Introduction

Lottie is a library for Android, iOS, Web, and Windows developed by Airbnb that parses Adobe After Effects animations and renders them natively on mobile and on the web [1]. The source code for this project is on GitHub with over 33K stars and written in Java/Kotlin. The code was first released on Feb 1, 2017 and the latest version (5.2.0) was released on May 30, 2022 [2]. This project's source code was analyzed using the tool Understand [3] and 10 version releases spanning across five years. This analysis was performed in Oct 2022.

Analysis Results

Summary

Table 1 summarizes the software quality metrics used to analyze the project per release. Test classes and methods were excluded from the analysis. A two-tailed T-Test was performed on the lines of code between each release and the values were not statistically significant.

Release	Release Date	LOC p-value	CountLineCode	AvgCyclomaticComplexity	RatioCommentToCode	AvgMaxInheritanceTree	AvgCountDeclInstanceMethod	AvgCountDeclInstanceVariable
1.0.3	2/12/2017	-	4,443	2.12	0.04	1.63	3.81	2.28
2.6.0	8/4/2018	0.29	10,366	1.94	0.12	1.41	4.99	2.62
3.0.0	4/25/2019	0.79	11,723	2.02	0.12	1.40	5.13	2.82
3.3.0	12/2/2019	0.44	14,430	2.17	0.17	1.39	5.29	2.77
3.4.0	2/22/2020	0.95	14,542	2.18	0.17	1.39	5.32	2.80
3.5.0	11/8/2020	0.88	14,929	2.17	0.17	1.39	5.15	2.67
3.7.1	7/7/2021	0.44	15,256	2.19	0.18	1.39	5.22	2.69
4.0.0	7/28/2021	0.98	15,318	2.20	0.18	1.39	5.23	2.70
4.2.1	11/12/2021	0.99	15,637	2.22	0.18	1.39	5.22	2.78
5.0.1	2/21/2022	0.40	16,022	2.22	0.19	1.39	5.97	3.28
5.2.0	5/30/2022	0.97	16,121	2.23	0.19	1.39	5.98	3.28

Table 1a: Lottie Quality Metrics

Release	Release Date	AvgCountDeclMethod	AvgCountClassCoupled	AvgPercentLackOfCohesion	CountDeclMethodPublic	CountDeclClass	CountDeclClassMethod	CountDeclClassVariable
1.0.3	2/12/2017	4.00	7.61	20.73	322	100	36	24
2.6.0	8/4/2018	5.64	9.38	27.76	1,512	228	240	98
3.0.0	4/25/2019	5.86	9.36	28.55	1,654	218	276	100
3.3.0	12/2/2019	6.04	9.14	31.93	1,896	245	324	266
3.4.0	2/22/2020	6.09	9.14	32.06	1,908	245	328	266
3.5.0	11/8/2020	5.89	8.96	30.54	1,968	262	340	278
3.7.1	7/7/2021	5.97	9.00	29.97	1,982	261	344	278
4.0.0	7/28/2021	5.99	8.95	30.01	1,988	261	344	278
4.2.1	11/12/2021	5.96	9.02	27.47	2,026	265	342	286
5.0.1	2/21/2022	6.82	9.96	31.61	2,048	241	348	288
5.2.0	5/30/2022	6.84	10.06	31.74	2,052	241	350	288

Table 1b: Lottie Quality Metrics

The metrics analyzed were:

Complexity Metrics

1. **AvgCyclomaticComplexity** - The average cyclomatic complexity across all nested functions or methods. The lower the number the less complex the conditional logic.
2. **RatioCommentToCode** - The ratio of comment lines to code lines. The smaller the number, the more lines of code there are compared to comments.
3. **AvgMaxInheritanceTree [DIT]** - The average maximum inheritance tree depth. The higher the number, the deeper into the inheritance tree a class is and the more methods a class can inherit, therefore increasing complexity possibly.

