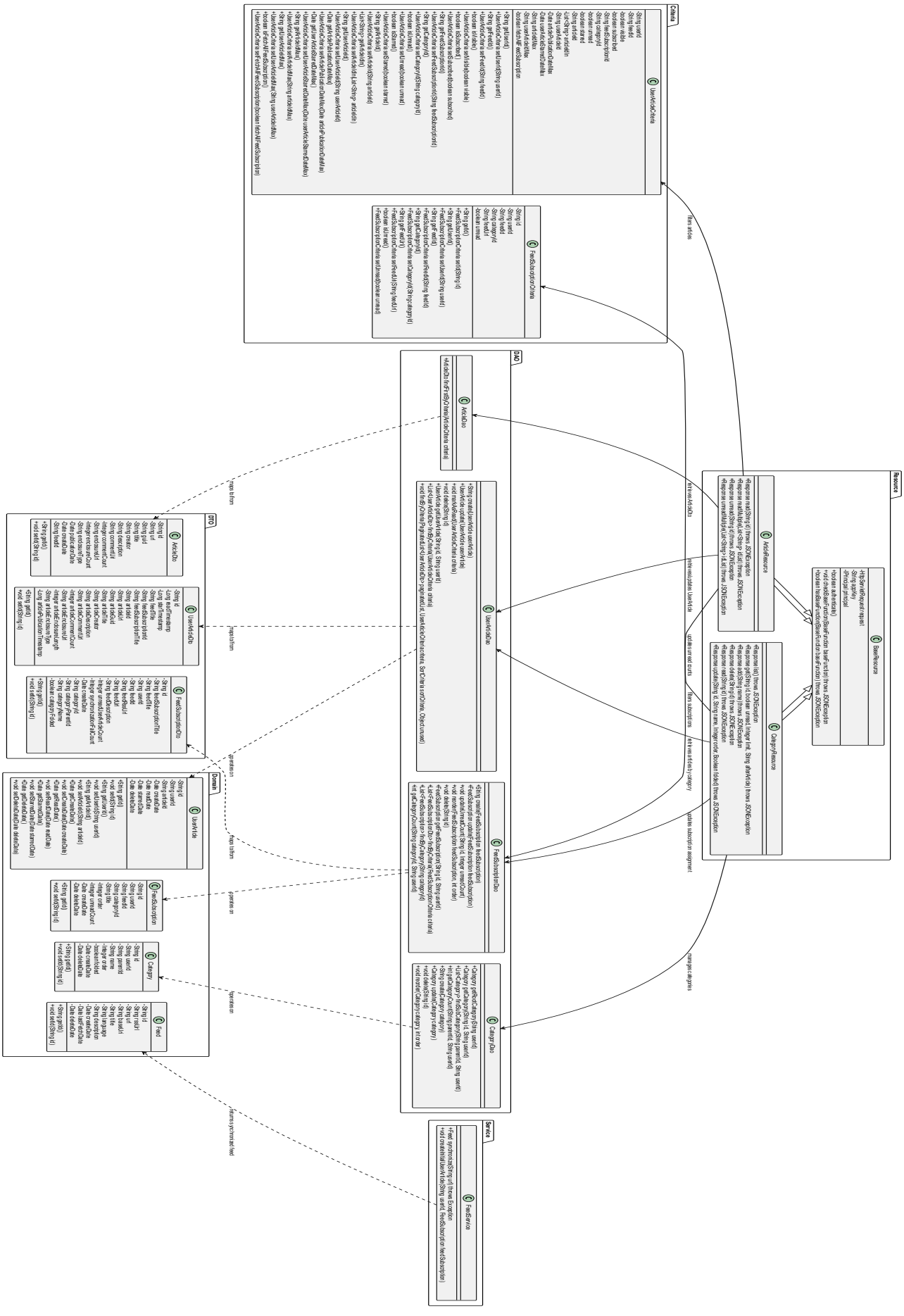
Software Engineering
Project - 1

Report
Team – 22

Team Members	
Ashish Lakhmani	2023202008
Rohit Joshi	2023202016
Rugved Thakare	2023201049
Pratik Singh	2023201082

1. UML Diagrams and Descriptions :

a. Article and Content Management System :



- ***Layered Architecture Overview :***

⇒ *Resource Layer:*
Contains the classes that handle HTTP requests and serve as entry points for API calls. These classes perform authentication, authorization, and delegate tasks to lower layers (e.g., DAO).

⇒ *Criteria Layer:*
Defines filtering and search criteria objects used to query and retrieve specific data from the persistence layer. They encapsulate parameters like user IDs, feed IDs, dates, and status flags.

⇒ *DAO - Data Access Object Layer:*
Provides methods to interact with the underlying database or persistence mechanism. Each DAO typically supports CRUD operations and maps between domain objects and DTOs.

⇒ *DTO - Data Transfer Object - Layer:*
Contains simple data carrier classes used to transfer data between processes, layers, or over the network. They usually have getters and setters for their attributes.

⇒ *Domain Layer:*
Models the core business entities (or domain objects) such as UserArticle, Category, Feed, and FeedSubscription. These classes capture the state and behavior of the primary concepts within the module.

⇒ *Service Layer:*
Contains business logic that coordinates operations between multiple DAOs or applies higher-level business rules. In this diagram, it includes operations like synchronizing a feed.

- ***Detailed Package & Class Descriptions :***

⇒ *BaseResource* **:**

*Attributes***:**

- Request : HttpServletRequest – Represents HTTP request data.
- appKey : String - Used for API authentication
- principal : IPrincipal - Holds user or system-level credentials.

Methods:

- authenticate() – Validates the request, returning a boolean.
- checkBaseFunction(BaseFunction baseFunction) – Checks if a given base function is allowed; may throw a JSONException.
- hasBaseFunction(BaseFunction baseFunction) – Returns a boolean indicating if the function is present.

⇒ *ArticleResource (inherits from BaseResource):*

Methods:

- read(String id) – Retrieves a specific article.
- readMultiple(List<String> idList) – Retrieves multiple articles at once.

- unread(String id) – Marks an article as unread.
- unreadMultiple(List<String> idList) – Marks multiple articles as unread.

Dependencies:

Uses DAOs (e.g., *UserArticleDao*, *FeedSubscriptionDao*, *ArticleDao*) and criteria (*UserArticleCriteria*) for its operations.

⇒ *CategoryResource (inherits from BaseResource):*

Methods:

- list() – Lists all categories.
- get(String id, boolean unread, Integer limit, String afterArticle) – Retrieves a specific category, with options for unread status and pagination.
- add(String name) – Adds a new category.
- delete(String id) – Deletes a category.
- read(String id) – Marks a category (or its articles) as read.
- update(String id, String name, Integer order, Boolean folded) – Updates category details.

Dependencies:

Relies on *CategoryDao*, *FeedSubscriptionDao*, *UserArticleDao*, and *FeedSubscriptionCriteria*.

⇒ *FeedSubscriptionCriteria:*

Attributes:

id, userId, feedId, categoryId, feedUrl, and a boolean unread - Used to filter and query feed subscription data based on various attributes.

Methods:

Getters and setters (with fluent interface style where setters return the criteria instance).

⇒ *UserArticleCriteria:*

Attributes:

Contains numerous fields such as userId, feedId, visible, subscribed, feedSubscriptionId, categoryId, unread, starred, articleId, a list articleIdIn, date limits (e.g., articlePublicationDateMax), and more. - Facilitates complex filtering for user articles, supporting various conditions and pagination boundaries.

Methods:

Getters and Setters to build a detailed query object.

⇒ *UserArticleDao*:

Methods:

- create(UserArticle userArticle) – Persists a new user article, returning its ID.
- update(UserArticle userArticle) – Updates an existing user article.
- markAsRead(UserArticleCriteria criteria) – Marks articles as read based on criteria.
- delete(String id) – Deletes a user article by ID.
- getUserArticle(String id, String userId) – Retrieves a specific user article.
- findByCriteria(UserArticleCriteria criteria) – Retrieves a list of *UserArticleDto* objects based on criteria.
- An overloaded findByCriteria method that supports pagination and sorting.

⇒ *CategoryDao:*

Methods:

- getRootCategory(String userId) – Retrieves the root category for a user.
- getCategory(String id, String userId) – Retrieves a specific category.
- findSubCategory(String parentId, String userId) – Finds subcategories.
- getCategoryCount(String parentId, String userId) – Counts categories under a parent.
- create(Category category) – Creates a new category.
- update(Category category) – Updates a category.
- delete(String id) – Deletes a category.
- reorder(Category category, int order) – Changes the order of a category.

⇒ *FeedSubscriptionDao:*

Methods:

- create(FeedSubscription feedSubscription) – Creates a new feed subscription.
- update(FeedSubscription feedSubscription) – Updates a subscription.
- updateUnreadCount(String id, Integer unreadCount) – Updates the unread count for a subscription.
- reorder(FeedSubscription feedSubscription, int order) – Reorders subscriptions.
- delete(String id) – Deletes a subscription.
- getFeedSubscription(String id, String userId) – Retrieves a specific subscription.
- findByCriteria(FeedSubscriptionCriteria criteria) – Retrieves subscriptions matching given criteria.
- findByCategory(String categoryId) – Finds subscriptions within a category.
- getCategoryCount(String categoryId, String userId) – Counts subscriptions for a category.

⇒ *ArticleDao:*

Method:

findFirstByCriteria(ArticleCriteria criteria) – Retrieves the first article matching the given criteria as an *ArticleDto*.

⇒ *ArticleDto:*

Attributes:

Fields such as id, url, guid, title, creator, description, commentUrl, commentCount, enclosureUrl, enclosureCount, enclosureType, publicationDate, createDate, and feedId - Used to transfer article data between layers or across the network.

⇒ *UserArticleDto:*

*Attributes***:**

Contains fields like id, readTimestamp, starTimestamp, feedTitle, feedSubscriptionId, feedSubscriptionTitle, articleId, articleUrl, articleGuid, articleTitle, articleCreator, articleDescription, articleCommentUrl, articleCommentCount, articleEnclosureUrl, articleEnclosureLength, articleEnclosureType, and articlePublicationTimestamp - Represents user specific article data for use in the presentation or service layers.

