Criterion C: Development

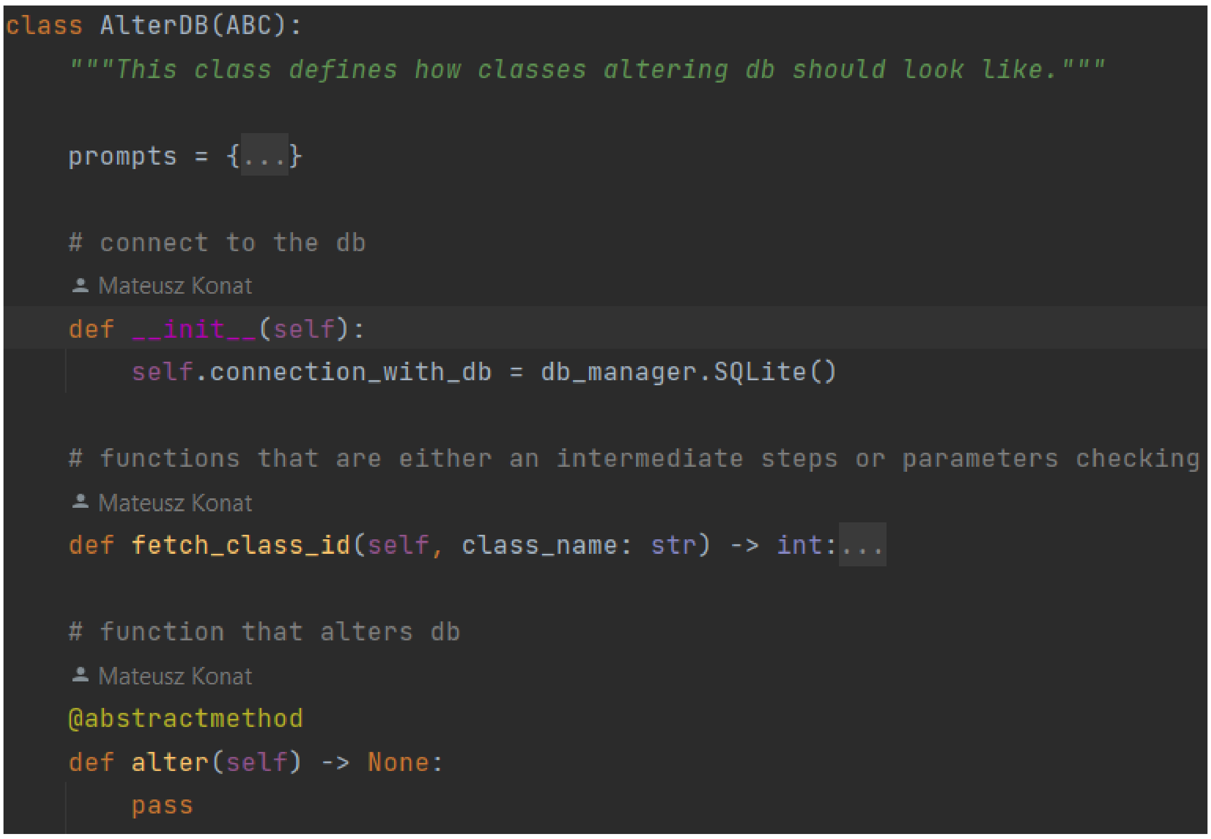
Techniques Used in CAS Manager Development

