Technical programming documentation

Application for estimate the development effort of software projects

Index

List of figures

1. Introduction

The purpose of this file is to offer a detailed guide on the technical documentation of programming the application, making it easier for any developer with computer knowledge to understand in depth the operation of the project.

In the following sections, the development environment used will be detailed, with a particular focus on the Qt Creator IDE (Integrated Development Environment) and its tools. The structure of the project will also be described, including the modules into which it is divided, the libraries used and the documentation generated.

On the other hand, instructions will be provided for the deployment of the application, ensuring that any developer can replicate the production environment and ensure its proper operation.

Ultimately, this documentation is designed to provide a comprehensive understanding of the development process, facilitating both future maintenance of the application and its potential expansion or improvement.

2. Development Environment

The development environment is critical to the efficient creation of the application. Below are the technologies used in conjunction with the Qt Creator IDE configuration.

2.1. Technologies used

The technologies used for the correct development and operation of the application are the following:

- **Qt Creator IDE**[1]: Qt Creator is a high-performance integrated development environment (IDE) for cross-platform application development. It offers advanced tools for graphical interface design, code writing, debugging, and performance analysis. It is especially useful for projects that use the Qt framework and its bindings for Python, PySide6.

Illustration 1. Logo Qt

- **PySide6**[2]: PySide6 is the official Python binding for Qt 6. It provides access to all the functionalities of the Qt framework, including graphics, multimedia, networking, and much more. PySide6 allows you to develop desktop applications with modern, professional graphical interfaces using Python.

Illustration 2. Logo Pyside6

- **Python**[3]: Python is a high-level programming language known for its simplicity and readability. It is used as the primary language for developing application logic.

Figure 3. Logo Python

2.2. Configuring the Development Environment

To correctly configure the development environment, the following operations must be performed.

2.2.1. Installing Qt Creator and PySide6

1. **Download and install Qt Creator:**

    - Visit the official Qt download page: https://www.qt.io/download.
    - Download the Qt installer for our operating system (Windows, macOS, Linux).
    - Run the installer and follow the on-screen instructions.

2. **Select components during installation:**

    - During installation, make sure to select the following options:

        - Qt for Python: This will include PySide6 and all the necessary libraries.
        - Qt Creator: The integrated development environment that will be used to develop the application.

3. **Check the Installation:**

    - Open Qt Creator.
    - Go to Help > About Qt Creator, to verify that the installed version is correct.
    - Go to Tools > Options > Kits to make sure that the development kits for Python and PySide6 are configured.

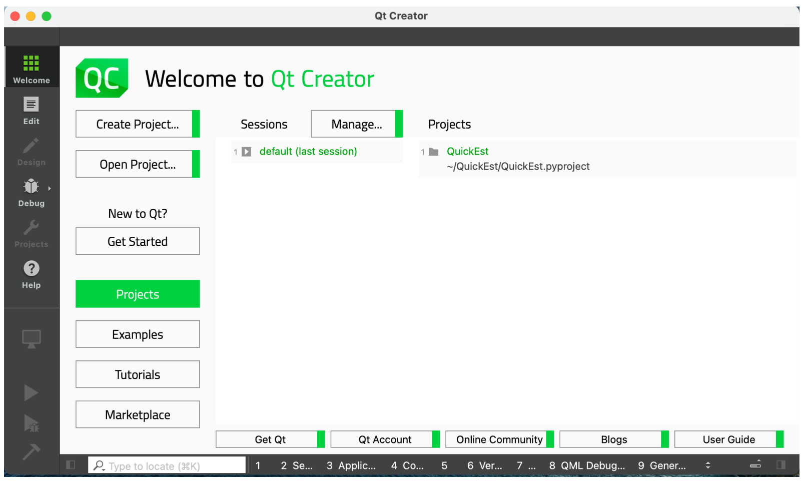


Figure 4. Qt Creator IDE Interface

2.2.2. Setting up a project in Qt Creator

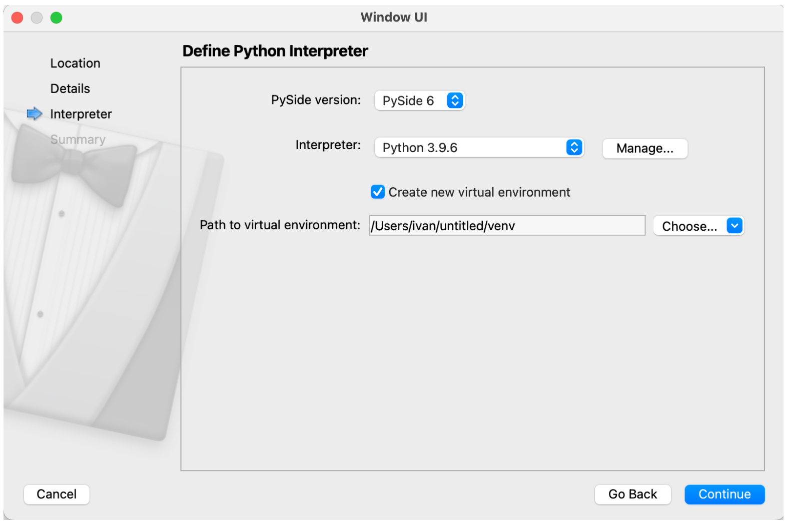
1. **Create a Python project with PySide6:**

    - Open Qt Creator.
    - Select File > New File or Project.
    - Choose "Python Application" and follow the instructions in the wizard.
    - Make sure to select the development kit that includes PySide6. The tool will automatically detect the version of Python installed on the system. If you do not have Python installed, follow these steps:

        - Visit the official Python download page: https://www.python.org/downloads/.
        - Download the appropriate installer for our operating system (Windows, macOS, Linux).
        - Run the installer and make sure to select the "Add Python to PATH" option before proceeding with the installation.

