SE - Project

Project-1: Sismics Music Server

Team-19

Abhishek Reddy Gaddam - 2022201025

Srikant Konduri - 2022201017

Ravada Sai Venkatesh - 2022201072

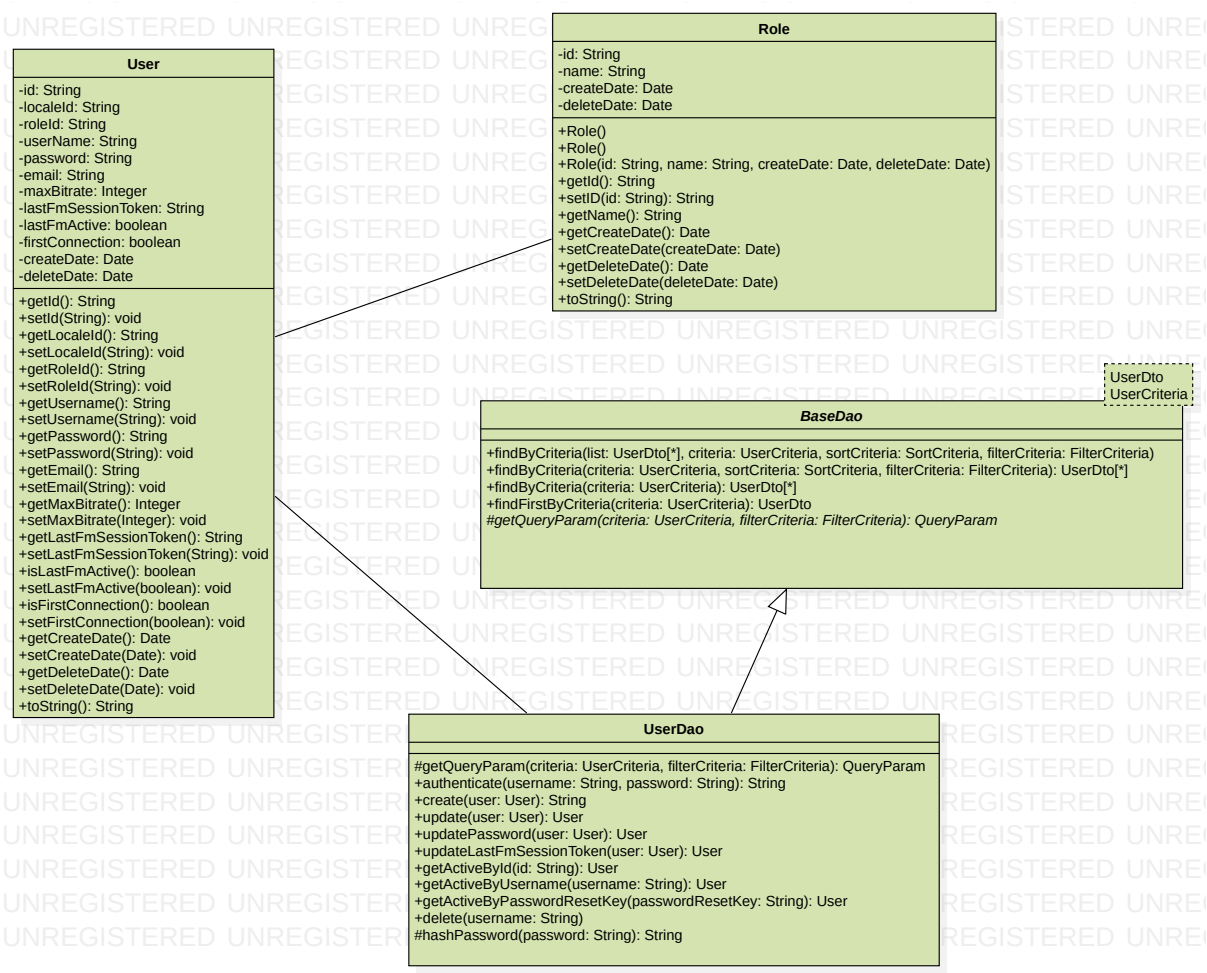
Venkata Sriram D - 2022201015

Kambampati Barath - 2022202025

Notion Document

Task-1: Mining Repository

User Management



Overview

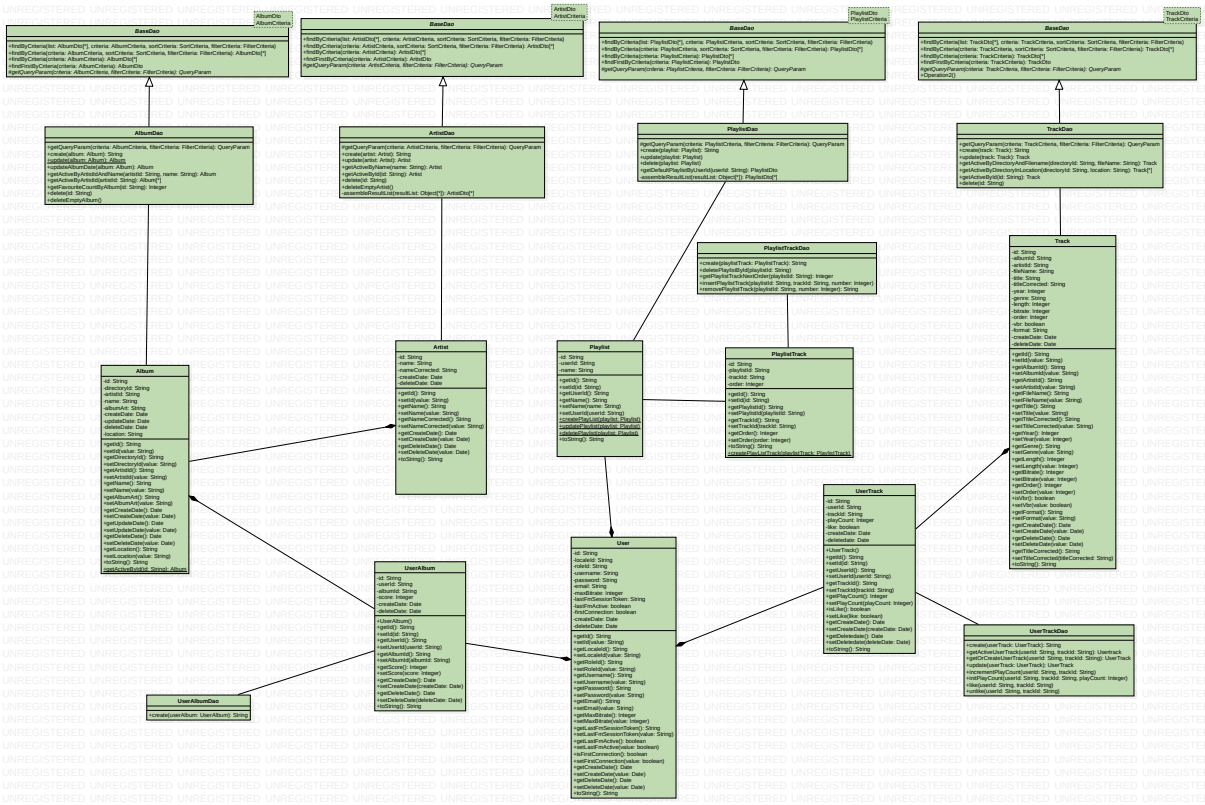
- User management system is responsible for login/logout of users.
- This feature is accomplished by several classes such as UserResource, UserDao, UserDto, UserDtoMapper.
- UserDao(inherited from BaseDao) contains all the CRUD methods for the User Model. UserResource contains all the endpoints from the web-application which invokes the methods present in UserDao.

