Design Plan

Generalised Design Plan

This is a generalised, phased plan for the development of the automarker, designed to be relatively foolproof to follow. (Note: This plan is iterative development not waterfall)

Raw Data Gathering - Phase 1

Unzip the folders into an organised directory structure:

```
submissions/

    student1dir/

    student2dir/
```

- Read the data and contents of each directory.

Handling Submission Data - Phase 2

- We will process one submission folder at a time.
- The data inside the folder/submission will be read then graded to conform to pre-defined data schema.

Grading/Marking The submission - Phase 3

- This phase is self-explanatory. To simplify, it consists of reading, processing, and grading the Java files within a given folder/directory.
- Finally, a single output object data storage structure will encapsulate the results.

Generate PDF - Phase 4

- Once the data conforms to the set schema, we will generate a PDF document using the data storage structure for grades/marks.4

Repeat - Phase *

- This process will be repeated for all submissions in the directory.

Phase 1 - Reading Submission :)

Returns: A list of file paths corresponding to each student's unzipped submission or just pointing to the directory but the first option probably better

Obtain the Paths:
Retrieve the paths of all the unzipped files as a `List<String>`.

Unzip the Files:
Unzip each file from the list.

Traverse and Gather Paths:
Traverse the unzipped directories and collect the file paths corresponding to each unzipped submission into a `List<String>`.

Corresponding of the structure such that

```
submissions/

    student1dir/

    student2dir/
```

Implementation

The SubmissionManager oversees the extraction process, employing a ZipFileExtractor instance for file handling, following the Strategy pattern.

It begins by creating an output directory, then retrieves and validates the input stream of the submission's zip file.

Upon successful validation, it extracts the zip entries and organises them into directories for each student.

The organizeStudentSubmissions method traverses the unzipped content, creating a directory for each student and populating it with their respective files.

The ZipFileExtractor class manages the extraction of zip entries and files, utilising buffered streams for efficient writing.

Phase 2,3 - Processing submissions

Each submission's data will be stored in a `totalAssignmentGrades` object. This object will contain four sub-objects for each class, representing their specific grading breakdowns as follows:

- `ChatBot` accounts for 36 marks
- `ChatBotPlatform` accounts for 20 marks
- `ChatBotGenerator` accounts for 7 marks
- `ChatBotSimulation` accounts for 12 marks

(These breakdowns follow the rubric provided.)

Next, for each submission, grades will be retrieved using `GradeManager.getGrade(submissionLocation)`.

Each sub-object, for example **ChatBotGrade**, will include an `allocateWeightings` method. This method assigns a test name and its corresponding weighting. For instance, the autograder will test the chatbot's name and assign a passing grade of 1:

i.e. **ChatBotTest.put(***"chatBotName", 1***);**

This call refers to the **chatBotName** test within the **ChatBotGrade** class. Grades are then collated for each class via **AssignmentGrade**.

Development WorkFlow:

1. Write tests for each of the four main classes (**ChatBot**, **ChatBotPlatform**, etc) ensuring they include test suites that match the grading rubric.
2. Create a class that combines these four test classes and runs them all, returning a *List<Result>* from JUnit.
3.

Developing the **Grade** Interface: The **Grade** Interface can now be implemented with methods like **getMarks()** and **getTotalMarks()**. Via a template class **GradeTemplate** for code deduplication.

Concrete subclasses such as **ChatBotGrade**, will use JUnit test results, apply the corresponding weightings, and interpret the results to calculate the marks.

These grading subtypes (subclasses) will then be stored in an **AssignmentGrade** data model class, which will map criteria to the marks obtained, e.g, **Map<Criteria, Mark>**.

Phase 4 - Generate PDF

`StudentDetails` data class to store student information - name and ID

`AssignmentGrade` which has the marks obtained for each class (`ChatBot`, `ChatBotPlatform`, `ChatBotGenerator`, `ChatBotSimulation`)

`AssignmentDetails` class consists of `StudentDetails` and `AssignmentGrade` and contains a `toPDFData()` method which returns the necessary data for PDF generation in a map, `Map<String,String>()`.

A `PDF` interface is implemented which contains the method,

```
generate(AssignmentDetails)
```

The `PDFGenerator` class implements the `PDF` interface. The `generate(AssignmentDetails)` method fills out the fields of the PDF using the `populateFields(pdfData, pdfFields)` method.

The `GradeManager` class implements a method `GeneratePDFGrade()` which accepts the `AssignmentDetails` object and generates a PDF,

```
PDFGenerator.generate(AssignmentDetails)
```

The generated PDF should be in the same directory as the student's java files.

Testing:

Ensure that the `PDFGenerator` functionality is covered by unit tests. These tests should validate:

- Correctness of populated fields in the generated PDF.
- Handling of missing or incorrect data in `AssignmentDetails`.
- The file path consistency for saving PDFs.

