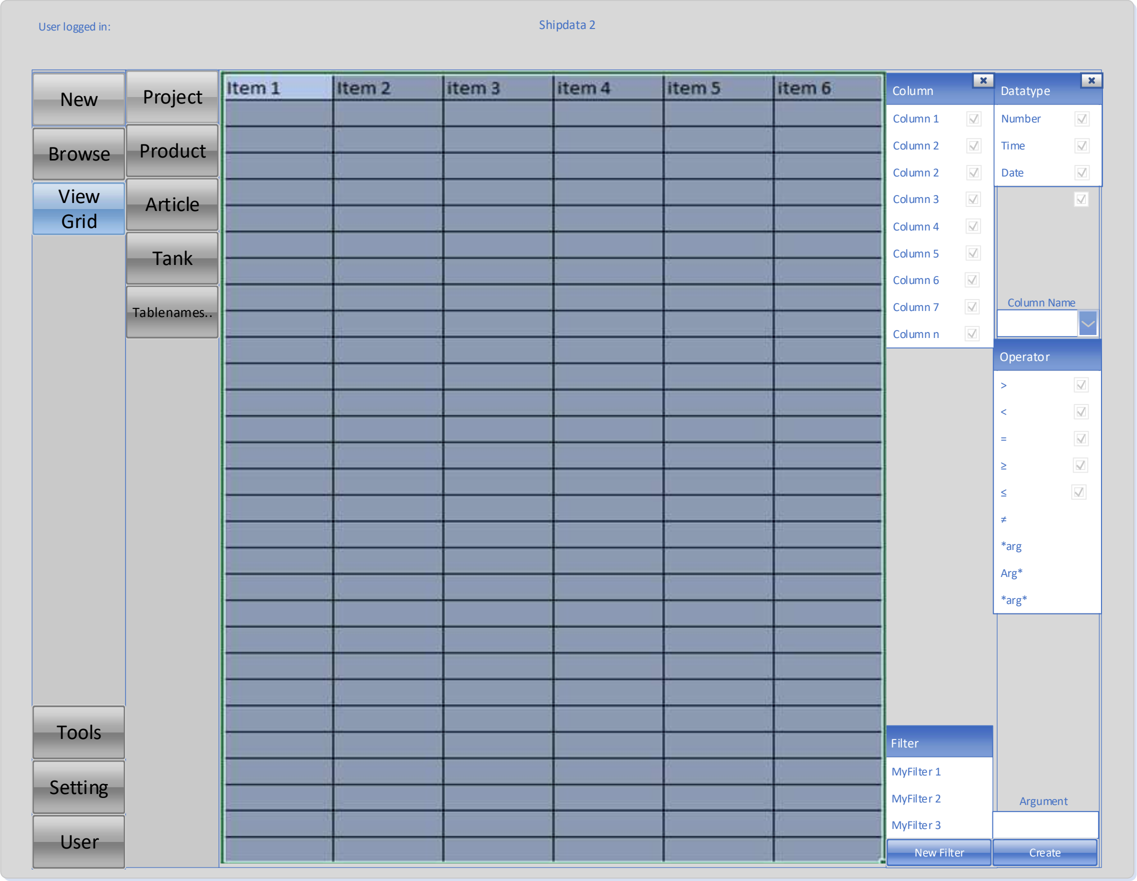
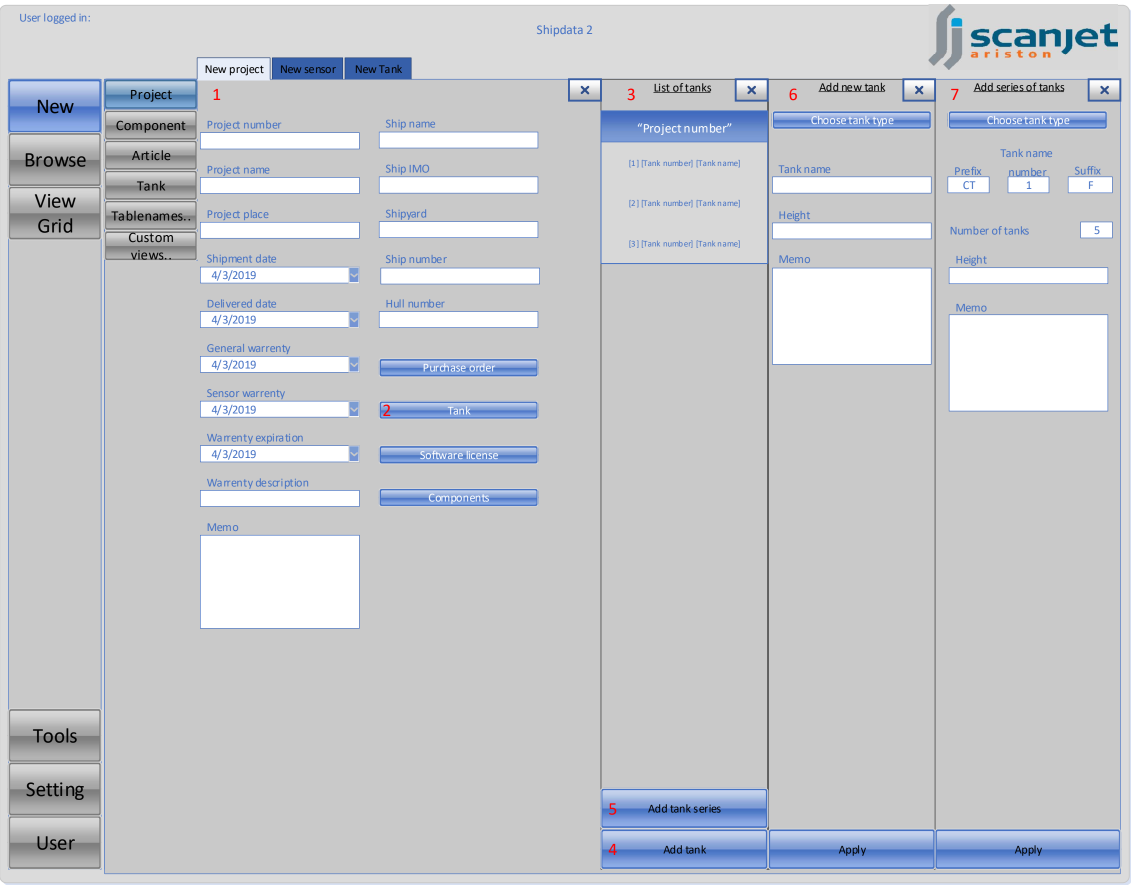


*Figure 5-4: Design concept of view grid function*

### 5.2.5 New record / Edit record

*Figure 5-5* shows an example for adding a new project with the application. The main panel (1) contains the information needed to add a project, e.g. project number and project name. A project is connected to one or more tanks which can be accessed by pressing the function button *Tank* (2). This opens a new panel (3) on the right side with a list of tanks connected to the project and a function button for adding one new tank (4) or a series of tanks (5). When adding a single tank, a panel (6) will be opened to the right with the necessary information to fill inn regarding one tank. If adding a series of tanks, a panel (7) will be opened instead. The user must then define the tank name with a prefix, a number which is to be iterated, a suffix and the number of tanks to add.

*Figure 5-5* illustrates both functions, adding a tank and a series of tanks, however the functions cannot be done at the same time.

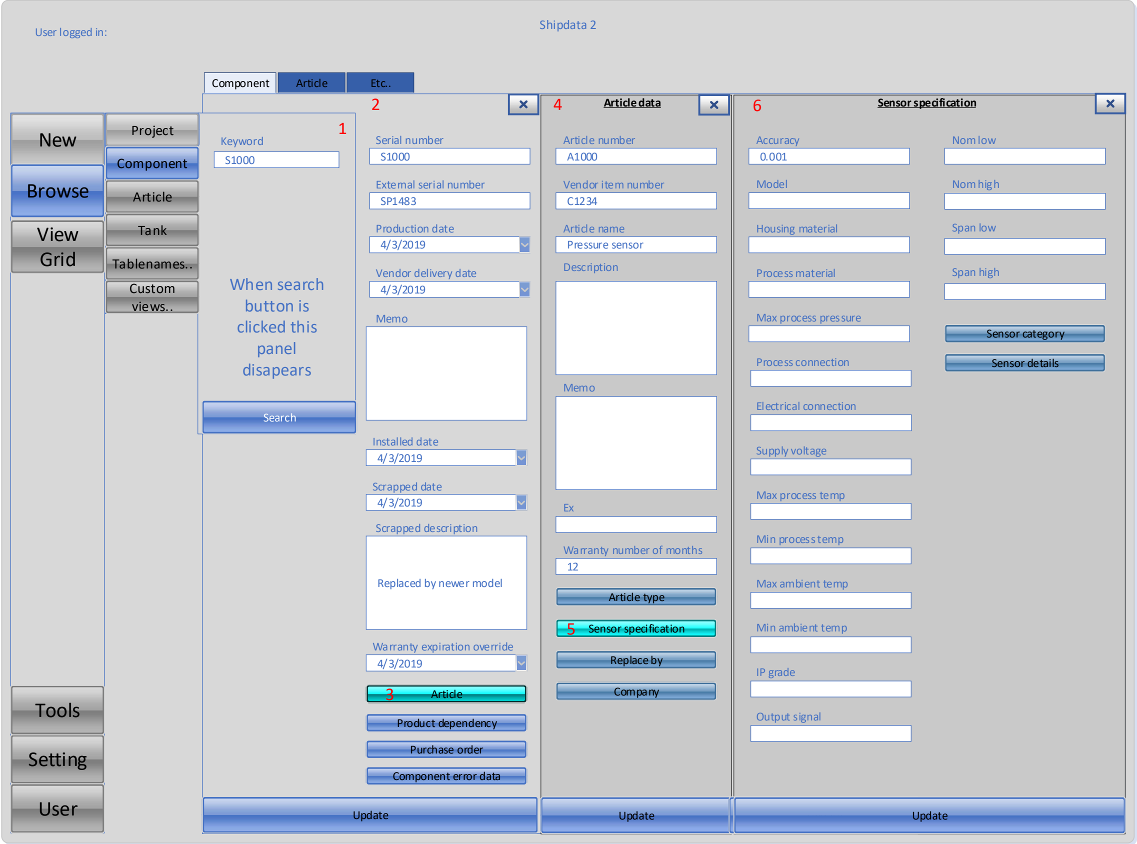


Shipment date: 4/3/2019
Delivered date: 4/3/2019
General warrenty: 4/3/2019
Sensor warrenty: 4/3/2019
Warrenty expiration: 4/3/2019
Warrenty description

Prefix: CT
number: 1
Suffix: F
Number of tanks: 5

*Figure 5-5: Design example of adding a new project with connected tanks*

*Figure 5-6* illustrates an example of how the browse function will be presented. The example includes a search for a component by serial number, viewing article data and sensor specifications for the component. After navigating through the menu, a panel (1) opens where a keyword must be specified before the search can be executed. If there is a match for the keyword, a second panel (2) opens containing the most important information for the item, in this case component with serial number S1000. The user can then view the components article data by pressing the button *Article* (3). A third panel (4) containing article data for the component is presented. In this case the article is a sensor and it is possible to view sensor specifications by pressing the button *Sensor specification* (5) which opens a panel containing the associated information in the *sensor_specification* table. If any information should be edited, simply write in the correct information and press update.



Keyword: S1000

Serial number: S1000
External serial number: SP1483
Production date: 4/3/2019
Vendor delivery date: 4/3/2019
Memo:
Installed date: 4/3/2019
Scrapped date: 4/3/2019
Scrapped description: Replaced by newer model
Warranty expiration override: 4/3/2019

Article number: A1000
Vendor item number: C1234
Article name: Pressure sensor
Description:
Memo:
Ex:
Warranty number of months: 12

Accuracy: 0.001
Model:
Housing material:
Process material:
Max process pressure:
Process connection:
Electrical connection:
Supply voltage:
Max process temp:
Min process temp:
Max ambient temp:
Min ambient temp:
IP grade:
Output signal:
Nom low:
Nom high:
Span low:
Span high:

*Figure 5-6: Design example of browse function. Searching for a component, a sensor, with article details and sensor specification data*

# 5.3 Code structure

To achieve the goal of an extensible system, the code structures need not only be thoroughly documented and commented, but also serve as solid foundation for future development. In this sub-chapter the higher-level structures will be visualized, and their underlying philosophies explained. By the end of chapter 5 the reader will have a fundamental understanding of how the application works and how to properly extend its functionality in the future.

## 5.3.1 Overview

The application was designed to consist of four main code parts. The first is the predefined windows form provided by the IDE. It consists of a lot of hidden code, including, most notably, the program loop as well as event handlers. This report referres to this part as *mainsource*. Here, the application gets tied together.

Then there are the customized GUI control classes, imported through the *s_gui* header file. This file attempts to define class hierarchies for panels, buttons, tabs and other GUI components. Some of which are derived from Embarcadero classes. E.g. panels and buttons might need class members for storing which database table it's working with.

Third in line is the configuration layer, named *s_datastructure*. Most importantly, it describes the database structure, but also resources for GUI elements.

Finally, there is the SQL interface, named *s_sql*. It simply defines functions for connecting to and running queries in a database.

*Figure 5-7* shows how the different parts are dependent on each other.

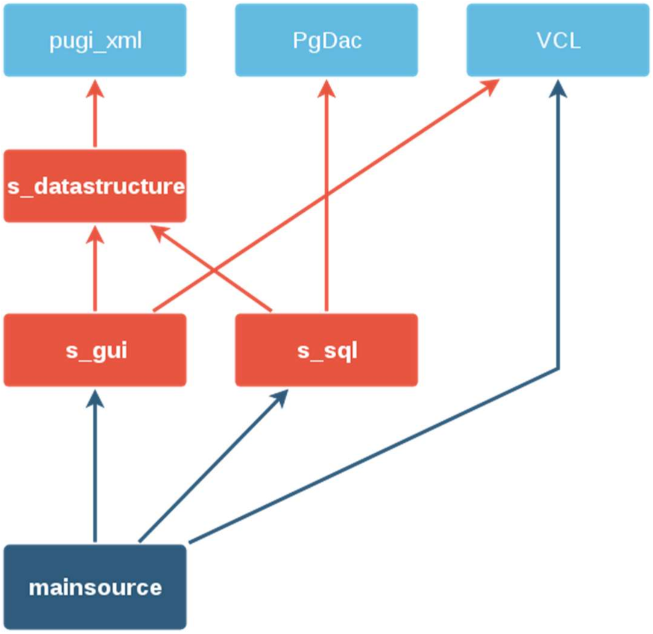


*Figure 5-7: High level code dependecies. The rectangles in the top row all represent external libraries.*

A central idea the design is based upon, is the ability to change the database structure without having to change the code. Of course, some changes somewhere must happen, unless exhaustively detailed by the database itself, but the project aimed to make these changes isolated to a single application layer. Therefore, the application otherwise stays unopinionated on the matters handled by this new layer.

All GUI components introduced in this project are part of VCL. The nature of this library, and how it's used in the project is described below.

### 5.3.2 VCL controls and derived classes (s_gui)

This is a proprietary library for the Delphi language, made and ported to C++ by Embarcadero. Usage is in a few cases limited due to Delphi compatibility, but this rarely leads to any serious problems.

To cover the GUI specification, two components were identified as being more important than others: panels and buttons. Other GUI components are indeed present, but they don't necessarily demand extended functionality from their original classes. Panels and buttons require additional class members for different reasons described below.

Panels are the main building blocks of the GUI. All other visual components are supposed to stay within these rectangular shapes. The most appealing reason for this is to have one unified way of positioning other components. Positions, while within panels, are relative to panel edges. Something which simplifies the placement of components and leaves the updating mechanisms of positions to the base class.

Panels have been designed to store the pointers for its child components instead of storing them all directly in the form class. The VCL class *TPanel* will therefore serve as the base class for a family of new panel classes. These new classes are primarily split between static and dynamic. In this case meaning panels that should only get instantiated once, and panels that may have any number of instances, respectively. Dynamic panels are then branched into the inner and outer category. Inner panels belong within outer panels. Outer panels are the panels that populate the tab vector (explained in a later section in this chapter [4]) and grow to the right on the screen. *Figure 5-8* shows the inheritance tree of the panel classes. Note that only the rectangular classes are ever instantiated directly. The circular classes are used as abstract classes.

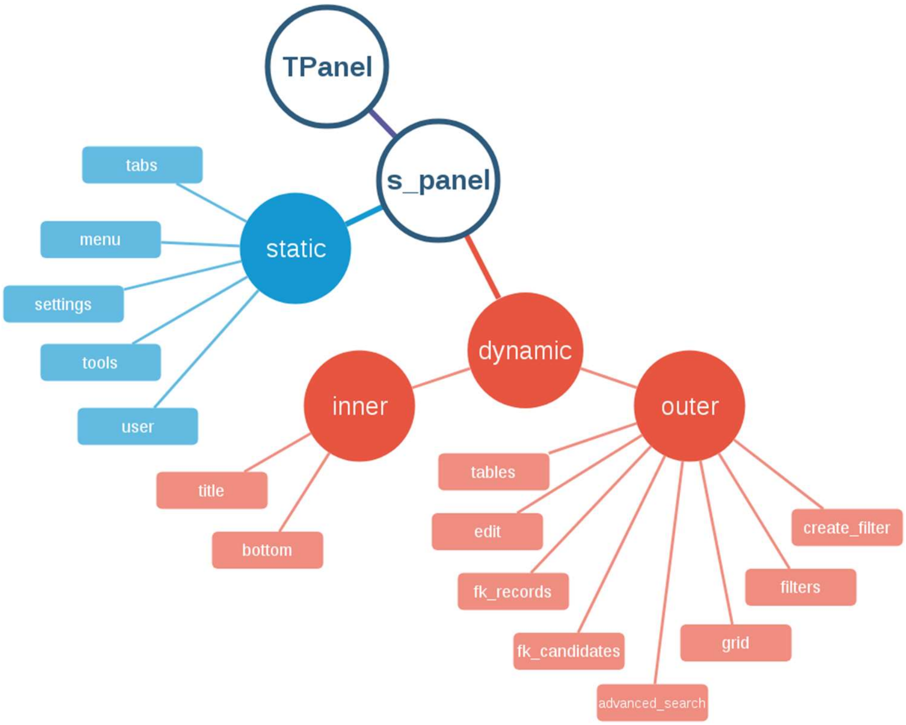


*Figure 5-8: Panel inheritance hierarchy.*

Together the following panels handle the “New” and “Edit” functionality described in 5.1.1, and 5.1.2: *tables*, *edit*, *fk_records*, *fk_candidates*. The requirements for the grid view described in 5.1.3 is covered by *grid*, *filters*, and *create_filter.* The rest of the requirements in chapter 5.1 are covered by the static panels.

Why buttons get the same treatment as panels, at least when discussing classes derived from *TButton*, comes down to a single problem: Referencing the panel on which buttons exist, from any event function. The solution proved to be simple as long as the button object calling the event also had a class member pointing to its parent panel. Any event function may with this design cast the calling *TObject* (ultimate ancestor for all embarcadero classes) into its respective derived button class, and then access the parent panel. The downside to this, is that all event functions need to know exactly which derived button class it is meant for, and near-duplicates of functions may occur. *Figure 5-9* shows an example of an event being fired, involving both buttons, panels and casting as mentioned above.

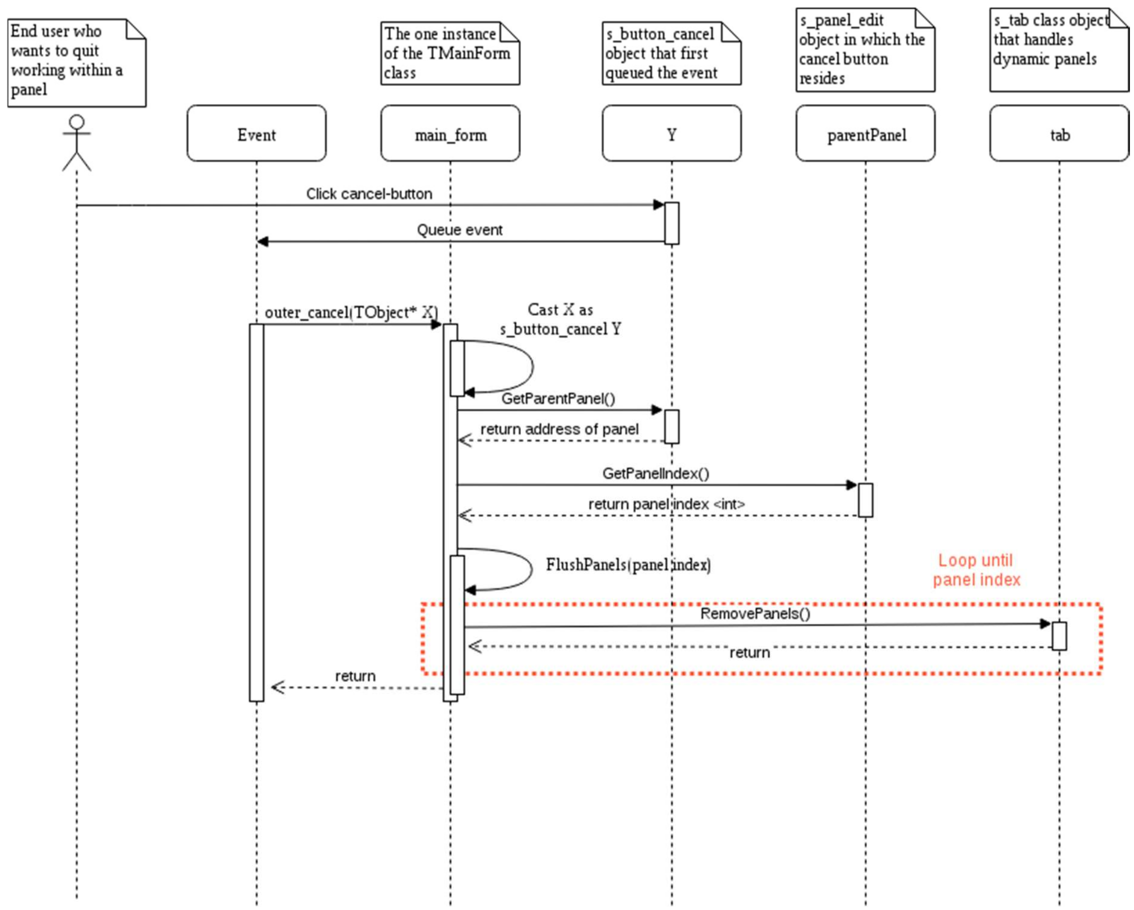

End user who wants to quit working within a panel
The one instance of the TMainForm class
s_button_cancel object that first queued the event
s_panel_edit object in which the cancel button resides
s_tab class object that handles dynamic panels
Event
main_form
Y
parentPanel
tab
Click cancel-button
Queue event
outer_cancel(TObject* X)
Cast X as s_button_cancel Y
GetParentPanel()
return address of panel
GetPanelIndex()
return panel index <int>
FlushPanels(panel index)
Loop until panel index
RemovePanels()
return
return

*Figure 5-9: Sequence diagram for pressing the cancel button.*

Otherwise, the new button classes are branched out in much the same way as panels. Static buttons belong to static panels, and dynamic buttons to dynamic panels etc. This aims to create a similar structure between the two controls, even if it mostly serves the purpose of readability, rather than functional needs. *Figure 5-10* displays the inheritance tree for the custom button classes derived from *TButton*. As with panels, the circular classes are implemented as abstract classes.

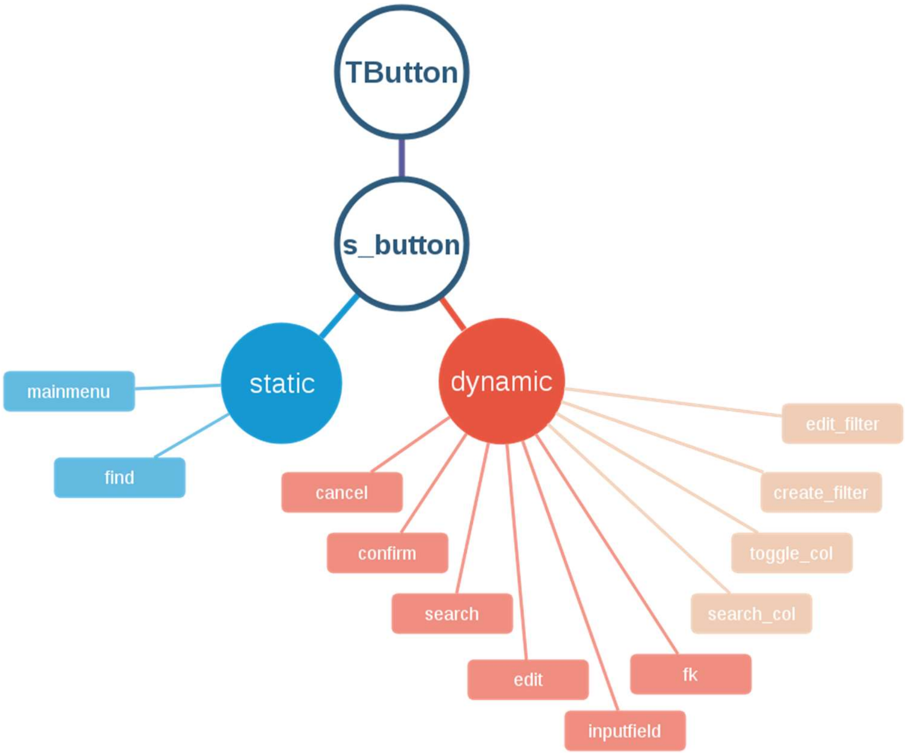


*Figure 5-10: Button inheritance hierarchy.*

Two other concepts handled by *s_gui* are the input boxes displayed in *Figure 5-5*, and tabs (think internet browsers). The input boxes are in this project dubbed *datafields*, hence *s_datafield.cpp*.

*Datafields* are supposed to solve a few issues related to editing a single database record. Because the database is composed of several different data types, which data type is expected from any input control should be intuitive. Upon committing this data to the database, and because the database interface requires data to be passed as *AnsiString*[2], all *datafields* would have to return their input data as *AnsiString* in a controlled way. Therefore, the *datafield* class hierarchy not only handles positioning, size and other visuals, but also a unified way of retrieving, and setting, input data. *Figure 5-11*

[2] *AnsiString is Embarcadero's own string library and data type made for the Delphi language.*

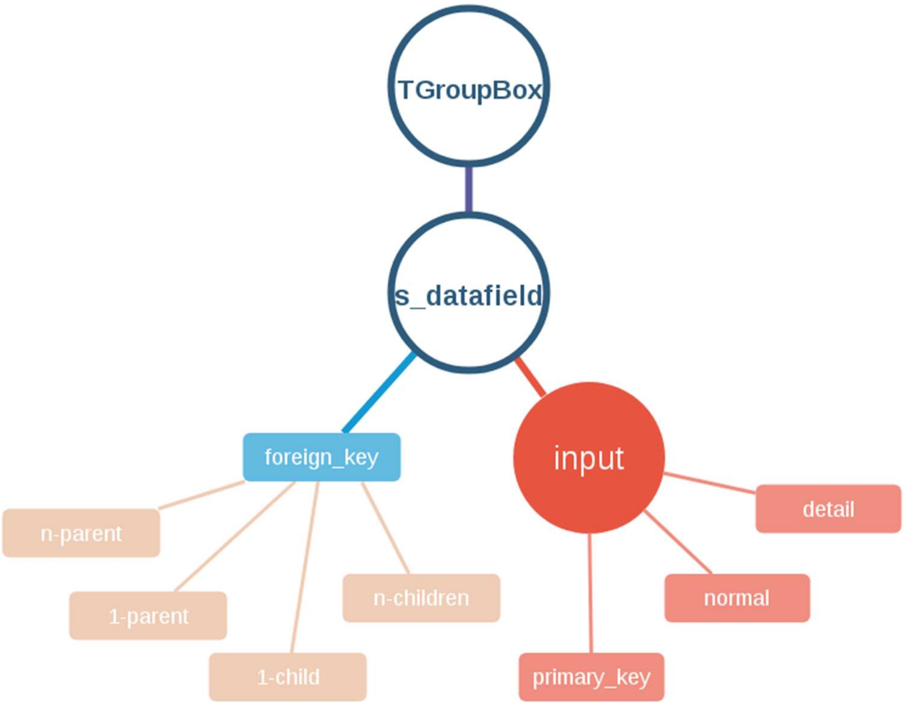
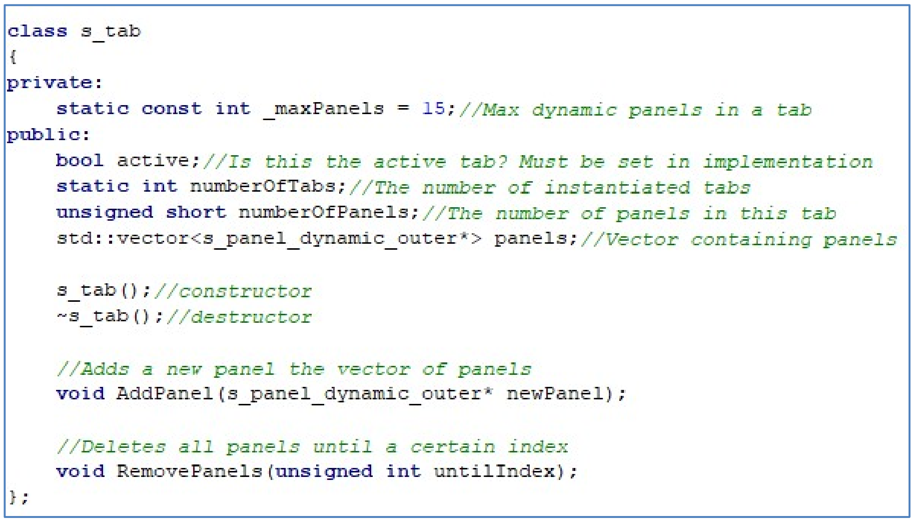


*Figure 5-11: Data field inheritance hierarchy*

Lastly there is the tab class. While a fully functional tab system as the ones implemented in modern browsers would be nice, a simple to use and easy to understand system of handling visibility and memory allocation for dynamic panels is plenty for this project.
The tab class has no base class, and no derived classes. It is invisible in the GUI, but handles panels, and therefore included in the same file. *Figure 5-12* shows the class members of *s_tab*. *Figure 5-13* illustrates how a *s_tab* class is declared in the *mainsource*.



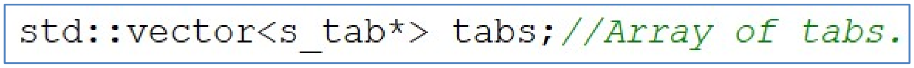
```
class s_tab
{
private:
    static const int _maxPanels = 15;//Max dynamic panels in a tab
public:
    bool active;//Is this the active tab? Must be set in implementation
    static int numberOfTabs;//The number of instantiated tabs
    unsigned short numberOfPanels;//The number of panels in this tab
    std::vector<s_panel_dynamic_outer*> panels;//Vector containing panels

    s_tab();//constructor
    ~s_tab();//destructor

    //Adds a new panel the vector of panels
    void AddPanel(s_panel_dynamic_outer* newPanel);

    //Deletes all panels until a certain index
    void RemovePanels(unsigned int untilIndex);
};
```

*Figure 5-12: Tab class declaration in s_gui.hpp*

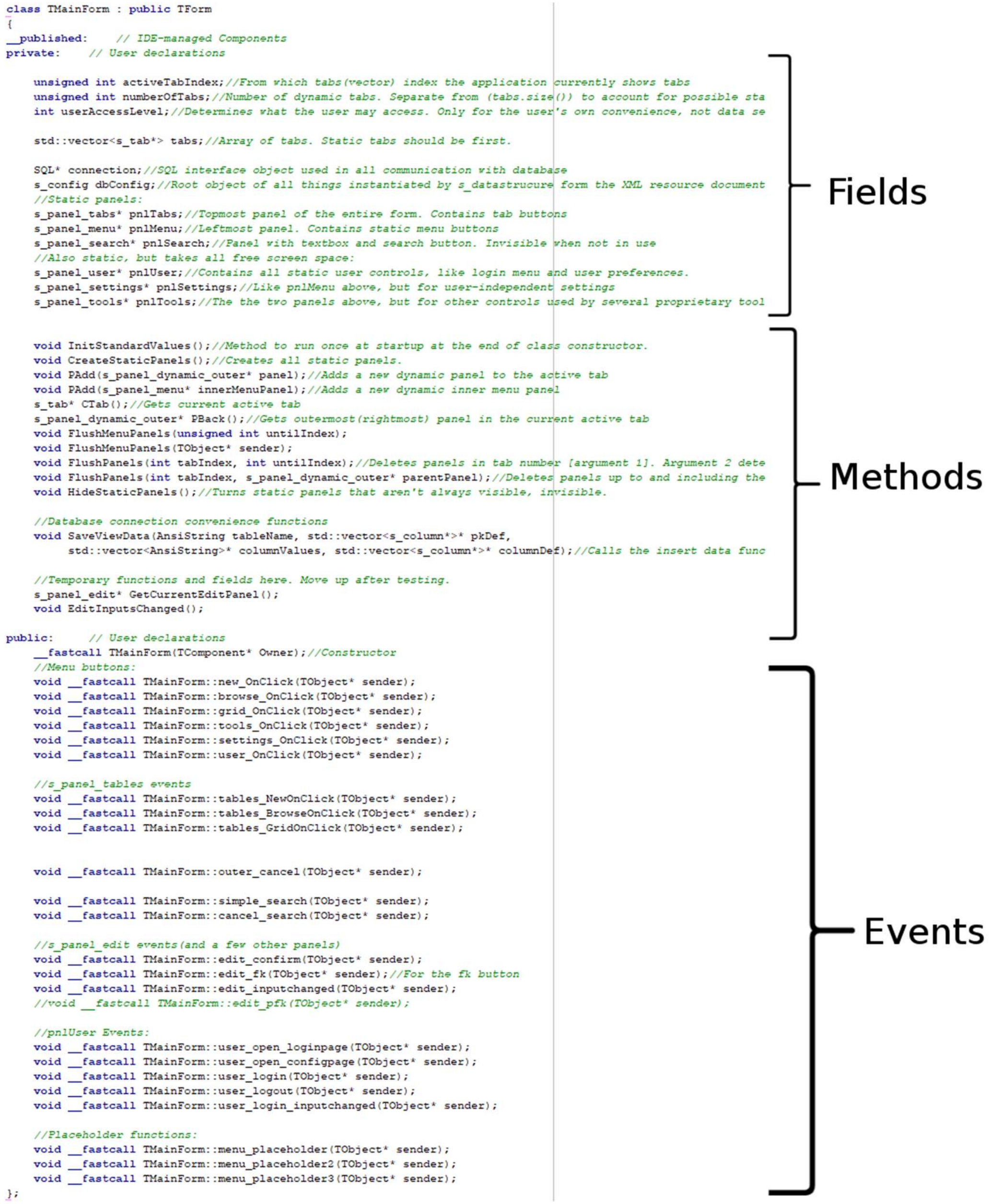


```
std::vector<s_tab*> tabs;//Array of tabs.
```

*Figure 5-13: Vector of tabs as declared in main form class*

### 5.3.3 Main form (mainsource)

The *mainsource* documents ties it all together. It consists of one class: *TMainForm*, which is a Windows form class derived from *TForm*. Here the XML configuration and SQL interface objects get instantiated, as well as all GUI objects, dynamic or static. Events are defined here, as well as methods and functions meant to assist in tying together this particular application. This is the least designed of the four code parts but is no less important to the final product. *Figure 5-14* shows a screenshot of the main header file.



```
class TMainForm : public TForm
{
__published:    // IDE-managed Components
private:    // User declarations

    unsigned int activeTabIndex;//From which tabs(vector) index the application currently shows tabs
    unsigned int numberOfTabs;//Number of dynamic tabs. Separate from (tabs.size()) to account for possible sta
    int userAccessLevel;//Determines what the user may access. Only for the user's own convenience, not data se

    std::vector<s_tab*> tabs;//Array of tabs. Static tabs should be first.

    SQL* connection;//SQL interface object used in all communication with database
    s_config dbConfig;//Root object of all things instantiated by s_datastrucure form the XML resource document
    //Static panels:
    s_panel_tabs* pnlTabs;//Topmost panel of the entire form. Contains tab buttons
    s_panel_menu* pnlMenu;//Leftmost panel. Contains static menu buttons
    s_panel_search* pnlSearch;//Panel with textbox and search button. Invisible when not in use
    //Also static, but takes all free screen space:
    s_panel_user* pnlUser;//Contains all static user controls, like login menu and user preferences.
    s_panel_settings* pnlSettings;//Like pnlMenu above, but for user-independent settings
    s_panel_tools* pnlTools;//The the two panels above, but for other controls used by several proprietary tool


    void InitStandardValues();//Method to run once at startup at the end of class constructor.
    void CreateStaticPanels();//Creates all static panels.
    void PAdd(s_panel_dynamic_outer* panel);//Adds a new dynamic panel to the active tab
    void PAdd(s_panel_menu* innerMenuPanel);//Adds a new dynamic inner menu panel
    s_tab* CTab();//Gets current active tab
    s_panel_dynamic_outer* PBack();//Gets outermost(rightmost) panel in the current active tab
    void FlushMenuPanels(unsigned int untilIndex);
    void FlushMenuPanels(TObject* sender);
    void FlushPanels(int tabIndex, int untilIndex);//Deletes panels in tab number [argument 1]. Argument 2 dete
    void FlushPanels(int tabIndex, s_panel_dynamic_outer* parentPanel);//Deletes panels up to and including the
    void HideStaticPanels();//Turns static panels that aren't always visible, invisible.

    //Database connection convenience functions
    void SaveViewData(AnsiString tableName, std::vector<s_column*>* pkDef,
         std::vector<AnsiString>* columnValues, std::vector<s_column*>* columnDef);//Calls the insert data func

    //Temporary functions and fields here. Move up after testing.
    s_panel_edit* GetCurrentEditPanel();
    void EditInputsChanged();

public:     // User declarations
    __fastcall TMainForm(TComponent* Owner);//Constructor
    //Menu buttons:
    void __fastcall TMainForm::new_OnClick(TObject* sender);
    void __fastcall TMainForm::browse_OnClick(TObject* sender);
    void __fastcall TMainForm::grid_OnClick(TObject* sender);
    void __fastcall TMainForm::tools_OnClick(TObject* sender);
    void __fastcall TMainForm::settings_OnClick(TObject* sender);
    void __fastcall TMainForm::user_OnClick(TObject* sender);

    //s_panel_tables events
    void __fastcall TMainForm::tables_NewOnClick(TObject* sender);
    void __fastcall TMainForm::tables_BrowseOnClick(TObject* sender);
    void __fastcall TMainForm::tables_GridOnClick(TObject* sender);


    void __fastcall TMainForm::outer_cancel(TObject* sender);

    void __fastcall TMainForm::simple_search(TObject* sender);
    void __fastcall TMainForm::cancel_search(TObject* sender);

    //s_panel_edit events(and a few other panels)
    void __fastcall TMainForm::edit_confirm(TObject* sender);
    void __fastcall TMainForm::edit_fk(TObject* sender);//For the fk button
    void __fastcall TMainForm::edit_inputchanged(TObject* sender);
    //void __fastcall TMainForm::edit_pfk(TObject* sender);

    //pnlUser Events:
    void __fastcall TMainForm::user_open_loginpage(TObject* sender);
    void __fastcall TMainForm::user_open_configpage(TObject* sender);
    void __fastcall TMainForm::user_login(TObject* sender);
    void __fastcall TMainForm::user_logout(TObject* sender);
    void __fastcall TMainForm::user_login_inputchanged(TObject* sender);

    //Placeholder functions:
    void __fastcall TMainForm::menu_placeholder(TObject* sender);
    void __fastcall TMainForm::menu_placeholder2(TObject* sender);
    void __fastcall TMainForm::menu_placeholder3(TObject* sender);
};
```

*Figure 5-14: Example structure of the main form class. Note that the complete application needs many more events than displayed here.*

Aside from events and the class constructor, which are public, are all other members private. In the field domain, primitive data types are declared before other types. Below that are methods.

# 5.4 XML configuration file

In order to comply with the overarching goal of making an extensible application, there seemed to loom a particular problem over the entirety of the system. Would changes to the database result in complex changes in the code? If so, the system is hardly extensible at all. The natural response seemed to be a third layer in the application to serve as a configuration for how the application should behave. This behavior is based on two inputs: The database structure (metadata), and customer demands (manual changes). The next few pages describes in full what this means, and how to utilize it.

## 5.4.1 Data sources

This section explains what the configuration layer depends on, and from where data originate. *Figure 5-15* shows an overview of this.

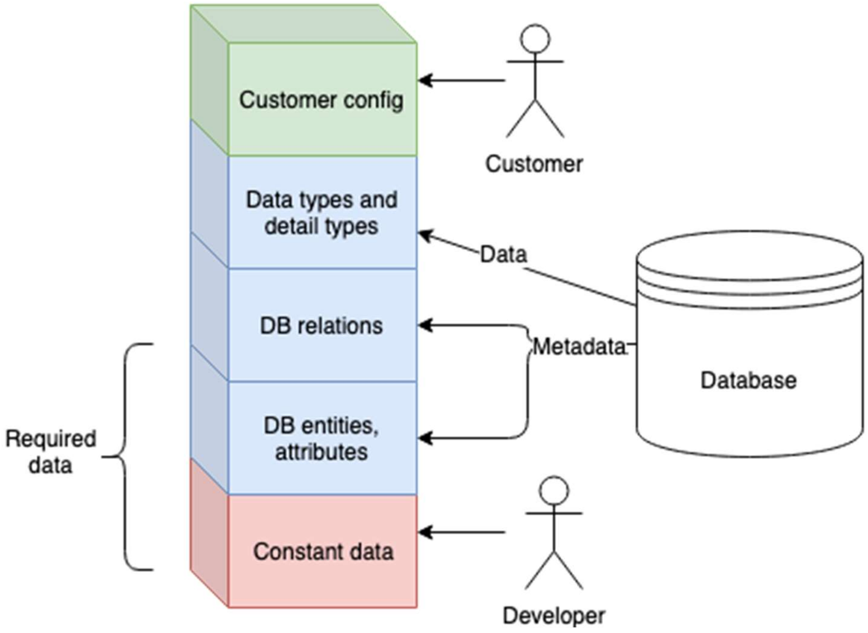


*Figure 5-15: Data origins on the config "stack"*

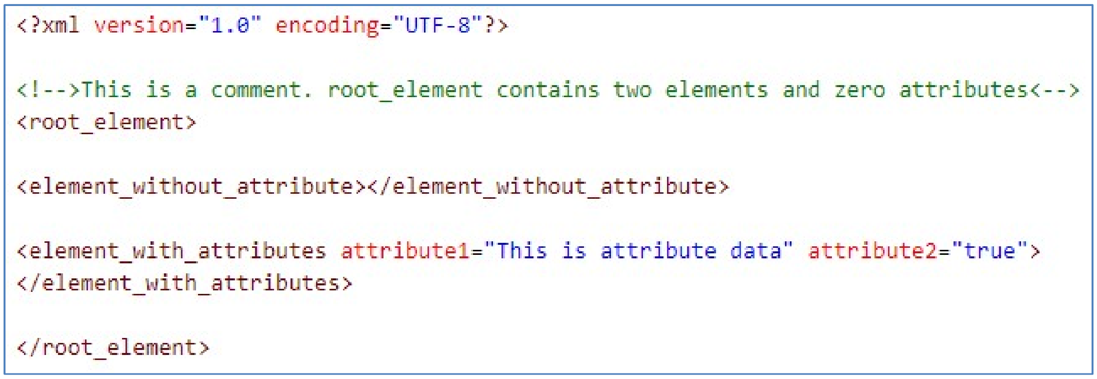
First off, the configuration layer has two absolute requirements outside its own domain: First, GUI input control types must be defined in code. This is because these control types must be hard coded in the XML configuration as well, for when custom data types need to refer to a user interface representation. The second requirement is a valid database. Meaning a database with at least one table, and where all tables have a primary key.

