Software Design Documentation

Professor Lushan Shaun Gao
CP317 Software Engineering
Group Project
Friday, November 24, 2023

Group 5:

Lashani Dharmasena (212378870)
Deergh Gandhi (210629500)
Prabhjot Chahal (200912570)
Nicole Jeong (200358950)

1. Introduction

1.1 Background

The **Text File Formatter App** is an intricate solution created to carefully read and format two different text files with the goal of producing a well formatted output, in response to the growing need for more efficient text file processing. The app, which was developed in Java, is a prime example of modern software engineering. It is focused on the ideas of object-oriented programming (OOP) and emphasizes encapsulation, polymorphism, and abstraction. With its operation in the Windows environment, the Text File Formatter App is prepared to meet the demands for a powerful yet clear application that will reduce the challenges involved in manipulating text files.

1.2 Objectives

Collaboratively managing diverse text files can be difficult for both individuals and organizations, which can result in errors in interpretation of data and inefficiencies. In order to handle the technical complexities of file manipulation as well as to improve user experience through intuitive design and encapsulated functionality, the following Text File Formatter App was created.

The app accomplishes this by offering a seamless and feature-rich platform. Through increasing extensibility and maintainability, the application ensures a modular and scalable design. The encapsulation principle, a cornerstone of software architecture, safeguards users from needless complexity while ensuring a safe and effective processing environment.

1.3 Scope

The scope of this app includes the following functionalities:

- File reading - the applications reads two input text files of various file sizes and formats
- Formatting logic - the app will convert unstructured text data into a readable structured format by implementing a given formatting logic into practice.
- Output generation - following the processing, an updated text file with formatting will be created, maintaining the original data integrity.

While the app itself is a comprehensive tool, it is essential to acknowledge certain limitations:

- File types - the app is designed specifically for handling text files and does not support proprietary formats or binary data.
- Real-time processing - the app does not offer real-time processing; it operates on a file-by-file basis.

- Complex formatting - although the program guarantees basic formatting, very intricate formatting needs are outside its scope.

To manage expectations, it is essential to identify what the app will not undertake:

- Multimedia handling - no multimedia files will be altered by the application.
- Advanced data analysis - the program is not meant for complicated data analysis activities other than simple formatting.
- GUI

We recognize that there may be more updates in the future and that the modular architecture of the program makes it possible to easily incorporate new features and upgrades as needed.

1.4 Target environment

Our target environment for the Text File Formatter App will be a fully integrated, single-threaded program with a component based architecture. Users will have a single, convenient location to download and use the application as the source code and related classes will be housed in a dedicated Github repository. The app can be used locally by users by cloning the repository and running the App.java file using the java command. The environment's classes will be linked together to provide effective communication modules, guaranteeing a responsive and seamless user experience. Our platform also encourages cooperation and extensibility allowing users to improve or modify the application to meet their particular needs by leveraging and expanding our codebase. Users must download the required APIs in order to integrate and operate the Text File Formatter App without any issues. The project's GitHub repository has comprehensive installation and usage instructions that make the process simple to follow.

2. System Architecture

2.1 High-Level Architecture

The applications architecture is created as a system of components. The simplicity and efficiency of this strategy contribute to quicker development, implementation, and production and delivery timelines. Since the requirements are expected to remain constant, the application is not meant to be extremely flexible or scalable, which fits perfectly with the component-based approach.

Advantages of chosen architecture:

1. Functionality decomposition: Functionality of the program is divided up into discrete parts, encouraging the separation of issues and assisting in the efficient management of dependencies.
2. Ease of management: Because interfaces and boundaries between components are well defined.
3. Streamlined development: Ability to work on several system components in parallel due to the component based design speeds up the development process as a whole.
4. Design simplicity: Simplifies the system's complexity by allowing for a simple design in which each component has a simple function.

Although a monolithic design was considered, due to components of this structure being closely connected, it may be more difficult to isolate and fix problems. Since a modular strategy where each component may be built, tested, and maintained independently fits the needs and limitations of the project, the following architecture was used.

