

ZAMBIA UNIVERSITY COLLEGE OF TECNOLOGY

*ADVANCING KNOWLEDGE AND INNOVATION*

ASSIGNMENT 2

PROGRAMME: .................SOFTWARE ENGINEERING

COURSE: ........................SOFTWARE DESIGN

COURSRCODE: ............BSE 2210

LECTURER..................Mr JACOB CHIKWANDA

STUDENTS NAME …......Mr. JAMES CHISULO (2300745)

DATE GIVEN.........................21/08/2024

DUE DATE.............................27/082024

# SOFTWARE DEVELOPMENT PRINCIPLES

The Software development principles are essential guidelines that help in creating reliable, maintainable, and efficient software. Below is a given example of a Student Management System written in Python later being broken down into issues the code has as pertaining to the following principles

1. SOLID Principle
2. DRY Principles
3. KISS Principle
4. YAGNI Principle

Additionally, this code has user feedback

1. Administrator Feedback: "The system is functional, but whenever we request new features or changes, it takes a long time to implement. We've had to wait weeks for even simple updates."
2. IT Department Feedback: "The codebase is difficult to maintain and scale. We've had several issues with bugs that are hard to trace because the logic is scattered everywhere."
3. New Developer Feedback: "The code is hard to understand and lacks clear documentation. It’s challenging to figure out where to start when making changes."

The example code (Student Management System )

|  |
| --- |
| class Student: def **init**(self, id, name, age, major): self.id = id self.name = name self.age = age self.major = major  *def update\_student(self, name=None, age=None, major=None):*  *if name:*  *self.name = name*  *if age:*  *self.age = age*  *if major:*  *self.major = major*  *def display\_student(self):*  *print(f"ID: {self.id}, Name: {self.name}, Age: {self.age}, Major: {self.major}")*  class StudentDatabase: def **init**(self): self.students = []  *def add\_student(self, student):*  *self.students.append(student)*  *def remove\_student(self, student\_id):*  *for student in self.students:*  *if student.id == student\_id:*  *self.students.remove(student)*  *break*  *def display\_all\_students(self):*  *for student in self.students:*  *student.display\_student()*  class StudentManagementSystem: def **init**(self): self.database = StudentDatabase()  *def add\_new\_student(self, id, name, age, major):*  *student = Student(id, name, age, major)*  *self.database.add\_student(student)*  *def delete\_student(self, student\_id):*  *self.database.remove\_student(student\_id)*  *def update\_student\_info(self, student\_id, name=None, age=None, major=None):*  *for student in self.database.students:*  *if student.id == student\_id:*  *student.update\_student(name, age, major)*  *def show\_all\_students(self):*  *self.database.display\_all\_students()* Example Usage system = StudentManagementSystem() system.add\_new\_student(1, "John Doe", 20, "Computer Science") system.add\_new\_student(2, "Jane Smith", 22, "Mathematics") system.show\_all\_students() system.update\_student\_info(1, name="Johnathan Doe") system.show\_all\_students() system.delete\_student(2) system.show\_all\_students() |

The above code defies the mentioned principles and here is the are the noted issues and improvements

SOLID Principles

**Single Responsibility Principle (SRP)**:

Issue: The *Student* class handles both data storage and updating student information. This could be split into separate classes or methods to adhere more strictly to SRP.

Improvement: Create a separate *StudentUpdater* class to handle updates.

**Open/Closed Principle (OCP):**

Issue: The *Student* class is not easily extendable without modification. For example, adding new attributes would require changes to the class.

Improvement: Use inheritance or composition to extend functionality without modifying existing code.

**Liskov Substitution Principle (LSP):**

Issue: Not directly applicable here, but if you were to introduce subclasses, ensure they can replace the *Student* class without altering the program's behavior.

Improvement: Ensure all subclasses adhere to the expected behavior of the *Student*  class.

**Interface Segregation Principle (ISP)**:

Issue: Not applicable as there are no interfaces. However, if interfaces were used, they should be specific to the clients' needs.

Improvement: Introduce interfaces if the system grows more complex.

**Dependency Inversion Principle (DIP):**

Issue: The *StudentManagementSystem* directly depends on the *StudentDatabase*  class. This could be improved by depending on an abstraction.

