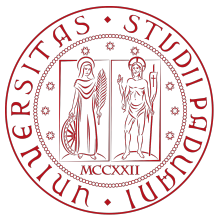

UNIVERSITY OF PADUA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN ICT FOR INTERNET & MULTIMEDIA

Design and Implementation of a Cloud-based Test Executor of Microcontroller Algorithms

Supervisor: Prof. NICOLA FERRO

Co-Supervisor:
Ing. ANDREA BASCHIROTTO
Dott. SERGIO SPINATO
Dott. SERGIO VENDEMMIATI

Candidate: SILVIO MENEGUZZO
Badge: 1206743

21 February 2022
Academic Year 2021-2022

“Learn from yesterday, live for
today, hope for tomorrow.
The important thing is not
to stop questioning”

Albert Einstein

Abstract

This thesis work is the result of an internship in the Microcontroller division of Infineon Padua Development Center.
The aim of this thesis was to carry out a feasibility study for the creation of a specific cloud system, developed as a plugin, which would allow the exchange of test packets used and executed on Jazz (parent tool - Test executor). This work required to study the existing toolchain and the existing tools in particular the tools for generating tests in order to understand the structure of test packets. The intent of Jazz Cloud is to add functionalities for sharing test packets, uploading, downloading and exporting (implementation of a support algorithm to optimize the correct extraction of files). All this by relying on a intuitive and dynamic GUI (Graphic User Interface). Once the code was developed, and its correctness was proved, it was committed and made available to all the team. The access point to Jazz Cloud and its additional features, are reachable through Jazz external tools. Jazz is a tool used all over the world within Infineon Technologies, and Jazz Cloud aims to increase its diffusion by making it easier to use.

Sommario

Questo lavoro di tesi è il risultato del tirocinio effettuato nella divisione di Microcontrollori del centro di Ricerca e Sviluppo della sede di Padova di Infineon Technologies.
Lo scopo di questa tesi era quello di svolgere uno studio di fattibilità per la creazione di uno specifico sistema cloud, sviluppato come plugin, che permettesse lo scambio di pacchetti di test utilizzati ed eseguiti su Jazz (tool padre - esecutore di Test). Per svolgere questo lavoro è stato necessario studiare la toolchain e i tool esistenti, in particolare i tool di generazione dei Test per poter capire la struttura di quest'ultimi. L'intento di Jazz Cloud è quello di aggiungere delle funzionalità di condivisione dei flussi di Test, di upload, di download; con il supporto di un Exporter (implementazione di un algoritmo di supporto per otimizzare l'estrazione corretta dei files), tutto ciò appoggiandosi a una GUI intuitiva e dinamica. Una volta che il codice è stato sviluppato e testato, è stato reso disponibile a tutto il team. L'accesso a Jazz Cloud e alle funzionalità che esso mette a disposizione, avviene tramite l'interfaccia grafica di Jazz, più nello specifico, nella sezione external tools.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The aim of this chapter is to introduce the company as well as the team in which this thesis was conceived and then developed. It explains the task assigned and its importance within the team. Then, in conclusion, a first look of the structure of the work is presented in order to have an initial approach on the main topics of the thesis.

1.1 Infineon Technologies

Ranked one of the global top 10 semiconductor companies, Infineon Technologies is a world leader in semiconductor solutions. Founded in April 1999, it is headquarted in Munich (Germany) but there are multiple sites located in more than 100 countries: 60 of those are Research and Development centers and 19 of those are Production sites allocated all over Europe, the Americas and the Pacific Regions.

Figure 1.1: Infineon Logo

Combining entrepreneurial success with responsible actions, "Infineon makes life easier, safer and greener". As a matter of fact, Infineon today takes care of four trends:

- Automotive (ATV) : this segment provides semiconductors solutions for automotive applications. This trend powers the transition to hybrid and purely electric vehicles. It supports the next stages of automated driving as well as higher electrification, connectivity, digitalization and security. It drives safety, instrument cluster, infotainment, convenience and lighting innovations.

- Industrial Power Control (IPC) : this division includes leading semiconductors solutions for smart and efficient energy generation, storage, transmission and consumption. The application spectrum goes from photovoltaic installations, to wind turbines and high-voltage

DC transmission system, to charging infrastructures for electric vehicles and finally to trains and home appliances.

- Power & Sensor Systems (PSS) : this division provides sensing solutions such as MEMS microphones, pressure sensors and radar sensors, advanced 3D ToF imagers are bringing "human" senses to IoT devices, allowing in this way them to react to their surroundings. These technologies are combined between them to make human-to-machine and machine-to-machine interaction energy efficient, safe and seamless.

- Connected Secure Systems (CSS): this divisions provide security for the connected world. It provides security solutions that allow to guarantee robust protection of devices, machines, identities, intellectual property and data.

All of them are driven by the four key trends of the company which are: energy efficiency, mobility, security and IoT & big data.
In the 2020 fiscal year (ended 30 September), the company reported sales of 8,567 billion, divided on the four business area as can be seen in Figure 1.2 and counts total 46.700 employees worldwide.

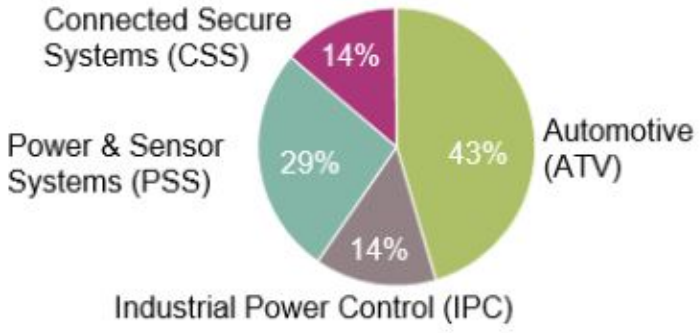

Figure 1.2: Financial Year 2020, Revenue split by segment

Furthermore Infineon keeps close customer relationship based on system know-how and application understanding, see Figure 1.3.

Figure 1.3: Infineon's relationship with customers

1.2 Infineon Technologies Italy

Infineon Technologies Italy S.r.l. was established in 1999 and became operational since 2000 with a sales organization based in Milan. In 2001 a R&D center for Automotive applications opened in Padua. After the acquisition of International Rectifier by Infineon Technologies in 2015, a second Italian R&D center was opened in Pavia. Infineon Italy maintains profitable relationships with universities to ensure consistent levels of knowledge sharing which is absolutely necessary in an industry where technological innovation is the first item of business success. Overall, as a matter of fact, Infineon Italy represents the Infineon Group in the country, is fully integrated in global organization and is a complete, cohesive entity in its own right.

1.2.1 Automotive MC team Padua

The Automotive MC team is the center of competence for embedded Non Volatile Memories (NVM) analysis/characterization and test methodologies.
NVM memory is about 50% chip area, therefore chip quality is directly dominated by NVM quality.
The MC team ensure eFlash product functionality, robustness and quality providing:

- Embedded Flash NVM robustness validation (analysis and characterization)
- TestWare for validation, analysis, qualification and production test: deliver ready to use Flash Test Suite for available Test Platforms
- tools and methodologies for validation, analysis, characterization and production test

To provide these tasks Padua's MC team is divided into three sub-teams:

Test Engineering: this sub-team develops Testware (Firmware for Testing) embedded software for analysis and testing of production. The quality, which is developed primarily for production tests, is in focus in order to reach the automotive quality standard requested.

Product Engineering: this sub-team characterizes Embedded Flash and validates robustness of non-volatile memories. The main purpose is to ensure the quality of the memory through the validation of the customers requirements, statistical analysis and by extending the validation to many samples. The group takes care of the analysis, searches the causes that generate the problem to solve and therefore offers solutions for the designers and technology experts. The characterization activities try to push to the limit parameters like the temperature, power and system frequencies.
This group is involved in automotive micro-controller development activities, from first silicon analysis, to massive production support, through customer validation, product qualification and testing support.

Technical Marketing and Concept Engineering : This sub-team understand the customers applications and define innovative concepts for the best microcontrollers (Software and Hardware) in terms of features, cost, power and time-to-market with the ultimate goal to make human life safer, cleaner and more comfortable. Provide best-in-class second level customer support and tools to enable effective and easy usage of the products.

1.3 Goal and Thesis Description

The goal of the project is to develop a tool (Jazz Cloud) that consists of a Cloud system for interchange of Jazz packets (Test packets), from upload to download with attached exporter functionality and other secondary functions. In this way we obtain a cloud system that will contain all the test workflows needed by the various Infineon Technologies teams with related information on where it was created by whom and any specific descriptions of the test packets, increasing the efficiency and effectiveness of test sharing in a single tool made with the intent to maximise the user experience to simplify the process as much as possible.
Jazz Cloud is made up by 2 main functionalities:

- **Upload**: It allows to upload packages starting from a *.tfl* (single test flow) or *.fls* (collection of tests flow) file.
- **Download**: It allows to download previously uploaded packets.

and other secondary functionalities.
After a brief introduction to Flash memories and Scientific and technological background (chapter 2 and 3), the thesis will discuss about Jazz Cloud, describing its Design. At the end of the thesis will be drawn some final considerations.

CHAPTER 2

AUTOMOTIVE MICROCONTROLLERS

A microcontroller is a small computer on a single metal-oxide-semiconductor (MOS) integrated circuit (IC) chip. A microcontroller contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals. Different Microcontrollers used in an automobile can communicate one with another through a multiplexing. These microcontrollers can manage related systems separately by using a BUS to communicate with other networks when they are required to perform a function. The combination of several linked networks includes the CAN (controller area networks). Present controller area networks permit complex interactions, that involve sensory systems, car speed, outdoor rain fall interactions, in car temperatures with performance controls for air conditioning maintenance, the audio visual multimedia systems and braking mechanisms.

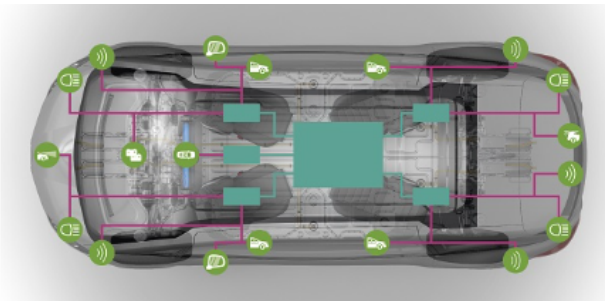

Figure 2.1: Parts of the car under the control of the Microcontrollers

The communication in the automobiles, which is established by different microcontrollers, has control over both fail safe systems and automotive fault tolerant systems wherein the microcontrollers can not only serve to respond mishaps and faults that occur to the car (anti lock brake interference, accelerator and broken lights), but also to duplicate as secondary units that continuously check the primary microcontroller in the event of microcontroller itself failing. An example of fault tolerance is, when the car tires slip on a snow filled road. The incident not only activated a response from the car driver, but the incident is also sensed by a sensor microcontroller, which will then activate the anti clock braking system when the car driver bangs on the brakes.
This chapter's aim is to give a general view of the NVM (Non Volatile Memories) in order to understand their architecture and give the reader a general knowledge. After a brief description of memories types, we will focus on Flash memories and analyze them and their structures. Then a first view on the microcontrollers developed at Infineon will be given. Finally the general enviroment of Jazz Cloud.

2.1 Memory Types

Over the last years memories have occupied a leading role in the sector of the semiconductor market founding multiple usage in the fields of electronics including automotive applications. Different types of memories can be distinguished depending on the base working technology. In particular memories can be identified on their storage capacity, power requirement, price and latency, see Figure 2.2. As a matter of fact, memories can be divided in two main types: volatile and non-volatile which differ between them on their ability to retain information with or without a power supply.

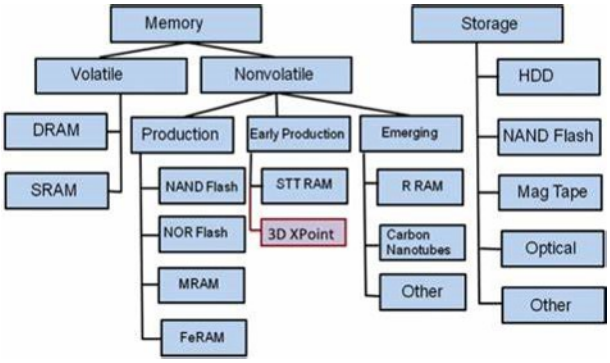

Figure 2.2: Types of digital memories

2.1.1 Volatile Memories

Volatile memories are memories which have the capability to retain information as long as they are attached to a power supply. This category includes the Random Access Memories (RAM) where the data can be written and read in any cell with an access time that is more or less the same for writing and programming. RAM memories in turn can be divided in two main types:

Dynamic RAM (DRAM): Each cell is composed by a transistor and a capacitor as a store unit, see Figure 2.3. However, in a DRAM cell, the transistor is not an ideal switch and this may cause a loss of charge in the cell. Therefore, in order to avoid to loose information, a refresh operation occurs frequently. From this characteristic came the term "Dynamic".

Static RAM (SRAM): the main characteristic of a SRAM cell is its ability to maintain information as long as a power supply is applied. In fact, in this kind of cells the refreshing operations are not needed. SRAM cells make the difference in terms of high speed and low power consumption but, on the other hand, has a much more complex structure requiring at least 4 transistors (see Fig 2.4)

2.1.2 Nonvolatile Memories

Nonvolatile memories (NVM) have the ability to retain information even without a constant power supply and can be divided on their ability of erasing data. On one hand we have NVM's which are not erasable, namely:

Read Only Memories (ROM) which are programmed in the production phase and can be only read afterwards.

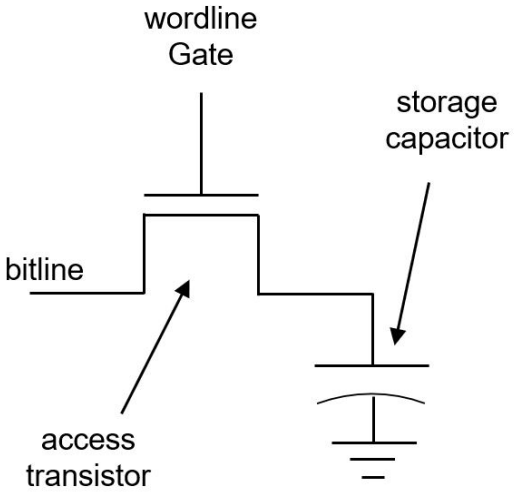

Figure 2.3: Single DRAM cell topology

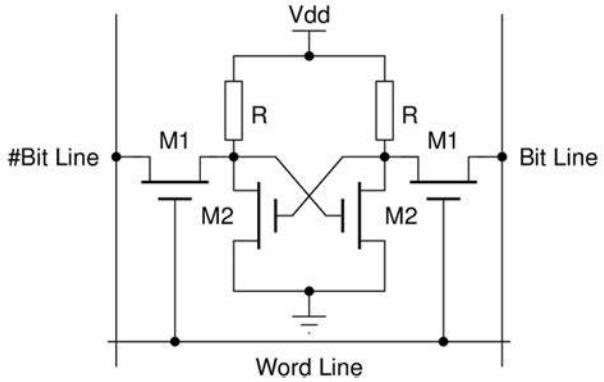

Figure 2.4: 4T SRAM cell topology

Programmable ROMs (PROM) in which the information can again be written only once but in this case it's written by the user.

On the other hand, there are the erasable NVM's:

Erasable PROMs (EPROM): need the most complex erasing procedure because they need to be exposed to UV rays for a time of 20/30 minutes. This kind of erasing procedure is really expensive and time consuming.

Electrically Erasable PROM (E^2PROM): are the evolution of the EPROMs because the erasing procedure in this case is done in an electric way allowing to keep the device in the system in which it is installed during the deletion of data process.

Flash Memories: most important NVM type which occupies the largest part in the market of nonvolatile memories. This kind of technology is nowadays the base for memories in particular for the ones embedded in the microcontrollers produced at Infineon. For this reason, flash memories will be discussed more in detail in the next section.

2.2 Flash Memory

Flash Memories are based on the floating gate transistor model. The floating gate transistor has a structure similar to the classical MOSFET with the difference that has two overlapping gates over the channel, see Fig 2.5. The upper gate is the control gate, while the lower one, completely

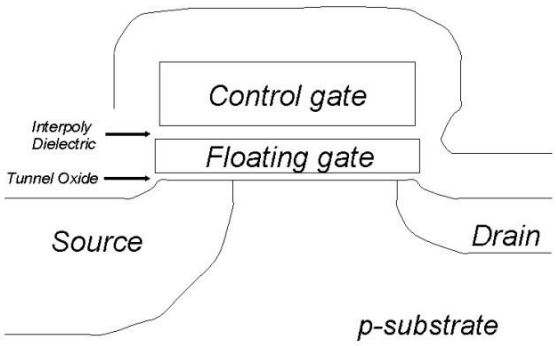

Figure 2.5: Floating Gate model structure

surrounded by dielectric, is called "floating gate". A thin oxide layer, called "gate oxide", divides the floating gate from the channel. Then, a triple layer, called "interpoly dielectric" separates again the floating gate from the overlaying layer ("control gate"). The floating gate acts as a potential well: if a charge is forced into the well it can not move from there without applying any external force. For this reason, the floating gate becomes a storing element for charge actively changing the threshold voltage of the transistor and thus inducing a change between high and low states. Furthermore, all this allows to distinguish two states in the cell which are defined as programmed and erased or as 1 and 0. Three operation can be performed in flash memories: programming, erasing and reading.

2.2.1 Charge transferring operations

Different mechanisms are used in order to transfer electric charge to and from the floating gate. The main ones are:

Hot-electron injection: in this technique a lateral electric field between source and drain heats the electrons and a transversal electric field between channel and control gate promotes the injection of the carriers through the oxide, see Fig 2.6. This is due to the fact that either an electron or a hole gains sufficient kinetic energy to overcome a potential barrier necessary to break an interface state. The term hot electron comes from the effective temperature used to model carrier density.

Fowler-Nordheim tunneling: it is a quantum mechanical phenomena that takes place when there is an high electric field through a thin oxide. When these conditions happen, the energy band diagram of the oxide region is very steep therefore there is an high probabilty for electrons to pass through the energy barrier itself, see Fig 2.7.

2.2.2 Equivalent electrical model

The electrical model of FG device is shown in Fig 2.8. The capacitances C_C, C_S, C_D, C_B are the capacitances between FG and control gate, source, drain and bulk regions, respectively.

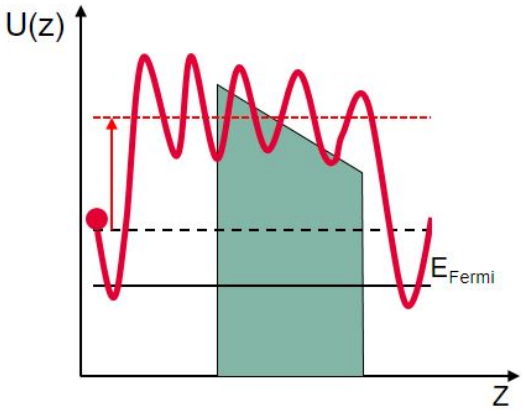

Figure 2.6: Hot Electron injection technique

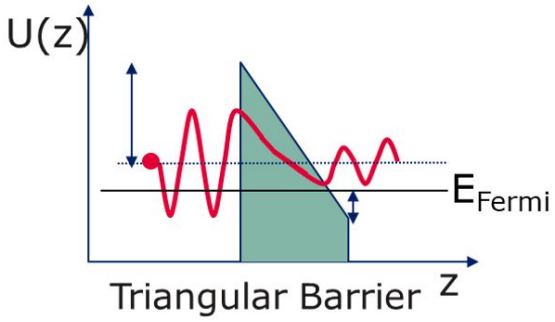

Figure 2.7: Tunneling through a thin barrier

The functionality of the device can be easily understood by determining the relationships that relate the potential of the FG (which controls the conductivity of the channel) to the control gate potential (which is controlled by the external circuitry). The floating gate potential is:

$$V_F = \frac{C_C}{C_T}V_C + \frac{C_S}{C_T}V_S + \frac{C_D}{C_T}V_D + \frac{C_B}{C_T}V_B + \frac{Q}{C_T} \tag{2.1}$$

where $C_T = C_C + C_S + C_D + C_B$ is the total capacitance.
The equation (2.1) can be rewritten, considering that bulk and source potential are grounded and all the potentials are referred to the source, as follows:

$$V_F = \frac{C_C}{C_T}V_{CS} + \frac{C_D}{C_T}V_{DS} + \frac{Q}{C_T} \tag{2.2}$$

The equation can be further rearranged by defining the *coupling factor* $\alpha_C = \frac{C_C}{C_T}$ and $f = \frac{C_D}{C_C}$ obtaining:

$$V_{FS} = \alpha_C \left(V_{CS} + fV_{DS} + \frac{Q}{C_T} \right) \tag{2.3}$$

A FG device has a characteristic that depends on the threshold voltage (potential $V_{T_{FS}}$ to apply to the FG with $V_{DS} = 0$ in order to have inversion in the surface population). Since for MOSFETs in general we have that the gate can not be accessed directly (in this case we are talking of floating gate), $V_{T_{FS}}$ is applied to the floating gate when a suitable voltage, derived from (2.3), is

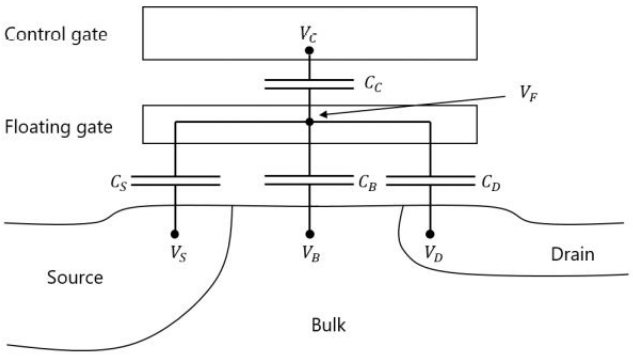

Figure 2.8: Equivalent Electric model for floating gate transistor

applied to the control gate. This voltage is:

$$V_{T_{CS}} = \frac{1}{\alpha_C} V_{T_{FS}} - \frac{Q}{C_C} \tag{2.4}$$

While the threshold voltage depends uniquely on the device technology, on the other hand,$V_{T_{CS}}$ varies within the charge into the FG. Due to this dependence it is possible to choose a suitable "threshold shift" ($|\frac{Q}{C_C}|$) allowing in this way to define two different device states: *erased* for $Q = 0$ and *programmed* for $Q \ll 0$. Each of these two states has its own threshold voltage, respectively : V_{T_E} and V_{T_P}. The I-V trans-characteristic can be seen below:

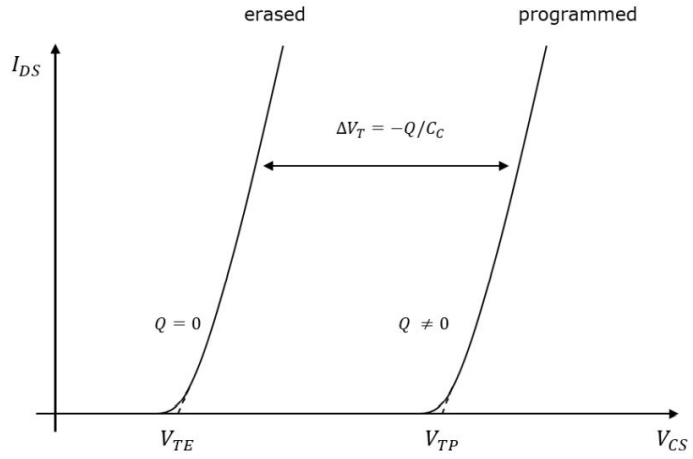

Figure 2.9: I-V Trans-Characteristic of FG device

2.2.3 Types of operations

In flash memories there are three types of operations that can be carried out:

Read Reading operations are needed in order to inspect the state of the cell. To inspect the presence of information inside a cell, polarization is needed. Polarizing the cell allows for different behaviours of it, depending on the presence or absence of charge in the Floating Gate. All can be done applying a positive voltage at the gate and drain terminals and by keeping grounded the source terminal. Under these conditions, a conductive channel

under the oxide is created by the cell inducting a current if the Floating Gate is devoid of charge. If charge is already present in the FG it does not allow the flow of current.
A sensing circuit is used to compare the current flowing through source and drain I_{cell} with a reference current I_{ref}. If $I_{cell} > I_{ref}$ the cell is programmed (logical state "1"), otherwise the cell is erased (logical state "0").

