Class Design Document

Class 1: FileManager

Purpose

Manages all JSON file read/write operations with built-in error handling and data validation.

Attributes

data_dir: Optional[str]	*Directory path for data files (None = current dir)*
students_file: str	*Full path to students.json*
resources_file: str	*Full path to resources.json*
transactions_file: str	*Full path to transactions.json*

Methods

__init__(students_file, resources_file, transactions_file, data_dir=None)
Initialize file manager and create missing files

load_students() -> List[Dict]
Load student data from JSON file

save_students(students: List[Dict])
Save student data to JSON file with atomic write

load_resources() -> List[Dict]
Load resource data from JSON file

save_resources(resources: List[Dict])
Save resource data to JSON file with atomic write

load_transactions() -> List[Dict]
Load transaction data from JSON file

save_transactions(transactions: List[Dict])
Save transaction data to JSON file with atomic write

Key Features

- Auto-creates missing files with empty array []
- Validates JSON structure on load
- Uses atomic writes (temp file → replace) to prevent corruption
- Handles empty files gracefully

Exception Classes

FileManagerError	*Base exception for file operations*
FileCorruptionError	*Raised when JSON is invalid or corrupted*

Class 2: RoleConfig

Purpose

Stores email domain configuration for automatic role assignment.

Attributes

student_domains: tuple	*Valid student email domains*
staff_domains: tuple	*Valid staff email domains*

Methods

__init__(student_domains=("student.campus.edu",),
    staff_domains=("campus.edu",))

Initialize with default or custom email domains

Class 3: SystemManager

Purpose

Core business logic class that manages all students, resources, and transactions while enforcing borrowing rules.

Attributes

file_manager: FileManager	*Instance for data persistence*
due_days: int	*Number of days until resource due (default: 3)*
role_config: RoleConfig	*Configuration for role determination*
students: List[Dict]	*In-memory list of student dictionaries*
resources: List[Dict]	*In-memory list of resource dictionaries*
transactions: List[Dict]	*In-memory list of transaction dictionaries*

Methods

Data Management

load_all()
Load all data from JSON files into memory

save_all()

Save all in-memory data to JSON files

Role Management

determine_role(email: str) -> str
Determine user role based on email domain
Returns: "student" or "staff"

Raises: ValidationError if email invalid or domain not recognized

Student Operations

add_student(student_id: str, name: str, email: str)

Add new student to system
Raises: ConflictError if student_id already exists
Raises: ValidationError if any field is empty

find_student(student_id: str) -> Optional[Dict]

Find student by ID

Returns: Student dictionary or None if not found

Resource Management

add_resource(resource_id: str, name: str, rtype: str, quantity: int)

Add new resource to system
Raises: ConflictError if resource_id already exists
Raises: ValidationError if invalid input

update_resource_quantity(resource_id: str, new_quantity: int)

Update resource quantity
Raises: NotFoundError if resource not found
Raises: ValidationError if quantity invalid

remove_resource(resource_id: str)

Remove resource from system
Raises: NotFoundError if resource not found
Raises: ConflictError if resource is currently borrowed

list_resources() -> List[Dict]

Return all resources

list_available_resources() -> List[Dict]

Return only resources with quantity > 0

find_resource(resource_id: str) -> Optional[Dict]

Find resource by ID

Returns: Resource dictionary or None if not found

Borrowing & Returning

borrow_resource(student_id: str, resource_id: str, borrow_date: Optional[str] = None) -> Dict

Process resource borrowing
Creates transaction, reduces quantity, sets due date
Returns: Transaction dictionary
Raises: NotFoundError if student or resource not found
Raises: ConflictError if resource unavailable or already borrowed

return_resource(transaction_id: str, return_date: Optional[str] = None) -> Dict

Process resource return by transaction ID
Updates transaction, increases quantity
Returns: Updated transaction dictionary
Raises: NotFoundError if transaction not found

Raises: ConflictError if already returned

return_resource_by_student_resource(student_id: str, resource_id: str,
return_date: Optional[str] = None) -> Dict

Process resource return by student and resource
Returns: Updated transaction dictionary
Raises: NotFoundError if no active transaction found

Raises: ConflictError if multiple active transactions

Exception Classes

SystemManagerError	*Base exception for system operations*
NotFoundError	*Entity (student, resource, transaction) not found*
ValidationError	*Invalid input data*
ConflictError	*Operation not allowed (e.g., duplicate ID, unavailable resource)*

Data Storage Format

Student Dictionary

```
{
  "student_id": "S001",
  "name": "Princess Addai",
  "email": "P.addai@alustudent.com"
}
```

