Final Report: Conflict-Free Scheduling System for Small Businesses

GitHub Repository: https://github.com/osshb6/CS4090-Scheduling-App/

Team Members and Roles:

- **Oscar Sargent – Backend Development and Integration**

  Oscar was responsible for designing and implementing the core backend functionality, including the database schema, scheduling logic, and connection handlers. He also led the effort to integrate backend modules with the user interface, ensuring seamless data flow and proper function calls between layers.

- **Payton Labbee – User Interface and Backend Development**

  Payton contributed heavily to both the frontend and backend. On the UI side, he built out key screens and user interactions using Tkinter. He also worked closely with Oscar on backend logic, helping structure modules for scheduling, employee management, and constraint handling.

- **Dane Shuler – System Architecture and User Experience**

  Dane led the architectural design of the application, developing UML and sequence diagrams to define system structure. He focused on aligning design patterns with component responsibilities and enhancing the overall user experience through consistent layouts, role-based flows, and usability feedback.

- **Logan Gray – Testing and Documentation**

  Logan was in charge of the testing infrastructure, writing unit and integration tests to validate system behavior. He also ensured the project was well-documented, creating internal guides, progress reports, and supporting materials to maintain clarity and traceability throughout development.

---

1. Problem Statement and Motivation

Small businesses such as restaurants and retail shops frequently rely on manual scheduling tools like paper calendars or Excel spreadsheets. These traditional methods are prone to human error, difficult to update dynamically, and inefficient in handling constraints like employee availability or role requirements. Our project seeks to create a computer-based scheduling application that enables conflict-free, optimized shift planning while enhancing employee and manager experience through a digital interface.

---

2. Requirements Specification

Our requirements were guided by real-world constraints and business needs:

- Managers must be able to create, modify, and assign schedules.
- Employees should view and modify their availability.
- The system should store employee, shift, availability, and constraint data persistently.
- Schedules should be generated based on multiple criteria (e.g., fairness, coverage, preferences).
- A simple UI must support both managerial and regular users.
- A SQLite database is used to maintain persistence.

Key user stories included functionality such as viewing individual or team schedules, submitting time-off requests, and using constraints to generate multiple optimized schedules. Advanced security, concurrency, and payroll systems were explicitly excluded from scope.

---

3. System Design

Architecture Overview: The application is structured using a layered architecture:

- **Presentation Layer** (GUI): Uses the Observer pattern for real-time updates.
- **Application Layer**: Implements Factory (user management), Command (shift operations), and Strategy (scheduling optimization) patterns.
- **Persistence Layer**: Singleton pattern ensures one database connection for reliability.

Architectural Layers and Justification

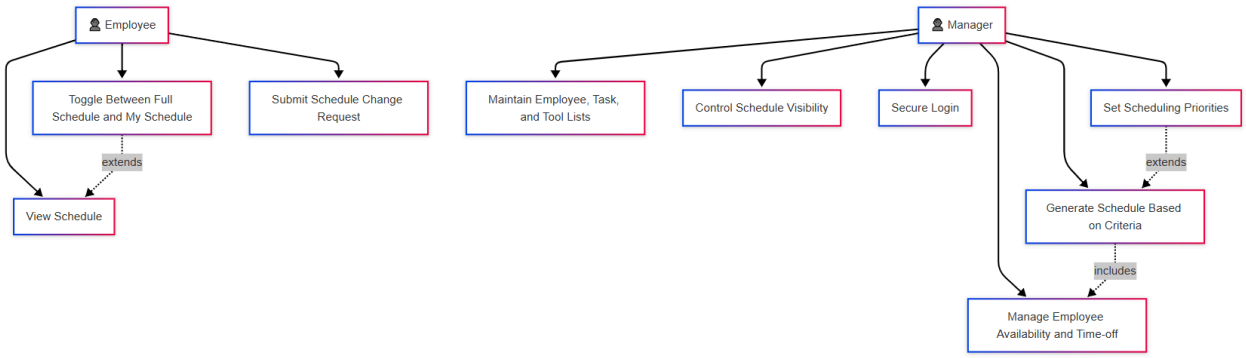
Our system follows a three-tier architecture, Presentation, Application, and Data Persistence, designed to promote modularity and scalability. The **Presentation Layer** (Tkinter GUI) isolates UI logic, allowing future upgrades such as migrating to a web or mobile interface. The **Application Layer** contains core functionalities like scheduling, user management, and optimization. This layer is highly pattern-driven, using Factory for user instantiation, Command for schedule actions, and Observer to propagate changes across modules. Finally, the **Data Layer** (SQLite + Singleton) provides a lightweight, consistent interface for storing and retrieving data, ensuring shared access across components without redundant connections.

