DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Software toolkit to manage quality reports in automotive industry

Report submitted to Federal University of Santa Catarina

as requisite to the approval of discipline:

DAS 5511: Final Project Work

Vinícius Heck Peiter

Dingolfing, December 16th, 2019

Software toolkit to manage quality reports in automotive industry

Vinícius Heck Peiter

This monograph was corrected in the context of the discipline

DAS 5511: Final Project Work

and approved in its final form by

Control and Automation Engineering Course

Prof. Ricardo José Rabelo

____

Examination Board:

Stephan Görgens
Company Advisor

Prof. Ricardo José Rabelo
Professor Advisor

Prof. Marcelo de Lellis Costa de Oliveira
President of the Board

Prof. Hector Bessa Silveira
Evaluator

Nicholas Wagner
Student Debater

Ricardo Fileti Marcon
Student Debater

Acknowledgements

I would like to thank my family and friends for their support in this journey. Thanks for my parents who have always supported me and believed in my potential.

Special thanks to my fiance Carol who not only encouraged me to pursue an experience abroad for my personal and professional growth, but also came with me to Germany and was always by my side when I needed. You are wonderful and I am really lucky to have you.

Thanks to the team that welcomed me in BMW Group Plant Dingolfing, TG-300. Thanks for the instruction, guidance and friendships. Each one of you was very important to me throughout the last year and I am glad to have had the opportunity to be a part of your team.

Thanks to the former intern Luiz Arthur, who started the research that was the base for my work. Without your assistance I would not have gotten this far.

Thanks to my supervisor Stephan, who was always knowledgeable and did what he could to help me improve myself and my work. You were always there to discuss technical and theoretical aspects of my work, send me to training opportunities that were relevant to my work and even make a book club to read and discuss a book that supports the theoretical background of my work. You always make sure that interns should do relevant work that will be interesting for them and teach them important skills for their professional life, and I am thankful for that.

Thanks to my advisor Professor Ricardo, who helped me to ensure that this opportunity would benefit my career both professionally and academically. Thank you for being always available to give me your experienced insights first on the paperwork that made this opportunity real, then on my work, making it more relevant.

When software is done right, it requires a fraction of the human resources to create and maintain. Changes are simple and rapid. Defects are few and far between. Effort is minimized, and functionality and flexibility are maximized.

—Robert C. Martin, *Clean Architecture*

Abstract

Within the data analysis tools used in BMW Group Quality IT, SAP BusinessObjects Business Intelligence Suite is the most widely used, processing massive amounts of data from eleven different BMW Group Plants and automatically generating tens of thousands of quality reports every day. However, this tool was not designed to work in such a large scale and lacks basic functionality to manage its platform, motivating the internal development of extensions to customize its usage. In 2018 the Quality IT obtained access to the platform through the RESTful API and started making script prototypes to explore its possibilities. Although the results showed that this approach was promising, some issues in the API required careful handling and the script prototypes were not flexible enough to be extended or modified for different use cases. This project consists of a framework and a set of tools that standardize the way the RESTful API of SAP BO can be used in BMW Group, solving the common issues encountered in previous prototypes and using best practices of software development and architecture in order to enable effortless extension and maintainability. The framework was designed using SOLID principles for Object-Oriented Programming and implemented using Scrum and Test-Driven Development in Python. Three prototype tools were developed using this framework to perform platform management tasks saving thousands of work hours from the Quality IT department.

Keywords: Software Architecture, SOLID Principles, RESTful API, Python, SAP BO, Business Intelligence

Resumo

Dentro das ferramentas de análise de dados utilizadas no departamento de TI de Qualidade do Grupo BMW, o SAP BusinessObjects Business Intelligence Suite é o mais utilizado, processando grandes quantidades de dados de onze fábricas diferentes do Grupo BMW e gerando automaticamente dezenas de milhares de relatórios de qualidade todos os dias. No entanto, essa ferramenta não foi projetada para funcionar em tão grande escala e carece de funcionalidades básicas para gerenciar sua plataforma, motivando o desenvolvimento interno de extensões para customizar seu uso. Em 2018 o TI de Qualidade obteve acesso à plataforma através da API RESTful e começou a fazer protótipos de scripts para explorar suas possibilidades. Embora os resultados mostrassem que essa abordagem era promissora, algumas questões na API exigiam um manuseio cuidadoso e os protótipos de script não eram suficientemente flexíveis para serem estendidos ou modificados para diferentes casos de uso. Este projeto consiste em um framework e um conjunto de ferramentas que padronizam a forma como a API RESTful do SAP BO pode ser utilizada no Grupo BMW, resolvendo os problemas comuns encontrados em protótipos anteriores e utilizando boas práticas de desenvolvimento e arquitetura de software para permitir a extensão e a manutenção descomplicada. O framework foi projetado usando princípios SOLID para Programação Orientada a Objetos e implementado usando Scrum e Desenvolvimento Orientado a Testes em Python. Três protótipos de ferramentas foram desenvolvidos utilizando este framework para executar tarefas de gerenciamento da plataforma, economizando milhares de horas de trabalho do departamento de TI de Qualidade.

Palavras-chave: Arquitetura de software, Princípios SOLID, API RESTful, Python, SAP BO, Business Intelligence

List of Figures

Figure 1 – SAP BO Platforms in BMW Group 27
Figure 2 – Instance statuses in SAP BO 29
Figure 3 – The Cockpit home page of TLK Dingolfing 31
Figure 4 – KPI visualization in the Cockpit 31
Figure 5 – The TDD Cycle 38
Figure 6 – Component Diagram of SAP Toolkit Project 46
Figure 7 – First attempt to model the restful module 50
Figure 8 – Defining Request and Response data structures 51
Figure 9 – The RequestSenderBase interface 52
Figure 10 – The RequestFactory class 53
Figure 11 – Class diagram of the *sap_framework* component 57
Figure 12 – Git branching illustration 61
Figure 13 – *sap_status* repository structure with *sap_framework* as submodule . . 63
Figure 14 – *sap_framework* directory structure 65
Figure 15 – Implementation of data structure classes in Python 67
Figure 16 – Interface implementation with abstract class in Python 68
Figure 17 – The ColtrollerMaker class implementation 69
Figure 18 – Example facade method implemented in RequestColtroller class 70
Figure 19 – Example method implemented in RequestFactory class 71
Figure 20 – Example ResponseParserBase abstract class implementation 72
Figure 21 – How the script in *sap_status* uses the *sap_framework* 74
Figure 22 – JSON Configuration file to reschedule script 75

List of Tables

Table 1 – Statistics of the Painted Bodywork Quality IT department in Plant Dingolfing 29
Table 2 – Frequency which *RWS 00008* error occurs 48
Table 3 – Statistics of the `sap_framework` component 79
Table 4 – Statistics of the `sap_reschedule` component during the NAS migration in 2019 82

List of abbreviations and acronyms

SDK Software Development Kit

BMW Bayeriche Motoren Werke

IT Information Technology

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

FTP File Transfer Protocol

API Application Programming Interface

HTML Hypertext Markup Language

XML Extensible Markup Language

JSON JavaScript Object Notation

TDD Test Driven Development

XP Extreme Programming

FIRST Fast, Independent, Repeatable, Self-Validating and Timely

SAP BO SAP BusinessObjects Business Intelligence Suite

PDF Portable Document Format

CSV Comma-Separated Values

OOP Object-Oriented Programming

SOLID SRP, OCP, LSP, ISP, ISP and DIP

SRP Single Responsibility Principle

OCP Open-Closed Principle

LSP Liskov Substitution Principle

ISP Interface Segregation Principle

DIP Dependency Inversion Principle

SQL Structured Query Language

NAS Network Attached Storage

KPI Key Performance Indicator

URL Uniform Resource Locator

IP Internet Protocol

IDE Integrated Development Environment

Contents

1 INTRODUCTION . . . **21**
1.1 Objectives . . . **23**
1.1.1 Main Objective . . . 23
1.1.2 Specific Objectives . . . 23
1.2 Structure of the document . . . **24**

2 MOTIVATION AND PROBLEM DESCRIPTION . . . **25**
2.1 The Company . . . **25**
2.2 Quality Information Management in BMW Group . . . **25**
2.3 SAP BO for Quality Control . . . **26**
2.4 Key Performance Indicators and The Cockpit . . . **30**

3 THEORETICAL BACKGROUND . . . **33**
3.1 Object-Oriented Programming . . . **33**
3.2 The SOLID Design Principles for OOP . . . **34**
3.3 Software Components . . . **35**
3.3.1 Principles for Component Cohesion . . . 35
3.3.2 The Main Component . . . 36
3.4 Software Tests . . . **37**
3.4.1 Test Driven Development . . . 37
3.5 RESTful API . . . **40**

4 REQUIREMENTS AND DESIGN CONSIDERATIONS . . . **43**
4.1 General Requirements . . . **43**
4.1.1 Functional Requirements . . . 43
4.1.2 Nonfunctional Requirements . . . 44
4.2 Architecture . . . **45**
4.3 The *sap_framework* component and its modules . . . **46**
4.3.1 The restful module requirements . . . 46
4.3.2 The false unauthorized problem . . . 47
4.3.3 The false *logoff* problem . . . 48
4.3.4 The session expiring problem . . . 49
4.3.5 The first model restful module . . . 50
4.3.6 The problems and solutions of this first module . . . 51
4.3.7 Entities Module . . . 53
4.3.8 The *date_converter* Module . . . 54

4.3.9 Conclusion . . . 54
4.4 Use Cases . . . 55
4.4.1 The *sap_status* component . . . 55
4.4.2 The *sap_reschedule* component . . . 56
4.4.3 The *sap_backup* component . . . 56

5 IMPLEMENTATION . . . 59
5.1 Scrum Methodology for Project Organization . . . 59
5.2 Git Version Control System . . . 60
5.2.1 Workflow . . . 61
5.2.2 Git Submodules for Component Source Management . . . 62
5.3 Python 3 . . . 63
5.3.1 Standard Libraries Limitation . . . 64
5.3.2 Components and Modules . . . 64
5.4 Test Driven Development . . . 64
5.5 The *sap_framework* Component . . . 66
5.5.1 Data Structures . . . 66
5.5.2 Interface Classes . . . 67
5.5.3 The *Main Component* of *sap_framework* . . . 67
5.5.4 Information Flow . . . 68
5.5.5 Extensibility . . . 72
5.6 The *sap_status* Component . . . 73
5.7 The *sap_reschedule* Component . . . 74
5.8 The *sap_backup* Component . . . 78

6 RESULTS . . . 79
6.1 The `sap_framework` component . . . 79
6.2 The `sap_status` component . . . 80
6.3 The `sap_reschedule` component . . . 81
6.4 The `sap_backup` component . . . 82

7 CONCLUSIONS AND PERSPECTIVES . . . 85

REFERENCES . . . 87

1 Introduction

The BMW Group takes quality control very seriously. Being an automotive manufacturer focused on the premium sector, it would be impossible not to. Every car in each of the group's brands, BMW, Mini and Rolls Royce, is manufactured using high precision processes in each step of the production line to guarantee the best quality product.

However every manufacturing process may introduce defects, therefore the monitoring systems should be able to capture when a defect happens so that the parts affected can be replaced or fixed. The quality control processes use these monitoring systems in the production line to find the defects and decide how they should be handled.

To optimize the production processes, minimize the defects, rework times and production costs while maximizing the overall quality it is necessary to understand what caused the introduction of which defect. These analyses and decisions are done using quality reports generated by Business Intelligence tools.

This is the task of the Quality IT team, which uses the data generated by the monitoring systems with data analysis tools to develop quality reports that aim to map the status of the production quality as well as find processes that can be optimized even further.

The data analysis tool most widely used in BMW Group is the SAP BusinessObjects Business Intelligence Suite (SAP BO), which is used to process massive amounts of data from eleven different BMW Group Plants and automatically generate tens of thousands of quality reports every day.

Although the thirteen SAP BO servers in BMW Group have been constantly upgraded to handle all the processing power this application needs when used in this scale, some of the functionality of the suite seems not to be designed to work with such a large scale of data analyses.

The Quality IT teams develop the report templates in SAP BO and automate the report generation using schedules. Each report schedule has customizable filters such as the car model, which allow many variations of the same report to be scheduled using the same report template. Thus each Quality IT team in each plant is responsible to manage hundreds, sometimes thousands of report schedules.

Because SAP BO does not provide a tool in their suite that allows changing report schedules in batch and it takes roughly 7 minutes for a team member to manually change each report schedule, required changes to all schedules can lead to weeks of manual labor to the Quality IT teams until the quality reports can be generated again, and this

is not acceptable. Such changes are required in situations like implementations of new password policies (which for the technical users of SAP BO should happen once a year) and migration of file servers.

Another problem faced in BMW Group is related to the stability of the SAP BO platform. Since so many databases and systems are integrated into it there are many things that can go wrong and make a scheduled report fail to be generated. In such situations the Quality IT team must detect, diagnose and fix the problems as soon as possible to make sure the affected quality reports will be available when the managers and production supervisors need them. The functionality of the monitoring tools provided with the SAP BO suite is limited though, which makes the monitoring and diagnosing processes inefficient.

To fulfill some of the needs of the company, the Quality IT teams started to develop some tools that extend the functionality of SAP BO. SAP provides some ways to extend their product using language specific Software Development Kits (SDKs), however lately their focus is increasingly moving towards the RESTful Web Services SDK.

In 2018 the Quality IT team got access to the RESTful Web Services SDK and started to experiment with it and develop the first scripts. After a relatively short period the results were analyzed and the team reached two conclusions: There was a big potential for tools that access the SAP BO platform through this Application Programming Interface (API) to cause a huge positive impact on the way the company uses this platform, and the lack of documentation and support for this type of project within the custom installation of SAP BO in the BMW Group servers were a big challenge that needed to be overcome.

The work that resulted in this undergraduate thesis was developed throughout twelve months of internship in the Paintshop Quality IT department of BMW Group Plant Dingolfing. During this period I have contacted people from over 13 Quality IT Departments in BMW Group, from 8 different BMW, Mini and Rolls Royce plants in 5 countries. These people helped defining requirements for a system that could be used in all BMW Group Plants, testing the tools that I developed and giving feedback, which was essential to achieve good results with this project.

The tools developed within this project automated high demanding tasks to manage the SAP BO Platforms in BMW Group which saved more than 100 thousand euros in working hours throughout 2019 and have the potential to increase this impact in the years to come.

1.1 Objectives

The aim of this project is to build the foundations on which tools that use the SAP BO RESTful Web Services SDK can be built upon.

1.1.1 Main Objective

Develop a framework that can handle session management and all different types of requests needed from the SAP BO RESTful Web Services SDK. Provide solutions in this framework to the most common problems found in directly using the SAP BO RESTful Web Services SDK in BMW Group custom installations, to ensure stability to the applications that are developed using this framework.

Develop a set of script applications (tools) that implement functionality needed by BMW Group Quality IT teams using the developed framework, to manage the automation of quality report generation (report schedules) and monitor the SAP BO platform stability.

1.1.2 Specific Objectives

- Design the architecture of the framework in a way that will allow future extension and maintenance with least effort.
- Develop the session management functionality in the framework considering the user authentication requirements and the sessions issues in BMW Group's custom installations of SAP BO.
- Provide in the framework the necessary data structures for managing the instances of reports (report schedules and statuses) in the *SAP BO* platforms, such as folder, report template, report schedule, statuses, platforms and environments of BMW Group *SAP BO* installations.
- Develop functionality in the framework to deal with the many problems in the SAP BO RESTful Web Services SDK in BMW Group *SAP BO* installations and validate the responses for the implemented requests.
- Develop script application to monitor SAP BO platform status and stability using the newly developed framework. This script should be integrated with other tools developed by the team in this department to make the data available in web dashboard.
- Develop script application to change parameters of automatically generated quality reports (report schedules) in batch using the newly developed framework.

- Develop script application to backup and restore report schedules using the newly developed framework.

1.2 Structure of the document

This document is organized in a structure where the contextualization of the problem, the theoretical background necessary for the solution, the design, implementation and results of the solution are presented gradually.

Chapter 2 briefly presents BMW Group as a company as well as the Quality IT departments and how SAP BO is used to generate the quality reports. The problems faced to manage SAP BO are described and the motivation of this project is contextualized.

Chapter 3 covers the theoretical background needed to design and develop the proposed solutions, which will be referenced in the following chapters when they are used.

Chapter 4 exposes the requirements of this project and explains in detail the design process of the proposed solutions. The model of the framework component to handle the communication of the SAP BO Platforms in BMW Group through the Restful API is explained as well as some of the problems in this API which the framework should be able to handle.

Chapter 5 explains the methodologies used in the implementation of this project, exposes the technologies used to implement the design and matching the requirements from Chapter 4.

Chapter 6 exposes the results of the framework and the tools developed to use it and automate some of the management tasks that SAP BO lacks, as well as how much time and money this project has saved BMW Group.