1. Adding and updating

1.1 Dependency inversion

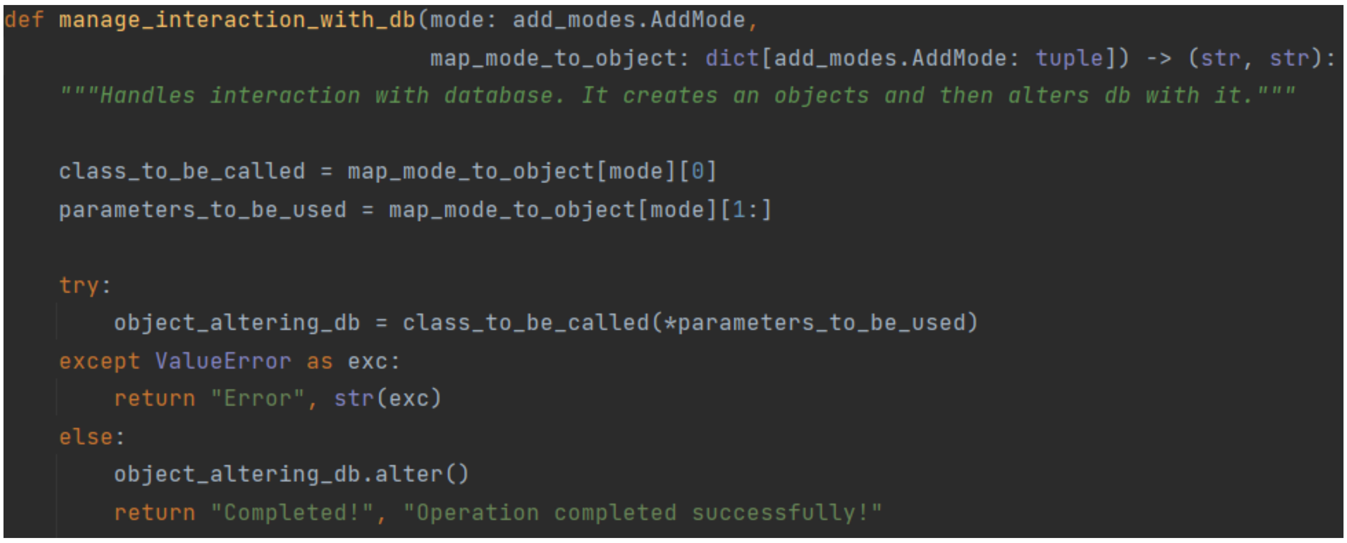
Dependency inversion allows high-level modules to be independent of the specific implementation details of low-level modules (Wikipedia, 2021). In the CAS Manager program, this concept is utilized through the creation of an abstract layer known as an Abstract Base Class (ABC). The purpose of this ABC is to separate the management of database transactions—such as adding and updating records—from the actual implementation of these transactions (Success Criteria 1-2).

By employing **dependency inversion**, CAS Manager gains several advantages. Firstly, it becomes more extensible, as introducing new functionalities for altering the database only requires creating another implementation for the ABC. This addition does not impact other parts of the codebase, making future updates and modifications easier to manage. Additionally, **dependency inversion** simplifies the debugging process by providing clearly-defined interfaces for each database-altering functionality. This clarity helps ensure the development of error-free software, enhancing the overall quality of the product delivered to the client.



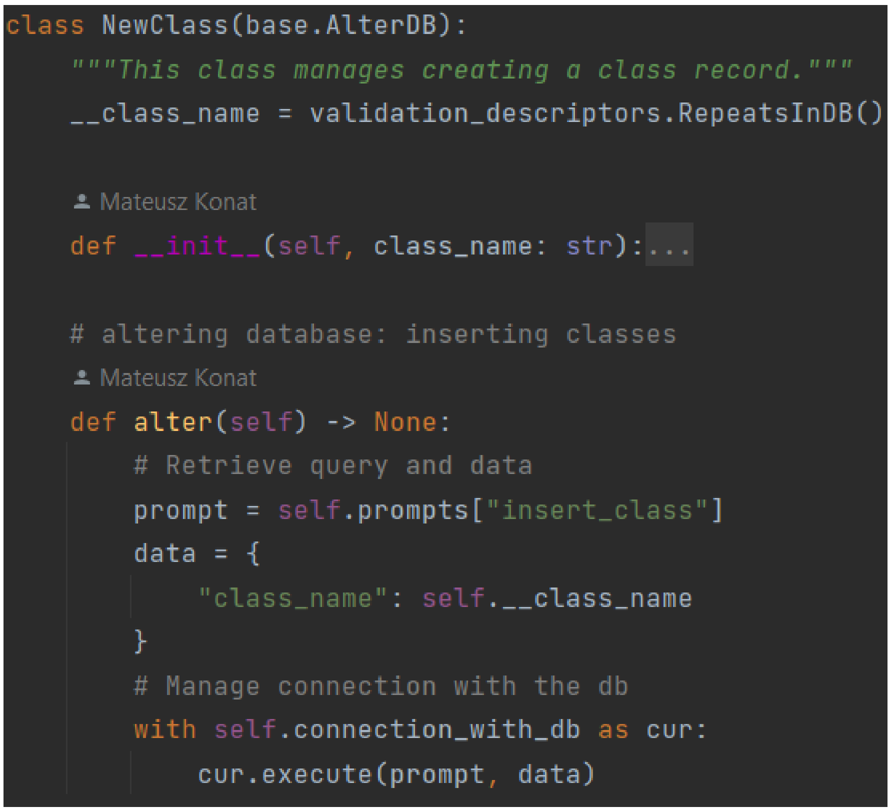
```python
class AlterDB(ABC):
    """This class defines how classes altering db should look like."""

    prompts = {...}

    # connect to the db
    Mateusz Konat
    def __init__(self):
        self.connection_with_db = db_manager.SQLite()

    # functions that are either an intermediate steps or parameters checking
    Mateusz Konat
    def fetch_class_id(self, class_name: str) -> int:...

    # function that alters db
    Mateusz Konat
    @abstractmethod
    def alter(self) -> None:
        pass
```

