Documentation

Project 1

Team 27
C Swaroop (2022101114)
Masumi Desai (2022102057)
Manan Chichra (2022102058)
Samanvitha (2022102027)
Shravan Gadbail (2022102025)

Feed Organisation Subsystem :

1. Feed Ordering

This subsystem is responsible for managing the ordering, synchronization, and organization of RSS feeds and their associated articles. It ensures that feeds are fetched, ordered, and updated correctly.

Classes in Feed Ordering

1.1 FeedService

- **Functionality:**
    - Manages the synchronization of RSS feeds and the ordering of articles within those feeds.
    - Provides methods to retrieve the oldest and newest articles, as well as articles to remove.
- **Key Methods:**

    - **synchronize(String url):** Fetches and updates the feed from the provided URL.
    - **getOldestArticle(List<Article> articleList):** Retrieves the oldest article from a list.
    - **getNewerArticleList(List<Article> articleList, Article oldestArticle):** Retrieves articles newer than the specified oldest article.
    - **getArticleToRemove(List<Article> articleList):** Identifies articles to remove based on criteria.
- **Behavior:**
    - Interacts with FeedSubscriptionDao, ArticleDao, and RssReader to fetch and manage feeds and articles.
    - Ensures feeds are synchronized and articles are ordered correctly.
- **Relationships:**
    - Uses FeedSubscriptionDao to manage feed subscriptions.
    - Uses ArticleDao to manage articles.
    - Uses RssReader to fetch RSS feeds.

1.2 FeedSubscriptionDao

- **Functionality**:
    - Handles database operations related to feed subscriptions, including reordering, moving, and updating subscriptions.
- **Key Methods:**
    - **reorder(FeedSubscription feedSubscription, int order):** Reorders a subscription.
    - **moveUp(FeedSubscription feedSubscription):** Moves a subscription up in the order.

    - **moveDown(FeedSubscription feedSubscription):** Moves a subscription down in the order.
    - **getNextOrderValue(String userId):** Retrieves the next order value for a user's subscriptions.
    - **adjustOrderAfterDeletion(FeedSubscription feedSubscription):** Adjusts the order after a subscription is deleted.
    - **getSortedSubscriptions(FeedSubscriptionCriteria criteria):** Retrieves sorted subscriptions based on criteria.
    - **updateUnreadCount(String feedSubscriptionId, int unreadCount):** Updates the unread count for a subscription.
- **Behavior:**
    - Manages the ordering and state of feed subscriptions in the database.
    - Ensures subscriptions are correctly ordered and updated.
- **Relationships:**
    - Manages FeedSubscription entities.
    - Uses FeedSubscriptionCriteria for filtering and sorting.

1.3 FeedSubscription

- **Functionality:**
    - Represents a user's subscription to a feed, including its order and metadata.
- **Key Attributes:**
    - order: The order of the subscription in the user's list.
- **Key Methods:**
    - **getOrder():** Retrieves the order of the subscription.

    - **setOrder(int order):** Sets the order of the subscription.
    - **compareTo(FeedSubscription other):** Compares this subscription to another based on order.
- **Behavior:**
    - Encapsulates the state of a feed subscription.
    - Used by FeedSubscriptionDao for ordering and management.
- **Relationships:**
    - Managed by FeedSubscriptionDao.
    - Mapped to FeedSubscriptionDto by FeedSubscriptionMapper.

1.4 FeedSubscriptionCriteria

- **Functionality:**
    - Encapsulates criteria for querying and sorting feed subscriptions.
- **Key Attributes:**
    - sortCriteria: The sorting criteria for subscriptions.
- **Key Methods:**
    - **setSortCriteria(SortCriteria sortCriteria):** Sets the sorting criteria.
    - **getSortCriteria():** Retrieves the sorting criteria.
    - **applyFilters():** Applies filters to the query.
    - **setUserId(String userId):** Sets the user ID for filtering.
- **Behavior:**
    - Used by FeedSubscriptionDao to filter and sort subscriptions.
- **Relationships:**
    - Extends SortCriteria.

- Used by FeedSubscriptionDao.

1.5 FeedSubscriptionMapper

- **Functionality:**
  - Maps between FeedSubscription entities and FeedSubscriptionDto objects.
- **Key Methods:**
  - **mapToDto(FeedSubscription entity):** Maps an entity to a DTO.
  - **mapToEntity(FeedSubscriptionDto dto):** Maps a DTO to an entity.
- **Behavior:**
  - Facilitates data transfer between layers by converting entities to DTOs and vice versa.
- **Relationships:**
  - Maps FeedSubscription to FeedSubscriptionDto.

2. Marking as Read and Starring

This subsystem handles the marking of articles as read or unread, as well as starring and unstarring articles.

Classes in Marking as Read and Starring Subsystem

2.1 StarredResource

- **Functionality:**
  - Manages operations related to starring and unstarring articles.
- **Key Methods:**

    - **get(limit: int, afterArticle: String)**: Retrieves starred articles.
    - **star(id: String):** Stars an article.
    - **unstar(id: String)**: Unstars an article.
    - **starMultiple(idList: List<String>):** Stars multiple articles.
    - unstarMultiple(idList: List<String>): Unstars multiple articles.
- **Behavior:**
    - Interacts with UserArticleDao to update the starred status of articles.
- **Relationships:**
    - Inherits from BaseResource.
    - Uses UserArticleDao to manage starred articles.

2.2 ArticleResource

- **Functionality**:
    - Handles marking articles as read or unread.
- **Key Methods:**
    - readMultiple(context: Context, idList: Set<String>, responseHandler: JsonHttpResponseHandler): Marks multiple articles as read.
    - unreadMultiple(context: Context, idList: Set<String>, responseHandler: JsonHttpResponseHandler): Marks multiple articles as unread.
    - unread(context: Context, id: String, responseHandler: JsonHttpResponseHandler): Marks a single article as unread.
- **Behavior:**
    - Updates the read status of articles in the database.
- **Relationships:**
    - Inherits from BaseResource.
    - Uses UserArticleDao to manage article read status.

2.3 UserArticleDao

- **Functionality:**
    - Manages database operations related to user-specific article data, including read status and starring.
- **Key Methods:**
    - markAsRead(UserArticleCriteria criteria): Marks articles as read based on criteria.
    - create(UserArticle userArticle): Creates a new user article entry.
    - update(UserArticle userArticle): Updates an existing user article entry.
    - delete(String id): Deletes a user article entry.
    - getUserArticle(String id, String userId): Retrieves a user article by ID and user ID.
- **Behavior:**
    - Manages the state of user-specific article data, including read status and starring.
- **Relationships:**
    - Manages UserArticle entities.
    - Used by StarredResource and ArticleResource.

3. Folder Organization

This subsystem manages the organization of feeds and articles into folders or categories.

Classes in Folder Organization

3.1 Category

- **Functionality:**

    - Represents a folder or category for organizing feeds and articles.
- **Key Attributes:**
    - id: Unique identifier for the category.
    - userId: ID of the user who owns the category.
    - parentId: ID of the parent category (for nested categories).
    - name: Name of the category.
    - order: Order of the category in the user's list.
    - folded: Indicates whether the category is folded (collapsed).
- **Key Methods:**
    - **getId():** Retrieves the category ID.
    - **getName()**: Retrieves the category name.
    - **getOrder():** Retrieves the category order.
    - **setOrder(Integer):** Sets the category order.
    - **isFolded():** Checks if the category is folded.
- **Behavior:**
    - Encapsulates the state of a category.
    - Used by CategoryDao for management.
- **Relationships:**
    - Managed by CategoryDao.
    - Contains FeedSubscription and UserArticle.

3.2 CategoryDao

- **Functionality:**
    - Handles database operations related to categories.
- **Key Methods:**
    - **create(Category):** Creates a new category.
    - **update(Category):** Updates an existing category.

    - **delete(String):** Deletes a category.
    - **getRootCategory(String):** Retrieves the root category for a user.
    - **findSubCategory(String, String):** Finds subcategories for a given category.
    - **reorder(Category, int):** Reorders a category.
- **Behavior:**
    - Manages the creation, updating, and deletion of categories.
    - Ensures categories are correctly ordered and nested.
- **Relationships:**
    - Manages Category entities.
    - Used by FeedService for organizing feeds.

4. Sharing

This subsystem handles the sharing of articles through various platforms.

Classes in Sharing Subsystem

4.1 ArticleActivity

- **Functionality:**
    - Manages the UI and logic for sharing articles.
- **Key Attributes:**
    - shareIntent: Intent used for sharing.
    - starMenuItem: Menu item for starring articles.
    - viewPager: ViewPager for navigating articles.

    - sharedAdapterHelper: Helper for managing shared articles.
- **Key Methods:**
    - **onCreate(Bundle):** Initializes the activity.
    - **onCreateOptionsMenu(Menu):** Creates the options menu.
    - **onOptionsItemSelected(MenuItem):** Handles menu item selection.
    - **updateActionBar():** Updates the action bar.
- **Behavior:**
    - Handles sharing of articles through social media, email, etc.
- **Relationships:**
    - Uses Intent for sharing.
    - Uses SharedArticlesAdapterHelper for managing shared articles.

4.2 SharedArticlesAdapterHelper

- **Functionality:**
    - Manages the state and data of shared articles.
- **Key Attributes:**
    - articleItems: List of shared articles.
    - adapters: Set of adapters for displaying articles.
    - listeners: Set of listeners for article events.
- **Key Methods:**
    - **getInstance():** Retrieves the singleton instance.
    - **getArticleItems()**: Retrieves the list of shared articles.
    - **addAdapter(Object, ArticlesHelperListener):** Adds an adapter.
    - **removeAdapter(Object, ArticlesHelperListener):** Removes an adapter.

    - **onDataChanged():** Notifies listeners of data changes.
- **Behavior:**
    - Manages the state of shared articles and notifies listeners of changes.
- **Relationships:**
    - Used by ArticleActivity for managing shared articles.

Observations :

Strengths

a. Modular and Layered Architecture

The system is divided into four main packages, each with a distinct responsibility:

- **Feed Ordering**: Manages feed synchronization, subscriptions, and article ordering.
- **Marking as Read and Starring**: Handles user interactions like starring articles and marking them as read/unread.
- **Sharing**: Provides sharing functionality via social media, email, etc.
- **Folder Organization**: Manages hierarchical organization of feeds and articles.

This modular structure ensures minimal cross-impact when changes are made, adhering to the **Single Responsibility Principle (SRP)**.

b. Reusability of Components

- **SortCriteria** and **FilterCriteria** are used across multiple classes, ensuring consistency and reducing redundancy.
- **BaseResource** and **BaseDao** provide shared functionality, promoting code reuse.

c. Clear Separation of Concerns

- **FeedService** manages high-level feed operations, while **FeedSubscriptionDao** and **ArticleDao** handle data access.
- **StarredResource** and **ArticleResource** manage user interactions separately from **UserArticleDao**, ensuring business logic remains distinct from data operations.

d. Use of Design Patterns

- **Inheritance**: Used in **BaseResource, BaseDao, and SortCriteria** to eliminate redundancy.
- **Composition**: Ensures ownership relationships (e.g., **FeedService** owns **FeedSubscriptionDao** and **ArticleDao**) for better lifecycle management.
- **Aggregation**: Allows flexible interactions (e.g., **AllResource** references **UserArticleDao** and **FeedSubscriptionDao**).

e. Efficient Data Management

- The **DAO (Data Access Object) pattern** (**FeedSubscriptionDao, ArticleDao, UserArticleDao**) abstracts database operations, improving maintainability.
- **Criteria Classes** (**FeedSubscriptionCriteria, UserArticleCriteria**) enable dynamic queries, supporting filtering, sorting, and pagination.

Weaknesses

a. Tight Coupling in Feed Ordering

- **FeedService** is tightly coupled with **FeedSubscriptionDao, ArticleDao, and RssReader**, making unit testing difficult.

b. Lack of Asynchronous Operations

- The system processes feeds sequentially, which may cause delays with large datasets.

c. Scalability Concerns

- **FeedService** synchronizes feeds **sequentially**, limiting scalability.
- **ArticleDao** and **UserArticleDao** do not support **pagination** or **batch processing**, leading to performance issues.

Assumptions:

- Feed synchronization and article updates are performed sequentially, with no concurrent modifications.
- The network and database are always available and responsive.

User Management Subsystem

Below is the list of various classes included in this subsystem along with their main attributes, methods and functionality.

1. User

Functionality: The **User** class represents a user in the system. It encapsulates all core user attributes such as ID, username, password, email, role, and locale, and serves as the central entity for user-related operations.

Methods/Attributes:

- **id**: *String* - Unique identifier for the user.
- **username**: *String* - The username used for login and identification.
- **password**: *String* - The user's password (stored in plaintext or hashed).
- **email**: *String* - The user's email address.
- **roleId**: *String* - The role assigned to the user (e.g., admin, user).
- **localeId**: *String* - The user's preferred locale (e.g., language and region).
- **createDate**: *Date* - The date and time when the user account was created.
- **deleteDate**: *Date* - The date and time when the user account was deleted (if applicable).

Behavior:

- The **User** class is a data model that stores user information and is used across the subsystem for CRUD operations, authentication, and authorization.
- It is persisted in a database and accessed via **DAO** classes or repositories.
- It is related to **AuthenticationToken** through composition, as a user can own multiple tokens for session management.

Relationship to Subsystem:

- The **User** class is the core entity of the **User Management** subsystem. All other classes (e.g., **UserResource**, **AuthenticationToken**) revolve around managing or interacting with **User** objects.

2. AuthenticationToken

Functionality: The **AuthenticationToken** class represents a session token used for user authentication. It tracks token creation, last connection time, and whether the token is long-lived or short-lived.

Methods/Attributes:

- **id**: *String* - Unique identifier for the token.
- **userId**: *String* - The ID of the user associated with the token.
- **creationDate**: *Date* - The date and time when the token was created.
- **lastConnectionDate**: *Date* - The date and time of the user's last connection using this token.
- **longLasted**: *boolean* - Indicates whether the token is long-lived (e.g., "remember me" tokens).

Behavior:

- Tokens are created during login and deleted during logout or when they expire.
- The **AuthenticationTokenDao** class manages the persistence and life cycle of tokens.
- Tokens are validated by the **CookieAuthenticationFilter** during each request to ensure the user is authenticated.