### 5.4.2 Choosing the right technology

The only real requirement for the technology of the configuration file was that it must be easily editable by any normal installation of the Windows operating system. Text files fit this requirement perfectly. However, plain text files can quickly become unmanageable when handling large amounts of structured data. In this case, as the rest of the subchapter describes, the amount of data is relatively large, also rigidly structured. Therefore, some other kind of text-based technology is needed.

XML (Extensible Markup Language) seemed to be the solution after adding the requirement of structuring this data. XML is a simple, well tested markup language that many other technologies are based on (e.g. Scalable Vector Graphics [5]). Markup languages provide annotation for text documents in a platform independent manner. In practice, this means data may more consistently get structured in other forms than one dimensional text.

XML documents are built using elements. Elements may consist of any number of other elements and attributes. Much like HTML, XML uses opening and closing tags to define elements. *Figure 5-16* displays some basic XML.



```
<?xml version="1.0" encoding="UTF-8"?>

<!-->This is a comment. root_element contains two elements and zero attributes<-->
<root_element>

<element_without_attribute></element_without_attribute>

<element_with_attributes attribute1="This is attribute data" attribute2="true">
</element_with_attributes>

</root_element>
```

*Figure 5-16: Basic example XML code.*

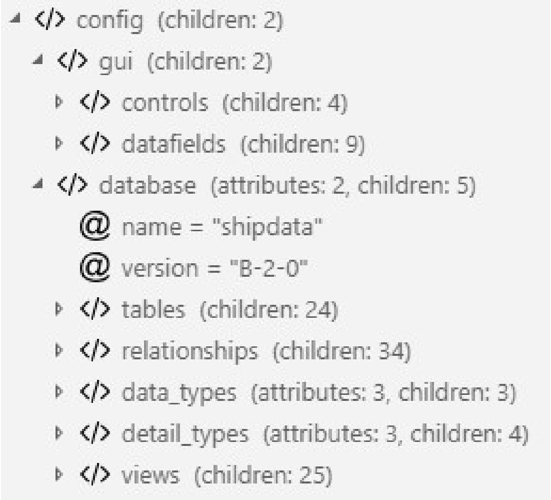
A fundamental choice befalls the users of XML that the specification doesn't take for us: Whether to store data within elements or attributes. This project consistently stores data in attributes, and never directly within element tags. Not for any performance reasons, although the two might have different performance impact, but to strengthen the readability of the raw XML file.

Using the DOM (Document Object Model), an XML document is treated as a tree structure. In the next section the implementation of this tree structure is explained.

### 5.4.3 Document structure

As with other parts of the application, this layer should also be structured in the most natural way feasible. The question then arises as for whom this document should most naturally be structured for? The answer is a middle ground between application usage (tailored for in-memory data. See chapter 5.4.5), database metadata (tables, columns, and relations) and human readability.

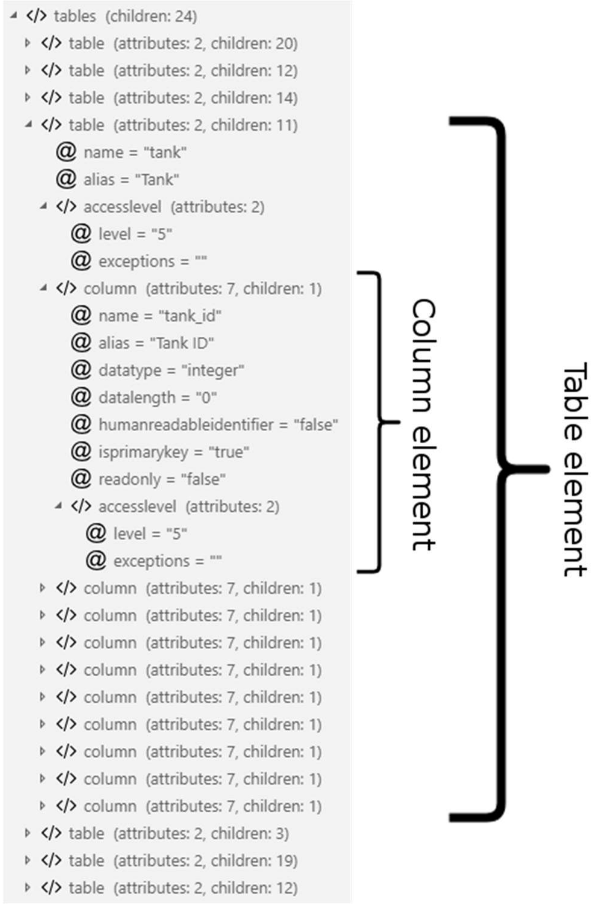
*Figure 5-17* displays the root DOM node (*config*), its two child nodes (*gui* and *database*) with their attributes and children. These nodes (denoted with a closing tag) are all XML elements represented as DOM nodes.



```
◢ </> config  (children: 2)
  ◢ </> gui  (children: 2)
    ▷ </> controls  (children: 4)
    ▷ </> datafields  (children: 9)
  ◢ </> database  (attributes: 2, children: 5)
        @ name = "shipdata"
        @ version = "B-2-0"
    ▷ </> tables  (children: 24)
    ▷ </> relationships  (children: 34)
    ▷ </> data_types  (attributes: 3, children: 3)
    ▷ </> detail_types  (attributes: 3, children: 4)
    ▷ </> views  (children: 25)
```

*Figure 5-17: Root node in the DOM tree with children and grandchildren*

The figure above shows the nodes closest to the root but there are many more nodes further out in the branches. *Figure 5-18* shows the nodes and attributes of the *tank* table, and one of its columns (*tank_id*). This node is found within the *tables* node.



```
◢ </> tables  (children: 24)
  ▷ </> table  (attributes: 2, children: 20)
  ▷ </> table  (attributes: 2, children: 12)
  ▷ </> table  (attributes: 2, children: 14)
  ◢ </> table  (attributes: 2, children: 11)
        @ name = "tank"
        @ alias = "Tank"
      ◢ </> accesslevel  (attributes: 2)
          @ level = "5"
          @ exceptions = ""
      ◢ </> column  (attributes: 7, children: 1)
          @ name = "tank_id"
          @ alias = "Tank ID"
          @ datatype = "integer"
          @ datalength = "0"
          @ humanreadableidentifier = "false"
          @ isprimarykey = "true"
          @ readonly = "false"
        ◢ </> accesslevel  (attributes: 2)
            @ level = "5"
            @ exceptions = ""
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
      ▷ </> column  (attributes: 7, children: 1)
  ▷ </> table  (attributes: 2, children: 3)
  ▷ </> table  (attributes: 2, children: 19)
  ▷ </> table  (attributes: 2, children: 12)
```

*Figure 5-18: Inspecting a table in the DOM tree.*

A DTD (Document Type Definition) has been developed for this document. See Appendix J for the full specification.

This document defines how the DOM tree should be structured using XML. The DTD may even assist XML parsers in determining the validity of documents, as well as providing default values and other functionality. Within this project, the DTD isn't actually enforced, but instead serves as documentation, and a syntax for end users when manually editing the config document. The reason this isn't enforced comes down to which XML parser is used, as described in the following section.

### 5.4.4 XML parser

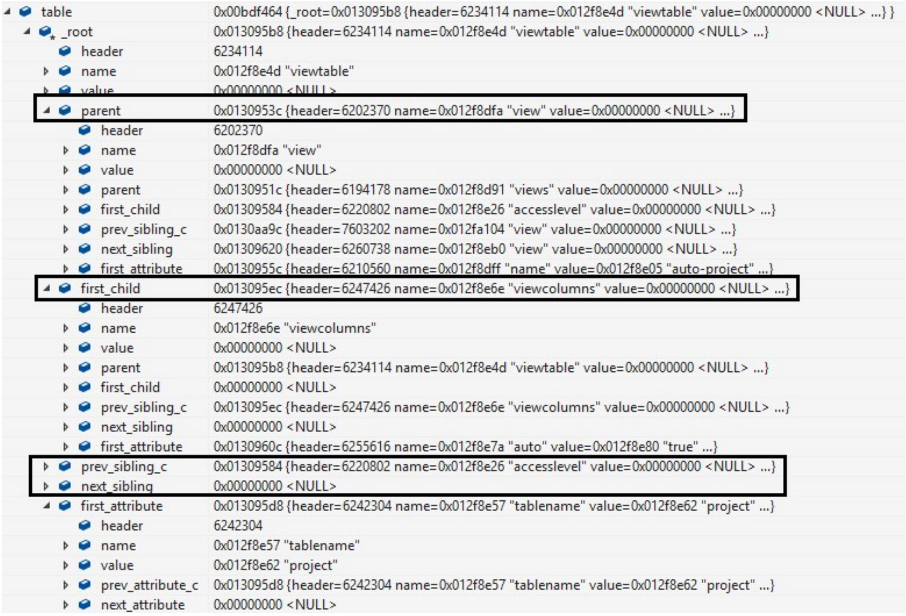
An XML parser is a piece of software that interprets XML and makes the DOM tree readable in memory. In this project, the C++ application needed some convenient way of extracting information from the configuration file.

There are many XML parsers available serving different use cases. For use in this project it would have to be in C++, and open source. Other than that, it would be beneficial if it was simple to use. Many parsers like this exist, but none of them are fully XML + DOM compliant. Meaning, in this specific case, that the DTD document described earlier wouldn't be considered when parsing. A fair trade-off for an easy-to-use parser.

After testing two different parsers, *pugixml* seemed slightly easier to use and was added to the project dependencies. It should be noted that the tree structure *pugixml* uses, is DOM-like only, but this report will continue to use "DOM tree".
Implementing *Xerxes,* or any other complicated, and fully DOM compliant parser, would have resulted in a more fail-safe system from the end user's standpoint, but the idea was put down to stay within the time constraints of the project. [6][7]

After calling the parsing function upon launching the application, the DOM tree lies available in memory to read and write to. However, as briefly explained in chapter 5.4.3, the document structure isn't fully tailored for usage by the application. Therefore, this data will now get transformed for more optimized computation.

*Figure 5-19* displays an example of how one such pugixml DOM node looks like in memory. *parent* is a pointer to the parent element, *first_child* points to the first child node, *prev*- and *next_sibling* point to sibling elements. If elements like these don't exist, they are simply replaced by NULL pointers. First_attribute however, is a pointer to an attribute, which contains fewer data fields.



```
◢ table              0x00bdf464 {_root=0x013095b8 {header=6234114 name=0x012f8e4d "viewtable" value=0x00000000 <NULL> ...} }
  ◢ _root            0x013095b8 {header=6234114 name=0x012f8e4d "viewtable" value=0x00000000 <NULL> ...}
      header         6234114
    ▷ name           0x012f8e4d "viewtable"
    ▷ value          0x00000000 <NULL>
    ◢ parent         0x0130953c {header=6202370 name=0x012f8dfa "view" value=0x00000000 <NULL> ...}
        header       6202370
      ▷ name         0x012f8dfa "view"
      ▷ value        0x00000000 <NULL>
      ▷ parent       0x0130951c {header=6194178 name=0x012f8d91 "views" value=0x00000000 <NULL> ...}
      ▷ first_child  0x01309584 {header=6220802 name=0x012f8e26 "accesslevel" value=0x00000000 <NULL> ...}
      ▷ prev_sibling_c 0x0130aa9c {header=7603202 name=0x012fa104 "view" value=0x00000000 <NULL> ...}
      ▷ next_sibling 0x01309620 {header=6260738 name=0x012f8eb0 "view" value=0x00000000 <NULL> ...}
      ▷ first_attribute 0x0130955c {header=6210560 name=0x012f8dff "name" value=0x012f8e05 "auto-project" ...}
    ◢ first_child    0x013095ec {header=6247426 name=0x012f8e6e "viewcolumns" value=0x00000000 <NULL> ...}
        header       6247426
      ▷ name         0x012f8e6e "viewcolumns"
      ▷ value        0x00000000 <NULL>
      ▷ parent       0x013095b8 {header=6234114 name=0x012f8e4d "viewtable" value=0x00000000 <NULL> ...}
      ▷ first_child  0x00000000 <NULL>
      ▷ prev_sibling_c 0x013095ec {header=6247426 name=0x012f8e6e "viewcolumns" value=0x00000000 <NULL> ...}
      ▷ next_sibling 0x00000000 <NULL>
      ▷ first_attribute 0x0130960c {header=6255616 name=0x012f8e7a "auto" value=0x012f8e80 "true" ...}
    ▷ prev_sibling_c 0x01309584 {header=6220802 name=0x012f8e26 "accesslevel" value=0x00000000 <NULL> ...}
    ▷ next_sibling   0x00000000 <NULL>
    ◢ first_attribute 0x013095d8 {header=6242304 name=0x012f8e57 "tablename" value=0x012f8e62 "project" ...}
        header       6242304
      ▷ name         0x012f8e57 "tablename"
      ▷ value        0x012f8e62 "project"
      ▷ prev_attribute_c 0x013095d8 {header=6242304 name=0x012f8e57 "tablename" value=0x012f8e62 "project" ...}
      ▷ next_attribute 0x00000000 <NULL>
```

*Figure 5-19: A pugicxml node highlighting related nodes.*

### 5.4.5 Instantiated data structure (s_datastructure)

To transform data from the DOM tree correctly to a new data structure, a set order needed to be defined. *Figure 5-20* shows the instantiation dependencies from the bottom to top.

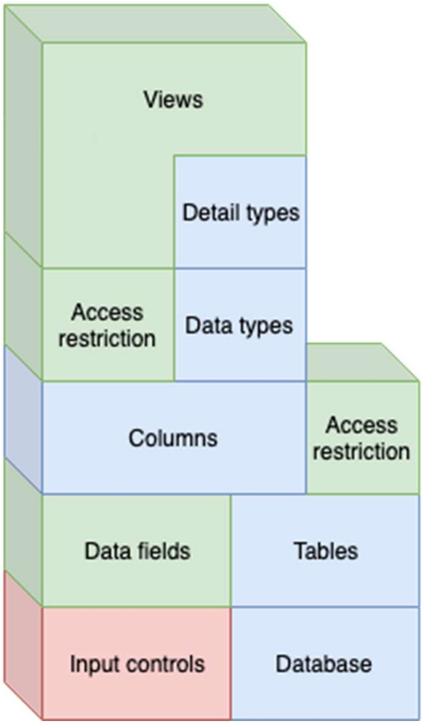
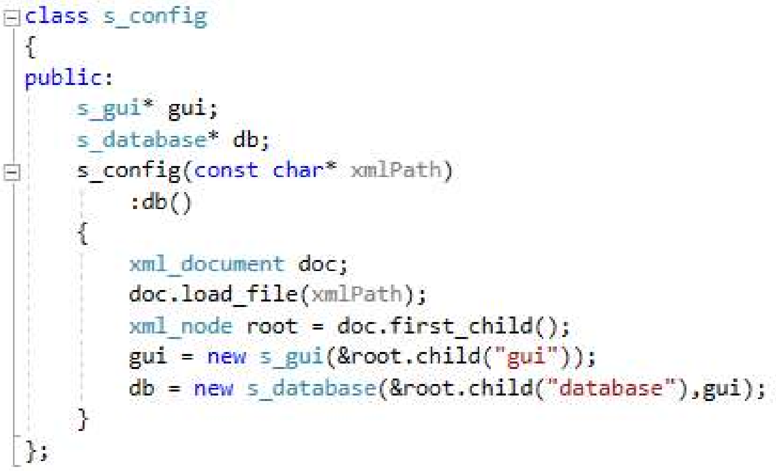


*Figure 5-20: Dependencies in the configuration*

*Figure 5-21* shows the container class for all configuration data. It has only two fields: *gui* and *db*, which are pointers to instantiated data from the *gui* and *database* element, respectively as the two child elements seen in *Figure 5-17*. This class, with the constructor shown in fig, is how *s_datastructure* is implemented in *mainsource*, by passing the file path to the XML document.



```
class s_config
{
public:
    s_gui* gui;
    s_database* db;
    s_config(const char* xmlPath)
        :db()
    {
        xml_document doc;
        doc.load_file(xmlPath);
        xml_node root = doc.first_child();
        gui = new s_gui(&root.child("gui"));
        db = new s_database(&root.child("database"),gui);
    }
};
```

*Figure 5-21: The container class for all of s_datastructure*

First off to get instantiated are all the static data, which is strictly defined by developers of the application. There aren't a lot, but nonetheless important as other layers may depend on them. Afterwards comes the database structure (metadata).

Here access levels come into play. Tables and columns all have a set access restriction, but it's impossible for the program to know what this should be and must therefore be provided manually if full access for all users isn't desirable.

Now comes the optional steps. First are data which are read from selected tables in the database. In this project *data type*s and *detail type*s (see chapter 3) are included so that they may be configured with more variables than what is saved in the database.

Lastly customer configuration is instantiated. After all this layer is what is the most dependent on existing data. In a few cases however, like table and column access levels, customer configuration data is instantiated earlier, but these cases always provide valid placeholder values.

To stay within project scope, the parsing process has not gotten any exception handling implemented to guide the end user in debugging xml code. Instead the parsing function returns early and leaves the DOM tree empty.

One of the weaknesses with using the *pugixml* data structure is that it only stores strings. Iterating through a number of tables and columns looking for a specific match, after casting them to the correct data type every time, would put a lot of delay in the system. The cost of using a little more memory seems insignificant when the new data structure can find all relations to a table in the manner of a few pointer operations. Compare that to using the DOM tree directly, where the list of all relations would have to be iterated through, compared, and cast correctly for every operation the application would use it for.

# 5.5 SQL interface

Besides connecting the application to the server, there also needs to be some sort of API that can be utilized in code to properly execute SQL queries. This subchapter gives some insight into the higher-level logic present in the code, and ideas behind the entire interface.

## 5.5.1 Interface foundations

On handling data exchange between application and server within the constraints of the task requirements, two basic solutions emerged. Both solutions would need to take an arbitrary amount of data as an argument, and metadata on where that information is headed, before somehow executing the correct SQL query.

Solution.1: In many software stacks this could be achieved by creating parametrized functions server-side (in the SQL language itself), and then call those functions through a simple abstraction layer in code.

Solution.2: *PgDac* a precompiled library developed by Devart, handles PostgreSQL database-functionality. Instead of coding anything server-side are all functions exclusively created application-side and SQL queries are built and executed dynamically.

Initially, solution 1 was put into development. However, upon encountering issues passing table names as arguments, this idea was scrapped in favor of solution 2.

## 5.5.2 PgDac intro

PgDAC is a library for RAD Studio used for PostgreSQL connectivity developed by Devart. It encapsulates many components used for data transaction between an application and a PostgreSQL server. There is very little documentation about advantages and disadvantages between these. Therefore, three of these objects were tested for transactions between the application and database, *Query, Stored Procedure* and *Table*.

*Query* directly uses SQL queries and will return a cursor of what is selected, or a cursor may be inserted or edited with a query. It was first thought to be unsafe for SQL-injection but should be safe by using parameterized queries.[8]

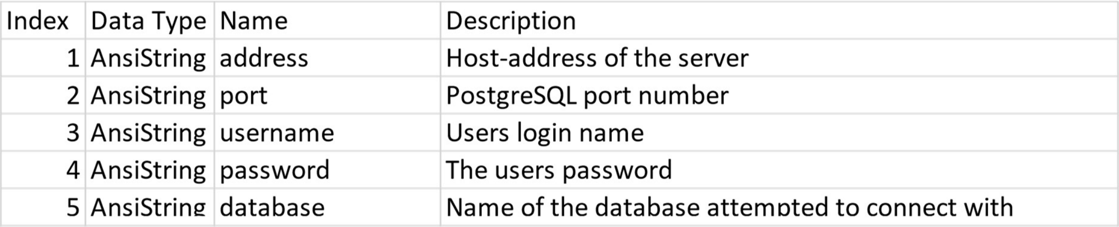
*Stored procedure* requires functions to be pre-defined, there were attempts to create a dynamic procedure, which could take table names as parameter. This proved hard, but again parameterizing SQL code may have solved the issue. It was however discovered at a time when development was started using *Table.* [9]

*Table* reflects a database table and only requires the program to know which table and column it wants to select, insert, edit or delete to or from. It appeared to be the most dynamic solution and was chosen for this reason.

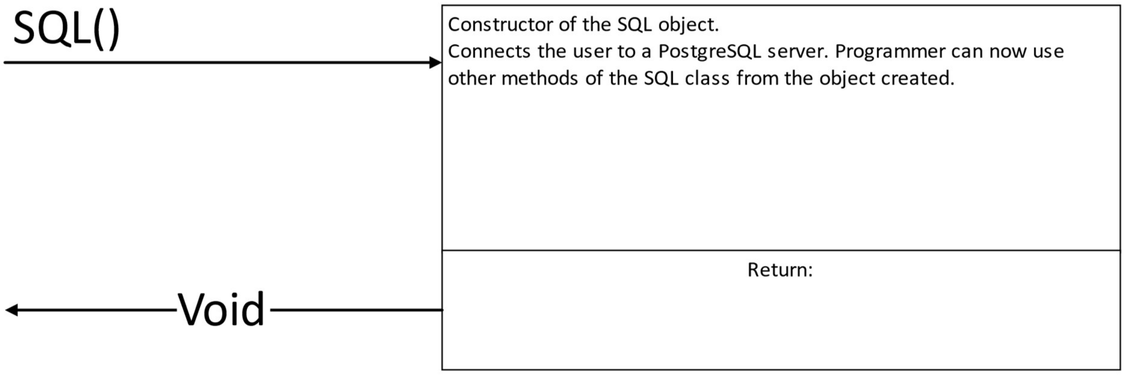
## 5.5.3 Usage/high level logic

The interface consists if a single class that needs to be instantiated by connecting to a database. Below are the constructor and destructor described in respectively *Figure 5-22* and

*Figure 5-23*. These figures list the parameters in the top and describe both what the function does and what it returns below.

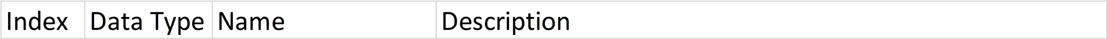


| Index | Data Type | Name | Description |
|---|---|---|---|
| 1 | AnsiString | address | Host-address of the server |
| 2 | AnsiString | port | PostgreSQL port number |
| 3 | AnsiString | username | Users login name |
| 4 | AnsiString | password | The users password |
| 5 | AnsiString | database | Name of the database attempted to connect with |

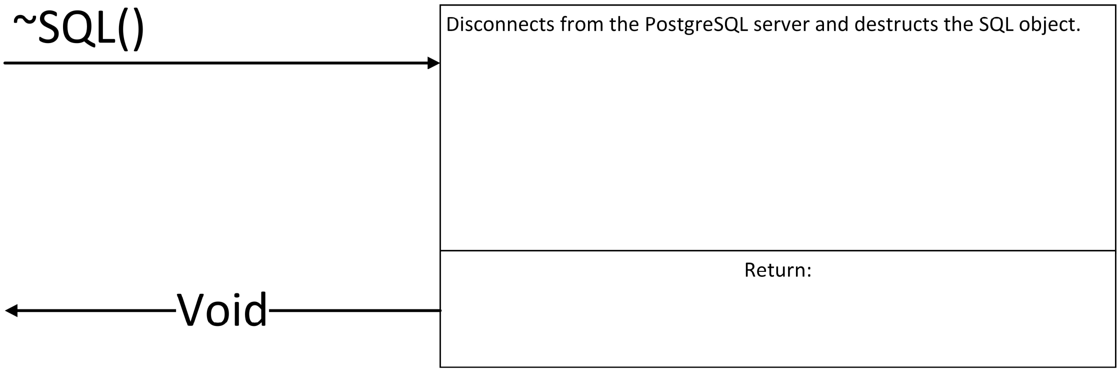


Constructor of the SQL object.
Connects the user to a PostgreSQL server. Programmer can now use other methods of the SQL class from the object created.

*Figure 5-22 SQL constructor*



| Index | Data Type | Name | Description |
|---|---|---|---|



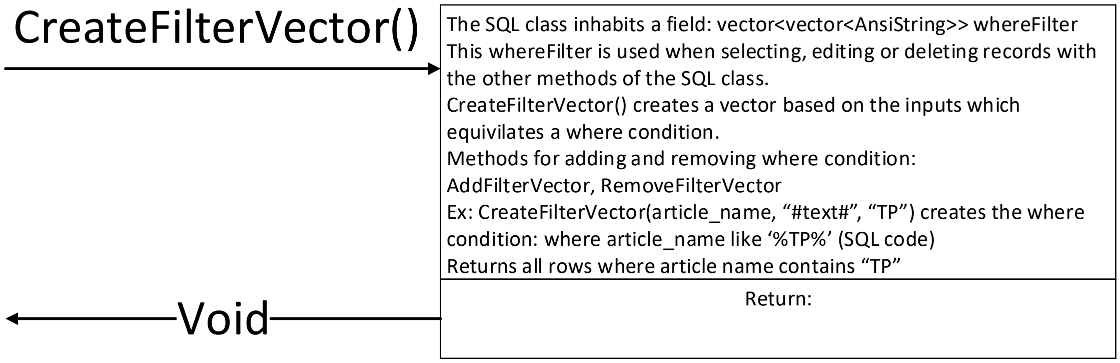
Disconnects from the PostgreSQL server and destructs the SQL object.

*Figure 5-23 SQL destructor*

This next section describes individually the most important methods of the SQL interface. The methods are all that is needed for an interface between an application and the PostgreSQL database. The following methods are illustrated below:

- CreateFilterVector in *Figure 5-24*, creating where conditions
- SelectTableValues in *Figure 5-25*, viewing a table
- InsertTableValues in *Figure 5-26*, inserting or editing a table
- InsertLogData in *Figure 5-27*, inserting or updating log data
- GetDBGrid in *Figure 5-28*, creating a grid view for a table
- DeleteRow in *Figure 5-29*, deleting an entry

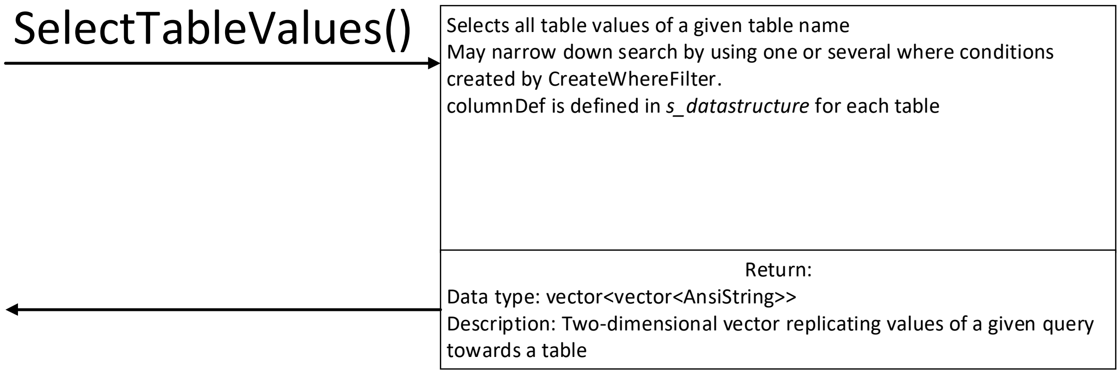
| Index | Data Type | Name | Description |
|---|---|---|---|
| 1 | s_column | column | Describes a given column's data type and name |
| 2 | AnsiString | operation | Operators for SQL where condition |
| 3 | AnsiString | columnValue | Comparing value of where condition |



The SQL class inhabits a field: vector<vector<AnsiString>> whereFilter
This whereFilter is used when selecting, editing or deleting records with the other methods of the SQL class.
CreateFilterVector() creates a vector based on the inputs which equivilates a where condition.
Methods for adding and removing where condition: AddFilterVector, RemoveFilterVector
Ex: CreateFilterVector(article_name, "#text#", "TP") creates the where condition: where article_name like '%TP%' (SQL code)
Returns all rows where article name contains "TP"

*Figure 5-24 CreateFilterVector method*

| Data Type | Name | Description | |
|---|---|---|---|
| AnsiString | tableName | Name of the table to be selected | |
| vector<s_column> | columnDef | Describes the columns of a given table | |

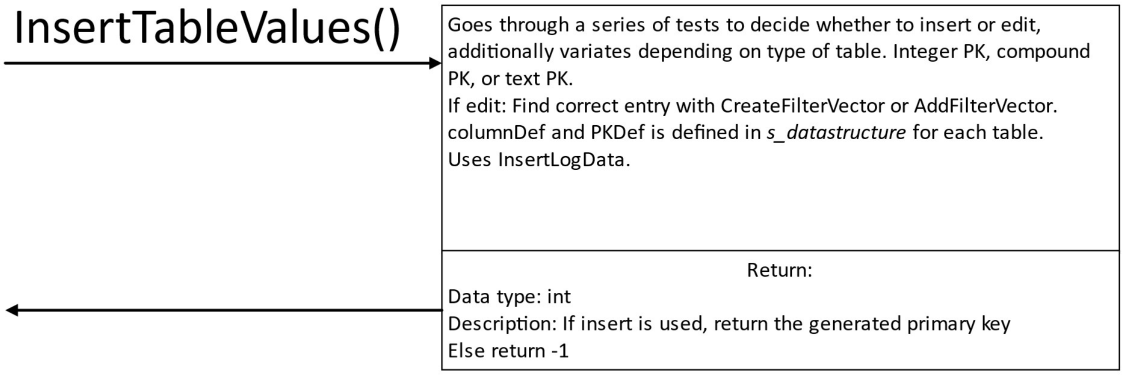


Selects all table values of a given table name
May narrow down search by using one or several where conditions created by CreateWhereFilter.
columnDef is defined in *s_datastructure* for each table

Data type: vector<vector<AnsiString>>
Description: Two-dimensional vector replicating values of a given query towards a table

*Figure 5-25 SelectTableValues method*

| Index | Data Type | Name | Description |
|---|---|---|---|
| 1 | AnsiString | tableName | Name of the table to be selected |
| 2 | vector<s_column> | PKDef | Describes the primary keys of a table |
| 3 | vector<AnsiString> | columnValue | Holds all values to be inserted or edited |
| 4 | vector<s_column> | columnDef | Describes the columns of a given table |

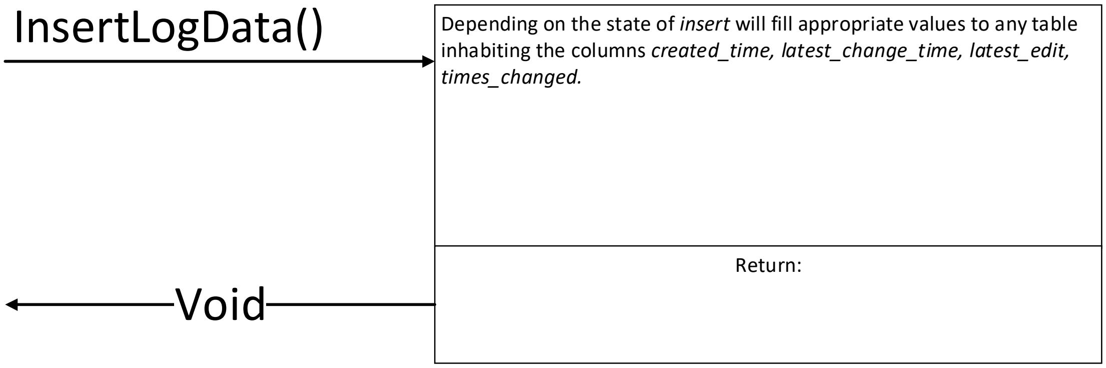


Goes through a series of tests to decide whether to insert or edit, additionally variates depending on type of table. Integer PK, compound PK, or text PK.
If edit: Find correct entry with CreateFilterVector or AddFilterVector.
columnDef and PKDef is defined in *s_datastructure* for each table.
Uses InsertLogData.

Data type: int
Description: If insert is used, return the generated primary key
Else return -1

*Figure 5-26 InsertTableValues method*

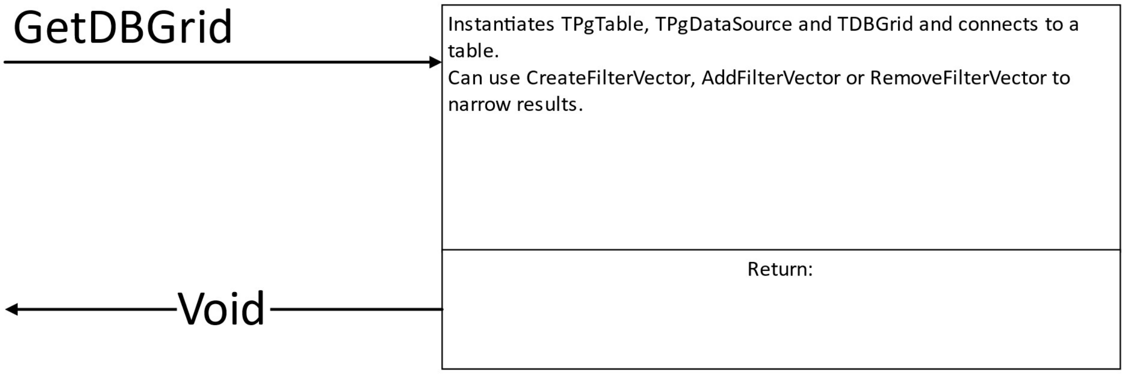
| Index | Data Type | Name | Description |
|---|---|---|---|
| 1 | TPgTable | tbl | TPgTable used in InsertTableValues |
| 2 | Boolean | insert | Describes whether to insert or edit |



Depending on the state of *insert* will fill appropriate values to any table inhabiting the columns *created_time, latest_change_time, latest_edit, times_changed.*

*Figure 5-27 InsertLogData method*

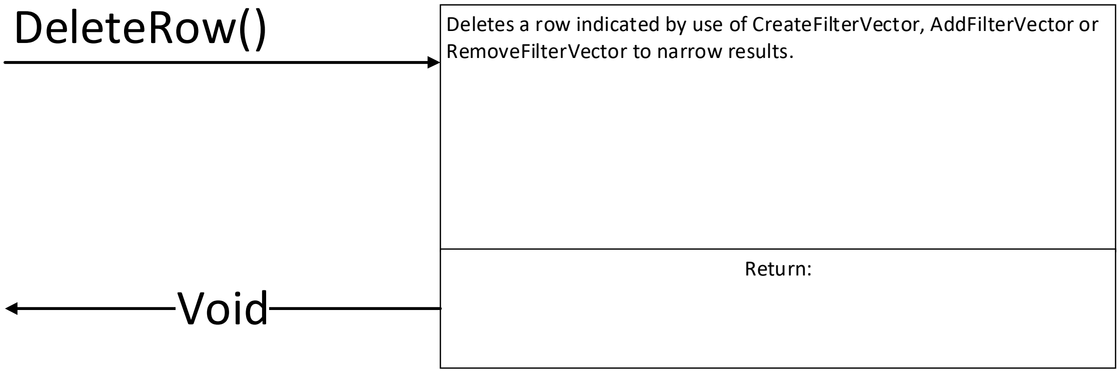
| Index | Data Type | Name | Description |
|---|---|---|---|
| 1 | AnsiString | tablename | A postgreSQL table name |
| 2 | TWinControl | parent | parent decides which object the grid will be part of |
| 3 | TPgTable | tbl | Non-instantiated TPgTable |
| 4 | TPgDataSource | ds | Non-instantiated TPgDataSource |
| 5 | TDBGrid | grid | Non-instantiated TDBGrid |



Instantiates TPgTable, TPgDataSource and TDBGrid and connects to a table.
Can use CreateFilterVector, AddFilterVector or RemoveFilterVector to narrow results.

*Figure 5-28 GetDBGrid method*

| Index | Data Type | Name | Description |
|---|---|---|---|
| 1 | AnsiString | tablename | A postgreSQL table name |



Deletes a row indicated by use of CreateFilterVector, AddFilterVector or RemoveFilterVector to narrow results.

*Figure 5-29 DeleteRow method*

### 5.5.4 Underlying algorithms

There are mainly three algorithms worth mentioning. These are the algorithms for *CreateFilterVector*, *InsertTableValues* and *SelectTableValues*. *InsertTableValues* uses the same principle as *SelectTableValues* in the *Insert values to dataset process* in *Figure 5-31* and will only be explained in the *SelectTableValues*.

*CreateFilterVector* creates a vector for a where-condition in SQL code. Here the algorithm must create varying where-conditions based on the data type of the column it tests against. *Table 5-1* shows respectively how operators are transformed and values formatted. Note that all variables are text strings.

*Table 5-1 Input vs output in CreateFilterVector*

| **Inputs** | | | | **Output vector** | |
|---|---|---|---|---|---|
| Data type | operations | columnValue | | operator | columnValue |
| dtInteger | <,>,= | a | | <,>,= | a |
| dtReal | <,>,= | a | | <,>,= | a |
| dtBool | = | a | | = | a |
| dtTimeStamp | <,>,= | a | | ::date + <,>,= | date + a |
| dtDate | <,>,= | a | | <,>,= | a |
| dtText | text#, #text, #text#, = | a | | like,= | a%, %a, %a%, = |

*SelectTableValues* uses one or more where-conditions created by the previously mentioned method. Then it starts to iterate through all columns in a given table, checking that data types match the column that is attempted to be read. The information flow is shown in *Figure 5-30*.

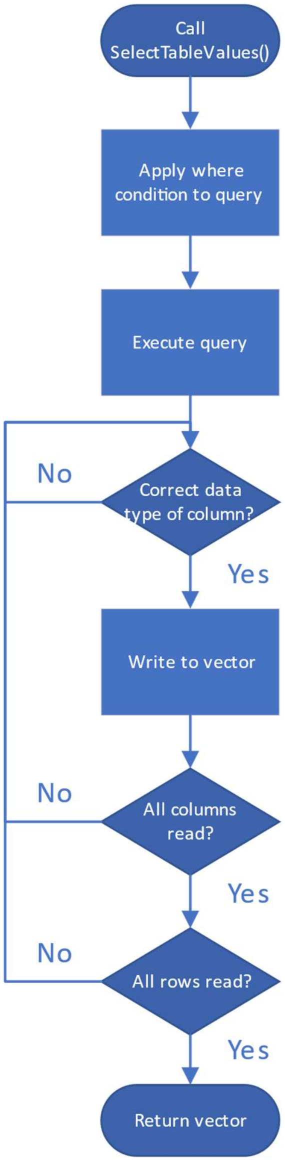


*Figure 5-30 Flowchart for SelectTableValues*

The *InsertTableValues* method is used both for new entries and for editing records. In *Figure 5-31* the method chooses between three paths which will process the primary keys differently and then converge when primary key processing is complete. The *Insert values to dataset* process uses the same logic as seen in the previous figure regarding data types and columns. However, the method *InserTableValues* reads from the input vector instead of writing to a vector.

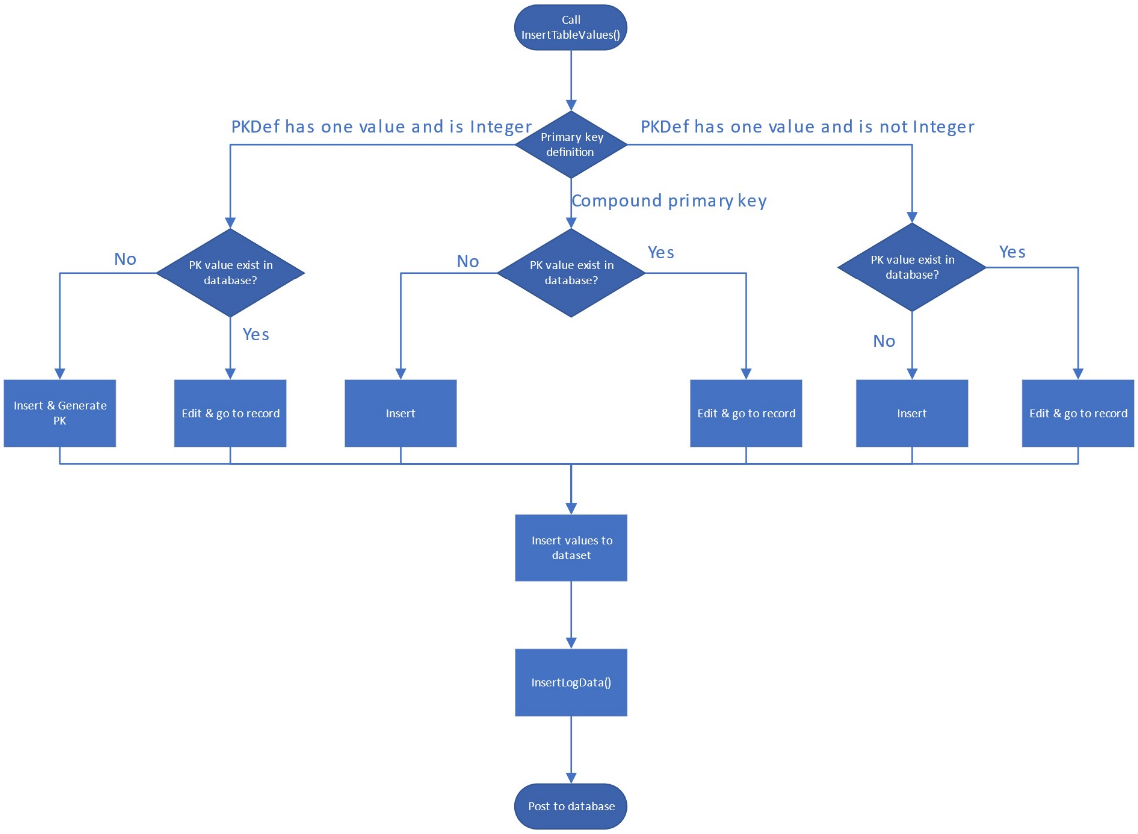


*Figure 5-31 Flowchart of InsertTableValues method*

# 6 Status of prototypes and solutions

This chapter gives the reader an overview of which solutions that has been implemented and status of prototypes developed.

## 6.1 Database

The database solution was implemented to Scanjet’s central server rather early and updated with changes continuously during the project. At the end of this project, a final version was installed on the server due to attribute misplacements in tables during updates.

Today, the database will allow foreign key columns to contain null values. If desired, this can be changed by not accepting null in the relationships between tables.