Figure 1. Defining ABC ('AlterDB()').



```python
def manage_interaction_with_db(mode: add_modes.AddMode,
                               map_mode_to_object: dict[add_modes.AddMode: tuple]) -> (str, str):
    """Handles interaction with database. It creates an objects and then alters db with it."""

    class_to_be_called = map_mode_to_object[mode][0]
    parameters_to_be_used = map_mode_to_object[mode][1:]

    try:
        object_altering_db = class_to_be_called(*parameters_to_be_used)
    except ValueError as exc:
        return "Error", str(exc)
    else:
        object_altering_db.alter()
        return "Completed!", "Operation completed successfully!"
```

Figure 2. Using abstraction layer – method 'alter()'.



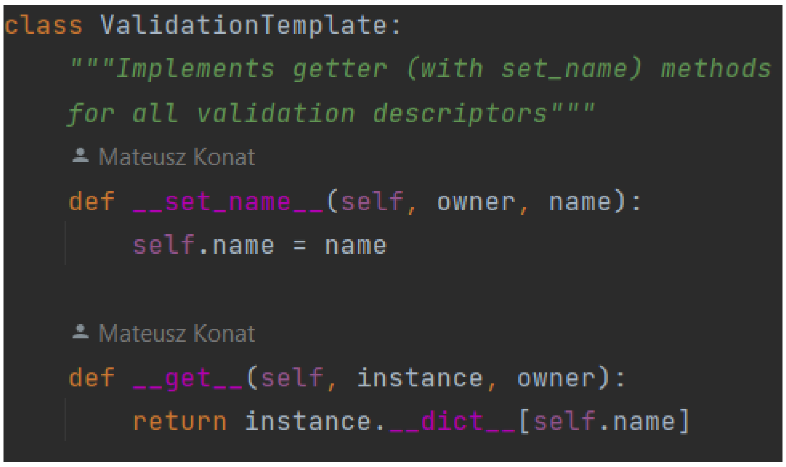
```python
class NewClass(base.AlterDB):
    """This class manages creating a class record."""
    __class_name = validation_descriptors.RepeatsInDB()

    Mateusz Konat
    def __init__(self, class_name: str):...

    # altering database: inserting classes
    Mateusz Konat
    def alter(self) -> None:
        # Retrieve query and data
        prompt = self.prompts["insert_class"]
        data = {
            "class_name": self.__class_name
        }
        # Manage connection with the db
        with self.connection_with_db as cur:
            cur.execute(prompt, data)
```

Figure 3. Example implementation of the ABC.

1.2 Descriptors

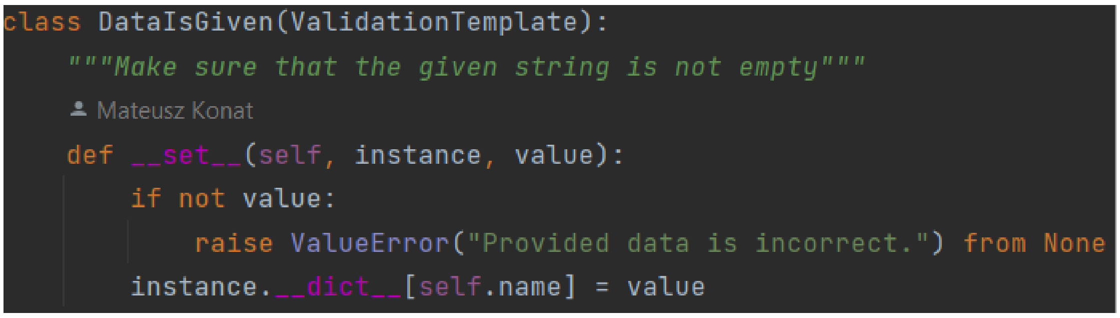
In Python, **descriptors** are objects that customize how attributes are accessed, stored, and deleted (Python documentation, n.d.; Real Python, 2019). In CAS Manager, they validate data before insertion into the database (Success Criterion 11).