Relationship to Subsystem:

- The **AuthenticationToken** class is essential for session management. It enables secure user authentication and maintains user sessions across requests.
- It has a composition relationship with **User**, as tokens cannot exist without an associated user.

3. UserPrincipal

Functionality: The **UserPrincipal** class implements the **IPrincipal** interface and represents the security context of an authenticated user. It provides user details such as ID, name, locale, and timezone for authorization purposes.

Methods/Attributes:

- **id**: *String* - The ID of the authenticated user.

- **name**: *String* - The name of the authenticated user.
- **locale**: *Locale* - The user's preferred locale.
- **dateTimeZone**: *DateTimeZone* - The user's preferred timezone.
- **email**: *String* - The user's email address.
- **isAnonymous()**: *boolean* - Checks if the user is anonymous (not authenticated).
- **getId()**: *String* - Returns the user's ID.
- **getName()**: *String* - Returns the user's name.
- **getLocale()**: *Locale* - Returns the user's locale.
- **getDateTimeZone()**: *DateTimeZone* - Returns the user's timezone.
- **getEmail()**: *String* - Returns the user's email.

Behavior:

- The **UserPrincipal** class is created during authentication and used by security filters (e.g., **CookieAuthenticationFilter**) to authorize requests.
- It decouples security concerns from the **User** class, ensuring that sensitive user data is not exposed unnecessarily.

Relationship to Subsystem:

- The **UserPrincipal** class is critical for authorization. It ensures that only authenticated users can access protected resources.
- It implements the **IPrincipal** interface, providing a standardized way to interact with user security contexts.

4. UserResource

Functionality: The **UserResource** class provides REST API endpoints for user management (e.g., register, update, delete) and authentication (e.g., login, logout).

Key Methods:

- **register(User)**: *void* - Registers a new user.
- **login(String username, String password)**: *AuthenticationToken* - Authenticates a user and returns a session token.

- **logout(String tokenId)**: *void* - Logs out a user by deleting their session token.
- **changePassword(String userId, String newPassword)**: *void* - Updates a user's password.
- **info(String userId)**: *User* - Retrieves user information.

Behavior:

- The **UserResource** class acts as the entry point for all user-related HTTP requests.
- It interacts with **AuthenticationTokenDao** for token management and **ApplicationContext** for user state management.
- It raises events (e.g., **UserCreatedEvent**, **PasswordChangedEvent**) to notify other subsystems of changes.

Relationship to Subsystem:

- The **UserResource** class is the public interface of the **User Management** subsystem. It exposes functionality to external clients (e.g., mobile apps, web apps).
- It depends on **User**, **AuthenticationToken**, and **AuthenticationTokenDao** to perform its operations.

5. AuthenticationTokenDao

Functionality: The **AuthenticationTokenDao** class is a Data Access Object (DAO) that manages CRUD operations for **AuthenticationToken** entities.

Key Methods:

- **get(String id)**: *AuthenticationToken* - Retrieves a token by its ID.
- **create(AuthenticationToken)**: *String* - Creates a new token and returns its ID.
- **delete(String tokenId)**: *void* - Deletes a token by its ID.
- **deleteOldSessionToken(String userId)**: *void* - Deletes expired tokens for a user.

- **updateLastConnectionDate(String id)**: *void* - Updates the last connection date for a token.

Relationship to Subsystem:

- The **AuthenticationTokenDao** class is essential for session persistence. It ensures that tokens are stored securely and can be retrieved or deleted as needed.
- It depends on the **AuthenticationToken** class to perform its operations.

6. ApplicationContext

Functionality: The **ApplicationContext** class is a singleton that holds global application state, such as the currently logged-in user.

Key Methods:

- **getInstance()**: *ApplicationContext* - Returns the singleton instance of **ApplicationContext**.
- **fetchUserInfo()**: *void* - Retrieves user information from the server.
- **setUserInfo(User)**: *void* - Updates the currently logged-in user.
- **isLoggedIn()**: *boolean* - Checks if a user is logged in.

Behavior:

- The **ApplicationContext** class maintains the state of the logged-in user across the application.
- It interacts with **PreferenceUtil** to persist user data (e.g., cached user info).

Relationship to Subsystem:

- The **ApplicationContext** class is critical for state management. It ensures that user state is consistent across the application.
- It depends on **User** and **PreferenceUtil** to perform its operations.

7. CookieAuthenticationFilter

Functionality: The **CookieAuthenticationFilter** class validates authentication tokens from cookies and creates **UserPrincipal** objects for authenticated users.

Key Methods:

- **doFilter(ServletRequest, ServletResponse, FilterChain)**: *void* - Validates tokens and authorizes requests.

Behavior:

- The **CookieAuthenticationFilter** class intercepts incoming requests, extracts tokens from cookies, and validates them using **AuthenticationTokenDao**.
- It creates a **UserPrincipal** object for authenticated users and passes it to the next filter or controller.

Relationship to Subsystem:

- The **CookieAuthenticationFilter** class is essential for request authorization. It ensures that only authenticated users can access protected resources.
- It depends on **AuthenticationToken** and **UserPrincipal** to perform its operations.

Observations and Comments

Strengths

- **Separation of Concerns:** The subsystem is well-structured, with clear boundaries between entities (**User**), security (**UserPrincipal**), and REST endpoints (**UserResource**). For example, **UserPrincipal** decouples security concerns from the **User** class, ensuring that sensitive data is not exposed unnecessarily.
- **Persistence Abstraction:** The **AuthenticationTokenDao** class isolates database operations, adhering to the **Single Responsibility Principle (SRP)**.

This makes it easier to switch persistence mechanisms (e.g., from raw SQL to an ORM framework).
- **Modular Design:** The use of interfaces (e.g., **IPrincipal**) and composition (e.g., **User** owns **AuthenticationToken**) promotes modularity and reusability.
- **Session Management:** The **AuthenticationToken** class effectively manages user sessions, supporting both short-lived and long-lived tokens.

Weaknesses

- **God Class:** The **UserResource** class handles too many responsibilities, including user CRUD, authentication, and password management. This violates the **Single Responsibility Principle (SRP)** and makes the class difficult to maintain and test.
- **Primitive Obsession:** Overuse of primitive types (e.g., **String** for IDs, **boolean** for token longevity) reduces type safety and readability. For example, **userId: String** could be replaced with a **UserId** value object.
- **Security Gaps:** Sensitive fields like **User.password** are stored without encryption, posing a security risk. Token validation lacks mechanisms for **rate-limiting** or **revocation**, making the system vulnerable to brute-force attacks.
- **Tight Coupling:** The **ApplicationContext** class is tightly coupled with **PreferenceUtil** and **User**, hindering testability and scalability. For example, **ApplicationContext** directly manipulates **PreferenceUtil** for caching user data.
- **Lack of Abstraction:** The **AuthenticationTokenDao** class uses raw SQL queries instead of an abstract repository interface, violating the **Dependency Inversion Principle (DIP)**.

Assumptions

I assumed only the classes which involve either add, delete, update users or password change are relevant for this subclass as it was mentioned in the project documentation.

Subscription and Content subsystem:

I)Description of key classes:

1.Domain Entities-

Feed

Functionality: Represents an RSS feed, storing metadata and URL details, and handling updates when new articles are fetched.

Key Methods and Attributes:

- `id: String` – Unique identifier for the feed.
- `rssUrl: String` – URL of the RSS feed.
- `url: String` – Main website URL.
- `baseUri: String` – Base URI for resolving relative links.
- `title, language, description: String` – Metadata fields.
- `createDate, lastFetchDate, deleteDate: Date` – Lifecycle tracking attributes.
- `synchronizeFeed(): void` – Updates feed with new articles.
- `updateMetadata(): void` – Refreshes feed title, description, and favicon.

Behavior:

- A new feed is created during synchronization.
- Metadata is refreshed periodically or upon user action.
- Articles within the feed are fetched, parsed, and stored.

Relationship to Subsystem:

- Central to the content subsystem.
- Works with `FeedSubscription`, `Article`, and `FeedDao`.

Article

Functionality: Represents an individual article retrieved from an RSS feed, storing its details and ensuring safe rendering.

Key Methods and Attributes:

- `id: String` – **Unique identifier.**
- `feedId: String` – **Associated feed.**
- `title, content, link, author: String` – **Article metadata.**
- `publishDate, fetchDate: Date` – **Timestamps for article retrieval.**
- `sanitizeContent(): String` – **Ensures the article is displayed correctly and safely.**
- `markAsRead(userId: String): void` – **Updates read status for a user.**

Behavior:

- **Articles are extracted from the RSS feed using `RssReader`.**
- **Content is sanitized before being displayed.**
- **Articles are stored and indexed for fast retrieval.**

Relationship to Subsystem:

- **Core entity in content management.**
- **Works with `Feed`, `UserArticle`, and `ArticleDao`.**

FeedSubscription

Functionality: Links a user to a feed, tracking their subscription and unread article count.

Key Methods and Attributes:

- `id, userId, feedId: String` – Identifiers.
- `categoryId: String` – Organizes subscriptions.
- `title: String` – Optional override of feed title.
- `unreadCount: int` – Tracks unread articles.
- `subscribe(userId, feedId): void` – Creates a subscription.
- `markArticlesAsRead(): void` – Updates unread count.

Behavior:

- Created when a user adds a feed manually or via OPML import.
- Unread count updates as new articles arrive.
- Deleted when a user unsubscribes.

Relationship to Subsystem:

- Part of both the subscription and content subsystems.
- Works with `Feed`, `User`, and `FeedSubscriptionDao`.

Category

Functionality: Allows users to group feed subscriptions into categories like "Tech News" or "Sports".

Key Methods and Attributes:

- `id, userId: String` – Identifiers.
- `parentId: String` – Supports hierarchical categorization.
- `name: String` – Category name.

- `order: int` – Display order.
- `createCategory(userId, name): void` – Adds a new category.
- `deleteCategory(id): void` – Removes a category.

Behavior:

- Created automatically during OPML import if missing.
- Organizes subscriptions visually in the UI.
- Users can customize and reorder categories.

Relationship to Subsystem:

- Part of the subscription subsystem.
- Works with `FeedSubscription`.

UserArticle

Functionality: Tracks whether a user has read or starred an article.

Key Methods and Attributes:

- `id, userId, articleId: String` – Identifiers.
- `readDate, starredDate: Date` – User interactions with the article.
- `markAsRead(): void` – Updates read status.
- `toggleStarred(): void` – Marks/unmarks an article as favorite.

Behavior:

- Created when a user reads an article.
- Determines unread article count.
- Allows users to save favorite articles.

Relationship to Subsystem:

- Part of the content subsystem.
- Works with `Article` and `User`.

2.Data Access Objects (DAOs)

1. FeedDao

Functionality

The `FeedDao` class is responsible for handling all database operations related to RSS feeds. It interacts with the persistence layer to store, retrieve, update, and delete feed information.

Key Attributes

- **None** (DAO classes typically do not maintain state; they execute database queries.)

Key Methods

- `create(Feed feed)`: Inserts a new feed into the database.
- `update(Feed feed)`: Updates an existing feed's metadata.
- `delete(String feedId)`: Deletes a feed based on its unique identifier.
- `getFeedById(String feedId)`: Retrieves a specific feed using its unique identifier.
- `getFeedsForUser(String userId)`: Fetches all feeds that a given user has subscribed to.
- `getAllFeeds()`: Retrieves all available feeds in the system.
- `getFeedsToSync(int limit)`: Retrieves feeds that need to be synchronized based on last fetch time.

Behavior

- This class interacts with the **Feed** entity in the database.

- It enables retrieval of feed information when users want to browse their subscribed feeds.
- Supports bulk retrieval of feeds for synchronization, ensuring new articles are regularly fetched.
- Prevents duplicate feeds by checking RSS URLs before insertion.

Relationship to the Subscription & Content Subsystem

- The `FeedDao` plays a crucial role in managing the **Subscription & Content Subsystem** by ensuring that feed data is correctly stored and retrieved.
- It is used by **FeedService** and other service classes to synchronize feeds and provide content to users.

2. ArticleDao

Functionality

The `ArticleDao` class handles the persistence and retrieval of articles that belong to RSS feeds. It ensures articles are efficiently stored and managed.

Key Attributes

- **None (DAO classes typically do not maintain state; they execute database queries.)**

Key Methods

- **`insertArticles(List<Article> articles)`: Inserts multiple new articles into the database.**
- **`deleteOldArticles(int retentionDays)`: Deletes articles older than a certain number of days.**
- **`getArticlesByFeedId(String feedId)`: Retrieves all articles belonging to a specific feed.**

- **`getArticleById(String articleId)`: Fetches a specific article by its ID.**
- **`searchArticles(String query)`: Performs a search across article content based on user queries.**

Behavior

- **This class manages the Article entity, ensuring articles fetched from RSS feeds are stored and retrieved efficiently.**
- **Implements retention policies to prevent database overload by deleting outdated articles.**
- **Enables keyword-based searches for users.**

Relationship to the Subscription & Content Subsystem

- **`ArticleDao` is a core component of the Content Subsystem, ensuring articles from RSS feeds are stored properly.**
- **It interacts with FeedService and the UserArticleDao to track read/unread articles and article metadata.**

. FeedSubscriptionDao

Functionality

The `FeedSubscriptionDao` class manages the relationship between users and the feeds they subscribe to. It enables users to subscribe, unsubscribe, and organize feeds.

Key Attributes

- **None** (DAO classes typically do not maintain state; they execute database queries.)

Key Methods

- `addSubscription(FeedSubscription subscription)`: Adds a new subscription to a user's account.
- `removeSubscription(String subscriptionId)`: Removes a user's subscription.
- `getSubscriptionsByUserId(String userId)`: Fetches all feeds subscribed to by a given user.
- `updateUnreadCount(String subscriptionId, int newCount)`: Updates the unread article count for a given subscription.

Behavior

- Manages user subscriptions, ensuring that they can follow and unfollow feeds.
- Keeps track of unread article counts for each subscription.
- Supports category-based organization of feeds.

Relationship to the Subscription & Content Subsystem

- `FeedSubscriptionDao` is essential for managing user interactions within the **Subscription Subsystem**.
- It works with **FeedDao** to retrieve subscription data and ensure feeds are correctly linked to users.

4. UserArticleDao

Functionality

The `UserArticleDao` class tracks user interactions with articles, including whether an article has been read or starred. It plays a key role in personalizing the user experience.

