LOG8430 - TP1

Group: Omega

2487723 - Ting-Hsuan Li
2486632 - Yu-Chi Shih
2104222 - Ali Chahbour
2207081 - Skander Hellal

October 18, 2025

Table of Contents

1 Q1. Software Architecture Analysis 3

1.1 Architecture UML Diagram . . . 3

1.2 Identified Styles . . . 3

1.3 Roles of the architecture modules . . . 3

1.3.1 UI (presentation). . . . 3

1.3.2 Application Services. . . . 3

1.3.3 Domain Model. . . . 4

1.3.4 Infrastructure. . . . 4

1.3.5 Adapter Parsers. . . . 4

1.3.6 Event Bus. . . . 4

2 Q2. SOLID design principles 4

2.1 PictureInPictureUtil (obey SRP) . . . 4

2.2 SplashActivity SRP (violate SRP) . . . 5

2.3 AspectRatioVideoView (obey OCP) . . . 6

2.4 PreferenceUpgrader (violate OCP) . . . 7

3 Q3: Measuring the design quality of AntennaPod 8

3.1 Metrics . . . 8

3.1.1 Size Metrics . . . 8

3.1.2 Complexity Metrics . . . 9

3.1.3 Inheritance Metrics . . . 9

3.1.4 Coupling and Cohesion Metrics . . . 10

3.2 Overall quality of the system and its impact . . . 12

3.2.1 Size . . . 12

3.2.2 Complexity . . . 12

3.2.3 Inheritance . . . 12

3.2.4 Coupling and Cohesion . . . 12

4 Q4. Finding and Refactoring Anomalies 13

4.1 Presentation of the Detected Anomalies . . . 13

4.2 Classification of Code Smells . . . 14

4.3 Refactoring Approach . . . 15

4.4 Justification of the Refactoring . . . 16

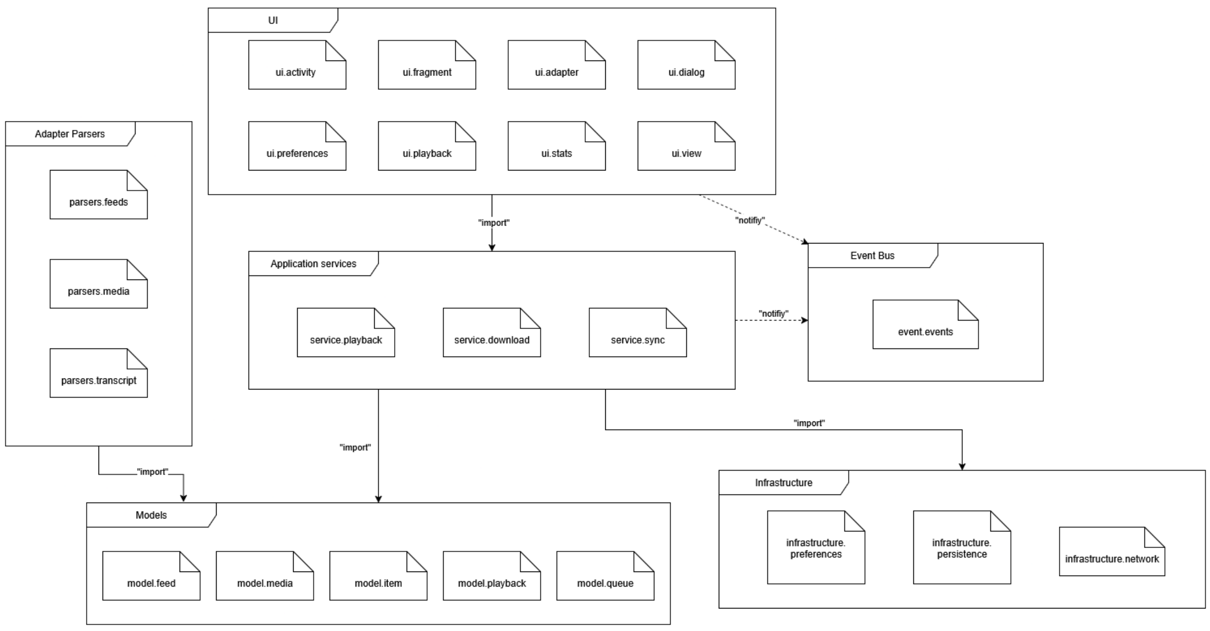


Figure 1: UML Architecture Diagram

1 Q1. Software Architecture Analysis

1.1 Architecture UML Diagram

Figure 1 represents the UML Diagram of AntennaPod's architecture.

1.2 Identified Styles

The architecture of AntennaPod primarily follows a **Layered Architecture**, where responsibilities are separated across distinct layers: the Presentation layer (UI), the Application Services layer (playback, download, synchronization), the Domain layer (models), and the Infrastructure layer (persistence, preferences, and networking). A dedicated Event layer is also present, represented by the Event Bus, which acts as a cross-cutting mechanism to decouple components and enable async updates. Dependencies flow downward between layers, consistent with the definition of the layered style we saw in class. In addition, the system exhibits characteristics of a **Service-Oriented Architecture**, since playback, download, and synchronization are encapsulated as distinct services exposed behind clear interfaces. Finally, it incorporates elements of an **Event-Driven Architecture**, as the Event Bus enables communication between UI and services.