Utilizing **descriptors** offers several benefits. It notably reduces code redundancy compared to using setters and getters method, or a property decorator, which would require separate implementations in each class. It also aids in debugging, minimizing errors as code is written once. This not only enhances the user experience for the client but also opens doors for more flexible and robust future developments in CAS Manager.



```python
class ValidationTemplate:
    """Implements getter (with set_name) methods
    for all validation descriptors"""
    Mateusz Konat
    def __set_name__(self, owner, name):
        self.name = name

    Mateusz Konat
    def __get__(self, instance, owner):
        return instance.__dict__[self.name]
```

Figure 4. Template for accessing attributes.



```python
class DataIsGiven(ValidationTemplate):
    """Make sure that the given string is not empty"""
    Mateusz Konat
    def __set__(self, instance, value):
        if not value:
            raise ValueError("Provided data is incorrect.") from None
        instance.__dict__[self.name] = value
```

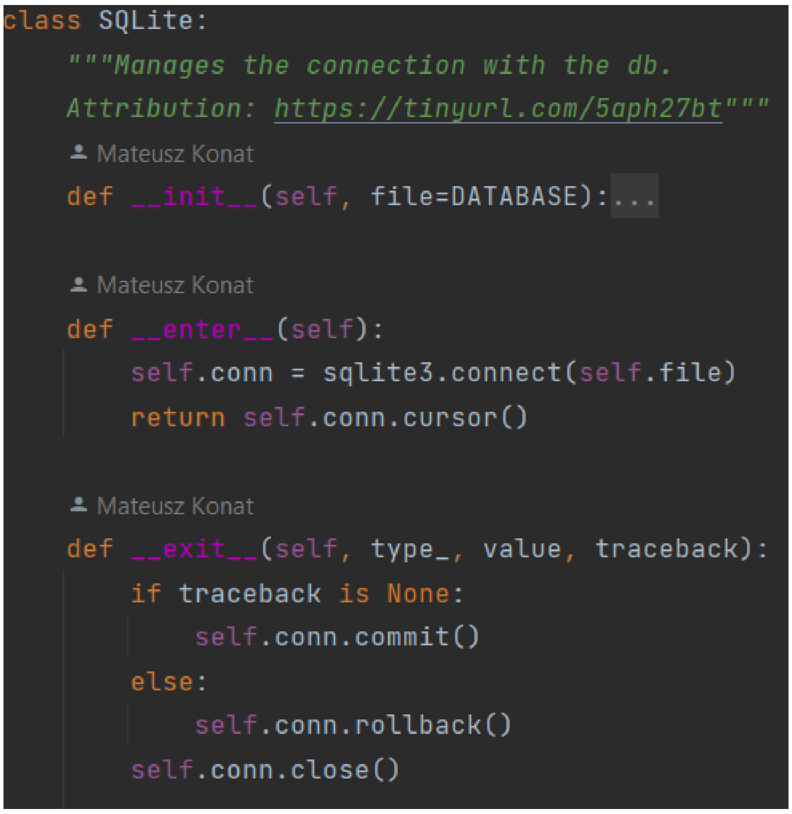
Figure 5. Ensuring that all data were provided.

2. Database

2.1 Context manager

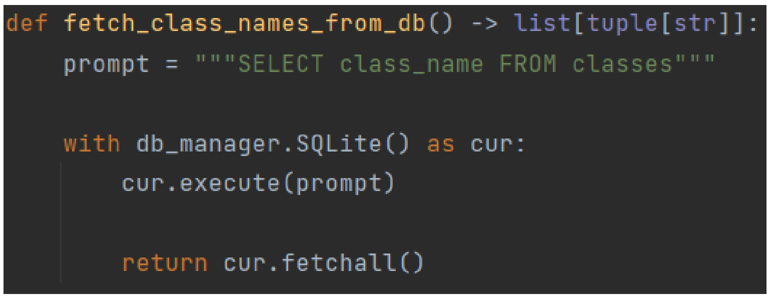
Context managers in Python offer a streamlined and error-free approach to managing resources like files and network connections (Python documentation, n.d.; Real Python, 2021). By leveraging the 'with' statement, Python creates a runtime context that automates the opening and closing of connections to resources.

In CAS Manager, a **custom context manager** is implemented to handle database connections seamlessly (Success Criteria 1-8). This **custom context manager** ensures the proper management of database connections, preventing common issues such as denied access due to too many open connections. This approach optimizes the performance and reliability of the application, contributing to a smoother user experience.



```
class SQLite:
    """Manages the connection with the db.
    Attribution: https://tinyurl.com/5aph27bt"""
    Mateusz Konat
    def __init__(self, file=DATABASE):...

    Mateusz Konat
    def __enter__(self):
        self.conn = sqlite3.connect(self.file)
        return self.conn.cursor()

    Mateusz Konat
    def __exit__(self, type_, value, traceback):
        if traceback is None:
            self.conn.commit()
        else:
            self.conn.rollback()
        self.conn.close()
```

Figure 6. Custom context manager implementation ('SQLite()').