Key Attributes

- **None** (DAO classes typically do not maintain state; they execute database queries.)

Key Methods

- `markAsRead(String userId, String articleId)`: Marks an article as read for a user.
- `markAsStarred(String userId, String articleId)`: Marks an article as starred (saved for later).
- `getUnreadArticlesByUser(String userId)`: Retrieves all unread articles for a specific user.
- `removeUserArticleRecord(String userId, String articleId)`: Deletes a user-specific article record when necessary.

Behavior

- Ensures that user preferences (read/unread/starred status) are saved.
- Allows users to manage their reading history and saved articles.
- Helps in generating unread article counts for subscriptions.

Relationship to the Subscription & Content Subsystem

- This class forms the bridge between the **Subscription Subsystem** and the **Content Subsystem** by tracking how users engage with articles.
- Works with **ArticleDao** to retrieve article data for specific users.

3.Import and Parsing Components-

) OpmlReader

Functionality

The `OpmlReader` class is responsible for parsing OPML (Outline Processor Markup Language) files, which are commonly used for importing and exporting lists of RSS feed subscriptions.

Key Attributes

- **None** (This class functions as a parser and does not maintain persistent state.)

Key Methods

- `parseOpml(String opmlContent)`: Parses the OPML file content.
- `extractSubscriptions(Document opmlDocument)`: Extracts the list of feed URLs and categories from the OPML structure.
- `validateOpmlFormat(String opmlContent)`: Ensures that the OPML file is well-formed and meets required standards.

Behavior

- Reads OPML file content and extracts feed subscription details.
- Extracted feed URLs and categories are passed to the **Subscription Creation Process** for further processing.
- Ensures a valid OPML format before processing to avoid errors in importing feeds.

Relationship to the Subscription & Content Subsystem

- Works with `SubscriptionImportAsyncListener` to import user subscriptions from OPML files.
- Plays a key role in onboarding users by allowing them to import feeds in bulk.

b) RssReader

Functionality

The `RssReader` class is responsible for parsing RSS feeds and extracting article data, ensuring content is retrieved and structured correctly.

Key Attributes

- **`feed`: Stores the parsed feed metadata.**
- **`articleList`: Stores the list of parsed articles extracted from the feed.**
- **`fatalErrorCount`: Tracks the number of parsing failures.**

Key Methods

- **`parseRss(String rssUrl)`: Fetches and parses the RSS feed from the given URL.**
- **`extractArticles(Document rssDocument)`: Extracts articles from the parsed RSS XML document.**
- **`handleParsingErrors(Exception e)`: Logs and handles errors during RSS feed parsing.**

Behavior

- **Reads RSS XML data from a given feed URL.**
- **Extracts article content, sanitizes it, and prepares it for storage.**
- **Implements error handling to skip faulty feeds without breaking the entire synchronization process.**

Relationship to the Subscription & Content Subsystem

- **Works with `FeedService` to process feed content and extract articles.**
- **Ensures articles are properly formatted before being stored in the database.**

. Service Layer

a) FeedService

Functionality

The `FeedService` class is responsible for managing feed synchronization, article retrieval, and updating user subscriptions.

Key Attributes

- **None (Service classes typically operate through methods and do not store persistent state.)**

Key Methods

- `syncFeeds()`**: Triggers synchronization for all feeds to fetch new articles.**
- `fetchNewArticles(Feed feed)`**: Retrieves new articles for a given feed.**
- `updateFeedMetadata(Feed feed)`**: Updates metadata (title, description, etc.) for a feed.**
- `removeInactiveFeeds()`**: Deletes feeds that are no longer active or have been removed.**

Behavior

- **Periodically synchronizes all feeds to ensure new content is retrieved.**
- **Works with `RssReader` to fetch new articles and process them.**
- **Manages feed metadata updates and ensures user subscriptions are up to date.**

Relationship to the Subscription & Content Subsystem

- **The core service responsible for ensuring users have up-to-date feed content.**
- **Works closely with `RssReader`, `FeedDao`, and `ArticleDao` to retrieve and store article data.**

. Event Listeners and Asynchronous Processing

a) SubscriptionImportAsyncListener

Functionality

The `SubscriptionImportAsyncListener` class handles the asynchronous import of user subscriptions, ensuring the process does not block user interactions.

Key Attributes

- **None (Event listener classes listen for triggers and process events asynchronously.)**

Key Methods

- **`onSubscriptionImportEvent(SubscriptionImportEvent event)`: Listens for subscription import events and processes them.**
- **`processOpmlFile(String opmlFilePath)`: Extracts feed data from an OPML file and adds subscriptions.**
- **`handleImportErrors(Exception e)`: Manages errors that occur during the import process.**

Behavior

- **Listens for subscription import events triggered when a user imports feeds via OPML.**

- **Processes OPML files asynchronously to ensure large imports do not slow down the system.**
- **Creates feeds and user subscriptions based on imported data.**

Relationship to the Subscription & Content Subsystem

- **Works with `OpmlReader` to parse imported subscription data.**
- **Updates user subscription records using `FeedSubscriptionDao`.**

6. Utility Classes

a) ReaderHttpClient

Functionality

The `ReaderHttpClient` class is a utility class that handles HTTP requests for fetching RSS feed content from external websites.

Key Attributes

- **None (This class functions as a network client and does not maintain state.)**

Key Methods

- **`fetchUrlContent(String url)`: Sends an HTTP request to retrieve content from a given URL.**
- **`configureSslSettings()`: Configures SSL settings to securely connect to RSS feed URLs.**
- **`setRequestHeaders(Map<String, String> headers)`: Sets HTTP headers, including User-Agent and authentication details.**

- **`handleRedirects(HttpResponse response)`: Manages HTTP redirects to ensure the correct feed URL is followed.**

Behavior

- **Manages HTTP connections and redirects while fetching RSS feeds.**
- **Configures necessary SSL settings for secure feed retrieval.**
- **Prevents common HTTP errors such as request timeouts and connection failures.**

Relationship to the Subscription & Content Subsystem

- **Used by `RssReader` to fetch RSS XML content from remote sources.**
- **Ensures the system can retrieve feeds securely and efficiently.**

---

b) ArticleSanitizer

Functionality

The `ArticleSanitizer` class is responsible for cleaning and formatting article HTML content before storage and display.

Key Attributes

- **None (This class is purely utility-based and does not maintain state.)**

Key Methods

- **`sanitizeHtml(String rawHtml)`: Removes unwanted HTML tags and scripts to ensure safe rendering.**

- `stripMaliciousScripts(String content)`: Detects and removes potential XSS vulnerabilities from article content.
- `normalizeEncoding(String text)`: Ensures text encoding is consistent across all articles.

Behavior

- Cleans raw article HTML to remove unnecessary or dangerous content.
- Ensures articles are properly formatted for safe rendering in the UI.
- Prevents security risks such as cross-site scripting (XSS) attacks.

Relationship to the Subscription & Content Subsystem

- Works with `RssReader` to ensure all articles are sanitized before storage.
- Guarantees that feed content is displayed safely in the RSS Reader UI.

General Notes

- Import and Parsing Components ensure that RSS feeds and OPML files are processed efficiently.
- Service Layer handles synchronization and feed management.
- Event Listeners enable asynchronous processing for better performance.
- Utility Classes support network communication and security.

3. Observations and Comments,

Strengths-

- **Modular Design:**
  The system separates concerns among entities, DAOs, services, event listeners, and utility classes. This modularity makes it easier to maintain and extend.
- **Event-Driven Architecture:**
  Using synchronous and asynchronous event buses decouples long-running tasks (e.g., feed import, indexing, favicon updates) from user-triggered actions, improving scalability and responsiveness.
- **Robust Import Mechanism:**
  The import functionality handles multiple file formats (ZIP archives, OPML, Google Takeout's JSON) and uses helper classes (e.g., `OpmlReader`, `StarredReader`) to parse these files.
- **Scheduled Synchronization:**
  The FeedService uses a scheduled service to synchronize feeds periodically, ensuring that content is kept up-to-date without user intervention.
- **Full-Text Search Integration:**
  Integration with Lucene (through the IndexingService and ArticleDao search methods) provides powerful search capabilities over the articles.

Weaknesses-

No Caching for Frequently Accessed Data

- **Issue:** The system retrieves feeds and articles from the database or external sources every time they are requested.
- **Impact:** This can lead to performance issues due to repeated database queries.
- **Solution:** Introduce caching (e.g., Redis) for frequently accessed feeds, articles, and user subscriptions.

No Archiving or Expiration Policy for Old Articles

- **Issue:** All fetched articles are stored indefinitely.
- **Impact:** This can cause **unnecessary database bloat** over time.
- **Solution:** Implement **an automatic cleanup mechanism** for old articles.

No Support for Notifications

- **Issue:** Users are not notified when new articles are available.
- **Impact:** Users must **manually check** for new content.
- **Solution:** Implement **email or push notifications** for new articles.

4.Assumptions taken during the documentation process-

In designing the high-level class diagrams, several components were intentionally omitted to maintain clarity and focus on the core domain entities and key service components. The following elements were not included in the diagram:

1. **DTOs and Mapper Classes –** Data Transfer Objects (DTOs) and their associated mapping utilities were excluded to avoid clutter and focus on core business logic rather than data transformation layers.
2. **Utility Classes –** Helper classes that facilitate specific functionalities (e.g., string utilities, date formatters) were omitted since they do not directly contribute to the domain model.
3. **Event Handling Mechanisms –** Any event-driven interactions (such as feed update events or asynchronous message processing) were left out, as they do not fundamentally alter the structural relationships in the system.

A **UI layer** was introduced to represent the user input interface, where users can interact with the system by entering RSS feed URLs or importing feeds via OPML/XML files. This layer initiates actions in the system, acting as the entry point for feed subscription and article retrieval. The diagram assumes a scenario where a user inputs an RSS feed into the system, and from there, the backend processes the feed, retrieves articles, and displays them in the UI.

NOTE:

-DTOs are simple objects used to transfer data between different parts of the system, like from the backend to the UI.

-They help keep the actual database models separate from what is sent to users, making the system more secure and efficient.

-Mapper classes convert data between DTOs and actual database models (e.g., `FeedDTO` → `Feed`).

-They help keep the business logic clean by handling data transformation separately.

In the class diagram, DTOs and mappers are not included to keep the focus on the main system functions and relationships instead of data conversion details.

Task 2a Identifying design smells

Task 2b Code Metrics

Tools used: PMD, ck, Lizard, Sonarqube,

Code Metric 1: Cyclomatic Complexity Analysis

Done using Lizard.

Cyclomatic Complexity is more than 10 for the following functions:

Reader-web

1. **list() - CC = 12**

   File: SubscriptionResource

   Location: reader-web/src/main/java/com/sismics/reader/rest/resource/SubscriptionResource.java

   Lines: 71-158

2. **update() - CC = 15**

   File: UserResource

   Location: reader-web/src/main/java/com/sismics/reader/rest/resource/UserResource.java

   Line: 146-217

3. **info() - CC = 13**

   File: UserResource

   Location: reader-web/src/main/java/com/sismics/reader/rest/resource/UserResource.java

   Line: 501-579

Reader-core

1. **processImportFile()** - CC = 16
   File: SubscriptionImportAsyncListener
   Location:
   reader-core/src/main/java/com/sismics/reader/core/listener/async/SubscriptionImportAsyncListener.java
   Lines: 203-295
2. **importOutline()** - CC = 15
   File: SubscriptionImportAsyncListener
   Location:
   reader-core/src/main/java/com/sismics/reader/core/listener/async/SubscriptionImportAsyncListener.java
   Lines: 304-422

3. **importFeedFromStarred()** - CC = 13
   File: SubscriptionImportAsyncListener
   Location:
   reader-core/src/main/java/com/sismics/reader/core/listener/async/SubscriptionImportAsyncListener.java
   Lines: 431-510
4. **sanitize()** - CC = 14
   File: ArticleSanitizer
   Location: reader-core/src/main/java/com/sismics/reader/core/util/sanitizer/ArticleSanitizer.java
   Lines: 50-154
5. **handle()** - CC = 12
   File: TransactionUtil
   Location: reader-core/src/main/java/com/sismics/reader/core/util/TransactionUtil.java
   Lines: 28-83
6. **XmlReader()** - CC = 20
   File: XmlReader
   Location: reader-core/src/main/java/com/sismics/reader/core/dao/file/rss/XmlReader.java
   Lines: 52-102
7. **startElement()** - CC = 133
   File: RssReader
   Location: reader-core/src/main/java/com/sismics/reader/core/dao/file/rss/RssReader.java
   Lines: 246-412
8. **endElement()** - CC = 58
   File: RssReader
   Location: reader-core/src/main/java/com/sismics/reader/core/dao/file/rss/RssReader.java
   Lines: 415-485
9. **startElement()** - CC = 13
   File: OpmlReader
   Location: reader-core/src/main/java/com/sismics/reader/core/dao/file/opml/OpmlReader.java
   Lines: 81-114
10. **read()** - CC = 12
    File: StarredReader
    Location: reader-core/src/main/java/com/sismics/reader/core/dao/file/json/StarredReader.java
    Lines: 32-125

11. **getQueryParam()** - CC = 20
    File: UserArticleDao
    Location: reader-core/src/main/java/com/sismics/reader/core/dao/jpa/UserArticleDao.java
    Lines: 26-112
12. **synchronize()** - CC = 23
    File: FeedService
    Location: reader-core/src/main/java/com/sismics/reader/core/service/FeedService.java
    Lines: 138-316
13. **guessMimeType()** - CC = 20
    File: MimeTypeUtil
    Location: reader-core/src/main/java/com/sismics/util/mime/MimeTypeUtil.java
    Lines: 19-48
14. **list()** - CC = 13
    File: ResourceUtil
    Location: reader-core/src/main/java/com/sismics/util/ResourceUtil.java
    Lines: 32-87

Reader-web-common

1. **doFilter() - CC = 14**

   File: RequestContextFilter

   Location: reader-web-common/src/main/java/com/sismics/util/filter/RequestContextFilter.java

   Line: 91-151

So, in total there are **18 methods** with **CC** of greater than 10.

Code Metric 2: Code Duplication Percentage

Done using SonarQube.

Code Metric	reader-web	reader-core	reader-web-common
Code Duplication Percentage	1.9%	4.6%	1.2%

Code Metric 3: Lines of Code (Class-level and method-level) **(ck)**

Used CK for LOC.

