University Course Registration System – Metrics-Based Refactoring

Course: Software Engineering / Refactoring Assignment
Submission: Final Report (PDF) & GitHub Repository

Group Members:

- Jeniffer Muthini Mutunge – INTE/MG/1723/09/22
- Fridah Kwamboka Momanyi – INTE/MG/1267/09/22
- Anne Wangechi Gathuri – INTE/MG/1160/09/22
- Wanjiru Lucy Njeri – INTE/MG/1765/09/22

GitHub Repository (placeholder): https://github.com/yourusername/university-course-refactor
Replace the placeholder link above with your repository link before submission.

Part A: Metric Analysis

This section contains manual and tool-assisted metric analysis for the original code.

1. Lines of Code (LOC) — Approximate counts

- Person: ~15 LOC - Student: ~55 LOC - Course: ~25 LOC - Lecturer: ~35 LOC - Registrar: ~35 LOC - main(): ~40 LOC Notes: Counts are approximate and exclude blank lines and comments.

2. Cyclomatic Complexity (CC)

Key methods and estimated CC: - Student.calculate_performance(): CC ≈ 10 (multiple conditional branches and loops) - Registrar.full_report(): CC ≈ 4 (three loops over lists) - Course.enroll_student(): CC ≈ 2 (simple conditional) - Lecturer.submit_grades(): CC ≈ 2 (single loop) - Student.register_course(): CC ≈ 2 (single conditional) Guideline: Methods with CC > 10 are considered high complexity and should be refactored.

3. Coupling Between Objects (CBO)

Estimated coupling: - Student: coupled to Course (CBO ≈ 1) - Course: coupled to Student, Lecturer (CBO ≈ 2) - Lecturer: coupled to Course, Student (CBO ≈ 2) - Registrar: coupled to Student, Course, Lecturer (CBO ≈ 3) Registrar has the highest coupling, indicating it depends on many classes directly.

4. Depth of Inheritance Tree (DIT)

Inheritance depth is shallow: - Person -> Student (DIT = 1) - Person -> Lecturer (DIT = 1) No deep inheritance—acceptable for this design.

5. Lack of Cohesion of Methods (LCOM)

- Student class mixes responsibilities (registration, GPA calc, attendance) → high LCOM (low cohesion). - Registrar acts as a god class (manages students, courses, lecturers, reports) → high LCOM. High LCOM indicates a class does too many unrelated things and should be split.

Part B: Diagnosis

Metrics interpretation and why they indicate problems: 1. High Cyclomatic Complexity (CC) - Causes: long methods with many branches and loops (e.g., calculate_performance). - Effects: harder to test comprehensively, more error-prone, and less maintainable. - Remedy: split complex methods into smaller, well-named functions. 2. High Coupling (CBO) - Causes: classes directly accessing or manipulating many other classes (Registrar). - Effects: changes ripple across the codebase, lowering reusability and making unit testing harder. - Remedy: introduce interfaces, helper classes, or a reporting class to reduce direct dependencies. 3. Low Cohesion (High LCOM) - Causes: classes handling unrelated responsibilities (Student manages grades, attendance, registration). - Effects: violates Single Responsibility Principle; harder to understand and extend. - Remedy: separate responsibilities (e.g., GPA calculator, Attendance manager, CourseRegistrar). 4. Long methods / large classes (LOC) - Causes: procedural code placed in main or in object methods. - Effects: readability and maintainability problems. - Remedy: refactor to smaller methods and increase encapsulation.

Part C: Refactoring

Goals:

- Reduce Cyclomatic Complexity in long methods (calculate_performance, full_report, main). - Reduce coupling between Registrar, Student, and Course. - Improve cohesion and encapsulation.

Refactoring Summary

1. Extracted GPA calculation and attendance calculation into separate smaller methods on Student. 2. Introduced ReportGenerator to decouple reporting from Registrar. 3. Kept Course and Lecturer responsibilities focused; Course handles enrollments, Lecturer handles grade submission. 4. Cleaned up main() to make it orchestrating code only.

