Ranjandeep	A1909181
Sourav Debnath	A1900755

Technical Report on Modifications to the `ImportHandler` Class

Introduction

This report details the modifications made to the `ImportHandler` class in a Java application. The `ImportHandler` class is integral to handling various import operations, including managing bibliographic data, processing files, and resolving duplicates. The primary objective of these modifications was to enhance the functionality and efficiency of the class by incorporating file preferences and providing improved path handling.

Original Class Structure

The original `ImportHandler` class in the Java application was designed to manage various aspects of importing bibliographic data and associated files. Below is a detailed description of its components:

Static Fields

- **LOGGER**:

```
private static final Logger LOGGER =
LoggerFactory.getLogger(ImportHandler.class);
```

This is a static final logger used for logging purposes throughout the class. It is initialized using the LoggerFactory and provides a standardized way to log messages, errors, and other information, aiding in debugging and monitoring the class's behavior.

Final Fields

- **bibDatabaseContext**:

```
private final BibDatabaseContext bibDatabaseContext;
```

This field holds the context of the bibliographic database, which includes information about the database such as its location, associated files, and other metadata. It is essential for managing and accessing the bibliographic entries.

- **preferences**:

```
      private final ImporterPreferences preferences;
```

  This field manages user preferences related to importing operations. It stores settings and configurations that influence how imports are handled, ensuring that the process aligns with user expectations and requirements.

- **fileUpdateMonitor**:

```
private final FileUpdateMonitor fileUpdateMonitor;
```

  This field monitors file updates, tracking changes to files that are part of the bibliographic database. It helps ensure that the application is aware of any modifications to these files, allowing it to respond appropriately.

- **linker**:

```
private final ExternalFileTypesManager linker;
```

  This field is responsible for linking external files to bibliographic entries. It manages the associations between entries in the database and their corresponding files, facilitating easy access and management of related documents.

- **contentImporter**:

```
private final ContentImporter contentImporter;
```

  This field imports content from external files into the bibliographic database. It handles the actual process of reading data from files and integrating it into the application's data structures.

- **undoManager**:

```
private final UndoManager undoManager;
```

  This field manages undo operations, allowing users to revert changes made during the import process. It provides functionality to track and reverse actions, enhancing user control and error recovery.

- **stateManager**:

```
private final StateManager stateManager;
```

This field manages the state of the application, keeping track of its current status and ensuring that it operates correctly. It helps coordinate different components and maintain consistency throughout the application.

- **dialogService**:

```
private final DialogService dialogService;
```

This field handles dialog interactions with the user. It provides methods to display messages, prompts, and other dialogs, facilitating communication between the user and the application.

- **taskExecutor**:

```
private final TaskExecutor taskExecutor;
```

This field executes background tasks, allowing the application to perform import operations asynchronously. It helps improve performance and responsiveness by offloading time-consuming tasks to separate threads.

Constructor

```
public ImportHandler(BibDatabaseContext bibDatabaseContext,
ImporterPreferences preferences, FileUpdateMonitor
fileUpdateMonitor, ExternalFileTypesManager linker, ContentImporter
contentImporter, UndoManager undoManager, StateManager
stateManager, DialogService dialogService, TaskExecutor
taskExecutor) {
          this.bibDatabaseContext = bibDatabaseContext;
          this.preferences = preferences;
          this.fileUpdateMonitor = fileUpdateMonitor;
          this.linker = linker;
          this.contentImporter = contentImporter;
          this.undoManager = undoManager;
          this.stateManager = stateManager;
          this.dialogService = dialogService;
          this.taskExecutor = taskExecutor;

}
```

The constructor initializes the above fields, setting up the `ImportHandler` class for handling imports. It takes parameters corresponding to each field and assigns them to the respective class variables, ensuring that all necessary components are available for the class's operations.

Methods

- **importFilesInBackground**:

```
public void importFilesInBackground(List<Path> files) {
          // Implementation

}
```

This method handles the background import of files. It takes a list of file paths and processes them asynchronously, ensuring that the import operation does not block the main application thread.

- **importEntries**:

```
public void importEntries(List<BibEntry> entries) {
          // Implementation

}
```

This method imports a list of bibliographic entries. It processes the entries and integrates them into the bibliographic database, handling any necessary cleanup and validation.

- **importCleanedEntries**:

```
public void importCleanedEntries(List<BibEntry> entries) {
          // Implementation

}
```

This method adds cleaned bibliographic entries to the database. It ensures that the entries are properly formatted and validated before integrating them into the database.

- **importEntryWithDuplicateCheck**:

```
public void importEntryWithDuplicateCheck(BibEntry entry) {
          // Implementation

}
```

This method imports a single bibliographic entry while checking for duplicates. It ensures that the same entry is not imported multiple times, maintaining the integrity of the database.

- **handleBibTeXData**:

```
public void handleBibTeXData(String bibtex) {
          // Implementation

}
```

This method processes BibTeX data. It takes a string containing BibTeX entries and parses it, extracting the relevant information and integrating it into the database.

- **handleStringData**:

```
public void handleStringData(String data) {
          // Implementation

}
```

This method processes string data. It handles generic string input, parsing and extracting bibliographic information for integration into the database.

- **downloadLinkedFiles**:

```
public void downloadLinkedFiles(List<Path> paths) {
          // Implementation

}
```

This method downloads files linked to bibliographic entries. It takes a list of file paths and retrieves the corresponding files, ensuring that all necessary documents are available locally.

- **generateKeys**:

```
public void generateKeys(List<BibEntry> entries) {
          // Implementation

}
```

This method generates citation keys for bibliographic entries. It ensures that each entry has a unique identifier, facilitating easy referencing and management.

Figure 1 depicts the UML class diagram of ModifiedImportHandler.java. The private methods section is highlighted in green, indicating the recent modifications made to the codebase.

Fig 1 : UML diagram of modified code

Impact of Changes

The introduction of the `filePreferences` field allows the ImportHandler class to access and utilize user-defined preferences, thereby enhancing the user experience by making the application more customizable and adaptable to individual needs. This change ensures that file operations respect user settings, providing a more consistent and reliable experience.

The `relativize(Path)` method significantly improves path management by converting absolute paths to relative paths. This improvement is particularly beneficial in scenarios where the application needs to handle files across different environments or user settings. By using relative paths, the application can maintain consistency and avoid potential issues related to absolute path dependencies.

Conclusion

The modifications to the ImportHandler class address the issue in JabRef where linked PDF paths were stored as absolute paths instead of relative paths when creating an entry by dragging and dropping a PDF onto the main table. By adding the `filePreferences` field and the `relativize(Path)` method, the class now ensures that PDF paths are stored as relative paths, fixing the issue.

The `filePreferences` field allows ImportHandler to respect user-specific file handling settings. The `relativize(Path)` method converts absolute file paths to relative ones based on these preferences, enhancing the application's portability and consistency across different environments. This change ensures that file paths are managed more reliably, benefiting collaborative use where file structures may differ.