# ****W5D3 -- Enhancement Phase: Code Refactoring****

JTC Program: Tech Pathways Cohort: S25 Lesson Plan: Code Refactoring Type: Lesson Plan Week / Day: W5D3 Version Date: 05/27/2025

## ****Focus Concepts****

* Understanding what code refactoring is and why it's essential for maintainable software
* Identifying code quality issues and areas for improvement in existing code
* Applying fundamental refactoring techniques to improve code organization
* Learning to write clean, readable code with meaningful variable and function names
* Implementing the DRY (Don't Repeat Yourself) principle to eliminate code duplication
* Organizing code using functions and classes for better structure and reusability

## ****Learning Objectives****

By the end of this session, fellows will be able to:

* Define code refactoring and explain its importance in software development
* Identify common code quality issues such as poor naming, duplication, and lack of organization
* Extract functions from repetitive code blocks to improve maintainability
* Choose descriptive variable and function names that make code self-documenting
* Apply the DRY principle to eliminate code duplication
* Organize related data and functions using basic class structures

## ****Out-of-Scope Objectives****

* Advanced design patterns (Observer, Factory, Strategy, etc.)
* Complex inheritance hierarchies and polymorphism
* Performance optimization techniques
* Database refactoring and schema changes
* Legacy code migration strategies
* Advanced testing frameworks and test-driven refactoring

## ****Required Competencies****

* Basic Python syntax and control structures (variables, functions, loops, conditionals)
* Understanding of basic data types (strings, integers, lists, dictionaries)
* Familiarity with writing and calling functions with parameters
* Experience with basic file operations and code organization
* Comfort with debugging simple Python programs

## ****Technical Requirements****

* Python 3.x installed
* Code editor or IDE (VS Code, PyCharm, or similar)
* Ability to run Python scripts from command line or IDE
* Access to provided Python example files
* Basic understanding of file management and directory structure

## ****Prerequisites****

* Completion of Python fundamentals (variables, functions, control structures)
* Basic understanding of programming logic and problem-solving
* Experience writing simple Python programs (at least 20-50 lines)
* Familiarity with reading and understanding existing code
* Understanding of basic programming concepts like variables, functions, and loops

## ****Assigned Reading & Pre-Class Learning****

Estimated Time: 25 minutes

Resources:

* [Clean Code Basics - Writing Better Code](https://www.freecodecamp.org/news/clean-coding-for-beginners/) - Introduction to clean code principles - 12 minutes
* [Python Code Style Guide (PEP 8) - Naming Conventions](https://pep8.org/" \l "naming-conventions) - Learn Python naming best practices - 8 minutes
* [What is Code Refactoring?](https://refactoring.guru/refactoring/what-is-refactoring) - Understanding refactoring fundamentals - 5 minutes

## ****Before-Class Mini Quiz Questions (5 questions)****

1. What is code refactoring?
   * A) Writing new code from scratch
   * \*B) Improving existing code without changing its functionality
   * C) Fixing bugs in broken code
   * D) Adding new features to a program
2. Which of these is an example of poor variable naming?
   * A) student\_name
   * B) total\_score
   * \*C) x1
   * D) is\_valid
3. What does the DRY principle stand for?
   * A) Debug Regularly, Yearly
   * \*B) Don't Repeat Yourself
   * C) Deploy, Run, Yield
   * D) Data Requires Yelling
4. Why should you avoid code duplication?
   * A) It makes programs run slower
   * \*B) Changes need to be made in multiple places, increasing the chance of errors
   * C) It uses too much memory
   * D) It's against Python syntax rules
5. What is the main benefit of extracting repeated code into a function?
   * A) It makes the program run faster
   * B) It reduces the file size
   * \*C) The code can be reused and maintained more easily
   * D) It automatically fixes bugs

## ****Key Terms****

* **Refactoring**: The process of improving code structure and readability without changing its functionality
* **Code Quality**: A measure of how readable, maintainable, and efficient code is
* **DRY Principle**: "Don't Repeat Yourself" - avoid duplicating code by creating reusable functions
* **Function Extraction**: Taking repeated code and creating a separate function for it
* **Variable Naming**: Choosing descriptive names that clearly indicate what a variable contains
* **Code Organization**: Structuring code in a logical, easy-to-follow manner
* **Maintainability**: How easy it is to modify, debug, and extend code over time
* **Readability**: How easily other programmers (or yourself later) can understand the code
* **Code Duplication**: When the same or similar code appears in multiple places
* **Function Decomposition**: Breaking down large, complex functions into smaller, focused ones
* **Modularity**: Organizing code into separate, independent components
* **Class**: A blueprint for creating objects that group related data and functions together
* **Method**: A function that belongs to a class
* **Encapsulation**: Keeping related data and functions together in a class
* **Self-Documenting Code**: Code that is so clear it explains itself without comments
* **Technical Debt**: The cost of choosing quick solutions over clean code
* **Code Smell**: Signs that code needs refactoring (long functions, repeated code, poor names)
* **Separation of Concerns**: Each function or class should have one clear responsibility

## ****Lesson Schedule & Detailed Script****

### ****6:30 PM -- 6:45 PM: Interactive Check-In****

**Instructor Script:** "Welcome to Week 5, Day 3! We're now in the Enhancement phase of our Python journey. You've learned the fundamentals, and now we're going to focus on writing better, cleaner code. Today's topic is code refactoring - the art of improving existing code to make it more readable, maintainable, and professional. Think of it like editing an essay: the ideas are there, but we're going to make them clearer and better organized. By the end of today, you'll be able to transform messy, hard-to-read code into clean, professional-quality code."