The detailed report for Lines of Code (LOC) specific to the methods as well as classes, can be found in the report generated by CK.

The average LOC per class in the original code is: 51.2122641509434

The average LOC per method in the original code is: 8.901873327386262

The average LOC per class in reader-core original code is: 46.550898203592816

The average LOC per method in reader-core original code is: 7.465462274176408

The average LOC per class in reader-web original code is: 86.75

The average LOC per method in reader-web original code is: 29.209876543209877

The average LOC per class in reader-web-common original code is: 38.470588235294116

The average LOC per method in reader-web-common original code is: 5.9393939393939394

Code Metric 4: Fan in (ck)

Dataset: Full Project

The average fanin per class in Full Project original code is: 2.8632075471698113

The average fanin per method in Full Project original code is: 2.090990187332739

Dataset: reader-core

The average fanin per class in reader-core original code is: 2.3473053892215567

The average fanin per method in reader-core original code is: 1.622741764080765

Dataset: reader-web

The average fanin per class in reader-web original code is: 0.7857142857142857

The average fanin per method in reader-web original code is: 0.8148148148148148

Dataset: reader-web-common

The average fanin per class in reader-web-common original code is: 1.1764705882352942

The average fanin per method in reader-web-common original code is: 0.5959595959595959

Code Metric 5: Fan out (ck)

Dataset: Full Project

The average fanout per class in Full Project original code is: 5.221698113207547

The average fanout per method in Full Project original code is: 3.1507582515611063

Dataset: reader-core

The average fanout per class in reader-core original code is: 4.059880239520958

The average fanout per method in reader-core original code is: 1.842720510095643

Dataset: reader-web

The average fanout per class in reader-web original code is: 10.357142857142858

The average fanout per method in reader-web original code is: 0.9135802469135802

Dataset: reader-web-common

The average fanout per class in reader-web-common original code is: 5.647058823529412

The average fanout per method in reader-web-common original code is: 0.5959595959595959

Code Metric 6: DIT (ck)

Dataset: Full Project

The average dit per class in Full Project original code is: 1.3537735849056605

Dataset: reader-core

The average dit per class in reader-core original code is: 1.1497005988023952

Dataset: reader-web

The average dit per class in reader-web original code is: 1.7857142857142858

Dataset: reader-web-common

The average dit per class in reader-web-common original code is: 1.411764705882353

Code Metric 7: Number of Methods (ck)

Dataset: Full Project

The average totalMethodsQty per class in Full Project original code is: 5.278301886792453

Dataset: reader-core

The average totalMethodsQty per class in reader-core original code is: 5.622754491017964

Dataset: reader-web

The average totalMethodsQty per class in reader-web original code is: 2.892857142857143

Dataset: reader-web-common

The average totalMethodsQty per class in reader-web-common original code is: 5.823529411764706

Code Metric 8: Cognitive Complexity (SonarQube)

The cognitive complexity of the code Before refactoring: 2112

Cognitive complexity refers to the ease of understandability of the code.

Task 3a Refactoring:

Identifying Design Smells

1. Broken Hierarchy in ThemeResource

The ThemeResource class improperly inherits from BaseResource, creating a Broken Hierarchy due to the following reasons:

- **Incorrect Generalization:** BaseResource is designed for authentication and security, but ThemeResource does not require these features, leading to unnecessary coupling and violating the Single Responsibility Principle (SRP).
- **Hidden Dependencies:** ThemeResource relies on the request field from BaseResource, making it unclear where dependencies originate and reducing code maintainability.
- **Better Alternative –** Composition Over Inheritance: Instead of inheriting from BaseResource, ThemeResource should directly inject required dependencies (like SecurityService or HttpServletRequest) to improve modularity and maintainability.

2. Broken Hierarchy in ForbiddenClientException

- The ClientException class is meant to serve as the parent for all client-related exceptions.
- However, ForbiddenClientException breaks this hierarchy by skipping ClientException and directly inheriting from WebApplicationException.

Refactoring Approach

Remove Broken Hierarchy smell by properly inheriting from ClientException.

3. Deficient Encapsulation in AnonymousPrincipal

1. In the original design, AnonymousPrincipal is mutable because it allows external code to change its locale and dateTimeZone at any time. This means an AnonymousPrincipal instance can be modified after creation, which is undesirable for a security-related class. Once an instance of AnonymousPrincipal is injected into an HTTP request (via

injectAnonymousUser), another part of the system could accidentally or maliciously modify its state.

Refactoring Approach

Fixed this deficient encapsulation by intializing locale and dataTimeZone into constructor so that it can be set the first time an object is created but cannot be re-modified accidentally or maliciously by any other class throughout its lifetime.

4. Deficient Encapsulation in HeaderBasedSecurityFilter

Issue 1: The **enabled** flag is **mutable** (private boolean enabled;)

- This flag is set during init(), but **no safeguards prevent its modification later**.
- Since it controls authentication, accidental modification could lead to security vulnerabilities.

Refactoring Approach

Make enabled **final** and initialize it in the constructor to **ensure immutability**.

Issue 2: Direct UserDao Instantiation

- Authenticate() **directly instantiates UserDao**, violating encapsulation.
- This means **the filter is tightly coupled to UserDao**, making it **hard to test and replace** with another authentication mechanism.

Refactoring Approach

Inject UserDao via the constructor instead of creating it inside authenticate().

5. Unnecessary Hierarchy in ThemeResource

Issue: The ThemeResource class exhibits Unnecessary Hierarchy because it extends BaseResource but only relies on it for the request object from the BaseResource class. This hierarchy is unnecessary.

This is an issue because it creates tight coupling with no significant upside.

Refactoring approach:

Replace Inheritance with composition by injecting HttpServletRequest using @Context, improving encapsulation and reducing coupling.

6. Deficient Encapsulation in ThreadLocalContext:

Issue: The attribute threadLocalContext is set to public, though not accessed anywhere outside the class but is a potential issue because it is subject to being accessed/set.

Refactoring Approach:

Simple refactoring includes changing the variable to private. We can also include getter for it based on the functionality, but as it isn't used anywhere outside the class, we don't need to.

7. Deficient Encapsulation in Subscription:

Issue: Subscription class has all variables public with no getters and setters. This is deficient encapsulation because accessing class attributes directly will make it harder for future modifications.

Refactoring Approach:

Change the public variables to private, add getters and setters for every attribute and then make necessary changes in all the files where Subscription class has been used.

This may take time but is a long term solution for better encapsulation as it restricts direct access to fields, ensuring better data integrity and maintainability.

8. Broken Modularisation in AdBlockUtil:

Issue: The method getSubscriptions has the responsibility of parsing the adblock/subscriptions.xml file and then returning them as a list.

Refactoring Approach:

Move parsing logic in another helper method and call this method in getSubscriptions.

This promotes single responsibility and better readability of the code logic.

9. Wide Hierarchy in BaseResource:

Issue: All the classes in the directory reader-web/src/main/java/com/sismics/reader/rest/resource inherit from BaseResource. This creates the problem of Wide Hierarchy because BaseResource class has too many children.

Refactoring Approach:

Make BaseResource into a concrete class and replace inheritance with composition. This will remove the wide Hierarchy problem between all resource classes with BaseResource and also remove the unnecessary coupling between all these classes.

10. ArticleDto exhibits Insufficient Modularization

Issue: ArticleDto contains too many attributes and getters and setters for all those attributes. This leads to too much data in the ArticleDto class and too many primitive data types being used in one class.

Refactoring Approach:

We can group the data into multiple small classes which will promote single responsibility and better modularisation.

It will also promote better encapsulation because when the data is grouped, it is easier to manage and prevent it from unintended modifications.

11. Article Class exhibits Insufficient Modularization

Issue: The Article class currently handles too many responsibilities including:

Article metadata (ID, URL, GUID)

Content (title, description, creator)

Comments (commentUrl, commentCount)

Enclosures (enclosureUrl, enclosureLength, enclosureType)

Dates (publicationDate, createDate, deleteDate)

Refactoring Approach:

We can create additional classes such as ArticleMetada, Content, Enclosure, etc which can help shift some of the responsibilities of article class.

12. **Role Class exhibits Missing Abstraction**

Issue: The Role class appears to be an isolated entity that's not effectively integrated into the application's functionality. While it has a relationship with RoleBaseFunction, there's no service layer to manage role-related operations.

This can lead to:

Scattered role management logic across the application

Lack of centralized role-based access control

Difficulty in maintaining and extending role-related features

Refactoring Approach:

We'll create a service layer to properly utilize the Role abstraction by:

Creating a RoleService class to manage role operations

Adding necessary business logic methods

Ensuring proper integration with RoleBaseFunction

13. **Constants Class exhibits Broken Modularization**

Issue: The current Constants.java class exhibits broken modularization by mixing different types of constants that should be logically separated.

Refactoring Approach:

Split the current Constants.java into multiple focused constant classes:

UserConstants.java (for DEFAULT_LOCALE_ID, DEFAULT_TIMEZONE_ID, DEFAULT_THEME_ID, DEFAULT_ADMIN_PASSWORD, DEFAULT_USER_ROLE)

LuceneConstants.java (for LUCENE_DIRECTORY_STORAGE_RAM, LUCENE_DIRECTORY_STORAGE_FILE)

JobConstants.java (for all JOB_* related constants)

14. **Favicon Downloader cases exhibits Deficient Encapsulation**

Issue: The `FaviconDownloader` class in `com.sismics.reader.core.dao.file.html` package exhibits poor encapsulation and violates the Single Responsibility Principle (SRP) and The `FAVICON_MIME_TYPE_MAP` is publicly exposed, allowing external modification of MIME type mappings

Class has multiple responsibilities:

- MIME type management
- URL construction/validation

- File downloading
- File system operations

Refactoring Approach:

Create new classes:

- `MimeTypeConstants.java`: Encapsulate MIME type mappings
- `FaviconUrlBuilder.java`: Handle URL construction logic

Refactor `FaviconDownloader.java`:

- Make fields and methods private where possible
- Break down large methods into smaller, focused ones
- Delegate responsibilities to appropriate new classes

15**. ReaderStandardAnalyzer class exhibits Deficient Encapsulation**

Issue: The ReaderStandardAnalyzer class exposes internal state through public fields (DEFAULT_MAX_TOKEN_LENGTH and STOP_WORDS_SET) and lacks proper access control, violating encapsulation principles and potentially allowing unsafe modifications to internal state.

Refactoring Approach:

The ReaderStandardAnalyzer class exposes internal state through public fields (DEFAULT_MAX_TOKEN_LENGTH and STOP_WORDS_SET) and lacks proper access control, violating encapsulation principles and potentially allowing unsafe modifications to internal state.

16. **MimeType class exhibits Unnecessary Abstraction**

Issue: Just a Container for Constants:

The MimeType class appears to only hold static final String constants for MIME types

It doesn't provide any additional behavior or functionality

It doesn't encapsulate any complex logic or state

No Object-Oriented Benefits:

The class doesn't leverage any OO principles like inheritance, polymorphism, or encapsulation

There's no need for instantiation or instance-specific behavior

The constants could be placed directly where they're needed

Refactoring Approach:

Put all of the constants in reader-core/src/main/java/com/sismics/reader/core/dao/file/html/MimeTypeConstants.java directly

17. Cyclic-Dependent Modularization in AppContext

Issue 1: AppContext directly initializes and depends on FeedService, IndexingService, and ConfigDao. This creates cyclic dependencies, making it harder to test, extend, and maintain.

Issue 2: Single Responsibility Principle (SRP) violation

AppContext is responsible for both dependency management and event handling (via EventBus and its variations). The application context should only provide access to services, not handle event subscriptions.

Root Cause of Cyclic Dependencies

The cyclic dependency arises because AppContext tries to act as both a service locator and an application bootstrapper while also depending on lower-level database access

(ConfigDao). This violates the separation of concerns principle, as infrastructure-related components (database access) should not directly influence core application management.

Refactoring Approach

- Move ConfigDao initialization outside AppContext
    - Pass pre-configured service instances to AppContext rather than initializing them inside the constructor.
- Creating ServiceInitializer (Service Factory)
    - Moves service creation logic out of AppContext and ensures AppContext only receives ready-to-use services.
- Creating EventBusManager
    - Previously, AppContext was managing event handling, leading to unnecessary coupling.
    - Handles event publishing in a decoupled way.

18. Unnecessary Abstraction in UserCriteria

The UserCriteria class was empty and not used functionally.

Fix:

By replacing it with Object in UserDao and passing null where instances were created, we eliminate the unnecessary abstraction.

Task 3b

Code Metric 1: Cyclomatic complexity remains the same before and after.

Code Metric 2: Code Dupication

Before Refactoring

Code Metric	reader-web	reader-core	reader-web-common
Code Duplication Percentage	1.9%	4.6%	1.2%

After Refactoring

Code Metric	reader-web	reader-core	reader-web-common
Code Duplication Percentage	1.9%	3.5%	1.2%

As we solved a lot of issues from reader-core, the code duplication percentage reduced significantly in that.