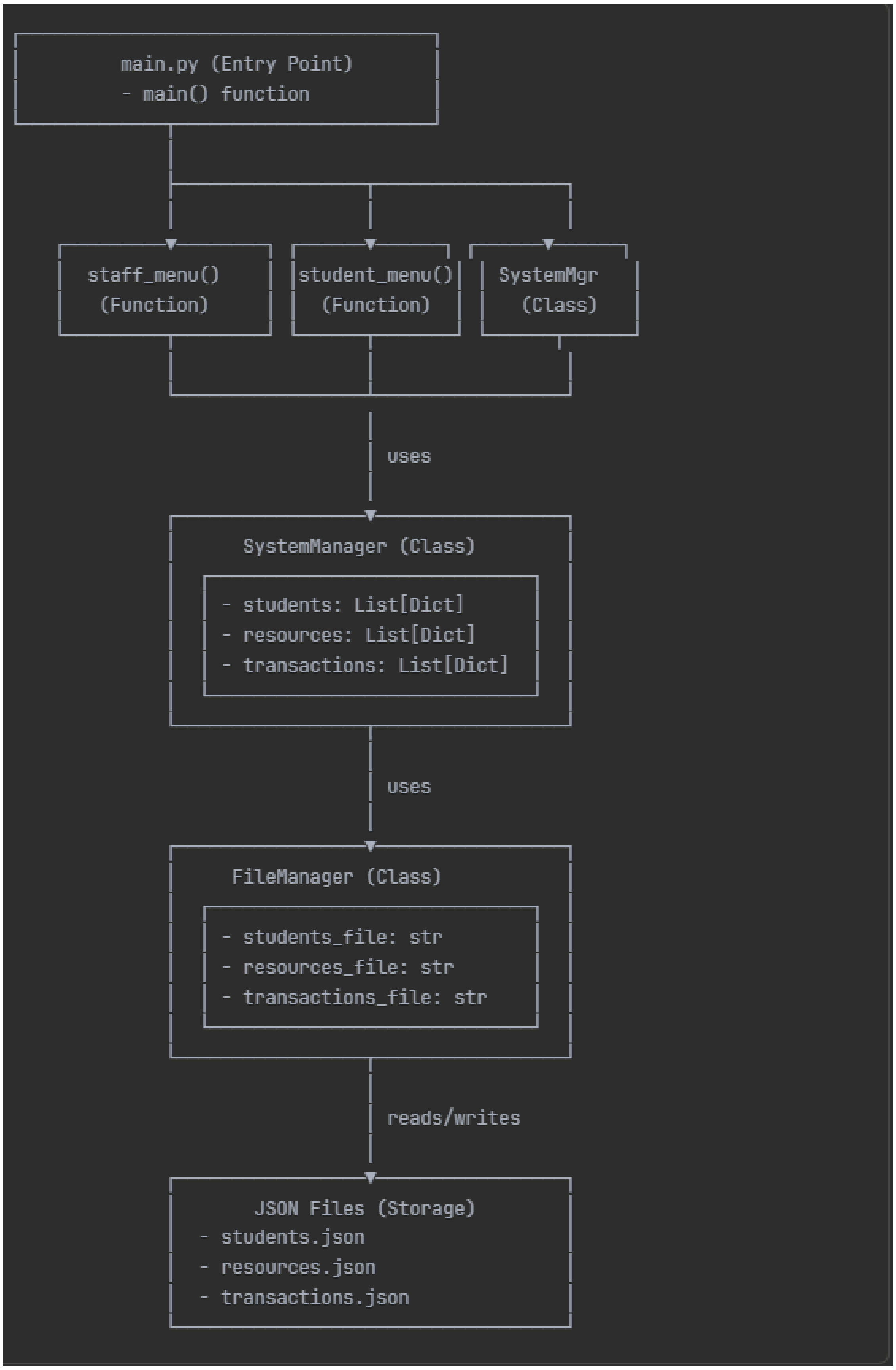
Resource Dictionary

```
{
  "resource_id": "L001",
  "name": "Dell Laptop",
  "type": "IT Equipment",
  "quantity": 5
}
```

Transaction Dictionary

```
{
  "transaction_id": "T001",
  "student_id": "S001",
  "resource_id": "L001",
  "borrow_date": "2025-01-05",
  "due_date": "2025-01-08",
  "return_date": null,
  "status": "borrowed"
}
```

SECTION 2: CLASS STRUCTURE DIAGRAM



Section 3. Sample Test Plan

The following test scenarios cover **normal**, **edge**, and **invalid** cases. Each class has at least five test cases.

3.1 Student Class – Test Scenarios

1. **Create a student with valid ID, name, and email** → student object created successfully.
2. **Create a student account with empty name** → system handles or rejects invalid data.
3. **Create two students accounts with the same ID** → system prevents duplication.
4. **Display student details** → correct information is shown.
5. **Load student data from file** → student objects created correctly.

3.2 Resource Class – Test Scenarios

1. **Add a resource with quantity > 0** → resource is available.
2. **Borrow a resource when quantity = 1** → quantity decreases to 0.
3. **Borrow a resource when quantity = 0** → borrowing is denied.
4. **Return a resource** → quantity increases by 1.
5. **Add a resource with a negative quantity** → system rejects invalid input.
6. **Add different types of resources** (e.g., projector, lab kit) → system handles all types correctly.

3.3 BorrowTransaction Class – Test Scenarios

1. **Create a transaction with valid borrow and due dates**, and set the status to "borrowed".
2. **Return a resource before due date** → status changes to "returned".
3. **Return a resource after the due date** → status marked as "overdue".
4. **Check overdue status when the current date exceeds the due date** → overdue detected.
5. **Attempt to return an already returned resource** → system prevents duplicate return.

6. **Borrow multiple resources by the same student** → transactions recorded correctly.

3.4 FileManager Class – Test Scenarios

1. **Load data from existing JSON files** → data loaded successfully.
2. **Save updated resource data** → changes persist after restart.
3. **Load from an empty JSON file** → system handles gracefully.
4. **Load from missing file** → system creates a new file.
5. **Save invalid data format** → system prevents file corruption.
6. **Load and save multiple resource types** → ensures all resource types are correctly handled.

3.5 SystemManager Class – Test Scenarios

1. **Borrow a resource with a valid student and an available resource** → borrowing successful.
2. **Borrow a resource with an invalid student ID** → borrowing denied.
3. **Return a borrowed resource** → transaction updated correctly.
4. **Return a resource that was not borrowed** → system shows error.
5. **View available resources** → system lists all resources with quantity > 0.
6. **View overdue resources** → system lists all overdue transactions.
7. **Borrow and return different resource types** → system correctly handles books, projectors, lab kits, etc.