Technische Universität München
Department of Mechanical Engineering
Institute of Automation and Information Systems
Prof. Dr.-Ing. Birgit Vogel-Heuser

Bachelor's Thesis

# Automated Generation of Modular and Dynamic Industrial Process Plant Visualizations in a Manufacturing Execution System (MES)

| | |
|---|---|
| Author: | Miguel Romero Karam |
| Matriculation No.: | 03675217 |
| Advisor: | Dr.-Ing. Daniel Schütz |
| Supervisor: | Emanuel Trunzer, M.Sc. |
| Begin Date: | 15 April 2018 |
| Submission Date: | 20 September 2018 |

# Statutory Declaration

I hereby confirm to have written the present bachelor's thesis independently and only with the use of the sources and resources I have indicated. Both content and literal content were identified as such. The work has not been available in this or similar form to any other panel of examiners.

Date: 07.09.208 Signature: 

# Abstract

The objective of this bachelor's thesis is the development of a system for the automated generation of industrial plant piping and instrumentation diagram (P&ID) visualizations within the scope of the Process Application Composer (ProcAppCom) project. The task of generating dynamic process plant visualizations and keeping them up-to-date demands significant efforts due to the frequent and often manual reconfiguration and adjustment required to do so. By means of the developed solution, both technical effort and time for the creation, updating and modification of P&ID visualizations is to be decreased. The developed solution is to be implemented and tested in the Legato Manufacturing Execution System (MES) web application for an example brewery facility and finally evaluated for an eventual application in a real industrial context.

# Index

# 1 Introduction

## 1.1 Overview and Motivation

Software has become indispensable in modern industrial contexts, but the increasing complexity of the industrial contexts themselves makes its implementation in an efficient, economical and advantageous manner more challenging now than ever.

The development of control software and visualization interfaces for the operation of smaller process engineering systems is a major cost driver in process engineering automation projects. Additional to the high up-front implementation cost for the connection and configuration of a *Manufacturing Execution System (MES)*, operational costs rise rapidly during the MES product life cycle; Creating and later modifying plant-specific visualization interfaces represents a significant technical effort, which translates to these continued increments in operational expenditure. The MES software architecture is often deeply intertwined; Adjustments in any area usually have consequences in others, even rather simple modifications can propagate and lead to important sources of errors, imposing constant software adjustments. A slight shop floor modification, be it a physical change in the plant like the disabling of a temperature sensor or a change to the order of production, might result in significant number of adjustments for the MES. Process visualizations in the *Graphical User Interface (GUI)* are similarly influenced; being virtual representations of the physical process facility, they demand frequent adjustments which result in significant overhead for its implementation.

The *Process Application Composer (ProcAppCom)* research project behind this bachelor thesis represents a cooperation between multiple industrial partners, namely *3S-Smart Software Solutions GmbH, Gefasoft GmbH, Johann Albrecht Brautechnik GmbH* and *APE Engineering GmbH* with the *Technical University of Munich*. The main objective of the ProcAppCom research project is the automated configuration and generation of control code and visualizations for production plants in the field of process engineering.

Gefasoft GmbH is a leading and innovative provider of production-related software solutions. With the product *Legato Sapient®, Gefasoft* offers a web-based *MES* for the distributed control of production. Main functions of a *MES* are production management, supervisory control, maintenance management and the real-time data acquisition, storage and integration to other information systems. These include e*nterprise resource planning (ERP)* and s*upervisory control and data acquisition (SCADA)* systems as well as *programmable logic controllers (PLC)*. A *MES* therefore typically spans from the operational management level, where it is implemented, through the process management (*SCADA*) and the control levels (*PLC*), unto the field layer or shop floor.

Motivation of this bachelor thesis is the development of a system for the automated generation of dynamic *Piping and Instrumentation Diagram (P&ID)* visualizations for industrial plants with the goal of reducing implementation and operation expenditure for a *MES*, so that any enterprises can profit from these software solutions.

## 1.2 Problem Definition

The present trends in automation technology lead to a permanent increase in the complexity of industrial process facilities and to permanent technical changes. These changes propagate through the documentation, maintenance and operation of mentioned facilities, which represents a major engineering challenge. This leads to the need for the frequent and often manual reconfiguration and adjustment of such systems during their life cycle. Plant-specific visualizations in the *graphical user interface (GUI)* demand significant efforts for their creation and modification after technical changes of any kind. As virtual representations of the process facility and with the imminent increase in changes due to accelerated technological advancements, they are subject to constant manual updating. Moreover, this constant change leads to deviation from the industry standards for visualizations, like those for *P&ID* visualizations. As a result, different companies from different sectors end up each with different standards, which further increases the engineering complexity and results in counterproductive *GUIs*. For these reasons, production software requires adapting to these demanding trends in the process engineering industry. With respect to process visualizations in the process engineering industry, a system must be developed for the automated generation of modular and dynamic plant P&ID visualizations with minimal user configuration and integration to the MES software at hand.

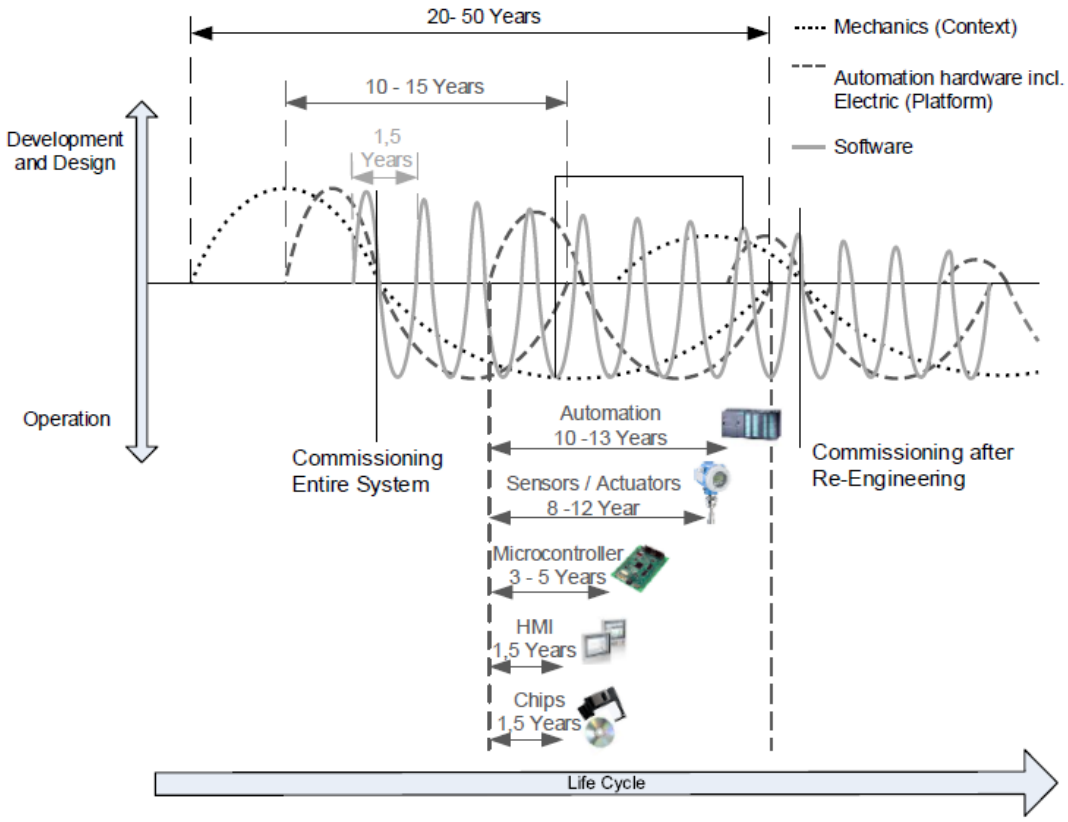



*Figure 1 Software life cycles in comparison and the rise in its importance. Source: TUM AIS Chair*

## 1.3 Initial Situation

The foundations of this project were already set by previous projects in the context of the *ProcAppCom* research project at *Gefasoft*. A general description model for process engineering plants was initially developed. Before the start of this project, it was also possible for plant models to be read and directly transcribed to database tables of the *MES Legato Sapient.* This enables the automated connection of the MES to the plant's control and field levels via the factory edge gateways. A system for the automated generation of P&ID visualizations for the user selected site, area, production unit, process cell or equipment module of the modelled process engineering plant is the culmination of this research project.

The *Legato Graphic Designer* boardlet of *Legato Sapient* was developed for the dynamic rendering of visualizations in the form of a single, static xml file. In favor of preventing repetition and to seamlessly integrate to the *Legato Sapient Environment*, existing code should be leveraged as much as possible. The final product is a dashboard for the creation and rendering of the user selected process engineering plant instance.

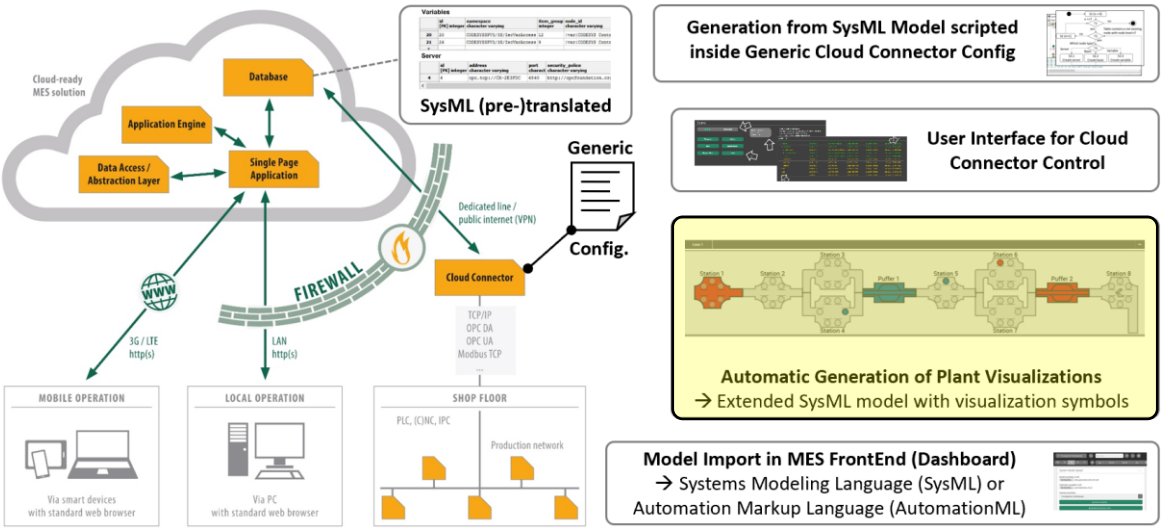



*Figure 2 Global ProcAppCom project overview (scope of this project in yellow). Source and Copyright: Gefasoft GmbH*

## 1.4 Goals of the Bachelor Thesis

The goals of the bachelor thesis give an overview of what is ultimately intended. They enabled a plan to be defined initially in terms of the desired outcomes of the project. The specifics for the fulfillment of the goals came later with the requirements definition. The following goals were set to define what was to be ultimately achieved by means of the developed solution:

- Reduce technical effort and accelerate the generation and modification of *Piping and Instrumentation Diagram (P&ID)* visualizations for industrial process plants.
- Standardization of the visualization components according to industrial standards and best practices for generation of consistent *P&ID* visualizations.
- Prototypal implementation of the software solution in the web-based *MES Legato Sapient* in the form of a user-friendly *GUI* dashboard for the generation and viewing of *P&ID* visualizations with abstraction of configurations for the user.
- Verification, validation and evaluation of the solution for the implementation in an industrial grade *manufacturing execution system*.

## 1.5 Project Requirements

Specific requirements where set apart from the project goals for the technical and conceptual aspects of the project. Figure 3 illustrates the conceptual overview of the solution departing from what was previously developed within the encompassing ProcAppCom project. It was intended for all requirements to be met by the end of the project, but the development cycle brought slight variations to some during the course of the project. The project requirements define the intended outcome of the practical

implementation of the project at *Gefasoft* by means of the proposed technical solution. Furthermore, the development of new methodologies to achieve the project goals outlined in section 1.4 represents the research part of the project.

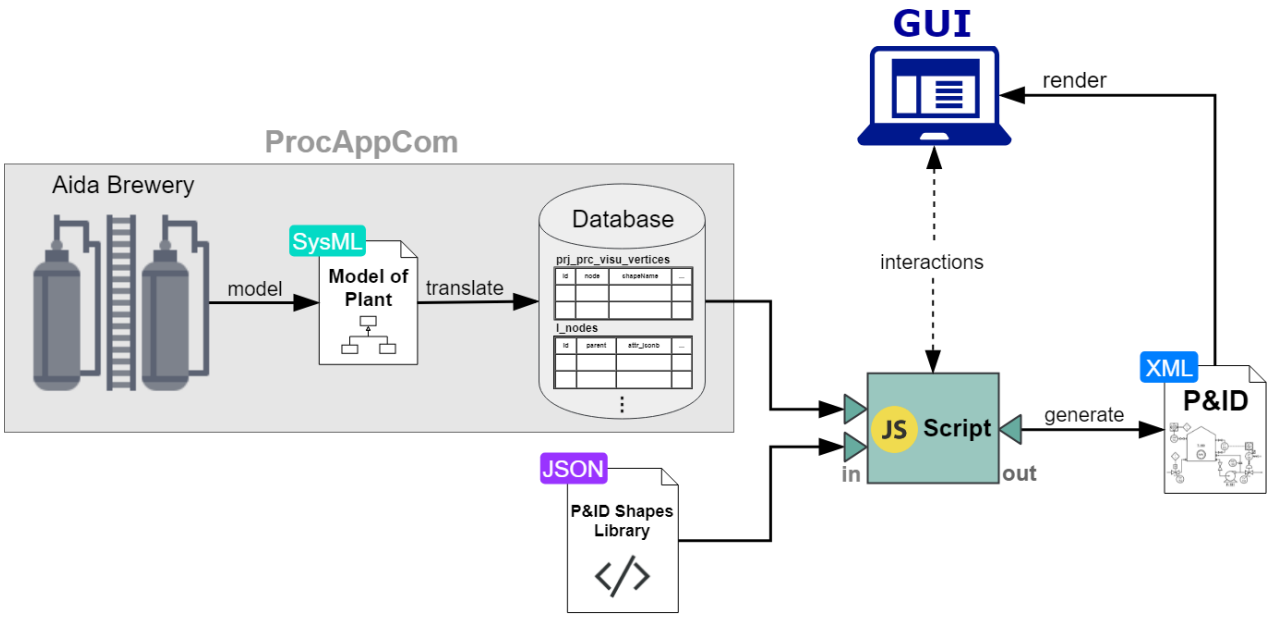



*Figure 3 Conceptual overview of the solution departing from previous works of the ProcAppCom project.*

**R1: Library of modular P&ID Visualization Components According to Industry Standards**

A library of *P&ID* symbols is to be developed with an *object-oriented* approach, by means of which shared characteristics are to be propagated (via *inheritance*). The symbols themselves should be compliant with the ANSI/ISA S5.1-1984 standards for *P&ID* visualizations and should allow for user-configurability and the real-time dynamic display of data (for example of process variables) components.

**R2: User Friendly Graphical User Interface (GUI) Boardlet for Creation of P&ID Visualizations**

A simple and user-friendly *graphical user interface* is to be implemented in the form of a *Legato Web Application* dashboard composed of boardlets. The dashboard should include the functionalities to create new, update existing, and render *P&ID* visualizations. The creation of the *P&ID* should abstract configuration details from the user, and require only minimal user action, for instance, the selection of the corresponding shape library and the node for which the visualization is to be generated.

**R3: Client-Side Script for the Automated P&ID Visualization Generation as an XML File**

The required *business logic* for the generation of *P&ID* visualizations is to run entirely on the client and require little-to-no user configurations, and at best, work at the click of single a button. This abstracts

unnecessary complexities for the user to handle. The script itself should be structured in a modular and composable way and packaged inside a single, well-documented *JavaScript* file.

**R4: Mapping of Physical Plant Instances to Corresponding Visualization Component**

Mapping of the modelled plant instances (in database) is to work seamlessly and unobtrusively with minimal modification to the model (and thus to the database tables), if any. At best, this should happen with the minimal requirement: a single mapping property (`shapeName`) through which all plant instances can be appropriately matched to its corresponding geometric *P&ID* shape in the library.

**R5: Automatic Type Detection and Simplification of Connections**

Contrary to vertices (or plant instances like pumps, tanks, and valves), connections (like the pipe lines and data lines between them) should not require any changes in the model in order to be correctly mapped to *P&ID* shapes. This task will require some conditional logic, as to determine which connections are of which type. That being said, the script should differentiate between data, process, connection and signal lines. Hence, no need to specify a *shapeName* for connections in the data model.

**R6: Declarative specification of Graphing Constraints in Form of Tags**

The graph layout and/or positioning logic should be implemented in a declarative form, as to provide readability and allow for the progressive enhancement of the algorithm. How the positioning logic is actually implemented should be decoupled from the constraints, which define which rules apply to what.

**R7: P&ID Graphing Algorithm**

No comparable solutions might exist for this purpose. Still, an in-depth research of available and similar graphing algorithms is to be done to borrow characteristics from similar existing solutions, if possible. The graph should optimize the area of the visualization so that it decently fits on desktop-sized screens.

**R8: Dynamic Real-Time Display of Process Variables in the P&ID Visualization**

Although the file which contains the *P&ID* is to be a static *XML* file, it should include dynamic data bindings to external variables, as to allow for the dynamic rendering of this data. This should support various data types and implement distinct animations and or labels according to that and the type of process engineering element. The *data-bindings* are to be implemented via the `sapient-bind` property, so that the available *Legato Graphic Designer* boardlet can handle them correctly.

**R9: Prototypal Implementation in the Infrastructure of a MES (Legato Web Application) and Documentation**

The developed solution is to be implemented in the infrastructure of the *Legato Web Application* and by means of its frameworks, libraries and tools. The solution is finally to be evaluated for functionality, performance and scalability, especially for the eventual implementation in an industrial context in the process engineering field. The development should stick to best-practices and code should be well-documented.

## 1.6 Composition of the Bachelor Thesis

### 1.6.1 Project Management

In favor of lightweight and flexible project management, the *GIST* methodology was preferred over more traditional agile methods. *GIST* is called after its main building blocks: Goals, Ideas, Sprints and Tasks, each with distinct planning perspective and frequency of change [1]. Instead of initially declaring tasks, goals where set, which enabled a plan to be defined in terms of the desired project outcomes. The goals stated in section 1.4 lead the decisions from beginning to end of the project and where maintained for the most part. Ideas where tracked during the entire project's life cycled and reconsidered for implementation or discarded at the beginning of each sprint. Sprints where executed until all tasks where completed, though tasks of previous and future sprints where sometimes worked upon outside of the corresponding sprint. Tasks themselves where reconsidered weekly for the current sprint and tracked with a *Kanban* board.

### 1.6.2 Project Sprints

**S1: Research and Choice of Tools and Technologies**

Before any work was done, the tools and technologies with which the goals would be achieved had to be at least preliminary decided. Though many diagramming frameworks and libraries exist, not all where optimal for the task at hand, therefore comparisons where done between the available technologies after meticulous analysis of the projects technical and conceptual requirements. The open-source *mxGraph* diagramming library (*JavaScript*) was chosen due to its lightweightedness, robustness between distinct web browsers and compatibility with the diagramming tool *draw.io*, built using *mxGraph*. Furthermore, the *Legato Graphic Designer* boardlet in which the visualization is to be rendered is implemented also with the *mxGraph* library.

**S2: Creation of a P&ID Shapes Library**

The second project sprint was the conception of an *object-oriented* library of modular shapes conforming to the industry standards for *P&ID* visualizations. This task was further divided into subtasks. first of which was the analysis of the *mxGraph application programming interface (API)*, with which the visualizations were to be implemented in the browser. These consisted of the manual creation and analysis of example visualizations as well as a thorough reading to the *API*'s documentation. *mxGraph* is a fully client-side *JavaScript* diagramming library that uses *SVG* and *HTML* for rendering. The predefined process engineering shape library was used as a base for the next step: the creation of a statically defined, modular and composable *object-oriented* shapes library for *P&ID* elements. It was decided that this library was to be defined in *JSON* format, to facilitate user modification and tuning of the geometries, rather than in *XML* format like the visualization file itself.

**S3: Requirements and Design of Software Architecture**

After the creation of the statically defined *P&ID* shapes library file in *JSON* format came the conceptual elaboration of a preliminary software architecture for the project's technical implementation in the *MES Legato Sapient*. This task preceded the commencement of the agile development life cycle.

**S4: Boardlet Design**

Aligning to the *Legato Sapient* design and coding principles implemented in the component based *Ember.js* framework, the start of the development phase consisted in setting up boilerplate code for the *P&ID* Creator Boardlet. After the creation of both a *JavaScript (.js)* and *handlebars (.hbs)* templating file, the preliminary wireframe design for the boardlet was made and coded. Attaining to principles of component-based design, the *handlebars* template was designed and developed modularly with both new and reused *Ember* components. After having the boardlet up and running on the *Sapient Engine* it became possible to start the progressive development of the *business logic* for the automated *P&ID* visualization generation.

**S5: Generation of the XML file of the P&ID Visualization**

The first development sprint for the generation of the *XML* file of the visualization where made before establishing a database connection for fetching of the plant instances on a separate testing boardlet. This testing boardlet allowed for constant modification and experimentation of the algorithms with pure *JavaScript, HTML5* and *CSS*. These allowed for rapid coding without needing to be connected to the full sapient architecture. The plant hierarchy was first modelled statically in form of *JSON* files in place of the database queries which return equivalent *JSON* responses. The file input component for the uploading of the *P&ID* shapes library was recycled to directly load the needed files in the client, thus enabling faster trials and testing of the script in development until *XML* of unplaced, overlapping vertices was correctly generated.

**S6: Connection and Fetching of Plant Instances from Database**

After the script successfully and automatically generated an *XML* file of the *P&ID* from static *JSON* files of plant instances, the connection to the database and registering of database tables in the *Legato Configuration Center (LC2)* followed. This allowed to test the *XML* generation script now with actual plant data queried from the database. This required a global data map of all required tables and fields. With the data map, name mappings where done and a function to fetch the data with custom filters implemented via a Legato specific function call. Modification of the database queries could now be done only by modification of the passed parameters for the query. The result of the *XML* generation algorithm with the actual plant data corresponded now with what was originally modelled for the plant. By now, all vertices and edges where correctly instantiated in the diagram, but vertices where placed one on top of the other. This led to the start of the graphing algorithm for the placement of these vertices.

**S7: P&ID Graphing Algorithm**

The main task during this sprint was the development of a *vertex placement algorithm* to set the x and y properties of each vertex according to a defined set of positioning rules. Both a declarative, rule-based approach and an algorithmic approach for the optimization of area where used. First part of the algorithm consisted in the declaration of constraints in the form of tags based on shape attributes. This way, the positioning logic could be later better targeted at the distinct tags individually, since distinct shapes are to be positioned based on a distinct set of rules. The loosely-coupled tags where specified first and apart from the positioning logic, as this part was based on algorithmic optimization rather than classification. Afterwards, distinct sets of positioning rules where defined for each of the tags. The shapes would thus be iteratively positioned by the algorithm in a distinct and independent way. Furthermore, a *block-packing algorithm* was implemented for the positioning of groups in groups to minimize the area. Though much progress was made in a short time, a standstill was later reached. Though the algorithm could still have been made better, the time required to do so was too much compared to the results it would yield. Because of this lack of time, the algorithm was left as is, for the continuation of the bachelor thesis, or more specifically, with the next sprint.

**S8: Dynamic Real-Time Display of Process Variables in the P&ID Visualization**

Although the output of the *P&ID Creator* boardlet and the input for the *P&ID Viewer* is a static *XML* file of the *P&ID* Visualization, it must contain the *data bindings* for the dynamic display of the process variables. The *data binding* should be independently set based on the data type of the process variable via the `sapient-bind` attribute in each *XML* object. This logic had to be set before the string generation so that the distinct data types of the process values result in distinct labels or colors for each shape instance. The *Legato Graphic Designer* requires the `id` (*primary key*) of the value in the database and automatically fetches the value in the background whenever it changes. This functionality is implemented as a *mxGraph*

placeholders and allows for the *data bindings* to be also included in the static XML file of the P&ID visualization.

**S9: Prototypal Implementation and Evaluation in the Infrastructure of MES Legato Sapient®**

Although plant instances where already modelled and available from the database during the project, no connection to the gateway of the example *Aida brewery* plant existed. This prevented the testing of real time display of process variables with the systems, though the feature was indeed implemented, and thus, the requirement fulfilled. Still, the solution was implemented entirely on the *Legato Sapient infrastructure* and evaluated for feasibility for industrial production environments.

# 2 Technological Standpoint

## 2.1 Industrial Process Control

### 2.1.1 Overview

*Control engineering* is an engineering subfield which applies *automatic control theory* for designing systems with certain behaviors [2]. Figure 2 shows some applications of control engineering in different contexts. This thesis will focus on the application of control theory in industrial production contexts, more specifically, in industrial processes.

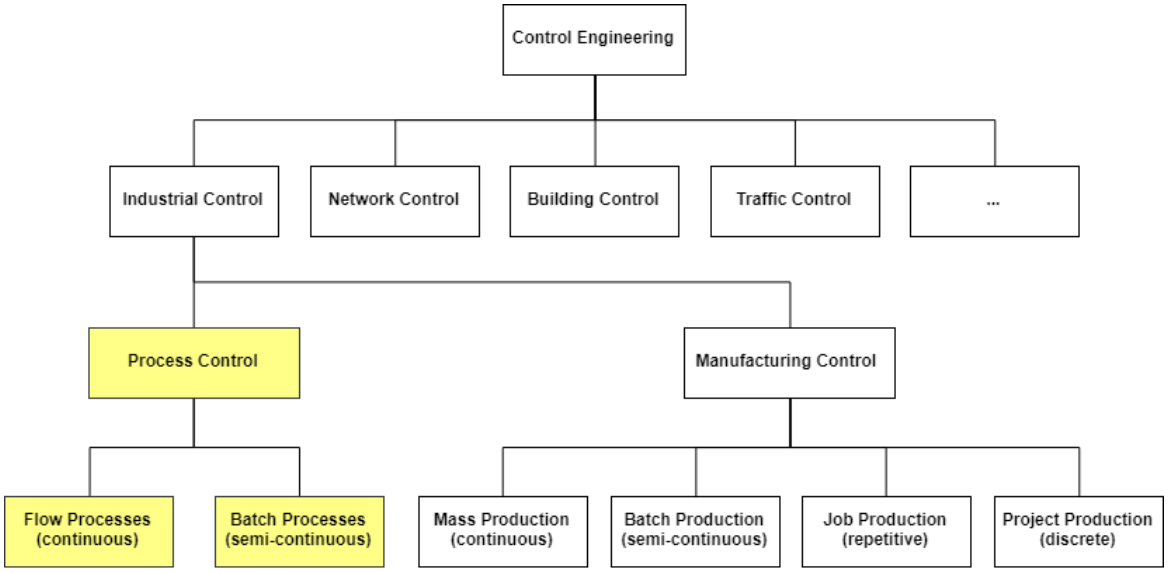



*Figure 4 Scope of the bachelor thesis within the applications of control engineering. Source: Chair AIS TUM*

A *process* is defined as a series of actions taken to achieve an objective. More specifically, an *industrial process* is the systematic application of mechanical and/or chemical operations for the production or manufacture of something.