**Admin Tasks:**

* Take attendance
* Ensure everyone can run Python and access the example files
* Check for any issues with previous assignments

**Prompting Questions:**

* "Have you ever looked at code you wrote a week ago and thought 'What was I thinking?' That's exactly why we refactor!"
* "What makes some code easier to read than others?"

**Poll Questions:**

* "On a scale of 1-5, how confident do you feel about reading other people's code?"
* "What's your biggest challenge when writing code: getting it to work, or making it look good?"

### ****6:45 PM -- 7:05 PM: Session 1 -- What is Code Refactoring and Why Does it Matter?****

**Objective:** Understand the fundamentals of code refactoring and its importance in software development.

**Instructor Script:** "Let's start by understanding what refactoring really means and why it's one of the most important skills for any programmer to develop."

#### ****Definition and Importance of Code Refactoring:****

Code refactoring is the process of improving the internal structure of code without changing its external behavior. Think of it like renovating a house - you're making it better organized and more beautiful, but it still serves the same purpose.

**Why Refactor Code?**

1. **Readability**: Makes code easier to understand for yourself and others
2. **Maintainability**: Easier to fix bugs and add new features
3. **Reusability**: Well-organized code can be reused in other projects
4. **Professional Quality**: Clean code is a mark of a skilled programmer
5. **Team Collaboration**: Others can easily work with your code

# BEFORE REFACTORING - Hard to understand

def calc(a, b, c):

x = a + b + c

y = x / 3

if y >= 90:

return "A"

elif y >= 80:

return "B"

else:

return "C"

# AFTER REFACTORING - Clear and self-explanatory

def calculate\_letter\_grade(math\_score, english\_score, science\_score):

total\_points = math\_score + english\_score + science\_score

average\_score = total\_points / 3

if average\_score >= 90:

return "A"

elif average\_score >= 80:

return "B"

else:

return "C"

#### ****Common Code Quality Issues:****

**Instructor Script:** "Let's look at the most common problems that make code hard to work with:"

1. **Poor Variable Names**

# Bad

x = 25

y = "John"

z = x \* 0.1

# Good

student\_age = 25

student\_name = "John"

age\_bonus\_points = student\_age \* 0.1

1. **Code Duplication**

# Bad - Same logic repeated

print(f"Student 1: {name1}, Grade: {grade1}")

if grade1 >= 90:

print("Excellent!")

print(f"Student 2: {name2}, Grade: {grade2}")

if grade2 >= 90:

print("Excellent!")

# Good - Logic in a reusable function

def display\_student\_info(name, grade):

print(f"Student: {name}, Grade: {grade}")

if grade >= 90:

print("Excellent!")

1. **Functions That Do Too Much**

# Bad - One function doing everything

def process\_student():

# Get input (20 lines)

# Validate data (15 lines)

# Calculate grades (25 lines)

# Display results (10 lines)

# Save to file (12 lines)

# Good - Separate functions for each task

def get\_student\_input():

# Just handle input

def validate\_student\_data():

# Just validate

def calculate\_grades():

# Just calculate

#### ****The Cost of Poor Code Quality:****

**Instructor Script:** "Bad code isn't just ugly - it has real costs:"

* **Time Cost**: Developers spend 60-80% of their time reading code, not writing it
* **Bug Risk**: Messy code leads to more bugs and harder debugging
* **Team Problems**: Other developers can't easily work with poorly written code
* **Career Impact**: Clean code is a sign of professionalism and skill

#### ****When to Refactor:****

**Good Times to Refactor:**

* Before adding new features to existing code
* When you find bugs in existing code
* During code reviews
* When code becomes hard to understand
* As part of regular maintenance

**When NOT to Refactor:**

* When you're close to a deadline (unless it's blocking progress)
* When the code works well and won't be changed
* When you don't have good tests to ensure you don't break functionality

### ****7:05 PM -- 7:30 PM: Session 2 -- Basic Refactoring Techniques****

**Objective:** Learn and practice fundamental refactoring techniques including function extraction and improved naming.

**Instructor Script:** "Now let's get hands-on with the most important refactoring techniques. We'll start with simple examples and build up your skills."

#### ****Technique 1: Extracting Functions from Repetitive Code****

**Instructor Script:** "The most common refactoring technique is extracting functions. When you see the same code repeated, that's a signal to create a function."

# BEFORE: Repetitive code that's hard to maintain

print("=== Student Report Card System - BEFORE ===")

# Process Student 1

student1\_name = "Alice"

math\_grade = 85

english\_grade = 92

science\_grade = 78

total = math\_grade + english\_grade + science\_grade

average = total / 3

print(f"Student: {student1\_name}")

print(f"Math: {math\_grade}, English: {english\_grade}, Science: {science\_grade}")

print(f"Average: {average:.1f}")

if average >= 90:

letter\_grade = "A"

elif average >= 80:

letter\_grade = "B"

elif average >= 70:

letter\_grade = "C"

else:

letter\_grade = "F"

print(f"Letter Grade: {letter\_grade}")

print("-" \* 30)

# Process Student 2 (same code repeated!)

student2\_name = "Bob"

math\_grade = 78

english\_grade = 85

science\_grade = 90

total = math\_grade + english\_grade + science\_grade

average = total / 3

print(f"Student: {student2\_name}")

print(f"Math: {math\_grade}, English: {english\_grade}, Science: {science\_grade}")

print(f"Average: {average:.1f}")

if average >= 90:

letter\_grade = "A"

elif average >= 80:

letter\_grade = "B"

elif average >= 70:

letter\_grade = "C"

else:

letter\_grade = "F"

print(f"Letter Grade: {letter\_grade}")

# AFTER: Clean functions that eliminate repetition

print("=== Student Report Card System - AFTER ===")

def calculate\_average(math\_score, english\_score, science\_score):