⇒ *FeedSubscriptionDto:*

Attributes:

Contains fields id, feedSubscriptionTitle, feedTitle, userId, feedId, feedRssUrl, feedUrl, feedDescription, unreadUserArticleCount, synchronizationFailCount, createDate, categoryId, categoryParentId, categoryName, and a boolean categoryFolded - Transfers feed subscription data between the persistence layer and the higher layers.

⇒ *UserArticle:*

Attributes:

id, userId, articleId, createDate, readDate, starredDate, deleteDate - Represents a user's interaction with an article in the business domain.

⇒ *Category:*

Attributes:

id, userId, parentId, name, order, folded, createDate, deleteDate - Models the organizational structure (or hierarchy) of articles/feeds into categories.

⇒ *Feed:*

Attributes:

id, rssUrl, url, baseUri, title, language, description, createDate, lastFetchDate, deleteDate - Represents the feed entity that aggregates articles, typically fetched via RSS.

⇒ *FeedSubscription:*

*Attributes***:**

id, userId, feedId, categoryId, title, order, unreadCount, createDate, deleteDate - Connects a user to a specific feed and categorizes it, managing subscription order and unread counts.

⇒ *FeedService:*

Encapsulates business logic related to feeds, such as synchronization and initial setup for user articles.

Methods:

- synchronize(String url) – Fetches and synchronizes a feed from a given URL; returns a *Feed* domain object.
- createInitialUserArticle(String userId, FeedSubscription feedSubscription) – Initializes user-specific articles when a new feed subscription is created.

- ***Relationships Between Components:***

⇒ *Resource → DAO & Criteria:*

- *ArticleResource* and *CategoryResource* make use of DAO classes (e.g., *UserArticleDao*, *CategoryDao*, *FeedSubscriptionDao*, *ArticleDao*) to perform CRUD operations and data retrieval.
- They also utilize criteria objects (*UserArticleCriteria* and *FeedSubscriptionCriteria*) to filter results when querying the database.

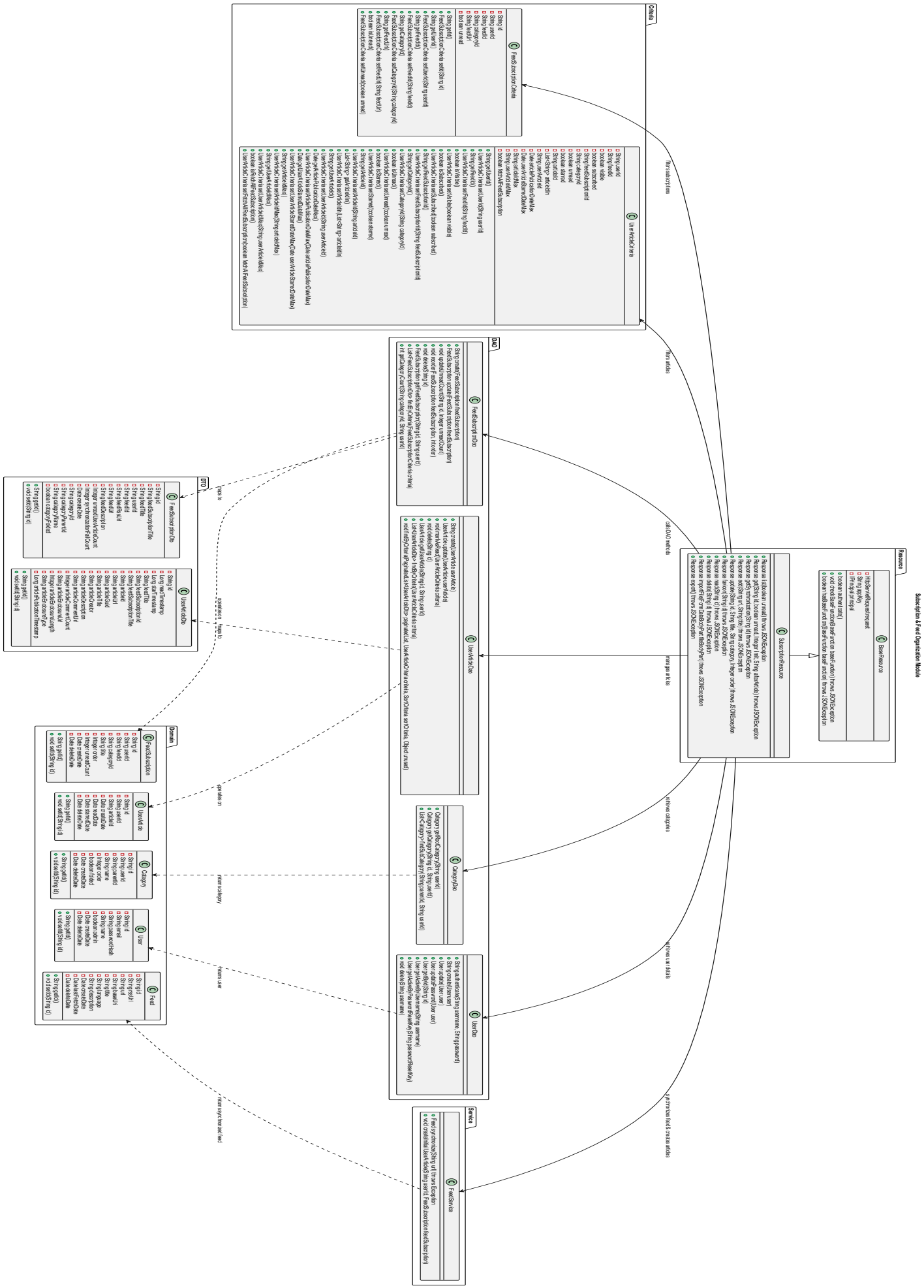
⇒ *DAO ↔ Domain & DTO:*

DAOs operate directly on **Domain** entities (e.g., *UserArticle*, *Category*, *FeedSubscription*, *Feed*) and convert or map these to and from **DTO** classes (e.g., *UserArticleDto*, *ArticleDto*, *FeedSubscriptionDto*) for data transfer purposes.

⇒ *Service → Domain:*

- The *FeedService* returns domain objects (like *Feed*) after performing operations such as synchronization.
- Inheritance in Resource Layer **:** Both *ArticleResource* and *CategoryResource* inherit common behavior and attributes from *BaseResource*, ensuring consistent authentication and request handling.
- Dependency Arrows in the Diagram:
  i. Arrows indicate that, for instance, *ArticleResource* "retrieves/updates UserArticle" through *UserArticleDao* and "updates unread counts" via *FeedSubscriptionDao*.
  ii. Similarly, *CategoryResource* depends on *CategoryDao* and *FeedSubscriptionDao* for its operations.

b. Subscription & Feed Organization Module:



- ***Layered Architecture Overview:***

⇒ *Resource Layer:*
This layer handles incoming HTTP requests. It is responsible for authentication, authorization, and routing requests to the appropriate service or data access methods. Classes in this layer represent API endpoints.

⇒ *Criteria Layer:*
These classes encapsulate the filtering and query parameters used when searching for or retrieving data. They make it easy to build complex queries by providing getters and setters for various criteria.

⇒ *DAO (Data Access Object) Layer:*
This layer interacts with the underlying data store. It provides CRUD operations and query methods to retrieve and manipulate persistent data. The DAOs also map between domain objects and data transfer objects (DTOs).

⇒ *DTO (Data Transfer Object) Layer:*
DTOs are simple objects used for transferring data between processes or layers. They often mirror the structure of domain objects but are tailored for efficient data exchange.

⇒ *Domain Layer:*
This layer models the core business entities. It includes classes for entities such as User, Feed, Category, FeedSubscription, and UserArticle. These objects hold the business state and logic.

⇒ *Service Layer:*
Contains higher-level business logic that orchestrates interactions between DAOs, domain objects, and other services. For example, synchronizing feeds and initializing user articles are tasks handled here.

- ***Detailed Package & Class Descriptions:***

⇒ *SubscriptionResource (extends BaseResource):*

Methods:

- list(boolean unread): Retrieves a list of subscriptions, optionally filtering by unread status.
- get(String id, boolean unread, Integer limit, String afterArticle): Gets details for a specific subscription with support for pagination and filtering.
- getSynchronization(String id): Retrieves synchronization information for a given subscription.
- add(String url, String title): Adds a new subscription using the provided URL and title.
- update(String id, String title, String category, Integer order): Updates subscription details like title, associated category, and order.
- favicon(String id): Retrieves the favicon for the feed associated with the subscription.
- read(String id): Marks the subscription (or its related articles) as read.
- delete(String id): Deletes a subscription.
- importFile(FormDataBodyPart fileBodyPart): Imports subscriptions from a file.
- export(): Exports subscription data.

Dependencies:

- This resource interacts with criteria classes (*FeedSubscriptionCriteria* and *UserArticleCriteria*), various DAOs (*FeedSubscriptionDao*, *UserArticleDao*, *CategoryDao*, *UserDao*), and the *FeedService* for synchronization and user article creation.

⇒ *FeedSubscriptionCriteria:*

Attributes:

- id, userId, feedId, categoryId, feedUrl - Various identifiers and URL used to filter subscriptions.
- boolean unread: Indicates whether to filter based on unread status.

Methods:

Getters and fluent setters for each attribute, allowing easy construction of query objects.

⇒ *UserArticleCriteria:*

Attributes:

- Includes parameters such as userId, feedId, visible, subscribed, feedSubscriptionId, categoryId, unread, starred, articleId, a list articleIdIn, and date boundaries like articlePublicationDateMax and userArticleStarredDateMax.
- Also contains pagination/boundary fields such as articleIdMax and userArticleIdMax, and a boolean fetchAllFeedSubscription.

Methods:

Comprehensive getters and setters to construct detailed queries for retrieving user articles.