Finally, Chapter 7 presents the conclusions and perspectives for future development of this project in BMW Group.

2 Motivation and Problem Description

This chapter will describe the company and departments concerned in the scope of this project, as well as the processes and problems that motivate it.

2.1 The Company

Bayeriche Motoren Werke AG, or BMW, is a company in the automotive industry that manufactures cars for the luxury market sector nowadays. It was founded in 1917 with the union of two manufacturers of internal combustion engines for airplanes. (BMW GROUP, 2019)

The name of the company in German reads "Engine Factory of Bavaria" and the emblem displayed in all products since its foundation has the colors of the Bavarian flag.

BMW started producing motorcycles in 1923 and automobiles in 1928. During the Second World War BMW was classified as an armament manufacturer due to the production of airplane parts and had the production plants completely dismantled with the end of the war in 1945. The first post-war motorcycle started being produced in 1948 and the first BMW car in 1952. The company struggled for some time until the launch of a successful model in the early 1960s.

In the mid-1960s BMW Plant Munich reached maximum production capacity and BMW purchased the company Hans Glas GmbH along with its facilities. BMW Plant Dingolfing was created in 1967 to be the largest BMW Plant for decades.

Up to the present-day BMW Plant Dingolfing is still the largest BMW Plant in Europe with 288 hectares. It is the only plant to produce the most premium models because of its reference in production quality and capacity, producing 1600 vehicles a day. (BMW GROUP, 2018)

2.2 Quality Information Management in BMW Group

Quality Control is a topic that is taken very seriously in BMW Group.

In all of the plants, there are many systems to measure and capture data from the cars during production. Teams of Quality Specialists design quality control reports based on this data to optimize production processes and provide an overview of the quality standards of the production.

The Quality Information Management in BMW Group Plants, which manages the

Information Technologies (ITs) that support quality control, goal management, occupational safety and environmental management, is responsibility of two different departments that are respectively concerned about two different steps of the production. The first one is the Painted Bodywork, which concerns all the process chain from the Press Shop, Body Shop and Paint Shop technologies. The second is the Assembly Shop, which concerns the assembly of all parts of the car to its body.

Every plant has a Quality Information Management Plant Coordinator for each Painted Body and Assembly, and for each of those technologies, there is one Technology Coordinator, both of which work in Plant Dingolfing. For this reason, Plant Dingolfing plays an important role in defining standards to be followed in all of the BMW Group Plants.

The department where this work was developed, Quality Information Management and Quality Control TG-300, is responsible for Quality Information Management for Painted Bodywork of Plant Dingolfing, as well as for the development of some tools that integrate and manage the information systems which will be explained in more detail in Section 2.4.

2.3 SAP BO for Quality Control

The tool most widely used to monitor quality control in the production lines in BMW Group is the SAP BusinessObjects Business Intelligence Suite, currently on version 4.2 SP5. This business intelligence tool is connected to most of the databases that store all kinds of quality control information and is used by Quality Specialists to generate reports about any type of defect that can happen in any part of the car. These reports are then used by many departments to monitor, diagnose problems and guide the optimization of the production processes.

The BMW Group Plants that use SAP BO for quality control on vehicle production are listed in Figure 1. They are divided in five different *Platforms*, and each of them is composed by three *environments*: development (DEV), integration (INT) and production (PROD). Every environment has around thirty servers in each of the two server *nodes*, for redundancy. For political reasons, the Chinese Plants do not exchange data with other BMW Group Plants, and because of this, they are not going to be supported by this project. The tools that I develop are going to be used in the German, United Kingdom, South Africa and United States SAP BO platforms of BMW Group. It is noteworthy that the Araquari Plant in Brazil and the San Luis Potosi Plant in Mexico do not have their own platforms and use the German and United States platforms respectively, as illustrated in Figure 1.

SAP BO provides the Web Intelligence tool, a web-based report creation tool that

Figure 1 – SAP BO Platforms in BMW Group

Source: (PEKELMAN, 2019).

is used by quality specialists of both Painted Bodywork and Assembly Shop technologies in all BMW Group plants to develop the quality reports. It provides many information analysis methods to access the data sets and find patterns, trends, means and deviations, that can be used with many different types of data visualization models, graphs and charts, according to the preferences of the analysts. This tool is used to create a *Report Template*, which can also have multiple parameters related to filters that are applied to the data for the specific analysis, such as the period, car series, type of process, rework stations, types of defects, company's cost centers, etc.

To generate a quality report the analyst must run the *Report Template* with the chosen parameters and choose the output file format and destination. The SAP BO server then reaches the databases and other back-end assets, processes the information, generates the report and sends it to the destination. The output formats most used include Portable Document Format (PDF), Microsoft Excel, Comma-Separated Values (CSV) or Hypertext Markup Language (HTML), and the destination is normally one of the company's file servers or e-mail.

Most of the reports created are recurring reports, which are scheduled to be automatically generated hourly, daily, weekly, monthly or following a specific calendar. For such reports, a *Schedule* is created in the same way a *report template* is put to run manually, however, the recurrence information is also needed. This process generates a special instance of the *Report Template* called *Recurring Instance*, which is a type of *Schedule*.

SAP BO gives the *children* of a *Report Template* the name of *instances*. The

relationship between instances of a *Report Template* is analogous to object instances of a class. The main property of an *Instance* in SAP BO is the *status*, which can assume the values (SAP, 2017a):

- Running
- Completed
- Recurring
- Failed
- Paused
- Pending
- Expired
- Warning

The *Recurring*, *Paused* and *Expired* statuses are related to *schedules* of the *Report Template*. *Recurring* means that it is enabled to automatically generate reports, *Paused* means that the *schedule* is temporarily disabled and *Expired* means that the end date of the schedule has been reached and therefore it was suspended.

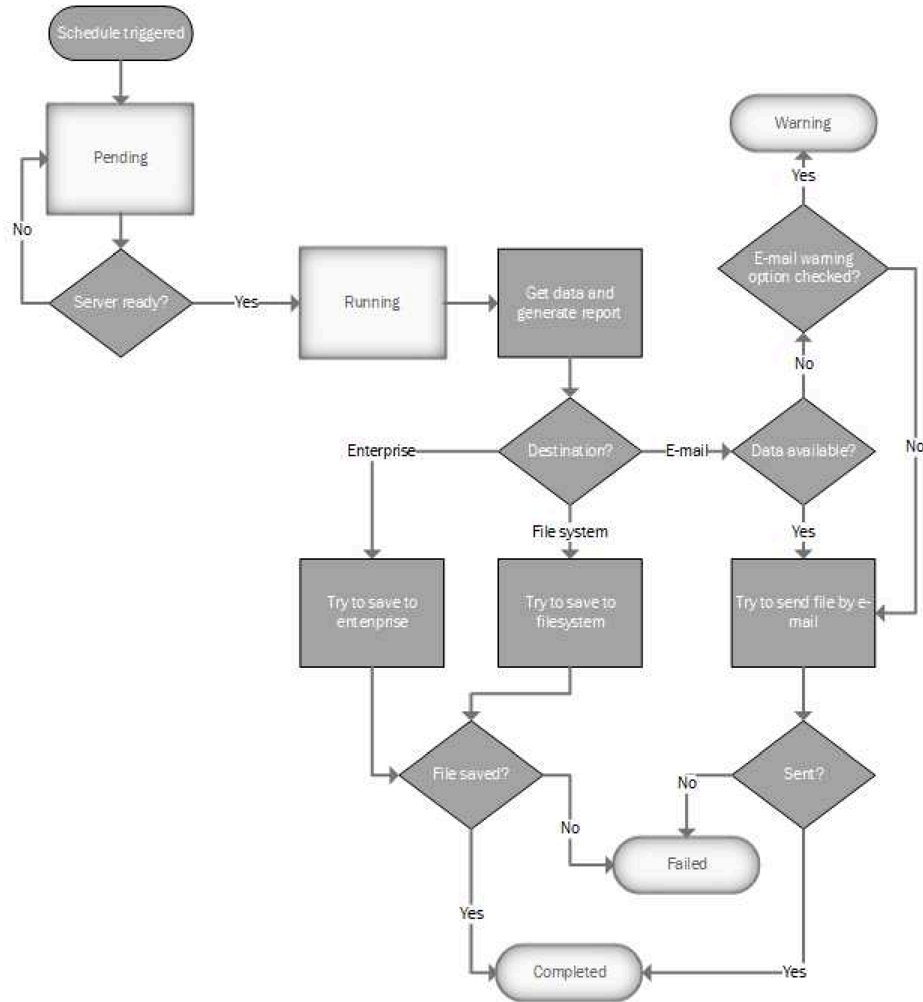
The remaining statuses are related to the status of a *Report Template* that was triggered to run either manually or via a *schedule*. The meaning of each status and the situations in which they can happen can be seen in Figure 2.

An instance which is a successfully generated quality report has the status *Completed*, however many situations can lead to other statuses which will not generate a report, thus the importance of verifying the status of the instances and make necessary changes in order to make them run again.

The flowchart in Figure 2 represents the possible statuses of a report instance in SAP BO and the situations which make the status change. Once a report is triggered to run, either by a schedule or by a platform user who manually triggers it, a new instance is created with the status *Pending*. The blocks lightly colored in the flowchart show the possible statuses that an instance can have in SAP BO. The statuses that are not present in the flowchart are *Recurring*, *Paused* and *Expired*, which correspond to statuses of a *Report Schedule*.

The SAP BO is a complete tool for data analysis and is the main tool being used by Quality IT departments in BMW Group today. There are 89 production databases connected to it that generate a massive amounts of data, most of which is already organized into *universes* and can be used to create quality reports and optimize the production.

Figure 2 – Instance statuses in SAP BO



Source: made by the author.

Table 1 has some statistics of the Quality IT department responsible for the Painted Bodywork technology in Plant Dingolfing to illustrate how much data the platform handles considering the same structure is used in 11 Plants.

Table 1 – Statistics of the Painted Bodywork Quality IT department in Plant Dingolfing

Information type	Amount
Defect Types	168
Defect Places	32
Cost Centers	375
Car Series	25
Cars produced daily	1600
Key Performance Indicators	1178
Report Schedules in SAP BO	1934

Source: made by the author.

2.4 Key Performance Indicators and The Cockpit

Most of the Report Schedules in the SAP BO Platform, besides the detailed report, also calculate a Key Performance Indicator (KPI) defined by a set of rules. Each KPI has an acceptable range of values which are the targets of the production managers.

Sometimes a manager is responsible for a large set of those KPIs and opening each report file on the Network Attached Storage (NAS) to find the calculated KPI becomes a high demanding task.

To provide a solution for this problem the Painted Bodywork Quality IT department (where this project was developed) develops and supports a project which is used by most of the Quality IT departments in BMW. The so called Cockpit is a web application that uses the Oracle Apex Framework to provide the users of the reporting systems a user friendly way to access the KPIs and reports, providing a dashboard overview of the production in the plant.

Each Quality IT department that uses the Cockpit has a tool running in a server called Wrapper Tool, also developed and maintained by our department, which scans through the folders that contain their reports in the NAS, reads the report files and finds the information related to the KPIs. This information is gathered and saved to the Cockpit's database hourly.

The Cockpit has general information pages, home pages with information of each Quality IT department (Figure 3), and the dashboard pages that show the KPIs of a specific category of report(Figure 4).

The KPI views show the current and last values of each KPI, colored green if the KPI value is within the acceptable range and red otherwise. Clicking on a KPI opens the report file from which the KPI was read by the Wrapper.

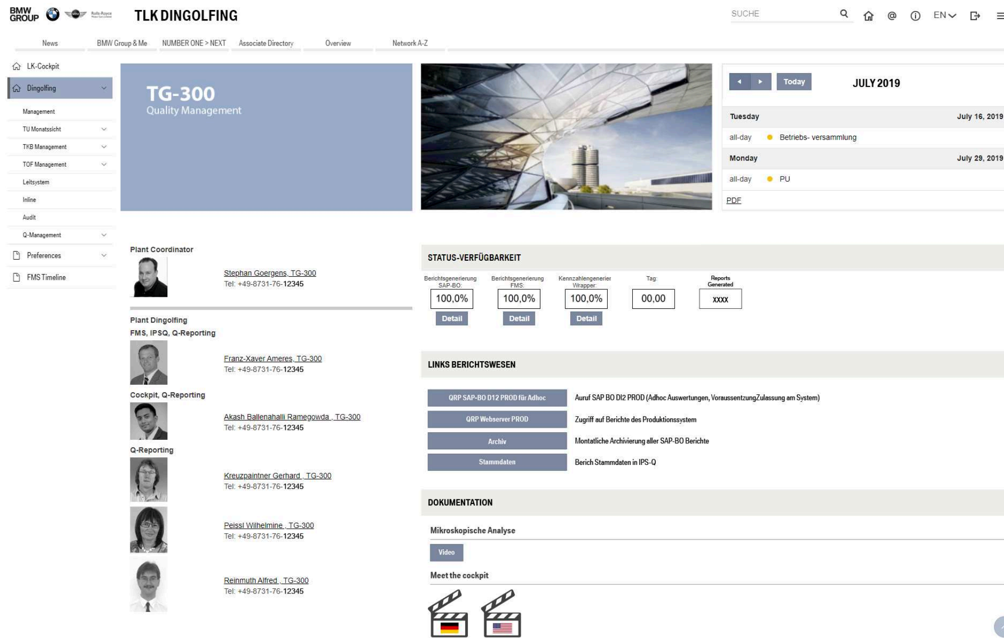
Both tools have many more functions that will not be covered here because are not relevant to the work developed in this project.

The scope of the project developed in this internship is enabling the development of plugin applications that extend the functionality of the SAP BO to better match the requirements of BMW Group just like the Wrapper Application and the Cockpit do. However these tools originally extend the functionality by reading the files in the NAS only, and if for some reason the SAP BO platform was not able to add a file there (see figure 2) they cannot give the user any information related to what happened inside *SAP BO*. This project is going to use the RESTful SDK of SAP BO to access relevant information inside the SAP BO Platform, as well as manage the *Schedules* of the *Report Templates* created in SAP BO.

One of the tools developed within this project is also integrated with these systems

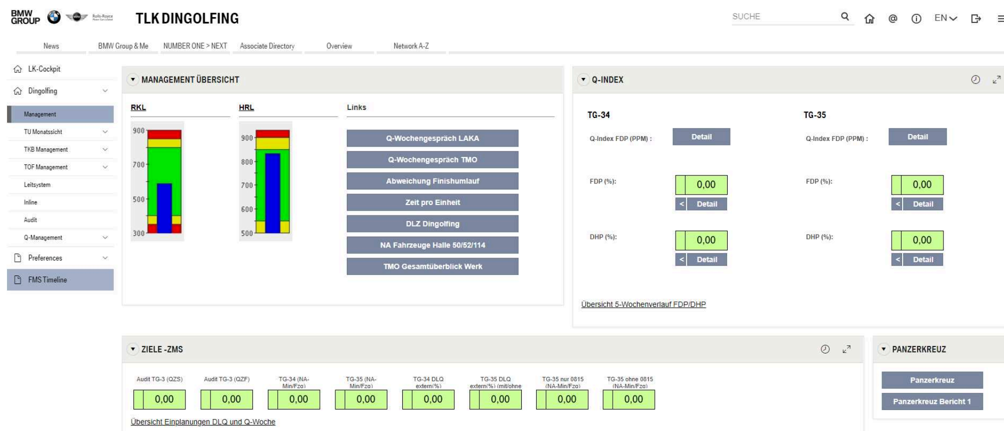
to deliver information on the page in Figure 3. This tool will be explained in detail in Sections 4.4.1 and 5.6, and the relationship with this image will be explained in Section 6.2.

Figure 3 – The Cockpit home page of TLK Dingolfing



Source: made by the author.

Figure 4 – KPI visualization in the Cockpit



Source: made by the author.

3 Theoretical Background

This chapter explains the theoretical backgrounds that guided the design processes, the development and analyses of the results of this project.

3.1 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of *objects*. *Objects* are usually models of things in the real world. They have data fields (called *attributes* or *properties*) and procedures called *methods*.

Objects are generalized into *Classes*, which define a type of object and allow the creation of multiple similar objects that reuse the same code. In other words, the classes define a the data formats and methods available in a type of object and the objects are the *instances* of a class.

However, OOP is more than just using classes and objects to organize code, it is a programming paradigm *based* on classes an objects that enables a set of features:

- Encapsulation
- Inheritance
- Polymorphism

As previously stated, programming languages that support OOP provide an easy and effective encapsulation of data an function into cohesive objects.

Inheritance is a way of redeclaration of a set of attributes and methods from a *base class* into a *subclass*, which *inherits* from it. This is useful to define *subtypes*. For example, both *Employee* and *Customer* are persons, so they have a set of information that is common, thus the classes that represent them could inherit from a *Person* class that define what is common to all types of persons.