"""Calculate the average of three test scores."""

total\_points = math\_score + english\_score + science\_score

return total\_points / 3

def get\_letter\_grade(average\_score):

"""Convert a numerical average to a letter grade."""

if average\_score >= 90:

return "A"

elif average\_score >= 80:

return "B"

elif average\_score >= 70:

return "C"

else:

return "F"

def display\_student\_report(name, math\_score, english\_score, science\_score):

"""Display a complete report card for one student."""

print(f"Student: {name}")

print(f"Math: {math\_score}, English: {english\_score}, Science: {science\_score}")

average = calculate\_average(math\_score, english\_score, science\_score)

print(f"Average: {average:.1f}")

letter\_grade = get\_letter\_grade(average)

print(f"Letter Grade: {letter\_grade}")

print("-" \* 30)

# Now processing students is simple and consistent

display\_student\_report("Alice", 85, 92, 78)

display\_student\_report("Bob", 78, 85, 90)

display\_student\_report("Carol", 94, 88, 91) # Easy to add more!

**Benefits of Function Extraction:**

* **DRY Principle**: Don't Repeat Yourself
* **Single Responsibility**: Each function has one clear job
* **Easier Testing**: Can test each function independently
* **Easier Maintenance**: Changes only need to be made in one place

#### ****Technique 2: Improving Variable and Function Names****

**Instructor Script:** "Good names make code self-documenting. Let's practice choosing names that clearly communicate intent."

# BEFORE: Confusing names

def process\_data(x, y, z):

result = []

for item in x:

if item > y:

temp = item \* z

result.append(temp)

return result

data = [85, 92, 78, 96, 88]

threshold = 80

multiplier = 1.1

final = process\_data(data, threshold, multiplier)

# AFTER: Clear, descriptive names

def apply\_bonus\_to\_high\_scores(test\_scores, minimum\_score, bonus\_multiplier):

"""Apply a bonus multiplier to scores above the minimum threshold."""

scores\_with\_bonus = []

for score in test\_scores:

if score > minimum\_score:

boosted\_score = score \* bonus\_multiplier

scores\_with\_bonus.append(boosted\_score)

return scores\_with\_bonus

student\_test\_scores = [85, 92, 78, 96, 88]

passing\_threshold = 80

excellence\_bonus = 1.1

final\_scores = apply\_bonus\_to\_high\_scores(student\_test\_scores, passing\_threshold, excellence\_bonus)

**Naming Best Practices:**

* Use full words, not abbreviations (score not sc)
* Be specific (student\_name not just name)
* Use verbs for functions (calculate\_total not total)
* Use nouns for variables (total\_score not calculate)
* Use boolean names that read like questions (is\_valid, has\_passed)

#### ****Technique 3: Breaking Down Large Functions****

# BEFORE: One large function doing everything

def student\_management\_system():

# Get student info (10 lines of input code)

print("Enter student information:")

name = input("Name: ")

math = int(input("Math score: "))

english = int(input("English score: "))

science = int(input("Science score: "))

# Validate scores (8 lines of validation)

if math < 0 or math > 100:

print("Invalid math score")

return

if english < 0 or english > 100:

print("Invalid english score")

return

if science < 0 or science > 100:

print("Invalid science score")

return

# Calculate grades (5 lines of calculation)

total = math + english + science

average = total / 3

if average >= 90:

grade = "A"

elif average >= 80:

grade = "B"

else:

grade = "C"

# Display results (6 lines of output)

print(f"Student: {name}")

print(f"Scores: Math {math}, English {english}, Science {science}")

print(f"Average: {average:.1f}")

print(f"Grade: {grade}")

# Save to file (8 lines of file operations)

with open("grades.txt", "a") as file:

file.write(f"{name},{math},{english},{science},{average:.1f},{grade}\n")

print("Data saved successfully!")

# AFTER: Organized into focused functions

def get\_student\_information():

"""Get student name and test scores from user input."""

print("Enter student information:")

name = input("Name: ")

math\_score = int(input("Math score: "))

english\_score = int(input("English score: "))

science\_score = int(input("Science score: "))

return name, math\_score, english\_score, science\_score

def validate\_score(score, subject\_name):

"""Check if a score is within the valid range (0-100)."""

if score < 0 or score > 100:

print(f"Invalid {subject\_name} score: {score}")

return False

return True

def validate\_all\_scores(math\_score, english\_score, science\_score):

"""Validate all three test scores."""

return (validate\_score(math\_score, "math") and

validate\_score(english\_score, "english") and

validate\_score(science\_score, "science"))

def calculate\_grade\_info(math\_score, english\_score, science\_score):

"""Calculate average score and letter grade."""

total\_points = math\_score + english\_score + science\_score

average\_score = total\_points / 3

if average\_score >= 90:

letter\_grade = "A"

elif average\_score >= 80:

letter\_grade = "B"

else:

letter\_grade = "C"

return average\_score, letter\_grade

def display\_student\_results(name, math\_score, english\_score, science\_score, average\_score, letter\_grade):

"""Display the student's complete grade report."""

print(f"Student: {name}")

print(f"Scores: Math {math\_score}, English {english\_score}, Science {science\_score}")

print(f"Average: {average\_score:.1f}")

print(f"Grade: {letter\_grade}")

def save\_student\_data(name, math\_score, english\_score, science\_score, average\_score, letter\_grade):

"""Save student data to a file."""

with open("grades.txt", "a") as file:

file.write(f"{name},{math\_score},{english\_score},{science\_score},{average\_score:.1f},{letter\_grade}\n")

print("Data saved successfully!")

def student\_management\_system():

"""Main function that coordinates the student grading process."""

