Usability and maintainability of the software tools VIOLIN and CORAL

Bachelor Thesis

for the examination of

Bachelor of Science

of the study course Information Technology

at the Baden-Wuerttemberg Cooperative State University

by

Maikhanh Dang

August 2022

Time of Project	07.06.2022 - 30.08.2022
Student ID, Course	9572551, TINF19IT1
Company	Deutsches Zentrum für Luft- und Raumfahrt e.V.
Location	Berlin-Charlottenburg
Supervisor in the Company	M. Sc. Stephen Schade
Reviewer	Prof. Dr. Holger Gerhards

Author's declaration

Hereby I solemnly declare:

1. that this Bachelor Thesis, titled *Usability and maintainability of the software tools VIOLIN and CORAL* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. I have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Bachelor Thesis has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. I have not published this Bachelor Thesis in the past;
5. the printed version is equivalent to the submitted electronic one.

I am aware that a dishonest declaration will entail legal consequences.

Berlin, 29. August 2022

Maikhanh Dang

Abstract

The software tools **Vi**rtual Ac**o**ustic F**L**yover S**i**mulatio**n** (VIOLIN) and Air**C**raft Engine N**O**ise Au**RAL**ization (CORAL) have been developed and continuously expanded in the course of this bachelor's program. So far, the focus was on a clean software programming development as well as the content of the software. However, in the process, the software quality of the tools was less prioritized. In the scope of this thesis, various modifications are implemented and designed in interest of the quality characteristics usability and maintainability and evaluated quantitatively. For this purpose, the quality model of the ISO/IEC 25010 and the complementary metrics defined in the ISO/IEC 25023 are used. The evaluation shows an evident quality improvement for both software tools. However, in doing so, the quality measures of ISO/IEC 25023 are critically reviewed and their informative value are put into question.

Kurzfassung

Im Verlaufe dieses Bachelorstudiums wurden die Software-Tools VIOLIN and CORAL entwickelt und kontinuierlich erweitert. Hierbei lag der Fokus bisher auf der geschickten programmiertechnischen Umsetzung und auf den Inhalten der Tools, wobei der Aspekt der Softwarequalität in diesem Prozess vernachlässigt wurde. Im Rahmen dieser Bachelorarbeit werden Implementierungen zur Verbesserung der Qualitätseigenschaften Nutzbarkeit und Wartbarkeit umgesetzt und quantitativ bewertet. Zu diesem Zweck werden das Qualitätsmodell aus der ISO/IEC 25010 und die dazugehörigen Metriken aus der ISO/IEC 25023 verwendet. Die Evaluierung bestätigt eine erkennbare Verbesserung der Softwarequalität beider Tools. Allerdings werden dabei die Metriken kritisch reflektiert und die Aussagekraft dieser in Frage gestellt.

Contents

Acronyms

API	**a**pplication **p**rogramming **i**nterface
CD	**C**ontinuous **D**elivery
CD	**C**ontinuous **D**eployment
CI	**C**ontinuous **I**ntegration
CORAL	AirCraft Engine N**O**ise Au**RAL**ization
dB(A)	**A**-weighted **d**eci**b**el
DFT	**D**iscrete **F**ourier **T**ransform
DLR	**D**eutsches Zentrum für **L**uft und **R**aumfahrt
E2E	**e**nd-**to**-**e**nd
EPNL	**e**ffective **p**erceived **n**oise **l**evel
f-string	**f**ormatted **string**
GIGO	**g**arbage **i**n, **g**arbage **o**ut
GUI	**G**rahical **U**ser **I**nterface
HDF	**H**ierarchical **D**ata **F**ormat
HTML	**H**yper**T**ext **M**arkup **L**anguage
ICAO	**I**nternational **C**ivil **A**viation **O**rganization
ID	**id**entification
IOC	**I**nversion **o**f **C**ontrol
ISTFT	**I**nverse **S**hort-**T**ime **F**ourier **T**ransform
JSON	**J**ava**S**cript **O**bject **N**otation
PN	**P**rop**N**oise++
QME	**Q**uality **M**easure **E**lement
reST	**re**S**tructured**T**ext
RMS	**r**oot **m**ean **s**quare
SARPs	**S**tandards **A**nd **R**ecommended **P**ractices
SPL	**s**ound **p**ressure **l**evel
SQuaRE	**S**ystems and software **Qua**lity **R**equirements and **E**valuation
TOB	**t**hird **o**ctave **b**and
UI	**U**ser **I**nterface
UML	**U**nified **M**odeling **L**anguage
UN	**U**nited **N**ations
VIOLIN	**Vi**rtual Ac**o**ustic F**L**yover S**i**mulatio**n**

List of Figures

List of Tables

Glossary

For the purposes of this thesis, the following terms and definitions apply.

analyzability

degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified (adopted from ISO/IEC 25010:2011).

appropriateness recognizability

degree to which users can recognize whether a product or system is appropriate for their needs (adopted from ISO/IEC 25010:2011).

assertion

Boolean expression at a specific point in the software that should always evaluate to true.

atmospheric absorption

resonance absorption by air molecules.

auralization

process for artificially making an acoustic phenomena audible.

compatibility

degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment (adopted from ISO/IEC 25010:2011).

docstring

string literal to document a specific segment of code.

Doppler effect

time compression or expansion of a sound wave with changes in the distance between transmitter and receiver.

fake

testing double with working but usually simplified implementation using shortcuts, thus not suitable for production.

functional suitability

degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions (adopted from ISO/IEC 25010:2011).

golden file

expected output file of a software test.

golden master

last-known verified or acceptable output of a software.

introspection

ability to exam the type and properties of objects like classes and functions at runtime.

job

user-defined unit of work that is to be accomplished by a computer (adopted from ISO/IEC 25023:2016).

lambda function

anonymous function that is not bound to an identifier.

lazy evaluation

evaluation of an expression when needed.

learnability

degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use (adopted from ISO/IEC 25010:2011).

legacy code

still active old or outdated computer source code.

logging

automatic generation of a log of software processes.

maintainability

degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers (adopted from ISO/IEC 25010:2011).

measure

variable to which a value is assigned as the result of measurement (adopted from ISO/IEC 25023:2016).

measurement

set of operations having the object of determining a value of measure (adopted from ISO/IEC 25023:2016).

measurement function

algorithm or calculation performed to combine two or more quality measure elements (adopted from ISO/IEC 25023:2016).

mock

testing double substituting a real object by mimicking behavior.

modifiability

degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality (adopted from ISO/IEC 25010:2011).

modularity

degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components (adopted from ISO/IEC 25010:2011).

module

package of code and data for reuse.

operability

degree to which a product or system has attributes that make it easy to operate and control (adopted from ISO/IEC 25010:2011).

performance efficiency

performance relative to the amount of resources used under stated conditions (adopted from ISO/IEC 25010:2011).

portability

degree of effectiveness and efficiency with which a s system, product or component can be transferred from one hardware, software or other operational or usage environment to another (adopted from ISO/IEC 25010:2011).

quality characteristic

category of quality attributes that bears on software product or system quality (adopted from ISO/IEC 25023:2016).

quality measure

derived measure that is defined as a measurement function of two or more values of quality measure elements (adopted from ISO/IEC 25023:2016).

quality model

defined set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality (adopted from ISO/IEC 25023:2016).

reliability

degree to which a system, product or component performs specified functions under specified conditions for a specified period of time (adopted from ISO/IEC 25010:2011).

reusability

degree to which an asset can be used in more than one system, or in building other assets (adopted from ISO/IEC 25010:2011).

security

degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization (adopted from ISO/IEC 25010:2011).

software quality

degree to which a software product conforms to user specifications.

software stakeholder

person who has a stake in the software.

software testing

process of evaluating and verifying the software.

spectrogram

time-varying frequency spectrum.

strict evaluation

evaluation of function parameters before evaluation of function body.

stub

testing double containing predefined data and answers to function calls..

test coverage

percentage measure of the degree to which testing is performed by a set of tests.

test suite

collection of software test cases.

testability

degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met (adopted from ISO/IEC 25010:2011).

testing pyramid

visual metaphor for software testing standard.

tool

set of utilities or programs that help with the software developing process.

unit

small testable part of a software.

usability

degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use (adopted from ISO/IEC 25010:2011).

user error protection

degree to which a system protects users against making errors (adopted from ISO/IEC 25010:2011).

Foreword

The German Aerospace Center (**D**eutsches Zentrum für **L**uft und **R**aumfahrt (DLR)) is the research center of the Federal Republic of Germany for aerospace. However, the activities of the DLR reach far beyond this area. The DLR is engaged in a wide range of research in areas of energy and transport as well as security and digitalization. The extensive research and development work in these areas is conducted in national and international partnerships with other companies and research institutes. In addition to researching the earth and solar system, the DLR is also developing environmentally friendly technologies for the mobility, energy supply and security of the future. [1]

The DLR institute for Propulsion Technology focuses on the development of efficient and environmentally friendly aircraft propulsion systems and power plant turbines. Highly efficient methods and fast simulation processes are developed and applied in research. Additionally, demanding measuring methods and unique test facilities are used to investigate powerful and quiet propulsion concepts as well as environmentally friendly turbomachinery components and low-emission combustion chambers. [2]

The department for Engine Acoustics, a department of this institute, deals specifically with the acoustics of turbomachines and gas turbines, especially aircraft engines. The research activities of the department focus on fan noise. In particular, the noise generation, propagation and radiation are investigated. The noise levels are predicted either analytically, numerically or experimentally. The goal is to identify the noise sources and, above all, to reduce this noise. [3]

1. Introduction

As part of the dual bachelor's program in Computer Science of the Baden-Wuerttemberg Cooperative State University, two analytical software tools were developed in the practical phases at the German Aerospace Center (DLR): **Vi**rtual Ac**o**ustic F**L**yover S**i**mulatio**n** (VIOLIN) and Air**C**raft Engine N**O**ise Au**RAL**ization (CORAL). Both tools process data from an analytical noise prediction. The tool VIOLIN is a post-processing and noise immission tool. With VIOLIN, sound generated from an aircraft, e.g. fan or jet noise, can be propagated to one or more observer positions during a virtual flyover. The resulting sound levels are evaluated using various noise metrics. This is used for virtual certification in the design phase. For further evaluation, the resulting sound fields are auralized using the tool CORAL. Auralization is the artificial process to make an acoustic phenomena audible.

During the development phase, the focus was on the modular software design and structure as well as the content of the software. Both tools were continuously expanded by new acoustic modules. In doing so, the software quality was less prioritized. Quality assurance is a vital part of the software development cycle. This is because software quality encompasses desirable properties of a software that conform to the requirements placed on the software. It ensures that the software performs its intended functions without failure in a safe and fault-free manner. Therefore, the subject of this thesis is to improve the software quality of the tools VIOLIN and CORAL. The research is limited to the quality aspects around the software user. This includes usability and maintainability. To improve these software characteristics of VIOLIN and CORAL, the following approach is taken in this thesis:

To establish a theoretical basis, the terms ”software quality”, ”usability” and ”maintainability” are defined. For this purpose, existing definitions and various quality models that include those quality characteristics in the academic literature are compared and reviewed. A suitable quality model is chosen. In preparation of the quality improvement, the analyzed software tools are introduced. Based on the users of both tools, general user requirements to the software are identified. The user requirements are further specified with respect to the quality characteristics usability and maintainability as defined in the chosen model. On the basis of the specified requirements, appropriate modifications are proposed and implemented to improve the quality characteristics usability and maintainability of both tools. Lastly, the proposed implementations are evaluated quantitatively using the quality metrics of the chosen quality model.

2. Theoretical framework

This chapter provides an overview of relevant literature and current knowledge on the subject. It starts with the definition of software quality according to academic literature, followed by well-established software quality models that propose usability and maintainability as factors of software quality. Furthermore, core concepts of software testing and documentation are summarized in the following sections. These will be relevant for the modifications to improve the software quality.

2.1. The meaning of software quality

To consider and analyze specific software quality factors in detail, it is first necessary to understand what software quality means. Numerous definitions are proposed throughout the academic literature. In the compendium produced in a Ph. D. course on ”Quality attributes and trade-offs” [4], various perspectives of different authors and researches on (software) quality are explored in the first chapter by D. Milicic. Essentially, Milicic points out two views on the definition of (software) quality:

- Conformance to specification: Quality whose measurable characteristics satisfy fixed specifications defined in beforehand.
- Meeting customer needs: Quality as the capability to meet customer expectations.

To offer more insight on the two perspectives, Milicic gives six examples:

1. According to Crosby [5], quality must be defined as ”conformance to requirements”. It is important to define quality to be able to manage the concept of quality. Crosby is a strong advocate of quality as prevention and not as appraisal or inspection. In other words, quality assurance to satisfy specified requirements should be part of the developing process and not assessed in retrospect. Nonconformance is the absence of quality. This clearly adheres to the quality definition ”conformance to specifications”.
2. In contrast to Crosby, Feigenbaum's philosophy of quality [6] represents the view ”meeting customer needs”. Quality is based on the customer's experience with the product or system. Feigenbaum emphasizes the importance of satisfying the customer's actual and expected needs, stated or not stated.

3. Similarly to Feigenbaum, Ishikawa's definition [7] of quality complies with a "meeting customer needs" perspective. Ishikawa further interprets quality as a dynamic concept as customer needs and expectations change continuously. Consequently, quality must be defined dynamically. Meeting standards only is insufficient, even if frequently updated. The standards simply cannot keep up with the pace of customer needs.

4. A counter example is Deming's opinion [8] on quality: Quality specifications must be defined in consideration of future user needs. This is a much wider concept as Deming combines two perspectives "conformance to specifications" and "meeting customer needs". Deming points out the difficulty of translating future user needs into measurable characteristics. In his opinion, this requires a management system that enables responsibility of one's own work. For this purpose, he introduces 14 points or steps for management.

5. Also fitting into both philosophies is Shewhart's definition of quality from the 1920s [9]. Shewhart, who is referred to as "the master" by Deming due to his widely-accepted definition, identifies an objective and subjective aspect of quality: The first refers to the quality that is independent of the human factor and the latter describes the resulting thoughts and feelings about the objective quality. Shewhart's definition is one of the oldest but still deemed to be the most superior.

6. An alternative view is provided by Juran [10]. He defines quality as "fitness for use" and proposes three steps for managing quality: planning, control and improvement. This involves the identification of customers, requirements etc., the examination or evaluation of the product against the requirements and lastly the practice of methods to continuously sustain quality. According to Milicic, Juran's definition indicates references to the "conformance to specification" view, thus characterizing it as such more than "meeting a customer needs".

Within the scope of this thesis, the definition of software quality agrees with that of Deming's. In this context, quality also means the conformance to specifications which are defined in accordance to the user. The goal of this thesis is to improve software quality in consideration of user needs and typical application scenarios. From these aspects, requirements and specifications for the software tools are derived as well as necessary steps to achieve the goal. This clearly follows both definitions "conformance to specifications" and "meeting customer needs" similar to Deming's view on software quality. In this sense, it is also important to consider user needs that are relevant in the future as well.

2.2. Usability and maintainability in software Engineering

The terms ”usability” and ”maintainability” within the scope of software engineering are typically integral aspects of software quality. Many software quality models exist, thus providing a range of definitions for both terms. In this section, the following popular models that include usability and maintainability are presented:

- The Boehm model is described in Subsec. 2.2.1.
- The McCall model is described in Subsec. 2.2.2.
- The International Standard ISO/IEC 9126 is described in Subsec. 2.2.3.
- The International Standard ISO/IEC 25010 is described in Subsec. 2.2.4.

In Subsec. 2.2.5, the various definitions of usability and maintainability are reviewed by comparing their attributes of the presented models to determine the most suitable model and definition for both terms.

2.2.1. Boehm’s model

In 1976, Boehm et al. [11] first introduced a hierarchical model, later to be known as ”Boehm’s Software Quality Model” (see Fig. 2.1). It is one of the earliest models. In this model, maintainability is characterized as one of the three so-called ”primary uses”. The model suggests that the primary uses are necessary conditions for software quality and therefore considered as high level characteristics. Usability is referred to as ”human-engineering” [12][13] and is one of the mid level characteristics that are associated with the three primary uses. These are called ”intermediate constructs”. The primary use maintainability, for example, is decomposed into the characteristics understandability, modifiability and testability, and, according to Boehm, is aided by human-engineering. Intermediate constructs are further classified into so-called ”primitive constructs”. For example, human-engineering is broken down into the lower level characteristics communicativeness, accessibility and robustness/integrity.

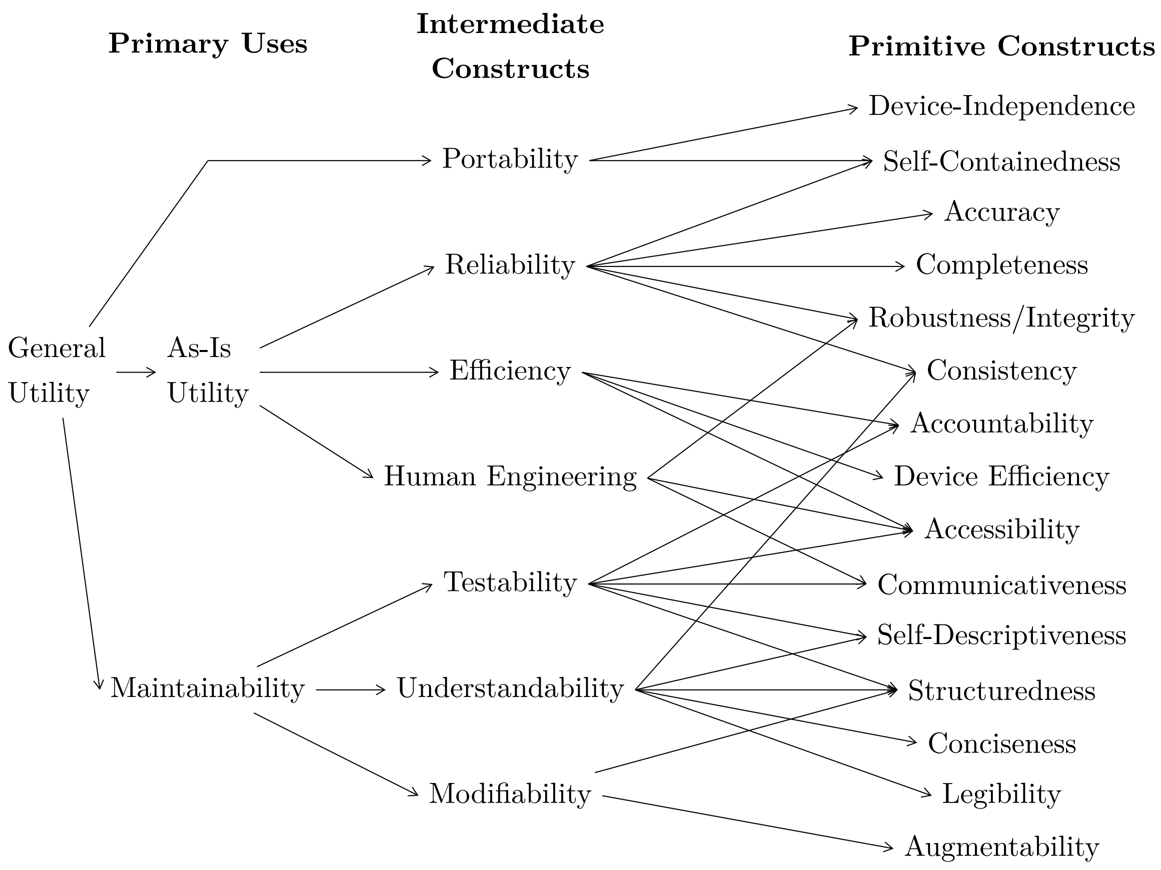


Figure 2.1.: Software Quality Characteristics Tree by Boehm [11]

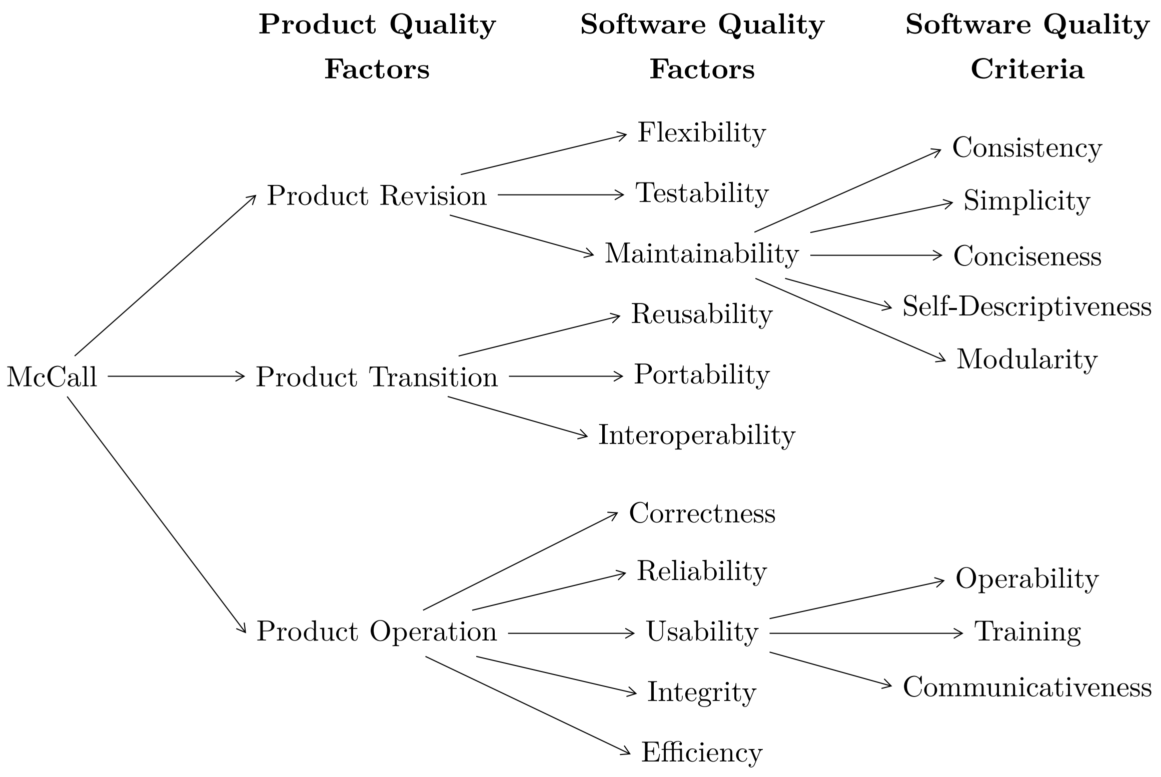


Figure 2.2.: Truncated software quality model of McCall [14]

2.2.2. McCall's model

McCall's model [14] was proposed in 1977 and represents similar to Boehm's model a hierarchical software quality model. It is based on three product quality factors:

- Product Operation is the requirement that affects the software operation providing better user experience.
- Product Revision is the requirement for software testing and maintenance.
- Product Transition is the requirement for software adaptation to new environments.