⇒ *FeedSubscriptionDao:*

Methods:

- create(FeedSubscription feedSubscription): Persists a new feed subscription and returns its generated ID.
- update(FeedSubscription feedSubscription): Updates an existing subscription.
- updateUnreadCount(String id, Integer unreadCount): Modifies the unread count for a subscription.
- reorder(FeedSubscription feedSubscription, int order): Adjusts the ordering of subscriptions.
- delete(String id): Removes a subscription.
- getFeedSubscription(String id, String userId): Retrieves a specific subscription for a user.

- findByCriteria(FeedSubscriptionCriteria criteria): Finds subscriptions matching the specified criteria.
- getCategoryCount(String categoryId, String userId): Returns the number of subscriptions in a particular category.

⇒ *UserArticleDao:*

Methods:

- create(UserArticle userArticle): Creates a new user article record.
- update(UserArticle userArticle): Updates an existing user article.
- markAsRead(UserArticleCriteria criteria): Marks articles as read based on the given criteria.
- delete(String id): Deletes a user article.
- getUserArticle(String id, String userId): Retrieves a specific user article.
- findByCriteria(UserArticleCriteria criteria): Returns a list of *UserArticleDto* objects matching the criteria.
- An overloaded findByCriteria supports pagination and sorting.

⇒ *CategoryDao:*

Methods:

- getRootCategory(String userId): Retrieves the root category for a user.
- getCategory(String id, String userId): Gets a specific category by ID.
- findSubCategory(String parentId, String userId): Lists subcategories under a given parent.

⇒ *UserDao:*

Methods:

- authenticate(String username, String password): Authenticates a user and returns a token or identifier.
- create(User user): Creates a new user.
- update(User user): Updates user details.
- updatePassword(User user): Updates the user's password.
- getById(String id): Retrieves a user by ID.
- getActiveByUsername(String username): Retrieves an active user by username.
- getActiveByPasswordResetKey(String passwordResetKey): Finds an active user using a password reset key.
- delete(String username): Deletes a user by username.

⇒ *FeedSubscriptionDto:*

Attributes:

Contains fields such as id, feedSubscriptionTitle, feedTitle, userId, feedId, feedRssUrl, feedUrl, feedDescription, and several attributes related to unread counts, synchronization failures, and category details.

Methods:

Standard getters and setters .

⇒ *UserArticleDto:*

Attributes:

Includes id, timestamps for reading and starring (readTimestamp, starTimestamp), feed and article details (titles, IDs, URLs, descriptions, etc.), and additional metadata like comment counts.

Methods:

Standard getters and setters.

⇒ *Category:*

Attributes:

id, userId, parentId, name, order, folded, createDate, deleteDate.

*Methods***:**

Getters and setters for accessing and modifying the category's properties.

⇒ *Feed:*

Attributes:

Contains fields such as id, rssUrl, url, baseUri, title, language, description, and various dates (createDate, lastFetchDate, deleteDate).

Methods:

Standard getters and setters.

⇒ *User:*

Attributes:

Fields include id, email, passwordHash, name, an admin flag, and timestamps (createDate, deleteDate).

Methods:

Getters and setters for user properties.

⇒ *FeedSubscription:*

Attributes:

Contains id, userId, feedId, categoryId, title, order, unreadCount, createDate, and deleteDate.

Methods:

Standard getters and setters.

⇒ *UserArticle:*

Attributes:

Contains id, userId, articleId, and dates for creation, reading, starring, and deletion.

Methods:

Getters and setters for managing user article data.

⇒ *FeedService:*

Encapsulates the business logic required for synchronizing feeds and initializing related articles, thereby bridging the Resource and DAO layers.

Methods:

- synchronize(String url): Fetches and synchronizes feed data from the provided URL. This method returns a *Feed* domain object and may throw an exception if the synchronization fails.
- createInitialUserArticle(String userId, FeedSubscription feedSubscription): Creates initial user article records when a new feed subscription is added.

- ***Relationships Between Components:***

⇒ *Resource-to-Criteria & DAO Interactions:*

- SubscriptionResource utilizes the FeedSubscriptionCriteria to filter subscriptions and UserArticleCriteria to filter user articles.

- It calls methods on various DAOs:
  i. *FeedSubscriptionDao* for creating, updating, and deleting subscriptions.
  ii. *UserArticleDao* for managing articles related to subscriptions.
  iii. *CategoryDao* to retrieve category details.
  iv. *UserDao* to retrieve user details.

- It also leverages the FeedService to synchronize feeds and initialize user articles.

⇒ *DAO Mapping & Domain Association:*

FeedSubscriptionDao maps data to/from FeedSubscriptionDto and operates on the FeedSubscription domain object.

i. *UserArticleDao* similarly maps between UserArticleDto and the UserArticle domain object.
ii. *CategoryDao* returns Category domain objects.
iii. *UserDao* works with the User domain entity.

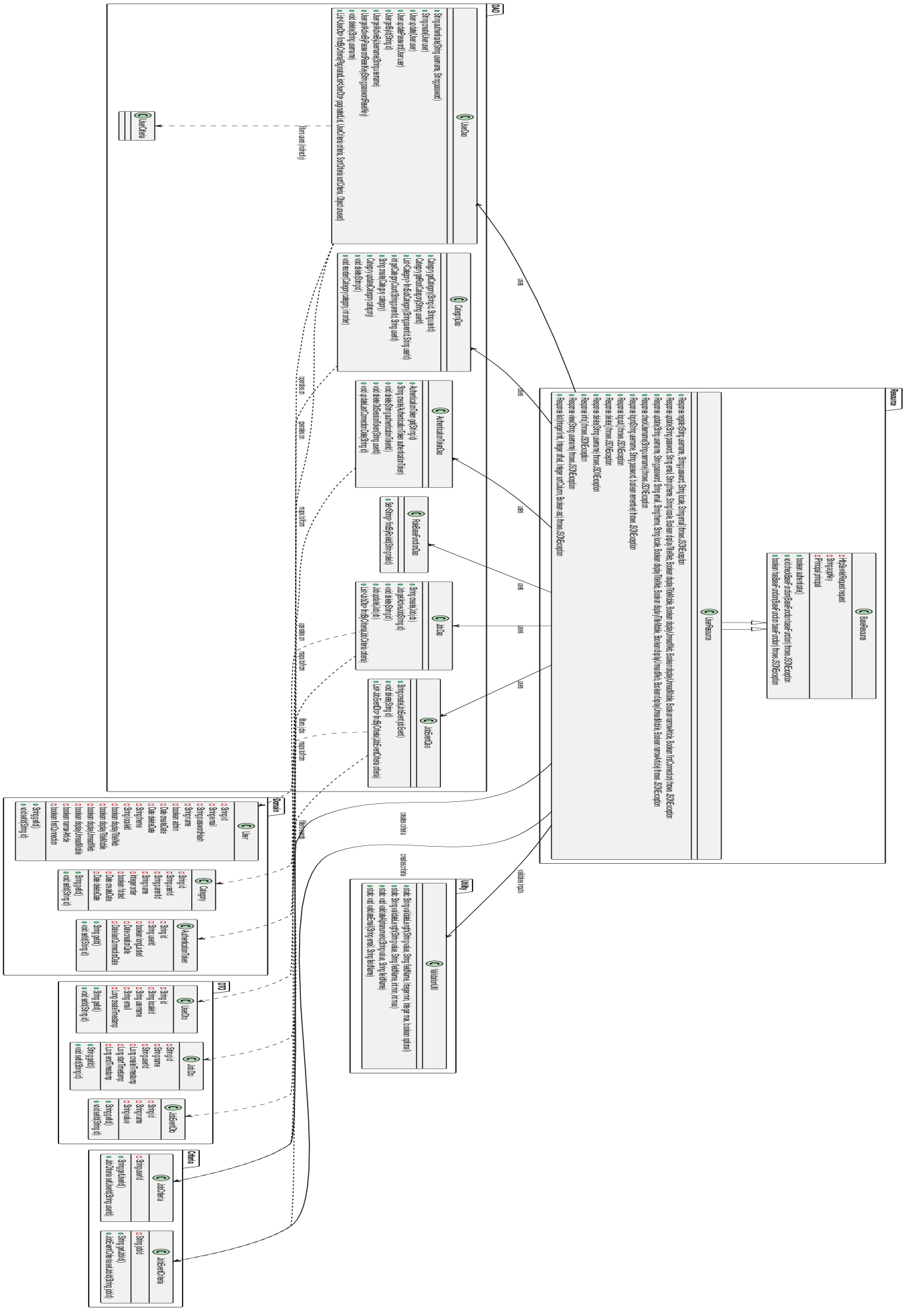
⇒ *Service Layer Dependency:*

The FeedService interacts with the Feed domain object, returning it as a result of a synchronization process. It also helps in creating initial user articles for new subscriptions.

⇒ *Inheritance in the Resource Layer:*

SubscriptionResource extends BaseResource, inheriting common request-handling, authentication, and function-checking logic.

c. ***User Account & Authentication Module:***



- ***Layered Architecture Overview:***

⇒ *Resource Layer:*
Handles HTTP requests and serves as the entry point for user management operations. It deals with input validation, authentication, and invoking the appropriate persistence or utility methods.

⇒ *DAO (Data Access Object) Layer:*
Provides methods for interacting with the database to perform CRUD operations on users, categories, authentication tokens, roles, and job-related data.

⇒ *DTO (Data Transfer Object) Layer:*
Contains simple objects that carry data between layers. DTOs help in mapping domain entities to a format suitable for communication, especially across network boundaries.

⇒ *Domain Layer:*
Models the core business entities such as User, Category, and AuthenticationToken. These classes represent the system's state and behavior.

⇒ *Criteria Layer:*
Provides criteria objects (like UserCriteria, JobCriteria, and JobEventCriteria) for filtering and querying data from the database.

⇒ *Utility Layer:*
Contains helper classes, such as ValidationUtil, which provide static methods for input validation and other common tasks.