*Industrial process control* is thus the systematic optimization of consistency, economy and safety of continuous production industrial processes. To achieve this, process control incorporates the fields of control and chemical engineering to automate production of *continuous* or *batch processes*. In contrast to *manufacturing control systems* which typically favor flexibility because of the heterogeneous and usually discontinuous nature of manufacturing, process control systems favor robustness, real-time reactivity and safety above flexibility. This way, process control systems are implemented to run non-stop for decades.

The size and complexity of industrial process plants generally represents a significant engineering challenge, but its long-lasting nature has historically taken the process industry to the vanguard in industrial production control.

*Process control systems* have traditionally been hierarchically structured. Nonetheless, the high degree of networking and the availability of substantial amounts of data has led the trend for their decentralization. Enabled by an interconnected network of *Cyber Physical Systems (CPS*s*)*, modern process control systems now allow for adaptive and self-configuring production. Figure 3 shows the trend for the decentralization of traditional automation into *Cyber Physical Systems (CPS)* for flexible, data-driven and efficient production.

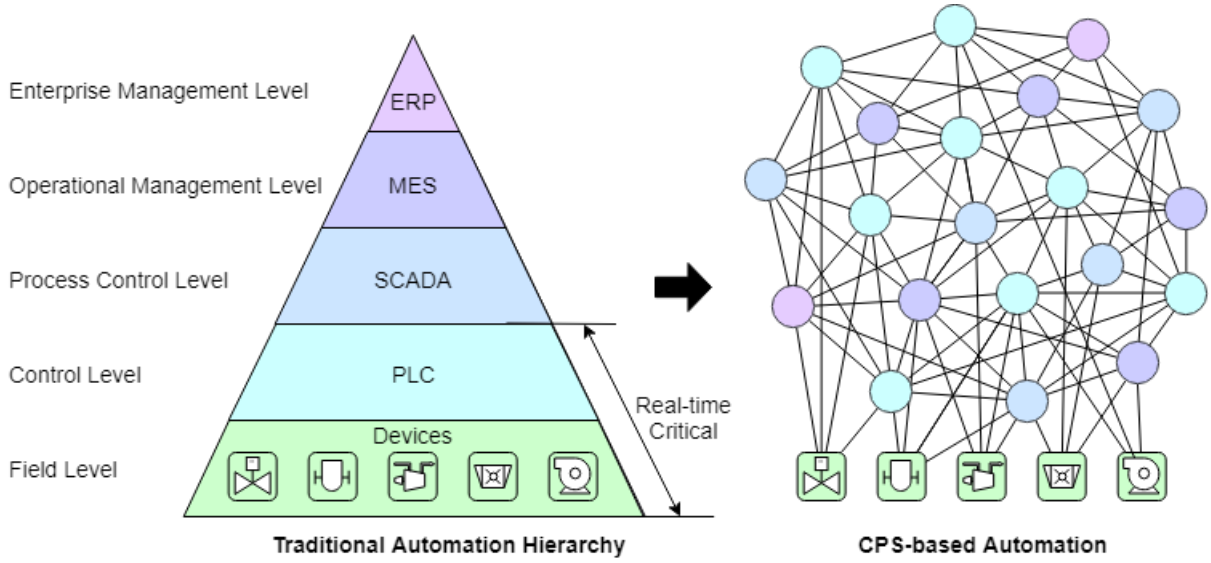



*Figure 5 Decentralization of traditional automation hierarchical structure via Cyber Physical Systems (CPS).*

### 2.1.2 Plant Hierarchy, Procedural Control and Process Models

The *Plant Hierarchy Model* or *ISA-95*, as it is more commonly referred to, is an international standard for developing an automated interface between enterprise and control systems [3]. With it an industrial organization can structured in a hierarchical manner, in which the starting root node is the hierarchy itself. The *Plant Hierarchy Model* described by the ISA-95 standard directly mirrors the physical model, and thus serves as a virtual clone of the actual physical instances. This is important for *industrial process automation* since the physical elements can then be abstracted into classes and programmed with an object-oriented paradigm. This results in huge benefits for the software engineers, since *object-oriented development* facilitates code reuse, enhances encapsulation and encapsulation and lessens need for code maintenance. Contiguous to the *Plant Hierarchy Model*, the *ISA-88* standard describes the *Procedural Control* and *Process Models*, particularly for process control – the scope for this project. Figure 4 associates all three mentioned models together, and highlights what the ProcAppCom project of this bachelor thesis encompasses.

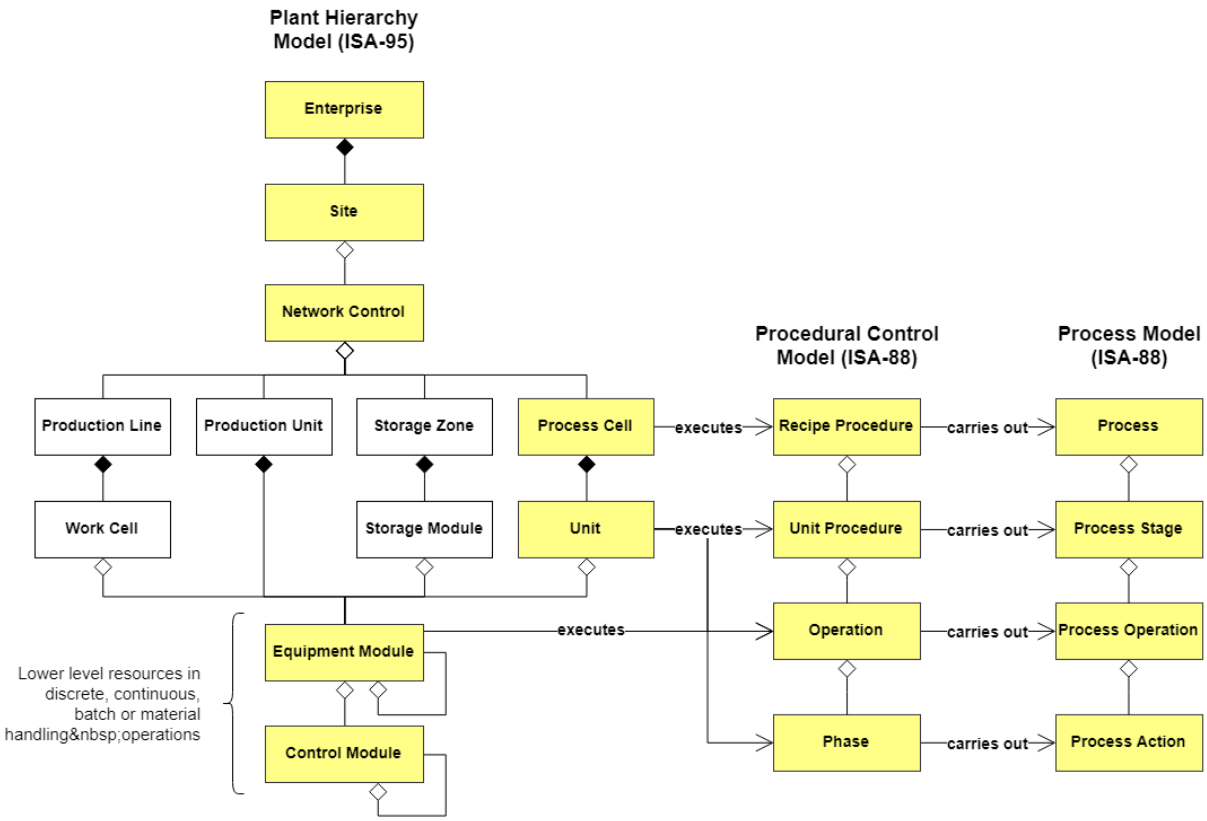



*Figure 6 Correlation of Plant Hierarchy Model (ISA-95) and Procedural Control and Process Models (ISA-88) with the scope of this project in yellow.*

### 2.1.3 Formalized Process Description

The interdisciplinarity required in the field of *industrial process engineering* requires engineers of distinct backgrounds to work together. The source of information for *industrial control systems* in the field is thus typically distributed across chemical, control, electrical, mechanical and software engineering disciplines. This results in the need for a *formalized process description*, or a standardized description from all process kinds [4]. The goals of this standard, as described by the VDI 3682 standard include: universal applicability for all kinds of processes and plants (including non-technical ones), to serve as a clear (visual) aid for interdisciplinary understanding, to be formalized to prevent misunderstandings, and to serve as basis for the plant models. Discipline-specific examples of formalized process descriptions for an example brewery could be:

- Chemical: batch process models, chemical reaction equations, etc.
- Control: control theory block diagrams, etc.
- Electrical: electrical layout plans, etc.
- Mechanical: *computer aided designs (CAD)*, plant floor plans, piping plans, etc.
- Software: class diagrams, activity diagrams, entity relationship maps, etc.

Due to the high degree of interconnectivity of the information itself, it makes sense to have an interface for all information sources, rather than separate documents for each. The following shows the stages of the *formalized process description* for industrial process control in ascending order of concretization:

1. Graphical representation of the process (process-oriented)
2. Information model of the process objects and their relationships (information-oriented)
3. Objects attribution by identifiers and characteristics (process-oriented)
4. Administration of attributes (information-oriented)
5. Export and import procedures with computer aided engineering (CAE) systems (information-oriented)

The *ProcAppCom* project comprised all five stages for the abstraction of the example *Aida brewery* plant. However, the following discipline-specific flow diagrams for industrial process control are especially relevant to this bachelor thesis. Listed in ascending order of concretization, these are:

- **Basic Flow Diagram (ISO 10628):** an abstract representation of the process steps, operations, plants and plant parts in the form of a block diagram
- **Process Flow Diagram (ISO 10628):** a more concrete representation of a process or plant characterized by equipment and apparatus
- **Piping and Instrumentation Diagram (ISO 3511):** the most detailed representation of procedures and equipment with description of *process control engineering (PCE)* tasks.

This project deals with the generation of the third and most concrete *process flow diagram* and is therefore handled in more detail in the next section.

### 2.1.4 Piping and Instrumentation Diagram (P&ID)

A *Piping and Instrumentation Diagram (P&ID)* is the most detailed flow chart used in the process industry for the description of processes and process plants. A *P&ID* shows the piping, equipment, instruments, control systems and connections of an industrial process plant in a relatively structured manner and is therefore a vital interdisciplinary document for chemical, control, electrical, mechanical and software engineers alike.

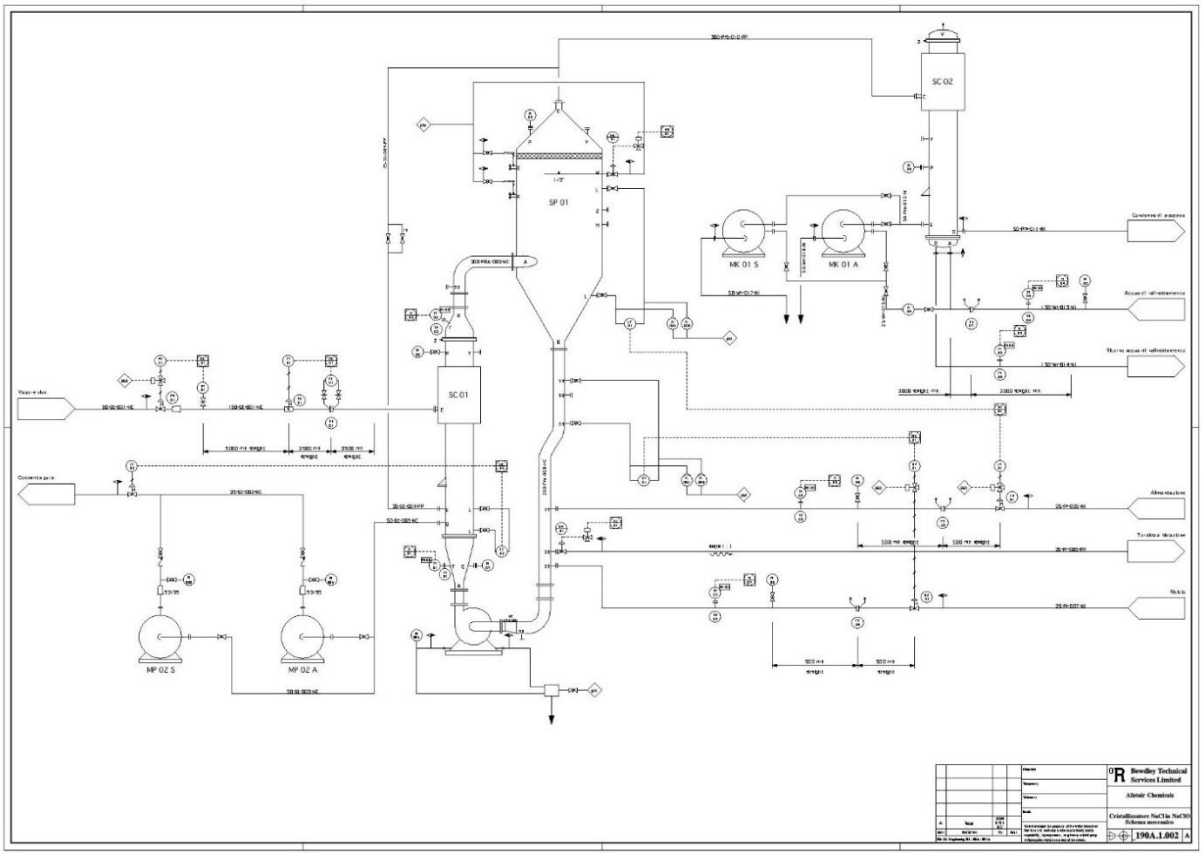



*Figure 7 An example Piping and Instrumentation Diagram of an evaporative crystallizer. Source: Wikipedia*

**Contents**

In spite of the formalized nature of *P&ID* diagrams, no exact standard for how to draw *P&IDs* exists. The *Process Industry Practice (PIP)*, a consortium of contractors and owners in the process industry indicates in their PIC001 document, what a *P&ID* should contain [5]. Nevertheless, due to space and complexity constraints, it was determined that the P&IDs to be generated within this project should only contain the points marked with a check-mark from the following taken from the PIC001 specification:

- ✓ Mechanical equipment with names and numbers
- ✓ All valves and their identifications
- ✓ Process piping, sizes and identification
- ▪ Miscellaneous - vents, drains, special fittings, sampling lines, reducers, increasers and swagers
- ▪ Permanent start-up and flush lines
- ✓ Flow directions
- ✓ Interconnections reference
- ▪ Control inputs and outputs, interlock
- ▪ Seismic category
- ▪ Interfaces for class changes
- ▪ Quality level

- Annunciation inputs
- Computer control system input
- Vendor and contractor interfaces
- Identification of components and subsystems delivered by others
- Intended physical sequence of the equipment
- Equipment rating or capacity

*P&ID* visualizations have the advantage of being simple whilst having the ability to further refine and improve them. On the other hand, a major disadvantage are the inconsistencies that exist between industries and companies due to a lack of promulgation of the standards.

**Functions**

The functions of a *P&ID* are essential in the development, modification and maintenance of processing facilities. According to [5], these include:

- Produce documents that explain how the process works
- Develop guidelines and standards for facility operation
- Serve as a basis for control programming
- Design a conceptual layout of a chemical or manufacturing plant
- Form recommendations for cost estimates, equipment design, and pipe design
- Provide a common language for discussing plant operations
- Create and implement philosophies for safety and control
- Evaluate construction processes

**Symbols and Notation**

Contrary to the contents of a *P&ID*, the symbols and notations used in *P&IDs* are vastly standardized. The instrumentations symbols serve to understand the data presented in the *P&ID* and should adhere to the ANSI/ISA S5.1-1984 (R 1992) standards. This ensures that communication of instrumentation, control and automation is consistent and system independent, for its interdisciplinary understanding. Figure 6 shows the four graphical elements defined by ISA S5.1: discrete instruments, shared displays or controls, computer functions and *programmable logic controllers (PLCs)*:

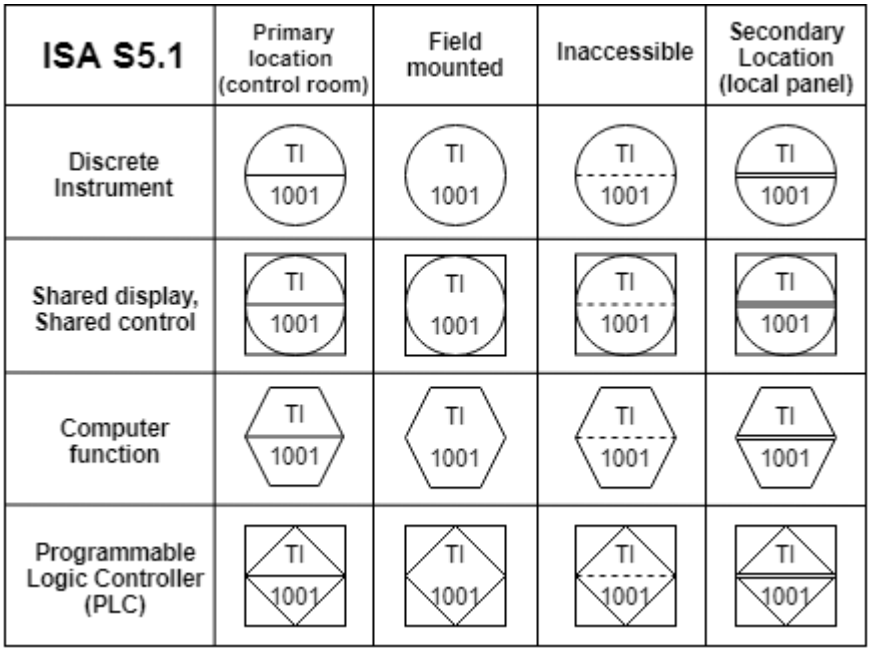



*Figure 8 General P&ID instrument or function symbols according to ISA S5.1*

## 2.2 Manufacturing Execution Systems

### 2.2.1 Overview

A *Manufacturing Execution System (MES)* is a software system used in manufacturing for the tracking and documentation of industrial production from raw material to finished products. A *MES* is responsible for providing valuable information for the management and control of manufacturing by working real time to control the multiple elements of the production process like machines, personnel, support services, process variables, amongst others. A *MES* sits between *Enterprise Resource Planning (ERP)* software and *Supervisory Control and Data Acquisition (SCADA)* systems in the automation hierarchy and is capable of real-time, field-level process control as well as some resource planning functionalities.

### 2.2.2 Functions

According to the *VDI (Verein Deutscher Ingenieure)* and *ISA (International Society of Automation)*, a *MES* encompasses the following main functionalities:

- Material/product stock management
- Data collection
- Performance analysis
- Order planning and control
- Quality management
- Information management

- Human resource management

### 2.2.3 Legato Sapient®

*MES Legato Sapient*® revamps the traditional, platform specific and sometimes antiquated *MES* software solutions still widely used in the industry with the *Legato Web Application*: an entirely web-based, flexible and scalable software solution to keep up with the industry 4.0 requirements and fast-moving environment. *Legato Sapient* covers a wide range of applications and is designed modular. Stand-alone functional modules are available for all key *MES* functions previously mentioned. This allows user implementations to be configured individually, module by module, up to a complete production management system.

**Features**

Unique to the *Legato Sapient* product are its modular implementation of basic functionalities of *MES*, called *Legato Efficiency Boosters* or *EB*s for short. These can be activated just in time and not only upfront, what allows *Legato* to grow together with its requirements. The following overview lists the available *efficiency boosters* according to *Gefasoft*:

- **EB Work Time Models:** to define shifts and breaks for a production area
- **EB Alarm Management:** for recording, archiving and further processing of alarms
- **EB Key Performance Indicators:** provides machine-related *KPIs* like production time, failure times, piece counters, availability, *overall equipment efficiency* (*OEE*), *mean time to repair (MTTR)* or *mean time before failure* (*MTBF*), etc.
- **EB Machine States:** *Machine Data Acquisition (MDA)* based on configurable machine states to aid in machine management
- **EB Document Management:** to administrate machine-related documents like *NC* or *PLC* programs, manuals, *CAD*-drawings, maintenance instructions, etc.
- **EB Measurement Controlling:** supports measurement planning after SixSigma DMAIC method (define, measure, analyze, improve, control)
- **EB Shift Logs:** to dispose digital notes to production and document them
- **EB Best Cycle Time:** to measure cycle times and calculate the *best cycle times (BCT)* for selectable stations or components
- **EB Energy Management:** to measure and calculate energy consumptions
- **EB TPM and Spare Parts:** *total productive maintenance (TPM)* to support preventive maintenance
- **EB Web Visualization:** to create, visualize and maintain *SVG* based graphics
- **EB Web Reporting:** for integrated *business intelligence (BI)* reports using *BIRT* library

- **EB Supervisory Station:** provides a *Gant diagram* display of operation sequences
- **EB Tracking and Tracing:** identifies and tracks *production units (PU)* like parts, lots and batches, depending on production system
- **EB Work Time Management:** to record and manage employee work times
- **EB Recipe Management:** to manage machine parameter sets and recipes
- **EB Quality Management:** provides quality guidelines, levels and checklists for quality control
- **EB Product Stocks and Logistics:** for efficient planning of warehouses, warehouse locations, stocks, stock movements, etc.
- **EB Mobile Ap – Legato Info:** to access real-time information for management and maintenance anytime and anywhere (supports *iOS*, *Android* and *Windows phone*)

**Hardware Architecture**

Legato Sapient® permits software components to be run either on Windows, Linux or Unix *Operation Systems (OS)* with the exception of OPC servers in the *Legato Database Gateway*. Additionally, processes like the *Legato Web Application* or the *Legato Application Engine* can be operated through virtual environments. The scaling up of an application to hundreds of web users and thousands of data sources (from *PLCs* for example), common to the automotive industries can be achieved by splitting the software processes to different server platforms: 1 to many communication servers with the *Legato Database Gateway*, a database server and an application server with *Legato Web Application* and the *Legato Application Engine* [6].

**Software Architecture**

The entire *Legato* software architecture is based on a central *relational database* to store both configuration and process data. The *Legato Database Gateway* manages the real-time connections to field devices by means of distinct interface technologies and flexible project specific scripting solutions. The *Graphical User Interface (GUI)* is entirely web-based as it is built up entirely using *HTML5*, *CSS3* and *JavaScript*; The *Legato Web Application* provides the views for the *GUI* in form of boardlets which build up customizable and modular dashboards and are implemented with the *Ember.js JavaScript* Framework and the its component model. *SOAP (Simple Object Access Protocol)* web services are used to integrate web app and other shop floor IT systems. This means that a standardized web browser is the only requirement for the basic functions of a Legato client. The *Legato Application Engine* runs distinct jobs for different tasks like data management, importing and exporting data and for data processing and calculation of *Key Performance Indicators* (KPIs). Additionally, the *Legato Sapient API* provides user-specific and tailored solutions by providing a simple interface for the interconnection of customer boardlets with other boardlets, the database, and other standard elements.

## 2.3 Web Applications

### 2.3.1 Overview

The *World Wide Web* was originally conceived as an information space for the distribution of static documents and resources interlinked by hypertext links and accessible via the internet [7]. The web has since then been critical for the development of the information age. Initially, web content was limited to being static, simple text documents navigable via embedded hyperlinks. The web browser was nothing more than a search engine, hence the name "*browser*". Nevertheless, with the later development of scripting languages like *JavaScript*, code could be programmed to run directly in the client's browser. This empowered web browser to dynamically generate and update content in the client-side, what lead the shift to the development of web applications.

A *web application* is a full client-server computer program with a user interface and client-logic running in the web browser. Although initially limited by the web browsers capabilities, new *Application Programming Interfaces (APIs)* have been developed which empower the browser with capabilities that were previously exclusive to native applications. These *APIs* allow web apps to leverage and consume services like *Bluetooth*, the camera, the local file system, sensors and accelerometers, *GPS (Global Positioning System)*, amongst many others. Web and native are thus at par regarding many key functionalities for application development. Furthermore, the cross-platform nature of the web platform is a major advantage of developing web and not native applications. While web applications require only to be programmed once to run in all devices with a web browser and an internet connection, the code for native applications is usually platform-specific. By virtue of the recent development of the web platform, more and more applications are being advantageously developed for the web. The *MES Legato Sapient* exemplifies how companies today leverage these benefits to build robust, industrial grade web applications.

### 2.3.2 List of Technologies

The accelerated technological evolution in recent years has brought significant changes to the web platform. Meanwhile, web application technologies have had to swiftly adapt to not lose ground against native application technologies. Driven by the constant evolution of the web platform, the amount of web application technologies has dramatically increased in parallel. The following list covers the concepts and technologies employed for the technical implementation of this project.

**HTML**

*Hypertext Markup Language (HTML)* is the standard markup language for the creation of web content. *HTML* elements or tags are the building block of the web and are used to declaratively define the structure and content of the web page.

### CSS

*Cascading Style Sheets (CSS)* is a style sheet language used for describing the presentational aspect (layout, colors, fonts, etc.) of content defined in a markup language like *HTML* [8]. *CSS* enables the separation of style from content and allows for sharing of styles throughout multiple web pages. *CSS* has selectors to target specific *HTML* elements and style rules which follow a priority scheme to determine the styles to be applied for the selected elements.

### JavaScript

*JavaScript (JS)* is a dynamic, weakly-typed, high-level interpreted programming language. Along with *HTML* and *CSS*, *JavaScript* is one of the three core technologies of the *world wide web* [9]. *JavaScript* enables web pages to be interactive and is thus essential for web applications. Being a multi-paradigm language, *JavaScript* allows for event-driven, functional, imperative and object-oriented programming [10]. Although it was initially developed for the *front-end* as a client-side implementation for the web browser, it has now been extended to support *back-end* implementation for web servers and database development.