For each one of these, the model defines different software quality factors. They describe important external attributes and are considered higher level characteristics. Each quality factor further contains internal attributes called software quality criteria. In comparison to quality factors which can be accessed directly, the lower level quality criteria can be accessed subjectively or objectively. This model classifies maintainability and usability as quality factors. Figure 2.2 depicts a truncated model that only lists the internal attributes of usability and maintainability. McCall defines both as follows:

- Usability: "Effort required to learn, operate, prepare input, and interpret output of a program" [14]
- Maintainability: "Effort required to locate and fix an error in an operational program" [14]

2.2.3. The ISO/IEC 9126-1

The first part of ISO/IEC 9126 [15] describes a software product quality model. It is the revision of ISO/IEC 9126 (1991) and retains the same software quality characteristics. Software quality is broken down into the following six characteristics:: functionality, reliability, usability, efficiency, maintainability and portability. These are further broken down into normative subcharacteristics with measurable attributes. Appropriate metrics are proposed in the other parts of ISO/IEC 9126. This standard provides the following definitions for usability and maintainability:

- Usability: "The capability of the software product to be understood, learned, used and attractive to the user, when used under specific conditions." [15]
- Maintainability: "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in the requirements and functional specifications." [15]

2.2.4. The ISO/IEC 25010

The ISO/IEC 25010 [16] revises and replaces ISO/IEC 9126-1, incorporating the same software quality characteristics with some amendments. It is part of the **S**ystems and software **Qua**lity **R**equirements and **E**valuation (SQuaRE) series of International Standards and defines system and software quality models. The following definitions apply:

- Usability: ”degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use” [16]
- Maintainability: ”degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers” [16]

2.2.5. Literature review

The presented models offer different definitions of the terms usability and maintainability. However, when comparing the varying attributes of each quality factor, it becomes clear that all sources share a similar core concept of the terms. In Tab. 2.1 for usability and Tab. 2.2 for maintainability, these definitional attributes are summarized respectively. Each row lists an area of apparent agreement on the respective quality attributes. For example, all sources except for Boehm’s model define the term ”operability” as an attribute of usability (see Tab. 2.1). Not all sources use the same exact term for an area. In some cases, synonymous terms are used. Overall, it becomes noticeable that the ISO/IEC 25010 seemingly consolidates all attributes of the other presented sources in one model. This observation is examined in the following, starting off with usability.

Table 2.1.: Usability attributes of various quality models

Boehm (1976)	McCall (1977)	ISO 9126-1 (2001)	ISO 25010 (2011)
Communicativeness	Communicativeness	Understandability	Appropriateness recognizability
	Training	Learnability	Learnability
	Operability	Operability	Operability
Robustness			User error protection
		Attractiveness	User interface aesthetics
Accessibility			Accessibility
		Usability compliance	

The ISO/IEC 25010 is the revision of ISO/IEC 9126. Therefore, it is logical that attributes of the earlier model are encapsulated in the new one. According to ISO/IEC 25010, usability was an implicit quality and made explicit. For this purpose, the subcharacteristics understandability and attractiveness underwent a change of name to be more accurate (see Tab. 2.1). Two new subcharacteristics are introduced: user error protection (to achieve freedom of risks) and accessibility. Learnability and operability are adopted as is. Only the attribute usability compliance was not incorporated. However, it is redundant to include a requirement of the quality usability for a software product to adhere to standards, regulations etc. relating to usability. Boehm and McCall list three subcharacteristics each, sharing the attribute communicativeness. This criterion, as McCall refers to, is defined as the extent that software provides inputs and outputs that are useful and easy to assimilate. It can be considered a subset of the attribute appropriate recognizability and therefore is listed in this area. ISO/IEC 25010 defines this subcharacteristic as the "degree to which users can recognize whether a product or system is appropriate for their needs". Similarly, the criterion training by McCall is subsidiary to the area of learnability. A short but precise definition of the latter is given by ISO/IEC 9126: Learnability is the "capability of the software product to enable the user to learn its application". According to McCall, training refers to software attributes that provide initial familiarization, thus aiding the attribute learnability and characterizing it as such. The classification of the remaining attributes is self-explanatory. Robustness (Boehm) is synonymous with the subcharacteristic user error protection. The particular terms accessibility (Boehm) and operability (McCall) are both specified in the ISO/IEC 25010 standard.

In Tab. 2.2, the varying attributes of maintainability are depicted. The subcharacteristics analyzability and testability from ISO/IEC 9126 are adopted in the ISO/IEC 25010. Modifiability is a combination of changeability and stability. Both, testability and modifiability, can be found in Boehm's model as attributes of maintainability. The subcharacteristic analyzability, however, is only implied in the two earlier models. According to Boehm, the premise for any code maintenance is the understanding of the code. In a broader sense, understandability is also a logical subset of analyzability. McCall lists the criteria self-descriptiveness and simplicity which inherently describe the term understandability. Therefore, they can also be considered in the area of analyzability. The subcharacteristic maintenance compliance of ISO/IEC 9126 is redundant with the same reasoning given for the attribute usability compliance. Lastly, two new subcharacteristics are introduced in ISO/IEC 25010: reusability and modularity. The latter is represented in McCall's model and defined as the attributes that provide a structure of highly independent modules. Furthermore, McCall proposes the criterion conciseness which is "the ability to satisfy functional requirements with a minimum amount of software" according to himself. In this sense, conciseness belongs to the subcharacteristic modularity.

Table 2.2.: Maintainability attributes of various quality models

Boehm (1976)	McCall (1977)	ISO 9126-1 (2001)	ISO 25010 (2011)
	Modularity, Conciseness		Modularity
			Reusability
Understandability	Self-descriptiveness, Simplicity	Analyzability	Analyzability
Modifiability		Changeability, Stability	Modifiability
Testability		Testability	Testability
		Maintenance compliance	

The comparison shows that the quality model as defined by ISO/IEC 25010 is a well-rounded model, consolidating usability and maintainability attributes of prominent models. Further inspections also reveal this standard to propose suitable measurement functions for each characteristic and subcharacteristics. Boehm makes an attempt by defining a categorical rating scale. McCall essentially proposes a checklist, a binary measure determining the existence or absence of something, and a relative quantity measure. Additionally, McCall sets the following rule for the units of the metric: ”The units of the metric will be chosen as the ratio of actual occurrences to the possible number of occurrences” [14]. This shows similarities to the measurement functions of the International Standard. The SQuaRE series provides a detailed set of quality measures for a quantitative evaluation of the software quality factors defined in the ISO/IEC 25010. These are specified in the complementary standard ISO/IEC 25023 [17]. Further details are given in Sec. 3.4.

Based on this literature review, the quality model of ISO/IEC 25010 proves to be the most comprehensive model. Conclusively, the definitions of usability and maintainability according to this standard apply for the purpose of this thesis.

2.3. Software Testing

An integral step of the software development cycle is software testing. Following the design plan of a software, the actual implementation has to be tested before deployment. The software is evaluated and verified to ensure that all defined requirements are met. The purpose of software testing is to find missing requirements and catch bugs and errors. Software testing is also an important part of software maintenance. Regular testing ensures a high software quality and the longevity of a product.

2.3.1. The Testing Pyramid

There are different approaches to (automated) software testing. A popular one is the so-called ”Testing Pyramid”. The concept is believed to have been introduced by Mike Cohn in his book [18] published in 2009. The general idea of the pyramid has caught on since and multiple variants have been proposed. Figure 2.3 depicts the testing pyramid. The pyramid is a visual metaphor for thinking about software testing in different layers:

1. Unit Tests: The base of the pyramid comprises unit tests. Unit testing is a process where the smallest testable components of a software are isolated and scrutinized individually. A unit can be a function/method, a subroutine or a property for example. For the purpose of isolation, mock, stub or fake objects or so-called ”test doubles” are used.

2. Service Tests: Service tests are integration tests. In contrast to unit testing, integration testing appraises the individual components of a software as one combined entity. The objective is the evaluation of the compliance of a software or system with specified functional requirements. This is achieved by integrating the various components and testing the interfaces and interaction between the modules.

3. UI Tests: The top of the pyramid consists of **U**ser **I**nterface (UI) tests, also known as **e**nd-**to**-**e**nd (E2E) tests. Unlike the name entails, the tests are entirely automated and not conducted by human. Every user interaction is simulated.

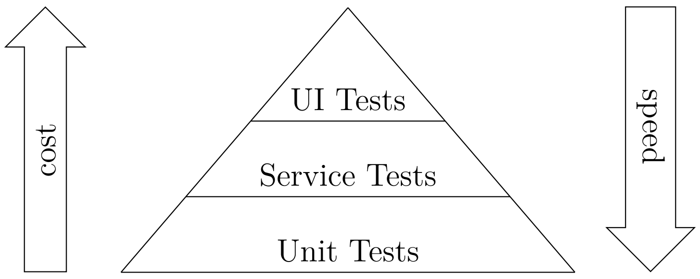


Figure 2.3.: The software testing pyramid

Each level of the pyramid covers a different scope. The integration of software components increases with each level, starting with the isolation of units at the bottom of the pyramid. However, more integration also means more time and consequently more expenses. Conversely, unit tests are fast and less expensive. For this reason, unit tests constitute the base of the pyramid, which is the widest part. The visual metaphor of the pyramid also specifies the amount of testing for each level in relation the other levels.

2.3.2. Golden Master Testing

Golden Master Testing [19] is a software testing method to protect legacy code from unintended changes via automated testing. It enables and provides a safety net for the extension and restructuring of code with insufficient or no adequate unit tests. This method is also known as a characterization test, a term coined by Mike Feathers [20].

In contrast to the typical approach of assertion-based software testing, the complex result of the tested software is validated against a reference outcome of a previous version of the software. The reference is referred to as the Golden Master. A golden master can be any type of output containing a challenging amount of data. For example, the characterization test can be used to verify an image or some type of data format file. In these cases, the simple comparison between a current output and a reference is far more cheaper and appropriate than the usual assertion of each individual value or property. The process of the Golden Master Testing is visualized in Fig. 2.4. A characterization test passes if the output and golden master match. Otherwise, the test fails. However, this does not ultimately mean that the code is wrong. On the contrary, this kind of test serves as a simple change detector. If a test fails, the developer has to check the results and decide whether the changes in the code need to be fixed or the golden master needs to be replaced. The latter is another advantage to this software testing technique. Software changes continuously. Therefore, the corresponding tests have to be easily adaptable. Golden Master Testing enables this by just updating the reference with the new acceptable output. Additionally, due to the fact that this technique is based on existing code, it is possible to automate the tests. This is an interesting aspect as a measure to uphold the software quality maintainability. A disadvantage is that characterization tests merely verify an observed behavior of a software. They cannot determine the correctness of the code. This is up to the developer who analyzes and evaluates the detected change. Furthermore, Golden Master Testing depends on repeatability. Characterization tests are not suitable for volatile or non-deterministic results. In this case, traditional assertion-based software testing is the better approach.

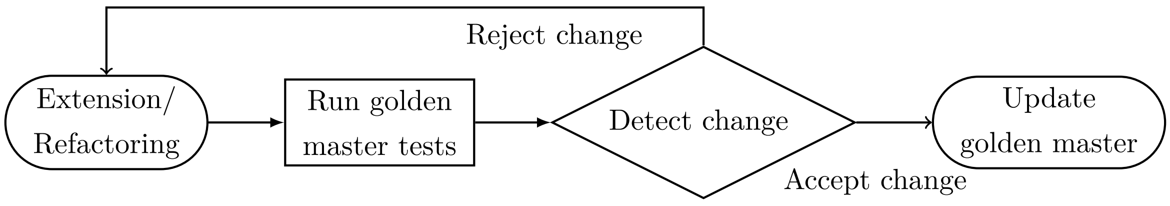


Figure 2.4.: Process steps of the Golden Master Testing

2.4. Software Documentation

Software documentation is an integral part of a software application. Forward [21] describes in his master's thesis: ”Documentation is an abstraction of knowledge about a software system [...]”. A document or any other artifact forms part of a software's documentation as long as it can effectively communicate knowledge. By artifacts, Forward includes, for example, software models and source code. To determine which attributes contribute to the effectiveness of a documentation, Forward conducts a survey of software professionals. On this topic, the results of the survey reveal the following:

- Content is the most important factor as well as the target audience. The target should be kept in mind to produce effective documentation. Furthermore, the chosen documentation technology must allow easy creation and maintenance of content-rich documents.
- Secondly, the degree to which a document is up-to-date is an important factor. However, it is even more important that up-to-date documentation is readily available and easily located. Up-to-date documentation that is not available to users is as useless as outdated documentation. Examples are great methods for effectiveness.
- Lastly, a document's file format or the quality of spelling and grammar has low correlation with the documentation's effectiveness.

Continuing the first revelation, academic literature on software engineering suggest multiple types of software documentation. They include but are not limited to the following:

- Requirements documentation describe the foundation for the implementation.
- Technical documentation include information on code, algorithms, interfaces etc.
- User documentation include operation manuals for the end-user.

Each type of documentation addresses a different target group. As Forward concludes, the target audience has to be kept in mind when choosing the type of documentation.

3. Methods and materials

For the realization and evaluation of this thesis, a range of different tools and methods is utilized. They are presented in this chapter as follows:

- In Sec. 3.1, the Requirements-Properties-Matrix is presented.
- In Sec. 3.2, the tool GitLab CI/CD for continuous methodologies is presented.
- In Sec. 3.3, various Python modules, frameworks and tools are presented.
- In Sec. 3.4, the quality measures for the quantitative evaluation are presented.

3.1. The Requirements-Properties Matrix

In an attempt to avoid ambiguity in the software requirements specifications, Boehm et al. [11] suggest a technique to explicitly analyze implicit quality requirements: the Requirements-Properties Matrix. The difficulty is, as Deming [8] also points out, the translation of qualitative user needs into measurable or quantitative characteristics. When defining the software requirements, the Requirements-Properties Matrix is supposed to help to identify additional and more feasible specifications in consideration of quality aspects. As shown in Tab. 3.1, it is a matrix whose rows consist of the desired qualities or properties and whose columns list the individual requirements. The elements of the matrix contain further specifications of an overall requirement and are less ambiguous. As an example, Boehm et al. [11] use this technique on the following requirement: ”Terminate the simulation at an appropriate shift break”. This specification leaves room for interpretation. It is ambiguous and non-testable. Table 3.1 depicts the corresponding exemplary Requirements-Properties Matrix. The resulting specifications define clear instructions and conditions for the initial software quality requirement, differentiating between the specific quality factors. This way, it is easy to evaluate if a software requirement is satisfied. In case of the given example, an appropriate shift break for termination is further specified as a time span of eight hours for testability. Allowing the user to set the termination time as an input parameter accounts for modifiability of the software. Lastly, an alternate termination condition in case of an exception is a requirement for robustness. The Requirements-Properties Matrix shows to be a reasonable technique to determine precise requirements from qualitative user needs.

Table 3.1.: Portion of a Requirements-Properties Matrix adopted from Boehm [11]

Requirement / Property	Terminate the simulation at an appropriate shift break	. . .
Testability	Terminate the simulation after 8 hours of simulated time	. . .
Modifiability	Allow user to specify termination time as an input parameter, with a default value of 8 hours	. . .
Robustness	Provide an alternate termination condition in case the time criterion cannot be reached	. . .
. . .	. . .	. . .

3.2. GitLab CI/CD

GitLab is an open-source software development platform. Aside from built-in version control and issue tracking, GitLab additionally offers a tool for continuous methodologies. This tool is called GitLab CI/CD [22]. **C**ontinuous **I**ntegration (CI), **C**ontinuous **D**elivery (CD) and **C**ontinuous **D**eployment (CD) are popular practices to catch software bugs and errors early in the development cycle. The fundamental element of the mentioned methodologies is CI. It is the practice to build and test each submitted change automatically and continuously. In addition to this, **C**ontinuous **D**elivery manually triggers the deployment of the changes. **C**ontinuous **D**eployment does so automatically. GitLab CI/CD can automatically build, test, deploy and monitor a software project, complying to code standards that the user defined.

In the configuration file `.gitlab-ci.yml`, the user defines the pipeline containing specific instructions. It is a top-level component of CI/CD. A pipeline comprises jobs and stages. Jobs define what to do, whereas stages define when to run the jobs. A job must at least contain the `script` clause with one command. The open-source application GitLab Runner works with GitLab CI/CD and executes the jobs automatically. A user can register and configure own runners or use runners hosted by GitLab. Typically, multiple jobs of the same stage are executed concurrently if enough runners exist. If all jobs in a stage succeed, GitLab continues with the next stage. Otherwise, the pipeline is halted and fails. Jobs can output an archive of files. The output is called job artifact.

3.3. Python modules, frameworks and tools

The programming language Python offers a variety of built-in library modules, frameworks and external tools. Modules contain separate resources with different functionalities. A module is a package of code and data for reuse including a definition of Python objects like classes, functions, variables and constants. The term tool is often used synonymously with the term module. However, in this context, a tool describes a set of utilities or programs that help a developer with the developing process of a software application. A framework is a set of libraries providing a program architecture. Unlike a library module, a framework is not called to access reusable code and included to a software application as a developer chooses to. Source code is integrated into the framework instead. This is the defining characteristic of a framework: **I**nversion **o**f **C**ontrol (IOC) [23]. Instead of the application controlling the flow of control and using only standard functions, control of the execution of certain subroutines is handed over to a framework. The flow of control and data is managed by the framework.

The following Python frameworks, libraries or tools are relevant to this thesis:

- The `logging` module is introduced in Subsec. 3.3.1.
- The framework `unittest` is introduced in Subsec. 3.3.2.
- The tool `pdoc` is used to auto-generate **a**pplication **p**rogramming **i**nterface (API) documentation from inline documentation and is introduced in Subsec. 3.3.3.
- The tool `pyreverse` is used to auto-generate **U**nified **M**odeling **L**anguage (UML) diagrams from source code and is introduced in Subsec. 3.3.4.

3.3.1. The module "logging"

Logging is a monitoring method for software applications or systems. It allows to understand the behaviour of a software and to find the problems within by simply tracking events. For this purpose, Python offers the standard library module `logging` [24]. The module contains classes and functions to implement a flexible logging system. An event is described with a descriptive message. Optionally, that message can contain a variable. Each event is assigned a level or severity of importance. All standard levels and their applicability are noted in Tab. 3.2. The default severity is `WARNING`. That means all events at this level or above will be tracked, unless configured otherwise. In other words, the levels `INFO` and `DEBUG` are not tracked by default. Besides that, it is possible to define custom levels. Complementary to the standard levels, the library offers a set of homonymous functions:

- `debug()` to report events in detail for diagnostic purposes.
- `info()` to report events during normal operation for status monitoring.
- `warning()` to issue a warning regarding a runtime event.
- `error()` and `critical()` to report the suppression of an error without raising an exception. The functions are not suitable to actually report an event error.

This logging module is also not suitable for mere console output of ordinary usage. For this purpose, a standard `print()`-statement is the best option.

Table 3.2.: The standard levels or severity of events and their applicability adopted from [24] (in increasing order of severity)

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

3.3.2. The framework "unittest"

Python's `unittest` [25] is a built-in framework including a test runner to test Python source code. As explained in Subsec. 2.3.1, the smallest testable components of a software are scrutinized during unit testing. This is an assertion-based software testing technique. Logically, `unittest` offers a range of assert methods (see appendix Tab. B.1). According to the official documentation, the framework supports "test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework". It is a convenient built-in framework providing a rich set of tools for unit testing without using third-party modules.