To access the database, user groups are implemented on the server as described in chapter 3.1.2. This enables Scanjet to add new users and connect them to a user group which handles restrictions. Procedures for login authentication (with a return value representing access level) and adding users has been created for the database.

## 6.2 Migration – Solutions

The general approach to the migration is shown in *Figure 6-1*. Data must be exported from the Clarion database and imported to Excel. Most data transformation is executed in Excel. Where required, a Python script is used for advanced transformations. Data is then inserted to Excel again and uploaded to the new database with help of Devart for Excel.

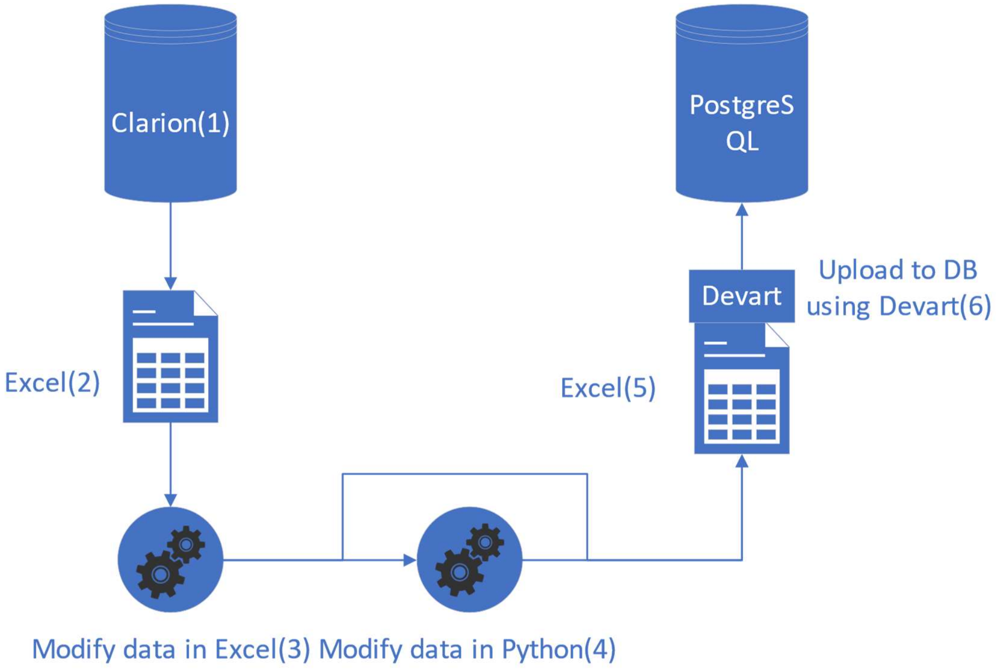


*Figure 6-1 Overview of data migration from Clarion to PostgreSQL*

The order of tables migrated requires some attention, as there are many foreign keys which should or must be attended to before a table is inserted. An overview of the order can be seen in *Figure 6-2*. The order starts at the top with tables which does not have foreign keys and moves downward to tables which uses primary keys of the filled-out tables.

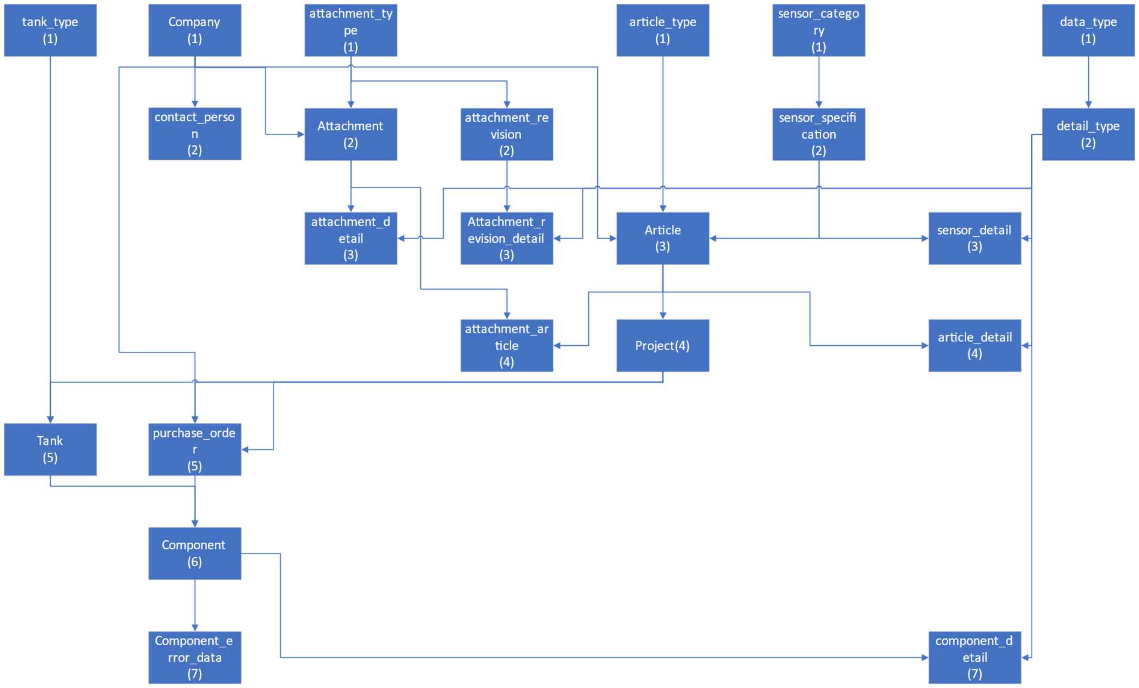


*Figure 6-2 Order of data migration*

### 6.2.1 SoftVelocity Database Scanner

Before exporting data, formatting in SoftVelocity Database Scanner is required. Hidden fields must be selected to retrieve all wanted data and some must be hidden to avoid data corruption. The column *Dallasid* must be reformatted to show all digits and several columns in *sensordata*. These processes, where they apply and how to export data can be read in Appendix G Chapter 1.1.

### 6.2.2 New data

Not all data required for the new system to operate exist. Some will have to be created, some must be acquired from Navision, Scanjet's ERP system. This data is stored in an Excel book Appendix H and covers all tables marked with 1 in *Figure 6-2* and additionally *detail_type, sensor_specification* and *article.*

### 6.2.3 Python transformation

Some tables in the new database require significant changes in the data structure, this applies to the detail tables shown in *Figure 6-3*.

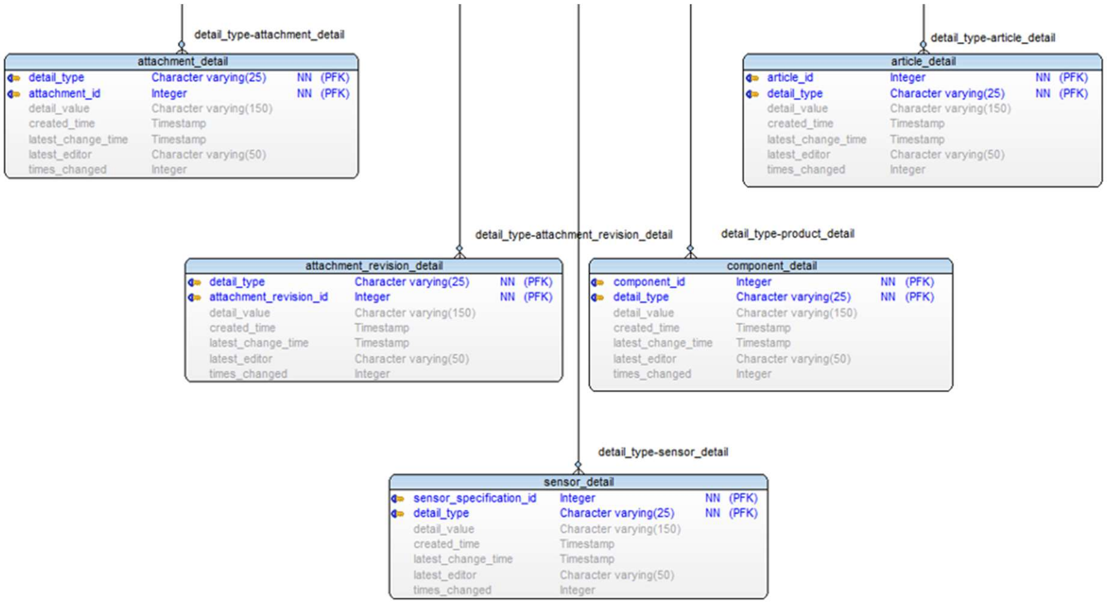
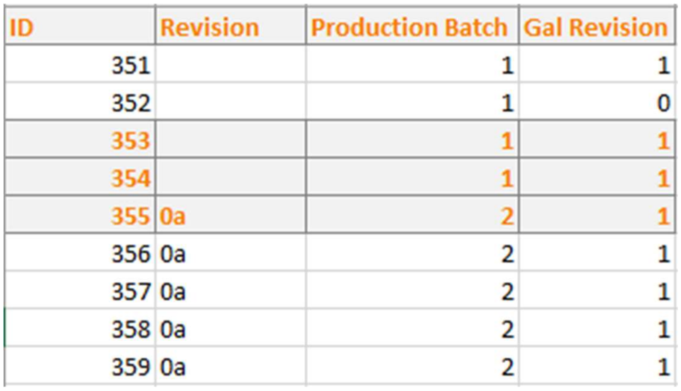

detail_type-attachment_detail
attachment_detail
detail_type Character varying(25) NN (PFK)
attachment_id Integer NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
detail_type-article_detail
article_detail
article_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
detail_type-attachment_revision_detail
detail_type-product_detail
attachment_revision_detail
detail_type Character varying(25) NN (PFK)
attachment_revision_id Integer NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
component_detail
component_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
detail_type-sensor_detail
sensor_detail
sensor_specification_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer

*Figure 6-3 PostgreSQL tables that are more easily transformed outside Excel*

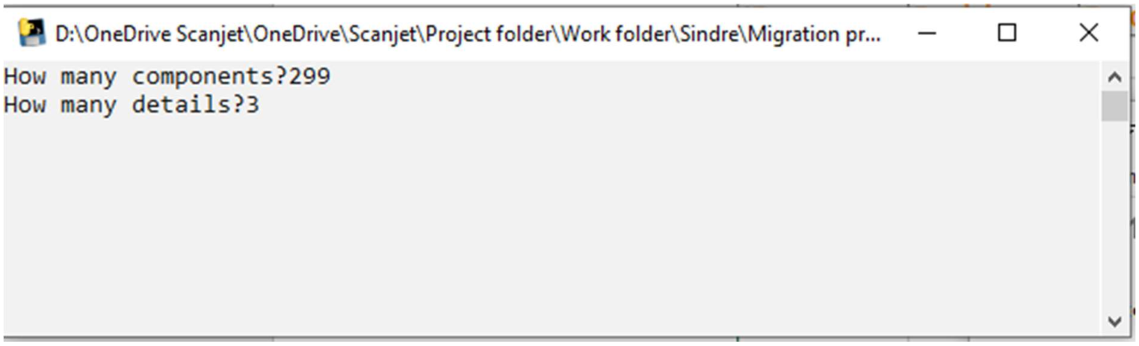
The Clarion format of the data is shown in *Figure 6-4*. This data will go through the Python script and be transformed to fit the PostgreSQL detail table.



| ID | Revision | Production Batch | Gal Revision |
|---|---|---|---|
| 351 | | 1 | 1 |
| 352 | | 1 | 0 |
| 353 | | 1 | 1 |
| 354 | | 1 | 1 |
| 355 | 0a | 2 | 1 |
| 356 | 0a | 2 | 1 |
| 357 | 0a | 2 | 1 |
| 358 | 0a | 2 | 1 |
| 359 | 0a | 2 | 1 |

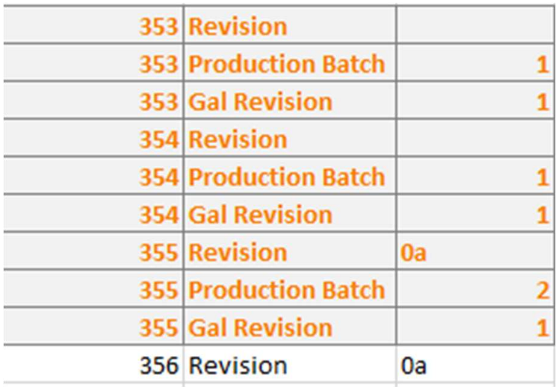
*Figure 6-4 Data before transformation of Inandata details*

The script is an executable file which will be available for Scanjet on a USB drive. When run, CMD will open and ask for number of components and details as shown in *Figure 6-5*, once given, the script will attempt to transform the data. As long as it does not go out of bounds (saying there is more components or details than it actually is) there should be no issues. The result can be seen in *Figure 6-6*.


D:\OneDrive Scanjet\OneDrive\Scanjet\Project folder\Work folder\Sindre\Migration pr...

```
How many components?299
How many details?3
```

*Figure 6-5 Executing the transformation script for one of the component tables, Inandata*



| | | |
|---|---|---|
| 353 | Revision | |
| 353 | Production Batch | 1 |
| 353 | Gal Revision | 1 |
| 354 | Revision | |
| 354 | Production Batch | 1 |
| 354 | Gal Revision | 1 |
| 355 | Revision | 0a |
| 355 | Production Batch | 2 |
| 355 | Gal Revision | 1 |
| 356 | Revision | 0a |

*Figure 6-6 Data after transformation of Inandata details*

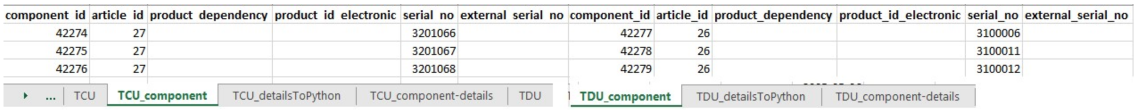
Each column today except *ID* shown in *Figure 6-4* , will be its own record in the new database illustrated in *Figure 6-6*.

### 6.2.4 Excel

The migration will revolve around an Excel book which can be seen in Appendix I. This book has sheets reflecting both the Clarion and PostgreSQL databases. It's used to read imported data from the Clarion database, then transform the data and putting it in sheets fitting the PostgreSQL database.

As the migration will take place after this bachelor project, the migration tool must be able to handle new entries in the old database. To mitigate this, several Excel-sheets are made for each of the old database's component tables.

Each Excel-sheet for *components* to the new database will increment primary keys based on the previous *component* sheet. This gives room to expand on primary keys for each of the Clarion database's component tables. *Figure 6-7* shows two sheets of the new database's *component* sheets, TDU will find the max *component_id* from *TCU* and create IDs for itself based on what *TCU's* highest ID is. This means components must be arranged in a given order to avoid duplicate primary keys. In Appendix I that order is from left to right, starting with *ABU_component*. Note that other tables must be uploaded before components can be uploaded according to *Figure 6-2*.



| component id | article id | product dependency | product id electronic | serial no | external serial no |
|---|---|---|---|---|---|
| 42274 | 27 | | | 3201066 | |
| 42275 | 27 | | | 3201067 | |
| 42276 | 27 | | | 3201068 | |

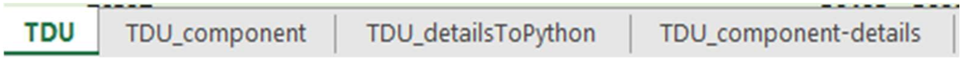
▸ ... | TCU | TCU_component | TCU_detailsToPython | TCU_component-details | TDU

| component_id | article_id | product_dependency | product_id_electronic | serial_no | external_serial_no |
|---|---|---|---|---|---|
| 42277 | 26 | | | 3100006 | |
| 42278 | 26 | | | 3100011 | |
| 42279 | 26 | | | 3100012 | |

TDU_component | TDU_detailsToPython | TDU_component-details

*Figure 6-7 Shows two component tables from the new database, where one (TDU) increments component_id from the other (TCU)*

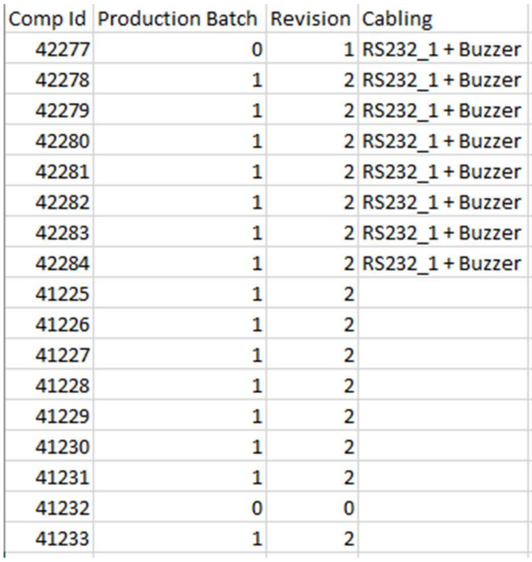
The system shown in *Figure 6-8* repeats itself for all components except for *sensordata*. *Sensordata* has several *_detailToPython* as the data stored has varying details.



TDU | TDU_component | TDU_detailsToPython | TDU_component-details

*Figure 6-8 An example from Appendix I showing the most common sheet types*

Having multiple sheets for the new database's *component_detail* is a necessity as the the Clarion database's tables have varying extra columns. These columns must go through a Python script and must have a *component_id*, a column name and a value. In *Figure 6-9* details from TDU and TCU are put into one *detailsToPython.* This will not give an error message, but will however indicate that TCU can have *cabling,* but does not have installed. This is not possible and should be avoided by using several sheets.



| Comp Id | Production Batch | Revision | Cabling |
|---|---|---|---|
| 42277 | 0 | 1 | RS232_1 + Buzzer |
| 42278 | 1 | 2 | RS232_1 + Buzzer |
| 42279 | 1 | 2 | RS232_1 + Buzzer |
| 42280 | 1 | 2 | RS232_1 + Buzzer |
| 42281 | 1 | 2 | RS232_1 + Buzzer |
| 42282 | 1 | 2 | RS232_1 + Buzzer |
| 42283 | 1 | 2 | RS232_1 + Buzzer |
| 42284 | 1 | 2 | RS232_1 + Buzzer |
| 41225 | 1 | 2 | |
| 41226 | 1 | 2 | |
| 41227 | 1 | 2 | |
| 41228 | 1 | 2 | |
| 41229 | 1 | 2 | |
| 41230 | 1 | 2 | |
| 41231 | 1 | 2 | |
| 41232 | 0 | 0 | |
| 41233 | 1 | 2 | |

*Figure 6-9 Putting TDU and TCU's detail in one sheet, TDU starting from component_id 42277 and TCU from 41225*

### 6.2.5 Formatting

The old dates are wrong with about 100 years, formula (6-1 rectifies the date value. Formatting to date are done by Excel as the value is written in days.

$$n = o - (365 \times 99) + 26 \qquad (6\text{-}1)$$

*n = calculated date*

*o = date to be calculated*

Formatting Boolean values are done in several ways, by replacing a symbol with *true* or *false.*

Several column names have been changed manually but will not be required to be done again during migration.

## 6.3 Application status

This chapter summarizes the status of the desktop application. First, each part of the application is individually inspected and evaluated. In the end, more general issues are explained.

### 6.3.1 Configuration layer

The XML configuration layer currently covers most of the initial design. The instantiated data structure has been tested and iteratively improved on over the course of the project. In the end, it serves as a solid foundation for dynamically created GUI controls, and their underlying logic.

Primitive data types and custom data types are now defined using the same enumerated list. This is not optimal, because custom data types depend on primitive data types. Additionally, custom data types may, in reality, consist of several different primitives, and the current implementation doesn’t handle this.

Otherwise the configuration only lacks one central function: Serializing database metadata automatically. Instead it has to be written semi-manually, as a set of useful code snippets for Visual Studio Code are included with the source code.

Currently there is no convenient way of debugging the configuration document within the application. To identify and fix issues isolated to the configuration document, there is no better way than using third-party options.

### 6.3.2 SQL interface

The SQL interface have been completed with satisfactory functionality when considering reading and modifying tables. The performance of reading and editing data is slow due to iterative control statements of Strings and conversion of data types. The best solution as of now appears to be through multithreading and should be considered in further development.

### 6.3.3 GUI classes

The GUI classes consist of three main branches and one standalone class. Most of these are implemented as designed, and cover the requirements specified for the application.

The *tab* class handles dynamic panels, but is itself not the base for, nor derived from, any other classes. The tab class covers its required functionality, and has been tested.

For the other three class hierarchies, most of the functionality is in place. However, the hierarchies themselves are shallow and is prone to design changes in the future. Additionally, constructors for the derived classes aren’t simplified enough. To support a more strictly structured main form, constructors for derived GUI classes should either always take all required data as parameters, or only data that are common for all custom GUI controls. Instead, the current solution bends this rule which makes for a more confusing source code.

*s_datafield* and its derivatives are mostly complete, but lack a unified way of setting data as AnsiString in controls that handle time-related data types. Otherwise, their visuals are roughly in place, and have three placeholder buttons implemented for further development.

The panel and button classes are the domain for most of the missing pieces. *Static* panels and buttons are in place. As for the *dynamic* controls, the filter related panels are completely untouched. The foreign key panels contain a grid, although they are both very unpolished. The rest of the panels, including buttons have their base functionality in place, but has not been properly tested.

All GUI components have to be heap-allocated. Because the design for using VCL components have been derived from testing, rather than designed top-to-bottom with prior knowledge, there are objects in the application still with clear signs of ignorant implementation. Memory leaks may occur which, depending on severity, could go unnoticed for a very long time.

### 6.3.4 Main source

The main form class is unfinished mostly as a product of missing pieces from the other code parts/layers. Almost all custom GUI classes are represented in this code, but some remain unreachable because of missing SQL connectivity and bugs in the implementation of *s_datastructure*.

Most observations on this are written in chapter 7.

### 6.3.5 General issues

Due to time constraints, the visibility of class members is in most cases set to public. This could easily lead to unintentional usage. In relation to this, the use of const and mutable is also highly sporadic. Although these points aren't directly mentioned in the requirements specification, it certainly impacts the overall code quality, and should be taken into consideration when continuing development.

Exception handling has been ignored and left for future work in all corners of the software. The upside to this is that no systems in the current solution have hidden flaws covered by clumsy exception handling.

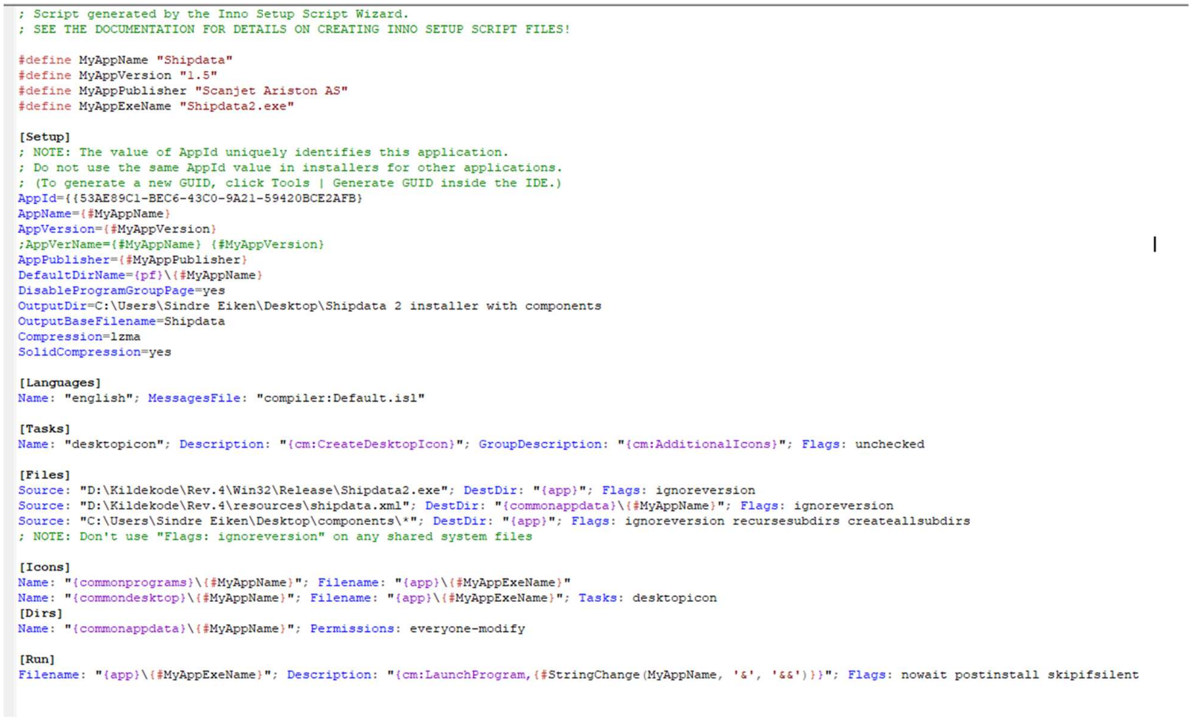
As with many software systems, string handling is a major issue. VCL objects only accept AnsiString with very little implicit conversion functionality. Due to the amount of different ways strings can be stored, this leads to equally many issues. The solution to this has mostly been to fix it once an issue arises, rather than an application-encompassing rule to avoid them in the first place. It should be noted that AnsiString is used more in the main source and GUI classes, while std::string is used in *s_datastructure*. Exceptions to this rule still occur, sometimes by necessity. [10]

Some of these points are discussed further in chapter 7.

## 6.4 Deployment

To deploy the application as an installation file, Inno Setup has been explored. It is a free installer for Windows applications made by Jordan Russel and Martijn Laan. A benefit using this program is that it's well documented. An installation file of the prototype was made with Inno Setup and tested through a virtual machine using Oracle Virtual Box. [11][12]

A code example is shown in *Figure 6-10*. The Inno Setup installer generates most of the code through a simple-to-use wizard. The exception is where to put the configuration file which must be manually scripted. During this test, it was decided to put the configuration file into the *commonappdata* directory.



```
; Script generated by the Inno Setup Script Wizard.
; SEE THE DOCUMENTATION FOR DETAILS ON CREATING INNO SETUP SCRIPT FILES!

#define MyAppName "Shipdata"
#define MyAppVersion "1.5"
#define MyAppPublisher "Scanjet Ariston AS"
#define MyAppExeName "Shipdata2.exe"

[Setup]
; NOTE: The value of AppId uniquely identifies this application.
; Do not use the same AppId value in installers for other applications.
; (To generate a new GUID, click Tools | Generate GUID inside the IDE.)
AppId={{53AE89C1-BEC6-43C0-9A21-59420BCE2AFB}
AppName={#MyAppName}
AppVersion={#MyAppVersion}
;AppVerName={#MyAppName} {#MyAppVersion}
AppPublisher={#MyAppPublisher}
DefaultDirName={pf}\{#MyAppName}
DisableProgramGroupPage=yes
OutputDir=C:\Users\Sindre Eiken\Desktop\Shipdata 2 installer with components
OutputBaseFilename=Shipdata
Compression=lzma
SolidCompression=yes

[Languages]
Name: "english"; MessagesFile: "compiler:Default.isl"

[Tasks]
Name: "desktopicon"; Description: "{cm:CreateDesktopIcon}"; GroupDescription: "{cm:AdditionalIcons}"; Flags: unchecked

[Files]
Source: "D:\Kildekode\Rev.4\Win32\Release\Shipdata2.exe"; DestDir: "{app}"; Flags: ignoreversion
Source: "D:\Kildekode\Rev.4\resources\shipdata.xml"; DestDir: "{commonappdata}\{#MyAppName}"; Flags: ignoreversion
Source: "C:\Users\Sindre Eiken\Desktop\components\*"; DestDir: "{app}"; Flags: ignoreversion recursesubdirs createallsubdirs
; NOTE: Don't use "Flags: ignoreversion" on any shared system files

[Icons]
Name: "{commonprograms}\{#MyAppName}"; Filename: "{app}\{#MyAppExeName}"
Name: "{commondesktop}\{#MyAppName}"; Filename: "{app}\{#MyAppExeName}"; Tasks: desktopicon
[Dirs]
Name: "{commonappdata}\{#MyAppName}"; Permissions: everyone-modify

[Run]
Filename: "{app}\{#MyAppExeName}"; Description: "{cm:LaunchProgram,{#StringChange(MyAppName, '&', '&&')}}"; Flags: nowait postinstall skipifsilent
```

*Figure 6-10: Program code for the installation file*

For the migration, Scanjet will receive a USB stick with several folders with Excel sheets and an executable file to run the migration at their behest. The main folder will contain two folders, one for the main Excel sheets:

- *01 Predefined data.xlsx*
- *02 Read Clarion files.xlsx*
- *03 Transformation.xlsx*
- *04 Insert book.xlsx*
- *Data migration guide.pdf*

The other folder will contain the executable file and its associated Excel files in a subfolder. The executable called *Migration.exe,* and in the subfolder *src* two Excel files:

- *01_To_transformation.csv*
- *02_Finished.csv*

By following the instructions of *Data migration guide* in Appendix G*,* Scanjet should be able to execute the migration.

Seeing as the database and migration parts of the project have reached a satisfactory level, Scanjet Ariston should consider additional possibilities regarding the future of the desktop application.

- Continue development on the current project as it was initially planned
- Reevaluate the project plan, but continue working on the current source code

- Inspect the application parts as they are now, and move some of them into a new project if they are considered usable
- Restart application development and only refer to the old source code when needed.

Out of the four main possibilities for continuing development, Scanjet is recommended to pursue the third option. The reason for this is that the project development plan and main form class need serious overhauls to account for a project capable of running with multiple developers. On the other hand, the SQL interface and XML configuration are fully functional, and may be transferred into a new project as they are. This is not to say these layers are flawless and demand absolutely no polish, but can be built upon rather than extracting code parts.

# 7 Discussion

When starting this project, the group was recommended by the supervisor to use the development methodology called Scrum. For two of the group members, this methodology was completely unknown. Additionally, the group had very little experience with planning software development. As a result, the planning could have been more efficient in the start with a little more research on software planning.

When developing the ER-model for the database, we used most of the aspects of Scrum. We had regular meetings with the customer to discuss the model, planned sprints of one week with tasks to be done and had an early prototype to evolve. However, when we started developing the application, we moved away from Scrum. Reasons might be that we were not too experienced with developing an application. We had to try and fail to get any progress and were struggling to work parallel with the same application. At this point we were going towards a more traditional sequential development approach where we had to finish a part completely before advancing to the next part. If something was not correct, we had to go back to the previous part and fix the problem before advancing.

## 7.1 The database

Judging by the initial requirements, the database had no real restrictions in terms of shape. As long as it could handle all the data from the old database and was made in PostgreSQL, we had the opportunity to design it as we saw fit. One major issue kept prodding at our project: Whether to sacrifice data type integrity for the sake of a cleaner database structure? Down the line we knew this would lead to a more complex desktop application. Because if the database can't keep the data type integrity, the application would have to. Looking back, this philosophy of generalization launched the project onto a slippery slope of unfortunate design choices. Simple problems suddenly called for complex solutions to account for variables where there once were constants.

Further design choices begged the question: to which degree do we need to model reality? Many columns today store strings instead of selecting from a predefined set of data. E.g., the *country* column in the *company* table, which, of course, only has a few hundred valid values. Single-column tables like this are useful to define *things that exist*, but also turns the design process into a philosophical discussion on the structural reality of all things. To draw the line somewhere, we decided to only create relations like this where it significantly helped the logic of the application. We mention this as a disclaimer for readers who might think we attempted to create a "perfect" database. In truth however, we have strived to create a product that is both intuitive and simple, all the while modelling it for the future of the desktop application and the data itself.

A choice was made during development of database to focus on designing main tables to comply with the 3NF standard. For tables not considered main tables, it was to fulfill at least the requirements for 1NF or higher where it was a significant advantage for the application.

After deciding on the ER model, user groups and access restrictions had to be defined. On this point, the customer requirements were very unclear. Some of this was of course because Scanjet themselves had not yet decided if the solutions we provided were good enough for deployment outside offices in Norway. Another reason was that the restrictions had to be defined within the structure of the new database, which we in the project group naturally understood better than them. What we believe we should have done, was to have Scanjet define these restrictions in a few sentences before getting mixed up in the thought process of the model design. We could then, based on this, design access restrictions on our own.

## 7.2 Migration

Starting the data migration project, no guidelines had been given on what data was to be transferred. What was given was that the data structure was greatly differing between the databases and a suggestion to use Devart for Excel. As the group had not partaken in any data migration before it was easy, and a good choice to use Excel. This gave a good overview of the data and an easy interface between databases, where one can simply point to cells in the other database. The Excel books created are performance wise, not efficient. Several Excel books must be used to perform tedious copying between books. Using more scripts and directly uploading to the databases would likely save time during the migration process but would require more planning. Due to the many tasks and little time, only one person has worked on this task.

For a larger database, Excel is not recommended. In the relatively small database migrated Excel uses significant time on processing tables.

For some transformation of data, PyCharm with Python was used. Before using Python, we attempted to use RAD Studio with C++. The documentation proved lacking and the group had issues reading CSV files. The work revolved around using loops and putting together two-dimensional arrays which any language should be able to perform. After some time attempting to use RAD studio, the group chose to use PyCharm as it has easy-to-use plugins for reading and writing CSV files.

A full-scale test may take over a week to complete. There has not been time to do this, however the concepts are tested. The issues, if any, will likely revolve around foreign and primary keys. Keys aren't always derived from a unique identifier, and may give duplicate or wrong keys, though unlikely. The other likely issue revolves around *memo* type fields, in other words, fields that have more than one value in the Clarion database. These fields are known to corrupt tables by being misplaced during import to Excel. An important lesson learned, is that the database should be locked for changes when migration design is started.

## 7.3 Application

On developing the desktop application, a lot of things could've been done differently. The sheer ambition of the sub-project didn't match that of e.g., the database. Nor were the hours disposed for the desktop application successfully split between project members. One of the issues with this, stems from the development methodology. Because the intention was to develop all project parts iteratively using a Scrum-like methodology, too little time was put aside for pure planning, design and brainstorming in general. Each sprint, the idea was for individual members to "make something work". This turned away the possibility of an overarching design. We don't blame this on the wrong choice of methodology, but rather our usage of it. Perhaps the outcome would have been different had we spent the first few sprints solely on planning and design tasks.

Had the overarching design been put into place early on, there would be a great deal more room for working in parallel. This has been one of the most damning factors for the development of the desktop application. During most times there was only one member working on it at a time. A solution to this could have been achieved by designing the GUI, SQL interface and XML configuration to be independent of each other. This would have led to a more complex development process when tying it all together, but perhaps an overall advantage.

Some of the individual design choices were also ill advised due to them demanding exponentially more time testing and therefore caused the project to halt at code implementation. An example of this is the idea of virtual views. No longer would problems be contained within a single table upon editing a record, but instead involve multiple chains of tables that all could have individual flaws related to many different layers of the application. Instead of defining these virtual views in the configuration document, they could be hard coded in the GUI. While a weaker, less extensible solution, it would also be simpler to test and therefore not block further development for as long as it did.

One of the requirements not met is the user manual. This was supposed to document the use cases for the application, but as it hasn't reached a satisfactory level, we decided against spending time creating this. However, the time was used to improve database documentation.

PgDAC objects appears to work best as designer objects and used for defined tasks, typically for *grids*. This is something PgDAC seem to do very well in the demo project that follows with the installation, it is however written in Delphi (programming language). The demo project mostly considers using grids which was something the group did not aim for. The documentation of PgDAC is quite weak, especially for C++, and few code examples are found online. Some code can be found on Devart's forums, again it's mostly for Delphi but was helpful as most methods uses close to same syntax and names. Devart employees are eager to assist in any problems and should have been used more during the development stage. [13]

One goal for the application was to make it dynamic and being able to adapt to the database. Thus, requiring general methods in the server-client interface. From the XML configuration file, resources as table names and column descriptions were available. These parameters were used to create a dynamic method for selecting and modifying data. In design, it appeared a good solution, realistically it was a slow method.

Alternatively, more processing should be server side. Either by use of specific stored procedures or dynamic SQL to move a lot of processing to the server. Triggers could also be used on some fields, most notably on the log data fields found in most tables.

JSON (JavaScript Object Notation) was considered as an alternative to XML. The format is used for many of the same use cases as XML, and is in many ways simpler. The main arguments for the use of JSON include array functionality, its lower verbosity and no need for end tags. However, *pugixml* proved very simple to use, with no noticeable drawbacks in terms of performance, and was therefore never replaced.

Considering the amount of trouble we encountered trying to develop an ambitious application in a new language in a new development environment, we looked back at completely different and possible approaches. Since the intention of the system is to provide database access abstraction for both in-office and international Scanjet employees, maybe one solution could fit both? The wish for a C++ based application was clear to us at the start of the project, but with sufficient research maybe a more adapted solution would have come into sight? Even if Scanjet has no intention of learning a new language, there are possibilities for dealing with HTML/CSS through so-called What-You-See-Is-What-You-Get editors, and the back-end code can execute programs with lower-level functionality.

## 7.4 Further development

During the discussions regarding user access groups, one of the topics were if users in Korea should only be able to see data in database regarding projects they were involved in. Although this topic was researched, no obvious solution was found. As part of the further development of user restrictions, this topic is recommended to look deeper into.

The use of proper namespaces was not considered in the original project design. This would be helpful in organizing the volume of symbols introduced in the project, at least when implementing in a larger project.

One of the software layers that we functionally finished was the SQL interface. While covering the demands we had when planning, it ended up slower than expected. This was due to heavy string handling, and layers upon layers of control-statements and loops. To combat this, the code should get a thorough inspection to see if any unnecessary copy operations are happening. In addition to this, if such measures still provide unsatisfactory results, asynchronous programming should get introduced. PgDAC are considered thread-safe, although, it may require many other code changes to make all related methods being thread-safe. Still, when dealing with possibly slow internet connections, this could yield very positive results in responsivity and overall user experience.[14]

One of the many problems that could emerge in the future are custom data types that are incompatible with any one input control, and therefore demand multiple.  For this problem the current design takes no measure, and would instead require multiple datafields for a dataset that is stored in only one string. To fix this, for say an array of tuples, this would have to be defined in a new class inherited from s_inputfield., or better yet streamline it by implementing some unified field to hold any number of input controls in the base class. Either way, this was beyond our reach for this project, but we urge future development to prioritize this issue before it becomes a more complex issue.

# 8 Summary

Scanjet Ariston AS want a new database and an associated application to store information about their projects and products to ensure product traceability. The following points summaries the goals achieved during this project.

- A new database has been developed using PostgreSQL and has been installed on Scanjet's central server.
- User groups and their restrictions has been defined for the new database and implemented on the server.
- A guide to understand the ER-model has been created for Scanjet Ariston AS to assist with further development of the system.
- Methods for migrating data from the existing system to the new database has been developed using Devart for MS Excel and a Python app.
- A complete guide on how to perform the migration is created and will be delivered to Scanjet Ariston AS. The migration itself will be done after this project by Scanjet themselves.
- A configuration layer using an XML resource file has been made in Visual Studio and can be imported to any Embarcadero or Visual C++ project through one header file. This header file depends on pugixml.
- An SQL interface has been developed using PgDAC that handles any database structure and can be imported to any Embarcadero C++ project through one header file. Its general design functionally works, but comes at a significant performance cost. It is dependent on the new configuration layer.
- GUI classes are partly designed and implemented in a desktop application. Some of its functionality is ready for use, but demands additional work to reach project requirements. It is also dependent on the new configuration layer.
- A prototype of the application has been developed using the three software parts above. It isn't finished, some of which is a product of the GUI classes being incomplete

# References

[1] Capesoft, what is clarion, 2019, available: https://www.capesoft.com/accessories/WhatIsClarion.htm , accessed on: 19.04.19

[2] It knowledge portal, Software development methodologies, available: http://www.itinfo.am/eng/software-development-methodologies/, accessed on:

[3] Wikipedia, Database normalization, available: https://en.wikipedia.org/wiki/Database_normalization, accessed on: 14.04.19

[4] Embarcadero, TObject class, available: http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN/html/delphivclwin32/System__TObject.html accessed on: 12.05.19

[5] W3cschools, SVG Tutorial, available: https://www.w3schools.com/graphics/svg_intro.asp

[6] Pugixml, Pugixml, available: https://pugixml.org/ accessed on: 12.05.19

[7] Github, Topic: xml, 2019, available: https://github.com/topics/xml?l=c%2B%2B accessed on: 02.05.19

[8] Stack overflow, parameterized query, available: https://stackoverflow.com/questions/4712037/what-is-parameterized-query, accessed on: 14.05.19

[9] Stack exchange, parameterized table name in dynamic sql, available: https://dba.stackexchange.com/questions/212815/parameterize-table-name-in-dynamic-sql, accessed on: 14.05.19

[10] Jonathan Corbet, improving kernel string handling, available: https://lwn.net/Articles/643376/, accessed on: 10.05.19

[11] Jrsoftware, Inno Setup, available: http://www.jrsoftware.org/isinfo.php accessed on: 13.05.19

[12] Virtualbox, Virtualbox, available: https://www.virtualbox.org/, accessed on: 13.05.19

[13] PgDAC, Component List, available: https://www.devart.com/pgdac/docs/components_pgdac.htm, accessed on: 14.05.19

[14] PgDAC, Are the PgDAC connections components thread-safe?, available: https://www.devart.com/pgdac/docs/faq.htm, accessed on: 14.05.19

# Appendices

Appendix A

Universitetet
i Sørøst-Norge

**Fakultet for Teknologi, Naturvitenskap og Maritime fag**

# PRH612 Bachelor thesis

| | |
|---|---|
| **Study:** | Information technology and automation |
| **Project group:** | IA6-10-19 |
| **Title:** | Project and inventory manager - creating an expandable application in C++ and a future oriented database with PostgreSQL |
| **Supervisor:** | Hans-Petter Halvorsen |
| **Collaborator:** | Scanjet Ariston AS |

## Project background:

Scanjet Ariston AS, henceforth referred to as Scanjet, is a company that delivers and develops a universal and flexible solution for tank monitoring and control. To ensure traceability of their products, a database is used to store information of each product.
Today Scanjet uses a database based on Clarion, which is a file-based proprietary solution. Because of the uncertainty of Clarions future, Scanjet evaluates it to be a considerable risk to continue using Clarion. Because Scanjet has departments localized in several countries such as China, Korea, Indonesia, Norway and Sweden there is now a need for a client/server-based solution.

## Task description

The customer Scanjet wants a new database solution to replace their old Clarion database and an application to manipulate the new database.
The task consists making a client/server solution for a PostgreSQL database with server-side user authentication and creating an application in C++ with Rad Studio which manipulates data in the database. The bachelor group will have to make requirements for the new application based on Scanjet's old application and conversations with Scanjet. Scanjet requires a test document to verify the system, a user manual and that the entire system is in English.
Additional tasks consist of creating a web application, functions for programming Scanjet's components, an optimization tool for SQL tables and a new configuration generator for components in their system.

## Scope

The project scope will consist of three main parts: Database, desktop application and data migration from existing database. The database part will consist of creating an ER-model for the database structure, server-controlled user structure and authentication, and SQL-injection proof data access.