# Get information

name, math\_score, english\_score, science\_score = get\_student\_information()

# Validate scores

if not validate\_all\_scores(math\_score, english\_score, science\_score):

return # Exit if validation fails

# Calculate grades

average\_score, letter\_grade = calculate\_grade\_info(math\_score, english\_score, science\_score)

# Display results

display\_student\_results(name, math\_score, english\_score, science\_score, average\_score, letter\_grade)

# Save data

save\_student\_data(name, math\_score, english\_score, science\_score, average\_score, letter\_grade)

**Benefits of Breaking Down Functions:**

* **Single Responsibility**: Each function has one clear purpose
* **Easier Testing**: Can test each function independently
* **Better Error Handling**: Can isolate and handle errors more precisely
* **Improved Readability**: The main function reads like a step-by-step plan

### ****7:30 PM -- 7:50 PM: Capstone Work Session****

**Activity:** Work on capstone project, applying refactoring techniques to improve existing code.

### ****7:50 PM -- 8:00 PM: Break****

10-minute break

### ****8:00 PM -- 8:35 PM: Session 3 -- Organization with Classes and Advanced Techniques****

**Objective:** Learn to organize code using classes and apply advanced refactoring concepts.

**Instructor Script:** "Now that you understand basic refactoring, let's explore how to organize code using classes. Classes help us group related data and functions together, creating more sophisticated and maintainable programs."

#### ****Introduction to Classes for Code Organization****

**Instructor Script:** "Think of a class as a blueprint or template. Just like a blueprint for a house defines what every house built from it will have, a class defines what objects created from it will contain."

# BEFORE: Functions and data scattered everywhere

def display\_student\_info(name, age, grade, subjects):

print(f"Student: {name}, Age: {age}, Grade: {grade}")

for subject, score in subjects.items():

print(f" {subject}: {score}")

def calculate\_student\_average(subjects):

if not subjects:

return 0

return sum(subjects.values()) / len(subjects)

def add\_subject\_score(subjects, subject, score):

subjects[subject] = score

# Student data stored separately

student1\_name = "Alice"

student1\_age = 16

student1\_grade = 10

student1\_subjects = {"Math": 85, "English": 92}

student2\_name = "Bob"

student2\_age = 17

student2\_grade = 11

student2\_subjects = {"Math": 78, "Science": 88}

# Functions called with many parameters

display\_student\_info(student1\_name, student1\_age, student1\_grade, student1\_subjects)

avg1 = calculate\_student\_average(student1\_subjects)

# AFTER: Organized with a Student class

class Student:

"""A class to represent a student and manage their academic information."""

def \_\_init\_\_(self, name, age, grade):

"""Create a new student with basic information."""

self.name = name

self.age = age

self.grade = grade

self.subjects = {} # Dictionary to store subject scores

def add\_subject\_score(self, subject, score):

"""Add or update a score for a specific subject."""

if 0 <= score <= 100:

self.subjects[subject] = score

print(f"Added {subject} score of {score} for {self.name}")

else:

print(f"Invalid score: {score}. Score must be between 0 and 100.")

def calculate\_average(self):

"""Calculate the average score across all subjects."""

if not self.subjects:

return 0

return sum(self.subjects.values()) / len(self.subjects)

def get\_letter\_grade(self):

"""Get letter grade based on average score."""

average = self.calculate\_average()

if average >= 90:

return "A"

elif average >= 80:

return "B"

elif average >= 70:

return "C"

elif average >= 60:

return "D"

else:

return "F"

def display\_report\_card(self):

"""Display complete student information and grades."""

print(f"\n--- Report Card for {self.name} ---")

print(f"Age: {self.age}, Grade: {self.grade}")

print("Subject Scores:")

if not self.subjects:

print(" No scores recorded yet.")

else:

for subject, score in self.subjects.items():

print(f" {subject}: {score}")

average = self.calculate\_average()

letter\_grade = self.get\_letter\_grade()

print(f"Average: {average:.1f}")

print(f"Letter Grade: {letter\_grade}")

# Using the Student class

alice = Student("Alice", 16, 10)

alice.add\_subject\_score("Math", 85)

alice.add\_subject\_score("English", 92)

alice.add\_subject\_score("Science", 78)

alice.display\_report\_card()

bob = Student("Bob", 17, 11)

bob.add\_subject\_score("Math", 78)

bob.add\_subject\_score("Science", 88)

bob.display\_report\_card()

**Benefits of Using Classes:**

* **Encapsulation**: Related data and functions are grouped together
* **Data Protection**: Each student manages their own data
* **Code Reuse**: Easy to create multiple students
* **Maintainability**: Changes to student logic only need to be made in one place

#### ****Advanced Class Example: Classroom Management****

class Classroom:

"""A class to manage multiple students in a classroom."""

def \_\_init\_\_(self, class\_name, teacher\_name):

"""Create a new classroom."""

self.class\_name = class\_name

self.teacher\_name = teacher\_name

self.students = [] # List to store Student objects

def add\_student(self, student):

"""Add a student to the classroom."""

self.students.append(student)

print(f"{student.name} has been added to {self.class\_name}")

def remove\_student(self, student\_name):

"""Remove a student from the classroom by name."""

for student in self.students:

if student.name == student\_name:

self.students.remove(student)

print(f"{student\_name} has been removed from {self.class\_name}")

return

print(f"Student {student\_name} not found in {self.class\_name}")

def find\_student(self, student\_name):

"""Find a student by name."""

for student in self.students:

if student.name == student\_name:

return student

return None

def calculate\_class\_average(self):

"""Calculate the average score for the entire class."""

