High-level Design

- We are using the Pipe and Filter Pattern for our structure. We use this to get more data in between each 'pipe'. For example, we may need to check what format the code is in, then check what language the code is in, then start processing and understanding, then documenting.
- The filters we want to implement are:
    - Code Entry: Where code is analyzed first and is responsible for reading source code files from various inputs such as local/remote directories and repositories.
    - Language Detector: Next, we identify what programming language it is written in. The code is now labeled for the next step of processing.
    - Parser: Based on what language the code is in, this filter parses through the code into a formatted structure where each function and class are split up and grouped in likeness.
    - Analyzer: The analyzer then can check for any previous commented material and then prepare for more documentation.
    - Documentation Generator: Here, we finally generate all the documentation needed while keeping previous documentation.
    - Output Manager: Pipe and Filter patterns can be rough to output files by themselves, so this output manager helps handle all output code, formats it properly, and makes sure it is saved in case of any error.
- Extensibility: We want to add more filters when considering adding new languages or more formats besides HTML or PDF. These filters can be added effectively and easily making for a highly extensive system.
- Scalability: There can be a lot of scaling with this program as there can be multiple specific filters that can be deployed. The Parser/Analyzer filters may need to be enhanced to do so.
- Error Handling: The program will be implemented with error-handling filters. These filters would be mainly placed at the start and the parsing filters so that there is no loss or false code.

Low-level Design

We will be using the Factory Method design pattern for AutoDoc. This method is the best for AutoDoc because it enhances flexibility and scalability in creating documentation for various programming languages, which is what our tool aims to do. This pattern is a part of the creational pattern family, which focuses on object creation - trying to create objects in a manner suitable to the situation.

Factory Method Pattern: The factory method pattern defines an interface for creating objects, but it allows subclasses to decide which class to instantiate. For the AutoDoc app, a DocumentationGenerator could serve as the main layer that defines a method for creating documentation generator instances. Subclasses of this factory would then implement the instantiation of specific documentation generators, such as PythonDocGenerator or JavaDocGenerator, based on the programming language selected by the user. This approach would allow easy integration for new languages. All that would need to be changed would be extending the factory with new subclasses, which follows the Open/Closed Principle (open for extension, closed for modification).

Benefits: With using the Factory Method pattern, our application is flexible when adding new changes. We can add additional features without having to refactor our whole codebase. This allows for people who haven't been working on the codebase through its entirety to be able to understand what is going on and maintain the codebase.

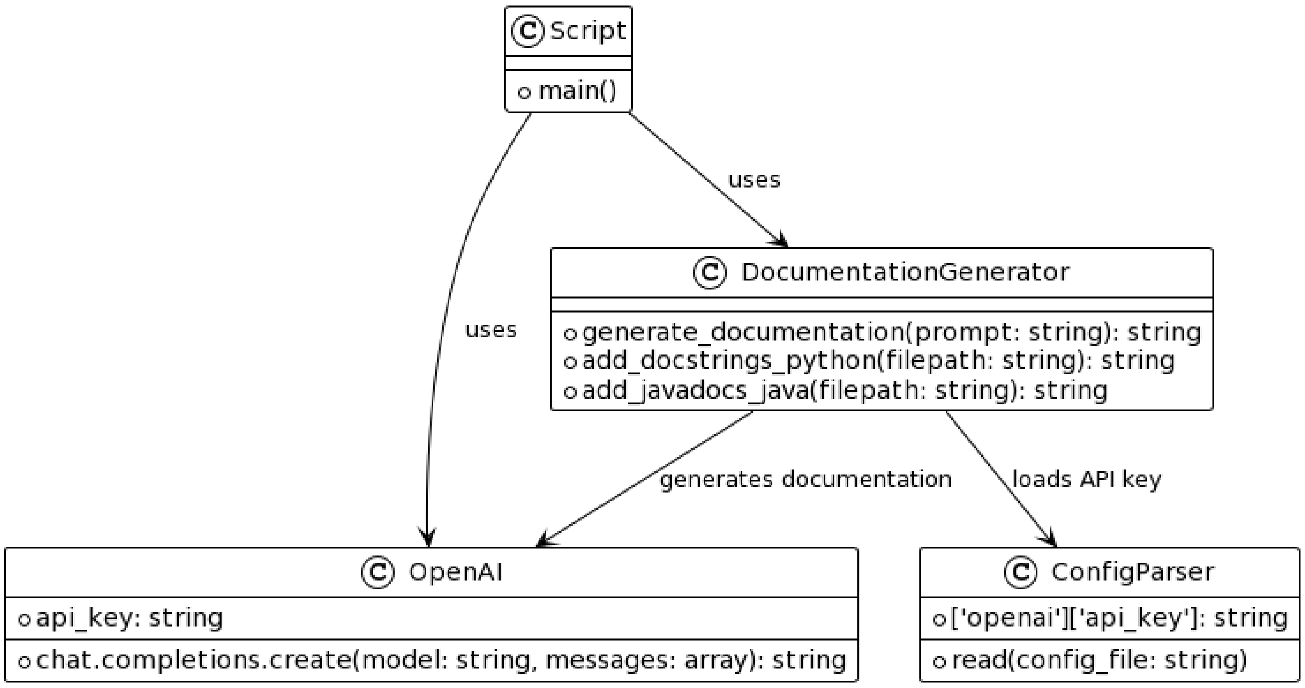
Application: Because we first ask the user to select the language, then utilize the OpenAI's API to generate comments, we can easily add more languages without having to rework the current code. This exemplifies the concept of encapsulation because the codebase is flexible and scalable and allows for new changes.