    - Select the "Create virtual environment" option. This is recommended for isolating project dependencies. To activate the virtual environment, run the following command in the IDE terminal from the project's root directory:

        - On Linux and MacOs: `source venv/bin/activate`
        - On Windows: `venv\Scripts\activate`



Define Python Interpreter

PySide version: PySide 6

Interpreter: Python 3.9.6

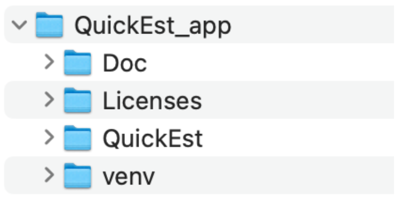
☑ Create new virtual environment

Path to virtual environment: /Users/ivan/untitled/venv

Figure 5. Creating a project with Pyside6

3. Project structure

The structure of the project is critical to keeping the code organized and making it easier to maintain and scale. A general outline of this structure is presented below, followed by a brief description of each of the main directories and files that make up the application. This section is designed to provide a clear and understandable view of how source code is organized, allowing developers to easily navigate and understand each component of the project.



```
QuickEst_app
    Doc
    Licenses
    QuickEst
    venv
```

Figure 6. General structure of the project

The QuickEst **directory** is the core of the application and contains all the essential components for its operation. Within this directory are subdirectories and files that handle the business logic, user interface, controllers, data models, and resources needed. Each subdirectory has a specific responsibility, ensuring a clear separation of concerns and facilitating code maintenance and scalability. In the following section, each of the modules that make up the application and its associated files will be described in detail.

The venv **directory** contains the Python virtual environment for the project. This virtual environment isolates project dependencies, ensuring that the necessary packages and libraries do not conflict with other projects or with the global Python libraries installed on the system. Using a virtual environment is a best practice in software development to ensure the reproducibility and consistency of the development environment.

As for the project documentation, the Doc **directory** allows us to store all the relevant information for the developers. This includes all the technical programming documentation necessary to understand and use the application. Keeping documentation up-to-date and organized is crucial to facilitating collaboration and long-term maintenance of the project.

On the other hand, in the Licenses directory, we will find the documentation of the licenses of use for the software. This directory includes the full texts of the licenses, providing the necessary legal information about the terms of use, distribution, and limitations of the various dependencies and components of the software. The licenses ensure that the use of each part of the software complies with the legal agreements established by the original creators of the libraries and tools used.

3.1. Organisation of the application

In this section, each of the modules of the main core of the application will be described in detail, detailing the files contained in them.

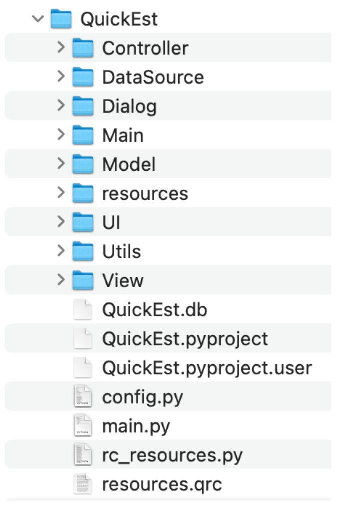


Figure 7. "QuickEst" Module

Before going into specific details about the organization of each module, we will proceed to describe the most relevant files:

- **QuickEst.db**: When the application is run for the first time, this file is generated, which represents the SQLite database used by the application and contains all the tables and data necessary for its proper functioning.

- **config.py:** This file defines several constants used throughout the application, including the states of database operations, file extensions, and resource paths.

- **main.py**: This file is the primary entry point for the QuickEst application. Configures the application, ensuring that only one instance is running and setting the locale and UI style. It also installs an event filter for dialogs and verifies the correct initialization of the database before displaying the main application window.

- **rc_resources.py**: This file is automatically generated and contains references to the graphic and other resources incorporated in the application. It is used by PySide to access images, icons, and other media resources defined in resources.qrc and stored within the "resources" folder.

- **resources.qrc**: This file is a Qt resource file that defines all the graphical and media resources used in the application. It is used to generate the rc_resources.py file, allowing resources to be accessible within the Python code.

As for the configuration of the project with Qt Creator we find two reelevenates files:

- **QuickEst.pyproject**: Project configuration file, used by Qt Creator to manage and store project configuration. This file contains information about file paths, build configurations, development kits, and other IDE-specific configurations. It is essential to ensure that the development environment is properly configured every time the project is opened in Qt Creator.

- **QuickEst.pyproject.user**: This is a user-specific configuration file that uses Qt Creator. This file stores personal and local settings, such as user preferences, system-specific paths, and other settings that should not be shared between different developers. This file ensures that each developer can customize their development environment without affecting the other team members.

3.1.1. 'Main' Module

The Main module contains the main files that launch the application and configure the graphical user interface (GUI). This module is critical to the operation of the application, as it establishes the MVC (Model-View-Controller) architecture and handles user interactions with the main interface.

