# Introduction

ZXing ("Zebra Crossing") is an open-source, multi-format 1D/2D barcode image processing library implemented in Java, with ports to other languages [1]. The project that was analyzed, using Understand [2], is the Barcode Scanner application that appears on the Google Play store. It was downloaded over 100 million times, has close to 650k reviews, and a 3.9-star rating.

The source code for the app is hosted and maintained on GitHub [[2]](#_bookmark0). The source code for each release is maintained as a tag. In total we analyzed 6 releases of the app. The releases range from 2014 (when the project was moved to GitHub) to 2022 (the most recent release when running the analysis). Most of the releases were minor version releases and the last three releases analyzed each spanned about three years between releases. The first two releases (2.3.0 and 3.0.0) analyzed occurred only one month apart but were included to show the shift between major versions 2.0 and 3.0.

# Analysis

The table below shows the metrics that were captured per release. The metrics include complexity, volume, and object-oriented (CK) metrics.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Release** | **Release Date** | **Average Cyclomatic**  **Complexity** | **Average Comment /Code** | **Average DIT** | **Average NIM** | **Average NIV** | **Average Local Methods** | **Average CBO** | **Average LCOM** | **Average NL** | **Average LOC** | **Average NSC** | **Average Number of functions** | **Total Classes** | **Total Lines of Code** | **Total Files** | **Total Functions** |
| 2.3.0 | 2014-01-18 | 2.57 | 0.28 | 1.69 | 5.91 | 2.39 | 7.64 | 6.88 | 18.23 | 57.00 | 40.25 | 20.51 | 5.91 | 587 | 64,884 | 476 | 2,815 |
| 3.0.0 | 2014-02-28 | 2.51 | 0.29 | 1.72 | 6.10 | 2.45 | 7.90 | 6.94 | 18.96 | 56.93 | 40.14 | 20.40 | 4.49 | 555 | 75,695 | 643 | 2,886 |
| 3.2.0 | 2015-02-15 | 2.58 | 0.29 | 1.72 | 6.17 | 2.46 | 8.01 | 7.02 | 19.11 | 57.26 | 40.50 | 20.37 | 6.10 | 564 | 65,399 | 471 | 2,871 |
| 3.3.0 | 2016-09-16 | 2.55 | 0.30 | 1.72 | 6.08 | 2.49 | 7.91 | 7.00 | 18.81 | 57.26 | 40.35 | 20.19 | 6.00 | 582 | 66,692 | 485 | 2,910 |
| 3.4.0 | 2019-05-17 | 2.56 | 0.31 | 1.74 | 6.12 | 2.43 | 7.98 | 7.13 | 18.68 | 57.78 | 40.60 | 20.30 | 5.87 | 561 | 67,682 | 499 | 2,931 |
| 3.5.0 | 2022-05-01 | 2.55 | 0.33 | 1.72 | 6.46 | 2.53 | 8.44 | 7.17 | 19.23 | 40.77 | 58.02 | 20.41 | 6.27 | 589 | 72,676 | 513 | 3,215 |

*Figure 1: Release Metrics*

# Complexity Metrics

The *Cyclomatic Complexity* of the project shows a sharp decline from versions 2.3.0 and 3.0.0, followed by a climb from the initial major release to the second minor release (3.0.0 to 3.2.0). After this, the complexity trends downward. The decrease between major versions is probably due to refactoring the code as seen between versions 3.2.0 to 3.5.0. However, after the new major version is released, the sudden climb might be due to adding features or fixing unforeseen bugs.

In the second graph, the *Ratio of Comment to Code* increases continuously across all versions. Good comments make the code easier to understand and easier to maintain. Too few comments can lead to confusions while too many comments may indicate poor structure or naming conventions. The consistency for ZXing's ratio likely shows the code is well commented from version 2.3.0 and got even better by the most recent release.

The *Max Inheritance Tree* (DIT) shows the maximum length from a class to its root. The deeper the tree, the greater the design complexity. The graph above shows that the complexity increased slightly between major versions, but the developers appeared to try and keep the code relatively simple.

In summary, the complexity metrics indicate the ZXing codebase is not overly complex and is probably clean and well documented.

# Object Oriented Metrics

The second set of graphs all focus on metrics tied to Object-oriented programming. The graphs appear to be correlated as each has a similar sinusoidal-like curve, trending upwards apart from a shallow drop around release versions 3.3.0 and 3.4.0. The top row of graphs includes the averages between versions for *Number of Instance Methods* (NIM), *Number of Instance Variables* (NIV), and *Average Number of Local Methods* and the second row of graphs show *Class Coupling* (CBO) and *Lack of Cohesion in Methods* (LCOM).

These metrics are often used to help detect anti-patterns and design smells such as Blog or Spaghetti Code. Looking at the CBO alone, it seems the application is gaining in complexity

"""

High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion. This metric evaluates the design implementation s well as reusability.

Since good object-oriented design requires a balance between coupling and inheritance, coupling measures focus on non-inheritance coupling.

CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is reuse in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a module is harder to understand, change or correct by itself if it is interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules. This improves modularity and promotes encapsulation. CBO evaluates design implementation and reusability."""

The LCOM graph shows an upward trend; this is an indication of an increase in complexity. An approach to resolve this is to split the classes, with high values, into multiple sub-classes.

The last three releases show a decrease in the LCOM metric. This could possibly be related to the decrease of NIM and NIV in the same releases.

The amount of coupling between classes increases from release 1.4.1 to 2.0. Even though the *Total Classes* graph shows a reduction from

1.4.1 to 2.0, the developers would have increased the number of links between existing classes.

However, it seems that the developers refactored the code at release

2.1 to improve the coupling. An increase in classes could have helped reduce the CBO value (possibly by moving coupled methods to a single class).

From release 2.1, coupling starts increasing again. To improve (reduce

the value), developers must promote modularity in the code.

# Volume Metrics

The average *Line Count* remained consistent over the years and then suddenly dropped slightly between the 2019 (3.4.0) and 2022 (3.5.0) releases. Despite this, the *Average Lines of Code* mirrored this behavior, instead increasing the number of lines of code for the most recent version. This is likely due to removing dead code, code comments, and blank lines, i.e., cleaning up the codebase. The *Number of Semicolons* decreased in the until the 3.3.0 release and then began to climb again. This is possibly due to refactoring and cleaning the code and then later adding more functionality. The next metric, *Average Number of Functions* also shows a tiny uptick toward the final release but is difficult to characterize its overall impact due to is relatively tiny variability footprint. The chart for *Total Classes* shows that when the product first arrived on Github in 2014 with the 2.3.0 release, it had close to 600 classes. Between that release and the 3.0.0 release, we can see a sharp decline in the number of classes and again between the 3.3.0 and 3.4.0 releases. It then begins to climb again. Too many classes can lead to more complexity, especially if they are dependent on one another or have high coupling. The *Total Lines of Code* and *Total Functions*  both increased at a higher rate than the *Total Files* which suggests the developers wanted to keep the files to a smaller number while still introducing more functionality, which inevitably leads to more lines of code. It seems there was a larger increase in functionality from version 2.3.0 to 3.5.0 but using the Mann-Whitney U-Test \figure{2}, it appears the changes were not statistically significant from one release to the next.

# References

1. Barcode Scanner - Android Apps on Google Play. Available at: <https://play.google.com/store/apps/details?id=com.google.zxing.client.android> [Accessed Oct 30, 2022]
2. GitHub - ZXing Project. Available at: <https://github.com/zxing> [Accessed Oct 29, 2022]
3. SciTools – Metrics. Available at: <https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have-> [Accessed February 02, 2017]