1.3 Roles of the architecture modules

1.3.1 UI (presentation).

The UI packages implement the Presentation layer of the layered architecture. They are responsible for handling user interaction and rendering the domain information. In the service-oriented perspective, the UI acts as a *client* that invokes playback, download, and sync services. In the event-driven style, the UI is both a producer of events (user actions) and a consumer of events (such as playback status or queue changes).

1.3.2 Application Services.

They orchestrate the use of domain logic, persistence, and network modules. In the service-oriented view, each service is a distinct module exposing a clear responsibility. From the event-driven perspective, they act as producers of events like playback state changes, download progress, or sync updates.

1.3.3 Domain Model.

The domain model defines entities such as feeds, episodes, media, and playback items. In the layered organization, this layer provides the core business abstractions. In the service-oriented style, it represents the contract shared between services and clients. In the event-driven approach, these entities often serve as the payloads carried inside published events.

1.3.4 Infrastructure.

The storage, preferences, and network packages represent the Data Access layer. They provide persistence, configuration, and external communication facilities to the services. In the service-oriented view, they act as providers of data and connectivity. In the event-driven style, they play only a secondary role serving sometimes as sources of low-level events.

1.3.5 Adapter Parsers.

The parser packages function as adapters, transforming external inputs such as RSS feeds, metadata, or transcripts into domain objects. In the layered architecture, they connect external data sources to the domain layer. In the service-oriented style, they support services by supplying normalized domain data. They are not direct producers of events, but their outputs may be propagated through the event system.

1.3.6 Event Bus.

The event package implements the cross-cutting event layer. In the layered architecture, it belongs to the infrastructure, enabling communication without breaking dependency direction. It is not a business service in the service-oriented view, but in the event-driven style it plays the role of a control tower, receiving published events from UI or services and dispatching them to subscribers asynchronously.

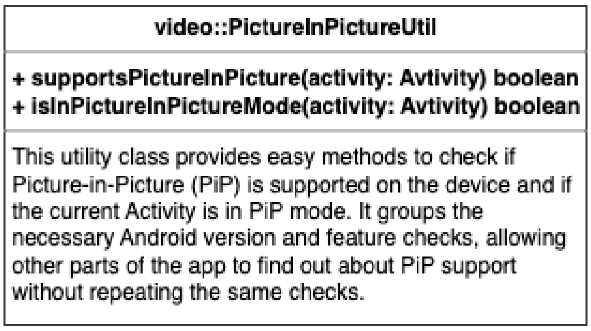
2 Q2. SOLID design principles

AntennaPod features several designs that adhere to the Single Responsibility Principle (SRP). The SRP states that a class should have only one reason to change, meaning it should focus on a single responsibility or task.

Another example of a SOLID principle observed in AntennaPod is the Open-Closed Principle (OCP). The OCP suggests that software entities, such as classes or modules, should be open to extension but closed to modification.

2.1 PictureInPictureUtil (obey SRP)

The **PictureInPictureUtil** class follows the Single Responsibility Principle (SRP) because it provides utility methods to check Picture-in-Picture (PiP) support and state in Android. Both methods (supportsPictureInPicture & isInPictureInPictureMode) serve the same purpose, determining the PiP capability and mode, and would only require modification if the underlying Android API for PiP detection changes. Since the class does not handle unrelated concerns like UI rendering or playback control, it maintains high cohesion and adheres to SRP.

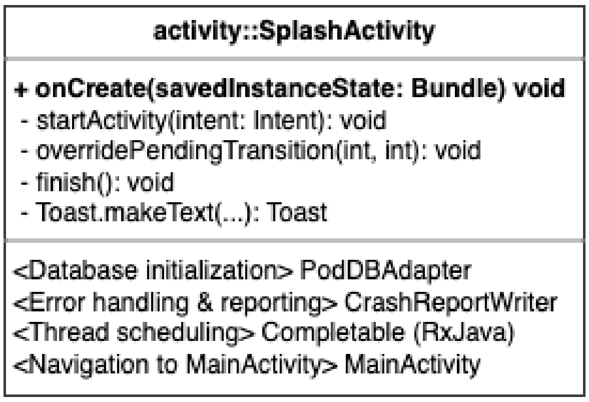

video::PictureInPictureUtil
+ supportsPictureInPicture(activity: Avtivity) boolean
+ isInPictureInPictureMode(activity: Avtivity) boolean

This utility class provides easy methods to check if Picture-in-Picture (PiP) is supported on the device and if the current Activity is in PiP mode. It groups the necessary Android version and feature checks, allowing other parts of the app to find out about PiP support without repeating the same checks.

Figure 2: PictureInPictureUtil UML diagram

2.2 SplashActivity SRP (violate SRP)

The **SplashActivity** class violates the Single Responsibility Principle (SRP) because it mixes multiple unrelated responsibilities, such as handling the splash screen UI and navigation, managing database initialization, writing crash reports, and orchestrating background threading. Each of these could change for different reasons (UI redesign, database schema update, or error handling changes), which means that the class has several reasons to change. According to SRP, it should focus only on presenting the splash screen and delegate other tasks to dedicated helper classes.