3.3.3. The tool "pdoc"

The software package `pdoc` [26] is an external Python tool to auto-generate API documentation from a Python's module hierarchy. By means of introspection, `pdoc` extracts information from inline source code documentation in form of docstrings. While plain Markdown docstrings are preferred, `pdoc` also understands and supports `numpydoc`, Google-style and **re**Structured**T**ext (reST) docstring. The acquired information is rendered and automatically transformed into **H**yper**T**ext **M**arkup **L**anguage (HTML) documentation. A built-in web server with live reloading is provided for the result. Alternatively, the documentation can also be saved as a HTML file. The tool `pdoc` only extracts API documentation of public code objects like modules, submodules, classes, functions and all kind of variables. Conversely, objects marked with an underscore are ignored unless configured otherwise. An example for the command-line application of the tool is given in Listing 3.1.

```
pdoc [-o DIR] [-d {markdown,google,numpy,restructuredtext}]
```

Listing 3.1: Exemplary usage of `pdoc`'s command-line application

3.3.4. The tool "pyreverse"

The Pylint tool suite provides the tool `pyreverse` [27]. It is an external tool to generate UML diagrams from Python source code, using the open source graph visualization software Graphviz as backend. All modules and classes of a user-specified location are analyzed. The resulting diagrams include:

- Class and instance attributes (if possible with their type) and methods
- Inheritance links between classes
- Association links between classes
- Representation of exceptions and interfaces

Similarly to `pdoc`, the tool `pyreverse` only considers public code objects unless configured otherwise when using the command-line application. An example is given in Listing 3.2. The optional argument `--filter-mode` filters attributes and functions according to the specified mode. Mode `'ALL'` filters nothing, thus including private members.

```
pyreverse [-o FORMAT] [--filter-mode/-f MODE]
```

Listing 3.2: Exemplary usage of `pyreverse`'s command-line application

3.4. Software product quality measures of the ISO/IEC 25023 standard

The standard ISO/IEC 25023 [17] of the SQuaRE series defines a set of quality measures for the quality characteristics and subcharacteristics defined in ISO/IEC 25010 (see Subsec. 2.2.4) and is intended to be used in conjunction with ISO/IEC 25010. However, the standard does not assign ranges of values of the measures to grades or rated levels of conformance. The main use of ISO/IEC 25023 is rather quality assurance and improvement of software products during and after the development process. The majority of the quality measures produce a result that is relative to a target value that is established as a requirement. Each is expressed as a measurement function.

This section presents the measures relevant to this thesis and is structured as follows:

- Subsection 3.4.1 depicts an obligatory set of conditions for quality conformance.
- Subsection 3.4.2 depicts the general documenting format of the quality measures.
- Subsection 3.4.3 depicts the quality measures of usability.
- Subsection 3.4.4 depicts the quality measures of maintainability.

3.4.1. Conformance conditions

According to the ISO/IEC 25023 standard [17], any quality requirement specification or quality evaluation that conforms to this standard ought to follow a specific set of rules. These are summarized in the following:

1. The quality characteristics or subcharacteristics to be specified or evaluated are selected as defined in ISO/IEC 25010.
2. A distinction is made between so-called ”Generic (G)” and ”Specific (S)” quality measures. For each selected characteristic or subcharacteristic, all Generic measures must be used. If any Generic measures are excluded, a rationale has to be provided. Specific measures are optional and selected when relevant only. In the scope of this thesis, the selection is mostly limited to Generic measures due to time constraints.
3. If any quality measure is modified, the reason for changes has to be provided.
4. Additional quality measures and **Q**uality **M**easure **E**lements (QMEs) can be defined if not included in ISO/IEC 25023.

3.4.2. Identification code of the quality measures

Each quality measure is given a unique **id**entification (ID) code which consists of the following information:

- An abbreviated alphabetic code representing the quality characteristics as a capital letter and the subcharacteristics as one capital letter followed by a lowercase letter
- A serial number of sequential order within a quality subcharacteristic
- A ”G” or ”S” to denote Generic or Specific measures

For example, the ID code ”UOp-7-S” describes the seventh (Specific) measure of the subcharacteristic ”Operability” for the quality characteristic ”Usability”.

3.4.3. Usability measures

This subsection lists the measures for the quality characteristic ”usability” that are relevant to this thesis. Usability measures are used to evaluate the ”degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use” [16]. Usability is composed of six subcharacteristics: appropriateness recognizability, learnability, operability, user error protection, user interface aesthetics and accessibility. However, the last two subcharacteristics are not relevant to this thesis with the following rationale:

- Firstly, to be able to assess appearance aesthetics, a **G**rahical **U**ser **I**nterface (GUI) is required. This does not apply to the analyzed software tools of this project. Besides, the one and only measure of the subcharacteristic user interface aesthetics is a Specific measure anyway.
- Secondly, the users of the software tools consist of a small, closed group of scientists. While inclusiveness is and remains an important topic, the consideration of users with disabilities[1] is not necessary for this scale of a project. Therefore, the subcharacteristic accessibility is excluded as well.

Consequently, four subcharacteristics are considered in this thesis. Table 3.3 lists all Generic measures of the selected subcharacteristics plus one additional Specific measure. The measures are grouped by the corresponding subcharacteristics. For each quality measure, the unique ID and name are given. The respective measurement functions are depicted in Appendix Sec. A.1.

[1] Disabilities include cognitive, physical, hearing/voice and visual disabilities.

Table 3.3.: Selection of usability measures proposed in ISO/IEC 25023 [17]

Subcharacteristic	ID	Name
Appropriateness recognizability	UAp-1-G	Description completeness
Learnability	ULe-1-G	User guidance completeness
	ULe-3-S	Error messages understandability
Operability	UOp-1-G	Operational consistency
	UOp-2-G	Message clarity
User error protection	UEp-1-G	Avoidance of user operation error

The selected measures shown in Tab. 3.3 are defined according to the standard as follows:

- Description completeness is the proportion of usage scenarios described in the product description or user documents relative to the actual number of usage scenarios.
- User guidance completeness is the proportion of functions explained in sufficient detail in user documentation and/or help facility[1] in relation to the number of functions implemented that require to be documented.
- Error messages understandability is the proportion of the error messages stating the cause and a solution in relation to the number of error messages implemented.
- Operational consistency is the extent of interactive tasks with a behavior and appearance that is consistent both within the task and across similar tasks in relation to the number of specific interactive tasks that need to be consistent.
- Message clarity is the proportion of messages[2] from a system conveying the right outcome or instructions to the user in relation to the number of messages implemented.
- Avoidance of user operation error is the proportion of user actions and inputs protected against causing any system malfunction in relation to the number of user actions and inputs that could be protected from causing any system malfunction.[3]

[1] e.g. on-line help, operational guide video, operational instruction system

[2] "Messages that provide all available information that could help the user, and when possible explain how to resolve the error." [17]

[3] It is helpful to find erroneous user actions and inputs for a better measurement. Possible measures against user operation errors include system confirmation requests before carrying out actions with significant consequences that cannot be undone.

3.4.4. Maintainability measures

This subsection lists the measures for the quality characteristic ”maintainability” that are relevant to this thesis. Maintainability measures are used to evaluate the ”degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers” [16]. Maintainability comprises five subcharacteristics: modularity, reusability, analyzability, modifiability and testability. In a similar format to Tab. 3.3, Tab. 3.4 lists all Generic measures for maintainability.

Table 3.4.: Selection of maintainability measures proposed in ISO/IEC 25023 [17]

Subcharacteristic	ID	Measure
Modularity	MMo-1-G	Coupling of components
Reusability	MRe-1-G	Reusability of assets
Analyzability	MAn-1-G	System log completeness
Modifiability	MMd-1-G	Modification efficiency
	MMd-2-G	Modification correctness
Testability	MTe-1-G	Test function completeness

The measures shown in Tab. 3.4 are defined according to the standard as follows:

- Coupling of components is the degree of independence between between components and the number of implemented components with no impact from change on others in relation to the number of specified components which are required to be independent.
- Reusability of assets[1] is the number of reusable assets in a system in relation to the actual number of assets which are designed and implemented.
- System log completeness is the extent of system logs to trace operations in relation to the number of logs for which audit trails are required during operation.
- Modification efficiency is the required time compared to the expected time.
- Modification correctness is the proportion of correctly implemented modifications in relation to the number of modifications implemented.
- Test function completeness the number of implemented test functions in relation to the number of required test functions.

[1] e.g. source code modules, testing modules, hardware, requirements documents

4. The software process chain

In this thesis, the following software tools are subject to a software quality analysis:

- **Vi**rtual Ac**o**ustic F**L**yover S**i**mulatio**n** (VIOLIN)
- Air**C**raft Engine N**O**ise Au**RAL**ization (CORAL)

VIOLIN provides a virtual acoustic flyover simulation and noise assessment based on various noise metrics. CORAL transforms the simulated noise into audible files. This process of making an artificial spectrogram audible is called auralization. Both tools establish a consecutive process chain with **P**rop**N**oise++ (PN) and HEIDI. These are software tools for the analytical prediction of engine noise and provide the input data for VIOLIN. The overall process chain is visualized in Fig. 4.1.

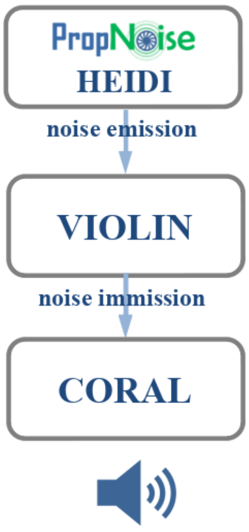


Figure 4.1.: Flowchart of the overall process chain

For better insight, the current state of the software tools and the overall process chain are introduced in this chapter. In Sec. 4.1 the tool VIOLIN is briefly summarized and the tool CORAL in Sec. 4.2. A more detailed description of both tools is given in this project report [28].

4.1. The flyover tool ”VIOLIN”

Virtual Ac**o**ustic **FL**yover **Si**mulatio**n** (VIOLIN) is a post-processing and sound immission tool where the sound field generated at the noise source is virtually propagated to an observer during a pre-defined flyover event. An illustrative example is given in Fig. 4.2.

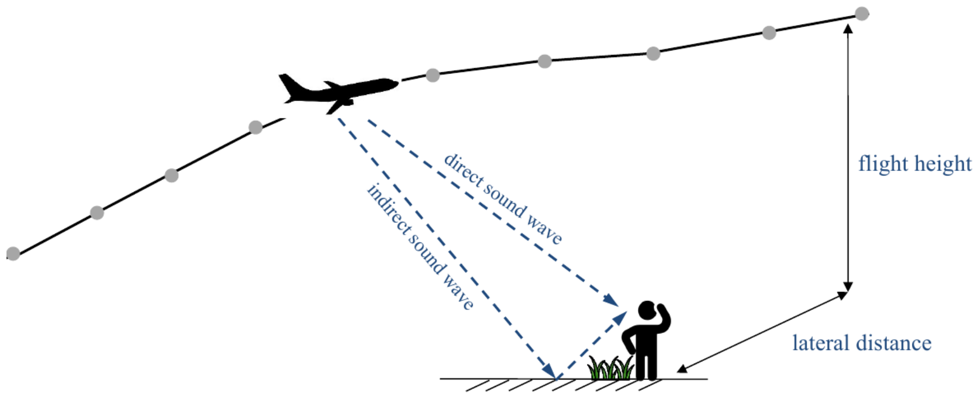


Figure 4.2.: 3D-Illustration of an exemplary flyover event in VIOLIN

The generated noise field is provided by one of the emission tools PN or HEIDI. Frequency- and angle-dependent **s**ound **p**ressure **l**evels (SPLs) or so-called sound directivities are predicted for several noise sources. VIOLIN processes this information as specified by the **S**tandards **A**nd **R**ecommended **P**ractices (SARPs) of the **I**nternational **C**ivil **A**viation **O**rganization (ICAO). The ICAO is a specialized agency of the **U**nited **N**ations (UN) with the goal of promoting the sustainability of the global civil aviation system and standardizing this system. During the sound propagation, various physical effects are considered. These include the Doppler effect, the atmospheric absorption and the ground attenuation or reflection. In Fig. 4.2, the indirect sound wave represents the sound wave reflected on the ground. In addition, this sound wave might also be attenuated by the ground. Both types of sound waves, direct and indirect, experience the Doppler frequency shift and an attenuation due to atmospheric absorption. The resulting frequency- and time-dependent SPLs reach the observer and represent the received spectrogram. Lastly, the received spectrogram is assessed by use of various metrics like the annoyance-based **e**ffective **p**erceived **n**oise **l**evel (EPNL) or the loudness-based **A**-weighted **d**eci**b**el (dB(A))[1]. Figure 4.3 depicts the described process chain. Next to the input provided by PN or HEIDI, the user defines the flight path and other settings like the time discretization at the beginning of the process.

[1] unit of measurement of the sound pressure level according to the internationally standardized frequency weighting curve A

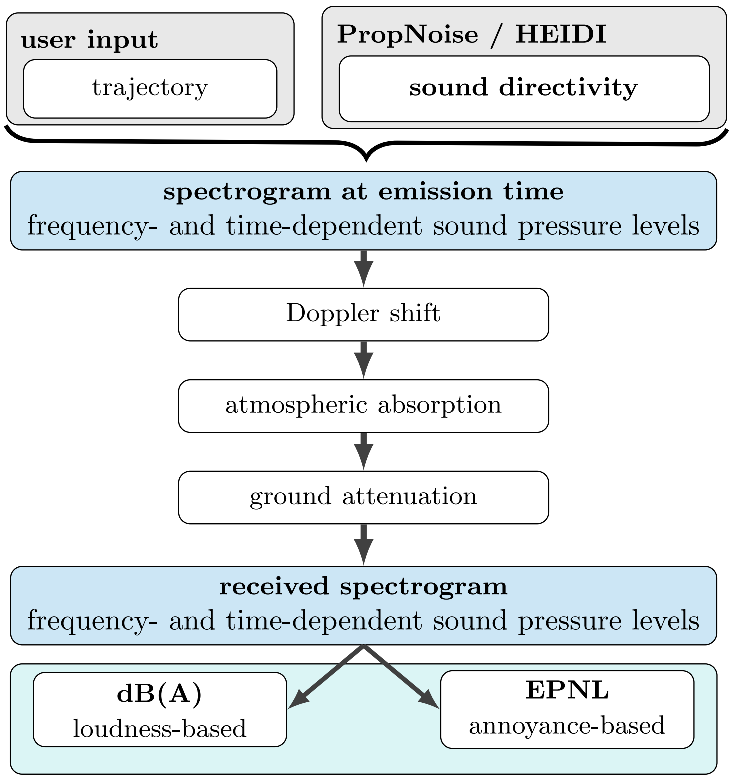


Figure 4.3.: Calculation steps of the software tool VIOLIN

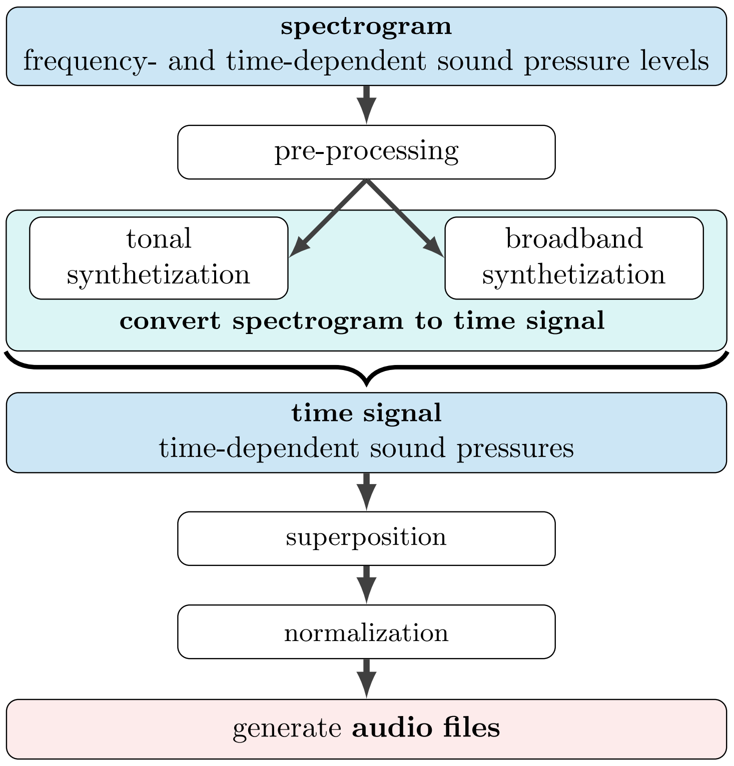


Figure 4.4.: Calculation steps of the software tool CORAL

4.2. The auralization tool "CORAL"

Air**C**raft Engine N**O**ise Au**RAL**ization (CORAL) was developed as a post-processing tool of VIOLIN to make the predicted flyover immission noise audible. For this purpose, CORAL converts the frequency- and time-dependent SPLs or received spectrogram into a time signal of sound pressures. Figure 4.4 depicts the overall auralization process.

The auralization process can be divided into three steps:

1. Pre-processing of the input
2. Conversion of the spectrograms to time signals
3. Generation of audio files

The first step is the pre-processing of the spectrogram input. In this part, the spectrograms are resampled in preparation for the conversion to time signals if necessary. The frequency and time values are adjusted. Following this step, the resampled spectrograms are converted into time signals. This is the main part of the auralization process. CORAL distinguishes between noise sources. Both noise source types provide a different spectrogram input and thus require differing transformation routines:

- In case of a tonal source, the input is a spectrogram of complex values (unit: Pa) with phase information. To generate the corresponding time signal, the tonal spectrogram is simply split into frequency batches. For each frequency chunk, a signal is synthesized by modulating a sine function. The sum of the individually synthesized signals equals the overall tonal time signal.

- In case of a broadband source, the input is a spectrogram of real values (unit: Pa^2/Hz) without phase information owed to the stochastic nature of broadband noise. The broadband time signal is generated by means of the **I**nverse **S**hort-**T**ime **F**ourier **T**ransform (ISTFT). However, due to lack of information about the phase, the phase relation has to be reconstructed first. The complete process is depicted in Fig. 4.5. For the reconstruction, it is necessary to generate a random white noise of the same time domain as the broadband source. In other words, a time signal of the same length with random values of uniform distribution is to be generated. By computing the **D**iscrete **F**ourier **T**ransform (DFT) of the white noise, the phase relation is attained from the complex spectrum of the white noise. The transformed white noise is then filtered with the broadband spectrogram. This way, the phase information which is missing in the input spectrogram is reconstructed. Finally, the ISTFT of the spectrogram can be computed and the broadband time signal is synthesized.

The conversion of a tonal or broadband spectrogram results in a time signal of real values (unit: Pa). In the next step, the time signal is made audible.

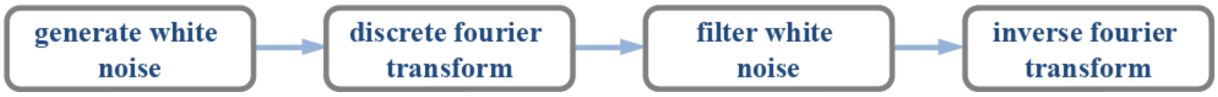


Figure 4.5.: Steps to synthesize a broadband time signal in CORAL

Before writing the resulting time signal to an audio file, the values of the time signal are normalized in reference to the maximum value. In addition to the time signals derived from the input, it is interesting to inspect the superposition between the noise sources. With the superposition, three more time signals are auralized:

- The sum of all tonal time signals
- The sum of all broadband time signals
- The superposition of all noise sources

These are normalized before the generation of respective audio files as well.

5. Definition of requirements

Software quality means conformance to user specifications. This requires the definition of software requirements and is the subject of this chapter:

- Section 5.1 defines the general user needs for the software tools. In this regard, a user profile is created. The assessment of the user profile enables a more funded insight into the user needs and requirements.
- Section 5.2 defines the typical application scenarios of both software tools. The scenarios are relevant to derive specific use cases for the software as well as further requirements.
- Section 5.3 uses the Requirements-Properties Matrix on the requirements identified in Sec. 5.1 and Sec. 5.2 and brings everything together in a clear manner.

5.1. User profile and needs

The software tools VIOLIN and CORAL are used by scientists of the group for analytical and numerical noise prediction of aircraft engine noise. The tools are operated separately as independent programs or consecutively in a process chain for different research purposes. However, in addition to purely using the software, the scientists also improve and work on the software like the software developers. The developers comprise another user group of the tools. For this reason, the term ”user” is used in a broader sense. The *SWEBOK: Guide to the Software Engineering Body of Knowledge* [29] refers to this as the ”software stakeholder”. Software stakeholders include but are not limited to the following:

- Users: This group comprises those who operate the software.
- Customers: This group comprises those who commissioned the software.
- Developer: This group comprises those who develop the software.

Using this terminology, the scientists of the group represent all three stakeholder groups. For simplification, the scientists and developers are referred to as users of the software who operate and develop the software in the scope of this thesis. With these responsibilities in mind, the following general requirements are identified in consultation with the users:

1. The software offers different (optional) features for the various stakeholders to choose from aside from the main application. A scientist has different requirements to the software as a user than as a developer. Additionally, the scientist as a developer may have slightly different needs than the actual software developer. The different user requirements to the software have to be considered. A useful strategy is the implementation of different execution modes accustomed to the various groups of users. In the scope of this thesis, such will be referred to us application modes or just modes in short. The term ”mode” is not to be understood as the acoustic mode in this context.