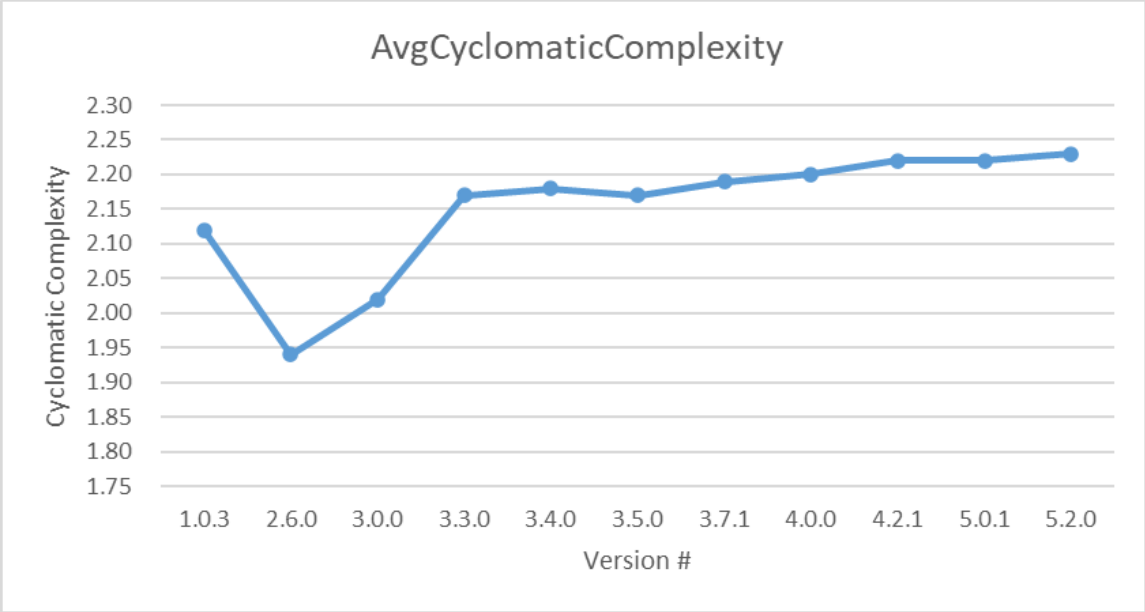
Object Oriented Metrics

1. **AvgCountDeclInstanceMethod [NIM]** - The average number of instance (non-static) methods .
2. **AvgCountDeclInstanceVariable [NIV]** - The average number of instance (non-static) variables .
3. **AvgCountDeclMethod** - The average number of non-inherited methods .
4. **AvgCountClassCoupled [CBO]** - The average number of class coupling between a class. The higher the number, the more coupling between classes which increases complexity.
5. **AvgPercentLackOfCohesion [LOCM]** - The average percentage of class methods that use a given class instance variables. A lower number means higher cohesion between class data and methods.