Class Responsibilities

Class	Responsibility
UserDao	Provides an interface to interact with the underlying user data store
UserDto	Encapsulates user data in a format that can be easily transferred over the network

Class	Responsibility
UserDtoMapper	Maps user data between the `User` and `UserDto` classes
User	Represents the model of the user with attributes, getters and setters.
UserResource	Handles `HTTP` requests and responses related to user data. It contains methods for handling `REST API` endpoints.

Library Management



Overview

- The main business logic concerned with uploading music, organizing music into playlists and albums, editing metadata and album art information.

Relationships Between Entities Involved

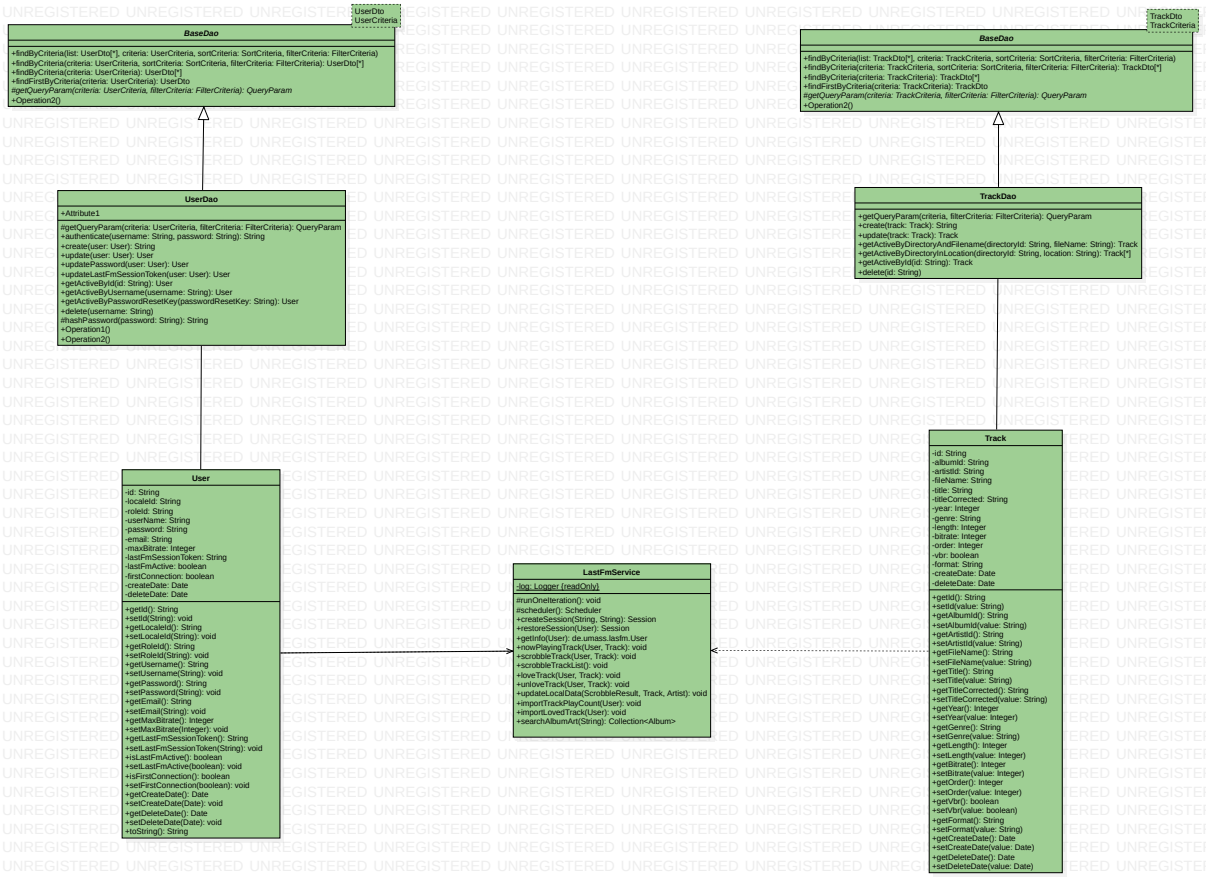
- Album and Artist Relationship
  - `Album` cannot exist without an `Artist`
  - **Observation**: Each `Album` is composed with only one `Artist`
- Album and UserAlbum Relationship
  - `UserAlbum` cannot exist without an `Album`
  - **Observation**: Each `UserAlbum` is associated with `Album` indicating `User` specific information regarding to this `Album`
- User and UserAlbum Relationship
  - `UserAlbum` cannot exist without a `User`
  - **Observation**: `UserAlbum` has a foreign key reference to `User` object
- Playlist and User Relationship
  - `Playlist` cannot exist without a `User`
  - Observation: A `Playlist` is created by `User`
- User and UserTrack Relationship
  - `UserTrack` does not exist without a `User`
  - Observation: `UserTrack` has all the track-metrics related to a particular `User`
- UserTrack and Track Relationship
  - `UserTrack` cannot exist without a `Track`
  - Observation: `UserTrack` has a trackId and userId

Class Responsibilities

Class	Responsibilities
AlbumDao	Has methods for creating, reading, updating, and deleting album records in the data store
ArtistDao	Has methods for creating, reading, updating, and deleting artist records in the data store

