SWE 261P Software Testing and Analysis - Part 1 Report

PDFsam Basic: Functional Testing and Partitioning

JAVA 21 | JUNIT 5 | MAVEN

Course SWE 261P | Quarter Winter 2026 | Team Kingson & Zian & Zhenyu

Repo Github Link: https://github.com/eric-song-dev/pdfsam

Team Members:

- Kingson Zhang: kxzhang@uci.edu
- Zian Xu: zianx11@uci.edu
- Zhenyu Song: zhenyus4@uci.edu

This report documents the systematic functional testing process of **PDFsam Basic**, focusing on equivalence partitioning across core modules: Merge, Rotate, and Extract.

📂 Quick Navigation

🚀 1. Introduction

1.1 Repo Introduction

PDFsam (PDF Split And Merge) Basic is a free, open-source, multi-platform desktop application designed to perform various operations on PDF files. PDFsam has become one of the most popular tools for PDF manipulation, offering functionality ranging from simple page extraction to complex document merging operations.

1.2 Purpose and Features

PDFsam Basic provides the following core functionalities:

Feature	Description
Alternate Mix	Interleave pages from multiple PDF documents
Backpages	Add backpages to existing PDF documents
Extract	Extract specific pages or page ranges from PDF documents
Merge	Combine multiple PDF files into a single document with options for bookmarks, forms, and page normalization
Rotate	Rotate PDF pages by 90°, 180°, or 270° degrees
Split	Divide PDFs by page count, size, or bookmarks

1.3 Technical Overview

> [!IMPORTANT] **Total Project Scale:** 734 files, **36,281 Logical Lines of Code (LLOC)**, and 15,045 comments.

▶ 📊 Click to View LOC Details

Languages

language	files	code	comment	blank	total
Java	622	29,515	14,734	6,099	50,348
XML	45	4,087	48	153	4,288
PostCSS	20	1,642	152	366	2,160
Markdown	3	360	0	135	495
YAML	5	176	14	45	235
Java Properties	22	134	0	4	138
Batch	4	107	33	39	179
HTML	2	92	2	2	96
Shell Script	4	85	62	23	170
C#	1	48	0	3	51
JSON	6	35	0	0	35

The project is primarily written in Java (approx. 82%), consisting of roughly 35000 lines of code. View LOC

1.4 Project Architecture

PDFsam follows a modular multi-project Maven structure:

```
pdfsam/
├── pdfsam-basic          # Main application entry point
├── pdfsam-core           # Core utilities and support classes
├── pdfsam-fonts          # Font resources
├── pdfsam-gui            # GUI components and controllers
├── pdfsam-i18n           # Internationalization
├── pdfsam-model          # Domain model classes
├── pdfsam-persistence    # Data persistence layer
├── pdfsam-service        # Business services
├── pdfsam-test           # Test utilities
├── pdfsam-themes         # UI themes
├── pdfsam-ui-components  # Reusable UI components
└── pdfsam-tools/         # PDF manipulation tools
    ├── pdfsam-merge
    ├── pdfsam-rotate
    ├── pdfsam-extract
    ├── pdfsam-simple-split
    ├── pdfsam-split-by-size
    ├── pdfsam-split-by-bookmarks
    ├── pdfsam-alternate-mix
    └── pdfsam-backpages
```

The application relies heavily on the **Sejda** library for low-level PDF operations, providing a robust foundation for document manipulation.

📝2. Build Documentation

2.1 Prerequisites

Before building PDFsam, ensure the following tools are installed:

- **Java Development Kit (JDK)**: Version 21 (NOT JDK 11)
- **Apache Maven**: Build tool for dependency management
- **Git**: For cloning the repository
- **Gnu gettext**: Required for internationalization
  - **Windows**: Download from mlocati/gettext-iconv-windows
  - **macOS**: Install via Homebrew (`brew install gettext`) or use Docker
  - **Linux**: Usually pre-installed or available via package manager

2.2 Cloning the Repository

```
git clone https://github.com/torakiki/pdfsam.git
cd pdfsam
```

2.3 Building the Project

PDFsam uses Java 21's preview features (Foreign Function & Memory API), so the `--enable-preview` flag is required:

```
# Compile the project
mvn clean compile
```

```
# Package the application
mvn clean package -DskipTests
```

```
# Build with tests
mvn clean install -DskipTests
```