Modularization Strategy

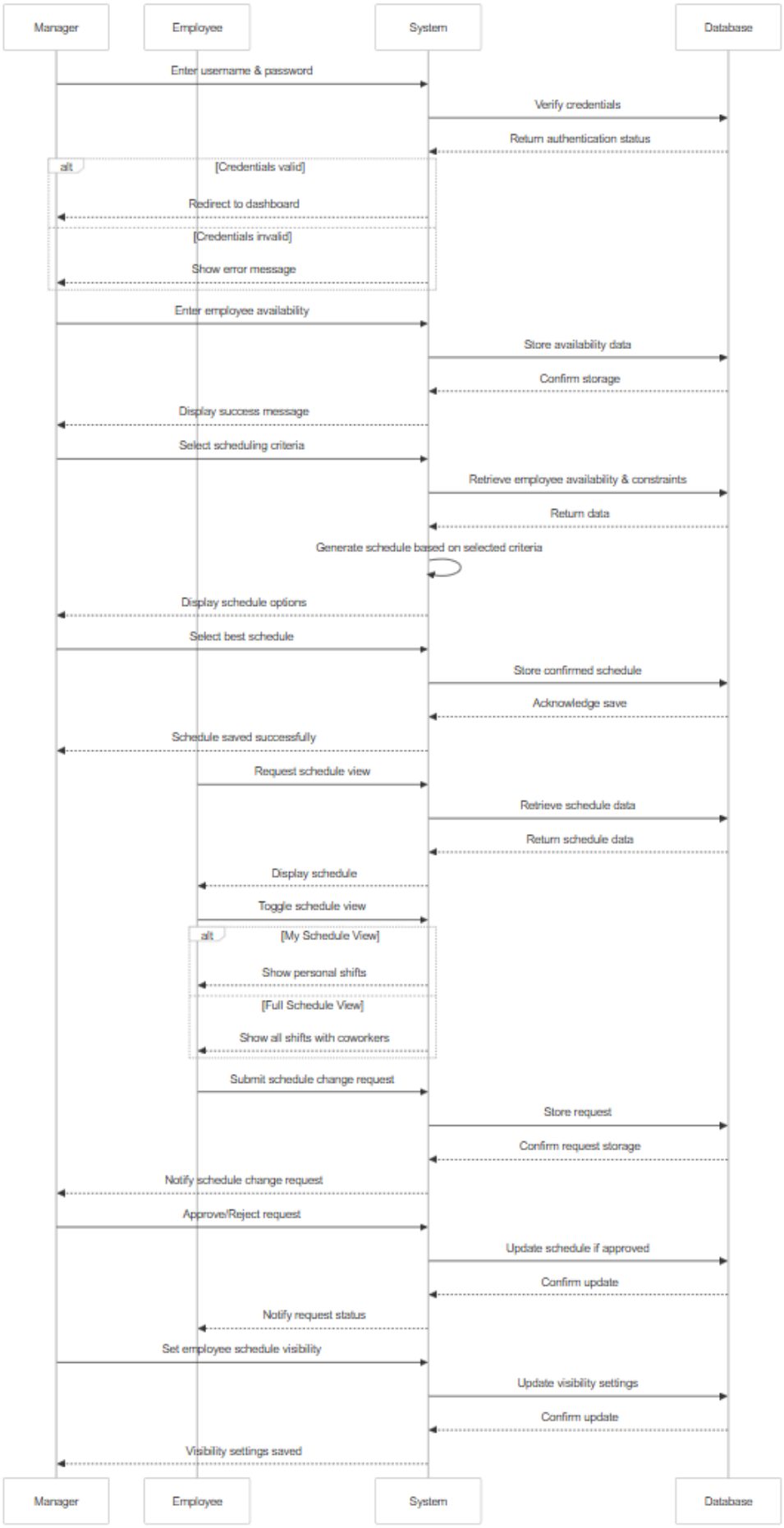
We restructured the project into clearly defined directories (`src/ui/`, `src/backend/`, storage/, `database/`, and `tests/`), each responsible for distinct tasks. This separation of concerns made collaborative development easier, improved readability, and supported more efficient debugging and testing.