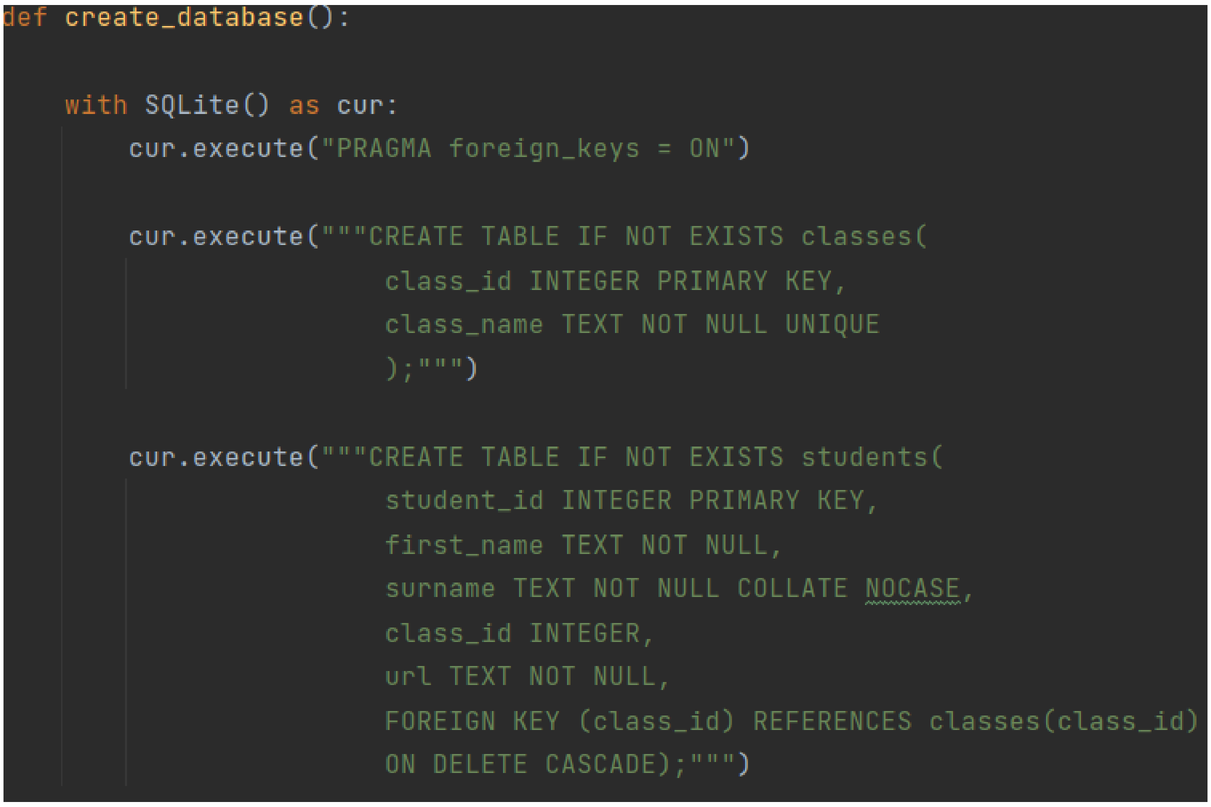
```
def fetch_class_names_from_db() -> list[tuple[str]]:
    prompt = """SELECT class_name FROM classes"""

    with db_manager.SQLite() as cur:
        cur.execute(prompt)

        return cur.fetchall()
```

Figure 7. Using context manager to retrieve class names.

2.2 Functional programming

Functional programming, a programming paradigm centered on evaluating functions for computation (Real Python, 2021; Wikipedia, 2019), forms the foundation of CAS Manager's database operations and transaction designs. The database schema, as defined in Criterion B.2, is constructed and managed using a functional programming approach, notably for tasks like retrieving students in ascending alphabetical order (Success Criteria 1-8).

Incorporating a functional approach offers several advantages within CAS Manager's development. By reducing the need for boilerplate code often associated with OOP solutions, the codebase becomes clearer and more concise. This clarity not only simplifies the understanding of the code but also streamlines the testing process, ensuring the reliability and efficiency of CAS Manager's functionalities.



```python
def create_database():

    with SQLite() as cur:
        cur.execute("PRAGMA foreign_keys = ON")

        cur.execute("""CREATE TABLE IF NOT EXISTS classes(
                        class_id INTEGER PRIMARY KEY,
                        class_name TEXT NOT NULL UNIQUE
                        );""")

        cur.execute("""CREATE TABLE IF NOT EXISTS students(
                        student_id INTEGER PRIMARY KEY,
                        first_name TEXT NOT NULL,
                        surname TEXT NOT NULL COLLATE NOCASE,
                        class_id INTEGER,
                        url TEXT NOT NULL,
                        FOREIGN KEY (class_id) REFERENCES classes(class_id)
                        ON DELETE CASCADE);""")
```