2.4 Running the Application

After successful compilation, the application can be run using:

```
cd pdfsam-basic
```

```
mvn exec:exec
```

2.5 IDE Setup

For IntelliJ IDEA or Eclipse:

1. Import as Maven project
2. Enable preview features in compiler settings
3. Set Java 21 as the project SDK
4. Find and run ./pdfsam-basic/src/main/java/org/pdfsam/basic/App.java

🧪 3. Existing Test Cases

3.1 Testing Frameworks

PDFsam employs a comprehensive testing stack:

Framework	Version	Purpose
JUnit 5 (Jupiter)	Latest	Unit testing framework
Mockito	Latest	Mock object creation
AssertJ	Latest	Fluent assertions

3.2 Test Organization

Tests are organized following Maven conventions:

```
src/
├── main/java/        # Production code
└── test/java/        # Test code
    └── org/pdfsam/
        └── tools/
            ├── merge/
            │   ├── MergeParametersBuilderTest.java
            │   ├── MergeOptionsPaneTest.java
            │   └── MergeSelectionPaneTest.java
            ├── rotate/
            │   └── ...
            └── extract/
                └── ...
```

3.3 Test Categories

Our code creates a few new testing components. For the first lab testing, we will be primarily testing the pdf manipulation tools found in the tools folder, specifically the extract, merge, and rotate functionality. Below are following currently implemented. As the project continues, we will continue to add more testing.

1. **Unit Tests**: Isolated component testing with mocks
2. **Integration Tests**: Testing component interactions

3.4 Running Tests

```
# Run all tests
mvn test
```

```
# Run tests for a specific module
mvn test -pl pdfsam-tools/pdfsam-rotate
```

```
# Run a specific test class
cd pdfsam-tools/pdfsam-rotate
mvn test -Dtest=RotateParametersBuilderTest
```

✨4. Partition Testing

4.1 Motivation for Systematic Functional Testing

Software testing faces a fundamental challenge: **exhaustive testing is impossible**. For any non-trivial program, the space of possible inputs is effectively infinite. Consider a simple function that takes a 32-bit integer—testing all 4.3 billion possible values is impractical, and real-world inputs are far more complex.

Systematic functional testing addresses this by:

- Treating the software as a "black box" based on its specification
- Identifying meaningful categories of inputs
- Ensuring representative coverage of the input domain

This approach is essential because:

1. **Ad-hoc testing** misses edge cases and boundary conditions
2. **Random testing** provides poor coverage of critical scenarios
3. **Developer intuition** often overlooks non-obvious failure modes

4.2 Partition Testing Concepts

Partition testing (also known as equivalence partitioning) is a systematic technique that:

1. **Divides the input domain** into partitions where the program is expected to behave equivalently for all values within each partition
2. **Selects representative values** from each partition
3. **Tests boundary values** at partition edges where defects often lurk

Key principles:

- **Completeness**: Partitions cover the entire input domain
- **Disjointness**: Partitions don't overlap (each input belongs to exactly one partition)
- **Homogeneity**: All values in a partition should trigger similar behavior

Benefits of partition testing:

- Reduces test cases while maintaining effectiveness
- Provides systematic coverage documentation
- Identifies missing test cases
- Focuses testing effort on distinct behaviors

4.3 Zhenyu's Partition Testing: Merge Feature

4.3.1 Feature Description

The feature under test is the **PDF Merge Configuration**, specifically the `MergeParametersBuilder` class. This component is responsible for collecting user inputs and settings to construct a valid `MergeParameters` object, which drives the actual merge process. It handles critical configuration options such as:

- Input PDF files and their order.
- Output file destination and overwrite policies.
- Processing options (compression, versioning).
- Content policies (Outline/Bookmarks, Table of Contents, AcroForms handling).
- Page manipulation (normalization, blank pages for odd-numbered files).