Phase 5 - Refactor

The refactoring phase aimed to improve code maintainability, enhance testing capabilities, and ensure modular design. Key changes and improvements include:

1. **Test Refactoring**:
    - Refactored tests for **ChatBot**, **ChatBotPlatform**, and **ChatBotSimulation** to improve clarity and maintainability.
    - Added **ConfigLoaderStrategyTest** to ensure configuration loading strategies are working as expected.
2. **Code Organization**:
    - Moved **SubmissionManager** to a more modular structure by placing it within a processor and a helper package.
    - Updated folder structure and paths to align with project organisation standards, enhancing readability and ease of navigation.
3. **Data Handling Improvements**:
    - Utilised **BufferedReader** for loading YAML data, which streamlines the data loading process and handles exceptions more gracefully.
    - Introduced **TextProcessor** in the utils package for lenient string comparisons, improving data handling flexibility.
4. **Design Patterns and Constants**:
    - Refactored the submission process to use the Command Pattern for compilation, encapsulating the compilation logic and allowing for easier expansion.
    - Added constants to **Grade** classes to standardise and centralise the usage of values across the grading logic.
5. **Facade Pattern and Error Handling**:
    - Introduced a facade to simplify interactions with complex subsystems, reducing dependencies across components.
    - Fixed errors related to zipping submissions and patched issues with **GradeConfigLoader**, enhancing robustness in file handling.
6. **AssignmentGrade Refactoring**:
    - Refactored **AssignmentGrade** and added a toString method to improve debugging and logging capabilities.

Testing

The refactoring phase also involved thorough testing to ensure no functionality was lost and new structures were working as intended. All new modules and design pattern implementations were verified for correct integration.

Phase 6 - Feedback Refactor

The purpose of this phase was to overhaul the feedback system, making it more modular, comprehensive, and maintainable. Key changes and improvements include:

1. **New Feedback Implementation**:
    - Introduced a redesigned feedback system to provide clearer, more detailed feedback to users.
    - Modularized feedback logic to enable easier modifications and future enhancements, making it adaptable to various grading scenarios and student needs.
2. **YAML Structure Optimization**:
    - Flattened the YAML configuration structure, simplifying data access and reducing nested data complexity.
    - Updated the factory pattern to align with the new YAML structure, ensuring compatibility and consistency across data sources.
3. **Improved Data Loading**:
    - Enhanced the **MarkschemeLoader** to handle grading configurations more reliably, fixing previous issues with data parsing and loading.
    - Updated **AssignmentGrade** to integrate with the new feedback structure, ensuring accurate feedback generation based on grading results.
4. **Dependency Removal**:
    - Removed the **SnakeYAML** dependency, optimising the project's dependency footprint and streamlining YAML processing.
5. **Constants and General Clean-up**:
    - Updated constants related to feedback to centralise values and improve maintainability.
    - Refactored code in the feedback system to improve readability and ensure consistency.

Testing

Phase 6 included thorough testing of the feedback system to validate that it produced accurate and clear output. Tests covered the new feedback logic, YAML configuration compatibility, and data loading reliability to ensure robust performance and usability.

Test Phase 1 - Unit and Performance Testing

This phase was focused on establishing a robust testing framework, covering unit and performance tests to ensure system reliability and efficiency. Key components tested include submission processing, PDF generation, grading, and system configuration.

1. **Performance Testing**:
    - Added performance tests for various modules, including **TestRunner**, **PDF generation**, **Facade**, **ConfigLoader**, and **SubmissionExtractor**.
    - Implemented **CompileCommandPerformanceTest** and **GradePerformanceTest** to measure the execution efficiency of compilation and grading processes.
    - Introduced **PerformanceChecker** to wrap system tests and monitor performance metrics across different components.
2. **Unit Testing**:
    - Developed unit tests focused on interfaces rather than specific implementations to ensure that all components adhere to expected behaviours and maintain flexibility for future changes.
    - Tests for interfaces like `Grade`, `GradeFactory`, and `Compile & Facade` validate functionality at a high level, ensuring that any implementation of these interfaces will conform to required standards.
    - This approach enhances modularity and future-proofing by focusing on contract-based testing rather than implementation-specific details.
3. **Testing Constants and Cleanup**:
    - Introduced constants for testing to maintain consistency across test cases, improving readability and ease of updates.
    - Removed outdated or redundant tests to ensure the test suite remains concise and effective.
4. **Bug Fixes and Refinements**:
    - Addressed issues identified during testing, including a bug in the performance test runner and minor errors in the submission processor.
    - Standardised naming conventions and terminology across tests to ensure clarity and consistency.

Test Phase 1 laid the foundation for a reliable testing framework by implementing both unit and performance tests for key system modules. By focusing on interface-based unit tests, this phase ensures that the system's core functionalities are well-defined and flexible, supporting future modifications and varying implementations.

Test Phase 2 - Additional Unit and Functional Testing

This phase expanded the testing framework by introducing new unit and functional tests, refining the existing tests, and adding sample data for enhanced test coverage.