The use of *subtypes* that have a different implementations overriding the same set of methods, as in the definition of an *interface*, allows the usage of this subtypes to be standardized. This is called *Polymorphism*.

As Robert C. Martin wrote on Clean Architecture: A Craftsman's Guide to Software Structure and Design (2017), these features are not entirely exclusive to OOP languages but they are definitely made easier and more convenient in these languages. The real advantage that OOP offers in a Software Architecture point-of-view, according to him,

is the full control of dependencies and the possibility to use dependency inversion. This aspect will be explained further in section 3.2.

3.2 The SOLID Design Principles for OOP

SOLID is a name created to help remembering the five principles of good OOP design. Each letter corresponds to one of the following acronyms:

- **SRP:** Single Responsibility Principle
- **OCP:** Open-Closed Principle
- **LSP:** Liskov Substitution Principle
- **ISP:** Interface Segregation Principle
- **DIP:** Dependency Inversion Principle

According to (MARTIN, 2017, PART III Design Principles) the target of this principles is to design modules that are easy to understand, tolerate change and serve as basis for components that can be used in many systems.

The Single Responsibility Principle (SRP) states that each module should have one single responsibility, or should be responsible to a single *actor*. If this principle is ignored, one function or class may be used by different use cases or actors. If one actor requests a functionality to be changed and this function or class needs to be changed for this purpose, it might affect the other use cases that rely on it, leading to unexpected behavior.

The Open-Closed Principle (OCP) states that a class, module or component should be open for extension but close for modification. This means that in order to extend the functionality of the module, there should be no need to make big changes on source code apart from adding the new functionality code. This is accomplished by partitioning source code and managing dependencies to protect higher-level modules from changes in lower-level modules.

The Liskov Substitution Principle (LSP) guides the use of Inheritance in OOP and is based on the definition of subtypes by Barbara Liskov (LISKOV, 1988). This principle states that inheritance should be avoided when the subclasses are not interchangeable like *real subtypes*, or in other words, do not implement the same interface thus allowing other classes to use the objects in the same way.

The Interface Segregation Principle (ISP) states that when a class A has a method that is only used by a class B, this method should be segregated into an interface to prevent class B from depending on the rest of class A's source code, forcing recompilation

and redeployment of class B whenever changes are made to class A. This principle is mostly used is statically typed languages such as Java or C#, because dynamically typed languages such as Ruby or Python have implied interfaces that are inferred at runtime, so they do not force redeployment (and are not compiled either). In such cases the important idea behind this principle is that a software artifact should not be forced to depend on things it does not need.

The Dependency Inversion Principle (DIP) states that source code dependencies should refer to abstractions instead of volatile concrete implementations that are susceptible to frequent changes. This means that the lower-level modules that contain the implementation details should depend on the higher-level modules, despite the flow of control being in the opposite direction, which is the reason the principle is called Dependency *Inversion*.

3.3 Software Components

Components are a set of modules that are deployed together. In compiled languages they are an aggregation of binary files, while in interpreted languages they are aggregations of source code files. Well designed components contain a cohesive set of classes and well defined boundaries with dependency management that ensure they could be independently *deployable*, therefore being independently *developable* as well. (MARTIN, 2017, Chapter 12)

3.3.1 Principles for Component Cohesion

Selecting which modules should be part of which component is a challenging and important part of the application's architecture. Well-designed components should follow the three principles of component cohesion: (MARTIN, 2017, Chapter 13)

- The Reuse/Release Equivalence Principle
- The Common Closure Principle
- The Common Reuse Principle

The Reuse/Release Equivalence Principle states that *"The granule of reuse is the granule of release."*. It means that classes and modules that are formed into a component need to be a cohesive group. It also means that components should have a versioning system with *release numbers* that will be referenced in other components for compatibility purposes. The release process of a component should also include documentation of changes in order to allow the developers of other components to make the necessary changes for

interfaces with the new component or even ignore the new release and use the old version instead.

The Common Closure Principle is the component version of the SRP (section 3.2). It states that the classes and modules that change for the same reasons should be grouped in the same component, and those which change for different reasons should not. This is also important for maintainability of the application:

> For most applications, maintainability is more important than reusability. If the code in an application must change, you would rather that all of the changes occur in one component, rather than being distributed across many components.1 If changes are confined to a single component, then we need to redeploy only the one changed component. Other components that don't depend on the changed component do not need to be revalidated or redeployed. (MARTIN, 2017)

The Common Reuse Principle states that classes that are independent should not be part of the same component, thus not enforcing other components that depend on it to depend on classes within it that they would not need. It is the component of the ISP (section 3.2).

While the first two component principles define which classes and modules should be grouped in a component, the third defines which ones should not. It is not possible to completely achieve all three principles in all components of an application, therefore the architecture should focus on achieving a balance between the effort to reuse the component, frequency of releases, and number of components that will require change in each release, in order to fulfill the current development concerns.

3.3.2 The Main Component

When the principles in Sections 3.2 and 3.3.1 are used to implement a clean architecture and the dependency management is properly dealt with, higher-level components such as those which implement the *Business Rules* or *Use Cases* should not be affected by implementation details of lower-level components.

In practice, this means that the names of the lower-level classes should not be mentioned in the higher-level ones that use them, although this information needs to be available somehow. To solve this, *dependency injection* frameworks should be used or a special kind of component should be developed that links everything, the *Main* component.

This component is responsible for instantiating the lower-level classes and giving these objects to the higher-level modules. It injects the dependencies and builds component structures of the application, thus is considered to be the *dirty* component of the *clean architecture*. (MARTIN, 2017, Chapter 26)

3.4 Software Tests

Quality is determined by how a product matches the expectations set, and in case of software, by the software requirements. In other words, in order to assure quality in software production there need to be software tests that show if and how the software product matches its requirements.

This tests can be done in different levels, manually or automatically. The three levels of software tests are:

- Unit Tests
- Integration Tests
- Acceptance Tests

Unit Tests are done in the lowest level, covering the classes's methods, while Integration Tests cover the component level and component relations and Acceptance Tests cover the application level and use cases.

Most of the languages today have a unit testing framework (or sometimes more then one available) to write automated unit tests to the software being programmed.

3.4.1 Test Driven Development

Test Driven Development (TDD) is a software development technique that became popular in the late 1990s as part of the Extreme Programming (XP) and was later adopted by Scrum and other Agile methods. The so called "Test First Programming" has three laws:

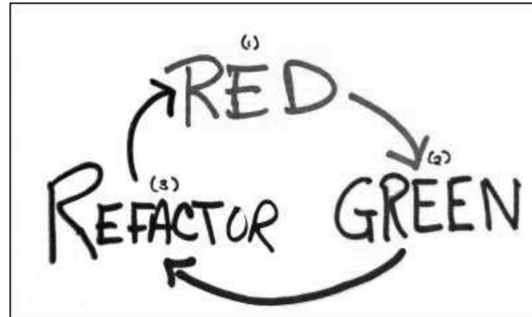
1. You are not allowed to write any production code until you have first written a failing unit test.
2. You are not allowed to write more of a unit test than is sufficient to fail.
3. You are not allowed to write more production code that is sufficient to pass the currently failing unit test.

This three laws define a cycle that should be followed during the software development. The cycle is perhaps less than a minute long and ensures that tests and production code grow together.

The cycle starts with the first lines of the unit test. At some point a name of a class or method that was not yet declared will be mentioned and will make the test fail.

This means that it is time to write the production code that declares this resource, but not more, so that the unit test now passes. Then time to write more test code again.

Figure 5 – The TDD Cycle



Source: The Cycles of TDD (2014)

A visual representation of the TDD cycle can be found in figure 5. The first status is represented by the word "red" and it refers to the step when the developer writes a unit test that fails. The second status is represented by the word "green" and it refers to the step when the developer writes production code that makes the failing unit test to pass. Finally, the third status is represented by the word "refactor" and it refers to the step when the developer cleans the production code just written to improve readability or resource management. (MARTIN, 2014)

According to Robert C. Martin (MARTIN, 2011, Chapter 5) the benefits of using TDD are:

- Certainty
- Defect injection Rate
- Courage
- Documentation
- Design

Dozens of new tests are written for each new module added to the application, that can and will be run every time a change is made to production code. Whenever a change is made in any line of any file, the unit tests will show with **Certainty** if the methods that worked before still work. If however something broke, they are going to show where so that it can be fixed right away, reducing the **Defect injection Rate**.

Developers that work in big projects sometimes encounter parts of code that are not organized, hard to understand and hard to modify, and think that it should be fixed. The problem is that this kinds of fixes have the potential to break other things, and the developer who tried to fix the mess would be responsible for it. TDD makes it simpler to

change pieces of code and verify that everything still works as expected, so it gives the developers the **Courage** to change what should be changed to improve the quality of the product.

Code examples are the parts of the documentation of any library or framework that is most appreciated by developers. The unit tests created with TDD have code samples covering all the different ways each class was designed and developed, and that can definitely be used as **documentation** by the developers in the team.

The difficulty of writing unit tests is that for a method to be tested correctly it needs to be isolated from others. Writing the tests first forces the developer to consider this things during development, which are characteristics of good **design** practices.

Although the unit tests are not part of the production code, the design and organization in their code must be made with the same care and hold the same quality standards that production code. Messy tests can indeed lead to more overhead when requirements change or make production code that was designed to be volatile harder to change because of their intrinsic dependency relationship. Tests always depend on production code. (MARTIN, 2017, Chapter 28)

Well written tests should be Fast, Independent, Repeatable, Self-Validating and Timely, which form the acronym FIRST.

Fast: Tests should be fast to execute. If they are slow, developers will not run them as frequently, leading to more problems not being found in an early stage and therefore harder to fix.

Independent: Tests should not depend on each other, they should be able to run separately and on any order. If one test sets up the conditions for others, that leads to a cascade of failures making the debugging process difficult.

Repeatable: Tests should be repeatable in any environment. They should not depend on databases or internet connection or computer architecture. If the tests are not repeatable in every environment, they are not able to run when the environment is not available and there will always be doubts whether tests failed because of a problem in the code or in the environment used to run them.

Self-Validating: Tests should have a boolean output. They can only be successful or failed, and the evaluation should not depend on subjective analysis of logs or output files.

Timely: Tests should be written in a timely fashion, seconds before the production code they test. If they are written after the production code, it may be too hard to test because the production code was not designed to be testable. (MARTIN, 2009, Chapter 9)

3.5 RESTful API

Most of the software applications today are not stand alone and have features that need a centralized database or some kind of integration with other agents over a network.

According to Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (2004), there are six different patterns of integration architecture, which are:

- Information Portals
- Data Replication
- Shared Business Functions
- Distributed Business Processes
- Business-to-Business Direct Integration
- Service-Oriented Architecture

The Service-Oriented Architecture is the most advanced pattern used nowadays, and its goal is to take advantage from new and old functionality in order to widely reuse the business rules. This functionalities are developed in a decoupled way from the applications and are accessed remotely as if they were local methods. This methods are generally called *web services*. (HOHPE et al., 2004)

A web server is a computer or a piece of software running on a computer that provides other computers access to files or databases (MOZILLA DEVELOPER NETWORK, 2019b). A web server can be either static or dynamic. A static web server receives a request for access to a file and sends the file "as is" and sends it to the client. This requests usually use a protocol like Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). A dynamic web server, on the other hand, has some extra software to process data and is commonly divided into an application server and a database. In this configuration, the application server receives the HTTP request, processes the request, accesses the database to read or write information, and sends a response to the client.

The HTTP is a stateless application-layer protocol that follows a classic client-server model. It defines how the messages between client and server should be structured, with predefined Uniform Resource Locator (URL), headers, the message body and the HTTP Method, most commonly GET or POST (MOZILLA DEVELOPER NETWORK, 2019a). HTTP is the most used protocol on the internet, and it is used for example to define how the internet browser in a computer communicates with web sites' servers.

A software application that is built to be accessed by other applications should implement the interface that allows this interaction. In software development, this is called API.

One of the standard architectural styles for distributed hypermedia systems is the Representational State Transfer (REST) pattern (RESTFULAPI.NET, 2019), which is an implementation of the Remote Procedure Invoking integration approach. REST defines 6 guiding constraints that must be satisfied by an API in order to be called RESTful, which are:

- Client–server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand (optional)

A RESTful API is implemented through the HTTP protocol to give client applications access to the server's resources. REST uses a resource identifier to identify the particular resource requested. The state of a resource is called resource representation and its data format is some kind of hypermedia, usually HTML, Extensible Markup Language (XML) or JavaScript Object Notation (JSON).

HTTP Response Status Codes are also used to identify whether the request was successful, if the response has a body, what kind of error might have occurred and if the error was on the client-side or server-side (FIELDING et al., 1999).

4 Requirements and Design Considerations

The four SAP BusinessObjects Platforms in BMW GROUP and their dozen environments in total, constitute a massive infrastructure that is crucial for the Quality Reporting in the entire company, as discussed in Chapter 2. In order to keep this infrastructure running healthily, there are some management tasks required. Some of these crucial tasks require a huge collective effort from the Quality Reporting teams in all BMW GROUP Plants due to the extensive scale.

Although developers have created automated solutions for some tasks in the past using the RESTful API, those solutions were usually to a specific use case which solved a specific problem of one of the departments described in section 2.2. Other technologies and other Plants with similar problems would not be able to use the same script because the changes in requirements that were necessary were too hard to be implemented in a non-modular project that was not designed to allow changes.

The main goal of this project is to provide a software framework that enables developers to make automated solutions for this platform management tasks, in form of scripts, services or applications, without being concerned with the lower-level implementation details related to the use of the different APIs available. This chapter is going to explain the design process that enabled this goal to be achieved.

4.1 General Requirements

This project will contain the SAP BO platform management framework and script solutions that use this framework to perform platform management tasks.

The definition of a framework helps us to understand some of the requirements of this project:

> A framework, or software framework, is a platform for developing software applications. It provides a foundation on which software developers can build programs for a specific platform. For example, a framework may include predefined classes and functions that can be used to process input, manage hardware devices, and interact with system software. This streamlines the development process since programmers don't need to reinvent the wheel each time they develop a new application. (CHRISTENSSON, 2013)

4.1.1 Functional Requirements

1. Connect to any environment of any BMW SAP Platform

2. Manage sessions for different users

3. Detect expired session and reconnect
4. Have persistence of sessions for manual logoff in case of crashes
5. Retrieve data with different APIs available in the platform
6. Parse the responses and return the relevant information
7. Provide necessary data structures to represent the requested data

The first requirement is to connect to any SAP BO environment in BMW because the tools that are going to be developed using this framework should be able to access Development, Integration and Production environments within the four Platforms in different countries, as explained in Section 2.3.

Requirements 3 to 5 are related to problems the RESTful API of SAP BO has which the framework should be able to handle. These problems will be further explained in Sections 4.3.2, 4.3.3 and 4.3.4.

Finally, requirements 6 and 7 are related to converting the text based response of the RESTful API into data structure objects which can be read by the tools. This is necessary to ensure not only that the tools will be able to access the information they need, but also that the integrity of the responses is validated within the framework. More information in Section 4.3.

4.1.2 Nonfunctional Requirements

1. Version control
2. Independently developable and deployable components
3. Testability
4. Documentation
5. Extensibility
6. Internationalization and Localization
7. Maintainability
8. Portability
9. Readability

The first two nonfunctional requirements are related to the versioning of the framework and the tools that use it. The tools and the framework component need to be independently developable so that changes made in the framework component to fulfill new requirements of one of the tools do not enforce changes in the other tools components to keep them compatible. With version control, each tool component can use a different *Release* of the framework component, according to the Common Reuse/Release Principle of Component Cohesion explained in Section 3.3.1. For more details see Sections 4.2 and 5.2.2.

The requirements 3, 4, 5, 7 and 9 are related to the use of good practices of Software Engineering that will enable other developers to maintain the tools that I create and develop new ones after the period of my internship is over. These requirements were defined with the Quality IT team that would be responsible for this project in the future, since they explained that it is normally hard to continue the work of a former intern and usually the new interns have to start new projects from the beginning.

Requirement 6 about *Internationalization and Localization* refers to SAP BO in BMW Group being used in Platforms in multiple countries, with different languages, which affect the operation of these Platforms through the RESTful API, mostly but not limited to, in the formats of date strings. This problem is detailed in Section 4.3.8.

Finally, requirement 8 about *Portability* refers to the problem that in a big company such as BMW Group there are many IT rules and the employees usually do not have administrator rights in all computers and servers that are going to use the tools developed in this project, which means that no additional installations should be required.