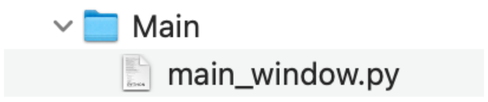


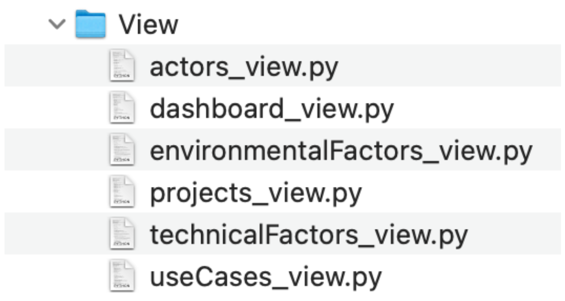
Figure 8. Main Module

It consists of the following files:

- **main_window.py:** This file implements the MainWindow class, which represents the main window of the application. It is responsible for initializing the graphical user interface (GUI), configuring the MVC architecture, and handling user interactions. In addition, it manages text font loading, menu settings, and in-app navigation.

3.1.2. 'View' Module

The View module contains the views that handle the presentation of data. These views are responsible for the graphical user interface (GUI) and how the data is displayed on screen.



View
actors_view.py
dashboard_view.py
environmentalFactors_view.py
projects_view.py
technicalFactors_view.py
useCases_view.py

Figure 9. View Module

The files contained in this module are as follows:

- **actors_view.py**: This implements the ActorsView class, which represents the view for actor management. It is in charge of the user interface related to the management of actors, providing functionalities to view, add, edit and delete them. It includes the configuration of tables that show data from the actors, buttons to perform actions, and a search bar to filter information.

- **dashboard_view.py**: It implements the DashboardView class, which represents the dashboard view, responsible for displaying various metrics and graphs related to performance and the distribution of efforts in the application. In addition, it allows for reporting and conversion factor settings for the ongoing project.

- **environmentalFactors_view.py**: It implements the EnvironmentalFactorsView class, which represents the view for environmental factor management. It is responsible for the user interface related to the configuration and monitoring of environmental factors that can influence the project. It provides functionality for adding, editing, and deleting these factors, as well as for viewing a summary of the data. It includes the configuration of tables that show the factors and allows users to manage these efficiently, ensuring that all relevant environmental aspects are considered in the planning and execution of the project.

- **'projects_view.py'**: This is where the ProjectsView class is implemented, which represents the view for project management. It is in charge of the user interface related to the configuration and management of projects, providing functionalities to create, open, edit and delete them. It also includes options to import and export projects, as well as to mark projects as favorites. The view manages a table that displays project details, allows for project search and filtering, and provides action buttons for performing specific operations. This view makes it easy for users to fully manage projects within the app.

- **'technicalFactors_view.py'**: This implements the TechnicalFactorsView class, which represents the view for managing technical factors. It is responsible for the user interface related to the configuration and monitoring of technical factors that may influence the project. It provides functionality for adding, editing, and deleting factors, as well as for viewing a summary of the data. It includes the configuration of tables that show the technical factors and allows users to manage these efficiently, ensuring that all relevant technical aspects are considered in the planning and execution of the project.

- **'useCases_view.py'**: This implements the UseCasesView class, which represents the view for use case management. It is in charge of the user interface related to the management of use cases, providing functionalities to visualize, add, edit and delete them. It includes the configuration of tables that show the data of the use cases, buttons to perform actions on them, and a search bar to filter information.

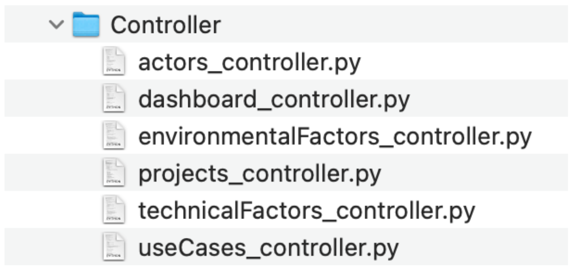
The following is an outline of the general structure followed to implement each of the views in the application:

```python
from PySide6.QtWidgets import QWidget, QVBoxLayout, QPushButton
from PySide6.QtCore import Signal

class View(QWidget):
    # Signals that will be connected to the controller methods
    user_action = Signal(dict)
    another_action = Signal(int)

    def __init__(self):
        super().__init__()
        self.init_ui()

    def init_ui(self):
        # User Interface Settings
        layout = QVBoxLayout()
        self.some_button = QPushButton("Do Something")
        layout.addWidget(self.some_button)
        self.setLayout(layout)

        # Connect signals to view slots
        self.some_button.clicked.connect(self.on_some_button_clicked)

    def on_some_button_clicked(self):
        # Emit signals with sample data
        self.user_action.emit({"key": "value"})
        self.another_action.emit(7)

    def update_display(self, result):
        # Method to update the UI with results
        self.result_label.setText(result)
```

The view defines the user interface and emits signals, which will be heard by the controller, when the user interacts with the UI. It also contains methods for updating the visualization based on the results coming from the driver.

3.1.3. 'Controller' module

The Controller module is a fundamental part of the application architecture, taking care of managing the business logic and the interaction between the user interface and the data logic. It acts as an intermediary that handles user events, updates the view, and processes data from the model. In addition, it ensures that each component of the application works consistently and efficiently, providing a smooth user experience.



```
Controller
    actors_controller.py
    dashboard_controller.py
    environmentalFactors_controller.py
    projects_controller.py
    technicalFactors_controller.py
    useCases_controller.py
```

Figure 10. Controller Module

In this module we find the following files:

- **actors_controller.py**: This file contains the ActorsController class, which is responsible for handling the business logic related to the actors in the project. It connects the view and the actor model, handling the loading, creating, updating, deleting and viewing of these. In addition, it manages the opening of dialogues for the management and updating of the weights of actors.