Pseudocode Representation –

```
Program AutoDocApp

Class ConfigParser
  Function read(config_file: String)
    // Load configuration file and return configuration settings
  End Function
End Class

Class OpenAI
  Variable api_key as String

  Constructor(api_key: String)
    // Initialize OpenAI client with API key
    This.api_key = api_key
  End Constructor

  Function chat_completions_create(model: String, messages: Array)
    // Make an API call to OpenAI to generate documentation based on the provided prompt
    // Return generated documentation
  End Function
End Class

Class DocumentationGenerator
  Variable client as OpenAI

  Constructor(api_key: String)
    // Initialize DocumentationGenerator with an OpenAI client
    This.client = new OpenAI(api_key)
  End Constructor

  Function generate_documentation(prompt: String)
    // Use OpenAI client to generate documentation for the given prompt
    // Return documentation or error message
  End Function

  Function add_docstrings_python(filepath: String)
    // Read Python file, generate and insert docstrings
    // Save the modified file
  End Function

  Function add_javadocs_java(filepath: String)
    // Read Java file, generate and insert Javadocs
    // Save the modified file
  End Function
End Class

Class Script
  Function main()
    // Load API key using ConfigParser
    config = new ConfigParser()
    api_key = config.read("config.ini")['openai']['api_key']

    // Initialize documentation generator with API key
    docGenerator = new DocumentationGenerator(api_key)

    // Determine language and filepath from command line arguments
    // language, filepath = get_command_line_arguments()
```

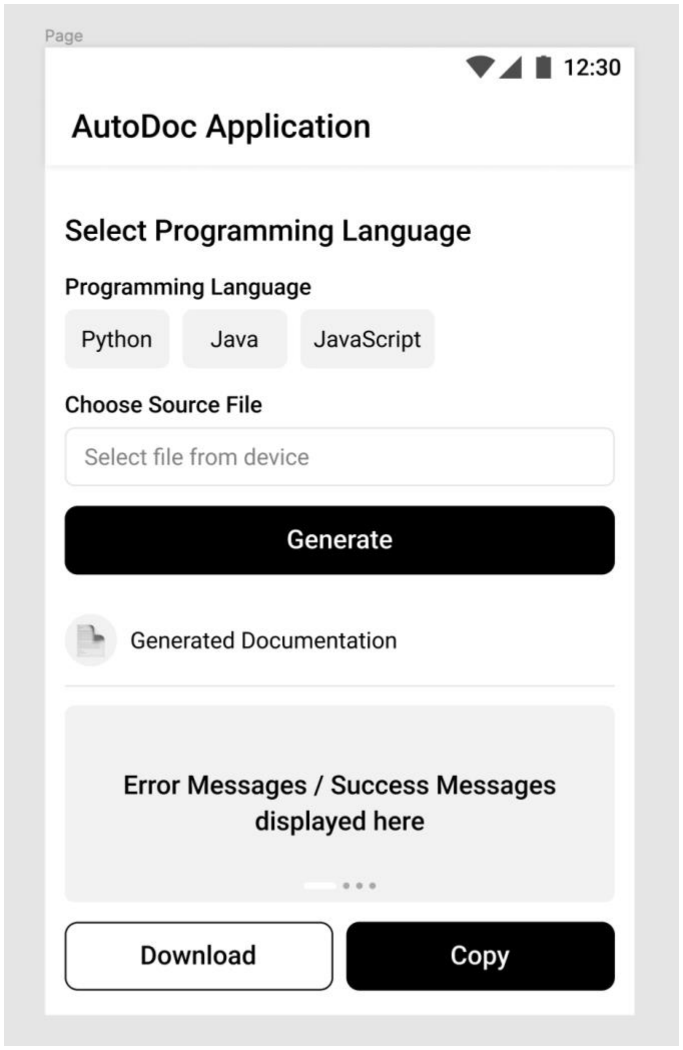
```
        // Generate and insert documentation based on the language
        If language == "python" Then
            docGenerator.add_docstrings_python(filepath)
        Else If language == "java" Then
            docGenerator.add_javadocs_java(filepath)
        Else
            Print("Unsupported language. Only Python and Java are supported.")
        End If

        Print("Documented file created.")
    End Function
End Class

End Program
```