if not self.students:

return 0

total\_average = 0

students\_with\_scores = 0

for student in self.students:

student\_avg = student.calculate\_average()

if student\_avg > 0: # Only count students with recorded scores

total\_average += student\_avg

students\_with\_scores += 1

if students\_with\_scores == 0:

return 0

return total\_average / students\_with\_scores

def display\_class\_roster(self):

"""Display information about all students in the class."""

print(f"\n=== {self.class\_name} - Teacher: {self.teacher\_name} ===")

print(f"Total Students: {len(self.students)}")

if not self.students:

print("No students enrolled yet.")

return

print("\nStudent Roster:")

for i, student in enumerate(self.students, 1):

avg = student.calculate\_average()

grade = student.get\_letter\_grade()

print(f"{i}. {student.name} (Age {student.age}) - Average: {avg:.1f} ({grade})")

class\_avg = self.calculate\_class\_average()

print(f"\nClass Average: {class\_avg:.1f}")

def get\_top\_students(self, count=3):

"""Get the top performing students in the class."""

# Sort students by their average score (highest first)

students\_with\_scores = [s for s in self.students if s.calculate\_average() > 0]

sorted\_students = sorted(students\_with\_scores,

key=lambda s: s.calculate\_average(),

reverse=True)

return sorted\_students[:count]

# Demonstration of the complete system

def main():

"""Demonstrate the classroom management system."""

# Create a classroom

math\_class = Classroom("Advanced Mathematics", "Ms. Johnson")

# Create and add students

alice = Student("Alice", 16, 10)

alice.add\_subject\_score("Math", 95)

alice.add\_subject\_score("Physics", 92)

alice.add\_subject\_score("Chemistry", 88)

bob = Student("Bob", 17, 11)

bob.add\_subject\_score("Math", 78)

bob.add\_subject\_score("Physics", 85)

bob.add\_subject\_score("Chemistry", 82)

carol = Student("Carol", 16, 10)

carol.add\_subject\_score("Math", 92)

carol.add\_subject\_score("Physics", 89)

carol.add\_subject\_score("Chemistry", 94)

# Add students to classroom

math\_class.add\_student(alice)

math\_class.add\_student(bob)

math\_class.add\_student(carol)

# Display classroom information

math\_class.display\_class\_roster()

# Show top students

print(f"\nTop 2 Students:")

top\_students = math\_class.get\_top\_students(2)

for i, student in enumerate(top\_students, 1):

print(f"{i}. {student.name}: {student.calculate\_average():.1f}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

#### ****Key Refactoring Principles Demonstrated:****

1. **Single Responsibility Principle**: Each class and method has one clear purpose
2. **DRY (Don't Repeat Yourself)**: No duplicated code
3. **Encapsulation**: Related data and methods are grouped together
4. **Clear Naming**: All variables and functions have descriptive names
5. **Modular Design**: Code is organized into logical, reusable components

### ****8:35 PM -- 9:05 PM: Session 4 -- Hands-On Refactoring Practice****

**Objective:** Apply refactoring techniques to real code examples through guided practice.

**Instructor Script:** "Now it's time to put your refactoring skills to work! We'll practice with some messy code examples that need improvement. I'll guide you through the process step by step."

#### ****Practice Exercise 1: Refactoring a Simple Calculator****

**Instructor Script:** "Here's a calculator program that works, but it's poorly organized. Let's refactor it together."

# MESSY CODE TO REFACTOR

print("Calculator Program")

while True:

print("1. Add")

print("2. Subtract")

print("3. Multiply")

print("4. Divide")

print("5. Exit")

c = input("Choose operation: ")

if c == "1":

n1 = float(input("First number: "))

n2 = float(input("Second number: "))

r = n1 + n2

print(f"Result: {r}")

elif c == "2":

n1 = float(input("First number: "))

n2 = float(input("Second number: "))

r = n1 - n2

print(f"Result: {r}")

elif c == "3":

n1 = float(input("First number: "))

n2 = float(input("Second number: "))

r = n1 \* n2

print(f"Result: {r}")

elif c == "4":

n1 = float(input("First number: "))

n2 = float(input("Second number: "))

if n2 == 0:

print("Error: Division by zero!")

else:

r = n1 / n2

print(f"Result: {r}")

elif c == "5":

break

else:

print("Invalid choice!")

**Problems with this code:**

* Poor variable names (c, n1, n2, r)
* Repeated code for getting user input
* No functions - everything in one big block
* Hard to test individual operations
* No error handling for invalid input

**Step-by-step refactoring:**

# REFACTORED VERSION

class Calculator:

"""A simple calculator class with basic arithmetic operations."""

def \_\_init\_\_(self):

"""Initialize the calculator."""

self.operations = {

"1": ("Addition", self.add),

"2": ("Subtraction", self.subtract),

"3": ("Multiplication", self.multiply),

"4": ("Division", self.divide)

}

def add(self, first\_number, second\_number):

"""Add two numbers."""

return first\_number + second\_number

def subtract(self, first\_number, second\_number):

"""Subtract second number from first number."""

return first\_number - second\_number

def multiply(self, first\_number, second\_number):

"""Multiply two numbers."""

return first\_number \* second\_number

def divide(self, first\_number, second\_number):

"""Divide first number by second number."""

if second\_number == 0:

raise ValueError("Cannot divide by zero!")

return first\_number / second\_number

def get\_user\_numbers(self):

"""Get two numbers from the user with error handling."""

try:

first\_number = float(input("Enter first number: "))

second\_number = float(input("Enter second number: "))

return first\_number, second\_number

except ValueError:

print("Error: Please enter valid numbers!")

return None, None

def display\_menu(self):

"""Display the calculator menu options."""