Code Metric 3: Average LOC After:

Dataset: Full Project

The average loc per class in Full Project original code is: 51.2122641509434

The average loc per class in Full Project refactored code is: 49.38766519823788

Dataset: reader-core

The average loc per class in reader-core original code is: 46.550898203592816

The average loc per class in reader-core refactored code is: 44.285714285714285

Dataset: reader-web

The average loc per class in reader-web original code is: 86.75

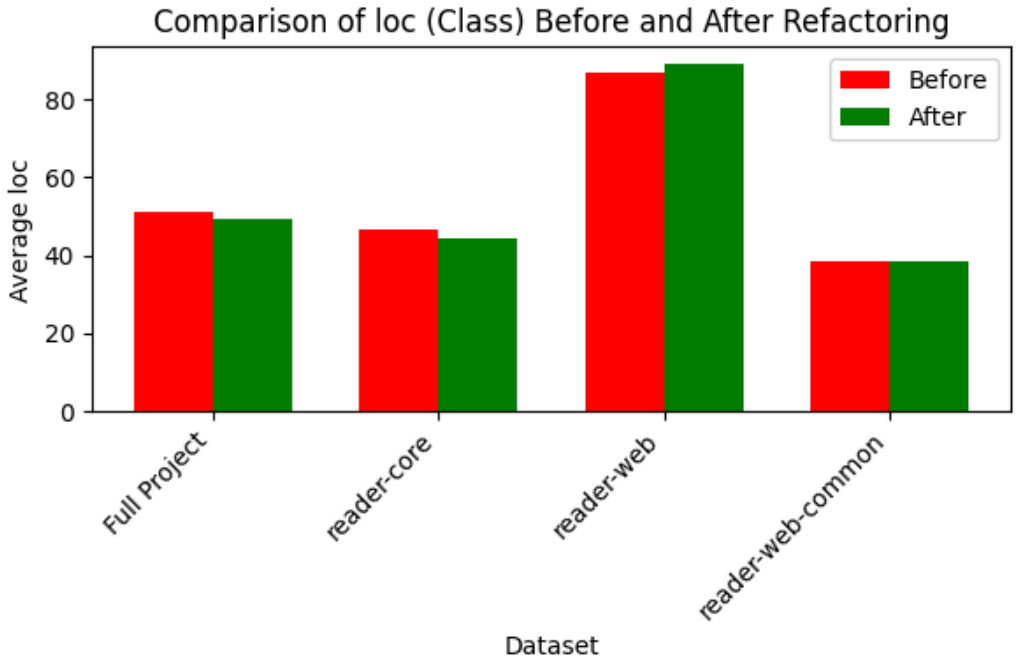
The average loc per class in reader-web refactored code is: 89.07142857142857

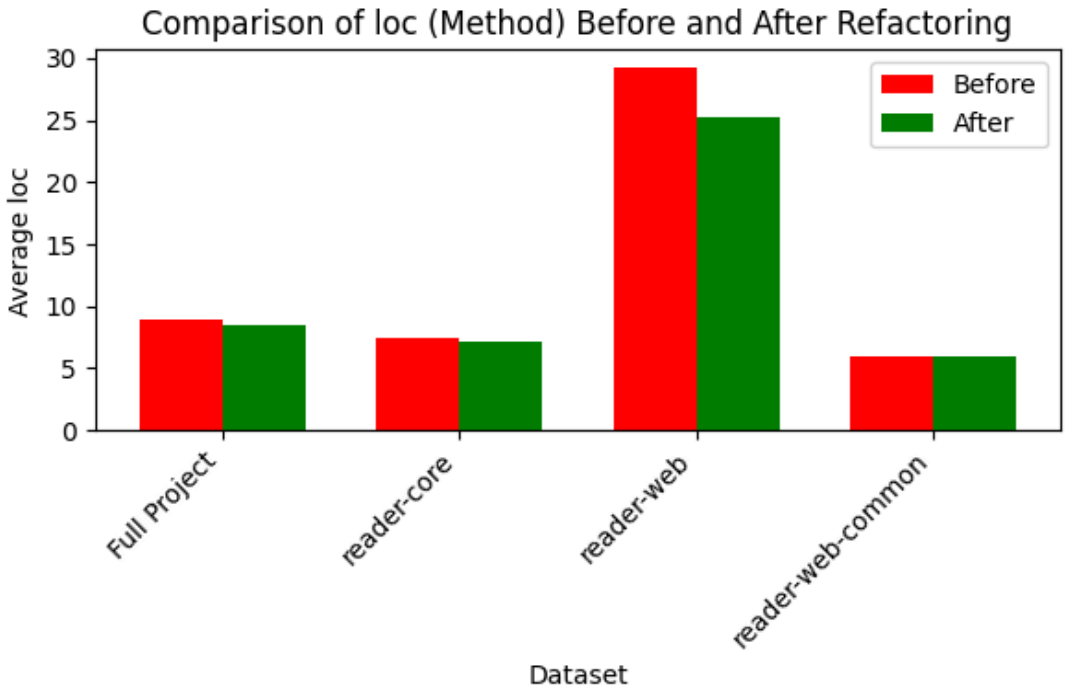
Dataset: reader-web-common

The average loc per class in reader-web-common original code is: 38.470588235294116

The average loc per class in reader-web-common refactored code is: 38.64705882352941

Plot:





Code Metric 4: Fan in (ck)

Dataset: Full Project

The average fanin per class in Full Project original code is: 2.8632075471698113

The average fanin per class in Full Project refactored code is: 2.894273127753304

The average fanin per method in Full Project original code is: 2.090990187332739

The average fanin per method in Full Project refactored code is: 2.0429752066115703

Dataset: reader-core

The average fanin per class in reader-core original code is: 2.3473053892215567

The average fanin per class in reader-core refactored code is: 2.368131868131868

The average fanin per method in reader-core original code is: 1.622741764080765

The average fanin per method in reader-core refactored code is: 1.5767716535433072

Dataset: reader-web

The average fanin per class in reader-web original code is: 0.7857142857142857

The average fanin per class in reader-web refactored code is: 0.7857142857142857

The average fanin per method in reader-web original code is: 0.8148148148148148

The average fanin per method in reader-web refactored code is: 1.2083333333333333

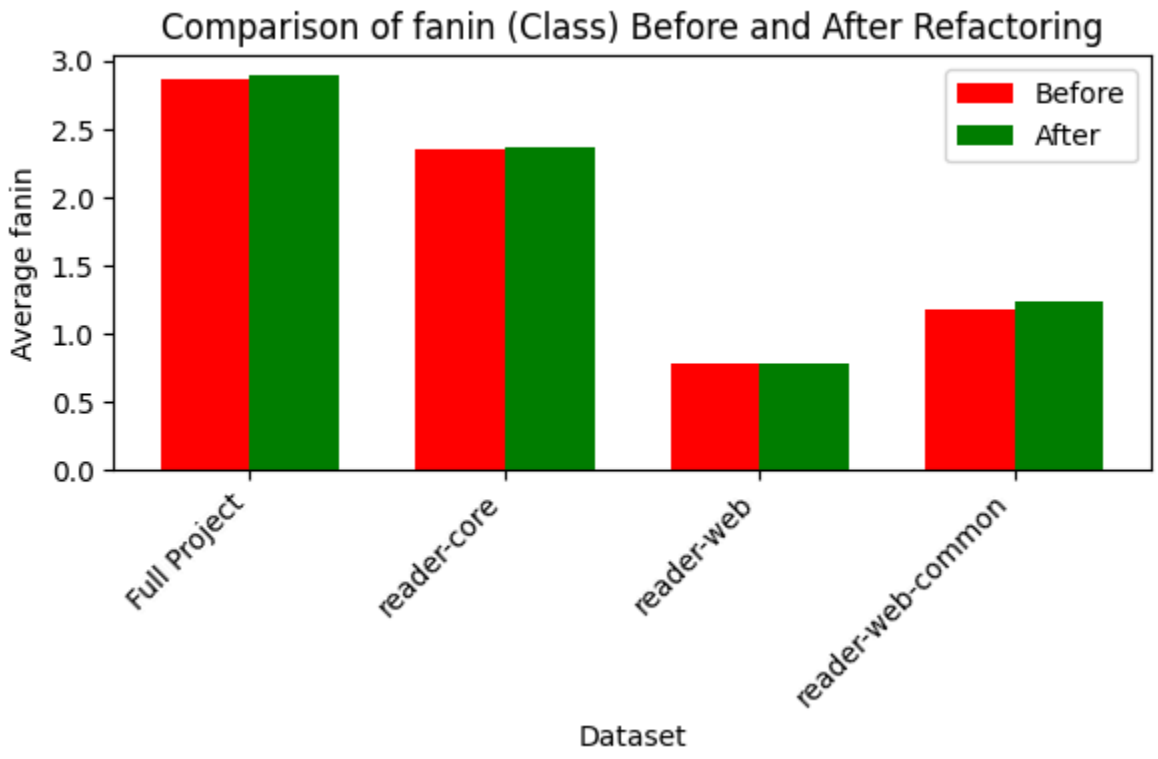
Dataset: reader-web-common

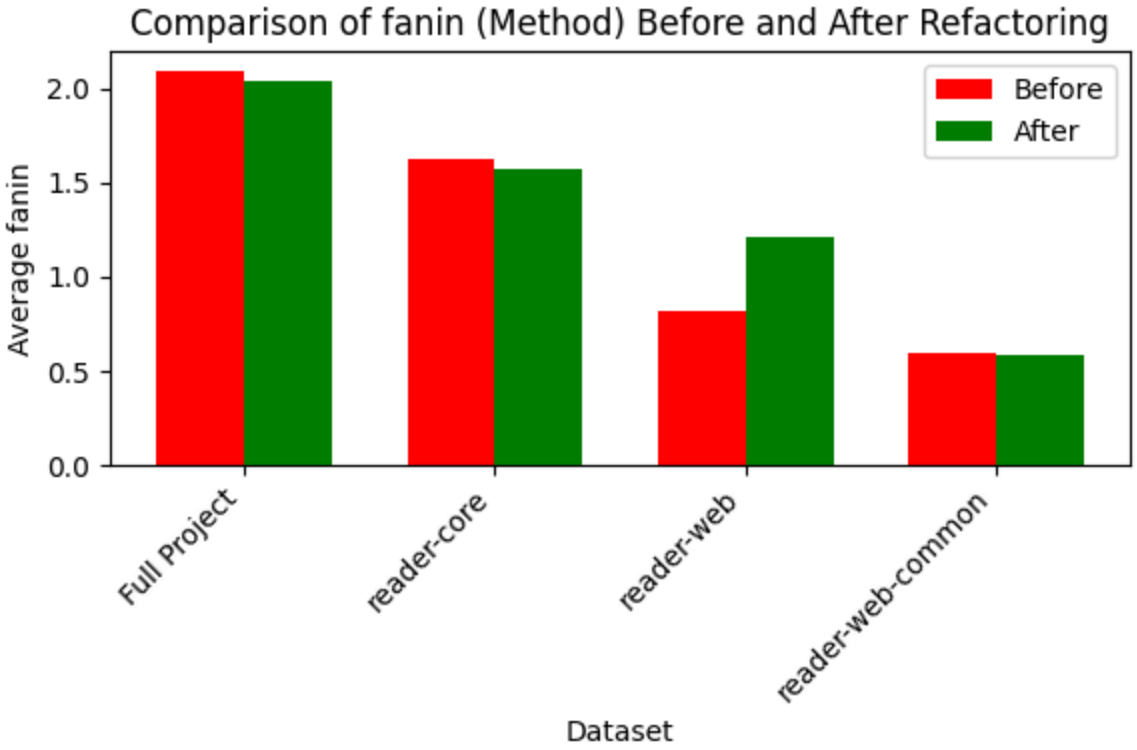
The average fanin per class in reader-web-common original code is: 1.1764705882352942

The average fanin per class in reader-web-common refactored code is: 1.2352941176470589

The average fanin per method in reader-web-common original code is: 0.5959595959595959

The average fanin per method in reader-web-common refactored code is: 0.5816326530612245





Code Metric 5: Fan out (ck)

Dataset: Full Project

The average fanout per class in Full Project original code is: 5.221698113207547

The average fanout per class in Full Project refactored code is: 5.26431718061674

The average fanout per method in Full Project original code is: 3.1507582515611063

The average fanout per method in Full Project refactored code is: 2.3578512396694213

Dataset: reader-core

The average fanout per class in reader-core original code is: 4.059880239520958

The average fanout per class in reader-core refactored code is: 3.9945054945054945

The average fanout per method in reader-core original code is: 1.842720510095643

The average fanout per method in reader-core refactored code is: 1.796259842519685

Dataset: reader-web

The average fanout per class in reader-web original code is: 10.357142857142858

The average fanout per class in reader-web refactored code is: 11.25

The average fanout per method in reader-web original code is: 0.9135802469135802

The average fanout per method in reader-web refactored code is: 1.2916666666666667