2. The software tools have to be intuitive for easy operation. This is achieved by standardization within each tool and among the tools. Standardization is the foundation for a common knowledge and code base. By using this base, faster implementation and better collaboration are enabled. Structured methods and reliable data are greatly beneficial for the innovation process as well as software operation, development and maintenance.

3. Generally important but especially relevant for maintainability is software testing. Evaluating and verifying the results of a software application prevents bugs and assures that the simulation results are correct. Normally, software testing is of no interest to the classic user and thus only affects the software quality maintainability. Maintainability, in fact, contains the subcharacteristic testability[1]. However, as concluded in the assessment of the user profile, the user in the context of this thesis refers to a group who operates and develops the software. Therefore, software testing indirectly becomes of relevance to the software quality usability.

4. All the requirements above imply the operation of execution scripts. Different application modes require an execution method. An intuitive and easy operation calls for a simple execution method but also with standardization within and among the software tools in mind. The same applies to the execution of software testing. Therefore, execution scripts, preferably Bash scripts for simplicity, are an indirect requirement.

5. The software tools must implement easily adjustable input parameters for a more flexible program workflow. In this regard, the following principle must be remembered: ”**g**arbage **i**n, **g**arbage **o**ut (GIGO)”. GIGO describes that nonsense input produces nonsense output. In other words, it is important to determine which input is relevant and protect such input from user operation errors.

[1] according to ISO/IEC 25010 [16]

5.2. Typical application scenarios

An application scenario describes a certain situation or manner a user interacts with VIOLIN and/or CORAL by means of a realistic example. It represents a use case of the software tool/s, which is frequently applied by the scientists during research. Application scenarios can have varying values of input and go through individual program workflows. Furthermore, the scenarios serve as categories of additional test cases for both tools individually or in a process chain. The following six application scenarios are defined[1]:

- Scenario 1: Auralization of a flyover event with PN, presented in Subsec. 5.2.1
- Scenario 2: Auralization of a flyover event with HEIDI, presented in Subsec. 5.2.2
- Scenario 3: Developer, presented in Subsec. 5.2.3
- Scenario 4: Auralization of sound emitted from distributed engines, presented in Subsec. 5.2.4
- Scenario 5: Certification and community noise, presented in Subsec. 5.2.5
- Scenario 6: Auralization of static engine tests without flyover, presented in Subsec. 5.2.6

Section 5.2.7 presents the varying input options for each test case of a scenario.

5.2.1. Auralization of flyover with PN

This application scenario includes the complete process chain as depicted in Fig. 4.1, starting with **P**rop**N**oise++ (PN). The emission tool PN provides sound directivities for VIOLIN with the following attributes for the different source types:

- Tonal sources produce complex, zero-to-peak scaled sound pressure amplitudes.
- Broadband sources produce real[2], RMS[3]-scaled power spectral density values.

VIOLIN uses this input and performs a virtual acoustic flyover simulation with subsequent noise assessment. The resulting noise immission is auralized by CORAL. Figure 5.1 visualizes this application scenario.

[1] In preparation for this thesis, the application scenarios have been identified in this project report [30].
[2] no phase information
[3] **r**oot **m**ean **s**quare (RMS)

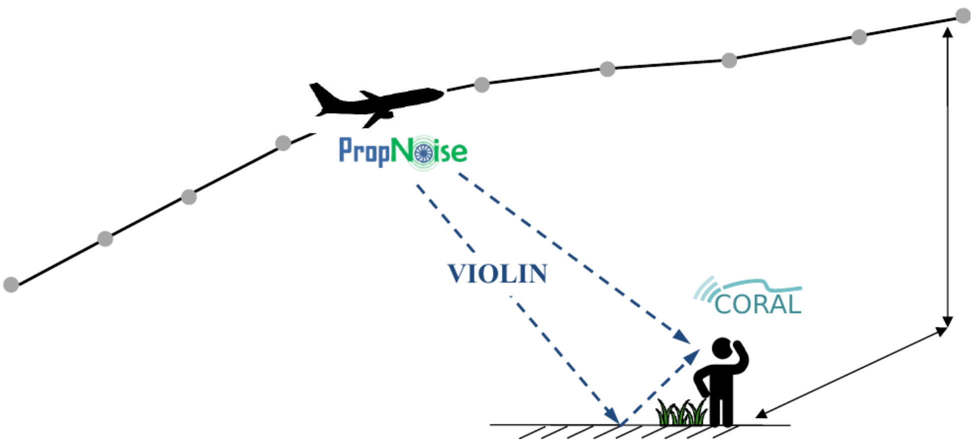


Figure 5.1.: Illustration of the application scenario ”Auralization of flyover with PN”

5.2.2. Auralization of flyover with HEIDI

The other version of the complete process chain, as shown in Fig. 4.1 starting with HEIDI, is covered in this scenario. The emission tool HEIDI provides sound directivities for VIOLIN with the following attributes for the different source types:

- Tonal sources produce **s**ound **p**ressure **l**evels (SPLs).
- Broadband sources produce SPLs in the **t**hird **o**ctave **b**and (TOB).

The input from HEIDI goes through the same process as described in Subsec. 5.2.1. Thus, sound immission from a flyover simulation with HEIDI is auralized. Figure 5.2 visualizes this application scenario.

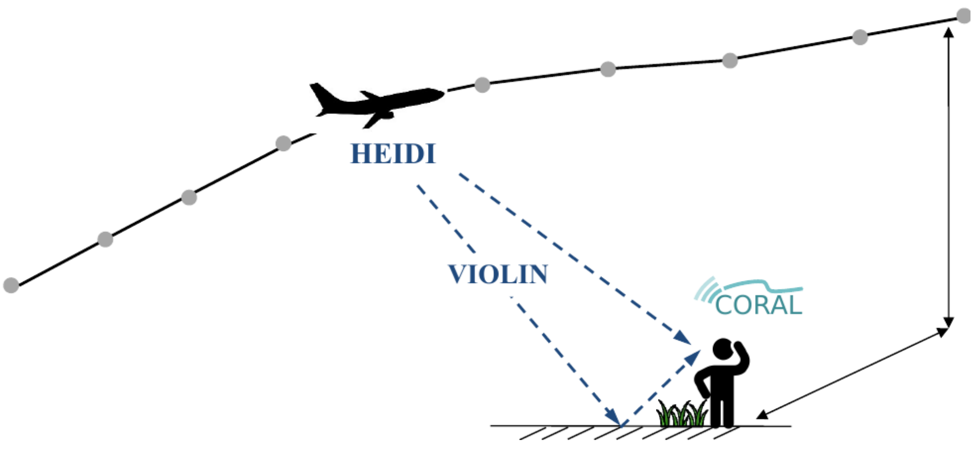


Figure 5.2.: Illustration of the application scenario ”Auralization of flyover with HEIDI”

5.2.3. Developer

In this application scenario, the complete process chain is scrutinized during further development. An application mode, as proposed in Sec. 5.1, dedicated to the developer would be used in this case. Logically, this scenario addresses the developer group. It is intended to help developers review new modules or functions for example. The concept of the corresponding application mode is described in Sec. 6.1.

5.2.4. Auralization of sound immission from distributed engines

A recent addition to VIOLIN is the implementation of multiple distributed engines. Prior to that, a point source at each aircraft position was considered. For this application scenario, two test cases with different types of propulsion system are defined: propeller and fan. Both cases have the following setting:

- Multiple distributed engines (i.e. eight engines)
- Identical operating point for each engine
- Varying operating points along the trajectory (identical for each engine)
- Relatively close-distanced position of observer microphone (i.e. several 100m)

This application scenario is visualized in Fig. 5.3.

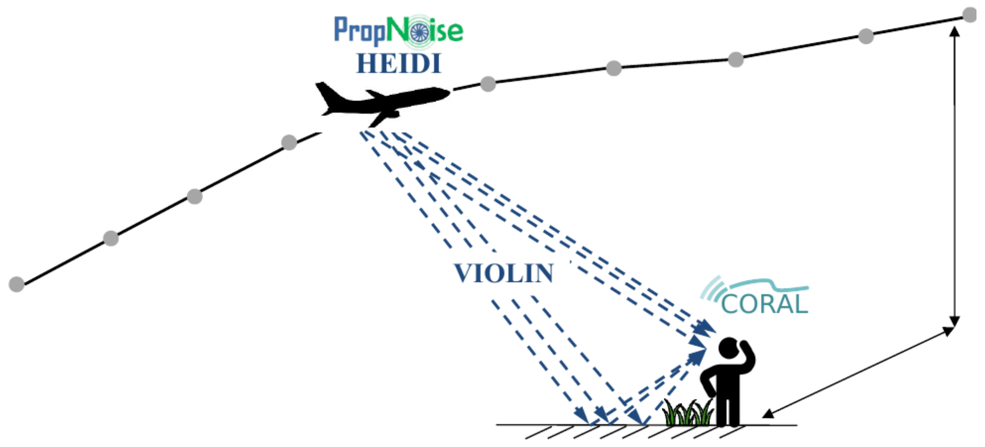


Figure 5.3.: Illustration of the application scenario "Auralization of sound immission from distributed engines"

5.2.5. Certification and community noise

This application scenario only involves a flyover, meaning the software tool VIOLIN. The purpose of this scenario is to calculate a noise carpet. Sound is no longer propagated to one microphone but to several microphones of a structured layout. The result is a noise map on the ground, also called noise carpet. A typical use case would be for residents in the vicinity of airports for example. Therefore, the noise carpet should be positioned relatively far away (i.e. several kilometers). One of the microphones represents the certification point for the acoustic noise certification. More details on the structure and implementation of the noise carpet are described in [31]. Figure 5.4 visualizes this application scenario.

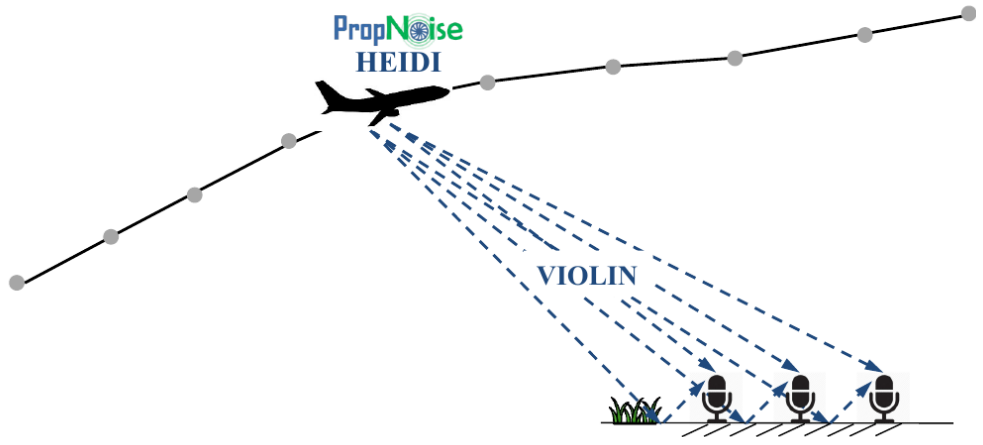


Figure 5.4.: Illustration of the application scenario ”Certification and community noise”

5.2.6. Auralization of static engine tests without flyover

In this scenario, a typical engine test is emulated. During an engine test, the observer is located in a room with the test engine. The observer stands at a certain distance and angle to the engine and listens to how the engine sounds. In this case, the user defines the relative position of the observer. There are two possible options to implement this scenario:

1. In VIOLIN, the trajectory is configured so the starting and end point have the same coordinates. The time parameter and observer position are adjusted accordingly. This way, no flyover is performed and the sound propagation of only one position is generated virtually. In other words, the aircraft does not move during the simulation to emulate the circumstances of an engine test. The resulting noise immission is auralized with CORAL as per usual.

2. VIOLIN is not used to simulate an engine test. Instead, a simple dummy case is directly provided for CORAL, taking sound emission data from PN for example.

This application scenario is visualized in Fig. 5.5.

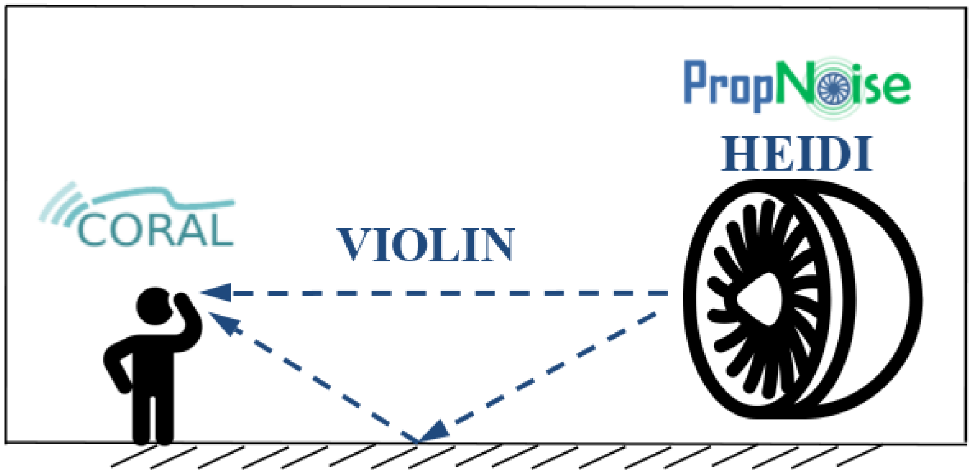


Figure 5.5.: Illustration of the application scenario ”Auralization of engine tests”

5.2.7. The input options of a test case

Each application scenario represents a testing category. One scenario can have multiple test cases. The following input options for a test case are available:

- Variable number, position and/or distance of engines
- Constant operating point along the trajectory
- Changing operating point along the trajectory (simultaneously for all engines)
- Varying linear flight sequences (horizontal, climb, descent)[1]
- One or multiple observer positions
- One noise source, a selection or all sources from input
- Different types of propulsion system (propeller, fan) including buzz saw noise source
- With or without specific physical effects[2]

[1] One flight sequence for each operating point.
[2] i.e. ground reflection, atmospheric absorption, Doppler effect

5.3. The software quality requirements

To further specify the requirements with respect to the quality characteristics usability and maintainability, the Requirements-Properties Matrix introduced in Sec. 3.1 is used on the requirements identified in Sec. 5.3 and Sec. 5.2. The result is depicted in Tab. 5.1. Each specification of a requirement is assigned a unique code consisting of a letter for the requirement, a letter for the characteristics and a number.

Table 5.1.: Overall Requirements-Properties Matrix of the software process chain

Property / Requirement	Usability (U)	Maintainability (M)
A: Different features according to different needs	• AU1: Implement at least two application modes with different outputs, i.e. user mode and developer mode. • AU2: Allow user to specify application mode at execution.	• AM1: Implement a framework to easily add or remove a mode. • AM2: Implement modes without interference of other modes. • AM3: Integrate output of different modes without affecting the clarity of the code. • AM4: Generate output files in each mode for testing.
B: Standardization within each tool and among the tools	• BU1: Standardize code documentation style. • BU2: Standardize software testing.	• BM1: Define documentation guidelines for automatic rendering. • BM2: Design reusable software testing framework.
C: Software testing (and Integration)	• CU1: Allow user to easily add and remove test cases. • CU2: Allow user to manually execute software testing.	• CM1: Implement a testing framework including integration and unit tests. • CM2: Integrate testing to the development process.
D: Custom user input	• DU1: Allow user to easily define relevant input parameters. • DU2: Notify user of error.	• DM1: Provide default input in case of invalid or no user input.
E: Consideration of application scenarios	• EU1: Provide each application scenario for the user.	• EM1: Implement at least one test case for each scenario.

6. Concept for implementation

Based on the requirements defined in Tab. 5.1, a number of approaches to improve software quality is determined. In total, five approaches are implemented within the scope of this thesis. A detailed concept development of each is presented as follows:

- The implementation of the application modes is outlined in Sec. 6.1.
- The implementation of a test suite is outlined in Sec. 6.2.
- The guideline for the documentation style is outlined in Sec. 6.3.
- The continuous integration of testing and documenting is outlined in Sec. 6.4.
- The implementation of a post-processing script is outlined in Sec. 6.5.

6.1. The application modes

Section 5.1 proposes the implementation of application modes that are accustomed to the various groups of users. The following modes are defined to satisfy AU1:

1. The release mode is the default mode of execution and is based on the software tool PN. This mode is clean and simple designed for the users who operate the software. In other words, the user is oblivious to the software process and only interested in the results of the software. Therefore, only the standard output files containing the results of the tools are provided. In case of VIOLIN, this includes six HDF5[1] files. CORAL produces an audio file for each auralized source. Further details on the output files are given in [28].

2. The debug mode is an extension to the release mode. In addition to the standard output, further information is provided for a deeper insight. This mode is useful for developers to spot errors more easily during the development phase. The output is extended by the following for easier evaluation of the process:

    - Code-intern values like data shape, size or dimension and intermediate results
    - Energy calculations with warnings for violation of conservation of energy

[1] **H**ierarchical **D**ata **F**ormat (HDF) is used to store large amount of data.

3. The plot mode is another extension to the release mode. In this mode, all kind of plots are created for post-processing purposes. To avoid clutter in the output, all generated plots are saved in a respective location and not shown at execution. This mode is dedicated to the scientist as a developer. The physical and functional correctness of the results is validated and reviewed. Errors and oddities are spotted in a visualization more easily than in a matrix of numbers. However, a further requirement has rendered this mode more useful as a post-processing routine than an execution mode that is tied to the entire software process. This is requirement D (Custom user input). The plots are generated using the results stored in the HDF5 files. Therefore, it is more reasonable to separate the plotting from the software process. In this way, it is not necessary to repeat the entire software process to adjust the visualization of the results, which saves time. This post-processing script is introduced in Sec. 6.5.

The implementation of the application modes adheres directly to requirement A (Different features according to different modes). Furthermore, the application modes are implemented for VIOLIN and CORAL. This indirectly satisfies the requirement B (Standardization within and among the tools). In this regard, the following two subsections propose two possible approaches to the implementation of the application modes. Subsection 6.1.3 evaluates both concepts based on the defined requirements to determine the more suitable one for this thesis.

6.1.1. Concept using lambda functions

This approach makes use of lambda functions to create an individual `print`-function for each application mode that is not the default mode (AM1). Each lambda function is set at the beginning of the software process. The user chooses the application mode by means of a unique command-line argument at execution (AU2). The arguments are parsed within the program and converted to Boolean variables. Listing 6.1 gives an example of a lambda function for an application mode. If the respective command-line argument is set, `debug_print` prints all arguments passed to the function. Otherwise, the function returns nothing. The individual lambda function is intended to be used like the regular `print()`-statement throughout the program. This way, the user only sees the output of the chosen mode while the developer can edit or ignore the lambda functions as the clarity of code is preserved (AM3). In comparison, the usage of `print()`-statements would require a constant `if`-clause to check the mode.

```
    debug_print = print if args.mode_debug else lambda *output: None
```

Listing 6.1: Exemplary lambda expression used for an application mode

The lambda functions are easy to use and sustain the code's readability. However, this approach has two striking disadvantages:

1. The lambda functions are not global. This means, to use the special `print`-statements throughout the program, the functions have to be passed around like variables. In case of multiple application modes, this can result in code clutter (infringing AM3).

2. A bigger problem is the strict evaluation. As explained in Sec. 6.1, the debug mode is used to provide intermediate outputs like energy calculations. Ideally, the energy calculation, a computationally expensive routine, is only executed in the debug mode. However, due to strict evaluation, an expression as shown in Listing 6.2 is evaluated regardless of the application mode (infringing AM2). In other words, the energy is computed either way and simply not printed in the default mode. This renders the extra `print`-functions superfluous.

```
    debug_print(f'Energy of signal: {Time_Signal_Energy(p,t)}')
```

Listing 6.2: Exemplary usage of the `debug_print`

6.1.2. Concept using the "logging" module

This approach uses Python's `logging` module introduced in Subsec. 3.3.1. In comparison to the concept approach described in Subsec. 6.1.1, this neither requires the implementation of new functions nor the passing around. The `logging` can be used throughout the program by simply importing the module (AM3). In case of the debug mode, the library already offers the event level `DEBUG` and the complementary logging function `debug()`. The functionalities for the debug mode are ready for use. Similarly to the other concept, the user chooses the application mode with command-line arguments (AU2). To track events of the debug level, the following configuration is set:

```
    logging.basicConfig(level=log.DEBUG, filename='debug.log',
        filemode='w', format='%(filename)s-%(lineno)d: %(message)s')
```

Listing 6.3: Configuration of the debug mode using the `logging` module

Another advantage compared to the lambda function is the option to automatically create a logging file (AM4). The keyword `filename` specifies that an instance of the class `FileHandler` be created instead of the default class `StreamHandler`. This way, the specified file is opened in the specified `filemode` and used as the stream for logging rather than the output console. Not only does this reduce clutter in the output but also serves as a suitable testing opportunity. The console is not spammed with information. The generated file is more readable and can be used to test the respective application mode.

```python
class Lazy(object):

    def __init__(self, f, *args, **kwargs):
        self.f = f
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return str(self.f(*self.args, **self.kwargs))
```

Listing 6.4: Source code of class used for lazy evaluation

However, the problem of strict evaluation remains. For this problem, a new class is introduced. Listing 6.4 presents a class to achieve lazy evaluation. This class takes a function and its arguments or keyword arguments as parameters. Because the function is only passed as an object and not called, no expression is evaluated. The function is only called with the specified arguments when the string representation of the class `Lazy` is invoked (AM2). For this to work, it is also important to not use **f**ormatted **string**s (f-strings) like in Listing 6.1. A f-string would immediately invoke the string representation of an object and therefore call the function that is not supposed to be evaluated strictly. Alternatively, the `logging` module is intended to be used in combination with lazy evaluation as follows:

```python
log.debug('Energy of signal: %s', Lazy(Time_Signal_Energy, p, t))
```

Listing 6.5: Exemplary usage of the `logging` module

Logically, the `Lazy` class is designed for the given problem: strict evaluation of string expressions. Therefore, it is assumed that the class only takes a function with a return value that can properly be transformed into a string.