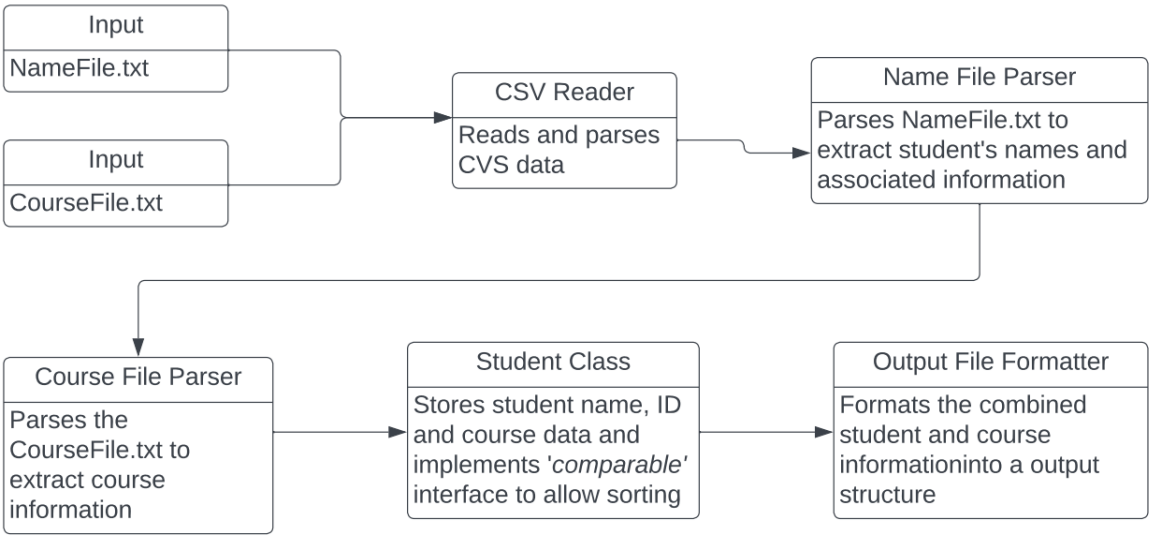
2.2 Component Design

Components of the architecture:

1. CSV reader - responsible for reading and parsing CSV data, acts as the entry point for the data processing flow.
2. Name file parser - parses the name file to extracts student names and associated information, works along with the CSV reader for data extraction
3. Course file parser - parses the course file to extract course information, supplies necessary course details to be associated with student records.
4. Student class - stores student information including name, ID and course data, implements the '*comparable*' interface to allow for sorting if needed.
5. Output file formatter - formats the combined student and course information into a specified output structure, manages the creation of the output file ensuring the data is presented clearly and correctly.

2.3 Data Flow

- Data Flow Diagram



3. Class Design

1. Student class (Implemented comparable)
    - Attributes:
        - '*name'* :string - stores the name of the student
        - *'id'* :int - a unique identifier for the student
        - '*Course'* :hashmap<string, float> - a collection that maps course names to scores or percentages
    - Methods:
        - '*compareTo*'
2. App (Main class)
    - Attributes:
        - '*Students*' :hashmap<integer, student> - stores all student objects with their ID as the key
    - Methods:
        - '*main()*' :void - entry point of the application that orchestrates the reading, parsing, and formatting process
    - Purpose: the main driver of the application, managing the flow of data and initialization of the process
3. File parser (Parent class)
    - Methods:
        - '*parseFile()*' :void - abstract or implemented method to parse a given file
    - Purpose: serves as a base class for file parsing operations

4. Name file (Child of File Parser)
    - Attributes: Inherited from '*FileParser*'
    - Methods: Inherits or overrides '*parseFile*()' from '*FileParser*'
    - Purpose: Specializes in parsing files that contain course information
5. Course file (Child of File Parser)
    - Attributes: Inherited from '*FileParser*'
    - Methods: Inherits or overrides '*parseFile*()' from '*FileParser*'
    - Purpose: Specializes in parsing files that contain course information
6. Output file formatter
    - Attributes:
        - '*Filename*' :PrintStream - the output stream to write the formatted data to a file
    - Methods:
        - '*write*()' :void - writes the formatted data to the output file
    - Purpose: Formats and writes the combined data of students and their courses to an output file

