Caitlyn Lefebvre, Travis Fine, and Jaice Arsenault
Professor Sirazum Tisha
CMS 270
2 December 2025

Final Group Project

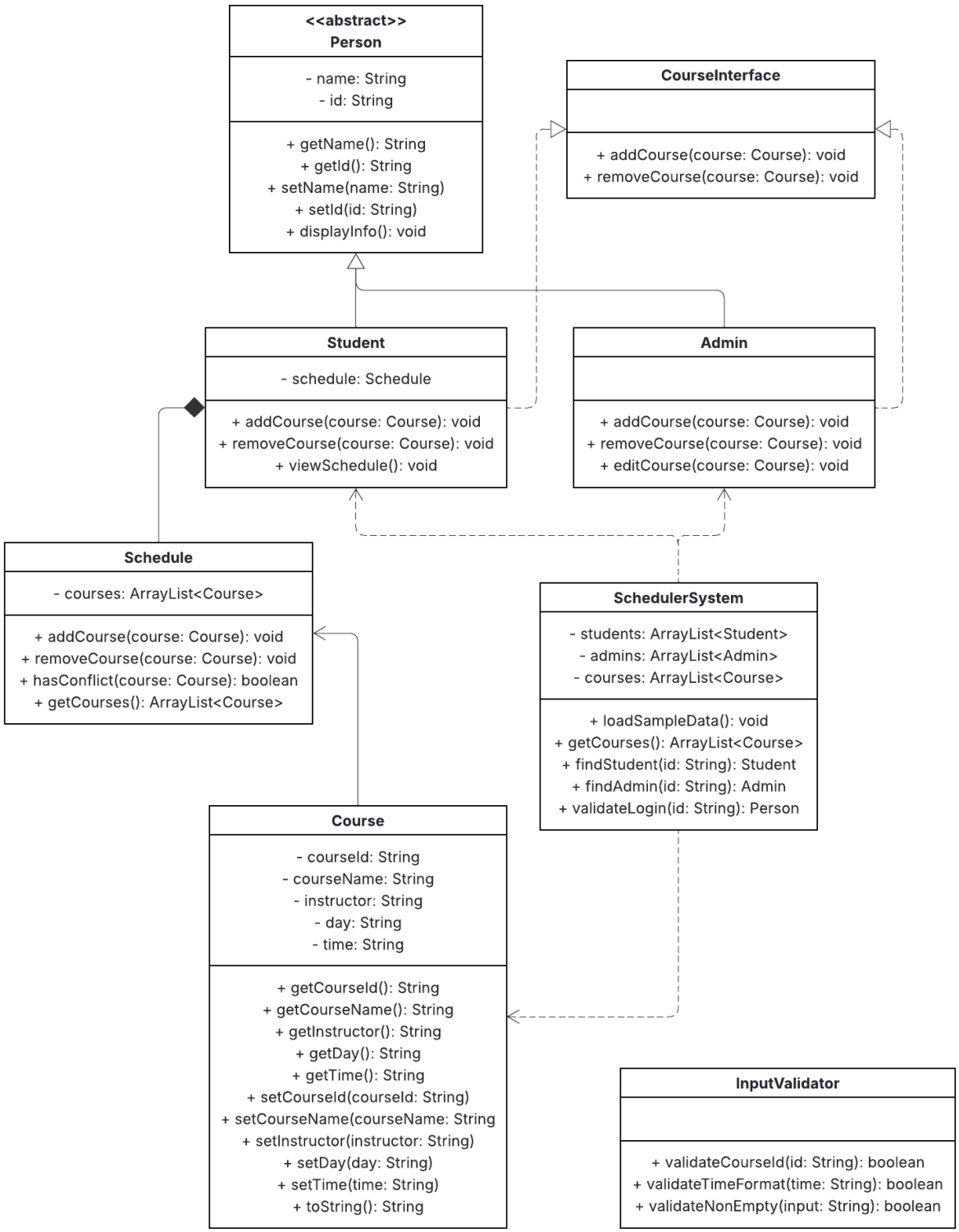
1. **Title Page**:
   Project Title: Rollins Class Scheduler
   Team Members and Roles:
   - Caitlyn Lefebvre: Lead Developer, implemented OOP logic, systemic architecture, and model layer
   - Travis Fine: Documentation writer, code reviewer, and logic consistency analyst
   - Jaice Arsenault: User Interface developer, Swing GUI implementation

2. **Introduction**:
   The purpose of the Rollins Class Scheduler project is to simulate a simplified version of the Rollins College course registration system using Java and object-oriented design principles. The system allows both students and administrators to interact with a set of course offerings through a graphical user interface (GUI).