Class	Responsibilities
PlaylistDao	Has methods for creating, reading, updating, and deleting playlist records in the data store
TrackDao	Has methods for creating, reading, updating, and deleting track records in the data store
AlbumDto and AlbumDtoMapper	`AlbumDto` encapsulates `Album` for transfer over network. `AlbumDtoMapper` converts `Album` object to `ALbumDto` object
ArtistDto and ArtistDtoMapper	`ArtistDto` encapsulates `Artist` for transfer over network. `ArtistDtoMapper` converts `Artist` object to `ArtistDto` object
PlaylistDto and PlaylistDtomapper	`PlaylistDto` encapsulates `Playlist` for transfer over network. `PlaylistDtoMapper` converts `Playlist` object to `PlaylistDto` object
TrackDto and TrackDtoMapper	`TrackDto` encapsulates `Track` for transfer over network. `TrackDtoMapper` converts `Track` object to `TrackDto` object
Album and UserAlbum	`Album` is a db model for album record, `UserAlbum` has user-specific information about the album such as rating, score etc
Track and UserTrack	`Track` is a db model for track record, `TrackAlbum` has user-specific information about the track such as playcount, like/dislike etc
Playlist	`Playlist` is a db model for playlist record
Artist	`Artist` is db model for artist
Resource Classes	Handles `HTTP` requests and responses related to user data. It contains methods for handling `REST API` endpoints such as upload, import etc

LastFm Integration



Overview of LastFm

- Last.fm is a music-based social networking service that uses a music recommendation system, "Audioscrobbler" to keep track of the songs users listen to across various devices and platforms.
- This listening data is used to create personalized music recommendations for each user based on their listening habits.

Relationships Between Entities Involved

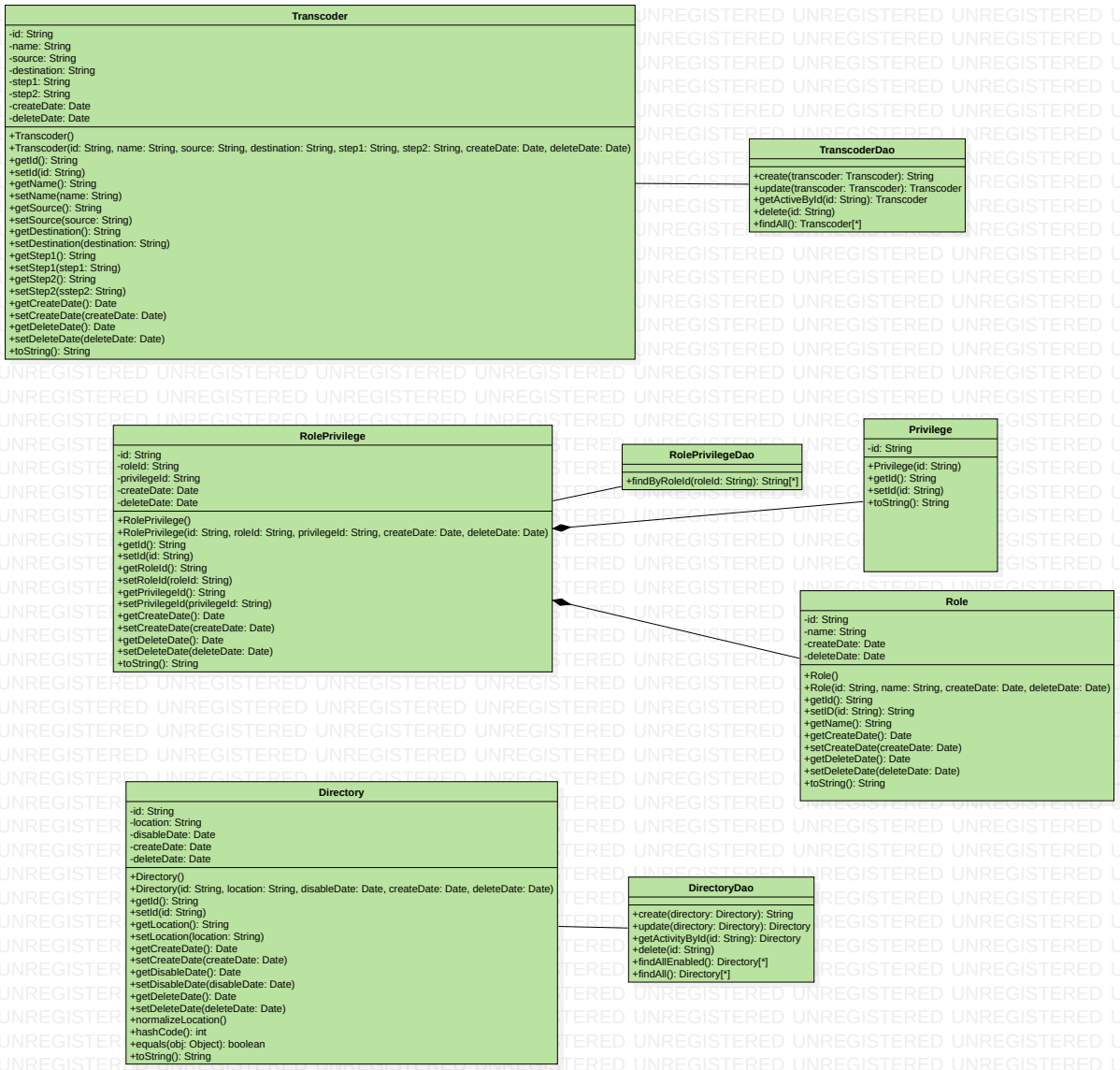
- `LastFMService` has dependencies with `User` and `Track`
- `User` has `lastFmSessionToken` created by `LastFmService.createSession()`

Classes and Responsibilities

Class	Responsibilities

Class	Responsibilities
LastFmService	`LastFmService` class is responsible for communicating with the Last.fm API to retrieve data about users and tracks. It provides methods for searching for users and tracks, retrieving information about a specific user or track, and updating information about a user.
User	`User` class is responsible for representing user data in the application. It has properties such as the user details, Last.fm session token.
Track	`Track` class is responsible for representing track data in the application. It has properties such as the track's name, artist, format, bitrate.
UserDao	Provides an interface to interact with the user data store to
TrackDao	Has methods for creating, reading, updating, and deleting track records in the data store
BaseDao	`BaseDao` class is a generic data access object that provides common functionality for interacting with the data store

Administration Features



Overview

- The administrator of music server has special privileges such as creating user, deleting user, create and delete the local directory which stores the music.
- The administrator can also manage the `Transcoders` supporting the platform.
- `User` object with **highest privilage** can accomplish these tasks.