The desktop application part consists of making a suitable GUI for adding, reading and changing data in the database, a solution for generating reports and enforcing user access. Configuration generator, sensor programming, web application and optimization tool for SQL tables are excluded from the scope but will be implemented in the application by the company itself.

**Signatur**

Veileder (dato og signatur):

Studenter (dato og signatur (alle må signere)):

4/3-19

4/3-19 Martin Holm

04/03-19

# System requirement specification for

PostgreSQL database “Shipdata 2” for Scanjet Ariston AS

Version 1.0

Prepared by Sindre Eiken, Martin Holm and Espen Buø

Date..

# Contents

# 1. Introduction

## 1.1. Purpose

This specification document describes the requirements and functions specified this “Shipdata 2” database for Scanjet Ariston AS. The existing database is based on Clarion, which is a file-based proprietary solution. Scanjet Ariston evaluates it to be considerable risk to continue using Clarion because of its uncertain future. As the company has grown into an international company, there is now need for a client/server-based solution.

## 1.2. Definitions, acronyms and abbreviations

- DB – Database
- ER – Entity-Relation

## 1.3. References

https://krazytech.com/projects/sample-software-requirements-specificationsrs-report-airline-database

# 2. General description

## 2.1. Product perspective

The database system stores the following information:

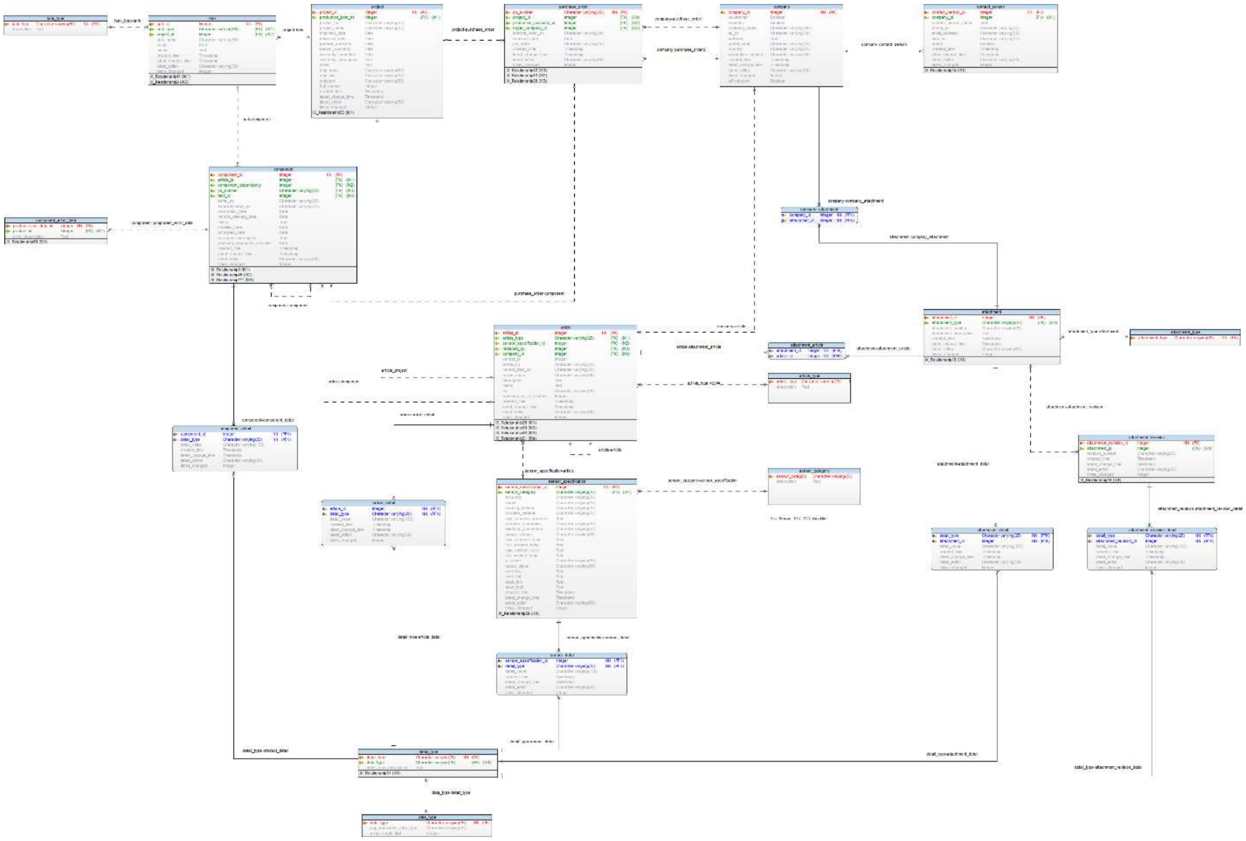
- **Product details**

  It includes serial numbers, article number, product type, calibration data, cable length etc.

- **Project data**

  It includes project number, project name, location, ship number, ship name, shipyard etc.

## 2.2. Product features



## 2.3. User class and characteristics

Users of this system should be able to retrieve and add information about projects and equipment.

<u>**User function group 0:**</u> System operator (Sysop)

- User access.
- Full access.
- Add, edit and delete tables in database.

- Show, edit and delete information.
- Configuration file generation.

**User function group 1:** Administrator (Admin)

- Show, edit and delete information.
- Configuration file generation.

**User function group 2:** User

- Show and edit information.
- Configuration file generation.

**User function group 3:** KR admin

- See only components that are available and not
- Limited access to certain tables.
  - Be able to replace sensors.
  - Add tank data.

**User function group 4:** KR user

- Read only access to certain tables.
  - Available sensors and electronics.
  - Limited product information to Kora production.

## 2.4. Operating environment

Following points is the operating environment for the “Shipdata 2” system:

- Client/server system
- Operating system: Windows
- Database: PostgreSQL
- Platform: C++

# 3. Specific requirements

## 3.1. Database

### 1.1.1. Functional requirements

**Database structure:**

- The database should be made using PostgreSQL.
- The database should be manipulated using PGadmin.
- The database should be based on an ER database model.
- All existing data must be shaped into entity relation model.

**Data logging:**

- All independent tables must include simple log data.

**Accessibility:**

- Accessible from offices in Norway.

**Security:**

- Unauthorized attempts to retrieve data should fail.
- Database must be SQL injection proof.

**Migration:**

- All existing data must be transferred to the new database.

### 1.1.2. Non-functional requirements

**Structure:**

- Database should be extensible.
- Relationships between project, products and articles should be unambiguous.

**Documentation:**

- All SQL procedures and views must be documented outside of code.

## 3.2. Application

### 1.1.3. Functional requirements

**Project functions:**

- User should be able to add new, edit and delete projects.
- User should be able to add new, edit and delete customer details.
- User should be able to add new, edit and delete case handler details.
- User should be able to add new, edit and delete ship name and ship yard details.

**Tank functions:**

- User should be able to add new, edit and delete tank data.
- Tank data should always be linked to a project.
- User should be able to add, replace and delete tank sensors.
- User should be able to copy tank lists.

**Sensor functions:**

- Users should be able to add new, edit and delete sensors and sensor data.
- Users should be able to program sensors.

**Electronics:**

- User should be able to add new, edit and delete electronics components. This includes AD cards, amplifiers and Zener-barriers.

**Hloa sensor:**

- Users should be able to add new, edit and delete Hloa Sensors.
- Users should be able to add new, edit and delete Hloa electronics.
- User should be able to program Hloa equipment.

**Surveyor models:**

Surveyor models include TCU, TDU, ANZB485, ANZBANAR6, ANZBANAR7, Water ingress, TPC350, TPC196 and TPC 140.

- Users should be able to add, edit and delete surveyor models.
- User should be able to see Old modules.

**Print reports**

Users should be able to print reports for following items:

- System overview.
- Sensor list for one system.
- Sensor labels for one system.
- Print free sensors.

- Print all sensors.
- Show total sensors.

**Other products:**

Other products include Inclinometer, Monitor and PC.

- Users should be able to add, edit and delete and delete products.

**Users should be able to generate a configuration file based on data from database.**

**Log in function.**

**Users should be able to use a search in the software.**

**Users should be able to find fault history.**

**Support/help.**

### 1.1.4. Non-functional requirements

**Language:**

- English

**GUI**

- The application should feel better than the original.

**Source-code:**

- All code should be human-readable.

## 3.3. Migration of data

For an overview over which tables are transferred during the migration, see Figure 3-1.

- Generally, all data in the tables should be transferred with its relations intact.
- Most of the yellow tables should be linked to an article number
- A guide to executing the migration should be provided
- Purchase orders should be generated with suffix PO: and the project number
- The tool created for migrating data should be extensible to allow migration at a later time when new data is stored in the Clarion database

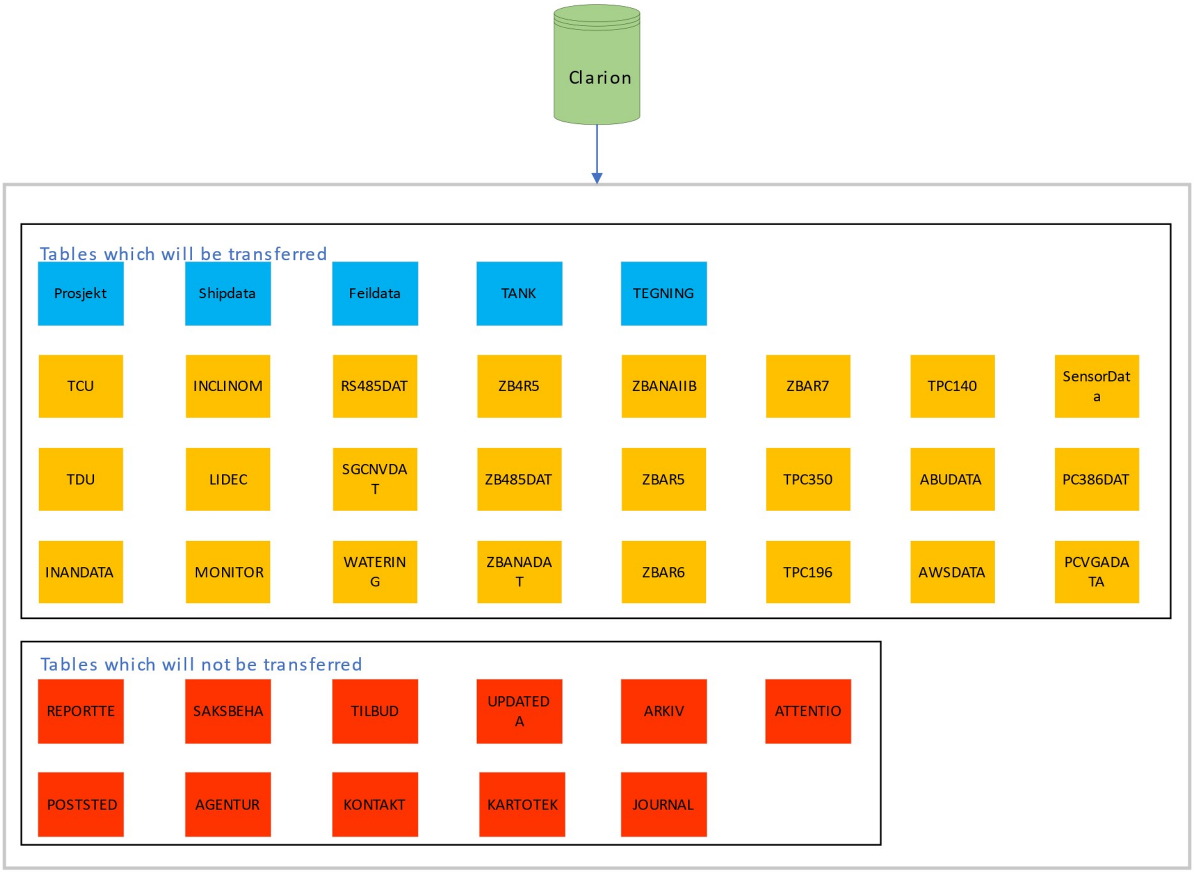


*Figure 3-1 Shows which tables will be migrated*

There are however exceptions:

- Sensordata should not be connected to article numbers
- ABU and AWS should not have details (Non-general columns)
- Many of the non-general columns in yellow tables should be distributed to component_detail, article_detail and sensor_detail, but will not, all should be used as component_details. Only new data should use the other detail tables.

Scanjet Ariston will supply data for some tables in PostgreSQL:

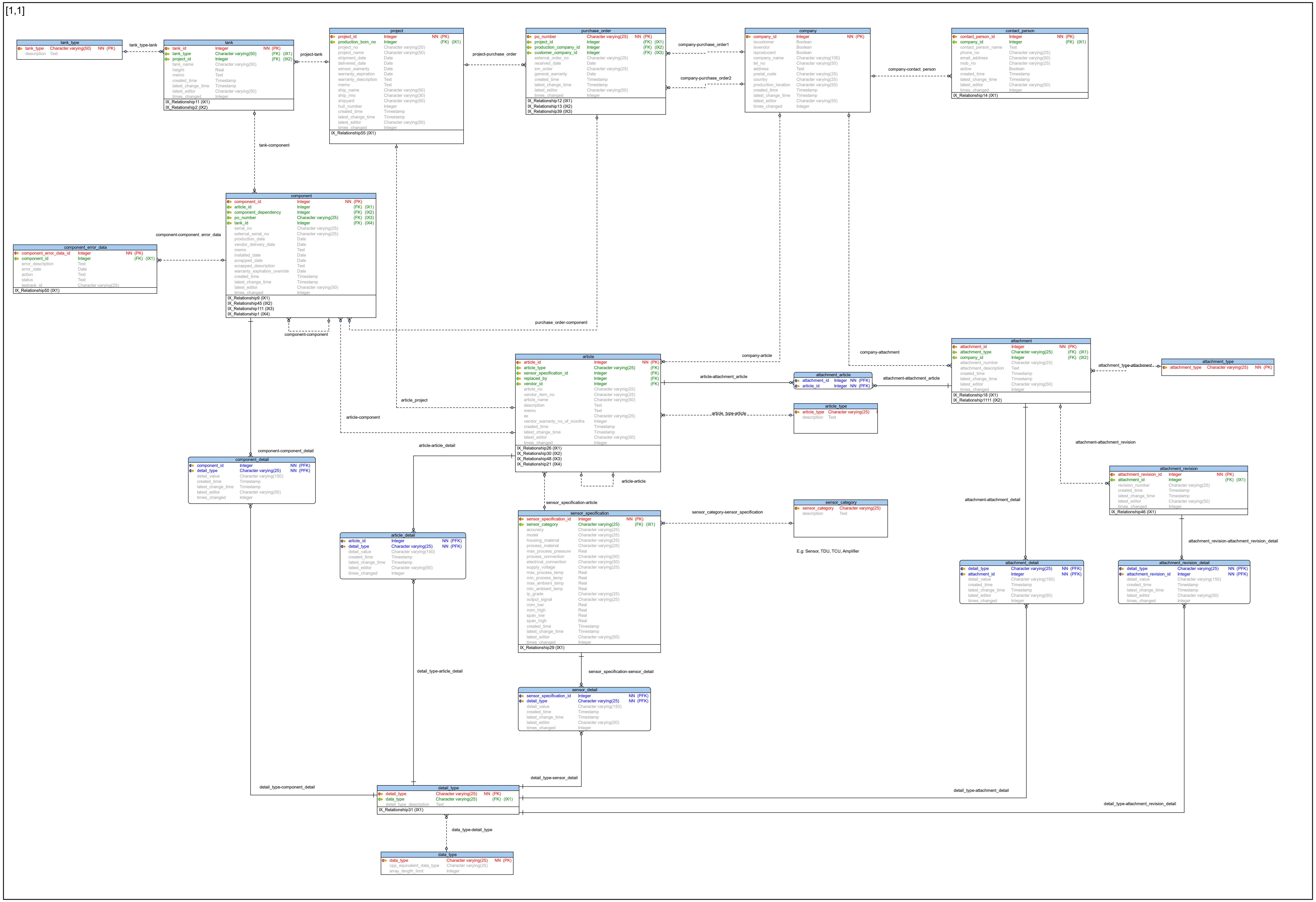
- Article
- Article type
- Sensor specification table
- Sensor category
- Company
- Contact Person

# 4. External interface requirements

## 4.1. Software requirements

- Operating system: Windows 10 32/64 bit.
- PostgreSQL database.

# Appendix C


[1,1]
tank_type
tank_type Character varying(50) NN (PK)
description Text
tank_type-tank
tank
tank_id Integer NN (PK)
tank_type Character varying(50) (FK) (IX1)
project_id Integer (FK) (IX2)
tank_name Character varying(50)
height Real
memo Text
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship11 (IX1)
IX_Relationship2 (IX2)
project-tank
project
project_id Integer NN (PK)
production_bom_no Integer (FK) (IX1)
project_no Character varying(20)
project_name Character varying(50)
shipment_date Date
delivered_date Date
sensor_warranty Date
warranty_expiration Date
warranty_description Text
memo Text
ship_name Character varying(50)
ship_imo Character varying(30)
shipyard Character varying(50)
hull_number Integer
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship55 (IX1)
project-purchase_order
purchase_order
po_number Character varying(25) NN (PK)
project_id Integer (FK) (IX1)
production_company_id Integer (FK) (IX2)
customer_company_id Integer (FK) (IX3)
external_order_no Character varying(25)
received_date Date
sm_order Character varying(25)
general_warranty Date
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship12 (IX1)
IX_Relationship13 (IX2)
IX_Relationship39 (IX3)
company-purchase_order1
company-purchase_order2
company
company_id Integer NN (PK)
iscustomer Boolean
isvendor Boolean
isproducent Boolean
company_name Character varying(100)
tel_no Character varying(50)
address Text
postal_code Character varying(25)
country Character varying(25)
production_location Character varying(50)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
company-contact_person
contact_person
contact_person_id Integer NN (PK)
company_id Integer (FK) (IX1)
contact_person_name Text
phone_no Character varying(25)
email_address Character varying(50)
mob_no Character varying(25)
active Boolean
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship14 (IX1)
tank-component
component-component_error_data
component_error_data
component_error_data_id Integer NN (PK)
component_id Integer (FK) (IX1)
error_description Text
error_date Date
action Text
status Text
testrack_id Character varying(25)
IX_Relationship50 (IX1)
component
component_id Integer NN (PK)
article_id Integer (FK) (IX1)
component_dependency Integer (FK) (IX2)
po_number Character varying(25) (FK) (IX3)
tank_id Integer (FK) (IX4)
serial_no Character varying(25)
external_serial_no Character varying(25)
production_date Date
vendor_delivery_date Date
memo Text
installed_date Date
scrapped_date Date
scrapped_description Text
warranty_expiration_override Date
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)
component-component
purchase_order-component
company-article
company-attachment
article
article_id Integer NN (PK)
article_type Character varying(25) (FK)
sensor_specification_id Integer (FK)
replaced_by Integer (FK)
vendor_id Integer (FK)
article_no Character varying(25)
vendor_item_no Character varying(25)
article_name Character varying(50)
description Text
memo Text
ex Character varying(25)
vendor_warranty_no_of_months Integer
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
article-attachment_article
attachment_article
attachment_id Integer NN (PFK)
article_id Integer NN (PFK)
attachment-attachment_article
attachment
attachment_id Integer NN (PK)
attachment_type Character varying(25) (FK) (IX1)
company_id Integer (FK) (IX2)
attachment_number Character varying(25)
attachment_description Text
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship18 (IX1)
IX_Relationship1111 (IX2)
attachment_type-attachment
attachment_type
attachment_type Character varying(25) NN (PK)
article_project
article-component
article_type-article
article_type
article_type Character varying(25)
description Text
component-component_detail
component_detail
component_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
article-article_detail
article-article
attachment-attachment_revision
attachment_revision
attachment_revision_id Integer NN (PK)
attachment_id Integer (FK) (IX1)
revision_number Character varying(25)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship46 (IX1)
sensor_specification-article
attachment-attachment_detail
sensor_category-sensor_specification
sensor_category
sensor_category Character varying(25)
description Text
E.g: Sensor, TDU, TCU, Amplifier
sensor_specification
sensor_specification_id Integer NN (PK)
sensor_category Character varying(25) (FK) (IX1)
accuracy Character varying(25)
model Character varying(25)
housing_material Character varying(25)
process_material Character varying(25)
max_process_pressure Real
process_connection Character varying(50)
electrical_connection Character varying(50)
supply_voltage Character varying(25)
max_process_temp Real
min_process_temp Real
max_ambient_temp Real
min_ambient_temp Real
ip_grade Character varying(25)
output_signal Character varying(25)
nom_low Real
nom_high Real
span_low Real
span_high Real
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
IX_Relationship29 (IX1)
article_detail
article_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
attachment_revision-attachment_revision_detail
attachment_detail
detail_type Character varying(25) NN (PFK)
attachment_id Integer NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
attachment_revision_detail
detail_type Character varying(25) NN (PFK)
attachment_revision_id Integer NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
detail_type-article_detail
sensor_specification-sensor_detail
sensor_detail
sensor_specification_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_value Character varying(150)
created_time Timestamp
latest_change_time Timestamp
latest_editor Character varying(50)
times_changed Integer
detail_type-sensor_detail
detail_type-component_detail
detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
detail_type_description Text
IX_Relationship31 (IX1)
detail_type-attachment_detail
detail_type-attachment_revision_detail
data_type-detail_type
data_type
data_type Character varying(25) NN (PK)
cpp_equivalent_data_type Character varying(25)
array_length_limit Integer

Appendix D

# Shipdata 2 ER-model guide

by Sindre Eiken, Martin Holm and Espen Buø

07.05.19, Porsgrunn

# 1 Introduction

This document is intended as a guide to understand the entity-relationship model developed for Shipdata 2 for Scanjet Aristion AS. The structure of this document is done in such a way that it can be used as a “encyclopedia” when working with the database. The table of content in this document has hyperlinks to each chapter for easier navigation.

## 1.1 Nomenclature list

| | | |
|---|---|---|
| Attribute | – | A column in a database table. |
| Entity | – | In this context, a table in the database. |
| ER | – | Entity relation. |
| Relationship | – | A connection between tables in the database. |

# 2 Overview

Figure 2-1 illustrates an overview of the ER-model designed for the database *shipdata*. The overview does not show normal attributes, only primary keys and foreign keys.

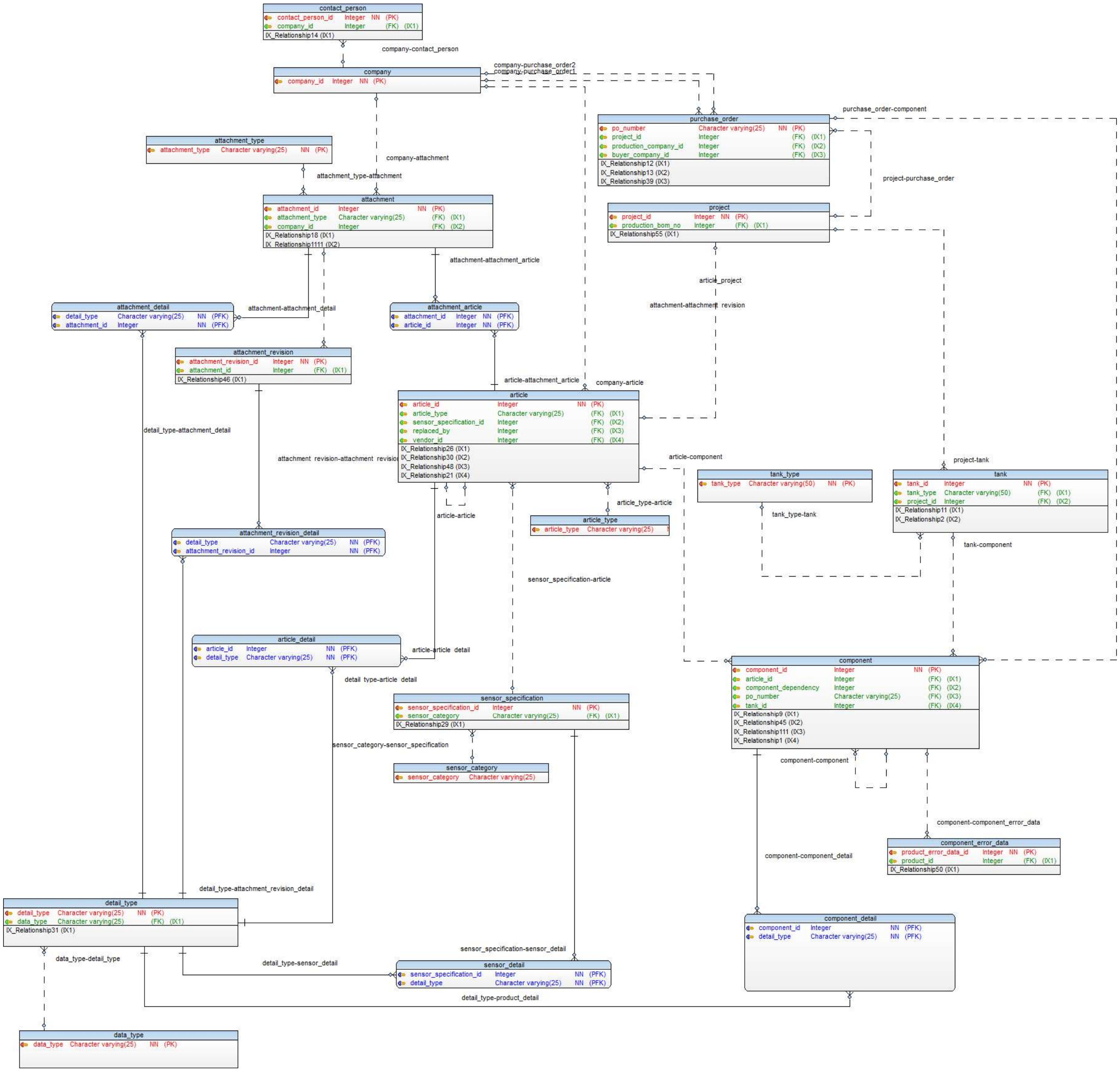

contact_person
contact_person_id Integer NN (PK)
company_id Integer (FK) (IX1)
IX_Relationship14 (IX1)
company-contact_person
company
company_id Integer NN (PK)
company-purchase_order2
company-purchase_order1
purchase_order-component
purchase_order
po_number Character varying(25) NN (PK)
project_id Integer (FK) (IX1)
production_company_id Integer (FK) (IX2)
buyer_company_id Integer (FK) (IX3)
IX_Relationship12 (IX1)
IX_Relationship13 (IX2)
IX_Relationship39 (IX3)
project-purchase_order
attachment_type
attachment_type Character varying(25) NN (PK)
company-attachment
attachment_type-attachment
attachment
attachment_id Integer NN (PK)
attachment_type Character varying(25) (FK) (IX1)
company_id Integer (FK) (IX2)
IX_Relationship18 (IX1)
IX_Relationship1111 (IX2)
project
project_id Integer NN (PK)
production_bom_no Integer (FK) (IX1)
IX_Relationship55 (IX1)
attachment-attachment_article
article_project
attachment_detail
detail_type Character varying(25) NN (PFK)
attachment_id Integer NN (PFK)
attachment-attachment_detail
attachment_article
attachment_id Integer NN (PFK)
article_id Integer NN (PFK)
attachment-attachment revision
attachment_revision
attachment_revision_id Integer NN (PK)
attachment_id Integer (FK) (IX1)
IX_Relationship46 (IX1)
article-attachment_article
company-article
article
article_id Integer NN (PK)
article_type Character varying(25) (FK) (IX1)
sensor_specification_id Integer (FK) (IX2)
replaced_by Integer (FK) (IX3)
vendor_id Integer (FK) (IX4)
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
detail_type-attachment_detail
attachment revision-attachment revisio
article-component
project-tank
tank_type
tank_type Character varying(50) NN (PK)
tank
tank_id Integer NN (PK)
tank_type Character varying(50) (FK) (IX1)
project_id Integer (FK) (IX2)
IX_Relationship11 (IX1)
IX_Relationship2 (IX2)
article-article
article_type-article
article_type
article_type Character varying(25)
tank_type-tank
tank-component
attachment_revision_detail
detail_type Character varying(25) NN (PFK)
attachment_revision_id Integer NN (PFK)
sensor_specification-article
article_detail
article_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
article-article detail
detail type-article detail
component
component_id Integer NN (PK)
article_id Integer (FK) (IX1)
component_dependency Integer (FK) (IX2)
po_number Character varying(25) (FK) (IX3)
tank_id Integer (FK) (IX4)
IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)
sensor_specification
sensor_specification_id Integer NN (PK)
sensor_category Character varying(25) (FK) (IX1)
IX_Relationship29 (IX1)
sensor_category-sensor_specification
sensor_category
sensor_category Character varying(25)
component-component
component-component_error_data
component_error_data
product_error_data_id Integer NN (PK)
product_id Integer (FK) (IX1)
IX_Relationship50 (IX1)
component-component_detail
detail_type-attachment_revision_detail
detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
IX_Relationship31 (IX1)
component_detail
component_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
data_type-detail_type
sensor_specification-sensor_detail
detail_type-sensor_detail
sensor_detail
sensor_specification_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_type-product_detail
data_type
data_type Character varying(25) NN (PK)

*Figure 2-1: overview of tables in the database model with primary and foreign keys.*

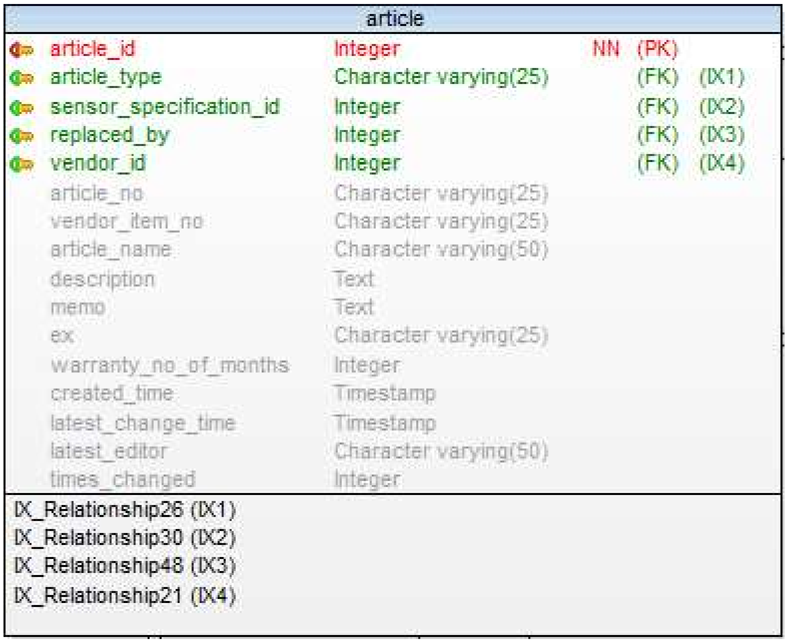
# 3 Entities and attributes

In entities where records of significant data are stored the following attributes are included:

- **Created_time** – A timestamp for when the record was created.
- **Latest_change_time** – A timestamp for when record was last changed.
- **Latest_editor** – Username of the latest editor of a record.
- **Times_changed** – A counter for how many times a record has been changed.

## 3.1 Article

Article is an entity shown in *Figure 3-1* that contains common information about a group of components.



| article | | | | |
|---|---|---|---|---|
| article_id | Integer | NN | (PK) | |
| article_type | Character varying(25) | | (FK) | (IX1) |
| sensor_specification_id | Integer | | (FK) | (IX2) |
| replaced_by | Integer | | (FK) | (IX3) |
| vendor_id | Integer | | (FK) | (IX4) |
| article_no | Character varying(25) | | | |
| vendor_item_no | Character varying(25) | | | |
| article_name | Character varying(50) | | | |
| description | Text | | | |
| memo | Text | | | |
| ex | Character varying(25) | | | |
| warranty_no_of_months | Integer | | | |
| created_time | Timestamp | | | |
| latest_change_time | Timestamp | | | |
| latest_editor | Character varying(50) | | | |
| times_changed | Integer | | | |

IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)

*Figure 3-1: Article table*

### 3.1.1 Article - Primary and foreign keys:

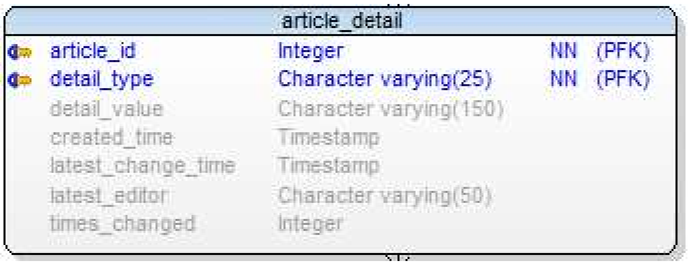
- **Article_id** – The primary key of article, each record must have a unique number.
- **Article_type** – A foreign key attribute from the entity *article_type*.
- **Sensor_specification_id** – A foreign key attribute from the entity sensor_specification. Used when an article is a sensor.
- **Replaced_by** – A foreign key from the entity *article*. Used when an article is obsolete and replace by another article number.
- **Vendor_id** – A foreign key from the entity company. Used to connect a company as vendor for articles.

### 3.1.2 Article - attributes

- **Article_no –** Attribute for article numbers. In practice used to identify each article, but duplicates may occur.
- **Vendor_item_no** – Attribute storing the vendor's item number for an article.
- **Article name** – Descriptive name of an article.
- **Description**
- **Memo** – attribute reserved if any extra information should be stored.
- **ex** – explosion proof classification description.
- **Warranty_no_of_months**

## 3.2 Article detail

The entity *article_detail* shown in *Figure 3-2* contains records of details which are not common for all articles. The entity enables new details in the future to be added.



| article_detail | | | |
|---|---|---|---|
| article_id | Integer | NN | (PFK) |
| detail_type | Character varying(25) | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

*Figure 3-2: Article detail entity*

### 3.2.1 Article detail - Primary and foreign keys:

This tables have two attributes combined as a primary key:

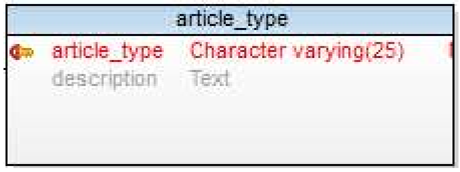
- **Article_id** – Primary foreign key from the entity *article.*
- **Detail_type** – Primary foreign key from the entity *detail_type.*

### 3.2.2 Article detail – Attributes

- **Detail_value** – Attribute for storing a value of a detail record.

## 3.3 Article type

The entity *article_type* shown in *Figure 3-3* contains records of article types. Examples: sensor, TDU.



| article_type | |
|---|---|
| article_type | Character varying(25) |
| description | Text |

*Figure 3-3: article type entity.*

### 3.3.1 Article type - Primary and foreign keys:

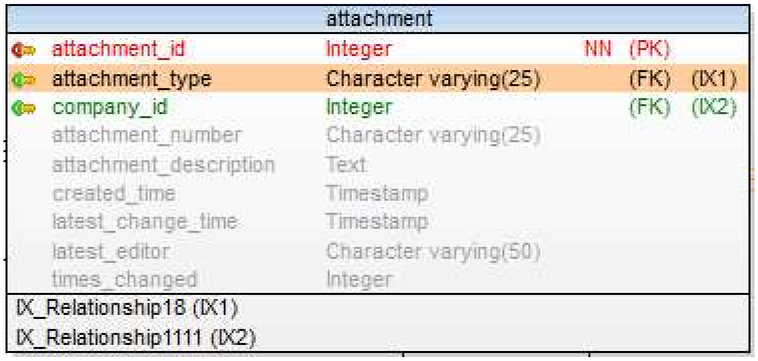
- **Article_type –** Descriptive name of an article.

### 3.3.2 Article type – attributes

- **Description –** Free text describing the article type.

## 3.4 Attachment

Attachment shown in *Figure 3-4* is an entity that contains records of information for different types of attachments. Examples are drawings and certificates.



| attachment | | | | |
|---|---|---|---|---|
| attachment_id | Integer | NN | (PK) | |
| attachment_type | Character varying(25) | | (FK) | (IX1) |
| company_id | Integer | | (FK) | (IX2) |
| attachment_number | Character varying(25) | | | |
| attachment_description | Text | | | |
| created_time | Timestamp | | | |
| latest_change_time | Timestamp | | | |
| latest_editor | Character varying(50) | | | |
| times_changed | Integer | | | |
| IX_Relationship18 (IX1) | | | | |
| IX_Relationship1111 (IX2) | | | | |

*Figure 3-4: Attachment entity.*

### 3.4.1 Attachment – Primary and foreign keys:

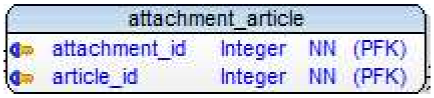
- **Attachment_id** – The entity's primary key.
- **Attachment_type** – A foreign key from the entity *attachment_type*.
- **Company_id –** A foreign key from the entity *company*.

### 3.4.2 Attachment – Attributes

- **Attachment_number –** Scanjet's internal attachment number.
- **Attachment_description**

## 3.5 Attachment article

The entity attachment article shown in *Figure 3-5* is a connecting entity between *article* and *attachment*.



attachment_article

| | | | |
|---|---|---|---|
| attachment_id | Integer | NN | (PFK) |
| article_id | Integer | NN | (PFK) |

*Figure 3-5: Attachment article entity*

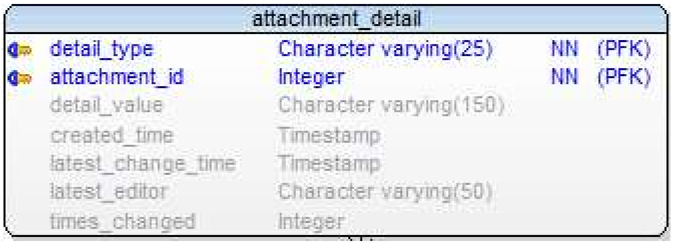
### 3.5.1 Attachment article – Primary and foreign keys:

The entity *attachment_article* has a combined primary key that consists of the following attributes:

- **Attachment_id –** Primary foreign key from entity *attachment*.
- **Article_id –** Primary foreign key from entity *article.*

## 3.6 Attachment detail

The entity *attachment_detail* shown in *Figure 3-6* contains records of details not common for all attachments, as well as enables new attributes to be added to an attachment in the future.



attachment_detail

| | | | |
|---|---|---|---|
| detail_type | Character varying(25) | NN | (PFK) |
| attachment_id | Integer | NN | (PFK) |
| detail_value | Character varying(150) | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

*Figure 3-6: Attachment detail entity*

### 3.6.1 Attachment detail – Primary and foreign keys:

The entity *attachment_detail* uses a combined primary key from the following foreign keys:

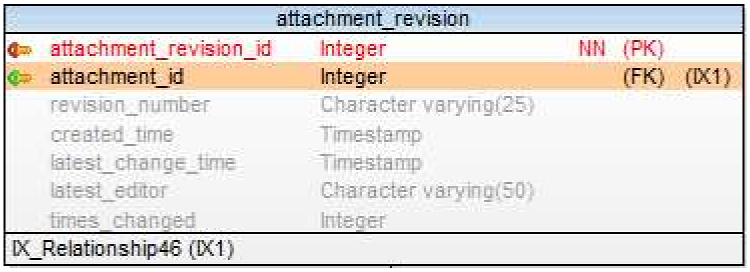
- **Detail_type** – Primary foreign key from entity *detail_type.*
- **Attachment_id –** Primary foreign key from entity *attachment_id*.

### 3.6.2 Attachment detail – Attributes

- **Detail_value –** Attribute to store a value to each record.

## 3.7 Attachment revision

The entity *attachment_revision* shown in *Figure 3-7* holds records regarding revision information for attachments.



| attachment_revision | | |
|---|---|---|
| attachment_revision_id | Integer | NN (PK) |
| attachment_id | Integer | (FK) (IX1) |
| revision_number | Character varying(25) | |
| created_time | Timestamp | |
| latest_change_time | Timestamp | |
| latest_editor | Character varying(50) | |
| times_changed | Integer | |
| IX_Relationship46 (IX1) | | |

*Figure 3-7: Attachment revision entity*

### 3.7.1 Attachment revision – Primary and foreign keys:

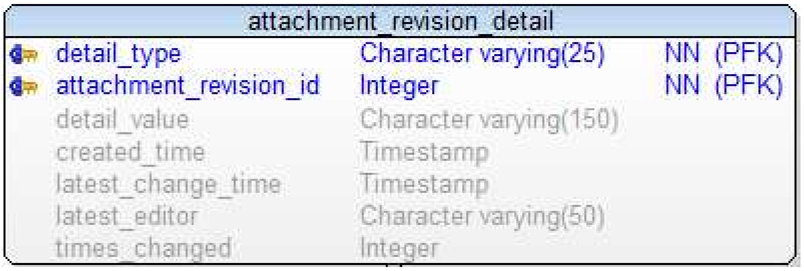
- **Attachment_revision_id –** Primary key attribute for each record.
- **Attachment_id –** Foreign key from the entity *attachment* to connect a revision record.

### 3.7.2 Attachment revision – Attributes

- **Revision_number –** Revision number for an attachment.

## 3.8 Attachment revision detail

The entity *attachment_revision_detail* shown in figure *Figure 3-8* holds records of uncommon details regarding attachment revisions. This enables new attributes to be added in the future.



| attachment_revision_detail | | |
|---|---|---|
| detail_type | Character varying(25) | NN (PFK) |
| attachment_revision_id | Integer | NN (PFK) |
| detail_value | Character varying(150) | |
| created_time | Timestamp | |
| latest_change_time | Timestamp | |
| latest_editor | Character varying(50) | |
| times_changed | Integer | |

*Figure 3-8: Attachment revision detail entity*

### 3.8.1 Attachment revision detail – Primary and foreign keys:

Attachment revison detail uses a combined primary key from the following foreign keys:

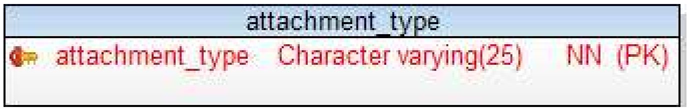
- **Detail_type –** Primary foreign key from the entity *detail_type*.
- **Attachment_revision_id –** Primary foreign key from entity *attachment_revision.*

### 3.8.2 Attachment revision detail – attributes

- **Detail_value –** Attributing holding a value for each record.

## 3.9 Attachment type

The entity *attachment_type* shown in *Figure 3-9* contains records describing each type of attachments.



attachment_type

| | | |
|---|---|---|
| attachment_type | Character varying(25) | NN (PK) |

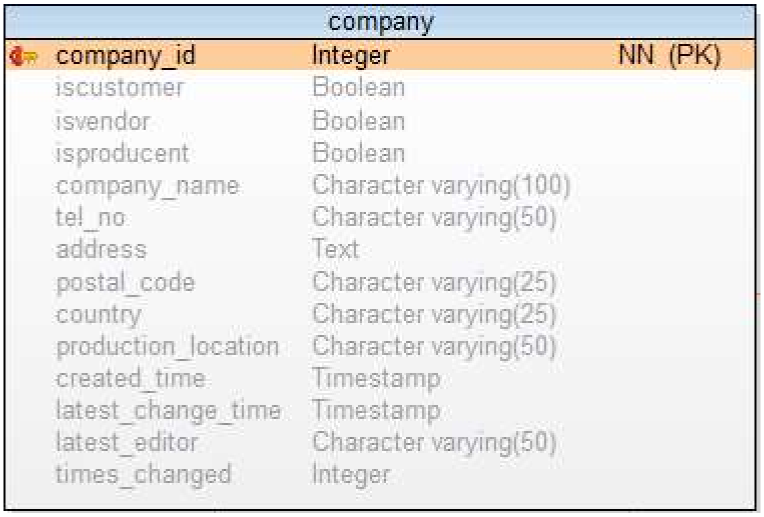
*Figure 3-9: Attachment type entity*

### 3.9.1 Attachment type – Primary and foreign keys:

- **Attachment_type** – Primary key for the entity. A short type description of various types of attachments.