The concepts of encapsulation, polymorphism and abstraction are implemented in the app as follows:

Encapsulation

The idea of bundling data with methods that operate on that data is put into practice by limiting access to parts of an object's components. The attributes are made private and public getter and setter methods are made available to access and modify the values of a private variable to do this.

For example:

- Attributes like '*name*', '*id*', and '*courses*' are private to prevent external direct access
- Methods are public to access and modify the attributes safely

Polymorphism

Objects can be treated as instances of their parent class rather than their own class through polymorphism. This is done by implementing an interface and overriding methods.

For example:

- '*FileParser*' class be an abstract class with a method '*parseFile*()' that is overridden by its subclasses '*NameFile*' and '*CourseFile*'
- *'Comparable'* interface implemented by the '*Student*' class, allowing different implementation of the '*compareTo*' method

Abstraction

The idea of abstraction is to reveal only the elements that are essential while concealing the complex sections.

In the app:

- '*FileParser*' class is an example. It offers a generalized form that its subclasses further define. Just the '*parseFile*' method, which must be implemented by its subclasses is exposed; all other file parsing details are hidden
- Interfaces like '*Comparable*' are also used to abstract the method signatures that the implementing classes need to adhere to.

4. Workflow

The workflow involves several classes working together to read, parse, and format text data. A step-by-step outline of this process provided here:

Step 1: Initialize Application

- The program begins by executing the main method of the '*App*' class.
- In order to hold student data, it initializes the relevant parts and data structures, such as the students HashMap.

Step 2: Read Input Files

- **'***CSV Reader***'** is called to read the raw data from the CSV file.
- The '*NameFileParser*' reads and extracts student names and IDs if there is a Name File.

Step 3: Parse Data

- '*Course File Parser*' a subclass of '*FileParser*', reads and manipulates course-specific data from an external file.
- Both parsers insert their individual bits of data into the intermediate data structure.

Step 4: Combine Data

- Using the information from earlier phases, student objects are instantiated using the '*Student*' class.
- Every '*Student*' object would have its '*name*', '*id*', and '*courses*' (with relevant scores) filled in.

Step 5: Format Data

- The compiled student data is passed into the **'***Output File Formatter***',** which formats it in accordance with the output specifications.
- It would format the representation using the data found in the **'***Student***'** objects.

Step 6: Write to Output File

- The '*Output File Formatter*' is responsible for managing the output file where the prepared data is subsequently written.
- All I/O activities would be handled by the '*Output File Formatter*', guaranteeing accurate data writing to the file system.

Step 7: Error Handling

- Any class might have exceptions or problems during execution, which should be reported and notified to the user using an **'***ExceptionHandler***'** class or error handling inside each class.

Step 8: Close

- Once all operations are finished, the application guarantees that all resources are released, such as shutting file streams.
- The program then closes, giving the operating system back control.

Class Collaboration:

1. '*App'* class: Initializes the overall process from start to calling other components.
2. *'CSV Reader'* and *'FileParser'* Subclasses (*'NameFileParser'* and *'CourseFileParser'*): Read and parse input files and pass the data to the *'Student'* objects.
3. *'Student'* class: Serves as a placeholder for individual student data.
4. *'Output File Formatter'*: Uses the data within *'Student'* to create a formatted output and writes it to a file.
5. *'ExceptionHandler*': Used to handle exceptions in a consistent manner.

5. Data Management

Reading Data

Input Acquisition: The program must first locate and launch the required text files. To accomplish this, the user is prompted for file locations or names upon startup.

File Reading: The '*CSV Reader'* reads the raw data from the CSV file by using file I/O operations made available by the java API. An extension of the '*FileParser*' class, '*NameFileParser*' would open and read the name file line by line.