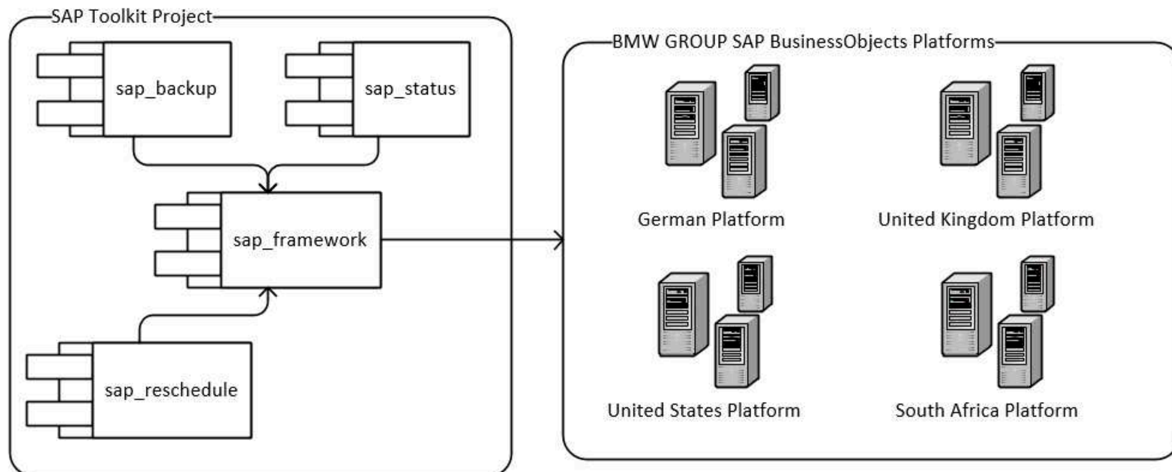
4.2 Architecture

The software architecture is essential in the development of this project since it will contain many tools that, with good software architecture techniques, will use the same modules to implement the commonly needed resources.

These common resources will be part of a component called *sap_framework*, as illustrated in Figure 6. The other components that use it represent the tools developed. These tools will implement the main applications, Use Cases, GUIs, inputs and outputs, while the *sap_framework* component will implement the *entities* (main data structures), the module that handles the RESTful requests, the necessary converters and other common pieces of software.

These components, specially the *sap_framework* which is depended on by the others, will have a release process following the Reuse/Release Equivalence Principle discussed in section 3.3.1.

Figure 6 – Component Diagram of SAP Toolkit Project



Source: made by the author.

Figure 6 shows some of the software components that will be part of this project, although the project itself is not limited to them. The architecture should allow new tools to be added at any given time. The block on the right called *BMW Group SAP BusinessObjects Platforms* correspond to the platforms of SAP BO in BMW Group as explained in section 2.3. The block on the left called *SAP Toolkit Project* shows the project developed by me during this internship, with its software components.

The *sap_framework* component may also be updated to support new features, thus requiring that the components are source code independent and use a version control system that allow components to use different versions of the framework, enabling independent development.

This means that each of the *tool* components should use a *release* of the *sap_framework*, and every time there is a new release the developers of the tools can have the option to upgrade the tool to use the new release of the framework, and plan the next release of the tool accordingly. The implementation of this process will be described in section 5.2.2.

4.3 The *sap_framework* component and its modules

4.3.1 The restful module requirements

This is going to be the module that handles the communication with the SAP BO environment through the RESTful API, and it should hide the lower-level communication details of the API and the HTTP Requests that it uses.

To design this module it is necessary to understand how the APIs work using both the documentation provided by SAP and the situations that are not documented and had

to be reverse-engineered in our specific installation of SAP BO.

As explained in section 3.5, to communicate with a RESTful API, a module needs to be able to create HTTP requests, send the requests, receive the responses, decode and parse the responses, and give the requested information back to the module that called it.

In the SAP BO platform specifically, the first request should be a *logon* request that sends in the *body* the *username* and *password* of the user, as well as the authentication method. The *body* of the response for this request will contain a *logonToken* string which will be used in the remaining requests as part of the *headers*.

There are at least three APIs in the SAP platform that use this same kind of interface *Raylight*, *Infostore* and *CMSQuery*. Basically the first is the main API that SAP provides to connect tools to their platform, the second gives access to some limited data on Infostore objects such as folders, documents and instances (see section 2.3), and the third enables the usage of Structured Query Language (SQL) queries to the platform's databases.

After the requests are over, a *logoff* request with the *logonToken* must be sent to the platform in order to release the session currently being used.

Each time this *logon* request is processed by the SAP BO platform it generates a *session* for this user. Each user has a maximum of ten sessions available independently of the client type used (API or other applications from the SAP BO Suite), which means that whenever the maximum number of sessions for a user is reached, this user is blocked until the open sessions are closed. According to the SAP BO documentation the opens sessions should expire and be deleted after one hour, which is not always the case in the BMW Group installation of SAP BO as my tests monitoring the open sessions using an administrator account have showed.

Because of this problem with the management of the sessions it is necessary that the *restful* module manages the sessions it creates, making sure that every session opened by it should be reused as many times as necessary and then properly closed afterwards. The session management in this module should also detect when a session has been expired and reconnect this user.

4.3.2 The false unauthorized problem

One of the most common problems that happens with this APIs is that most of the requests are not responded properly. Most of the requests through any of the APIs mentioned get a response with HTTP Status 401 (Unauthorized) and message *"No logon token provided in the X-SAP LogonToken HTTP header or query parameter. (RWS 00008)"*, even though there was a logon token in the *headers* with this exact tag. In this cases we send the same request again until the response does not have this error anymore. This

happens randomly for every kind of request tested. See Table 2 to see how often this error occurs in some tests I have made on October 10th in the Integration environment of the German Platform of SAP BO.

Table 2 – Frequency which *RWS 00008* error occurs

Number of requests made	Number of responses with RWS 00008 error	Average errors per request
40	38	0.95
40	39	0.975
40	37	0.925
40	37	0.925
40	1	0.025
40	40	1
	TOTAL AVERAGE	0.8

Source: made by the author.

The workaround used to this problem is to ignore a response like this and send the same request again until the client-side receives a response considered to be valid.

4.3.3 The false *logoff* problem

Although the API documentation states that to release a session a *logoff* request should be sent and answered with HTTP Status 200 (OK) (SAP, 2017b), that response does not always mean that the session was indeed closed.

To test this, an account with full administrator rights was used in a controlled environment of SAP BO, with which it was possible to monitor the open sessions of a specific user. Sessions were opened through the RESTful API and it was observed that after a *logoff* request through the API the session would still be blocked in SAP BO, even though the response through the API was HTTP Status 200 (OK).

This problem was a known issue which caused users to be blocked by having ten sessions open indefinitely, until someone in the SAP BO Operation Team, from a supplier company, would manually clear the sessions for that user. SAP has provided us with a private fix for this problem which would supposedly make inactive sessions to be automatically cleared after one hour.

The tests described earlier showed, however, that this fix does not always work properly, and still sometimes leaves sessions open indefinitely. This problem was reported to the SAP BO Operation Team with test results and a solution was never found.

The workaround that we use to ensure that the session was properly closed was discovered by a former intern in the department (SILVA PRAZERES, 2018). He realized that after the *logoff* request was properly processed, following requests using the same token are responded to with HTTP Status 401 (Unauthorized) and message *"Not a valid*

logon token. (FWB 00003)" and proposed the method of sending the *logoff* request multiple times until this response is received.

My tests show that, although the converse in not true and this response does not always mean that the session is disconnected, this method can be used to check whether a *logoff* attempt was successful.

4.3.4 The session expiring problem

Another problem in the SAP BO RESTful API that was not documented is related to the expiration of sessions.

After some time a session token can be disconnected by the server-side, which would then require the client to send a new *login* request and get a new token. The SAP BO Operations Team tried to support this project and conducted an investigation to determine the reason of sessions being closed by the server-side but it was inconclusive.

The problem is that when this happens there is no clear message to the client-side stating that the token should be renewed. There are two error messages related the token, both of which have their own problems explained in Sections 4.3.2 and 4.3.3.

By reading this description of the two error messages, one would assume that when a session is closed in the server-side, the response sent would have HTTP Status 401 (Unauthorized) and message *"Not a valid logon token. (FWB 00003)"*, as explained in section 4.3.3, but this is not true.

Most of the times a session is closed by the server-side, the following requests containing the same token are actually responded with HTTP Status 401 (Unauthorized) and message *"No logon token provided in the X-SAP LogonToken HTTP header or query parameter. (RWS 00008)"*, which, because of the problem explained in Section 4.3.2, is an ambiguous message that usually has no real meaning.

This could be tested using an account with full administrator rights in a controlled environment of SAP BO and manually closing, on the server-side, the session being used on the RESTful API on the client-side.

A consequence of this problem is that the workaround suggested for the problem in section 4.3.2 is not always valid. Repeating requests that are responded with this message will *usually* eventually result in a valid response, but if this process is repeated too many times and no valid response is detected, it *probably* means that the session disconnected and the user needs a new token.

4.3.5 The first model restful module

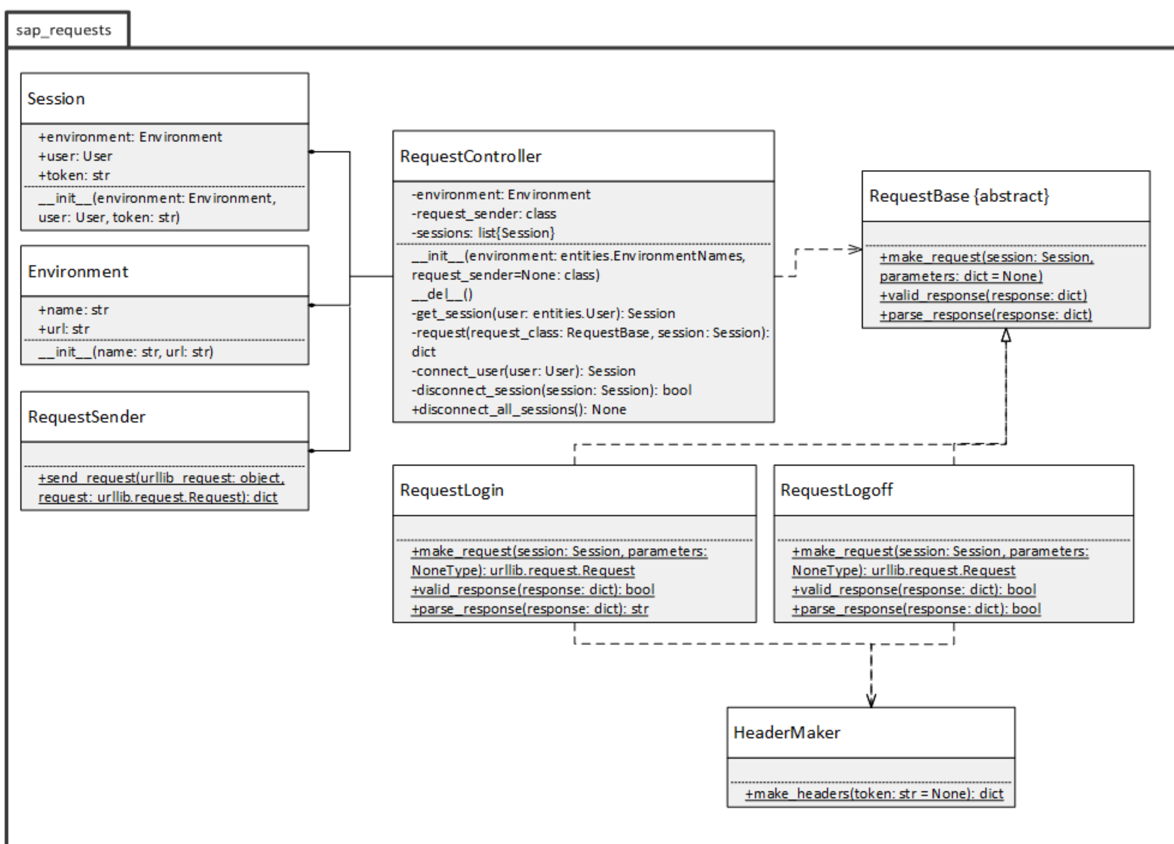
Based on this few requirements I planned a first version of the module's structure as can be seen in the class diagram on Figure 7.

The main class here is the *RequestController*, which is composed by one *Environment* object and a list of *Session* objects. It has methods to execute the requests managing the sessions automatically, creating a session for a user if it is not available and disconnecting all sessions at the end (upon deletion).

Each different request method of the *RequestController* class uses the *_request* private method with a different implementation of the *RequestBase* class to make the *Request* object, verify if the response received is valid and parse the response dictionary finding the information that should be returned to the higher-level module. This *_request* method uses the *RequestSender* class and retries the request some times until the implementation of *RequestBase.valid_response* method returns true. This is meant to avoid the problem with the *RWS 00008* error described in section 4.3.2.

In Figure 7 there are two of these *RequestBase* implementations, *RequestLogin* and *RequestLogoff*, and the idea was that more implementations of *RequestBase* could be added for each type of request that the framework would support.

Figure 7 – First attempt to model the restful module



Source: made by the author.

4.3.6 The problems and solutions of this first module

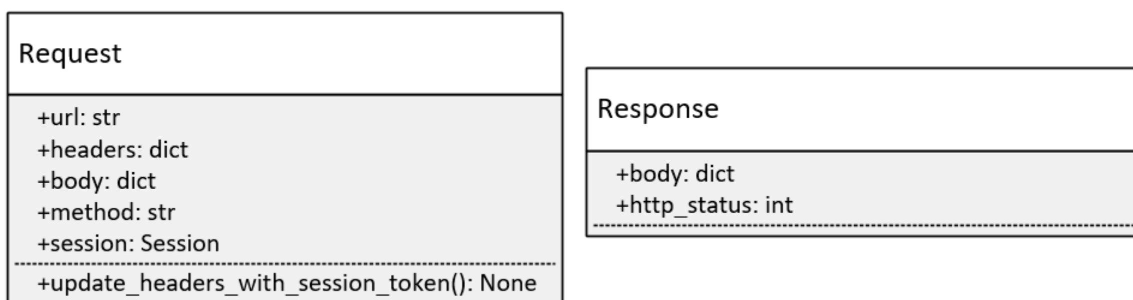
This first model has some advantages but many problems as well. This section will discuss the problems from a software architecture point of view applying the principles from section 3.2.

The first problem is that although the *RequestSender* class is the only one that actually use an external library to handle the communication with the platform, the *Request* object used by many of the classes in this structure is part of the same library, Python's *urllib*, that handles HTTP requests. If in the future someone decides to reuse this structure with a different library such as *urllib2*, *urllib3* or *requests*, many of the classes will have to be changed, which would require a huge effort.

The second problem is that instead of defining a *Response* class, the response is only a dictionary that contains the body of the response. This is a problem because some of the errors that may happen in the platform such as HTTP Error *503 Service Unavailable* have no body, so the response would be an empty dictionary with no information of which problem happened.

To address this first two problems I modeled two classes: *Request* and *Response*, illustrated in Figure 8. The first has the basic HTTP request properties discussed earlier as attributes along with the *Session* object used to make the request, which is relevant because if the module detects that a session has expired during a request, it has all the necessary information to reconnect and update the request with a new token to try again, as discussed earlier in this section. The second is composed of *http_status* and *body*, so that the error type is still available in case the error has no body.

Figure 8 – Defining Request and Response data structures

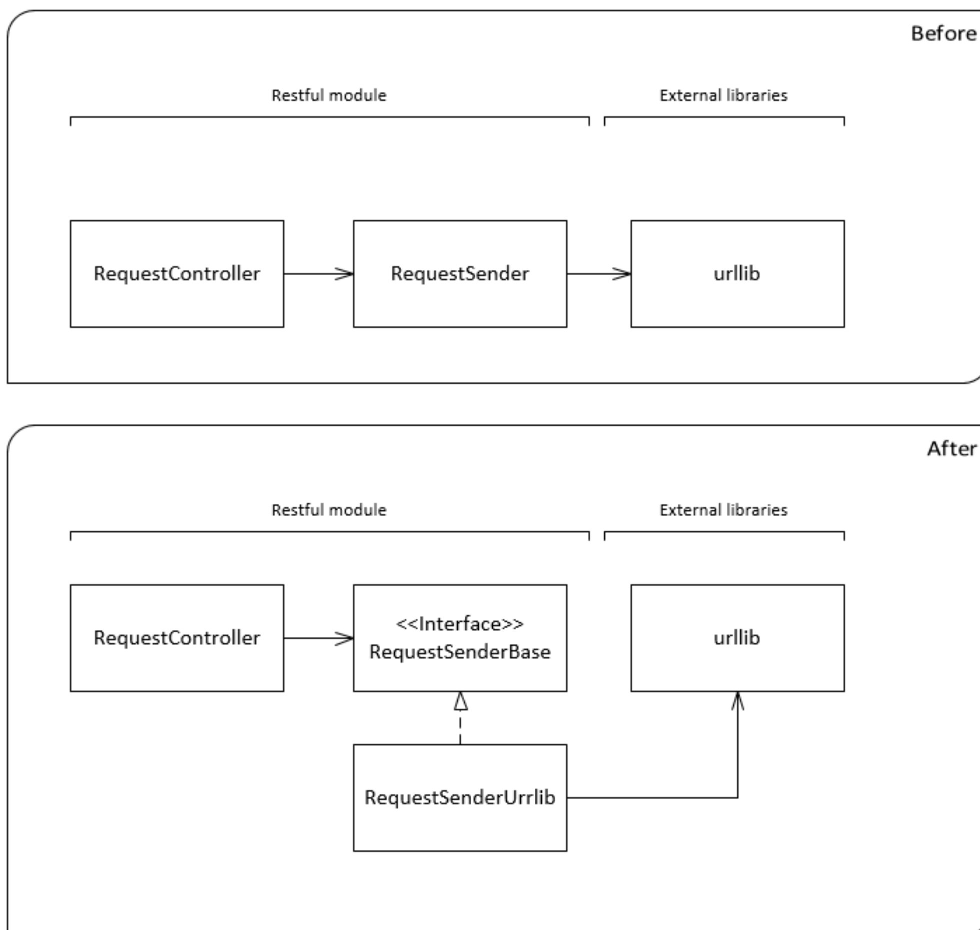


Source: made by the author.

