Design Document: Course Registration System (Unsafe Version)

Overview

This application simulates a multi-threaded course registration system with a graphical user interface for manual entering of student and course information. Each student is modeled as a thread attempting to improve their schedule by dropping and enrolling in courses over time.

The first version of the system deliberately lacks synchronization in critical areas to expose data races, demonstrating the need for concurrency control in real-world software systems.

Architecture

Registrar:
- Central manager of all courses and students.
- Maintains shared data structures: Map<String, Course> courses and Map<String, Student> students.
- Handles enrollment (tryAdd), dropping (tryDrop), and management of two course shopping carts (mostDesired and alsoOk).

Student:
- Each student is a thread.
- Has mostDesired and alsoOk course lists, and a currentCourses set.
- Attempts to improve schedule until success or max attempts reached.

Course:
- Contains name, capacity, and a shared mutable list of enrolled students.

RegistrationGUI:
- Swing-based interface for manually and automatically adding students and courses.
- Allows thread execution, enrollment, dropping, and state inspection.

What Counts as “Correct” Behavior

- No student enrolled in a course more than once.
- Course enrollment must not exceed capacity.
- Students must be enrolled in at most 4 courses.
- A dropped course should be recoverable unless full.

What Can Go Wrong (Incorrect States)

- Simultaneous enrollments exceed course capacity.
- A dropped course cannot be recovered because another thread takes the spot.
- Inconsistent views of enrollment (student thinks they're in, course disagrees).

How Data Races Are Triggered

Thread.sleep() calls simulate delays that increase likelihood of interleaved execution.

Examples:
- Sleep before modifying course enrollment list.
- Sleep between drop and add operations.
- Sleep before re-enrolling in a dropped class (in the event that the student was not able to enroll in the new course)

These simulate real concurrency bugs caused by timing.

Example of One Possible Race

(In Registrar.java, line 49 in unsafe version, line 52 in safe version)

```
if (c.enrolled.size() >= c.capacity)
```

This line catches the condition where the current number of enrolled students in the course is greater than or equal to the capacity limit for the course. This check is referred to as "check" in the Sequential Consistency Diagram below. Let us assume that `c.enrolled.size()` is 0 at the start of the diagram running, and that `c.capacity` is 1. This comes from the code where a thread is attempting to enroll itself (a Student) in a course. Assume that Thread A and Thread B are both attempting to enroll in the Course cs1.

Time	Thread A	Thread B	Num Enrolled
T1	check is true		0
T2		check is true	0
T3	A enrolls in cs1		1
T4		B enrolls in cs1	2

The result is that the number of students enrolled in Course cs1 is 2, but the capacity limit was 1.

Summary

This version intentionally allows problematic concurrency to illustrate the need for synchronization. It shows how shared data without protection leads to real-world failures.