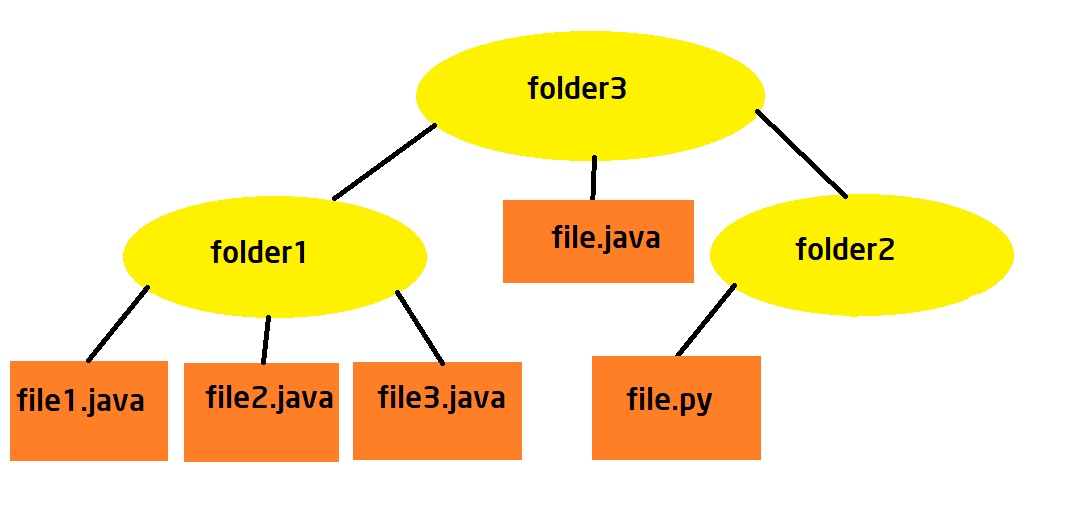
**Automating Code Documentation with GPT-3.5**

**Intro**

Goal: Generating a general description/documentation for the top directory in a source code repository

How: python + GPT 3.5

**Algorithm Explanation**

****

*FIGURE 1- TREE-LIKE PROJECT MODEL*

The algorithm loops over the source code repository (also called “codebase”) from the bottom of the structure up to the high level directory for which we want to create a general description/documentation.

Specifically, the algorithm starts by reading the code content from individual files within the codebase (in this case file.java, file1.java etc)

It uses the OpenAI GPT-3.5 model to generate descriptions for each code file.

The descriptions of the files are accumulated within the directory containing them files.

Then, going up in the structure, it generates descriptions for directories containing only files (no subdirectories)

It then utilizes the OpenAI model to generate a description for the directory, taking into account the descriptions of the individual files it contains.

The goal is to create a coherent and contextually relevant summary of the directory's purpose based on the content of its files.

For directories containing both files and subdirectories, the algorithm recursively analyzes the contents.

It generates descriptions for subdirectories by considering the descriptions of the files and subdirectories within them.

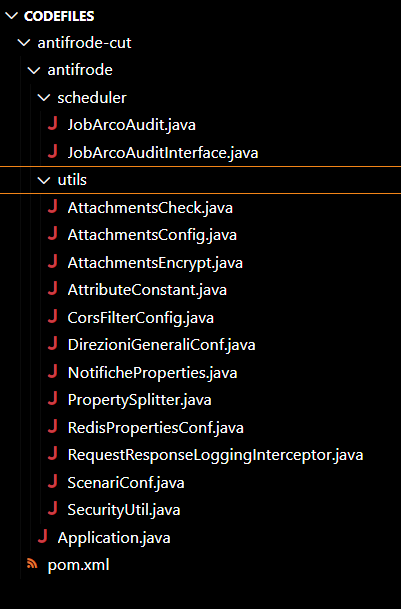
This recursive process ensures that the generated descriptions provide a holistic understanding of the structure and purpose of each directory.

The algorithm automatically saves the generated descriptions to text files.

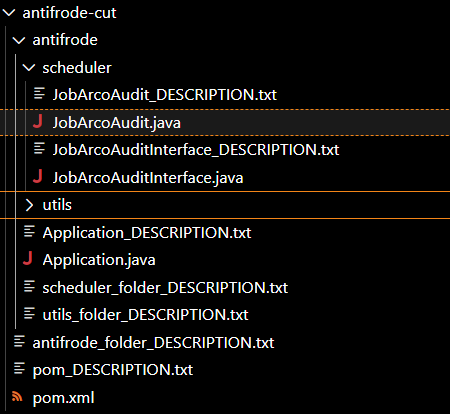
Each description is stored in a separate file, typically with "\_DESCRIPTION.txt" appended to the original file or directory name.