### XML

*Extensible Markup Language (XML)* is a textual data format and markup language for encoding documents in a human- and machine-readable format. Although *XML* was specifically designed for documents, the language is also widely used for the representation of arbitrary data structures such as those used in the web [11]. *XML* documents use user defined tags and implementations can be specified with the use of any of the available schema systems. Moreover, many *APIs* exist for distinct programming languages for working with *XML*. In this project, the *XML* format is used to define the generated *P&ID* visualization, or more specifically the data structures that define it.

### JSON

*JavaScript Object Notation (JSON)* is a lightweight data interchange format used to represents arbitrary data structures as data objects consisting of attribute-value-pairs and array data types [12]. The *JSON* format is commonly used for asynchronous browser-server communication and has replaced *XML* in many applications. *JSON* is a language independent data format, nevertheless, it was derived from *JavaScript* and thus is syntactically similar to a common *JavaScript* object. This, its high readability, amongst other advantages facilitate working with data structures in *JSON*, reason for which it is the chosen data format for the *P&ID* shapes library of this project.

### Scalable Vector Graphics (SVG)

*Scalable vector graphics (SVG)* are an open-standard *vector-image* format developed by the web to represent two-dimensional interactive and animable graphics in an *XML*-based format. *SVG* allows for

three types of graphics: vector graphic shapes such as outlines and paths, text and bitmap images. *Vector-graphics* enable geometries to be defined in mathematical terms, instead of as a *bitmap* of pixels, which in turn keep the image quality when scaling and zooming. The standard supports the grouping of graphical objects, as well as linking, animation, font-selection, metadata, filter-addition, amongst others, all of which can be directly modified in standard text editors. All major browsers support *SVG* making it a preferred format for high-quality graphics in the web.

**mxGraph and Draw.io**

*mxGraph* is an open-source diagramming library for the rapid creation of interactive charts and diagrams that run natively in web browsers. The *mxGraph API* exposes a number of functionalities for the implementation of the library and is available in various programming languages, including *JavaScript, Java, PHP* and *.NET*. The *API* provides a robust package of high-level methods and services for integration in web applications. Built using the *mxGraph* library, *draw.io* is a visual diagramming tool with rich functionality (that of *mxGraph*) for the creation of fully integrated charts and diagrams in other popular software platforms like *Google GSuite*, *Confluence* and *Jira*.

**Web Development Frameworks: Ember JS**

*Ember.js* is a *front-end* JavaScript framework that facilitates building websites with rich and complex user interactions. *Ember.js* provides developers with features to manage complexity in modern web applications, as well as an integrated development toolkit for rapid development and iteration. Features of the framework include: routing to drive the application state via common *URLs*, a data layer to manage application state and provide a consistent way for external *API* communication, a Handlebars based templating engine, a robust *Command Line Interface (CLI)* toolkit to create, develop and build ember applications, among others. Built on the principle of component-driven design and development, the *Ember.js* framework facilitates the creation of modular, encapsulated components of code. This easies and promotes code reuse as well as a modular front-end design which tends to be essential in the long run, especially for larger products.

**Relational Database: PostgreSQL and SQL**

A *relational database* is a collection of data structured based on a *relational model* of the data [13]. The *relational model* organizes the data into one or more tables of columns (or attributes) and rows (or records) [14], with a unique key to identify each row. Normally, each table represents a distinct type of entity, like a student or an exam, and rows represent individual instances of that entity, like “Mike” or “Math”. In relation to object-oriented programming, it can be said that tables are analogous to classes, rows to objects and columns to the object’s attributes, with fields being the specific values of those attributes for the corresponding object.

Normally, *structured query language (SQL)* is used to manage a relational database. *SQL* is a domain-specific language designed manage data through *relational database management Systems (RDBMS).*

Features of the language include data definition, data manipulation (update, insert and delete), data query and data access control. *SQL* defines certain statements for working with the data at hand using relational algebra and tuple relational calculus. A typical *SQL* statement or query consist of clauses like `SELECT`, `UPDATE`, `SET`, `WHERE`, expressions and predicates and allow to retrieve and persist changes to data.

The database at hand for this research product is implemented in *PostgreSQL*. *PostgreSQL* is an open-source *object-relational database system* with an extensive and powerful set of features for developers. The *PostgreSQL* database and an available *Legato*-specific database *API* were available before the start of the project. More on the database implementation for the context of this project will be discussed later.

**Data transmission: AJAX and JSON API**

The *Fetch API* provides a simple interface for the fetching of resources, including across the network [15]. The *Legato Engine* implements its own high-end functionalities to query the database via the *Fetch API* in the form of *Asynchronous JavaScript And XML (AJAX)* requests that return *JSON* objects. These return objects are specified by the *JSON API* to ensure best practices. The *JSON API* is responsible for the definition of standards and best practices for data exchange in *JSON* format [16]. It specifies how a client should request to fetch or modify resources, and how a server should respond to those requests. The correct use of *JSON API* can minimize the number of requests and the amount of data transmitted between clients and servers.

## 2.4 Unified Modelling Language (UML)

The *Unified Modelling Language (UML)* is a graphical description language for specification, visualization, construction and documentation of systems. The use of standardized graphical modelling languages like *UML* increase the clarity and thus understanding of the model even across different engineering disciplines. The different types of *UML 2.0* diagrams can be classified into two categories: structure models and behavior models and further classified as Figure 8 shows.

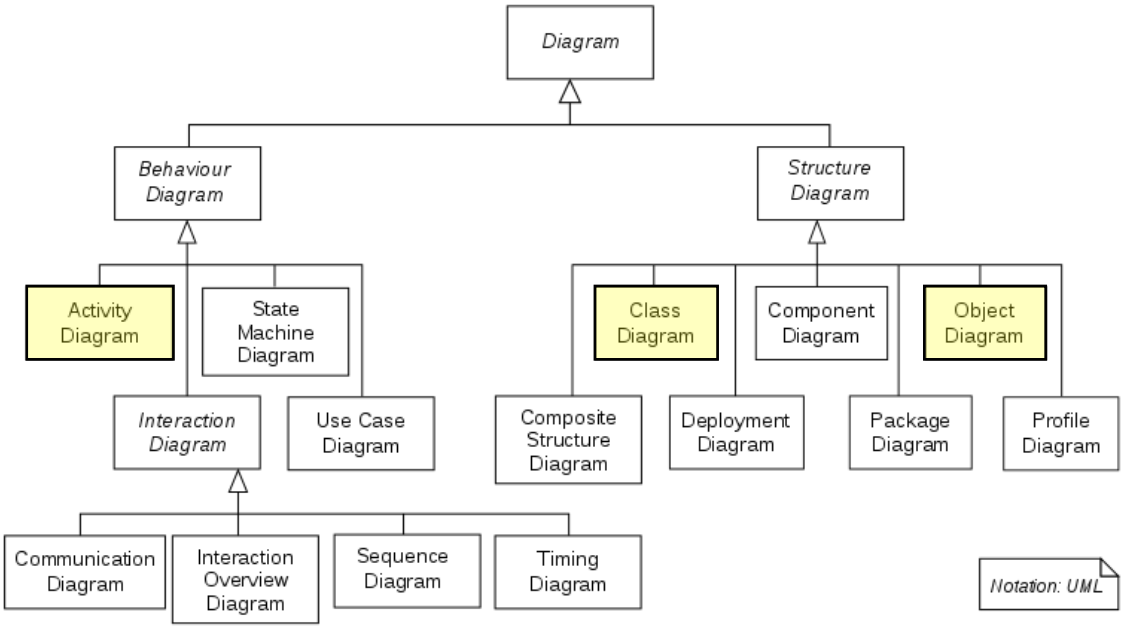



*Figure 9 UML 2.0 diagram hierarchy shown as a class diagram, with graph types used throughout this writing in yellow.*

Apart from the standard *UML 2.0* implementation, the standard allows for its extension by means of new element declarations using stereotypes and constraints to ease domain specific implementations. In production automation systems (both process and manufacturing), extension profiles like *SysML (Systems Modeling Language)* and *UML-PA (UML for Process Automation)* serve to model the system description. The system description can be further broken down as follows [16]:

- **Plant structure:** component hierarchy, topology, component relationships
- **Plant components:** Mechatronic structure, properties/parameters, economic data
- **Networks**: electrical construction, communication systems
- **Behavior:** component behavior, control system design, robot processes
- **Geometry and Kinematics:** mechanical construction, motion planning, electrical construction

This enables *UML* to directly address the specific modelling requirements for systems of distinct nature. Throughout this document, following *UML 2.0* and extension diagram types will be used, some of which with slight modifications and simplifications:

### 2.4.1 Class Diagram

*Class diagrams* are a type of static structure diagrams for the description of systems in the form of classes, their attributes, methods, and the relationships to other classes. A *class diagram* can be considered the main block of object-oriented programming, as it abstracts systems into detailed conceptual models which can be used to generate programming code. *Class diagrams* can also be used for data modelling and are thus crucial for the structuring of a system into its corresponding data model.

### 2.4.2 Object Diagram

An *object diagram* is a graph of instances, including objects and data values. A static *object diagram* is thus an actual representation of the state of the system at any point in time, encompassing both objects and relationships at that moment in the system. Although similar to class diagrams in appearance, an object diagram further serves software developers to examine specific iterations of a more general system and consequently, to achieve a detailed overview of the system at the object level, rather than the more general class level. *Object Relational Mapping (ORM)* is facilitated by virtue of this diagramming method. *ORM* systems serve to convert data between incompatible type systems using *object-oriented* code. In this project for instance, the data in the *relational database* tables had to be mapped to object instances for its manipulation with code. An *object diagram* was used correspondingly to define the data model, which would later be mapped using *JavaScript* at runtime to *JavaScript*-compatible object instances.

### 2.4.3 Entity Relationship Diagram

*Entity Relationship Diagrams (ERD)* are a type of flowchart used to map how entities relate to each other within a system. In software engineering, *ERDs* are most commonly used to design and debug *relational databases*. This type of diagram defines a set of graphical symbols to depict entities with their attributes and relationships in which entities are usually expressed nouns and relationships as verbs. Several notations of such graphical symbols exist; In this writing, a simplified version of the *Crow Foot style for Information Engineering* will be used.

### 2.4.4 Activity Diagram

*Activity Diagrams* are a subset of *UML* behavior diagrams used to describe the dynamic aspects of a system's workflow as a set of step-by-step activities or actions. As such, activity diagrams are key in the representation of algorithm logic, to model software architecture elements like methods and functions, and in clarifying complicated use cases for the simplification and improvement of processes. They consist of actions, decision nodes for conditional divergence of flow, control flows, and a start and end node.

## 2.5 Graphing Algorithms

### 2.5.1 Graph Theory

*Graph theory* stems from discrete mathematics and is the study of *graphs*, mathematical structures used to model pairwise relations between objects [18]. Graphs have many applications, most notably in modeling of social, biological, physical and information systems [19].

**Graph**

A *graph* can be mathematically described as an ordered pair G = (V, E) composed of a set of *nodes* or *vertices* V and a set of *lines* or *edges* E, which are two-element subsets of V, since each edge is associated to two vertices. The edges of a graph can be either *directed* or *undirected*, and a vertex's degree describes the number of edges connected to it. The sets V and E are usually finite, and the *order of the graph* is its number of vertices |V| while the *size of the graph* corresponds to its number of edges |E|.

Graph edges can have *cycles*, and edges can have either qualitative or quantitative data types called *values* or *weights* on them. Figure 9 further categorizes the types of data an edge can acquire as values. A *Tree* describes a subset of general graphs with no cycles, directed edges and a specially designated, single root vertex.

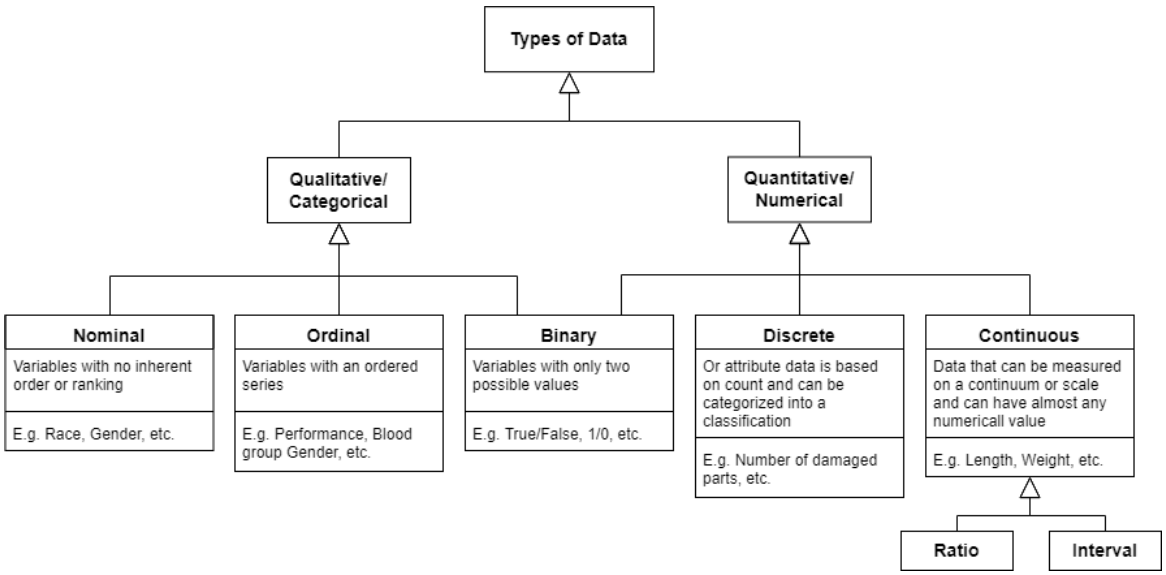



*Figure 10 Class diagram of possible types of data an edge can acquire. Source: sixsigma-institute.org*

**Adjacency**

The *adjacency* of a graph is the representation of the graph itself, including vertex instances along with their relationships to other vertices, from which edges can be derived. *Adjacency* representations support both directed and undirected graphs and can be expressed in graph form, as *node-link diagrams*, in matrix form, the *adjacency matrix*, or in list form, an *adjacency list*, all of which contain the same information, each specific for distinct applications. *Adjacency* of the *P&ID* visualizations required for this project was for instance read from relational database tables via a parent attribute and parsed into an *adjacency list* to pass later to the graph layout algorithm for its generation.

### 2.5.2 Types of Graphs

The flexible nature of graphs results in the possibility of breaking them down into countless different types, some of which are trivial for most cases, and most of which attain to no particular standard in their

nomenclature. Rather than defining all graph type possibilities, only those relevant to this project's scope and context are considered. The categorization is split into two subcategories based on either their relationship characteristics or *layout heuristics*.

**Based on Relationship Characteristics**

Based on the relationships or more specifically the nature of their edges, graphs are broken into:

- **Undirected Graph:** a graph with edges with no direction between vertices
- **Directed Graph:** a graph with directed edges, which connect vertices in a specific way
- **Weighted Graph:** a graph in which each edge is given a numerical value or weight to represent distinct things like costs, lengths or capacities. Although usually numerical, weights can be nominal, ordinal or quantitative

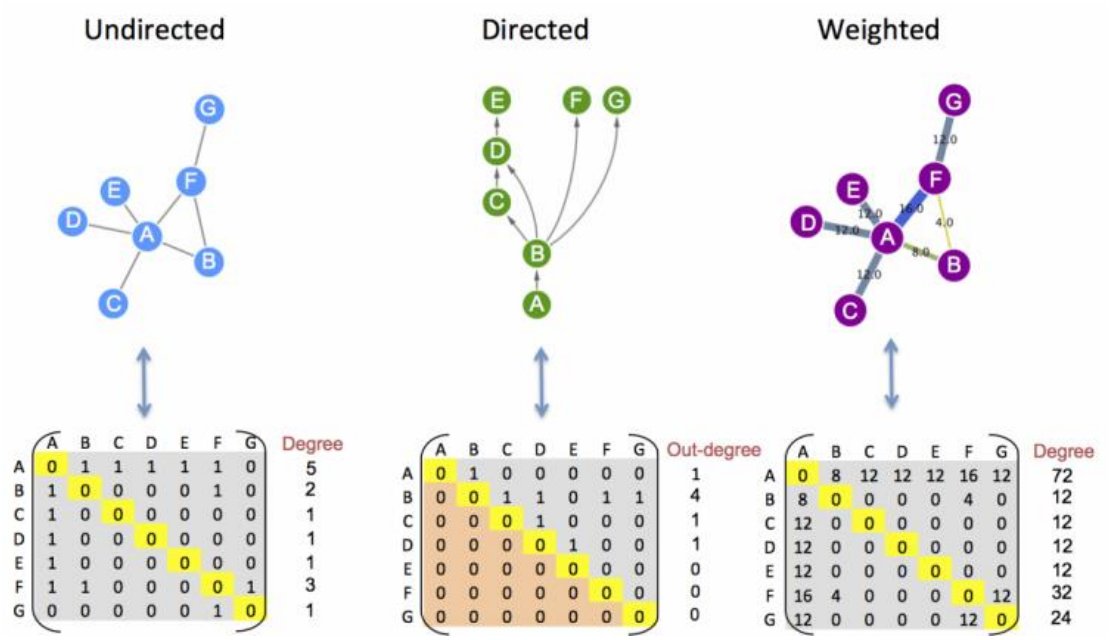



*Figure 11 Types of graphs (categorized based on their relationships) relevant to this project and their corresponding adjacency matrices. Source: EMBL-EBI*

**Based on Layout Heuristics**

*Layout heuristics*, or how the graphs are laid out, significantly influence the visual appearance of graphs. Based on *layout heuristics* and with focus on the scope of this thesis, *P&ID* visualizations shares similarities to the following graph types:

- Planar Graph
- Grid-based Graph
- Orthogonal Graph
- Hierarchical Graph
- Horizontal/Vertical Flow Graph

- Horizontal/Vertical Tree Graph
- Circular Graph
- Organic Graph

### 2.5.3 Graph Drawing in Software

*Graph drawing* is a field of math and computer science for the conception of two-dimensional depictions of graphs. In software, this can be done by combination of existing graph layout algorithms for the defining of the graph into its visual representation. Although many algorithms exist for the efficient drawing of graphs, a number of aspects directly proportional to the size of the graph must be considered and several challenges remain [20]. This include:

- **Graph layout and positioning:** which consists in making a concrete render of an abstract graph or mathematical representation. This is a particularly complex task, since vertices can have any degree (any number of connected edges) and can thus quickly become computationally expensive. Further sub-challenges include:
- **Rank Assignment:** compute which nodes have larger degree to place them at center of clusters
- **Crossing Minimization:** swap nodes to rearrange edges in favor of minimizing crossings
- **Subgraph Extraction:** ability to identify and pull out clusters of nodes
- **Planarization:** pull out a set of nodes that can lay out on plane
- **Scaling:** challenging for large graphs which cannot fit vertices and edges into screen space. Scaling can also significantly slow down an algorithm
- **Navigation and Interaction:** how to support user moving around graph and changing focus without entailing a new render
- **Vertex issues:** defining the shape, color, size, location and label for vertices of different types, with different positioning, etc.
- **Edge issues:** defining the shape, color, size, label, form, if polyline, straight line, orthogonal, etc.

Additionally, graph drawing involves the following complexity considerations:

- **Edge Crossings**: to minimize edge crossings towards a planar graph layout
- **Total Edge Length:** to minimize towards proper scaling
- **Area:** to minimize for efficient use of space
- **Maximum Edge Length:** to minimize the longest edge for compactness
- **Uniform Edge Length:** to minimize variances in lengths of edges for uniformity
- **Total Bends:** minimize orthogonal bends in favor or straighter lines for clarity

Several studies have found that of the mentioned complexity factors, what seems to be most important for the optics of the visualization is the minimization of edge crossings (Purchase, Graph Drawing ’97, Ware et al, Info Vis 1(2), June ’02 and Ghoniem et al, Info Vis 4(2), Summer ’05).

### 2.5.4 Graph Layout Algorithms

Likewise, many graph layout algorithms for different graphing types exist. With respect to this project, the following are relevant and where used as reference in development of this project’s *P&ID* graph layout algorithm:

- **Grid Layout Algorithm:** consists in placing vertices on a constrained two-dimensional grid
- **Tree Layout Algorithm:** consists in traversing the tree hierarchy in order (for example by using a *depth-first* or *breadth-first algorithm*) and setting the vertices in that order (either vertically or horizontally)
- **Force Directed Layout Algorithm:** models graph as a set of masses connected by springs, and simulates this *mass-spring system* for the layout or separation of the vertices
- **Planar Layout Algorithm:** draws graph by laying vertices out in order to avoid edge crossings. A graph is said to be *planar* if it can be properly drawn in the plane with no edge crossings.

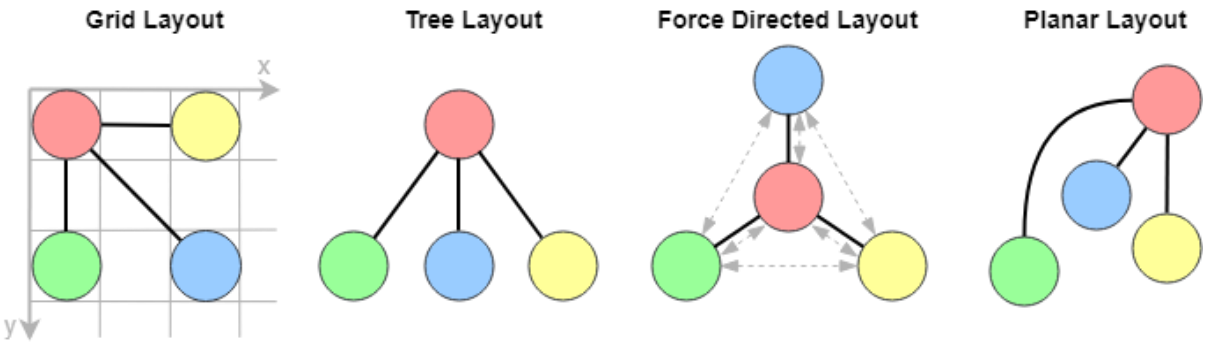



*Figure 12 A simple graph G=(V,E) with |V|=4 and |E|=3 as laid out using the four main graph layout algorithm strategies relevant to this project.*

### 2.5.5 Graphs and P&IDs

The *P&ID* visualizations required for this project borrow characteristics from all graph types mentioned, nevertheless, *tree* structures are the best match. *Trees* represent a subset of a *general graph* with no cycles and a single, designated root vertex, typically only with directed edges. *Trees* are analogous to the *plant instance hierarchy* modelled by the *SysML* model and translated to the *relational database* in form of flat tables. Despite this translation, the intrinsic hierarchical structure of the model remains in the flat database tables in the form of parent and children attributes. Being a *tree* structure what best embodies the structure

of the *P&ID* visualization to be generated, the *tree layout algorithm* heavily influenced the developed algorithm.

## 2.6 Related Works

### 2.6.1 Overview of Related Works

The topic of automatic visualization generation is much investigated, and many concepts exist for different purposes, nevertheless, not many exclusively targeted at industrial process applications. Consequently, no one research nor practical project was identified, which coincides with the majority of this project's requirements entirely. Following is a list of related works which do indeed assimilate some of this project's requirements outlined in section 1.5 and listed again below in table 2.

### 2.6.2 Comparison of Related Works

The technical and conceptual requirements to be addressed for this project compare as follows to the previously listed set of related works. The requirements were regarded with a more general perspective (underlined part) so that they matched better, rather than with a narrower focus on P&ID visualizations exclusively:

**R1** Library of modular P&ID Visualization Components According to Industry Standards

**R2** User Friendly Graphical User Interface (GUI) Boardlet for Creation of P&ID Visualizations

**R3** Client-Side Script for the Automated P&ID Visualization Generation as an XML File

**R4** Mapping of Physical Plant Instances to Corresponding Visualization Component

**R5** Automatic Type Detection and Simplification of Connections

**R6** Declarative specification of Graphing Constraints in Form of Tags

**R7** P&ID Graphing Algorithm

**R8** Dynamic Real-Time Display of Process Variables in the P&ID Visualization

**R9** Prototypal Implementation in the Infrastructure of a MES (Legato Web Application) and Documentation

| Key | Title |
|---|---|
| **W1** | An Automated Generation Approach of Simulation Models for Checking Control/Monitoring System [Prat, 2017] |
| **W2** | Automatic Model Generation for Virtual Commissioning based on Plant Engineering Data [Oppelt, Wolf, Drumm, Lutz, 2014] |
| **W3** | A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications [Yang, Prasad, Xie, 2013] |
| **W4** | Object-oriented engineering data exchange as a base for automatic generation of simulation models [Barth, Strube, Fay, Weber, Greifeneder, 2009] |
| **W5** | Automated Generation of Modular and Dynamic Industrial Process Plant Visualizations in a Manufacturing Execution System (MES) [Romero Karam, 2018] |

*Table 1 List of related works*

| Work | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 |
|---|---|---|---|---|---|---|---|---|---|
| **W1** | ✓ | ○ | ○ | ✓✓ | ✓ | ○ | ○ | ○ | ○ |
| **W2** | ✓ | ✓✓ | ✓ | ✓✓ | ✓ | ✗ | ○ | ✓ | ✓✓ |
| **W3** | ○ | ○ | ✓✓ | ✓ | ○ | ○ | ○ | ✓ | ○ |
| **W4** | ✓✓ | ✓ | ✓✓ | ✓ | ○ | ○ | ○ | ✓✓ | ✓ |
| **W5** | ✓✓ | ✓✓ | ✓✓ | ✓✓ | ✓✓ | ✓✓ | ✓ | ✓ | ✓✓ |

*Table 2 Summary of related works based on the addressed requirements.*

✓✓ – Implemented (very relevant)
✓ – Addressed (relevant)
○ – Not addressed (not relevant)
✗ – Not implemented (but relevant)

In general, several projects were found that deal with the consumption of *P&ID* diagrams for the automated generation of simulation models, but not the other way around: the consumption of models for the automated generation of *P&ID* visualizations. While the former requires a robust *(object relational) mapping* logic in order to appropriately bind the information parsed from the *P&ID*, the generation of *P&ID* requires complex graphing algorithms, which none of the related works implements. Not surprisingly though, the compared projects all implement the *XML* format for the representation and templating of the visualizations or models. Hence, similarities could indeed be found, when comparing the related works with a broader lens. It is by the unique intention of this bachelor thesis that it contributes with something new and of significant value to the field: generating intricate *P&ID* visualizations departing from a model and not the way around.