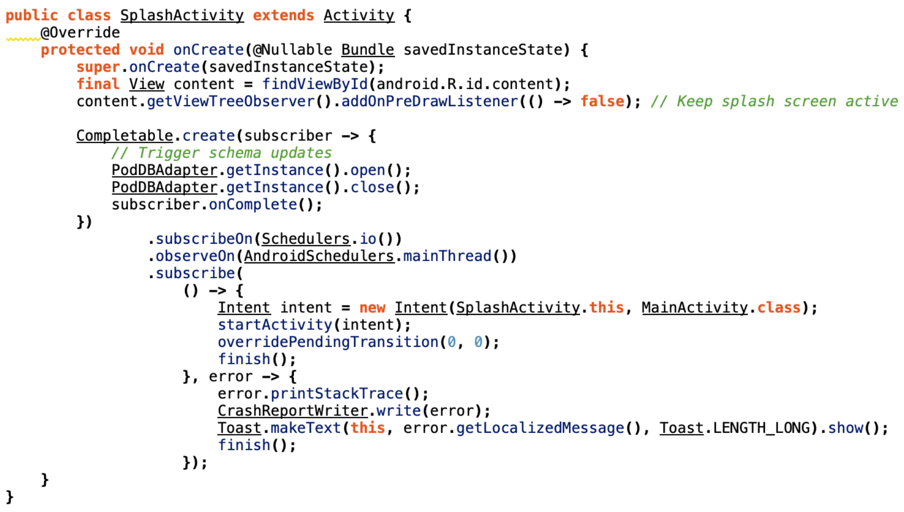


activity::SplashActivity

+ onCreate(savedInstanceState: Bundle) void
- startActivity(intent: Intent): void
- overridePendingTransition(int, int): void
- finish(): void
- Toast.makeText(...): Toast

<Database initialization> PodDBAdapter
<Error handling & reporting> CrashReportWriter
<Thread scheduling> Completable (RxJava)
<Navigation to MainActivity> MainActivity

Figure 3: SplashActivity UML diagram



```
public class SplashActivity extends Activity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        final View content = findViewById(android.R.id.content);
        content.getViewTreeObserver().addOnPreDrawListener(() -> false); // Keep splash screen active

        Completable.create(subscriber -> {
            // Trigger schema updates
            PodDBAdapter.getInstance().open();
            PodDBAdapter.getInstance().close();
            subscriber.onComplete();
        })
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(
                    () -> {
                        Intent intent = new Intent(SplashActivity.this, MainActivity.class);
                        startActivity(intent);
                        overridePendingTransition(0, 0);
                        finish();
                    }, error -> {
                        error.printStackTrace();
                        CrashReportWriter.write(error);
                        Toast.makeText(this, error.getLocalizedMessage(), Toast.LENGTH_LONG).show();
                        finish();
                    });
    }
}
```

Figure 4: SplashActivity code snippet

2.3 AspectRatioVideoView (obey OCP)

The AspectRatioVideoView class is an example that respects Open-Closed Principle (OCP), which means this class is open for extension and closed for modification. It extends the **VideoView** class and overrides the **onMeasure** method to manage layout behavior, but still preserves the original functionality. This design allows developers to customize video view behavior by subclassing and overriding methods, while the existing source code does not need to be changed. Through polymorphism, **AspectRatioVideoView** can also be used wherever a **VideoView** is expected, allowing multiple specialized video views to coexist in the system design.

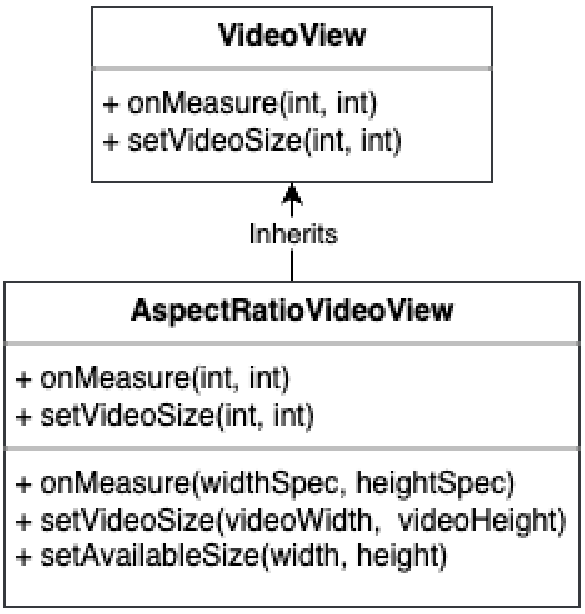
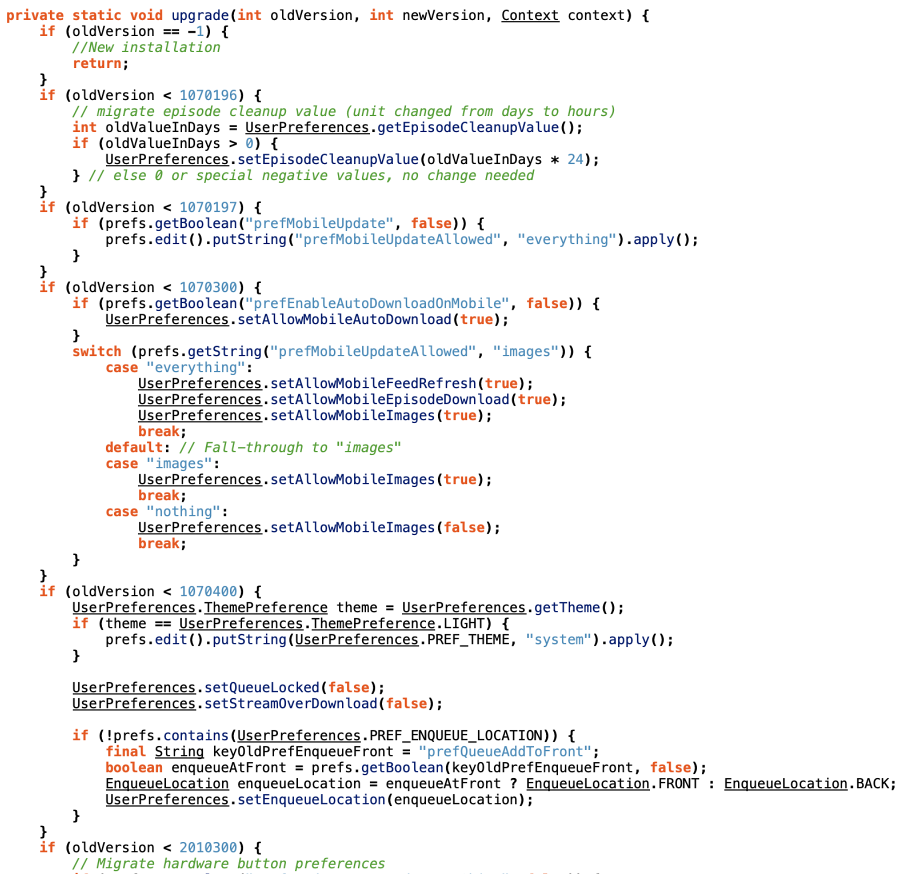