- ***Detailed Package & Class Descriptions:***

⇒ *UserResource (Extends BaseResource):*

Methods:

- register(String username, String password, String locale, String email): Registers a new user.
- Two overloaded update(...) methods allow updating a user's password, email, theme, locale, and various display preferences.
- checkUsername(String username): Checks if a username is available.
- info(): Retrieves information about the currently authenticated user.
- view(String username): Views details for a specific user.
- login(String username, String password, boolean remember): Logs in a user and may set a persistent session if remember is true.
- logout(): Ends the user session.
- Two versions of delete() allow the deletion of the current user or a user specified by username.
- list(Integer limit, Integer offset, Integer sortColumn, Boolean asc): Lists users with pagination and sorting support.

Dependencies:

Uses several DAOs (UserDao, CategoryDao, AuthenticationTokenDao, RoleBaseFunctionDao, JobDao, JobEventDao), creates criteria objects (JobCriteria, JobEventCriteria), and utilizes the ValidationUtil for input checking.

⇒ *UserDao:*

Methods:

- authenticate(String username, String password): Validates user credentials and returns an authentication token or identifier.
- create(User user): Persists a new user and returns its ID.
- update(User user): Updates user details.
- updatePassword(User user): Specifically updates the user's password.
- getById(String id), getActiveByUsername(String username), getActiveByPasswordResetKey(String passwordResetKey): Retrieve users based on various identifiers.
- delete(String username): Deletes a user.
- findByCriteria(...): Supports querying users using a criteria object (UserCriteria) along with pagination and sorting.

⇒ *CategoryDao:*

Methods:

- getCategory(String id, String userId): Retrieves a specific category for a user.
- getRootCategory(String userId): Retrieves the root category for the user.
- findSubCategory(String parentId, String userId): Lists subcategories under a parent.
- getCategoryCount(String parentId, String userId), create(Category category), update(Category category), delete(String id), reorder(Category category, int order): Manage category lifecycle and ordering.

⇒ *AuthenticationTokenDao:*

Methods:

- get(String id): Retrieves a token by its ID.
- create(AuthenticationToken authenticationToken): Creates a new token.
- delete(String authenticationTokenId): Deletes a token.
- deleteOldSessionToken(String userId): Cleans up old tokens for a user.
- updateLastConnectionDate(String id): Updates the token with the last connection timestamp.

⇒ *RoleBaseFunctionDao:*

Method:

findByRoleId(String roleId): Returns a set of base functions available to that role.

⇒ *JobDao:*

Methods:

- create(Job job): Creates a new job record.
- getActiveJob(String id): Retrieves an active job.
- delete(String id): Deletes a job record.
- update(Job job): Updates an existing job.
- findByCriteria(JobCriteria criteria): Finds jobs based on filtering criteria.

⇒ *JobEventDao:*

Methods:

- create(JobEvent jobEvent): Creates a new job event.
- delete(String id): Deletes an event.
- findByCriteria(JobEventCriteria criteria): Retrieves job events using filtering criteria.

⇒ *UserDto:*
A data transfer object that represents a user when transferring data between layers.

Attributes:

id, localeId, username, email, createTimestamp among others : Provides a simplified view of a user's data for external communication.

⇒ *JobDto:*
Represents job data with fields like id, name, userId, and timestamps (createTimestamp, startTimestamp, endTimestamp).

⇒ *JobEventDto:*
Represents events associated with a job.

Attributes:

id, name, value.

⇒ *User:*
Models the core user entity in the system.

Attributes:

id, email, passwordHash, name, admin (flag for administrative rights), creation and deletion dates, as well as preferences like theme, localeId, and various display options (e.g., displayTitleWeb, displayUnreadMobile, etc.) - Encapsulates user-related business logic and state.

⇒ *Category:*
Represents a category that a user might organize or associate with content.

Attributes:

id, userId, parentId, name, order, folded, and timestamps - Helps in organizing content or user data into hierarchies.

⇒ *AuthenticationToken:*
Models an authentication token used for maintaining a user session.

Attributes:

id, userId, longLasted (to indicate whether the session should persist), creationDate, and lastConnectionDate - Supports session management and security by tracking token validity.

⇒ *UserCriteria:*
Provides filtering capabilities for querying users. Attributes (not fully detailed in the diagram) allow tailoring queries based on specific user properties.

⇒ *JobCriteria:*
Contains filtering attributes for job queries, such as userId.

Methods:

Getters and setters (fluent interface style) to build criteria for filtering job records.

⇒ *JobEventCriteria:*
Used for filtering job events based on a jobId.

Methods:

Provides getters and setters for the jobId attribute.

⇒ *ValidationUtil:*
Helps maintain data integrity and ensures that user inputs meet required constraints

Methods:

- validateLength(...): Validates the length of a string, with overloaded versions to support optional values.
- validateAlphanumeric(...): Ensures a value is alphanumeric.
- validateEmail(...): Validates the format of an email address.

- ***Relationships Between Components:***

⇒ *Resource Inheritance & Usage:*

- UserResource extends BaseResource, inheriting its authentication and request-handling capabilities.
- UserResource depends on multiple DAO classes (UserDao, CategoryDao, AuthenticationTokenDao, RoleBaseFunctionDao, JobDao, JobEventDao) to perform persistence operations.
- It also creates instances of JobCriteria and JobEventCriteria to filter data during job-related operations and uses ValidationUtil to validate inputs.

⇒ *DAO and Domain/DTO Mapping:*

- UserDao operates on User domain objects and maps data to/from UserDto.
- CategoryDao works with Category domain objects.
- AuthenticationTokenDao works with AuthenticationToken domain objects.
- JobDao and JobEventDao map between their respective domain entities and JobDto/JobEventDto.

⇒ *Criteria Usage:*
- UserCriteria is used indirectly by UserDao to filter queries.
- JobCriteria and JobEventCriteria are directly used by JobDao and JobEventDao to filter job-related queries.

⇒ *Utility Integration:*

ValidationUtil is used by UserResource to ensure that input parameters (like username, password, and email) conform to expected formats and length restrictions.

2. Design Smells:

a. Smell Detection and Correction(3a):

- ***UserResource.java - Missing Abstraction (Abstraction Smell) :***

⇒ *Description:*

"Missing Abstraction" arises when a piece of functionality (or concept) is duplicated across multiple parts of the codebase instead of being encapsulated in a single, coherent abstraction (class, method, or module). In this scenario, the two update methods in UserResource both contain overlapping logic that should be extracted into a shared abstraction.

⇒ *Symptoms:*

- Duplicate Code: Repeated logic for validation, data transformation, or other update steps.
- Maintenance Overhead: Any change to the update process must be made in multiple places.
- Inconsistent Behavior: If one duplicated method is updated but the other is not, the system can behave inconsistently.

⇒ *Implications:*

- Increased Maintenance Cost: More effort is required to keep the code in sync when updates are needed.
- Higher Risk of Bugs: Duplicated logic can lead to discrepancies if not all instances are updated uniformly.
- Code Bloat: Multiple methods with similar code add unnecessary bulk to the codebase, making it harder to read and navigate.

⇒ *Refactoring:*

The common user-property update logic is extracted into a private helper method. This method is then reused by both update endpoints (one for the current user and one for an admin updating another user). Comments within the code explain the changes.

- ***ForbiddenClientException.java, ClientException.java, ServerException.java, GenericExceptionMapper.java - Missing Abstraction (Abstraction Smell) :***

⇒ *Description:*

The "Missing Abstraction" smell occurs when a common behavior—in this case, exception handling—is not abstracted into a unified framework or strategy. Instead, individual modules define and throw their own exceptions (e.g., ForbiddenClientException, ClientException, ServerException), even when these exceptions overlap in responsibility. This leads to inconsistent handling of similar error cases.

⇒ *Symptoms:*

- Diverse Exception Types: Multiple exception classes are used to represent similar error conditions.
- Inconsistent Error Handling: Different modules handle errors in their own way, leading to varied responses to similar problems.
- Overlapping Concerns: Exceptions with similar roles exist under different names, causing confusion about which to use.
- Scattered Exception Logic: Error handling logic is spread out rather than centralized, making it harder to manage and update.

⇒ *Implications:*

- Maintenance Overhead: Changes to error handling need to be replicated across multiple modules, increasing the risk of discrepancies.
- Error-Prone Behavior: Inconsistent exception types can lead to improper handling or unintentional propagation of errors.
- Reduced Clarity: Developers may struggle to understand the intended error-handling strategy, leading to misuse or misinterpretation of exceptions.

- Integration Issues: A lack of standardized exception handling can complicate integration with other systems or frameworks that expect a uniform approach.

⇒ *Refactoring:*

The idea is to create a common “ErrorResponse” class (a simple POJO) that encapsulates the error “type” and “message” and then refactor your custom exceptions to use it instead of manually constructing JSON objects (and throwing checked JSONException). This approach eliminates the need for checked JSONException in exception constructors, provides a consistent error format across your application, decouples error representation from a specific JSON library.

- ***UserResource.java - Modularization Smell (Insufficient Modularization) :***

⇒ *Description:*

The "Long Methods" smell is a classic example of insufficient modularization. It occurs when a single method takes on multiple responsibilities—such as assembling user information and aggregating job statistics—resulting in a lengthy, monolithic block of code. This makes it hard to understand, maintain, and test, as the method doesn't clearly separate distinct concerns.

⇒ *Symptoms:*

- Excessive Length: The method spans many lines of code, often hundreds, making it difficult to follow.
- Multiple Responsibilities: The method performs several distinct tasks, each of which could logically be handled by separate methods.
- Complex Control Flow: The method contains nested conditionals, loops, or other complex logic that further complicates readability.
- Difficult Debugging: Pinpointing bugs is challenging because an error in one part of the method might affect seemingly unrelated functionalities.

⇒ *Implications:*

- Reduced Readability: The longer and more complex the method, the harder it is for developers to quickly understand its purpose and behavior.
- Higher Risk of Bugs: Intermingling different responsibilities increases the chance that a change intended for one part of the method might inadvertently impact another.
- Poor Maintainability: Maintaining and updating the method becomes laborious, and the risk of introducing regression errors is heightened.
- Hindered Testing: Unit testing becomes less effective because it's difficult to isolate and test individual pieces of functionality within the monolithic method.