4.3.2 Partitioning Scheme

To ensure robust coverage of the configuration logic, the input space was partitioned based on **builder state complexity** and **input validity**:

1. **Default State Partition**: The builder is used without any explicit configuration.
   - *Goal*: specific verification of safe defaults.
   - *Representative Input*: An empty `MergeParametersBuilder` instance.
2. **Fully Configured Partition**: The builder is provided with explicit, non-default values for every available setting.
   - *Goal*: Verify that all user choices are correctly captured and propagated.
   - *Representative Input*: A builder with inputs, `PdfVersion.VERSION_1_6`, `OutlinePolicy.ONE_ENTRY_EACH_DOC`, `ToCPolicy.DOC_TITLES`, etc.
3. **Input Sequence Partition**: Multiple inputs added in a specific order.
   - *Goal*: specific verification that the merge order respects the user's input sequence.
   - *Representative Input*: Inputs `["1.pdf", "2.pdf", "3.pdf"]` added sequentially.
4. **Redundant/Edge-Case Input Partition**: Duplicate or redundant inputs.
   - *Goal*: specific verification of deduplication logic.
   - *Representative Input*: The same `PdfMergeInput` object added twice.
5. **Invalid Input Partition**: Null or missing values.
   - *Goal*: Ensure null safety and robustness.
   - *Representative Input*: `addInput(null)` and setting policies to `null`.

4.3.3 Test Implementation

The partition tests are implemented in `ZhenyuMergePartitionTest.java` using JUnit 5 and Mockito.

- **`testDefaults()`**: Covers the *Default State Partition*. Asserts that a fresh builder produces parameters with expected defaults (e.g., `OutlinePolicy.RETAIN`, `ToCPolicy.NONE`, `isCompress` false).
- **`testFullConfiguration()`**: Covers the *Fully Configured Partition*. Sets every property (e.g., `compress(true)`, `version(1.6)`) and asserts the resulting `MergeParameters` object reflects these exact values.
- **`testAddInput_PreservesOrder()`**: Covers the *Input Sequence Partition*. Adds three mock inputs and verifies they appear in the exact same order in the final list.
- **`testAddInput_Deduplicates()`**: Covers *Redundant Input Partition*. Adds the same input object twice and asserts the list size is 1.
- **`testAddInput_IgnoresNull()`** and **`testNullPolicies_AreAllowed()`**: Covers the *Invalid Input Partition*. Verifies that adding `null` inputs serves no operation and that setting null policies doesn't crash the builder.

4.4 Zian's Partition Testing: Rotate Feature

4.4.1 Feature Description

The `RotateParametersBuilder` class constructs parameters for PDF page rotation. It handles:

- Rotation angle selection
- Page selection (all, odd, even, or custom ranges)
- Multiple input sources
- Output file naming

4.4.2 Partitioning Scheme

This test suite implements a systematic **Input Domain Partitioning** strategy to validate the `RotateParametersBuilder`. The logic is decomposed into four primary dimensions:

Dimension 1: Angular Transformation Mapping

- `[P1a]` Systematic Rotation (90°)😗* Validates the fundamental mapping of a quadrant clockwise rotation using `Rotation.DEGREES_90`. It ensures that the Builder correctly encapsulates the angular intent into the final task parameters.

Dimension 2: Predefined Page Selection Strategy

- `[P2a]` (Identity Mapping)😗* Verifies the "Rotate All" logic using `PredefinedSetOfPages.ALL_PAGES`. For a standard 10-page document, it asserts that all 10 pages are correctly targeted for transformation.
- `[P2b]` (Odd Parity Filter)😗* Evaluates the parity-based filtering using `PredefinedSetOfPages.ODD_PAGES`. It confirms that for a 10-page document, only the 5 odd-indexed pages are selected.

Dimension 3: Parameter Precedence & Override Logic

- `[P3a]` (Fallback Mechanism)😗* Confirms the system's "Safe Default" behavior. When no custom ranges are provided (`null`), the system successfully falls back to the global predefined strategy (e.g., Odd Pages).
- `[P3b]` (Complex Range Merging)😗* Validates the override mechanism using a `HashSet` of multiple disjoint `PageRange` objects (e.g., pages 1-3 and 7-9). It ensures custom user input takes precedence over global settings.

Dimension 4: Input Batch Cardinality

- `[P4a]` (Zero Input)😗* Tests the system's state when no files are added, ensuring `hasInput()` correctly returns `false` to prevent null-pointer operations.
- `[P4b]` (Multi-Source)😗* Verifies the bulk processing capability by injecting 3 distinct PDF sources. It asserts that the `InputSet` size matches the expected count of 3.

Combined Scenario: Integration Verification

- **Cross-Partition Validation:** A composite test case that simultaneously evaluates 90° rotation, multiple input sources with heterogeneous selection strategies (one file with custom ranges, another with predefined sets), and output target consistency.

4.4.3 Test Implementation

The partition tests are implemented in `ZianRotatePartitionTest.java` using JUnit 5, AssertJ, and Mockito.