- **dashboard_controller.py**: This file contains the DashboardController class, which is responsible for managing the functionality of the dashboard. It is responsible for connecting the view and the dashboard model, managing the loading of the necessary data and updating information on actors, use cases, technical factors, environmental factors, distribution and results of effort estimation and report generation. In addition, it manages the opening of dialogs to adjust conversion factors and percentages.

- **environmentalFactors_controller.py**: This file contains the EnvironmentalFactorsController class, which is responsible for handling the business logic related to the environmental or environmental factors of the project.

It connects the view and model of environmental factors, managing the loading, updating and visualization of these.

- **projects_controller.py**: This file contains the ProjectsController class, which is responsible for handling the business logic related to projects. Coordinates the opening, creation, editing, viewing, deletion, and import/export of projects, connecting the view and the corresponding model. In addition, it handles the generation of reports in Excel format and the management of favorite projects.

- **technicalFactors_controller.py**: This file contains the TechnicalFactorsController class, which is responsible for handling the business logic related to the environmental or environmental factors of the project. It connects the view and model of environmental factors, managing the loading, updating and visualization of these.

- **useCases_controller.py**: This file contains the UseCasesController class, which is responsible for handling the business logic related to the project's use cases. Connect the view and the use case model, handling the loading, creating, updating, deleting, and viewing of use cases. In addition, it manages the opening of dialogs for the management and updating of use case weights.

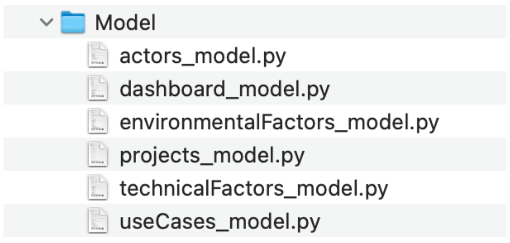
The following is a general outline of the structure followed to implement the application drivers:

```python
class Controller:
    def __init__(self, view, model):
        self.view = view
        self.model = model
        self.connect_signals()

    def connect_signals(self):
        # Connect sight signals to controller methods
        self.view.user_action.connect(self.handle_user_action)
        self.view.another_action.connect(self.handle_another_action)

    def handle_user_action(self, data):
        result = self.model.process_data(data)
        self.view.update_display(result)

    def handle_another_action(self, value):
        processed_value = self.model.process_value(value)
        self.view.update_display(processed_value)
```

The controller listens for signals from the view, calls methods from the model to get and process data, and then updates the view with the results.

3.1.4. 'Model' Module

The Model module is responsible for managing the data logic of the application. It acts as an intermediate layer between the database and the controllers, providing methods for accessing, manipulating, and storing the necessary data. Each file within this module contains a class that represents a specific entity and its interaction with the database, implementing functions to perform CRUD (Create, Read, Update, Delete) operations and other related operations.



Model
actors_model.py
dashboard_model.py
environmentalFactors_model.py
projects_model.py
technicalFactors_model.py
useCases_model.py

Figure 11. Model Module

The description of each of the files that make up this module is detailed below:

- **actors_model.py**: This file contains the ActorsModel class, which is responsible for managing the actors' data in the project. It provides methods for creating, updating, deleting, and loading actors from the database, as well as for updating weights and information related to their complexity. It also offers methods for exporting and obtaining summary data from actors.

- **dashboard_model.py**: This file contains the DashboardModel class, which is responsible for managing the dashboard data in the project. It provides methods for updating effort distribution percentages and conversion factor, as well as for calculating and obtaining key metrics related to project effort estimation.
- **environmentalFactors_model.py**: This file contains the EnvironmentalFactorsModel class, which is responsible for managing environmental factors in your project. It provides methods for updating environmental factors in the database, categorizing influences, loading data, and obtaining results from them. In addition, it allows you to insert default environmental factors for new projects and export existing ones.

- **projects_model.py**: This file contains the ProjectsModel class, which is responsible for managing the persistence and manipulation of projects in the application. Its functionalities include creating, updating, deleting, and retrieving projects in the database. In addition, it handles the import and export of project data to binary files, providing methods for calculating file hashes and handling database transactions. This class is critical to ensuring the integrity and consistency of project data throughout its lifecycle.

- **technicalFactors_model.py**: This file contains the TechnicalFactorsModel class, which is responsible for managing the technical factors of the project. It provides methods for updating technical factors in the database, categorizing influences, loading data, and obtaining results from them. In addition, it allows you to insert predetermined technical factors for new projects and export existing data.

- **useCases_model.py**: This file contains the UseCasesModel class, which is responsible for handling the use cases in the project. It provides methods for creating, updating, deleting, and loading use cases from the database, as well as for updating weights and information related to their complexity. It also provides methods for exporting and getting summary data for use cases.

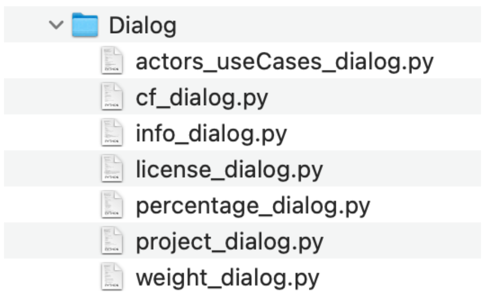
Below, we will show the general scheme followed for the implementation of the model classes:

```python
class Model:
    def process_data(self, data):
        # Code to process the data obtained
        return f"processed {data}"

    def process_value(self, value):
        # Code to process the entered value
        return f"processed {value}"
```

The model manages data and business logic, performing data query, processing, and updating operations. Returns the results to the controller for later viewing.