Write To perform writing operations an electric field needs to be applied to allow the charges to cross into the Floating Gate (hot electron injection technique described in Section 2.2.1). The charge injection involve a reduction of the FG potential which implies an increase in the threshold voltage.

Erase The erase operations permit to lower the threshold voltage of all the cells belonging to a sector to the logical state "0". Most used technique is the tunneling one described in Section 2.2.1 .

2.3 Flash Architectures

An architecture is the arrangement of cells of the memory in order to form a memory bank. The FG transistors which compose the memory are organized in a grid. In this way we can distinguish between Bit Lines (BL), which represent the vertical lines of the grid, and Word Lines (WL) which represent the horizontal ones. From the way in which cells are connected to the WL and BL, two main architectures can be distinguished: NOR and NAND.

2.3.1 NOR architecture

In a NOR architecture cells are arranged in rows (WL) and columns (BL). All the gates of the cells in a row are connected to the same wordline, while all the drains of the cells in a column are connected to the same bitline; the sources of all the cells in the sector are connected to a common source line, see Fig 2.10. This organization is repeated 8 or 16 times, thus obtaining either byte or word output. Reading is done by byte or by word, therefore one cell for each

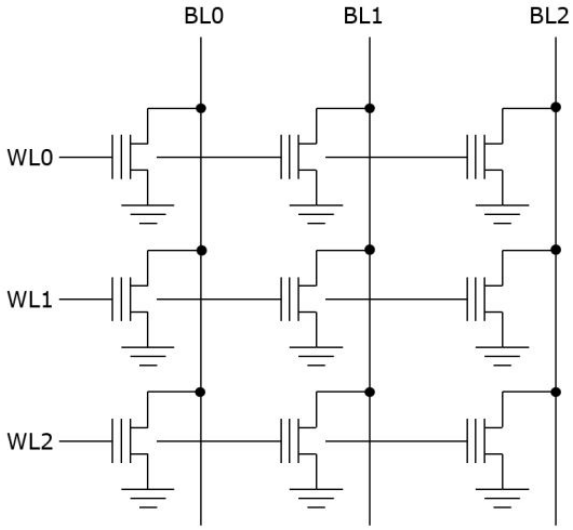

Figure 2.10: NOR architecture

output is addressed. Read operation is the method to evaluate the content of the selected cell. The most used way to identify the cell status compares the current of the matrix cell with the one of a reference cell; the result of the comparison is then converted into a voltage which is fed to the output.
The program operation is used to raise the threshold voltage of the selected cell, thus changing its state to "programmed". The physical mechanism responsible for this effect is called "channel hot electron". The gate of the selected cell is connected to an high voltage which can be either provided externally or generated internally.
The erase operation is used to lower the threshold voltage of all cells inside a sector, thus changing their state to "erased". Two erase scheme can be used: *source erase* or *negative gate erase* and the erase operations can be done using one of the following methods:

1. by applying a fixed voltage at the source terminal;

2. by forcing a constant source current by means of a current regulator.

2.3.2 NAND architecture

The NAND Flash cell is similar to the NOR one but the access to the matrix information is different. The cells are arranged inside the array in serial chains: the drain of one cell is connected to the source of the following cell, see Fig 2.11. The name comes from the way in which the

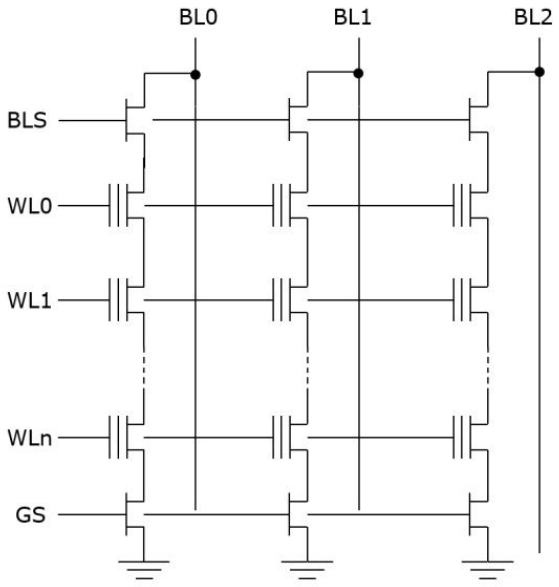

Figure 2.11: NAND architecture

read operations is performed; data sensing is serial and therefore the access time is very slow. For this reason this architecture is not much suitable for fast random access time applications. The voltages used to program and erase the cell are very high and the availability of this high voltage inside the system can be a serious constraint.

2.4 Infineon's microcontroller embedded flash memory

Infineon's microcontrollers use a flash memory with a NOR architecture. This kind of architecture is the most suitable in the field of automotive because it allows faster reading operations in an environment in which time is often critical. The microcontrollers used throughout this thesis work belong to the product of the TriCore family: Aurix TC38x and Aurix TC39x (see Fig 2.12). This family of microcontrollers is a 32-bit multicore product for the automotive market developed and produced by Infineon Technologies. The AURIX TC3xx family offers increased flash memory sizes of up to 16 MByte, over 6 MByte of integrated SRAM and up to six TriCore 1.62 embedded cores, each with a full clock frequency of 300 MHz. Due to their complexity they are often called Sistem On Chip (SoC) and the devices are tested using the SoC resources, in particular using the device's RAM onto which a dedicated Operating System will be downloaded.

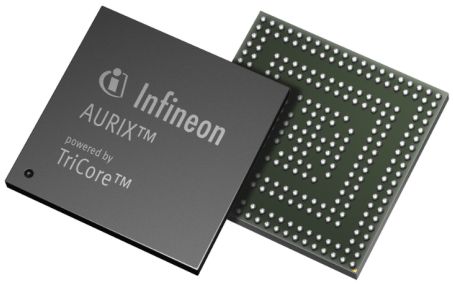

Figure 2.12: Aurix TC3xx chip

Infineon Padua is centre of competence in NVM characterization. The NVM represents more or less the 50% of the chip area, so, it is really important to test and characterize the memory in order to ensure product functionality, robustness and quality to respect the "zero defect" motto of the company. In order to characterize the NVM the most important tool, part of a toolchain totally developed in Padua, is Jazz which will be discussed more in detail in the next chapter and will be the starting point of this thesis work focused on Jazz Cloud.

CHAPTER 3

ENVIRONMENT

This chapter's aim is to give a first look at the Jazz Cloud operations and all the elements involved in the development phase. The subsections will therefore follow the order:

- Jazz description. Jazz is the tool which manages the communication between the computer and the automatic test equipment (ATE) and also between the computer and the system on a chip (SoC). Jazz acts also as an interface for running tests. It is presented as the "parent" tool of JazzCloud.
- Description of all the tools incorporated in Jazz and used to create Jazz Cloud.
- Jazz Cloud operations overview.

3.1 Jazz Tool

Every product of a microcontroller families needs to be tested. JAZZ is an Infineon proprietary analysis tool, which runs on Windows based platform, designed to provide a low cost platform to perform automated characterization for product test engineers' activities. It can be run on a normal desktop PC and interacts with lab instruments 24h/day very complex test flows. Jazz tool provides an interface to the device under test, typically with a JTAG protocol access. Moreover, it can drive a lot of control and measure instruments like power supplies, multimeters or thermal chambers with different types of access like GPIB. Object Orienting Programming concepts stand at the base of the development of this tool. This leads to reusability and portability of the tool in fact, once a test is created for a specific chip, it can be reused and applied to each present or future SoC, leaving the effort of differences management just to the inner object structure. Jazz can be synchronized and connected between different PCs to test many device in parallel on board setups or lab instrumentation. JAZZ tool can also import or export test patterns from and to other testers, following the direction of strict collaboration between Product and Test engineers. JAZZ tool is based on a graphical user interface (GUI), easing user work and training, minimizing human errors probability.
The goal of this analysis and characterization tool are:

- To be Standard, Indipendent and Integrated

- To be product indipendent which means to support all 8/16/32 bit Infineon MC families (configuration files allow to define any new device)
- Bring automation to the design validation activities
- Bring automation to the design characterization activities
- Suitable for all Product Engineers in order to provide electrical failure analysis activities
- User friendly GUI
- Support different test flow for different analysis
- To log the result of each operation
- To be independent from staff's availability

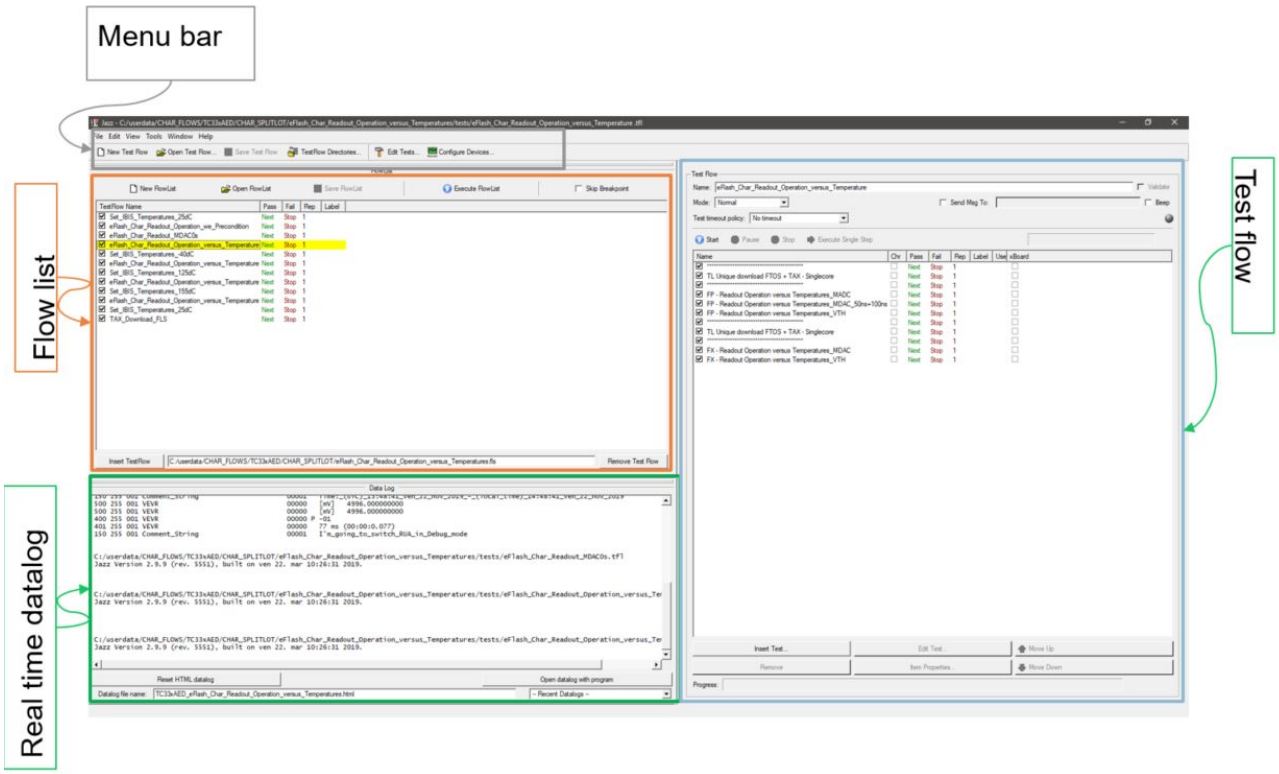

Figure 3.1: Jazz tool interface

Furthermore, by means of Jazz it is possible to interface using the personal PC with:

- Chip: device initialization, scripts and hex files download
- Flash: Read, Write and Erase
- Flash: DMA (Direct Memory Access) measures
- SRAM: Read, Write, Erase
- Power Supply: On/Off, set value, measure
- Oscilloscope: On/Off, set up, measure
- Thermostream, Thermal Chamber: On/Off, set values

- Function Generators: On/Off, force waveforms
- ATE: Automatic Test Equipment

A test is each operation performed with or without a device. Example of tests can be the chip connection, "set temperature", "get time and date", "erase flash" etc.. Every test returns a value which can be "PASS" or "FAIL" and it is identified by an unique number, a name and several properties values. For each device, in Jazz tool, it is possible to create a test collection. A test collection allows to identify a test flow. A test flow is a subset of a test collection used to perform an analysis or characterization task which has the same properties of single test (returns "PASS" or "FAIL" depending on the result of each test meaning that even if a single test fails then all the test flow fails). Also testflows are identified with a name and have an execution mode; "to run a test flow" means to run, sequentially and step by step, all the tests belonging to it. Moreover, each test can be executed several times before going through the next test and can be labelled to enable jumps or just improve readability. Each test allows to choose the action to perform on "PASS" or "FAIL". Different execution modes are available:

1. Normal: Testflow is executed only once.
2. Cycle: Testflow is executed a certain number of times until a fail occurs.
3. Test Characterization: Each single test can be characterized and each single test can be run a certain number of times but the testflow runs only once.
4. Flow Characterization: The whole testflow is characterized

Results are then logged in different ways like:

- html file
- xml file
- Line or shmoo plots, in png and csv format

JAZZ tool is a powerful solution to get cheap, flexible and common tests in both lab and production environment. Automation, reliability, stability, portability, reusability and object oriented concept are the main characteristics, allowing the tool to aim to act as a standard solution for semiconductor testing process.
The idea of this thesis work is to promote the Test Flows sharing, by enabling user to access a cloud system through a Jazz plug-in (aka Jazz Cloud).

3.2 Related Tools

Before starting with JazzCloud presentation, it is useful to describe all the tools incorporated in it and planned to be used in the next release.

3.2.1 Python

Python is an interpreted high-level general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant indentation. Its language constructs as well as its object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Figure 3.2: Python Logo

The code of Jazz Cloud is developed in Python, it is the core of the entire project.

3.2.2 Git

Git is a software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development. Its goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different systems).

Figure 3.3: Git Logo

In the 4th chapter, there is the architecture of the third release in which integration with *Jenkins* and *Artifactory* is foreseen.

3.2.3 Jenkins

Jenkins is a free and open source automation server. It helps in the automation of the parts of software development related to building, testing, and deploying, facilitating in this way continuous integration and continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat. It supports version control tools, including AccuRev, CVS,

Figure 3.4: Jenkins Logo

Subversion, Git, Mercurial, Perforce, ClearCase and RTC, and can execute Apache Ant, Apache Maven and sbt based projects as well as arbitrary shell scripts and Windows batch commands. Jenkins is organized in Job Jenkins, in which each job is referred to a particular automated process. Every time a Job is built, then a process automatically runs in a specified node (computer), that can be local or remote. Jenkins reports the node prompt messages and, if the process produces some files, the files themselves in a Jenkins cloud (artifacts) where then in our case will be deploy automatically in *artifactory*. In the Job Jenkins' settings is possible to set its organization:

- **General Configurations**: In this section it is possible to set the general configurations, like the node in which the code has to run, the parameters of the project, enforces a minimum time between builds based on the desired maximum rate, or discard old building records based on a maximum count of build.

- **Source Code Management**: It defines if Jenkins has to check out some repositories. This check is mostly used to perform the polling of the repository itself.

- **Build Triggers**: This part defines the build triggers. There are multiple options: trigger builds remotely, build periodically or polling a repository. In particular this last setting let Jenkins trigger the build every time a repository, reported in "Source Code Management", changes.

- **Build Environment**: Here the Job Jenkins environment is set up, such as the elimination of the workspace before every build, it provides the configuration files, or inject environment variables to the build process.

- **Build Steps**: This part defines all the steps the Job Jenkins has to do during its build. It is possible to Execute windows batch (or shell) command automatically, generate documentations, or trigger another Job.

- **Post-build Actions**: It defines all the post-build actions. In particular it is possible to publish the results in different formats, build another project or retire build after its failure.

Jenkins is used to trigger the entire upload and run process in a remote computer (node) in the third release of Jazz Cloud.

3.2.4 Artifactory

Artifactory is a product by *JFrog* that serves as a binary repository manager. The binary repository is a natural extension to the source code repository, in that it will store the outcome of your build process, often denoted as artifacts. Most of the times one would not use the binary repository directly but through a package manager that comes with the chosen technology. You can host your own repositories, but also use Artifactory as a proxy for public repositories. With such a proxy the time to receive an artifact is reduced and it saves bandwidth. Artifactory allows you to host your private build artifacts.

Figure 3.5: Artifactory Logo

Artifactory in the first releaase of Jazz Cloud cointain only the binary of the entire project, in the next releases it will contain also the test flow uploaded by the users (now contained in a Network Folder, see chapter 4).

3.2.5 Qt 6 and Pyside6

PySide is a Python binding of the cross-platform GUI toolkit Qt developed by The Qt Company, as part of the *Qt for Python* project. It is one of the alternatives to the standard library package Tkinter. PySide is free software. PySide supports Linux/X11, macOS, and Microsoft Windows. There have been three major versions of PySide:

- PySide supports Qt 4
- PySide2 supports Qt 5
- PySide6 supports Qt 6

PySide6 was released in December 2020. It added support for Qt 6 and removed support for all Python versions older than 3.6. Qt 6 and Pyside6 are used in Jazz Cloud project for create and manage GUI operation.

3.2.6 SQL and MySQL Workbench 8

SQL (*Structured Query Language*) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e. data incorporating relations among entities and variables.

MySQL Workbench (3.6) is a unified visual tool for database architects, developers, and DBAs. MySQL Workbench provides data modeling, SQL development, and comprehensive administration tools for server configuration, user administration, backup, and much more.

A specific database has been created for Jazz Cloud in order to maintain the information listed in the requirements exposed in chapter 4. Furthermore, stored procedures have been created relating to the entries that can be updated via the Jazz Cloud tool.

Figure 3.6: MySQL Workbench Logo

3.3 Jazz Cloud overview

Jazz cloud is a project born 6 years ago within Infineon Technologies Padua. Initially created as a web app, it was launched in 2019. However, it has never been used by users due to problems of complexity in accessing it and due to a low user friendly interface.
Once the state of art was analyzed, the decision was to maintain the *old* requirements and moreover to add some new ones, resulting in a project that included 67 high-level requirements. The next step was to prioritize those requirement by the project's stackholders and subsequently to unpack them into different low-level requirements.
From these carried out subdivisions, **3 MVPs** (Minimum Viable Product) have been obtained, which will be introduced below and which we will see in detail in the fourth chapter.

- **MVP-1**: Upload, Download and filtering function. Focus on create a cloud system for manage internal Infineon Test Flow. (Develop stable release)
- **MVP-2**: Add filtering Sharing function (develop a first release of MVP-2)
- **MVP-3**:Run and Download in Remote environment (planned a complete architecture for this step)

CHAPTER 4

INTRODUCTION TO JAZZ CLOUD

The aim of this chapter is to give an overview on the state of the art of Jazz Cloud, define the steps taken to arrive at the solution and describe the approach adopted for developing it, exposing the architectural/design concept created.

4.1 State of the Art

The first step was to analyze the previous state of art of the already existing Jazz Cloud. In the first phase, a usability test of the functions offered, through the Jazz Cloud web page (see Figure 4.1), was carried out.

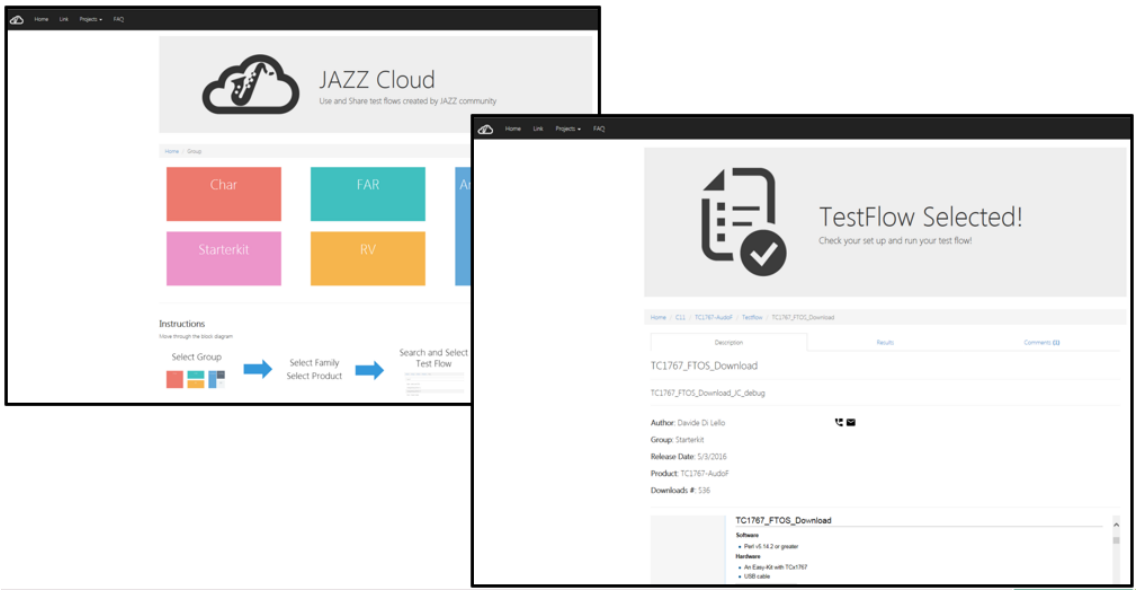

Figure 4.1: Web page Jazz Cloud

Using reverse engineering has revealed complications in terms of login time, some branching of functionalities once logged in and problems due to security leak. The use of this existing tool

was for these reasons avoided, thus at the moment it exists as a container without information, since even just loading a test flow is not so immediate. Anyway, in order to better understand the issues which relies on the base of the former Jazz cloud, a study of its algorithms was made. The source code was analyzed, deriving various *activity diagrams* that contribute to the understanding. Taking into consideration the high level *activity diagram* present in Figure 4.2, we can see how the flow is composed of 4 macro areas.

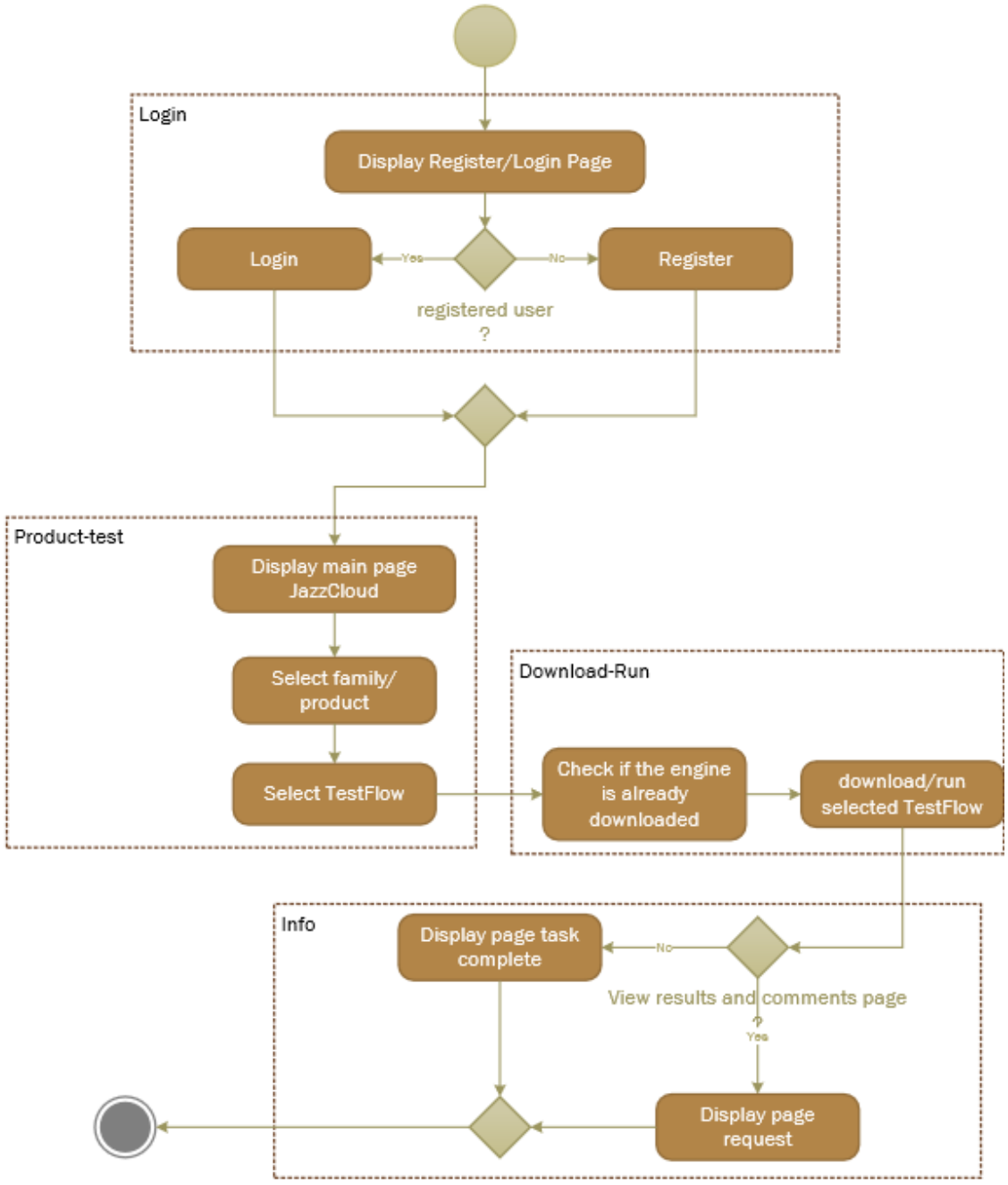

Figure 4.2: Jazz Cloud state of the art algorithm activity diagram

The four macro areas identified are respectively:

- **Login**: registration/login section
- **Product-test**: select/research previous uploaded test based on product family
- **Download-Run**: download and/or run test on device

- **Info**: display result of the test or other related information

To simplify the view of the diagram we can grasp the general idea of the previous Jazz Cloud from the image 4.3.