⇒ *Refactoring:*

Below is one refactored version of your info() method in UserResource.java. In this version, we've extracted the long, complex logic into three helper methods:

- assembleAnonymousResponse() – builds the JSON response for an anonymous user.
- assembleUserInfo(User user) – builds the JSON representation for the authenticated user's basic details.
- aggregateJobStatistics(User user) – aggregates job statistics for the user.

- ***Constants.java, messages.properties - Missing Abstraction (Abstraction Smell) :***

⇒ *Description:*
The "Missing Abstraction" smell in this context arises when hard-coded values and magic strings are embedded directly within the code rather than being abstracted into a centralized configuration or constants resource. This practice bypasses the benefits of abstraction by scattering important values throughout the codebase, making the system less flexible and more error-prone.

⇒ *Symptoms:*
  - Scattered Hard-Coded Values: Default error messages, passwords, or identifiers appear directly in multiple classes.
  - Inconsistent Updates: Changing a hard-coded value requires modifying it in several places, increasing the risk of oversights.
  - Reduced Readability: Magic strings make it harder to understand the purpose or context of the values.
  - Hidden Business Logic: Important configuration details are embedded in the code, making them less visible and manageable.

⇒ *Implications:*
  - Maintenance Overhead: Multiple code modifications are necessary for what should be a simple configuration change.
  - Increased Risk of Bugs: Inconsistencies may arise if not all instances of the hard-coded value are updated correctly.
  - Limited Flexibility: The system becomes rigid since changing these values requires code changes and redeployment.
  - Poor Localization and Internationalization: Hard-coded messages make it difficult to support multiple languages or regions.

⇒ *Refactoring:*
We externalize error messages and default values into configuration files using ResouceBundle and centralize constants in a dedicated class. This way, if any of these values need to change, you only update one place rather than modifying multiple files like update Exception Classes to use Constants.

- ***FeedService.java - Multifaceted abstraction (Abstraction Smells):***

⇒ *Description:*
The Multifaceted Abstraction smell arises when a single class is burdened with multiple, unrelated responsibilities. In this scenario, the class is responsible for tasks such as feed synchronization, article management, RSS feed and page parsing, favicon management, and event handling. This violates the Single Responsibility Principle (SRP) and creates a class that is overly complex and hard to understand.

⇒ *Symptoms:*
  - Overloaded Class: The class contains methods that serve distinct purposes, often unrelated to one another.
  - Inconsistent Naming: Method and variable names within the class might not reflect a single, unified concept.
  - Complex Dependencies: The class likely depends on various other components or libraries to support its diverse roles.

- Difficulty in Testing: Unit testing becomes challenging because isolating the different functionalities is hard.

⇒ *Implications:*

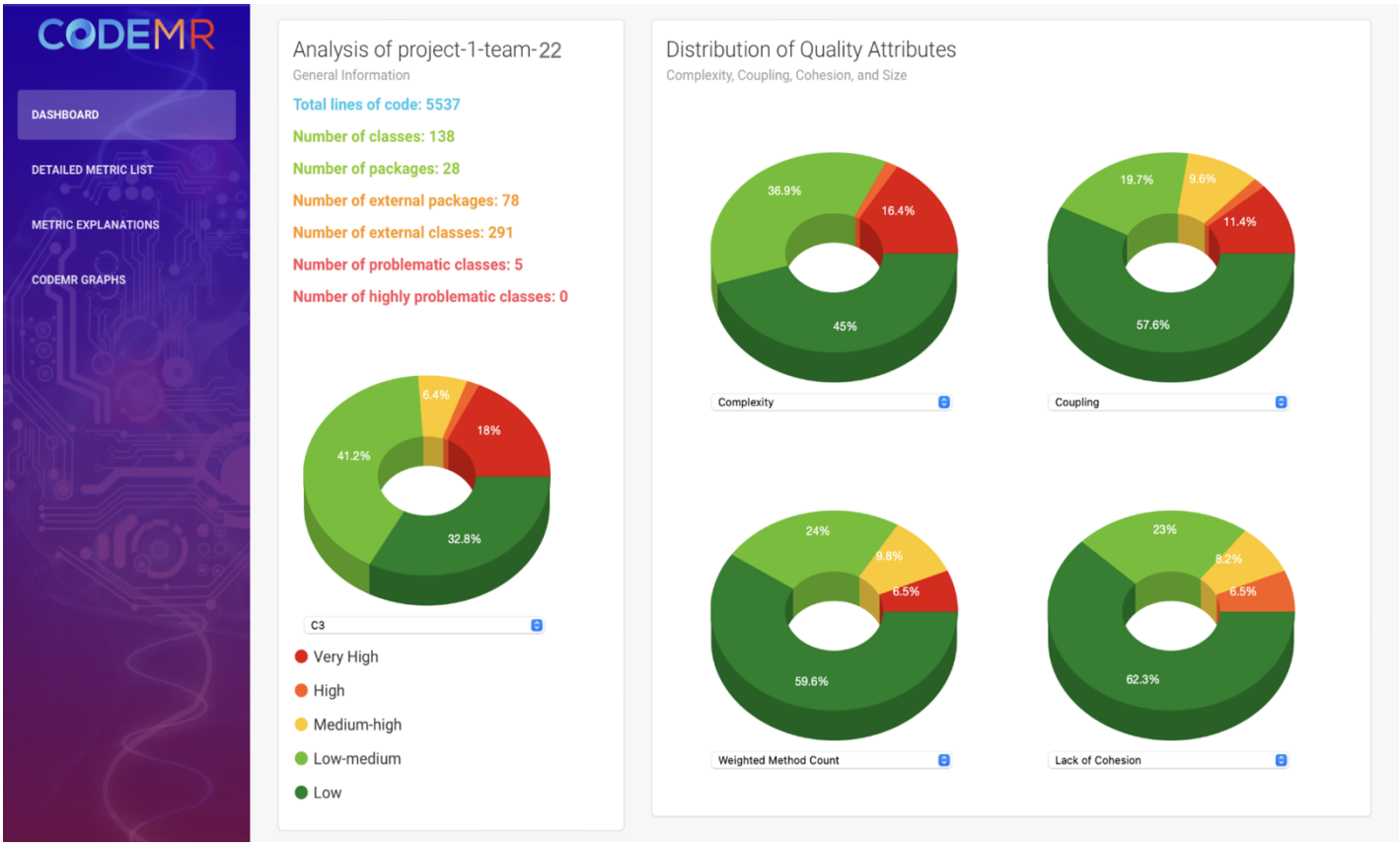
- Maintenance Challenges: Any change in one responsibility could inadvertently impact other functionalities within the class.
- Increased Bug Risk: Mixing multiple concerns increases the likelihood of introducing bugs when modifications are made.
- Poor Readability: Developers may struggle to quickly grasp the class's purpose, making onboarding and debugging more difficult.
- Limited Reusability: A class with mixed responsibilities is less adaptable for reuse in other contexts where only one of its functions might be needed.

⇒ *Refactoring:*

Split the multifaceted class into several smaller, cohesive classes like :

- ArticleService.java : Handle all article management operations.
- FeedParser.java : Focus solely on parsing RSS feeds and pages.
- FaviconUtil.java : Manage favicon-related tasks.

b. Code Metrics Analysis:



- ***CodeMR Report Analysis:***

Overview of Key Metrics:

- Total Lines of Code: 5537
- Classes: 138
- Packages: 28
- External Packages: 78

- External Classes: 291
- Problematic Classes: 5
- Highly Problematic Classes: 0

- ***Strengths:***

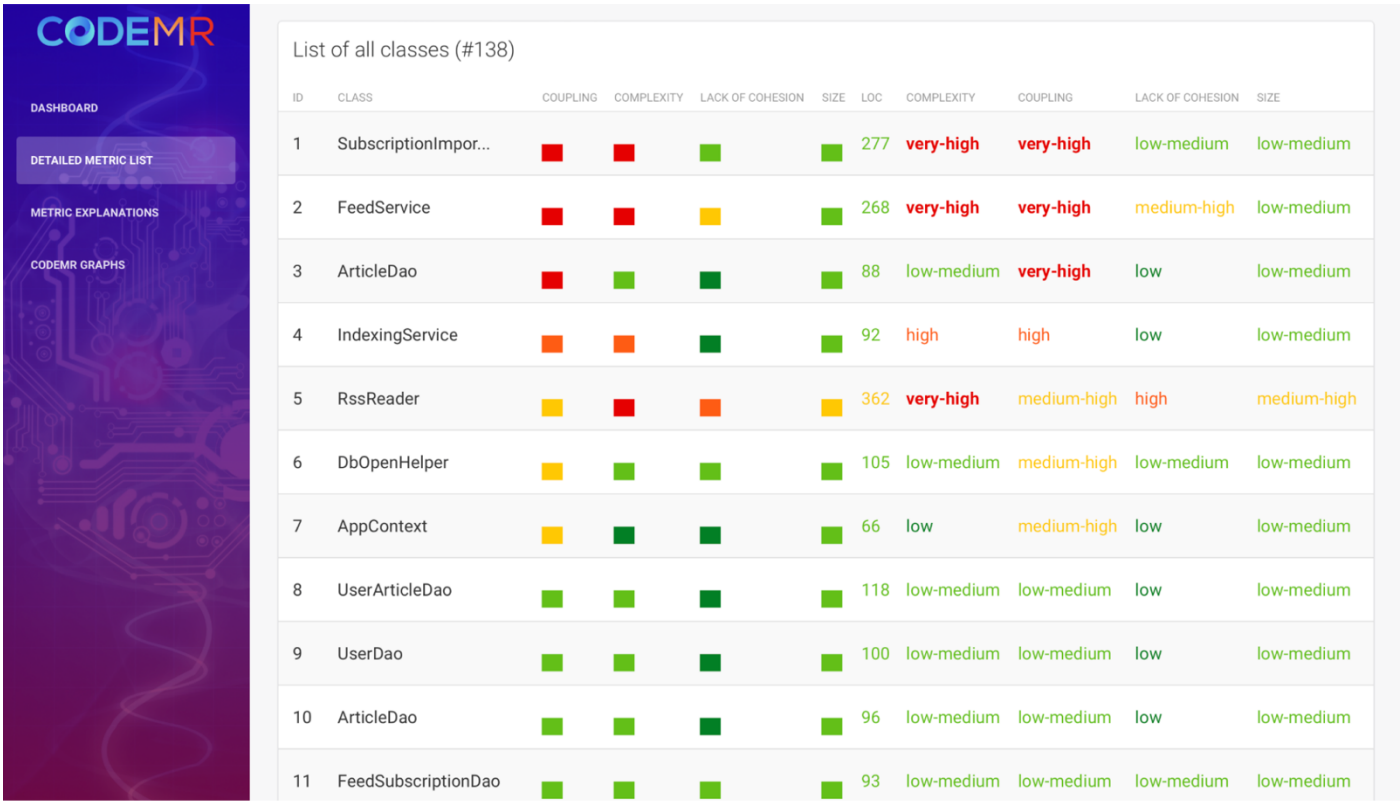
  - No Highly Problematic Classes: The codebase shows no signs of extremely flawed modules, indicating a stable overall design.
  - Balanced Complexity and Coupling: Most classes are categorized under low or low-medium complexity and coupling, which simplifies maintenance and reduces the risk of intricate interdependencies.
  - Manageable Complexity: With 45% of classes at low complexity and 41.2% at low-medium complexity, the logic across the classes remains straightforward and accessible.
  - Low Coupling Prevalence: About 57.6% of classes exhibit low coupling, fostering modular design and easing future modifications.
  - Efficient Method Usage: Approximately 59.6% of methods have a low weighted method count (WMC), suggesting minimal code bloat.
  - Strong Cohesion in Majority: 62.3% of classes demonstrate high cohesion, meaning their methods work well together toward common goals, which enhances readability and maintainability.