3.1.5. 'Dialog' Module

The Dialog module contains the dialog boxes used to interact with the user in various situations. These dialogs allow the user to make configurations, enter data, and manage different aspects of the application.



```
Dialog
    actors_useCases_dialog.py
    cf_dialog.py
    info_dialog.py
    license_dialog.py
    percentage_dialog.py
    project_dialog.py
    weight_dialog.py
```

Figure 12. Dialog Module

Each of the files within this module is detailed below:

- **actors_useCases_dialog.py**: This file implements the ActorsUCDialog class, which is responsible for managing the dialog for managing actors and use cases within the application. It is responsible for the user interface that allows you to create or edit actors and use cases, providing input fields for the details that are deemed necessary. It also includes validating the data entered and emitting signals when the data is saved, thus facilitating the user's interaction with these key elements of the application.

- **cf_dialog.py**: This file implements the CFDialog class, which is responsible for handling the dialog for conversion factor settings within the application. It is responsible for providing a user interface that allows you to set the conversion factor value. It includes an input field to edit this value and buttons to save or cancel the operation. The dialog box emits a signal when the data is saved, allowing the application to update and use the new conversion factor value as needed.

- **info_dialog.py**: This file implements the QuickestInfoDialog class, which is responsible for handling the information dialog about the QuickEst application. Use a Singleton pattern to ensure that only one instance of the dialog exists. It provides a user interface that displays details about the app, such as version, developer, contact information, and license. This dialog box is useful for providing users with relevant information about the app and their usage rights.

- **license _dialog.py**: This file implements the QuickestLicenseDialog class, which is responsible for handling the QuickEst application license information dialog. Use a Singleton pattern to ensure that only one instance of the dialogue exists. It provides a user interface that displays the terms of the GNU Lesser General Public License v3.0, as well as the disclaimer and copyright. This dialog box is useful for informing users about the terms under which the application is distributed and can be used.

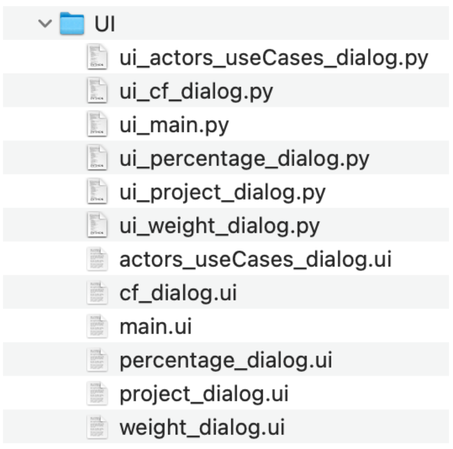
- **percentage _dialog.py**: This file implements the PercentageDialog class, which is responsible for managing the dialog box for setting percentages in the application. It is responsible for providing a user interface that allows adjusting percentages in different categories such as analysis, design, overhead, scheduling, and testing. It includes input fields for each category and a progress bar that shows the total percentage. It also emits a signal when data is saved, allowing the application to update and use the new percentage values as needed.

- **project _dialog.py**: This file implements the ProjectDialog class, which is responsible for managing the dialog box for project configuration within the application. It provides a user interface that allows you to create or edit projects, entering specific details about them. A signal is emitted when data is saved, allowing the application to update project information.

- **weight_dialog.py**: This file implements the WeightDialog class, which is responsible for managing the dialog box for weight configuration within the application. It provides a user interface that allows you to adjust the weights of different categories (simple, average, and complex) by using input fields. The dialog box is responsible for loading the initial values of the weights and emitting a signal when the data is saved, allowing the application to update and use the new weight values as needed.

3.1.6. 'UI' Module

The UI module contains the files automatically generated by the Qt UI compiler. These files define the structure and graphical user interface (GUI) elements for various components of the application. Using PySide6, they allow you to create, configure and manage the different dialogs and windows of the application, thus facilitating a consistent and functional user experience.



```
UI
    ui_actors_useCases_dialog.py
    ui_cf_dialog.py
    ui_main.py
    ui_percentage_dialog.py
    ui_project_dialog.py
    ui_weight_dialog.py
    actors_useCases_dialog.ui
    cf_dialog.ui
    main.ui
    percentage_dialog.ui
    project_dialog.ui
    weight_dialog.ui
```

Figure 13. UI Module

The .ui define the structure and appearance of graphical user interfaces (GUIs) in a Qt application. They were automatically generated from the designer of the IDE itself. Each .ui contains a description in XML format of the widgets, layouts, and properties that make up a specific window or dialog in the application. These files are essential for maintaining a clear separation between the application's logic and its visual presentation, making it easier to create and maintain complex user interfaces.

On the other hand, the corresponding .py files are generated from the .ui by using the Qt User Interface Compiler (uic). These contain the code necessary to configure and display the interfaces defined in the .ui files. They don't include application logic, but instead focus on setting the properties and layouts of UI elements, allowing developers to connect these elements to application logic in other modules.

To generate the ui_form.py files, the following command must be executed:

```
pyside6-uic form.ui -o ui_form.py
```

Below is the list of .ui and its corresponding .py :

- **actors_useCases_dialog.ui**: Defines the graphical interface for a dialogue that manages actors and use cases.
    - **ui_actors_useCases_dialog.py**: Contains the Ui_ActorsUCDialog class, which configures and displays the interface defined in actors_useCases_dialog.ui**.**

- **cf_dialog.ui**: Contains the definition of a dialog to adjust the conversion factor.
    - **ui_cf_dialog.py**: Contains class Ui_CFDialog, which figuresand displays the interface defined in cf_dialog.ui**.**

- **main.ui**: Defines the main interface of the application, encompassing the entire visual design. It includes all menus and views, providing a coherent and navigable structure for the user. This file centralizes the design of the application, ensuring that all interface elements are properly integrated and accessible.
    - **ui_main.py**: Contains the Ui_Main class, which configures and displays the interface defined in main.ui.