Figure 5: AspectRatioVideoView UML diagram

2.4 PreferenceUpgrader (violate OCP)

The **PreferenceUpgrader** class in the application demonstrates a violation of the Open–Closed Principle (OCP) . As mentioned before, OCP states that a class should be open for extension but closed for modification, which means that new method or function should be added without changing the existing code. In this class, the **upgrade** method contains a long sequence of **if** statements to check the old version number and apply migrations directly within the same method. This results in every time a new version is released, developers must modify this class by inserting additional conditions. By tightly coupling the class to version-specific upgrade logic, this design strongly violates the OCP and makes the code harder to maintain and test. A better approach would be to refactor the upgrade steps into separate, extendable migration classes or handlers, so that adding new upgrade rules can be done by extension rather than modification.



```java
private static void upgrade(int oldVersion, int newVersion, Context context) {
    if (oldVersion == -1) {
        //New installation
        return;
    }
    if (oldVersion < 1070196) {
        // migrate episode cleanup value (unit changed from days to hours)
        int oldValueInDays = UserPreferences.getEpisodeCleanupValue();
        if (oldValueInDays > 0) {
            UserPreferences.setEpisodeCleanupValue(oldValueInDays * 24);
        } // else 0 or special negative values, no change needed
    }
    if (oldVersion < 1070197) {
        if (prefs.getBoolean("prefMobileUpdate", false)) {
            prefs.edit().putString("prefMobileUpdateAllowed", "everything").apply();
        }
    }
    if (oldVersion < 1070300) {
        if (prefs.getBoolean("prefEnableAutoDownloadOnMobile", false)) {
            UserPreferences.setAllowMobileAutoDownload(true);
        }
        switch (prefs.getString("prefMobileUpdateAllowed", "images")) {
            case "everything":
                UserPreferences.setAllowMobileFeedRefresh(true);
                UserPreferences.setAllowMobileEpisodeDownload(true);
                UserPreferences.setAllowMobileImages(true);
                break;
            default: // Fall-through to "images"
            case "images":
                UserPreferences.setAllowMobileImages(true);
                break;
            case "nothing":
                UserPreferences.setAllowMobileImages(false);
                break;
        }
    }
    if (oldVersion < 1070400) {
        UserPreferences.ThemePreference theme = UserPreferences.getTheme();
        if (theme == UserPreferences.ThemePreference.LIGHT) {
            prefs.edit().putString(UserPreferences.PREF_THEME, "system").apply();
        }

        UserPreferences.setQueueLocked(false);
        UserPreferences.setStreamOverDownload(false);

        if (!prefs.contains(UserPreferences.PREF_ENQUEUE_LOCATION)) {
            final String keyOldPrefEnqueueFront = "prefQueueAddToFront";
            boolean enqueueAtFront = prefs.getBoolean(keyOldPrefEnqueueFront, false);
            EnqueueLocation enqueueLocation = enqueueAtFront ? EnqueueLocation.FRONT : EnqueueLocation.BACK;
            UserPreferences.setEnqueueLocation(enqueueLocation);
        }
    }
    if (oldVersion < 2010300) {
        // Migrate hardware button preferences
```

Figure 6: PreferenceUpgrader code snippet

3 Q3: Measuring the design quality of AntennaPod

3.1 Metrics

In this section, we will examine the classic metrics. These metrics have been analyzed using the IDE Intellij with the plugins MetricTree [1] and MetricReloaded [2]. Each metric is measured at different levels: project, package, class, and method. For each metric, we will provide the level, total value, average, minimum value, maximum value, and standard deviation.

3.1.1 Size Metrics

Size metrics assess the overall scale of a software system by counting structural elements such as lines of code (LOC) and number of methods (NOM). These measures provide a basic indication of implementation effort, maintainability, and complexity.

Lines of Code(LOC): counts the number of return characters(e.g. \r) in a method, a class, or a compilation unit.