- ***Weaknesses:***

  - **Problematic Classes:** Although there are only 5 problematic classes, they warrant careful attention and potential refactoring to avoid future complications.
  - **High Complexity Concerns:** Nearly half of the classes (18% high and 32.8% medium-high) possess a complexity level that might lead to maintenance challenges, increased testing needs, and bugs.
  - **Tight Coupling:** Some classes exhibit higher degrees of coupling (11.4% high and 19.7% medium), which can complicate changes and updates.
  - **Cohesion Issues:** While a majority show strong cohesion, 23% of classes have low cohesion, indicating unclear responsibilities and the potential for duplicated code.

- ***Recommendations:***

  - **Refactor Problematic Classes:** Investigate the 5 problematic classes for issues related to complexity, coupling, or cohesion, and refactor them to enhance overall maintainability.
  - **Simplify Complex Classes:** For classes with medium-high or high complexity, consider breaking down large methods into smaller ones or redistributing responsibilities across multiple classes.
  - **Reduce Coupling:** Examine the classes with high coupling and explore design patterns like Dependency Injection or Facade to decrease interdependencies.
  - **Improve Cohesion:** For classes with low cohesion, reassess their responsibilities and possibly divide them into more focused, task-specific classes.
  - **Monitor Method Weight:** Keep an ongoing review of the weighted method count to ensure that methods remain efficient and maintainable over time.



ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	SubscriptionImpor...					277	very-high	very-high	low-medium	low-medium
2	FeedService					268	very-high	very-high	medium-high	low-medium
3	ArticleDao					88	low-medium	very-high	low	low-medium
4	IndexingService					92	high	high	low	low-medium
5	RssReader					362	very-high	medium-high	high	medium-high
6	DbOpenHelper					105	low-medium	medium-high	low-medium	low-medium
7	AppContext					66	low	medium-high	low	low-medium
8	UserArticleDao					118	low-medium	low-medium	low	low-medium
9	UserDao					100	low-medium	low-medium	low	low-medium
10	ArticleDao					96	low-medium	low-medium	low	low-medium
11	FeedSubscriptionDao					93	low-medium	low-medium	low-medium	low-medium

- ***Overview of Class Quality Analysis:***
  This document reviews 138 classes by evaluating key metrics such as coupling, complexity, cohesion, lines of code (LOC), and size. The goal is to identify classes that fall into high-risk, medium-risk, and low-risk categories, and to offer targeted recommendations for enhancement.

  ⇒ *High-Risk Classes:*
  These classes are complex and tightly interconnected, indicating that they might be shouldering multiple responsibilities or have dependencies that span several modules. If not refactored, they could lead to bugs, hinder maintainability, and complicate future updates. Some classes are flagged as high-risk due to their extremely high complexity and coupling. Notable examples include:

  - *SubscriptionImporter:*
    Complexity: Very high
    Coupling: Very high
    Cohesion: Low-medium

  - *FeedService:*
    Complexity: Very high
    Coupling: Very high
    Cohesion: Medium-high

  - *RssReader:*
    Complexity: Very high
    Coupling: Very high
    Cohesion: High

  ⇒ *Low-Risk and Well-Designed Classes:*

Several classes demonstrate low complexity, low coupling, and high cohesion, which indicates a robust and maintainable design. These classes are streamlined, cohesive, and have minimal dependencies, making them easy to maintain and extend. They serve as prime examples of clean and effective design.

⇒ *Medium-Risk Classes:*

Some classes are categorized as medium-risk, falling into the medium-high range for complexity or coupling. These classes may be handling more responsibilities or dependencies than necessary, which could evolve into maintainability challenges over time.

- ***Tools Used:***

  CodeMR :

  CodeMR is a static code analysis tool that assists developers in visualizing and understanding the complexities within their software architecture. It offers metrics on coupling, cohesion, and various quality attributes. Two key features of CodeMR are:

  - It provides a detailed visual map of the codebase, making it easier to spot and resolve architectural or design issues.
  - The metrics it delivers serve as early indicators of potential maintainability challenges, allowing teams to prioritize necessary refactoring efforts.

- ***Implications Discussion:***
  - Complexity:
    High complexity may indicate that a class or method is undertaking too many tasks, making it difficult to understand, test, and maintain. Simplifying such code often requires refactoring into smaller, more focused methods or classes.
  - Coupling:
    High coupling suggests that classes are overly interdependent, meaning changes in one class could have ripple effects throughout the system. Lower coupling is preferred for enhanced maintainability and flexibility.
  - Cohesion:
    Low cohesion might imply that a class is handling multiple, unrelated responsibilities, which can necessitate breaking it into more targeted, cohesive components.
  - Size (LOC - Lines of Code):
    Classes or methods with a large number of lines can be hard to maintain and may benefit from being divided into smaller, more manageable sections.
  - Number of Problematic Classes:
    This metric highlights classes that require immediate attention due to poor coding practices, potentially impacting overall maintainability and performance.
  - Weighted Method Count (WMC):
    WMC, defined as the sum of the complexities of all methods in a class, indicates the level of effort required to maintain and develop that class.

- ***Comprehensive Analysis for Task 2B:***
  Based on the attached images and CodeMR metrics, the following analysis details several key metrics:

⇒ *Lines of Code (LOC):*

- Definition: LOC measures the total number of lines in a class, method, or project.
- Observed Range: Values range from 66 lines (e.g., AppContext) to 594 lines (e.g., RssReader).
- Implications:
  - Classes like RssReader (594 LOC) and SubscriptionImporter (510 LOC) are quite large, suggesting higher complexity.
  - Increased LOC can lead to a greater risk of bugs, reduced readability, and more challenging maintenance, whereas smaller classes are generally easier to manage.

⇒ *Cyclomatic Complexity:*

- Definition: This metric counts the number of independent execution paths through a program, reflecting the number of decision points.
- Observed Issues: CodeMR indicates that classes such as FeedService and SubscriptionImporter exhibit very high complexity.
- Implications:
  - High cyclomatic complexity means numerous decision points (e.g., if statements, switch cases) which can make testing more difficult and increase the likelihood of defects.
  - Debugging and maintaining such classes may require significant time and effort.

⇒ *Coupling Between Objects (CBO):*

- Definition: CBO quantifies the number of other classes with which a given class interacts.
- Observed Issues: Classes like FeedService, SubscriptionImporter, and RssReader show very high coupling, indicating they depend on many other classes.
- Implications: High coupling means that a change in one class can trigger widespread effects, reducing reusability and complicating modifications and testing.

⇒ *Lack of Cohesion of Methods (LCOM):*

- Definition: LCOM measures how closely related the methods within a class are.
- Observed Issues: Classes such as DbOpenHelper and RssReader demonstrate low cohesion, meaning their methods do not share a common purpose.
- Implications: Low cohesion may suggest that a class is burdened with multiple, unrelated responsibilities, which violates the Single Responsibility Principle (SRP) and can lead to code duplication and diminished maintainability.

⇒ *Weighted Methods per Class (WMC):*

- Definition: WMC is the total sum of the complexities of all methods in a class.
- Observed Issues: High WMC values are observed in classes like SubscriptionImporter and FeedService, indicating an overload of complex methods.
- Implications:
  - A high WMC makes it challenging to understand a class's overall behavior due to its many responsibilities.
  - These classes are generally more prone to defects and are harder to debug and extend.