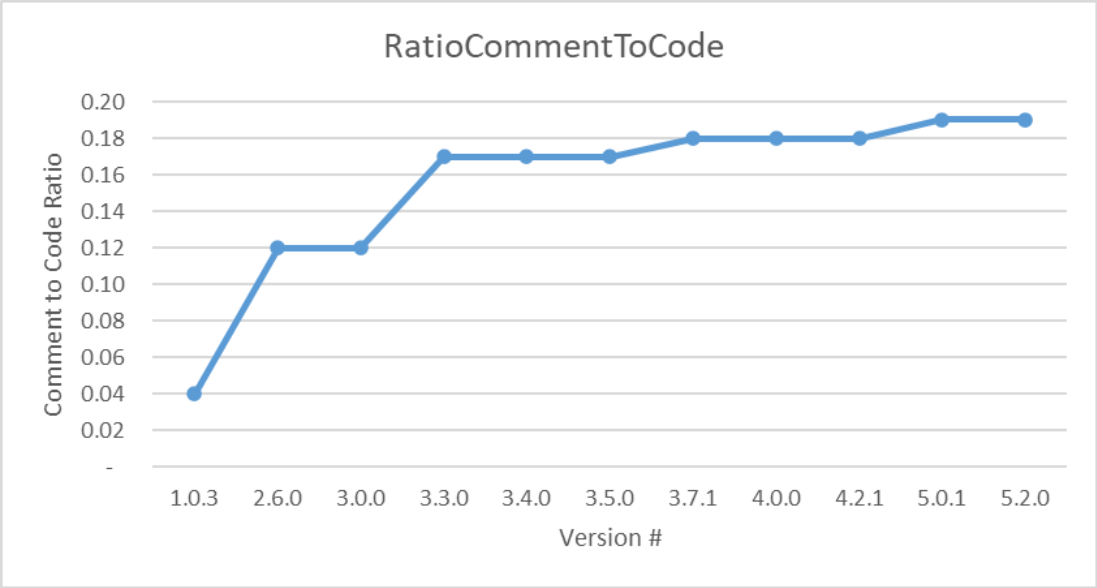
Count Metrics

1. **CountLineCode** - The total number of lines of code.
2. **CountDeclMethodPublic [NPRM]** - The total number of declared non-inherited public methods.
3. **CountDeclClass** - The total number of Java classes.
4. **CountDeclClassMethod** - The total number of static methods.
5. **CountDeclClassVariable** - The total number of static variables.

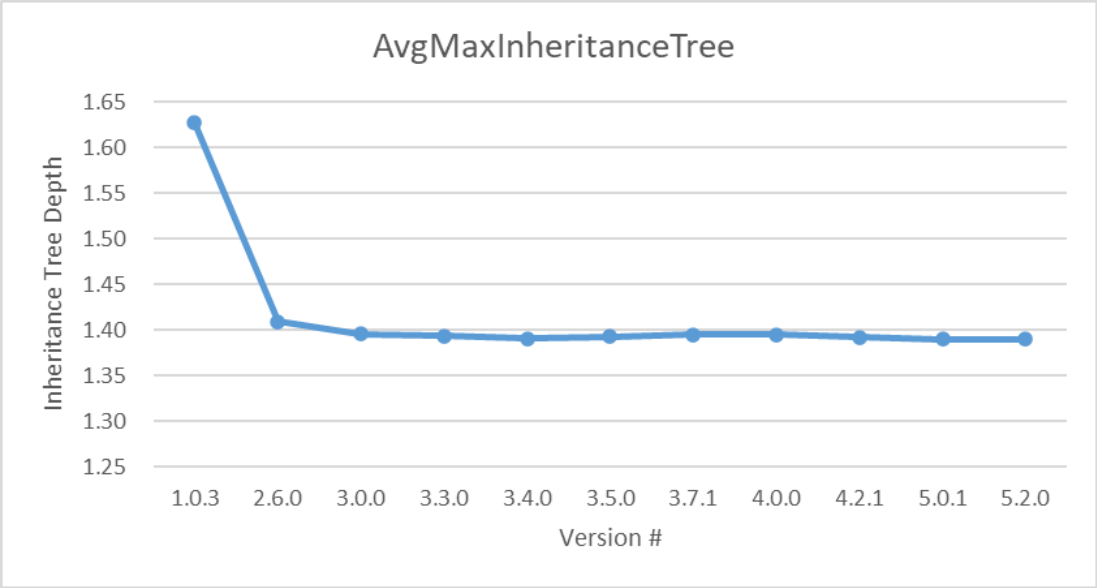
Complexity Metrics



The cyclomatic complexity for this project started off with complexity similar to the later releases, but then experienced a reduction in complexity in version 2.6.0 and 3.0.0. Overall, as the project evolved the complexity is trending up, but not by a large rate which is good.