Relationships Between Entities Involved

- `RolePrivilage` is composed of `Privilage` and `Role` classes.

- `RolePrivilageDao` is associated with `RilePrivilage`
- `DirectoryDao` is associated with `Directory`
- `TranscoderDao` is associated with `Transcoder`

Class Responsibilities

Class	Responsibilities
Role	`Role` is responsible for representing a role in the application containing ID and name
Privilage	`Privilage` is responsible for privilage in application containing ID
RolePrivilage	`RolePrivilege` is responsible for representing the relationship between a role and a privilege in the application. It has properties such as the role ID and privilege ID.
Directory	`Directory` is responsible for representing a directory in the application. It has properties such as the directory `id` and `location`
Transcoder	`Transcoder` is responsible for representing a transcoder in the application. It has properties such as the transcoder source and destination etc
RolePrivilageDao	Has methods for creating, reading, updating, and deleting records in the role privilege store
DirectoryDao	Has methods for creating, reading, updating, and deleting records in the directory store
TranscoderDao	Has methods for creating, reading, updating, and deleting records in the transcoder store

Task-2: Analysis

2a: Design Smells

1. **Missing Abstraction:**
   - In `User` class, `lastFmSessionToken` , `lastFmActive` , `FirstConnection` attributes are related to LastFM which can be combined to a new class `FMclass`

2. **Missing Abstraction:**
   - All the classes in `./model` folder are using attributes such as `createDate` `deleteDate` `updateDate` which can be combined to a single class i.e `Dateclass`

3. **Imperative Abstraction:**
   - `TrackLikedAsyncListener` `TrackUnlikedAsycListener` are similar and are combined to a single class named `TrackReactedAsyncListener`
   - `TrackLikedAsyncEvent` `TrackUnlikedAsyncEvent` are similar and are combined to a single class named `TrackReactedAsyncEvent`
   - **Dependencies:**
     - `trackResource.java` and event methods in `AppContext` are modified accordingly.

4. **Imperative Abstraction:**
   - `PlayStartedEvent` `PlayCompletedEvent` are similar and are combined to a single class `PlayActionEvent`
   - `PlayStartedAsyncListener` `PlayCompletedAsyncListener` are similar and are combined to a single class named `PlayActionAsyncListener`.
   - **Dependencies:**
     - `playService.java` and `App Context` folder are modified accordingly.

5. **Broken Modularisation:**

   In `model` folder `Role` and `Privilage` class is redundant compared to `RolePrivilege` , so other classes can be removed.

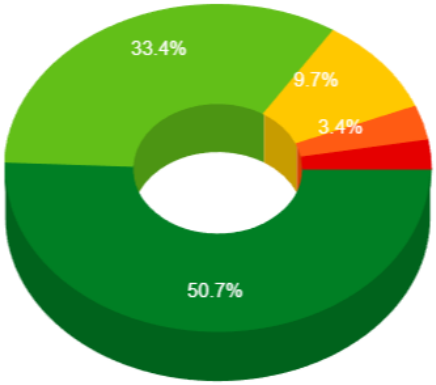
2b: Code metrics

- To record code metrics we used CodeMR plugin for IntellijIDEA.
- CodeMR is a code metrics analysis tool that can be used to evaluate and improve the quality of software code. It offers a range of features such as code analysis, visualization, and reporting.
- Summary

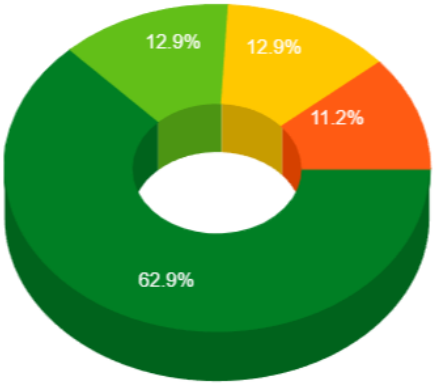
Analysis	Count
Total lines of code	4850
Number of classes	142
Number of packages	28
Number of external packages	55
Number of external classes	251
Number of problematic classes	5
Number of highly problematic classes	0

- Complexity
  - Complexity is indicated by the involvement a lot of entities and interactions, which can make it difficult to understand. Greater complexity raises the likelihood of accidentally disrupting these interactions, which in turn increases the risk of introducing defects when changes are made.
  - Mitigations:
    - Working on writing better documentation.
    - Eliminating the dead code.
    - Dividing the method into smaller pieces.

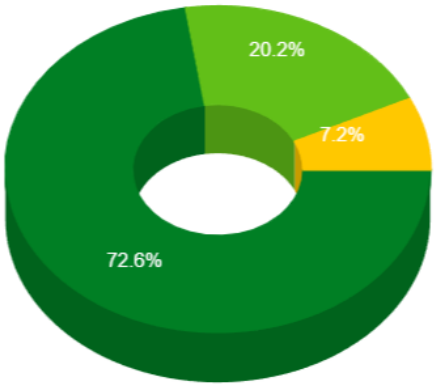
- Coupling
    - Coupling is a measure of how closely two classes or components are related. If two classes or components are tightly coupled, they are highly dependent on each other, meaning that a change to one class or component could cause a ripple effect of changes to the other. Tightly coupled systems tend to be more difficult to maintain, as any change to one class or component can have a significant effect on the other.
    - To reduce the issue of tight coupling, it is important to establish well-defined interfaces between the components that are coupled. By defining a clear, consistent interface, it becomes easier to modify one component without having to modify the other components that rely on it. Additionally, using dependency injection can help to decouple components, as the object does not need to be tightly coupled to the dependencies.
- Lack of Cohesion
    - Determine how well a class's methods are related to one another. High cohesion is preferable because it is associated with several desirable software characteristics such as robustness, reliability, reusability, and understandability. Low cohesion, on the other hand, is associated with undesirable characteristics such as being difficult to maintain, test, reuse, or even understand.
- Size
    - Size is one of the most basic and widely used types of software measurement. The number of lines or methods in the code is counted. A high count may indicate that a class or method is attempting to do too much work and should be divided. It could also indicate that the class will be difficult to maintain.