6.1.3. Comparison of concepts

The two concepts presented in Subsec. 6.1.1 and Subsec. 6.1.2 were developed consecutively with the latter fixing the flaws of the first concept. To confirm the improvement, both concepts are evaluated based on the requirements defined in Tab. 5.1. A simple checklist containing the relevant requirements is depicted in Tab. 6.1. When comparing both concepts, it becomes clear that the approach using the `logging` module with lazy evaluation is more suitable. This approach satisfies all requirements. Therefore, the concept proposed in Subsec. 6.1.2 is implemented in the scope of this thesis.

Table 6.1.: Evaluation of concepts for the implementation of application modes

ID	Requirement	Concept 6.1.1	Concept 6.1.2
AU1	Implement at least two application modes with different outputs.	✓	✓
AU2	Allow user to specify application mode at execution.	✓	✓
AM1	Implement a framework to easily add or remove a mode.	✓	✓
AM2	Implement modes without interference of other modes.		✓[1]
AM3	Integrate output of different modes without affecting the clarity of the code.	(✓)[2]	✓
AM4	Produce appropriate output files in each mode for testing.		✓

6.2. The test suite

Regarding software testing, the initial situation of both software tools is the following:

- In VIOLIN, a test suite has been implemented. However, it is inactive and outdated with minimal test coverage. Therefore, it can be assumed that no test suite exists.
- CORAL has no test suite and only produces audio files as output.

Before introducing a testing framework that is applicable to both software tools, testable output has to be generated in CORAL. To satisfy BU2, it is reasonable to use VIOLIN as a reference. VIOLIN writes HDF5 files as output. Accordingly, CORAL can produce HDF5 output files containing the time signals for the respective audio files. With this in mind, the content of this section is structured as follows:

- Subsection 6.2.1 presents the structure of the testing framework.
- Subsection 6.2.2 presents the integration tests.
- Subsection 6.2.3 presents the unit tests.
- Subsection 6.2.4 checks the implementation against the requirements.

[1] Due to lazy evaluation
[2] The lambda functions have to be passed around the program.

6.2.1. Structure of the testing framework

The testing framework follows the concepts introduced in Subsec. 2.3.1 and Subsec. 2.3.2.

```
testsuite
├─ Integration_Tests
│  ├─ 01_Engine_Tests
│  ├─ 02_Flyover_with_PN
│  ├─ 03_Flyover_with_HEIDI
│  ├─ 04_Developer
│  ├─ 05_Certification_and_Community_Noise  .............This is for VIOLIN only.
│  ├─ 06_Distributed_Engines
│  ├─ projects.txt
│  └─ projects_debug.txt
├─ Unit_Tests
│  ├─ Unit_1
│  ├─ ...
│  └─ projects.txt
├─ build_integration_tests.sh  .................Build tests for default or debug mode.
├─ compare_debug.sh  ..........................................................Compare debug.log.
├─ compare_results.sh  ....................................................Compare HDF5 files.
├─ create_reference.sh  ..................................................Update reference files.
├─ move_data.sh  ..........................................Move output files to output directory.
├─ run_debug_tests.sh  ........................................Run test cases of 04_Developer.
├─ run_integration_tests.sh  .............................................Run integration tests.
├─ run_unit_tests.sh  ..................................................................Run unit tests.
├─ test_debug.py
└─ test_units.py
```

Figure 6.1.: Directory tree of testing framework

Figure 6.1 depicts the general structure of the testing framework. The test suite includes unit tests and integration tests (CM1). The highest level of the Testing Pyramid, UI tests (see Subsec. 2.3.1), is not considered. Neither VIOLIN nor CORAL provide a fully developed **U**ser **I**nterface (UI). Therefore, there is no need for the simulation and assessment of user interactions. All test cases are located in the directory of their respective level. The `Integration_Tests` directory is further categorized by application scenarios. Each scenario contains at least one test case with the following generalized[1] structure (EU1, EM1):

```
integration_test_case
├─ input ........................ Input files for VIOLIN/CORAL.
├─ output ....................... Output files of VIOLIN/CORAL.
├─ reference .................... Golden master files of VIOLIN/CORAL.
├─ Input.json ................... User input file of VIOLIN/CORAL.
└─ run.sh ....................... Execution script of VIOLIN/CORAL.
```

Figure 6.2.: Generalized directory tree of an integration test case

The `Unit_Tests` directory has a simpler structure and encompasses of individual test cases for one software unit each with an equally simple structure:

```
unit_test_case
├─ output ....................... Output files of unit test.
├─ reference .................... Golden master files of unit test.
└─ Unit_input.h5 ................ Input file of unit test.
```

Figure 6.3.: Directory tree of a unit test case

A unit test case essentially is a simplified version of an integration test case. However, both types of test follow the principle of Golden Master Testing (see Subsec. 2.3.2). The implementation of this testing method is explained in the following sections. In doing so, the rest of the files depicted in Fig. 6.1 are discussed.

[1] Input and output files are individual to each software tool. However, the general structure is the same. The focus of this thesis is to provide a solution concept for for the given subject. For this reason, a more detailed graph is not given. The visualization also remains simpler this way.

6.2.2. The integration tests

The integration tests are composed of two steps: build and compare.

In the first step, all necessary steps for the tests are prepared. The software is run for each test case to produce outputs of the current software version. This process is automated by the script `build_integration_tests.sh`. For this purpose, all test cases are listed in the file `projects.txt` with the exception of the developer test cases. Test cases of the application scenario `Developer` are run in the debug mode. They are denoted in a separate file named `projects_debug.txt`. The script to build integration tests takes the name of the file as an argument. Depending on the specified file, the test cases are built in the release mode or the debug mode (BM2). After building the test cases in the respective mode, an intermediate step is executed. The script `move_data.sh` moves all testable output files to the directory `output` of each test case (BM2). This includes all HDF5 files and, if available, the `debug.log`.

When all preparations are completed, the results are compared to the golden master. The comparison of HDF5 files is executed by the script `compare_results.sh` using the command-line tool `h5diff` (BM2). This tool compares two HDF5 files and reports the differences. In case of failure, meaning a difference was detected, an error file is generated. The `debug.log` is automatically tested with the script `compare_debug.sh` but uses a different approach than the comparison of HDF5 files. The logging files are tested assertion-based using the Python framework `unittest` introduced in Subsec. 3.3.2. This is implemented in the file `test_debug.py` (see Listing B.1). The Python script is automatically run for each test case of the scenario `Developer` by the `compare_debug.sh` script.

The steps above are concatenated in the script `run_integration_tests.sh`. For test cases of the scenario `Developer`, the script `run_debug_tests.sh` automatically executes the mentioned steps. Both are intended to be used by the user for manual local testing during the development phase. The user simply has to specify the relative path to the desired test cases in the files `projects.txt` and `projects_debug.txt` (CU1) and execute the respective Bash scripts (CU2). No further specifications or arguments are required. The only condition is that the test cases are prepared with the correct file structure. If this is satisfied, the user can add and remove test cases.

Lastly, to realize the advantage of Golden Master Testing, the script `create_reference.sh` automatically updates the golden files of a specified test case. The output files of a software test are moved to the reference file to replace the golden master. Logically, it is assumed that software tests were run. For reference, the testing process should follow the cycle depicted in Fig. 2.4.

6.2.3. The unit tests

As explained in Subsec. 6.2.1, the structure of unit tests is a simplified version of integration tests. Both follow the Golden Master Testing technique. However, unit tests do not require preparations and only consist of the assertion-based testing. Similarly to the integration tests, the test cases are defined in a file called `projects.txt`. All specified tests are automatically run with the script `run_unit_tests.sh`. The tests are implemented in separate Python scripts using the framework `unittest`. An example is given in Listing B.2. The script `create_reference.sh` to update the golden master also works for unit tests.

6.2.4. Conformance of the testing framework to the requirements

In Tab. 6.2, the concept of the testing framework is checked against the requirements.

Table 6.2.: Evaluation of the testing framework

ID	Requirement	
BU2	Standardize software testing.	✓ The testing framework is applicable to VIOLIN and CORAL.
BM2	Design reusable software testing framework.	✓ The individual scripts are usable for both types of integration tests.
CM1	Implement a testing framework including integration and unit tests.	✓ The testing framework implements integration and unit tests.
CU1	Allow user to easily add and remove test cases.	✓ All active test cases are listed in the user-definable `projects.txt` .
CU2	Allow user to manually execute software testing.	✓ All software tests can be run with the respective `run` execution scripts.
EU1 & EM1	Provide each application scenario for the user. Implement at least one test case for each scenario.	✓ All application scenarios are represented by at least one integration test.

[1] **J**ava**S**cript **O**bject **N**otation (JSON) is a standard lightweight file format.

6.3. Guideline for software documentation

Based on Sec. 2.4 and the user profile created in Sec. 5.1, a technical documentation and a user documentation are most reasonable (BU1, BM1). The key attributes for effective documentation presented in Sec. 2.4 are listed in the following as requirements to the documents:

- The documentation should be up-to-date.
- The documentation should be readily available and easily located.
- The documentation technology must allow easy creation and maintenance.

The documentation guideline in this thesis is limited to two artifacts:

- The user documentation is represented by the `README.md` file of a project.
- The technical documentation is provided by the source code.

6.3.1. The user documentation

Every GitLab repository of a project contains a `README.md`. This Markdown file serves as a manual for the user and specifies how to operate the software tool with all its features. The file should be structured as follows in a precise and compact manner without leaving out any relevant details:

1. First and foremost, it is important to note what software and, if known, what respective software version is required for the application. This way, the user can prepare the required software like specific versions of Python modules before usage. Other prerequisites like relevant hardware must be listed as well.
2. The most important part of the manual explains the following aspects:
    - How to run the software tool? Provide or refer to an example.
    - What input files are required? Where must they be located?
    - Which application modes are available? How to choose a mode?
3. The post-processing routine needs to be introduced and explained.
4. Lastly, the structure and functionalities of the test suite are explained. The user should know how to run the test suite, what test cases are currently active, how to add or remove a test case and to which degree the test suite is integrated.

6.3.2. The technical documentation

For a clear and structured inline code documentation, the usage of reST or `numpydoc` is considered. Both are commonly used in the department of Engine Acoustics and can be rendered by the tool `pdoc` (see Subsec. 3.3.3). The inline documentation serves as an artifact to effectively communicate knowledge to the developer. However, it also is the base for the documentation tools `pdoc` and `pyreverse` (see Subsec. 3.3.4) to provide additional artifacts for the developer and user (BM1). Therefore, the source code must be documented thoroughly. The following guideline should be followed:

- The description of a class starts with the phrase ”Class used to/for” and is followed by the general purpose of that class in a precise manner.
- The description of a function is expressed in the imperative form. In case of a constructor, the following pattern is used: ”Construct a new ’[name of class]’ object.”.
- The data type of a variable must be denoted.
- All parameters and return parameters are shortly explained. If possible, the unit of a variable is specified. If relevant to the context comprehension, the shape or dimensions of a variable are specified, i.e. in case of arrays.
- A class is further described by defining its string representation. The string representation should clearly reveal what kind of object this class is.

Listing 6.6 gives a demonstrative example for the guidelines.

```python
class Example:
    """Class used to demonstrate the guidelines."""

    def __init__(first: int):
        """
        Construct a new 'Example' object.
        :param first: The first parameter [unit]
        """
        self.first: str = str(first)
        """The first instance attribute"""

    def do_something(self):
        """
        Do something for a specific purpose.
        :return: something
        """
        return 'something'
```

Listing 6.6: Exemplary code following the guidelines for a technical documentation using reST

Using the documentation tools pdoc and pyreverse as explained in Subsec. 3.3.3 and Subsec. 3.3.4, additional artifacts are generated from the source code. Figure 6.4 depicts the overall file structure of a project including the location of the artifacts. This applies to VIOLIN and CORAL for the sake of standardization (B).

```
project
├─ documentation ........................................Output files of unit test.
│  ├─ class_diagram ..............................Artifacts generated by pyreverse.
│  │  ├─ classes.pdf
│  │  └─ packages.pdf
│  └─ html ..............................................Artifacts generated by pdoc.
│     ├─ Class.html
│     └─ ...
├─ example ............................Examples for the user (see Subsec. 6.3.1).
├─ post ..............................................Post-processing (see Sec. 6.5).
├─ src ......................................................Source code of project.
└─ testsuite ..............................................Test suite (see Sec. 6.2).
```

Figure 6.4.: General directory tree of a software project with focus on the documentation artifacts

6.4. Continuous integration

To automatically and continuously test changes to the software (CM2), the test suite is integrated using GitLab CI/CD (see Sec. 3.2). Additionally, the tools pdoc and pyreverse are used to keep the technical documentation up-to-date (BM1). The following pipeline is built:

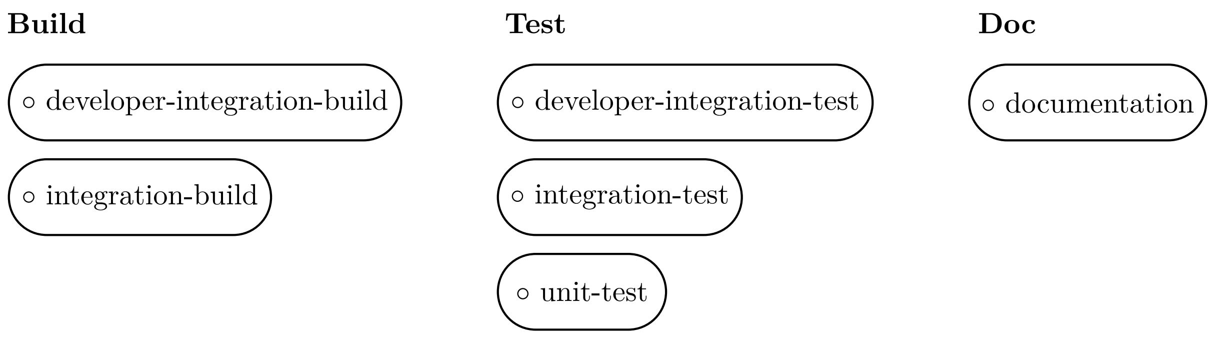


Figure 6.5.: General structure of the GitLab CI/CD pipeline of both software tools

The pipeline basically executes all steps of the test suite as described in Sec. 6.2 extended by the automatic documentation. For this purpose, three stages are defined:

1. The stage `build` prepares the integration tests as explained in Subsec. 6.2.2. The script `build_integration_tests.sh` is run for normal and `Developer` integration tests with the respective arguments. Relevant outputs are moved to the output directory using the script `move_data.sh`.
2. The stage `test` executes the actual software testing. All integration tests and unit test cases are run by using the scripts `compare_results.sh` and `compare_debug.sh`.
3. The stage `doc` makes sure all changes are documented. All artifacts generated with `pdoc` and `pyreverse` are updated in this stage.

The choice to execute each step separately is made to easily distinguish between runtime errors and software changes that affected the results. Furthermore, unnecessary process time is saved. If a job of one stage fails, the pipeline is halted. It is also reasonable to activate the pipeline for the main branch and develop branches only. To avoid clutter, it might even be more reasonable to automatically document changes of the main branch only. An exemplary `.gitlab-ci.yml` is given in Listing B.3.

6.5. A post-processing script

As explained in Sec. 6.1, the post mode is changed to a post-processing routine due to requirement DU1: "Allow user to easily define relevant input parameters.". In the case of plots, these are parameters that can change the appearance of a plot. Such parameters should be arbitrarily adjustable after the software tool is run without requiring the repeated execution of the tool. This option is relevant for the scientist as a user. For example, when presenting the results in a paper, the graphics need to be adjusted to fit the style and format of the document. In context with the requirement, this further means the following: The scientist can change parameters of the graphs after the execution without having to know about the details of the plotting implementation. With this in my mind, the concept of the post-processing script is presented in this section as follows:

- Subsection 6.5.1 depicts the structure of the post-processing routine.
- Subsection 6.5.2 defines the relevant plots.
- Subsection 6.5.3 presents the user-defined parameters.

6.5.1. Structure of the post-routine

The structure of the directory containing the post-processing routine is depicted in Fig. 6.6. Only the main Python script as well as all `settings` files are of relevance to the user. To execute the post-processing routine, the user simple runs the `main.py`. All adjustable parameters are defined in the settings files. These files are explained in Subsec. 6.5.3.

```
post
├─ main.py
├─ parser.py   ........................Command-line argument(s) parser.
├─ plots.py    ..................................Implementation of plots.
├─ reader.py   ...................................Reader for output files.
├─ settings_global.py
├─ settings_matplotlib.py
└─ settings_plot.py
```

Figure 6.6.: Directory tree of the post-processing routine

The post-processing routine proceeds as follows:

1. The execution of the main script has the following options:
   - An obligatory argument is the specification of the path to the output files. The validity of the path as well as the existence of all necessary files are checked before execution. In case of an exception, the user is notified of the error.
   - Optionally, the user can choose to save all figures. If no location is provided, the figures are saved in the working directory.
   - Alternatively, the user can specify to not show the figures at all. This option is useful for the developer working on the implementation of the plots.

   The command-line parsing is implemented in the file `parser.py`.
2. The necessary data for the plots is extracted from the output files of the specified location. This is implemented in the file `reader.py`.
3. The plotting routine is implemented in the `plots.py` file. Here, all plots are created using the parameters specified by the user.

An important aspect for the implementation is that the individual plotting routines are completely independent of another. This includes the reading routine. Even if some plots require the same data, the routine should be implemented, so the data can be accessed individually and multiple times. This way, the addition or removal of plots is possible without interfering with unrelated plots. For this sake, redundancies are acceptable.

6.5.2. The relevant plots

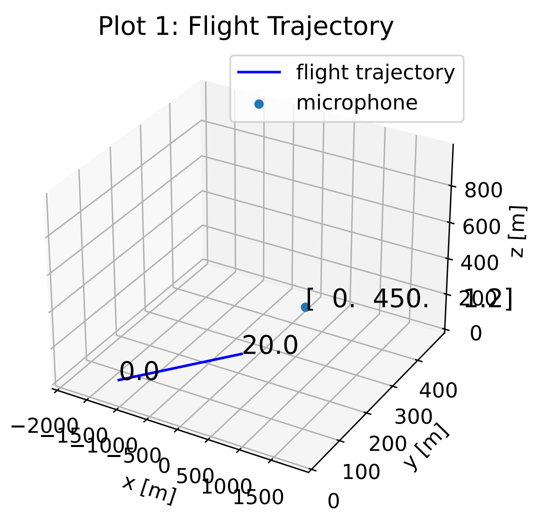
In this subsection, the required plots of the post-processing routine are defined.

The software tool VIOLIN should produce the following plots of the results:

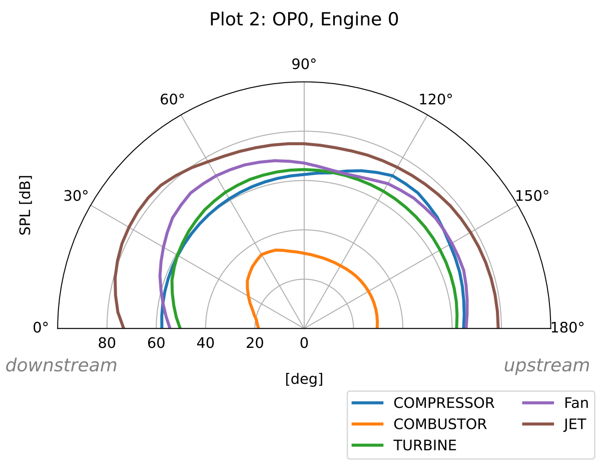
1. The flight trajectory is visualized in one 3D line plot including the start and end position of each operating point. Additionally, the location of the microphone or of all microphones in case of a noise carpet is highlighted (see Fig. 6.7a).
2. The sound directivities of all noise sources are visualized for each operating point. For this purpose, a line plot on a polar axis is used respectively (see Fig. 6.7b). Therefore, each angle-dependent frequency spectrum of a source is summed over the frequencies to obtain the total angle-dependent total values. In case of noise sources provided by the emission tool HEIDI, all fan and jet noise sources are respectively combined to total `Fan` and `JET` by means of addition as sound pressures.
3. The resulting spectrograms are visualized as 2D contour plots (see Fig. 6.7c). This includes the overall emitted spectrogram, the overall received spectrogram and the overall received spectrogram with dB(A) weighting. In this context, "overall" refers to the spectrogram as a total of all noise sources of the complete trajectory. The overall spectrogram must be calculated carefully and heed the following rules:
   - The total of tonal sources is obtained by concatenating all tonal spectrograms. The frequency-dependent spectra for each point in time are merged together. The result is one big spectrogram containing all tonal spectra sorted by the frequencies in increasing order. For spectra of the same frequencies, their sound pressure values are summed linearly. Likewise, the corresponding time-varying frequency values are merged and sorted to a matrix of the same shape as the total spectrogram. Duplicate frequencies are removed. Altogether, this means that the concatenation never changes the size of the time axis but can increase the size of the frequency axis.
   - The total tonal spectrogram is converted from complex amplitude-scaled pressure values to real RMS-scaled squared pressure values.