   Students can log in, browse available courses, add or remove courses from their personal schedule, and view their complete schedule. Administrators have enhanced privileges, including the ability to create new courses, modify existing ones, and remove them entirely from the system.

   We chose this topic because course scheduling is a universal process that affects all Rollins students every semester. By recreating this system, from the backend data management to the user interface, we gained valuable insight into how registration systems enforce rules such as time conflicts, credit limits, and instructor scheduling. This topic allowed our team to apply core Computer Science concepts—especially abstraction, inheritance, and encapsulation—through a real-world context.

3. **UML Diagram**:
   We created this as a baseline to build our program off of. Classes have been changed and added since.



4. **Implementation Description**:
   Our project was built intentionally around core object-oriented programming concepts. Below is a breakdown of how each principle was applied with concrete examples from our code.

   Encapsulation
   Encapsulation was used throughout the system to hide internal data structures and expose only necessary methods.

   Example:
   The Course class keeps its list of enrolled students private:
   ```
   private HashSet<String> enrolledStudents = new HashSet<>();
   ```
   Access is only possible through methods such as enrollStudent() and removeStudent(). This prevents external classes from accidentally corrupting course enrollment data. The Student class also encapsulates a Schedule object rather than exposing the underlying ArrayList<Course> directly.

   Inheritance
   Inheritance helps reduce repeated code and express shared traits.

   Example:
   Student and Admin both extend:
   ```
   Public abstract class Person
   ```
   This means both share fields such as name and id, as well as common behavior like displaying user information. Admin and Student also both implement the CourseInterface, ensuring consistency in how course operations are defined even though behavior differs.

   Polymorphism
   Polymorphism appears through the shared interface and method overriding.

   Example:
   Both Student and Admin implement:
   ```
   void addCourse(Course course);
   void removeCourse(Course course);
   ```
   But they execute very different logic:
   - Students check time conflicts, credit limits, and course capacity.
   - Admins bypass restrictions and simply add or remove courses.

   The controller can call person.addCourse() without knowing whether the user is a Student or an Admin—this is runtime polymorphism.

   Abstraction
   Abstraction appears through high-level classes and hidden complexity.

   Example:
   The SchedulerSystem class abstracts away:

- Loading sample data
- Preventing instructor schedule conflicts
- Searching for students, admin, and courses
- Adding/removing users and courses with validation

The controller does not need to understand how these processes work internally—it only calls simple methods like:

```
system.addCourse(c);
system.findStudent(id);
system.removeCourse(c);
```

5. **Challenges and Solutions**:
   a) Time Conflict Detection
   Challenge: Ensuring that students could not register for overlapping courses.
   Solution: We implemented a helper method inside Schedule:
   `private boolean timeOverlap(String start1, String end1, String start2, String end2)`
   along with logic that only checks conflicts for matching days. This modular design prevented bugs and made the rule easy to adjust.
   b) Instructor Department Enforcement
   Challenge: Sample data needed instructors to only teach within one department.
   Solution: A HashMap<String, String> tracks which department each instructor belongs to. If a new sample course violates that rule, it is skipped.
   c) GUI and Controller Interaction
   Challenge: Swing is strict about how data must be updated, especially table models and switching screens.
   Solution: We organized all event listeners inside SchedulerController and kept the GUI simple to avoid mixing logic and UI code. We also used CardLayout to cleanly switch between screens.
   d) Duplicate ID Handling
   Challenge: Avoiding duplicate student/admin IDs when creating new users.
   Solution: We created a custom exception:
   `public class DuplicatedIdException extends Exception`
   that is thrown during addStudent() and addAdmin() if the ID already exists.

6. **Contribution Summary**:

Team Member	**Contributions**
Caitlyn Lefebvre	Developed the core OOP architecture, implemented the model classes (Course, Student, Admin, Schedule), designed validation rules, created sample data logic, integrated course conflict detection
Travis Fine	Created written documentation, ensured consistent logic between system components, reviewed class interactions, helped refine method behavior

Jaice Arsenault	Built the GUI using Swing (SchedulerView, layout design, styling), implemented user navigation, added branding through custom RLogo component, assisted with controller testing