Metric	Total	Average	Std Dev	Max	Min
LOC(method)	16696	12.08	13.71	136	2
LOC(class)	18320	88.08	109.62	681	3
LOC(interface)	65	13	5.14	18	3
LOC(Package)	21813	752.17	497.18	1851	17
LOC(Module)	21813	-	-	-	-
LOC(Project)	21813	-	-	-	-

Table 1: Lines of code obtained by MetricsReloaded according to different scopes

Table 1 presents the Lines of Code (LOC) metrics collected using the MetricsReloaded plugin in IntelliJ IDEA. At the method level, the project contains a total of 16,696 lines of code, with an average of 12.08 lines per method, which suggests relatively concise implementations. At the class level, the total LOC reaches 18,320, corresponding to an average of 88.08 lines per class and a maximum of 681, indicating that while most classes are moderately sized, a few large classes contribute significantly to the total. Interfaces, on the other hand, are much smaller, averaging 13 lines each, consistent with their role in defining abstractions without implementation details.

At higher abstraction levels such as package, module, and project, the LOC values mainly represent the aggregate size of lower-level entities. Since the project consists of a single module, the LOC totals are identical across these levels (21,813 LOC). Therefore, these numbers provide limited insight into design quality or internal structure. The method and class-level LOC metrics are more informative for evaluating cohesion, complexity, and code organization.

Number of Methods(NOM): counts the number of methods declared in a class, not the derived methods.

Metric	Total	Average	Std Dev	Max	Min
NOM(class)	1382	13.62	6.6	41	1

Table 2: NOM of class obtained by MetricsTree

Table 2 displays the Number of Methods (NOM) metrics collected using the MetricsTree plugin at the class level. The project contains an average of 13.62 methods per class, with a standard deviation of 6.6. While most classes fall within a reasonable range, the maximum value of 41 methods stands out and may indicate the presence of a large-class anti-pattern. Nevertheless, the overall distribution suggests that classes generally contain a moderate number of methods relative to their attributes, and there is no clear evidence that they violate the single-responsibility principle or handle multiple unrelated concerns.

3.1.2 Complexity Metrics

The complexity of a program can be measured using metrics like cyclomatic complexity (CC), weighted method count (WMC), and response for classes (RFC).

Cyclomatic Complexity(CC): It counts the number of independent paths in a code slice.

Upon examining the results of these metrics in Table 3, several observations can be made. The average Cyclomatic Complexity (CC) is 2.58, indicating that the code is generally easy to understand. However, the notably high maximum value of 36 suggests the presence of a large method, which introduces complexity and can make the code difficult to read.

Metric	Total	Average	Std Dev	Max	Min
CC(method)	3569	2.58	2.82	36	1

Table 3: Cyclomatic complexity of method obtained by MetricsReloaded

Weighted method count(WMC): counts the sum of a metric over all methods of a class. The metric could be the complexity, the LOC, or none (unweighted)

Response for Class(RFC): counts the number of public methods of a class and all methods accessed directly by these public methods.

The Weighted Methods per Class (WMC) metric assesses the overall complexity of the methods in each class. With an average WMC value of 15.69 (SD = 20.48), it suggests that most classes have a manageable level of internal complexity. However, the maximum WMC of 137 indicates that some classes may be overly complex, which could lead to reduced maintainability and increased costs for modifications or debugging.

The Response for a Class (RFC) metric helps quantify the number of methods a class invokes in response to a received message, including both internal and external calls. The average RFC value is 12.18, with a standard deviation of 14.11, which indicates that most classes have limited dependencies and demonstrate good modularity. However, the maximum RFC of 90 suggests that some classes have extensive interaction surfaces. If these classes are not refactored, it could complicate testing and maintenance.

Overall, the low average values of CC (Cyclomatic Complexity), WMC (Weighted Methods per Class), and RFC (Response for a Class) suggest that the system is well-structured, maintainable, and easy to evolve. However, a few outlier classes exhibit high complexity and interaction counts, which should be closely examined to avoid potential reliability or scalability issues in the future.

Metric	Total	Average	Std Dev	Max	Min
WMC(class)	3264	15.69	20.48	137	0
RFC(class)	2643	12.18	14.11	90	1

Table 4: WMC and RFC of class obtained by MetricsReloaded

3.1.3 Inheritance Metrics

The inheritance of a can be described by metrics such as the depth of inheritance (DIT) and the number of children (NOC).

Number of children (NOC): counts the number of classes immediately derived from a basic class.

Depth of Inheritance (DIT): calculates the maximal length from a derived class to a basic class in the inheritance structure of the system.

Observing the values in Table 5, we can observe that the average DIT and NOC values generally fall within reasonable ranges, which is a positive sign for code maintainability.

The average Depth of Inheritance Tree (DIT) value is 1.2, with a maximum of 3. This indicates that most classes are relatively shallow in the inheritance hierarchy, which positively contributes to code maintainability and comprehensibility. For the Number of Children (NOC), the average value is 0.28, with a maximum of 9. This suggests that most classes have few or no subclasses, which is generally a positive sign as it indicates a limited degree of inheritance complexity.

Overall, the DIT and NOC metrics suggest that the system's class hierarchy is reasonably well-balanced. However, classes with high NOC values should be reviewed to ensure a maintainable and modular design.

Metric	Total	Average	Std Dev	Max	Min
NOC	56	0.28	1.04	9	0
DIT	242	1.2	0.43	3	1

Table 5: NOC and DIT obtained by MetricsReloaded

3.1.4 Coupling and Cohesion Metrics

Coupling refers to the extent to which two modules are interconnected, while cohesion measures how well the various elements within a single module are related to one another. To assess coupling, we can use efferent coupling (Ce) and afferent coupling (Ca). To evaluate cohesion, the lack of cohesion in a method can be measured using LCOM (Lack of Cohesion in Methods).

