ENGI 9839 - Software Verification and Validation

Leonardo David Eras Delgado

July 2025

Contents

1 Introduction

Around 75% of people report experiencing glossophobia, the fear of public speaking [4]. For some individuals, this manifests as mild nervousness, while for others it can cause intense anxiety and panic.

Although numerous free tools exist to support public speaking and presentation delivery [3], few specifically analyze the visual content of presentation slides. Slides are often considered merely *supporting materials*, and their quality is frequently overlooked when aiming to improve overall presentation effectiveness [1].

This gap presents an opportunity for software solutions that assist speakers in evaluating and improving their slides. The *PresentationApp* addresses this need by performing automated analyses of slides, offering insights into aspects such as visual contrast, text density, font sizes.

However, the primary goal of this project is not solely to introduce the tool itself, but to demonstrate a rigorous process of software verification and validation (V&V). This document focuses on how various testing techniques—including unit tests, system tests, and exploratory testing—were employed to ensure the correctness, reliability, and usability of the *PresentationApp*. Where appropriate, code excerpts are provided to illustrate specific testing approaches.

2 Related Work

Audience engagement is widely regarded as a key indicator of presentation effectiveness [1]. Traditionally, public speaking training has relied on in-person coaching to help speakers reduce glossophobia and improve delivery skills.

In recent years, technological solutions have emerged to support public speaking practice, many of which leverage machine learning to analyze nonverbal communication, voice characteristics, and speech fluency [5], [7]. However, these tools typically require audio and video input and focus on aspects such as body posture, vocal modulation, and speech disfluencies.

By contrast, relatively few tools explicitly analyze the visual quality and readability of slide content, despite slides being an important supporting element in presentations. The *PresentationApp* seeks to address this gap by providing automated analysis and feedback on slide attributes such as text density, color contrast, and layout clarity.

3 Methodology

3.1 System Overview

PresentationApp enables users to register, log in, and upload ***Portable Document Format (PDF)*** files for analysis. The tool evaluates three major slide features, each mapped to a 0–5 scale for user-friendly interpretation [2]. A report is then generated, displaying scores and tailored feedback on potential improvements. To reduce visual overload, only the last three analyses are visible on the *dashboard* screen.

- ***Contrast ratio*** Measures the contrast between text and background, based on definitions and formulas from the *Web Content Accessibility Guidelines* [9].

While originally designed for web content, these calculations are adapted here for slides in oral presentations.

- ***Number of words*** Computed by counting words per slide, excluding symbols, page numbers, and similar noise. Guidance from [6] informs recommended word thresholds for slides. The slide title is included in the total count.
- ***Font sizes*** The tool identifies the smallest font size used on each slide to assess readability.

3.2 Tools and Frameworks

PresentationApp was developed in Python using the **Django** web framework. Django's architecture enables rapid development of secure and maintainable web applications, and its integrated testing tools facilitate systematic verification and validation activities. Among the libraries used for the project, the most important are:

- PyMuPDF, PyPDF for PDF content extraction and validation.
- OpenCV for image analysis.

Django's integrated testing framework provided efficient tools for writing and running automated tests, ensuring the reliability of the application.

3.3 Testing Approach

PresentationApp applies several layers of testing to ensure the correctness and reliability of its analytical processes. The testing approach is divided into **unit testing**,

integration/system testing, and **exploratory testing**.

The project includes a comprehensive suite of unit and system tests. While only a few representative unit tests are shown here, the complete set can be accessed from the project's GitHub repository. [2].

3.3.1 Unit Testing Approach

This section describes the methodology followed for writing tests and explains at least two unit tests in detail, in alignment with the recommended testing process:

1. Identify the function to test.
2. Analyze and list dependencies.
3. Isolate the function.
4. Write and run the tests.

These are pure unit tests with no authentication, no file uploads, no filesystem interactions, just validating logic in isolation.

Unit Test 1 - Slide Word Count Scoring

Target function: $score\_num\_words(pages\_data)$

Why test it: This function evaluates how densely packed a slide is with text. Instead of returning the raw word count, it converts this into a 0–5 star score to guide users on whether the slide is too wordy or appropriately concise. This metric is derived from established presentation heuristics like Kawasaki's 10/20/30 rule [6].

The word count thresholds used to compute star ratings are:

- **5 stars**: ≤ 25 words
- **4 stars**: 26–40 words
- **3 stars**: 41–60 words
- **2 stars**: 61–80 words
- **1 star**: > 80 words

Goal: Ensure that various slide configurations are mapped to the correct score, particularly for edge cases like no text or overly wordy slides.

Dependencies: The function depends on the structure of *pages_data*, a list of dictionaries generated by the *PyMuPDF* library. Each dictionary contains:

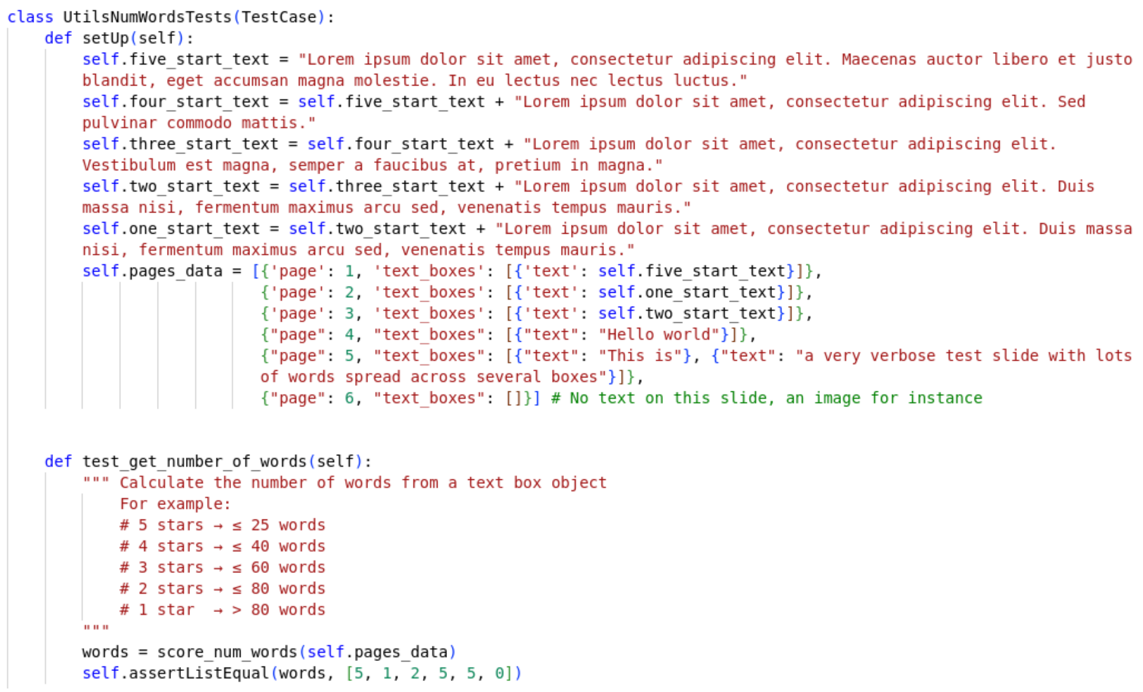
- A page number.
- A a list of *text_boxes*, each with:
  - A text string.
  - Extra attributes, ex. font, color and size

Isolation plan: To isolate this function from its dependency on actual PDF files, we mock the *pages_data* structure. This way, we can test scoring logic directly without needing to parse a real document.

Test scenarios:

- Three slides with different *Lorem Ipsum* length texts, measured to produce a specific value from the score function.

- A slide with a single text box containing the sentence “Hello world” should return 2 words.
- A slide with multiple text boxes should sum all words from each box.
- A slide with no text at all.



```python
class UtilsNumWordsTests(TestCase):
    def setUp(self):
        self.five_start_text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas auctor libero et justo
        blandit, eget accumsan magna molestie. In eu lectus nec lectus luctus."
        self.four_start_text = self.five_start_text + "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed
        pulvinar commodo mattis."
        self.three_start_text = self.four_start_text + "Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Vestibulum est magna, semper a faucibus at, pretium in magna."
        self.two_start_text = self.three_start_text + "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis
        massa nisi, fermentum maximus arcu sed, venenatis tempus mauris."
        self.one_start_text = self.two_start_text + "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis massa
        nisi, fermentum maximus arcu sed, venenatis tempus mauris."
        self.pages_data = [{'page': 1, 'text_boxes': [{'text': self.five_start_text}]},
                           {'page': 2, 'text_boxes': [{'text': self.one_start_text}]},
                           {'page': 3, 'text_boxes': [{'text': self.two_start_text}]},
                           {"page": 4, "text_boxes": [{"text": "Hello world"}]},
                           {"page": 5, "text_boxes": [{"text": "This is"}, {"text": "a very verbose test slide with lots
                           of words spread across several boxes"}]},
                           {"page": 6, "text_boxes": []}] # No text on this slide, an image for instance


    def test_get_number_of_words(self):
        """ Calculate the number of words from a text box object
            For example:
            # 5 stars → ≤ 25 words
            # 4 stars → ≤ 40 words
            # 3 stars → ≤ 60 words
            # 2 stars → ≤ 80 words
            # 1 star  → > 80 words
        """
        words = score_num_words(self.pages_data)
        self.assertListEqual(words, [5, 1, 2, 5, 5, 0])
```

Figure 1: Unit test for the `luminance` function.

Unit Test 2 - Contrast Calculation for Black on White Test

Target functions: *luminace*, *contrast_to_stars*

Why test it: These calculations directly affect whether the system provides accurate accessibility feedback for slides [9]. If the contrast computation is wrong, users might get misleading recommendations.

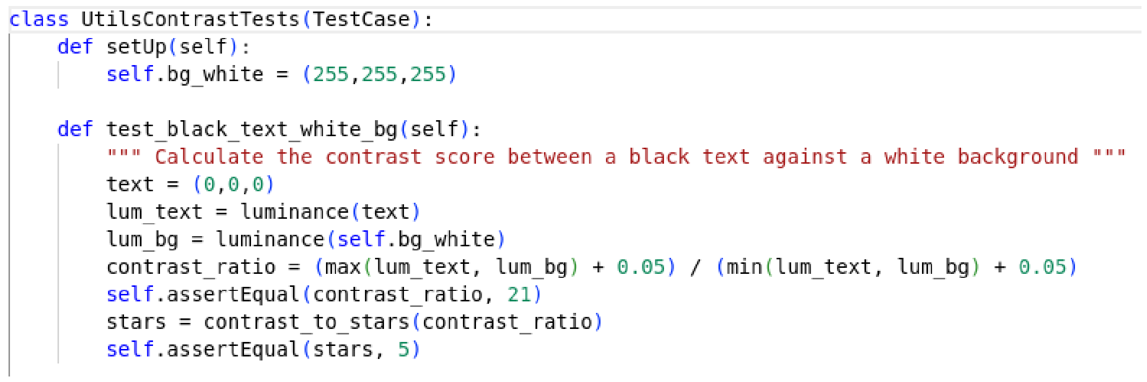
Goal: Given well-known color pairs (e.g. pure black text on pure white background), the contrast ratio should be exactly 21, which WCAG defines as the highest possible ratio [8]. This should map to a 5-star score.