Sample Project

I tested this python script to this partial project after placing it in the same path of the high level folder for which I aim to generate the documentation/description



*FIGURE 2 - SOURCE CODE BEFORE ELABORATION*

**

*FIGURE 3 – GENERATED DESCRIPTIONS AFTER ELABORATION*

AI-prompts used

For generating file descriptions based on the code:

***“Describe the purpose of this code: “.***

For generating folder (containing only files) descriptions based on the descriptions of the files contained within it:

***“Describe the purpose or function of this directory in a codebase, considering the descriptions of the files contained within this directory. Here are the descriptions of those files:”***

For generating folder (containing both files and subfolders) descriptions based on the descriptions of the files and subfolders contained within it:

***“Describe the purpose or function of this directory in a codebase, considering the descriptions of the files and the folders contained within this directory. Here are the descriptions of them:”***

Descriptions of files and folders

# directory antifrode-cut

The purpose or function of this directory in the codebase is to contain code related to the scheduling and execution of tasks for the ArcoAudit system. It includes an interface called JobArcoAuditInterface, which defines methods for importing, updating, and deleting records in the ArcoSegnalazioni database table. The directory also contains a file called JobArcoAudit, which implements the functionality for scheduling and executing these tasks. This file uses Spring's scheduling functionality and includes methods for calling external services and processing retrieved data. Overall, this directory is responsible for managing the periodic tasks and data operations related to the ArcoAudit system.

Additionally, this directory contains utility classes and configuration classes for various functionalities such as attachment management, encryption, CORS filtering, configuration management, logging, scenario management, and security/authentication. These classes provide methods and properties for performing checks on attachments, managing attachment configurations, encrypting and decrypting attachments, defining constant values for specific attributes, configuring CORS filters, storing and accessing lists of objects, managing BPMN notifications, splitting and mapping properties, managing Redis properties, implementing logging interceptors for HTTP requests and responses, managing scenarios, and providing security and authentication-related utility methods and properties.

The directory also contains a file called pom.xml, which is used to configure the Maven build for a Spring Boot project. It specifies the project's dependencies, plugins, and other settings. It also includes profiles for different environments (coll and dev) that can be activated based on the project's needs. The code in this file also includes the configuration for generating server stubs using OpenAPI specifications.

------------------------------------------------------------------------------------------------------------------------------------

# directory antifrode-cut\antifrode

The purpose or function of this directory in the codebase is to contain code related to the scheduling and execution of tasks for the ArcoAudit system. It includes an interface called JobArcoAuditInterface, which defines methods for importing, updating, and deleting records in the ArcoSegnalazioni database table. The directory also contains a file called JobArcoAudit, which implements the functionality for scheduling and executing these tasks. This file uses Spring's scheduling functionality and includes methods for calling external services and processing retrieved data. Overall, this directory is responsible for managing the periodic tasks and data operations related to the ArcoAudit system.

Additionally, this directory contains utility classes and configuration classes for various functionalities such as attachment management, encryption, CORS filtering, configuration management, logging, scenario management, and security/authentication. These classes provide methods and properties for performing checks on attachments, managing attachment configurations, encrypting and decrypting attachments, defining constant values for specific attributes, configuring CORS filters, storing and accessing lists of objects, managing BPMN notifications, splitting and mapping properties, managing Redis properties, implementing logging interceptors for HTTP requests and responses, managing scenarios, and providing security and authentication-related utility methods and properties.

# file pom.xml

The purpose of this code is to configure the Maven build for a Spring Boot project. It specifies the project's dependencies, plugins, and other settings. It also includes profiles for different environments (coll and dev) that can be activated based on the project's needs. The code also includes the configuration for generating server stubs using OpenAPI specifications.

# file Application.java

The purpose of this code is to configure and start a Spring Boot application. It includes the necessary annotations and configurations to run the application, as well as a bean definition for a Jackson JSON converter. The Jackson converter is configured to handle serialization and deserialization of JSON data, with specific features such as ignoring unknown properties and not failing on empty beans.

# directory antifrode-cut\antifrode\scheduler

The purpose or function of this directory in the codebase is to contain code related to the scheduling and execution of tasks for the ArcoAudit system. It includes an interface called JobArcoAuditInterface, which defines methods for importing, updating, and deleting records in the ArcoSegnalazioni database table. The directory also contains a file called JobArcoAudit, which implements the functionality for scheduling and executing these tasks. This file uses Spring's scheduling functionality and includes methods for calling external services and processing retrieved data. Overall, this directory is responsible for managing the periodic tasks and data operations related to the ArcoAudit system.