In order to make the rest of the module independent from the library that handles the HTTP requests we can use the Dependency Inversion Principle and *hide* the *RequestSender* class behind an interface as shown in Figure 9. After this change, the *restful* module depends only on the interface *RequestSenderBase* which does not have dependencies on external libraries. The class which contains the implementation using the

urllib is now *RequestSenderUrllib*, which is one implementation of the *RequestSenderBase* interface. In case someone decides in the future that the framework should use other library to handle the HTTP requests, another class can be created inheriting from the same interface and implement the *send_request* method with the other library. Therefore this model also makes the module *open for extension but closed for modification*, following the Open-Closed Principle.

Figure 9 – The RequestSenderBase interface



Source: made by the author.

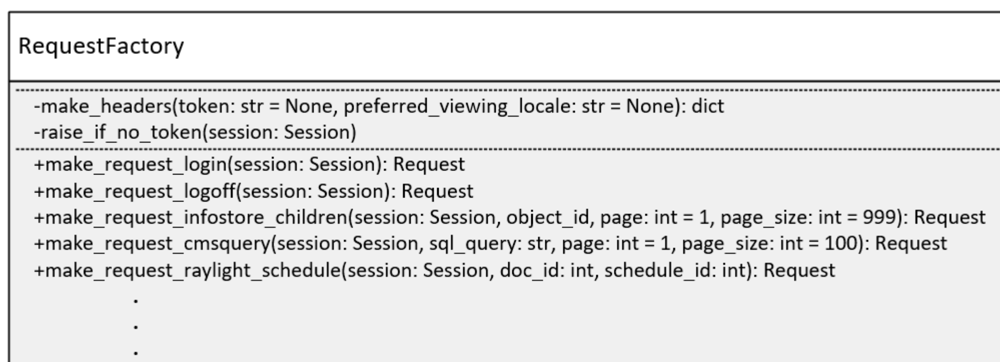
The third problem is the *RequestBase* class and its many implementations. The methods of *RequestBase*, as well of its implementations, have the responsibility of making the *Request* object **and** reading the *Response*. This violates the Single Responsibility Principle because the *RequestBase* class has more than one reason to change. In this case this problem is aggravated due to *RequestBase* behaving like an interface which will have many different implementations, therefore requiring change in all of the implementations as well.

The *make_request* method of *RequestBase* has different parameters depending on the type of the request. All the requests require parameters that will be attached either to

the *url*, the *headers* or the *body*, depending on which API and resource is being requested, as well as the *method*. In this first model these parameters were hidden, as a dictionary argument called *parameters*, to enable the use of the interface. Although this would work to some extent, the fact that the dictionary's keys are not defined and will be different for each kind of request shows that it is only being used to hide that the implementations of this method are not interchangeable and do not satisfy the Liskov Substitution Principle, therefore this method should not be part of the same interface.

To solve this issue with the I modeled the *RequestFactory* class that is responsible for making the *Request* objects, seen on Figure 10. It has different methods that make the different types of requests supported taking the necessary arguments directly, without the use of an ambiguous dictionary. It also has private methods that are used internally implementing functions that will be used for more than one request type.

Figure 10 – The RequestFactory class



Source: made by the author.

The public methods of *RequestFactory* have their names starting with *make_request* prefix before the descriptive name that relates to the API and resource type used. Some of this methods were added to Figure 10 to illustrate how the different methods can take different parameters, but to improve readability the remaining methods were omitted. Another noteworthy change is that the *make_headers* that before was a part of the standalone *HeaderMaker* class was incorporated here as a private method since it is unnecessary outside this class.

4.3.7 Entities Module

The *entities* module contains the higher-level data structure classes, which are used to communicate information between the *sap_framework* component and the tool components. These data structures do not contain any lower-level implementation details.

One of the data structures defined here, for example, is an Enumeration of all the environments of SAP BO platforms in BMW Group. It defines the names of each

environment so that both the *sap_framework* component and the tool components know which names are defined. Then inside the *restful* module the lower-level implementation details such as base URL and server Internet Protocol (IP) addresses are defined for each environment name defined in this Enumeration.

4.3.8 The *date_converter* Module

The *date_converter* module is used for internationalization of the date and time representations. One of the biggest problems that this project faced is that BMW Group has multiple SAP BO platforms in multiple countries, as explained in section 2.3, which are set to different default *locales*.

Normally a system like this would be implemented using *datetime* objects in the database and *back-end* services, which would be then converted to string representations in the user defined locale to be presented in the *front-end* applications.

The implementation of the SAP BO RESTful API, however, is a *back-end* service and uses dates and times in string representations which are formatted to either the user's or the platform's locale settings.

Another problem that was faced to enable this project to be used in all BMW Group SAP BO platforms is that the *Report Templates* and *Instances* objects are also saved with locale information from the last user who edited it. When creating a new schedule with the RESTful API, dates and times filled in the parameters need to be converted from the string representations of these locales to the generic string representation which the RESTful API supports.

This problems inspired the creation of the *date_converter* module, which is used by the *restful* module to convert the string representations to date objects and back, which falls into the *Internationalization and Localization* requirement in section 4.1.2.

4.3.9 Conclusion

The *sap_framework* component includes the *entities*, *date_converter*, and *restful* modules.

The complete class diagram of the framework component can be seen in figure 11. The *date_converter* was omitted from this class diagram because since the last upgrade patch was installed in BMW Group SAP BO environments the problems described in Section 4.3.8 that made it necessary have been fixed. Finally the *environments* module, which fulfills the functional requirement of Section 4.1.1 to "Connect to any environment of any BMW SAP Platform", was also omitted because it contains sensitive information related to the BMW Group servers configuration which I cannot share.

4.4 Use Cases

As previously explained in Section 4.2, the use cases will be implemented in separate components that use the *sap_framework* component to implement the communication to the SAP BO platforms in BMW Group.

4.4.1 The *sap_status* component

Each Quality IT department has their own folder in the SAP BO platform where all the report templates are saved. This report templates are scheduled to automatically generate quality reports. Most of the times a report template will have multiple schedules which have different filters selected, such as *Car Series* or *Cost Center*, or different recurrence information, such as *hourly*, *daily* or *monthly*.

It is responsibility of the Plant Coordinator in the Quality IT team to make sure that the quality reports are properly generated and available to the end users. If one of the systems used by a report template is not available such as a database, the NAS server or the authentication server, or if there is some other problem with the analysis, the schedule may not generate an output report. The output report is only generated when the new report instance reaches status *Completed* in figure 2, which may not happen for many reasons.

Every morning the Plant Coordinator should check the report instances in the three environments, Development, Integration and Production, verify if the reports are generated as expected and, when reports have generated *Failed* instances, verify why they have failed.
To automate this task with a script, it must use the framework to do the following steps:

1. Receive configuration via parameters
2. Connect to one of the SAP BO Environments
3. Given a folder, navigate through the inner folders and get the report templates, recursively
4. Count the total number of instances from these report templates
5. Count the number of instances with status *Recurring* from these report templates
6. Count the number of instances with status *Paused* from these report templates
7. Get information from the instances with status *Failed* from these report templates
8. Organize information in JSON format to be exported and read by external tool

The first and last steps are related to the integration with an external tool that

runs in the server of the *Wrapper Application*. This external tool will use this script with different configurations to read the status of the *Report Schedules* in the SAP BO Platform in folders of multiple departments and in the DEV, INT and PROD environments (see Section 2.3). This external tool will also read the output files generated by this script and save the information in the Apex Database of the Cockpit to be displayed in the web dashboard to all users (see Section 2.4).

4.4.2 The *sap_reschedule* component

The automation of report generation is done in the SAP BO platforms of BMW group using *Schedules*, and since the SAP BusinessObjects Business Intelligence Suite does not provide a tool that support editing schedules in batch, managing them manually in the scale that they are used in BMW Group becomes a high effort task.

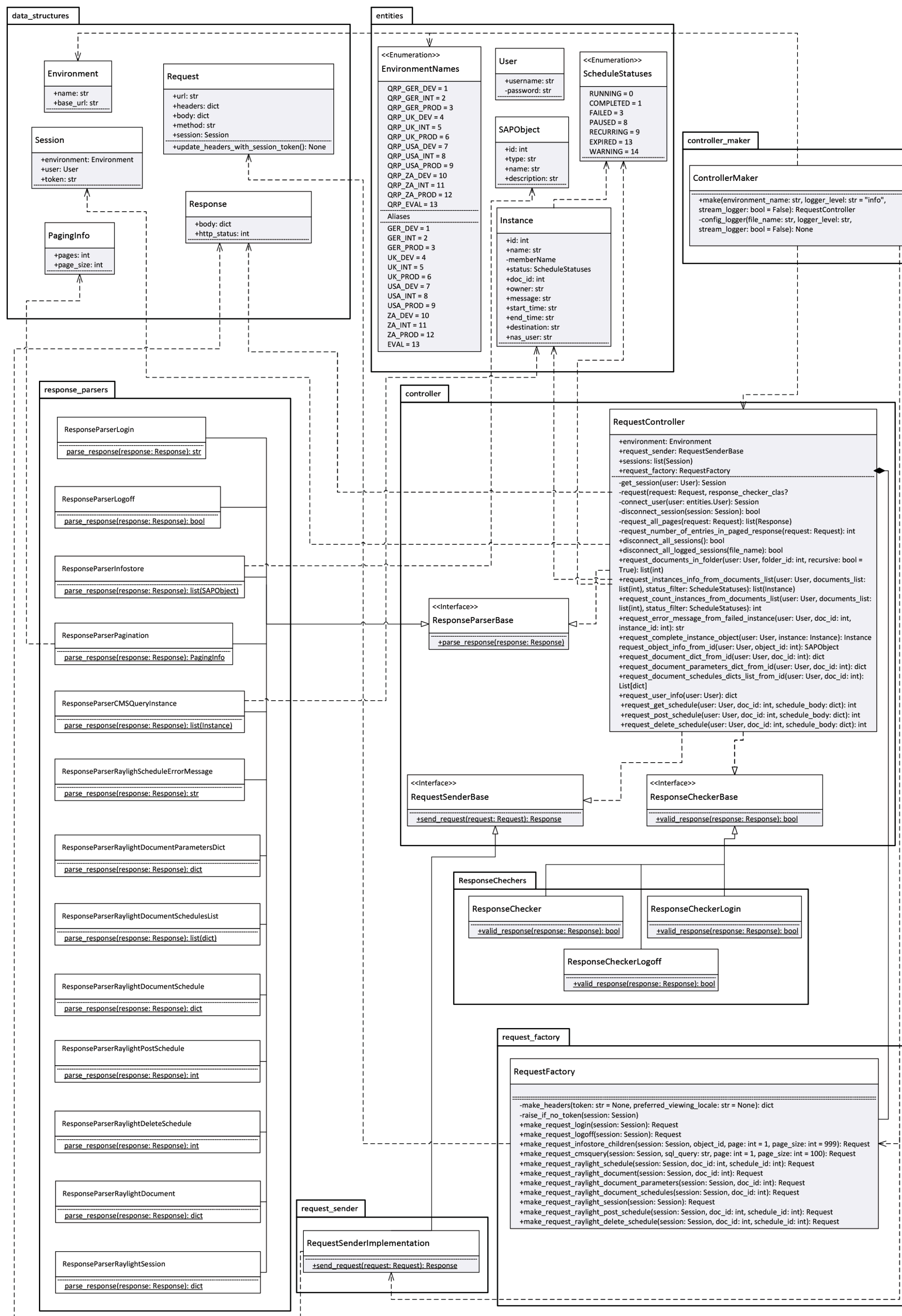
To automate this task and save thousands of work hours per year across BMW Group, the *sap_reschedule* component will implement a script which can select all schedules of the *Report Templates* within a specific folder in SAP BO, alter parameters predefined in a configuration file and post them back into the SAP BO platform.

4.4.3 The *sap_backup* component

The *sap_backup* component will implement two configurable scripts, one of them should be able to select select all schedules of the *Report Templates* within a specific folder in SAP BO, and back them up into a file in JSON format, while the other should be able to read the backup file and restore the schedules into the SAP BO Platform.

The implementation of both scripts will be similar to the *sap_reschedule* script, except that the schedules should not have their parameters altered, allowing the restored schedules to produce the exact same reports that the backed up schedules would.

Figure 11 – Class diagram of the *sap_framework* component



Source: made by the author.

5 Implementation

This chapter will expose the details about the implementation of the project, along with the technologies, processes and techniques chosen to support the requirements and design choices discussed in Chapter 4.

Some sections in this chapter that explain project management processes will often refer to a collective term such as *development team*, simply because that is the proper technical term. Since I am the only developer in this project by the time of writing this document, these terms will usually be referring to myself only, unless otherwise noted.

5.1 Scrum Methodology for Project Organization

The chosen project management methodology to be used in this project was the *Scrum* (SUTHERLAND, 2015). It consists of a framework to organize the tasks of the project and plan their execution. The time of the project is divided into fixed periods called *Sprints* that normally are between one week and four weeks depending on the project.

New tasks are added to the *Project Backlog*. In the beginning of each *sprint* there is a meeting called *sprint planning* where the Scrum Team decides which tasks from the *backlog* should be assigned to the next *sprint*. All members vote for the number of points each task deserves according to difficulty and time estimate to complete it. This points can follow the Fibonacci sequence in order to represent clearly the difference in each step, although because this measurement is subjective, at the end of the *sprint* the team counts the number of points accomplished which is the team's speed. The speed is taken in consideration in the *sprint planning* when choosing how many and which tasks are going to be assigned to the *sprint*.

During the *sprint* the team has meetings called *daily stand up*, which as the name suggests, happen daily and with the members standing in a circle. This meeting is supposed to take very little time, around fifteen minutes considering a team as big as eight people. Each team member should report on what she worked the day before, what is planned for the current day, and which difficulty they might be having. This is important to let the leader (*Scrum Master*) or other teammates know when help is needed to remove obstacles to the productivity.

When a *sprint* ends, the team meets again for *sprint retrospective*, when the work done in the *sprint* is analyzed and the team thinks about what can be improved for the next sprint.

Since I work on the same project full time, it was decided that a period of one

week only would be enough for each Sprint.

The tasks scheduled in this project fall into the following categories: research, development, operations and documentation. Research tasks are related to research on how different tasks can be developed, which include learning how to use the RESTful APIs, using the documentation and tests designed for reverse engineer what is not documented. Development include tasks of development of new functionality in either component on the project. Operations include tasks related to configuring and running the tools that were developed in the project. Finally, documentation tasks include tasks related to documenting the findings of research tasks, the development made, how to use the tools and what was done with them in the operation tasks, as well as the topics that are written in this document.

The *stakeholders* of this project are the *Plant Coordinators*, responsible for quality reporting in all BMW, Mini and Rolls Royce plants, as explained in 2.2. Together they represent all the users of SAP BO in BMW Group who will be benefited from this project, and have helped set the requirements for most of the tools developed up to now.

Finally, the roles of *Scrum Master* and *Product Owner*, who respectively lead the team and decide what should go in the *backlog* and in each order of priority, were shared between my supervisor and I since we did not have a regular sized, five to eight people, *Scrum Team*.

The tools used in BMW Group for project management and documentation are part of the so called Agile Tool-chain, supplied by the Australian enterprise software company Atlassian Corporation. The *Scrum* project management described in this session was implemented in a web tool contained in this package called *Jira*. The internal documentation for the tools developed was written in another web based tool from this package called *Confluence*.