Dependencies: For the *luminance* calculation, we can mock the text and the background colors. The *contrast_to_stars* ratio can be calculated from the division defined by [8], using the result of the *luminance* function.

Isolation plan: Run the calculations with hard-coded RGB values in a standalone test.

Test scenario:

- Black text color on a white background.



```
class UtilsContrastTests(TestCase):
    def setUp(self):
        self.bg_white = (255,255,255)

    def test_black_text_white_bg(self):
        """ Calculate the contrast score between a black text against a white background """
        text = (0,0,0)
        lum_text = luminance(text)
        lum_bg = luminance(self.bg_white)
        contrast_ratio = (max(lum_text, lum_bg) + 0.05) / (min(lum_text, lum_bg) + 0.05)
        self.assertEqual(contrast_ratio, 21)
        stars = contrast_to_stars(contrast_ratio)
        self.assertEqual(stars, 5)
```

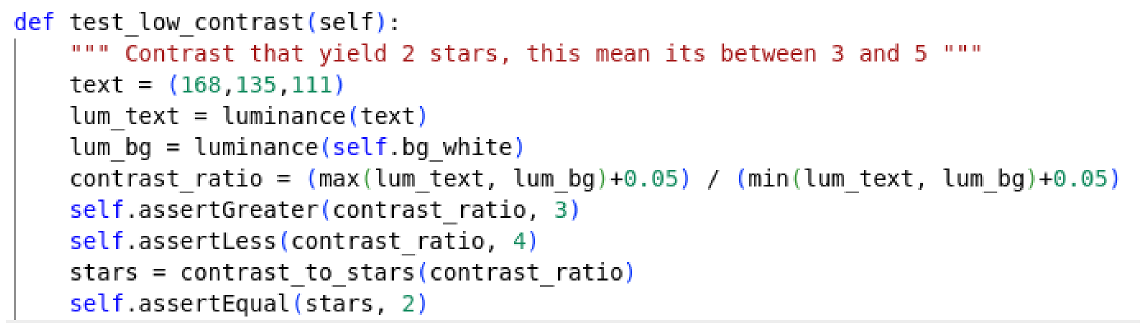
Figure 2: Unit test for the `luminance` and `contrast_to_stars` functions.

I have implemented multiple unit test cases to verify that the *contrast_to_stars* function correctly maps contrast ratios to star ratings, based on the following scale:

- **5 stars**: ≥ 12 contrast ratio.
- **4 stars**: 8–12 contrast ratio.

- **3 stars**: 5–8 contrast ratio.
- **2 stars**: 3–5 contrast ratio.
- **1 star**: < 3 contrast ratio.

Each test simulates a specific RGB color against a white background and asserts that the computed contrast ratio yields the expected number of stars. Below is a sample test case:



```python
def test_low_contrast(self):
    """ Contrast that yield 2 stars, this mean its between 3 and 5 """
    text = (168,135,111)
    lum_text = luminance(text)
    lum_bg = luminance(self.bg_white)
    contrast_ratio = (max(lum_text, lum_bg)+0.05) / (min(lum_text, lum_bg)+0.05)
    self.assertGreater(contrast_ratio, 3)
    self.assertLess(contrast_ratio, 4)
    stars = contrast_to_stars(contrast_ratio)
    self.assertEqual(stars, 2)
```

Figure 3: Sample test case.

3.3.2 System Testing Approach

Five system-level test cases are outlined following the Category-Partition Method, these are described on the Table 1, which emphasizes choosing representative inputs, imposing constraints, and identifying independently testable functionality. The steps used for Category-Partition-Method are the following:

1. Identify Testable Functions.
2. Identify Parameters and Environment Characteristics (Choices).

3. Identify Categories and Partitions (Values).

4. Impose Constraints.

5. Generate Test Cases.

Table 1 will be referenced throughout this section. A few representative test cases are discussed in detail, while the remaining ones are summarized in the table for brevity.

Table 1: System Test Cases Using the Category-Partition Method

Test Case	**Function Under Test**	**Choices**	**Partitions**	**Constraints**
TC1	Contrast Calculation	*text_boxes*, Image folder, filepath	Valid, Invalid Format. File/Folder permissions. Text box data	File existence, permissions. Text on slides
TC2	PDF Upload Pipeline	Title, File	Several Partitions Defined	Must be logged in, file must be PDF, title constraints
TC3	Font Size Detection	*text_boxes*	Valid, Invalid Format	Must be logged in, PDF must be valid.
TC4	Text Box Extraction	Filepath	Valid/Invalid path	Must be logged in, PDF must be valid.
TC5	User login	Username, Password	Format, Account status	Correct Password, user exists on database

System Test 1 (TC1) - Contrast Calculation

The following is an example of a full system test case that simulates a user scenario from login to PDF upload to grading. It involves creating a test user, simulating the upload of a PDF file, extracting visual elements, computing the contrast, and verifying the final grades. Setup and teardown functions are used to clean up generated files and ensure isolated test environments.

Step 1 = Identify Testable Functions: *socre_contrast* This function returns the contrast score per slide on an uploaded PDF file.