Informal Class Diagram -


Script
main()
uses
uses
DocumentationGenerator
generate_documentation(prompt: string): string
add_docstrings_python(filepath: string): string
add_javadocs_java(filepath: string): string
generates documentation
loads API key
OpenAI
api_key: string
chat.completions.create(model: string, messages: array): string
ConfigParser
['openai']['api_key']: string
read(config_file: string)

Design Sketch

The mockup for our user interface is pretty simple. This user view includes first the types of programming language that the code is written in. Then, the user would choose the actual file by uploading it. The application/website could return the file with comments appended inside the file. From that, the user would be able to either copy the contents or download the entire file.



Process Deliverable – Code and Fix

GitHub - https://github.com/rudyP123/CS3704-DocDoctors

```
# How to use
# python doc_generator.py python path/to/your_file.py
# python doc_generator.py java path/to/your_file.java
```

```python
import ast
import sys
import os
import configparser
from openai import OpenAI

def load_api_key(config_file='config.ini'):
    config = configparser.ConfigParser()
    config.read(config_file)
    api_key = config['openai']['api_key']
    return api_key

client = OpenAI(api_key=load_api_key())

def generate_documentation(prompt):
    try:
        response = client.chat.completions.create(model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ])
        return response.choices[0].message.content.strip()
    except Exception as e:
        print(f"An error occurred: {e}")
        return "Documentation generation failed."

def add_docstrings_python(filepath):
    output_filepath = f"{os.path.splitext(filepath)[0]}_documented{os.path.splitext(filepath)[1]}"
    with open(filepath, "r") as file:
        lines = file.readlines()

    def get_function_start_line_numbers(lines):
        function_start_lines = []
        for i, line in enumerate(lines):
            if line.strip().startswith("def "):
                function_start_lines.append(i)
        return function_start_lines
    function_start_lines = get_function_start_line_numbers(lines)
    for start_line in function_start_lines:
        function_lines = []
        for line in lines[start_line:]:
            function_lines.append(line)
            if line.strip().startswith("def ") or "return" in line:
                break
        function_text = "".join(function_lines)
        description = "Provide a concise summary for this Python function."
        docstring = generate_documentation(description)
        docstring_formatted = f'"""{docstring}"""\n'
        lines.insert(start_line + 1, docstring_formatted)

    new_source = "".join(lines)
    with open(output_filepath, "w") as file:
        file.write(new_source)

    return output_filepath


def add_javadocs_java(filepath):
    """Reads a Java file, adds Javadoc comments to methods."""
    output_filepath = f"{os.path.splitext(filepath)[0]}_documented{os.path.splitext(filepath)[1]}"

    with open(filepath, "r") as file:
        lines = file.readlines()

    new_lines = []
    for i, line in enumerate(lines):
        if (line.strip().startswith(("public", "private", "protected")) and "(" in line and ")" in line and "{" in line
                and not any(keyword in line for keyword in ["class ", "=", ";"])):
            method_signature = line.strip()
            description = f"""Write a detailed but concise Javadoc comment for the following
            Java method: {method_signature}"""
            javadoc = generate_documentation(description)
            new_lines.append(javadoc + "\n")
        new_lines.append(line)

    with open(output_filepath, "w") as file:
        file.writelines(new_lines)
    return output_filepath

def main():
    if len(sys.argv) < 3:
        print("Usage: script.py <language> <input_file_path>")
        sys.exit(1)

    language = sys.argv[1].lower()
    input_filepath = sys.argv[2]
    output_filepath = ""

    if language == "python":
        output_filepath = add_docstrings_python(input_filepath)
    elif language == "java":
        output_filepath = add_javadocs_java(input_filepath)
    else:
        print("Unsupported language. Only Python and Java are supported.")

    print(f"Documented file created: {output_filepath}")

if __name__ == "__main__":
    main()
```