Figure 4.3: Jazz Cloud state of the art general idea

Each of the four macro areas will be seen in more detail below.

4.1.1 Login

Depending from the login or registration choice taken from the form by the user, the algorithms proceeds to one branch rather than another (see 4.4) and, in case of an error, a pop-up is generated which subsequently takes the user back to the previous form. On the other hand, in the case of a positive outcome of the procedure, the user is redirected to the main page (4.1 figure on the left).

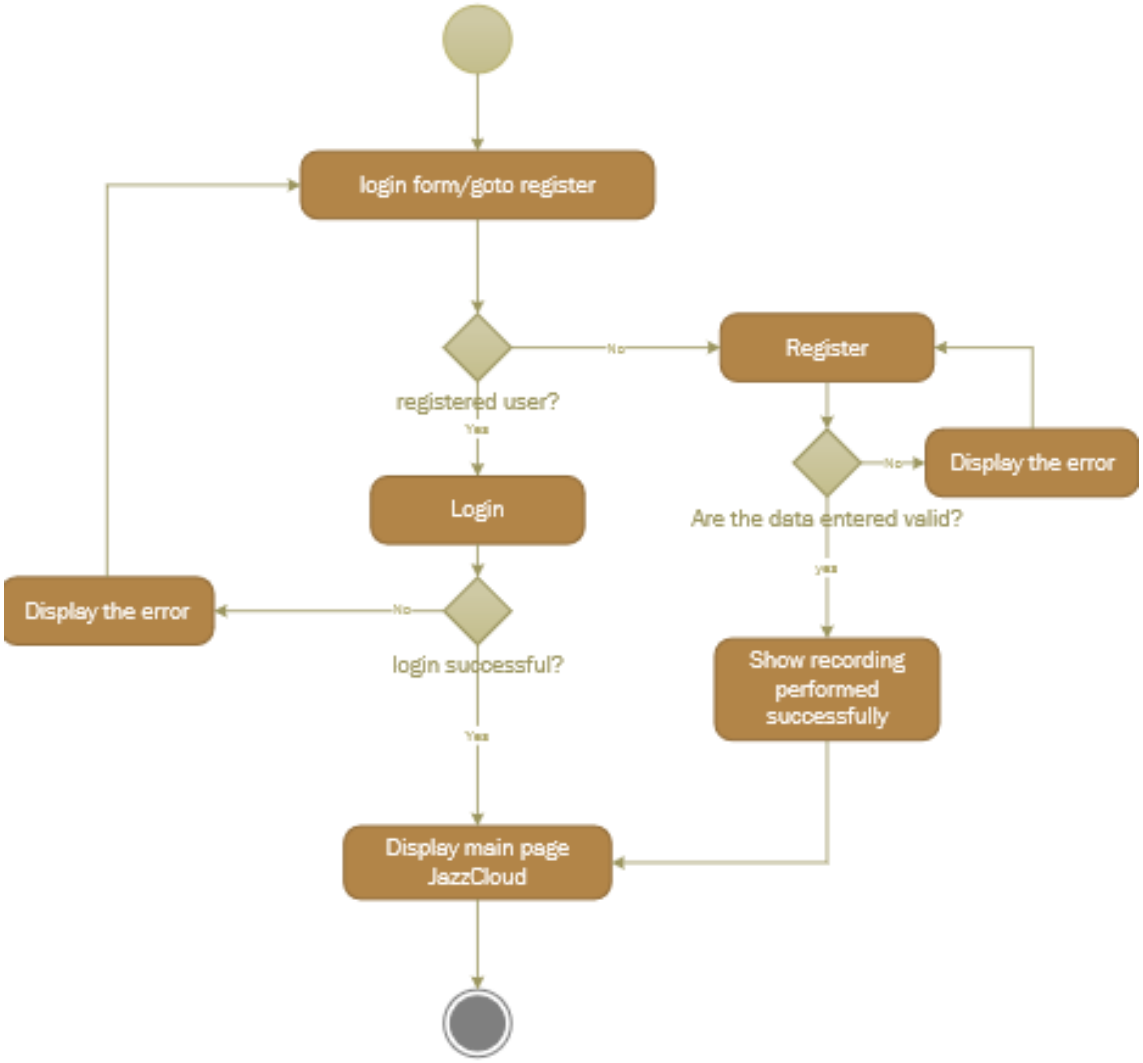

Figure 4.4: Jazz Cloud state of the art login/registration activity diagram

4.1.2 Product-test

This section deals with showing the available branch.
Once the user is logged in, based on login credentials, the product families with their respective tests are shown. Once the user has selected the test that wants to run/download the information corresponding to that test are visualized.

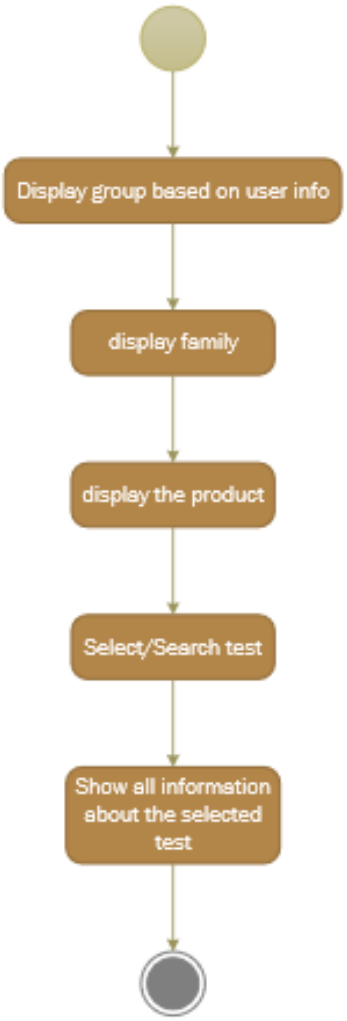

Figure 4.5: Jazz Cloud state of the art Product-test activity diagram

4.1.3 Download-Run

This section represents one of the main features of the former Jazz Cloud.
At this point, we are asked to download our *engine*, consisting of a daemon written in *perl* that runs in *background* on the PC where we will perform the operations related to the test that we will download or run. Once we have downloaded our engine, we can decide whether to download or even run the test on a specific device. The weak point of this approach is that the daemon forces us to keep a specific door constantly open, an issue that has been seen as a potential flaw security on Infineon's most sensitive sites.

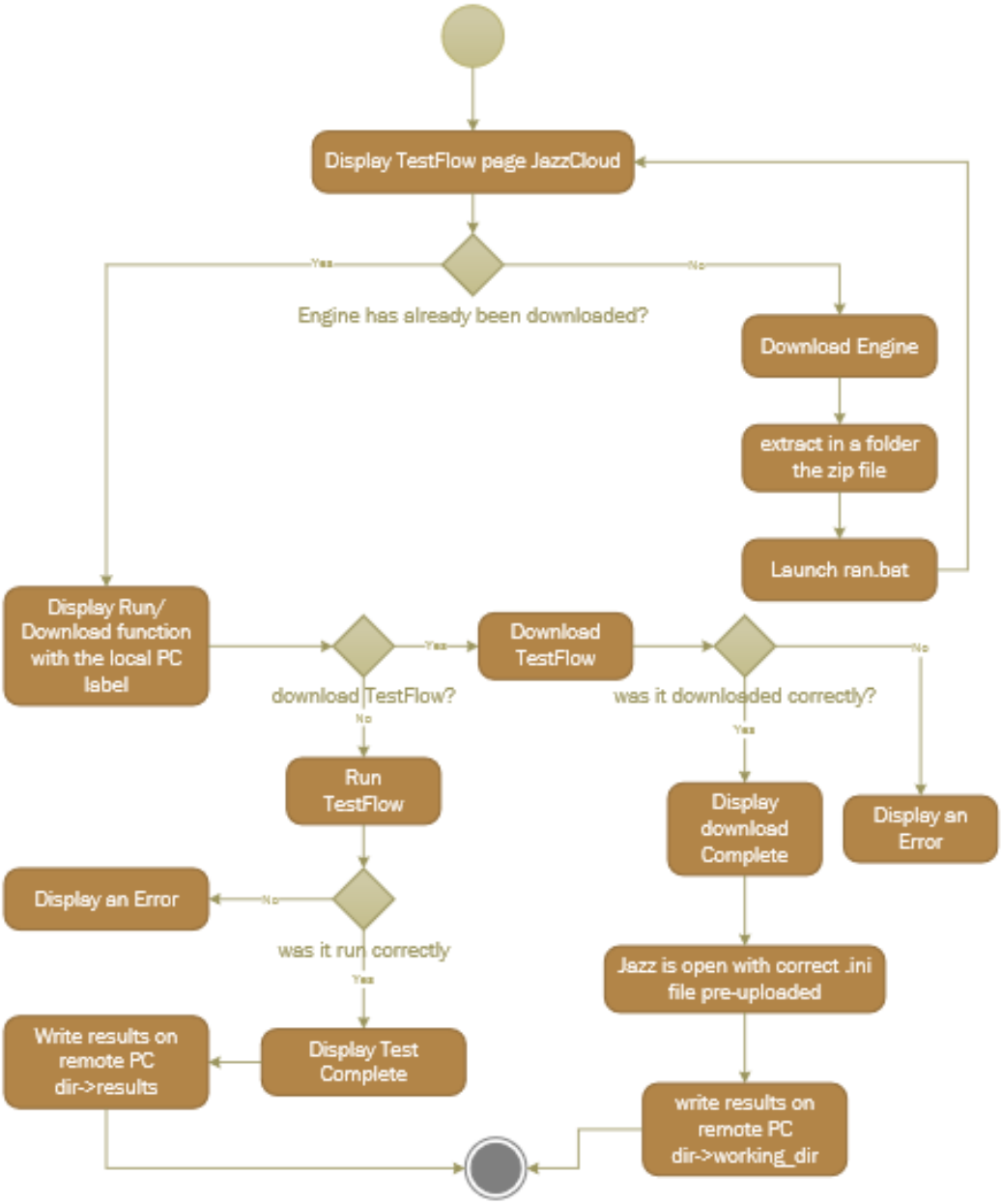

Figure 4.6: Jazz Cloud state of the art Download-Run activity diagram

4.1.4 Info

On the basis of the previously selected and downloaded / executed test flow, this section allows us to see the test result or to view the comments page related to the tests with the possibility of commenting on our own.

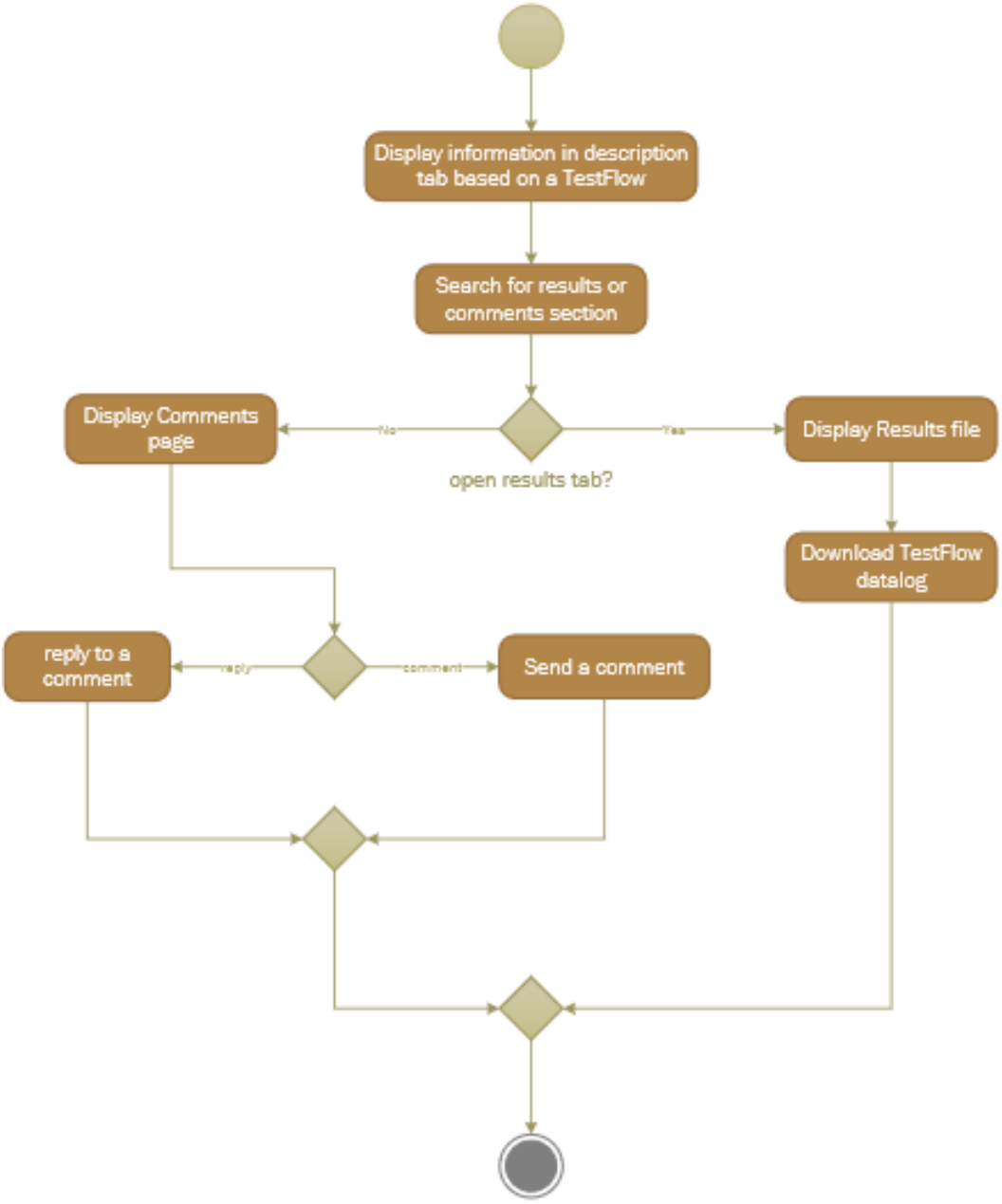

Figure 4.7: Jazz Cloud state of the art Info activity diagram

4.2 Requirements Collection

Once the state of art has been analyzed, the next step taken was the collection of the requirements. There are basically three different sources of requirements:

- **Stakeholders**: Persons or institutions who directly or indirectly impact the system.

- **Documents**: Laws, standards, manuals or other documentation that may be used to determine requirements.
- **Systems**: It is often helpful to analyze a previous system or a competing product.

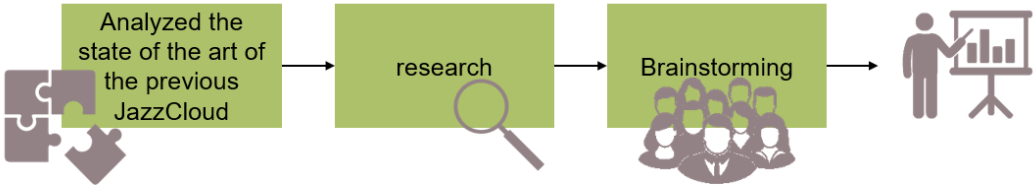

Figure 4.8: Requirements Collection methodology

So, based on this methodology, a list of all the requirements was made. The list was based on purpose to create a Cloud system for the interchange of Jazz packets, in this way improving the current Jazz Cloud tool by creating a better user experience and delete all the weak points such as ports constantly opened (potential leak of security). In the first place 67 high level requirements have been collected. After that, they were unpacked into low-level requirements and subsequently voted in order of importance. Then, they were assigned to a certain *MVP-release* (see section 3.3). In the table below the list of the requirements generated are presented following the process in figure 4.8 with their respective MVPs (Minimum Viable Product) and ratings (From 1 - Minimum to 3 - Maximum) after the screening which took place from the votes (47 REQs selected tot) assigned by stakeholders.

Req ID	Description	Ratings	MVP
1	Jazz Cloud MVP shall provides the execution of Test Flow in local env	3	1
2	Jazz Cloud MVP shall provides the execution of Test Flow in remote env	3	3
3	Jazz Cloud MVP shall provides storage of Test Flows	3	1
4	Jazz Cloud MVP shall provides share of Test Flows	3	2
5	JC MVP shall ensure a minimum set of functionalities	3	1
6	JC shan't change system user settings of PC where we have no control over	3	1
7	JC shall provide change only to remote PCs where we have control over	3	3
8	JC shall give user an automated method for uploading new flows/packages	3	1
9	JC shall have a system for store packets	3	1
10	JC shall have a system for trace packets	3	1
11	JC shall manage access to the resource, guaranteeing use one at a time	3	3
12	JC shall check that the config files are not touched during the run phase	3	3
13	JC shall bind flow and config file to a certain version of Jazz	3	1
14	JC shall have access to the various config file available	3	1
15	JC shall have access to the various versions of Jazz available	3	1
16	JC shall have access to the various Test flow available	3	1
17	JC shall provide the triad complete of Jazz, flow and config file	3	1
18	JC shall provide flow and config file only	3	3
19	JAZZ Cloud shall be able to execute eBeam Board flows	3	1
20	JC shall require to enter product and author, before to upload a jazz flow	2	1
21	JC shall provide dedicated input space as stand alone/plugin enviroment	2	1
22	JC shall provide dedicated input space in web	2	3
23	JC information shall be available at Executor level (J4)	2	1
24	JC shall use as much as possible, professional third parties software	2	1
25	JC shall provide a wiki (description page) for describing Jazz flow	2	3
26	JC shall allow only downloading the flow without execution	2	1
27	JC shall allow the User to be able to share one of his/her flows	2	2
28	JC shall allow the User to be able to share one of the flows (web)	2	3
29	JC shall handle flow versioning	2	1
30	Jazz Cloud shall provide a minimal graphic user interface (plugin)	2	1
31	JC shall provide a minimalal graphic user interface (in a web app)	2	3
32	Jazz Cloud shall run the flowlist consistency checker	2	1
33	JAZZ Cloud shall be able to trace all the relevant SW version	2	1
34	JAZZ Cloud shall be able to trace Python software version	2	1
35	JAZZ Cloud shall be able to trace Perl software version	2	1
36	JAZZ Cloud shall be able to trace Jazz software version	2	1
37	JAZZ Cloud shall provide client SW (for Jazz)	2	3
38	Jazz Cloud Client shall notify to final user the client status	2	3
39	Jazz Cloud Client shall notify to server the client status	2	3
40	JAZZ Cloud shall handle JAZZ version/testflows via its own VCS	2	1
41	JAZZ Cloud shall upload version/testflows via its own repository	2	1
42	JC shall consider BP (body power) products as pilot objects	1	1
43	JC shall track the packets maturity state	1	1
44	JC shall allow flow execution on remote setups	1	3
45	JC shall execute automatically flows without user knowledge of Jazz	1	3

Table 4.1: Jazz Cloud requirements collection pt.1

The collection and analysis of the requirements directed us towards a plug-in structure rather

Req ID	Description	Ratings	MVP
46	Jazz Cloud shall be based on a server side system	1	1
47	JC shall give the possibility to send an email to other user	1	1

Table 4.2: Jazz Cloud requirements collection pt.2

than web for the first release and laid the foundations for the creation of a software architecture focused on package storage (in our case the packets are *tests* with relative configuration file *.ini* and *jazz version*).

4.3 Jazz Cloud Architecture

Software architecture is responsible for the skeleton and the high-level infrastructure of software, whereas software design is responsible for the code level design such as: what each module is doing, the classes scope, and the functions purposes, etc. Generally, the architecture and design both explain the idea but architecture focuses on the abstract view of idea while design focuses on the implementation view of idea. Design patterns solves reoccurring problems in the software Design. These design patterns are divided mainly into three groups:

1. **Creational Patterns** : Those patterns deal with object creation mechanisms. They focus on how to instantiate an object or group of related objects.

2. **Structural Patterns**: Those patterns ease the design by identifying a simple way to realize relationships among entities.

3. **Behavioral Patterns**: They identify common communication patterns between objects and realize these patterns.

Architecture patterns provide a proven reusable solution for commonly encountered structural design problems, such as layouts and how different layers should interact with each other, within the software development industry. They, however, overly complicate the software development process if an inappropriate pattern is selected. Some architectural and design patterns have been included in the architectural design of the JazzCloud plugin. Some of these are:

- Three-layer architecture
- MVP Architecture
- Facade design pattern

4.3.1 Three-layer architecture

Decompose the design into logical groupings of software components. These logical groupings are called layers. Layers help to differentiate between the different kinds of tasks performed by the components, making it easier to create a design that supports reusability of components. Each logical layer contains a number of discrete component types grouped into sub layers, with each sub layer performing a specific type of task.
By identifying the generic types of components that exist in most solutions, you can construct

a meaningful map of an application or service, and then use this map as a blueprint for your design. Dividing an application into separate layers that have distinct roles and functionalities helps to maximize maintainability of the code, optimize the way that the application works when deployed in different ways, and provides a clear delineation between locations where certain technology or design decisions must be made.

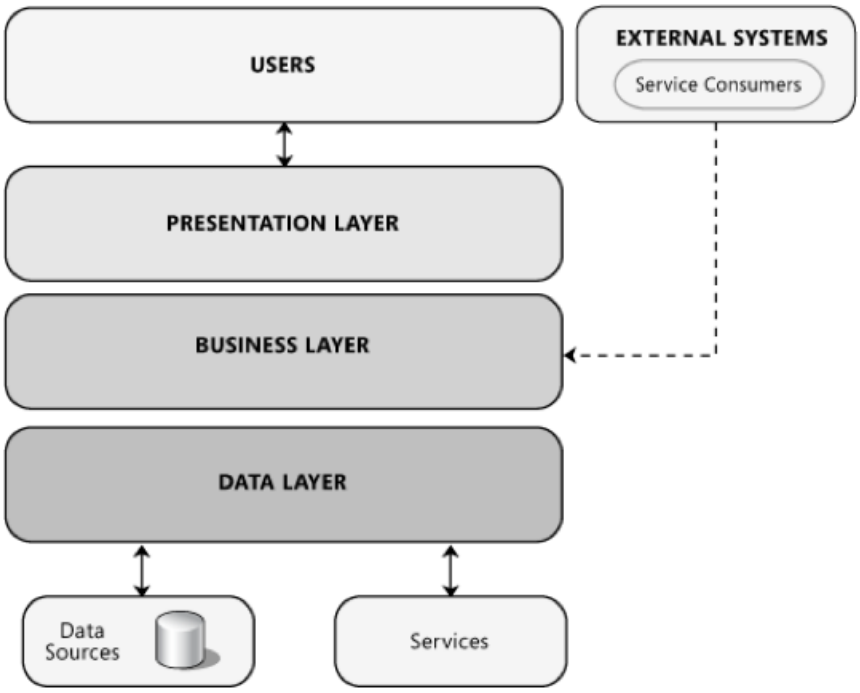

Figure 4.9: Three-layer architecture scheme

- **Presentation layer**: This layer contains the user oriented functionality responsible for managing user interaction with the system, and generally consists of components that provide a common bridge into the core business logic encapsulated in the business layer.
- **Business layer**: This layer implements the core functionality of the system, and encapsulates the relevant business logic. It generally consists of components, some of which may expose service interfaces that other callers can use.
- **Data layer**: This layer provides access to data hosted within the boundaries of the system, and data exposed by other networked systems; perhaps accessed through services. The data layer exposes generic interfaces that the components in the business layer can consume.