Step 2 - Identify Parameters and Environment Characteristics (Choices). Input variables:

- `text_boxes`: a custom data structure returned by `PyMyPDF`, used to extract data from the PDF.
  - Format: e.g., valid size, valid data.
- `image_folder`: string; the relative path of the image folder.
  - Format: valid data.
  - Correctness: folder exists.
  - Accessibility: folder has read permission.
- `filename`: string; represents the name of the PDF file.
  - Format: e.g., valid size.
  - Correctness: file exists.
- *Environmental Factors:*
  - The PDF file contains valid, readable content (e.g., has text).

  - File system permissions are correct.
  - Network connectivity is stable (if relevant).
  - Database connection is active (if used during scoring).

Step 3 – Identify Categories and Partitions:

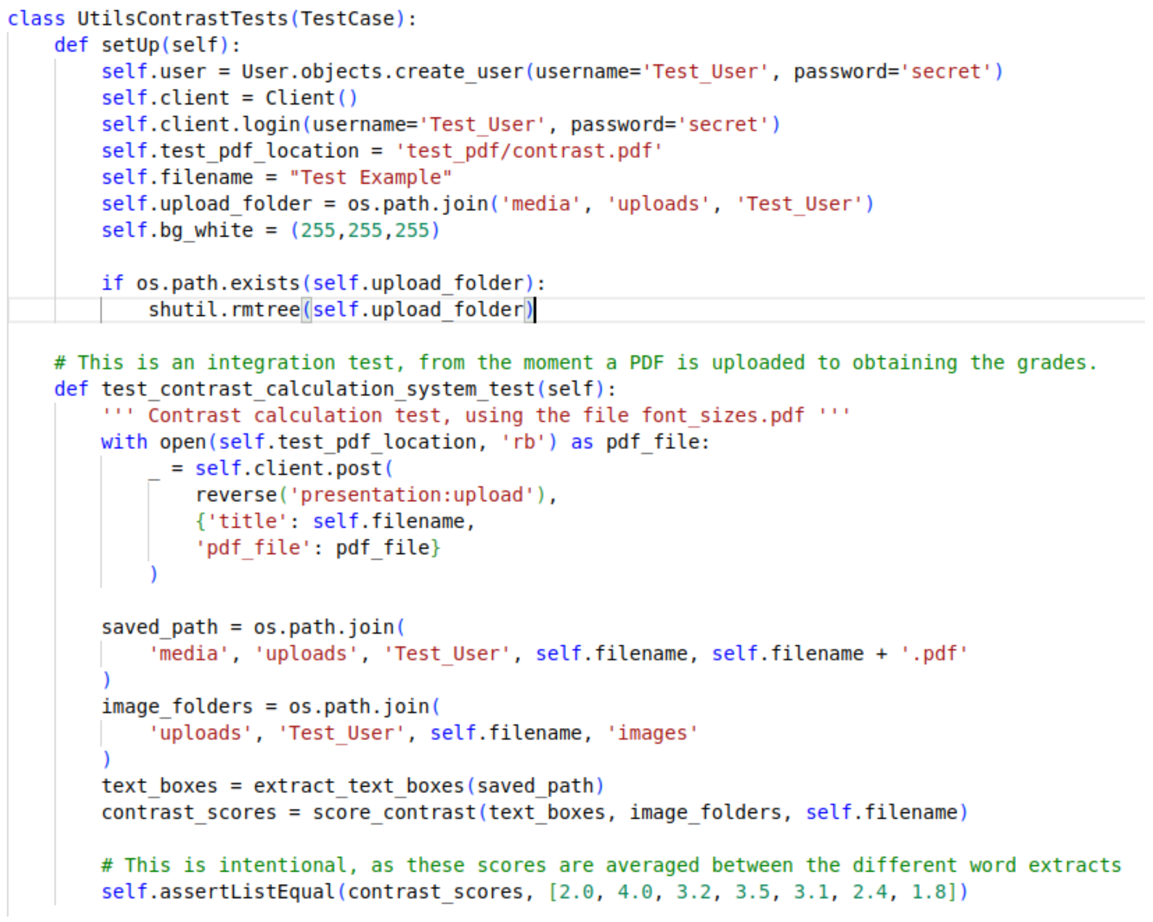
- `text_boxes` – Format:
  - Number of pages: 0 or multiple.
  - Blocks of text: 0 or multiple.
  - Format: Valid / Invalid.
- `image_folder` – Format:
  - Valid / Invalid path.
- `image_folder` – Existence:
  - Folder exists / Folder missing.
- `image_folder` – Permission:
  - Read permission available / Not available.
- `filename` – Format:
  - Valid format / Invalid format.
- `filename` – Existence:

  - File exists / File missing.

Step 4 – Impose Constraints:

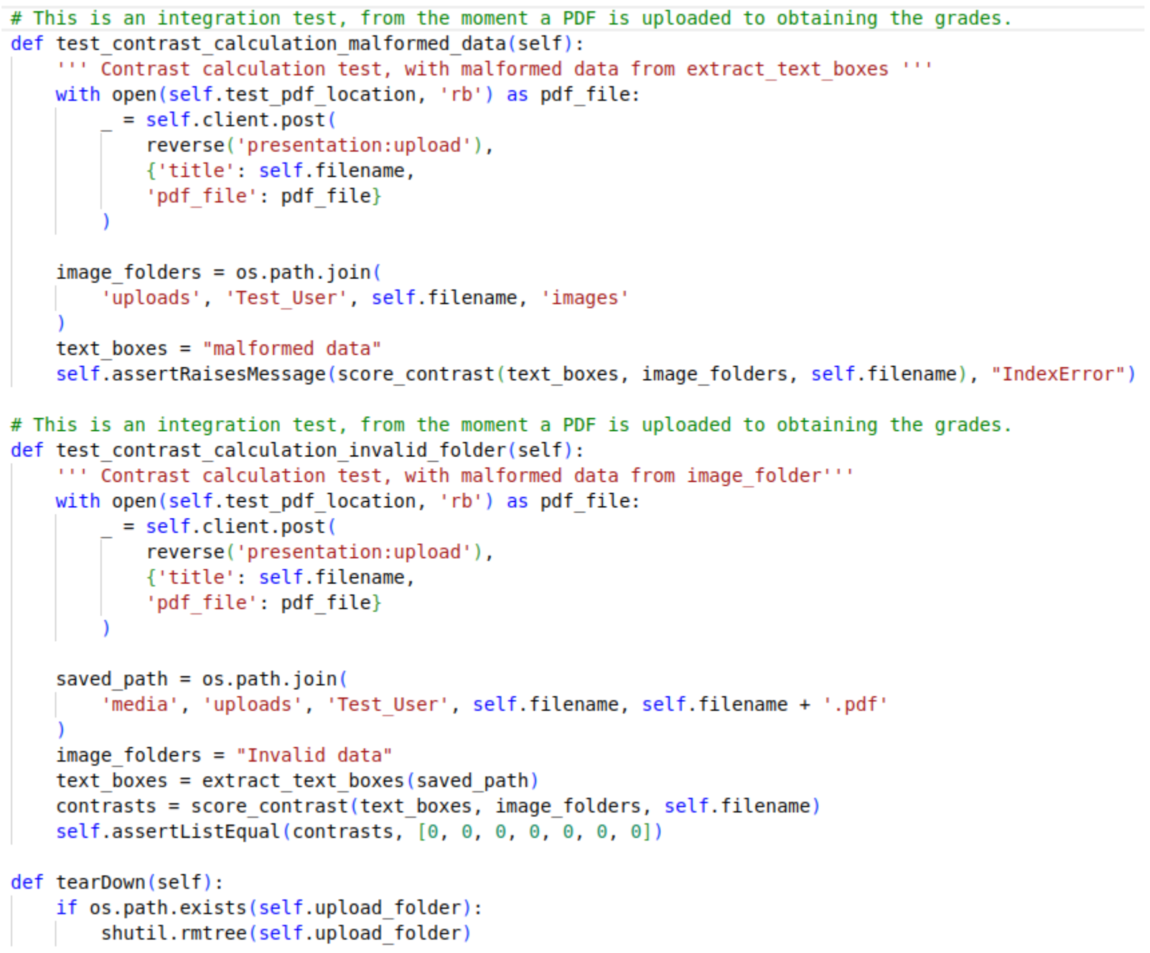
- *Error Constraints:*
  - If the folder does not exist, the function raises a FileNotFoundError or returns a specific error code indicating missing images.
  - If the folder is not accessible due to read permission issues, the function raises a PermissionError or returns a relevant error code.
- *Single Constraints:*
  - If no text is detected in any slide, the function returns a contrast score of 0 for that slide.
- *Conditional Constraints:*
  - If there is exactly one text element on a slide, the contrast score is computed solely based on that element's foreground and background contrast.

Step 5 – Generate Test Cases: As shown in Figure 4, the system test validates the contrast score under normal conditions. Figure 5 illustrates the failure cases with improper file inputs.



```python
class UtilsContrastTests(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='Test_User', password='secret')
        self.client = Client()
        self.client.login(username='Test_User', password='secret')
        self.test_pdf_location = 'test_pdf/contrast.pdf'
        self.filename = "Test Example"
        self.upload_folder = os.path.join('media', 'uploads', 'Test_User')
        self.bg_white = (255,255,255)

        if os.path.exists(self.upload_folder):
            shutil.rmtree(self.upload_folder)

    # This is an integration test, from the moment a PDF is uploaded to obtaining the grades.
    def test_contrast_calculation_system_test(self):
        ''' Contrast calculation test, using the file font_sizes.pdf '''
        with open(self.test_pdf_location, 'rb') as pdf_file:
            _ = self.client.post(
                reverse('presentation:upload'),
                {'title': self.filename,
                'pdf_file': pdf_file}
            )

        saved_path = os.path.join(
            'media', 'uploads', 'Test_User', self.filename, self.filename + '.pdf'
        )
        image_folders = os.path.join(
            'uploads', 'Test_User', self.filename, 'images'
        )
        text_boxes = extract_text_boxes(saved_path)
        contrast_scores = score_contrast(text_boxes, image_folders, self.filename)

        # This is intentional, as these scores are averaged between the different word extracts
        self.assertListEqual(contrast_scores, [2.0, 4.0, 3.2, 3.5, 3.1, 2.4, 1.8])
```

Figure 4: System test for the contrast calculation on a valid PDF file.



```python
# This is an integration test, from the moment a PDF is uploaded to obtaining the grades.
def test_contrast_calculation_malformed_data(self):
    ''' Contrast calculation test, with malformed data from extract_text_boxes '''
    with open(self.test_pdf_location, 'rb') as pdf_file:
        _ = self.client.post(
            reverse('presentation:upload'),
            {'title': self.filename,
            'pdf_file': pdf_file}
        )

    image_folders = os.path.join(
        'uploads', 'Test_User', self.filename, 'images'
    )
    text_boxes = "malformed data"
    self.assertRaisesMessage(score_contrast(text_boxes, image_folders, self.filename), "IndexError")

# This is an integration test, from the moment a PDF is uploaded to obtaining the grades.
def test_contrast_calculation_invalid_folder(self):
    ''' Contrast calculation test, with malformed data from image_folder'''
    with open(self.test_pdf_location, 'rb') as pdf_file:
        _ = self.client.post(
            reverse('presentation:upload'),
            {'title': self.filename,
            'pdf_file': pdf_file}
        )

    saved_path = os.path.join(
        'media', 'uploads', 'Test_User', self.filename, self.filename + '.pdf'
    )
    image_folders = "Invalid data"
    text_boxes = extract_text_boxes(saved_path)
    contrasts = score_contrast(text_boxes, image_folders, self.filename)
    self.assertListEqual(contrasts, [0, 0, 0, 0, 0, 0, 0])

def tearDown(self):
    if os.path.exists(self.upload_folder):
        shutil.rmtree(self.upload_folder)
```

Figure 5: System test for contrast calculation using an invalid folder structure and an incorrect image filename. The *tearDown* method is used for cleanup.