# directory antifrode-cut\antifrode\utils

The purpose or function of this directory in the codebase is to provide utility classes and configuration classes related to various functionalities.

1. AttachmentsCheck.java: This code provides a utility class for performing checks on attachments, such as checking the name, content, description, and type of a document attachment. It also checks if a process instance already has an attachment with a given name.

2. AttachmentsConfig.java: This code defines a configuration class for managing attachments in an application. It includes properties such as the maximum file size, maximum file name length, maximum description length, maximum length for a document type, a list of allowed file extensions, and allowed patterns for file names and descriptions.

3. AttachmentsEncrypt.java: This code provides utility methods for encrypting and decrypting attachments. It includes methods for encrypting and decrypting data using a specified algorithm, key, and initialization vector (IV). It also includes methods for generating a secret key from a password and salt, and for generating an IV.

4. AttributeConstant.java: This code defines an enumeration that represents a set of constant values for specific attributes. Each constant value has an overridden "toString()" method that returns a string representation of the attribute.

5. CorsFilterConfig.java: This code defines a configuration class for a CORS (Cross-Origin Resource Sharing) filter. It specifies a list of allowed origins for cross-origin requests.

6. DirezioniGeneraliConf.java: This code defines a configuration class for storing and accessing a list of "DirezioneGenerale" objects. It provides a method to retrieve the description of a "DirezioneGenerale" object based on its unique identifier.

7. NotificheProperties.java: This code defines a configuration class for storing and retrieving a map of BPMN notifications.

8. PropertySplitter.java: This code provides utility methods for splitting and mapping properties.

9. RedisPropertiesConf.java: This code defines a configuration class for Redis properties related to antifraud functionality. It sets the time-to-live (TTL) values for different cache scenarios.

10. RequestResponseLoggingInterceptor.java: This code implements a logging interceptor for HTTP requests and responses. It logs the details of the requests and responses, including the URI, method, headers, and body.

11. ScenariConf.java: This code defines a utility class for managing a list of scenarios. It provides methods to retrieve the code or value of a scenario based on the other, and vice versa.

12. SecurityUtil.java: This code provides utility methods and properties related to security and authentication. It includes methods for retrieving user information, checking user roles, and retrieving ACL for a specific task.

Overall, this directory contains utility classes and configuration classes for various functionalities such as attachment management, encryption, CORS filtering, configuration management, logging, scenario management, and security/authentication.

# file JobArcoAuditInterface.java

The purpose of this code is to define an interface called JobArcoAuditInterface. This interface contains three methods: importScheduled(), updateScheduled(), and deleteScheduled(). These methods are intended to be implemented by classes that need to perform periodic tasks related to importing, updating, and deleting records in a database table called ArcoSegnalazioni.

# file JobArcoAudit.java

The purpose of this code is to schedule and execute various tasks related to the ArcoAudit system. It imports data from ArcoAudit, updates information in the ArcoSegnalazioni table, and deletes obsolete records from the ArcoSegnalazioni table. The code uses Spring's scheduling functionality to execute these tasks periodically. It also includes methods to call various external services and retrieve data from them. The retrieved data is then processed and stored in the database.

# file AttachmentsCheck.java

The purpose of this code is to provide a utility class called "AttachmentsCheck" that is used to perform various checks on attachments. The class has methods to check the name, content, description, and type of a document attachment. It also has a method to check if a process instance already has an attachment with a given name. The class uses various configurations and services to perform these checks and returns an outcome object indicating the result of the checks.

# file AttachmentsConfig.java

The purpose of this code is to define a configuration class for managing attachments in an application. It includes properties such as the maximum file size, maximum file name length, maximum description length, maximum length for a document type, a list of allowed file extensions, and allowed patterns for file names and descriptions. This class is annotated with Spring annotations to indicate that it is a component and to configure it with properties from a configuration file.

# file AttachmentsEncrypt.java

The purpose of this code is to provide utility methods for encrypting and decrypting attachments. It includes methods for encrypting and decrypting data using a specified algorithm, key, and initialization vector (IV). It also includes a method for generating a secret key from a password and salt, and a method for generating an IV. The code handles exceptions and logs any errors that occur.

# file AttributeConstant.java

The purpose of this code is to define an enumeration called "AttributeConstant" that represents a set of constant values. Each constant value represents a specific attribute and has an overridden "toString()" method that returns a string representation of the attribute. The constants defined in this enumeration are "AREE", "COMPETENZA", and "US", which represent the attributes "Aree", "Competenza", and "United States" respectively.