- The total broadband spectrogram and corresponding frequency matrix are obtained in the same manner as of tonal sources. However, the source types are treated separately because broadband sources already have real RMS-scaled squared pressure values.
- The total spectrogram is obtained by merging the total tonal spectrogram and the total broadband spectrogram in the same manner. The same applies to the total frequency matrix from the matrices of the two source types.
- The total spectrogram is converted to **s**ound **p**ressure **l**evel (SPL) values before visualization.

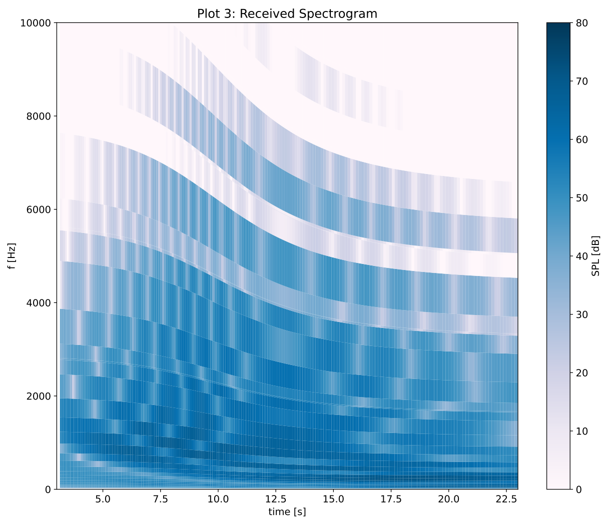
4. The overall time-dependent received **s**ound **p**ressure **l**evels (SPLs) of each source are visualized in one 2D line plot (see Fig. 6.7d). In this context, ”overall” refers to the complete trajectory and ”received” refers to the received spectrograms and the received spectrograms with dB(A) weighting. For this purpose, each spectrogram is summed over the frequencies to obtain the time course of the sound pressures for each source. This step must heed the following steps carefully:
   - Tonal spectrograms have to be converted from complex amplitude-scaled pressure values to real RMS-scaled squared pressure values except for fan and jet noise sources provided by HEIDI.
   - Each spectrogram is summed over frequencies as real RMS-scaled squared pressure values except for fan and jet noise sources provided by HEIDI.
   - Fan and jet noise sources of HEIDI are combined to the noise sources `Fan` and `JET`. However, the components can be of different source types. The source types of each component group are summed over frequencies separately. After converting the tonal time course of sound pressures to real RMS-scaled squared pressure values. The components are combined as real RMS-scaled squared pressure values to total `Fan` and `JET`.
   - In addition to the resulting time-varying sound pressures of all noise source, an overall tonal time course of sound pressures is calculated and visualized.
   - All resulting time-varying sound pressures are converted to **s**ound **p**ressure **l**evel (SPL) values.



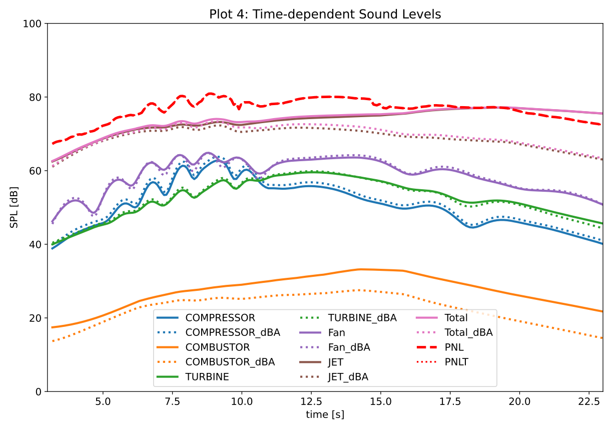
(a) Flight trajectory



(b) Sound directivities



(c) Spectrogram



(d) Time-varying SPL values

Figure 6.7.: Exemplary VIOLIN post-processing of the test case `HEIDI_NASA_STCA55_Sideline`

The software tool CORAL should produce the following plots:

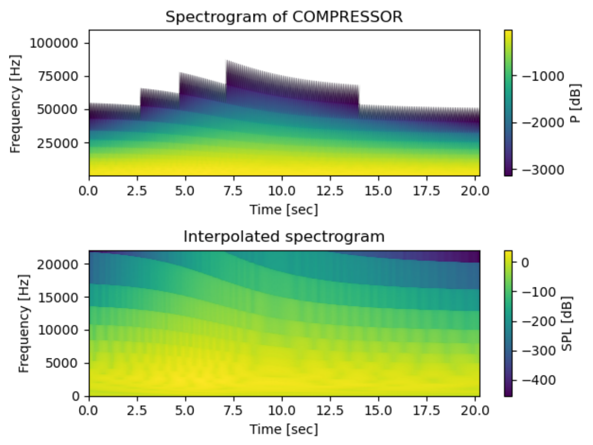
1. The original spectrogram and the resampled spectrogram of each noise source are visualized in a 2D contour plot (see Fig. 6.8a). This is an intermediate output. As explained in Sec. 4.2, the spectrograms are resampled in the pre-processing step. The two source types are treated differently:

   - Tonal sources require no resampling in the pre-processing step.
   - Broadband sources are adjusted to fit the randomly generated white noise spectrogram of uniform distribution by means of interpolation. For this purpose, the frequency matrix is interpolated to the shape of the white noise with a linear frequency scale. The resulting frequencies are used to resample the spectrogram.

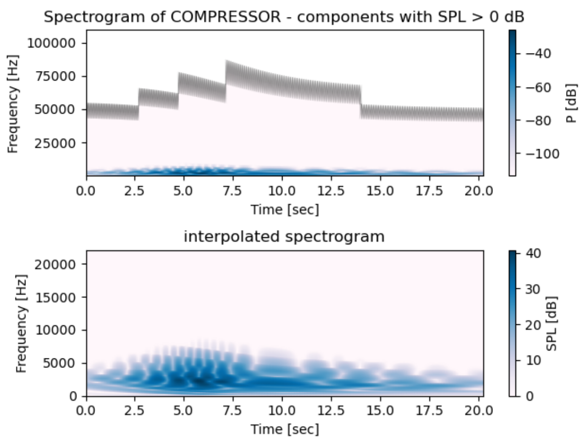
2. The original spectrogram and the resampled spectrogram of each noise source are visualized in a 2D contour plot, containing only positive values (see Fig. 6.8b). This means all SPL values greater than zero.

3. The randomly generated white noise signal of uniform distribution for each broadband source is visualized in a 2D line plot (see Fig. 6.8c). It is generated using the function `numpy.random.randn()`, creating `L` values of uniform distribution. The variable `L` equals the length of time vector.

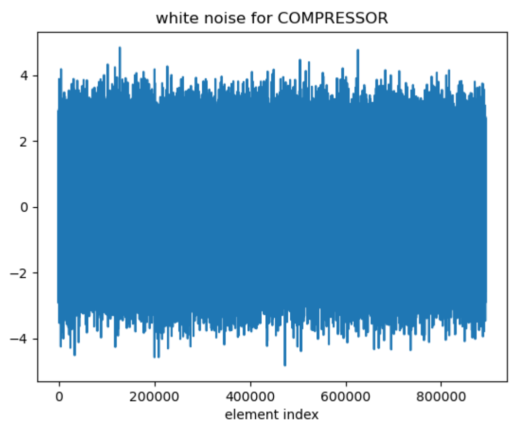
4. The time signal of each noise source is visualized in a 2D line plot (see Fig. 6.8d). This output visualizes the results. Each time signal has a corresponding audio file. All time signals should the same length.



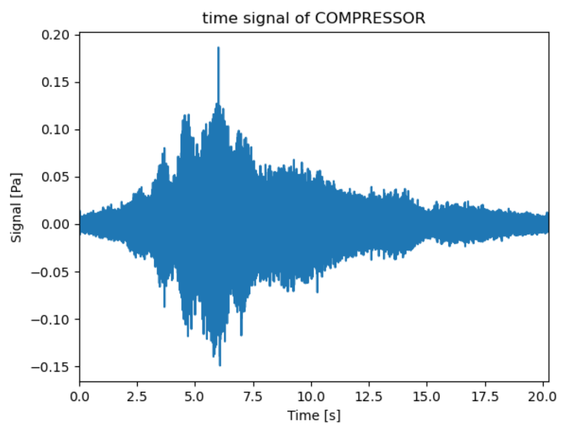
(a) Spectrogram



(b) Spectrogram with positive values only



(c) White noise signal



(d) Time signal

Figure 6.8.: Exemplary CORAL post-processing of the test case `HEIDI_NASA_STCA55_Sideline`

6.5.3. The adjustable parameters

The file `settings_matplotlib.py` contains customizable parameters that apply to all plots. This includes the following `rcParams` of `matplotlib`:

- `font.family`: The font style
- `axes.titlesize`: The font size of the plot title
- `axes.labelsize`: The label size of each axis
- `xtick.labelsize`: The label size of all ticks on the x-axis
- `ytick.labelsize`: The label size of all ticks on the y-axis
- `legend.fontsize`: The font size of the legend
- `savefig.bbox`: The bounding box in inches of the saved figure
- `savefig.dpi`: The resolution in dots per inch of the saved figure
- `savefig.edgecolor`: The edgecolor of the saved figure
- `savefig.facecolor`: The facecolor of the saved figure
- `savefig.format`: The file format of the saved figure
- `savefig.orientation`: The orientation of the saved figure
- `savefig.pad_inches`: The amount of padding around the saved figure
- `savefig.transparent`: The transparency of the Axes patches

The parameters of each individual plot are specified in `settings_plot.py`. A distinction between the variables is made by using the prefix with the format ”p[plot number]_”. In general, the following parameters can be adjusted for each plot individually if applicable:

- The size and title of the figure
- The file name of the saved figure
- The label, limits and ticks of the x-axis, y-axis and z-axis
- The color, style and width of the line(s)
- The location and column number of the legend
- The shading, colormap and colorbar of the contour plot

New parameters can be added easily and simply must be used in the `plots.py`.

7. Evaluation of the software quality

In this chapter, the presented implementation concepts are evaluated based on their software quality using the quality measures introduced and elaborated in Sec. 3.4. The approach to evaluate the usability and maintainability improvement of VIOLIN and CORAL is explained in Sec. 7.1. Based on the general approach, the rest of the chapter is struvtured as follows:

- A score system for the measures is presented in Sec. 7.2.
- The software quality of VIOLIN is evaluated in Sec. 7.3.
- The software quality of CORAL is evaluated in Sec. 7.4.

The implementation and evaluation results are discussed in Ch. 8.

7.1. General approach

To evaluate the software quality, the chosen quality measures of ISO/IEC 25023 presented and explained in Sec. 3.4 are used. For this purpose, the proposed implementation described in Ch. 6 is compared to an older version of each software tool. Table 7.1 depicts which version of each tool is analyzed. The "Before" version is a software state before the proposed implementation was integrated. Therefore, it is assumed that an improvement of software quality is reflected in the evaluation result of the "After" version which includes all proposed implementations.

Figure 7.1.: Calendar date of software version used for the evaluation

	Before	After
VIOLIN	09.12.21	08.08.22
CORAL	21.12.21	08.08.22

Due to the limited time frame, the selected quality measures are further narrowed down. The measurement functions MMd-1-G (Modification efficiency) and MMd-2-G (Modification correctness) of the maintainability subcharacteristic modifiability are excluded. Both measurement functions include a time variable (see Tab. A.8). This means the application of the function requires time to obtain evaluation results. Modification efficiency is determined by comparing the required time to the expected time of modifications. At least two modifications have to be implemented and timed for each version of the software tool by at least two different developers for a proper assessment. Modification correctness checks the failure rate of modifications within a defined period of time after being implemented. With the limited given time, the measure cannot be properly evaluated. Therefore, it is reasonable to postpone such time-dependent measurement functions to the future.

The remaining measurement functions can be used without further ado. For a better understanding of the evaluation approach, the structure of the functions is elaborated. Each function describes a result that is relative to a unique target value which is established as a requirement. This proportion can be expressed for each measurement function with the generalized equation[1]

$$X = A/B \tag{7.1}$$

where X is the resulting score of the measure calculated from the ratio of the achieved subset A to the specified target value B. Alternatively, the ratio X can be calculated as

$$X = 1 - \bar{A}/B \tag{7.2}$$

where $\bar{A}$ is the complement to the value A of Eq. 7.1[2]. For simplification, the result of a measure in the evaluation is always depicted in the form of Eq. 7.1. The variables A and B of each function are defined for general use in the ISO/IEC 25023 (see Appendix A). In the scope of this thesis, the measurement functions are further specified to the software tools VIOLIN and CORAL. For this purpose, the following questions are answered for each selected measure:

- What is the definition of the target value as per ISO/IEC 25023?
- How is the target value further defined in regard of the software tool?

Based on this, a score system is developed to determine how a target value is achieved. The score system is applied in the pre- and post-analysis evaluation of both software tools. For each version of a tool, the unique target value is described qualitatively and specified quantitatively according to the system. The result of each measure is depicted as the ratio of the achieved value A to the defined target value B and as the corresponding value X in percentage.

[1] This generalized equation does not apply to the measure MMd-1-G, which was excluded in the course of this thesis (see Tab. A.8).

[2] In the ISO/IEC 25023 [17], the variable A is used regardless of the type of subset. To distinguish, the value of A is defined accordingly. An example of both can be found in Tab. A.3.

7.2. Score system of the evaluation

In this section, a score system for the evaluation is established based on the definition of a target value as per ISO/IEC 25023 (see Appendix A)[1]:

- The measure UAp-1-G (Description completeness) appraises the usage scenarios in regard of user documentation. The target value is the number of usage scenarios. Each documented usage scenario equals one point. The function is defined as

$$X = \frac{\text{Number of documented usage scenarios}}{\text{Number of existing usage scenarios}} . \tag{7.3}$$

  In the scope of this thesis, a usage scenario is considered documented when any user document describes and/or visualizes the usage scenario. The term "usage scenario" is understood as follows: "How can a user utilize the given software tool?".

- The measure ULe-1-G (User guidance completeness) appraises the software functions. The target value is the number of implemented functions that require documentation. Such functions are examined and equal one point if sufficient documentation is provided. Existing but insufficient documentation of a function results in half a point. The function is defined as

$$X = \frac{\text{Number of functions described in user documentation as required}}{\text{Number of implemented functions that require documentation}} . \tag{7.4}$$

  The term "function" can mean either of the following: a user functionality of a software or a code function of the program. On the grounds that this measure belongs to the quality usability and further the subcharacteristic learnability, the first definition of function is chosen for the evaluation. In this sense, the available functions are documented, so a new user can learn how to operate the software. A new user would not be interested in the source code. The documentation of code functions would be more of concern to developers. An example is any command-line option of a usage scenario.

- The measure ULe-3-S (Error messages understandability) appraises the error messages. The target value is the number of implemented error messages. An error message should provide the following: the cause and a solution. Each piece of information equals half a point for an error. The function is defined as

$$X = \frac{\text{Number of error messages stating the reason and a possible solution}}{\text{Number of implemented error messages}} . \tag{7.5}$$

[1] The following definitions equations are adapted from ISO/IEC 25023 (see Appendix A). The exact measurement function as is can be found in Appendix A.

If the solution is given indirectly, one quarter point is assigned. A concrete example is the following error message: ”Error: The chosen sources do not match the given input files!”. While the occurred error is clear and the user can infer from that message to adjust the selection of sources, it does not clearly state how or where to change the selection of noise sources. Therefore, this would equal 0.75 points.

- The measure UOp-1-G (Operational consistency) appraises the behavior and appearance of interactive tasks. The target value is the number of interactive tasks that need to be consistent. Each interactive task that should be consistent and is so equals one point. The function is defined as[1]

  $$X = \frac{\text{Number of interactive tasks that are performed consistently}}{\text{Number of interactive tasks that need to be consistent}} \,. \tag{7.6}$$

  A counterexample for operational consistency is the following: The individual application modes are activated in different manners via a command-line argument, a file input or an edit in the source code. This would be inconsistent.

- The measure UOp-2-G (Message clarity) appraises the system messages. The target value is the number of implemented messages. Each message equals one point if the conveyed information is correct. The function is defined as

  $$X = \frac{\text{Number of messages that convey the right information}}{\text{Number of implemented messages}} \,. \tag{7.7}$$

  The software tools VIOLIN and CORAL are command-line applications. Therefore, each message that is printed in the console output except for error messages is considered for the target value.

- The measure UEp-1-G (Avoidance of user operation) error appraises user actions and inputs. The target value is the number of user operations that could be protected. Each protected user action or input equals one point. The function is defined as

  $$X = \frac{\text{Number of user operations that are protected}}{\text{Number of user actions and inputs that could be protected}} \,. \tag{7.8}$$

  Besides catching errors caused by wrong or invalid values, this could also include the implementation of default values. If this is applicable, each preventive measure of the two equals half a point.

[1] In ISO/IEC 25023 [17], the measure UOp-1-G is expressed in form of Eq. 7.2 (see Tab. A.3).

- The measure MMo-1-G (Coupling of components) appraises the software components. The target value is the number of specified components that are required to be independent. Such a component equals one point. The function is defined as

$$X = \frac{\text{Number of independent components}}{\text{Number of implemented components with no impact on others}} \ . \tag{7.9}$$

  Modules and components within modules should be independent. If the latter is not a case, half a point is deducted.

- The measure MRe-1-G (Reusability of assets) appraises the software assets. The target value is the number of assets. Each asset which is designed and implemented to be reusable equals one point. The function is defined as

$$X = \frac{\text{Number of reusable assets}}{\text{Number of existing assets}} \ . \tag{7.10}$$

  Examples of assets are source code modules containing code, testing modules, specific hardware and any documentation artifact. In VIOLIN and CORAL, the assets are the Python modules[1], the testing framework and user/technical documents.

- The measure MAn-1-G (System log completeness) appraises the system logs. The target value is the number of logs which are required. Each operation that should be and is recorded equals one point. The function is defined as

$$X = \frac{\text{Number of recorded logs}}{\text{Number of logs which are required during operation}} \ . \tag{7.11}$$

- The measure MTe-1-G (Test function completeness) appraises the software test functions. The target value is the number of required test functions. Each implemented test function equals one point. The function is defined as follows:

$$X = \frac{\text{Number of implemented test functions as specified}}{\text{Number of required test functions}} \tag{7.12}$$

  The term ”test function” is not further defined in the ISO/IEC 25023. In the scope of this thesis, the following definition is used: A test function tests a functionality of the software covering a specific layer of the Testing Pyramid (see Subsec. 2.3.1). Each tested functionality comprises a test case. In other words, the target value is the number of required test cases.

[1] Python modules are files with the extension `.py` that can be imported in another Python module.

7.3. Evaluation of the software tool VIOLIN

This section assesses the software quality of VIOLIN. For this purpose, the target values for each measure are defined for both versions in Tab. 7.1.

Table 7.1.: Target values of the measures for the pre- and post-quality-analysis versions of VIOLIN

Measure	Before	After
UAp-1-G (Description completeness)	• Flyover simulation with PN • Flyover simulation with HEIDI Target value $B = 2$ Achieved value $A = 2$	• Six application scenarios • Post-processing routine Target value $B = 7$ Achieved value $A = 7$
ULe-1-G (User guidance completeness)	• Run (default) application • Run with `--no-ground-refl`[1] • Run with `--noise-carpet`[2] Target value $B = 3$ Achieved value $A = 3$	• Run in default mode • Run in debug mode • Run with `--no-ground-refl` • Run post-processing • Run post with `-s`[3] • Run post with `--hide`[4] Target value $B = 6$ Achieved value $A = 6$
ULe-3-S (Error messages understand-ability)	• `Operating_Point.py` l. 52 • `Operating_Point.py` l. 60 • `User_Input.py` l. 106 Target value $B = 3$ Achieved value $A = 2.75$	• `Operating_Point.py` l. 61 • `Operating_Point.py` l. 69 • `User_Input.py` l. 120 • `Reader_H.py` l. 64 • `Reader_P.py` l. 71 Target value $B = 5$ Achieved value $A = 4.75$
UOp-1-G (Operational consistency)	• Preparation of data input • Execution of the flyover tool • Selection of physical modules Target value $B = 3$ Achieved value $A = 2$	• Preparation of data input • Selection of application mode • Selection of physical modules • Execution of post-processing Target value $B = 4$ Achieved value $A = 3$
UOp-2-G (Message clarity)	• Warning: `Propagation.py` l. 45 • Elapsed time Target value $B = 2$ Achieved value $A = 2$	• Warning messages: `Propagation.py` l. 134 `Reader_P.py` l. 82 `Third_Octave_Band.py` l. 78 `Third_Octave_Band.py` l. 96 `User_Input.py` l. 141

		• Elapsed time Target value $B = 6$ Achieved value $A = 6$
UEp-1-G (Avoidance of user operation error)	• `Input_user.json` • `Input_flight_path.xml` • Specification of path to sources • Command-line arguments Target value $B = 4$ Achieved value $A = 0.5$	• `Input_user.json` • `Input_flight_path.xml` • Specification of path to sources • Command-line arguments • Post-processing arguments Target value $B = 5$ Achieved value $A = 3.5$
MMo-1-G (Coupling of components)	• `User_Input.py` • `Reader` classes • `get_p_em_dist()` • `get_doppler()` • `Atmospheric_Absorption.py` • `Ground_Attenuation.py` • `Third_Octave_Band.py` • `Energy_Calculations.py` Target value $B = 8$ Achieved value $A = 5.5$	• `User_Input.py` • `Reader` classes • `get_p_em_dist()` • `get_doppler()` • `Atmospheric_Absorption.py` • `Ground_Attenuation.py` • `Third_Octave_Band.py` • `Energy_Calculations.py` • `Modes_Pre_Processing.py` Target value $B = 9$ Achieved value $A = 6.5$
MRe-1-G (Reusability of assets)	• User documentation • Technical documentation • Testing module • 25 Python modules Target value $B = 28$ Achieved value $A = 21$	• User documentation • Technical documentation • Testing module • 26 Python modules • Post-processing routine Target value $B = 30$ Achieved value $A = 24$
MAn-1-G (System log completeness)	• `debug.log` Target value $B = 1$ Achieved value $A = 0$	• `debug.log` Target value $B = 1$ Achieved value $A = 1$
MTe-1-G (Test function completeness)	• Unit: Atmospheric Absorption • Unit: Ground Attenuation • Unit: dB(A)-weighting • Unit: EPNL • Test: Flyover with PN • Test: Flyover with HEIDI Target value $B = 6$ Achieved value $A = 1$	• Unit: Atmospheric Absorption • Unit: Ground Attenuation • Unit: dB(A)-weighting • Unit: EPNL • Test: Six scenarios (Two for `Developer` with PN and HEIDI) Target value $B = 11$ Achieved value $A = 8$

Based on the defined target values in Tab. 7.1, the software quality of the pre- and post-analysis versions is evaluated. The achieved value A relative to the defined target value B is determined according the score distribution system presented in Sec. 7.2. For a quick and clean comparison, the pre- and post-analysis results are depicted in Tab. 7.2. The result is expressed as a percentage and rounded up to the second decimal place.