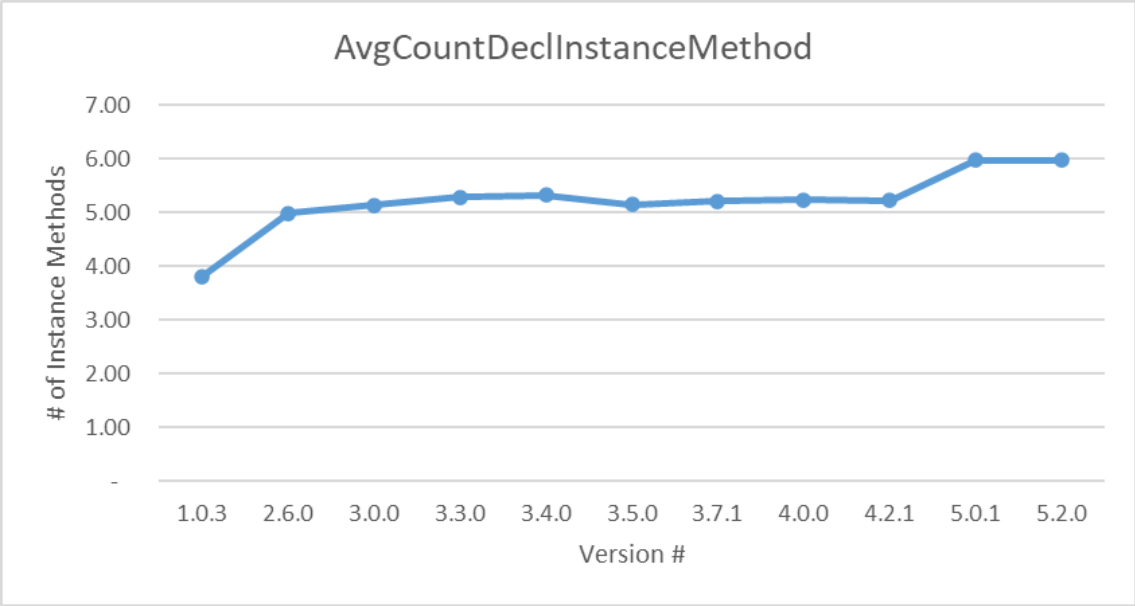


The ratio of comment to code is less than one meaning that there are more lines of code than lines of comments. The ratio quadrupled since the first release analyzed before leveling off at around 0.18 meaning much more comments were added in the first 4 versions analyzed of the project.

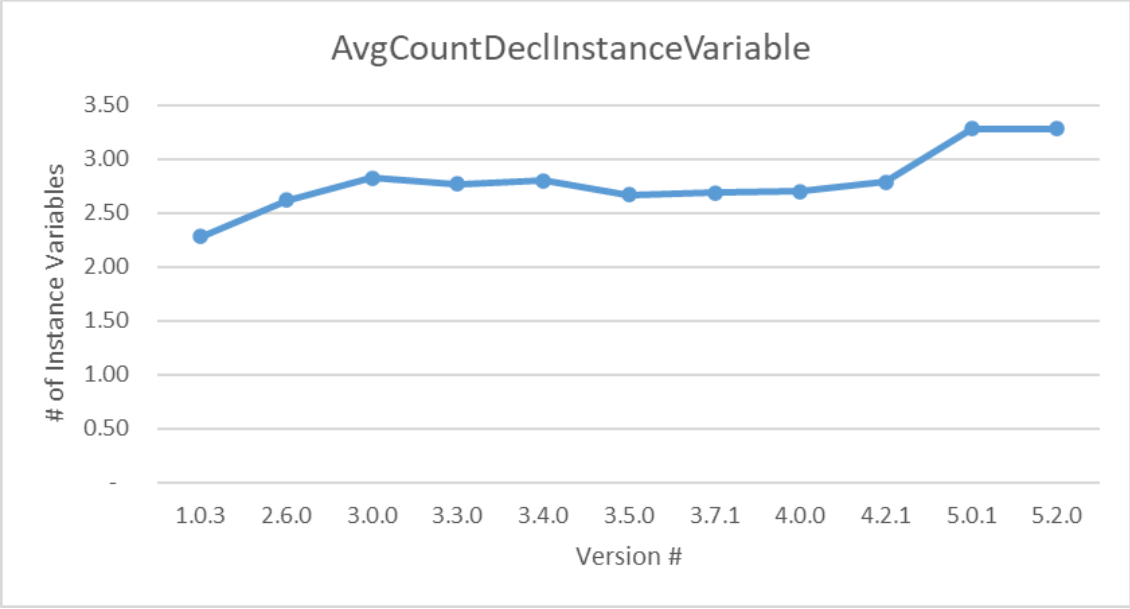


The maximum inheritance tree was the highest at 1.63 in the initial release analyzed and then remained around 1.40 throughout the rest of the project. This suggests that some refactoring could have been done to reduce the inheritance tree and since the average is over 1.0, this indicates that the project does inherit at least two levels deep from the parent classes.