# file CorsFilterConfig.java

The purpose of this code is to define a configuration class for a CORS (Cross-Origin Resource Sharing) filter. It is used to specify a list of allowed origins for cross-origin requests. The class is annotated with the Spring `@Component` annotation to indicate that it is a Spring bean and can be autowired into other components. The `@Data` annotation is from the Lombok library and automatically generates getter and setter methods for the class fields. The `@FieldDefaults` annotation sets the access level of the fields to private. The `@Value` annotation is used to inject the value of the `antifrode.corsfilter.allowedOrigins` property from the application's configuration file into the `allowedOrigins` field. The class has a default constructor that does not have any implementation.

# file DirezioniGeneraliConf.java

The purpose of this code is to define a configuration class called "DirezioniGeneraliConf" that is used to store a list of "DirezioneGenerale" objects. This class is annotated with Spring's "@Component" annotation, indicating that it is a Spring-managed component. It is also annotated with "@ConfigurationProperties" to specify that it should be configured using properties with the prefix "inail-static.table".

The "DirezioniGeneraliConf" class contains a method called "getById" that takes a "codSede" parameter and returns the description of the "DirezioneGenerale" object with the matching "codSede" value. If no matching object is found, it throws a "ResourceNotFoundException".

The "DirezioneGenerale" class is a nested static class within the "DirezioniGeneraliConf" class. It represents a specific "DirezioneGenerale" object and contains fields for "codSede" (a unique identifier), "descrizione" (a description), and "regione" (a region). It also provides getter and setter methods for these fields.

Overall, this code provides a way to configure and access a list of "DirezioneGenerale" objects, and retrieve the description of a specific object based on its unique identifier.

# file NotificheProperties.java

The purpose of this code is to define a configuration class called "NotificheProperties" that is used to store and retrieve a map of BPMN notifications. This class is annotated with @ConfigurationProperties to indicate that it is a configuration class and the prefix "flowable" is used to bind the properties from the application configuration file. The class has a constructor, getter, and setter methods for the "notificheBpmn" map.

# file PropertySplitter.java

The purpose of this code is to provide utility methods for splitting and mapping properties. It includes methods for splitting properties into lists or maps, and for grouping lists of properties. The code uses the Google Guava library's Splitter class to perform the splitting and mapping operations.

# file RedisPropertiesConf.java

The purpose of this code is to define a configuration class for Redis properties related to antifraud functionality. It is used to set the time-to-live (TTL) values for different cache scenarios. The class is annotated with various Spring and Lombok annotations to enable automatic configuration and dependency injection. The class has four properties: defaultValue, fallBack, notFound, and jwt, which represent the TTL values for different cache scenarios. The class also has a default constructor.

# file RequestResponseLoggingInterceptor.java

The purpose of this code is to implement a logging interceptor for HTTP requests and responses. It is used to log the details of the requests and responses, including the URI, method, headers, and body. This can be useful for debugging and troubleshooting purposes. The code defines a class called "RequestResponseLoggingInterceptor" that implements the "ClientHttpRequestInterceptor" interface. It overrides the "intercept" method to log the request and response details before and after the execution of the request. The logging is done using the SLF4J logging framework.

# file ScenariConf.java

The purpose of this code is to define a utility class called "ScenariConf" that is used to manage a list of scenarios.

The class has the following functionalities:

1. It stores a list of scenarios as instances of the inner class "Scenario".

2. It provides methods to retrieve the codice (code) of a scenario based on its valore (value) and vice versa.

3. The method "getByValue" takes a value as input and iterates through the list of scenarios to find a matching scenario. If a match is found, it returns the codice of the scenario. If no match is found, it returns 0L.

4. The method "getByCodice" takes a codice as input and uses a stream operation to filter the list of scenarios based on the codice. If a matching scenario is found, it returns its valore. If no match is found, it throws a "ResourceNotFoundException" with a specific error message.

5. The class also provides getters and setters for the codice and valore properties of the Scenario class.

Overall, this code provides a convenient way to manage and retrieve information about scenarios based on their codice or valore.

# file SecurityUtil.java

The purpose of this code is to provide utility methods and properties related to security and authentication. It includes methods for retrieving the current username, checking user roles, retrieving user details and token information, and getting the list of authorities assigned to the current user. It also includes a method for retrieving the ACL (Access Control List) for a specific task. The code is part of a larger application that uses the Flowable engine for workflow management and Spring Security for authentication and authorization.