Database Schema includes tables for Users, Shifts, Availability, Constraints, and OptimizationLogs. Each entity is uniquely identifiable and supports the required queries for schedule generation and reporting.

UML diagrams were created to visualize the class relationships and helped translate design into code efficiently.



Sequence diagrams were used to visualize and validate the step-by-step interactions between users, the interface, and backend components, ensuring proper flow and integration across the system.


Manager
Employee
System
Database
Enter username & password
Verify credentials
Return authentication status
alt
[Credentials valid]
Redirect to dashboard
[Credentials invalid]
Show error message
Enter employee availability
Store availability data
Confirm storage
Display success message
Select scheduling criteria
Retrieve employee availability & constraints
Return data
Generate schedule based on selected criteria
Display schedule options
Select best schedule
Store confirmed schedule
Acknowledge save
Schedule saved successfully
Request schedule view
Retrieve schedule data
Return schedule data
Display schedule
Toggle schedule view
alt
[My Schedule View]
Show personal shifts
[Full Schedule View]
Show all shifts with coworkers
Submit schedule change request
Store request
Confirm request storage
Notify schedule change request
Approve/Reject request
Update schedule if approved
Confirm update
Notify request status
Set employee schedule visibility
Update visibility settings
Confirm update
Visibility settings saved
Manager
Employee
System
Database

---

4. Implementation Summary

Technologies Used:

- **Programming Language**: Python 3.12
- **Database**: SQLite
- **Libraries**: Tkinter for GUI, Black for formatting, unittest for testing
- **Development Tools**: GitHub for version control, Makefile for build automation

Design Patterns Applied:

- **Observer**: Ensures UI updates reflect backend changes in schedules
- **Command**: Modularizes shift management actions (add, remove, modify)
- **Factory**: Creates users based on roles (Manager, Employee)
- **Strategy**: Allows scheduling to be generated using different algorithms
- **Singleton**: Guarantees one active connection to the database

Modules were organized in a modular directory structure including folders for GUI, models, scheduler logic, and database management.

---

5. Schedule Optimization: Strategy and Algorithm

One of the defining features of our application is its ability to generate optimized schedules based on manager-defined criteria. We implemented this using a **Strategy design pattern**, allowing different optimization approaches to be interchangeable within the system.

Currently, our primary strategy aims to **maximize shift coverage** while respecting employee availability. The algorithm proceeds in these general steps:

1. **Data Preparation:** Fetch all employees and their availability from the database.

2. **Slot Mapping:** Divide each day into 30-minute blocks and map each employee's availability to these blocks.

3. **Coverage Matching:** For each required shift slot, assign employees based on availability and constraint satisfaction.

4. **Constraint Handling:** If the system cannot fully meet coverage needs, it prioritizes critical times and raises a warning for manager review.

We designed the algorithm to be extensible in future versions, such as adding preferences for consecutive shifts, max hours, or even equity balancing across employees.

---

6. AI Tools and Contributions

We used AI in two ways:

1. **Mock Business Development**: Wwe used ChatGPT to develop a mock business to test our program with.
2. **Bug Identification and Testing Guidance**: ChatGPT helped debug integration issues and recommended best practices.

Limitations:

- AI tools required manual validation to ensure logic correctness.
- Code recommendations occasionally needed adaptation to our exact architecture.

---

7. Testing Strategy and Results

We implemented unit and integration tests to validate:

- Schedule creation
- Employee availability matching
- Conflict detection

Testing modules were written using Python's `unittest` framework and included coverage of edge cases (e.g., overlapping shifts, invalid time entries).

The Makefile facilitated testing and code formatting. We achieved partial coverage due to incomplete front-backend integration by the final report period, but key backend features were verified.

---

8. Ethical Considerations

Data Privacy and Role-Based Access

Although this is a local application, we designed it with security principles in mind. For instance, only managers can access scheduling tools and employee data, while regular employees are

restricted to their own views. We emphasized role-based access control even without password authentication, laying groundwork for future enhancements.