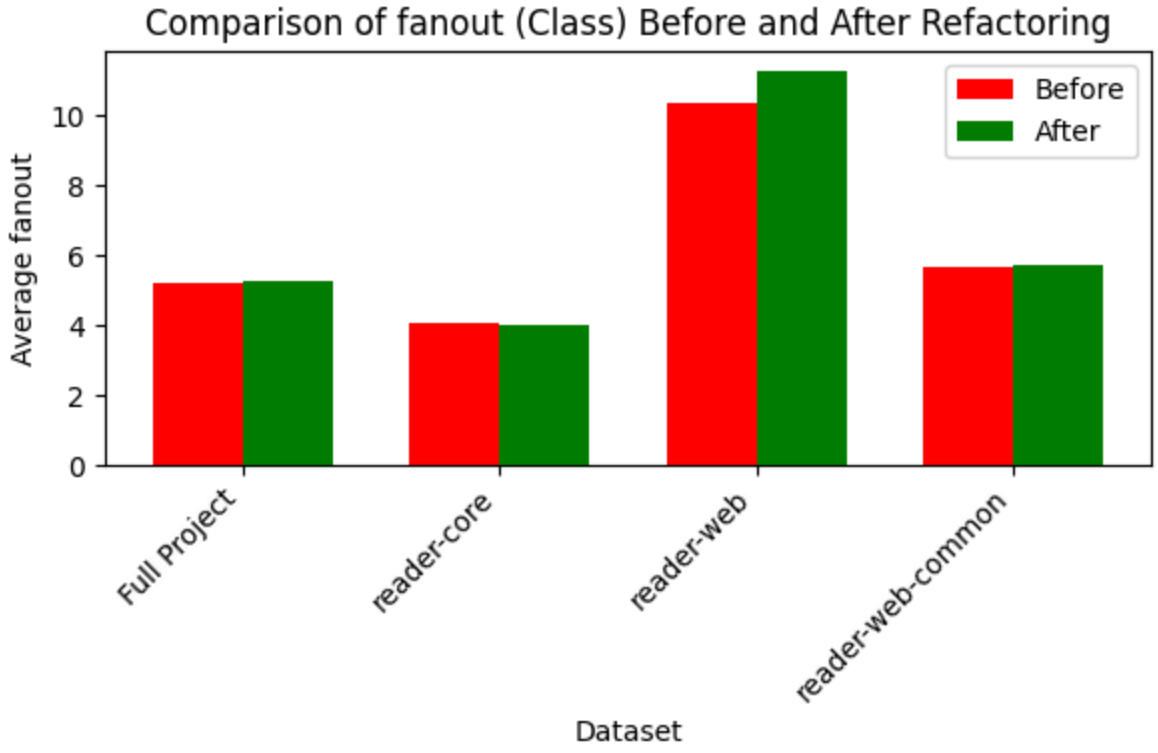
Dataset: reader-web-common

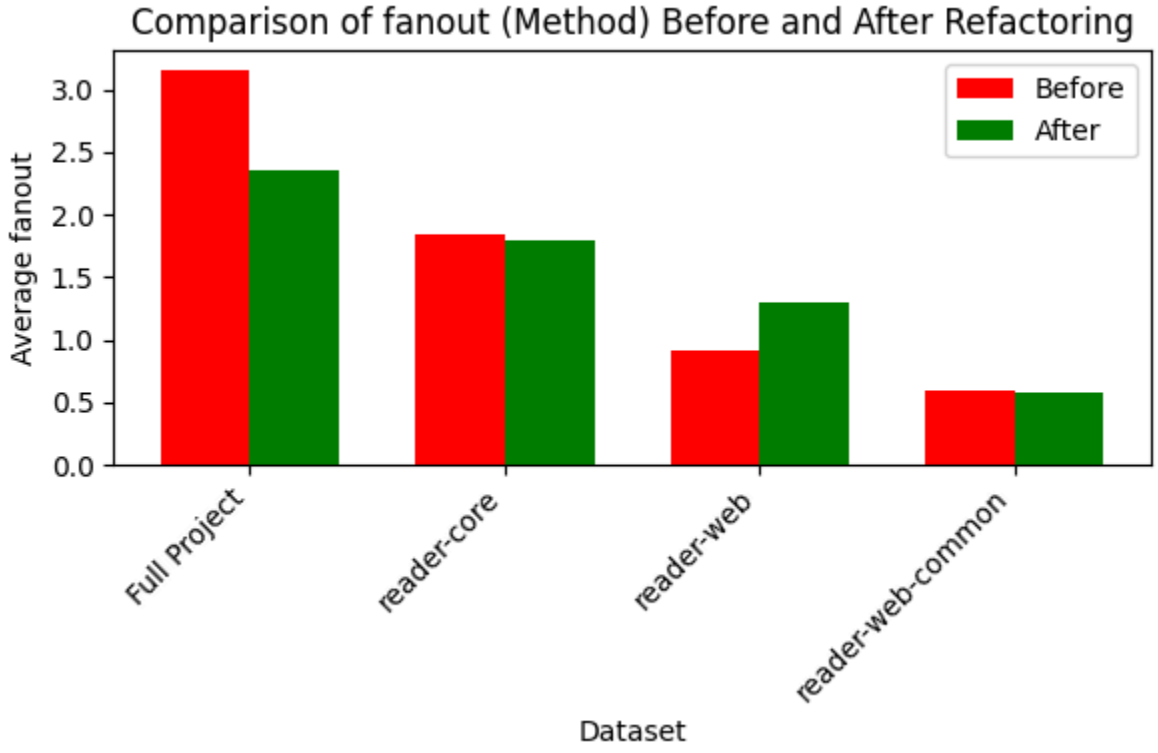
The average fanout per class in reader-web-common original code is: 5.647058823529412

The average fanout per class in reader-web-common refactored code is: 5.705882352941177

The average fanout per method in reader-web-common original code is: 0.5959595959595959

The average fanout per method in reader-web-common refactored code is: 0.5816326530612245


Comparison of fanout (Class) Before and After Refactoring
Before
After
Average fanout
0
2
4
6
8
10
Full Project
reader-core
reader-web
reader-web-common
Dataset


Comparison of fanout (Method) Before and After Refactoring
Before
After
Average fanout
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Full Project
reader-core
reader-web
reader-web-common
Dataset

Code Metric 6: DIT (ck)

Dataset: Full Project

The average dit per class in Full Project original code is: 1.3537735849056605

The average dit per class in Full Project refactored code is: 1.2422907488986785

Dataset: reader-core

The average dit per class in reader-core original code is: 1.1497005988023952

The average dit per class in reader-core refactored code is: 1.1373626373626373

Dataset: reader-web

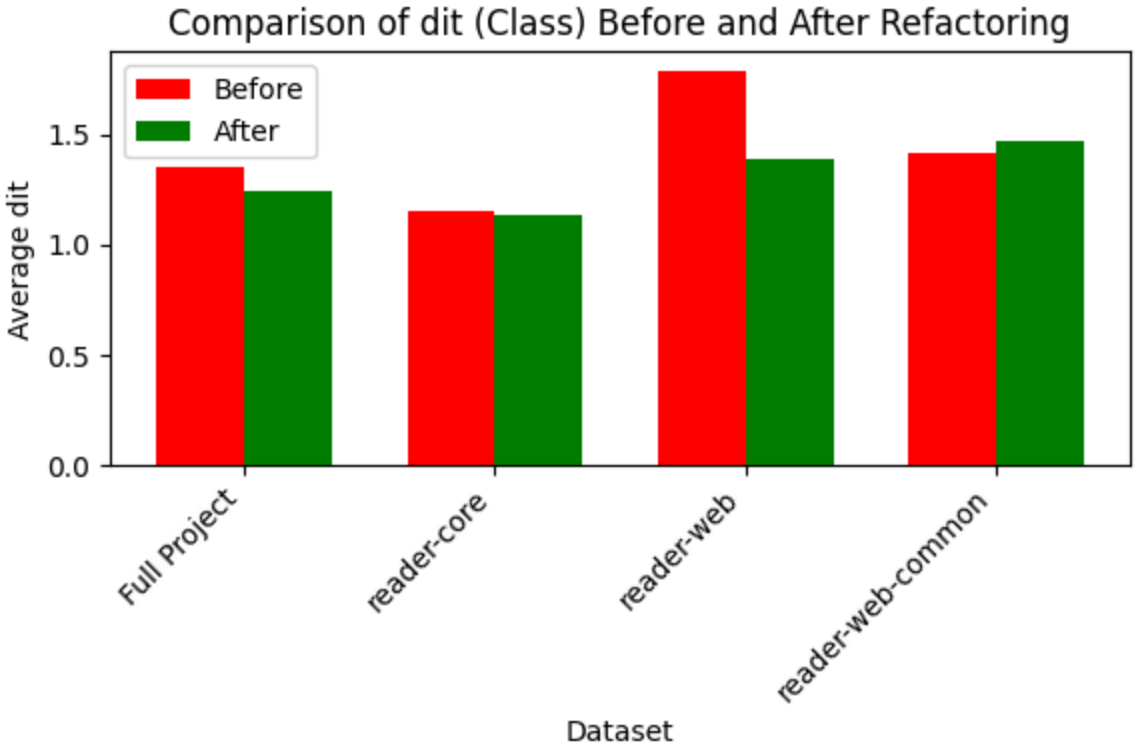
The average dit per class in reader-web original code is: 1.7857142857142858

The average dit per class in reader-web refactored code is: 1.3928571428571428

Dataset: reader-web-common

The average dit per class in reader-web-common original code is: 1.411764705882353

The average dit per class in reader-web-common refactored code is: 1.470588235294117



Code Metric 7: Number of Methods (ck)

Dataset: Full Project

The average totalMethodsQty per class in Full Project original code is: 5.278301886792453

The average totalMethodsQty per class in Full Project refactored code is: 5.3215859030837

Dataset: reader-core

The average totalMethodsQty per class in reader-core original code is: 5.622754491017964

The average totalMethodsQty per class in reader-core refactored code is: 5.571428571428571

Dataset: reader-web

The average totalMethodsQty per class in reader-web original code is: 2.892857142857143

The average totalMethodsQty per class in reader-web refactored code is: 3.4285714285714284

Dataset: reader-web-common

The average totalMethodsQty per class in reader-web-common original code is: 5.823529411764706

The average totalMethodsQty per class in reader-web-common refactored code is: 5.764705882352941

Code Metric 8: Cognitive Complexity (SonarQube)

The cognitive complexity of the code **BEFORE** refactoring: 2112

The cognitive complexity of the code **AFTER** refactoring: 2106

Cognitive complexity refers to the ease of understandability of the code

Code Metric 9: Variables Quantity

Dataset: Full Project

The average variablesQty per class in Full Project original code is: 8.820754716981131

The average variablesQty per class in Full Project refactored code is:
8.502202643171806

The average variablesQty per method in Full Project original code is:
1.2774308652988404

The average variablesQty per method in Full Project refactored code is:
1.2074380165289256

Dataset: reader-core

The average variablesQty per class in reader-core original code is:
8.059880239520957

The average variablesQty per class in reader-core refactored code is:
7.5989010989010985

The average variablesQty per method in reader-core original code is:
1.006376195536663

The average variablesQty per method in reader-core refactored code is:
0.9498031496062992

Dataset: reader-web

The average variablesQty per class in reader-web original code is:
15.714285714285714

The average variablesQty per class in reader-web refactored code is:
16.428571428571427

The average variablesQty per method in reader-web original code is:
5.345679012345679

The average variablesQty per method in reader-web refactored code is:
4.604166666666667

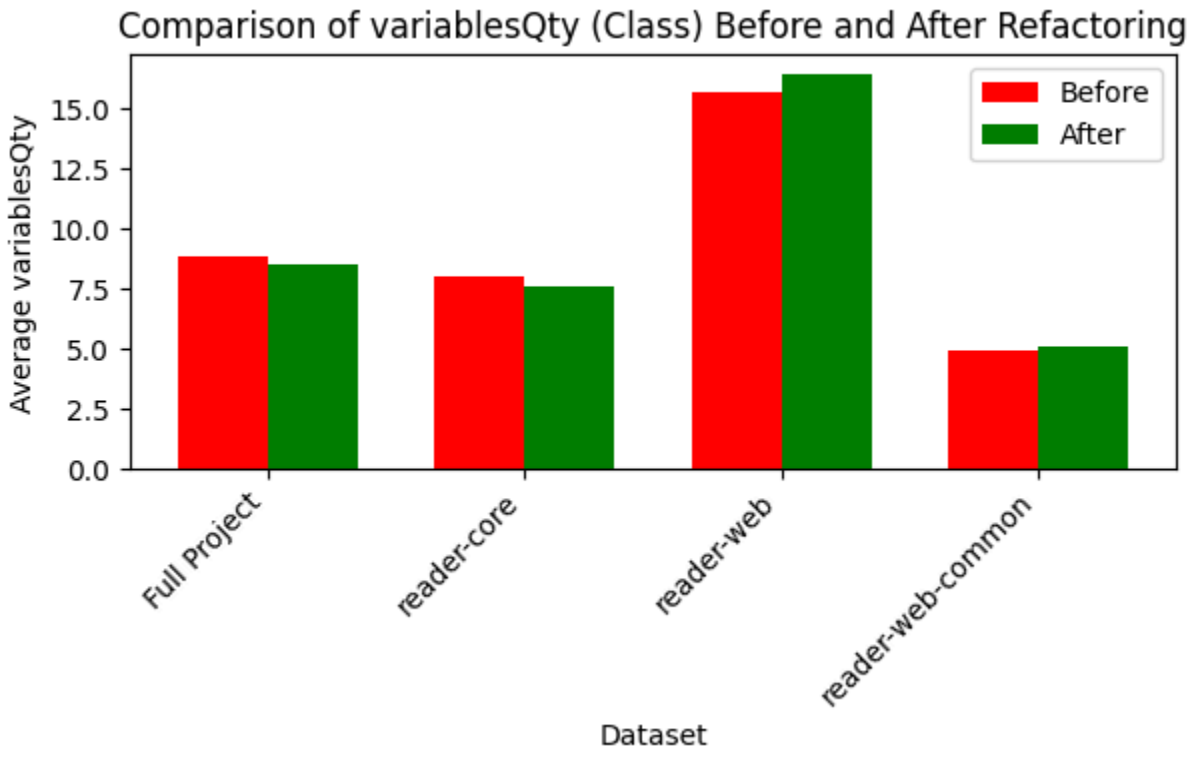
Dataset: reader-web-common

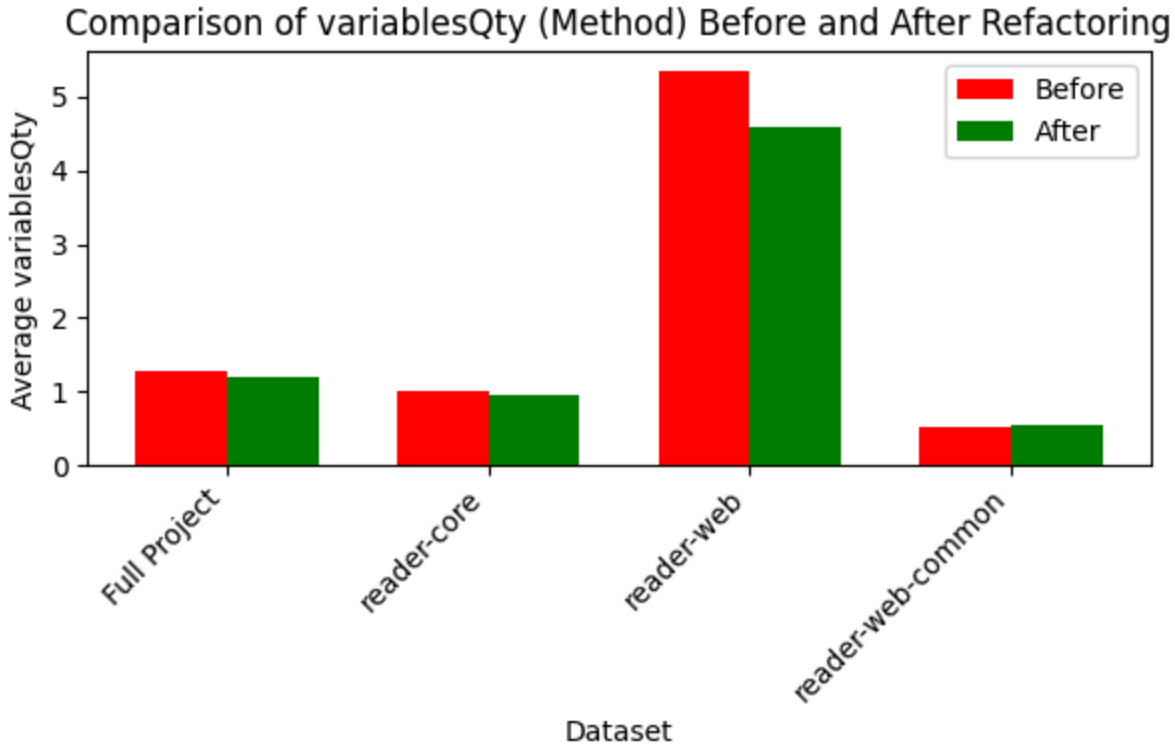
The average variablesQty per class in reader-web-common original code is: 4.9411764705882355