4.3.2 MVP (Model-View-Presenter) architecture

MVP (Model-View-Presenter) comes into the picture as an alternative to the traditional *MVC* (Model-View-Controller) architecture pattern. Using *MVC* as the software architecture, developers end up with the following difficulties:

- Most of the core business logic resides in Controller. During the lifetime of an application, this file grows bigger and it becomes difficult to maintain the code.

- Because of tightly-coupled UI and data access mechanisms, both Controller and View layer falls in the same activity or fragment. This cause problem in making changes in the features of the application.
- It becomes hard to carry out Unit testing of the different layer as most of the part which are under testing needs Android SDK components.

MVP pattern overcomes these challenges of *MVC* and provides an easy way to structure the project codes. The reason why MVP is widely accepted is that it provides modularity, testability, and a more clean and maintainable codebase. It is composed of the following three components:

- **Model**: Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.
- **View**: UI(User Interface) layer. It provides the visualization of the data and keep a track of the user's action in order to notify the Presenter.
- **Presenter**: Fetch the data from the model and applies the UI logic to decide what to display. It manages the state of the View and takes actions according to the user's input notification from the View.

Communication between View-Presenter and Presenter-Model happens via an interface(also called Contract). One Presenter class manages one View at a time i.e., there is a one-to-one relationship between Presenter and View. Model and View class doesn't have knowledge about each other's existence (see figure 4.10).

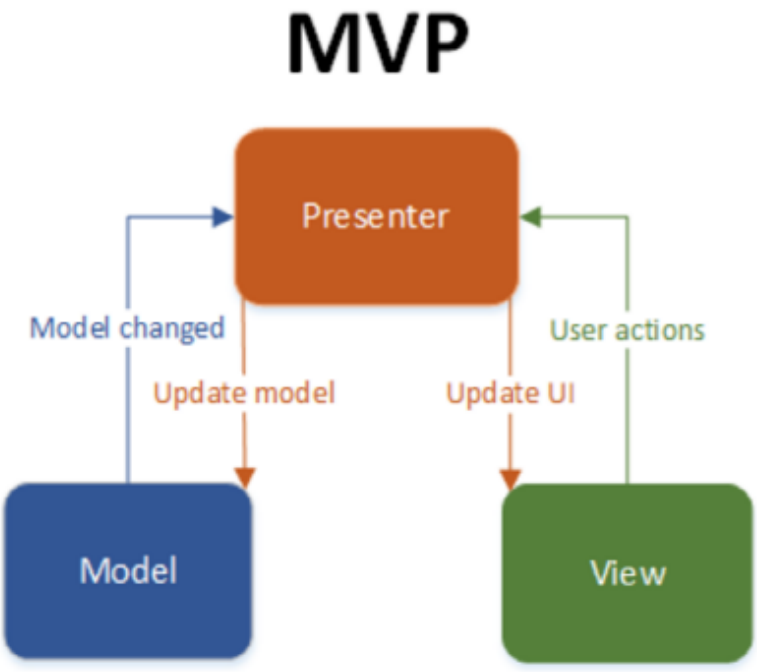

Figure 4.10: Model-View-Presenter scheme

4.3.3 Facade design pattern

This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities. This pattern involves a single class which provides

simplified methods required by client and delegates calls to methods of existing system classes. The main idea behind Software Architecture Methodologies such as Clean Architecture and Hexagonal Architecture is to create loosely coupled components that can be organized into layers. This way of writing code leverages the separation of concerns design principle and makes our application easier to maintain, i.e. we can easily modify our code and test it using stubs.
We can use the Facade Pattern to:

- Wrap Third-Party Integrations
- Third-party integrations (libraries, APIs, SDKs) are general-purpose tools designed to solve many different types of problems. Usually, we only require a small subset of the functionality a library provides.
- We can use the Facade Pattern to "wrap" our integration and only expose the functionality we require.
- If our clients requires additional functionality from a third-party integration, we can expand the interface of our Facade for that use case. Our abstraction starts to leak if clients start bypassing the Facade.

We can summarise the benefits of the Facade Pattern in three category, first: *reduces interface of 3rd party integrations*.

- Usually, we only require a small subset of functionality from third-party libraries. We can use the Facade Pattern to simplify a library's interface to only the subset we require.
- This can also improve our code's readability. Instead of directly integrating dependencies using each library's API, we can write business logic in the language of our problem domain.

second benefit is *Weak Coupling*.

- Our clients do not need to know about the underlying implementation of the integration. They only need to know the integration's interface: function names, what parameters it takes, what it sends back.
- We can change the implementation of the integration and our clients wouldn't know as long as the interface stayed the same. Another way to say this is: we "program to interfaces, not to implementations" .

Last but not least is the *separation of concerns*.

- We abstract parts of our code that change, from parts of our code that stay the same. This allows us to develop and test each component independently.

What we want to do is use what has been said about this design pattern and apply it to our third party software like *Jenkins* and *Artifactory* (see figure 4.11).

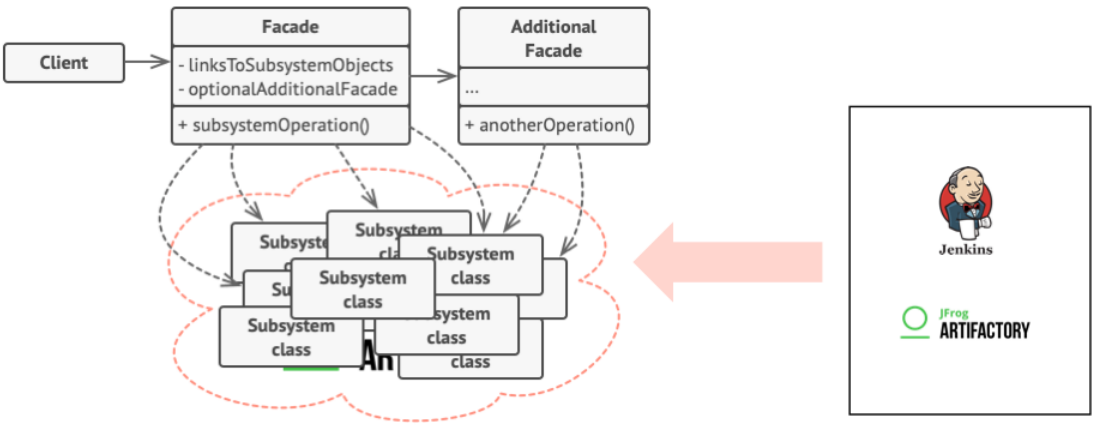

Figure 4.11: Facade Pattern in a nutshell

4.3.4 Jazz Cloud Architecture

The architecture of the cloud system that has been created is a fusion of the architectural and design patterns described in the previous paragraph. Furthermore, all the various requirements have been taken into consideration, thus constituting a scalable architecture aimed at covering all use cases up to the *third MVP*. The final architecture is shown in Figure 4.12.

The first release of Jazz Cloud takes its cue from this architecture (4.12) and without the use of Jenkins and Artifactory (using a *network folder* as a substitute). *MVP-1* manages to perform all the upload and download functions, also integrating a search function for retrieving packets uploaded by others users.
The second release will aim to share jazz packets among users.
The third release will aim to perform flows execution on remote hosts.

From the Figure 4.12 we can see the subdivision given by the three-layer architectural pattern, this architecture is finalized in the *MVP-3*, that as previous said includes function as Run and Download in Remote environment. The aim of the thesis however concerns the *first MVP* which focuses on Upload, Download and filtering function. The purpose is create a cloud system for manage internal Infineon Test Flow. In the next chapter we will see how the first release was developed and how the second MVP based on filtering/sharing function was subsequently implemented.

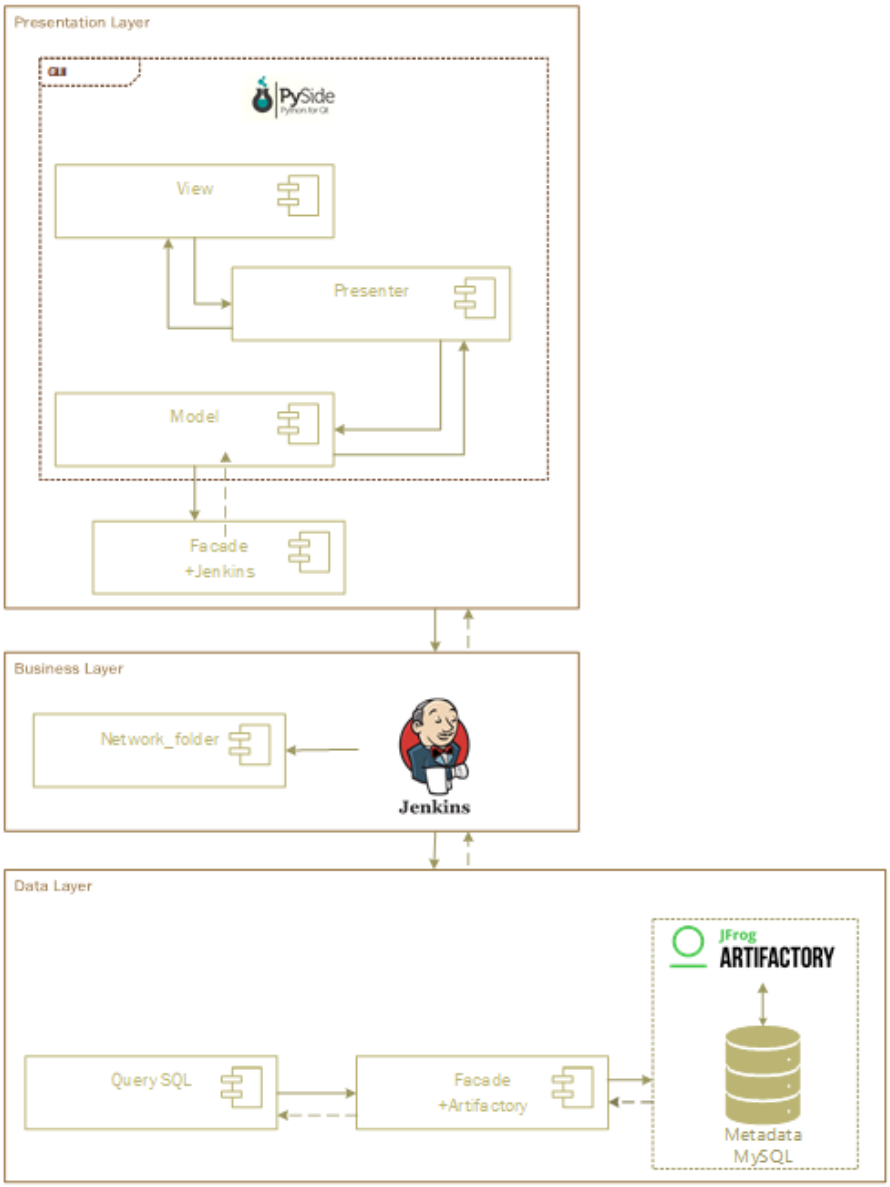

Figure 4.12: Jazz Cloud final architecture

CHAPTER 5

CONCEPT AND IMPLEMENTATION

In this section the algorithms which constitute the main parts of Jazz Cloud will be presented with the respective procedures shown through the graphical interface and subsequently analyzing the code. Then, other collateral functions that were made in order to enhance the performances of this cloud system will be shown. Finally we will see the database construct developed to maintain and track information in Jazz Cloud. From here forward all the discussion will be made taking for granted that we are developing a plugin software innested in Jazz tool needed for interchanging Jazz packets.

5.1 Development methodology

Having to develop a complete and functional software project in just 7 months, starting from the analysis of the requirements and coming to release the product within the company, we have adopted an approach called RAD that stands for *Rapid application development*. Rapid application development is an agile software development approach that focuses more on ongoing software projects and user feedback and less on following a strict plan. As such, it emphasizes rapid prototyping over costly planning. Though often mistaken for a specific model, rapid application development (RAD) is the idea that we benefit by treating our software projects like clay, rather than steel, which is how traditional development practices treat them.

The underlying rapid application development phases are:

1. **Define Requirements**: Rather than making you spend months developing specifications with users, RAD begins by defining a loose set of requirements. We say loose because among the key principles of rapid application development is the permission to change requirements at any point in the cycle.

2. **Prototype**:In this rapid application development phase, the goal is to build something that they can demonstrate to the client. This can be a prototype that satisfies all or only a portion of requirements (as in early stage prototyping).

3. **Absorb Feedback**: With a recent prototype prepared, we present our work to the client or end-users. We collect feedback on everything from interface to functionality, it is here where product requirements might come under scrutiny.

4. **Finalize Product**: During this stage, we may optimize or even re-engineer our implementation to improve stability, maintainability etc.. We may also spend this phase connecting the back-end to production data, writing thorough documentation, and doing any other maintenance tasks required before handing the product over with confidence.

Some Rapid Application Development Advantages are:

- RAD lets you break the project down into smaller, more manageable tasks.
- The task-oriented structure allows project managers to optimize their team's efficiency by assigning tasks according to members' specialties and experience.
- Clients get a working product delivered in a shorter time frame.
- Regular communication and constant feedback between team members and stakeholders increases the efficiency of the design and build process.

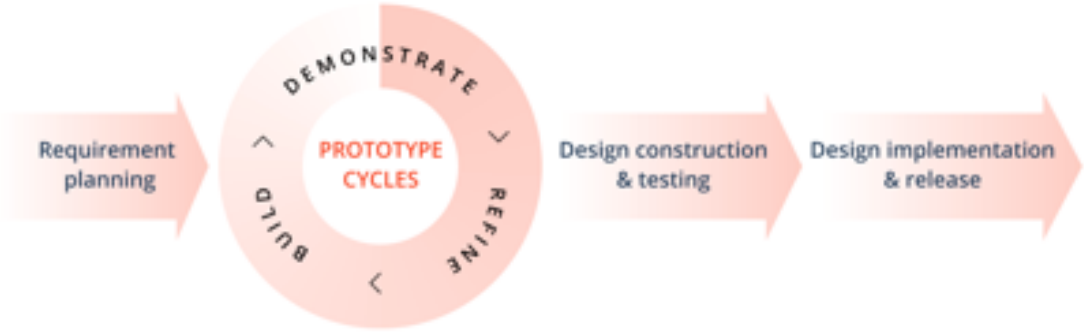

Figure 5.1: Rapid Application Development Scheme

5.2 Jazz Cloud Core

Jazz Cloud was developed using *pyside6* and *python 3.9* with the creation of a special virtual environment (*venv*). Jazz cloud has a tree structure similar to Jazz tool.

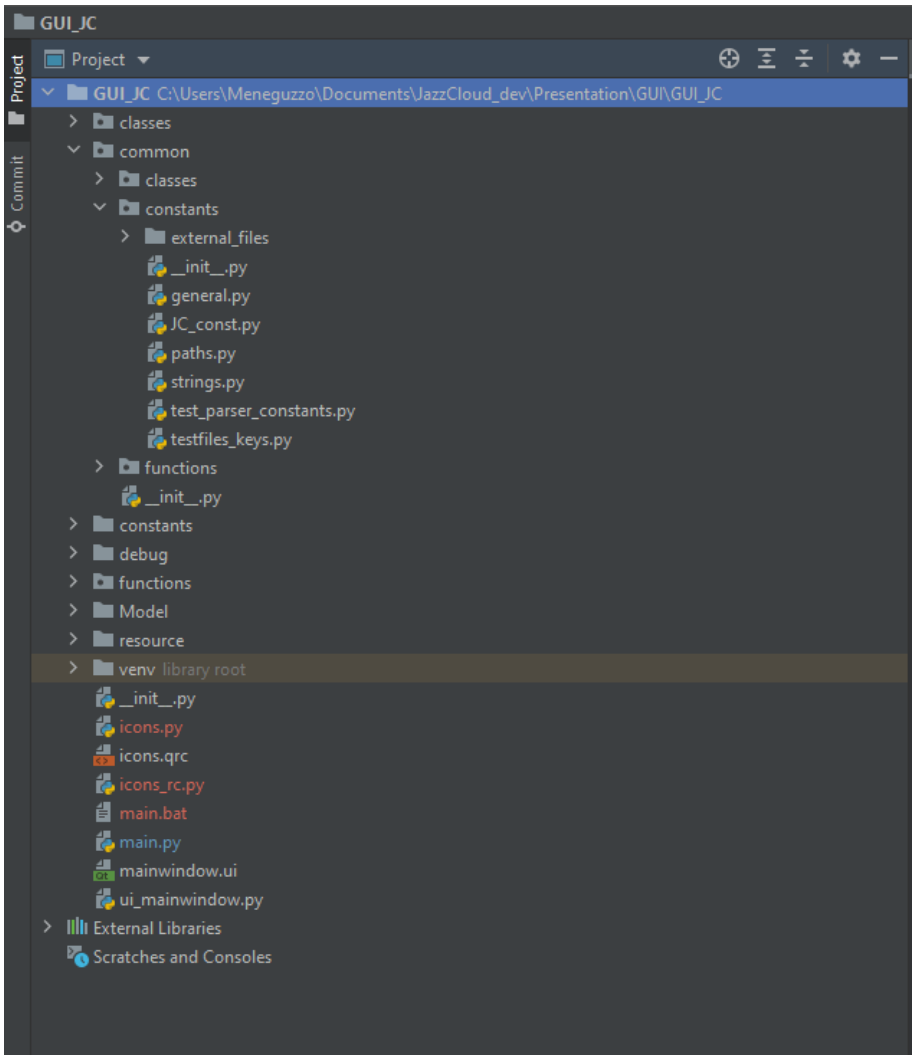

Figure 5.2: structure folder JazzCloud

From this representation we notice that the classes are grouped inside the classes folder, the functions inside the function folder, the constants inside the constants folder and this same structure is present inside the commons folder by replicating the Jazz functions / classes / constants customized for Jazz Cloud.

The Core, also called business layer, covers the part of functionality/engine. The related functionalities will be presented divided into macro sectors.

1. **Upload**

2. **Download**

3. **Side functionalities**

5.2.1 Upload function

One of the main features is the *upload function*. It allows to upload packets starting from a .tfl (single test flow, TestFLow packets) or .fls (collection of test flow, FlowLiSt packets) file.

- **.tst**: single test file, it may contain links to perl/python scripts, Intel hex/sre file or other proprietary format such as tde/pfd
- **.tfl**: test flow, set of tests (*.tst*) indexed by it
- **.fls**: flow list, set of *.tfl* indexed by it

The packets eligible to be uploaded are *.tfl* and *.fls* type with subsequent choice of the **.ini** file, which represents the hardware setup configuration (device under test and lab equipment configuration).
In the pages below, the activity diagrams of the various *MVPs* will be presented. After this, the first release, *MVP-1*, will become the subject of future arguments and we will see, step by step, the approach used, looking first at the graphic side and then at the code side.