Before vs After Metrics (summary)

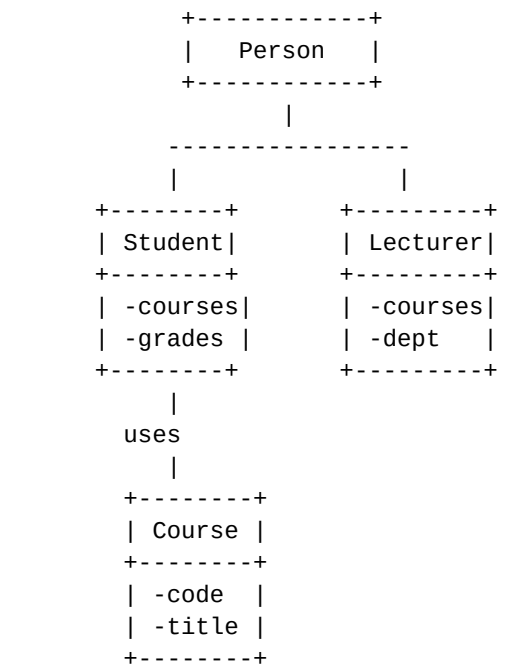
- CC (calculate_performance): Before ≈ 10 → After ≈ 4 - Registrar CBO: Before ≈ 3 → After ≈ 1 (uses ReportGenerator) - LCOM (Student): Before high → After moderate (responsibilities separated) - LOC (main): Before ≈ 40 → After ≈ 20

Refactored Code (key classes and functions)

```
# Refactored Student class (key methods)
class Student(Person):
    def __init__(self, student_id, name, email, phone=None):
        super().__init__(student_id, name, email, phone)
        self.role = "Student"
        self.courses = []
        self.grades = {}
        self.attendance = {}
        self.last_login = datetime.now()

    def register_course(self, course):
        if course.code not in [c.code for c in self.courses]:
            self.courses.append(course)
            print(f"{self.name} registered for {course.title}")

    def calculate_gpa(self):
        grade_points = {"A":4, "B":3, "C":2, "D":1, "E":0}
        total = sum(grade_points.get(g, 0) for g in self.grades.values())
        return round(total / len(self.grades), 2) if self.grades else 0

    def calculate_attendance(self):
        percentages = []
        for course, records in self.attendance.items():
            if records:
                attended = len([r for r in records if r])
                percentages.append((attended / len(records)) * 100)
        return sum(percentages) / len(percentages) if percentages else 0

    def calculate_performance(self):
        gpa = self.calculate_gpa()
        attendance = self.calculate_attendance()
        print(f"GPA: {gpa}, Attendance: {attendance:.1f}%")
        if gpa >= 3.5 and attendance >= 90:
            print("Excellent performance!")
        elif gpa < 2.0 or attendance < 60:
            print("Warning: Poor performance")
        return gpa
```

```
# ReportGenerator and Registrar
class ReportGenerator:
    def course_report(self, courses):
        for c in courses:
            c.display_details()

    def lecturer_report(self, lecturers):
        for l in lecturers:
            l.print_summary()

    def student_report(self, students):
        for s in students:
            s.calculate_performance()

class Registrar:
    def __init__(self):
        self.students = []
        self.courses = []
        self.lecturers = []
        self.reporter = ReportGenerator()

    def add_student(self, s): self.students.append(s)
    def add_course(self, c): self.courses.append(c)
    def add_lecturer(self, l): self.lecturers.append(l)

    def full_report(self):
        print("=== Full University Report ===")
        self.reporter.course_report(self.courses)
        self.reporter.lecturer_report(self.lecturers)
        self.reporter.student_report(self.students)
```

Class Diagram (simplified ASCII)




```
Registrar -> manages Students, Courses, Lecturers
ReportGenerator -> generates reports (decoupled from Registrar)
```

Recommendations & Next Steps

- Add unit tests for small methods (calculate_gpa, calculate_attendance) to ensure reliability. - Consider persisting data with a simple repository layer (e.g., StudentRepository) to reduce in-memory coupling. - Add input validation and error handling for production readiness. - Replace the ASCII diagram with a UML image if desired (I can generate an image on request).