Afferent Coupling (Ca): counts the number of external classes that are coupled (asper the definition of CBO) with classes from another package with respect toincoming coupling (i.e., external classes use methods of internal classes)

Efferent Coupling (Ce): counts the number of classes that are "coupled" with another class.

As shown in Table 6, for the Ce metric, a class with a Ce value of 164 indicates a high level of coupling with other classes, which could negatively impact its maintainability and understandability. Regarding the Ca metric, the range of values from 0 to 288 is particularly noteworthy. A package with 288 afferent couplings suggests it is heavily utilized and has a significant influence on other parts of the codebase. In both cases, the high standard deviations imply considerable variation in these metrics, indicating a need to closely examine the packages with high Ca values and the classes with high Ce values. Reducing such dependencies and adhering to sound design principles may enhance code maintainability and readability. The efferent coupling (Ce) measures 50.86, with a standard deviation of 52.25. In comparison, the afferent coupling (Ca) also has a value of 50.86, but with a standard deviation of 63.67. Typically, we prefer the value of Ce to be higher than that of Ca; however, in this instance, they are equal. It is encouraging to note that the maximum value of Ca exceeds that of Ce.

Metric	Total	Average	Std Dev	Max	Min
Ce(package)	1475	50.86	52.25	164	0
Ca(package)	1475	50.86	63.67	288	0

Table 6: Ce and Ca of package obtained by MetricsReloaded

The Tight Class Cohesion (TCC) metric measures the ratio of pairs of methods that directly access attributes to the total number of possible pairs of methods within the same class. This metric is applicable at the class level, as demonstrated.

High LCOM values in classes can increase complexity, reduce maintainability, and indicate a violation of the Single Responsibility Principle, necessitating careful review and potential refactoring to enhance code quality and adherence to design principles.

As shown in Table 7, the Tight Class Cohesion (TCC) metric has an average of 0.5 with a standard deviation of 0.3. Meanwhile, the Lack of Cohesion of Methods (LCOM) metric shows an average value of 1.88 with a standard deviation of 1.41. The relatively low TCC and LCOM values above 1 indicate a noticeable lack of cohesion between methods and their associated data in various parts of the system. Although the overall average suggests moderate cohesion, the high variability among classes points to inconsistent design quality. Of particular concern is the class with an LCOM value of 7, which signifies a severe cohesion issue and may indicate the presence of a "God Class" anti-pattern.

Metric	Total	Average	Std Dev	Max	Min
TCC	48.2	0.5	0.3	1	0
LCOM	391	1.88	1.41	7	0

Table 7: TCC and LCOM obtained by MetricsReloaded

3.2 Overall quality of the system and its impact

Building on the categorization of design metrics from the previous section, this section evaluates the overall system quality and discusses how these metrics affect system development and maintainability.

3.2.1 Size

When evaluating metrics like Lines of Code (LOC) and Number of Methods (NOM), higher values typically indicate a decrease in understandability, analyzability, and maintainability. This also results in reduced modifiability and testability. On the other hand, moderate or lower values in these metrics usually suggest better overall system quality and maintainability.

The project's lines of code (LOC) values indicate that most methods and classes are relatively small. This is a positive sign for software quality, as smaller methods typically focus on a single, well-defined task. This enhances readability, testability, and reusability. Such characteristics contribute to the system's understandability, adaptability, and ease of maintenance, which ultimately facilitate smoother development and evolution over time.

The presence of a class with an unusually high number of lines and methods indicates a potential violation of the Single Responsibility Principle (SRP) and suggests the existence of a large class (often referred to as a "God Class") anti-pattern. These situations can hinder maintainability, increase the risk of defects, and complicate future extensions. Overall, while most of the system demonstrates good structural quality, identifying and refactoring these problematic classes would enhance the system's maintainability and improve long-term development efficiency.

3.2.2 Complexity

The cyclomatic complexity (CC) metric reveals that most methods within the system exhibit low levels of complexity, indicating that they are generally not highly complicated. Similarly, the metrics for Weighted Methods per Class (WMC) and Response for Class (RFC) show low average values as well. This suggests that both methods and classes maintain manageable complexity levels. Such low complexity enhances maintainability, as it makes the code easier to understand, test, and adapt. As highlighted in "Clean Code," deeply nested logic structures can be challenging to comprehend, whereas implementing early returns can significantly improve readability.

However, the high maximum values of Weighted Methods per Class (WMC) and Response for a Class (RFC) indicate that certain classes may be outliers. These outliers could display design anti-patterns, such as "god classes" or "shotgun surgery." It is important to investigate these classes, as they may adversely affect the overall maintainability and quality of the system's design.

3.2.3 Inheritance

The inheritance metrics show that the system features a shallow and simple class hierarchy. Most classes are only one or two levels deep in the inheritance chain. This shallow structure positively impacts code maintainability, comprehensibility, and testability. In contrast, systems with excessive inheritance depth can be challenging to analyze and debug.

For the Number of Children (NOC), the average value is 0.28, indicating that most classes have few or no subclasses. While low inheritance can help prevent unnecessary complexity, it may also suggest an underutilization of polymorphism and design patterns. This can lead to code duplication and violations of the DRY (Don't Repeat Yourself) principle. Additionally, the class with nine children may represent a design hotspot or a potential violation of the Single Responsibility Principle.

Overall, the class hierarchy of the system is well-balanced; however, further examination of classes with high NOC values is advisable to ensure modularity and maintainability.