# 3 P&ID Shapes Library

## 3.1 Introduction

The *Legato Graphic Designer* boardlet where the generated *P&ID* visualization in form of a single, static *XML* file is to be uploaded for rendering, implements the *mxGraph API* already, from which the *draw.io* diagramming software tool is built upon. This heavily influenced the decision for the implementation of the *mxGraph API* for the project. Still, other alternative libraries and frameworks where considered and compared. Nonetheless, it was concluded that the intended functionalities could indeed be implemented via the *mxGraph* library and that it was the best option. Moreover, this mature, open-source *API* has many implementations from which to choose from. The *mxGraph JavaScript* library was selected for this project and implemented for the *object-oriented* abstraction of the geometries of the graphical elements. These graphical elements or shapes where abstracted in terms of the parameters specified by the *mxGraph* library for later compatibility. Apart from the adoption of the existing *mxGraph* parameters that define the shape's geometries, most which are directly derived from the *SVG* format, no other *mxGraph* methods or services where used; The geometrical parameter specification was analyzed and structured into a general *class diagram*, from which a static shapes library was assembled to be later manipulated with pure *JavaScript*. The abstraction of the *mxGraph API* allows for the developed solution to be virtually library-independent.

## 3.2 mxGraph API

The *JavaScript* implementation of *mxGraph* is an open-source developer library in form of a single *JavaScript* file that contains all functionalities to provide features aimed at applications with interactive diagrams and graphs. It is licensed under the *Apache 2.0 license* and thus, free for commercial usage. Due to its simple architecture, mxGraph requires only of a web server capable of serving *HTML* pages and a *JavaScript* enabled web browser to function. The library provides all required functionality to draw, interact, and associate a context to visuals and diagrams or graphs on the client-side. This means that round-trips to the server are not required for interaction events, which provides a quick and responsive native feel and allows offline usage. Besides that, the library provides several working examples to guide developers through its own functionalities. According to the official *mxGraph* user manual, key advantages of the technology are:

- No third-party plug-ins are required which removes vendor dependence.
- Technology is open source, so no risk of application becoming unfunctional due to vendor changes.
- Standardized technologies for reach to a maximum number of browsers without requiring additional installation or configuration at the client computer.

### 3.2.1 Geometrical Abstraction of Process Engineering Elements

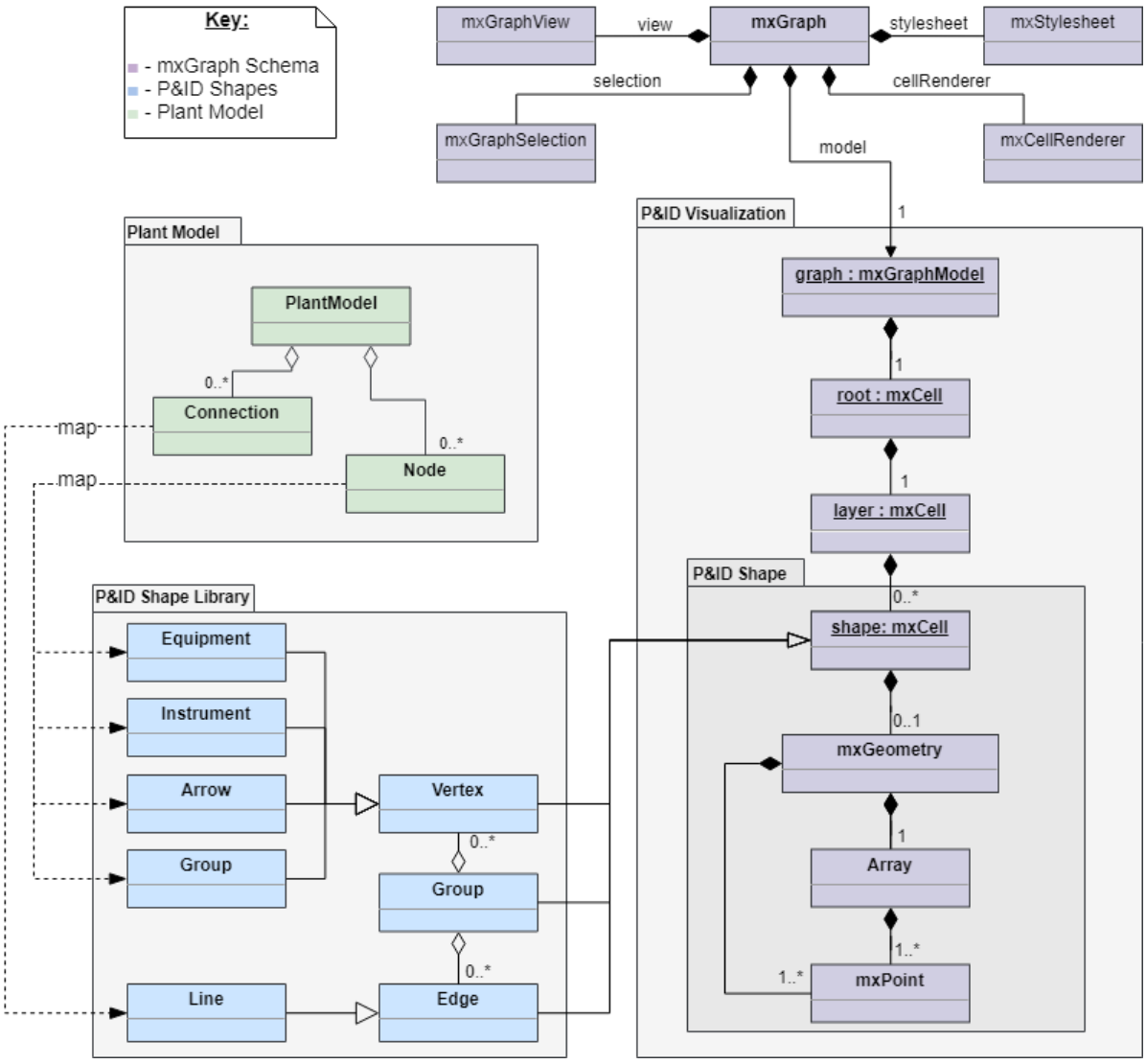



*Figure 13 Interface of plant model instances (green), P&ID shapes (blue) and mxGraph API (purple).*

For this project, it was decided that the Process Engineering Shapes provided by *draw.io* diagraming tool should be used for the *P&ID* shapes library. These shapes already strictly adhere to the *P&ID* industry standards for symbols and notations regulated by the *PIP*. This decision enabled the quick conception of the static *P&ID* shapes library to then continue with the code for the diagraming of the visualizations themselves. Still, analysis of the *mxGraph* API schema at hand was necessary for the *object-oriented* abstraction of the geometrical definition of the shapes based on their inherent geometrical and functional traits. Figure 12 describes the overview of the relationships between the physical plant instances, the *P&ID* shape library and the *mxGraph API* in form of a global *class diagram*.

The *mxGraph API* provides many functionalities and services for the creation of graphs with *JavaScript*, and the previous class diagram is merely an overview. Nevertheless, for the purpose of this project, many of these functionalities were not required. The *Legato Graphic Viewer* boardlet already implements many of these functionalities to render visualizations in the form of single *XML* files. The purpose of this project: the automated generation of *P&ID* visualizations in form of a single *XML* file for the Legato Graphic Viewer to load and render later, required but the general schema of how to define graphs and shapes. From this general schema, the final static definition of all *P&ID* shapes in the library was to be derived, with the following section showing how that was done.

### 3.2.2 Creation of the Object-oriented Shapes Library

**Concept**

In line with the project requirements outlined in section 1.5, the *P&ID* shapes library was to be modular and composable and define the shapes in *SVG* format. This was to be achieved by means of an *object-oriented* design of the library, to enable abstractions, hierarchies and inheritance between classes and the object instances. Inheritance was to provide the library with flexibility and customization; If a user were to modify the geometric aspect of a shape of a certain parent class, the changes were to be propagated throughout children of that same parent class to avoid having to modify each shape individually. On the other hand, despite the fact that the visualization was to be generated in *XML* file format, it was decided to implement the *P&ID* shapes library in form a single, static *JSON* file. The library should be clear and easily human-readable, and the *JSON* format is ideal for this. *JSON* format allows for easier modification of the shapes in the library, either by directly modifying the *JSON* object literal, or by using free online tooling to parse and convert the *JSON* file into a tabular *Microsoft Excel* compatible format like *XLSX*, for even easier modification. Important to note is the following: Equipment, Instruments, Arrows and Groups are represented as Nodes or Vertices. The flows (lines) between them on the other hand, are represented as connections.

**Implementation**

The implementation details as to how exactly the shapes in the library were to be defined, came with the analysis of the `mxGraphModel` object. The `mxGraphModel` is implemented as a common *XML* object, which holds all what is required for the instantiation, definition, rendering and data bindings of visualizations. Departing from the *XML* file of a manually-built *P&ID* diagram of the example *Aida brewery*, the schema was structured into a *class diagram* via advanced analysis tools and detailed inspection of the *mxGraph API* documentation. From there, the general structure of the *XML* objects for the graph and shape instances was derived from the class diagram in order to build the library as a single *JSON* file including all shapes (478 in total). The library itself was progressively enhanced by adding features along the way as they were required, and it finally served as the basis for coding the client-side script for the generation of *P&ID* visualizations in form of a single *XML* files. Figure 13 shows an overview of this abstraction process.

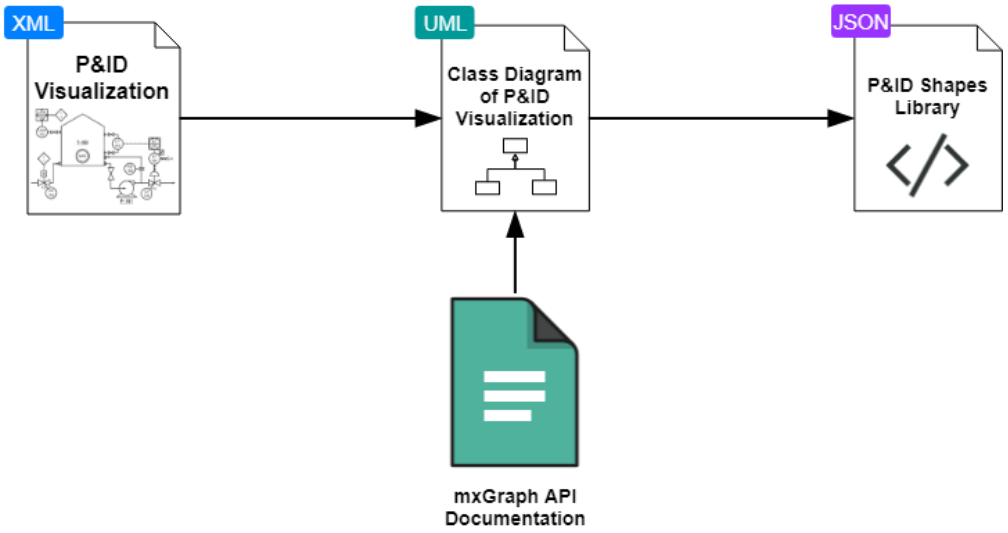



*Figure 14 Abstraction process from an example P&ID visualization into a class diagram of the general schema and concretion into an object-oriented P&ID shapes library.*

Similarly, figure 14 depicts the schema of the *P&ID* shapes library *JSON* file and illustrates its rather flat and not deeply nested structure. The nesting was flattened in favor of greater portability and simplicity, since the library was to be contained in a single, static *JSON* file. Likewise, in favor of a simpler interface for the *object relational mapping (ORM)* between the physical plant instances fetched from the database and their matching *P&ID* shapes retrieved from the *P&ID* shapes library. Due to the size of the library and the individual classes, no global class diagram was created, only analyzed via online tooling to in order to derive the library.

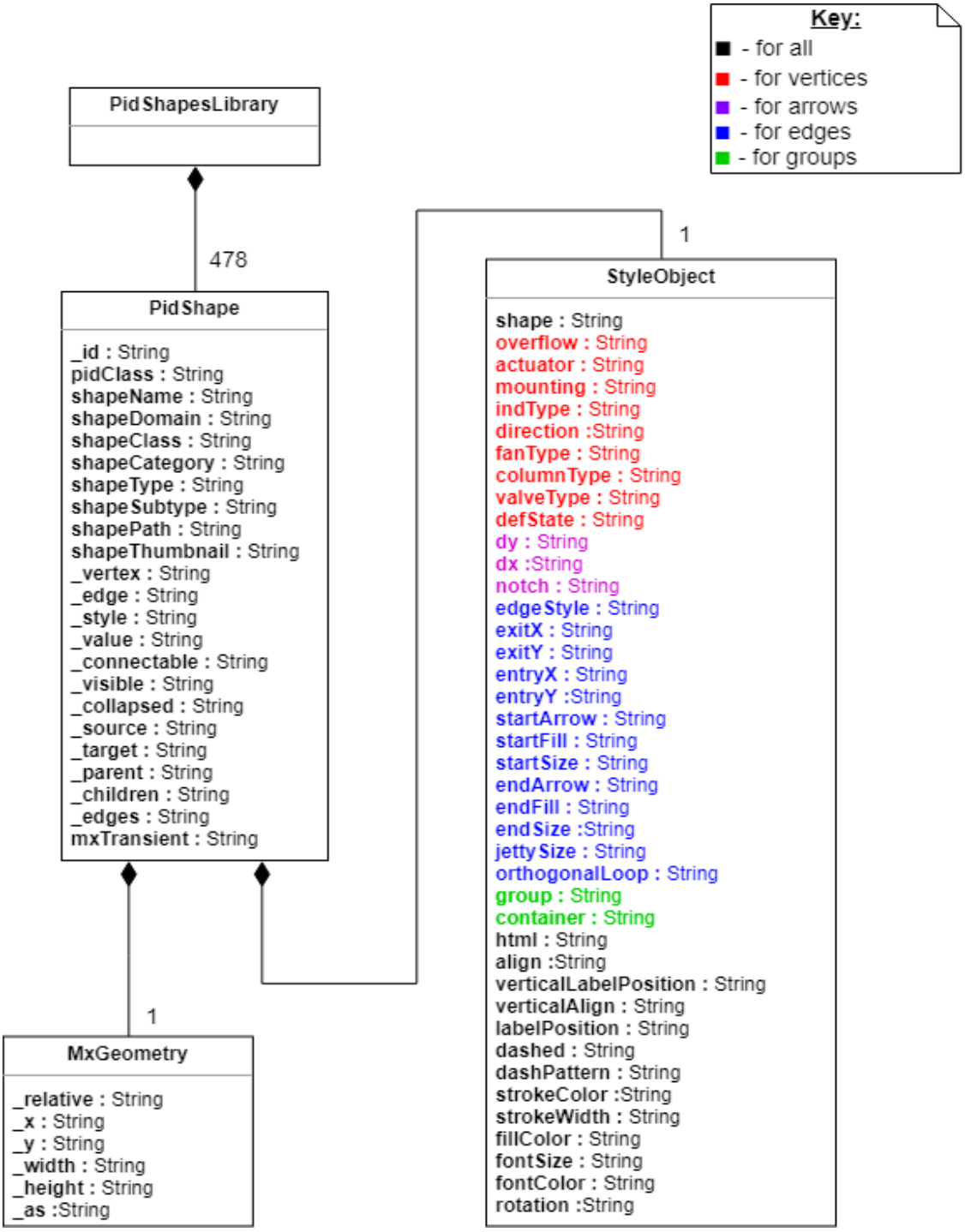


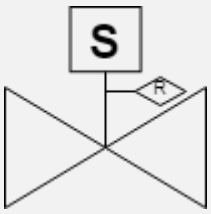

*Figure 15 Schema of the P&ID shapes library JSON file.*

The library consists of a single array of 478 objects, one for each *P&ID* shape available from the default *draw.io* process engineering shapes library. Each object thus entirely defines the corresponding shape by means of those properties. These include an id, metadata, geometry, styling, size and positioning, by means of which all shapes in the library are distinctly defined. Important to note is that mxGraph automatically provides default values for non-passed attributes and for empty strings passed to attributes of the final XML object. This means, that certain properties remain empty string in the library, as there is

no need to override the default value. Nonetheless, some properties do indeed override a default value in order to be instantiated in the final *XML* object of the shape to be rendered, to allow for data bindings to it for example.

As an example, the following *JSON* code snippet defines the shape for a process engineering valve of type gate with solenoid actuator and manual reset. This shape data is to be mapped to a corresponding physical plant instance, if any, to build a single object with joint geometric and plant instance data, from which the final *XML* object for the shape is to be generated to be included in the *XML* file of the corresponding *P&ID* visualization. All 478 shapes are analogously defined in the library based on both geometrical and functional aspects.

**Example JSON Object: Solenoid Gate Valve with Manual Reset (shape below)**





```
{
    "_id": "",
    "pidClass": "equipment",
    "shapeName": "gate_valve_(solenoid)",
    "shapeDomain": "mxgraph",
    "shapeClass": "pid2valves",
    "shapeCategory": "valve",
    "shapeType": "gate_valve",
    "shapeSubtype": "(solenoid)",
    "shapePath": "mxgraph.pid2valves.valve",
    "shapeThumbnail": "images\\gate_valve_(solenoid).svg",
    "_vertex": "1",
    "_style": "verticalLabelPosition=bottom;align=center;html=1;verticalAlign=top
;dashed=0;shape=mxgraph.pid2valves.valve;valveType=gate;actuator=solenoid;
strokeColor=#000000;fontSize=18;fontColor=#000000;",
    "styleObject": {
        "shape": "shape=mxgraph.pid2valves.valve",
        "actuator": "actuator=solenoid",
        "valveType": "valveType=gate",
        "html": "html=1",
        "align": "align=center",
        "verticalLabelPosition": "verticalLabelPosition=bottom",
        "verticalAlign": "verticalAlign=top",
        "dashed": "dashed=0",
        "strokeColor": "strokeColor=#000000",
        "fontSize": "fontSize=18",
        "fontColor": "fontColor=#000000"
    },
    "_parent": "1",
    "mxGeometry": {
        "_relative": "0",
        "_width": "100",
        "_height": "100",
        "_as": "geometry"
    }
}
```

In this way, each object in the library fully encapsulates all data needed for the correct instantiation of *XML* objects later by the visualization generating script, namely an id, metadata, geometry, styling, size and positioning. The object relational mapping of physical plant instances to shapes is done with a single attribute: `shapeName`. This means, that the only required modification to the database of the plant was a single field with the corresponding `shapeName` value for each physical instance (although more modifications where actually made to the database). The `_style` property is defined by the mxGraph API itself and required in the final XML object of the shape. The `_style` property takes in a string of semi-colon separated key-value pairs with all style such as shape type, fill color, font size and alignment for the corresponding shape. To facilitate the targeting and modification of individual style properties, the string was split into the `styleObject`, an object containing all key-value pairs.

### 3.2.3 Real-time Data Binding to Process Variables

As previously mentioned, the generated P&ID visualization was to be in form of a single, static XML file. Nevertheless, the shapes implement *data-bindings* to the database for the real-time dynamic rendering of process variables and styles. The *Legato Graphic Designer* is responsible for loading and rendering the XML file and already implements an API for data binding to dynamic database values and a service for the automatic, real-time re-rendering on each change event. For this reason, the P&ID shapes library does not implement the *data-bindings*, but only the static geometric definition of the shapes. Moreover, the *data-bindings* were not tested due to a lack of a real connection to the brewery, and thus a lack of real time data. Nevertheless, the *data-bindings* were prototypically proven to work in a testing environment. Figure 15 shows two of the possibilities: Boolean values to toggle between simple colors and integer values to be displayed as labels (also the case for string values) (more in section 4.3.5).

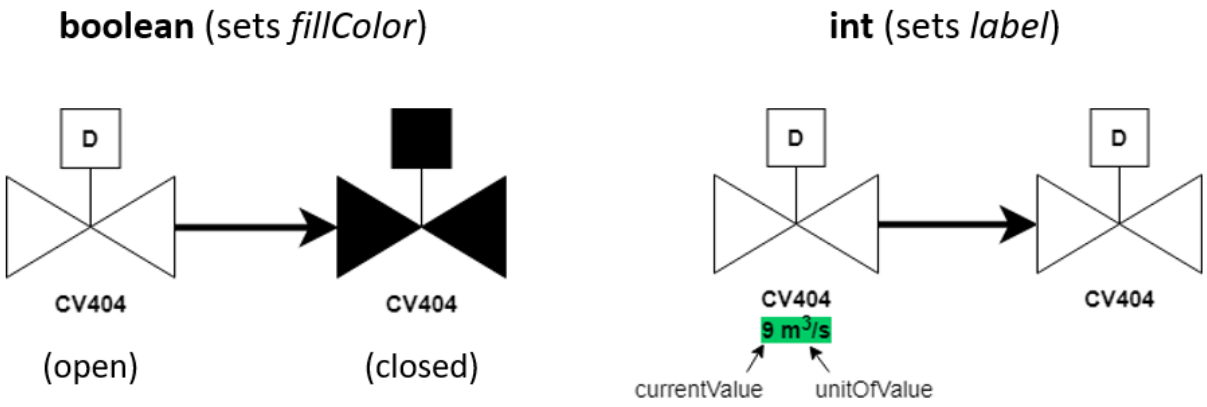



*Figure 16 Example data-binding to fillColor for Boolean values and to label for Integer (or String) values.*

# 4 Legato Dashboard – P&ID Viewer

## 4.1 Software Architecture

### 4.1.1 Requirements

Not surprisingly, the development of the dashboard for the *Legato Web Application* embodied most of the project requirements. The dashboard *front-end* consists of:

- **R2:** User Friendly Graphical User Interface (GUI) Boardlet for Creation of P&ID Visualizations

and the dashboard back-end of:

- **R3:** Client-Side Script for the Automated P&ID Visualization Generation as an XML File
- **R4:** Mapping of Physical Plant Instances to Corresponding Visualization Component
- **R5:** Automatic Type Detection and Simplification of Connections
- **R6:** Declarative specification of Graphing Constraints in Form of Tags
- **R7:** P&ID Graphing Algorithm
- **R8:** Dynamic Real-Time Display of Process Variables in the P&ID Visualization

Likewise, the project required:

- **R9:** Prototypal Implementation in the Infrastructure of a MES (Legato Sapient®) and Documentation

The above requirements served to align the software solution to the goals of the project and ensure that all of them where satisfactory achieved by means of the developed solution.

### 4.1.2 Alignment to the System Architecture

The design of the software architecture was greatly influenced by the constraints of the *Legato Sapient* system architecture. Due to the relative freedom in the conception of the software solution for this project, aligning the software architecture to the system's architecture and its context was required in order to ensure the quality of the solution to be developed. It is by means of the "big picture" of the system, including hardware and software, that the design of the software architecture was conceived. Figure 16 portrays this overview of the system architecture, from the client-side, through the server, and all the way to the plant level.

As can be seen in this figure 16, both presentation and business logic run entirely on the client-side (in the browser). The *SysML* model of the plant was previously translated into r*elational database tables*. This tables are filled with static and real-time data coming from the plant through the *Sapient Engine*. The *front-end* of the system is represented by the *GUI*, in this case the dashboard, outlined in blue. From there, user is prompted to select a shapes library file (1), a root node from the *Node Tree* boardlet (2) and to

click the “Generate P&ID” button to start the generation by calling the client-side script (3), outlined in red. From there, the script sequentially queries the database, filters the data (4) and generates the *XML* of the *P&ID* visualization (5). Finally, after the *XML* generation is done and rendered as text in the *P&ID Creator* boardlet, the download or upload of the *XML* to the local or server file system (6), outlined in green, is handled upon user request. As soon as the *XML* file is saved in the corresponding path in the local file system, it will be automatically rendered by the *P&ID Viewer* boardlet. The user is then free to repeat this process to create a new or update the existing *P&ID* to be re-rendered.

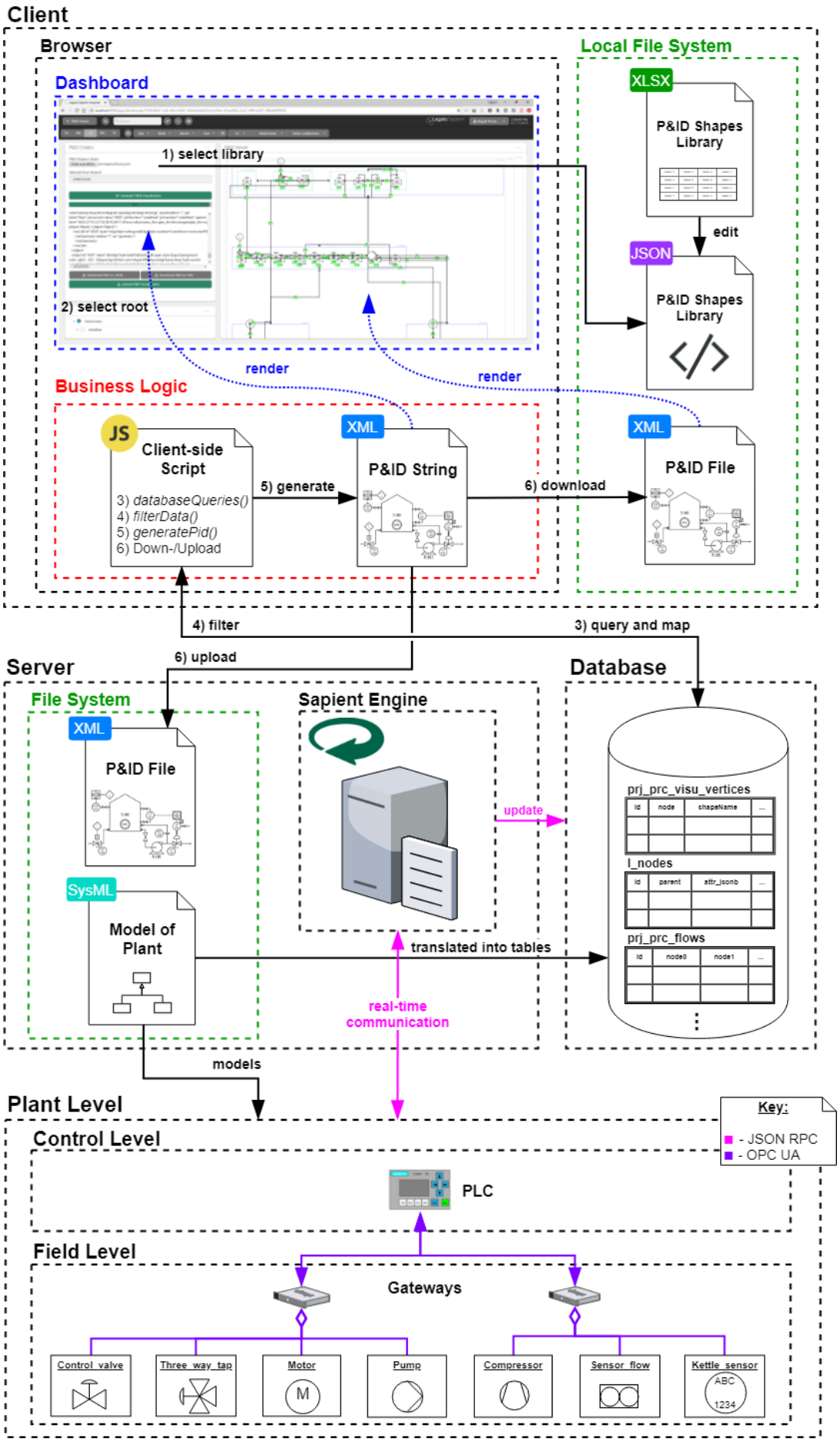



*Figure 17 Software Architecture Diagram.*

### 4.1.3 System Interactions

Figure 17 shows a detailed activity diagram of the system interactions and how they are implemented in the front- and back-end code. The *GUI* and the script or business logic run their corresponding activities entirely on the client, as does the local file system. On the other hand, the database queries for the physical plant instances require round trips to fetch the required data. Function blocks in blue corresponds to the presentational aspects of the graphical user interface and will be further detailed in section 4.2. Red function blocks on the other hand, correspond to the business logic to run on the client background for the *P&ID* generation and will be further detailed in section 4.3.