Table 7.2.: Comparison of pre- and post-quality-analysis results of VIOLIN using measurement functions proposed in [17]

Measure		Before			After		
ID	Name	A	B	X [%]	A	B	X [%]
UAp-1-G	Description completeness	2	2	100	7	7	100
ULe-1-G	User guidance completeness	3	3	100	6	6	100
ULe-3-S	Error messages understandability	2.75	3	91.67	4.75	5	95
UOp-1-G	Operational consistency	2	3	66.67	3	4	75
UOp-2-G	Message clarity	2	2	100	6	6	100
UEp-1-G	Avoidance of user operation error	0.5	4	12.50	3.5	5	70
MMo-1-G	Coupling of components	5.5	8	68.75	6.5	9	72.22
MRe-1-G	Reusability of assets	21	28	75	24	30	80
MAn-1-G	System log completeness	0	1	0	1	1	100
MTe-1-G	Test function completeness	1	6	16.67	8	11	72.73

Overall, an improvement of software quality is evident. This conclusion is made based on the average score of the result X of both versions. The average score increases from 63% to 87%. That equals a difference of 24%.

[1] The option to turn off the ground reflection module of the flyover simulation.
[2] The option to activate the noise carpet of the flyover simulation.
[3] The option to save all figures of the post-processing routine.
[4] The option to hide all figures of the post-processing routine.

7.4. Evaluation of the software tool CORAL

This section assesses the software quality of CORAL. For this purpose, the target values for each measure are defined for both versions in Tab. 7.3.

Table 7.3.: Target values of the measures for the pre- and post-quality-analysis versions of CORAL

Measure	Before	After
UAp-1-G (Description completeness)	• Auralization of dummy input • Auralization of VIOLIN input Target value $B = 2$ Achieved value $A = 0$	• Five application scenarios[1] Target value $B = 5$ Achieved value $A = 5$
ULe-1-G (User guidance completeness)	• Run (default) application Target value $B = 1$ Achieved value $A = 0$	• Run in default mode • Run in debug mode • Run in post mode Target value $B = 3$ Achieved value $A = 3$
ULe-3-S (Error messages understand-ability)	• `Func_Read_Input.py` l. 20 • `Func_Read_Input.py` l. 26 Target value $B = 2$ Achieved value $A = 1$	• `Func_Read_Input.py` l. 33 • `Func_Read_Input.py` l. 39 • `Func_Read_Input.py` l. 44 • `Func_Read_Input.py` l. 50 • `Settings.py` l. 60 Target value $B = 5$ Achieved value $A = 3.75$
UOp-1-G (Operational consistency)	• Preparation of data input Target value $B = 1$ Achieved value $A = 1$	• Preparation of data input • Selection of application mode Target value $B = 2$ Achieved value $A = 2$
UOp-2-G (Message clarity)	• Introduction message • Three calculation steps • Three intermediate values • Energy calculations Target value $B = 8$ Achieved value $A = 8$	• Introduction message • Three calculation steps • Warning: `Interpolations.py` l. 13 • Elapsed time Target value $B = 6$ Achieved value $A = 6$
UEp-1-G (Avoidance of user operation	• Specification of path to sources Target value $B = 1$ Achieved value $A = 0$	• Specification of path to sources • Command-line arguments • `User_Input.json`

error)		Target value $B = 3$ Achieved value $A = 1.5$
MMo-1-G (Coupling of components)	• `Calc_Pre_Processing.py` • `Calc_Convert_Spectrogram_Time_Signal.py` • `Calc_Generate_Audio.py` • `Spectrogram.py` • `Time_Signal.py` • `Reader` classes Target value $B = 6$ Achieved value $A = 5.5$	• `Calc_Pre_Processing.py` • `Calc_Convert_Spectrogram_Time_Signal.py` • `Calc_Generate_Audio.py` • `Spectrogram.py` • `Time_Signal.py` • `Reader` classes • `Func_Read_Input.py` Target value $B = 7$ Achieved value $A = 6.5$
MRe-1-G (Reusability of assets)	• Technical documentation • 14 Python modules Target value $B = 15$ Achieved value $A = 7$	• User documentation • Technical documentation • Testing module • 18 Python modules Target value $B = 21$ Achieved value $A = 13$
MAn-1-G (System log completeness)	• `debug.log` Target value $B = 1$ Achieved value $A = 0$	• `debug.log` Target value $B = 1$ Achieved value $A = 1$
MTe-1-G (Test function completeness)	• Unit: Generate time signal from tonal spectrogram • Unit: Generate time signal from broadband spectrogram • Test: Auralization with dummy • Test: Auralization with VIOLIN input Target value $B = 4$ Achieved value $A = 0$	• Unit: Generate time signal from tonal spectrogram • Unit: Generate time signal from broadband spectrogram • Unit: Six interpolation functions • Test: Five scenarios (Two for `Developer` with PN and HEIDI) Target value $B = 14$ Achieved value $A = 12$

Based on the defined target values in Tab. 7.3, the software quality of the pre- and post-analysis versions is evaluated. The achieved value A relative to the defined target value B is determined according the score distribution system presented in Sec. 7.2. For a quick and clean comparison, the pre- and post-analysis results are depicted in Tab. 7.4. The result is expressed as a percentage and rounded up to the second decimal place.

[1] The application scenario regarding the noise carpet is exclusive to VIOLIN (see Subsec. 5.2.5).

Table 7.4.: Comparison of pre- and post-quality-analysis results of CORAL using measurement functions proposed in [17]

Measure		Before			After		
ID	Name	A	B	X [%]	A	B	X [%]
UAp-1-G	Description completeness	0	2	0	5	5	100
ULe-1-G	User guidance completeness	0	1	0	3	3	100
ULe-3-S	Error messages understandability	1	2	50	3.75	5	75
UOp-1-G	Operational consistency	1	1	100	2	2	100
UOp-2-G	Message clarity	8	8	100	6	6	100
UEp-1-G	Avoidance of user operation error	0	1	0	1.5	3	50
MMo-1-G	Coupling of components	5.5	6	91.67	6.5	7	92.86
MRe-1-G	Reusability of assets	7	15	46.67	13	21	61.90
MAn-1-G	System log completeness	0	1	0	1	1	100
MTe-1-G	Test function completeness	0	4	0	12	14	85.71

Overall, an improvement of software quality is noticeable. The average score increases from 39% to 87%. This is a difference of 48%.

8. Summary & Discussion

In this chapter, the results of this thesis are summarized and critically analyzed. The chapter is structured as follows:

- In Sec. 8.1, the implementations proposed in Ch. 6 are summarized and checked against the defined requirements.
- In Sec. 8.2, the evaluation results of Ch. 7 are summarized and the key observations are determined for the discussion.
- In Sec. 8.3, the informative value of the measures is discussed with respect to the implementation results and the analyzed software tools VIOLIN and CORAL.

8.1. Requirements and implementation results

In Tab. 5.1, the general user needs for the software are further explicated in regard of the quality characteristics usability and maintainability using the Requirements-Properties-Matrix. Based on these requirements, five implementation concepts are proposed in Ch. 6. The implementations are summarized in the following:

1. The implementation of application modes is a direct specification of requirement A (Different features according to different needs). In the scope of this thesis, two application modes are defined (see Sec. 6.1):

    - The release mode is the default mode and provides a clean and simple execution of the software tool. This includes the standard output of the tool.
    - The debug mode is an extension to the release mode and is intended for developers to spot errors more easily. In addition to the standard output, further information is provided for a deeper insight.

    To conform to the requirements, the application modes are implemented using the `logging` module of Python. This module provides different tracking levels and complementary functions for each level, enabling easy addition of other application modes. The level is chosen by the user before the execution via command-line arguments. By default, the software is run in the release mode, which does not require any command-line specification.

2. The testing framework is a direct specification of requirement C (Software testing) and affected by requirement B (Standardization within and among the tools). In accordance with the Testing Pyramid principle (see Subsec. 2.3.1), the test suite of VIOLIN and CORAL comprise of the following (see Sec. 6.2):

    - The integration tests appraise the typical application scenario.
    - The unit tests appraise individual physical modules.

    Both types of tests follow the Golden Master Testing method (see Subsec. 2.3.2). The testing framework provides execution scripts to automatically run all test cases of each type of test. The test cases are specified in a file named `projects.txt` of the respective directory. This way, test cases can be added or removed easily. If a change is detected and accepted, the golden files can be updated automatically via another script.

3. For standardization (requirement B), documentation guidelines are introduced for the user and technical documentation of the software (see Sec. 6.3):

    - The guideline for the user documentation specifies what pieces of information the user document, the `README.md` of the repository, must include (see Subsec. 6.3.1).
    - The guideline for the technical documentation specifies the documentation style of the source code with respect to the tools `pdoc` and `pyreverse` for automatic documentation (see Subsec. 6.3.2).

4. The GitLab CI/CD pipeline is designed for automated software testing and automated, dynamic generation of technical documentation, resulting from requirements of B and C (see Sec. 6.4). It is divided into the stages containing the software testing jobs and the stage for dynamic documentation. The technical documents are only generated, if the jobs of the software testing stages succeeded.

5. A post-processing routine is implemented to plot the software results (see Sec. 6.5). This implementation originates from the plot mode (see Sec. 6.1). In regard of requirement DU1, the post-processing routine is designed for the user to easily adjust the parameters that can change the appearance of a plot. This means the user can customize the graphs without having to know about the details of the plotting implementation. Furthermore, the post routine offers the advantage that the appearance of the plots can be changed without having to run the tool again.

In Tab. 8.1, the results are checked against the requirements defined in Tab. 5.1.

Table 8.1.: Comparison of the proposed implementations to the defined software requirements

ID	Requirement	Implementations conforming to requirement
AU1	Implement at least two application modes with different outputs, i.e. user mode and developer mode.	The following modes are implemented: • Release mode • Debug mode
AU2	Allow user to specify application mode at execution.	By default, the release mode is run. Other modes are set via command-line arguments.
AM1	Implement a framework to easily add or remove a mode.	The `logging` module provides different tracking levels and complementary functions.
AM2	Implement modes without interference of other modes.	Lazy evaluation enables the evaluation of an expression of the selected mode only.
AM3	Integrate output of different modes without affecting the clarity of the code.	The `logging` module provides built-in functions for each level. Therefore, `if`-clauses can be avoided.
AM4	Generate output files in each mode for testing.	The `logging` module automatically generates a logging file.
BU1	Standardize code documentation style.	The documentation guideline is applied to VIOLIN and CORAL.
BU2	Standardize software testing	The testing framework is applicable to VIOLIN and CORAL.
BM1	Define documentation guidelines for automatic rendering.	The technical documentation is developed with regard to the use of `pdoc` and `pyreverse`
BM2	Design reusable software testing framework.	The individual test scripts are usable for both types of integration tests.
CU1	Allow user to easily add and remove test cases.	All active test cases are listed in the user-definable projects.txt.
CU2	Allow user to manually execute software testing.	All software tests can be run with the respective run execution scripts.
CM1	Implement a testing framework including integration and unit tests.	The testing framework implements integration and unit tests.
CM2	Integrate testing to the development process.	The software tests are integrated in the GitLab CI/CD pipeline.

DU1	Allow user to easily define relevant input parameters.	The post mode is changed into a post-processing routine in order for the scientists to easily change parameters of the graphs after the execution without repetitive execution of the software.
DU2	Notify user of error.	New error messages are implemented in the scope of this thesis (not included in the proposed implementation concepts).
DM1	Provide default input in case of invalid or no user input.	Default values are implemented in the scope of this thesis (not included in the proposed implementation concepts).
EU1	Provide each application scenario for the user.	Each application scenario is represented by a test case.
EM1	Implement at least one test case for each scenario.	The testing framework includes integration tests for each scenario.

As mentioned before, the requirements were formulated with respect to the quality characteristics usability and maintainability. The proposed implementations satisfy all the defined requirements. Therefore, a great improvement of software quality was expected. The implementation results are discussed in conjunction with the evaluation results in Sec. 8.3.

8.2. The evaluation results

The results of the implementations are evaluated using selected measurement functions of ISO/IEC 25023 (see Ch. 7) and are compared to a pre-analysis version of the software tool. Based on the average scores, an overall improvement of software quality is evident. For further insight, the evaluation results are visualized as bar charts as follows:

- The evaluation results of VIOLIN depicted in Tab. 7.2 are illustrated in Fig. 8.1.
- The evaluation results of CORAL depicted in Tab. 7.4 are illustrated in Fig. 8.2.

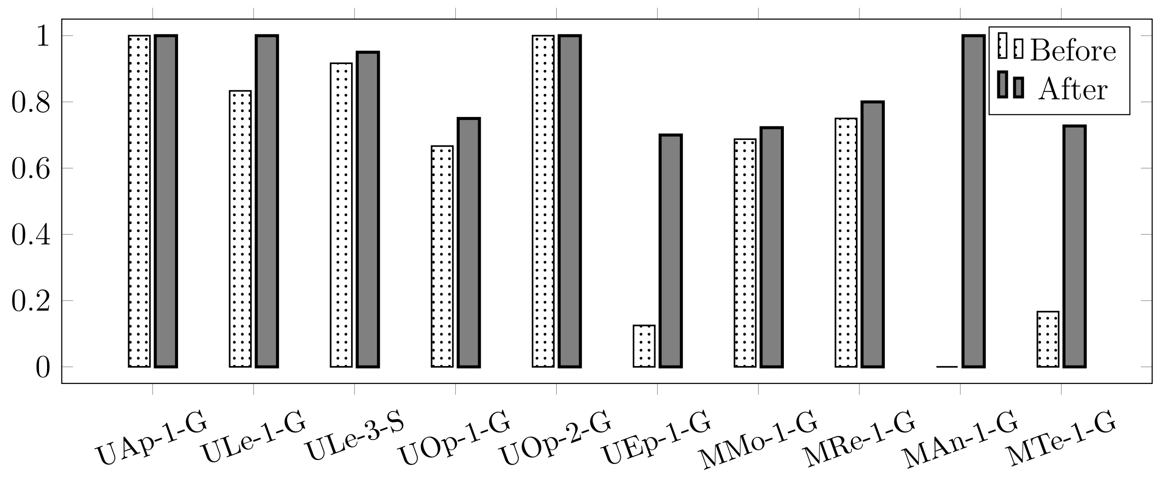


Figure 8.1.: Software quality evaluation results of VIOLIN

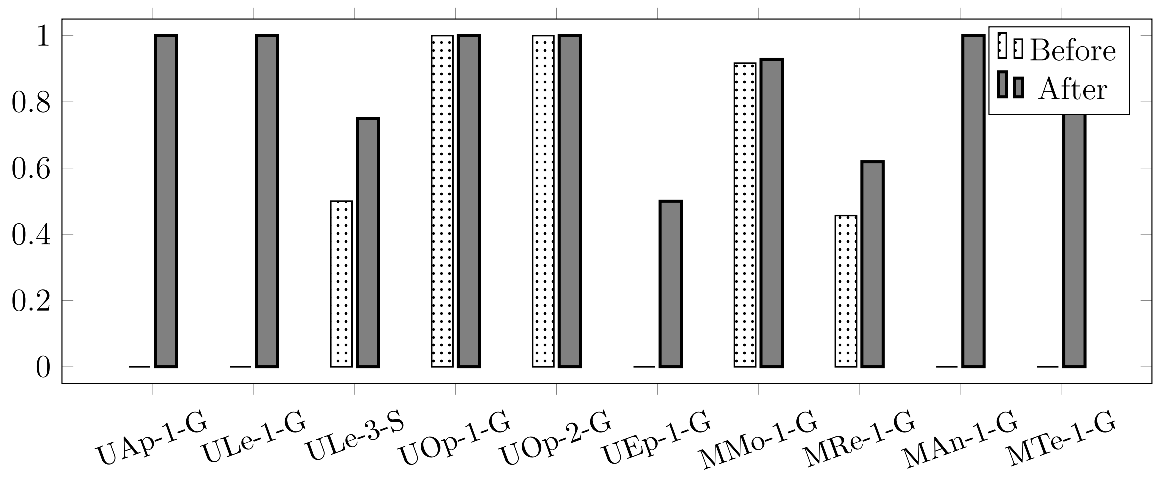


Figure 8.2.: Software quality evaluation results of CORAL

The following key observations or questions are concluded from Fig. 8.1 and Fig. 8.2:

1. In VIOLIN and CORAL, evident improvement is observed in the measurement results of UEp-1-G, MAn-1-G and MTe-1-G.
2. Why is an improvement from 0% to a perfect score of a 100% a recurring observation in the evaluation results?
3. Why is a perfect score achieved so often?
4. Why do some measures indicate little to no change in quality?
5. Why do the pre-analysis versions fare relatively well, even though no thought was given to usability and maintainability yet?

These questions are addressed and answered in the following section.

8.3. Interpretation and discussion

In general, the overall result of the evaluation meets the expectation formulated in Sec. 7.1: The implementation concepts designed in the interest of software quality contribute favorably to the overall software quality of both tools. This is particularly noticeable for the measurement functions that are directly affected by the implemented changes as pointed out in observation Q. 1. The observation is explained as follows:

- For the measure UEp-1-G (Avoidance of user operation error), an absolute improvement of 50-60% is observed. This is a direct result of requirement D defined in the overall Requirements-Properties Matrix (see Tab. 5.1). Not only does this requirement stipulate the interception of user input errors to notify the user but also inquires the definition of default values for such a case. While this is mostly implemented in the scope of this thesis, the pre-analysis software versions do not consider this at all. Therefore, the evident improvement is perceived in this area.
- An improvement from 0% to 100% is observed for the measure MAn-1-G (System log completeness). This is simply explained by the transition from no auditing to the logging of the energy calculations.
- A similar observation is made for the measure MTe-1-G (Test function completeness). In this case, the evident improvement is owed to the initial situation of the software testing. Software tests were either outdated or not implemented at all. If the unit tests were properly realizable, an improvement from 0% to 100% would have been observed for this measure as well. The problem with the unit tests is discussed later.

The listed observations were made for both software tools. Most of the quality improvement stems from these measures. Additionally, CORAL is significantly impacted by the introduction of documentation guidelines in Sec. 6.3. This is reflected in the measures UAp-1-G (Description completeness) and ULe-1-G (User guidance completeness). The obvious improvement results from the general lack of documentation in the pre-analysis version of CORAL, while the new version provides a detailed user and technical documentation.
From the observations mentioned so far, question Q. 2 and Q. 3 arise. Improvements from 0% to 100% and/or perfect scores seem to be a trending observation in the evaluation. Logically, a perfect score is easily achievable if a proportion to a small target value is calculated, answering question Q. 3. A concrete example is the measure MAn-1-G (System log completeness). This can happen if the quality aspect is not applicable and the software consequently requires or contains a small target value of that aspect. Furthermore, this means that the number of target values is not reflected in the evaluation score as it describes a proportion. An increase of target values does not show in the score. The informative value of the measure is arguable. This sparks the following discussion:

Initially, the complementary detailed set of measures of ISO/IEC 25023 was a determining factor for the selection of quality model proposed in ISO/IEC 25010, whereas McCall's model, for example, was criticized for the choice of a binary measure (see Subsec. 2.2.5). In retrospect, the evaluation in the scope of this thesis reveals that the measures of ISO/IEC 25023 are not quite exempt from this criticism. While the measures are introduced in the standard to produce a result that is relative to a target value that is established as a requirement, this is ultimately equivalent to a checklist on a finer scale in the case of some measures. This calls the informative value of the measures in question. For example, the measure UAp-1-G (Description completeness) is defined as the proportion of usage scenarios described in the user documents relative to the actual number of usage scenarios. Due to the lack of definition on the degree of description to a usage scenario, this is simply a checklist of existing documentation for every usage scenario. This is similarly the case for the measures ULe-1-G, UOp-1-G, UOp-2-G, UEp-1-G, MMo-1-G and MRe-1-G. A good counterexample is the measure ULe-3-S (Error messages understandability). While this function does calculate a proportion, a further specification is given to determine the degree of understandability of an error message. In other words, this measure reflects the proportion of the error messages that state the reason of occurrence and a way to resolve this error. The difference to the measure UAp-1-G is the contrast of meaning between proportion and degree:

- A measure of proportion like UAp-1-G counts the occurrence of a certain aspect that is generally accountable for (good) software quality like in measure.
- A measure of degree like ULe-3-S determines the actual degree of quality for each individual occurrence of an aspect.