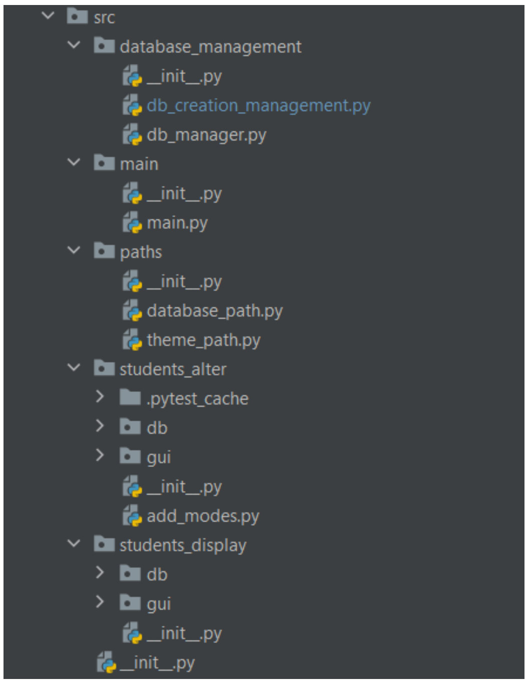
Figure 8. Implementing database schema via a built-in library.

3. Modular programming

Modularity plays a pivotal role in the design of CAS Manager, breaking down the entire codebase into smaller, manageable chunks (Success Criteria 1-11). This approach enhances both the maintainability and reusability of the software.

Maintenance: **modularity** reduces interdependencies between different parts of CAS Manager such that updates or changes can be made to individual modules with minimal impact on other parts.

Reusability: **modularity** promotes code reuse. Once a module is developed, it can be utilized across different sections of CAS Manager without the need for rewriting, which saves time and effort, and ensures consistency and reliability across the application.



```
src
  database_management
    __init__.py
    db_creation_management.py
    db_manager.py
  main
    __init__.py
    main.py
  paths
    __init__.py
    database_path.py
    theme_path.py
  students_alter
    .pytest_cache
    db
    gui
    __init__.py
    add_modes.py
  students_display
    db
    gui
    __init__.py
  __init__.py
```

Figure 9. Main body of CAS Manager.

3.1 Inheritance

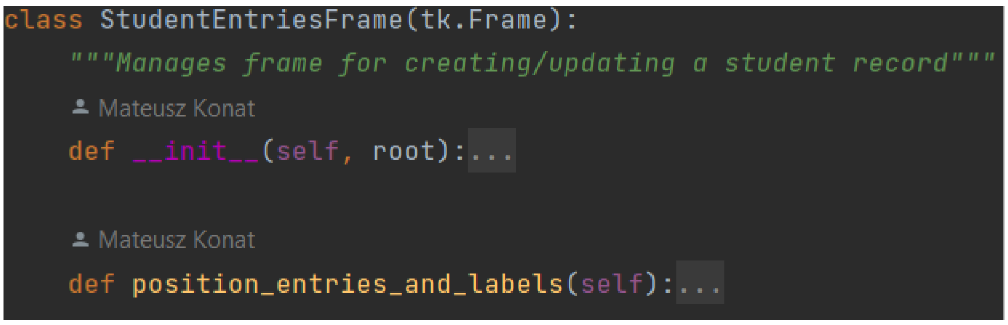
In OOP, **inheritance** establishes a class hierarchy, enabling child classes to inherit attributes and methods from a parent class (Python documentation, n.d.). CAS Manager effectively leverages **inheritance** to design modular GUI components (Criterion B.1.c) (). For example, CAS Manager segregates the creation of radio buttons for alteration modes (such as adding and updating classes and students) from the tree view of student display through inheritance.

This approach offers several advantages.

Firstly, it enhances testing by allowing individual testing of GUI elements, ensuring their functionality before integration.

Secondly, it promotes improved reusability as inherited classes facilitate the reuse of GUI elements across the application, thereby reducing redundant code.

Through these strategies, CAS Manager achieves a user interface that is flexible, scalable, and easy to maintain.

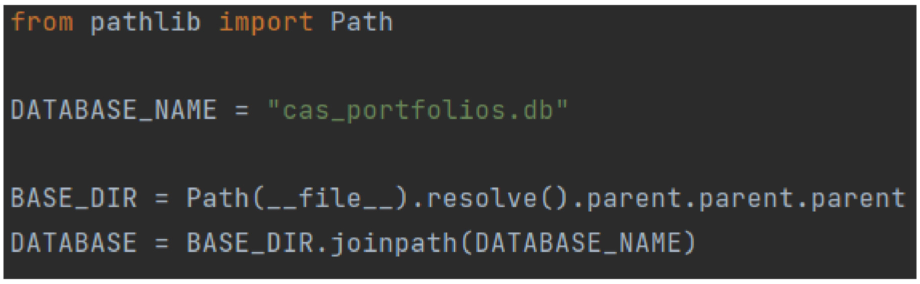


```python
class StudentEntriesFrame(tk.Frame):
    """Manages frame for creating/updating a student record"""
    Mateusz Konat
    def __init__(self, root):...

    Mateusz Konat
    def position_entries_and_labels(self):...
```

Figure 10. Example of inheritance – designing entries for student data.