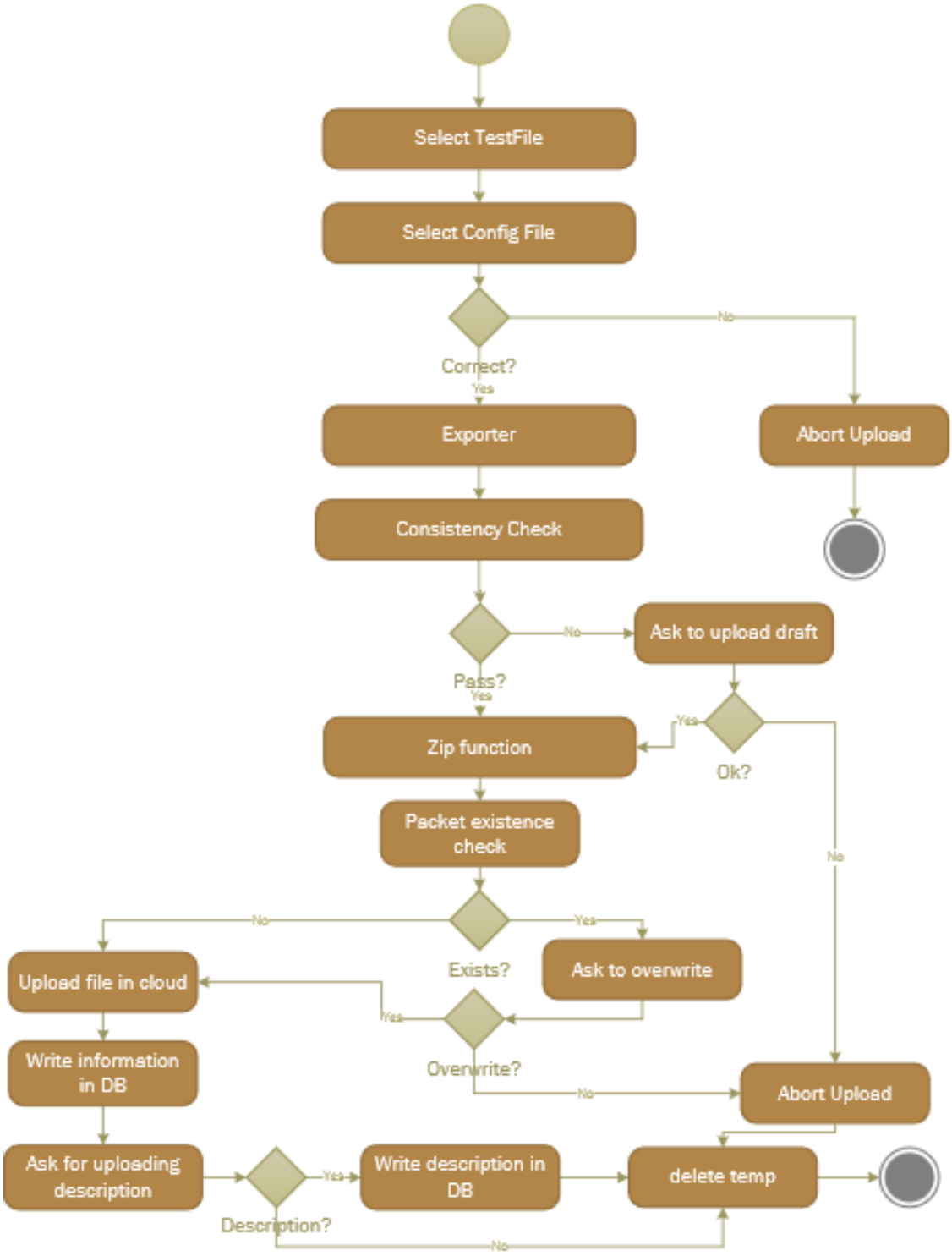

Figure 5.3: Jazz Cloud Upload activity diagram MVP-1

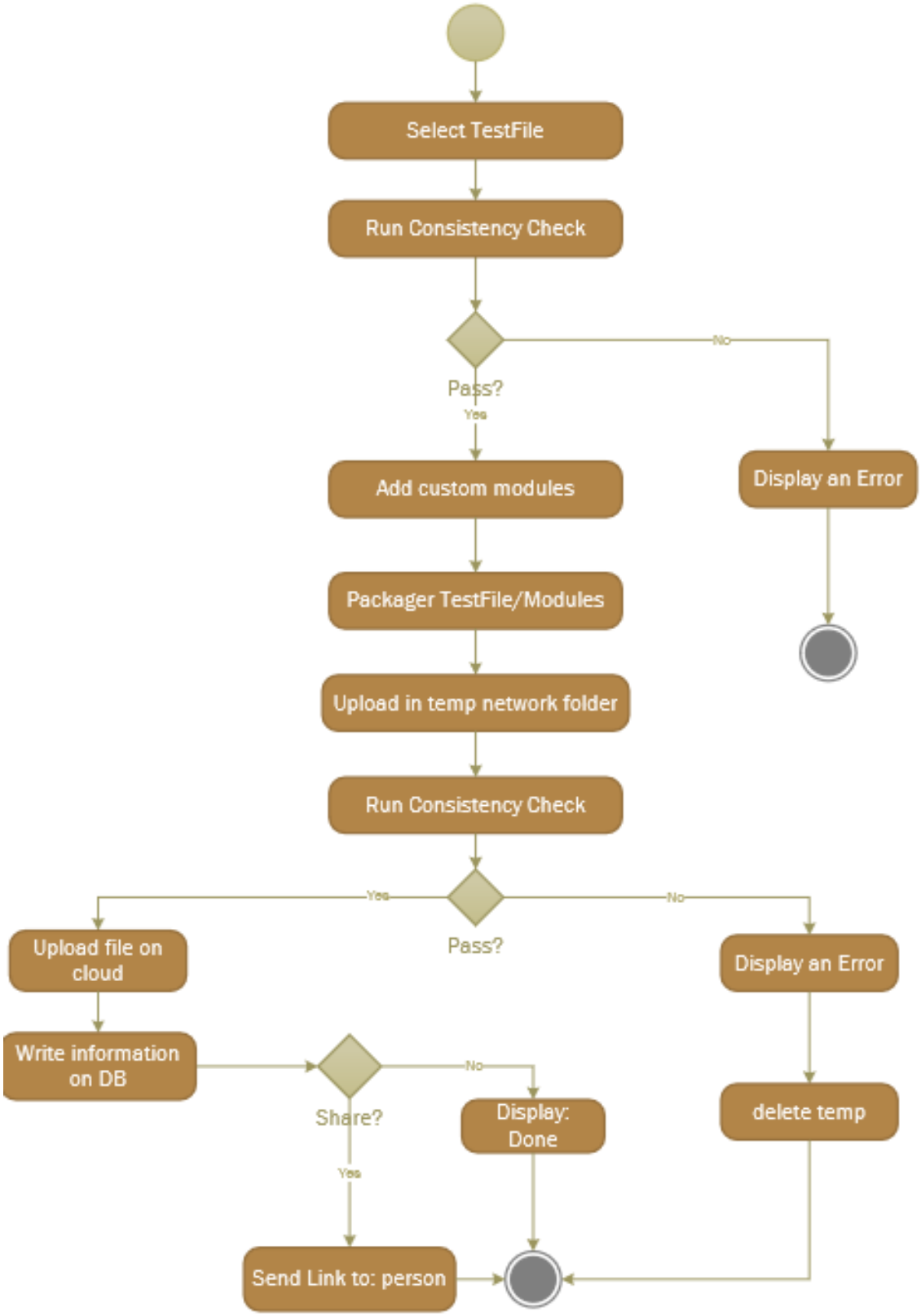

Figure 5.4: Jazz Cloud Upload activity diagram MVP-2

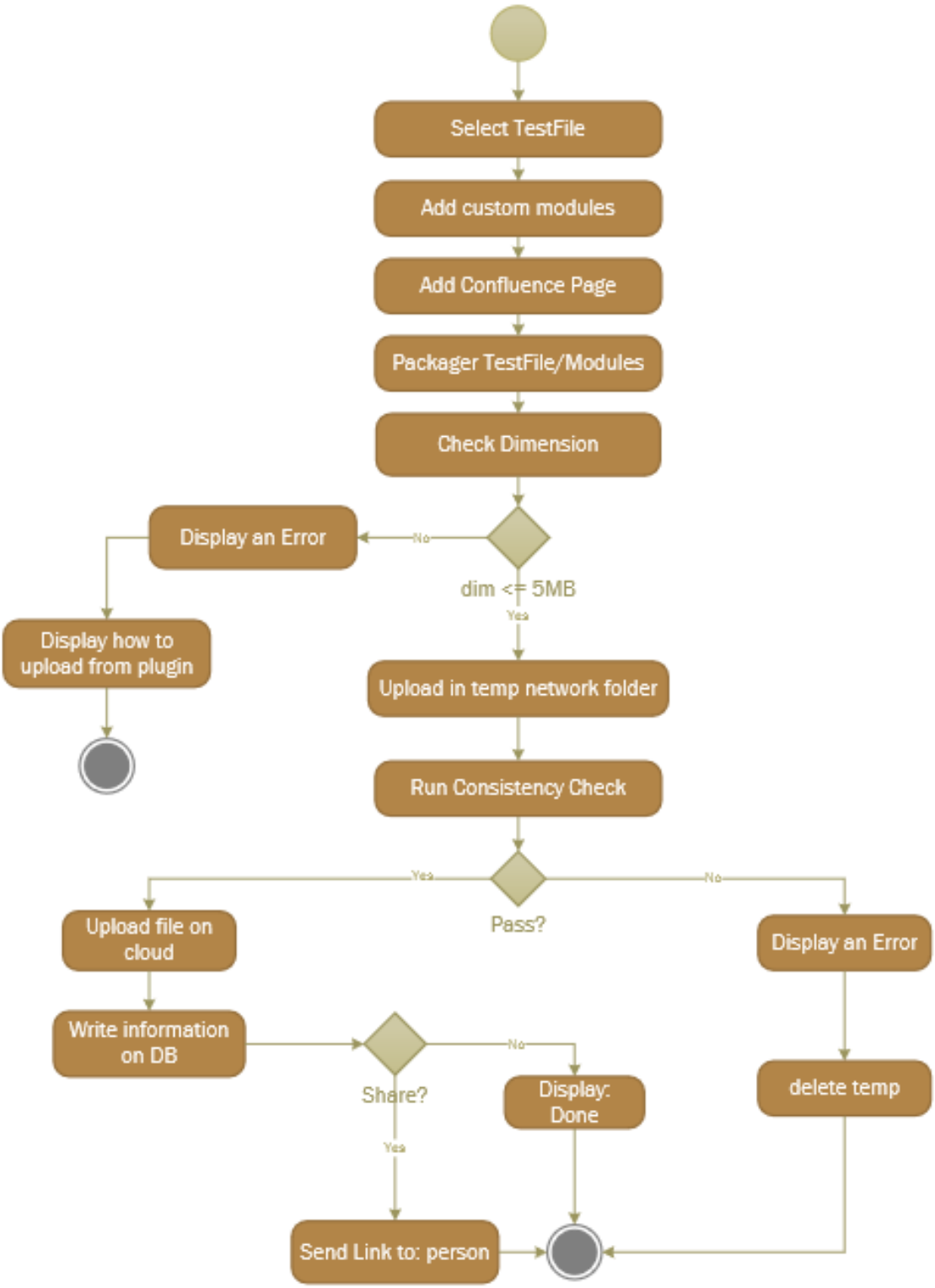

Figure 5.5: Jazz Cloud Upload activity diagram MVP-3

5.2.1.1 Upload function User interface point of view

The differences between *MVPs* are classified in the requirements listed in Table at *chapter 4.2*. As mentioned previously, following the activity diagram of the first MVP, let's look at the procedure on the graphical interface side.

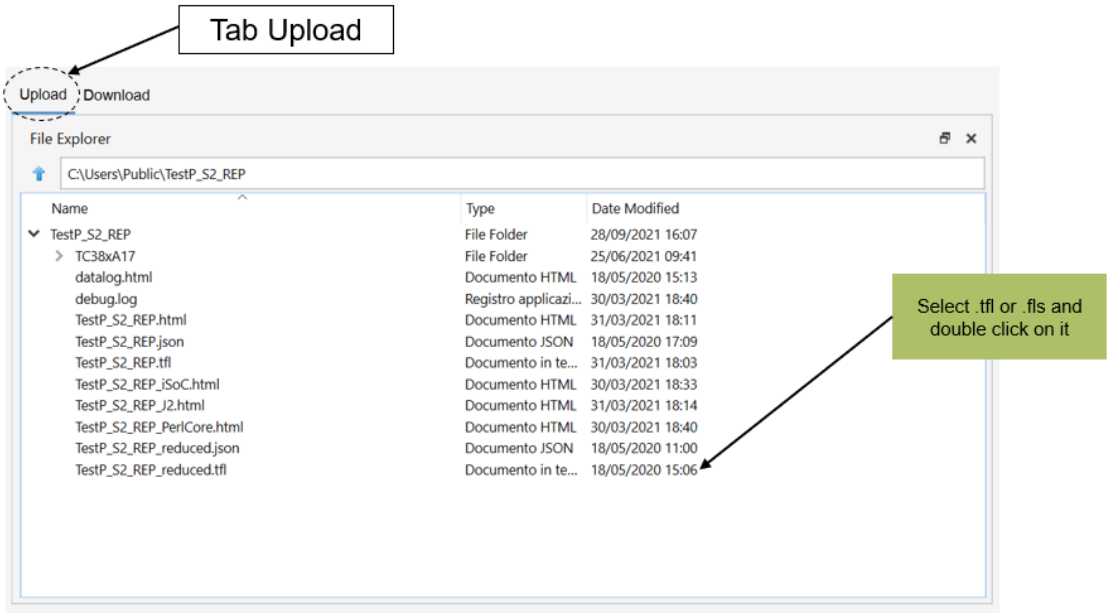

Figure 5.6: upload system window

We can access this function in the main window by clicking Upload Tab and then, on the folder tree, double click on the test file that we want to upload (5.6).
After have clicked on the .tfl/.fls file, a new Window appear asking for the insertion of the config file (5.7).

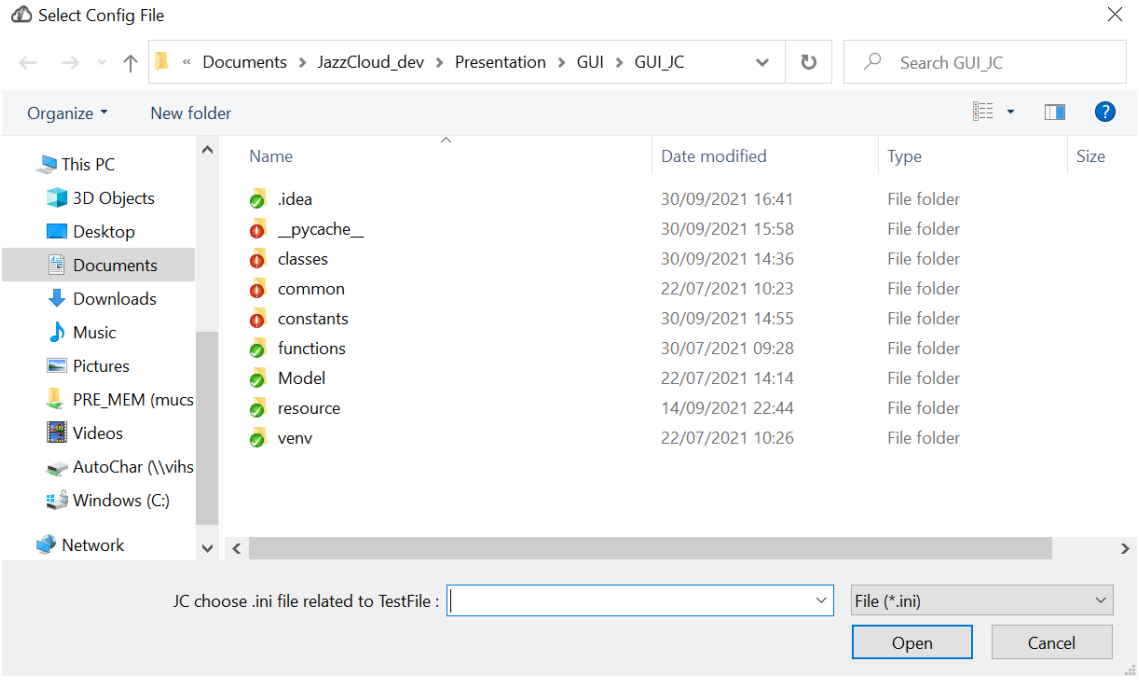

Figure 5.7: upload config file window - .ini

Now the *thread* that manages the *exporter* of the Test packets that you want to upload creates the environment for the upload, here we can find ourselves in front of a series of events:

1. The uploaded Test File is correct and all dependencies are satisfied
2. The Test File presents some dependencies not satisfied and therefore missing the files to which the .tfl / .fls is linked

If we find ourselves in the second case we will find a window that will ask us if we want to upload the Test packets anyway, even if it is incomplete (5.8).

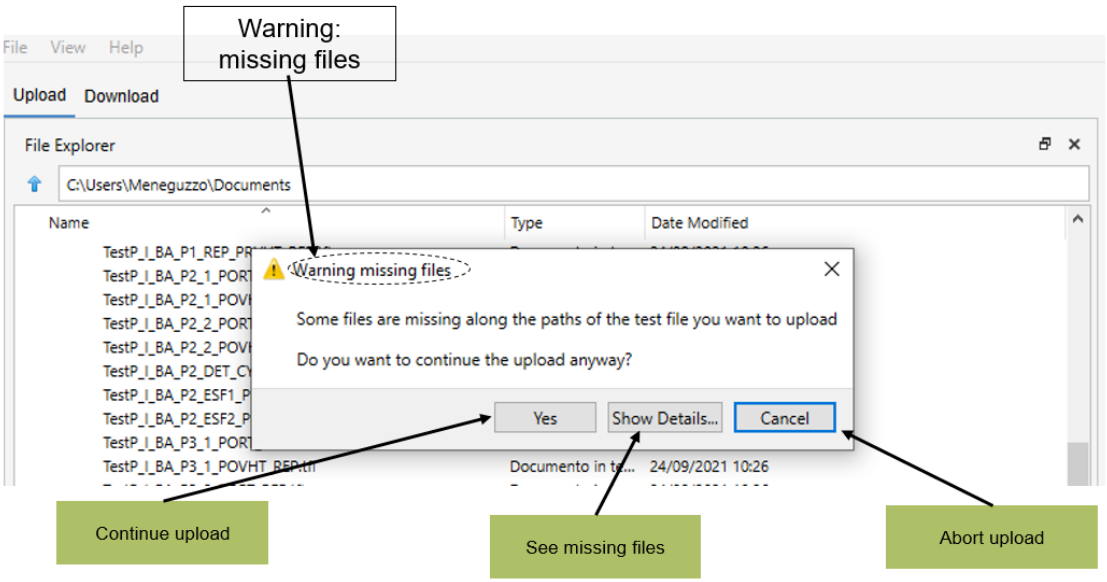

Figure 5.8: Missing File window

If we click on the *"Show Details..."* button, we could see the missing dependencies and the path where we expect to find these files (5.9).

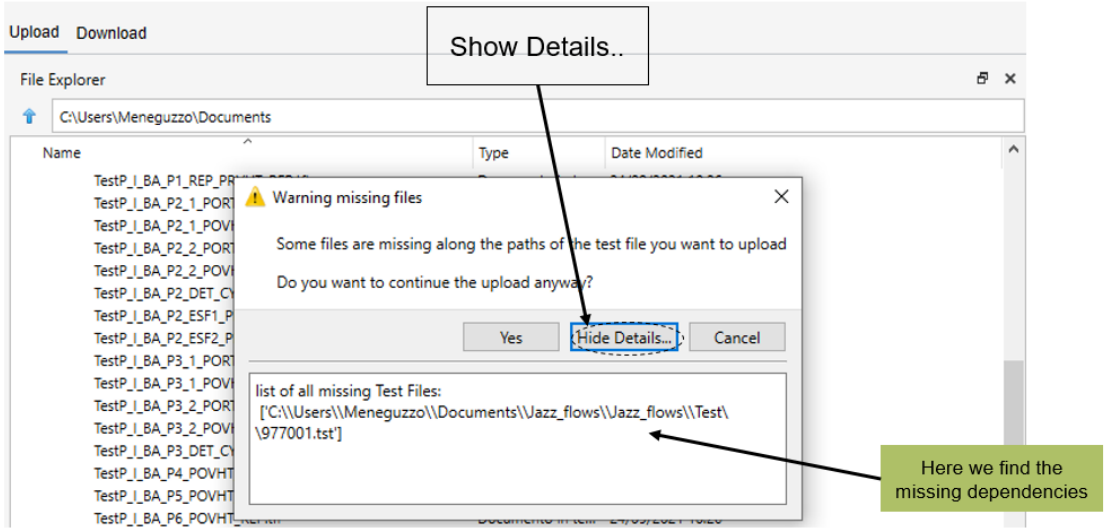

Figure 5.9: Show Details missing files

Continuing with the upload can be dangerous as you are uploading an incorrect Test Packets, it will be marked as a draft by Jazz Cloud System. In any case, being a corner case, it will appear infrequently.
Continuing with the Upload phase, now there may be two possibilities:

1. it is the first time you upload this test file so the upload will proceed independently.

2. This test file is already present in your personal cloud and a dialog window appears asking to overwrite the previously uploaded Test File.

Let's see what will appear if we were in the second case (5.10).

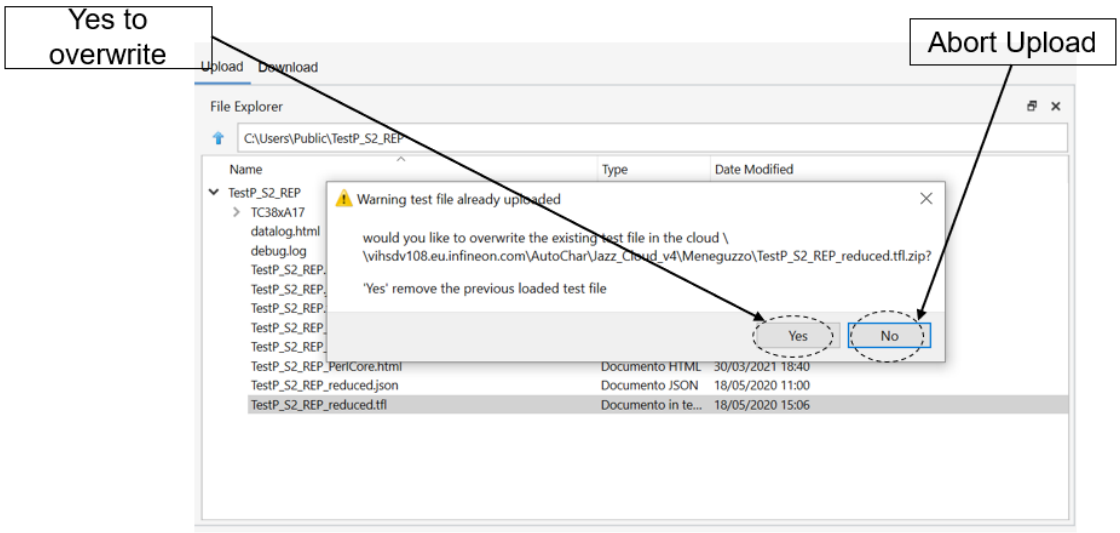

Figure 5.10: overwrite upload dialog window

Assuming we are in the first or second case with the choice of overwriting, what will happen will be the actual upload of the packet, after which you will be asked if you want to add a text description (5.11).

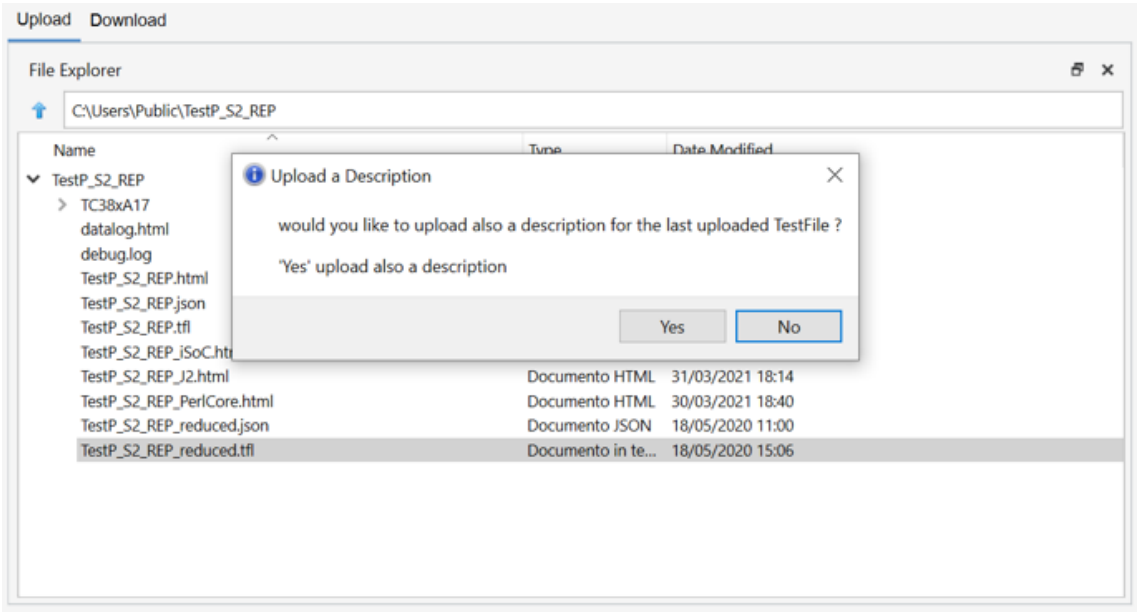

Figure 5.11: message for insert description

By clicking on Yes, a description window will appear (5.12).
After 5.12 the upload is completed, if you click on the download tab and perform a search inherent to the test file just uploaded, you will see its presence among the packets available for download, as shown in the application case in the Conclusion section, see figure 6.1.

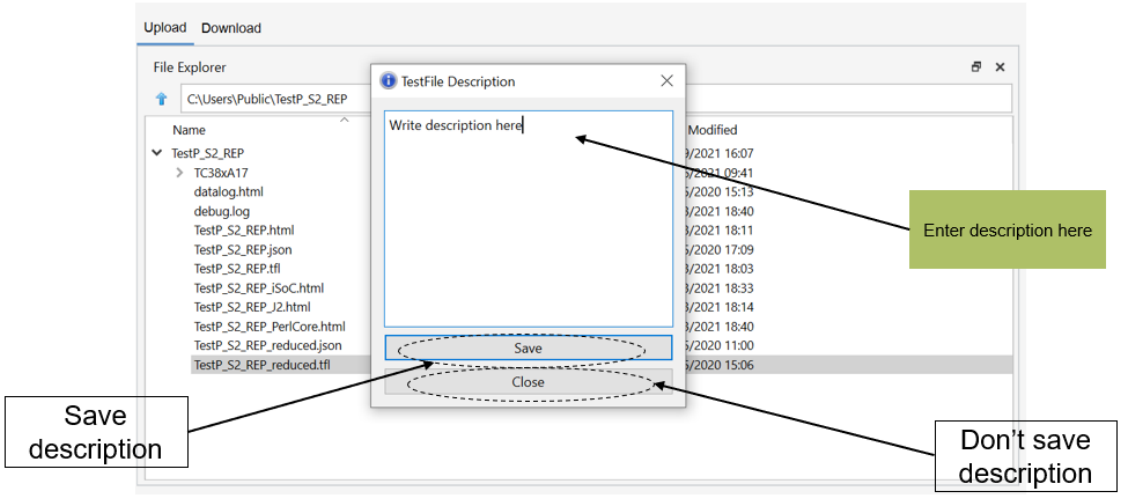

Figure 5.12: description window

5.2.1.2 Upload function code

Upload algorithm implements a multithreaded approach in such a way to allow the exporter and the compression procedure to work while the user still being able to navigate the GUI.
The upload function starts with a click / double click inside the GUI in the Folder Tree section. If the selected file is of type *.tfl* or *.fls*, the download procedure starts, which we will show in 5.13, otherwise a warning will be shown on the screen.

```
def load_testfile_from_explorer(self, path, configfiletmp):
    try:
        if path.endswith('.tfl') or path.endswith('.fls'):

            fileName = self.insert_config_file(configfiletmp)

            if fileName is not None:
                self.worker = Worker(self.zip_file, path, fileName)
                self.worker.signals.finish.connect(self.finish_worker)
                self.statusbar.showMessage(f"Wait for the current upload to finish..", 5000)

                self.threadpool.start(self.worker)

            else:
                self.statusbar.showMessage(f"Upload aborted, missing Config File", 5000)

        else:
            LOG.warning(f'Warning, try to upload different files from .tfl and .fls')
            print('Warning, try to upload different files from .tfl and .fls')
            QMessageBox.warning(self, 'Warning', 'You can only load ".tfl" or ".fls" files.')
            QMessageBox.setWindowIcon(":/resource/warning.png")

    except Exception as e:
        LOG.error("check:  Config File not selected", exc_info=True)
        print("Error: Config File not selected ")
```

Figure 5.13: load_testfile_from_explorer

Inside the **load_testfile_from_explorer** function it is confirmed that the file on which the user clicked is a *.tfl* or *.fls* and in case it has been called by the **_upload_last** function (this means that the user has clicked on *upload current*), the configfiletmp it will potentially be a valid config file or it will be None, in case it is None the user is given the opportunity to load a config file using

the **insert_config_file** function, if also in this case the config file will be None the upload will be aborted. If there is a config file, a *worker* (**QThreadPool()**) is created and will be launched.

```python
class LauncherSignals(QObject):
    gui_message = Signal(QMainWindow)
    finish = Signal()


class Worker(QRunnable):
    def __init__(self, zip_file, path, fileName):
        super().__init__()
        self.signals = LauncherSignals()
        self.zip_file = zip_file
        self.path = path
        self.fileName = fileName

        self.exporter = None
        self.path_local_zip = None
        self.basename = None
        self.missing_files = []

    @Slot()  # QtCore.Slot
    def run(self):
        '''
        Worker run function
        '''
        self.exporter = Exporter(self.path, self.fileName)
        if self.exporter.missing_tfl != [] or self.exporter.missing_tfl_tst != [] or self.exporter.missing_tst_reference_file != []:
            print("some file missing during the export: ")
            print(*self.exporter.missing_tfl, sep=", ")
            print(*self.exporter.missing_tfl_tst, sep=", ")
            print(*self.exporter.missing_tst_reference_file, sep=", ")
            self.missing_files.extend(self.exporter.missing_tfl)
            self.missing_files.extend(self.exporter.missing_tfl_tst)
            self.missing_files.extend(self.exporter.missing_tst_reference_file)

        self.path_local_zip = self.zip_file(self.exporter.root_tmp_folder)

        print(self.path_local_zip, self.path, self.fileName[0])
        print("worker exit and return to main thread")
        try:
            self.signals.finish.emit()
        except RuntimeError as err:
            LOG.error(f"signal error {err}", exc_info=True)
            print(err)
```

Figure 5.14: Worker(QRunnable)

QThreadPool manages and recyles individual *QThread objects* to help reduce thread creation costs in programs that use threads. Each Qt application has one global QThreadPool object, which can be accessed by calling *globalInstance()*.

