Guide for the implementation of additional functionalities

This document will provide a general guide for the addition of new functionalities into the verification tool. A separate section for new tests, new pipelines for a supported format, the introduction of new formats and new extraction methods is present. Before that an explanation of the general function of the program is provided and is recommended to be read before the following sections. In addition a separate section detailing the inclusion of new settings will be provided in the end. It is highly recommended to inspect the code while reading, the explanation might be lacking in clarity or depth but will provide the outline.

General functionality:

The internal structure of the program consists of four parts - the *tests*, the *pipelines,* the *program manager* and the *logger*. *Tests* are individual functions that compare or analyze files. For example, a test comparing the page count of documents is present in the program.

These tests serve as building blocks of the *pipelines*, which are specific functions consisting primarily of tests that facilitate the verification process between two files. Pipelines are implemented for comparison between two formats, for example a DOCX-PDF pipeline is present in the program. Note that pipelines are assigned based on PRONOM codes and not the file extensions.

The *program manager* serves as the element starting, controlling and monitoring the verification process. It is the program manager that is responsible for starting the pipelines. The functionality for data extraction has a mirroring implementation, but regarding only single files instead of pairs.

Inside the pipelines the test results are analyzed and recorded in the *logger*. Logger produces reports of the verification process. Test results are recorded either as failures or successes, with an error object passed in cases of failures.

Adding new test:

The addition of new tests is the easiest expansion of the tool. To add a new test, first simply create the desired test function. These usually are placed in the ComparingMethods folder. The function should return the result of the test, this can either be just the bool representing whether the test failed or not, or general results for analysis in the pipeline itself. In case of the test crashing or failing *null* is typically returned.

```
if(GlobalVariables.Options.GetMethod(Methods.MethodName))
{
    bool? res = ComparisonMethods.MethodName();

    if(res == null) { //Null, i.e. test execution failure
        compResults.AddTestResult(Methods.MethodName, false,
        comment: ["Could not execute MethodName."]);
    }
    else if(res.Value) { //True, i.e. test pass success
        compResults.AddTestResult(Methods.MethodName, true);
    }
    else { //False, i.e. test pass failure
        compResults.AddTestResult(Methods.MethodName, false,
            errors: [
                new Error(
                    "Method Name has failed",
                    "The test MethodName failed because...",
                    ErrorSeverity.High,
                    ErrorType.Visual,
                )
            ]
        );
    }
}
```

Once the test has been implemented it is ready to be added to the pipelines. Select the relevant pipeline function present in one of the files inside the

ComparisonPipelines folder. Other tests are already present there, and continuation of the pattern used is recommended. First, if the test is configurable, check if it is enabled in settings, then call the test and then analyze the test results. Based on the outcome, record the appropriate result in the same manner as other tests, by adding a test result. An example with the “MethodName” test is provided.

Adding new pipelines:

The addition of a new pipeline for a supported format is a two-step task, consisting first of the creation of the pipeline function and then of adding it to the selection.

Locate the file containing the pipelines for the desired format. Inside functions following the [*original format*]To[*new format*]Pipeline naming convention will be present. It is recommended to follow the naming convention. All pipelines must take the same parameters. Create the desired pipeline function ensuring the parameters are; a FilePair object, int, Action<int> and Action, the meaning behind the parameters is explained in code comments. The content of the pipeline function should consist of:

1. Execution of the ExecutePipeline function. This function centralizes the error handling, meaning it does not have to be handled in your implementation. The first parameter passed to this function should be the main logic of the pipeline as a lambda function, here all the tests should take place. For the rest of the parameters refer to comments in the code. Examine the implementation of other pipelines, or the image below, for reference and clarification.
2. The creation of a ComparisonResult object inside the function. This object is used to store test records before they are written to the log.
3. Ensure that at the end of the function the ComparisonResult is added to the logger.

```
//Ensure the correct parameters!
private static void OldFormatToNewFormatPipeline(FilePair pair, int additionalThreads,
                                                 Action<int> updateThreadCount, Action markDone)
{
    //Execute ExecutePipeline
    BasePipeline.ExecutePipeline(() =>
    {
        var compResult = new ComparisonResult(pair); //Creating compResults

        //Here goes the pipeline content, i.e. the tests that fill up the compResults

        GlobalVariables.Logger.AddComparisonResult(compResult); //Adding results to logger

    }, [pair.OriginalFilePath, pair.NewFilePath], additionalThreads, updateThreadCount, markDone);
    //  ^-- Pass the parameters like this, info on why is in the code
}
```

NOTE: If temporary folders are created (such as in DocxToTextDocPipeline function) remember to delete them!