## 3.10 Company

The entity *company* shown in *Figure 3-10* holds records for companies. Records can be either a customer, vendor or production company.



company

| | | |
|---|---|---|
| company_id | Integer | NN (PK) |
| iscustomer | Boolean | |
| isvendor | Boolean | |
| isproducent | Boolean | |
| company_name | Character varying(100) | |
| tel_no | Character varying(50) | |
| address | Text | |
| postal_code | Character varying(25) | |
| country | Character varying(25) | |
| production_location | Character varying(50) | |
| created_time | Timestamp | |
| latest_change_time | Timestamp | |
| latest_editor | Character varying(50) | |
| times_changed | Integer | |

*Figure 3-10: Company entity.*

### 3.10.1 Company – Primary and foreign keys:

- **Company_id** – Primary key of the entity. Represents a unique id for each company.

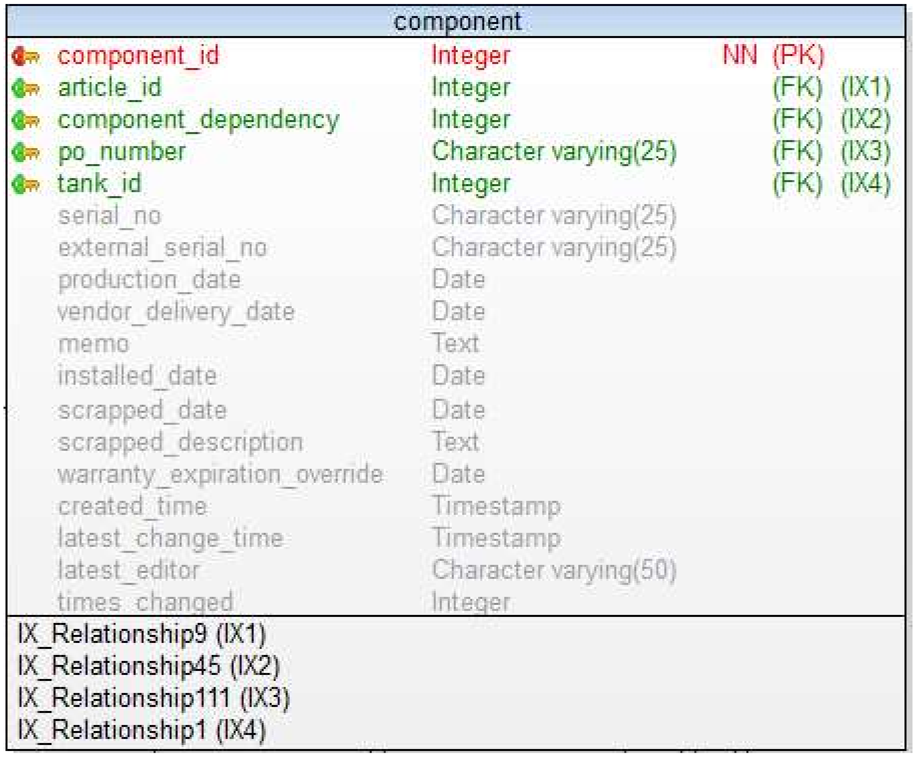
### 3.10.2 Company – attributes

- **Iscustomer** – An attribute which is either false or true depending if the record is a customer.
- **Isvendor** – An attribute which is either false or true depending if the record is a vendor.
- **Isproducent** – An attribute which is either false or true depending if the record is a producent.

- **Company_name** – The company’s name.
- **Tel_no** – the company’s phone number.
- **Address** – The company’s address.
- **Postal_code** – The company’s postal code.
- **Country** – which country the company is located in.
- **Production_location** – where the production is located of a company.

## 3.11 Component

The entity *component* shown in *Figure 3-11* holds records for each component sold by Scanjet Ariston AS.



| component | | | | |
|---|---|---|---|---|
| component_id | Integer | NN | (PK) | |
| article_id | Integer | | (FK) | (IX1) |
| component_dependency | Integer | | (FK) | (IX2) |
| po_number | Character varying(25) | | (FK) | (IX3) |
| tank_id | Integer | | (FK) | (IX4) |
| serial_no | Character varying(25) | | | |
| external_serial_no | Character varying(25) | | | |
| production_date | Date | | | |
| vendor_delivery_date | Date | | | |
| memo | Text | | | |
| installed_date | Date | | | |
| scrapped_date | Date | | | |
| scrapped_description | Text | | | |
| warranty_expiration_override | Date | | | |
| created_time | Timestamp | | | |
| latest_change_time | Timestamp | | | |
| latest_editor | Character varying(50) | | | |
| times_changed | Integer | | | |

IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)

*Figure 3-11: Component entity.*

### 3.11.1 Component – Primary and foreign keys:

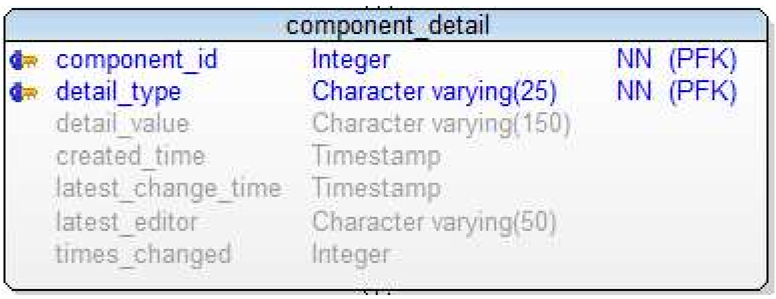
- **Component_id** – Primary key to separate each record.
- **Article_id** – Foreign key from the entity article. Each record can be connected to an article.
- **Component_dependency** – Foreign key from component itself. Some products are delivered as one piece but in practice consists of several registered components.
- **Po_number** – Foreign key from the entity *purchase_order*. Each component is connected to an order.
- **Tank_id** – Foreign key from the entity *tank*. Components like sensors are installed in tanks on a ship.

### 3.11.2 Component – attributes

- **Serial_no** – Attribute for Scanjet's serial number on components.
- **External_serial_no** – Attribute for vendors serial number on a component.
- **Production_date** – the date which the component was produced.
- **Vendor_delivery_date** – When the component was delivered by a vendor.
- **Memo** – free text attribute for any additional information.
- **Installed_date** – The date which the component was installed.
- **Scrapped_date** – If a component is scrapped it is possible to add a date to a serial number.
- **Scapped_description** – Descripting for scrapped components.
- **Warranty_expiration_override** – If a general warranty is overridden on one component.

## 3.12 Component detail

The entity *component_detail* shown in *Figure 3-12* holds records of additional attributes for records in *component* which are not common for all records.



| component_detail | | |
|---|---|---|
| component_id | Integer | NN (PFK) |
| detail_type | Character varying(25) | NN (PFK) |
| detail_value | Character varying(150) | |
| created_time | Timestamp | |
| latest_change_time | Timestamp | |
| latest_editor | Character varying(50) | |
| times_changed | Integer | |

*Figure 3-12: component detail entity.*

### 3.12.1 Component detail – Primary and foreign keys:

*Component_detail* uses a combined primary key of the following foreign keys:

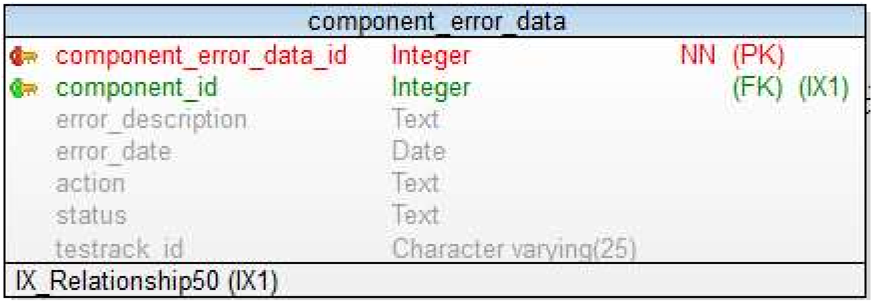
- **Component_id** – foreign key from the entity *component.*
- **Detail_type** – Foreign key from the entity *detail_type*.

### 3.12.2 Component detail – Attributes

- **Detail_value** – Attribute holding a value for each detail record.

## 3.13 Component error data

The entity *component_error_date* shown in *Figure 3-13* contains records of errors registered on a component.



component_error_data

| | | | |
|---|---|---|---|
| component_error_data_id | Integer | NN | (PK) |
| component_id | Integer | | (FK) (IX1) |
| error_description | Text | | |
| error_date | Date | | |
| action | Text | | |
| status | Text | | |
| testrack_id | Character varying(25) | | |

IX_Relationship50 (IX1)

*Figure 3-13: component error data entity*

### 3.13.1 Component error data – Primary and foreign keys:

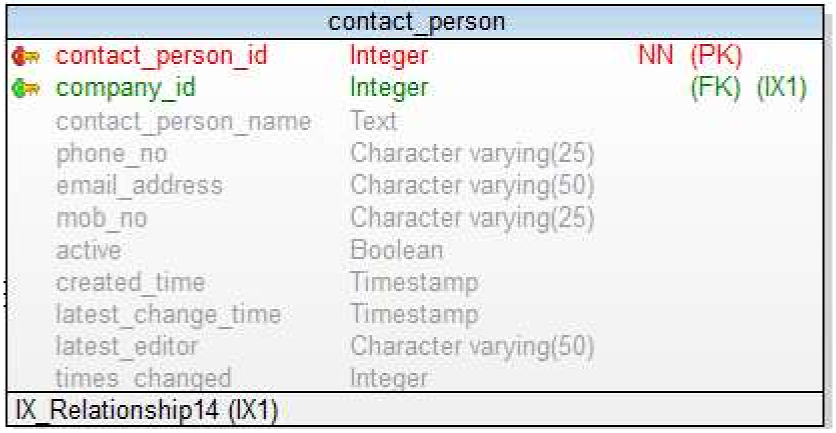
- **Component_error_data_id** – Primary key for the entity. A unique number for each record.
- **Component_id** – Foreign key from the entity *component* connecting an error record to a component.

### 3.13.2 Component error data – attributes

- **Error_description** – Free text attribute for describing the error registered.
- **Error_date -** Which date the error was registered.
- **Action** – Attribute describing which action is preformed to handle the error.
- **Status** – The status of the error. E.g. repaired, discarded etc.

## 3.14 Contact person

The entity *contact_person* shown in *Figure 3-14* holds records of contact persons for companies.



contact_person

| | | | |
|---|---|---|---|
| contact_person_id | Integer | NN | (PK) |
| company_id | Integer | | (FK) (IX1) |
| contact_person_name | Text | | |
| phone_no | Character varying(25) | | |
| email_address | Character varying(50) | | |
| mob_no | Character varying(25) | | |
| active | Boolean | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

IX_Relationship14 (IX1)

*Figure 3-14: Contact person entity*

### 3.14.1 Contact person – Primary and foreign keys:

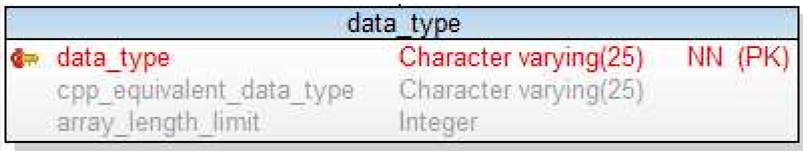
- **Contact_person_id** – Primary key for the entity. Requires a unique number for each record.
- **Company_id** – Foreign key from the entity *company*. Attribute required to connect a company to a contact person.

### 3.14.2 Contact person – attributes

- **Contact_person_name** – Name of the contact person.
- **Phone_no** – Landline number of the contact person.
- **Email_address** – Email address of the contact person.
- **Mob_no** – Cellphone number of the contact person.
- **Active** – An attribute that can either be false or true depending if the person is still active as a contact person.

## 3.15 Data type

The entity *data_type* shown in *Figure 3-15* holds records of different data types for details.



| data_type | | |
|---|---|---|
| data_type | Character varying(25) | NN (PK) |
| cpp_equivalent_data_type | Character varying(25) | |
| array_length_limit | Integer | |

*Figure 3-15: Data type entity.*

### 3.15.1 Data type – Primary and foreign keys:

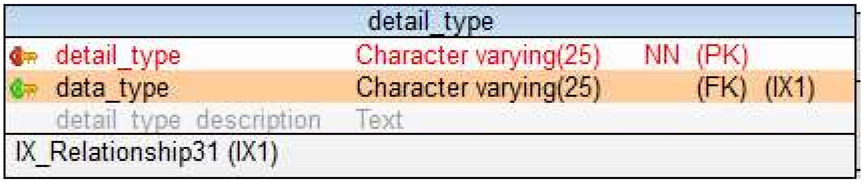
- **Data_type** – Primary key of the entity describing which data type the record is.

### 3.15.2 Data type – attributes

- **Cpp_equivalent_data_type** – The equivalent data type name in C++ language.
- **Array_length_limit** – The max length of the data type.

## 3.16 Detail type

The entity *detail_type* shown in *Figure 3-16* holds records of detail types used for each detail entities.



| detail_type | | |
|---|---|---|
| detail_type | Character varying(25) | NN (PK) |
| data_type | Character varying(25) | (FK) (IX1) |
| detail_type_description | Text | |
| IX_Relationship31 (IX1) | | |

*Figure 3-16: Detail type entity.*

### 3.16.1 Detail type – Primary and foreign keys:

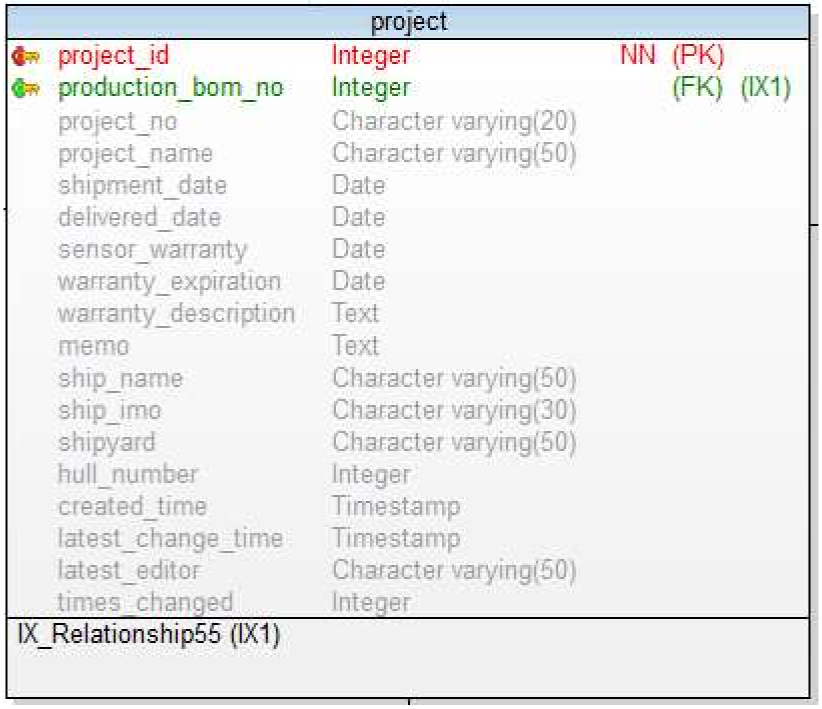
- **Detail_type –** Primary key of the entity. Short descriptive name of a detail type.
- **Data_type –** Foreign key from the entity *data_type* descripting which datatype the detail type is.

### 3.16.2 Detail type – attributes

- **Detail_type_desciption –** Free text describing the detail type.

## 3.17 Project

The entity *project* shown in *Figure 3-17* holds records of each project Scanjet Ariston has done.



| project | | | |
|---|---|---|---|
| project_id | Integer | NN (PK) | |
| production_bom_no | Integer | (FK) | (IX1) |
| project_no | Character varying(20) | | |
| project_name | Character varying(50) | | |
| shipment_date | Date | | |
| delivered_date | Date | | |
| sensor_warranty | Date | | |
| warranty_expiration | Date | | |
| warranty_description | Text | | |
| memo | Text | | |
| ship_name | Character varying(50) | | |
| ship_imo | Character varying(30) | | |
| shipyard | Character varying(50) | | |
| hull_number | Integer | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |
| IX_Relationship55 (IX1) | | | |

*Figure 3-17: Project entity.*

### 3.17.1 Project – Primary and foreign keys:

- **Project_id –** Primary key of the entity. A unique number for each record.
- **Production_bom_no –** A foreign key from the entity *article*. Each project has an article number.

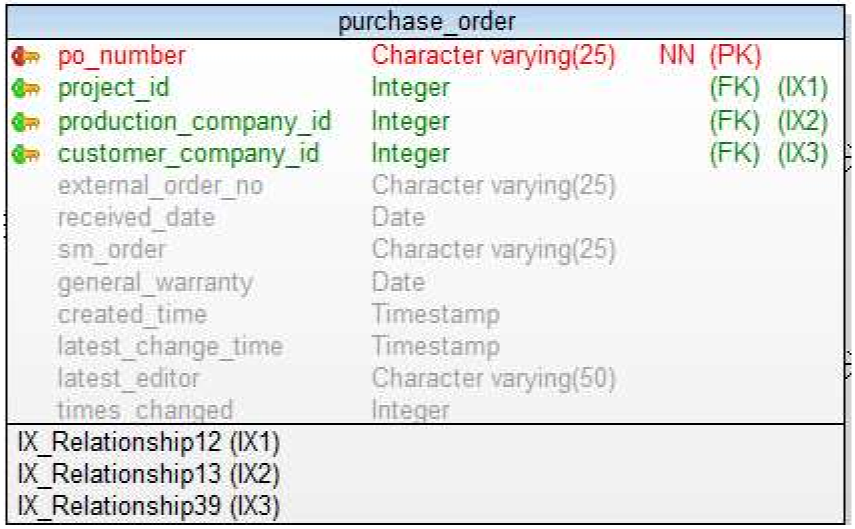
### 3.17.2 Project – attributes

- **Project_no –** Internal project number.
- **Project_name –** Name of the project.
- **Shipment_date –** When the project was shipped.

- **Delivered_date** – When the project was delivered.
- **Sensor_warranty** – The warranty expiration date of sensor equipment in the project.
- **Warranty_expiration** – The general warranty expiration for the project.
- **Warranty_description** – Description of the warranty.
- **Memo** – Free text for additional information.
- **Ship_name** – Name of the ship the record is for.
- **Ship_imo** – IMO number of the ship.
- **Shipyard** – Which shipyard the ship is being built.
- **Hull_number** – The ship's hull number.

## 3.18 Purchase order

The entity *purchase_order* shown in *Figure 3-18* holds records of each purchase order recived by Scanjet Ariston.



purchase_order

| | | | | |
|---|---|---|---|---|
| po_number | Character varying(25) | NN | (PK) | |
| project_id | Integer | | (FK) | (IX1) |
| production_company_id | Integer | | (FK) | (IX2) |
| customer_company_id | Integer | | (FK) | (IX3) |
| external_order_no | Character varying(25) | | | |
| received_date | Date | | | |
| sm_order | Character varying(25) | | | |
| general_warranty | Date | | | |
| created_time | Timestamp | | | |
| latest_change_time | Timestamp | | | |
| latest_editor | Character varying(50) | | | |
| times_changed | Integer | | | |

IX_Relationship12 (IX1)
IX_Relationship13 (IX2)
IX_Relationship39 (IX3)

*Figure 3-18: Purchase order entity.*

### 3.18.1 Purchase order – Primary and foreign keys:

- **Po_number** – Primary key of the entity. Each order has a unique internal PO number.
- **Project_id** – Foreign key from the entity *project*. Connecting a purchase order to a project.
- **Production_company_id** – Foreign key from the entity *company*. Which company has produced the order.
- **Customer_company_id** – Foreign key from the entity *company.* It describes which company is the customer of the order.

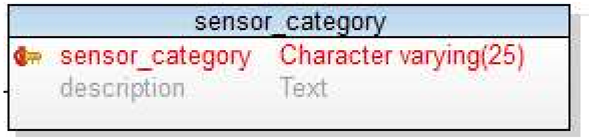
### 3.18.2 Purchase order – attributes

- **External_order_no** – External order number.
- **Recived_date** – which date was the order recived.

- **Sm_order** – Order number in Scanjet concern's system.
- **General_warranty** – The expiration date of the general warranty of a purchase order.

## 3.19 Sensor category

The entity *sensor_category* shown in *Figure 3-19* contains records of the different sensor categories used for articles. E.g. pressure, temperature etc.



| sensor_category | |
|---|---|
| sensor_category | Character varying(25) |
| description | Text |

*Figure 3-19: Sensor category entity*

### 3.19.1 Sensor category – Primary and foreign keys:

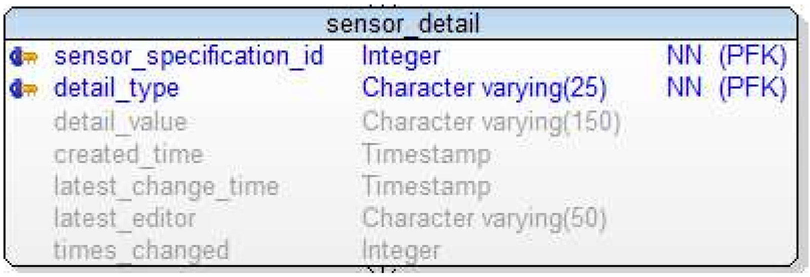
- **Sensor_category** – Primary key of the entity. Short descriptive text of the sensor category. Max length 25 characters.

### 3.19.2 Sensor category – attributes

- **Description** – Free text for describing the sensor category.

## 3.20 Sensor detail

The entity *sensor_detail* shown in *Figure 3-20* holds records of additional sensor specification details which are not common for *sensor* articles.



| sensor_detail | | |
|---|---|---|
| sensor_specification_id | Integer | NN (PFK) |
| detail_type | Character varying(25) | NN (PFK) |
| detail_value | Character varying(150) | |
| created_time | Timestamp | |
| latest_change_time | Timestamp | |
| latest_editor | Character varying(50) | |
| times_changed | Integer | |

*Figure 3-20: Sensor detail entity.*

### 3.20.1 Sensor detail – Primary and foreign keys:

*Sensor_detail* has a combined primary key of the following foreign keys:

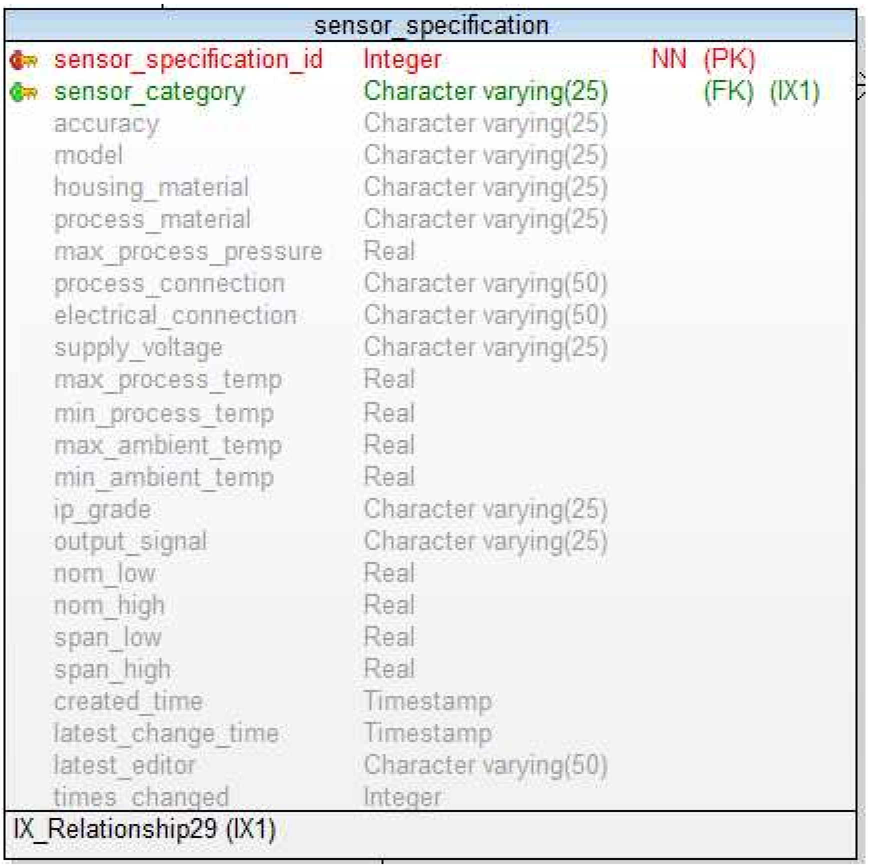
- **Sensor_specification_id** – Foreign key for the entity *sensor_specification*.
- **Detail_type** – Foreign key from the entity *detail_type.*

### 3.20.2 Sensor detail – attributes

- **Detail_value** – Attribute that holds a value for each sensor detail record.

## 3.21 Sensor specification

The entity *sensor_specification* shown in *Figure 3-21* holds records for articles of type sensor which require additional information.



sensor_specification

| | | | |
|---|---|---|---|
| sensor_specification_id | Integer | NN | (PK) |
| sensor_category | Character varying(25) | | (FK) (IX1) |
| accuracy | Character varying(25) | | |
| model | Character varying(25) | | |
| housing_material | Character varying(25) | | |
| process_material | Character varying(25) | | |
| max_process_pressure | Real | | |
| process_connection | Character varying(50) | | |
| electrical_connection | Character varying(50) | | |
| supply_voltage | Character varying(25) | | |
| max_process_temp | Real | | |
| min_process_temp | Real | | |
| max_ambient_temp | Real | | |
| min_ambient_temp | Real | | |
| ip_grade | Character varying(25) | | |
| output_signal | Character varying(25) | | |
| nom_low | Real | | |
| nom_high | Real | | |
| span_low | Real | | |
| span_high | Real | | |
| created_time | Timestamp | | |
| latest_change_time | Timestamp | | |
| latest_editor | Character varying(50) | | |
| times_changed | Integer | | |

IX_Relationship29 (IX1)

*Figure 3-21: Sensor specification entity.*

### 3.21.1 Sensor specification – Primary and foreign keys:

- **Sensor_specification_id** – Primary key of the entity. A unique identification number for each record.
- **Sensor_category** – Foreign key from the entity *sensor_category*. Describing the category of each sensor specification record.

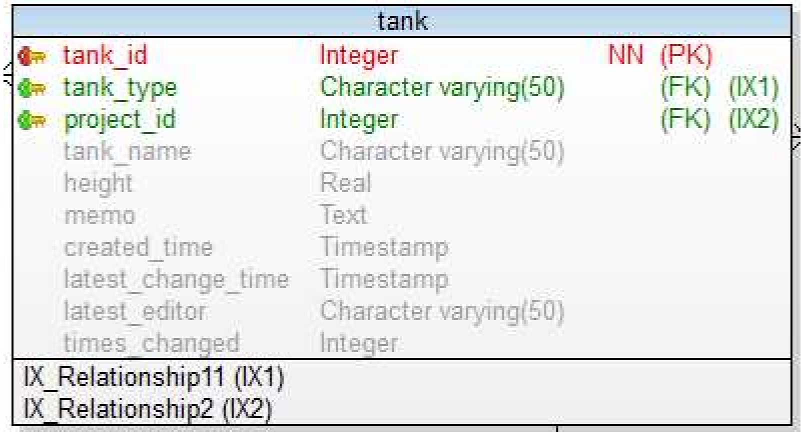
### 3.21.2 Sensor specification – attributes

- **Accuracy** – The accuracy of a sensor article.
- **Model** – model description of a sensor article.
- **House_material** – Which material the housing of a sensor article is made of.
- **Process_material** – Which material the process equipment of a sensor article is made of.
- **Max_prcoess_pressure** – Maximum process pressure the sensor article is made for.
- **Process_conection** – which connection the sensor article has to the process.

- **Electrical_connection** – which type of electrical connection the sensor article has.
- **Supply_voltage** – which supply voltage the sensor article is rated for.
- **Max_process_temp** – maximum process temperature the sensor article is rated for.
- **Min_process_temp** – minimum process temperature the sensor article is rated for.
- **Max_ambient_temp** – Maximum ambient temperature the sensor article is rated for.
- **Min_ambient_temp -** Minimum ambient temperature the sensor article is rated for.
- **Ip_grade** – The IP grade classification of the sensor article.
- **Output_signal** – which output signal the sensor article is rated for.
- **Nom_low** – which is the nominal low limit the sensor article is rated for.
- **Nom_high -** which the nominal high limit the sensor article is rated for.
- **Span_low** – Which zero point the sensor article is rated for.
- **Span_high** – Which range the sensor article is rated for.

## 3.22 Tank

The entity *tank* shown in *Figure 3-22* contains records with information for each tank components has been installed in.



tank

| | | | | |
|---|---|---|---|---|
| tank_id | Integer | NN | (PK) | |
| tank_type | Character varying(50) | | (FK) | (IX1) |
| project_id | Integer | | (FK) | (IX2) |
| tank_name | Character varying(50) | | | |
| height | Real | | | |
| memo | Text | | | |
| created_time | Timestamp | | | |
| latest_change_time | Timestamp | | | |
| latest_editor | Character varying(50) | | | |
| times_changed | Integer | | | |

IX_Relationship11 (IX1)
IX_Relationship2 (IX2)

*Figure 3-22: Tank entity.*

### 3.22.1 Tank – Primary and foreign keys:

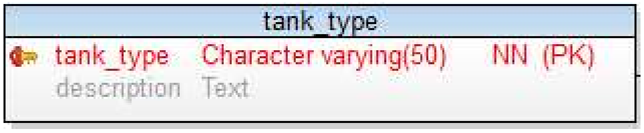
- **Tank_id** – Primary key of the entity. A unique number identifying each tank record.
- **Tank_type** – Foreign key from the entity *tank_type*. A type description of the tank. E.g. cargo, ballast.
- **Project_id** – Foreign key for the entity *project*. Connecting tank records to projects.

### 3.22.2 Tank – attributes

- **Tank_name** – Name of the tank.
- **Height** – the height of the tank.
- **Memo** – free text for additional information.

## 3.23 Tank type

The entity *tank_type* shown in *Figure 3-23* contains records describing each tank type.



| | tank_type | | |
|---|---|---|---|
| 🔑 | tank_type | Character varying(50) | NN (PK) |
| | description | Text | |

*Figure 3-23: tank_type entity.*

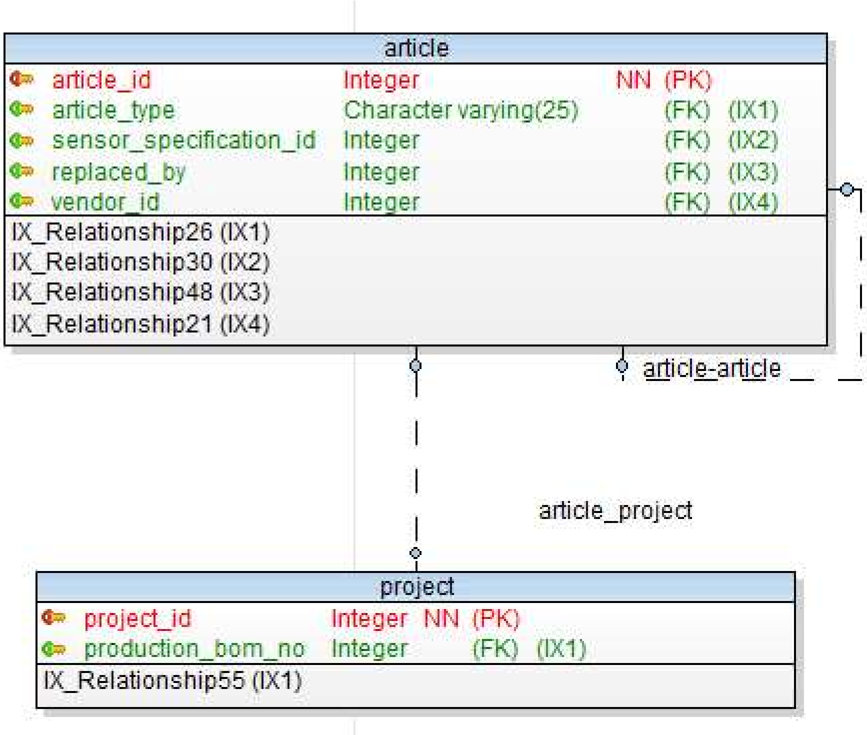
### 3.23.1 Tank type – Primary and foreign keys:

- **Tank_type –** Primary key of the entity. Short descriptive text regarding the tank type.
- **Decription –** free text for descripting the tank type.

# 4 Relationships

## 4.1 Article – project

Each record in the entity *project* can be connected to one record in *article*. The relationship is shown in *Figure 4-1.* In practice a project is allocated an article number.



| article | | | |
|---|---|---|---|
| article_id | Integer | NN (PK) | |
| article_type | Character varying(25) | (FK) | (IX1) |
| sensor_specification_id | Integer | (FK) | (IX2) |
| replaced_by | Integer | (FK) | (IX3) |
| vendor_id | Integer | (FK) | (IX4) |

IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)

| project | | | |
|---|---|---|---|
| project_id | Integer | NN (PK) | |
| production_bom_no | Integer | (FK) | (IX1) |

IX_Relationship55 (IX1)

*Figure 4-1: Article - project relationship*

## 4.2 Article type – article

Each record in the entity *article_type* can be connected to one or many articles. The relationship is shown in *Figure 4-2*.

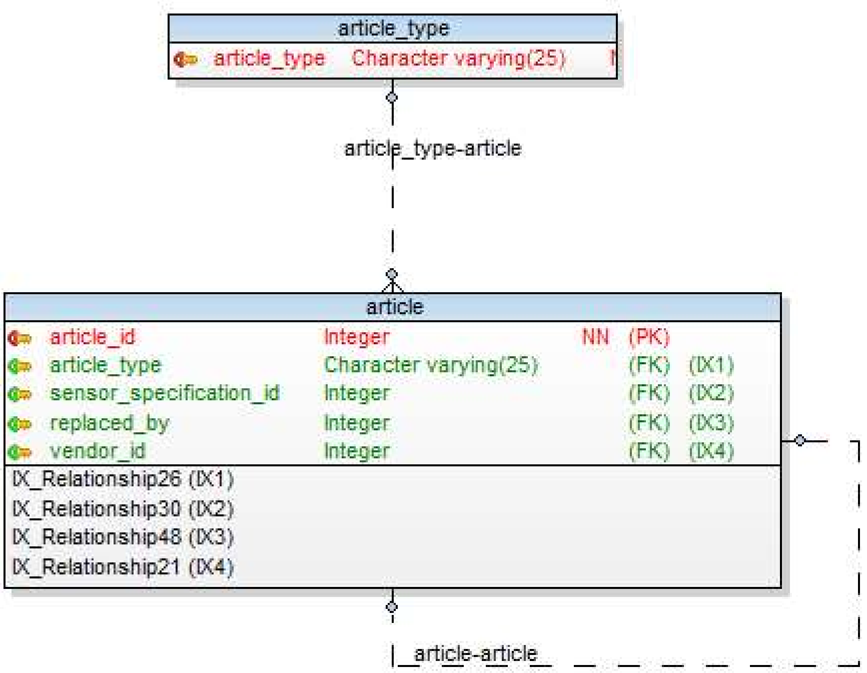

article_type
article_type Character varying(25) N
article_type-article
article
article_id Integer NN (PK)
article_type Character varying(25) (FK) (IX1)
sensor_specification_id Integer (FK) (IX2)
replaced_by Integer (FK) (IX3)
vendor_id Integer (FK) (IX4)
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
article-article

*Figure 4-2: article type - article relationship*

## 4.3 Article – article

Each record in *article* can be connected to one other record in article shown in *Figure 4-3.* An article may become obsolete and replace by a new article record. A connecting is designed to be able to trace such replacements.

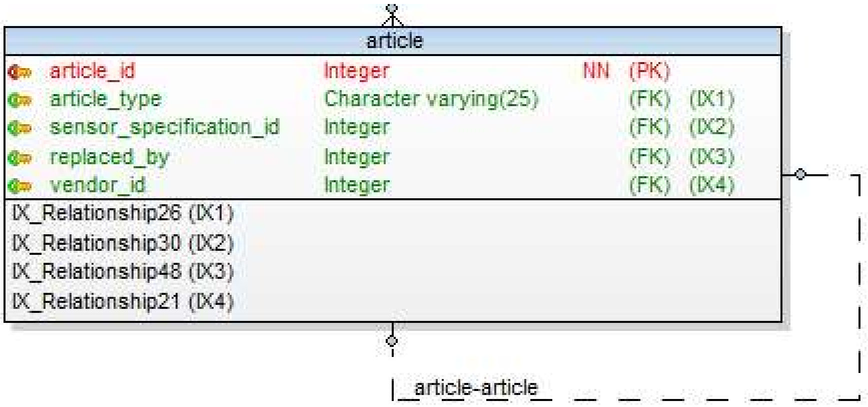
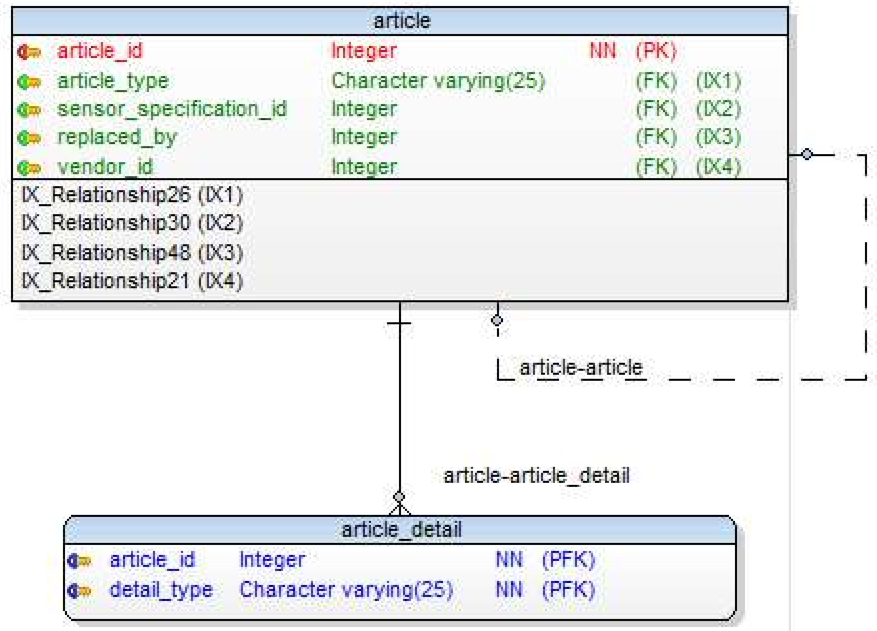

article
article_id Integer NN (PK)
article_type Character varying(25) (FK) (IX1)
sensor_specification_id Integer (FK) (IX2)
replaced_by Integer (FK) (IX3)
vendor_id Integer (FK) (IX4)
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
article-article

*Figure 4-3: Article - Article relationship*

## 4.4 Article – article detail

Each record in the entity *article* can be connected to one or many records in *article_detail* shown in *Figure 4-4.* The relationship is *identifying* because the foreign key *article_id* is part of *article_detail*'s primary key.



article

| | | | | |
|---|---|---|---|---|
| article_id | Integer | NN | (PK) | |
| article_type | Character varying(25) | | (FK) | (IX1) |
| sensor_specification_id | Integer | | (FK) | (IX2) |
| replaced_by | Integer | | (FK) | (IX3) |
| vendor_id | Integer | | (FK) | (IX4) |

IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)

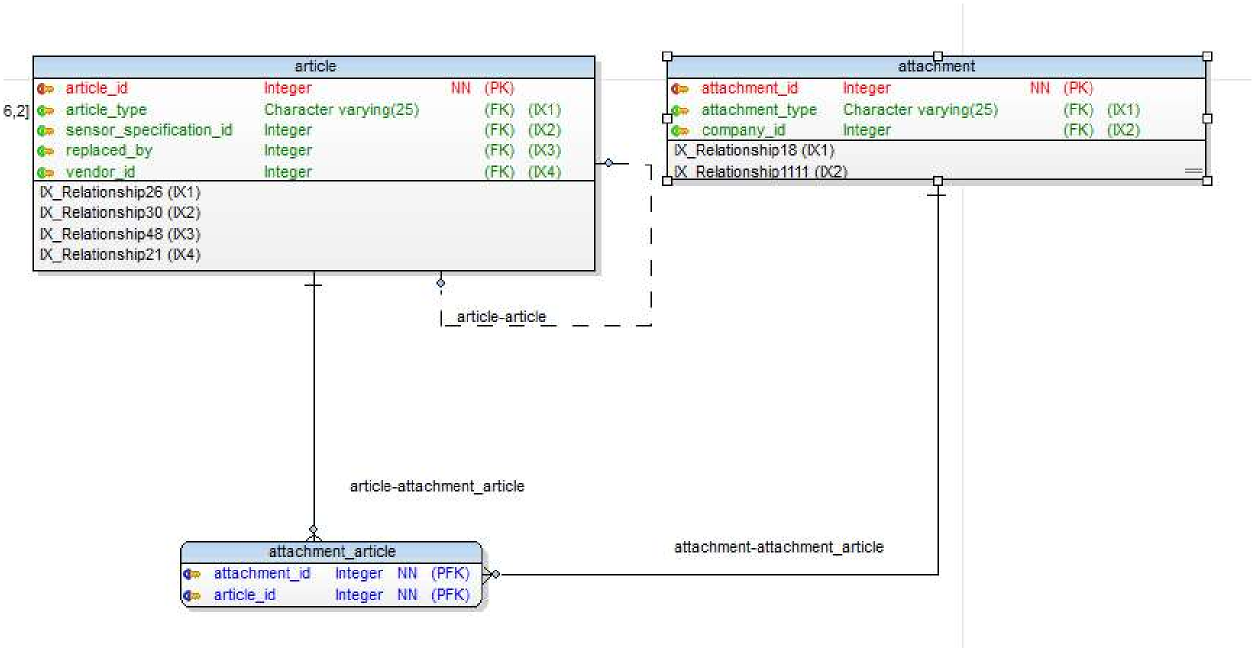
article_detail

| | | | |
|---|---|---|---|
| article_id | Integer | NN | (PFK) |
| detail_type | Character varying(25) | NN | (PFK) |

*Figure 4-4: Article - article detail relationship*

## 4.5 Article – attachment article – attachment

The relationship between the entities *article, attachment_article* and *attachment* are shown in *Figure 4-5.* The relationship between entities *article* and *attachment* is defined as a many-to-many relations. To accomplish this, a connecting entity *attachment_article* is designed. As a result, each record in *article* can be connected to many records in *attachment* and the opposite.



article

| | | | |
|---|---|---|---|
| article_id | Integer | NN | (PK) |
| article_type | Character varying(25) | | (FK) (IX1) |
| sensor_specification_id | Integer | | (FK) (IX2) |
| replaced_by | Integer | | (FK) (IX3) |
| vendor_id | Integer | | (FK) (IX4) |

IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)

attachment

| | | | |
|---|---|---|---|
| attachment_id | Integer | NN | (PK) |
| attachment_type | Character varying(25) | | (FK) (IX1) |
| company_id | Integer | | (FK) (IX2) |

IX_Relationship18 (IX1)
IX_Relationship1111 (IX2)

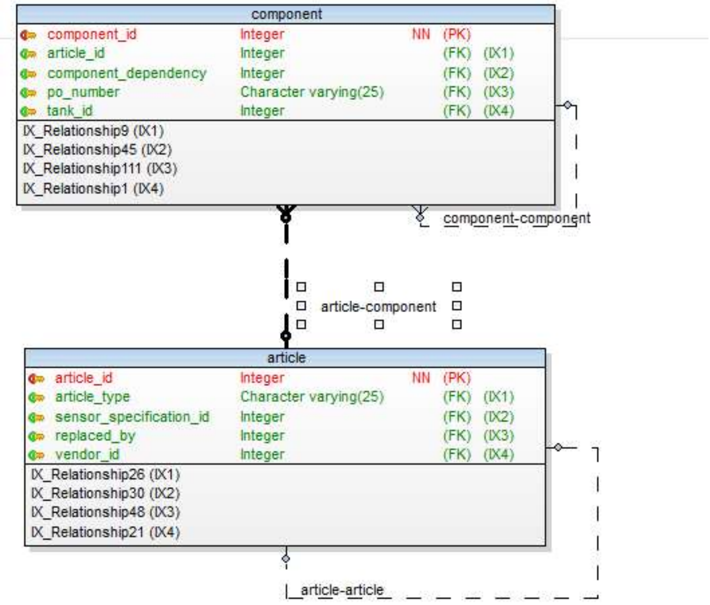
attachment_article

| | | | |
|---|---|---|---|
| attachment_id | Integer | NN | (PFK) |
| article_id | Integer | NN | (PFK) |

Figure 4-5: Article - attachment relationship

## 4.6 Article – component

The relationship between the entities *article* and *component* is shown in *Figure 4-6.* Each record in *article* can be connected to one or many components.



component

| | | | |
|---|---|---|---|
| component_id | Integer | NN | (PK) |
| article_id | Integer | | (FK) (IX1) |
| component_dependency | Integer | | (FK) (IX2) |
| po_number | Character varying(25) | | (FK) (IX3) |
| tank_id | Integer | | (FK) (IX4) |

IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)

article

| | | | |
|---|---|---|---|
| article_id | Integer | NN | (PK) |
| article_type | Character varying(25) | | (FK) (IX1) |
| sensor_specification_id | Integer | | (FK) (IX2) |
| replaced_by | Integer | | (FK) (IX3) |
| vendor_id | Integer | | (FK) (IX4) |

IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)

Figure 4-6: Article - component relationship

## 4.7 Attachment revision – attachment revision detail

The relationship between the entities *attachment_revision* and *attachment_revision_detail* is shown in *Figure 4-7.* Each record in *attachment_revision* can be connected to one or many revision details by a one-to-many relationship. The relationship is identifying because the foreign key from *attachment_revision* is part of *attachment_revision_detail's* primary key.