To use one of the QThreadPool threads, subclass QRunnable and implement the **run()** virtual function. Then create an object of that class and pass it to **start()**.

Inside our **Worker** we can see a class of **Signals** used to communicate, a function **zip_file** belonging to the *MainWindow* that will be used to zip the TestFile with its dependencies and an instantiation call of the **Exporter** class, which will check the testfile, check the presence of all dependencies and create an adequate structure to contain everything, modifying the paths where needed. All of this is done within the Worker to prevent *GUI freezing* and *user dissatisfaction*.

Once the worker has finished its tasks (Exporter and zip_file) a signal of finish of work is emitted, which is captured by the main thread that calls **finish_worker**, this checks if there have been any *missing dependencies* and in case it warns the user if you are sure you want to upload an *incomplete TestFile*. Once confirmed, the **upload_zip** function is called, which checks the destination path if it exists or not for the user who is using (constant **JC_const.DESTINATION** present in the folder common / costants file JC_const.py).

Consequently creates the final folder in the dedicated **ZTEMP** section, showing in the statusbar the result of the zip transfer operations within the appropriate space of the *destination network folder*, indicated by **os.path.join(JC_const.DESTINATION, user_name)**.

```
database = db.Database(self)
if database.connection:
    name = basename
    # TODO check box custom modules
    # Check box for include custom modules if True write on DB Custom modules path refer to Upload Packet
    # Now Default value = False
    c = False
    # c = self.checkBox.isChecked()  # check_box selected = true else false
    database.write_db(name, path, config_file, c, stat)

    insert_d = self.insert_description()
    if insert_d:
        # Description create for packet
        self.d.descr = self.get_description()

        print(self.d.descr)
        if self.d.descr is not None:
            database.write_db_description(self.d.descr)
    self.statusbar.showMessage(f"test file uploaded in {check}", 10000)
```

Figure 5.15: Last part of upload_zip

Once the physical movement of the package within the network folder shared with everyone has been completed, the information is added to the database by calling the **Database class**, which takes care of inserting all the data into the Database **J4_CLOUD** (db information present in the constants folder *DB_config.ini* file).

The **Database** class inside has the writing functions for each *entity* that is reported in the Database, these are filled through rules given by the **Stored Procedures** exposed by the DB **J4_CLOUD**, these are called by the individual functions exposed within the Database class.

5.2.2 Download function

The download function is developed on a special page accessible via a tab, and allows us to download all the packets previously uploaded, viewing the complete list or a restricted set obtain through a filtering given from the search bar.
We will present the activity diagrams of the various *MVPs*, after which we will focus on the first and see the various steps on the graphics side and then on the code side. In 5.16 we can see the activity diagram of the first MPV that is equivalent also to the second release. Moreover in figure 5.17 we can see the activity diagrams refer to *MVP-3*, download function from top-down prospective goes down deeper in the schemes below (5.18, 5.19, 5.20).

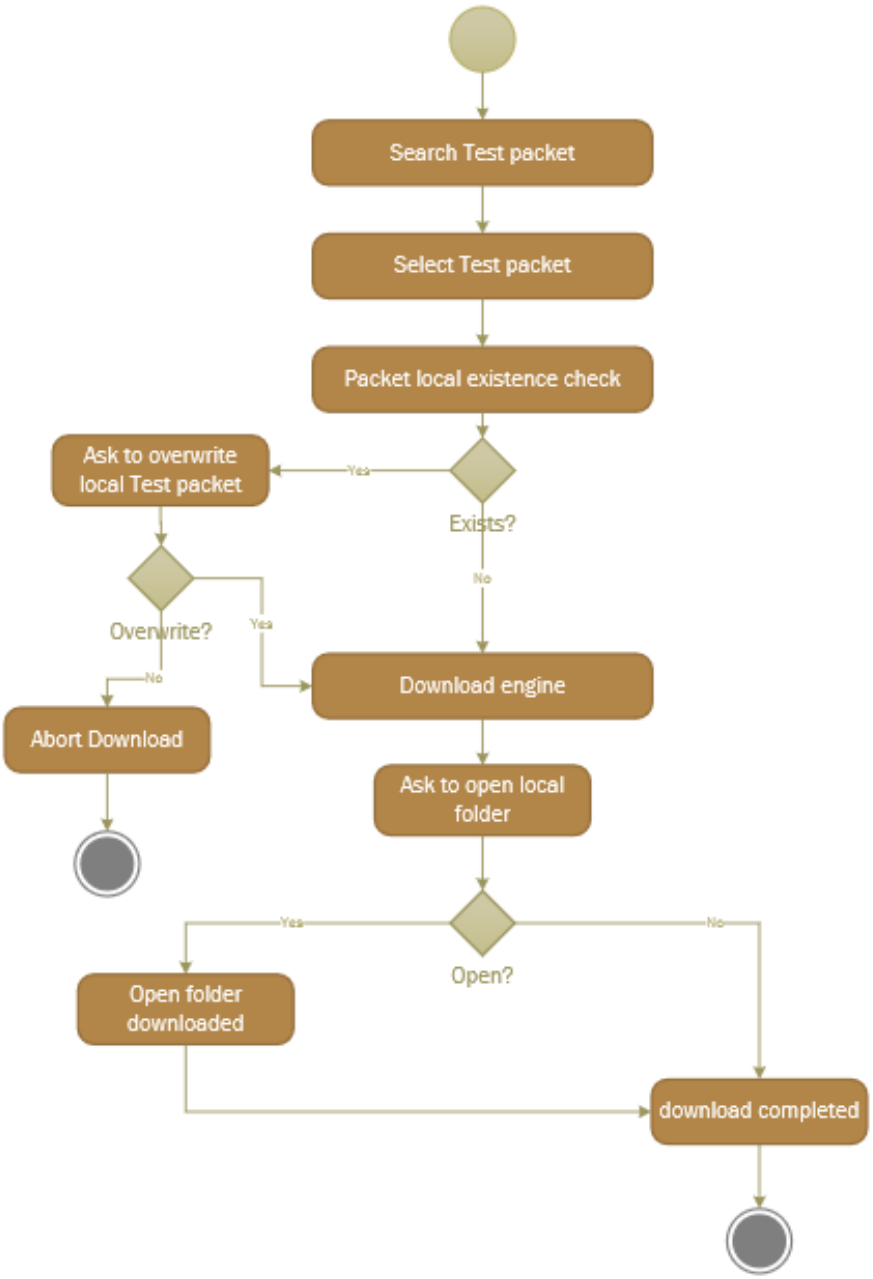

Figure 5.16: Jazz Cloud Download activity diagram MVP-1 and MVP-2

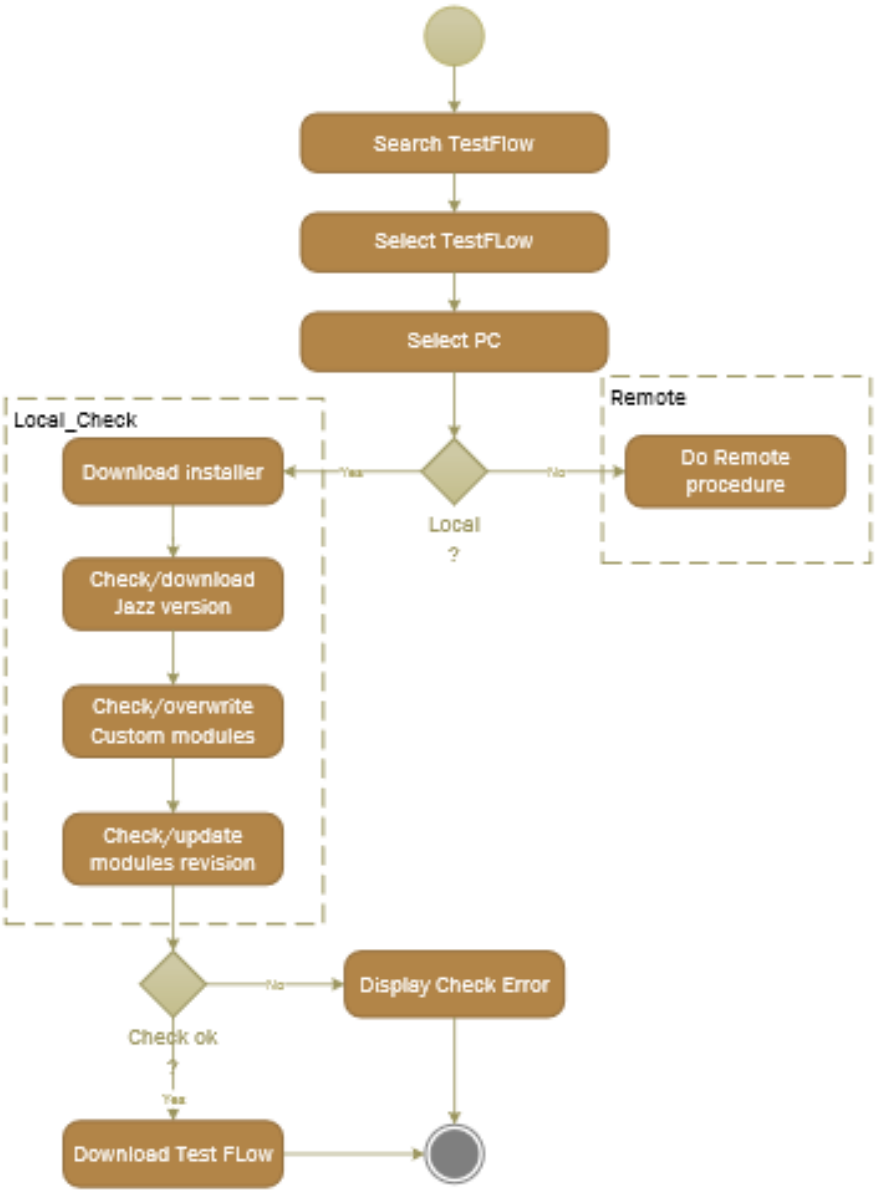

Figure 5.17: Jazz Cloud Download activity diagram MVP-3 Top-Down

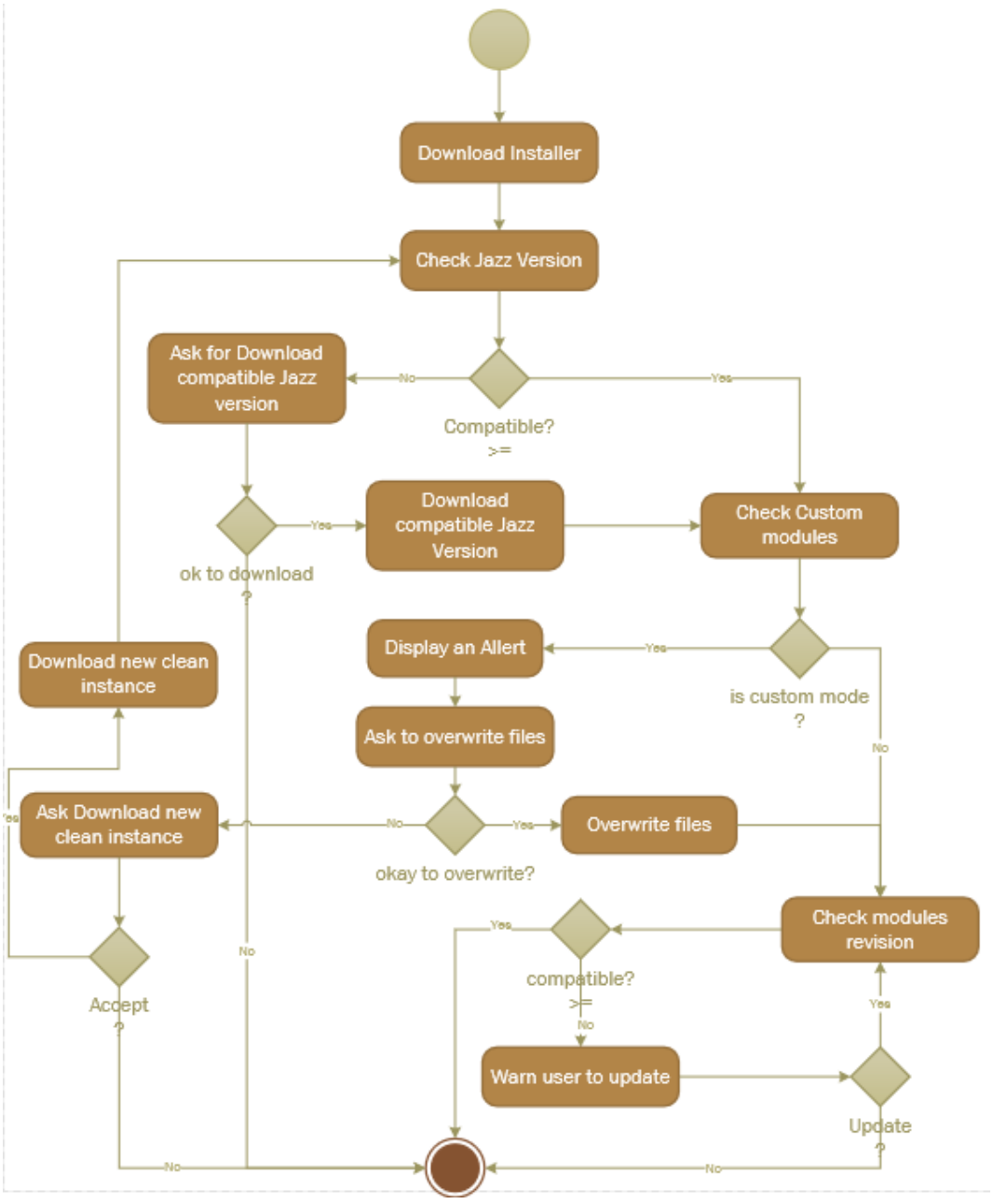

Figure 5.18: Jazz Cloud Download activity diagram MVP-3, download in local PC

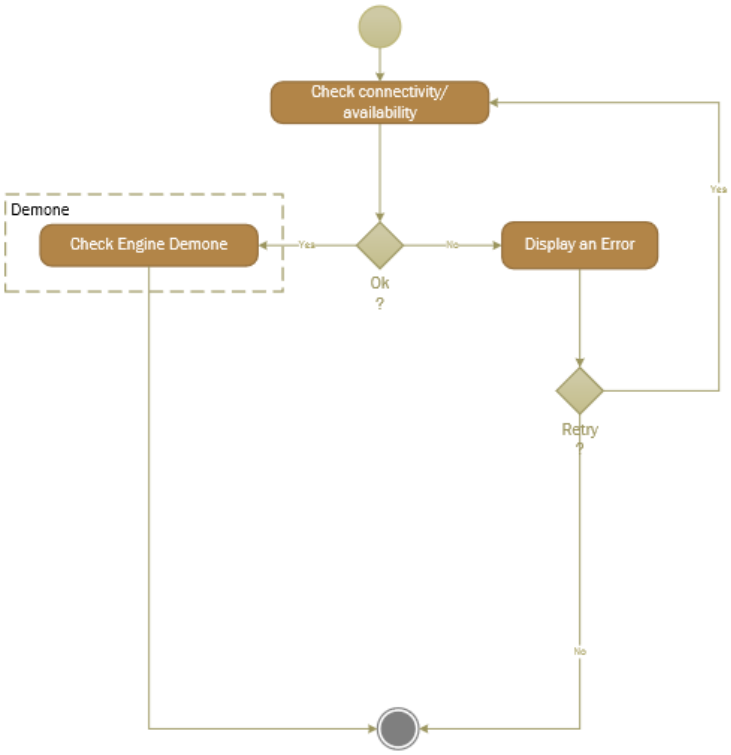

Figure 5.19: Jazz Cloud Download activity diagram MVP-3, download in remote PC

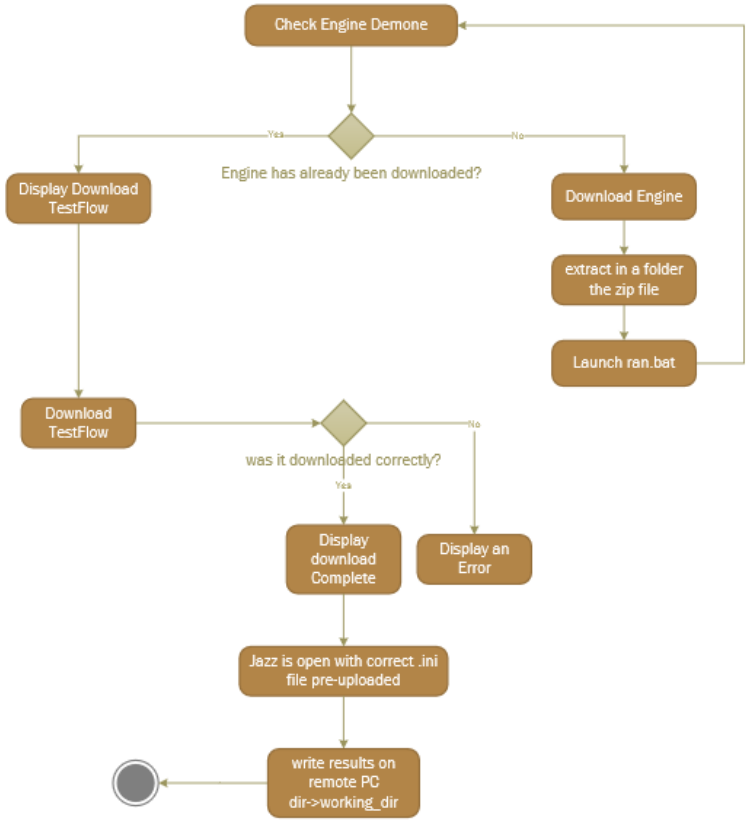

Figure 5.20: Jazz Cloud Download activity diagram MVP-3, download in remote PC, Demone procedure

5.2.2.1 Download function User interface point of view

The first release is the main focus of this thesis work, so for this reason download part reference to algorithm figure 5.16. The download function allows you to download previously uploaded packets. Clicking on the Download tab the below window will show you all the nodes, each node represents a packet, a search bar is also available for a quick and targeted search.

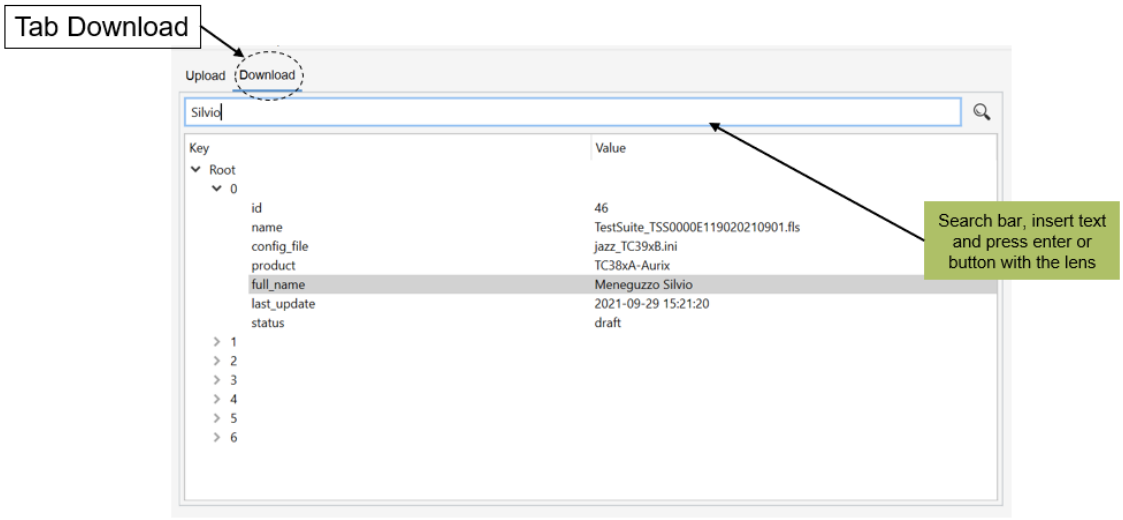

Figure 5.21: download system window

Once we have identified the packet that we want to download, just click on it in any position. A window will appear asking for confirmation of the download and in case a description has been previously uploaded, also the possibility to see it before a possible download.

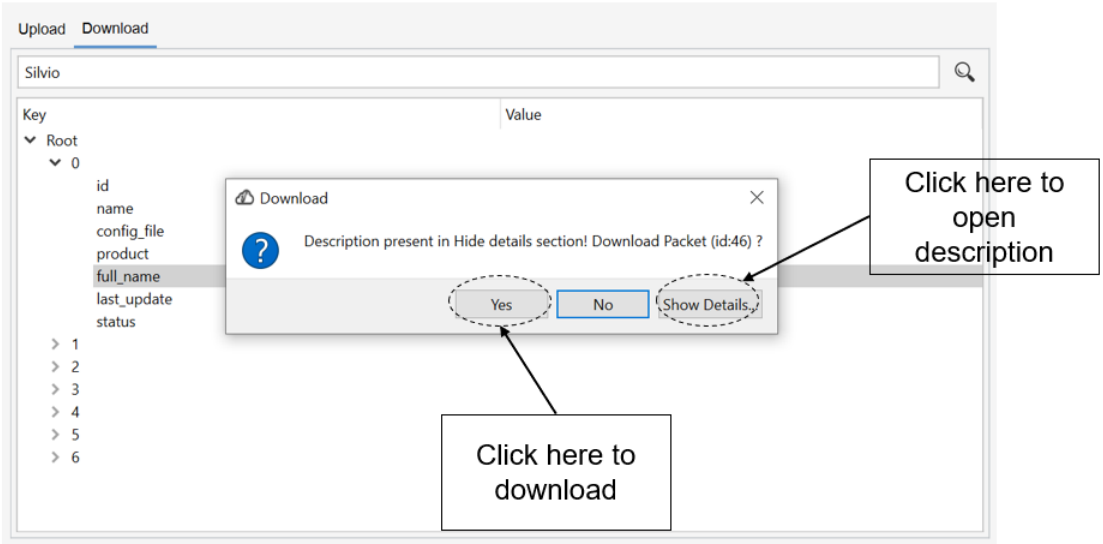

Figure 5.22: download window, with show description option

By click on Yes (5.22), the download will start, in this case two things can happen:

1. it is the first time that we download this packet so the download proceeds autonomously.
2. we had downloaded this packet previously and a dialog window appear to ask if we want to overwrite this packet locally.

In case two a pop-up appear (5.23).