Data parsing: It is the process of transforming raw data into a format that may be used. To do so, the text data must usually be converted into more organized data types like strings, integers, or floats. In addition, parsing also involves translating strings to other data types.

Compiling and Organizing Information

Data Storage: The data is momentarily kept in memory following parsing. For every student record, the '*Student*' class would be constructed with its field (name, id, etc.) populated with the parsed data.

Data Transformation: The '*Student*' class or a specific method within the '*App*' class would be used to aggregate, calculate, or transform any data that needs to be done (calculating total or average scores).

Data Formatting: The desired formatting is applied to the structured student data by the '*Output File Formatter*'.

File Writing: After formatting, the data needs to be written to an output file. The '*Output File Formatter*' does this by creating a file stream that points to the intended output file path.

Resource Management: To prevent leaks and guarantee that the data is saved correctly, file handles and streams need to be closed

6. Testing Strategy

10.1 Unit Testing

Involves verifying the smallest testable parts of the application in isolation. This was done by:

1. CSV reader tests - Make sure the CSV Reader can handle a variety of data types, read a variety of CSV files, and handle issues such as corrupted or missing files.
2. Tests for NameFileParser - Check for situations with odd characters or blank lines.
3. Tests for CourseFileParser
4. Student class test - Check that data is properly stored and retrieved.
5. Output file formatter -

10.2 Integration Testing

1. The purpose of integration testing is to highlight the testing/validation between multiple components in the code.
2. The scope of the integration testing is broader than unit testing and used to compare multiple functions of the code and how they interact together.
3. Dependencies are another area of testing where we delve into recognizing a problem that could arise and how it would affect the different elements of the code.
4. Integration testing was considered as FileParser communicates with the App class. This is done to ensure that the information from the Students class is correct and when parsing the file, all the authenticated information is stored and will be output.

7. Error Handling

Error handling plays a crucial role in ensuring the reliability and resilience of applications.This section delves into the error handling strategies employed by this application. Each file contributes to the comprehensive handling of potential errors during file parsing, input validation, and data formatting.
App.java:

- Catches a generic Exception for any unexpected errors during the execution of the main method.
- Prints the stack trace for debugging purposes.
- Exits the program with a status code of -1

FilePaser.java:

- Constructor
    - Throws a FileNotFoundException if the specified file doesn't exist.
    - Provides an error message indicating that the specified file doesn't exist.
- readNameFile Method
    - Throws a generic Exception for various parsing-related issues.
    - Provides specific error messages indicating the nature of the parsing issue.
- readCourseFile Method:
    - Throws a generic Exception for various parsing-related issues.
    - Provides specific error messages indicating the nature of the parsing issue.

OutputFileFormatter.java:

- Constructor
    - Constructor throws a FileNotFoundException if there is an issue creating the output file.
    - Does not provide a specific error message when handling FileNotFoundException.
- formatStudent Method
    - Catches a generic Exception when attempting to get grades for a course. It prints a message but does not propagate the exception.
    - Prints a message indicating that the course code doesn't exist.

Student.java:

- addCourse Method
    - Throws a generic Exception for various issues during the addition of a course, such as incorrect number of grades or invalid grades.
    - Provides specific error messages for different scenarios, such as incorrect number of grades or invalid grades.
- getGrades Method
    - Throws a generic Exception if the specified course code doesn't exist for the student.
    - Provides a specific error message indicating that the course code doesn't exist.

The error handling mechanisms implemented in the application can mitigate many potential issues during file parsing and data processing. In order to respond quickly to unexpected situations, the use of exception management ensures that try catch blocks in particular are employed. The program uses the FileNotFoundException to deal with missing input files in order to perform a thorough search for file existence. Custom exception messages are used to deal with parsing errors, such as incorrect field counts or nonnumeric identifiers, enabling developers and users to gain clear insight into the nature of encountered problems. The proactive approach to error detection and correction is reflected in this application's handling of errors, as well as the code demonstrating a robust strategy. In these strategies, we can make it easier for developers and users to understand encountered problems so that they are more effective in their debugging and troubleshooting during the ongoing development lifecycle.