print("\n=== Calculator Menu ===")

for key, (operation\_name, \_) in self.operations.items():

print(f"{key}. {operation\_name}")

print("5. Exit")

def get\_user\_choice(self):

"""Get the user's menu choice."""

return input("Choose an operation (1-5): ")

def perform\_operation(self, choice):

"""Perform the selected operation."""

if choice not in self.operations:

print("Invalid choice! Please select 1-5.")

return

operation\_name, operation\_function = self.operations[choice]

print(f"\n--- {operation\_name} ---")

first\_number, second\_number = self.get\_user\_numbers()

if first\_number is None or second\_number is None:

return # Error getting numbers

try:

result = operation\_function(first\_number, second\_number)

print(f"Result: {first\_number} {self.\_get\_operation\_symbol(choice)} {second\_number} = {result}")

except ValueError as error:

print(f"Error: {error}")

def \_get\_operation\_symbol(self, choice):

"""Get the mathematical symbol for display purposes."""

symbols = {"1": "+", "2": "-", "3": "×", "4": "÷"}

return symbols.get(choice, "?")

def run(self):

"""Run the calculator program."""

print("Welcome to the Calculator Program!")

while True:

self.display\_menu()

user\_choice = self.get\_user\_choice()

if user\_choice == "5":

print("Thank you for using the calculator!")

break

self.perform\_operation(user\_choice)

# Using the refactored calculator

if \_\_name\_\_ == "\_\_main\_\_":

calculator = Calculator()

calculator.run()

#### ****Practice Exercise 2: Student Progress Tracker****

**Instructor Script:** "Let's refactor a student progress tracking system. This code works but has several quality issues."

# BEFORE REFACTORING - Messy student tracker

students = {}

def add\_s(n, a):

students[n] = {"age": a, "grades": []}

def add\_g(n, g):

if n in students:

students[n]["grades"].append(g)

def calc\_avg(n):

if n in students and students[n]["grades"]:

return sum(students[n]["grades"]) / len(students[n]["grades"])

return 0

def show\_all():

for name, data in students.items():

avg = calc\_avg(name)

print(f"{name} ({data['age']}): {avg:.1f}")

# Usage (confusing and error-prone)

add\_s("Alice", 16)

add\_g("Alice", 85)

add\_g("Alice", 92)

show\_all()

**Guided Refactoring Steps:**

1. **Identify Problems:**
   * Cryptic function names (add\_s, add\_g, calc\_avg)
   * Global variable usage
   * No error handling
   * Poor parameter names (n, a, g)
2. **Apply Refactoring Techniques:**

# AFTER REFACTORING - Clean student tracker

class StudentProgressTracker:

"""A system to track student information and academic progress."""

def \_\_init\_\_(self):

"""Initialize an empty student database."""

self.students = {}

def add\_student(self, student\_name, student\_age):

"""Add a new student to the tracking system."""

if not student\_name or not student\_name.strip():

print("Error: Student name cannot be empty.")

return False

if student\_age < 0 or student\_age > 150:

print("Error: Invalid age. Age must be between 0 and 150.")

return False

if student\_name in self.students:

print(f"Student {student\_name} already exists.")

return False

self.students[student\_name] = {

"age": student\_age,

"grades": []

}

print(f"Successfully added student: {student\_name} (age {student\_age})")

return True

def add\_grade(self, student\_name, grade):

"""Add a grade for an existing student."""

if student\_name not in self.students:

print(f"Error: Student {student\_name} not found.")

return False

if not isinstance(grade, (int, float)):

print("Error: Grade must be a number.")

return False

if grade < 0 or grade > 100:

print("Error: Grade must be between 0 and 100.")

return False

self.students[student\_name]["grades"].append(grade)

print(f"Added grade {grade} for {student\_name}")

return True

def calculate\_student\_average(self, student\_name):

"""Calculate the average grade for a specific student."""

if student\_name not in self.students:

return None

grades = self.students[student\_name]["grades"]

if not grades:

return 0

return sum(grades) / len(grades)

def get\_student\_letter\_grade(self, student\_name):

"""Get the letter grade for a student based on their average."""

average = self.calculate\_student\_average(student\_name)

if average is None:

return "N/A"

if average >= 90:

return "A"

elif average >= 80:

return "B"

elif average >= 70:

return "C"

elif average >= 60:

return "D"

else:

return "F"

def display\_student\_report(self, student\_name):

"""Display detailed information for one student."""

if student\_name not in self.students:

print(f"Student {student\_name} not found.")

return

student = self.students[student\_name]

average = self.calculate\_student\_average(student\_name)

letter\_grade = self.get\_student\_letter\_grade(student\_name)

print(f"\n--- Report for {student\_name} ---")

print(f"Age: {student['age']}")

print(f"Grades: {student['grades']}")

print(f"Average: {average:.1f}")

print(f"Letter Grade: {letter\_grade}")

def display\_all\_students(self):

"""Display summary information for all students."""

if not self.students:

print("No students in the system yet.")

return

print("\n=== All Students Summary ===")

for student\_name, student\_data in self.students.items():

average = self.calculate\_student\_average(student\_name)

letter\_grade = self.get\_student\_letter\_grade(student\_name)

print(f"{student\_name} (Age {student\_data['age']}): {average:.1f} ({letter\_grade})")

def get\_class\_statistics(self):

"""Calculate and display class-wide statistics."""

if not self.students:

print("No students to analyze.")

return

all\_averages = []

for student\_name in self.students:

avg = self.calculate\_student\_average(student\_name)

if avg > 0: # Only include students with grades

all\_averages.append(avg)

if not all\_averages:

print("No grades recorded yet.")