- **percentage_dialog.ui**: Describes the interface of a dialog for configuring percentages related to the distribution of effort for a project.
    - **ui_percentage_dialog.py**: Contains the Ui_PercentageDialog class, which configures and displays the interface defined in percentage_dialog.ui.

- **project_dialog.ui**: Defines the structure of a dialog for managing project configurations.
    - **ui_project_dialog.py**: Contains the Ui_ProjectDialog class, which configures and displays the interface defined in project _dialog.ui.

- **weight_dialog.ui**: Describes a dialog for adjusting the weights of different categories.
    - **ui_weight_dialog.py**: Contains class Ui_WeightDialog, which configures and displays the interface defined in weight _dialog.ui.

3.1.7. 'Utils' Module

The Utils module contains utility classes and functions that facilitate various in-app operations, such as widget configuration, custom dialog handling, button utilities, and more. These tools allow for greater modularity and code reuse across the application.

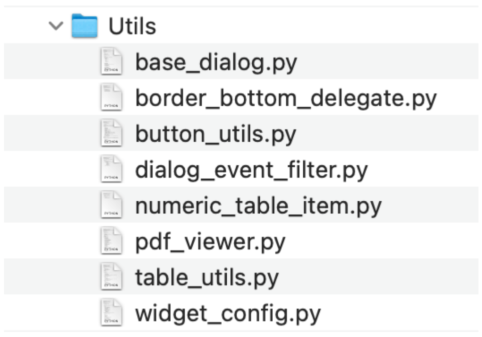


Figure 14. Utils Module

In this module we find the following utility files:

- **base_dialog.py**: Defines the BaseDialog class, which allows you to create custom dialogs with an icon and textual content, making it easier to standardize and reuse these components throughout your application.

- **border_bottom_delegate.py**: Defines the BorderBottomDelegate class, a custom delegate that draws a bottom border on table elements. This is accomplished by overwriting the paint method, providing a specific visual appearance for the tables in the user interface.

- **button_utils.py**: This file defines the ButtonUtils class, a class of utilities for configuring and managing button states and styles in the user interface. Provides methods for updating edit and delete buttons based on selecting rows in a table and for applying styles and icons to buttons.

- **dialog_event_filter.py**: This file defines the DialogEventFilter class, which is responsible for applying custom styles to certain dialogs when they are displayed in the application. Use an event filter to dynamically detect and apply the necessary styles.

- **numeric_table_item.py**: This file defines the NumericTableWidgetItem class, a custom class for items in a table that allows you to perform numeric sorting. The class overwrites the comparison method to ensure that items are compared and ordered numerically.

- **pdf_viewer.py**: This file defines the PDFViewer class, a widget for viewing PDF documents. It provides methods for uploading, navigating, and viewing PDF documents, as well as handling closing events. In addition, it includes a toolbar with buttons for forward, backward, and page selection.

- **table_utils.py**: This file defines the TableUtils class, which provides utilities for table operations, such as highlighting rows, handling move and click events, updating row numbers, and adding widgets such as buttons or checkboxes to table headers. This class allows for improved and more intuitive interaction with tables within graphical applications.

- **widget_config.py**: This file contains the WidgetConfig class, which provides utilities for configuring widgets in your application. It offers methods for enabling or disabling multiple widgets, creating comment buttons on tables, displaying message dialogs, and controlling the length of text entries and comments.

3.1.8. 'DataSource' Module

The DataSource module is crucial for database management within the application. This module provides an abstraction layer that allows you to handle the connection to the database and the initial configuration of the tables necessary for the operation of the application. Its main goal is to ensure that all interactions with the database are secure, efficient, and handled consistently.

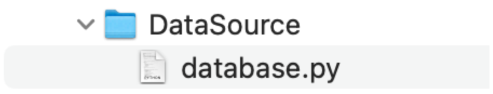


Figure 15. DataSource Module

In this module, we will find the following file:

- **database.py**: This defines the DataBase class, which implements a Singleton pattern to manage the connection to the local SQLite database. The class ensures that there is only a single instance of the database connection in the entire application, ensuring a single global access point to the application. It includes methods for starting and configuring the database, creating the necessary tables (projects, actors, use cases, technical factors, environmental factors, and parameters) if they do not exist. Database initialization also enables foreign key support to maintain referential integrity.

The structure followed for the implementation of the Singleton pattern is as follows:

1. Class _instance Attribute:

   ```
   _instance = None
   ```

   This class attribute stores the single instance of QSqlDatabase. It is initialized as None to indicate that the instance has not yet been created.

2. Class Method **get_instance()**:

   ```
   @classmethod
   def get_instance(CLS):
       if cls._instance is None:
           result = cls._init_db()
           if isinstance(result, str):
               Return result
           cls._instance = result
       Return cls._instance
   ```

   This method is the global access point to the single instance of the database. If the instance doesn't exist, call the _init_db() method to create and initialize it. If it already exists, simply return it. This ensures that the same instance is always used throughout the application.