5.2 Git Version Control System

To fulfill the *Version Control* and *Independently developable and deployable components* nonfunctional requirements (see Section 4.1.2) it was decided to use Git. Git is the most widely used version control system today, which is reliable, offers high performance and is fully distributed. In BMW Group the enterprise solution used to manage Git repositories is *Atlassian Bitbucket*. (ATLASSIAN CORPORATION, 2019b)

Git records and organizes the project's source code time line. When a change is saved onto Git, it is called *commit*. Every *commit* has date, time, user, and message information, and represents a specific version of the source code. Git also supports *branching* and *merging* to enable parallel time lines, enabling the developers to work in different

Figure 12 – Git branching illustration

Source: About Git (2019a)

features at the same time.

A *branch* in Git is pointer to a *commit*. It allows developers to load the source code from the *source tree* with all changes up to a specific *commit* and to place the next changes after this *commit*. (GIT SCM, 2019a)

This section will explain how Git was used in this project, although some extra knowledge might be necessary to fully understand it. This work is not meant to be a User Manual to Git, though, because despite the importance in defining the project's workflow, it is just a tool that was used. For full documentation on Git I recommend the book Pro Git (2014), which is available online and for free.

Bitbucket is integrated with the other *Atlassian* tools, *Jira* and *Confluence*, mentioned in Section 5.1. This enables improvements to the regular workflow used with Git.

5.2.1 Workflow

The Git repositories in this project follow a common pattern of branch organization known as Gitflow (ATLASSIAN CORPORATION, 2019a). The commits in the main branch, *master*, receive a release tag and the feature the development of new features is done in feature branches from the *develop* branch.

The process is enhance by using the integration between Atlassian's tools. Every change in the source code is related to a task in the *Scrum Board* in *Jira* as described in Section 5.1, so regardless if it is a new feature, bug fix, refactoring or optimization, there is a *Jira Issue* that describes what is the reason for the change. When a issue is created in *Jira*, it receives a tag which can be used to link to it within *Bitbucket* and *Confluence*.

Apart from the *master* and *develop* branches, the branches in this project are named

with the tag of the *Jira Issue* that it is related to, which helps to keep the repositories organized.

The *commit message*'s subject line also starts with the *Jira Issue* tag, this way after the branch is merged and deleted, the changes in the all commits can still be related to the original *Jira Issue*. Additionally, this reduces the need for long *commit messages*, with separate *subject line* and *body*, since the context of the reason for the change can be accessed in the linked *Jira Issue*.

With these simple process rules, every task in the scrum board in *Jira* also shows the number of *branches* and *commits* there are related to it in *Bitbucket*.

Another important part of the workflow is the *Pull Request*. Both *master* and *develop* branches in all repositories in this projects are blocked for changes without a *Pull Request*. When some change is ready to be merged, the developer will open a *Pull Request* in *Bitbucket*, explain the changes and add other *development team* members and supervisors as *reviewers*. The *reviewers* must approve the pull request for it to be merged automatically within *Bitbucket*, and they might request changes before approval.

Another benefit is that the *Pull Request* detects when *merge conflicts* are present and disables the *merge* button. The reason for *merge conflicts* in this stage is usually that the *feature branch* has outdated base code, and the conflicts are responsibility of the developer, the assignee of the *Jira Issue* and creator of the *Pull Request*, to fix. In order to approve the *Pull Request* the reviewers should require the conflicts to be solved by the developer, and since the developer has no change rights on the destination *branch*, the changes should be done with a *Rebase* procedure of the *feature branch*, updating the base code for it.

The procedures and guidelines described in this document have been fully documented in the project's *Confluence* page to help future team members to follow the same structure, with tutorials and examples.

5.2.2 Git Submodules for Component Source Management

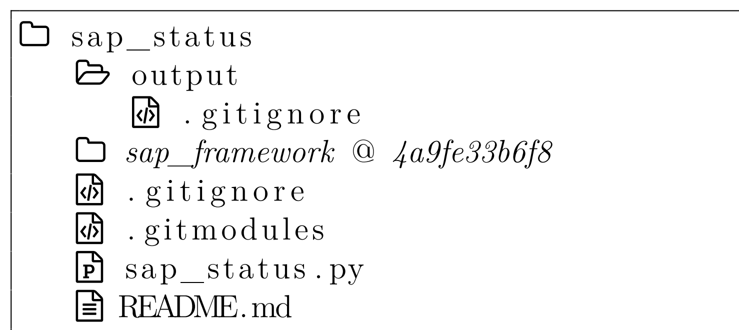
In Section 4.2 it was explained that the project will be composed by one framework component and other components that will use it to implement different use cases. Additionally, each component would need to use a version control system that would allow them to be independent of each other and use different versions of the framework component if necessary.

To implement this architecture, each of the components in Figure 6 will have their own Git repository in *Bitbucket*. The functionality of Git that allows a repository to be used by others is called *submodule* (GIT SCM, 2019b).

Whenever a new tool repository is set up, the *sap_framework* repository should be initialized as a *submodule*. This process is similar to *cloning* the *sap_framework* repository inside the other repository, however with the key difference that the changes in the *submodule* will not be tracked as source code changes. Instead, the *sap_framework* directory inside the root repository will be tracked as a *commit* from the *sap_framework* repository.

Figure 13 shows the folder structure of *sap_status* repository which uses the *sap_framework* as a submodule, and in this case the submodule is referencing the commit *4a9fe33b6f8* in this repository.

Figure 13 – *sap_status* repository structure with *sap_framework* as submodule



Source: made by the author.

With this configuration, each of the repositories of the remaining components in figure 6 will be linked to the *sap_framework* repository and have the hash of the *commit* which corresponds to the version of the *sap_framework* component they are using. Furthermore the *submodule* command in Git has many features to manage which commit is tracked, fetch source code and run other Git commands in the *submodule*'s repository.

5.3 Python 3

Python is an open-source programming language that in the last decade has grown to be one of the most important in many areas of computer science. It was decided by the department that this should be the language used by this project since it is relatively simple to learn and one of the most important aspects of this project is that it should be easily maintained by people who are new to this subject.

Python is a dynamically typed, interpreted, language, which meas that the Python interpreter reads the source code and interprets it in real time and there is no compilation required. It also means that the Python interpreter with the right version should be installed in the environment which will run the application.

BMW Group has an IT System which has custom builds of applications to be installed in the company's machines, and regular users must use the applications available

in this system. The Python interpreter version used for this project is therefore the most recent version that was available in this system when the project started, which is Python 3.7.

The source code in this project was developed following the Style Guide for Python Code (2001), also known as PEP-8. This code style standardizes the code layout style, naming conventions for variables, functions, classes and modules, comments and annotations in general. It was first created in 2001 but has been updated over the years, with the latest changes been made in 2013.

5.3.1 Standard Libraries Limitation

Because of this software installation constraints it was decided that this project should only use standard Python packages, that are included with the interpreter available for internal use, in order to allow the internal deployment to be done with no additional rights needed. This fulfills the *Portability* requirement from section 4.1.2.

5.3.2 Components and Modules

In Chapter 3 the names *component* and *module* were introduced as software engineering terms referring to different levels of functional parts of a software application.

However, the term *module* in Python usually refers to a file containing Python source code. When source code organized in more than one Python file in the same directory is to be used by modules outside this directory, the directory is called a Python *package*. (PYTHON SOFTWARE FOUNDATION, 2019, Section 6. Modules)

Therefore, to implement the structures described in section 4.2 with the term *software component* in Python, *packages* and *modules* will be developed.

5.4 Test Driven Development

TDD is a programming *process* that helps improve the software quality, as explained in Section 3.4.1. The software components in this project are supposed to be independently developed and different approaches can be used in the development of each application component though the *core* component, the *sap_framework*, used and relied on by all the tools in this project, must have its quality assured at all times, and will therefore be developed using TDD. It is also is deeply related to *Testability* and *Documentation* nonfunctional requirements of Section 4.1.2.

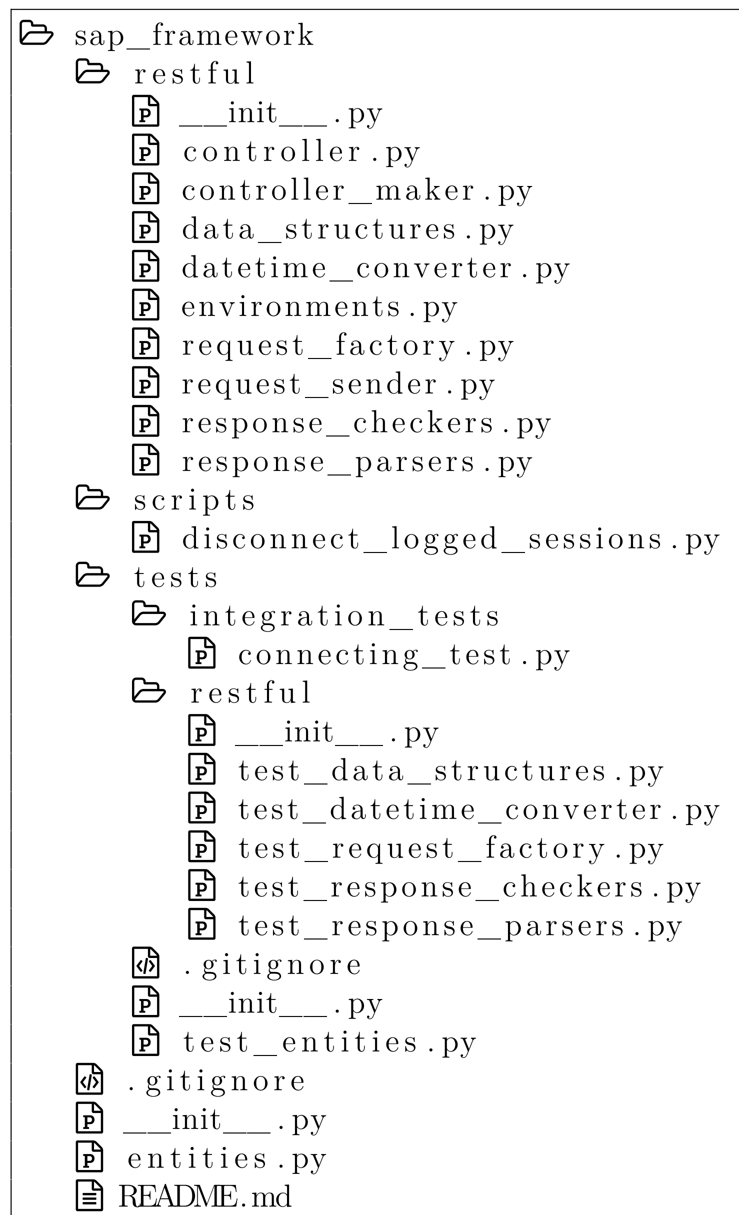
The unit test framework used is Python's *unittest* since it is the one that is included in the standard modules in this python version (see Section 5.3).

It was decided to use a *tests* directory inside the *sap_framework*'s repository root directory, mirroring the source code directory structure, and with a test module for each source code module on the repository. Test modules are named following the pattern *"test_*.py"* following the *unittest* standards, and using the original source code module name in place of the *wildcard* *. See Figure 14.

After the unit tests and source code classes are created following the TDD, the code is refactored so that for each class in the original source code module, there will be a test class (subclassing *unittest.TestCase*) with the same name prefixed by *Test*.

This structure facilitates the localization of test and implementation modules, which is good since they must always be edited together when using TDD.

Figure 14 – *sap_framework* directory structure



Source: made by the author.

5.5 The *sap_framework* Component

The core component in this project is the *sap_framework* component, which implements the module with the highest level of abstraction data structures, called *entities*, and the restful package, described in Section 4.3. This component is used by all the tool components as shown in Figure 6.

This component was implemented prioritizing maintainability and reusability following the Component Cohesion Principles described in Section 3.3.1.

The *restful* package was implemented following the described design from section 4.3. Although I am not allowed to share the source code for this project, some implementation option details are still worth mentioning, and I am authorized to share some parts that help illustrating the descriptions.

5.5.1 Data Structures

One of the functional requirements of the framework in section 4.1.1 is to "Provide necessary data structures to represent the requested data". This section is going to explain how this was implemented in Python.

The way the data structure classes were implemented both on the *entities* module and on the *restful* package is the same. A class is created and all the properties are added to the *__init__* magic method as a constructor. Usually *getters* and *setters* are not implemented since Python allows direct access to an object's properties. All the data structures also override Python's *__repr__* magic method, which is used to define a representation of the object that can be used to rebuild it with the *eval* function just like the standard types.

The example code in Figure 15 shows the implementation of the *PagingInfo* data structure in Python. The two attributes are loaded in the constructor, as well as the documentation in the default *docstring* format so that any developer can check it in within the Integrated Development Environment (IDE). *Docstrings* like these are used in most of the classes and methods to fulfill the *Documentation*, *Readability* and *Mainteinability* nonfunctional requirements from section 4.1.2.

The only data structure class that does not follow the this pattern completely is the *User* class, which hides the *password* attribute in the representation by replacing the characters of the representation with the character "*". This means that the representation of the object can still be used to recreate it with the *eval* method, but the password will not be restored. This is important to prevent a *User* object, or even a *Session* object which contains a *User* object by a composition relationship, to be logged with sensitive information (see class diagram in Figure 11).

Figure 15 – Implementation of data structure classes in Python

```
class PagingInfo:
    def __init__(self, pages: int, page_size: int):
        """
        This class contains the paging information of a the
        Response of a request which is paged.

        :param int pages: Number of pages.
        :param int page_size: Size of page.
        """
        self.pages = pages
        self.page_size = page_size

    def __repr__(self):
        return "PagingInfo(pages=%r, page_size=%r)" % (
                                    self.pages, self.page_size)
```

Source: made by the author.

5.5.2 Interface Classes

Python does not have *interfaces* defined like other languages such as Java or C# because it is not necessary since it is dynamically typed and every object has implicit interfaces that can be used with *duck typing*. However it supports abstract classes that can be used with abstract methods that define the interface of its subclasses, which will be required to override all abstract methods and implement them.

This type of class is very important to implement the Dependency Inversion Principle (DIP) described in section 3.2 and fulfill the *Extensibility* nonfunctional requirement from Section 4.1.2.

In python, the *abc* module (which stands for *Abstract Base Class*) is used to create such abstract classes. The implementation for the interface in figure 9 can be seen in Figure 16.

The abstract class in Figure 16 is a class that inherits from the *Abstract Base Class* and the abstract method uses the *abstractmethod* decorator, both from the *abc* module. There is no implementation of any method in this class, it contains only the *docstrings* shown. To implement this interface, a class must be created which inherits from it, and all the abstract methods should be overwritten, otherwise Python will raise an error.

5.5.3 The *Main Component* of *sap_framework*

The *Main Component* is the component which takes care of the *dependency injection* process, making instances of the lower-level classes that will be used by the higher-level ones, as explained in Section 3.3.2.

Figure 16 – Interface implementation with abstract class in Python

```
import abc

class RequestSenderBase(abc.ABC):
    """
    Defines the interface to be implemented by a
    RequestSender plugin using any library
    """

    @abc.abstractmethod
    def send_request(self, request: Request) -> Response:
        """
        Makes the HTTP request and gets the response

        :param request: Request object to be sent

        :return: Response object with info of the response of
        the request
        """
```

Source: made by the author.

In the *sap_framework* the higher-level module which contains the *business rules* is the *controller*. The constructor of the *RequestController* class receives as parameters instances of lower-level classes using the *dependency injection* technique, as well as additional settings. To enable other components that use the *sap_framework* to make an instance of this class without depending directly on those lower-level classes within the framework, a *Main Component* was implemented called *controller_maker*.

The class in Figure 17 has two methods, one public and one private. The *make* method configures the package logger according to the parameters by calling the private method *_config_logger*, validates the string with the name of the environment which is to be connected, instantiates and *Environment* object with the correct name and base URL, instantiates the *RequestSenderUrllib* and *RequestFactory* implementation classes, and then uses these to instantiate the *RequestController* class.

The dependency injection process is also very important to fulfill the *Testability* nonfunctional requirement from section 4.1.2, as it allows for mock classes to be used in unit tests and test the behavior of one module isolated.