System Test 2 (TC2) - PDF Upload Pipeline

The following is an example of a full system test case that simulates a user scenario from login to PDF upload. It involves creating a test user, simulating the upload of a PDF file testing on edge cases to get errors.

Step 1 = Identify Testable Functions: *upload* This function asks the user through an HTML form to upload a PDF file to be processed.

Step 2 - Identify Parameters and Environment Characteristics (Choices).

Input variables:

- `title`: Name for the new PDF file to be processed.
  - Format: e.g. valid data, valid length. Special characters are not allowed ('$%˙.,-*')
- `file`: PDF file.
  - Existence: File exists, File does not exist.
  - Size: Valid / Invalid (0 MB or more than 5 MB).
  - Extension: Valid (Only PDF file extension accepted) / Invalid otherwise.

Step 3 – Identify Categories and Partitions:

- `title` – Format:
  - Format Partitions.
    - Has special characters.
    - Is too short (less than 10 characters long)
    - Is too long (more than 80 characters long)
    - Valid format
- `title` – Size:
  - Size Partitions.

    * File is empty.
    * File is too big (greater than 5 MB)

- `title` – Extension:
  - Extension Partitions.
    * It is a valid PDF file.
    * It is another extension.

Step 4 – Impose Constraints:

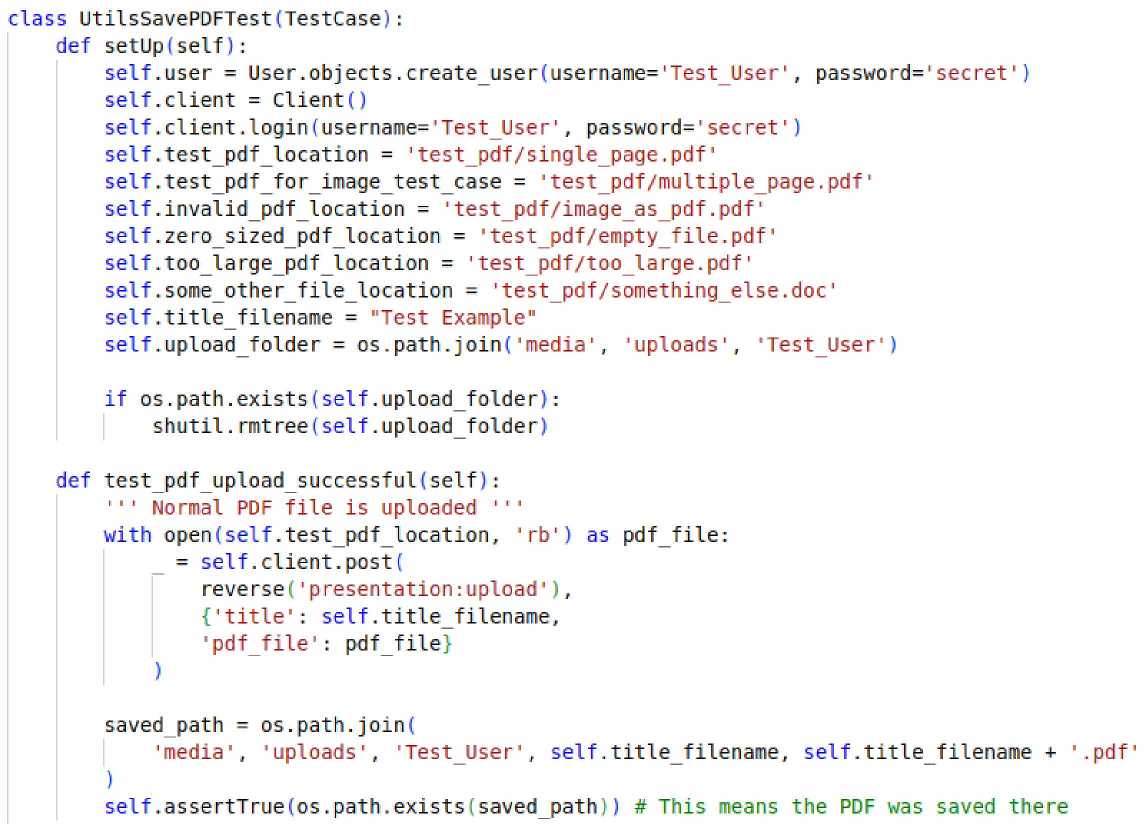
- *Error Constraints:*
  - If the file is not a PDF file, the function returns a *ValidationError* with the proper message.
  - If the file is an empty PDF file, the function returns a *ValidationError* with the proper message.
  - If the file is a too big PDF file, the function returns a *ValidationError* with the proper message.
- *Single Constraints:*
  - File must not be empty.
  - File must contain at least one page (i.e., be a valid PDF)
  - Title must be within allowed length
- *Conditional Constraints:*

– If title is provided and correct, the file must be provided as well.