3.2.4 Coupling and Cohesion

The analysis of the Ca, Ce, TCC, and LCOM metrics indicates that the system's classes exhibit low cohesion, suggesting that their responsibilities are distributed across multiple components. This lack of clear responsibility boundaries can complicate the system's understanding, maintenance, and testing, as classes become increasingly interdependent and less self-contained. While most classes are relatively short and contain only a few methods, these methods do not always align with the intended purpose of the class, indicating a potential violation of the Single Responsibility Principle (SRP).

In other words, although many classes maintain a reasonable level of internal consistency, the variability in cohesion suggests that certain parts of the system may be poorly structured or overloaded with responsibilities. This structural imbalance could hinder the long-term maintainability and evolution of the codebase.

At the package level, the Ce and Ca metrics further reveal that the system exhibits high and uneven coupling, where a few central packages either depend on or are relied upon by many others. This uneven distribution of dependencies implies that changes made to these core packages could have widespread effects, increasing the risk of introducing unintended side effects. From a cohesion standpoint, a high level of coupling typically correlates with low cohesion because packages that interact excessively with others often lack a single, well-defined purpose.

Overall, the system's design reveals concerns regarding both coupling and cohesion. This violates the principles of modularity and the Open-Closed Principle (OCP), making the system more challenging to extend and maintain. To improve software quality, the structures of the packages and classes should be refactored to reduce interdependencies and ensure that each component encapsulates a cohesive set of related responsibilities.

4 Q4. Finding and Refactoring Anomalies

4.1 Presentation of the Detected Anomalies

To locate potential anomalies, we used the heatmap visualization from the Understand tool [3], based on maximum cyclomatic complexity, to highlight the classes and methods with the highest control-flow complexity in the system. This visualization helped identify two clear hotspots: the class MainActivity, which presented a very high overall complexity, and the method PreferenceUpgrader.upgrade(), which contained the highest individual cyclomatic complexity value in the project. These findings were then confirmed through detailed metric inspection, leading to the identification of a God Class and a Long Method anomaly.

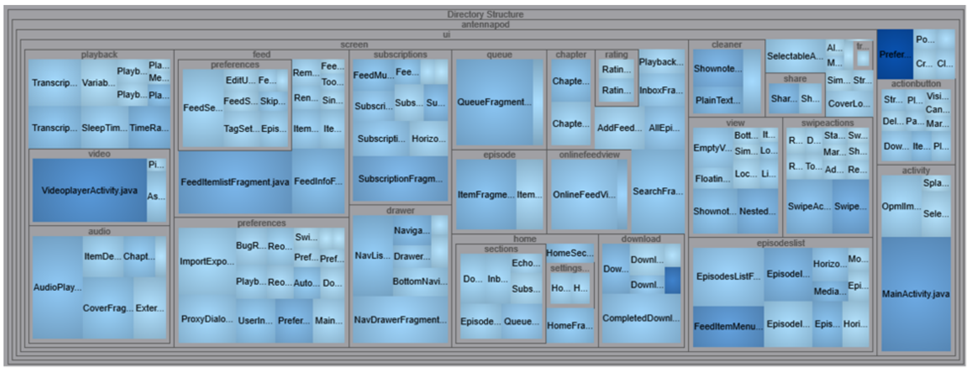


Figure 7: Understand [3] heatmap based on Max Cyclomatic Complexity showing complex classes and methods.

Anomaly 1 – `de.danoeh.antennapod.activity.MainActivity`

According to the metrics extracted from Understand, the `MainActivity` class contains 747 lines of code, 50 methods, 15 attributes, and a total cyclomatic complexity of 158. The comment-to-code ratio is only 3%, and the class shows 82 coupled classes with a lack of cohesion percentage of 92%. These values largely exceed common thresholds for maintainability (LOC > 500, WMC > 100, NOM > 20), indicating a class that centralizes multiple responsibilities.

In the heatmap, this class appeared as one of the darkest components, confirming its excessive complexity and density. Manual inspection revealed that it mixes logic for user interface control, playback state management, and navigation. This violates the Single Responsibility Principle (SRP) and corresponds to the God Class smell.

Anomaly 2 – `de.danoeh.antennapod.PreferenceUpgrader.upgrade()`

The `upgrade()` method has 128 lines of code and a cyclomatic complexity of 36, which is far above the acceptable threshold (CC > 15). It contains numerous nested conditional structures and repetitive logic to handle migration across multiple application versions. This design makes it difficult to extend or maintain the method and violates both the Single Responsibility Principle (SRP) and the Open/Closed Principle (OCP).

4.2 Classification of Code Smells

In this section, previously spotted anomalies will be classified in terms of code smells. This will be achieved by using the thresholds for metrics and conditions defined by Lanza and Marinescu [4] to help identify specific code smells. The relevant metrics thresholds defined based on 45 Java projects are presented in the following table.

Metric	Low	Average	High	Very High
CYCLO/method	1.1	2.0	3.1	4.7
LOC/method	7	10	13	19.5
WMC	5	14	31	47
LOC/class	28	70	130	195
NOM/class	4	7	10	15

Table 8: Statistical and derived thresholds of 45 Java systems [4]

The god class is a design flaw in which a class performs too much work and therefore has too many responsibilities. According to Lanza and Marinescu [4], the three conditions necessary to justify the god class code smell are:

- ATFD > FEW = 2 - 3
- WMC ≥ VERY HIGH = 47
- TCC < ONE THIRD

The ATFD metric measures the amount of direct access to attributes of other classes, and a high ATFD signify a lack of encapsulation and high coupling. In the case of the first anomaly `MainActivity`, the ATFD metric is not directly available in Understand. However, it is possible to manually look at the class and count that ATFD >> FEW.