5.5.4 Information Flow

The *sap_framework* component exposes the functionality of the SAP BO RESTful API using a facade design pattern in the *RequestController* class. This class implements public methods to every kind of request implemented in the framework. These methods

Figure 17 – The ColtrollerMaker class implementation

```
class ControllerMaker:
    @classmethod
    def make(cls, environment_name: str, logger_level="info",
             stream_logger=False) -> RequestController:
        """
        Makes a properly configured instance of RequestController.
        :param str environment_name: Name of SAP BO Environment in BMW.
        :param str logger_level: level of the package's logger.
        :param bool stream_logger: Send logger output to stdout stream.
        :return: RequestController
        """
        cls._config_logger("restful.log", logger_level, stream_logger)
        try:
            sap_env = EnvironmentNames[environment_name]
        except KeyError:
            env_error = "Environment {} not recognized." \
                        " Please check {}.{}".format(environment_name,
                                                     EnvironmentNames.__module__,
                                                     EnvironmentNames.__name__)
            logging.error(env_error)
            raise ValueError(env_error)
        environment = Environment(name=sap_env.name,
                                  base_url=ENVIRONMENTS[sap_env]["url"])
        request_sender = RequestSenderUrllib()
        request_factory = RequestFactory()
        controller = RequestController(environment,
                                       request_sender,
                                       request_factory)
        return controller

    @classmethod
    def _config_logger(cls, file_name, logger_level, stream_logger=False):
        numeric_level = getattr(logging, logger_level.upper(), None)
        if not isinstance(numeric_level, int):
            raise ValueError('Invalid log level: %s' % logger_level)
        handler = logging.FileHandler(file_name, "w", "utf-8")
        handlers = [handler]
        if stream_logger:
            handler = logging.StreamHandler(sys.stdout)
            handlers.append(handler)

        logging.basicConfig(handlers=handlers, level=numeric_level,
                            format="[%(asctime)s %(levelname)s %(filename)s:%(lineno)s|"
                                   "%(funcName)20s() ] %(message)s",
                            datefmt='%H:%M:%S')
```

Source: made by the author.

control the information flow in the modules of the framework and return the requested information to the tool component who called them.

The implementation of this methods follows a similar pattern in most of the cases, which can be summarized into these steps:

1. Get a session for the user

2. Use one method of the *RequestFactory* class to make a *Request* object

3. Send the request using the *_request* private method

4. Use the right *ResponseParserBase* implementation to parse the response

5. Return the parsed data

Please refer to Figure 11 for a visual representation of the modules and classes described in this section.

The information flow between the modules of the *sap_framework* component will be illustrated with an example which reads the error message from a report that failed to run (report instance with status *failed*, as explained in Section 2.3). This function is implemented in the method *request_error_message_from_failed_instance* shown in figure 18.

Figure 18 – Example facade method implemented in RequestColtroller class

```
def request_error_message_from_failed_instance(self, user: entities.User, doc_id: int,
                                               instance_id: int) -> str:
    """
    Gets the error message from a failed instance.

    :param User user: SAB BO user to be used in the requests.
    :param int doc_id: ID of the parent document in the platform.
    :param int instance_id: ID of the instance in the platform.
    :return: Error message.
    :rtype: str
    """
    logging.info("Requesting error message for instance {}".format(instance_id))
    session = self._get_session(user)
    request = self.request_factory.make_request_raylight_schedule(session, doc_id,
                                                                  instance_id)
    response = self._request(request)
    message = ResponseParserRaylightScheduleErrorMessage.parse_response(response)
    return message
```

Source: made by the author.

In step number 1, a the private method *RequestController._get_session* is used to get a *Session* object for the user.

One thing that is worth noting is that the public methods that implement the requests using the framework such as the one in Figure 18 do not use a authentication token as parameter, they require only the *User* object that should be used in the request. This is because all the session management is dealt with inside the framework. The private method *RequestController._get_session* checks if there is already a session open for this user to reuse the authentication token, otherwise it creates a new session by sending a login request, and stores this session for future uses. To properly disconnect all sessions in use the public method *RequestController.disconnect_all_sessions* can be called manually, but it is also called automatically in the routine that destroys the *RequestController* instance (Python's magic method *__dell__*). The representations of the *Session* objects (see section 5.5.1) created are also logged, and can be recreated to be disconnected by *RequestController.disconnect_all_logged_sessions*, which is also available as a standalone script found in *sap_framework/scripts/disconnect_logged_sessions.py* (see project structure in figure 14). Note that this fulfills the three functional requirements from Section 4.1.1 related to session management.

In step number 2, a *Request* object is made by the *RequestFactory* class. Each of

the public methods in this class implement a different type of request supported by the SAP BO RESTful API along with the necessary parameters and builds the *Request* object accordingly, as described in Section 4.3.6. The method used in the example in Figure 18 is the *RequestFactory.make_request_raylight_schedule* and its implementation can be seen in Figure 19.

Figure 19 – Example method implemented in RequestFactory class

```
def make_request_raylight_schedule(self, session: Session, doc_id: int,
                                   schedule_id: int) -> Request:
    """
    Makes a Request object for the details of a schedule (document instance) with the
    raylight API.

    :param Session session: Session object to be used for the request.
    :param int doc_id: id of the parent document of the desired schedule.
    :param int schedule_id: id of the schedule.
    :return: Request object.
    """
    self._raise_if_no_token(session)
    url = "{base_url}/raylight/v1/documents/{doc_id}/schedules/{schedule_id}".format(
            base_url=session.environment.base_url, doc_id=doc_id, schedule_id=schedule_id)
    body = {}
    headers = self._make_headers(session.token)
    method = "GET"
    return Request(url, headers, body, method, session)
```

Source: made by the author.

In step number 3, the *RequestController._request* method is used to send the *Request* object.

The *_request* private method of the *RequestController* class is the only place the *RequestSenderBase* implementation (see Figure 9) is used to send the request to the SAP BO Platform, and since this implementation is *dependency injected* into the *RequestController* class, a *mock* implementation can be used to replace this integration in unit tests. This private method also uses one of the *ResponseCheckerBase* implementations to verify if the response of the platform is proper, otherwise the request is sent again, solving the problems described in Sections 4.3.2 and 4.3.3. It also verifies if the session has expired and reconnects before retrying, fixing the problem described in Section 4.3.4.

In step number 4, one of the *parsers* that implement the *ResponseParserBase* abstract class is used to parse the *Response* object and get the relevant information or data structure that should be returned, which in this example is the error message of the *Failed* report. The implementation of the parser used in this example can be seen in Figure 20.

The *body* attribute of *Response* object in this example contains many information about the *Failed* report, such as scheduling information, destination and parameters, none of which is going to be read by this parser, since the only information relevant in this request is the error message. Other *parsers* may be implemented to read other information from the same type of *Response*.

The classes that implement the *ResponseParserBase* abstract class fulfill the

functional requirement "Parse the responses and terurn the relevant information" from Section 4.1.1.

Figure 20 – Example ResponseParserBase abstract class implementation

```
class ResponseParserRaylightScheduleErrorMessage(ResponseParserBase):
    """
    This class implements the ResponseParserBase interface for reading the error message
    of a failed schedule in the response of the raylight schedule request.
    """

    @staticmethod
    def parse_response(response: Response) -> str:
        try:
            return response.body["schedule"]["error"]["message"]
        except KeyError:
            return ""
```

Source: made by the author.

Finally, in step number 5, the data read by the parser is returned.

The framework was designed to access information using different APIs from *SAP BO* to fulfill the requirement "Retrieve data with different APIs available in the platform" from Section 4.1.1. Most of the documentation for the SAP BO RESTful Web Services SDK cover the *Raylight* API, which does not require additional features such as page navigation. These features had to be developed by reverse engineering the responses.

Some of the supported requests in *Infostore* and *CMSQuery* RESTful APIs for SAP BO may have a list of entries in their responses and those responses may have multiple pages, requiring multiple requests using *paging* parameters. To deal with this multiple page requests, the *ResponseParserPagination* was developed, which implements the *ResponseParserBase* abstract class and returns the data structure seen in figure 15.

To use pagination, the *RequestController._request* may be replaced with *RequestController._request_all_pages*, which will iterate through the pages and return a list of *Response* objects. However, if the number of entries only is required, the *RequestController._request_number_of_entries_in_paged_response* can be used instead, since it overwrites the *Request* object setting the maximum number of entries per page to 1, sends it with *RequestController._request*, uses the *ResponseParserPagination* to access the number of pages, which is much faster than having to request all pages and count the entries in the responses.

5.5.5 Extensibility

The architecture implemented as described in the previous sections can be easily extended to increase support to other functions that can be accessed via the he SAP BO RESTful Web Services SDK, which fulfills the *Extensibility* nonfunctional requirement from section 4.1.1.

In Section 5.5.4 it was explained that each function that the *sap_framework* gives access through the *RequestController* class uses (at least) one *request maker* method from *RequestFactory* to create the *Request* object and one *parser* which implements the *ResponseParserBase* abstract class. The example also shows that multiple *parsers* can be used for the same response (from the same request made by *RequestFactory*) to access different information.

If the framework needs to return other information from the same *Response* object, a *parser* may be added implementing the same abstract class, along with the addition of a new public method to the *RequestControler* facade. If the information to be returned is more complex than the error message of type string as showed in the example, however, a data structure can be added to the *entities* module so that the parser can instantiate it.

However, if a new type of request is necessary, besides additions mentioned above, the method that makes this new request type should be added to *RequestFactory* class.

Notice that the changes mentioned above are all *additions*. This is a clue that the architecture implemented follows the Open-Closed Principle (OCP) described in section 3.2 and therefore is *open to extension*.

5.6 The *sap_status* Component

This component contains a script which uses the *sap_framework* to read information related to instances of report templates in the SAP BO Platform.

The script uses the following parameters:

- user name of SAP BO user
- password of SAP BO user
- Folder in SAP BO
- Platform country
- Environment
- Path to output folder
- Output file name prefix

It validates the input parameters, creates a *User* object and uses the *Controler-Maker* constructor to to create a *RequestController* object, which will give access to the framework's methods.

It requests the *ID*s of the document templates inside the folder, recurrently. Then, with this list of *ID*s, it requests to count the instances of this report templates, in total

and filtered by status: *Recurring* and *Paused*, and to read the information of the instances with status *Failed*.

This is all it takes to read the necessary status information from the platform using the *sap_framework*: four function calls. The way the script in the *sap_status* component uses the *sap_framework* component is illustrated in Figure 21.

Figure 21 – How the script in *sap_status* uses the *sap_framework*

```
controller = sap_framework.ControllerMaker.make(environment,
                    logger_level="DEBUG", stream_logger=False)

user = sap_framework.User(username, password)

documents_list = controller.request_documents_in_folder(user, folder_id)

number_total_instances = controller.request_count_instances_from_documents_list(user,
                                                    documents_list, None)

number_recurring_instances = controller.request_count_instances_from_documents_list(user,
                                                    documents_list,
                                                    sap_framework.ScheduleStatuses.RECURRING)

number_paused_instances = controller.request_count_instances_from_documents_list(user,
                                                    documents_list,
                                                    sap_framework.ScheduleStatuses.PAUSED)

failed_instances = controller.request_instances_info_from_documents_list(user,
                                                    documents_list,
                                                    sap_framework.ScheduleStatuses.FAILED)
```

Source: made by the author.

The rest of the script prepares the JSON output with the platform information, run time of the script, number of *Recurring*, *Paused* and *Failed* instances, and the details of each failed instances.

The prototype of this script is run multiple times with a set of configurations for different folders and environments by another service in the same server of the Wrapper Application, and the output is stored in a set of tables in the Cockpit's database in order to show the status information on the Cockpit (see section 2.4).

5.7 The *sap_reschedule* Component

This component contains a script that uses the *sap_framework* to read all the schedules from a specific folder in SAP BO, select the ones whose parameters match a rescheduling criteria, and reschedule the reports.

This script uses a configuration file in JSON where the user specifies the folder in SAP BO, the environment, the users that authenticate in the SAP BO platform and in the NAS, whether this users should be overwritten, whether the *Completed* instances should be kept in the report template's history, possible paths for destination NAS changes, etc.

Figure 22 – JSON Configuration file to reschedule script


```
{
  "environment": "QRP_GER_PROD",
  "folder_id": "12345",
  "name_prefix": "TEST_",
  "planning_mode": true,
  "sap_bo_users": {
    "user1": "password1",
    "user2": "password2"
  },
  "overwrite_schedule_user": true,
  "nas_users": {
    "user2": "password2",
    "user3": "password3"
  },
  "overwrite_nas_user": false,
  "clear_old_instances_successfully_rescheduled": false,
  "overwrite_keep_instance_in_history": true,
  "keep_instance_in_history": true,
  "filesystem_path_changes": {
    "//server1/folderName1": "//server3/folderName1",
    "//server2/folderName2": "//server4/folderName2",
  },
  "ignore_schedules_with_different_filesystem_path": true,
  "filter": "Recurring",
  "run_limit": 20,
  "wait_time": 1
}
```


Source: made by the author.

Figure 22 shows the JSON structure of the configuration file used by this script. The parameters of the script can give an idea of the features that were implemented into it.

1. **"environment":** This parameter is the name of the environment of SAP BO that will be used and it is used to set the URL for all RESTful API requests.

2. **"folder_id":** This is the ID in SAP BO of the folder in which the script will look for folders, reports and schedules.

   When the script starts running it prompts the user to chose one or all folders inside this folder, in case it is required reschedule the subfolders one by one. Then it shows the user all the documents found in the selected folder(s) and prompts to choose one or all.

3. **"name_prefix":** This defines the prefix that will be added to the name of the resulting instances and its output files. Normally it is defined as "*TEST_*" when

testing rescheduling reports with the script and "" when rescheduling productive instances.

4. **"planning_mode":** This parameter defines whether the script is going to actually reschedule the instances or just check for how many instances are there which match the criteria given in the configuration file. This is helpful to check beforehand if there are users missing in the configuration file for example.

5. **"sap_bo_users":** This is the *usernames* and passwords of the users that will be used to login into SAP BO. It can be used with one or more users.

   The first user listed is going to be used to read all folders, documents and instances, the other ones will only be used to save the new schedules to SAP BO.

6. **"overwrite_schedule_user":** If this parameter is *false*, the script will not change the owner of any schedule, therefore ignoring schedules that have owners not listed in *sap_bo_users*.

   If *overwrite_schedule_user* is *true* however, the script will use the first user in *sap_bo_users* to post all new schedules, so they will always have the same owner and no schedule will be ignored for this reason.

7. **"nas_users":** This are the *usernames* and passwords for the users that will be used to save the reports to the NAS. It can be used with one or more users.

8. **"overwrite_nas_user":** If this parameter is *false* the script will not change the user used for NAS authentication, therefore ignoring schedules that have NAS users not listed in *nas_users*.

   If *overwrite_schedule_user* is true however, the script will use the first user in *nas_users* to post all new schedules, so they will always have the same NAS user and no schedule will be ignored for this reason.

9. **"clear_old_instances_successfully_rescheduled":** If this parameter is *false* the script will not delete the original schedules after creating the old ones. This is a common use case for tests, where you just want to create a duplicate of the schedules.

   If *clear_old_instances_successfully_rescheduled* is *true* however, the script will delete the old schedules after creating the new schedules successfully.

10. **"overwrite_keep_instance_in_history"** and **"keep_instance_in_history":** If *overwrite_keep_instance_in_history* is *false* the script will read this configuration from all the original schedules and keep the same in the new schedules.

If *overwrite_keep_instance_in_history* is *true* however, the script will not read this configuration from the original schedules and will write the value of *keep_instance_in_history* to all new schedules.

11. **"filesystem_path_changes":** This describes the changes of paths in the file system output. This was designed to migrate the NAS server or change the alias, but this configuration may also be useful to send the output of new test schedules to a different location so they do not overwrite the production reports in the production NAS.

12. **"ignore_schedules_with_different_filesystem_path":** This configuration is set to true when the script is running to change the NAS directory/alias. If set to *true*, the script will ignore the instances which the original destination does not match the original alias (keys) in "*filesystem_path_changes*".

13. **"filter":** This *filter* determines what kind of schedules is going to be rescheduled. It can be "*Recurring*" or "*Paused*".

    Please notice that although we can create new schedules based on "*Paused*" schedules, we cannot change the status of an instance via the API and the default status for a schedule is "*Recurring*", thus the schedules generated with the script will always be "*Recurring*".

14. **"run_limit":** Most of the schedules start running right after they are created, but the SAP BO platform cannot handle many running instances at the same time. The script uses *run_limit* to try not overloading the platform.

    run_limit is the maximum amount of running instances from the reports that the script is rescheduling. The script uses this number to check how many schedules can be created in each time slot. It counts the number of running instances and if it is less than *run_limit* it calculates the number of schedules to create in the next window as *run_limit - running_instances.*

    The default value of *"run_limit": 20* has been tested and is recommended that it is only changed in case of problems.

15. **"wait_time":** This is the time in seconds the script waits between each scheduling window. This is necessary to wait for the running instances to be processed and can be increased if facing problems.

After the script reads the schedules from the platform and selects which of them match the criteria in the configuration file, if goes through all of the selected ones, gets the schedule body, reads the parameters of their parent *Report templates* to combine the parameter structure with the answers of the schedule to create a new schedule body,

changes the parameters that should be changed according to the configuration file and creates the new schedule.

If the new schedules are successfully generated, a copy of the old ones is saved into a backup file and the old ones are removed from the platform if the option to clear old instances successfully rescheduled is checked in the configuration file.