The average variablesQty per class in reader-web-common refactored code is: 5.117647058823529

The average variablesQty per method in reader-web-common original code is: 0.5252525252525253

The average variablesQty per method in reader-web-common refactored code is: 0.5510204081632653





Many more plots can be found in the Issue-branch ck-analysis/Plots directory

Task 3c

1. **Code snippets from `TokenBasedSecurityFilter.java` showing Deficient Encapsulation:**

   Issue 1: Public Constants Exposed (`COOKIE_NAME`, `TOKEN_LONG_LIFETIME`, `TOKEN_SESSION_LIFETIME`)

```
public static final String COOKIE_NAME = "auth_token";
public static final int TOKEN_LONG_LIFETIME = 3600 * 24 *
365 * 20;
public static final int TOKEN_SESSION_LIFETIME = 3600 * 24;
```

Issue 2: Direct Instantiation of DAO Objects in `authenticate` Method

```
AuthenticationTokenDao authTokenDao = new AuthenticationTokenDao();
AuthenticationToken authToken = authTokenDao.get(authTokenID);
```

Issue 3: Mutable Collection (HttpServletRequest.getCookies()) Without Defensive Copy

```
private static String extractAuthToken(Cookie[] cookies) {
   if (cookies != null)
      for (Cookie cookie : cookies)
         if (COOKIE_NAME.equals(cookie.getName()) &&
!cookie.getValue().isEmpty())
            return cookie.getValue();

   return null;
}
```

Refactored Versions:

1. **LLM**

Version 2 (LLM Version - Using Factory Method for Instance Creation)

java Copy Edit

```java
package com.sismics.util.filter;

import com.sismics.reader.core.dao.jpa.AuthenticationTokenDao;
import com.sismics.reader.core.dao.jpa.UserDao;
import com.sismics.reader.core.model.jpa.AuthenticationToken;
import com.sismics.reader.core.model.jpa.User;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;

public class TokenBasedSecurityFilter extends SecurityFilter {

    private static final String COOKIE_NAME = "auth_token";
    private static final int TOKEN_LONG_LIFETIME = 3600 * 24 * 365 * 20;
    private static final int TOKEN_SESSION_LIFETIME = 3600 * 24;

    private AuthenticationTokenDao authTokenDao;
    private UserDao userDao;

    private TokenBasedSecurityFilter() {}

    public static TokenBasedSecurityFilter createInstance() {
        TokenBasedSecurityFilter filter = new TokenBasedSecurityFilter();
        filter.authTokenDao = new AuthenticationTokenDao();
        filter.userDao = new UserDao();
        return filter;
    }

    private static String extractAuthToken(Cookie[] cookies) {
        if (cookies == null) return null;
        for (Cookie cookie : cookies) {
            if (COOKIE_NAME.equals(cookie.getName()) && !cookie.getValue().isEmpty())
                return cookie.getValue();
            }
        }
        return null;
    }

    @Override
    protected User authenticate(HttpServletRequest request) {
        String authTokenID = extractAuthToken(request.getCookies());
        if (authTokenID == null) return null;

        AuthenticationToken authToken = authTokenDao.get(authTokenID);
        if (authToken == null || isTokenExpired(authToken)) {
            handleExpiredToken(authTokenDao, authTokenID);
            return null;
        }

        authTokenDao.updateLastConnectionDate(authToken.getId());
        return userDao.getById(authToken.getUserId());
    }
}
```

Message ChatGPT

2. Manual Version

```
package com.sismics.util.filter;

import com.sismics.reader.core.dao.jpa.AuthenticationTokenDao;
import com.sismics.reader.core.dao.jpa.UserDao;
import com.sismics.reader.core.model.jpa.AuthenticationToken;
import com.sismics.reader.core.model.jpa.User;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;

public class TokenBasedSecurityFilter extends SecurityFilter {

    private static final String COOKIE_NAME = "auth_token";

    private final AuthenticationTokenDao authTokenDao;
    private final UserDao userDao;

    public TokenBasedSecurityFilter(AuthenticationTokenDao
authTokenDao, UserDao userDao) {
    this.authTokenDao = authTokenDao;
    this.userDao = userDao;
    }
```

```
        private String getAuthToken(HttpServletRequest request) {
        if (request.getCookies() == null) return null;
        for (Cookie cookie : request.getCookies()) {
        if (COOKIE_NAME.equals(cookie.getName()) &&
!cookie.getValue().isEmpty()) {
                return cookie.getValue();
        }
        }
        return null;
        }

        @Override
        protected User authenticate(HttpServletRequest request) {
        String authTokenID = getAuthToken(request);
        if (authTokenID == null) return null;

        AuthenticationToken authToken =
authTokenDao.get(authTokenID);
        if (authToken == null || authToken.isExpired()) {
        authTokenDao.delete(authTokenID);
        return null;
```

```
        }


        authToken.updateLastConnectionDate();

        return userDao.getById(authToken.getUserId());

        }

    }
```

	Manual	LLM
Code Clarity	Clear & Simple structure without any Static methods	More Abstractions in comparison
Maintainability	Easier to modify and test.	More modular with clear separation between concerns
Efficiency	Fewer method calls, directly operates on objects	More overhead due to abstraction layers.

Strengths and Weaknesses

- Manual Refactoring is more concise and still follows solid design principles. It improves encapsulation without adding too many abstractions.
- LLM Refactoring further enhances encapsulation and modularity but may introduce unnecessary complexity if the project is small or does not require such a high level of abstraction.

2. **Code snippets from `FeedService.java`, `IndexingService.java` showing Cyclic Dependent Modularisation**

`FeedService` posts events (`ArticleCreatedAsyncEvent`, `ArticleDeletedAsyncEvent`).

`IndexingService` listens to these events and updates the index.

`FeedService` later retrieves data from `ArticleDao`, which `IndexingService` modifies

```
// FeedService.java
ArticleCreatedAsyncEvent articleCreatedAsyncEvent = new
ArticleCreatedAsyncEvent();
articleCreatedAsyncEvent.setArticleList(Lists.newArrayList(articleMap.
values()));
AppContext.getInstance().getAsyncEventBus().post(articleCreatedAsy
ncEvent);

// IndexingService.java
@Subscribe public void onArticleEvent(ArticleCreatedAsyncEvent
event) { for (Article article : event.getArticleList()) { index(article); } }

//ArticleDao
public void synchronizeAllFeeds() { List<Article> articles =
fetchNewArticles(feed.getRssUrl()); for (Article article : articles) { new
ArticleDao().create(article); // Modifies the same database } }
```

Refactored Versions:

1. **LLM**

This version **removes direct dependencies** by letting `IndexingService` listen to events **without knowing** `FeedService` **exists**.

Refactored `FeedService.java`

java Copy Edit

```java
public class FeedService {
    private final EventBus eventBus = AppContext.getInstance().getAsyncEventBus();

    public void synchronizeAllFeeds() {
        List<Article> articles = fetchNewArticles(feed.getRssUrl());
        for (Article article : articles) {
            eventBus.post(new ArticleEvent("CREATE", article));
        }
    }
}
```

New `ArticleEvent.java` **(Event Definition)**

java Copy Edit

```java
public class ArticleEvent {
    private final String action;
    private final Article article;

    public ArticleEvent(String action, Article article) {
        this.action = action;
        this.article = article;
    }

    public String getAction() { return action; }
    public Article getArticle() { return article; }
}
```

Refactored `IndexingService.java`

java Copy Edit

```java
public class IndexingService {
    @Subscribe
    public void onArticleEvent(ArticleEvent event) {
        if ("CREATE".equals(event.getAction())) {
            index(event.getArticle());
        } else if ("DELETE".equals(event.getAction())) {
            removeFromIndex(event.getArticle().getId());
        }
    }
}
```

Message ChatGPT

2. Manual

Eliminated cyclic dependency since both services only interact with `ArticleManager`.

```java
public class ArticleManager {
    private final ArticleDao articleDao = new ArticleDao();

    public void addArticle(Article article) {
        articleDao.create(article);
    }

    public void deleteArticle(String articleId) {
        articleDao.delete(articleId);
    }

    public List<Article> getArticlesForIndexing() {
        return articleDao.findAll();
    }
}


public class FeedService {
    private final ArticleManager articleManager = new ArticleManager();

    public void synchronizeAllFeeds() {
        List<Article> articles = fetchNewArticles(feed.getRssUrl());
        for (Article article : articles) {
            articleManager.addArticle(article); // Delegates to ArticleManager
        }
    }
}

public class IndexingService {
    private final ArticleManager articleManager = new ArticleManager();

    public void indexArticles() {
        List<Article> articles = articleManager.getArticlesForIndexing();
        for (Article article : articles) {
            index(article);
        }
    }
}
```

	Manual	LLM
Code Clarity	Easier due to Direct Method calls	Event Handling
Coupling	Loosely Coupled (interacts via ArticleManager)	Fully decoupled
Efficiency	Synchronous, blocking operations.	Asynchronous, better for scalability.

3. Code snippets from `Article.java` showing Insufficient Modularisation:

Manual Version :

Key Changes :

1. Introduction of `@Embeddable` Classes (`Comment` and `ImportantTimestamps`):
    - The fields `commentUrl`, `commentCount`, `publicationDate`, `createDate`, and `deleteDate` were moved into two new embeddable classes: `Comment` and `ImportantTimestamps`.
    - Why? This change improves modularity and reusability. By grouping related fields into separate classes, the code becomes more organized and adheres to the Single Responsibility Principle (SRP). These embeddable classes can now be reused in other entities if needed.
2. Removal of Direct Fields in `Article`:
    - Fields like `commentUrl`, `commentCount`, `publicationDate`, `createDate`, and `deleteDate` were removed from the `Article` class and replaced with instances of `Comment` and `ImportantTimestamps`.
    - Why? This reduces the complexity of the `Article` class by delegating responsibility for managing related data to the `Comment` and `ImportantTimestamps` classes. It also makes the `Article` class more focused on its core responsibilities.
3. Constructor Changes:
    - A new constructor was added to the `Article` class that accepts `Comment` and `ImportantTimestamps` objects.
    - Why? This allows for easier initialization of the `Article` class with its embedded objects, promoting better encapsulation and reducing the risk of inconsistent state.
4. Minor Formatting and Documentation Adjustments:

- Some Javadoc comments were adjusted or removed, and the code was formatted for better readability.
- Why? This improves readability and ensures that the code is easier to understand and maintain.

```java
@Embeddable
class Comment {
    @Column(name = "ART_COMMENTURL_C", length = 2000)
    private String commentUrl;

    @Column(name = "ART_COMMENTCOUNT_N")
    private Integer commentCount;

    public Comment() {}

    public Comment(String commentUrl, Integer commentCount) {
        this.commentUrl = commentUrl;
        this.commentCount = commentCount;
    }
}

@Embeddable
class ImportantTimestamps {
    @Column(name = "ART_PUBLICATIONDATE_D", nullable = false)
    private Date publicationDate;

    @Column(name = "ART_CREATEDATE_D", nullable = false)
    private Date createDate;

    @Column(name = "ART_DELETEDATE_D")
    private Date deleteDate;

    public ImportantTimestamps() {}

    public ImportantTimestamps(Date publicationDate, Date createDate, Date deleteDate) {
        this.publicationDate = publicationDate;
        this.createDate = createDate;
        this.deleteDate = deleteDate;
    }
}
```

LLM Version :

Changes Made in the LLM-Generated Refactored Code

1. Introduction of Multiple `@Embeddable` Classes:
    - The LLM introduced five embeddable classes: `ArticleMetadata`, `ContentDetails`, `CommentInfo`, `Enclosure`, and `Timestamps`.

    - **This change improves modularity and separation of concerns by grouping related fields into separate classes. Each class now has a clear and focused responsibility.**
2. **Removal of Direct Fields in `Article`:**
    - **All fields in the `Article` class were removed and replaced with instances of the embeddable classes.**
    - **This reduces the complexity of the `Article` class and delegates responsibility for managing related data to the embeddable classes.**
3. **Constructor Changes:**
    - **A new constructor was added to the `Article` class that accepts instances of all embeddable classes.**
    - **This allows for easier initialization of the `Article` class with all necessary data in a single step, promoting better encapsulation.**
4. **Simplified `toString()` Method:**
    - **The `toString()` method now references fields from the `ArticleMetadata` class (`metadata.getId()` and `metadata.getUrl()`).**
    - **This simplifies the `toString()` method and ensures it only includes the most relevant information.**
5. **Encapsulation of Fields:**
    - **Fields in the embeddable classes are encapsulated, with only getters provided for necessary fields (e.g., `getId()` and `getUrl()` in `ArticleMetadata`).**
    - **This promotes encapsulation and ensures that fields are accessed in a controlled manner.**

```java
@Entity
@Table(name = "T_ARTICLE")
public class Article {
    @Embedded
    private ArticleMetadata metadata;

    @Embedded
    private ContentDetails contentDetails;

    @Embedded
    private CommentInfo commentInfo;

    @Embedded
    private Enclosure enclosure;

    @Embedded
    private Timestamps timestamps;

    public Article() {}

    public Article(ArticleMetadata metadata, ContentDetails contentDetails, CommentInfo commentInfo, Enclosure enclosure, Timestamps times
        this.metadata = metadata;
        this.contentDetails = contentDetails;
        this.commentInfo = commentInfo;
        this.enclosure = enclosure;
        this.timestamps = timestamps;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("id", metadata.getId())
                .add("url", metadata.getUrl())
                .toString();
    }
}

@Embeddable
class ArticleMetadata {
    @Id
    @Column(name = "ART_ID_C", length = 36)
```

```
class ArticleMetadata {
    private String id;

    @Column(name = "ART_IDFEED_C", nullable = false, length = 36)
    private String feedId;

    @Column(name = "ART_URL_C", length = 2000)
    private String url;

    @Column(name = "ART_BASEURI_C", length = 2000)
    private String baseUri;

    @Column(name = "ART_GUID_C", nullable = false, length = 2000)
    private String guid;

    public ArticleMetadata() {}

    public String getId() { return id; }
    public String getUrl() { return url; }
}

@Embeddable
class ContentDetails {
    @Column(name = "ART_TITLE_C", length = 4000)
    private String title;

    @Column(name = "ART_CREATOR_C", length = 200)
    private String creator;

    @Lob
    @Column(name = "ART_DESCRIPTION_C")
    private String description;

    public ContentDetails() {}

    public ContentDetails(String title, String creator, String description) {
        this.title = title;
        this.creator = creator;
        this.description = description;
    }
}
```

```java
@Embeddable
class CommentInfo {
    @Column(name = "ART_COMMENTURL_C", length = 2000)
    private String commentUrl;

    @Column(name = "ART_COMMENTCOUNT_N")
    private Integer commentCount;

    public CommentInfo() {}

    public CommentInfo(String commentUrl, Integer commentCount) {
        this.commentUrl = commentUrl;
        this.commentCount = commentCount;
    }
}

@Embeddable
class Enclosure {
    @Column(name = "ART_ENCLOSUREURL_C", length = 2000)
    private String enclosureUrl;

    @Column(name = "ART_ENCLOSURELENGTH_N")
    private Integer enclosureLength;

    @Column(name = "ART_ENCLOSURETYPE_C", length = 2000)
    private String enclosureType;

    public Enclosure() {}

    public Enclosure(String enclosureUrl, Integer enclosureLength, String enclosureType) {
        this.enclosureUrl = enclosureUrl;
        this.enclosureLength = enclosureLength;
        this.enclosureType = enclosureType;
    }
}

@Embeddable
class Timestamps {
    @Column(name = "ART_PUBLICATIONDATE_D", nullable = false)
    private Date publicationDate;
```

```java
class Timestamps {

    @Column(name = "ART_CREATEDATE_D", nullable = false)
    private Date createDate;

    @Column(name = "ART_DELETEDATE_D")
    private Date deleteDate;

    public Timestamps() {}

    public Timestamps(Date publicationDate, Date createDate, Date deleteDate) {
        this.publicationDate = publicationDate;
        this.createDate = createDate;
        this.deleteDate = deleteDate;
    }
}
```