8. Deployment Plan

1. Pre-deployment:

- Describe the application in detail for the performance.
- Establish a secure configuration file system and the production environment.
- Create automation systems for deployment and backup.

2. Deployment:

- Prepare the application by packaging it and making a release candidate.
- Test in a staging environment, with the option to migrate data if necessary.
- Engage in user training for upcoming features.

3. Launch:

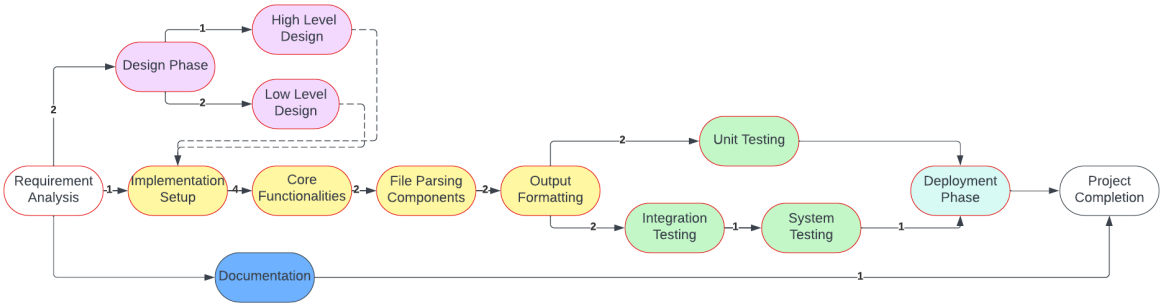
- Install in the live environment.
- Put monitoring into action and carry out smoke tests.
- For new features that are riskier, use feature toggles.

4. Post-deployment:

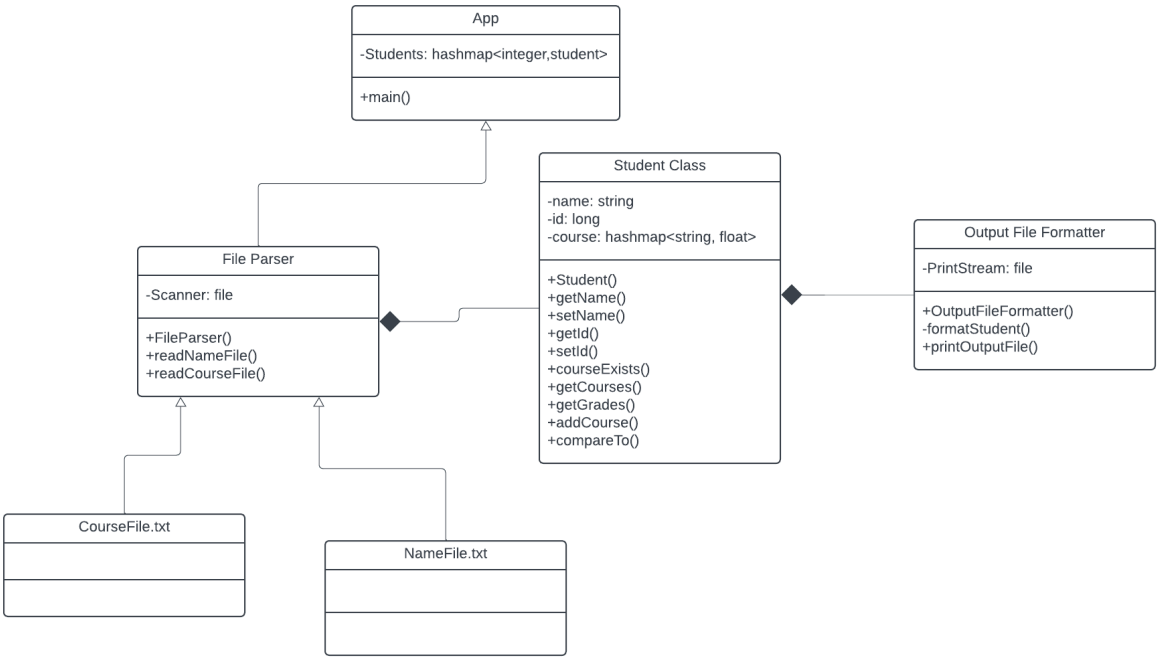
- Inspect for mistakes and keep an eye on the program.
- Notify users of the deployment.
- Make sure you have a backup and support strategy in place.