3. Static Method **_init_db():**

   ```
   @staticmethod
   def _init_db():
       script_dir = os.path.dirname(os.path.abspath(sys.argv[0]))
       db_path = os.path.join(script_dir, "QuickEst.db")
       db = QSqlDatabase.addDatabase("QSQLITE")
       db.setDatabaseName(db_path)
       if not db.open():
           return "Cannot connect to database."

       query = QSqlQuery(db)
       if not query.exec("PRAGMA foreign_keys = ON"):
           return f"Failed to enable foreign keys:
   {query.lastError().text()}"

       existing_tables = db.tables()
       if not existing_tables:
           result = DataBase._setup_database(db)
           if result is not None:
               Return result
       Return return (second match)
   ```

This method initializes the connection to the SQLite database by setting the SQLite database name and opening the connection. Enables foreign key support to maintain referential integrity. It then checks to see if the necessary tables exist, and if not, calls _setup_database() to create them.

4. Static Method **_setup_database():**

```
@staticmethod
def _setup_database(db):
    Queries = [
        """
        CREATE TABLE IF NOT EXISTS projects (
            id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
            favorite BOOLEAN NOT NULL DEFAULT False,
            name TEXT NOT NULL UNIQUE,
            description TEXT,
            created_at TEXT NOT NULL,
            last_access TEXT NOT NULL
        )""",
        # ... More queries for other tables
    ]

    for query_text in queries:
        q = QSqlQuery(db)
        if not q.prepare(query_text):
            return f"Failed to prepare query:
{q.lastError().text()}"
        if not q.exec():
            return f"Failed to execute query: {query_text} -
{q.lastError().text()}"
    return None
```

This method prepares and executes the SQL queries needed to create the tables in the database, ensuring that they do not previously exist. The tables that are created include projects, actors, use cases, technical factors, environmental factors, and parameters. Each query is verified to ensure that it is prepared and executed correctly.

This description provides a clear and comprehensive view of how the Singleton pattern is implemented and used in the DataBase class to manage the connection to the SQLite database, ensuring a single global access point, and maintaining referential integrity through foreign key support.

4. Libraries

The project makes use of a combination of standard Python libraries and Qt-specific libraries to implement its functionality. This section describes the ones used.

4.1. Python libraries

The Python modules and libraries[4] imported into the application are as follows:

- **sys**: Provides functions and variables used to manipulate different parts of the Python runtime environment. Allows access to system-specific parameters and functions.

- **locale**: Used for localization settings, which affects the presentation of data in different formats depending on the user's language and region.

- **pandas**: A library that facilitates the manipulation and analysis of tabular data. It uses data structures such as DataFrames, which allow you to efficiently read, write, cleanse, and analyze data. It is ideal for working with data in formats such as CSV, Excel and SQL, being an essential tool for data scientists and analysts.

- **datetime**: Provides classes for manipulating dates and times. It is a fundamental library for any operation that requires handling temporal data.

- **OS**: Provides a way to interact with the operating system. It includes functionalities for manipulating the file system, executing system commands, and more.

- **zipfile**: Allows you to work with ZIP files, providing tools to create, read, write and extract these types of files.

- **openpyxl**: A library for reading and writing Excel (.xlsx) files. It is very useful for manipulating Excel spreadsheets from Python.

4.1. Qt libraries

The application uses several PySide6-compatible Qt modules[5], including:

- **PySide6.QtWidgets**: This module is essential for the implementation of graphical user interfaces (GUIs) in PySide6. It provides a wide variety of widgets that allow you to efficiently build and manage the visual elements of an application. It includes tools for the arrangement, interaction and presentation of data, facilitating the creation of functionally rich and highly interactive applications.

- **PySide6.QtCore**: Provides the core capabilities and core of Qt, ranging from object management and inter-object communication to manipulation of files, directories, and text strings. It provides a robust set of classes and mechanisms that support the basic operation of any application, including event management, timing, internationalization, and more.

- **PySide6.QtGui**: It is responsible for the management and representation of graphic elements and events, providing classes and functions for the creation, manipulation and visualization of graphics, images and typographies. It makes it easy to draw and manage 2D graphics, handle user events related to graphical input, and create visually appealing interfaces.

- **PySide6.QtSql**: Enables database management and execution of SQL queries, providing a set of classes for establishing connections with different database systems and executing SQL commands. It facilitates the integration of databases into applications, allowing the manipulation and consultation of data efficiently and securely.

- **PySide6.QtCharts**: This module is designed for the creation and visualization of charts, offering a series of classes that allow you to build various types of charts to represent data in a visual and understandable way. It provides tools to create interactive and custom charts, integrating seamlessly with other parts of the application for effective data visualization.

- **PySide6.QtPdf**: This module provides functionalities for viewing and managing PDF documents, allowing you to upload, view, and navigate through PDF files within an application. It offers tools for accurate representation of PDF content, as well as interaction with it through bookmark navigation and text search.

- **Pyside6.QtPdfWidgets**: This module complements PySide6.QtPdf by providing specific widgets for viewing and navigating PDF documents within a graphical user interface. It offers classes that allow PDF viewers to be integrated directly into applications, providing a smooth and functional user experience when interacting with PDF documents.

5. Deployment

Deploying or freezing an application is an important part of a Python project, as it involves pooling all the resources needed for the application to find everything it needs to be able to run on a customer's machine. However, because most large projects don't rely on a single Python file, distributing these applications can be challenging[6].

Starting with version 6.4, Qt offers a new tool called pyside6-deploy, which deploys the PySide6 application on all desktop platforms: Windows, Linux, and macOS. This tool is used for the deployment of the application due to its ease of use and the ability to obtain a more optimized executable[7].

By following the steps detailed in the official Qt documentation for deploying applications with pyside6-deploy, we managed to obtain the final executable of the application.
First, we run the pyside6-deploy main.py command and, once the pysidedeploy.spec file is generated, we adjust the parameters according to the needs of our project, including the icon, name, dependencies...among other necessary aspects of the application.