1. Comparison of LLM-Based Refactoring vs. Manual Refactoring :

Similarities

1. **Use of `@Embeddable` Classes:**
    - **Both the manual and LLM refactoring introduced embeddable classes to group related fields.**
    - **Why? This improves modularity and adheres to the Single Responsibility Principle (SRP).**
2. **Removal of Direct Fields from `Article`:**
    - **Both refactorings removed direct fields from the `Article` class and replaced them with instances of embeddable classes.**
    - **Why? This reduces the complexity of the `Article` class and improves readability.**
3. **Constructor Changes:**
    - **Both refactorings added constructors to initialize the `Article` class with embeddable objects.**
    - **Why? This promotes better encapsulation and ensures consistent initialization.**

Differences

1. **Number of Embeddable Classes:**
    - **The manual refactoring introduced 2 embeddable classes (`Comment` and `ImportantTimestamps`).**
    - **The LLM refactoring introduced 5 embeddable classes (`ArticleMetadata`, `ContentDetails`, `CommentInfo`, `Enclosure`, and `Timestamps`).**
    - **Why? The LLM took a more granular approach, further separating concerns. However, this might introduce additional complexity.**
2. **Field Grouping:**
    - **The manual refactoring grouped fields based on their logical relationships (e.g., `commentUrl` and `commentCount` in `Comment`).**
    - **The LLM refactoring grouped fields more granularly (e.g., separating `ArticleMetadata` from `ContentDetails`).**
    - **Why? The LLM's approach is more modular but may be overkill for smaller applications.**

3. **Bug in `toString()` Method:**
    - **The manual refactoring introduced a bug in the `toString()` method (referencing non-existent `metadata` fields).**
    - **The LLM refactoring fixed this by correctly referencing `metadata.getId()` and `metadata.getUrl()`.**
    - **Why? The LLM likely identified and corrected the bug during refactoring.**

2. Strengths of the Manual Refactoring

1. **Pragmatic Approach:**
    - **The manual refactoring strikes a balance between modularity and simplicity, making it easier to understand and maintain.**
2. **Logical Grouping:**
    - **Fields are grouped based on their logical relationships, which aligns well with the application's context.**
3. **Readability:**
    - **The code is clean and easy to navigate, with fewer classes to manage.**

Weaknesses of the Manual Refactoring

1. **Limited Modularity:**
    - **The manual refactoring is less granular than the LLM's approach, which might limit reusability in some scenarios.**

3. Strengths of the LLM Refactoring

1. **Granular Modularity:**
    - **The LLM's approach of creating multiple embeddable classes ensures a high degree of separation of concerns.**
2. **Bug Fixes:**
    - **The LLM correctly implemented the `toString()` method, avoiding the bug present in the manual refactoring.**
3. **Consistency:**
    - **The LLM applied a consistent pattern across all embeddable classes, ensuring uniformity.**

Weaknesses of the LLM Refactoring

1. **Over-Engineering:**
    - **The introduction of 5 embeddable classes might be excessive for smaller applications, adding unnecessary complexity.**
2. **Lack of Context Awareness:**
    - **The LLM might not fully understand the application's context, leading to overly granular groupings.**
3. **Limited Documentation:**
    - **The LLM did not add Javadoc comments for the new classes, which could reduce readability for other developers.**

Task 3d

Automated Design Smell Detection and Refactoring Pipeline

Overview

This pipeline automates the detection and refactoring of design smells in Java projects hosted on GitHub. It leverages static code analysis tools, Gemini based refactoring, and Git automation to enhance software maintainability.

Tools Used

- **PMD – Static code analysis for detecting design smells.**
- **Checkstyle – Enforces coding standards and detects potential design flaws.**
- **Gemini AI – Performs intelligent refactoring based on detected issues.**
- **GitHub API – Automates pull request creation with refactored code.**
- **LangGraph – Manages AI workflow and refactoring context.**

---

1. Design Smell Detection

Process:

1. **The pipeline periodically scans a given GitHub repository for Java source files.**
2. **It applies PMD and Checkstyle to analyze code quality.**

3. **The tools detect various design smells, such as:**
    - **Feature Envy – A class that manipulates another class excessively.**
    - **God Class – A class that centralizes too many responsibilities.**
    - **Data Clump – Repeated groups of variables passed together frequently.**
4. **Detected issues are extracted and formatted for further processing.**

---

2. Refactoring Process

Steps:

1. **The detected design smells are passed to Gemini for refactoring.**
2. **The LLM is prompted with:**
    - **The affected code.**
    - **A second related file (to preserve inter-file logic).**
    - **Explicit instructions to maintain dependencies and function calls.**
3. **Gemini model modifies the code while ensuring structural integrity.**
4. **The response is parsed into a valid refactored code snippet.**

Example Prompt:

> **You are an expert refactoring assistant.**
>
> **Refactor the following Java code to remove design smells while maintaining logic between files.**
>
> **Issues Detected:**
>
> **-** Feature Envy: OrderManager manipulates NotificationService too much.
>
> **Rules:**
>
> - Do not break function calls across files.
>
> - Maintain dependencies.
>
> - Improve modularity.

---

3. Context Handling for Refactoring

Why Context Matters?

- **Many design smells span across multiple files.**
- **LLM needs access to dependent files to ensure safe modifications.**
- **The pipeline selects an additional relevant file for context.**

Context Selection Logic:

1. **Identify the primary file with detected smells.**
2. **Locate another related file from the repository (e.g., a file with method calls or dependencies).**
3. **Provide both files to Gemini to prevent breaking dependencies.**

---

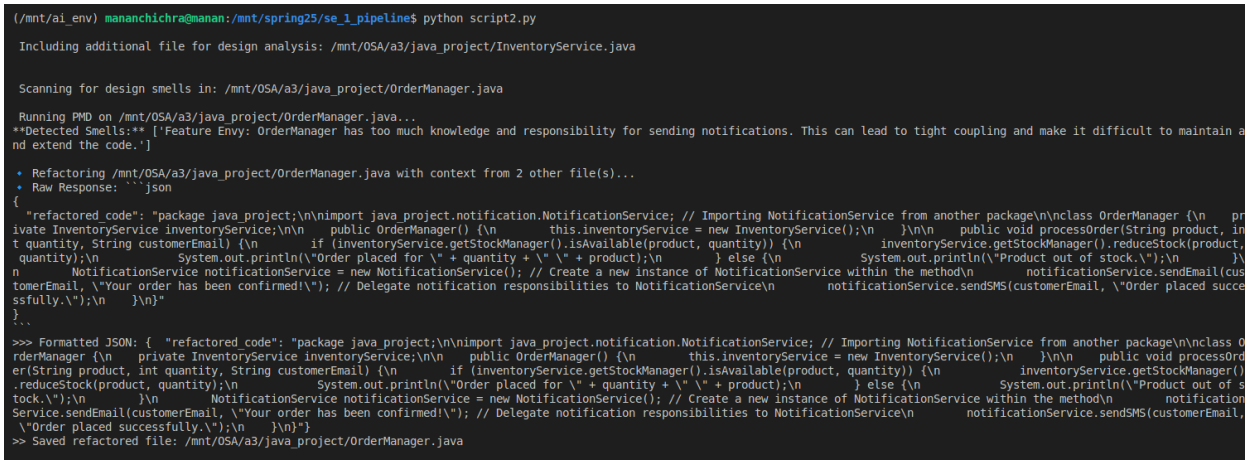
4. GitHub Pull Request Automation

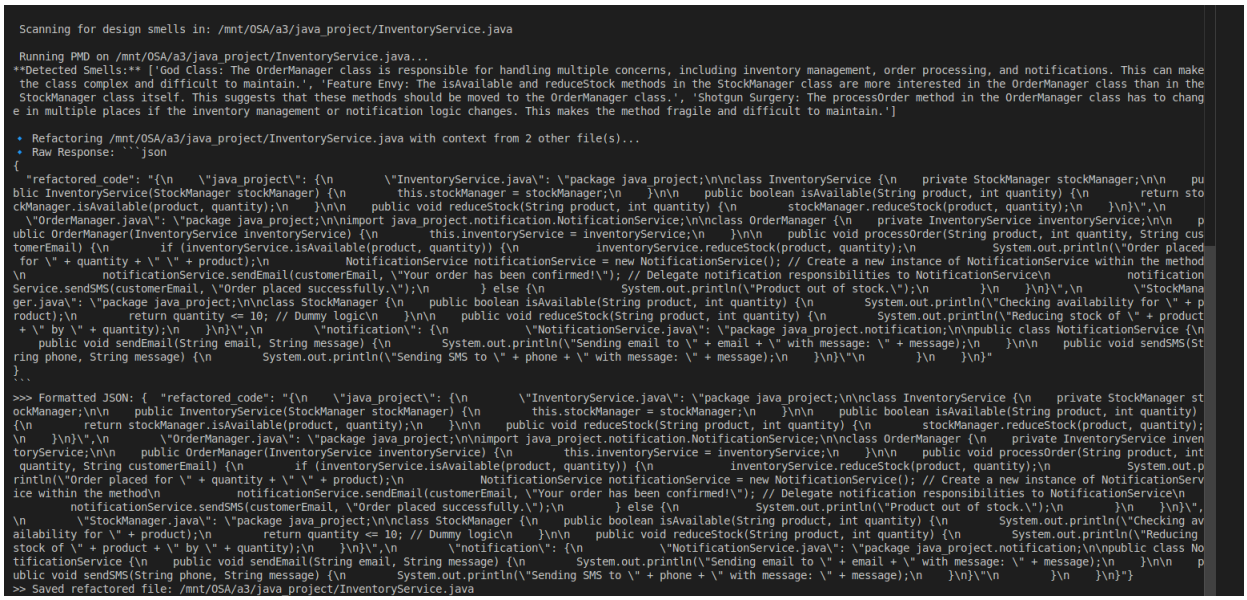
Steps:

1. **The pipeline commits the refactored code to a new Git branch.**
2. **A Pull Request (PR) is created automatically.**
3. **The PR includes:**
    - **A summary of detected issues.**
    - **Details of the refactored changes.**
    - **A request for code review before merging.**
4. **Developers can review and merge the improvements.**

---

5. Testing





Contributions

1. **Masumi Desai (2022102057):**
    a. **Task 1: Class Diagram for User Management Subsystem**
        - Complete class diagram using PlantUML
        - Detailed documentation

   b. **Task 2b: Code metrics Analysis using lizard**
      - Mainly did Cyclomatic complexity
      - Documented few of the other smells from ck report
   c. **Task 3b: Code metric analysis for cyclomatic complexity**
   d. **Task 3a: Refactoring design smells**
      - Issues #4, #6, #7, #15, #17
2. **Manan Chichra (2022102058):**
   a. **Task 3c: Leveraging LLMs for refactoring:**
      - Refactored two code snippets corresponding to insufficient modularisation manually and by using LLMs.
        1. Deficient encapsulation
        2. Cyclic modular modularisation
   b. **Task 3d: Automating Refactoring Pipeline**
      - **Automating the following processes:**
        1. Design Smell Detection
        2. Refactoring
        3. Pull request
3. **Samanvitha (2022102027):**
   a. **Task 1: Class Diagram for Feed Organisation Subsystem**
      - Class Diagram using PlantUML
      - Detailed Documentation

   b. **Task 3c: Leveraging Large Language Models for Refactoring**

      - Refactored two code snippets corresponding to Insufficient Modularisation manually and by using LLMs.
      - Detailed Documentation

4. **Shravan Gadbail (2022102025):**
   a. **Task 2b: Code metrics Analysis using ck**
      - Analysis of the full project directory, and individual directories reader-web, reader-core and reader-web-common using ‘ck’.
   b. **Task 3a: Refactoring design smells**
      - Issues #2, #9, #10, #13, #19, #20
   c. **Task 3b: Code metrics Analysis and Comparison**

- Comparison of these code metrics before and after refactoring using python code.
- Bar graph plots for comparing 6 code metrics' average value before and after refactoring.

5. **Swaroop C (2022101114):**

a.Task 1: Class Diagram for Subscription and Content Subsystem

- Complete class diagram using PlantUML
- Detailed documentation

B. Task 2a: Analysed the codebase using Sonarqube and DesigniteJava to identify code smells and design smells. Made a suitable documentation too.

C. Task 3a:Refactoring design smells

Issues #8, #11, #12, #14, #16, #18