3.2 File paths

File paths play a crucial role in modular programming as they provide access to shared resources like databases or theme colors across different parts of the program (Pitoru, 2012; Python documentation, n.d.). In CAS Manager, the built-in "pathlib" module is utilized for this purpose, offering an OOP to path management. This choice is preferred over the "os.path" module due to pathlib's simplicity and enhanced readability, making it easier to handle file paths in Python.



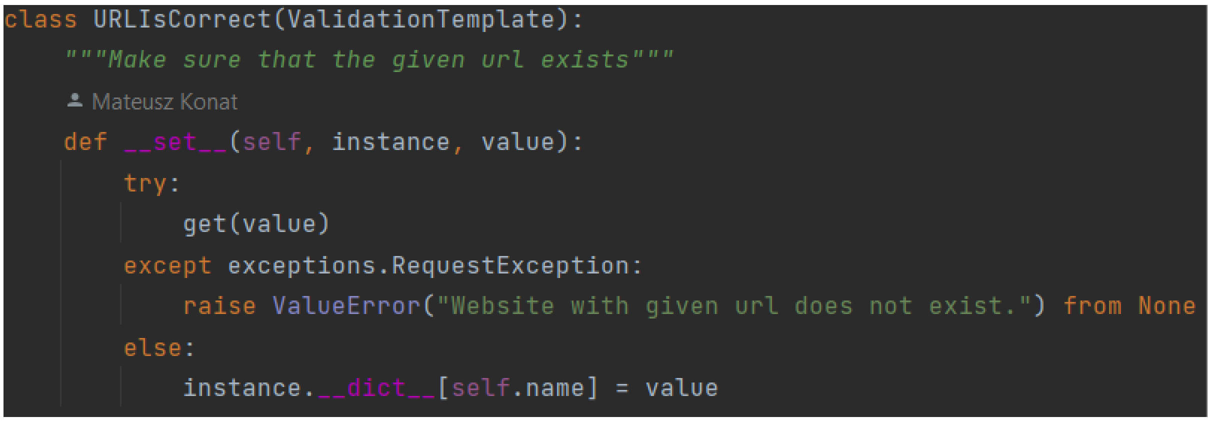
```python
from pathlib import Path

DATABASE_NAME = "cas_portfolios.db"

BASE_DIR = Path(__file__).resolve().parent.parent.parent
DATABASE = BASE_DIR.joinpath(DATABASE_NAME)
```

Figure 11. Path to the database.

4. Third-party software

4.1 Packages

a) Requests: CAS Manager employs the "requests" package to verify the existence of a portfolio (Success Criterion 11). This decision is based on the maturity of the "requests" package, which offers a secure and efficient method for handling such tasks.

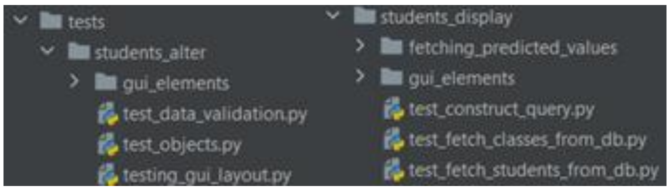


```python
class URLIsCorrect(ValidationTemplate):
    """Make sure that the given url exists"""
    Mateusz Konat
    def __set__(self, instance, value):
        try:
            get(value)
        except exceptions.RequestException:
            raise ValueError("Website with given url does not exist.") from None
        else:
            instance.__dict__[self.name] = value
```

Figure 12. Validating URL before insertion.

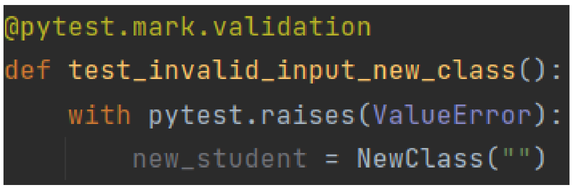
b) Pyinstaller: "pyinstaller" enables the creation of a single executable file, eliminating the need for the installation of the Python interpreter and other dependencies. This is especially important when delivering the program to clients who may not be familiar with setting up a programming environment.

c) **Pytest:** "pytest" is a testing package for Python. Its use it outlined in the Criterion B.4. This choice was made over the built-in unittest library because pytest simplifies test setup and teardown by providing a fixture mechanism, and supplies built-in support for test parameterization. These features enhance the testing capabilities of CAS Manager.



```
tests
  students_alter
    gui_elements
    test_data_validation.py
    test_objects.py
    testing_gui_layout.py
  students_display
    fetching_predicted_values
    gui_elements
    test_construct_query.py
    test_fetch_classes_from_db.py
    test_fetch_students_from_db.py
```

Figure 14. Dark GUI elements.



```python
@pytest.mark.validation
def test_invalid_input_new_class():
    with pytest.raises(ValueError):
        new_student = NewClass("")
```

Figure 15. White GUI elements.