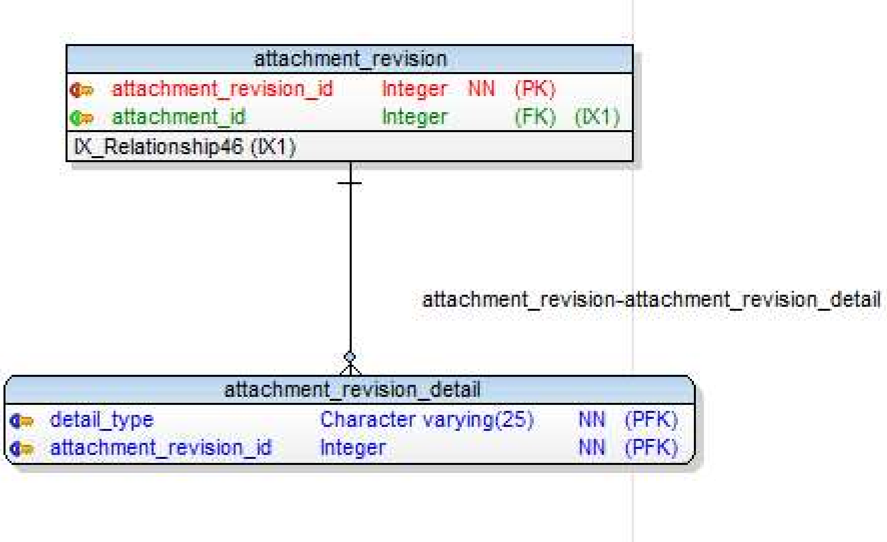

attachment_revision
attachment_revision_id Integer NN (PK)
attachment_id Integer (FK) (IX1)
IX_Relationship46 (IX1)
attachment_revision-attachment_revision_detail
attachment_revision_detail
detail_type Character varying(25) NN (PFK)
attachment_revision_id Integer NN (PFK)

*Figure 4-7: Attachment revision - attachment revision detail relationship*

## 4.8 Attachment type – attachment

The relationship between the entity *attachment_type* and *attachment* are shown in *Figure 4-8.* Each record in *attachment_type* can be connected to one or many records in *attachment.*

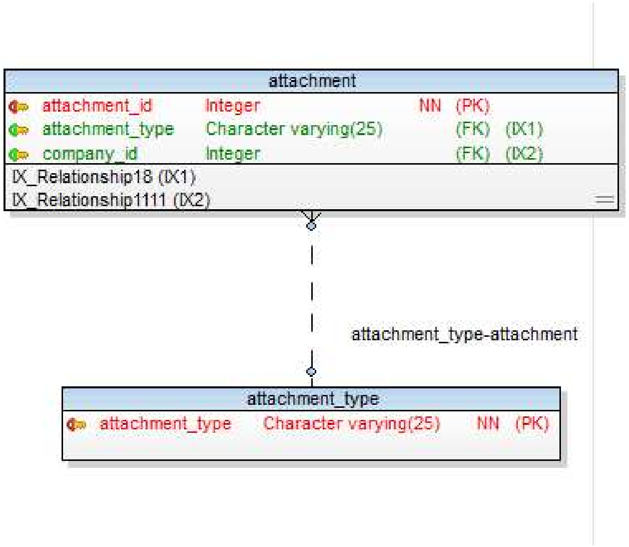
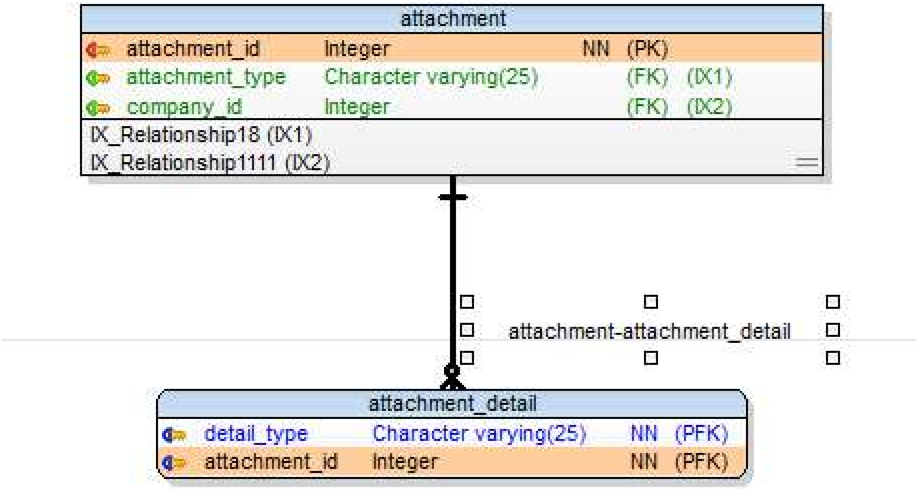

attachment
attachment_id Integer NN (PK)
attachment_type Character varying(25) (FK) (IX1)
company_id Integer (FK) (IX2)
IX_Relationship18 (IX1)
IX_Relationship1111 (IX2)
attachment_type-attachment
attachment_type
attachment_type Character varying(25) NN (PK)

*Figure 4-8: attachment type - attachment relationship*

## 4.9 Attachment – attachment detail

The relationship between the entity *attachment* and *attachment_detail* is shown in Figure 4-9. Each record in *attachment* can be connected to one or many *attachment_details.* The relationship is identifying because the foreign key *attachment_id* is part of the primary key of *attachment_detail.*



| attachment | | | | |
|---|---|---|---|---|
| attachment_id | Integer | NN | (PK) | |
| attachment_type | Character varying(25) | | (FK) | (IX1) |
| company_id | Integer | | (FK) | (IX2) |
| IX_Relationship18 (IX1) | | | | |
| IX_Relationship1111 (IX2) | | | | |

| attachment_detail | | | |
|---|---|---|---|
| detail_type | Character varying(25) | NN | (PFK) |
| attachment_id | Integer | NN | (PFK) |

*Figure 4-9: Attachment - attachment detail relationships*

## 4.10 Attachment – attachment revision

The relationship between the entity *attachment* and *attachment_revison* is shown in Figure 4-10. Each record in *attachment* can be connected to one or many records in *attachment_revision*.

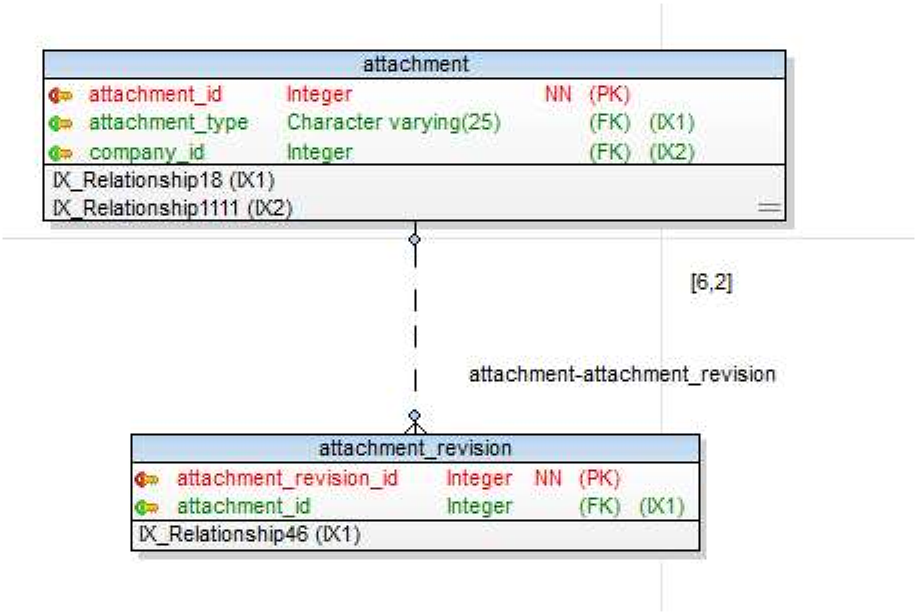

attachment
attachment_id Integer NN (PK)
attachment_type Character varying(25) (FK) (IX1)
company_id Integer (FK) (IX2)
IX_Relationship18 (IX1)
IX_Relationship1111 (IX2)
[6,2]
attachment-attachment_revision
attachment_revision
attachment_revision_id Integer NN (PK)
attachment_id Integer (FK) (IX1)
IX_Relationship46 (IX1)

*Figure 4-10: Attachment - attachment revision relationship*

## 4.11 Company – article

The relationship between the entities *company* and *article* is shown in Figure 4-11. Each company can be connected to one more record in article. An article can only be connected to one company. Each article has one vendor.

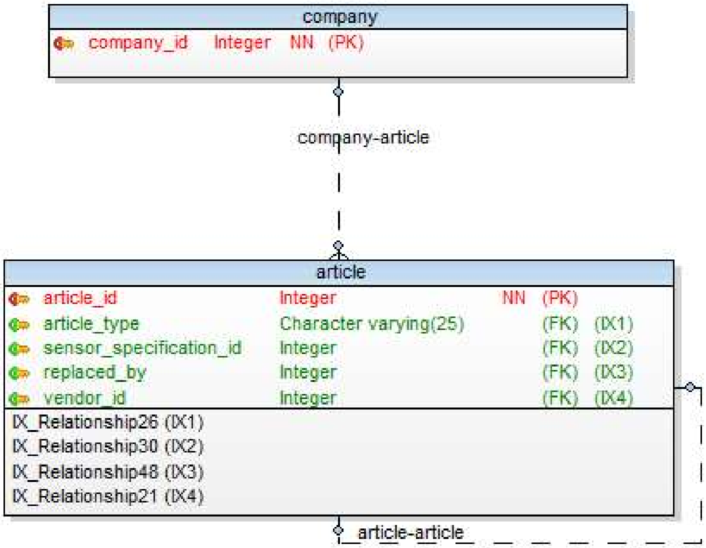

company
company_id Integer NN (PK)
company-article
article
article_id Integer NN (PK)
article_type Character varying(25) (FK) (IX1)
sensor_specification_id Integer (FK) (IX2)
replaced_by Integer (FK) (IX3)
vendor_id Integer (FK) (IX4)
IX_Relationship26 (IX1)
IX_Relationship30 (IX2)
IX_Relationship48 (IX3)
IX_Relationship21 (IX4)
article-article

*Figure 4-11: Company - article relationship*

## 4.12 Company – attachment

The relationship between *company* and *attachment* is shown in Figure 4-12. Each company can be connected to one or more records in *attachment*. An attachment record can only be connected to one company.

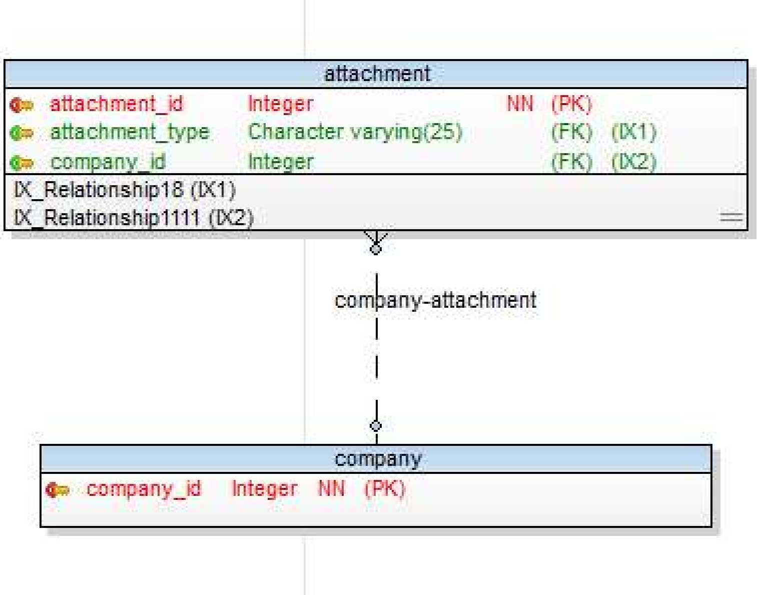

attachment
attachment_id Integer NN (PK)
attachment_type Character varying(25) (FK) (IX1)
company_id Integer (FK) (IX2)
IX_Relationship18 (IX1)
IX_Relationship1111 (IX2)
company-attachment
company
company_id Integer NN (PK)

*Figure 4-12: Company - attachment relationship*

## 4.13 Company – contact person

The relationship between *company* and *contact_person* is shown in *Figure 4-13.* Each record in *company* can have one or many records in *contact_person,* but a contact person can only be connected to one company.

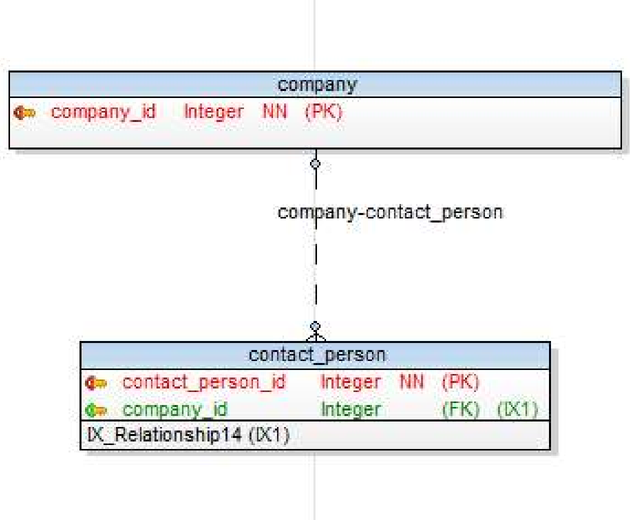

company
company_id Integer NN (PK)
company-contact_person
contact_person
contact_person_id Integer NN (PK)
company_id Integer (FK) (IX1)
IX_Relationship14 (IX1)

*Figure 4-13: Company - contact person relationship*

## 4.14 Company – purchase order 1 and 2

The relationships between *company* and *purchase_order* is shown in Figure 4-14. The reason behind two one-to-many relationships between company and purchase order is that one single record in *purchase_order* is connected to two companies, a production company and a customer company.

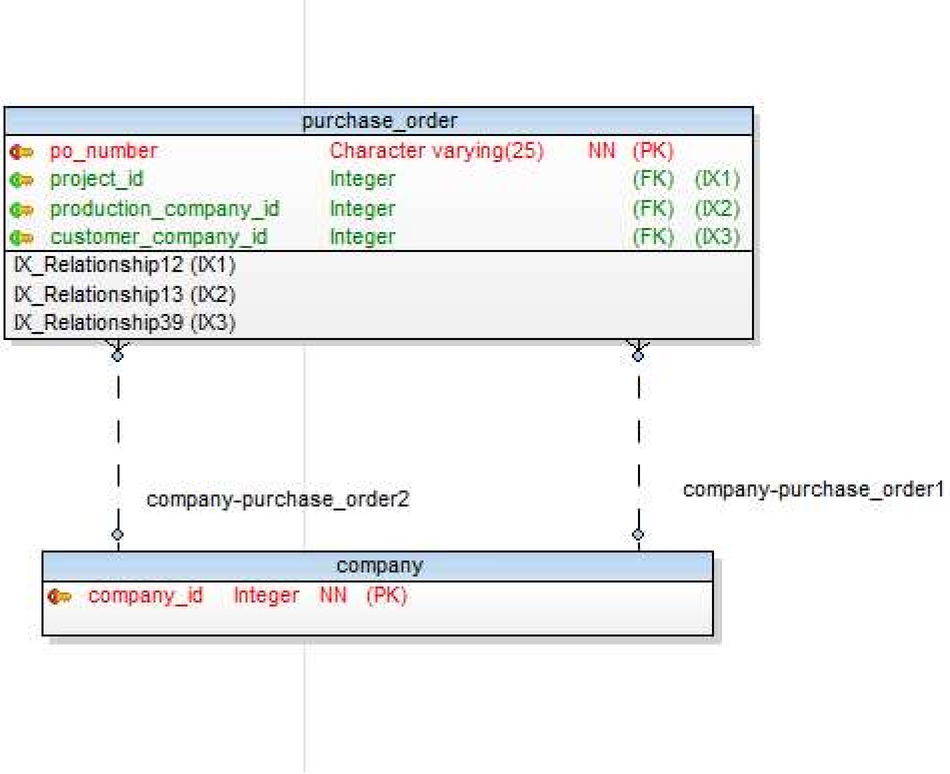

purchase_order
po_number Character varying(25) NN (PK)
project_id Integer (FK) (IX1)
production_company_id Integer (FK) (IX2)
customer_company_id Integer (FK) (IX3)
IX_Relationship12 (IX1)
IX_Relationship13 (IX2)
IX_Relationship39 (IX3)
company-purchase_order2
company-purchase_order1
company
company_id Integer NN (PK)

*Figure 4-14: Company - Purchase order relationships*

## 4.15 Component – component

The entity *component* has a one-to-many relationship with itself shown in Figure 4-15. A component can be part of complete product. The relationship is designed to ensure a connection between several components in a product. E.g. radar and radar electronics both have their own serial numbers but are sold as a complete product.

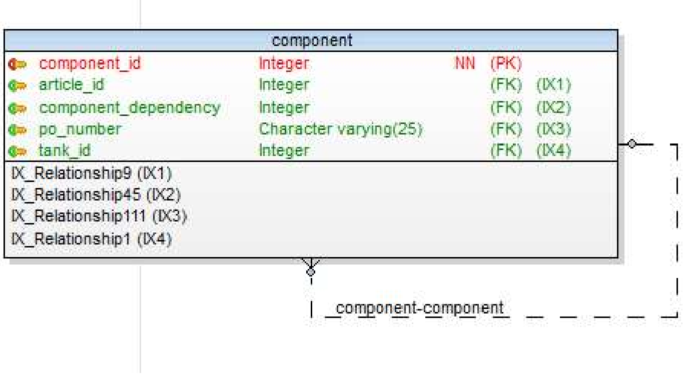

component
component_id Integer NN (PK)
article_id Integer (FK) (IX1)
component_dependency Integer (FK) (IX2)
po_number Character varying(25) (FK) (IX3)
tank_id Integer (FK) (IX4)
IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)
component-component

*Figure 4-15: Component - component relationship*

## 4.16 Component – component detail

The relationship between the entity *component* and *component_detail* is shown in Figure 4-16. Each record in component can have one or many component details. The relationship is identifying because the foreign key *component_id* is part of *component_detail's* primary key.

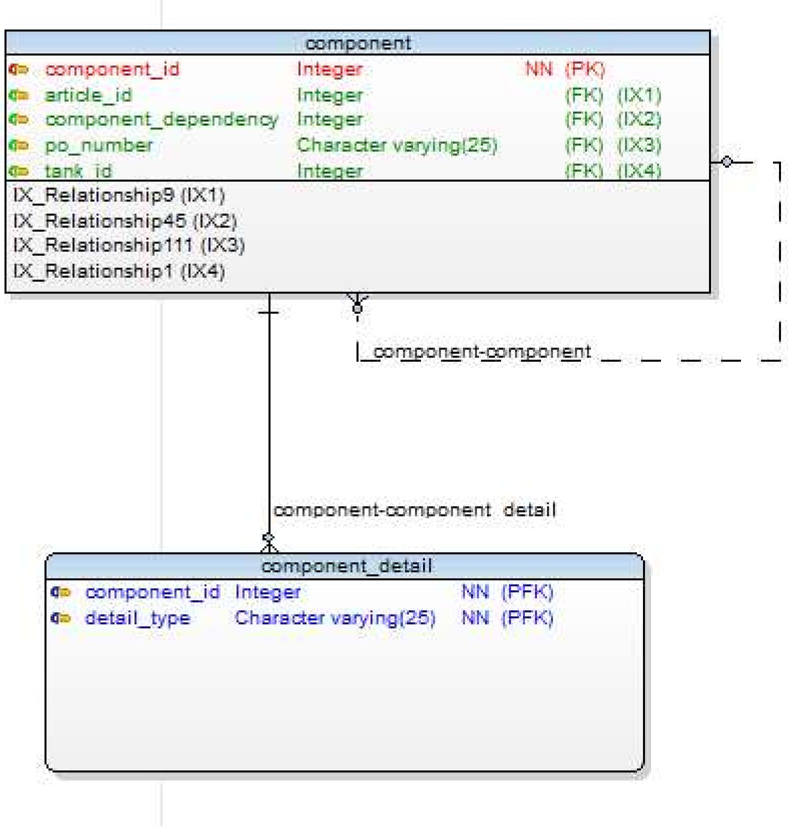

component
component_id Integer NN (PK)
article_id Integer (FK) (IX1)
component_dependency Integer (FK) (IX2)
po_number Character varying(25) (FK) (IX3)
tank_id Integer (FK) (IX4)
IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)
component-component
component-component detail
component_detail
component_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)

*Figure 4-16: component - component detail relationship*

## 4.17 Component – component error data

The relationship between *component* and *component_error_data* is shown in Figure 4-17. Each *component* record can have one or many connected records in *component_error_data.* A record in component error data can only be connected to one record in *component*.

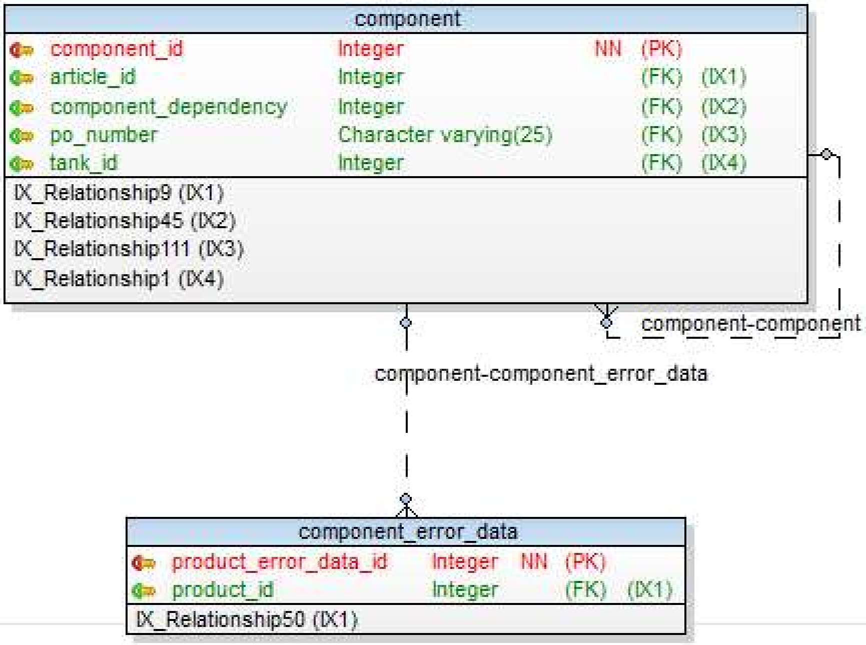

component
component_id Integer NN (PK)
article_id Integer (FK) (IX1)
component_dependency Integer (FK) (IX2)
po_number Character varying(25) (FK) (IX3)
tank_id Integer (FK) (IX4)
IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)
component-component
component-component_error_data
component_error_data
product_error_data_id Integer NN (PK)
product_id Integer (FK) (IX1)
IX_Relationship50 (IX1)

*Figure 4-17: component - component error data relationships*

## 4.18 Data type – detail type

The relationship between *data_type* and *detail_type* is shown in Figure 4-18. Each record in data type can be connected to one or many records in detail type.

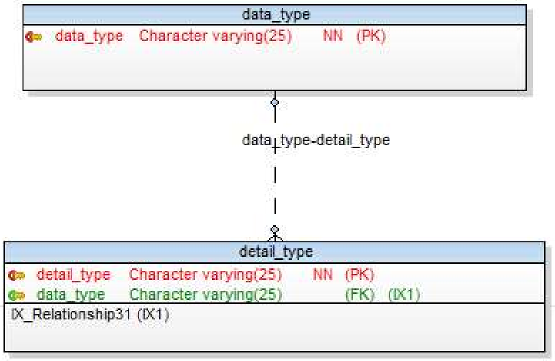
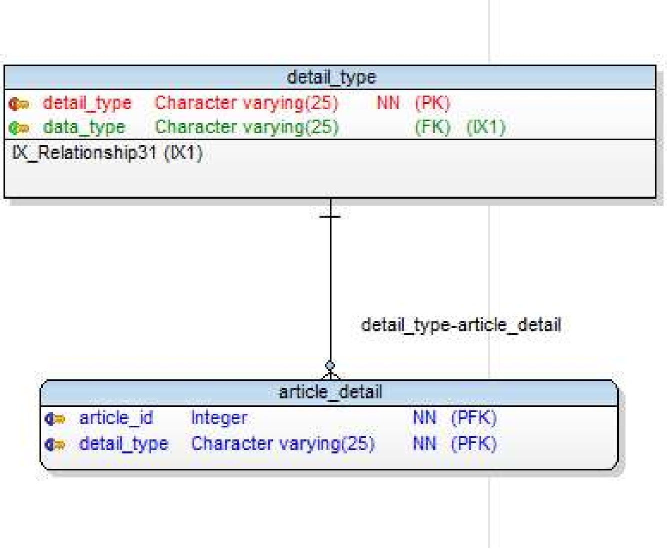

data_type
data_type Character varying(25) NN (PK)
data_type-detail_type
detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
IX_Relationship31 (IX1)

*Figure 4-18: Data type - detail type relationship*

## 4.19 Detail type – article detail

The relationship between *detail_type* and *article_detail* is shown in Figure 4-19. Each record in detail type can be connected to one or many records in article detail. The relationship is identifying because the foreign key *detail_type* is part of *article_detail's* primary key.

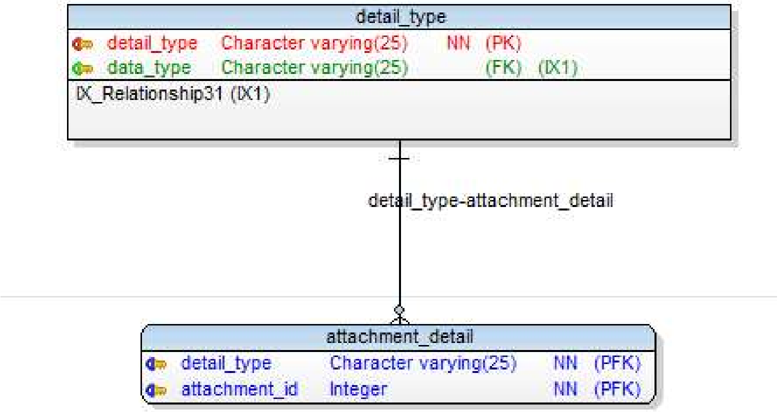


detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
IX_Relationship31 (IX1)
detail_type-article_detail
article_detail
article_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)

*Figure 4-19: Detail type - article detail relationship*

## 4.20 Detail type – attachment detail

The relationship between *detail_type* and *attachment_detail* is shown in Figure 4-20Figure 4-19. Each record in detail type can be connected to one or many records in *attachment_detail.* The relationship is identifying because the foreign key *detail_type* is part of *attachment_detail's* primary key.



detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
IX_Relationship31 (IX1)
detail_type-attachment_detail
attachment_detail
detail_type Character varying(25) NN (PFK)
attachment_id Integer NN (PFK)

*Figure 4-20: Detail type - attachment detail relationship*

## 4.21 Detail type – attachment revision detail

The relationship between *detail_type* and *attachment_revision_detail* is shown in Figure 4-21. Each record in detail type can be connected to one or many records in *attachment_revision_detail*. The relationship is identifying because the foreign key *detail_type* is part of *attachment_revision_detail's* primary key.

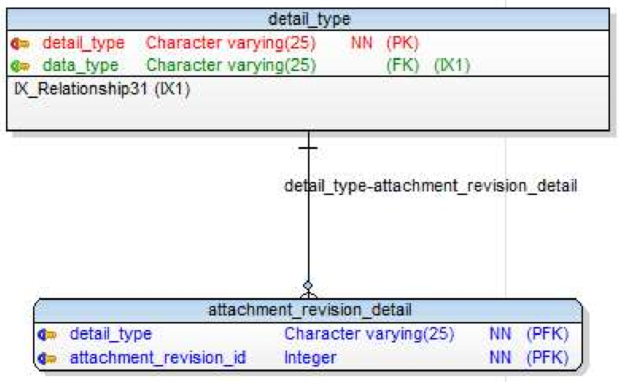

detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
IX_Relationship31 (IX1)
detail_type-attachment_revision_detail
attachment_revision_detail
detail_type Character varying(25) NN (PFK)
attachment_revision_id Integer NN (PFK)

*Figure 4-21: Detail type - attachment revision detail relationship*

## 4.22 Detail type – component detail

The relationship between *detail_type* and *component_detail* is shown in Figure 4-22. Each record in detail type can be connected to one or many records in *component_detail*. The relationship is identifying because the foreign key *detail_type* is part of *component_detail's* primary key.

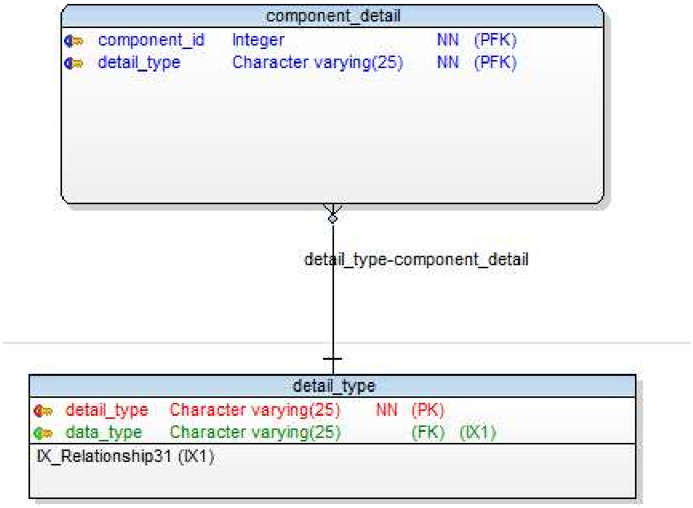

component_detail
component_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)
detail_type-component_detail
detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
IX_Relationship31 (IX1)

*Figure 4-22: Detail type - component relationship*

## 4.23 Detail type – sensor detail

The relationship between *detail_type* and *sensor_detail* is shown in Figure 4-23. Each record in detail type can be connected to one or many records in *sensor_detail*. The relationship is identifying because the foreign key *detail_type* is part of *sensor_detail's* primary key.

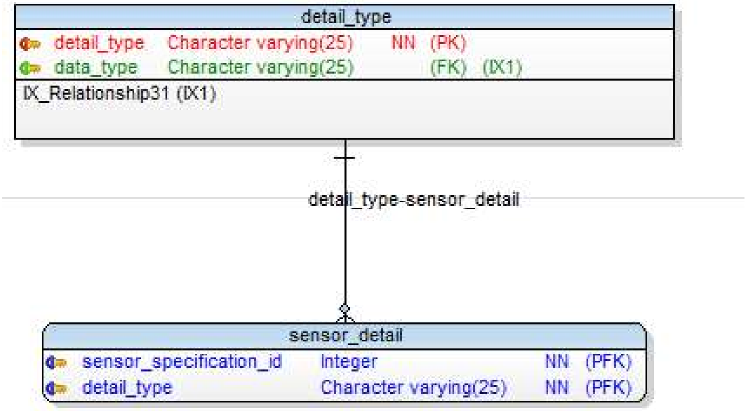

detail_type
detail_type Character varying(25) NN (PK)
data_type Character varying(25) (FK) (IX1)
IX_Relationship31 (IX1)
detail_type-sensor_detail
sensor_detail
sensor_specification_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)

*Figure 4-23: detail type – sensor detail relationship*

## 4.24 Project – purchase order

## 4.25 Project – tank

The relationship between *project* and *tank* is shown in Figure 4-24. Each record in *project* can be connected to one or more records in *tank.* A tank can only be connected to one project.

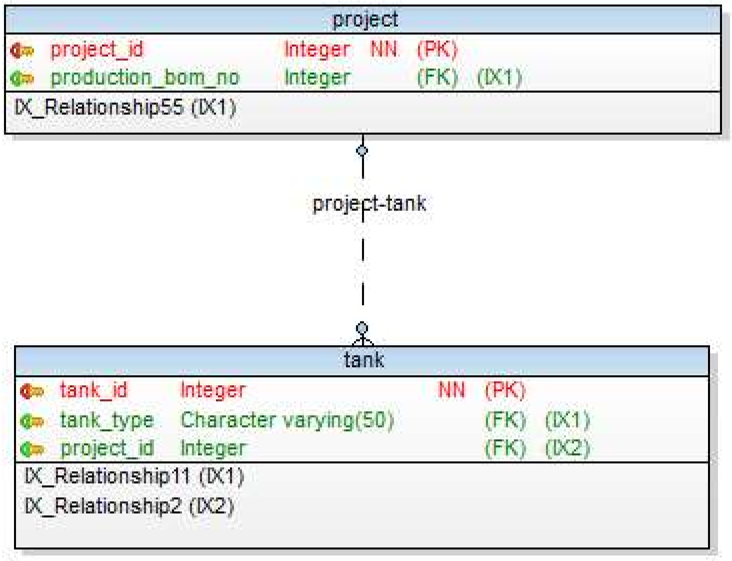

project
project_id Integer NN (PK)
production_bom_no Integer (FK) (IX1)
IX_Relationship55 (IX1)
project-tank
tank
tank_id Integer NN (PK)
tank_type Character varying(50) (FK) (IX1)
project_id Integer (FK) (IX2)
IX_Relationship11 (IX1)
IX_Relationship2 (IX2)

*Figure 4-24: Project - tank relationship*

## 4.26 Purchase order – component

The relationship between the entity *purchase_order* and *component* are shown in Figure 4-25. Each record in *purchase_order* can be connected to one or more records in *components.* A component can only be connected to one record in *purchase_order.*

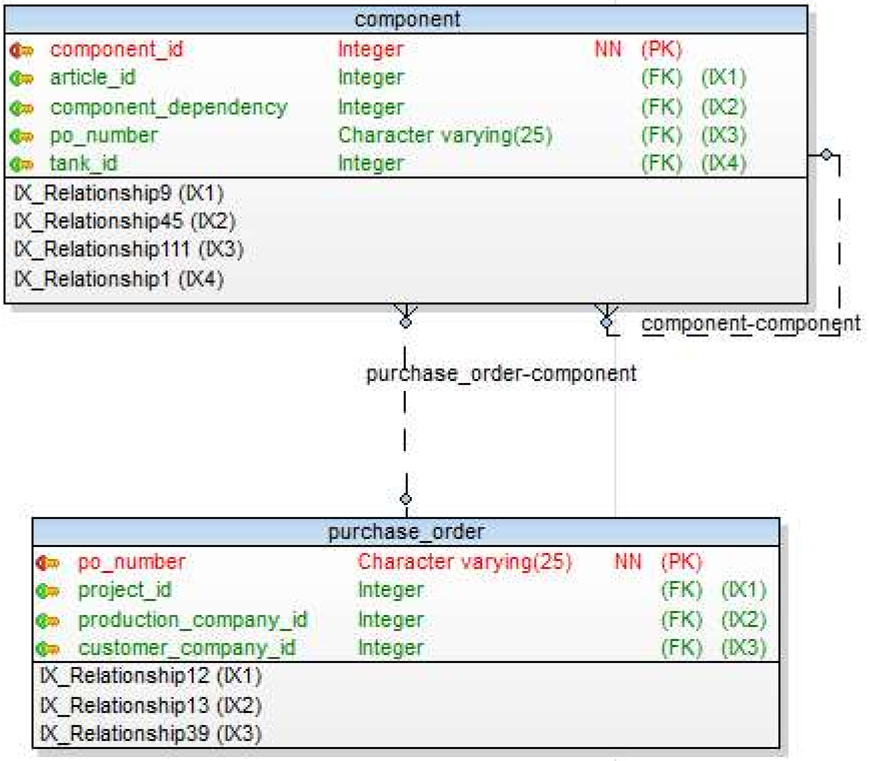

component
component_id Integer NN (PK)
article_id Integer (FK) (IX1)
component_dependency Integer (FK) (IX2)
po_number Character varying(25) (FK) (IX3)
tank_id Integer (FK) (IX4)
IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)
component-component
purchase_order-component
purchase_order
po_number Character varying(25) NN (PK)
project_id Integer (FK) (IX1)
production_company_id Integer (FK) (IX2)
customer_company_id Integer (FK) (IX3)
IX_Relationship12 (IX1)
IX_Relationship13 (IX2)
IX_Relationship39 (IX3)

*Figure 4-25: Purchase order - component relationship*

## 4.27 Sensor category – sensor specification

The relationship between *sensor_category* and *sensor_specification* is shown in Figure 4-26. A record in sensor category can be connected to one or more records in sensor specification. Each record in sensor specification can only be connected to one sensor category.

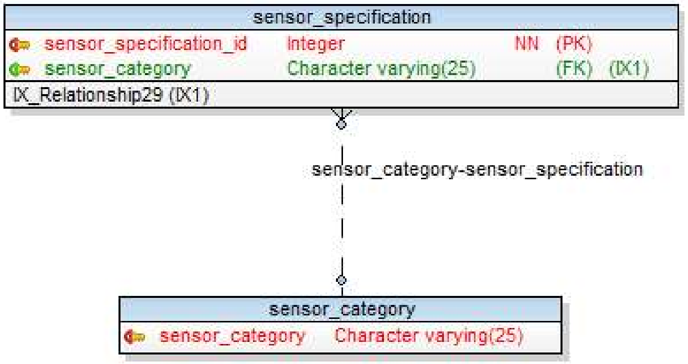

sensor_specification
sensor_specification_id Integer NN (PK)
sensor_category Character varying(25) (FK) (IX1)
IX_Relationship29 (IX1)
sensor_category-sensor_specification
sensor_category
sensor_category Character varying(25)

*Figure 4-26: Sensor category - sensor specification relationship*

## 4.28 Sensor specification – sensor detail

The relationship between *sensor_specification* and *sensor_detail* is shown in Figure 4-27. Each record in sensor specification can be connected to one or more records in sensor details. The relationship is identifying because of the foreign key *sensor_specification_id* being part of *sensor_detail's* primary key.

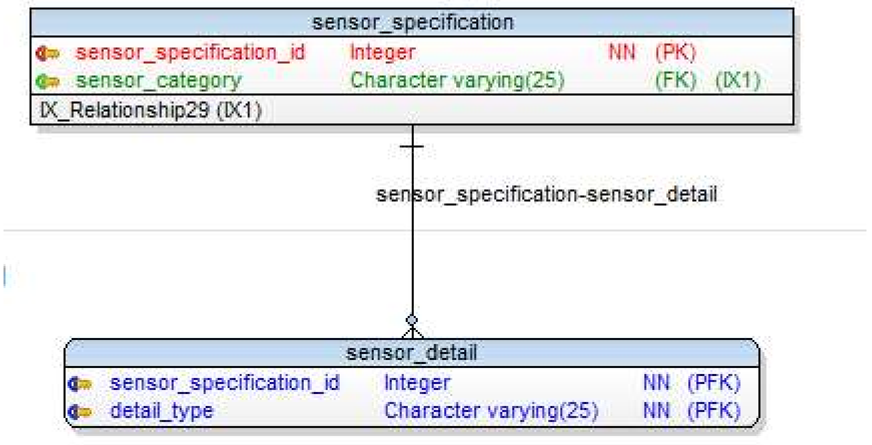

sensor_specification
sensor_specification_id Integer NN (PK)
sensor_category Character varying(25) (FK) (IX1)
IX_Relationship29 (IX1)
sensor_specification-sensor_detail
sensor_detail
sensor_specification_id Integer NN (PFK)
detail_type Character varying(25) NN (PFK)

*Figure 4-27: Sensor specification - sensor detail relationship*

## 4.29 Tank type – tank

The relationship between *tank_type* and *tank* are shown in Figure 4-28. Each record in tank type can be connected to one or many records in tank. A single record in tank can only be connected to one tank type.