Contributing to this distinction is the definition of the target value in the measure. For example, the measures UAp-1-G and ULe-1-G appear to be similar at first glance: Both seemingly count the occurrence of documentation for certain software aspects. However, the definition of the target value ultimately decides the type of measure:

- The target value of UAp-1-G is defined as the number of described usage scenarios.
- The target value of ULe-1-G is defined as the number of functions that require to be documented and are explained in sufficient detail.

The first is a simple case of counting the occurrences of something. The latter is defined to be individual to each software. This prompts the evaluator to further specify the target value in order to properly measure the degree of a quality aspect. In this sense, it is reasonable to propose to further define the target value of all measures before the evaluation, especially measures of proportion, to raise the informative value of the measurement functions.

For example, the following distinctions can be introduced for measures of proportion:

- For measure UAp-1-G (Description completeness), the evaluator can specify what pieces of information the documentation of a usage scenario must provide. One achievable point is distributed into the number of information pieces.
- Measure ULe-1-G (User guidance completeness) can be further specified like measure UAp-1-G.
- For measure UOp-1-G (Operational consistency), the operational consistency of each interactive task and what has to apply for this to be achieved can be defined.
- Measure UOp-2-G (Message clarity) can be further specified similarly to measure ULe-3-S. A message is clear if the message states the subject and if the conveyed content is correct.
- For measure UEp-1-G (Avoidance of user operation error), the required steps to protect user actions can be specified.
- For measure MMo-1-G (Coupling of components), it can be specified what constitutes the independence of a component by defining a few scenarios where the component should not have impact on other components.
- Similarly to measure MMo-1-G, the reusability of an asset can be further specified for measure MRe-1-G (Reusability of assets).

However, the further definition of measures of proportion is not only important to properly evaluate the degree of a quality aspect but also would resolve the questions Q. 4 and Q. 5. This is because the measures of proportion are the cause for the improvements from 0% to 100% as well as for the little to unchanging quality values. A concrete example is found in the evaluation results of the measures UAp-1-G and ULe-1-G for VIOLIN and CORAL:

- In CORAL, an improvement of quality from 0% to 100% is observed as pointed out in Q. 2 because of the transition from lacking documentation to providing a detailed user and technical documentation. The evaluation results of these measures only reflect if something is documented or not.
- In VIOLIN, little to no change is observed as well as the fact that the pre-analysis version performs well. The reason remains the same: The evaluation results of these measures only reflect if something is documented or not. It is not considered whether the depth of detail and consequently degree of quality increased with the proposed documentation guidelines. For this reason, the tools score relatively well before and the proposed implementations designed to improve the assessed characteristics have less of a impact on the scores than expected, answering Q. 4 and Q. 5.

The informative value of the selected measures of ISO/IEC 25023 is therefore questionable. This conclusion is further supported by the overall user feedback to the new implementations. A greater improvement is noted by the users than the evaluation results based on measures. This statement has to be further analyzed by means of a user study to substantially confirm the quality improvement.

Irrespective of the questionable informative value of the measures, the implementations and evaluation results reveal a general flaw in the software tools VIOLIN and CORAL: the lacking modularity of the components within a Python module. In Sec. 7.2, this is noted as a sub-requirement for the measure MMo-1-G (Coupling of components) aside from the independence between the modules. This specification was added during the implementation process as it turned out that this aspect was crucial for the unit tests. Before the consideration of software quality, the focus was on a clean and modular software structure, especially regarding the physical modules. As a result, these modules are designed to be independent. However, the individual routines within are linked together. Therefore, it was not possible to implement most of the unit tests as they assessed such routines. This problem is reflected in the results of the measure MTe-1-G (Test function completeness). While a detailed framework exists for the required integration and unit tests, a majority of the unit tests are not realizable resulting in an incomplete test coverage.

Also related to the software testing, the following issue was observed in the scope of this thesis: The format of the `debug.log` file which is produced to test the debug mode is not practical for assertion-based testing. Aside from the intermediate values, the `debug.log` contains the affected source code line. During the implementation and evaluation phase, this caused wrongful failing unit tests. The reason is the changing number of code line when the source code was edited. Thus, when comparing the `debug.log` to the golden master, the information for debugging is correct but the number of source code line might be different, resulting in a failed test.

In conclusion, a critical analysis of the implementation and evaluation results of VIOLIN and CORAL reveals that the proposed implementations developed in interest of usability and maintainability satisfy the defined requirements but an improvement may not be properly reflected by the selected measures. This requires further examination by means of a user study as well as a re-evaluation using the same measures but with further specifications to assess the degree of software quality.

9. Recommendations for future work

The critical analysis of the proposed implementations and of the measures of the ISO/IEC 25023 used in the scope of this thesis have indicated the following points for further work:

- A user study could be conducted to further analyze the evaluation results of the proposed implementation. The study could include the small group of currently active users of the analyzed software tools. In the study, the quality aspects assessed in the measures that are used in this thesis could be rated by the users according to a specified scale. This would provide a qualitative opinion on the quality aspects of usability and maintainability and is a great opportunity to compare quantitative quality measures to a qualitative quality assessments. Furthermore, the user study could be used to introduce the modifications and guidelines and find out the users' willingness to adhere to the changes.

- A re-evaluation could be performed using the same measures but with more detailed specifications regarding the target value completely. For this purpose, a set of specific characteristics could be defined for each measure before the evaluation that are required to achieve the target value. In doing so, measures of proportion could be transformed in to measures of degree. This step could increase the informative value of the measures on the degree of software quality. To confirm such improvement, the new results could be compared to the results of this thesis and a user study.

- A re-evaluation could be performed using the applicable measures of the type Specific (S) in addition to the Generic (G) measures. This way, a quality characteristic could be assessed more thoroughly, resulting in a different score. The new results could be compared to the results of this thesis and a user study to determine if the informative value changed. In this regard, this point could be combined with the previous one to obtain an even more detailed assessment of the software quality.

- An evaluation using the time-dependent measures MMd-1-G (Modification efficiency) and MMd-2-G (Modification correctness) that were removed from the evaluation of this thesis due to the limited time could be performed. For this purpose, the implementation for the distributed engines and/or the noise carpet in VIOLIN could be tested. Both modules are highly prone to errors so far. This is because these implementations are relatively new and require a certain degree of modification to the software. Therefore, these modules are perfectly suited for the modifications that are tested by the measures MMd-1-G and MMd-2-G.

- The measure MAn-1-G (System log completeness) could be used in the future to assess the computation efficiency of the tool. This could be of relevance when an optimization is conducted and would require the tracking of computation time of each routine. As a result, at least one more log exists next to the `debug.log`. This could change the evaluation results, especially if the target values is further specified.

- The modularity within the Python modules of both software tools could be improved. This would enable the implementation of all defined unit tests and more. For this purpose, the SOLID-principles [32] could be followed to achieve a more flexible and modular design. Afterwards, a re-evaluation using the same measures could be conducted and the new results could be compared to the results of this thesis. In this case, an improvement in the results of the measures MMo-1-G (Coupling of components) and MTe-1-G (Test function completeness) would be expected.

- The format of the `debug.log` file could be adjusted to be more suitable for testing purposes. This would require the removal of the affected code line, however consequently would affect the readability of the log file. Alternatively, the assertion-based testing functions of this file could be adjusted to ignore the code lines during the comparison to the golden master. This way, the test case would only fail if the actual assessed intermediate values have changed and not the number of source code line.

- A different quality model and consequently other quality measures could be applied and compared to the ISO/IEC 25010 [16] and ISO/IEC 25023 [17]. The other models presented in this thesis could be considered. These would be the models of Boehm [11] and McCall [14]. As explained in the literature review (see Subsec. 2.2.5), both models are included in the model of ISO/IEC 25010. Therefore, it would be interesting to examine if this is reflected in the respective evaluation results and confirm that ISO/IEC 25010 consolidates both models. Also interesting would be the critical analysis of the informative value of those models in comparison to the model used in this thesis.

- To broaden the software quality assessment, the software tools VIOLIN and CORAL could be evaluated in regard of the other quality characteristics of the ISO/IEC 25010 model. This includes functional suitability, performance efficiency, compatibility, reliability, security and portability. Furthermore, modifications to the software could be designed and proposed to improve these characteristics. As mentioned in an earlier point, an optimization of the calculation processes could be conducted. To evaluate this, the characteristic performance efficiency could be inspected. Furthermore, portability could be assessed and improved because the group of scientist uses the software tools in different environments.

Acknowledgements

At last, I would like to thank everyone who supported me during my bachelor thesis.

Words cannot express my gratitude to my supervisor Stephen Schade for his invaluable feedback and support. He helped me and was always there for me regarding structure- and content-related questions. I could not have undertaken this journey without him.

Special thanks go to my supervisor Prof. Dr. Holger Gerhards of the Baden-Wurttemberg Cooperative State University for his constructive feedback.

I would also like to thank Dr.-Ing. Antoine Moreau for his formal corrections.

Lastly, I would like to express my deepest appreciation to my fellow student Joel Winiecki who supported me in many situations until the end and kept my spirits high.

Bibliography

[1] *Das DLR im Überblick.* URL: https://www.dlr.de/DE/organisation-dlr/das-dlr/dlr-im-ueberblick.html (visited on 06/07/2022).

[2] *Institut für Antriebstechnik.* URL: https://www.dlr.de/at/desktopdefault.aspx/tabid-1490/2091_read-3604/ (visited on 06/07/2022).

[3] *Abteilung Triebwerksakustik.* URL: https://www.dlr.de/at/desktopdefault.aspx/tabid-1521/2269_read-3715/ (visited on 06/07/2022).

[4] Patrik Berander et al. "Software quality attributes and trade-offs". In: *Blekinge Institute of Technology* 97.98 (2005), p. 19.

[5] Philip B Crosby. *Quality is free: the art of making quality certain.* McGraw-Hill, 1979.

[6] Armand V Feigenbaum. *Total quality control.* McGraw-Hill, 1983.

[7] Kaoru Ishikawa. *What is total quality control?: the Japanese way.* Prentice-Hall, 1985.

[8] William E Deming. *Out of the crisis: quality, productivity and competitive position.* Cambridge Univ. Press, 1988.

[9] Walter A Shewhart. *Economic control of quality of manufactured product.* Van Nostrand, 1931.

[10] Joseph M Juran. *Joseph's Quality Control Handbook.* McGraw-Hill, 1988.

[11] Barry W Boehm, John R Brown, and Mlity Lipow. "Quantitative evaluation of software quality". In: *Proceedings of the 2nd international conference on Software engineering.* 1976, pp. 592–605.

[12] Euler Marinho and Rodolfo Resende. "Quality Factors in Development Best Practices for Mobile Applications". In: June 2012, pp. 632–645. ISBN: 978-3-642-31127-7. DOI: 10.1007/978-3-642-31128-4_47.

[13] Daniyal Farooque. *Boehm's Software Quality Model.* URL: https://www.geeksforgeeks.org/boehms-software-quality-model/ (visited on 07/16/2022).

[14] Jim A McCall, Paul K Richards, and Gene F Walters. *Factors in software quality. volume i. concepts and definitions of software quality.* Tech. rep. GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

[15] ISO/IEC 9126-1. *Software engineering — Product quality — Part 1: Quality model.* Standard ISO/IEC 9126:2001(E). Geneva, CH: International Organization for Standardization, June 2001.

[16] ISO/IEC 25010. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.* Standard ISO/IEC 25010:2011(E). Geneva, CH: International Organization for Standardization, Mar. 2011.

[17] ISO/IEC 25023. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality.* Standard ISO/IEC 25023:2016(E). Geneva, CH: International Organization for Standardization, June 2016.

[18] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum.* A Mike Cohen signature book. 2009.

[19] J B Rainsberger. *Surviving Legacy Code with Golden Master and Sampling.* URL: https://blog.thecodewhisperer.com/permalink/surviving-legacy-code-with-golden-master-and-sampling#welc (visited on 07/20/2022).

[20] Mike C Feathers. *Working Effectively with Legacy Code: WORK EFFECT LEG CODE.* Robert C. Martin Series. Pearson Education, 2004. ISBN: 9780132931755.

[21] Andrew Forward. "Software documentation: Building and maintaining artefacts of communication". MA thesis. University of Ottawa (Canada), 2002.

[22] *GitLab CICD.* URL: https://docs.gitlab.com/ee/ci/ (visited on 07/02/2022).

[23] Ritesh Ranjan. *What is a Framework in Programming & Why You Should Use One.* URL: https://www.netsolutions.com/insights/what-is-a-framework-in-programming/ (visited on 07/13/2022).

[24] *logging — Logging facility for Python.* URL: https://docs.python.org/3/library/logging.html (visited on 07/21/2022).

[25] *unittest — Unit testing framework.* URL: https://docs.python.org/3/library/unittest.html (visited on 07/21/2022).

[26] *pdoc.* URL: https://pdoc.dev/docs/pdoc.html (visited on 07/21/2022).

[27] *Pyreverse.* URL: https://pylint.pycqa.org/en/latest/pyreverse.html (visited on 07/21/2022).

[28] Maikhanh Dang. *Development of two modular Python tools for virtual acoustic flyover simulation with subsequent auralization of the simulated sound fields.* Report of practical phase 2. 2020.

[29] Pierre Bourque and Richard E Fairley, eds. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Version 3.0. IEEE Computer Society, 2014. URL: http://www.swebok.org/.

[30] Maikhanh Dang. *Identification of user needs and definition of typical application scenarios for the software tools VIOLIN and CORAL*. Tech. rep. Baden-Wuerttemberg Cooperative State University Mannheim, 2022.

[31] Maikhanh Dang. *Umsetzung eines modularen Python-Tools zur Simulation von Überfluglärm und der Lärmbewertung anhand verschiedener Metriken*. Tech. rep. Baden-Wuerttemberg Cooperative State University Mannheim, 2020.

[32] Robert C Martin. *The Principles of OOD*. URL: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (visited on 07/26/2022).

A. Quality measures of the ISO/IEC 25023 standard

This chapter contains a selection of measures proposed in ISO/IEC 25023 including the respective measures functions.

A.1. Usability measures

A selection of measures for the following usability subcharacteristics are depicted:

- Table A.1 shows selected measures for appropriateness recognizability.
- Table A.2 shows selected measures for learnability.
- Table A.3 shows selected measures for operability.
- Table A.4 shows selected measures for user error protection.

Table A.1.: Selected appropriateness recognizability measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
UAp-1-G	Description completeness	$X = A/B$ $A =$ Number of usage scenarios described in the product description or user documents $B =$ Number of usage scenarios of the product

Table A.2.: Selected learnability measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
ULe-1-G	User guidance completeness	$X = A/B$ A = Number of functions described in user documentation and/or help facility as required B = Number of functions implemented that are required to be documented
ULe-3-S	Error messages understandability	$X = A/B$ A = Number of error messages which state the reason of occurrence and suggest the ways of resolution where this is possible B = Number of error messages implemented

Table A.3.: Selected operability measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
UOp-1-G	Operational consistency	$X = 1 - A/B$ A = Number of specific interactive tasks that are performed inconsistently B = Number of specific interactive tasks that need to be consistent
UOp-2-G	Message clarity	$X = A/B$ A = Number of messages that convey the right outcome or instructions to the user B = Number of messages implemented

Table A.4.: Selected user error protection measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
UEp-1-G	Avoidance of user operation error	$X = A/B$ A = Number of user actions and inputs that are protected from causing any system malfunction B = Number of user actions and inputs that could be protected from causing any system malfunction

A.2. Maintainability measures

A selection of measures for the following maintainability subcharacteristics are depicted:

- Table A.5 shows selected measures for modularity.
- Table A.6 shows selected measures for reusability.
- Table A.7 shows selected measures for reusability.
- Table A.8 shows selected measures for modifiability.
- Table A.9 shows selected measures for testability.

Table A.5.: Selected modularity measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
MMo-1-G	Coupling of components	$X = A/B$ A = Number of components which are implemented with no impact on others B = Number of specified components which are required to be independent

Table A.6.: Selected reusability measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
MRe-1-G	Reusability of assets	$X = A/B$ A = Number of assets which are designed and implemented to be reusable B = Number of assets in a system

Table A.7.: Selected analyzability measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
MAn-1-G	System log completeness	$X = A/B$ A = Number of logs that are actually recorded in the system B = Number of logs for which audit trails are required during operation

Table A.8.: Selected modifiability measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
MMd-1-G	Modification efficiency[1]	$X = \sum_{i=1}^{n}(A_i/B_i)/n$ A = Total work time spent for making a specific type of modification i B = Expected time for making the specific type of modification i n = Number of modifications measured
MMd-2-G	Modification correctness	$X = 1 - A/B$ A = Number of modifications that caused an incident or failure within a defined period after being implemented B = Number of modifications implemented

Table A.9.: Selected testability measures of ISO/IEC 25023 [17]

ID	Name	Measurement function
MTe-1-G	Test function completeness	$X = A/B$ A = Number of test functions implemented as specified B = Number of test functions required

[1] X greater than 1 represents inefficient modifications and X less than 1 represents very efficient modifications

B. Appendix to software testing

This chapter contains additional data and material regarding the framework `unittest` including methods of `unittest` as well as some implementation snippets of test cases with `unittest`. The integration of the test suite is also included.

B.1. Methods of the testing framework

Table B.1 depicts commonly used assert methods of the framework `unittest`.

Table B.1.: Commonly used assert methods of the framework `unittest` adapted from [25]

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

B.2. Source code of the general test suite

This section contains code excerpts of the test suite:

- Listing B.1 presents the testing class for the debug mode.
- Listing B.2 presents an excert of a VIOLIN unit test.

```python
import unittest
import filecmp
import sys

path_to_project = sys.argv[1]

class TestDebug(unittest.TestCase):

    def test_debug(self):
        is_equal = filecmp.cmp(f'{path_to_project}/output/debug.log'
            , f'{path_to_project}/reference/debug.log')
        self.assertTrue(is_equal)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], verbosity=0)
```

Listing B.1: Source code of the testing class used for the debug mode

```python
class TestAA(unittest.TestCase):
"""Class used to test the atmospheric absorption routine."""

    def test_tonal(self):
        p_em_dist, f_doppler, r = self._read_input(path_unit_aa+
            input_aa_tonal)
        self._run_routine(p_em_dist, f_doppler, r, result=f'{
            path_unit_aa}/output/tonal.h5')
        self._compare_results('tonal.h5')

    def test_broadband(self):
        p_em_dist, f_doppler, r = self._read_input(path_unit_aa+
            input_aa_bb)
        self._run_routine(p_em_dist, f_doppler, r, result=f'{
            path_unit_aa}/output/broadband.h5')
        self._compare_results('broadband.h5')

    ...
```

Listing B.2: Source code excerpt of a class used for unit tests

B.3. Implementation of the pipeline

Listing B.3 presents an exemplary `.gitlab-ci.yml` file of the general pipeline.

```yaml
stages:
    - build
    - test
    - doc

integration-build:
    stage: build
    only:
        refs:
        - master
        - develop
    script:
        - testsuite/build_integration_tests.sh projects.txt
        - testsuite/move_data.sh projects.txt

developer-integration-build:
    stage: build
    only:
        refs:
        - master
        - develop
    script:
        - testsuite/build_integration_tests.sh projects_debug.txt
        - testsuite/move_data.sh projects_debug.txt

integration-testing:
    stage: test
    only:
        refs:
        - master
        - develop
    script: testsuite/compare_results.sh projects.txt

developer-integration-testing:
    stage: test
    only:
        refs:
        - master
        - develop
    script: testsuite/compare_results.sh projects_debug.txt
    artifacts:
        paths:
        - testsuite/Unit_Tests/Unit_AA/output/debug.log

unit-testing:
    stage: test
    only:
        refs:
        - master
        - develop
    script: testsuite/run_unit_tests.sh

documentation:
    stage: doc
    only:
        refs:
        - master
    script:
        - cd src/
        - rm -rf __pycache__/
        - pdoc -d restructuredtext *.py -o ../documentation/html/
        - cd
        - cd documentation/class_diagram/
        - pyreverse -f'ALL' -o pdf ../../src/*
        - cd
    artifacts:
        paths:
        - documentation/html/*
        - documentation/class_diagram/*
```

Listing B.3: The `.gitlab-ci.yml` file of the pipeline