Fairness in Scheduling

We considered fairness as an ethical constraint during schedule generation. Although our optimization algorithm is basic in this iteration, the system is designed to accommodate rules that ensure balanced workloads and respect employee preferences, especially critical in avoiding biased or overly demanding shift assignments.

AI Ethics in Development

We also recognized the importance of transparency when using AI tools. Any AI-generated content was reviewed, documented, and adjusted by team members, reinforcing accountability in software engineering practices.

---

9. Detailed Testing Strategy

Unit Testing

Each core module (e.g., `ManagerPage`, `CreateSchedule`, `DatabaseHandler`) was tested in isolation to confirm correctness of functionality. We used Python's `unittest` framework and included tests for edge cases such as overlapping availability or invalid shift times.

Integration Testing

We focused integration tests on key workflows, such as creating a new employee and assigning them a shift. These tests ensured communication between the front-end UI, back-end logic, and database worked as intended.

User Testing and Feedback

We conducted informal user testing with peers acting as managers and employees. Feedback led to changes such as improving the calendar visualization and simplifying navigation within the UI.

---

10. Challenges and Lessons Learned

Key Challenges:

- Integration of backend logic with GUI components using Tkinter.
- Modularizing a rapidly expanding codebase.
- Coordinating team contributions efficiently.

Lessons Learned:

- Organizing code early with clear directories, diagrams, and design patterns prevents confusion.
- Frequent testing and continuous integration help reduce bugs.
- AI can assist but never fully replace software engineering judgment.
- Delaying implementation to prioritize system diagrams (UML, sequence diagrams) and architecture planning saved time and confusion during coding phases.
- Regular check-ins and issue tracking through GitHub helped coordinate efforts, avoid redundant work, and foster a collaborative environment.
- Enforcing branch workflows and pull request reviews improved code quality and prevented accidental overwrites or regressions.
- We initially considered more features but learned to cut back and prioritize core functionality to meet deadlines without compromising quality.

---

11 Implementation Challenges by Module

UI Module (Tkinter)

- **Challenge:** Tkinter's layout system has strict rules around using `pack` and `grid`, which caused rendering errors.

- **Solution:** We resolved layout issues by separating layout logic from content logic and using only `grid()` in all frames.

- **Challenge:** Dynamically updating the calendar to reflect real-time changes required using **Observer patterns** and callbacks.

- **Solution:** We implemented a shared data model that UI elements subscribed to, which pushed updates when the underlying data changed.

Database Module (SQLite)

- **Challenge:** Maintaining a single, shared connection across the application without race conditions.

- **Solution:** We implemented the **Singleton pattern** in our `DatabaseManager` class to control connection instantiation.

- **Challenge:** Ensuring time consistency across all stored and retrieved data.

- **Solution:** All timestamps were normalized to 24-hour time using a utility formatter and stored as strings with strict input validation.

---

12. Future Work and Expansion Opportunities

Mobile and Web Interfaces

Currently, our application is a desktop-based system. However, to further increase accessibility and convenience for both managers and employees, we plan to expand to web and mobile platforms in the future. This will require:

- Migrating the backend to a web framework like Flask or Django

- Replacing Tkinter with React or Flutter

- Integrating secure user authentication (with password hashing and sessions)

Advanced Scheduling Features

Some future features include:

- **Automated conflict resolution:** The system could suggest optimal time slots when conflicts arise.

- **Skill-based scheduling:** Match employees with tasks that require specific certifications or experiences.

- **Machine Learning-enhanced predictions:** Use past data to forecast optimal shift patterns.

Audit Trail and Change Tracking

An improvement we envision is implementing **audit logs**, where all schedule changes (additions, deletions, edits) are timestamped and stored for later review. This adds accountability and can help resolve disputes.

---

13. Conclusion

Our application delivers a functional, scalable scheduling solution tailored to small businesses. It enhances usability, reduces errors, and improves transparency for employees and managers alike. While not yet production-ready, the solid foundation allows for future expansion including mobile access, web deployment, and advanced analytics.

This project highlighted the power of software engineering principles, particularly modularity, abstraction, and pattern-based design, alongside the supportive but limited role of AI tools.