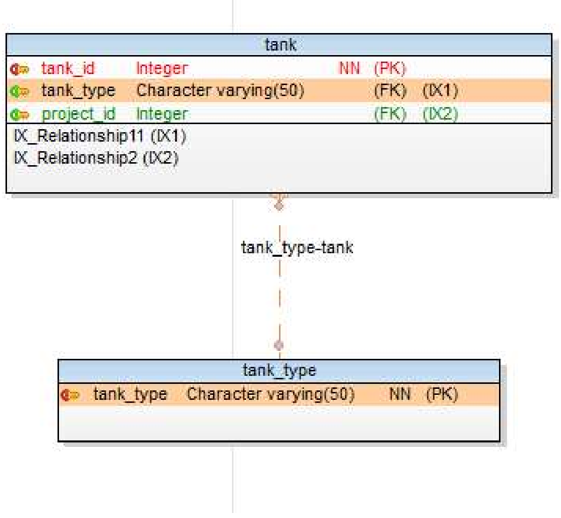
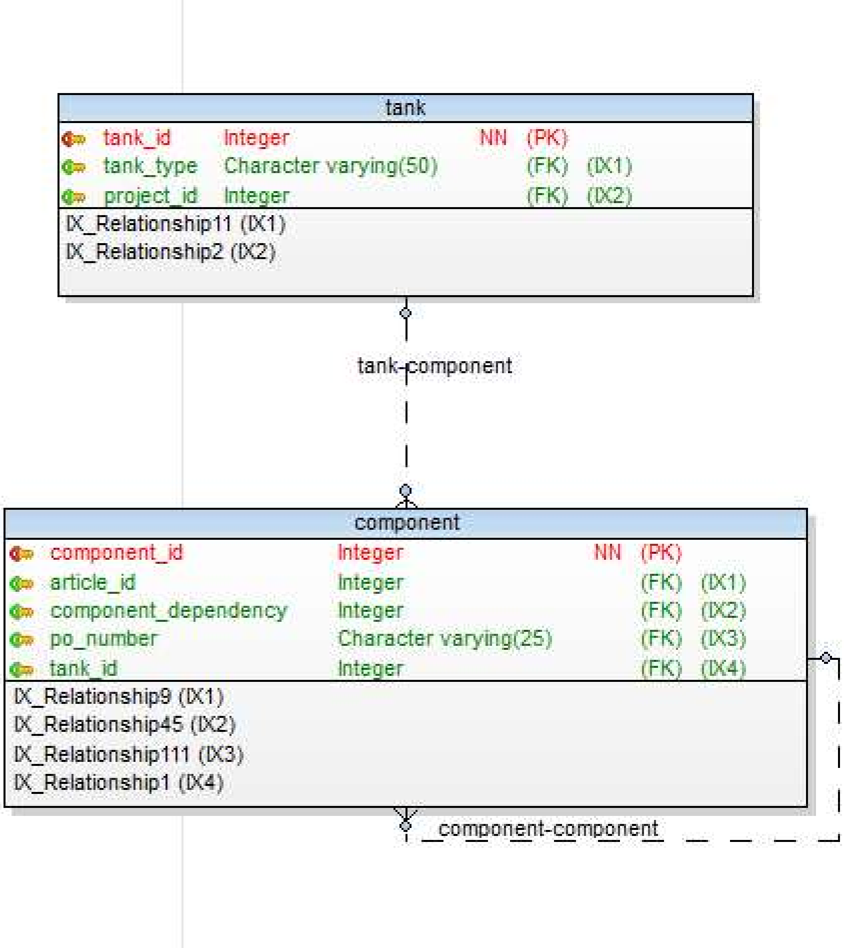

tank
tank_id Integer NN (PK)
tank_type Character varying(50) (FK) (IX1)
project_id Integer (FK) (IX2)
IX_Relationship11 (IX1)
IX_Relationship2 (IX2)
tank_type-tank
tank_type
tank_type Character varying(50) NN (PK)

*Figure 4-28: Tank type - tank relationships*

## 4.30 Tank – component

The relationships between *tank* and *component* is shown in Figure 4-29. Each record in tank can be connected to one or many components. A single record in component can only be connected to one record in tank. This design is done because several components can be installed in a tank.



tank

| | | | | |
|---|---|---|---|---|
| tank_id | Integer | NN | (PK) | |
| tank_type | Character varying(50) | | (FK) | (IX1) |
| project_id | Integer | | (FK) | (IX2) |

IX_Relationship11 (IX1)
IX_Relationship2 (IX2)

component

| | | | | |
|---|---|---|---|---|
| component_id | Integer | NN | (PK) | |
| article_id | Integer | | (FK) | (IX1) |
| component_dependency | Integer | | (FK) | (IX2) |
| po_number | Character varying(25) | | (FK) | (IX3) |
| tank_id | Integer | | (FK) | (IX4) |

IX_Relationship9 (IX1)
IX_Relationship45 (IX2)
IX_Relationship111 (IX3)
IX_Relationship1 (IX4)

*Figure 4-29: tank - component relationship*

**Appendix E**

| Table | attachment detail | component detail | Old names if changed |
|---|---|---|---|
| | Revision | Address | Adresse |
| | | Highrange | |
| | | Dallas Id | |
| | | Calibration mA | Set of Sen:Fro 0,  Sen:Fro 40,  Sen:Fro 60,   Sen:Fro 100 |
| | | Lowrange | |
| | | Production Batch | Prod Batch |
| | | COP8 Revision | |
| | | Gain | |
| | | Gal 2 Revision | |
| | | Gal Revision | |
| | | Nl1 | |
| | | Nl2 | |
| | | Null | |
| | | P906 Calibration mA | Set of Sen:Fro 0, Sen:Fro 0 G1, Sen:Fro 0 G2, Sen:Fro 0 G3, Sen:Fro 40, Sen:Fro 40 G1, Sen:Fro 40 G2, Sen:Fro 40 G3, Sen:Fro 60, Sen:Fro 60 G1, Sen:Fro 60 G2,Sen:Fro 60 G3, Sen:Fro 100, Sen:Fro 100 G1, Sen:Fro 100 G2, Sen:Fro 100 G3 |
| | | P906 Calibration1 mV | Set of Sen:Temp1,Sen:Temp Ohm 1, Sen:Fro1 0, Sen:Fro1 10, Sen:Fro1 20, Sen:Fro1 30, Sen:Fro1 40, Sen:Fro1 50, Sen:Fro1 60, Sen:Fro1 70, Sen:Fro1 80, Sen:Fro1 90, Sen:Fro1 100 |
| | | P906 Calibration2 mV | Set of Sen:Temp2,Sen:Temp Ohm 2, Sen:Fro2 0, Sen:Fro2 10, Sen:Fro2 20, Sen:Fro2 30, Sen:Fro2 40, Sen:Fro2 50, Sen:Fro2 60, Sen:Fro2 70, Sen:Fro2 80, Sen:Fro2 90, Sen:Fro2 100 |
| | | P906 Calibration3 mV | Set of Sen:Temp3,Sen:Temp Ohm 3, Sen:Fro3 0, Sen:Fro3 10, Sen:Fro3 20, Sen:Fro3 30, Sen:Fro3 40, Sen:Fro3 50, Sen:Fro3 60, Sen:Fro3 70, Sen:Fro3 80, Sen:Fro3 90, Sen:Fro3 100 |
| | | P906 Calibration4 mV | Set of Sen:Temp4,Sen:Temp Ohm 4, Sen:Fro4 0, Sen:Fro4 10, Sen:Fro4 20, Sen:Fro4 30, Sen:Fro4 40, Sen:Fro4 50, Sen:Fro4 60, Sen:Fro4 70, Sen:Fro4 80, Sen:Fro4 90, Sen:Fro4 100 |
| | | Power 24V | |
| | | Power 5V | |
| | | Revision | Revisjon |
| | | Position | Inclinometernummer, Tank:LGP, Tank: bottom, Tank: Mid, Tank:Top |
| | | Single | |
| | | Software version | |
| | | Tag number | |
| | | Tss | |
| | | Tzs | |
| | | Used | |
| | | Display | |
| | | BIOS Revision | |
| | | Cable length | Kabellengde |
| | | Mariner Approved | |
| | | Network card | |
| | | Connection protocol | |
| | | Connector | |
| | | CPU | |
| | | Housing | |
| | | Length High | Length 1 |
| | | Length Overflow | Length 2 |
| | | Flash memory | |
| | | HDD | |
| | | Welded | |
| | | Ex | NIS |
| | | Ram | |
| | | Type | |
| | | Cabling | |

**Appendix F**

| | | What column separates? | If two, first column value | | Article No= | Column value | | Article No= | Column value | | Article No= | Column value | Comment |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Table | | | | | | | | | | | | | |
| ABU | | | | | A5002 | | | | | | | | |
| AWS | | | | | A5003 | | | | | | | | |
| INANDATA | | | | | A5006 | | | | | | | | |
| Inclinom | | | | | C1364 | Ship uses ABU | | C1536 | Ship uses TCU | | | | |
| LIDEC | | Single | | | A5021 | Single | | A5022 | double | | | | |
| LIDEC(HLOA) | | Type | | | C2105 | API TLA | | C2679 | API UTS | | | | Typo API TPA rectify to API TLA |
| LIDEC(HLOA) | | Type, Single | ASL400 | | C1652 | Double | | C1747 | Single | | | | Empty=double, "information missing, double/single) I memo |
| LIDEC(HLOA) | | Type, Single | LSG | | C2048 | Double | | C2053 | Single | | | | Empty=double, "information missing, double/single) I memo |
| LIDEC(HLOA) | | Type | | | C2602 | Vegaswing63 o.l. | | | | | | | Rename all to Vegaswing 63 |
| LIDEC(HLOA) | | Type | | | C2218 | Vegacap62 | | | | | | | |
| LIDEC(HLOA) | | Type | | | C2452 | Vegacap64 | | | | | | | |
| LIDEC | | Type | | | | M1311B | | | | | | | Insert as detail, Typo MB1311B rectify to M1311B |
| LIDEC | | type | | | C1581 | L92C-HT | | C1581 | L92C-BPR | | | | |
| Monitord | | type | | | | | | | | | | | Insert as detail |
| PC386DAT | | | | | A5011 | | | | | | | | |
| PCVGADAt | | | | | C1445 | | | | | | | | |
| SGCNVDAT | | serienummer | | | A5023 | <256 | | a5036 | 256 | | A5037 | 257 | |
| TCU | | | | | A5026 | | | | | | | | |
| TDU | | Revisjon | | | A5025 | <5 | | A5041 | 5 | | | | |
| TPC140 | | | | | A5012 | | | | | | | | |
| TPC196 | | Software version | | | A5013 | 1 | | A5040 | 2 | | | | |
| TPC350 | | | | | A5000 | | | | | | | | |
| Watering | | Kabelengde | | | C1532 | 20 | | C1531 | 5 | | C1560 | 30 | |
| ZB4R5 | | Nis | | | A5027 | 0 | | A5031 | 1 | | | | |
| ZB485DAT | | | | | A5007 | | | | | | | | |
| ZBANADAT | | | | | A5008 | | | | | | | | |
| ZBANAIIB | | | | | A5009 | | | | | | | | |
| ZBAR5 | | Nis | | | A5028 | 0 | | A5032 | 1 | | | | |
| ZBAR6 | | | | | A5035 | | | | | | | | |
| ZBAR7 | | NIs | | | A5038 | 0 | | A5039 | 1 | | | | |
| RS485DAT | | | | | A5005 | | | | | | | | |

# 1 Data migration guide

This guide will describe how to execute the data migration from Clarion to PostgreSQL. It uses SoftVelocity Database Scanner(SVDS), Excel, Devart for Excel and a executable file. An excel file (*Transformation.xlsx*) is also included which will be used for most of the migration. It's recommended to dedicate a computer for this as there will be numerous hours of waiting.

The general approach to this migration can be seen in Figure 1-1.

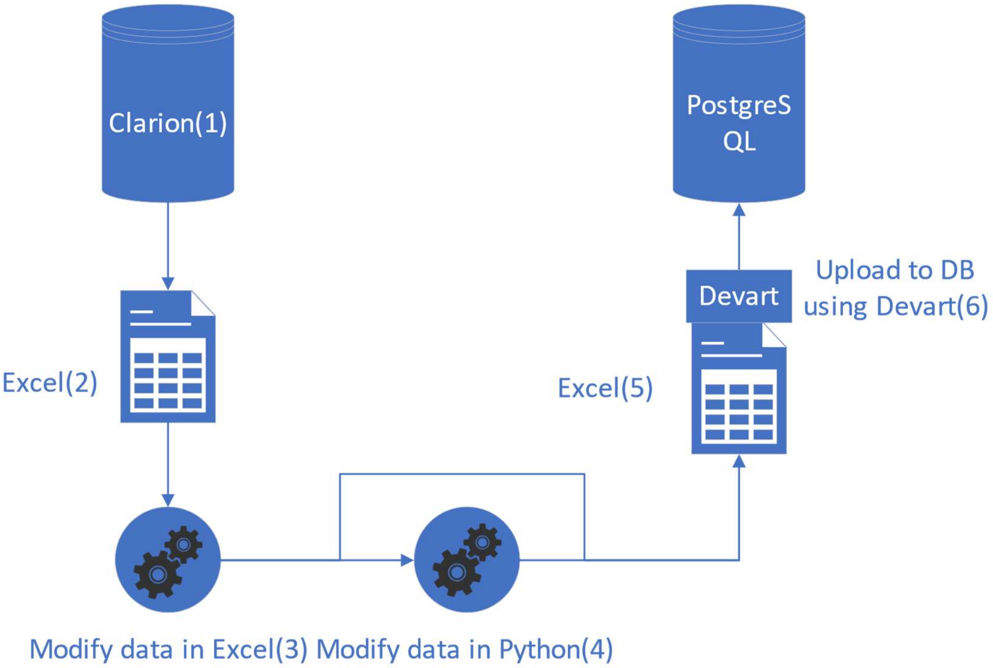


Figure 1-1 General approach to migration

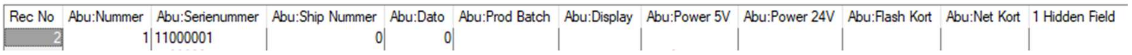
## Innhold

## 1.1 Export data from Clarion

Before exporting data, some modifications are required.

### 1.1.1 Hidden fields

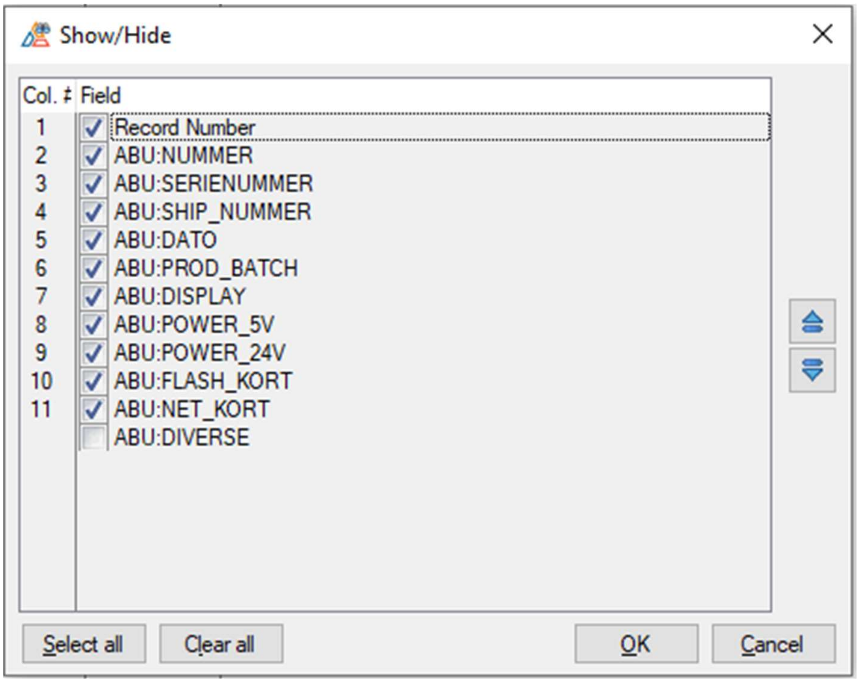
Some fields are hidden as shown in Figure 1-2.



| Rec No | Abu:Nummer | Abu:Serienummer | Abu:Ship Nummer | Abu:Dato | Abu:Prod Batch | Abu:Display | Abu:Power 5V | Abu:Power 24V | Abu:Flash Kort | Abu:Net Kort | 1 Hidden Field |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 11000001 | 0 | 0 | | | | | | | |

Figure 1-2 Hidden field at the end

To resolve this, double click on the Hidden field column. A window will pop up.



| Col. # | Field |
|---|---|
| 1 | ☑ Record Number |
| 2 | ☑ ABU:NUMMER |
| 3 | ☑ ABU:SERIENUMMER |
| 4 | ☑ ABU:SHIP_NUMMER |
| 5 | ☑ ABU:DATO |
| 6 | ☑ ABU:PROD_BATCH |
| 7 | ☑ ABU:DISPLAY |
| 8 | ☑ ABU:POWER_5V |
| 9 | ☑ ABU:POWER_24V |
| 10 | ☑ ABU:FLASH_KORT |
| 11 | ☑ ABU:NET_KORT |
| | □ ABU:DIVERSE |

Figure 1-3 Fields of the table

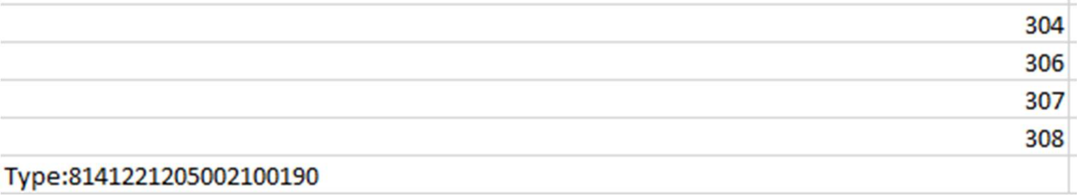
Figure 1-3 shows a hidden field DIVERSE in table ABU. Number of fields hidden may differ from table to table.

**In each table one should check for hidden fields!**

### 1.1.2 Corrupt data

#### 1.1.2.1 Misplaced data

It is recommended to keep data in columns as DIVERSE, although in some tables they may create trouble as it is uses multiline and will corrupt data integrity as can be seen in *Shipdata* table regarding *Eksternt Utstyr*. Figure 1-4 shows the column *rec no* in table *Shipdata*, it has misplaced data from column *eksternt utstyr* due to formatting faults. To resolve the issue, it is recommended to skip this column, another solution is to manually move this data to the correct column. SVDS(SoftVelocity Database Scanner) has been searched for solutions to this without luck.



| |
|---|
| 304 |
| 306 |
| 307 |
| 308 |
| Type:8141221205002100190 |

Figure 1-4 Misplaced data in Shipdata

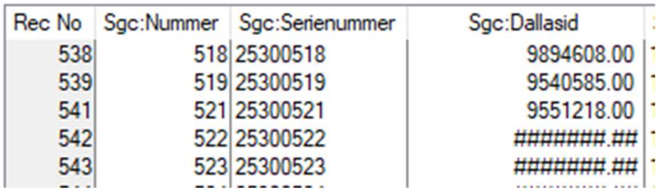
Additional places this has happened

- Feildata (Line 124)
- ABU(Line 172)

There may be more of this hidden in the data. This can only be checked during a full-scale test, which has not been done. New entries may contain this fault as well, which in that case, testing now wouldn't help. If encountered, values must either manually be rectified, or columns be hidden before exporting the data from the Clarion database. Choosing the latter will obviously mean data is lost.

### 1.1.2.2 Wrongly formatted data

*DallasID* is typically not readable in all records as can be seen in Figure 1-5. This also applies to some columns in *Sensordata.*



| Rec No | Sgc:Nummer | Sgc:Serienummer | Sgc:Dallasid |
|---|---|---|---|
| 538 | 518 | 25300518 | 9894608.00 |
| 539 | 519 | 25300519 | 9540585.00 |
| 541 | 521 | 25300521 | 9551218.00 |
| 542 | 522 | 25300522 | #######.## |
| 543 | 523 | 25300523 | #######.## |

Figure 1-5 Dallas ID not in readable format

This case applies to the following tables.

- Sensordata
- SGCNVDAT
- TPC140
- TPC196
- TPC350

To fix this issue, click in the column which has wrong format(1), then click column in the menu(2) and then format(3). See Figure 1-6 for visual instructions.

There are also occasions of numbers not being formatted correctly in Sensordata, there are occurances in several calibration data columns. As there are too many entries and columns to manually check, it's recommended to change all columns regarding calibration in addition to *DallasID.* There appears no obvious way to do this to more than one column at a time.

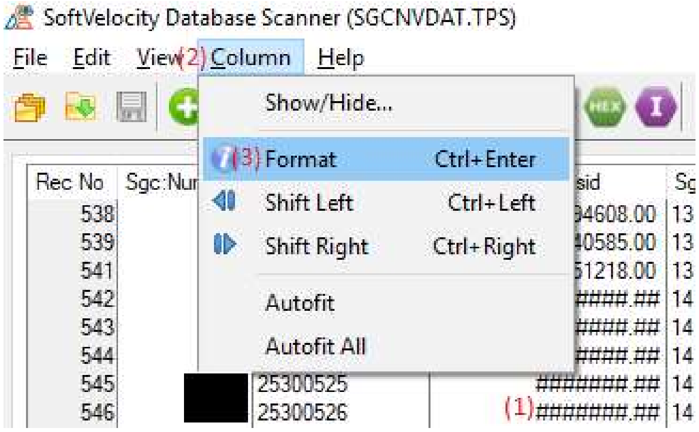
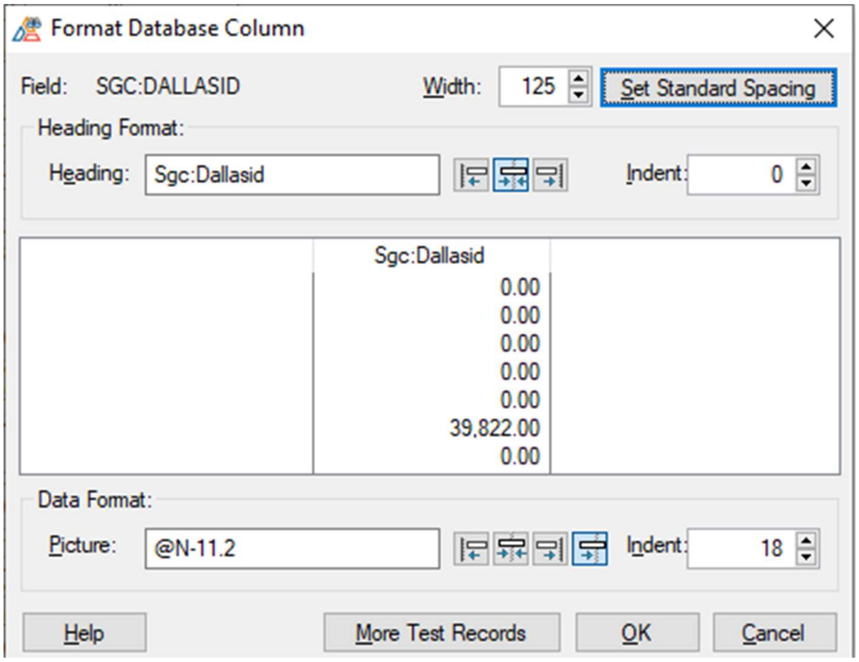


Figure 1-6 Finding the format window

Figure 1-7 shows the window that appears when last instruction is executed. Change Data format: Picture: @N-11.2 to @N-12 for *DallasId, @N-5 to @N-10 for calibration data*.


Format Database Column

Field: SGC:DALLASID Width: 125

Heading Format:

Heading: Sgc:Dallasid Indent: 0

Sgc:Dallasid
0.00
0.00
0.00
0.00
0.00
39,822.00
0.00

Data Format:

Picture: @N-11.2 Indent: 18

Figure 1-7 Format column window

## 1.1.3 Export data

When the preceding tasks have been finished, the exportation may start.

Following are instructions with pictures to visualize:

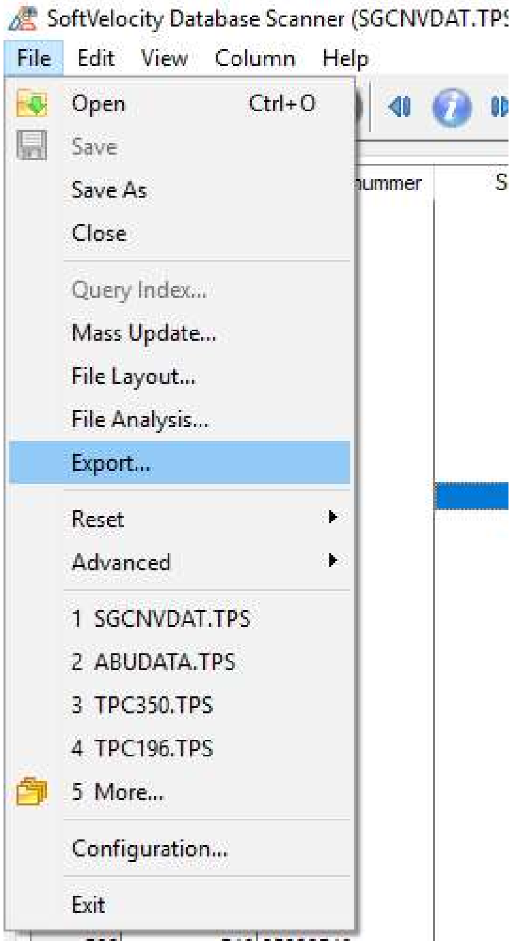
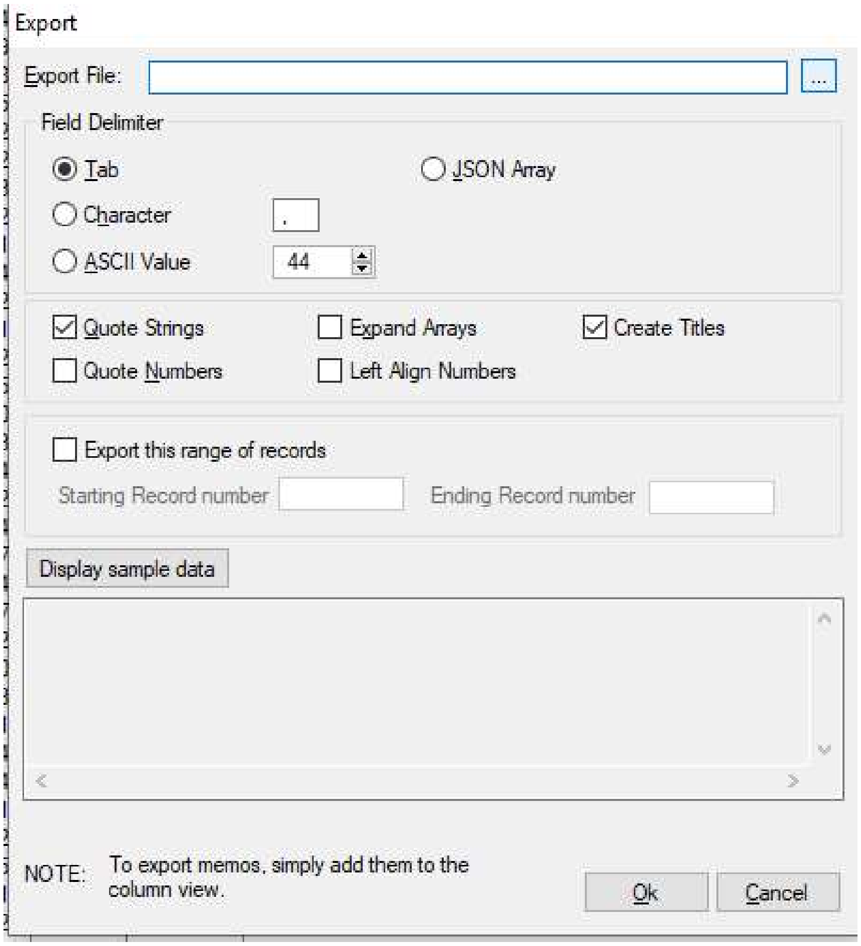


Figure 1-8 File->Export



Export

Export File:

Field Delimiter

● Tab ○ JSON Array

○ Character ,

○ ASCII Value 44

☑ Quote Strings □ Expand Arrays ☑ Create Titles

□ Quote Numbers □ Left Align Numbers

□ Export this range of records

Starting Record number Ending Record number

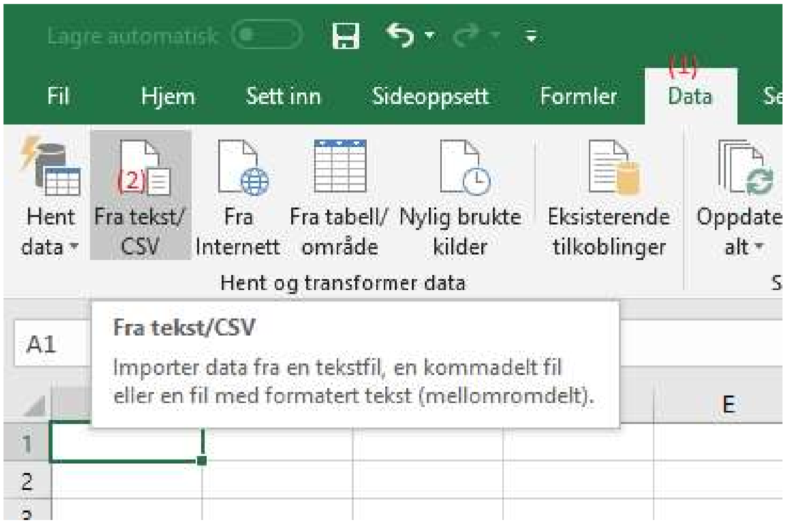
NOTE: To export memos, simply add them to the column view.

Figure 1-9 Choose where to export the file by clicking (…) and click Ok

## 1.2 Importing to Excel

It is recommended to change priority of Excel.exe in task manager(CTRL+Shift+ESC)->details->Right click on excel.exe-> set priority to high to increase resources to excel.

To import the files from Clarion, go to Data(1) then to Fra tekst/CSV(From text/CSV)(2). See Figure 1-10

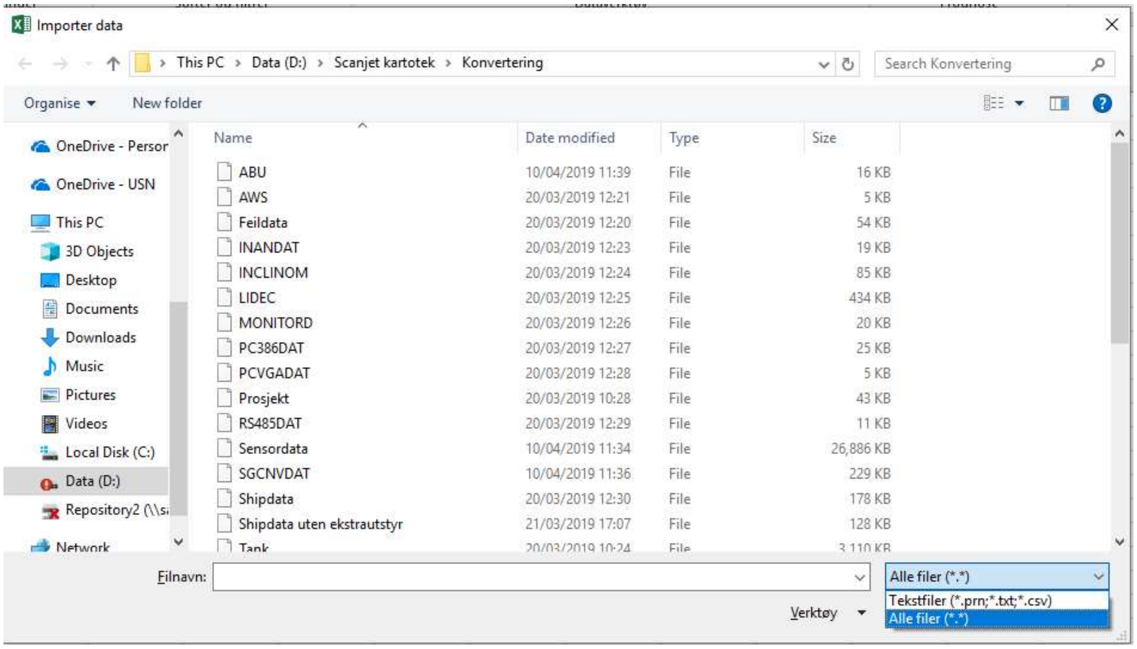


Fra tekst/CSV

Importer data fra en tekstfil, en kommadelt fil eller en fil med formatert tekst (mellomromdelt).

Figure 1-10 Importing from text

A new window appears, change file type from Text files(tekstfiler) to All files(Alle filer) as seen in the lower right corner of Figure 1-11.

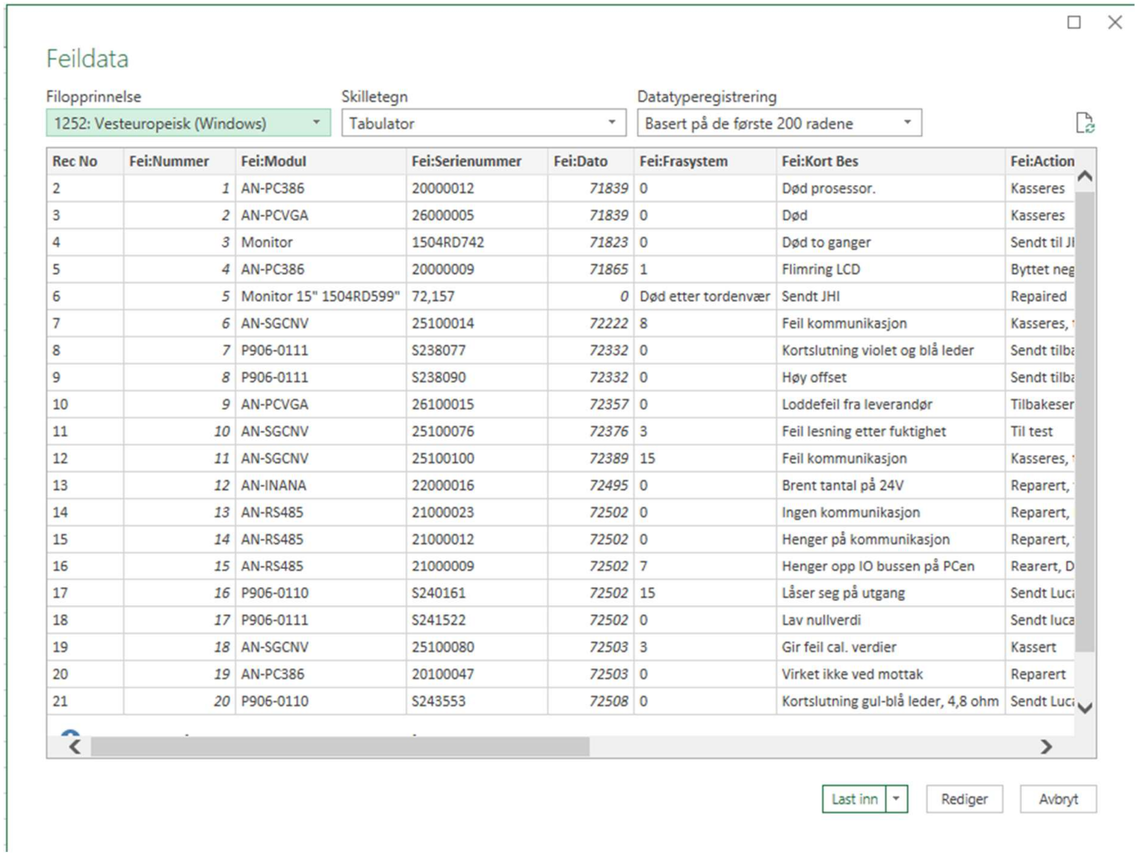


| Name | Date modified | Type | Size |
|---|---|---|---|
| ABU | 10/04/2019 11:39 | File | 16 KB |
| AWS | 20/03/2019 12:21 | File | 5 KB |
| Feildata | 20/03/2019 12:20 | File | 54 KB |
| INANDAT | 20/03/2019 12:23 | File | 19 KB |
| INCLINOM | 20/03/2019 12:24 | File | 85 KB |
| LIDEC | 20/03/2019 12:25 | File | 434 KB |
| MONITORD | 20/03/2019 12:26 | File | 20 KB |
| PC386DAT | 20/03/2019 12:27 | File | 25 KB |
| PCVGADAT | 20/03/2019 12:28 | File | 5 KB |
| Prosjekt | 20/03/2019 10:28 | File | 43 KB |
| RS485DAT | 20/03/2019 12:29 | File | 11 KB |
| Sensordata | 10/04/2019 11:34 | File | 26,886 KB |
| SGCNVDAT | 10/04/2019 11:36 | File | 229 KB |
| Shipdata | 20/03/2019 12:30 | File | 178 KB |
| Shipdata uten ekstrautstyr | 21/03/2019 17:07 | File | 128 KB |
| Tank | 20/03/2019 10:24 | File | 3,110 KB |

Alle filer (*.*)
Tekstfiler (*.prn;*.txt;*.csv)
Alle filer (*.*)

Figure 1-11 Choosing the file to import

This should be imported into a blank excel file. When a file has been chosen, a new window appears as in Figure 1-12. Click Load(Last inn) and data should appear in your worksheet. It is recommended to import all files before continuing to chapter 1.3.



Feildata

Filopprinnelse: 1252: Vesteuropeisk (Windows)

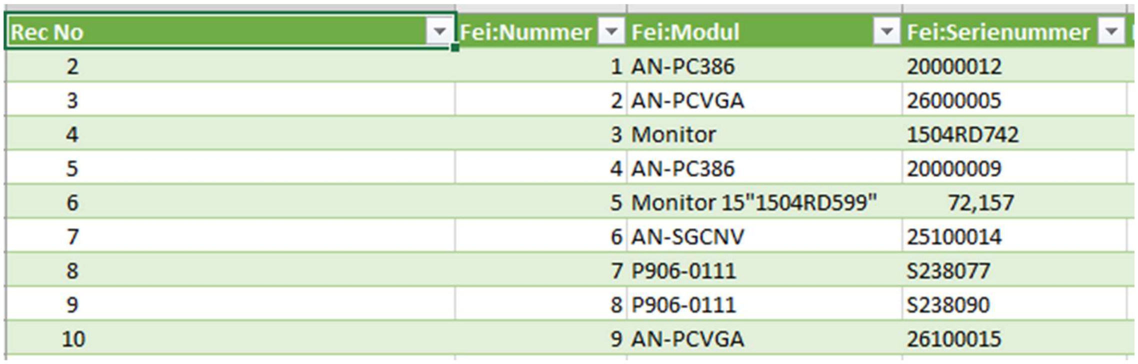
Skilletegn: Tabulator

Datatyperegistrering: Basert på de første 200 radene

| Rec No | Fei:Nummer | Fei:Modul | Fei:Serienummer | Fei:Dato | Fei:Frasystem | Fei:Kort Bes | Fei:Action |
|---|---|---|---|---|---|---|---|
| 2 | 1 | AN-PC386 | 20000012 | 71839 | 0 | Død prosessor. | Kasseres |
| 3 | 2 | AN-PCVGA | 26000005 | 71839 | 0 | Død | Kasseres |
| 4 | 3 | Monitor | 1504RD742 | 71823 | 0 | Død to ganger | Sendt til JI |
| 5 | 4 | AN-PC386 | 20000009 | 71865 | 1 | Flimring LCD | Byttet neg |
| 6 | 5 | Monitor 15" 1504RD599" | 72,157 | 0 | Død etter tordenvær | Sendt JHI | Repaired |
| 7 | 6 | AN-SGCNV | 25100014 | 72222 | 8 | Feil kommunikasjon | Kasseres, |
| 8 | 7 | P906-0111 | S238077 | 72332 | 0 | Kortslutning violet og blå leder | Sendt tilba |
| 9 | 8 | P906-0111 | S238090 | 72332 | 0 | Høy offset | Sendt tilba |
| 10 | 9 | AN-PCVGA | 26100015 | 72357 | 0 | Loddefeil fra leverandør | Tilbakeser |
| 11 | 10 | AN-SGCNV | 25100076 | 72376 | 3 | Feil lesning etter fuktighet | Til test |
| 12 | 11 | AN-SGCNV | 25100100 | 72389 | 15 | Feil kommunikasjon | Kasseres, |
| 13 | 12 | AN-INANA | 22000016 | 72495 | 0 | Brent tantal på 24V | Reparert, |
| 14 | 13 | AN-RS485 | 21000023 | 72502 | 0 | Ingen kommunikasjon | Reparert, |
| 15 | 14 | AN-RS485 | 21000012 | 72502 | 0 | Henger på kommunikasjon | Reparert, |
| 16 | 15 | AN-RS485 | 21000009 | 72502 | 7 | Henger opp IO bussen på PCen | Rearert, D |
| 17 | 16 | P906-0110 | S240161 | 72502 | 15 | Låser seg på utgang | Sendt Luc |
| 18 | 17 | P906-0111 | S241522 | 72502 | 0 | Lav nullverdi | Sendt luca |
| 19 | 18 | AN-SGCNV | 25100080 | 72503 | 3 | Gir feil cal. verdier | Kassert |
| 20 | 19 | AN-PC386 | 20100047 | 72503 | 0 | Virket ikke ved mottak | Reparert |
| 21 | 20 | P906-0110 | S243553 | 72508 | 0 | Kortslutning gul-blå leder, 4,8 ohm | Sendt Luc |

Figure 1-12 Loading data

Your excel sheet should look something like Figure 1-13



| Rec No | Fei:Nummer | Fei:Modul | Fei:Serienummer |
|---|---|---|---|
| 2 | 1 | AN-PC386 | 20000012 |
| 3 | 2 | AN-PCVGA | 26000005 |
| 4 | 3 | Monitor | 1504RD742 |
| 5 | 4 | AN-PC386 | 20000009 |
| 6 | 5 | Monitor 15"1504RD599" | 72,157 |
| 7 | 6 | AN-SGCNV | 25100014 |
| 8 | 7 | P906-0111 | S238077 |
| 9 | 8 | P906-0111 | S238090 |
| 10 | 9 | AN-PCVGA | 26100015 |

Figure 1-13 Clarion data in Excel

Feil I feildata linje 5

It’s recommended to import all data to the blank Excel file first, before proceeding.

## 1.3 Transformation of data

Each table in the old database may be a part of several tables in the new database, therefore a Clarion table will have many sheets in *03 Transformation.xlsx*. In the Excel book, the sheets will have the name type Old Table, Old table_To Python , Old Table_New Table, OldTable_details, Old table help or New table help.