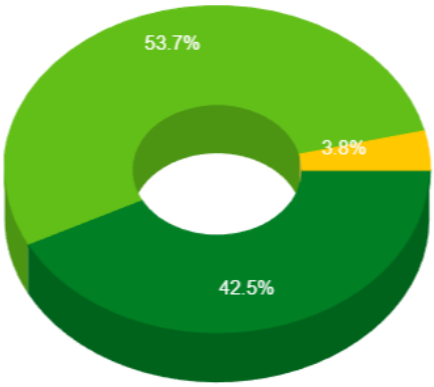
Complexity



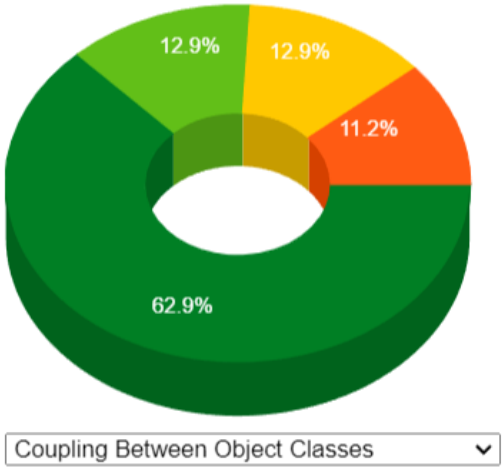
Coupling

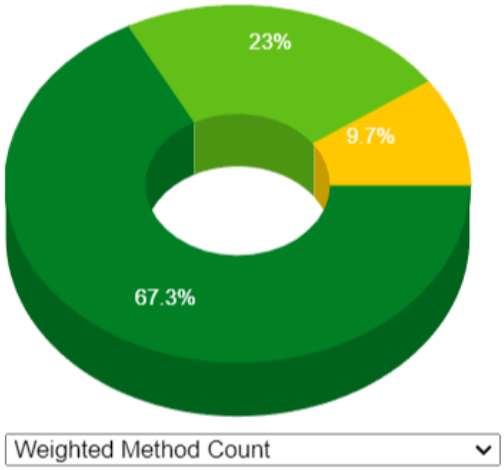


Lack of Cohesion



Size





Task -3: Refactoring

3a: Refactored Design Smells

FMclass

```java
package com.sismics.music.core.model.dbi;

public class FmClass {
    private String lastFmSessionToken;

    /**
     * Scrobbling on Last.fm active.
     */
    private boolean lastFmActive;

    /**
     * True if the user hasn't dismissed the first connection screen.
     */
    private boolean firstConnection;

    FmClass()
    {

    }
    FmClass( String lastFmSessionToken, boolean lastFmActive, boolean firstConnection)
    {
        this.lastFmSessionToken = lastFmSessionToken;
        this.lastFmActive = lastFmActive;
        this.firstConnection = firstConnection;
    }
    public String getLastFmSessionToken() {
        return lastFmSessionToken;
    }

    /**
     * Setter of lastFmSessionToken.
     *
     * @param lastFmSessionToken lastFmSessionToken
     */
    public void setLastFmSessionToken(String lastFmSessionToken) {
        this.lastFmSessionToken = lastFmSessionToken;
    }

    /**
     * Getter of lastFmActive.
     *
     * @return lastFmActive
     */
    public boolean isLastFmActive() {
        return lastFmActive;
    }

    /**
     * Setter of lastFmActive.
     *
     * @param lastFmActive lastFmActive
     */
    public void setLastFmActive(boolean lastFmActive) {
        this.lastFmActive = lastFmActive;
    }

    /**
     * Getter of firstConnection.
     *
     * @return firstConnection
     */
    public boolean isFirstConnection() {
        return firstConnection;
    }

    /**
     * Setter of firstConnection.
     *
     * @param firstConnection firstConnection
     */
    public void setFirstConnection(boolean firstConnection) {
        this.firstConnection = firstConnection;
    }
}
```

In `User` class, `lastFmSessionToken` , `lastFmActive` , `FirstConnection` attributes are related to LastFM which can be combined to a new class `FMclass`

Dateclass

```java
package com.sismics.music.core.model.dbi;

import java.util.Date;
public class DateClass {
    private Date createDate;
    private Date updateDate;
    private Date deleteDate;

    public Date getCreateDate() {
        return createDate;
    }

    public void setCreateDate(Date createDate) {
        this.createDate = createDate;
    }

    public Date getUpdateDate() {
        return updateDate;
    }

    public void setUpdateDate(Date updateDate) {
        this.updateDate = updateDate;
    }

    public Date getDeleteDate() {
        return deleteDate;
    }

    public void setDeleteDate(Date deleteDate) {
        this.deleteDate = deleteDate;
    }


    public DateClass(){

    }

    public DateClass(Date createDate,Date updateDate,Date deleteDate){
        this.createDate = createDate;
        this.updateDate = updateDate;
        this.deleteDate = deleteDate;
    }
}
```

All the classes in `./model` folder are using attributes such as `createDate` `deleteDate` `updateDate` which can be combined to a single class i.e `Dateclass`

TrackReactedAsyncEvent

```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;
import com.sismics.music.core.model.dbi.Track;
import com.sismics.music.core.model.dbi.User;

/**
 * Track reacted event.
 *
 * @author jtremeaux
 */
public class TrackReactedAsyncEvent {
    /**
     * Originating user.
     */
    private User user;

    /**
     * Liked track.
     */
    private Track track;

    private Boolean isLiked;
    public TrackReactedAsyncEvent(User user, Track track, Boolean isLiked) {
        this.user = user;
        this.track = track;
        this.isLiked = isLiked;
    }

    /**
     * Getter of user.
     *
     * @return user
     */
    public User getUser() {
        return user;
    }

    /**
     * Getter of track.
     *
     * @return track
     */
    public Track getTrack() {
        return track;
    }


    public Boolean getIsLiked(){return isLiked;}

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("user", user)
                .add("track", track)
                .toString();
    }
}
```

`TrackLikedAsyncEvent` `TrackUnlikedAsyncEvent` are similar and are combined to a single class named `TrackReactedAsyncEvent`