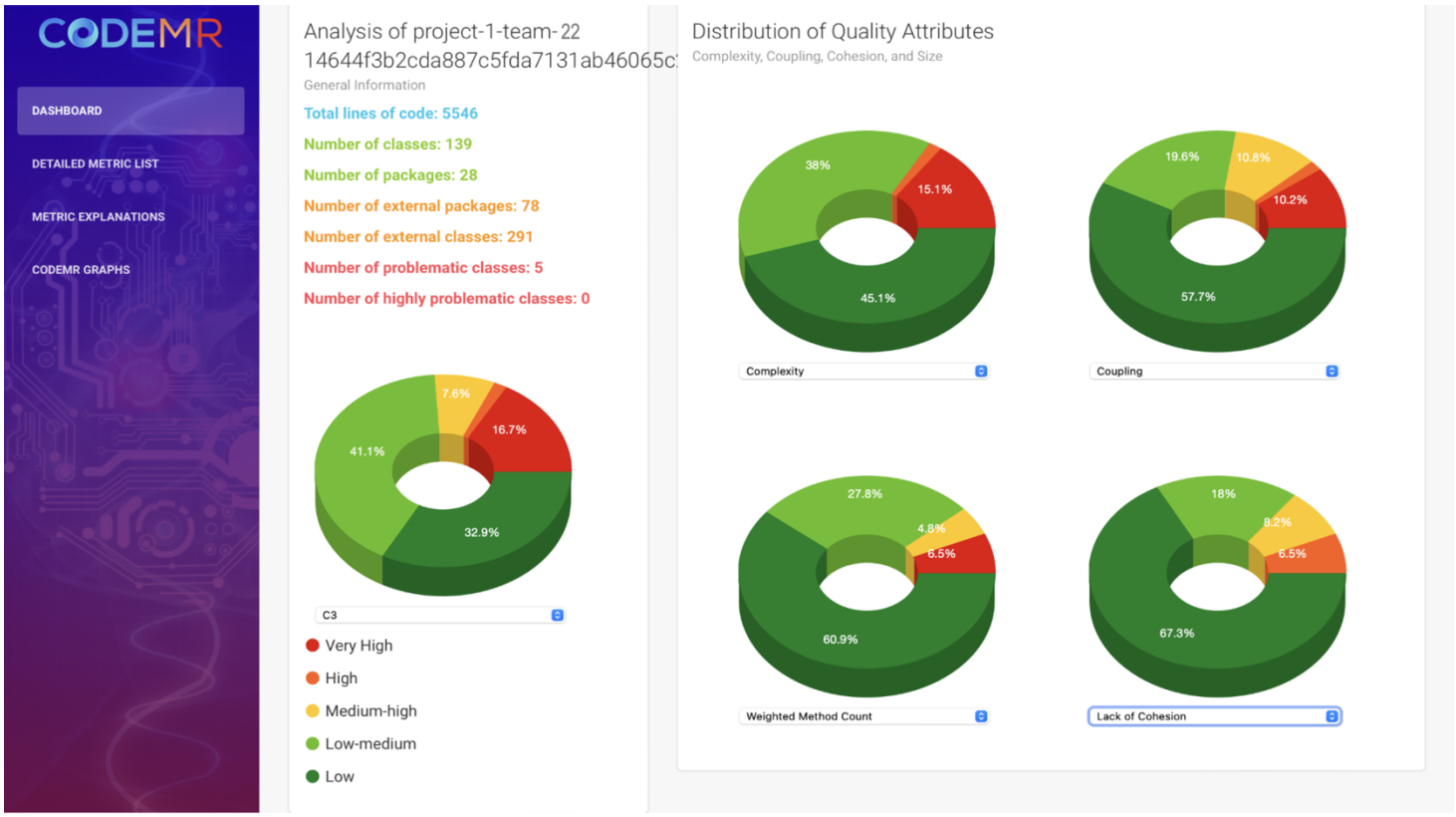
⇒ *Depth of Inheritance Tree (DIT):*

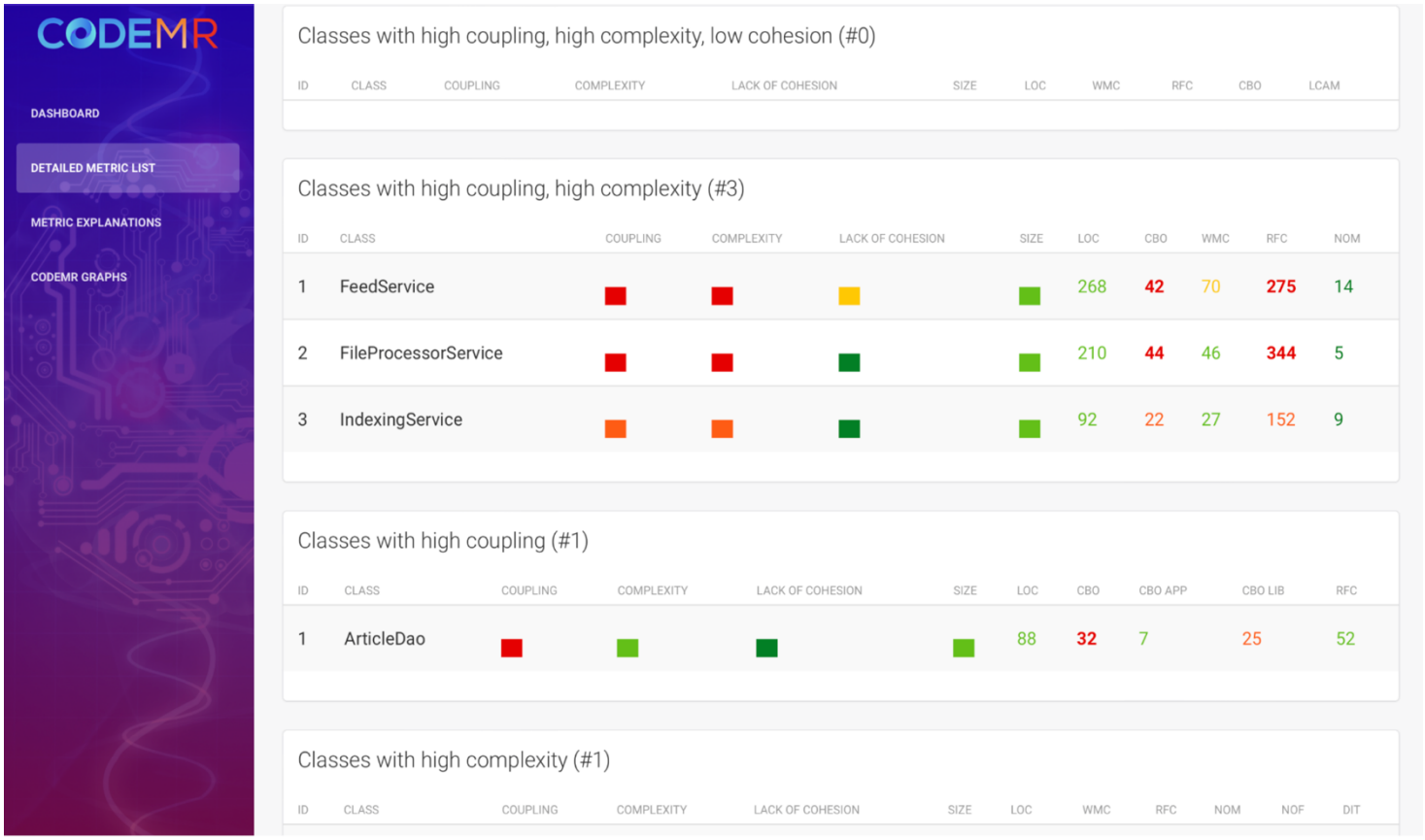
- Definition: DIT measures the number of inheritance levels from the base class to a particular class.
- Implications:
  - A high DIT can increase complexity due to more method overrides and dependencies.
  - Keeping the DIT low helps simplify the inheritance hierarchy.
  - Observed Issue: Although the current DIT levels are not extreme, they should be monitored—especially for classes within deep inheritance hierarchies.

3. Refactoring:

a. Smell Correction (Already done in 2a)

b. Code Metrics:





Quality Attribute Distribution:

- Complexity:
  Approximately 45.1% of the methods exhibit low complexity, signaling simpler, more maintainable code.

- Coupling:
  Most classes (57.7%) have low coupling, which enhances modularity and facilitates easier modifications and maintenance.

- Cohesion:
  A majority of methods (67.3%) demonstrate high cohesion, ensuring each component focuses on a single task. This improves understandability and reduces error rates.

- Problematic Classes:
  Detailed analysis identifies specific classes, such as FeedService, FileProcessorService, and IndexingService, that suffer from high coupling and complexity. Addressing these issues could enhance performance and scalability.

- Refactoring Impact:
  The distribution charts reveal that refactoring has successfully reduced the incidence of high complexity and coupling in several classes.

- Metric Explanation:
  Metrics like Lines of Code (LOC), Weighted Method Count (WMC), and Response for Class (RFC) measure software size and interactions. Lower values in these metrics after refactoring indicate improved software design.

- The refactoring efforts have positively influenced the software quality by reducing complexity and coupling while boosting cohesion. This suggests that the codebase is becoming increasingly maintainable and easier to extend.

c. LLM Refactor Code:

- ***ArticleResource.java - Tight Coupling:***
  We segregate the ArticleResource.java code into 3 sections for better separation of concerns, and makes the code more modular and testable i.e. Resource layer, Service Layer and DAO Layer

⇒ DAO Layer:

- We defined three DAO interfaces: IUserArticleDao, IArticleDao, and IFeedSubscriptionDao and provided sample implementations (e.g., UserArticleDaoImpl, ArticleDaoImpl, and FeedSubscriptionDaoImpl).
- These DAO classes encapsulate all data access operations.

⇒ Service Layer:

- The ArticleService interface defines the business operations for marking articles as read/unread (both individually and in bulk).
- The ArticleServiceImpl implements these operations by delegating to the DAO interfaces.
- Common functionality (like updating subscription unread counts) is encapsulated in helper methods within the service.

⇒ Resource Layer:

The ArticleResource now only handles authentication and HTTP concerns. It delegates article-related operations to the ArticleService.