The WMC metric measures the total complexity of the class and has a value of 136 for `MainActivity`, which is much higher than the VERY HIGH threshold of 47.

Finally, the TCC metric measures the cohesion of the class and is once again not available in Understand. However, the LCOM (lack of cohesion of method) which is another metric measuring the cohesion and has a value of 90 for `MainActivity` signifying a very low cohesion.

The anomaly for `MainActivity` is therefore a god class. It is also possible to note a very high LOC (line of code) of 640 and NOM (number of method) of 37 showing a very large class which is typical of god classes which have many responsibilities.

Brain methods are methods that centralize the functionality of a class and therefore do too much work on their own. When it comes to the brain method code smell, Lanza and Marinescu [4] define four necessary conditions:

- LOC > HIGH (class) / 2 = 65

- CYCLO $\geq$ HIGH = 3.1
- NOAV > MANY = 8
- MAXNESTING $\geq$ SEVERAL = 4 - 5

The LOC metric is the number of lines of code and has a value of 128 for `PreferenceUpgrader.upgrade()` which is much higher than the threshold of 65 signifying a very large method.

The second metric measures the amount of conditional branches, which should not be too high in a method that has a focused functionality. The method `PreferenceUpgrader.upgrade()` has a CYCLO value of 36 largely exceeding the threshold of 3.1.

The NOAV metric measures the number of accessed variables in the method which should be limited to 8. In the case of `PreferenceUpgrader.upgrade()`, the NOAV = 74 >> 8.

Finally, the MAXNESTING metric measures the nesting of the method and has a value of 2 which is lower than the threshold of 4 meaning that this condition is not fulfilled.

Despite the method `PreferenceUpgrader.upgrade()` not having a high nesting, the fact that all other metrics (LOC, CYCLO and NOAV) far exceed the established threshold justifies classifying this anomaly as a brain method due to its excessive length, high complexity, and high variable usage which are typically the flaws characterizing the brain method.

4.3 Refactoring Approach

Refactoring the god class

To correct the God Class anomaly, the responsibilities of `MainActivity` were separated into multiple classes, each with a clear and single purpose. For instance:

- A new class PlaybackController was extracted to handle playback logic.
- A separate NavigationHandler manages user navigation and activity transitions.
- The remaining class focuses solely on UI rendering and interaction.

Before Refactoring:

```
public class MainActivity extends AppCompatActivity {
    private void startPlayback() { ... }
    private void handleNavigation() { ... }
    private void updateUI() { ... }
}
```

After Refactoring:

```
public class MainActivity extends AppCompatActivity {
    private PlaybackController playbackController;
    private NavigationHandler navigationHandler;
    private void updateUI() { ... }
}
```

This separation improves cohesion, reduces coupling, and aligns the design with SRP.

Refactoring the long method

The long `upgrade()` method was refactored by applying the Strategy Pattern, separating version-specific upgrade logic into independent classes. Each migration step is encapsulated in its own class implementing a common interface, allowing new upgrade steps to be added without modifying existing code.

Before Refactoring:

```
private static void upgrade(int oldVersion, int newVersion, Context context) {
    if (oldVersion < 1070196) { ... }
    if (oldVersion < 1070197) { ... }
    if (oldVersion < 1070300) { ... }
}
```

After Refactoring:

```
interface UpgradeStep {
    boolean appliesTo(int oldVersion);
    void apply(Context context);
}

class Upgrade1070196 implements UpgradeStep { ... }
class Upgrade1070197 implements UpgradeStep { ... }

private static void upgrade(int oldVersion, int newVersion, Context context) {
    for (UpgradeStep step : upgrades) {
        if (step.appliesTo(oldVersion)) step.apply(context);
    }
}
```

This approach makes the code open for extension (adding new upgrade steps) but closed for modification, restoring compliance with the OCP.

4.4 Justification of the Refactoring

The proposed refactorings directly address the metrics-based anomalies identified earlier.

- **MainActivity (God Class):** Splitting the class into smaller components such as `PlaybackController` and `NavigationHandler` reduces its WMC and LOC, increasing cohesion and lowering coupling. Each class now has a single, focused responsibility, which improves readability, testability, and long-term maintainability. This correction aligns the design with the Single Responsibility Principle (SRP) and reduces architectural complexity.
- **PreferenceUpgrader.upgrade() (Long Method):** Refactoring the long conditional structure into multiple modular UpgradeStep classes distributes complexity and simplifies maintenance. The method now follows the Open–Closed Principle (OCP), as new migration steps can be added without altering existing logic. This significantly lowers cyclomatic complexity and improves clarity.

Overall, both refactorings improve cohesion, reduce code duplication, and enhance the maintainability index. The system becomes more modular, easier to extend, and more consistent with clean code and layered design principles.

References

[1] JetBrains, "MetricsTree (Version 2025.1.2) [Software]," 2025, available: https://plugins.jetbrains.com/plugin/13959-metricstree.

[2] ——, "MetricsReloaded (Version 1.12) [Software]," 2022, available: https://plugins.jetbrains.com/plugin/93-metricsreloaded.

[3] SciTools, "Understand (Version 6.4) [Software]," 2025, available: https://scitools.com/feature/understand/.

[4] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer, 2006. [Online]. Available: https://link.springer.com/book/10.1007/3-540-39500-7