– If the file fails validation (size, extension, or content), then it must not be processed or stored, regardless of title.

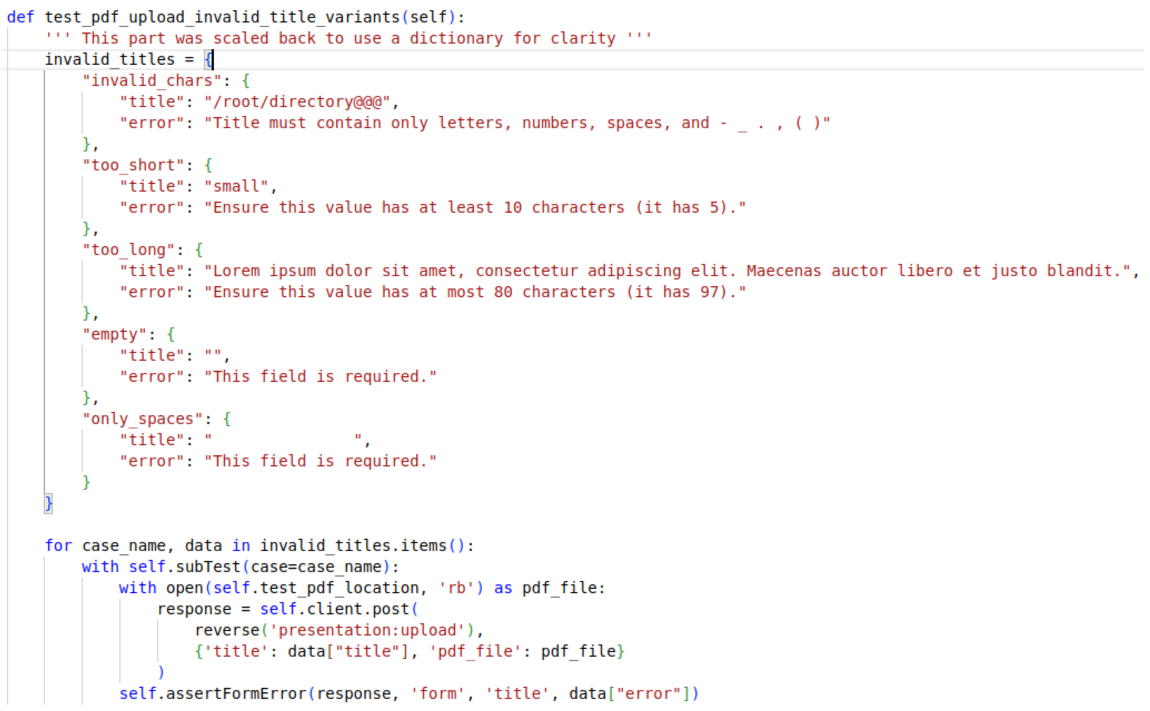
– Only authenticated users can upload files.

Step 5 – Generate Test Cases: Several test cases are generated from this particular analysis. The cases shown in this document, through Figure 6, and Figure 7 serve only as sample of the total of test cases.



```python
class UtilsSavePDFTest(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='Test_User', password='secret')
        self.client = Client()
        self.client.login(username='Test_User', password='secret')
        self.test_pdf_location = 'test_pdf/single_page.pdf'
        self.test_pdf_for_image_test_case = 'test_pdf/multiple_page.pdf'
        self.invalid_pdf_location = 'test_pdf/image_as_pdf.pdf'
        self.zero_sized_pdf_location = 'test_pdf/empty_file.pdf'
        self.too_large_pdf_location = 'test_pdf/too_large.pdf'
        self.some_other_file_location = 'test_pdf/something_else.doc'
        self.title_filename = "Test Example"
        self.upload_folder = os.path.join('media', 'uploads', 'Test_User')

        if os.path.exists(self.upload_folder):
            shutil.rmtree(self.upload_folder)

    def test_pdf_upload_successful(self):
        ''' Normal PDF file is uploaded '''
        with open(self.test_pdf_location, 'rb') as pdf_file:
            _ = self.client.post(
                reverse('presentation:upload'),
                {'title': self.title_filename,
                'pdf_file': pdf_file}
            )

        saved_path = os.path.join(
            'media', 'uploads', 'Test_User', self.title_filename, self.title_filename + '.pdf'
        )
        self.assertTrue(os.path.exists(saved_path)) # This means the PDF was saved there
```

Figure 6: System test for successful upload of a PDF file



```python
def test_pdf_upload_invalid_title_variants(self):
    ''' This part was scaled back to use a dictionary for clarity '''
    invalid_titles = {
        "invalid_chars": {
            "title": "/root/directory@@@",
            "error": "Title must contain only letters, numbers, spaces, and - _ . , ( )"
        },
        "too_short": {
            "title": "small",
            "error": "Ensure this value has at least 10 characters (it has 5)."
        },
        "too_long": {
            "title": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas auctor libero et justo blandit.",
            "error": "Ensure this value has at most 80 characters (it has 97)."
        },
        "empty": {
            "title": "",
            "error": "This field is required."
        },
        "only_spaces": {
            "title": "               ",
            "error": "This field is required."
        }
    }

    for case_name, data in invalid_titles.items():
        with self.subTest(case=case_name):
            with open(self.test_pdf_location, 'rb') as pdf_file:
                response = self.client.post(
                    reverse('presentation:upload'),
                    {'title': data["title"], 'pdf_file': pdf_file}
                )
            self.assertFormError(response, 'form', 'title', data["error"])
```

Figure 7: Testing example with filename errors, all these cases should raise exceptions within the Django Form framework

3.3.3 Exploratory testing applied to PDF file upload

Mission and Charter

- **Mission:** Assess the reliability, usability, and security of the file upload functionality.
- **Charter:**
  - Test standard and edge-case scenarios such as uploading valid/invalid files.
  - Explore validation behavior, feedback messages, and data flow.
  - Simulate user errors (e.g., uploading oversized or unsupported files).