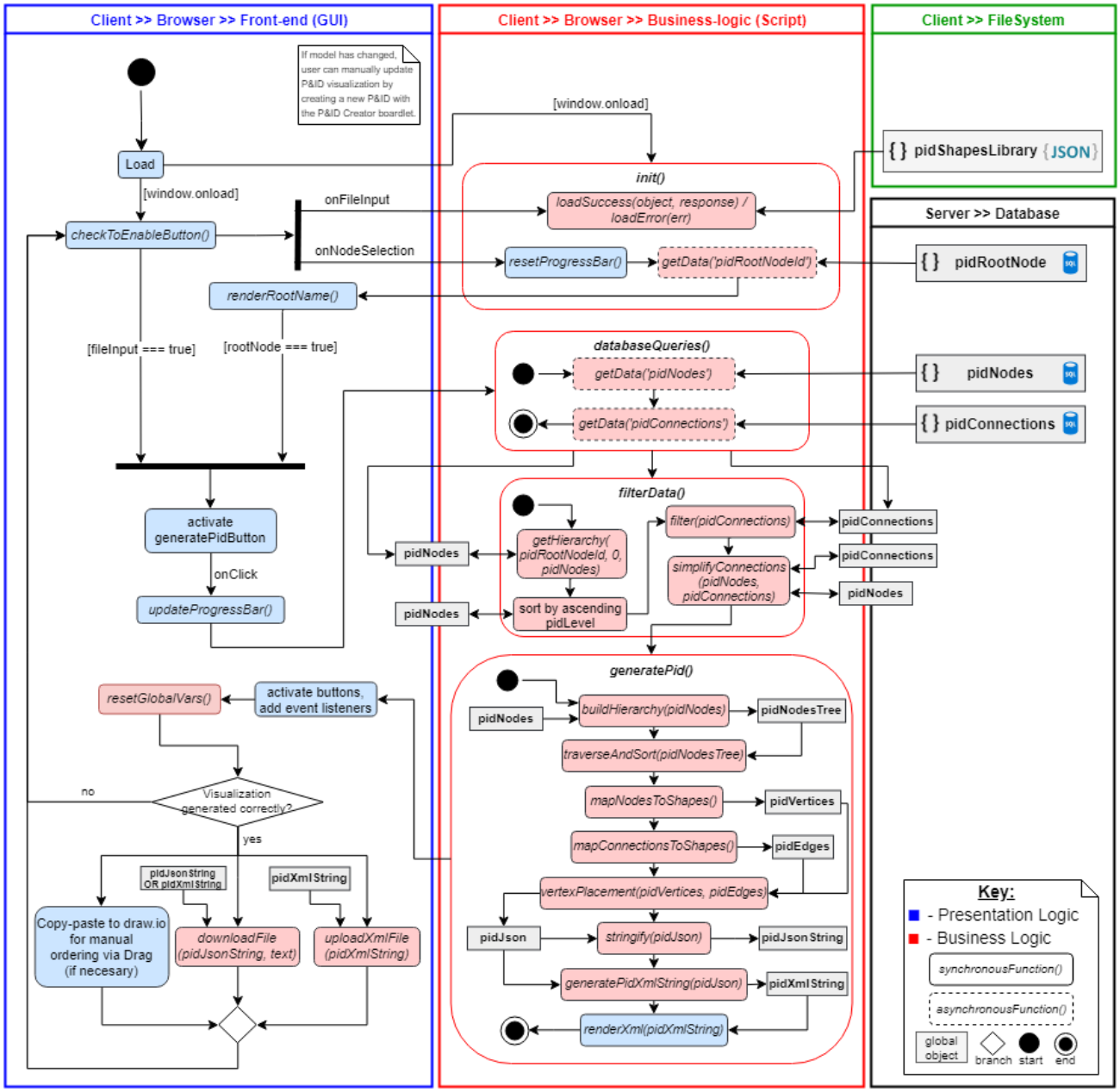



*Figure 18 UML 2.0 Software Activity Diagram.*

## 4.2 Presentation Logic

### 4.2.1 P&ID Creator Boardlet

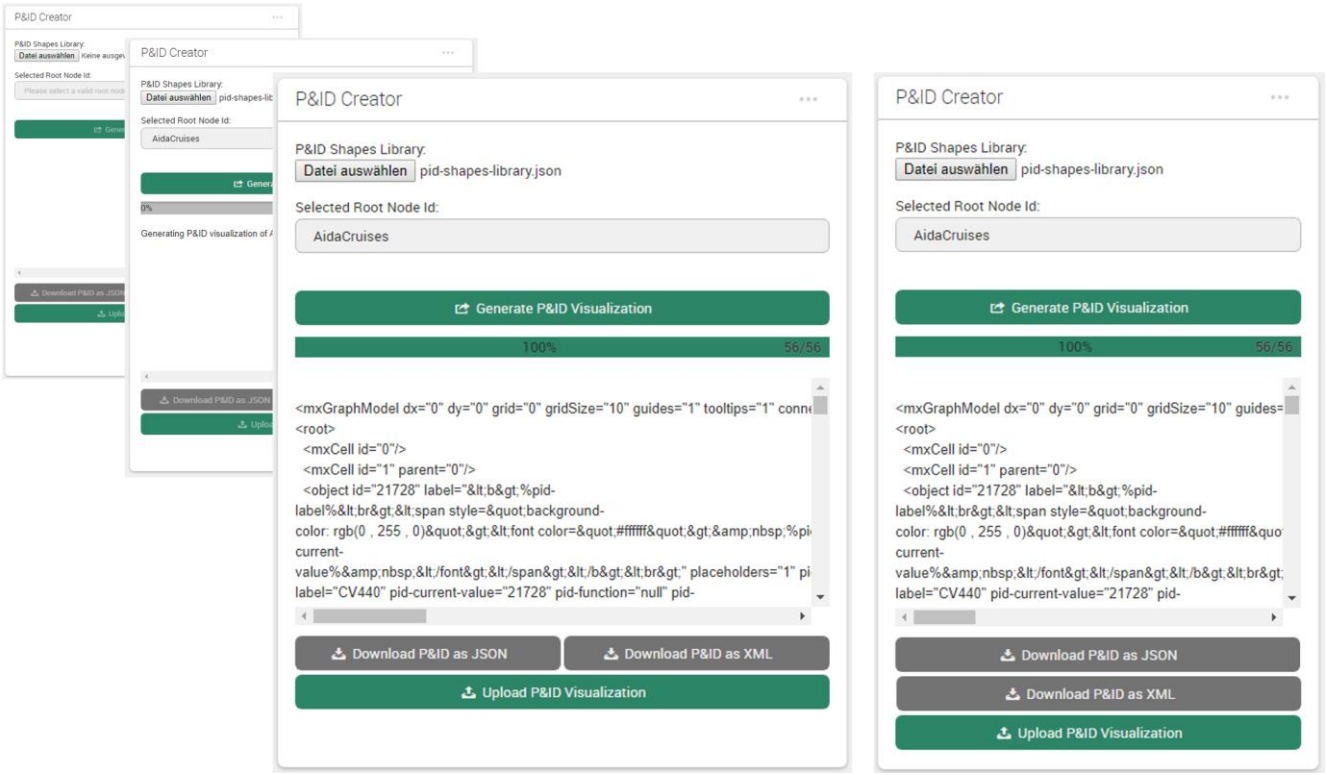



*Figure 19 Responsive design of P&ID boardlet through the P&ID generation process (both for desktop and mobile).*

The developed *P&ID Creator Boardlet* and its animations throughout the *P&ID* generation and carried out by the blue function blocks of figure 17 are shown in figure 18. For smaller screens like mobile devices or when resizing browser window, the boardlet behaves responsively by adapting the download buttons to one on top of the other. The boardlet follows *component-driven UI design* principles, and is composed of even smaller, modular *Ember Components* which encapsulate both presentation and business logic in small bundles. These has many benefits for developers like code reusability, accelerated development times, *user expierience (UX)* consistency.

Upon page load (back), the boardlet renders with a file input button active, a node selector input field, which updates upon node selection of the *Node Tree boardlet* and all other buttons deactivated (`checkToEnableButton`). This ensures an intuitive user expierience (*UX*) and makes it easy for everyone to use the boardlet. After an appropriate *JSON* file of the *P&ID* shapes library and a root node is selected, and thus its name rendered in the text field (`renderRootName`), the "Generate P&ID Visualization" button activates, prompting the user to click it. On click (middle) event, the script for the *P&ID* generation is called and a progress bar is shown (`updateProgressBar`), with the total number of plant instances to be modelled. The progress bar updates with a slight animation delay for each instantiated *XML* object. Finally, when the *P&ID* generation concludes, and all plant instances were instantiated as *XML* objects for shapes (front), the generated *XML* is rendered for user feedback and

control and the download and upload buttons are activated (`renderXml`). From this point, the user can either re-generate the same or a new *P&ID* without the need of page reload by selecting a different root node and/or shapes library.

### 4.2.2 P&ID Viewer Dashboard

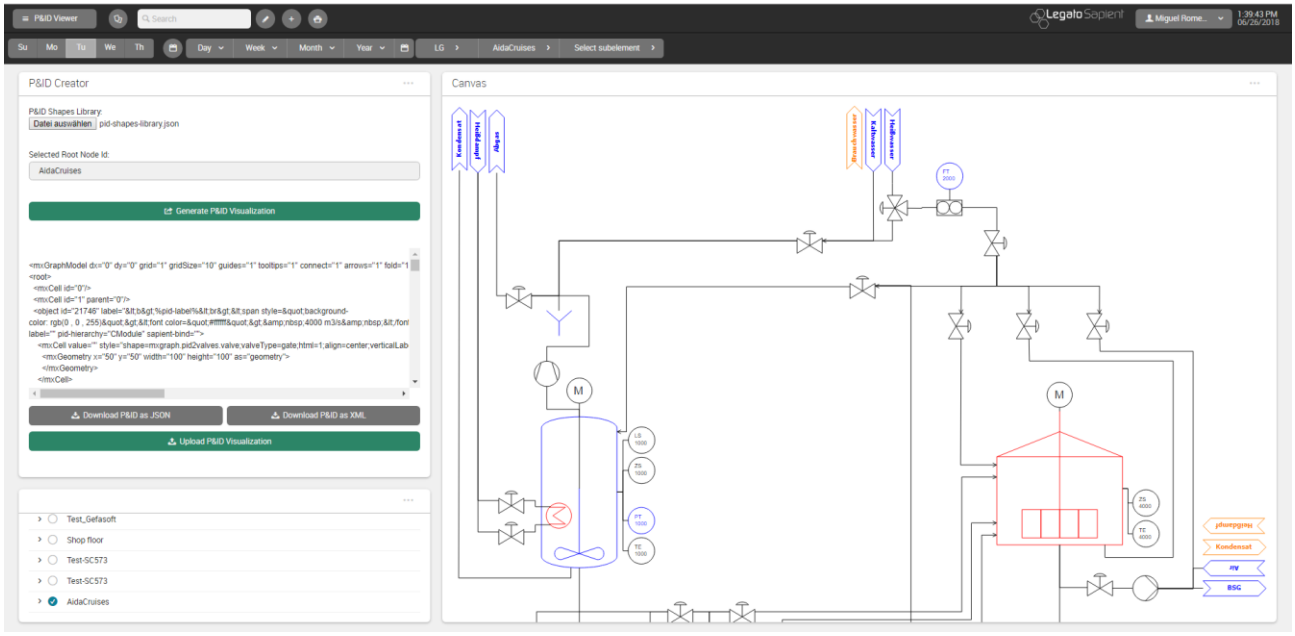



*Figure 20 P&ID Viewer Dashboard for the Aida Brewery (rendering a manually generated P&ID).*

The *Legato Web Application* is made up of composable dashboards. These are built by pre-defined boardlet components, which can be drag-and-dropped in place and communicate with each other. *Legato* services for the communication between boardlets exist already and where used for the root node selection; Once a root node was selected on the *Node Tree boardlet*, it communicated the id of the selected root node to the *P&ID Creator Boardlet*, which next fetched the node data from the database via the `id` (*SQL*: `SELECT * WHERE id=pidRootNodeId`), for example for the `short_name` of the node to be displayed in the field. After uploading or manually downloading and saving the *P&ID* generation in the corresponding path on the server, the *P&ID Viewer boardlet* automatically reads and re-renders the new or overwritten file. The *P&ID Viewer* corresponds to the *Legato Graphic Designer boardlet*, while the selection of the root node is implemented with the *Legato Node Tree boardlet*.

## 4.3 Business Logic

The business logic functionality can be broken down into a series of sequential activities (equally numbered in figure 16):

1. User selection of the desired version of the *P&ID* shapes library *JSON* file (from either the local or server file system) and of root node for the visualization (from node tree boardlet)

2. Script is called upon click event of the "Generate P&ID" button. This initiates the client-side sequence of function calls for the generation of the *P&ID* visualization while the selected *JSON* file is asynchronously read and loaded into a *JS* object for its later manipulation.

3. After the file is successfully loaded, the corresponding database queries of the physical plant instances (node and connection data) are carried out asynchronously. These responses are next loaded into *JS* objects, filtered and mapped with an *ORM*-function via the `shapeName` property to their corresponding shape in the library object. The function builds 2 new objects (`pidNodes` and `pidConnections`), which hold all nodes (Equipment, Instruments, Arrows, Groups) and connection (lines) by merging the physical plant instance data (from the database) and the data of its corresponding shape (from the shapes library).

4. The `pidNodes` and `pidConnections` objects are passed to the graphing algorithm, which consists of three sub steps:

    i. **`buildHierarchy`**: Filter out non-descendants (if any) and builds hierarchical/nested *JS* object of the instance hierarchy via the parent attribute.

    ii. **`traverseAndSort`**: traverses the instance hierarchy with a *post-order depth-first search (DFS*) and returns the traversal path.

    iii. **`vertexPlacement`**: determines positioning of vertices and sets the `x` and `y` properties of each.

5. Builds the *XML P&ID* visualization in form of a string using templating and recursion. This string contains all information for the proper rendering and data binding of the *P&ID*.

6. Download buttons (for *JSON* and *XML* file formats) and an upload button activate when *P&ID* generation is done. The *Legato Graphic Designer* boardlet automatically renders the *XML* file in the path and with the name configured in its settings.

For the sake of clarity, each activity will be examined separately and described in further detail isolated from the other activities, as they are actually implemented (modularly). The interactions each activity has with others can be interpreted from figures 16 and 17.

### 4.3.1 File and Root Node Selection

The file input is handled by the `fileUploadComponent` which encapsulates both presentation and business logic and exposes a simple *API*. The actual reading of the *JSON* file is handled by the `loadSuccess` and `loadError` functions, which are called on file input event, and parse the *JSON* file to a global *JS* object (`pidShapesLibrary`).

### 4.3.2 Querying and Mapping Data

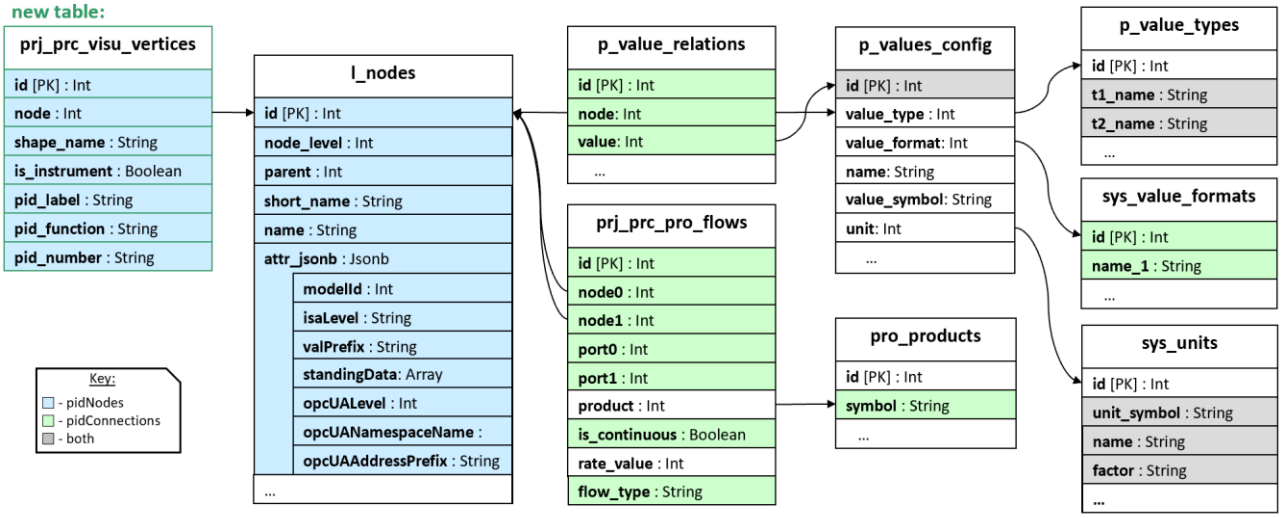



*Figure 21 Data Map of retrieved pidNodes and pidConnections information from database.*

**Querying Data from Database**

The data map for the required queries to the *PostgreSQL* relational database are shown in figure 20. According to this scheme, `JOIN`s over multiple tables retrieve all required data from the database. As advised by the *Legato* internal documentation, a generic `getData` function was developed which dynamically creates the query filters and asynchronously fetches the data. The `getData` function encapsulates the `getRecords` function provided already by *Legato*, to deliver a simple usage and can be re-used for all database queries through the specification of distinct query parameters with a simple if-statement (code lines 7-65). The queries are built using the following parameters which are then passed to the `getRecords` function (code lines 68-73):

- **`resource`** – name of the database table
- **`alias`** – an alias or short-name for tables to be queried
- **`fields`** – the fields to be queried
- **`relate`** – for joining several tables
- **`filter`** – to filter out certain values
- **`nameMappings`** – to map field names to keys for the final return object

**Mapping Data to Objects**

The `nameMappings` *JS* object handles the *object relational mapping* by cloning the response object to itself (code lines 79-83), thus translating the field names to more adequate variable names as specified in the code (code lines 38-64). Once the `getRecords` function asynchronously places the request in form of a promise, the code waits (unblocking) for the response object to be returned, to next continue with the

`.then` statements. The `databaseQueries` function concludes once all queries succeed to next call the `filterData` function (next section).

```
getData: function(data) {

    let resource, alias, fields, relate, filter;
    let rootId = this.get('pidRootNodeId');
    let nameMappings = [];

    // Build query parameters dynamically (shown for pidNodes)
    if (data === "pidRootNode") {...}
    if (data === "pidConnections") {...}
    if (data === "pidNodes") {
        resource = "l_nodes";
        alias = { "n": "l_nodes", "v": "prj_prc_visu_vertices", "r": "p_value_relations",
            "c": "p_values_config", "f": "sys_value_formats", "t": "p_value_types",
            "u": "sys_units"
        };
        fields = {
            "n": "id, node_level, parent, short_name, name, attr_jsonb",
            "v": "id, node, is_instrument, shape_name, pid_label, pid_function, pid_number",
            "r": "id, node, value",
            "c": "id as c_id, value_type, value_format, unit, value_symbol, name as c_name",
            "f": "id, name as f_name",
            "t": "id, name as t_name",
            "u": "id, name as u_name, unit_symbol"
        };
        relate = [
           { "src": "n", "dst": "v", "how": "left", "on": { "src": "id", "dst": "node" } },
           { "src": "n", "dst": "r", "how": "left", "on": { "src": "id", "dst": "node" } },
           { "src": "r", "dst": "c", "how": "left", "on": { "src": "value", "dst": "id" } },
           { "src": "c","dst": "f","how":"left","on": {"src": "value_format","dst": "id"} },
           { "src": "c","dst": "t","how": "left","on": {"src": "value_type","dst": "id" } },
           { "src": "c", "dst": "u", "how": "left", "on": { "src": "unit", "dst": "id" } }
        ];
        filter = {
            "field": "n.id",
            "op": "ge",
            "val": this.get("pidRootNodeId")
        };
        nameMappings = [
           // From l_nodes:
           { id: 'id' },                          // {Int} Primary Key
           { nodeLevel: 'node_level' },           // {Int} Hierarchy level of node
           { parentId: 'parent' },                // {Int} Id of parent in l_nodes
           { shortName: 'short_name' },           // {String} Short name
           { name: 'name' },                      // {String} Long name
           { details: 'attr_jsonb' },             // {Int} Other details (jsonB format)
           // From prj_prc_visu_vertices:
           { vId: 'id' },
           // {Int} Primary Key
           { isInstrument: 'is_instrument' },     // {Bool}  User defined boolean
           { shapeName: 'shape_name' },           // {String} Shape Name (required!)
           { pidLabel: 'pid_label'                // {String} Optional label for shapes
           { pidFunction: 'pid_function' },       // {String} Instrument function
           { pidNumber: 'pid_number' },           // {String} Instrument number
           // From p_values_config:
           { cValueId: 'c_id' },                  // {Int} Primary Key of value
           // From sys_value_formats
           { fDataType: 'f_name' },               // {String} Data type of value
           // From p_value_types:
           { tValueType: 't_name' },              // {String} (limit,energy,air...)
           // From sys_units
           { uUnitSymbol: 'unit_symbol' },        // {String} (Nm,J,kPa,°C, ...)
           { uName: 'u_name' },                   // {String} (Newton,Joule, ...)
```

```
63.             { uFactor: 'factor' }                    // {String} (1000, /1000, ...)
64.         ];
65.     }
66.
67.     let jsObject = [];
68.     this.get('server').getRecords(resource, {
69.         alias: alias,
70.         fields: fields,
71.         relate: relate,
72.         filter: filter
73.     }, undefined).then((result) => {
74.         if (result.content.length > 0) {
75.             let jsonClassArray = result.content;
76.             // Build jsObject with only fields in corresponding model
77.             jsonClassArray.forEach((row) => {
78.                 let object = {};
79.                 nameMappings.forEach((entry) => {
80.                     let attribute = Object.keys(entry);
81.                     let field = Object.values(entry);
82.                     object[attribute] = row[field];
83.                 }) jsObject.push(object);
84.             });
85.         }
86.     }).then(() => {
87.         // Set parsed jsObject to corresonding global varible (data)
88.         this.set(data, jsObject);
89.     }).then(() => {
90.         // Render root name when pidRootNode is queried data
91.         if ('pidRootNode' === data) this.renderRootName(this.get('pidRootNode'));
92.         this.filterData(data);
93.     });
94. }
```

### 4.3.3 Filtering Queried Data

Due to the complexity of the *SQL* query required to reclusively extract the complete hierarchy of a selected root node via its parent attribute, a workaround had to be developed. Instead of reclusively traversing through the selected root node's relatives (all ascendants and descendants) during the query, a simpler filtered query is made after it to filter the results recursively with *JavaScript* with the `filterData` function (refer to figure 17). An overview of the data filtering process from root node selection to is illustrated by figure 21.

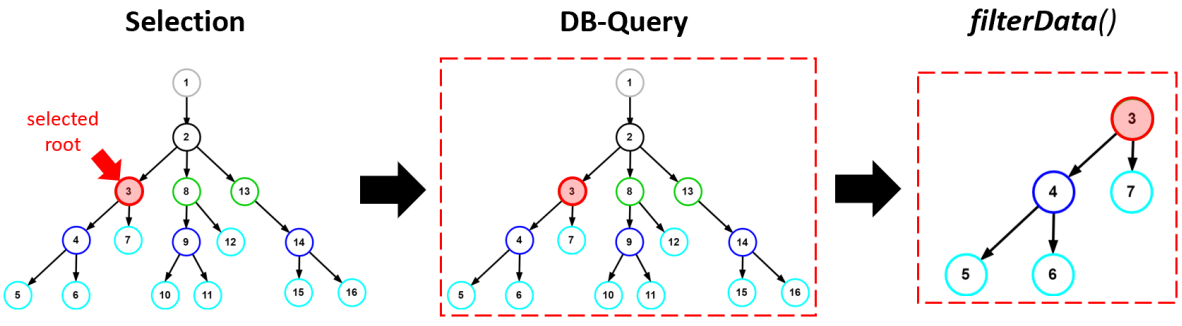



*Figure 22 Overview of the filtering process.*

The filtering of `pidNodes` and `pidConnections` is managed separately due to the distinct structure of the data object for `pidNodes` and `pidConnections` (refer to figure 20). On one hand, the `getHierarchy` function is passed the `pidNodes` object along with the selected root node to filter out all non-descendants of the `pidRootNode`. Additionally, during the level traversal, the function sets the `pidLevel` attribute to the objects metadata starting at `pidLevel=0` for the `pidRootNode`. Afterwards, the returned `pidNodes` array is sorted in ascending `pidLevel` order by means of an anonymous *ES6 arrow function*.