After the pipeline function has been implemented it has to be added to the selection.

Locate the Get[*format*]Pipeline function, which is usually the first function in the file. This function consists of conditional statements which based on the PRONOM code return the correct pipeline function. The selection function needs to *return*, not execute, the pipeline function. Note that the PRONOM codes are organized in lists in the FormatCodes class. Remember that the PRONOM code of the desired format could already be present in

another list, for example if a separate pipeline for the PDF/A2 format is desired its PRONOM code will already be contained in the PDF code list. Avoid removing codes, as it will affect the rest of the application. Simply place the new pipeline first in the selection.

```
//outputFormat is the PRONOM code of the new file. It is used to determine the pipeline.
public static Action<FilePair>, int, Action<int> GetOldFormatPipeline(string? outputFormat)
{
    if(FormatCodes.PronomCodesSomeFormat.Contains(outputFormat))
        return OldFormatToSomeFormatPipeline;

    if(FormatCodes.PronomCodesNewFormat.Contains(outputFormat))
        return OldFormatToNewFormatPipeline; //Our new pipeline added to selection

    return null;
}
```

Adding new formats:

Addition of a new format can either mean the inclusion of a new PRONOM code for an existing format (e.g. adding a brand new PDF format to the list) or the addition of a completely new file format with its PRONOM codes. Adding a new PRONOM code for an existing format is easy - simply navigate to the FormatCodes class, find the object storing PRONOM codes for the format and include it in the list. The addition of a completely new format will require additional steps:

1. Create a new FileFormat object in the FormatCodes class. It may also be included in relevant lists (e.g. text document list).
2. Create a new pipeline file. The creation of pipelines is described in the relevant section - create a file in ComparisonPipelines and create both the GetPipeline and pipeline functions for the new format.
3. Before being sent into the GetPipeline function, which determines the pipeline based on the PRONOM code of the new (i.e. converted) file, another function finds the correct GetPipeline function based on the PRONOM code of the original file (i.e. the original pre-conversion file). The new GetPipeline function needs to be added there. This function is present in the BasePipeline file and is called SelectPipeline and has the same structure as the GetPipeline function.

```
//Make sure to returns and not executes the GetPipeline function!
public static Action<FilePair, int, Action<int>, Action>? SelectPipeline(FilePair pair)
{
    if(FormatCodes.PronomCodesSomeFormat.Contains(pair.OriginalFileFormat))
        return SomeFormatPipelines.GetSomeFormatPipeline(pair.NewFileFormat);

    //Our new entry
    if(FormatCodes.PronomCodesNewFormat.Contains(pair.OriginalFileFormat))
        return NewFormatPipelines.GetSomeFormatPipeline(pair.NewFileFormat);

    return null;
}
```

Adding new extraction methods and pipelines:

The data extraction process uses separate pipelines and has a separate program manager. The implementation of data extraction directly mirrors the implementation of the normal verification process. Extraction methods are stored in the ExtractionMethods file and

pipelines in the BaseExtraction file. The addition of either happens for the most part in the same way as it would for normal verification, with the exception being the different parameters (due to it working with single files) and that pipelines return results. Pipelines during execution gather information and create a string-to-string directory storing the information, this directory is returned and represents the general result of the extraction process, and is what is written in the report. Extraction does not use the logger.

```
//Note the different parameters.
public static Dictionary<string, string>? NewExtractionPipeline(SingleFile file, Action updateThreads, Action markDone)
{
    //Again calling to ExecutePipeline (not the same one as in verification) to centralize error handling
    Dictionary<string, string>? res = ExecutePipeline(() => {
        Dictionary<string, string> results = new(); //Initializing the results

        int testResult = ExtractionMethods.ExampleTest(file);
        results["ExampleTest"] = $"{testResult}"; //Writing values

        //More extraction goes here

        return results; //Returning the from ExecutePipeline
    }, file.FilePath, updateThreads, markDone);

    return res; //Returning the results
}
```

Other than that, no differences exist and therefore no further details will be discussed.

Settings

Settings are stored in the Options class, located in FileVerifier/src/Options/Options.cs. The Options class is responsible only for the storing of Options values, so applying them will have to be done manually. Upon program start, an instance of the Options class will be created in GlobalVariables, which can be referred to by ‘GlobalVariables.Options’.

Additionally, new settings have to be implemented in the GUI in FileVerifier/Views/SettingsView.axaml, and linked to the FileVerifier/ViewModels/SettingsWindowViewModel.cs. The settings values are also loaded into the GUI in Synchronize() in FileVerifier/Views/SettingsView.axaml.cs.