9. Appendix

- PERT chart
  - Solid arrows: dependencies
  - Dotted arrows: dependencies without resources
  - Red outline around the node means it is part of the critical path



- UML class diagram



10. Conclusion

Key points summary:

1. Application architecture:
    - Explains a component-based framework that encourages modularity and simple upkeep.
    - Explains the functions and relationships between particular parts, such as the output formatter, file parsers, and CV reader.
2. Design principles include:
    - Emphasizes the value of object-oriented concepts like abstraction, polymorphism, and encapsulation.
    - Strives for a design that is both scalable and manageable.
3. Testing Approach:
    - Describes a thorough testing plan that includes system, integration, and unit testing.
    - Uses thorough testing to guarantee the application's robustness and dependability.
4. Deployment Strategy:
    - Describes the steps involved in preparing for, launching, and verifying the deployment, in that order.
    - Guarantees a flawless and error-free application rollout.
5. Maintenance & Upkeep:
    - Offers a schedule for continuous support, frequent updates, and help for users.

- Guarantees that the application will continue to be useful, safe, and functional throughout time.

6. Security and Performance:
    - Draws attention to possible security flaws and suggests solutions.
    - Goes over optimization strategies and performance standards.
7. Change management:
    - Explains how to record changes and amendments to the Software Design Document (SDD).
    - Ensures traceability and transparency throughout the development of the application.

Importance of the SDD:

1. Development Roadmap: Functions as a guiding map for the development process, ensuring that all team members are on the same page regarding the project's objectives and approaches.
2. Risk Handling: By identifying potential risks and their solutions, it aids in proactive risk management, which is vital for upholding the app's integrity and security.
3. Quality Assurance: The comprehensive testing strategies guarantee that the app adheres to quality standards and operates as intended.
4. Enhanced Communication: Serves as a point of reference for developers, testers, project managers, and stakeholders, improving communication and comprehension among teams.
5. Documentation for Future Reference: Crucial for new team members to comprehend the app and invaluable for future maintenance and updates.

11. References

Lucidchart. (n.d.).
https://www.lucidchart.com/pages/landing?utm_source=google&utm_medium=cpc&utm_campaign=_chart_en_us_mixed_search_brand_exact_&km_CPC_CampaignId=1457964857&km_CPC_AdGroupID=57044764032&km_CPC_Keyword=lucidchart&km_CPC_MatchType=e&km_CPC_ExtensionID=&km_CPC_Network=g&km_CPC_AdPosition=&km_CPC_Creative=442433231228&km_CPC_TargetID=kwd-335119361 69&km_CPC_Country=9001025&km_CPC_Device=c&km_CPC_placement=&km_CPC_target=&gad_source=1&gclid=CjwKCAiAsIGrBhAAEiwAEzMlC86PR5rGGGevEnd-Y2aC-zS4n7SIGaoCmtH5e4xgyv_e4oak9qFY1BoCYL0QAvD_BwE

Lucidchart. (n.d.). *UML Class Diagram Tutorial*. Lucidchart. https://www.lucidchart.com/pages/uml-class-diagram

Team Asana. (2023, July). *What is a PERT chart? Plus, how to create one (with examples)*. Asana. https://asana.com/resources/pert-chart

Lucidchart. (n.d.). *What is a data flow diagram*. Lucidchart. https://www.lucidchart.com/pages/data-flow-diagram