On the other hand, the `pidConnections` must be filtered out in case that either the `source` or `target` is not found in the now filtered `pidNodes` object. The `filterConnection` function takes in the `pidNodes` and `pidConnections` objects and returns the filtered `pidConnections` object, which the `simplifyConnections` function next takes in.

The *SysML* model of the plant, and thus the database represents connections from port-to-port, instead of globally from source to parent, therefore intermediate ports must be collapsed. The `simplifyConnections` function achieves this for edges from and to groups (`vertex.pidClass === 'group'`) by replacing both the `preEdge` and `postEdge` of that connection with a single, direct connection. Additionally, the function sets the `parentId` property of each simplified `edge` to the `id` of its `source` (required by *mxGraph* for the proper rendering of an edge). This means that the `simplifiedId` inherits all properties from the `startEdge` as well, which should nevertheless equal those of the `endEdge`. The functionality of the `simplifyConnections` function is analogous to clearing edge waypoints and is illustrated in figure 22.

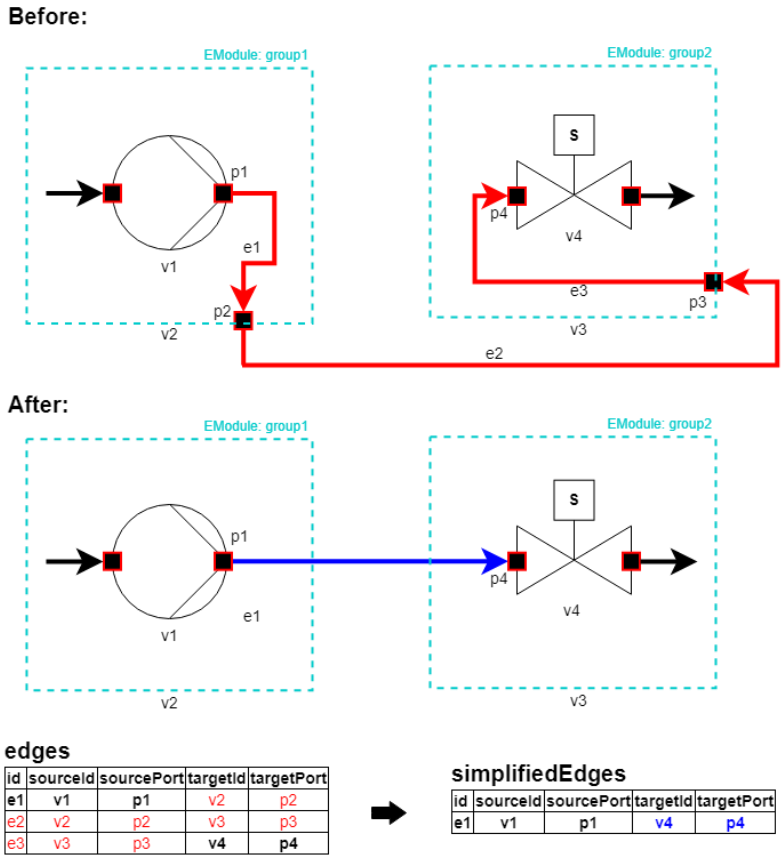



*Figure 23 Working principle of the simplifyConnections function.*

Moreover, for the example *Aida brewery* plant, only process flow where modelled. This means that contrary to nodes (equipment, instruments, arrows and groups), which are given a `shapeName` attribute in the database, the corresponding line shape for each connection must be determined by the script. Five process engineering classes exist, with the following `pidClass` values and an example shape:


*Figure 24 P&ID Shape Classes (pidClass).*

Similarly, four distinct *P&ID* line shapes exist with the following `shapeName` values and visual appearances:

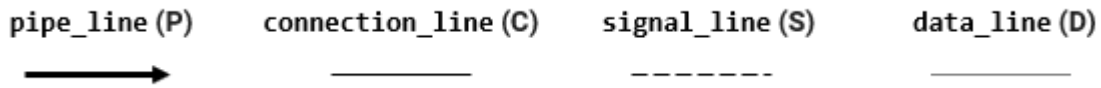



*Figure 25 P&ID Line Shapes (shapeName).*

For this project, only `pipe_line`–connections were required since only process flows were originally modelled. Nevertheless, a concept for the automatic identification of flow type was developed, to correspondingly set the `shapeName` attribute of each connection. The logic considers the `pidClass` of `source` and `target` nodes for each connection (after simplification) to determine the corresponding line type by means of an *adjacency matrix* of all possible relationships (table 3). Non-existent cases are greyed out but could be over-simplifications.

| Source\Target | Equipment | Instrument | Group | Arrow |
|---|---|---|---|---|
| **Equipment** | P | P,S | P | P |
| **Instrument** | P | D | D | P |
| **Group** | P | D | P | P |
| **Arrow** | P | P | P | P |

*Table 3 The adjacency matrix of all possible flows between pidClasses.*

The logic is indeed capable of identifying the `shapeName` based on the `pidClass` criteria but is limited to the accuracy of the *adjacency matrix*, which over-simplifies all possible relationships and misses out

on many special cases. Still, this conceptual matrix, which could eventually be further refined, was implemented as seen in table 3.

### 4.3.4 P&ID Generation

After the data is queried, mapped to *JS* objects and filtered, the `generatePid` function is called at the end of the `simplifyConnections` function. The `generatePid` function handles the calling of several functions for the creation of the *P&ID*. As a reminder, nodes or vertices represent equipment, instruments, arrows and groups, while connections represent lines. The functions, called in sequence within the `generatePid` function as shown by figure 17, are the following:

1. **`buildHierarchy`** – takes in a flat array (`pidNodes`) and builds a nested object via the `parent` attribute (`pidNodesTree`). Works even for more than one root node selections

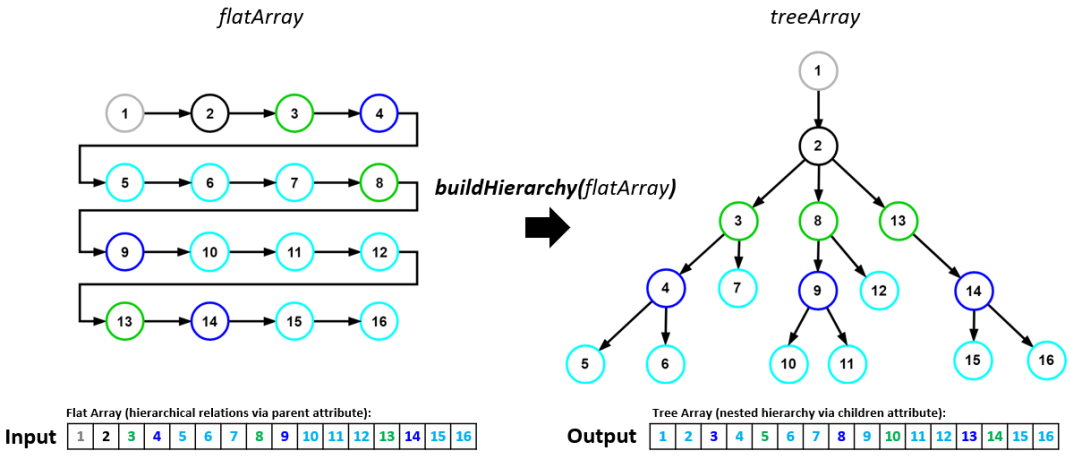



*Figure 26 Input and output of buildHierarchy function.*

2. **`traverseAndSort`** – takes in the `pidNodesTree` and traverses it (*post-order depth-first search*) via the `children` property (array) and overwrites it with the sorted path of traversed vertices (array of objects)

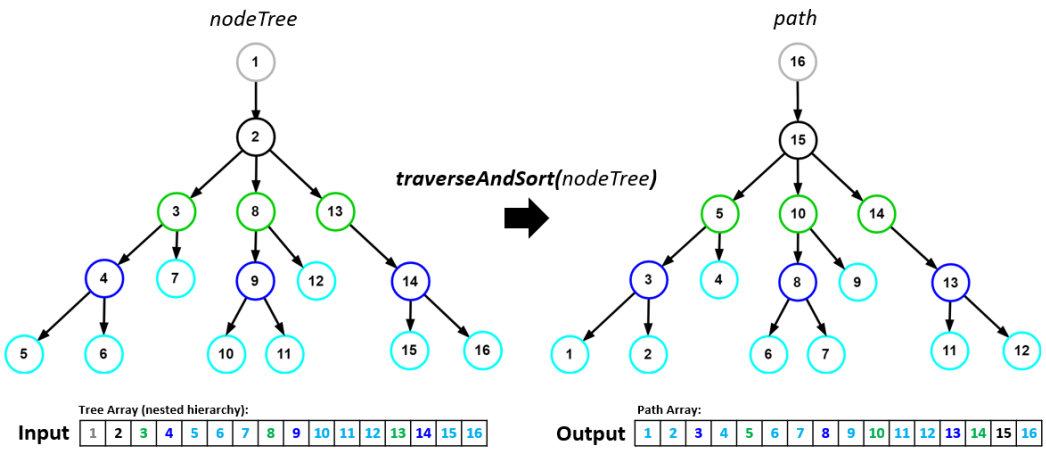



*Figure 27 Input and output of traverseAndSort function.*

3. **mapNodesToShapes** – maps each node from `pidNodes` to its corresponding vertex shape (equipment, instruments, groups and arrows) and creates a `pidVertex` instance with joint `pidNode` and `pidShape` properties (all of them). For `pidNodes` without a `shapeName` (`shapeName === ""`) like groups, it determines and sets it according to the `pidLevel`
4. **mapConnectionsToShapes** – maps each connection from `pidConnections` to its corresponding edge shape (lines) and creates a `pidEdge` instance with joint `pidConnection` and `pidShape` properties (all of them). Additionally, implements the conceptual logic to determine and set `shapeName` property for lines (described in previous section)
5. **vertexPlacement** – takes in `pidVertices` and `pidEdges` and loops through each `pidVertex` to set the `_x` and `_y` properties of the `mxGeometry` property of each (described in detail in chapter 5)
6. **stringify** – converts the pidJson object to a string
7. **generatePidXmlString** – generates the XML string of the visualization by looping through the `pidJson` instances and creating each XML object individually for each (described in detail in section 4.3.5)
8. **renderXml** – takes in the `pidXmlstring` and renders it in the text-box of the *P&ID Viewer* boardlet (calls `formatXml` and `escapeToHtmlValid` functions internally)

Analogous to a typical `main()`-function, the `generatePid` function serves to call functions declared elsewhere and additionally for presentational logic (not shown below). The function calls serve for easier unit-testing of the individual functions during development. Excerpts of this function are shown below:

```
generatePid: function() {
    this.set('pidNodesTree', this.buildHierarchy(this.get('pidNodes')));
    this.set('pidNodesInOrder', this.traverseAndSort(this.get('pidNodesTree')));
    this.set('pidVertices', this.mapNodesToShapes());
    this.set('pidEdges', this.mapConnectionsToShapes());
    this.set('pidJson', this.vertexPlacement(this.get('pidVertices'), this.get('pidEdges')));
    this.set('pidJsonString', JSON.stringify(this.get('pidJson')));
    this.set('pidXmlString', this.generatePidXmlString(this.get('pidJson')));
    this.set('loading', false);
    this.renderXml(this.get('pidXmlString'));
    this.resetGlobalVariables();
    this.checkToEnableButton();
}
```

### 4.3.5 XML String Generation

**Structure of the XML to be Generated**

The following code shows the general structure of the `pidXmlString` to be generated. The values in blue are all default values. Still, they can be directly changed from the template's code if the user desires to change the background color or to show a grid for example. In addition to the `mxGraphModel`, `root`,

and first two `mxCell` objects, which are boilerplate for the graph, two sample shapes (with empty property values) are shown. The number of *XML* objects after the generation corresponds to the number of mapped instances to shapes. These properties are to be filled-out dynamically with the corresponding variable for each shape using *ES6 template literals*.

```
<mxGraphModel dx="0" dy="0" grid="0" gridSize="10" guides="1" tooltips="1"
connect="1" arrows="1" fold="1" page="1" pageScale="1" pageWidth="1654"
pageHeight="1169" background="#ffffff" math="0" shadow="0">
<root>
  <mxCell id="0"/>
  <mxCell id="1" parent="0"/>

  <object id="" label="" placeholders="" pid-label="" pid-current-value=""
pid-function="" pid-number="" sapient-bind="">
    <mxCell style="" vertex="" connectable="" parent="">
      <mxGeometry x="" y="" width="" height="" as="">
      </mxGeometry>
    </mxCell>
  </object>

  <object id="" label="" placeholders="" pid-label="" pid-current-value=""
pid-function="" pid-number="" sapient-bind="">
    <mxCell style="" vertex="" connectable="" parent="">
      <mxGeometry x="" y="" width="" height="" as="">
      </mxGeometry>
    </mxCell>
  </object>

• • •

  </mxCell>
</object>
</root>
</mxGraphModel>
```

### Templates for the XML Generation

The `generatePidXmlString` function (refer to figure 17) iterates over the `pidJson` object to build a unique *XML* object for each shape. It does so by inserting the variable's value in the corresponding location in the template. The function implements a distinct template for each of the five shape classes (`pidClass`) since the *XML* objects must be composed of distinct combinations and logic. The following code shows the template code for the boilerplate and equipment shapes (`pidClass="equipment"`):

```
1.  generatePidXmlString: function(pidJson) {
2.
3.      // Builds an HTML-escaped label
4.      const htmlLabel = `&lt;b&gt;%pid-label%&lt;br&gt;&lt;span style=&quot;background-
    color: rgb(0 , 255 , 0)&quot;&gt;&lt;font color=&quot;#ffffff&quot;&gt;&amp;nbsp;%sapient-
    bind%&amp;nbsp;&lt;/font&gt;&lt;/span&gt;&lt;/b&gt;&lt;br&gt;`;
5.
6.      // Add mxGraph and mxGraphModel boilerplate settings
7.      let xmlString = `
```

```
8.     <mxGraphModel dx="${graphSettings.dx}" dy="${graphSettings.dy}" grid="${graphSettings.gr
   id}" gridSize="${graphSettings.gridSize}" guides="${graphSettings.guides}" tooltips="${grap
   hSettings.tooltips}" connect="${graphSettings.connect}" arrows="${graphSettings.arrows}" fo
   ld="${graphSettings.fold}" page="${graphSettings.page}" pageScale="${graphSettings.pageScal
   e}" pageWidth="${graphSettings.pageWidth}" pageHeight="${graphSettings.pageHeight}" backgro
   und="${graphSettings.background}" math="${graphSettings.math}" shadow="${graphSettings.shad
   ow}">
9.       <root>
10.         <mxCell id="0"/>
11.         <mxCell id="1" parent="0"/>`;
12.
13.     // Add Equipment (by iterating over each equipment):
14.     const equipmentCount = pidEquipments.length;
15.     console.log(`Generating XML-tags for ${equipmentCount} equipment instances...`);
16.     pidEquipments.forEach((pidEquipment) => {
17.         xmlString += `
18.           <object id="${pidEquipment.id ? pidEquipment.id : pidEquipment._id}" label="${pid
   Equipment._value !== '' ? pidEquipment._value : htmlLabel}" placeholders="1" pid-
   label="${pidEquipment.pidLabel ? pidEquipment.pidLabel : (pidEquipment.shortName ? pidEquip
   ment.shortName : (pidEquipment.name ? pidEquipment.name : null))}" pid-current-value=""
   pid-function="${pidEquipment.pidFunction}" pid-number="${pidEquipment.pidNumber}" sapient-
   bind=${this.getSapientBindpidEquipment)}">
19.               <mxCell style="${this.concatenateStyles(pidEquipment.styleObject)}"
   vertex="${pidEquipment._vertex}" connectable="1" parent="${0 === pidEquipment.pidLevel ?
   1 : pidEquipment.parentId}">
20.                   <mxGeometry x="${pidEquipment.mxGeometry._x ? pidEquipment.mxGeometry._x
   : 50}" y="${pidEquipment.mxGeometry._y ? pidEquipment.mxGeometry._y : 50}" width="${pidEqui
   pment.mxGeometry._width}" height="${pidEquipment.mxGeometry._height}" as="${pidEquipment.mx
   Geometry._as}">
21.                   </mxGeometry>
22.               </mxCell>
23.           </object>`;
24.     });
25.
26.     // Add instruments, arrows, groups and lines here analogously ...
27.
28.     // Add boilerplate closing tags
29.     xmlString += `
30.     </root>
31. </mxGraphModel>`;
32.
33.     return xmlString;
34. },
```

**Properties of the XML Objects**

For the sake of clarity, table 3 summarizes the properties each template takes in out of all available properties for each `pidVertex` and/or `pidEdge` instance. Properties of *mxGraph* objects should be private. As *JavaScript* does not allow the declaring of private properties, such properties are preceeded by an underscore (e.g. `_id`, `_style`, `_vertex`, etc.) in the P&ID shapes library *JSON* file to distinguish from non-mxGraph properties (e.g. `shapeCategory`, `shapePath`, `shapeThumbnail`, etc.). Table 4 also displays them with an underscore. Nevertheless, all properties are instanciated without an underscore by the template, as is required by the *XML* to be read by the *mxGraph API* loader.

A checkmark in the “Implemented by Templates for” columns means that the property is indeed recursively instanciated with its value for each `pidVertex` and/or `pidEdge` instance. This does not mean that all shapes have a non-empty value. In many cases, shapes of the same `pidClass` have distinct sets of non-empty properties, but the library contains all possible properties nonetheless, with empty-

string values if property is not relevant. For this reason, the properties are instanciated equal to empty-strings in the *XML* object by the template. The *P&ID* shapes library abstracts the geometrical definition of shapes; thus, it is not necessary to go in more detail than this as to which shapes require which properties.

For each iteration, the `_style` property is passed the return value of the `concatenateStyles(styleObject)`, which stringifies the in library configurable `styleObject` to a semi-colon separated string of key-value pairs. Table 5 breaks down the `styleObject` to its composing properties as was done in table 4 for each *XML* object.

**SL**: shape library (default static value)
**SC**: source code (dynamically generated value with code)
**DB**: database (static value fetched from database)
**B**: Boilerplate, **E**: Equipment, **I**: Instruments, **A**: Arrows, **G**: Groups, **L**: Line.

| XML Object | Property | Description | Src | Implemented by Templates for | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | B | E | I | A | G | L |
| **<mxGraphModel>** | | | | | | | | | |
| (graph configurations) | _dx | Specifies x-coordinate of translation/shift (either absolute or relative from parent) | SC | ✓ | - | - | - | - | - |
| | _dy | Specifies y-coordinate of translation/shift (either absolute or relative from parent) | SC | ✓ | - | - | - | - | - |
| | _grid | Boolean (1/0) to activate/deactivate grid | SC | ✓ | - | - | - | - | - |
| | _gridSize | Sets size of grid measures | SC | ✓ | - | - | - | - | - |
| | _tooltips | Boolean (1/0) to activate/deactivate tooltips | SC | ✓ | - | - | - | - | - |
| | _connect | Boolean (1/0) to set if connectable or not | SC | ✓ | - | - | - | - | - |
| | _arrows | Boolean (1/0) to activate/deactivate arrows | SC | ✓ | - | - | - | - | - |
| | _fold | Boolean (1/0) to activate/deactivate folding | SC | ✓ | - | - | - | - | - |
| | _page | Boolean (1/0) to set if page view or not | SC | ✓ | - | - | - | - | - |
| | _pageScale | Number to set page scaling | SC | ✓ | - | - | - | - | - |
| | _pageWidth | Number to set width of page | SC | ✓ | - | - | - | - | - |
| | _pageHeight | Number to set height of page | SC | ✓ | - | - | - | - | - |
| | _background | Color in hexadecimal to set background color | SC | ✓ | - | - | - | - | - |
| | _math | Boolean (1/0) to activate/deactivate math | SC | ✓ | - | - | - | - | - |
| | _shadow | Boolean (1/0) to activate/deactivate shadow | SC | ✓ | - | - | - | - | - |
| **<root>** | | | | | | | | | |
| **<object>** | | | | | | | | | |
| (data-binding and labels) | _id | Unique reference to itself | DB | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | label | HTML-escaped string template for shape label | SC | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | placeholders | Boolean (1/0) to activate/deactivate placeholders | SC | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | pid-label | Dynamically generated and set inside `label` | SC | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | pid-current-value | Default current value for empty string values | SC | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | pid-function | Instrument process engineering function | SC | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | pid-number | Instrument identification number | SC | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | sapient-bind | JSON string with data bindings to legato DB | SC | | ✓ | ✓ | ✓ | ✓ | ✓ |
| (shape metadata) | pidClass | Class of visualization element (E, I, A, G, L) | SL | - | | | | | |
| | shapeDomain | Shape domain in the draw.io standard library | SL | - | | | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | shapeClass | Shape class in the draw.io standard library | SL | - | | | | | |
| | shapeCategory | Shape category in the draw.io standard library | SL | - | | | | | |
| | shapeType | Shape type in in the draw.io standard library | SL | - | | | | | |
| | shapeSubtype | Shape subtype in the draw.io standard library | SL | - | | | | | |
| | shapePath | Shape path (concatenated shape metadata attributes) | SL | - | | | | | |
| | shapeThumbnail | File path to thumbnail for image embedding without mxGraph Viewer | SL | - | | | | | |
| **<mxCell>** | | | | | | | | | |
| (cell properties) | _style | String with semi-colon separated style properties built by the in-loop function call to concatenateStyles(styleObject) | SL | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | _vertex | Boolean (1/0) sets if vertex or not | SL | - | ✓ | ✓ | ✓ | ✓ | |
| | _edge | Boolean (1/0) sets if edge or not | SL | - | | | | | ✓ |
| | _value | HTML value for mxGraph label implementation (overwrites label) | SL | - | | | | | |
| | _connectable | Boolean (1/0) sets if edges can connect to shape | SL | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | _visible | Boolean (1/0) sets if visible or not | SL | - | | | | | |
| | _collapsed | Boolean (1/0) sets if collapsed or not (groups) | SL | - | | | | | |
| | _source | Reference to the source shape of edge (port) | SL | - | | | | | ✓ |
| | _target | Reference to the target shape of edge (port) | SL | - | | | | | ✓ |
| | _parent | Implements hierarchical relationship (important for proper rendering and shape positioning relative to parent) | SL | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| | _children | An array to hold children cells | SL | - | | | | | |
| | _edges | An array to hold edges | SL | - | | | | | |
| | _mxTransient | Array of members not to be cloned by clone | SL | - | | | | | |
| **<mxGeometry>** | | | SL | | | | | | |
| (size and positioning) | _relative | Boolean (1/0) to set mxCell positioning to absolute or relative to parent | | - | | | | | ✓ |
| | _x | Number to set x-coordinate (position) | SC | - | ✓ | ✓ | ✓ | ✓ | |
| | _y | Number to set y-coordinate (position) | SC | - | ✓ | ✓ | ✓ | ✓ | |
| | _width | Number to set width of shape | SL | - | ✓ | ✓ | ✓ | ✓ | |
| | _height | Number to set height of shape | SL | - | ✓ | ✓ | ✓ | ✓ | |
| | _as | Sets geometry type (default is “geometry”) | SL | - | ✓ | ✓ | ✓ | ✓ | ✓ |
| **<mxPoint>** | | | | | | | | | |
| (edge waypoints) | _x | Number to set x-coordinate (position) | SC | - | | | | | |
| | _y | Number to set y-coordinate (position) | SC | - | | | | | |
| | _as | Sets geometry type (default is “geometry”) | SC | - | | | | | |

*Table 4 Overview of all attributes (checkmark marks those implemented in the XML templates).*

.

| JS Object | Property | Description | Src |
|---|---|---|---|
| **styleObject** | | | |
| (all) | shape | Name of shape (maps to shapeName property from database) | SL |
| (equipment) | actuator | Type of actuator (manual, pneumatic, etc.) | SL |
| | direction | Direction of shape | SL |
| | fanType | Type of fan (for fans) | SL |
| (equipment) | columnType | Type of column (for columns) | SL |
| | valveType | Type of valve (for valves) | SL |
| | defState | Default state for valve (closed/open) | SL |
| | actuator | Type of actuator (for valves) | SL |
| (instruments) | mounting | Location of mounting (room, local, field, etc.) | SL |
| | overflow | Overflow value for instruments (fill) | SL |
| | indType | (inst, ctrl func, plc) | SL |
| (arrows) | dy | Geometry of arrow point (y-offset) | SL |
| | dx | Geometry of arrow point (x-offset) | SL |
| | notch | Angle of arrow point | SL |
| (groups) | group | Sets if group or not (for _vertex="1" shapes) | SL |
| | container | Sets if group is container for other cells or not | SL |
| (lines) | edgeStyle | Style of edge geometry (straight, orghogonal, rounded, etc.) | SL |
| | exitX | x-coordinate position of source connection | SL |
| | exitY | y-coordinate position of source connection | SL |
| | entryX | x-coordinate position of target connection | SL |
| | entryY | y-coordinate position of target connection | SL |
| | startArrow | Arrow type of starting edge (source) | SL |
| | startFill | Arrow color fill of starting edge (source) | SL |
| | startSize | Arrow size of starting edge (source) | SL |
| | endArrow | Arrow type of ending edge (target) | SL |
| | endFill | Arrow color fill of ending edge (target) | SL |
| | endSize | Arrow size of ending edge (target) | SL |
| | jettySize | Sets if jetty size should be used or not | SL |
| | orthogonalLoop | Activates/deactivates orthogonal loops | SL |
| (all) | html | Boolean (1/0) activates/deactivates html set in _value property of mxCell | SL |
| | align | Sets the horizontal alignment of the html | SL |
| | verticalLabelPosition | Sets the vertical alignment of the label | SL |
| | verticalAlign | Sets the vertical alignment of the html | SL |
| | labelPosition | Sets the horizontal alignment of the label | SL |
| | dashed | Sets if stroke/border should be dashed or not | SL |
| | dashPattern | Sets dash pattern | SL |
| | strokeColor | Sets color of stroke/border | SL |
| | strokeWidth | Sets width of stroke/border | SL |
| | fillColor | Sets fill color of mxCell | SL |
| | fontSize | Sets fontSize of label | SL |
| | fontColor | Sets fontColor of label | SL |
| | rotation | Sets rotation of mxCell (0, 90, 180, 360, etc.) | SL |

*Table 5 Detailed decomposition of the styleObject.*

# 5 P&ID Graphing Algorithm

## 5.1 Overview

This section will explain the intricacies and working principle of the `vertexPlacement` function mentioned previously in section 4.3.4. The vertexPlacement function encapsulates the *P&ID* graphing algorithm code entirely. Code snippets and diagrams will be presented to aid in the explanation, as the function itself spans over several hundred lines, too many to be presented in significantly long snippets in this document. The graphing algorithm for the generation of *P&ID* visualizations represented a significant challenge, as no similar works were found, with comparable characteristics and/or goals. The developed concept is thus the integration of research from distinct areas, as are those of graph layout and graph drawing.