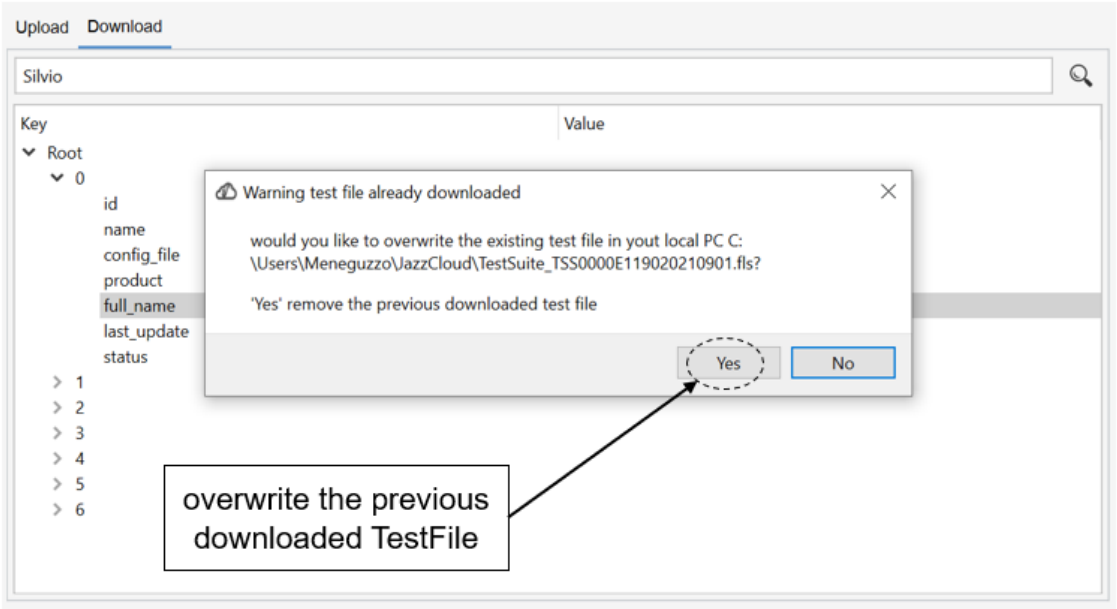

Figure 5.23: overwrite download dialog window

Assuming we are in the first or second case with choice of overwriting, We will be asked if we want to open the reference folder for all the downloads made by jazzcloud (5.24).

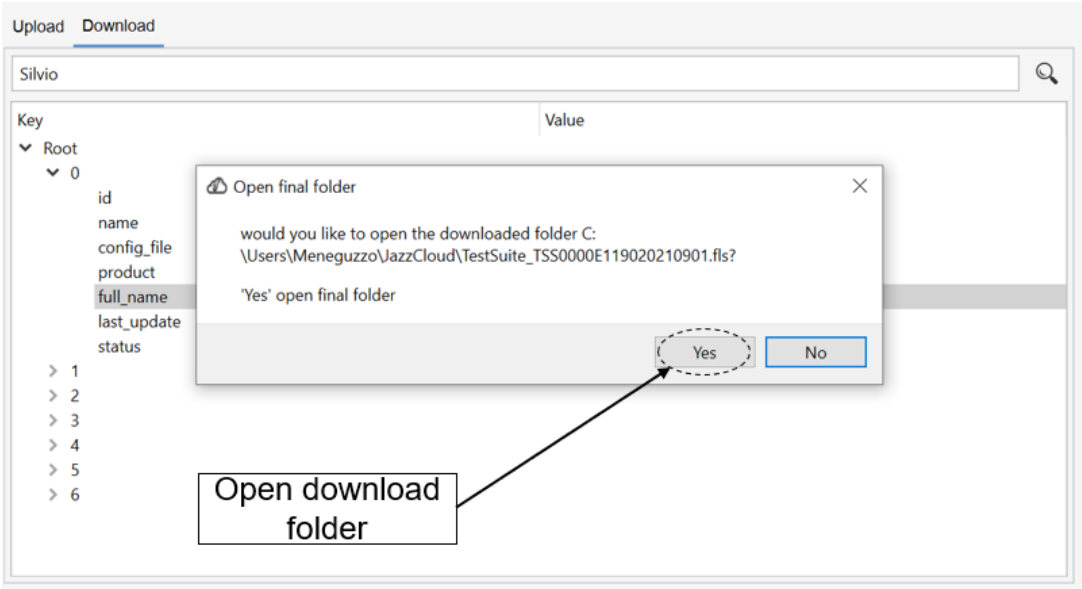

Figure 5.24: Open destination folder

In the destination folder we found all the previous downloaded *TestFile* or *Test packets* and the path is structured in this way: *C:/Users/Meneguzzo/JazzCloud*, where Meneguzzo is replaced with the Username of the JazzCloud User.

5.2.2.2 Download function code

The core of *download* consists of the **JsonView** class, this is instantiated immediately, when the JazzCloud tool is opened with the import of packets from the DB then saved in a dictionary.

```
def __init__(self, tab_d, window_description):
    try:
        super(JsonView, self).__init__()
        self.window_description = window_description
        self.find_box = None
        self.tab_d = tab_d
        self.tree_widget = tab_d.treeDownload
        self.text_to_titem = TextToTreeItem()
        self.find_str = ""
        self.found_titem_list = []
        self.found_idx = 0
        self.context_menu = None
        self.tree_widget.itemClicked.connect(self._click)
        self.tree_widget.itemDoubleClicked.connect(self.handler)

        self.database_d = db_download.Download(self)
        my_dictionary = {}
        if self.database_d.connection:
            cursor = self.database_d.conn.cursor(dictionary=True)
            cursor.callproc('download_info')
            my_dictionary = {}
            for result in cursor.stored_results():
                my_dictionary = result.fetchall()
            my_dictionary = json.dumps(my_dictionary, indent=4, sort_keys=False,
                                       default=str)  # for convert timestamp -> now readable
            print("my dictionary now is better")
            print(my_dictionary)
        jfile = my_dictionary
        jdata = json.loads(jfile, object_pairs_hook=collections.OrderedDict)
        self.jdata = jdata
        # Find UI
        self.make_find_ui()
        # Tree
        self.tree_widget = self.tab_d.treeDownload  # QtWidgets.QTreeWidget()
        self.tree_widget.setHeaderLabels(["List of available", "Data"])
        self.tree_widget.header().setSectionResizeMode(QtWidgets.QHeaderView.Stretch)
        root_item = QtWidgets.QTreeWidgetItem(["Packets:"])
        self.recurse_jdata(jdata, root_item)
        self.tree_widget.addTopLevelItem(root_item)
    except Exception as e:
        LOG.error(f"JsonView error : {e}", exc_info=True)
        print(f"JsonView error : {e}")
```

Figure 5.25: init JsonView class

JsonView, representing the center of the download part, presents a connection to the *J4_CLOUD database* that using the **download_info** stored procedure allows you to download all the packets and encode the information in a dictionary. JsonView presents an instantiation of the GUI where **QLineEdit**() and **SearchButton** are taken by the MainWindow and connects them to the search and clicked functions, moreover within the **QTreeWidget** instantiates and creates (*hardcoded*) the single **QTreeWidgetItem**, elements that represent the single packets downloaded from the database.

When on the GUI side the user clicks inside our **QTreeWidget** what happens is a call to the **_click** function that checks if the element on which you clicked refers to an element connected to a packets, in case of positive result a dialog window opens which presents the description (if available Through the **_single_click** function of the **window_description** class with the reference *packet id*). If the user proceeds with the download the **handler** function is called, passing him the reference column (identifier of the **QTreeWidgetItem**) which in turn makes a second check on the validity of the **QTreeWidgetItem** and calls the **download_packet** function of the **DB_download** class (5.26).

```
class Download:
    def __init__(self, _):
        self.conn = None
        self.connection = False
        self.id = None

        self.connection, self.conn = self.connect()

    def connect(self):
        """ Connect to MySQL database """

        db_config = read_db_config()
        try:
            LOG.info('Connecting to MySQL database..')
            self.conn = MySQLConnection(**db_config)

            if self.conn.is_connected():
                LOG.info('Connection established.')
                self.connection = True
            else:
                LOG.error('Connection failed.')

            return self.connection, self.conn

        except Error as e:
            LOG.error(f'Error occur: {e}', exc_info=True)
```

Figure 5.26: DB_download class

The **DB_download** class takes care of creating the folders in the *destination folder* of the user who is downloading the packet, and of downloading the packet passed by *JsonView* through the **download_packet** function, passing it as the *packet id* argument. **DB_download** provides some additional features such as *overwriting* downloads and *opening the final folder*.

5.3 Additional features

The additional features developed to lighten, speed up and support the use of Jazz Cloud are mainly 4.

1. Exporter
2. File
3. View
4. Help

We will present these features starting from a functional approach, showing the interaction and purpose, then looking at the code.

5.3.1 Exporter

Although the **exporter** is not one of the features considered as the focus of the thesis, once developed it has become the *nerve center* of the upload algorithm. This allows to parse *.fls* by checking that all dependencies on *.tfl* files are verified, then checks that all dependencies of *.tfl* files on related *.tst* files are checked. It also has an additional level of depth that verifies the characteristics of *.tst* files by putting them in clusters and verifying that in case they need support files or other perl files, they are imported correctly. Once all dependencies have been checked, they are brought and zipped into a new folder which will then be uploaded to the cloud. This allows for overhead-free and checked packets, without allowing incomplete tests to be loaded without the user being aware of it. It also performs the consistency check function to validate tests before they are uploaded.

5.3.2 File

File offers three macro functions: **Upload**, **Download**, **exit**. Upload allows you to move to the upload area or to upload the last Test File Open in Jazz, download allows you to move to the Download section and Exit to close the application.

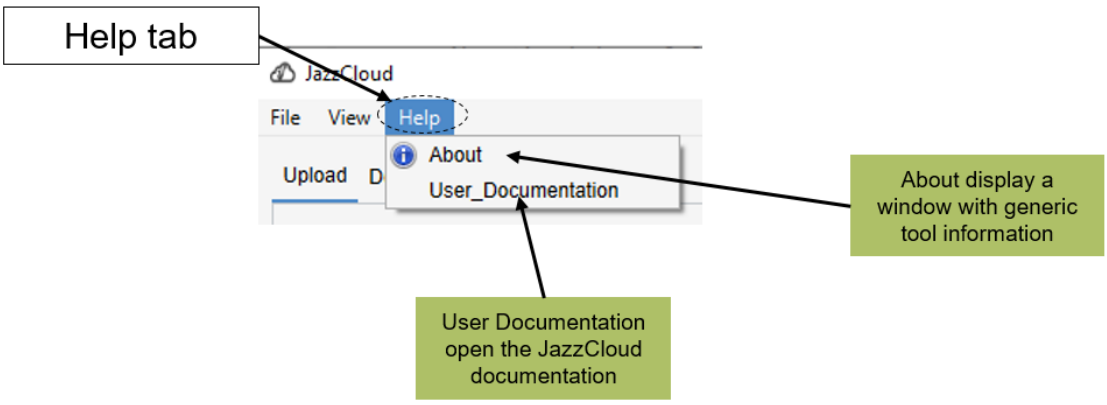

Figure 5.27: File menu

Upload item has a submenu that allows two different actions:

1. **Upload current** allows to upload the last open testfile in Jazz, reading the ini.conf file
2. **Search for upload** allows to move the tab to the Upload section so as to use the Folder tree for selection

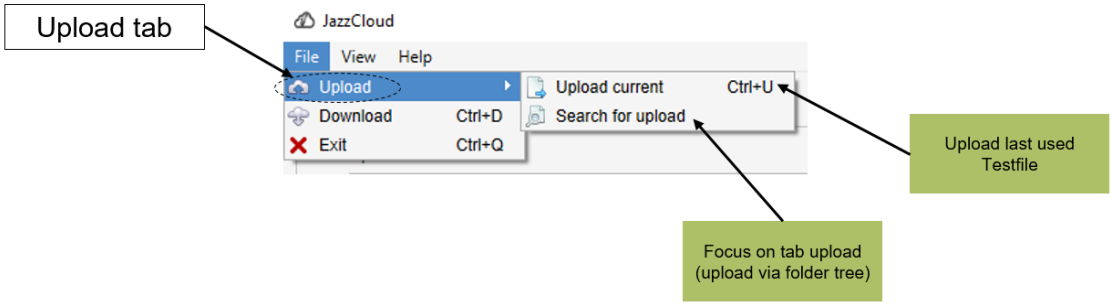

Figure 5.28: File -> Upload -> submenu

5.3.2.1 Upload current

This feature allows you to read the *ini.conf* file inside the CLI settings *ini.conf* folder of Jazz and pull up the information, this through the **_upload_last** function present in *main.py* which is connected to the GUI button.

```
def _upload_last(self):
    try:
        self.current = Upload()
        print(self.current.current_test_path, "test path")
        print(self.current.current_ini, "file conf ini")
        if self.current.current_test_path is not None and os.path.exists(self.current.current_test_path):

            if self.current.current_ini is None:
                a = self.ask_for_open_config(self.current.current_test_path)

                if a:
                    path_last_ini = None
                    dialog = QFileDialog(self)
                    dialog.setWindowTitle("Select Config File")
                    dialog.setLabelText(dialog.FileName,
                                        f"JC choose .ini file related to {os.path.basename(self.current.current_test_path)} :")
                    dialog.setNameFilter("File (*.ini)")
                    dialog.setViewMode(QFileDialog.Detail)
                    if dialog.exec():
                        self.current.current_ini = dialog.selectedFiles()

                if self.current.current_ini is None:
                    LOG.error("You must select a conf.ini file")
                    print("You must select a conf.ini file")
                print(f"name file conf {self.current.current_ini}")
            else:
                print("ok last files")
        if self.current.current_test_path is not None and self.current.current_ini is not None:
            filename = []
            filename.append(self.current.current_ini)
            self.load_testfile_from_explorer(self.current.current_test_path, filename)
        else:
            LOG.error("Impossible upload last TestFile")
            print("Impossible upload last TestFile")

    except Exception as e:
        LOG.error(f"generic error upload last {e}", exc_info=True)
```

Figure 5.29: _upload_last function

This function (5.29) calls the **Upload class**, creating an instantiation that takes care of obtaining, reading and, if necessary, writing the Jazz *ini.conf file*, extracting the data we are interested in and validating the existence of these paths. After which the **_upload_last** function in case of missing config file allows the user to select one through a **QFileDialog** in case of positive result the **load_testfile_from_explorer** function is called which allows to resume the upload in a canonical manner, already passing the path of the *test file* and that of the *config file*.

5.3.2.2 Download tab

The download section allows you to move to the correct tab using the function below and instantiating the *JsonView* class in case there were updates caused by an upload just performed, moving between tabs works as a **refresh**.

```
def position_tab_download(self):
    if self.j.tree_widget:
        self.j.tree_widget.clear()

    self.j = JsonView(self, self.window_description)
    self.features_tabs.setCurrentIndex(1)
```

Figure 5.30: Tab download piece of code

5.3.2.3 Exit

The **Exit** function iterates and deletes the children of the different layouts and possible open windows, so as to close all dependencies opened by the JazzCloud tool.

```
def closeEvent(self, event):
    button = QMessageBox.question(self, "Exit", f"{'Do you really want to exit JazzCloud?':80}",
                                  QMessageBox.Ok | QMessageBox.Cancel)
    if button == QMessageBox.Ok:

        for idx in reversed(range(self.gridLayout_7.count())):
            self.gridLayout_7.itemAt(idx).widget().setParent(None)

        for idx in reversed(range(self.gridLayout.count())):
            self.gridLayout.itemAt(idx).widget().setParent(None)

        if self.dlg.isVisible():
            self.dlg.close()

    else:
        event.ignore()
```

Figure 5.31: Exit function piece of code

5.3.3 View

View provides the user with a Display Log that allows you to see all the significant prints for understanding what happens in the *presentation layer*, *business layer* and in the *database layer*.

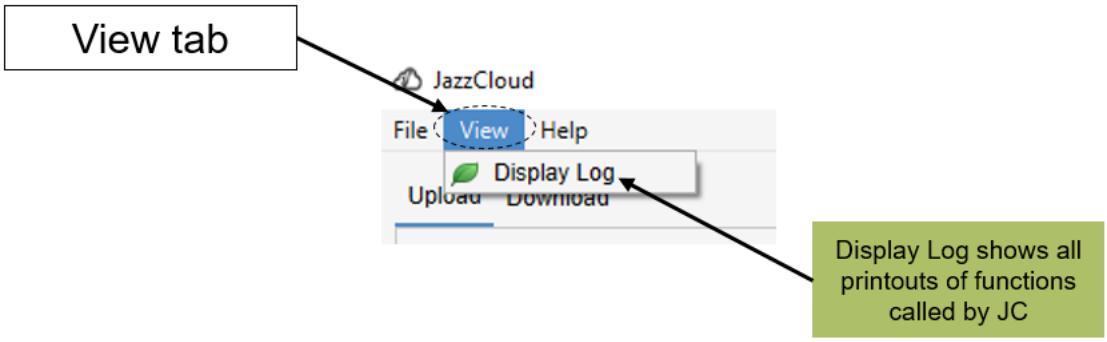

Figure 5.32: View menu

When Display log has been activated, a window opens (5.33) showing all the logs generated from that moment on by the JazzCloud tool.

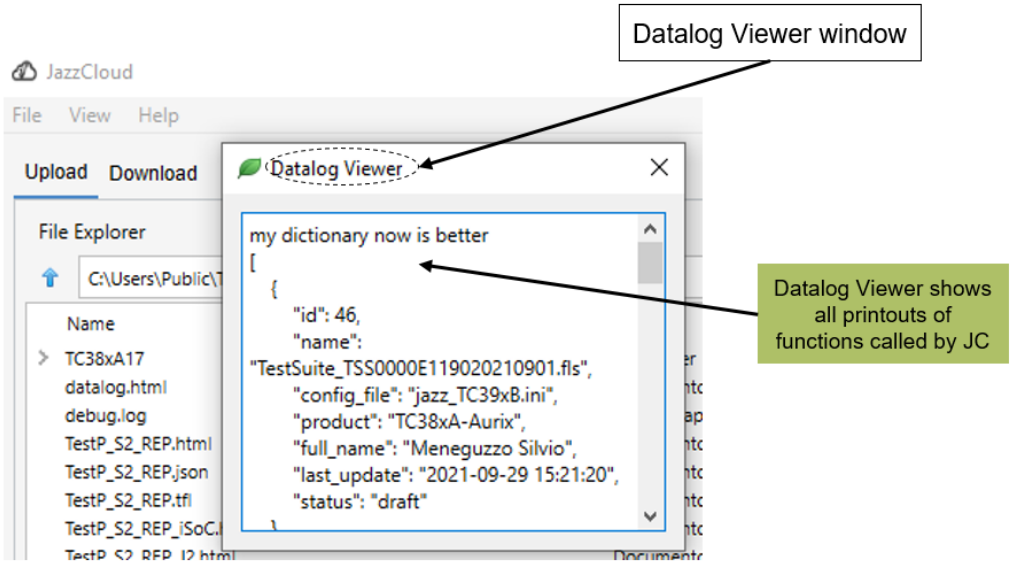

Figure 5.33: Datalog viewer window

5.3.3.1 Display Log code focus

Display Log is represented by the **MyDialog (QDialog)** class which is instantiated when the GUI side button is clicked, this class takes care of capturing all the **prints** scattered along the code when they are printed, this allows the user to check the *flow of actions* it is doing more precisely, the **LOGs** are redirected to the *debug.log* file in the *debug folder*.

5.3.4 Help

Help menu contains two items:

1. **About**: provides version and author information
2. **User Documentation**: provides the information to use and navigate within JazzCloud

```
class MyDialog(QDialog):
        def __init__(self):
                super().__init__()

                self._console = QTextBrowser(self)

                layout = QVBoxLayout()
                layout.addWidget(self._console)
                self.setLayout(layout)

                XStream.stdout().messageWritten.connect(self._console.insertPlainText)
                XStream.stderr().messageWritten.connect(self._console.insertPlainText)
```

Figure 5.34: Datalog viewer window piece of code

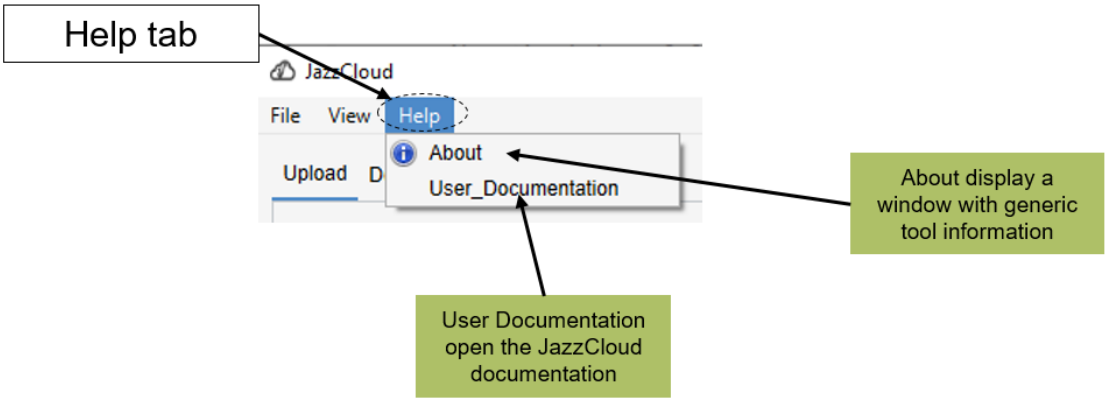

Figure 5.35: Help menu

5.3.4.1 About function

About is represented by a connect which call a **QMessageBox**:

```
self.actionAbout_JC.triggered.connect(lambda: QMessageBox.information(self, 'About', HELP_TEXT))
```

Figure 5.36: About connect

HELP_TEXT is linked in the constants/general.py directory.

```
HELP_TEXT = f"JazzCloud version 1.0.0\n\n"\
    "Developed by\n"\
    "+ Silvio Meneguzzo\n\n"\
    "Based on PySide6 library\n\n"\
    "Some icons by Yusuke Kamiyamane (Fugue Icons)\n"\
    "Licensed under a Creative Commons Attribution 3.0 License"
```

Figure 5.37: Help Text

5.3.4.2 User_Documentation

The *help* part of the documentation has been structured to have to modify only a *json file* in case you change the link to the documentation, this file is called **helps.json** and is present in the **constants external_files helps.json** folder, the function that obtains this file and to validate the path obtained is the **_init_help_tools** function which takes care of dynamically adding the action and connecting it to the appropriate connect that will call the **_launch_external_tool** function in case of click on the GUI, which will take care of opening the page written in the *helps.json* file already validated by the calling function *_init_help_tools*

```
def _init_help_tools(self):
    helps_dict = get_dict_from_json_file(paths.HELPS_JSON)
    if not helps_dict:
        return
    for key in helps_dict['helps'].keys():
        path = helps_dict['helps'][key]
        validator = URLValidator()
        try:
            validator(path)
        except ValidationError:
            if not os.path.exists(path):
                LOG.warning(f'Help {key} related path ({path}) does not exist')
                continue
        action = QAction(key, self.menuHelp)
        action.setData(path)
        action.triggered.connect(self._launch_external_tool)
        self.menuHelp.addAction(action)
```

Figure 5.38: _init_help_tools

5.4 Database

In our project we adopt MySQL relational database to store meta information.
A relational database is a type of database that stores and provides access to entities that are related to one another. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables. In a relational database, each row in the table is a record with a unique ID called key. The columns of the table hold attributes of the data, and each record usually has a value for each attribute, making it easy to establish the relationships among entities.

5.4.1 ER (Entity Relationship) Model

ER model stands for an **Entity-Relationship model**. It is a high-level data model. This model is used to define the data elements and relationship for a specified system. It develops a conceptual design for the database. Below we can see the Jazz Cloud ER Model.