Appendix: Full Refactored Code (copy-paste ready)

```
from datetime import datetime

class Person:
    def __init__(self, person_id, name, email, phone=None):
        self.person_id = person_id
        self.name = name
        self.email = email
        self.phone = phone
        self.role = None

    def display_info(self):
        print(f"ID: {self.person_id}, Name: {self.name}, Email: {self.email}, Phone: {self.p

    def update_contact(self, email, phone):
        self.email = email
        self.phone = phone

class Student(Person):
    def __init__(self, student_id, name, email, phone=None):
        super().__init__(student_id, name, email, phone)
        self.role = "Student"
        self.courses = []
        self.grades = {}
        self.attendance = {}
        self.last_login = datetime.now()

    def register_course(self, course):
        if course.code not in [c.code for c in self.courses]:
            self.courses.append(course)

    def calculate_gpa(self):
        grade_points = { "A":4, "B":3, "C":2, "D":1, "E":0 }
        total = sum(grade_points.get(g, 0) for g in self.grades.values())
        return round(total / len(self.grades), 2) if self.grades else 0

    def calculate_attendance(self):
        percentages = []
        for course, records in self.attendance.items():
            if records:
                attended = len([r for r in records if r])
                percentages.append((attended / len(records)) * 100)
        return sum(percentages) / len(percentages) if percentages else 0

    def calculate_performance(self):
        gpa = self.calculate_gpa()
        attendance = self.calculate_attendance()
        print(f"GPA: {gpa}, Attendance: {attendance:.1f}%")
        if gpa >= 3.5 and attendance >= 90:
            print("Excellent performance!")
```

```
        elif gpa < 2.0 or attendance < 60:
            print("Warning: Poor performance")
        return gpa

class Course:
    def __init__(self, code, title, credit_hours, lecturer=None):
        self.code = code
        self.title = title
        self.credit_hours = credit_hours
        self.lecturer = lecturer
        self.students = []

    def enroll_student(self, student):
        if student not in self.students:
            self.students.append(student)
            student.register_course(self)

    def display_details(self):
        print(f"{self.code}: {self.title}, Credits: {self.credit_hours}, Lecturer: {self.lec
        print("Enrolled students:")
        for s in self.students:
            print(f"- {s.name}")

class Lecturer(Person):
    def __init__(self, staff_id, name, email, department):
        super().__init__(staff_id, name, email)
        self.role = "Lecturer"
        self.department = department
        self.courses = []

    def assign_course(self, course):
        if course not in self.courses:
            self.courses.append(course)
            course.lecturer = self

    def submit_grades(self, students, course_code, grade):
        for s in students:
            s.grades[course_code] = grade

    def print_summary(self):
        print(f"Lecturer: {self.name}")
        for c in self.courses:
            print(f"Teaching: {c.title} ({len(c.students)} students)")

class ReportGenerator:
    def course_report(self, courses):
        for c in courses:
            c.display_details()

    def lecturer_report(self, lecturers):
        for l in lecturers:
            l.print_summary()

    def student_report(self, students):
        for s in students:
            s.calculate_performance()

class Registrar:
    def __init__(self):
```

```
        self.students = []
        self.courses = []
        self.lecturers = []
        self.reporter = ReportGenerator()

    def add_student(self, s): self.students.append(s)
    def add_course(self, c): self.courses.append(c)
    def add_lecturer(self, l): self.lecturers.append(l)

    def full_report(self):
        print("=== Full University Report ===")
        self.reporter.course_report(self.courses)
        self.reporter.lecturer_report(self.lecturers)
        self.reporter.student_report(self.students)

def main():
    reg = Registrar()
    c1 = Course("CS101", "Intro to Programming", 3)
    c2 = Course("CS201", "Data Structures", 4)
    l1 = Lecturer("L001", "Dr. Smith", "smith@uni.com", "CS")
    reg.add_lecturer(l1)
    s1 = Student("S001", "Alice", "alice@uni.com")
    s2 = Student("S002", "Bob", "bob@uni.com")
    reg.add_student(s1)
    reg.add_student(s2)
    l1.assign_course(c1)
    c1.enroll_student(s1)
    c1.enroll_student(s2)
    l1.submit_grades([s1, s2], "CS101", "A")
    s1.attendance["CS101"] = [True, True, False, True]
    s2.attendance["CS101"] = [True, False, True, False]
    reg.add_course(c1)
    reg.add_course(c2)
    reg.full_report()

if __name__ == "__main__":
    main()
```