**Initial Expectations**

- Simplicity over efficiency of the algorithm as to allow later improvements and since the creation of *P&ID* visualizations is not time critical
- Algorithm concept for *P&ID* visualizations works no matter the complexity of the modelled process engineering plant
- Optimal packing of the *P&ID* visualization for limited screen sizes
- Ability of progressively enhancing the algorithm for creation of better and more complex visualizations without change in concept
- Implementation of the algorithm for the example *Aida brewery* plant

**Encountered Issues**

- A *P&ID* requires complex and unapparent positioning logic, contrary to regular graphs, which cannot be laid out by means of existing algorithmic approaches
- Inconsistencies in the plant instance hierarchy model for the purpose of this algorithm, such as groups containing other groups but also other shapes as children
- Inconsistencies lead to irregular shape placement in the majority of cases for the *P&ID* visualization, what in turn calls for more programming logic, usually too much due to the declarative nature of the algorithm
- Relative positioning sometimes suboptimal, but still the best alternative
- Much coding effort and little progress in improving the algorithm towards the end

**Solution:**

Developed algorithm consists of two parts relatively independent from each other:

- **Specification of Constraints:** logically tagging shapes according to their properties (e.g. `pidClass`, `pidCategory`, or more specifically `shapeName`, `styleObject.mounting`) in order to later target tagged shapes specifically to apply certain positioning rules.
- **Vertex Placement via Positioning Rules:** rules apply specific positioning logic to shapes with specific tags and are programmed declaratively (more rules can be added later). Each rule encapsulates an algorithmic approach for the positioing logic (which can be later progressively enhanced for each rule).

The implementation details of both main parts of the *P&ID* graphing algorithm will be described in the following sections.

## 5.2 Specification of Constraints

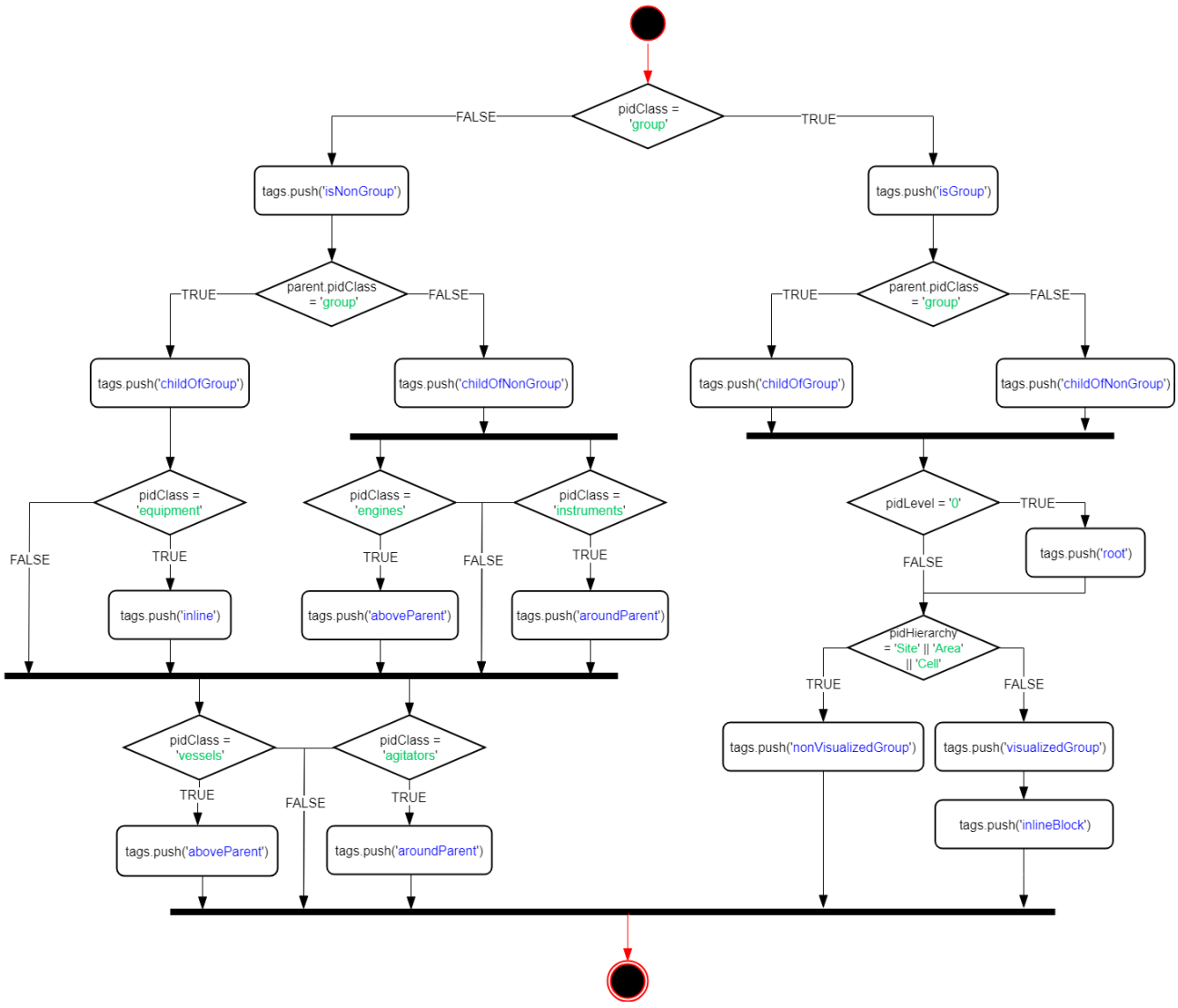



*Figure 28 UML 2.0 activity diagram of constraint specification for shapes.*

The specification of constraints is programmed declaratively in the first part of the `vertexPlacement` function. Figure 27 illustrates the activity diagram of the specification process through which all shapes are subject to be iteratively tagged. As can be seen from the diagram, the system for tagging can be easily enhanced and expanded for more specific tagging of the shapes, what in turn would allow for a more specific targeting of the positioning logic in form of rules. The system is implemented as a nested tree of conditional statements, in which the nesting level of the statement corresponds to the specificity of the tag, with more specific tags on the bottom and more general ones tending to the top. The specification of constraints in form of tags had several benefits for the specific case of P&ID graphing:

- Constraints in form of tags means any shape can be tagged as much as needed to keep refining the logic used for its positioning.
- Specification of constraints allows logic to be targeted specifically for certain shapes,
- Tags make it very clear as to which shapes are positioned with which rules, as the tags are contained in an array as properties of the shape itself and can thus be logged (see figure 28).
- Constraints allow progressive enhancement and are loosely coupled to the actual positioning logic itself

| lvl | id | name | pidClass | parentId | tag0 | tag1 | tag2 |
|---|---|---|---|---|---|---|---|
| 6 | 21728 | "CV440" | "equipment" | 21699 | "isShape" | "childOfGro… | "inline" |
| 6 | 21729 | "CP40" | "equipment" | 21699 | "isShape" | "childOfGro… | "inline" |
| 5 | 21699 | "WaterInjec… | "group" | 21695 | "isGroup" | "childOfGro… | "innerGroup" |
| 6 | 21730 | "Motor" | "equipment" | 21700 | "isShape" | "childOfNon… | "centeredAb… |
| 6 | 21731 | "KettleSens… | "instrument" | 21700 | "isShape" | "childOfNon… | "aroundPare… |
| 6 | 21732 | "KettleSens… | "instrument" | 21700 | "isShape" | "childOfNon… | "aroundPare… |
| 5 | 21700 | "Vessel" | "equipment" | 21695 | "isShape" | "childOfGro… | "nucleusGro… |
| 4 | 21695 | "LauterKett… | "group" | 21694 | "isGroup" | "childOfGro… | "innerGroup" |
| 5 | 21701 | "CV550" | "equipment" | 21696 | "isShape" | "childOfGro… | "inline" |
| 5 | 21702 | "Compressor" | "equipment" | 21696 | "isShape" | "childOfGro… | "inline" |

*Figure 29 Example log of several shape properties after constraint specification (tags highlighted in yellow)*

## 5.3 Vertex Placement

### 5.3.1 Conceptual Overview

The following points summarize the second part of the `vertexPlacement` function, for the actual positioning of the cells.

- Graph settings like `spacing`, `margin`, `pageWidth` and `pageHeight` implemented as variables allow fine tuning (progressive enhancement) of the algorithm during development
- Vertex positioning of cells is always relative to the origin (0, 0) of the cell's parent and is dictated by rules which apply to declaratively target specifically tagged elements
- Each rule encapsulates distinct positioning logic (algorithmic approach for the calculation and setting of proper `mxGeometry._x` and `mxGeometry._y` properties for the cells).

- Edge placement handled automatically by *mxGraph API* (optimized to minimize crossings and apply line jumps where needed)

Aligning to the data structure at hand (`pidJson` object), which consists of nested vertex and edge objects and represents the plant instance hierarchy, both an inclusion graphing and tree methodology stood out as appropriate approaches for positioning of *P&ID* elements parting from a hierarchical plant model. A conceptual overview of how the plant hierarchy maps to the visualized components or shapes is illustrated in figure 29. As mentioned before, the `traverseAndSort` function travels through the node tree of the hierarchy *depth-first* (starting from the root node and exploring as deep as possible along each branch before *backtracking*). The function returns the path of traversal, with which the order of the vertices is to be sorted (order shown by numbered nodes in figure 29). The `pidJson` object passed to the graphing algorithm is sorted and therefore iterated in this order. It is in this way that the *P&ID* graphing algorithm resembles a standard tree-like graphing algorithm, and by means of which the visualization logic can be sequentially applied to allow for inclusion graphing for children of groups.

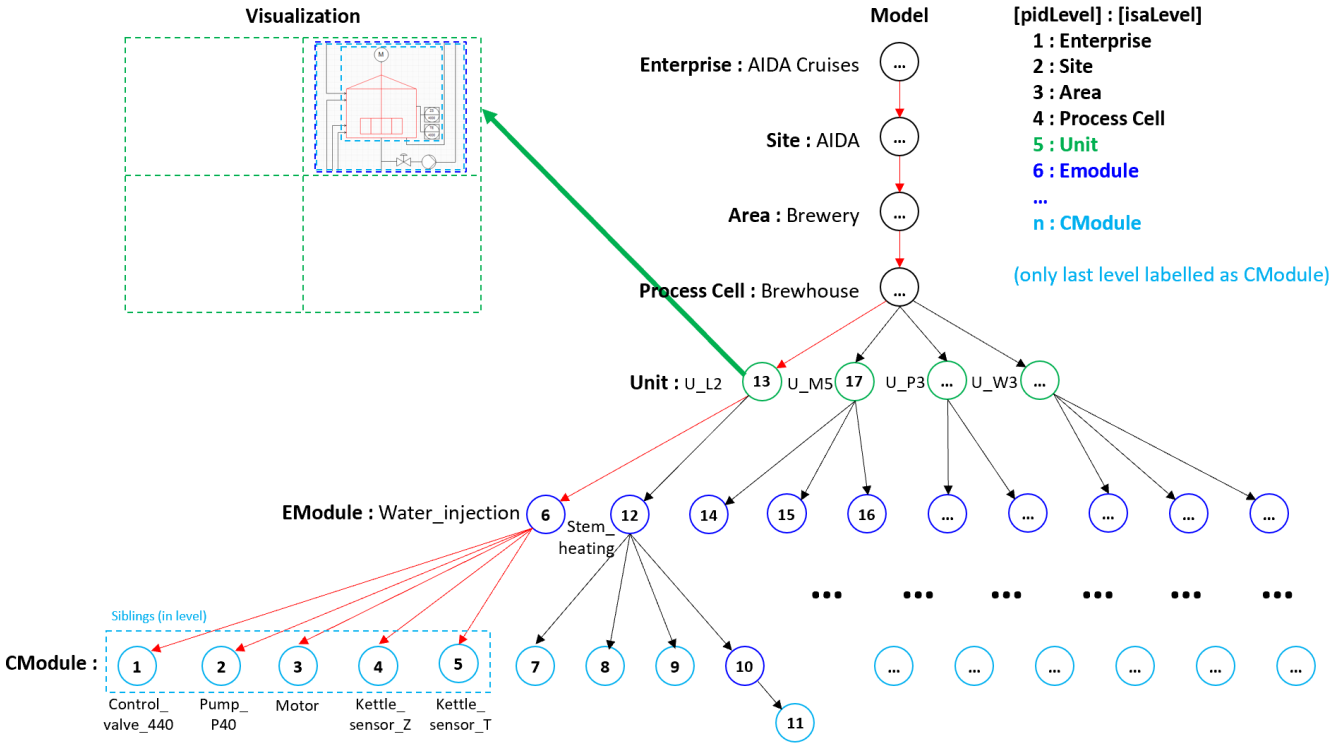



*Figure 30 Mapping concept of hierarchical model to visualization.*

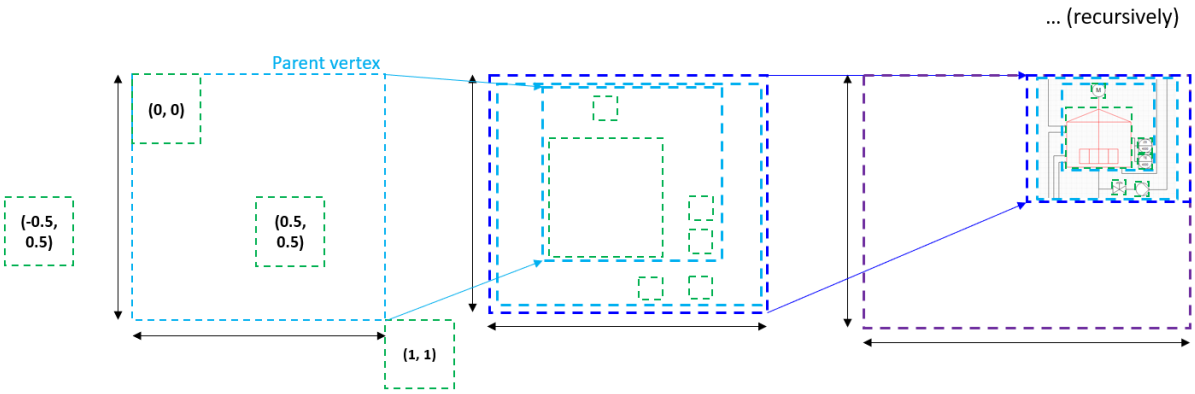



*Figure 31 Relative positioning of cells.*

### 5.3.2 Positioning Logic

As previously stated, the positioning logic is entirely decoupled to the constraint specification. It is after all shapes are tagged (tags pushed to `tags` array), that the vertex placement or the setting of the x- and y-coordinates of the *P&ID* shapes actually takes place. As determined at the start of this project, the positioning logic should at least handle all shapes present in the example *Aida brewery*, rather than all shapes present in the library (478 in total). Still, the positioning logic for each tag is too long to be presented in detail in this document. Figure 31 presents an overview of the positioning logic for all tags that can be specified according to figure 27. Next, 3 distinct cases will be exemplified (highlighted with color in figure 31).

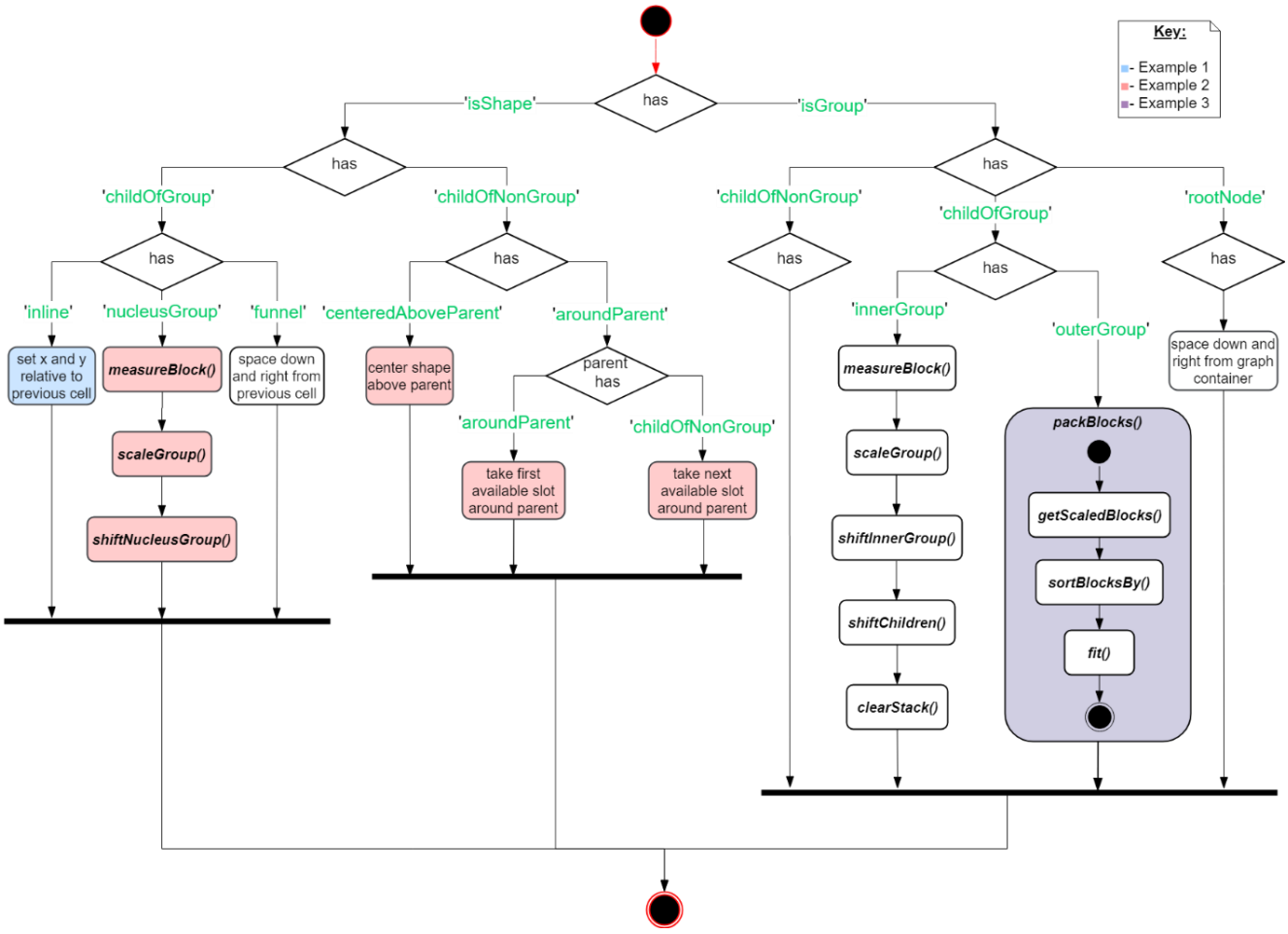



*Figure 32 UML 2.0 activity diagram of vertexPlacement function.*

### Example Case 1 – #inline

Tagged with `#inline` are shape cells to be positioned horizontally offset and vertically centered to its preceding cell. This is the case for consecutive valves on the same pipe for example, or a pump, or actually most other in-pipe elements. The geometry and positioning logic are shown bellow in figure 31, and the functions that implement this in blue in figure 30.

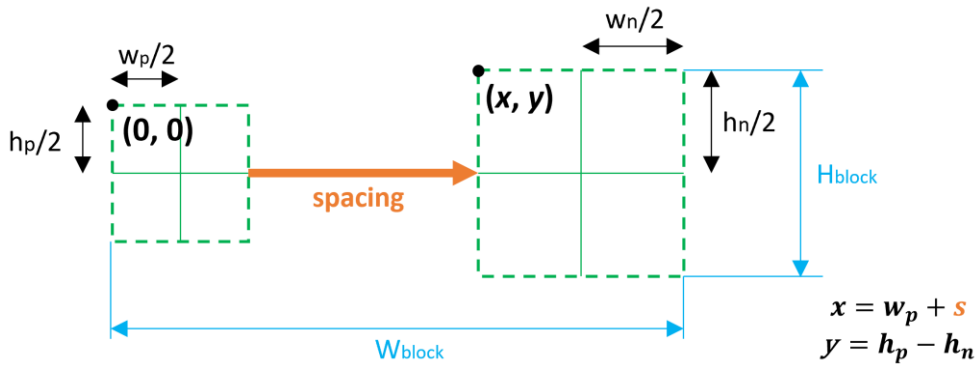



*Figure 33 Geometry and positioning of inline elements.*

**Example Case 2 – #nucleusGroup**

This case exemplifies the positioning logic of a nucleus group and its children (shown in red in figure 30). A sample case of this positioning is for that of a vessel, with sensors (`children`) to be positioned around it. Figure 32 shows the algorithmic approach of this logic for `1` to `n` number of sensors around the parent vessel.

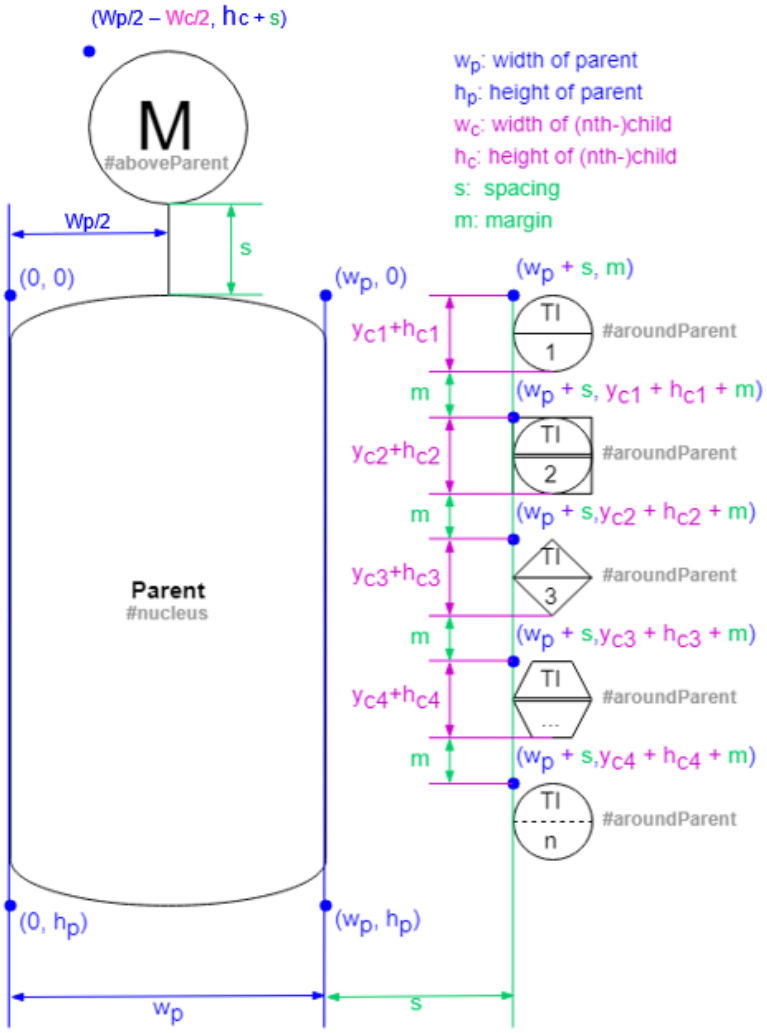



*Figure 34 Geometry and positioning of a nucleus group (pressurized vessel) and its children (motor and instruments).*

**Example Case 3 – #outerGroup**

This case exemplifies the positioning logic of group cells tagged with `#outerGroup`, which are group cells with only group cells as children. For these cases in which all children are groups (rectangular blocks), a *block packing algorithm* (orthogonally packing rectangles in an auto-scaling containing group) was determined as the best positioning logic. Contrariwise, when shape (non-group) cells are present as children, other logic like `#inline` logic in case of successive valves has to be applied, which inhibits the application of this algorithm for positioning of all children, regardless of their type. The working principle of this positioning logic is illustrated in figure 34 and the functions that implement this (shown in purple in figure 30) are further described bellow:

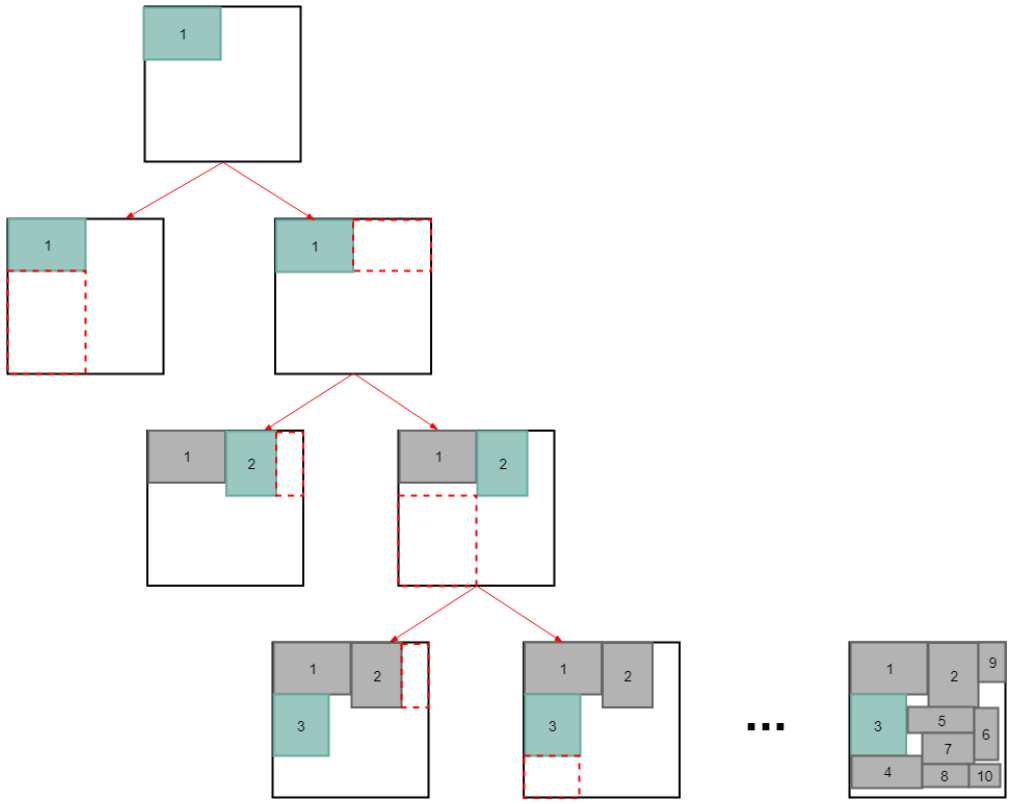



*Figure 35 Working principle of block packing algorithm.*

1. **`packBlocks`** – a function to encapsulate all others. Takes in the children groups in a certain order (`blocks` array), packs them in an optimized configuration (sets `mxGeometry._x` and `mxGeometry._y` of each) and scales the group to fit the blocks in that configuration (sets `mxGeometry._width` and `mxGeometry._height`). It does so by calling the following functions sequentially.
2. **`getScaledBlocks`** – takes in the blocks (children), adds a `margin` to each block side, and returns new scaled dimensions for each block.