4.2 Theme

In response to the client's request for a customized appearance in CAS Manager's GUI (Success Criterion 10), I employed a custom **theme** in Tkinter, the built-in Python module for GUI development. By integrating the "Forest theme for ttk," an open-source theme (rbende, 2021), CAS Manager now offers both dark and light versions of widgets, aligning perfectly with the client's desired visual style.

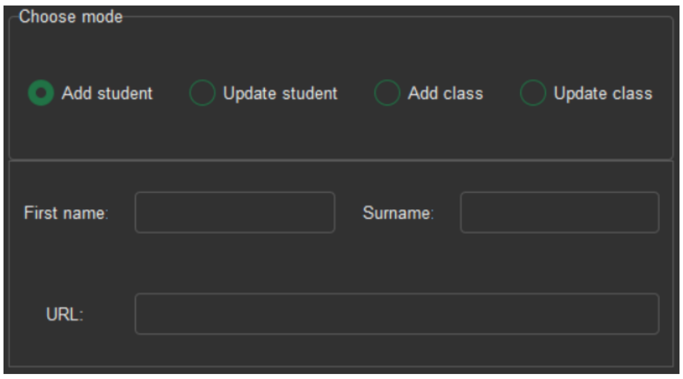


Figure 14. Dark GUI elements.

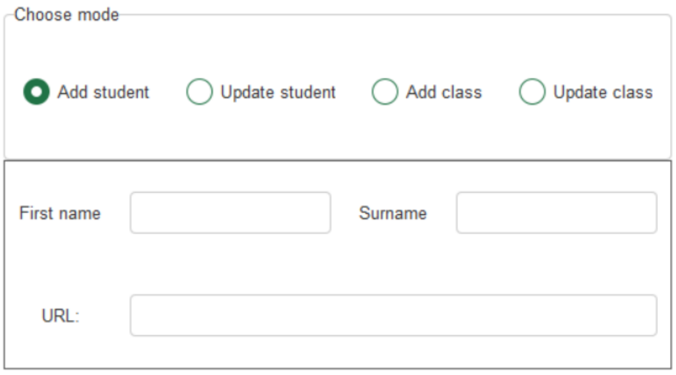


Figure 15. White GUI elements.

Word count: 960

List of references

Pitrou, A. (2012). *PEP 428 – The pathlib module – object-oriented filesystem paths | peps.python.org*. [online] Available at: https://peps.python.org/pep-0428/ [Accessed 8 Aug. 2023].

Python Documentation. (n.d.). *abc — Abstract Base Classes*. [online] Available at: https://docs.python.org/3/library/abc.html.

Python documentation. (n.d.). *Descriptor How To Guide*. [online] Available at: https://docs.python.org/3/howto/descriptor.html?highlight=descriptors.

Python documentation. (n.d.). *Built-in Types*. [online] Available at: https://docs.python.org/3/library/stdtypes.html#context-manager-types.

Python documentation. (n.d.). *9. Classes*. [online] Available at: https://docs.python.org/3/tutorial/classes.html#inheritance.

Python documentation (n.d.). *11.1. pathlib — Object-oriented filesystem paths — Python 3.7.0a2 documentation*. [online] Available at: https://python.readthedocs.io/en/latest/library/pathlib.html [Accessed 8 Aug. 2023].

rdbende (2023). *Forest theme for ttk*. [online] GitHub. Available at: https://github.com/rdbende/Forest-ttk-theme.

Real Python. (2021). *Context Managers and Python's with Statement – Real Python*. [online] realpython.com. Available at: https://realpython.com/python-with-statement/ [Accessed 8 Aug. 2023].

Real Python (2021). *Functional Programming in Python: When and How to Use It*. [online] realpython.com. Available at: https://realpython.com/python-functional-programming/ [Accessed 8 Aug. 2023].

Real Python. (2019). *Python Descriptors: An Introduction – Real Python*. [online] realpython.com. Available at: https://realpython.com/python-descriptors/ [Accessed 8 Aug. 2023].

Wikipedia. (2021). *Dependency inversion principle*. [online] Available at: https://en.wikipedia.org/wiki/Dependency_inversion_principle.

Wikipedia (2019). *Functional programming*. [online] Available at: https://en.wikipedia.org/wiki/Functional_programming.