TrackReactedAsyncListener

```java
package com.sismics.music.core.listener.async;

import com.google.common.base.Stopwatch;
import com.google.common.eventbus.Subscribe;
import com.sismics.music.core.event.async.TrackReactedAsyncEvent;
import com.sismics.music.core.model.context.AppContext;
import com.sismics.music.core.model.dbi.Track;
import com.sismics.music.core.model.dbi.User;
import com.sismics.music.core.service.lastfm.LastFmService;
import com.sismics.music.core.util.TransactionUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.text.MessageFormat;

/**
 * Track unliked listener.
 *
 * @author jtremeaux
 */
public class TrackReactedAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log =
LoggerFactory.getLogger(TrackReactedAsyncListener.class);

    /**
     * Process the event.
     *
     * @param trackReactedAsyncEvent New directory created event
     */
    @Subscribe
    public void onTrackLiked(final TrackReactedAsyncEvent trackReactedAsyncEvent) throws
Exception {
        if (log.isInfoEnabled()) {
            if(trackReactedAsyncEvent.getIsLiked())
            log.info("Track liked event: " + trackReactedAsyncEvent.toString());
            else
                log.info("Track unliked event: " + trackReactedAsyncEvent.toString());
        }
        Stopwatch stopwatch = Stopwatch.createStarted();

        final User user = trackReactedAsyncEvent.getUser();
        final Track track = trackReactedAsyncEvent.getTrack();

        TransactionUtil.handle(() -> {
            if (user.fmObject.getLastFmSessionToken() != null) {
                final LastFmService lastFmService =
AppContext.getInstance().getLastFmService();
                lastFmService.unloveTrack(user, track);
            }
        });

        if (log.isInfoEnabled()) {
            if(trackReactedAsyncEvent.getIsLiked())
                log.info(MessageFormat.format("Track liked completed in {0}", stopwatch));
            else
                log.info(MessageFormat.format("Track unliked completed in {0}", stopwatch));
        }
    }
}
```

`TrackLikedAsyncListener` `TrackUnlikedAsycListener` are similar and are combined to a single class named `TrackReactedAsyncListener`

PlayActionEvent

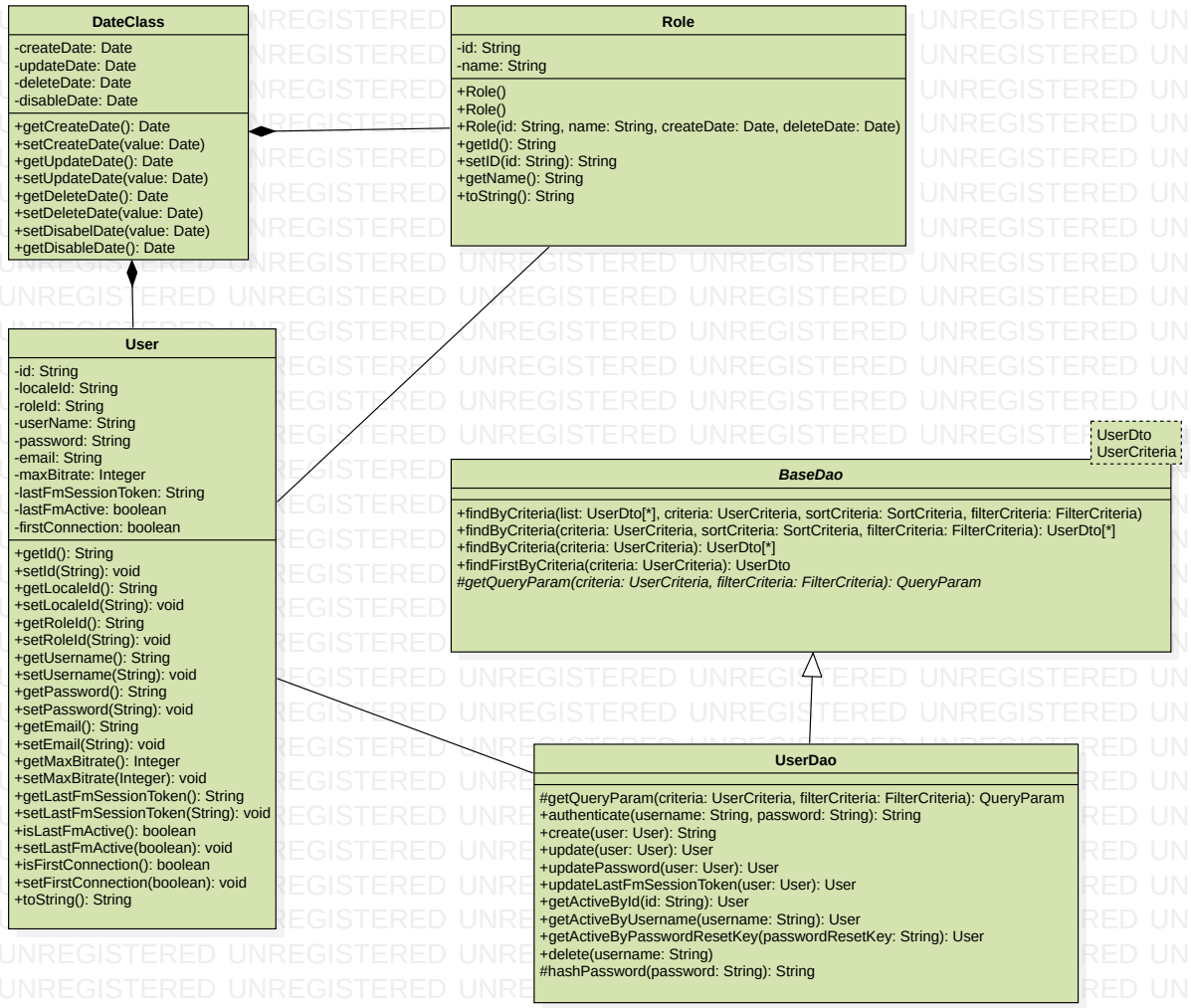
```java
package com.sismics.music.core.event.async;

import com.google.common.base.Objects;
import com.sismics.music.core.model.dbi.Track;

/**
 * Play action event.
 *
 * @author jtremeaux
 */
public class PlayActionEvent {
    /**
     * User ID.
     */
    private String userId;

    /**
     * Track.
     */
    private Track track;

    private Boolean isStarted;

    public PlayActionEvent(String userId, Track track, Boolean isStarted) {
        this.userId = userId;
        this.track = track;
        this.isStarted = isStarted;
    }

    public String getUserId() {
        return userId;
    }

    public Track getTrack() {
        return track;
    }

    public Boolean getIsStarted(){return isStarted;}

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
                .add("userId", userId)
                .add("trackId", track.getId())
                .toString();
    }
}
```