return

class\_average = sum(all\_averages) / len(all\_averages)

highest\_average = max(all\_averages)

lowest\_average = min(all\_averages)

print(f"\n=== Class Statistics ===")

print(f"Class Average: {class\_average:.1f}")

print(f"Highest Average: {highest\_average:.1f}")

print(f"Lowest Average: {lowest\_average:.1f}")

print(f"Total Students: {len(self.students)}")

print(f"Students with Grades: {len(all\_averages)}")

# Demonstration of the refactored system

def demonstrate\_student\_tracker():

"""Show how to use the refactored student tracking system."""

tracker = StudentProgressTracker()

# Add students

tracker.add\_student("Alice", 16)

tracker.add\_student("Bob", 17)

tracker.add\_student("Carol", 16)

# Add grades

tracker.add\_grade("Alice", 85)

tracker.add\_grade("Alice", 92)

tracker.add\_grade("Alice", 78)

tracker.add\_grade("Bob", 90)

tracker.add\_grade("Bob", 87)

tracker.add\_grade("Carol", 95)

tracker.add\_grade("Carol", 98)

tracker.add\_grade("Carol", 92)

# Display results

tracker.display\_all\_students()

tracker.display\_student\_report("Alice")

tracker.get\_class\_statistics()

if \_\_name\_\_ == "\_\_main\_\_":

demonstrate\_student\_tracker()

#### ****Guided Practice Session****

**Instructor Script:** "Now let's work together on identifying refactoring opportunities in this code. I'll show you a piece of messy code, and we'll discuss what needs to be improved."

# PRACTICE CODE - What needs refactoring?

def process\_grades():

data = []

while True:

n = input("Name (or 'done'): ")

if n == "done":

break

g1 = int(input("Grade 1: "))

g2 = int(input("Grade 2: "))

g3 = int(input("Grade 3: "))

avg = (g1 + g2 + g3) / 3

if avg >= 90:

ltr = "A"

elif avg >= 80:

ltr = "B"

else:

ltr = "C"

data.append([n, g1, g2, g3, avg, ltr])

for d in data:

print(f"{d[0]}: {d[1]}, {d[2]}, {d[3]} -> {d[4]:.1f} ({d[5]})")

process\_grades()

**Discussion Questions for Students:**

1. What variable names are confusing?
2. What code is repeated or could be extracted into functions?
3. How could we make this more organized?
4. What error handling is missing?

**Expected Student Responses and Guidance:**

* Variable names: n, g1, g2, g3, avg, ltr, d should be more descriptive
* Functions needed: input validation, grade calculation, letter grade assignment
* Organization: Could use a class or at least separate functions
* Error handling: What if user enters non-numeric grades?

### ****8:35 PM -- 9:05 PM: Session 4 -- Best Practices and Code Review****

**Objective:** Learn code review practices and establish standards for writing maintainable code.

**Instructor Script:** "Now that you can refactor code, let's talk about how to write good code from the start and how to review code effectively."

#### ****Code Review Checklist for Beginners****

**When reviewing code (your own or others'), ask these questions:**

1. **Naming and Clarity:**
   * Are variable and function names descriptive?
   * Can someone else understand what the code does without comments?
   * Are there any single-letter variables (except for loop counters)?
2. **Function Design:**
   * Does each function have a single, clear purpose?
   * Are functions too long (more than 20-30 lines might be too much)?
   * Do functions have too many parameters (more than 3-4 might be too many)?
3. **Code Duplication:**
   * Is any code repeated more than twice?
   * Could repeated logic be extracted into a function?
   * Are there similar but slightly different pieces of code that could be generalized?
4. **Error Handling:**
   * What happens if the user enters invalid input?
   * Are there try/except blocks where needed?
   * Do error messages help the user understand what went wrong?
5. **Organization:**
   * Is related functionality grouped together?
   * Would a class make sense for organizing related data and functions?
   * Is the code structured logically from top to bottom?

#### ****Establishing Coding Standards****

**Instructor Script:** "Here are some simple standards that will make your code much more professional:"

# GOOD CODING STANDARDS EXAMPLE

class ShoppingCart:

"""A shopping cart that manages items and calculates totals."""

def \_\_init\_\_(self):

"""Initialize an empty shopping cart."""

self.items = []

self.tax\_rate = 0.08 # 8% tax rate

def add\_item(self, item\_name, price, quantity=1):

"""Add an item to the cart.

Args:

item\_name (str): Name of the item

price (float): Price per unit

quantity (int): Number of items (default: 1)

Returns:

bool: True if item was added successfully

"""

if not item\_name or not item\_name.strip():

print("Error: Item name cannot be empty.")

return False

if price < 0:

print("Error: Price cannot be negative.")

return False

if quantity < 1:

print("Error: Quantity must be at least 1.")

return False

# Check if item already exists

for item in self.items:

if item['name'].lower() == item\_name.lower():

item['quantity'] += quantity

print(f"Updated {item\_name}: now {item['quantity']} in cart")

return True

# Add new item

new\_item = {

'name': item\_name,

'price': price,

'quantity': quantity

}

self.items.append(new\_item)

print(f"Added {quantity} x {item\_name} to cart")

return True

def calculate\_subtotal(self):

"""Calculate the subtotal before tax."""

subtotal = 0

for item in self.items:

subtotal += item['price'] \* item['quantity']

return subtotal

def calculate\_tax(self):

"""Calculate tax amount based on subtotal."""

return self.calculate\_subtotal() \* self.tax\_rate

def calculate\_total(self):

"""Calculate the final total including tax."""

return self.calculate\_subtotal() + self.calculate\_tax()

def display\_cart(self):

"""Display all items in the cart with totals."""

if not self.items:

print("Your cart is empty.")