Improvement: Introduce an interface for *StudentDatabase* and have *StudentManagementSystem* depend on that interface.

**DRY (Don't Repeat Yourself)**

Issue: The code is generally good at avoiding repetition, but if more methods are added to Student, consider abstracting common functionality.

Improvement: Ensure any new functionality is abstracted into reusable components.

**KISS (Keep It Simple, Stupid)**

Issue: The code is simple and straightforward, but as the system grows, ensuring it remains simple and avoid over-engineering.

Improvement: Regularly refactor to keep the codebase clean and simple.

**YAGNI (You Ain't Gonna Need It)**

Issue: The code currently adheres to YAGNI but be cautious of adding features that are not immediately necessary.

Improvement: Continually evaluate the necessity of new features before implementation.

Below is a refactored code according to the user feedback and the principles mentioned earlier. This code intends to improve the student management system by applying the principles as shown above. The StudentUpdater class now handles updates, adhering to SRP. It has IStudentDatabase interface to decouple StudentManagementSystem from the concrete StudentDatabase implementation and added a simple menu to allow users to interact with the system more effectively.

This refactored code should make the system easier to maintain, extend, and understand, addressing the user feedback

Refactored code

|  |
| --- |
| from abc import ABC, abstractmethod    class Student:  def \_\_init\_\_(self, id, name, age, major):  """  Initialize a Student object.    Args:  id (int): Student ID.  name (str): Student name.  age (int): Student age.  major (str): Student major.  """  self.id = id  self.name = name  self.age = age  self.major = major    def display\_student(self):  """  Display student information.    Prints the student's ID, name, age, and major.  """  print(f"ID: {self.id}, Name: {self.name}, Age: {self.age}, Major: {self.major}")    class StudentUpdater:  @staticmethod  def update\_student(student, name=None, age=None, major=None):  """  Update student information.    Args:  student (Student): The student object to update.  name (str, optional): New student name. Defaults to None.  age (int, optional): New student age. Defaults to None.  major (str, optional): New student major. Defaults to None.  """  if name:  student.name = name  if age:  student.age = age  if major:  student.major = major    class IStudentDatabase(ABC):  @abstractmethod  def add\_student(self, student):  """  Abstract method to add a student to the database.    Args:  student (Student): The student object to add.  """  pass    @abstractmethod  def remove\_student(self, student\_id):  """  Abstract method to remove a student from the database.    Args:  student\_id (int): ID of the student to remove.  """  pass    @abstractmethod  def display\_all\_students(self):  """  Abstract method to display all students in the database.  """  pass    class StudentDatabase(IStudentDatabase):  def \_\_init\_\_(self):  self.students = []    def add\_student(self, student):  """  Add a student to the database.    Args:  student (Student): The student object to add.  """  self.students.append(student)    def remove\_student(self, student\_id):  """  Remove a student from the database.    Args:  student\_id (int): ID of the student to remove.  """  for student in self.students:  if student.id == student\_id:  self.students.remove(student)  break    def display\_all\_students(self):  """  Display information for all students in the database.  """  for student in self.students:  student.display\_student()    # Example usage:  if \_\_name\_\_ == "\_\_main\_\_":  db = StudentDatabase()  student1 = Student(1, "Alice", 20, "Computer Science")  student2 = Student(2, "Bob", 22, "Mathematics")  db.add\_student(student1)  db.add\_student(student2)  db.display\_all\_students() |

# CONCLUSION

Software development principles serve as guiding rules and fundamental concepts that enhance the quality, efficiency, and reliability of software projects. From the example of a code from this research, we can note several issues that made the student management system difficult in maintainability and scalability, as mentioned by the user feedback. With the application of the software development principles which help guide to note issues and resolve them thereby, improving the quality and efficient system as provided in the refactored code section.

# REFFRENCE

Cris Norman P. Olipas(2020). *The design and development of student information and violation management system (SIVMS) for a higher educational institution.* Volume - 6, Issue - 8, Aug. [https://files.eric.ed.gov/fulltext/ED622596.pdf [accessed](https://files.eric.ed.gov/fulltext/ED622596.pdf  [accessed): 26/08/2024]

Innovera() *Student Data Management 101: Policies, Procedures & Practices.*<https://innovaresip.com/resources/blog/student-data-management-policies-procedures-practices/> [accessed: 26/08/2024]