To test this script I manually created multiple *schedules* of different *Report Templates* in a test folder in the DEV environment of the German Platform, with a wide variety of parameters and *schedule types* which would represent all the possible requirements then manually set the script to reschedule these *schedules* and verified the results. Once the results were approved, I contacted colleagues that work in the other platforms to let me repeat the tests on their environments and verify whether the requested functionality could be validated.

5.8 The *sap_backup* Component

This component contains two scripts. The first one uses the *sap_framework* to read all the schedules from a specific folder in SAP BO and save them in a file in JSON format. The second one restores the previously backed up instances to the SAP BO Platform.

The approach to read schedules from SAP BO reports and to post schedules back into SAP BO is similar to the one used in *sap_reschedule*, except that all schedules are saved and their parameters are left untouched.

The script that restores backups can also be used to restore a backup made by the *sap_reschedule*, which was actually the first functionality that was implemented in this component.

6 Results

This chapter will discuss the results from the implementations of this project and its components.

The implementation described in Chapter 5 fulfills all functional and nonfunctional requirements from Sections 4.1.1 and 4.1.2, as pointed out in the each section of this chapter.

The project was well received by the company and the results were considered successful. During the time of this internship there were frequent opportunities to receive insights and feedback by many colleagues that work in Quality IT departments throughout BMW Group (the project's *Stakeholders*, mentioned in Section 5.1), as well as colleagues from the supplier company responsible for operation of BMW Group's SAP Business Objects Platforms, environments and production databases. This opportunities were extremely valuable to make this project useful and versatile and better fit the company's needs.

6.1 The `sap_framework` component

The framework is the main component of this project and contains the necessary pieces of software needed by the projects tools to use the SAP BO RESTful Web Services.

It was designed following standardized software architecture principles (section 4.2) and implemented using the best practices in software development (sections 5.1, 5.2.1, 5.3, 5.4 and 5.5).

Table 3 – Statistics of the `sap_framework` component

Property	Amount
Lines of Code	1048
Lines of documentation	448
Classes	29
Modules	10
Unit tests	87

Source: made by the author.

The implementation with TDD helped not only to have a resulting component with a high test coverage, as can be seen by the number of unit tests in table 3, but also to ensure better design choices, which was one of the benefits described in Section 3.4.1. This was expected since the literature commonly mentions this as a consequence of applying TDD, but since this was the first time I have implemented it in a project from the beginning

it was a very good experience. By programming for testability, the developer is forced to make small and modular pieces of code that have well defined interfaces to fit together in a software component.

Along with the application of the SOLID Design Principles (see Section 3.2), this can explain why the resulting component has as many modules and classes as can be seen in table 3 or Figure 11.

The implementation details shared in Section 5.5 show how *flexible* and *extensible* the architecture of this component is.

However, even with the use of good practices such as illustrated in Figure 9 aiming to make it as *decoupled* as possible, it is still *strongly coupled* to the implementation of the RESTful API by SAP. This is not a problem of the architecture itself, though, since the objective of this project is to provide ways to automate the management of the SAP BO Platforms in BMW Group, there would be no point in trying to decouple it from SAP BO.

6.2 The `sap_status` component

This component contains a script that can be configured to check the status of the platform, by requesting relevant information from instances inside a specific folder in SAP BO, that otherwise would require manual searching in the tool Instance Manager available in the Central Management Console web application provided by SAP.

The process, when done manually, requires the plant Coordinator to login to each of the Production, Integration and Development environments and search for the instances of the reports that belong to the folder which he is responsible for. This process takes an average of 45 minutes and should be executed at least once daily.

The script developed in this component can be configured with a set of parameters and gets the same required information between two and four minutes depending on the amount of instances and the load on the platform itself.

This script is integrated in a prototype of another tool developed in this department that runs it in a server multiple times with different configurations to read the status of the folders of different Technologies (and folders) in all environments. This information is then written on our web server database from which the dashboard web application reads and shows it to the users of the platform. This prototype is currently running daily in a test environment and it is planned to go on production soon.

A view of this web dashboard can be seen in Figure 3, which is the homepage of the Paintshop's Quality IT department of Plant Dingolfing in the Cockpit (see Section 2.4). The status bar on the top right of the image shows the percentage of failed reports in each environment within the quality reports that this department is responsible for. The

buttons labeled "Details" open a pop-up window with the detailed information of the failed report instances of that environment.

6.3 The sap_reschedule component

The script that reschedules instances from a specific folder in SAP BO was the one which had most impact in the company.

Most of the quality reports generated in the BMW Group SAP BO platform are saved in a NAS, see Section 2.3. The NAS server is configured with folder specific user permissions and requires user authentication to save the reports, so scheduling a report in SAP BO to a file system destination requires filling the user data for authentication.

All employees are required to change their passwords monthly, so if the employees use their personal users to schedule reports in SAP BO, these schedules will fail after the password change, as can be seen in Figure 2. For this reason, the *Report Schedules* are created with *Technical Users*, which ideally would have password changes yearly.

The development of this script was first requested to support this password change for *Technical Users*, because some of this users were used in thousand or report schedules. Since each recurring instance takes around 7 minutes to be rescheduled manually, this process would take weeks to to be completed, leading to a huge downtime in the quality reporting systems. The script in this component was created for this task but it has not been used for it yet, since the IT department has postponed the date for the password change of the *technical users*.

Once a script to reschedule report instances was available, more use cases started to be requested. The IT department needed to deactivate the NAS server that was in use and requested the Quality IT *Plant Coordinators* (section 2.2) to reschedule the recurring instances of reports in SAP BO to save the reports to the new *NAS* server, which if not done before the deadline would cause all the reports to fail similarly to the password change use case mentioned above.

Eleven of the *Plant Coordinators* from seven different BMW, Mini and Rolls Royce Plants in four countries contacted me for support with the NAS migration using this tool. In a period of two weeks I rescheduled 8849 recurring instances for these departments, saving roughly 1100 hours of work from these colleagues, with estimated costs of around 100 thousand Euros. These statistics can be seen in table 4.

Since this script is still going to be used at least once a year when the passwords for all *Technical Users* of SAP BO are changed, it is expected that the similar gains to the ones shown in Table 4 are going to be obtained this frequently.

Responding to requests of the project's *stakeholders* (section 5.1), this tool was

Table 4 – Statistics of the `sap_reschedule` component during the NAS migration in 2019

Property	Amount
Schedules changed in PROD	8849
Departments benefited	11
BMW Group Plants	7
Countries	4
Work hours saved	1100
Estimated costs saved	100,000.00 Euros

Source: made by the author.

also modified to support changes of different schedule parameters as well, such as whether a successfully generated instance should be kept in the report's history, allowing them to reschedule instances in batch and standardize the report generation parameters the way they wish. The tool was used this way a handful of times as well.

6.4 The `sap_backup` component

The `sap_backup` component is meant to be able to backup schedules from a specific folder in SAP BO and to restore the schedules from a backup to the platform.

Since the SAP BO operations team in BMW Group has a backup functionality in all of the databases, this tool was not one of the initially planned use cases of this project. However, a series events happened and exposed a malfunction of one of this team's systems, which accidentally deleted schedules in the production environment of the German SAP BO Platform in BMW Group. When this problem happened, the operations team had no solution to it and it motivated us to develop this tool within our department. The sequence of events which caused the schedules to be accidentally deleted will be exposed in the next paragraphs.

Firstly, the rights to access the platform for the *Technical User* used to schedule all the reports in the Paint Shop in Plant Dingolfing expired, even though there was no expiration dates set. Secondly, because *Technical Users* have no e-mail address, the system did not generate any kind of notification to the responsible people.

This would normally not be considered a big problem because in the next production day someone would see that the reports were not been generated as expected, except that this problem happened during Plant Dingolfing's production break, which means that the report schedules were paused and the entire department was on vacation.

Sixteen days passed until the production break ended and the Quality IT team noticed that the schedules were deleted from the platform and opened a ticket for the IT department to rollback the instances from this folder. This was not possible, though, because the folder specific backups that they offer are deleted after fourteen days. Older

backups are only available for the entire platform, and since all the BMW Group Plants in Germany and Brazil use the same platform, a rollback of the entire platform would have bigger consequences and our only option, as it was decided, was to reschedule the 1700 report instances manually.

There were no plans of a backup tool within the scope of this project until then, but luckily this episode happened after we used the `sap_reschedule` tool to make the NAS migration described in Section 6.3. I had made a backup functionality to save the information of the old schedules to a file before deleting them from the platform in the `sap_reschedule` tool, in case something went wrong, and this backup could be used to recreate the deleted schedules.

For this reason the first script prototype related to the `sap_backup` tool was actually a script that allowed this backup made by the `sap_reschedule` tool to be used to restore the deleted report schedules to the platform. It was used to restore about 1700 schedules to the platform and saved around 198 hours of work as opposed to manually scheduling the reports.

This tool's prototype is not yet in a stage where it runs daily to make backups of multiple folders, although that is planned to be implemented in the near future.

7 Conclusions and Perspectives

The project was considered successful by the Quality IT departments that were involved and the direct manager. I have received many e-mails from the *stakeholders* (see Section 5.1) from the BMW Group Plants that I have helped during this internship with positive feedback. The final presentation of the project and its results that I have made for the management was also well received and responded with very positive feedback.

The first goal of this project was to build the foundations that would enable further development in the future and contain the knowledge acquired from documentation and reverse engineering. This was important for the department because they always hire students with software development skills but it is always a problem when they need a new student to continue the work of another that already left, which in most of the cases means that a new project is started from scratch.

Most of the functional and nonfunctional requirements from Sections 4.1.1 and 4.1.1 are related to the maintainability and extension of the project by other developers, as well as sharing of all the knowledge I have learned during this year with future team members after my internship ends. These requirements were fulfilled by applying the right design principles from Sections 3.2 and 3.3.1 and implementation described in Chapter 5.

One aspect that was important to overcome this problem was setting up the project management tools and processes. I was the first student in this department that proposed a project workflow like the one described in Section 5.2.1, asked to use Git, set up Jira Scrum boards and Bitbucket repositories. I proposed this workflow to be followed by a development team, and requested by my supervisor, wrote articles and tutorials about it for the next students that come here for internships.

Many problems of the SAP BO RESTful API that are not in the API documentation took a long time to solve in the framework.

The session management in the SAP BO RESTful API is a problem that was much more complex than expected. The API documentation (SAP, 2017a,b) explains that to receive a token, the client needs to send a *login* request and to release it, send a *logoff* request. The problems described in sessions 4.3.2, 4.3.3 and 4.3.4, though, were not documented anywhere in the API documentation and showed that session management is not as simple. These problems had to be reverse engineered, requiring tests design and execution, documentation and reporting to the SAP BO Operations team.

Another problem that is not documented is which requests require pagination of the response and how many items per page can each request type handle. This problem also

required many reverse engineering tests and inconclusive meetings with the responsible people in the SAP BO Operations team.

Both these problems were a huge obstacle for any work developed with the SAP BO RESTful API, however much thought and work was put into the framework in order to handle both of them the best way possible. Thanks to this, future developers working on the SAP BO Restful API in BMW Group can use the solutions provided and do not need to reinvent the wheel.

In terms of future perspectives, we have learned that when the SAP BO platform needs to be upgraded there are big changes in the RESTful API which may require changes in the framework. These changes are also unknown to the SAP BO team in BMW Group and not documented anywhere. For this reason, it is planned in future development of the framework to have enhanced integration tests that can show if the core functionality will still work after a platform version upgrade as well as show which functionality will require changes.

REFERENCES

ATLASSIAN CORPORATION. **Gitflow Workflow**. 2019. Available from: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>. Visited on: 10 Oct. 2019. Cit. on p. 61.

ATLASSIAN CORPORATION. **What is Git**. 2019. Available from: <https://www.atlassian.com/git/tutorials/what-is-git>. Visited on: 10 Oct. 2019. Cit. on p. 60.

BMW GROUP. **BMW Group Plant Dingolfing**. 2018. Available from: <https://www.bmwgroup-plants.com/dingolfing/en.html>. Visited on: 4 July 2019. Cit. on p. 25.

BMW GROUP. **History**: Difining Moments in The History of the BMW Group. 2019. Available from: <https://www.bmwgroup.com/en/company/history.html>. Visited on: 4 July 2019. Cit. on p. 25.

CHACON, Scott; STRAUB, Ben. **Pro Git**: Everything you need to know about Git. [S.l.]: Apress Berkely, 2014. ISBN 9781484200773. Available from: <https://git-scm.com/book/en/v2>. Cit. on p. 61.

CHRISTENSSON, Per. **Framework Definition**. 2013. Available from: <https://techterms.com/definition/framework>. Visited on: 10 Oct. 2019. Cit. on p. 43.

FIELDING, R. et al. **Hypertext Transfer Protocol (HTTP/1.1)**. The Internet Society. June 1999. Available from: <https://www.ietf.org/rfc/rfc2616.txt>. Visited on: 25 July 2019. Cit. on p. 41.

GIT SCM. **About Git**. 2019. Available from: <https://git-scm.com/about>. Visited on: 10 Oct. 2019. Cit. on p. 61.

GIT SCM. **git-submodule - Initialize, update or inspect submodules**. 2019. Available from: <https://git-scm.com/docs/git-submodule>. Visited on: 10 Oct. 2019. Cit. on p. 62.

HOHPE, G. et al. **Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions**. [S.l.]: Addison-Wesley, 2004. ISBN 9780321200686. Cit. on p. 40.

LISKOV, Barbara. Data abstraction and hierarchy. **SIGPLAN notices**, v. 23, n. 5, p. 17–34, 1988. Cit. on p. 34.

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Boston, MA: Prentice Hall, 2017. ISBN 978-0-13-449416-6. Cit. on pp. 33–36, 39.

MARTIN, Robert C. **Clean Code: A Handbook of Agile Software Craftsmanship**. 1. ed. [S.l.]: Prentice Hall, 2009. ISBN 978-0-13-235088-4. Cit. on p. 39.

MARTIN, Robert C. **The Clean Coder: A Code of Conduct for Professional Programmers**. 1. ed. [S.l.]: Prentice Hall, 2011. ISBN 978-0-13-708107-3. Cit. on p. 38.

MARTIN, Robert C. **The Cycles of TDD**. 2014. Available from: <https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html>. Visited on: 12 Nov. 2019. Cit. on p. 38.

MOZILLA DEVELOPER NETWORK. **HTTP**. 2019. Available from: <https://developer.mozilla.org/en-US/docs/Web/HTTP>. Visited on: 24 July 2019. Cit. on p. 40.

MOZILLA DEVELOPER NETWORK. **What is a web server?** 2019. Available from: <https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server>. Visited on: 24 July 2019. Cit. on p. 40.

PEKELMAN, Carol Jedwab. **REPORT OF SUPERVISED PROFESSIONAL PRACTICE**: UI and UX improvement of BMW's Quality Report dashboard environment. [S.l.], 2019. Cit. on p. 27.

PYTHON SOFTWARE FOUNDATION. **The Python Tutorial**. 2019. Available from: <https://docs.python.org/3/tutorial/index.html>. Visited on: 4 Sept. 2019. Cit. on p. 64.

RESTFULAPI.NET. **What is REST**. 2019. Available from: <https://restfulapi.net>. Visited on: 24 July 2019. Cit. on p. 41.

ROSSUM, Guido van; WARSAW, Barry; COGHLAN, Nick. **Style Guide for Python Code**. Python Software Foundation. 2001. Available from: <https://www.python.org/dev/peps/pep-0008/>. Visited on: 4 Sept. 2019. Cit. on p. 64.

SAP. **SAP BusinessObjects RESTful Web Service SDK User Guide for Web Intelligence and the BI Semantic Layer**. Version 4.2 Support Package 5 – 2017-12-15. 2017. Available from: <https://help.sap.com/doc/89ebd24bb71c4e82b47a44b7ad368bf5/4.2.5/en-US/webi42sp5_restful_web_service_sdk.pdf>. Visited on: 30 Aug. 2019. Cit. on pp. 28, 85.

SAP. **SAP Crystal Reports RESTful Web Services Developer Guide**. Version 4.2 Support Package 03 – 2017-05-12. 2017. Available from: <https://help.sap.com/doc/8f2e87893e944ec98a9c378aefdbbdd2/4.2.4/en-US/sbo42sp4_cr_restws_en.pdf>. Visited on: 30 Aug. 2019. Cit. on pp. 48, 85.

SILVA PRAZERES, Luiz Arthur d'Avila da. **Automation of the supervision process and analytic reports generation for the quality management of painted body shop production lines**. 2018. Undergraduate thesis – Universidade Federal de Santa Catarina. Cit. on p. 48.

SUTHERLAND, Jeff. **Scrum: The Art of Doing Twice the Work in Half the Time**. [S.l.]: Random House Business Books, 2015. ISBN 9781847941107. Cit. on p. 59.