### 1.3.1 Predefined data and connecting to the database from Excel

The data should be transferred in a certain order. This is shown in Figure 1-14. The first row, numbered 1 and additionally *detail_type(2), sensor_specification(2)* and *article*, are found in *01 Predefined data.xlsx* and should be uploaded to the database first. As Figure 1-14 suggests, *detail_type* and *sensor_specification* must be uploaded after *data_type* and *sensor_category* respectively. Lastly *article* must be uploaded. The other tables will be explained in due turn.

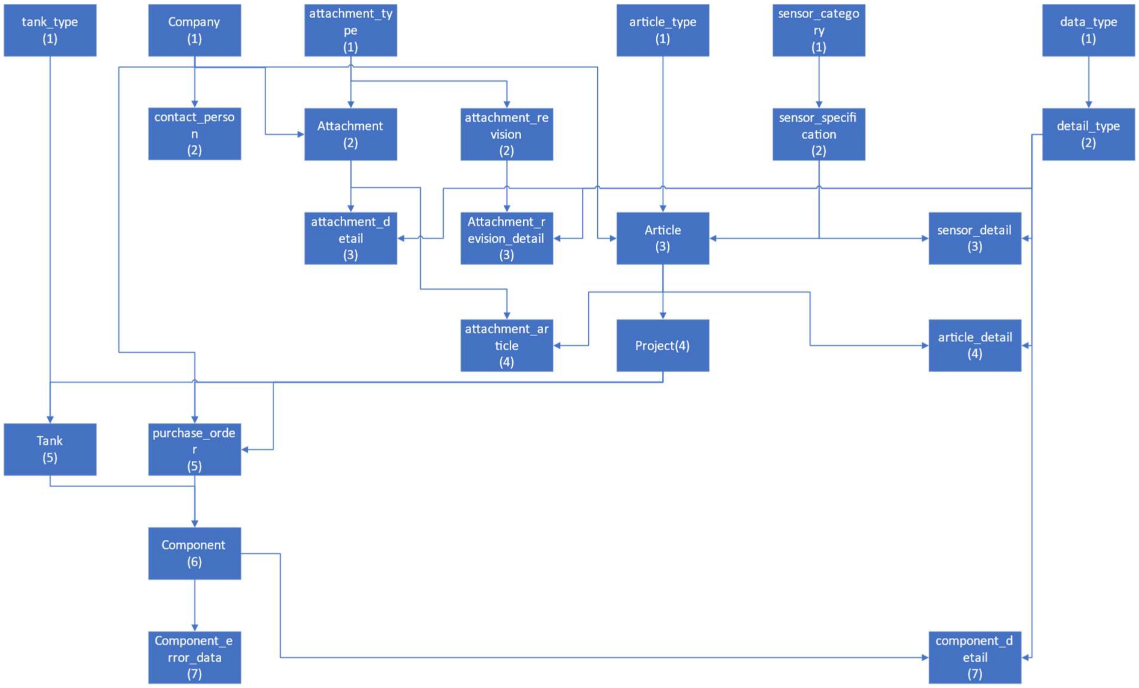
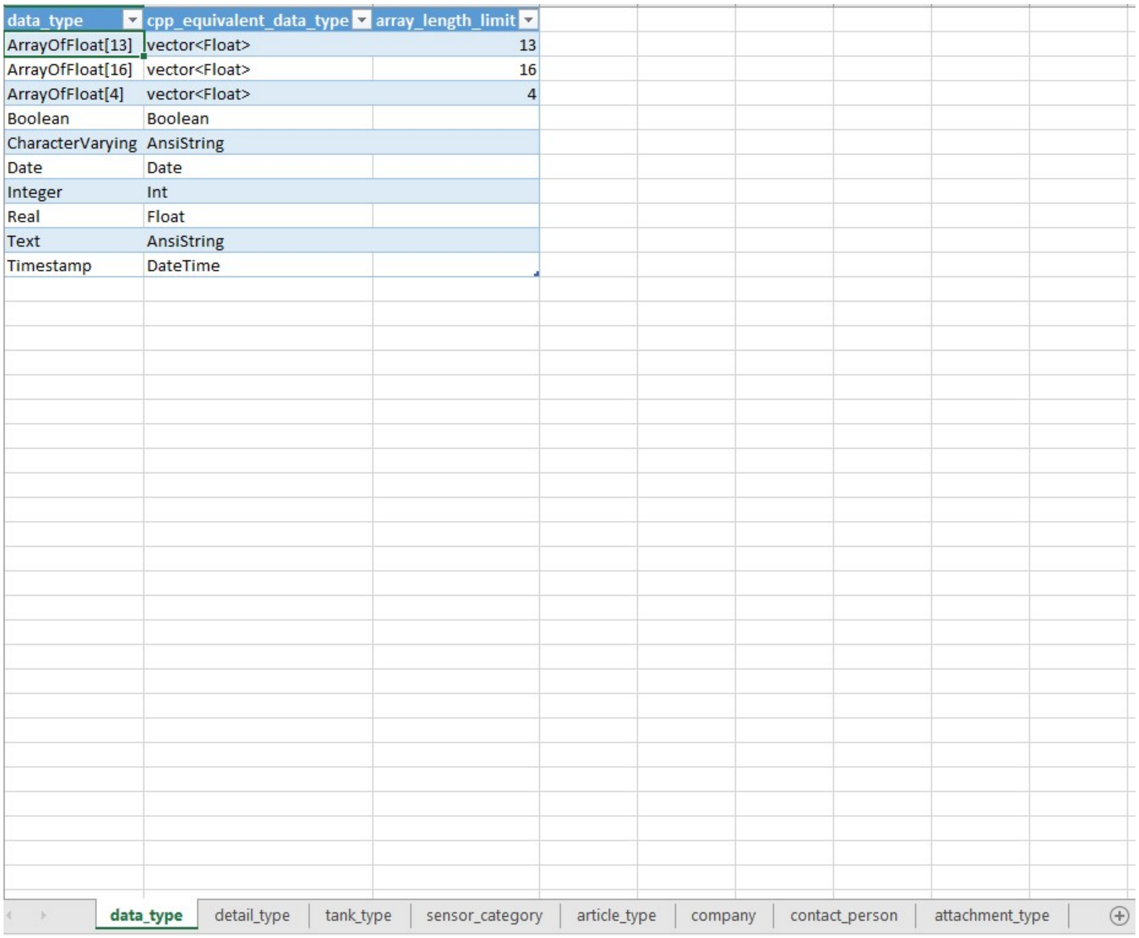


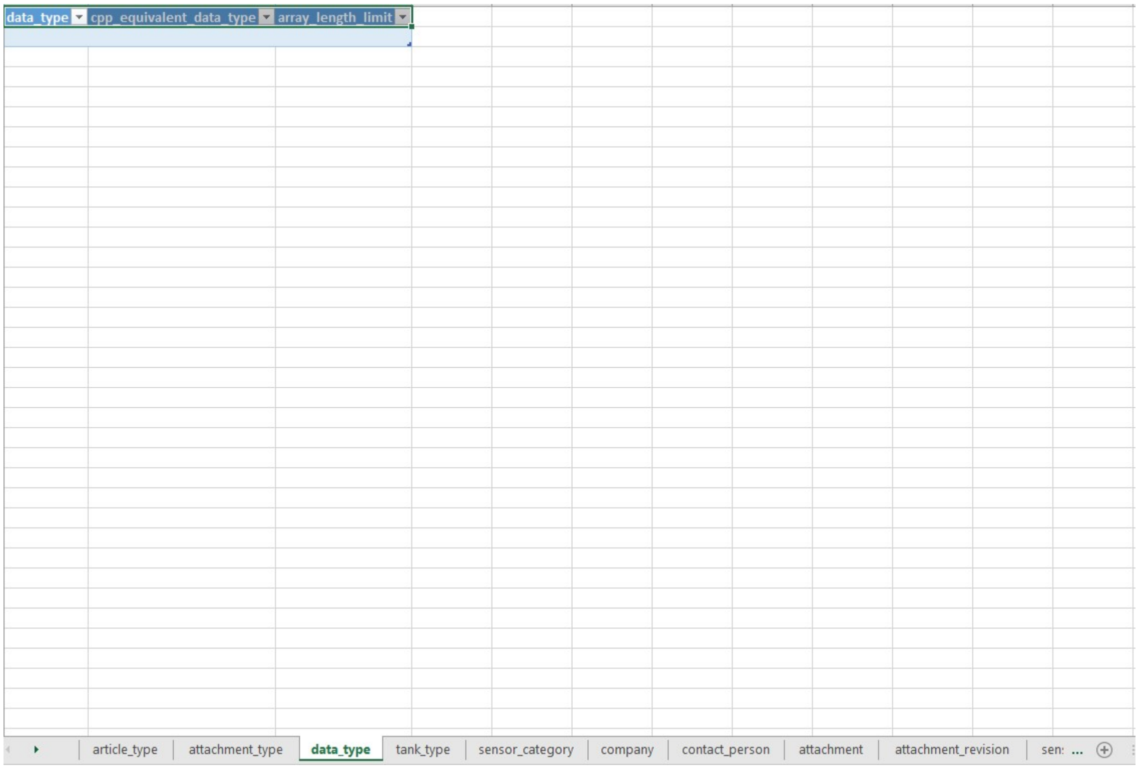
Figure 1-14 Order of data migration

To upload the first data, open the Excel files *01 Predefined data.xlsx* (visual in Figure 1-15) and *04 Insert sheet.xlsx* (visual in Figure 1-16).



| data_type | cpp_equivalent_data_type | array_length_limit |
|---|---|---|
| ArrayOfFloat[13] | vector<Float> | 13 |
| ArrayOfFloat[16] | vector<Float> | 16 |
| ArrayOfFloat[4] | vector<Float> | 4 |
| Boolean | Boolean | |
| CharacterVarying | AnsiString | |
| Date | Date | |
| Integer | Int | |
| Real | Float | |
| Text | AnsiString | |
| Timestamp | DateTime | |

Figure 1-15 *01 Predefined data*



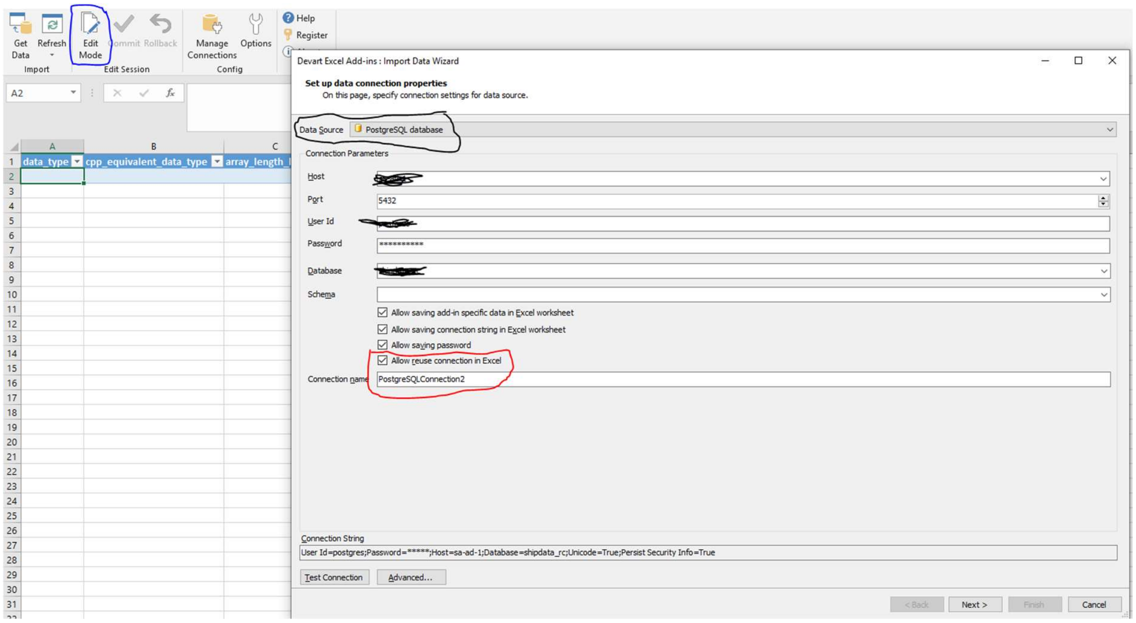
| data_type | cpp_equivalent_data_type | array_length_limit |
|---|---|---|
| | | |

Figure 1-16 *04 Insert book.xlsx*

In *04 Insert Sheet.xlsx* connect to the database by clicking *Edit mode* in the top menu (marked blue in following figure).

If a new database is created before migration is executed, the insert sheets must be recreated as they're connected to the current version. If this is not the case, skip the two next figures and related text.

You must then create a new connection to the new database by clicking *Get data* to the left of *Edit mode*. You will be prompted with a new window as shown in Figure 1-17. Give the host address, port, user id, password and database name. When entered, check the box *Allow reuse connection in Excel* and give a name to the connection (marked red in figure). The connection can now be used again by choosing a Data source (marked with black). Click next.



Devart Excel Add-ins : Import Data Wizard

**Set up data connection properties**
On this page, specify connection settings for data source.

Data Source: PostgreSQL database

Connection Parameters

Host: [redacted]
Port: 5432
User Id: [redacted]
Password: **********
Database: [redacted]
Schema:

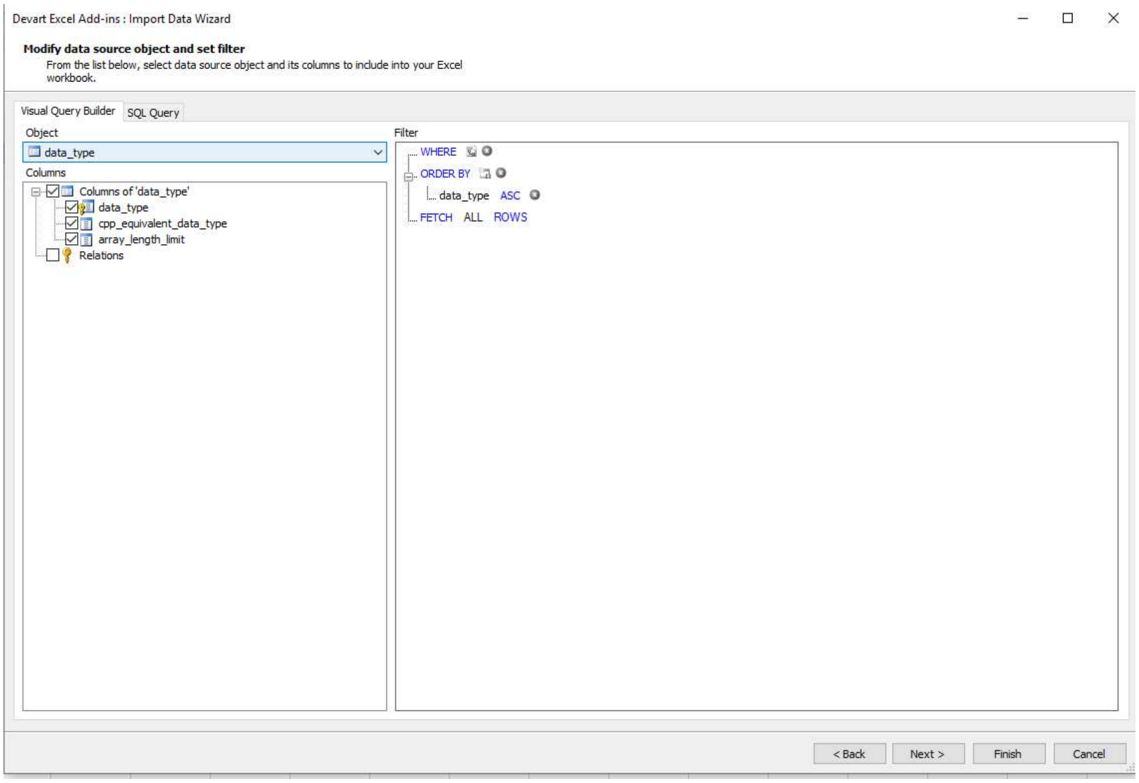
☑ Allow saving add-in specific data in Excel worksheet
☑ Allow saving connection string in Excel worksheet
☑ Allow saving password
☑ Allow reuse connection in Excel

Connection name: PostgreSQLConnection2

Connection String: User Id=postgres;Password=*****;Host=sa-ad-1;Database=shipdata_rc;Unicode=True;Persist Security Info=True

Figure 1-17 First step: connect to database

Choose a desired table, which in this case must be repeated for all tables in the database. When a table is chosen, click *Finish*. Devart will download the format of the chosen table. Now choose *Edit mode* again.


Devart Excel Add-ins : Import Data Wizard

**Modify data source object and set filter**
From the list below, select data source object and its columns to include into your Excel workbook.

Object: data_type

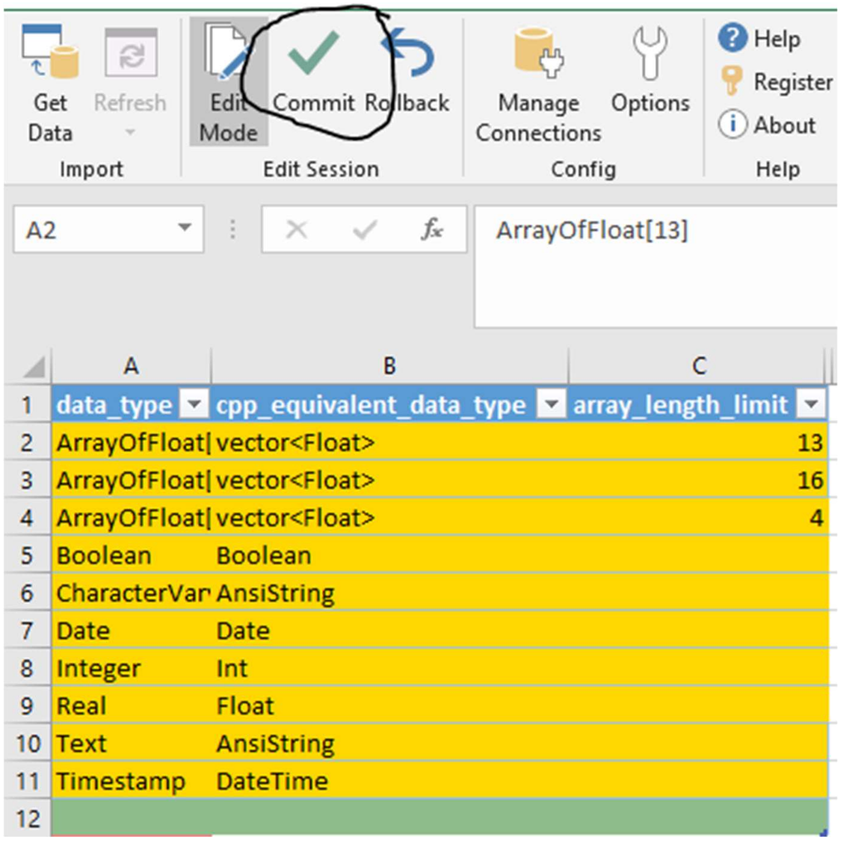
Columns
- ☑ Columns of 'data_type'
  - ☑ data_type
  - ☑ cpp_equivalent_data_type
  - ☑ array_length_limit
- □ Relations

Filter

```
WHERE
ORDER BY
    data_type ASC
FETCH ALL ROWS
```

Figure 1-18 Second step: choose a table

Now that Edit has loaded and connected to the database, copy the data (not column names) from *01 Predefined data.xlsx* into the corresponding sheet in *04 Insert book.xlsx.* When the data is loaded into the insert sheet. Click *Commit* as shown in Figure 1-19.



| | A | B | C |
|---|---|---|---|
| 1 | data_type | cpp_equivalent_data_type | array_length_limit |
| 2 | ArrayOfFloat[ | vector<Float> | 13 |
| 3 | ArrayOfFloat[ | vector<Float> | 16 |
| 4 | ArrayOfFloat[ | vector<Float> | 4 |
| 5 | Boolean | Boolean | |
| 6 | CharacterVar | AnsiString | |
| 7 | Date | Date | |
| 8 | Integer | Int | |
| 9 | Real | Float | |
| 10 | Text | AnsiString | |
| 11 | Timestamp | DateTime | |
| 12 | | | |

Figure 1-19 Data inserted to *04 Insert book.xlsx*

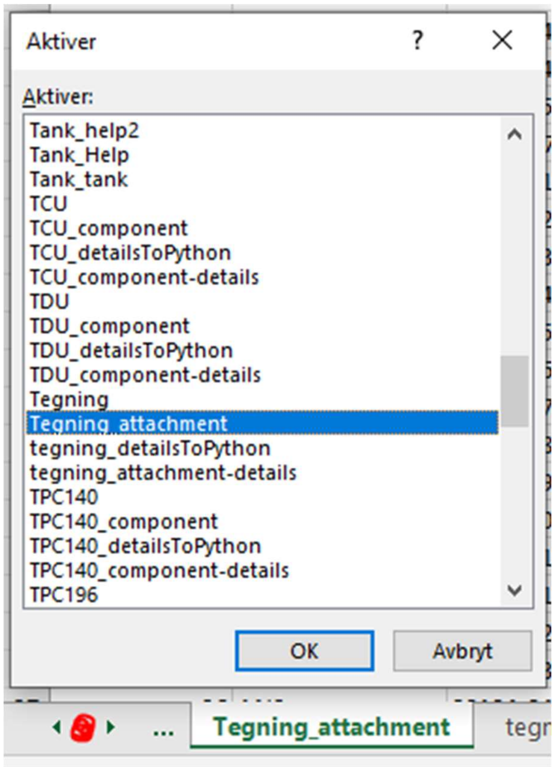
Complete all tables in *01 Predefined data* before moving on. Note that large amounts of data may use considerable amount of time to load. It's recommended to move primary keys alone as it appears creating new entries takes a lot of time compared to adding data to an existing record. Say about 1000 integer PK will likely take 15-20min to insert into the sheet, and some time after committing as well.

*01 Predefined data.xlsx* may now be closed.

## 1.3.2 Using the Transformation Excel book

*Transformation.xlsx* contains over 100 sheets formed as both databases and interfaces between them. This guide will tell where to start and how to navigate to execute the migration.

All data exported from Clarion should now be copied into *03 Transformation.xlsx* in appropriate sheets. E.g. TDU from clarion goes to the TDU sheet in *03 Transformation.xlsx*. To easily find the correct sheets see Figure 1-20. When all Clarion database data is copied into the workbook, a good place to start is by once again look at Figure 1-21.



Aktiver

Aktiver:

Tank_help2
Tank_Help
Tank_tank
TCU
TCU_component
TCU_detailsToPython
TCU_component-details
TDU
TDU_component
TDU_detailsToPython
TDU_component-details
Tegning
Tegning_attachment
tegning_detailsToPython
tegning_attachment-details
TPC140
TPC140_component
TPC140_detailsToPython
TPC140_component-details
TPC196

Figure 1-20 Easy navigation in Excel by right clicking at the red dot and choosing desired table

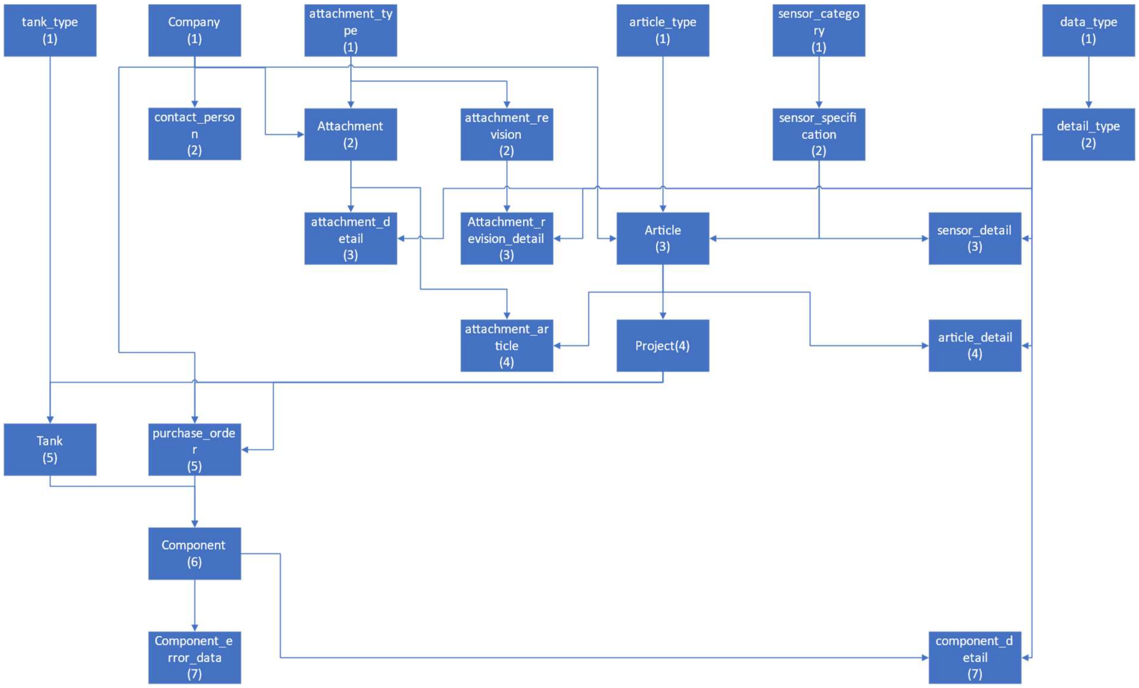


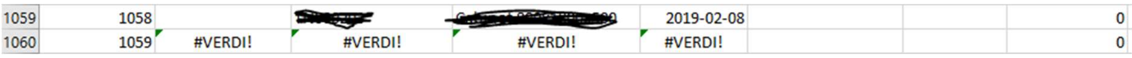
Figure 1-21 Order of data migration

Now that all the tables marked with (1), *detail_type* and *sensor_specification* is finished, tables marked with (2) should be attended. There exists no *contant_person* or *attachment revision* data, so these tables can be excluded. This leaves *Attachment* the obvious choice.

#### 1.3.2.1 Attachment and the general way to use most sheets in Transformation.xlsx

To migrate the content of this table, be sure to click the *Edit mode* in *04 Insert book.xlsx* at the *attachment* sheet. Open *03 Transformation.xlsx* and navigate to *Tegning_attachment*, once again, it's recommended not to copy big chunks of data, experiment with how many primary keys you want to insert at a given time, 1000 is likely a good start and should take 15-20minutes. When copying this data and all further data, it's **important to paste as values** as these values are derived with Excel formulas.

Make sure to scroll to the bottom of every sheet and look for values as shown in Figure 1-22. If this is not found, there are likely more entries which is not yet found. In this case, mark the last row which has values and copy the formulas by clicking and holder the lower right corner and dragging downward. The formulas will lookup in the Clarion database sheet and find additional entries. If the hashtag values are found, these rows should not be copied.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1059 | 1058 | | [redacted] | [redacted] | 2019-02-08 | | | 0 |
| 1060 | 1059 | #VERDI! | #VERDI! | #VERDI! | #VERDI! | | | 0 |

Figure 1-22 End of values from Clarion

When all primary keys are inserted and *Commited,* the rest of the data can be copied. Again, experiment with what number of cells you copy at a time, Devart for Excel does not need to create new entries now and can handle more data.

This approach can be used for all tables which are not detail tables. Still Figure 1-21 applies, the next step is to upload data from any table marked with (3) as all marked with (1) and (2) are finished.

#### 1.3.2.2 Attachment_detail

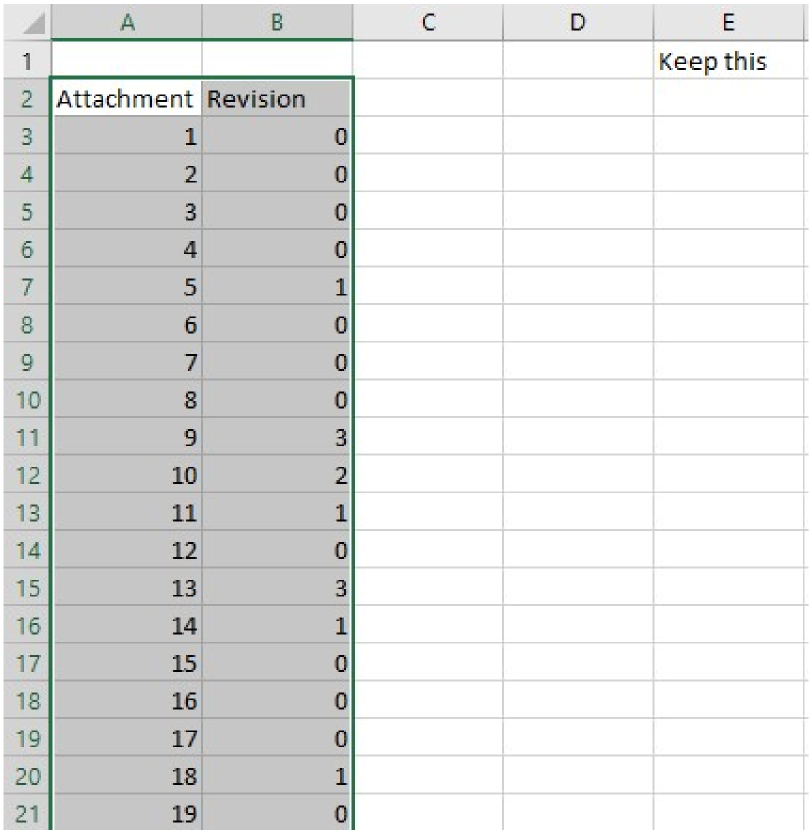
As *attachments* are now uploaded, this is a good time to explain the *attachment_details* and thus all other detail tables. In *03 Transformation.xlsx,* attachments and components have two additional sheets, generally called *old table_detailsToPython* and *old table_new table-details.* In the case of attachment, they're called *Tegning_detailToPython and Tegning_attachment-details.*

Here as Figure 1-1 indicated, it's needed to use a Python script to restructure the data.

There should now be updated data in *Tegning_detailsToPython,* again it's needed to check that all entries are present, as described in the second section of chapter 1.3.2.1.

When all entries are accounted for, the data, including column names, should be copied to *01_To_transformation.xlsx.* Figure 1-23 shows how this should look like. Note that the cell marked with "Keep this" is protected and must not be deleted. **This Excel book and *02_Finished* must not be moved** as an .exe file will use them.

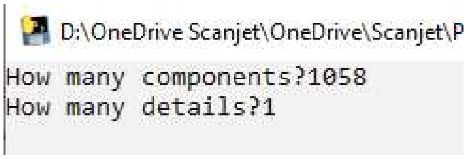
For the next step, note how many IDs and how many detail columns that are copied. In this case I have 1058 IDs and one detail column. The book should then be saved and closed.



| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | | | | | Keep this |
| 2 | Attachment | Revision | | | |
| 3 | 1 | 0 | | | |
| 4 | 2 | 0 | | | |
| 5 | 3 | 0 | | | |
| 6 | 4 | 0 | | | |
| 7 | 5 | 1 | | | |
| 8 | 6 | 0 | | | |
| 9 | 7 | 0 | | | |
| 10 | 8 | 0 | | | |
| 11 | 9 | 3 | | | |
| 12 | 10 | 2 | | | |
| 13 | 11 | 1 | | | |
| 14 | 12 | 0 | | | |
| 15 | 13 | 3 | | | |
| 16 | 14 | 1 | | | |
| 17 | 15 | 0 | | | |
| 18 | 16 | 0 | | | |
| 19 | 17 | 0 | | | |
| 20 | 18 | 1 | | | |
| 21 | 19 | 0 | | | |

Figure 1-23 Attachment details copied to *01_To_transformation*

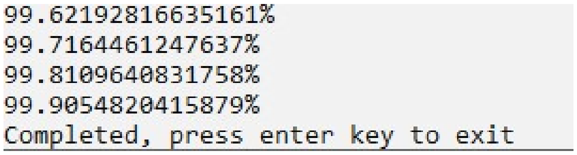
Now that *01_To_transformation* is saved and closed, run *Migration.exe.* In Figure 1-24 the noted numbers from last section is entered.



```
D:\OneDrive Scanjet\OneDrive\Scanjet\P
How many components?1058
How many details?1
```

Figure 1-24 *Migration.exe*

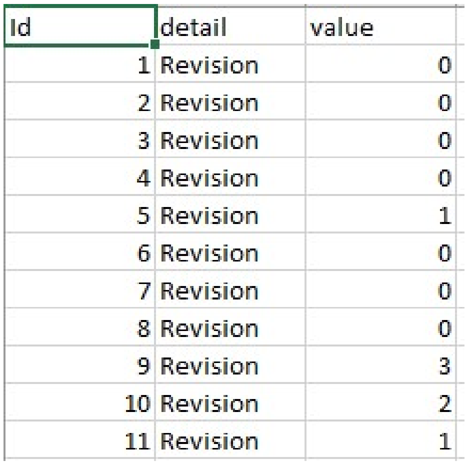
Pressing enter again will run the script, progressively showing how far it has come in terms of 0-100%. If the program should crash, this is likely because too many components or details are entered. The program will then exit, when the program succeeds it will halt and wait for the user to press *enter* as shown in Figure 1-25.



```
99.62192816635161%
99.7164461247637%
99.8109640831758%
99.9054820415879%
Completed, press enter key to exit
```

Figure 1-25 *Migration.exe* has successfully completed

*Migration.exe* has now updated *02_Finished.xlsx,* the product is shown in Figure 1-26. The data, without the column names can now be copied to either: *04 Insert book.xlsx* and be uploaded to the database, or to *03 Transformation.xlsx* in the *Tegning_attachment-details* sheet. The latter is not needed but may be preferable if you want to complete all transformations before uploading. If not, do as always, be sure to click the *Edit mode* button before pasting, and preferably paste and commit IDs before the rest of the data.



| Id | detail | value |
|---|---|---|
| 1 | Revision | 0 |
| 2 | Revision | 0 |
| 3 | Revision | 0 |
| 4 | Revision | 0 |
| 5 | Revision | 1 |
| 6 | Revision | 0 |
| 7 | Revision | 0 |
| 8 | Revision | 0 |
| 9 | Revision | 3 |
| 10 | Revision | 2 |
| 11 | Revision | 1 |

Figure 1-26 Details transformed to PostgreSQL format

### 1.3.3 Finally, almost

The procedures are from now on repeated for all tables with some lesser modifications on tables mentioned in the next subchapters.

Follow the instructions of Figure 1-27. Tables which are not a detail table uses the same approach as chapter 1.3.2.1. Detail tables follow the approach of 1.3.2.2. When it comes to c*omponent* and *component_detail* there are many sets of these tables. Start from the left in *03 Transformation.xlsx,* remember to check primary keys as explained in chapter 1.3.2. Additionally, remember for each sheet in *04 Insert book.xlsx* click *Edit mode.* This function often disconnects so check each time you want to upload something.

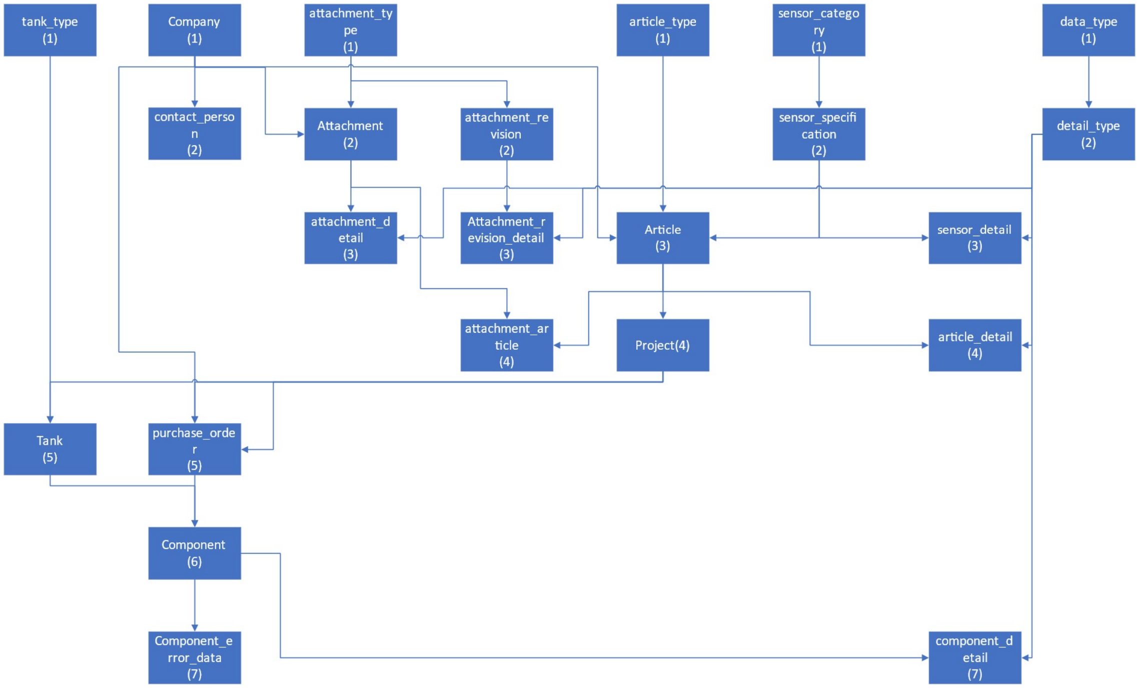


Figure 1-27 Order of data migration

## 1.3.4 Prosjekt-shipdata_project

Some *Prosjektnummer* from the Clarion database are empty, these will create issues in the table *purchase order* as *purchase order's* primary key will have the prefix "PO:" with project_number as variable. For now Q00000x is used and must be placed manually.

*Prosjektnummer* entries aren't unique and will cause trouble in this case as well, either they can be removed by marking and clicking *Data->Remove Duplicates* in Excel (two are found), or they may be searched for and given new values.

## 1.3.5 Sensordata_details

These sheets are named somewhat differently than other detail tables. This is because differing sensors have variable amount of detail columns. These have been split such that each type has their own sheet. These sheets are now called:

- *P906_details to python*
- *P986_details to python*
- *ATM_details to python*
- *VEGA_details to python*
- *P1728_details to python*
- *P_details to python*
- *PTM_details to python*
- *PULS_details to python*
- *SPT_details to python*

Note that *sensordata_details_help* should not be used during migration.

### 1.3.6 Tank

Tank names have been added as a detail of some components, this is due to late validation of how to store it. It is redundant data, but there is not time to rectify this. Scanjet may remove these

### 1.3.7 Feildata_component-error-data

This is one of the last tables to be uploaded and uses data which has been uploaded beforehand. To execute this, use Devart for Excel to download the new *component* table from the server. Insert the values into *component-erorr-data_help* in *03 Transformation.xlsx.*

From here on, use the Excel Insert book to upload the data to *component_error_data* as usual.

# Appendix J

```
<!DOCTYPE shipdata> [

<!ELEMENT config (gui, database)>

<!ELEMENT gui (datafields)>

<!ELEMENT controls (control*)>

<!LEMENT control EMPTY>
<!ATTLIST control
name CDATA #REQUIRED>

<!ELEMENT datafields (datafield*)>

<!ELEMENT datafield EMPTY>
<!ATTLIST datafield
name CDATA #REQUIRED
width CDATA #REQUIRED
height CDATA #REQUIRED
control CDATA #REQUIRED
readonly (true|false) #REQUIRED>

<!ELEMENT database (tables,relationships,data_types,detail_types,views)>
<!ATTLIST database
name CDATA #REQUIRED
version CDATA #REQUIRED>

<!ELEMENT tables (table*)>

<!ELEMENT table (accesslevel, column+)>
<!ATTLIST table
name ID #REQUIRED
alias CDATA #REQUIRED>

<!ELEMENT column (accesslevel, foreignkey?)>
<!ATTLIST column
name CDATA #REQUIRED
alias CDATA #REQUIRED
datatype (text|integer|real|bool|date|timestamp) #REQUIRED
datalength CDATA "0"
isprimarykey (true|false) "false"
readonly (true|false) "false">

<!ELEMENT accesslevel EMPTY>
<!ATTLIST accesslevel
level (0|1|2|3|4|5|6|7|8|9) "5"
exceptions CDATA #IMPLIED>

<!ELEMENT relationships (relationship*)>

<!ELEMENT relationship (parent+, child+)>
<!ATTLIST relationship
name CDATA #REQUIRED
cardinality (1|N) #REQUIRED
```

```
mandatory (true|false) #REQUIRED>

<!ELEMENT parent EMPTY>
<!ATTLIST parent
tablename CDATA #REQUIRED
columnname CDATA #REQUIRED>

<!ELEMENT child EMPTY>
<!ATTLIST child
tablename CDATA #REQUIRED
columnname CDATA #REQUIRED>

<!ELEMENT data_types (data_type*)>
<!ATTLIST data_types
tablename CDATA #REQUIRED
data_type_column CDATA #REQUIRED
array_length_column CDATA #REQUIRED>

<!ELEMENT data_type (gui_representation?)
<!ATTLIST data_type
name CDATA #REQUIRED
cpp_equivalent_data_type CDATA #REQUIRED
array_length_limit CDATA #REQUIRED
array_delimiter (,|.|-|/) #IMPLIED>

<!ELEMENT gui_representation EMPTY>
<!ATTLIST gui_representation
name CDATA #REQUIRED>

<!ELEMENT detail_types (detail_type*)>
<!ATTLIST detail_types
tablename CDATA #REQUIRED
detail_type_column CDATA #REQUIRED
data_type_relationship_name CDATA #REQUIRED>

<!ELEMENT data_type EMPTY>
<!ATTLIST data_type
name CDATA #REQUIRED
data_type_name CDATA #REQUIRED
unit CDATA #REQUIRED
description CDATA #REQUIRED>

<!ELEMENT view (accesslevel, viewtable)>
<!ATTLIST view
name CDATA #REQUIRED
alias CDATA #REQUIRED>

<!ELEMENT viewtable (viewcolumns, detailtable?, viewrelationships?)>
<!ATTLIST viewtable
tablename CDATA #IMPLIED>

<!ELEMENT viewcolumns (viewcolumn*)>
<!ATTLIST viewcolumns
auto (true|false) "true">
```

```
<!ELEMENT viewcolumn EMPTY>
<!ATTLIST viewcolumn
columnname CDATA #REQUIRED>

<!ELEMENT detailtable (detail+)>
<!ATTLIST detailtable
tablename CDATA #REQUIRED
detailcolumnname CDATA "detail_type"
valuecolumnname CDATA #REQUIRED>

<!ELEMENT detail EMPTY>
<!ATTLIST detail
property CDATA #REQUIRED>

<!ELEMENT viewrelationships (viewrelationship+)>

<!ELEMENT viewrelationship (viewtable, constantprimarykey?)>
<!ATTLIST viewrelationship
relationshipname CDATA #REQUIRED
constant (true|false) "true">

<!ELEMENT constantprimarykey (constantcolumn+)>

<!ELEMENT constantcolumn EMPTY>
<!ATTLIST constantcolumn
columnname CDATA #REQUIRED
columnvalue CDATA #REQUIRED>

]>

Other definitions:

<viewtable> @tablename: Only required for the first(most shallow)
occurence. Nested viewtable[s] already have this information through
<relationship>
<relationship>: <parent> and <child> elements must be ordered as follows:
parent1, parent2, parent3, child1, child2, child3


           Appendix J
```