⇒ ***DAO Interfaces and Implementations:***

- IUserArticleDao.java:

```java
package com.sismics.reader.core.dao;

import com.sismics.reader.core.dao.jpa.dto.UserArticleDto;
import com.sismics.reader.core.model.jpa.UserArticle;
import java.util.List;

public interface IUserArticleDao {
    UserArticle getUserArticle(String id, String userId);
    void update(UserArticle userArticle);
    String create(UserArticle userArticle);
    List<UserArticleDto> findByCriteria(Object criteria); // Simplified signature
}
```

- UserArticleDaoImpl.java:

```java
package com.sismics.reader.core.dao.impl;

import com.sismics.reader.core.dao.IUserArticleDao;
import com.sismics.reader.core.dao.jpa.dto.UserArticleDto;
import com.sismics.reader.core.model.jpa.UserArticle;
import java.util.List;

public class UserArticleDaoImpl implements IUserArticleDao {
    @Override
    public UserArticle getUserArticle(String id, String userId) {
        // TODO: Add persistence logic
        return null;
    }
    @Override
    public void update(UserArticle userArticle) {
        // TODO: Add update logic
    }
    @Override
    public String create(UserArticle userArticle) {
        // TODO: Add create logic
        return null;
    }
    @Override
    public List<UserArticleDto> findByCriteria(Object criteria) {
        // TODO: Add query logic
        return null;
    }
}
```

- IArticleDao.java:

```java
package com.sismics.reader.core.dao;

import com.sismics.reader.core.dao.jpa.dto.ArticleDto;
import com.sismics.reader.core.dao.jpa.criteria.ArticleCriteria;
import com.sismics.reader.core.model.jpa.Article;

public interface IArticleDao {
    ArticleDto findFirstByCriteria(ArticleCriteria criteria);
    void update(Article article);
    String create(Article article);
    void delete(String articleId);
}

```

- ArticleDaoImpl.java

```java
package com.sismics.reader.core.dao.impl;

import com.sismics.reader.core.dao.IArticleDao;
import com.sismics.reader.core.dao.jpa.criteria.ArticleCriteria;
import com.sismics.reader.core.dao.jpa.dto.ArticleDto;
import com.sismics.reader.core.model.jpa.Article;

public class ArticleDaoImpl implements IArticleDao {
    @Override
    public ArticleDto findFirstByCriteria(ArticleCriteria criteria) {
        // TODO: Implement query logic
        return null;
    }
    @Override
    public void update(Article article) {
        // TODO: Implement update logic
    }
    @Override
    public String create(Article article) {
        // TODO: Implement create logic
        return null;
    }
    @Override
    public void delete(String articleId) {
        // TODO: Implement delete logic
    }
}
```


```java
    @Override
    public void update(Article article) {
        // TODO: Implement update logic
    }
    @Override
    public String create(Article article) {
        // TODO: Implement create logic
        return null;
    }
    @Override
    public void delete(String articleId) {
        // TODO: Implement delete logic
    }
}
```

- IFeedSubscriptionDao.java

```java
package com.sismics.reader.core.dao;

import com.sismics.reader.core.dao.jpa.dto.FeedSubscriptionDto;
import com.sismics.reader.core.dao.jpa.criteria.FeedSubscriptionCriteria;
import java.util.List;

public interface IFeedSubscriptionDao {
    List<FeedSubscriptionDto> findByCriteria(FeedSubscriptionCriteria criteria);
    void updateUnreadCount(String id, int newCount);
}

```

- FeedSubscriptionDaoImpl.java

```java
package com.sismics.reader.core.dao.impl;

import com.sismics.reader.core.dao.IFeedSubscriptionDao;
import com.sismics.reader.core.dao.jpa.criteria.FeedSubscriptionCriteria;
import com.sismics.reader.core.dao.jpa.dto.FeedSubscriptionDto;
import java.util.List;

public class FeedSubscriptionDaoImpl implements IFeedSubscriptionDao {
    @Override
    public List<FeedSubscriptionDto> findByCriteria(FeedSubscriptionCriteria criteria) {
        // TODO: Implement query logic
        return null;
    }
    @Override
    public void updateUnreadCount(String id, int newCount) {
        // TODO: Implement update logic
    }
}

```

⇒ ***Service Interfaces and Implementations:***

- ArticleService.java

```java
package com.sismics.reader.core.service;

import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;
import java.util.List;

public interface ArticleService {
    JSONObject markArticleRead(String articleId, String userId) throws JSONException;
    JSONObject markArticlesRead(List<String> articleIds, String userId) throws JSONException;
    JSONObject markArticleUnread(String articleId, String userId) throws JSONException;
    JSONObject markArticlesUnread(List<String> articleIds, String userId) throws JSONException;
}
```

- ArticleServiceImpl.java

```java
package com.sismics.reader.core.service;

public class ArticleServiceImpl implements ArticleService {

    private IUserArticleDao userArticleDao;
    private IArticleDao articleDao;
    private IFeedSubscriptionDao feedSubscriptionDao;

    public ArticleServiceImpl(IUserArticleDao userArticleDao, IArticleDao articleDao, IFeedSubscriptionDao feedSubscriptionDao) {
        this.userArticleDao = userArticleDao;
        this.articleDao = articleDao;
        this.feedSubscriptionDao = feedSubscriptionDao;
    }

    @Override
    public JSONObject markArticleRead(String articleId, String userId) throws JSONException {
        UserArticle userArticle = userArticleDao.getUserArticle(articleId, userId);
        if(userArticle == null) {
            throw new ClientException("ArticleNotFound", MessageFormat.format("Article not found: {0}", articleId));
        }
        if(userArticle.getReadDate() == null) {
            userArticle.setReadDate(new Date());
            userArticleDao.update(userArticle);

            ArticleDto article = articleDao.findFirstByCriteria(new ArticleCriteria().setId(userArticle.getArticleId()));
            List<FeedSubscriptionDto> subscriptions = feedSubscriptionDao.findByCriteria(
                new FeedSubscriptionCriteria().setFeedId(article.getFeedId()).setUserId(userId)
            );
            for (FeedSubscriptionDto sub : subscriptions) {
                feedSubscriptionDao.updateUnreadCount(sub.getId(), sub.getUnreadUserArticleCount() - 1);
            }
        }
```

⇒ ***Resource Layer Update:***

- ArticleResource.java

```java
package com.sismics.reader.rest.resource;

@Path("/article")
public class ArticleResource extends BaseResource {

    // In a production environment, use dependency injection instead.
    private ArticleService articleService = new ArticleServiceImpl(
            new UserArticleDaoImpl(), new ArticleDaoImpl(), new FeedSubscriptionDaoImpl()
    );

    @POST
    @Path("{id: [a-z0-9\\-]+}/read")
    @Produces(MediaType.APPLICATION_JSON)
    public Response read(@PathParam("id") String id) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        JSONObject response = articleService.markArticleRead(id, principal.getId());
        return Response.ok().entity(response).build();
    }

    @POST
    @Path("read")
    @Produces(MediaType.APPLICATION_JSON)
    public Response readMultiple(@FormParam("id") List<String> idList) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        JSONObject response = articleService.markArticlesRead(idList, principal.getId());
        return Response.ok().entity(response).build();
    }

```

- ***CategoryResource.java - Tight Coupling:***
  Similar to ArticleResource.java – Tight Coupling case, we segregate the CategoryResource.java code into 3 sections for better separation of concerns, and makes the code more modular and testable i.e. Resource layer, Service Layer and DAO Layer.

- ***SubscriptionResource.java – Tight Coupling:***
  Similar to ArticleResource.java – Tight Coupling case, we segregate the SubscriptionResource.java code into 3 sections for better separation of concerns, and makes the code more modular and testable i.e. Resource layer, Service Layer and DAO Layer.

- ***UserResource.java – Tight Coupling:***
  Similar to ArticleResource.java – Tight Coupling case, we segregate the UserResource.java code into 3 sections for better separation of concerns, and makes the code more modular and testable i.e. Resource layer, Service Layer and DAO Layer.

- ***Areas Where LLM Refactoring Excels:***

⇒ Code Formatting & Cleanup: Efficient removal of redundant code, optimization of imports, and enhancement of readability.
⇒ Encapsulation & Modularization: Consolidating repetitive logic into helper classes or services for a cleaner structure.
⇒ Implementing Design Patterns: Applying patterns such as Factory, Singleton, or Strategy to improve code maintainability.
⇒ Decoupling Responsibilities: Separating concerns effectively to adhere to the Single Responsibility Principle (SRP).

- ***Areas Where Manual Refactoring Outperforms:***

⇒ Deep Business Logic Understanding: Requires domain expertise; complex logic often needs contextual business insights that LLMs might lack.
⇒ Handling Large Dependencies & Side Effects: Better suited to identify and manage hidden dependencies that span across multiple services.
⇒ Performance & Memory Optimization: Helps prevent the introduction of unnecessary object creation or inefficient algorithms that can affect performance.
⇒ Testing & Validation: Manual testing is essential to catch regressions and validate changes made during refactoring.

- ***Conclusion:***

⇒ Manual Refactoring: Demonstrates a deep understanding of the codebase's unique requirements and complexities—especially in areas like security, authentication, and modularization. It ensures clarity, conciseness, and adherence to best practices, making it ideal for handling intricate functionalities and dependencies.

⇒ LLM Refactoring: Excels in tasks such as code formatting, restructuring, and enforcing general best practices. However, it lacks the nuanced comprehension of business logic and hidden dependencies, making it best suited for simpler, well-contained tasks like removing redundancy, enhancing readability, and generating initial code structures.

d. <u>Automating Refactoring Pipeline:</u>

This script automates the refactoring of Java code in a GitHub repository using Gemini AI. It operates through the following steps:

⇒ **Setup and Authentication:** It imports necessary libraries (git, github, google.generativeai) and authenticates with GitHub and Gemini AI via API keys.
⇒ **Repository Management:** The script checks if the repository (REPO_URL) is already cloned locally. If it exists, it pulls the latest updates; if not, it clones the repository.
⇒ **Code Refactoring:** It recursively scans for .java files and sends the contents of each file to Gemini AI for refactoring. The updated code then replaces the original.
⇒ **Git Operations:** Every 24 hours, a new branch (named auto-refactor-Timestamp) is created. All modified files are committed and the branch is pushed to GitHub.
⇒ **Automated Pull Request:** A pull request is automatically generated from the new branch to the default branch, including a description that highlights the AI-driven refactoring aimed at improving readability and structure.

This process runs every 24 hours, ensuring continuous improvements to the codebase.

<u>Project Contribution:</u>

- **Task 1 :** Ashish Lakhmani, Rohit Joshi, Rugved Thakare
- **Task 2 :** Ashish Lakhmani, Pratik Singh
- **Task 3 :** Rugved Thakare, Pratik Singh, Rohit Joshi