3. **`sortBlocksBy`** – sorts the `blocks` array by defined configuration order set in the `sortOrder` variable (refer to figure 35):
    a. **`none`** – returns array in same order
    b. **`maxSide`** – sorts by descending greater side of each block (gave best results for minimizing area and therefore chosen as default)
    c. **`flows`** – sorts by descending number of flows with reference to the block
    d. **`width`** – sorts by descending block widths
    e. **`height`** – sorts by descending block heights
    f. **`area`** – sorts by descending calculated area (`width * height`)
    g. **`random`** – sorts by a random order
4. **`fit`** – takes in the by `sortOrder` value sorted blocks array and packs each block in that order to optimize the containing area on each placed block (decides where to place each block based on which placement will result in least total area of container). [Gordon and Contributors, 2016].

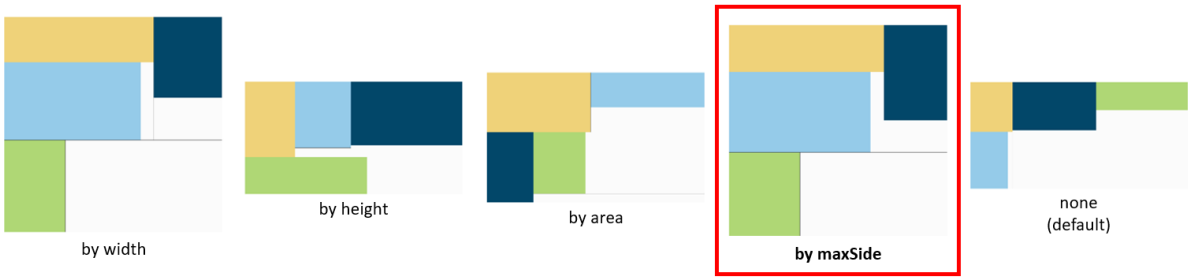



*Figure 36 Results of Binary Tree Packing algorithm simulation [21] for several sample blocks and distinct sortOrder values.*

After first developing the previously described *P&ID* graphing algorithm, the output *XML* files would not always render. This was due to inconsistencies in the model, for example, that connections (or line shapes) should have the parent attribute set to the `rootNodeId` in order to be rendered by *mxGraph*. Such technicalities presented themselves as obstacles which greatly delayed the progressive enhancement of the algorithm, in order to achieve better results. Nevertheless, these issues were tackled one-by-one, until the code was robust enough to render the corresponding *P&ID* visualization of any selected root node. After that of course, came the refinement of the algorithm, which represented the longest sprint of the project. After many time and effort, progress with the algorithm came to a stand still. At that point, it was determined to continue with the next sprint, due to time constraints. Figure 36 shows the best and final results achieved for the automated generation of the *P&ID* of the selected enterprise level root node *Aida Cruises*. The following issues are apparent: the layout is still not perfect, some connections are render collapsed to others due to the automatic *line-crossing optimization* of the algorithm, and the *data-bindings* are not yet implemented (the label includes the `id` of each node in place of the process variable, if any).

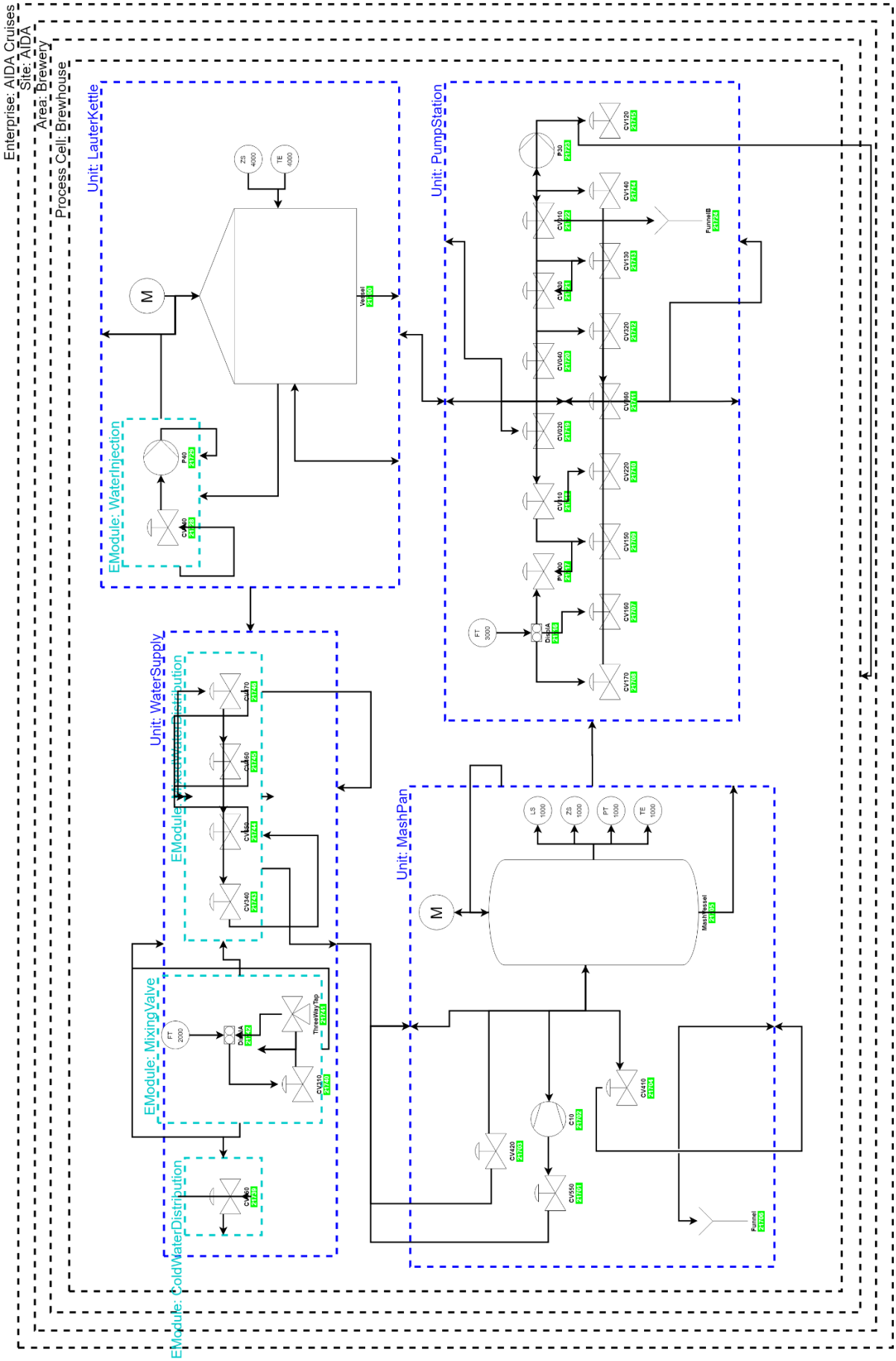



*Figure 37 Final result for the automatically generated P&ID of Aida Cruises (selected root node).*

# 6 Synopsis

This chapter summarizes once again, whether the results of this bachelor thesis fulfill its goals. For this purpose, a detailed breakdown of each requirement into its implementation details (or what was actually done) is presented before concluding if such requirement was met in terms of what was expected.

**R1: Library of modular P&ID Visualization Components According to Industry Standards**

- ✓ *Object-oriented* abstraction of industrial process engineering elements into *UML 2.0* classes based on their inherent geometrical and functional traits (class diagram).
- ✓ Conception of a library of modular and composable *P&ID* visualization components in *SVG* format according to the current industry standards.
- ✓ Static geometrical definition of visualization components in a *P&ID* shapes library implemented as a *JSON* file for efficiency, portability and ease of personalization and maintainability. Each shape corresponds to a *JSON* object with a defined set of properties with predefined values (defined by library version), or a default value if left blank.
- ✓ Shape instances inherit object properties from their parent *P&ID* class and category based on the data model defined as a *UML 2.0* class diagram. This enables automatic propagation of inherited and composed styles throughout library for ease of user personalization.
- ✓ Shapes display real-time plant process values if a data binding to process variable exists.
- ✓ Semi-colon separated string of styles de-structured into styles object for targeted configuration of individual styles. Styles object then concatenated back to string on *XML* generation.

→ **Requirement met**

**R2: User Friendly Graphical User Interface (GUI) Boardlet for Creation of P&ID Visualizations**

- ✓ User friendly boardlet with a simple interface for generating new or updating previously generated *P&ID* visualizations.
- ✓ No configurations needed and abstraction of inner workings for the user.
- ✓ User can easily select the root node he wants to create the *P&ID* for.
- ✓ File input for the selection of the desired version of the *P&ID* shapes library.
- ✓ Buttons for generating the *XML* of the visualization, for downloading the visualization in both *JSON* and *XML* format, and to upload the *XML* visualization file to *Sapient Engine* file system for the *Legato Graphic Designer* boardlet to import and render.
- ✓ Visual feedback: progress bar to display *XML* object generation process.
- ✓ Dashboard with boardlets: P*&ID Creator, Node Tree, P&ID Viewer*.

→ **Requirement met**

**R3: Client-Side Script for the Automated P&ID Visualization Generation as an XML File**

- ✓ Required from user is only the selection of the desired *P&ID* shapes library version for the visualization and no additional configurations.
- ✓ Script encapsulates all required business logic in a single modular and composable, well-documented *JavaScript* file.
- ✓ Separation of primary concerns: presentation logic, database queries, data mappings, graphing algorithm and *XML* generation are all separate and inter-independent from each other.

→ **Requirement met**

**R4: Mapping of Physical Plant Instances to Corresponding Visualization Component**

- ✓ Mapping to work with minimal changes to the original data model for an unobtrusive implementation of the automated *P&ID* visualization generator and to avoid the need for new tables and fields in database.
- ✓ Minimum database requirement is a `shapeName` attribute to be property set in the model and thus in the database.
- ✓ Connections don't require changes in model. The `shapeName` attribute for each line shape is determined via logic.

→ **Requirement met**

**R5: Automatic Type Detection and Simplification of Connections**

- ✓ Logic for setting the corresponding shapeName property to all connections: differentiate between data, process, connection and signal lines. Because of this, no need to specify a `shapeName` for connections in the data model.
- ✓ Connections with multiple waypoints simplified by skipping intermediate ports, until a shape to shape connection (from start source to end target) is reached without need of modifying how connections were originally modelled (from port to port).
- ✓ Orthogonal line shapes optimized for minimal crossings and shortest routing between source and target.

→ **Requirement met**

**R6: Declarative specification of Graphing Constraints in Form of Tags**

- ✓ Declarative approach of tags which allow targeting specific shapes to be positioned according to specific set of positioning rules.
- ✓ Tags are loosely coupled so they don't intervene with the algorithm, rather define the algorithm to be run.
- ✓ Separates vertex placement logic for shapes to be positioned with distinct positioning rules.
- ✓ Vertex placement algorithm can be easily progressively enhanced through the addition of more and more tags.

→ **Requirement met**

**R7: P&ID Graphing Algorithm**

- ✓ Research and analysis of state-of-the-art graphing algorithms.
- ✓ Graphing algorithm works for any user-selected root node to create the corresponding *P&ID* of any part of the modelled plant.
- ✓ Simplicity over efficiency of the algorithm as to allow later improvements and since the creation of *P&ID* visualizations is not time critical.
- ✓ *Depth-first post-order* instance hierarchy traversal to get nodes in graphing order.
- ✓ *Block-packing algorithm* for the positioning of groups in groups to minimize the area.
- ✓ Algorithm concept for *P&ID* visualizations will work no matter the complexity of the modelled process engineering plant (concept proved).
- ✓ Ability of progressively enhancing the algorithm for creation of better and more complex visualizations without change in concept.
- ✓ Implementation of the algorithm for the example *Aida brewery* plant.

→ **Requirement partially met**

Concept works and can be progressively enhanced to achieve even better layouts and clearer *P&ID* visualizations, but this would require exponentially more programming, what can turn out to be counter-productive. The layout of the example *Aida brewery* plant still has much room for improvement. The concept solution nevertheless works, its continuation represents therefore an interesting project for the future.

**R8: Dynamic Real-Time Display of Process Variables in the P&ID Visualization**

- ✓ *P&ID* visualization with real-time updating shapes and shape labels.
- ✓ Different types of display of process values depending on data type of process variable and on shape category (for example: Boolean values set fill color for valves, but not for tanks).
- ✓ Components encapsulate a uniform set of data bindings to the actual process values and display values in real time.
- ✓ Default settings to override labels for shapes with data bindings to empty values.
- ✓ One-way data bindings that update automatically on the client-side instead of on the server-side for optimizing performance.
- ✓ Data bindings implemented with the sapient-bind property of the shape's *XML* user-defined object which uses the *mxGraph API* already (placeholders).

→ **Requirement partially met**

Support for dynamic rendering implemented via the existing `sapient-bind` *Legato API* but not yet tested due to missing connection to real or testing gateway. The developed solution supports both Booleans and integer values, to both change shape color and render the value of process variables. Still, many more data types and configurations should be possible to allow for the appropriate rendering of this dynamic data in the *P&ID* visualization.

**R9: Prototypal Implementation in the Infrastructure of a MES (Legato Web Application) and Documentation**

- ✓ *Component-driven*, modular design of boardlet implemented with the *Ember.js* framework used in the *Legato Web Application.*
- ✓ Evaluation of the system: functionality, performance and scalability.
- ✓ Clear documentation of code.
- ✓ Document with next steps in case of interest on continuing development.
- → **Requirement met**

# 7 Implementation in an Industrial Context

## 7.1 Conclusions

In the previous chapters, both the working principles and the accomplished results where presented. It was demonstrated that the developed solution for the automated generation of *P&ID* visualizations works in concept. For an implementation in an industrial context though, where stricter, more demanding standards in quality and performance must be met, a couple of things need yet to be considered.

What seemed would be the greatest challenge at the start of the project, turned out to be one even greater than initially thought: the graphing algorithm for *P&ID* visualizations. *P&IDs* do indeed share similarities to ordinary (mathematical) graphs. Nonetheless, they cannot be positioned algorithmically by means of a mathematical approach or an existing methodology, but rather analytically, based on certain inherent positioning characteristics (for example, a child motor with a vessel as parent should be placed on top of its parent, the vessel). After many programming hours invested and no significant improvement in the vertex placement algorithm, it was evidenced that it had come to a still stand. Although a coarse placement of *P&ID* shapes was achieved for simple models (the *Aida brewery* plant), the refining of this positioning, something required by industrial grade *P&ID* visualizations, signifies an exponentially more challenging task.

In the case of the example *Aida brewery*, most if not all inherent logic was extracted and used for the creation of the positioning rules already. The *P&ID* itself did not have many more trivial positioning criteria, by which the algorithm could still have been empirically enhanced by humans, at least rapidly. In other words, everything for which there is a certain logic behind with respect to its positioning had already been placed according to this logic. Moreover, in the limited developing phase (total project time constrained to 6 months), it was common that fixing certain parts of the algorithm would break others, due to the relative positioning of all shapes and to complex interdependencies in the algorithm.

## 7.2 Evaluation

Despite the issues mentioned in the previous section, the *draw.io* diagraming tool makes it easy to manually edit the generated *P&ID* visualization via a simple, user-friendly interface. All the user has to do is copy and paste the *XML* string into the *draw.io* editor under *Extras >> Edit diagram…*. This way, the remaining vertices can be manually drag-and-dropped in place, what automatically reflects the changes directly in the copy-pasted *XML* string. This enables the user to further enhance the layout of his *P&ID* visualization as he wishes, to afterwards export is as an *XML* file for the *P&ID Viewer* boardlet to load and render in the *Legato Web Application*.

The concept of the solution works, and the implementation for the *Aida brewery* plant went well-enough to assert that an improved version of this solution will eventually be valid for an application in an industrial context. It is evident that much refining is to be done, if continued interest in this project remains, but the *proof of concept* was achieved successfully. Additionally, although the quality of the produced *P&ID* visualizations concerning its layout might not at first be suited for customers in the field, the other functionalities that the solution provides are not to pass unnoted. This include: a library of composable process engineering shapes according to industrial standards, the mapping of the selected root node's hierarchy in the database to *P&ID* shapes, the instantiation of these in the *XML* of the visualization (even though maybe not perfectly laid out), the data binding of each of this shapes for the dynamic, real-time rendering of process variables, the simplification of eventually inconsistently modelled connections and their optimization to minimize edge crossings, and most significantly, the abstraction of all this things into a user-friendly graphical user interface within the *MES* itself. Thus, and regardless if the *P&ID* must be manually edited at the end, the user already saves a significant amount of time for the creation of new and updating of existing *P&ID* visualizations if any changes in the plant model take place. The major goals of the project were therefore satisfactorily met, regardless of the graphing algorithm not being as performant as expected.

## 7.3 Next Steps

Further down the road, what could most certainly be of great value and results for this project is the application of machine learning for pattern recognition of more intricate, hidden positioning logic inherent to *P&ID* visualizations, by means of which more rules can be derived. Existing P&ID models could be fed to a neural network, for it to automatically detect the inherent positioning characteristics behind *P&IDs* in general. Furthermore, machine learning algorithms could also well be applied for the positioning of the vertices themselves by letting the neural network learn the appropriate positioning parameters by itself, valid for any *P&ID*. Computing power enables this way to iterate over the progressive enhancement process for thousands if not millions of times. What a human would take days or even months to program, a computer could achieve in minutes or even seconds. Considering this, the developed solution is, in concept, of great worth for a later pursuit of a deployable solution to the initially defined problem: the automated generation of modular and dynamic plant *P&ID* visualizations.

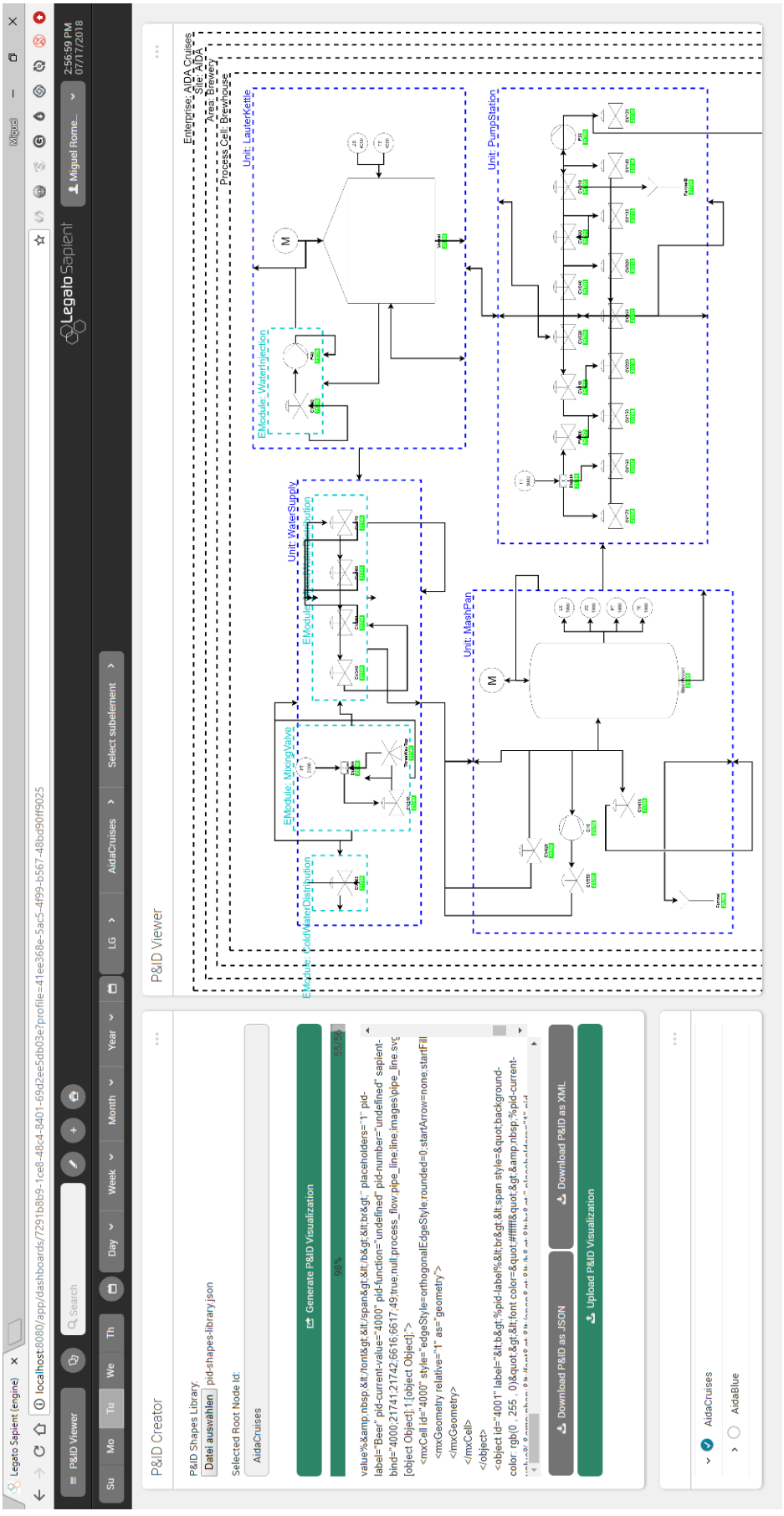



*Figure 38 Final result of the prototypal implementation of the developed solution (P&ID Viewer dashboard).*

# List of Figures

# List of Tables

# Abbreviations

| Abbreviation | Description |
|---|---|
| API | Application Programming Interface |
| CPS | Cyber Physical System |
| GUI | Graphical User Interface |
| ISA | International Society of Automation |
| JS | JavaScript |
| JSON | JavaScript Object Notation |
| MES | Manufacturing Execution System |
| ORM | Object Relational Mapping |
| PCS | Process Control System |
| PIP | Process Industry Practice |
| P&ID | Piping and Instrumentation Diagram |
| ProcAppCom | Process Application Composer |
| SVG | Scalable Vector Graphics |
| SysML | Systems Modelling Language |
| UML | Unified Modelling Language |
| UML-PA | Unified Modelling Language for Process Automation |
| VDI | Verein Deutscher Ingenieure |
| XML | Extensible Markup Language |

# 8 Bibliography

**[1]** Gilad, I. (2018). Why you should stop using product roadmaps and try GIST Planning [Blog]. Retrieved from https://hackernoon.com/why-i-stopped-using-product-roadmaps-and-switched-to-gist-planning-3b7f54e271d1

**[2]** "Systems & Control Engineering FAQ | Electrical Engineering and Computer Science". engineering.case.edu. Case Western Reserve University. 20 November 2015. Retrieved 27 June 2017.

**[3]** Enterprise control. (2018). Retrieved from https://en.wikipedia.org/wiki/Enterprise_control#ISA95_.E2.80.9Clevels.E2.80.9D_for_enterprise_integration

**[4]** VDI 3682 – Formalized Process Description - Professur für Automatisierungstechnik. (2018). Retrieved from https://www.hsu-hh.de/aut/forschung/forschungsthemen/ontology-engineering-for-collaborative-embedded-systems/ontology-design-patterns-for-the-manufacturing-domain/vdi-3682-formalized-process-description

**[5]** What are Piping & Instrumentation Diagrams. (2018). Retrieved from https://www.lucidchart.com/pages/p-and-id

**[6]** Short Track to MES. (2016). Close The Gap - Legato.

**[7]** "What is the difference between the Web and the Internet?". W3C Help and FAQ. W3C. 2009. Archived from the original on 9 July 2015. Retrieved 16 July2015.

**[8]** "CSS developer guide". Mozilla Developer Network. Retrieved 2015-09-24.

**[9]** Flanagan, David. JavaScript - The definitive guide (6 ed.). p. 1. JavaScript is part of the triad of technologies that all Web developers must learn: HTML to specify the content of web pages, CSS to specify the presentation of web pages, and JavaScript to specify the behavior of web pages.

**[10]** JavaScript. (2018). Retrieved from https://en.wikipedia.org/wiki/JavaScript

**[11]** Fennell, Philip (June 2013). "Extremes of XML". XML London 2013: 80–86. doi:10.14337/XMLLondon13.Fennell01. ISBN 978-0-9926471-0-0.

**[12]** "Doug Crockford "Google Tech Talks: JavaScript: The Good Parts"". 7 February 2009.

**[13]** A Relational Model of Data for Large Shared Data Banks". Communications of the ACM. 13 (6): 377–387. doi:10.1145/362384.362685.

**[14]** "A Relational Database Overview". oracle.com.

**[15]** Fetch API. (2018). Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

**[16]** JSON API — Latest Specification (v1.0). (2018). Retrieved from http://jsonapi.org/format/

**[17]** Vogel-Heuser, B. (2018). Softwareentwicklung für Betriebliche Anwendungen. Presentation, Technische Universität München - AIS Lehrstuhl.

**[18]** Graph Theory. (2018). Retrieved from https://en.wikipedia.org/wiki/Graph_theory

**[19]** Mashaghi, A.; et al. (2004). "Investigation of a protein complex network". European Physical Journal B. 41 (1): 113–121

**[20]** IAT 355 Graphs.ppt. (2018). Retrieved from https://slideplayer.com/slide/12488750/

**[21]** Code inComplete. (2018). Binary Tree Bin Packing Algorithm. https://codeincomplete.com/posts/bin-packing/.