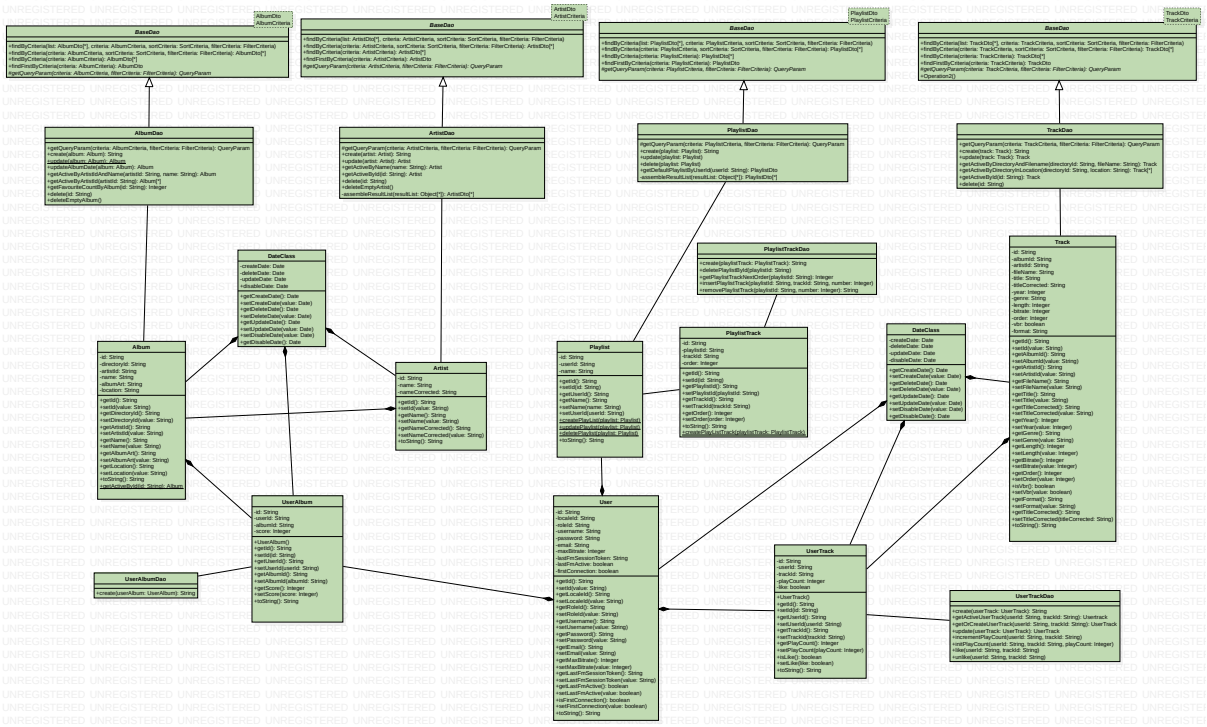
`PlayStartedEvent` `PlayCompletedEvent` are similar and are combined to a single class `PlayActionEvent`

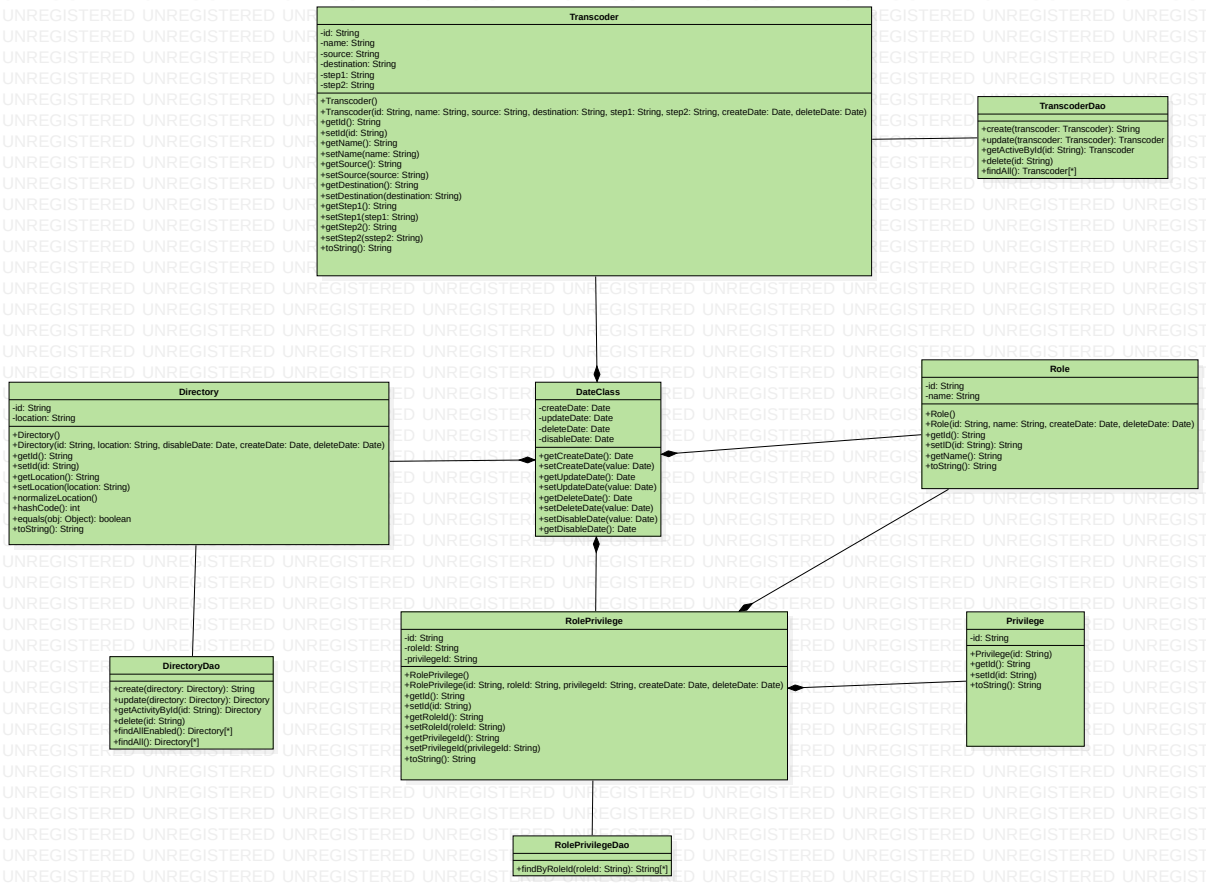
PlayActionAsyncListener

```java
package com.sismics.music.core.listener.async;

import com.google.common.eventbus.Subscribe;
import com.sismics.music.core.dao.dbi.UserDao;
import com.sismics.music.core.dao.dbi.UserTrackDao;
import com.sismics.music.core.event.async.PlayActionEvent;
import com.sismics.music.core.model.context.AppContext;
import com.sismics.music.core.model.dbi.Track;
import com.sismics.music.core.model.dbi.User;
import com.sismics.music.core.service.lastfm.LastFmService;
import com.sismics.music.core.util.TransactionUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * Play started listener.
 *
 * @author jtremeaux
 */
public class PlayActionAsyncListener {
    /**
     * Logger.
     */
    private static final Logger log = LoggerFactory.getLogger(PlayActionAsyncListener.class);

    /**
     * Process the event.
     *
     * @param playActionEvent Play started event
     */
    @Subscribe
    public void onPlayStarted(final PlayActionEvent playActionEvent) throws Exception {
        if (log.isInfoEnabled()) {
            if(playActionEvent.getIsStarted())
            log.info("Play started event: " + playActionEvent.toString());
            else
                log.info("Play completed event: " + playActionEvent.toString());
        }

        final String userId = playActionEvent.getUserId();
        final Track track = playActionEvent.getTrack();

        TransactionUtil.handle(() -> {
            if(!playActionEvent.getIsStarted()){
                UserTrackDao userTrackDao = new UserTrackDao();
                userTrackDao.incrementPlayCount(userId,track.getId());
            }
            final User user = new UserDao().getActiveById(userId);
            if (user != null && user.fmObject.getLastFmSessionToken() != null) {
                final LastFmService lastFmService =
AppContext.getInstance().getLastFmService();
                if(!playActionEvent.getIsStarted()){
                    lastFmService.scrobbleTrack(user,track);
                }
                else
                lastFmService.nowPlayingTrack(user, track);
            }
        });
    }
}
```