1. **Expanded Unit Testing**:
    - Added **ConfigLoaderStrategyUnitTest** to verify the loading strategy configurations, ensuring robust handling of different configurations.
    - Enhanced unit tests by adding coverage for the PDF generation process and updating existing tests to accommodate recent changes.
    - Focused on testing interfaces rather than specific implementations, maintaining a flexible and modular approach to unit testing.
2. **Functional Testing and Refinements**:
    - Introduced sample data to support more realistic and varied test cases, enabling tests that simulate actual user interactions and data processing scenarios.
    - Refined test names and reorganised test files for clarity, such as renaming testGenerateResponse_FieldIsPrivate for improved readability.
    - Made minor adjustments to test structures and logic, fixing issues discovered in Test Phase 1 and ensuring that test cases align with the current system configuration.
3. **Bug Fixes and Code Maintenance**:
    - Addressed minor bugs identified through testing, improving test stability and accuracy.
    - Regular maintenance, including renaming files and updating test descriptions, to ensure the test suite remains well-organised and comprehensible.

Video Plan

1. Team Members
2. Project Scope
3. Use Cases
4. System Demo
5. System Architecture
6. Core Entities
7. Grading Algorithm
8. Data Processing Strategies
9. Test Suite P1
10. Test Suite P2 (Brief)
11. Documentation

Roles:
Keshan - 1,2,3,11
Dmitri - 5,6
Felicia 6
Ando 4, 9,10
Brandon - 8

Core Classes:

- TestRunner
- Grade
- Factory
- PDF
- Facade
- Submission
- SubmissionExtractor (calls ZipFileExtractor)
- AssignmentDetails
- ~~AssignmentGrade~~
- ~~StudentDetails~~

Dmitri (Lead/Dev)

Good day, I'm Dmitri, the Project Lead. I primarily worked on core system development and implemented the performance testing microframework. Additionally, I was responsible for code reviews, with assistance from Felicia.

To begin, the system was written primarily in a layered architecture divided into Structures, Services, and Interface layers. Where structures correspond to real world domain entities, services represent the "business/grading logic" and Interfaces represent the abstraction/contract layer.

Moving into application design

The **TestRunner** interface defines a contract for running tests, returning results as a generic `List`. In this project, the `AssignmentTestRunner` specified JUnit's Result type for the `TestRunner`.

The **AssignmentTestRunner** runs a predefined list of test classes (`testClasses`) using `JUnitCore` and returns their results. Each indexed Result corresponds to the outcome of a specific indexed test class. Following JUnit's documentation, test execution logic is encapsulated in a private method.

The **Grade** interface manages assignment scoring data, including marks obtained, total marks, and feedback; the data can be encapsulated in any manner once it abides by the interface.

The `Grade` is implemented using an intermediate abstract `GradeTemplate` pattern, which enables specialisation for various assignment components while maintaining shared behaviours like mark processing and feedback handling. Each subtype of the template, such as `ChatBotGrade`, defines how marks and feedback are assigned based on results from the `TestRunner`.

The **Factory** interface standardised item creation. The **GradeFactory** uses a jump table to map indexes to grade types, enabling dynamic grade instantiation. The **AbstractGradeFactory** implements `createItems` to process a list of results into a list of grades, enforcing the `Grade` type.

Facade

The Facade interface acts as the core of the system which specifies the processSubmissions method. This method is implemented by the GradingFacade class to provide a streamlined interface for submission processing. It handles the workflow, including extracting submission files, resetting the Submission singleton instance, grading submissions and generating reports.

SubmissionExtractor (calls ZipFileExtractor)

The GradingFacade uses the SubmissionExtractor to process the provided zip file. The SubmissionExtractor uses a resource loader and the ZipFileExtractor to extract the zip file and organise student submissions into individual directories using the SubmissionOrganizer. This process returns a list of paths to all the extracted student submissions.

Submission

Each submission is represented by the Submission singleton which is initialised with a path provided by the SubmissionExtractor. It stores a submission's compilation status and includes a mechanism for dynamically loading classes directly from the submission for evaluation.

AssignmentDetails (AssignmentGrade, StudentDetails)

Once a submission is processed, the GradingFacade creates an AssignmentDetails object. This class models an assignment including StudentDetails which stores student information such as their student ID and name as well as the AssignmentGrade which stores the grades and feedback for the individual test cases.

PDF

The PDF interface defines the generate method, which is implemented by the PDFGenerator class. It populates a PDF template with the details from the AssignmentDetails object. The PDF generator gets a map of AssignmentDetails which has key-value pairs corresponding to the unique identifiers of the PDF's AcroFields.

Performance Test Microframework

We implemented a microframework to evaluate system performance. The testExecutionTime measures how long a given Runnable takes to execute by recording its start and end time. It compares the runtime against the specified threshold to determine if performance standards are met. The results of this test are encapsulated in a PerformanceTestResult object.