- **`rotation90Degrees()`**: Covers the *Angular Transformation Partition*. It asserts that when a 90° clockwise rotation is set, the builder correctly maps the `Rotation.DEGREES_90` constant to the resulting task parameters.
- **`allPages()` and `oddPages()`**: Cover the *Predefined Page Set Partition*. These tests simulate a 10-page document and verify that the selection logic correctly calculates the expected page count (e.g., all 10 pages for `ALL_PAGES` vs. 5 pages for `ODD_PAGES`).
- **`noCustomRange()`**: Covers the *Fallback Strategy Partition*. It verifies that if no specific page ranges are provided, the builder successfully defaults to the predefined selection type (e.g., rotating only odd pages).
- **`multipleCustomRanges()`**: Covers the *Custom Override Partition*. It uses a `HashSet` of multiple `PageRange` objects (e.g., pages 1-3 and 7-9) to ensure that explicit user-defined ranges correctly take precedence over global predefined settings.
- **`noInputs()` and `multipleSources()`**: Cover the *Input Cardinality Partition*. These tests establish the system's boundary behavior, asserting that `hasInput()` returns `false` when empty and correctly tracks the size of the input set when multiple PDF sources are injected.
- **`combinedPartitions()`**: Covers the *Integration Scenario*. This comprehensive test validates a complex state where multiple files are processed simultaneously using heterogeneous strategies—one file with a custom range and another using a predefined set—ensuring the builder maintains state integrity across bulk operations.

4.5 Kingson's Partition Testing: Extract Feature

4.5.1 Feature Description

The `ExtractParametersBuilder` class constructs parameters for extracting pages from PDFs. It handles:

- Page selection expressions (ranges, individual pages, keywords)
- Selection inversion
- Separate file output per range
- Optimization policies
- Bookmark handling

4.5.2 Partitioning Scheme

Partition 1: Page Selection Type

Partition	Description	Representative Value
P1a: Single page	One page number	`"5"`
P1b: Single range	Contiguous range	`"1-10"`
P1c: Multiple ranges	Non-contiguous ranges	`"1-5,10-15,20"`
P1d: Last keyword	Special 'last' page	`"last"`

Rationale: Different selection types exercise different parsing and processing paths.

Partition 2: Selection Inversion

Partition	Description	Representative Value
P2a: Normal	Extract specified pages	`invertSelection = false`
P2b: Inverted	Extract all EXCEPT specified	`invertSelection = true`

Rationale: Inversion fundamentally changes the extraction logic.

Partition 3: Output File Mode

Partition	Description	Representative Value
P3a: Single file	All pages in one output	`separateForEachRange = false`
P3b: Separate files	One file per range	`separateForEachRange = true`

Rationale: Affects output file generation strategy.

📋 5. Test Implementation Summary

5.1 New Test Files

File	Location	Team Member
`ZhenyuMergePartitionTest.java`	`pdfsam-tools/pdfsam-merge/src/test/java/org/pdfsam/tools/merge/`	Zhenyu Song
`ZianRotatePartitionTest.java`	`pdfsam-tools/pdfsam-rotate/src/test/java/org/pdfsam/tools/rotate/`	Zian Xu
`KingsonExtractPartitionTest.java`	`pdfsam-tools/pdfsam-extract/src/test/java/org/pdfsam/tools/extract/`	Kingson Zhang

5.2 Running the Partition Tests

```
# Run individual partition tests
mvn test -pl pdfsam-tools/pdfsam-merge -Dtest=ZhenyuMergePartitionTest
mvn test -pl pdfsam-tools/pdfsam-rotate -Dtest=ZianRotatePartitionTest
mvn test -pl pdfsam-tools/pdfsam-extract -Dtest=KingsonExtractPartitionTest
```

🎯 6. Conclusion

This report documents our analysis of PDFsam Basic, a robust PDF manipulation tool with a well-structured codebase and comprehensive existing test suite. Through systematic partition testing, we have:

1. **Identified key features** suitable for functional testing
2. **Designed partitioning schemes** that provide systematic coverage of input domains
3. **Implemented JUnit 5 tests** that exercise each partition with representative values
4. **Documented our approach** to serve as a foundation for future testing efforts

The partition testing methodology demonstrates its value in revealing potential edge cases and ensuring comprehensive coverage that ad-hoc testing might miss.

The next addition to our testing will include Finite State Machine testing, which will test the different states of the program.