To learn more about how to use the tool, check out the official pyside6-deploy documentation[7].

6. Documentation

This section details the technical programming documentation generated for the project. This documentation includes comprehensive information about all classes and functions implemented in the source code, providing a complete and accurate reference for developers and software administrators. It is essential to understand the structure of the code, its components and its operation, thus facilitating the process of development and maintenance of the project.

For the generation of the project documentation, the following has been usedSphinx[8], a powerful and flexible tool that converts source code and comments into well-structured documentation. Sphinx it is widely used in the Python community due to its ability to generate documentation in multiple formats, such as HTML, PDF, ePub, among others.

Figure 16. Logo Sphinx

Some of the features of this tool are:

- **Automation**: Sphinx can automatically extract docstrings and comments from the source code, generating documentation without the need for manual intervention.
- **Multiple Formats**: Allows the export of documentation in various formats, facilitating its distribution and access.
- **Extensibility**: It has a large number of extensions and themes that allow you to customize the appearance and functionality of the documentation.
- **Interactivity**: Supports the inclusion of executable code examples, graphics, and other interactive elements.

In the QuickEst project, comments for functions and methods follow the NumPy/SciPy formatting style, which is characterized by a clear and concise structure that includes sections for description, parameters, return values, and exceptions. This format makes it easier for other developers to understand and use the code.

The scheme followed to make this type of comment is as follows:

```
"""
[Brief Description of Function or Method]

Parameters
----------
param1 : type
    Description of the first parameter.
Param2 : Type, Optional
    Description of the second parameter (optional). The default value
is 'valor_por_defecto'.

Returns
-------
guy
    Description of the return value.

Raises
------
ExceptionType
    Description of the condition under which this exception is thrown.
"""
```

To view the documentation, simply open the "index.html" file with any web browser (such as Google Chrome). The following image shows the location of this file within the project.

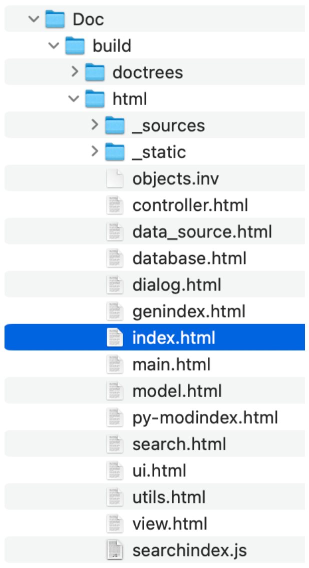
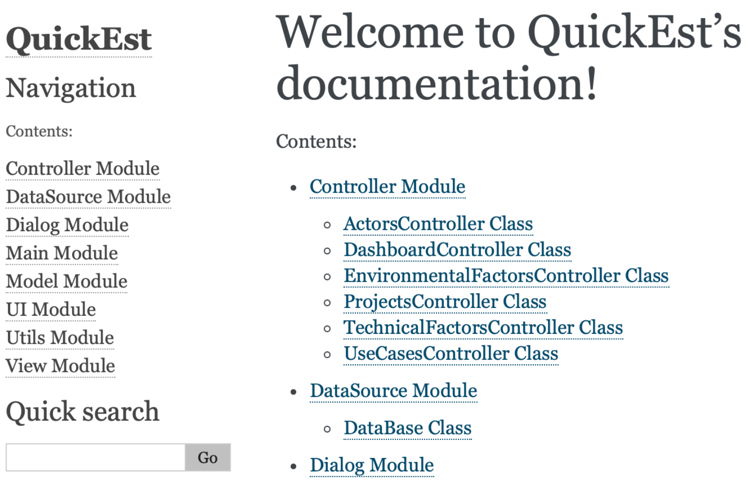

Doc
build
doctrees
html
_sources
_static
objects.inv
controller.html
data_source.html
database.html
dialog.html
genindex.html
index.html
main.html
model.html
py-modindex.html
search.html
ui.html
utils.html
view.html
searchindex.js

Figure 17. Documentation Files

In the following figure we can visualize the final result of the structure of the project documentation page:


QuickEst

Navigation

Contents:

Controller Module
DataSource Module
Dialog Module
Main Module
Model Module
UI Module
Utils Module
View Module

Quick search

Go

Welcome to QuickEst's documentation!

Contents:

- Controller Module
  - ActorsController Class
  - DashboardController Class
  - EnvironmentalFactorsController Class
  - ProjectsController Class
  - TechnicalFactorsController Class
  - UseCasesController Class
- DataSource Module
  - DataBase Class
- Dialog Module

Figure 18. Project Documentation

7. References

[1] The Qt Company. Qt Creator [Software]. https://doc.qt.io/qtcreator/

[2] The Qt Company. Qt for Python [Software]. https://doc.qt.io/qtforpython-6/

[3] Python Software Foundation. Python [Software]. https://www.python.org

[4] Python Software Foundation. Python 3.10 documentation [Documentation]. https://docs.python.org/3.10/

[5] The Qt Company. Qt 6 modules [Documentation]. https://doc.qt.io/qt-6/qtmodules.html

[6] The Qt Company. Deployment in Qt for Python [Documentation]. https://doc.qt.io/qtforpython-6/deployment/index.html

[7] The Qt Company. Deployment using Pyside6 deploy [Documentation]. https://doc.qt.io/qtforpython-6/deployment/deployment-pyside6-deploy.html - pyside6-deploy

[8] Sphinx. Sphinx documentation [Documentation]. https://www.sphinx-doc.org/en/master/