Key Exploratory Testing Tours for File Upload

A. Business District Tour

- Upload a valid PDF file and ensure successful submission.
- Upload a non-PDF file (e.g., .doc, .jpg); verify appropriate rejection message.
- Check for maximum file size handling (e.g., over 5MB).

B. Guidebook Tour

- Follow any help text, UI hints, or tooltips for file upload. The upload form returns errors on red text whenever something bad occurs.
- Compare actual UI behavior with documentation or tutorial content. *There is no documentation yet.*
- Check whether instructions (e.g., Title input number of characters range) match validation behavior.

C. Fed-Ex Tour

- Observe how uploaded files are processed and stored.
- Check if metadata (e.g., file name, size) is handled correctly.
- Test submissions across browsers; ensure consistent handling and file integrity.

D. Historic District Tour

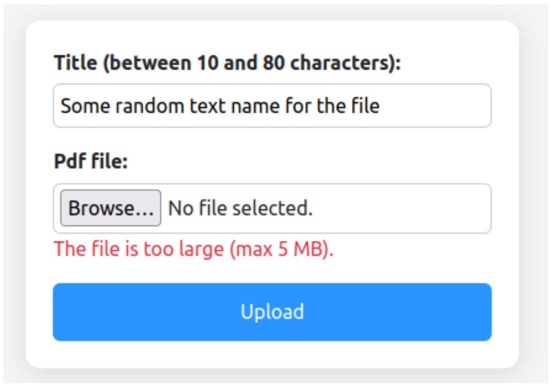
- Upload after reloading page; ensure session/state behaves as expected.

E. Seedy District Tour

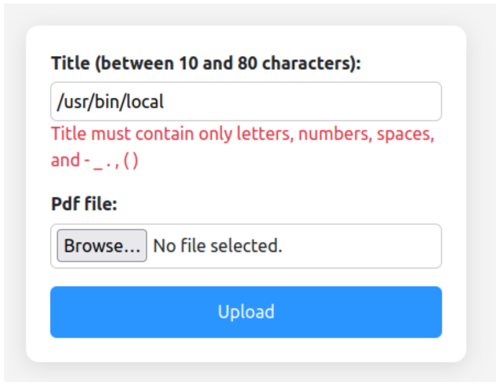
- Upload malformed or corrupted PDFs; check backend resilience.
- Attempt directory traversal via file names (e.g., `../../etc/passwd`).
- Try script injections in the file name or form title field.

Exploratory Test Session Example

Action	**Expected Result**	**Observed Behavior**
Upload valid PDF	Success message, file accepted	Success, file stored
Upload .doc file	Error message: invalid file type	Proper error displayed
Upload empty PDF	Error message: file is empty	Error shown
Upload over 5MB	Error message: file too large	File rejected. Figure 8a
Use filename with `../`	Error message: title is not correct	Proper error shown. Figure 8b


Title (between 10 and 80 characters):
Some random text name for the file
Pdf file:
Browse... No file selected.
The file is too large (max 5 MB).
Upload

(a) Uploading a too large PDF file.


Title (between 10 and 80 characters):
/usr/bin/local
Title must contain only letters, numbers, spaces, and - _ . , ( )
Pdf file:
Browse... No file selected.
Upload

(b) Using forbidden symbols in the filename.

Figure 8: Exploratory testing cases. (a) Oversized file. (b) Invalid filename.

Defects and Recommendations Issues to Detect

- Incomplete MIME/type validation or file sniffing.
- Lack of feedback on file processing errors.
- UI freezing or poor accessibility during large file upload.
- Path or filename vulnerabilities.

Improvement Suggestions

- Provide progress bars to accept bigger files.
- Validate file content type server-side, not just extension.
- Use title filename to check for document content validity.
- Accept other presentation files types (e.g. PPTX).

References

[1] Nancy Duarte. *Slide: ology: The art and science of creating great presentations.* O'Reilly Media, 2008.

[2] Leonardo Eras. Presentation app.

[3] Nurfazlina Haris, Sheela Faizura Nik Fauzi, Susana William Jalil, Muhammad Jastu Yusuf, and Asmahani Mahdi. Evaluating the efficacy of digital design platform in improving public speaking proficiencies. *International Journal of e-Learning and Higher Education (IJELHE)*, 20(1):137–155, 2025.

[4] HealthCentral. Glossophobia (fear of public speaking): Are you glossophobic?

[5] Yulieth Hoyos Vera et al. Hacia un sistema de reconocimiento del lenguaje corporal en presentaciones orales utilizando técnicas de machine learning. 2024.

[6] Guy Kawasaki. The 10/20/30 rule.

[7] Xavier Ochoa, Federico Domínguez, Bruno Guamán, Ricardo Maya, Gabriel Falcones, and Jaime Castells. The rap system: Automatic feedback of oral presentation skills using multimodal analysis and low-cost sensors. In *Proceedings of the 8th international conference on learning analytics and knowledge*, pages 360–364, 2018.

[8] World Wide Web Consortium (W3C). Relative luminance.

[9] World Wide Web Consortium (W3C). Success criterion 1.4.3 contrast (minimum).