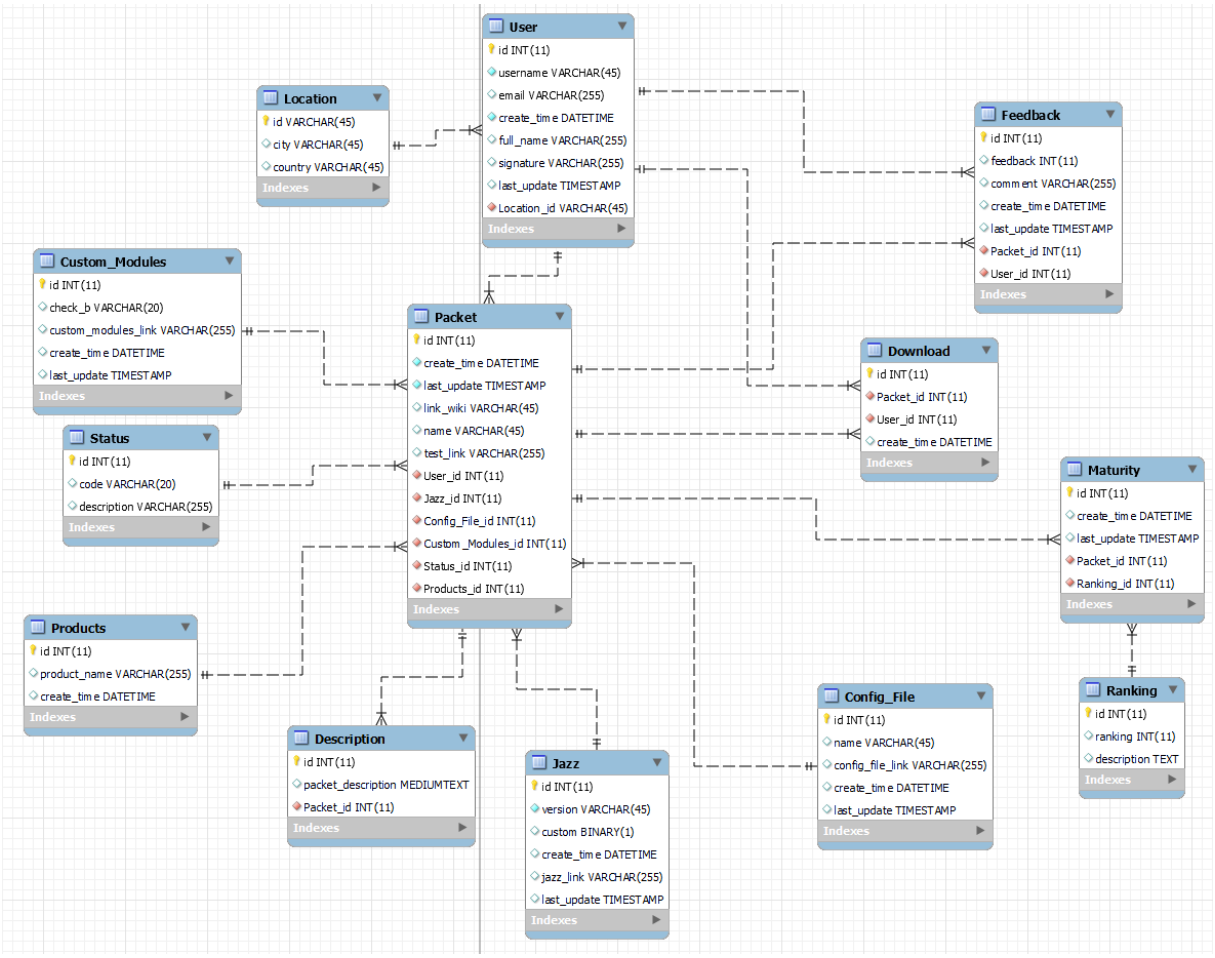

Figure 5.39: ER Model of J4_CLOUD

The database consists of 13 entities. In the ER diagram represented in figure 5.39 it can be seen that the fulcrum is the **packet entity** (see figure 5.40).seeing how the database works you can better understand how Jazz Cloud works.

5.4.1.1 Packet

Starting from *Packet* entity, the information is loaded through a stored procedure called by python code shown previously in the upload section, that allows us to insert data such as: a link to the Test File documentation (confluence page planned for the third release, *link_wiki*), package *name*, link to the package loaded on artifactory (planned for the third release, *test_link*), and a series of *foreign keys* referring to the user of the packets, information on the Jazz version used, information on config file (configuration *.ini* file), custom modules referring to perl / python modules modified by the user in local that can be used for the correct functioning of the Test file with Jazz, Status of the flow (complete, incomplete or eliminated), and product (refers to the product to which to apply the Test file). Id, create_time and last_update are self-filled attributes.

```
--
-- Table structure for table `Packet`
--

DROP TABLE IF EXISTS `Packet`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `Packet` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `link_wiki` varchar(45) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `name` varchar(45) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `test_link` varchar(45) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `User_id` int(11) unsigned NOT NULL,
  `Maturity_id` int(11) NOT NULL,
  `Jazz_id` int(11) NOT NULL,
  `Config_File_id` int(11) NOT NULL,
  `Custom_Modules_id` int(11) NOT NULL,
  `Status_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_Packet_User_idx` (`User_id`),
  KEY `fk_Packet_Maturity1_idx` (`Maturity_id`),
  KEY `fk_Packet_Jazz1_idx` (`Jazz_id`),
  KEY `fk_Packet_Config_File1_idx` (`Config_File_id`),
  KEY `fk_Packet_Custom_Modules1_idx` (`Custom_Modules_id`),
  KEY `fk_Packet_Status1_idx` (`Status_id`),
  CONSTRAINT `fk_Packet_Config_File1` FOREIGN KEY (`Config_File_id`) REFERENCES `Config_File` (`id`) ON UPDATE CASCADE,
  CONSTRAINT `fk_Packet_Custom_Modules1` FOREIGN KEY (`Custom_Modules_id`) REFERENCES `Custom_Modules` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `fk_Packet_Jazz1` FOREIGN KEY (`Jazz_id`) REFERENCES `Jazz` (`id`) ON UPDATE CASCADE,
  CONSTRAINT `fk_Packet_Maturity1` FOREIGN KEY (`Maturity_id`) REFERENCES `Maturity` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `fk_Packet_Status1` FOREIGN KEY (`Status_id`) REFERENCES `Status` (`id`) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `fk_Packet_User` FOREIGN KEY (`User_id`) REFERENCES `User` (`id`) ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
```

Figure 5.40: Packet table code

5.4.1.2 User and Location

User (5.41) is the one who uploads the *Packet* to jazz cloud, and is also the possible creator of this Test file, this information is not entered by hand by the user, but is automated by a python script inside Jazz Cloud which once the process of packet upload is running , the system obtains user information via signature and username referenced by the login system in the Infineon intranet. From these, the signature is broken down from which it is possible to obtain reference countries and cities which are also inserted through a stored procedure in the **Location** table (5.42), which in turn is indexed through a foreign key in the User table.

```
--
-- Table structure for table `User`
--

DROP TABLE IF EXISTS `User`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `User` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(45) COLLATE utf8mb4_general_ci NOT NULL,
  `email` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `full_name` varchar(45) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `last_update` timestamp NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  `Location_id` varchar(45) COLLATE utf8mb4_general_ci NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_User_Location1_idx` (`Location_id`),
  CONSTRAINT `fk_User_Location1` FOREIGN KEY (`Location_id`) REFERENCES `Location` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci COMMENT='JazzCloud user data';
/*!40101 SET character_set_client = @saved_cs_client */;
```

Figure 5.41: User table code

```
DROP TABLE IF EXISTS `Location`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `Location` (
  `id` varchar(45) COLLATE utf8mb4_general_ci NOT NULL,
  `ifx_department` varchar(45) COLLATE utf8mb4_general_ci NOT NULL,
  `city` varchar(45) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `country` varchar(45) COLLATE utf8mb4_general_ci DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
```

Figure 5.42: Location table code

5.4.1.3 Custom Modules, Products, Jazz and Config File

Custom Modules, Products, Jazz and Config File are entities that are autocompleted through stored procedures using the implicit data given by the Jazz parent system and by the config file uploaded. These informations are important for the correct use of the loaded test packet and to reproduce it correctly in a new system.

5.4.1.4 Download

The download table (5.43) has two foreign keys, one to packet and one to user, as well as having a create_time field. This entity is populated through a stored procedure, called by the python code when a packet is downloaded, tracing: who downloads that test, the test itself and the exact moment in which all this happens.

```
--
-- Table structure for table `Download`
--

DROP TABLE IF EXISTS `Download`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `Download` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `n_download` int(10) unsigned DEFAULT NULL,
  `Packet_id` int(11) unsigned NOT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_Download_Packet1_idx` (`Packet_id`),
  CONSTRAINT `fk_Download_Packet1` FOREIGN KEY (`Packet_id`) REFERENCES `Packet` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
/*!40101 SET character_set_client = @saved_cs_client */;
```

Figure 5.43: Location table code

5.4.1.5 Maturity and Ranking

The *maturity* table contains two foreign keys, one to packet and one to **Ranking**,moreover an update and create field. The maturity table indicates the maturity of a packet, this is given by the ranking associated with it, and this association takes place through a script that checks the status of the packet, if package descriptions have been entered and based on positive feedback. The Ranking table contains 4 values (see table below).

Id ranking	Description
1	lacking in information
2	packet contains a description
3	packet with description and status available
4	packet with description, status available and given feedback

Table 5.1: Ranking table values

5.4.1.6 Description

The packet_description attribute is a medium text that allows you to write a description of the test uploaded in the upload phase which will then be shown on the download side before downloading that test file, it is optional, however it contributes to the maturity of the packet and sets the bases for the confluence page that will be created by instantiating a first description of the test flow.

5.4.1.7 Status

The *status* table contains a code associated with a specific description, the status identifies if a packet has been loaded correctly and is complete with all dependencies, if it has been loaded in an incomplete manner, thus leaving out some files necessary for the correct functioning of the test flow, or if it was deleted after being uploaded.

Id ranking	Description
100	Draft version, incomplete test packet
200	Complete test packet
300	Removed test packet

Table 5.2: Status table codes

5.4.1.8 Feedback

The *feedback* table includes two foreign keys, one to packet and one to user, contains a feedback from the user with a rating from 1 to 10 and a possible comment regarding the packet.

5.4.2 Stored Procedure

There are **17 Stored Procedures** that allow to modify the Database in a restricted and targeted manner, these functions are used by the *python code* to analyze and make queries of various kinds to be able to manipulate information and keep the *DB consistent* with the reference to Jazz Cloud Network folder.

The stored procedure is SQL statements wrapped within the **CREATE PROCEDURE** statement. The procedure may contain a conditional statement like IF or CASE or the Loops. It can also execute another stored procedure or a function that modularizes the code.

Benefits of a stored procedure are:

1. **Reduce the Network Traffic:** Multiple SQL Statements are encapsulated in a stored procedure. When you execute it, instead of sending multiple queries, we are sending only the name and the parameters of the stored procedure

2. **Easy to maintain:** The stored procedure are reusable. We can implement the business logic within an SP, and it can be used by applications multiple times, or different modules of an application can use the same procedure. This way, a stored procedure makes the database more consistent. If any change is required, you need to make a change in the stored procedure only

3. **Secure:** The stored procedures are more secure than the AdHoc queries. The permission can be granted to the user to execute the stored procedure without giving permission to the tables used in the stored procedure. The stored procedure helps to prevent the database from SQL Injection

The syntax to create a MySQL Stored procedure is shown in Figure 5.44.

```
Create Procedure [Procedure Name] ([Parameter 1], [Parameter 2], [Para-
meter 3] )
Begin
SQL Queries..
End
```

Figure 5.44: Template Stored Procedure

In the syntax:

1. The name of the procedure must be specified after the **Create Procedure** keyword
2. After the name of the procedure, the list of parameters must be specified in the parenthesis. The parameter list must be comma-separated
3. The SQL Queries and code must be written between BEGIN and END keywords

To execute the store procedure, you can use the CALL keyword in this way:
CALL [Procedure Name] ([Parameters]..)

1. The procedure name must be specified after the CALL keyword
2. If the procedure has the parameters, then the parameter values must be specified in the parenthesis

In this thesis work, all the entities and stored procedures are contained within the *JC_v4* database.

CHAPTER 6

CONSIDERATIONS AND CONCLUSION

In this chapter the results obtained from the implementation are presented and some considerations are shown. For the sake of this thesis work, the final result is described and compared to the previous state of art.

6.1 Considerations

Jazz Cloud was conceived as a plugin applicable to Jazz, viewed as a desktop application. However the integration of this tool with other UI has been foreseen. This approach is very different from the former one (state of art) represented by a web interface. This paradigm shift required a great effort as the existing tool was eliminated and replaced entirely. The entire cycle of a software product was analyzed, from the collection of requirements to the release and subsequent steps of maintenance and product enhancement.

The verification of the functioning of the new tool was also carried out through the analysis of some application cases. The most common application case of the new Jazz Cloud is shown in figure 6.1.

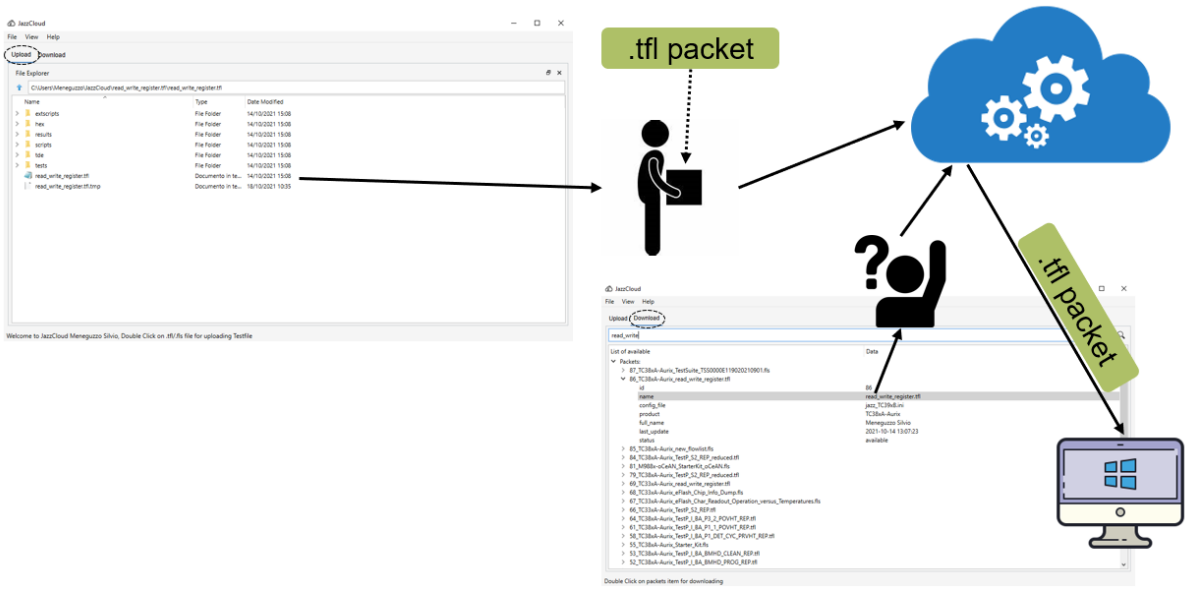

Figure 6.1: Jazz Cloud application case

In this case we can see as a *.tfl file* (file that includes multiple tests), is uploaded through the plug-in GUI, where, as illustrated above, the movement and information are traced through the relational databases (included in the speech bubble representing the cloud systems used) and then the packet is loaded into *JC network folder*. Subsequently, this same packet is made available via the Download graphical interface from any other computer that has access to the Infineon network, with the possibility of downloading the test file into the destination PC.

6.2 Results

The goal of this thesis was to develop a cloud system that is intuitive, easy to use and that automates the sharing process of tests flow. The purpose was completed with the creation of the first release of *Jazz Cloud v 1.0.0*. This has been tested and validated by the *Product Engineering team* and other stakeholders who have verified the solution. The first release covers all the requirements belonging to the *MVP-1*. In addition to this, a further beta release of the MVP-2 has also been developed and the next steps have been planned.

The result is already visible from the graphic interface, made as intuitive and automated as much possible, by avoiding the insertion of data that can be inferenced directly from the environment. The clean and simple interface can be seen in figure 6.2.

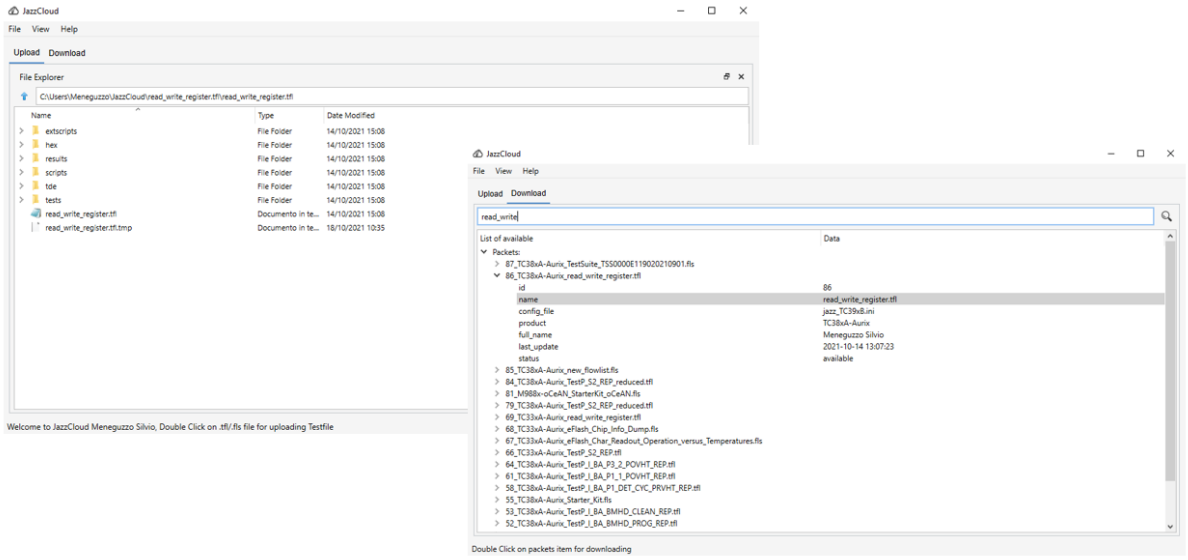

Figure 6.2: Jazz Cloud GUI upload and download view

6.3 Conclusion and Next Steps

The thesis project was evaluated in a very positive way, completing a tool designed and proposed in 2016, with an important paradigm change and with a modular and extensible structure. The goal was to release the **first MVP**, however it was also possible to release a beta version of the second MVP focused on more advanced and automated sharing features. The realization of the third MVP remains in the next steps (see figure 6.3), with a focus on correctly managing the three layer architecture.

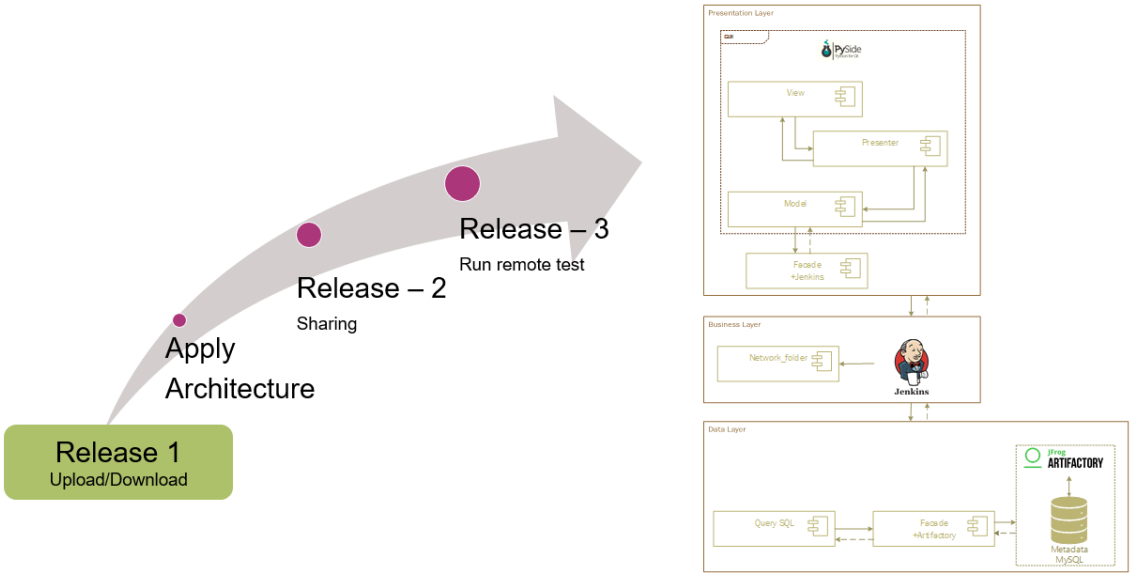

Figure 6.3: Steps Jazz Cloud

This thesis is the result of the work executed during a seven months stage at Infineon Technologies Development Center of Padua. The aims of the thesis are multiple: the first one is to analyze the software methodologies development used in the first version of Jazz Cloud, and all principles and rules given by stakeholders to create a consistent and reliable tool. Therefore a deep study of the actual implemented tool state of the art was the basic step to evaluate, if JAZZ Cloud, based on Jazz that is the Infineon tool used to read/write the memory content, and automate testing, of a microcontroller, satisfied the main requirements asked. Comparing it with the new requests makes possible to determinate the main limitations and issues of JAZZ Cloud state of art. Also users' feedback and software development measures offered a significant help to set up every necessary implementation choices. After a requirements and goals list creation, the second aim is the definition and implementation of a new application, which will improve the organization, manage and storage of test packets. So the main part of the project concerns the development of a standalone tool, Jazz Cloud, independent from JAZZ main tool but based on its classes. The achieved results are good and encouraging. Jazz Cloud works properly and does what it is designed for. The flexibility and the high level of portability of the tool simplify and improve the interchange of Jazz packets. Every step is realized following the software development principles and developed step by step in Python programming language by PyCharm and using the external libraries offered by Qt for Python, to build the Graphical User Interface (GUI). The first official Jazz Cloud version is released, as a complete and working release. The pre-release testing group has already provided good assessment and some hints for second MVP. This is the ending point of the thesis but also a starting point for future development. Improve all features, redesign GUI using latest version of Qt to get an even better appearance or provide an integration with Autochar Infineon tool needed for ultimate MVP-3, can all be new possible implementation choices to increment this new tool.

BIBLIOGRAPHY

[1] Infineon Technology official site
`https://www.infineon.com`

[2] Knoedler Stephanie, "*The company*", Infineon internal document

[3] Chiozzi Giorgio, "*Padova: about us*", Infineon internal document

[4] Cappelletti P., Golla C., Olivo P., Zanoni E., "*Flash Memories*", Kluwer Academic Publisher (1999)

[5] Infineon 32-bit TriCore microcontrollers
`https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller`

[6] Python main web site
`www.python.org`

[7] Bottin P., "*Introduction Flash Non Volatile Memories*", Infineon internal document, 2019

[8] Coppetta M., "*Flash test program development process – TestWare and Test Patterns*", Infineon internal document

[9] Carlesso L., "*Study, analysis and design of bitmapping tools for performances improvement in microcontroller analysis for embedded NVM*", Tesi di Laurea, Università degli Studi di Padova, 2018

[10] Rossi G., "*Study and realization of a "flow-sequencer" based on Apalis IMX6 system for automated test equipment*", Tesi di Laurea, Università degli Studi di Padova, 2020

[11] Gastaldo A., "*Concept, design and development of a software interface with memory modules and registers of automotive microcontrollers with embedded flash*", Tesi di Laurea, Università degli Studi di Ferrara, 2008

[12] Baschirotto A., "*JAZZ Cloud Phase2 Project I3 Milestone*", Infineon internal document, 2015

[13] Davide Di Lello, "*JAZZ Cloud User Guide*",Infineon internal document, 2016

[14] Davide Di Lello, "*JAZZ Cloud*",Infineon internal document, 2016

[15] Baschirotto A., "*JAZZ Cloud Project I7 Milestone*",Infineon internal document, 2017

[16] Coppetta M., "*TestWare Overview"*, Infineon internal document, 2016

[17] Spinato S., "*Jazz Training*", Infineon internal document, 2020

[18] DAP MiniWiggler
`www.infineon.com/miniWiggler`

[19] Shefkiu E., "*Algorithm implementation on IBIS and dedicated statistical studies on bitmaps*", Tesi di Laurea, Università degli Studi di Padova, 2021

[20] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides , "*Design Patterns: Elements of Reusable Object-Oriented Software* ", 1994

[21] George Reese , "*Cloud Application Architectures*", O'Reilly Media, Inc. (2009)

[22] Jenkins
`https://www.jenkins.io/`

[23] Artifactory
`https://jfrog.com/artifactory/`

[24] MySQL Workbench
`https://www.mysql.com/it/products/workbench/`

[25] Martin R., "*Clean Code: A Handbook of Agile Software Craftsmanship*", Prentice Hall PTR (2009)

[26] Structured Query Language, Wikipedia web site,
`https://en.wikipedia.org/wiki/SQL`