return

print("\n=== Shopping Cart ===")

for item in self.items:

item\_total = item['price'] \* item['quantity']

print(f"{item['name']}: ${item['price']:.2f} x {item['quantity']} = ${item\_total:.2f}")

subtotal = self.calculate\_subtotal()

tax = self.calculate\_tax()

total = self.calculate\_total()

print("-" \* 25)

print(f"Subtotal: ${subtotal:.2f}")

print(f"Tax ({self.tax\_rate\*100}%): ${tax:.2f}")

print(f"Total: ${total:.2f}")

# Example usage showing good practices

def demonstrate\_shopping\_cart():

"""Demonstrate the shopping cart functionality."""

cart = ShoppingCart()

# Add some items

cart.add\_item("Apple", 1.50, 3)

cart.add\_item("Bread", 2.99)

cart.add\_item("Milk", 3.49, 2)

# Display the cart

cart.display\_cart()

if \_\_name\_\_ == "\_\_main\_\_":

demonstrate\_shopping\_cart()

**Key Standards Demonstrated:**

1. **Clear class and method names**
2. **Docstrings explaining what each function does**
3. **Input validation with helpful error messages**
4. **Consistent formatting and indentation**
5. **Logical organization of methods**
6. **Appropriate use of constants (tax\_rate)**
7. **Good separation of concerns (separate methods for different calculations)**

#### ****Common Refactoring Mistakes to Avoid****

**Instructor Script:** "Here are some common mistakes beginners make when refactoring:"

1. **Over-refactoring:**

# DON'T do this - too many tiny functions

def get\_first\_name():

return input("First name: ")

def get\_last\_name():

return input("Last name: ")

def combine\_names(first, last):

return f"{first} {last}"

# Better - one function for related input

def get\_full\_name():

first\_name = input("First name: ")

last\_name = input("Last name: ")

return f"{first\_name} {last\_name}"

1. **Changing functionality while refactoring:**

# WRONG - this changes behavior

# Original: returns 0 for empty list

def calculate\_average\_old(numbers):

if not numbers:

return 0

return sum(numbers) / len(numbers)

# Refactored: now raises exception - this is a behavior change!

def calculate\_average\_new(numbers):

if not numbers:

raise ValueError("Cannot calculate average of empty list")

return sum(numbers) / len(numbers)

1. **Not testing after refactoring:**
   * Always test your refactored code to ensure it still works
   * Run the same inputs and verify you get the same outputs
   * Check edge cases (empty inputs, invalid data, etc.)

### ****9:05 PM -- 9:25 PM: Breakout #2: Capstone Working Session****

**Activity:** Apply refactoring techniques to capstone projects, focusing on improving existing code quality.

**Instructor Guidance:**

* Review students' capstone code for refactoring opportunities
* Help identify areas where functions can be extracted
* Suggest improvements to variable and function names
* Guide students in organizing code into classes where appropriate

### ****9:25 PM -- 9:30 PM: Wrap-Up & Final Questions****

**Instructor Script:** "Today we've learned the fundamentals of code refactoring - one of the most important skills for any programmer. Remember, refactoring isn't just about making code look pretty; it's about making it maintainable, readable, and professional. The techniques you've learned today - extracting functions, choosing good names, eliminating duplication, and organizing with classes - will serve you throughout your programming career."

**Review Key Points:**

* Refactoring improves code without changing functionality
* Good variable and function names make code self-documenting
* The DRY principle helps eliminate maintenance headaches
* Functions should have single, clear responsibilities
* Classes help organize related data and functionality
* Always test after refactoring to ensure nothing broke

**Final Thoughts:**

* Start writing clean code from the beginning - it's easier than refactoring later
* When you see code that's hard to understand, that's a signal it needs refactoring
* Good code is written for humans to read, not just computers to execute
* Refactoring is an ongoing process, not a one-time activity

**Prompting Question:** "What's one refactoring technique you're excited to apply to your own code?"

## ****After-Class Quiz (5 questions)****

1. What is the main goal of code refactoring?
   * A) To add new features to existing code
   * B) To fix bugs in broken code
   * \*C) To improve code structure and readability without changing functionality
   * D) To make code run faster
2. Which of these demonstrates the DRY principle?
   * A) Writing detailed comments for every line of code
   * B) Using short variable names to save typing
   * \*C) Creating a function for code that appears in multiple places
   * D) Writing code as quickly as possible
3. What makes a good variable name?
   * A) It should be as short as possible
   * B) It should use abbreviations to save space
   * \*C) It should clearly describe what the variable contains
   * D) It should use numbers instead of letters
4. When should you extract code into a separate function?
   * A) When the code is more than 5 lines long
   * B) When you need to use a loop
   * \*C) When you find yourself copying and pasting the same code
   * D) When you're using if statements
5. What is a key benefit of organizing code into classes?
   * A) Classes make code run faster
   * B) Classes are required in Python
   * \*C) Classes group related data and functions together
   * D) Classes automatically fix bugs in your code

## ****Homework Assignment****

**Practice Refactoring Exercise:**

Students will be provided with a messy Python program (a simple library book checkout system) that has the following problems:

* Poor variable names
* Repeated code
* No functions or classes
* No error handling

Their task is to refactor this code by:

1. Improving all variable and function names
2. Extracting repeated code into functions
3. Organizing the code into a class structure
4. Adding basic error handling
5. Writing a brief explanation of what they changed and why

**Estimated Time:** 45-60 minutes

**Deliverables:**

* Original messy code (provided)
* Refactored clean code
* Written explanation of improvements made (150-200 words)

This homework reinforces all the concepts learned in class and gives students hands-on practice with real refactoring scenarios.