`PlayStartedAsyncListener` `PlayCompletedAsyncListener` are similar and are combined to a single class named `PlayActionAsyncListener`



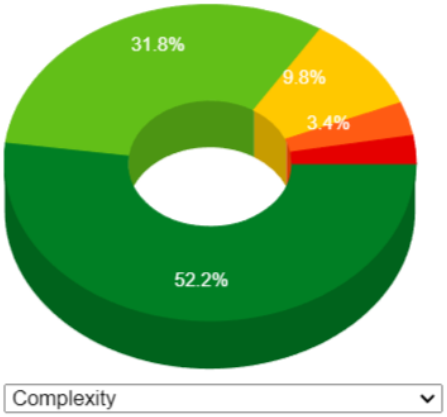

BaseDao
AlbumDao
ArtistDao
PlaylistDao
TrackDao
DateClass
PlaylistTrackDao
Track
Album
Artist
Playlist
PlaylistTrack
DateClass
UserAlbum
User
UserTrack
UserAlbumDao
UserTrackDao

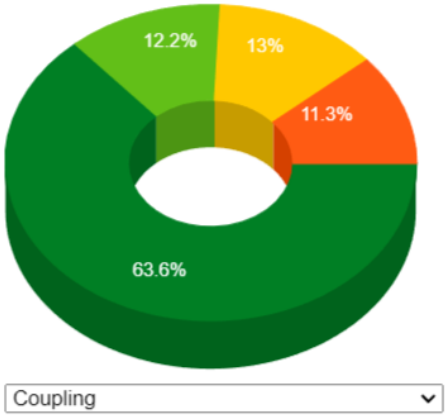

Transcoder
TranscoderDao
Directory
DateClass
Role
RolePrivilege
Privilege
DirectoryDao
RolePrivilegeDao

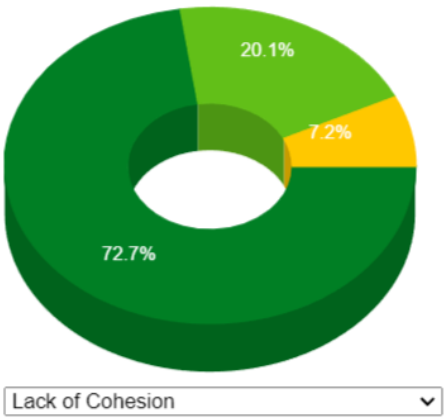
3b: Code Metrics After Refactoring

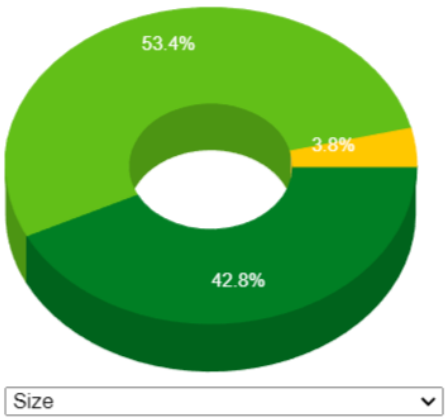
- Summary

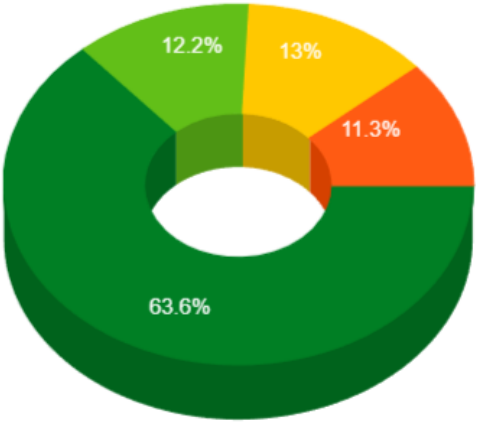
Analysis	Count
Total lines of code	4818
Number of classes	140
Number of packages	28
Number of external packages	55
Number of external classes	251
Number of problematic classes	5
Number of highly problematic classes	0



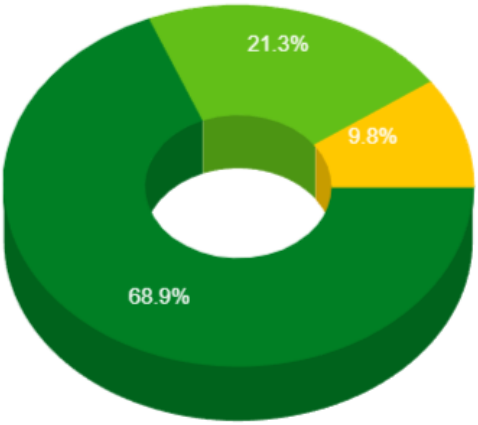








Coupling Between Object Classes



Weighted Method Count