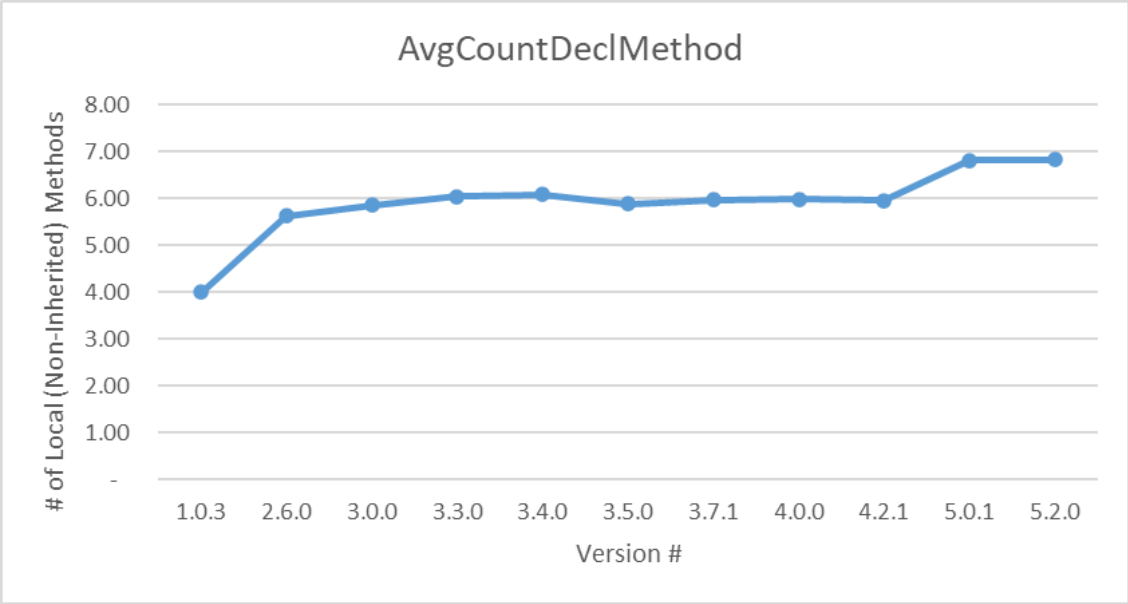
Object Oriented Metrics



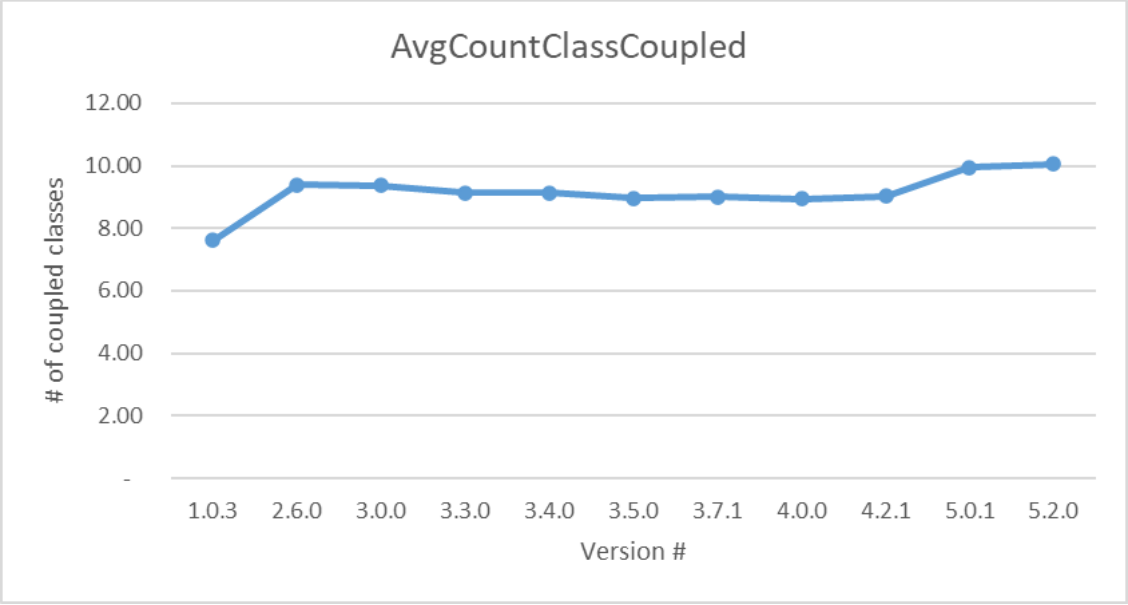
The average number of instance (non-static) methods for classes has an average around 5 for the majority of the project's life. The average increased from 4 to 6 across the analyzed releases. It is unclear if this is due to the increase in classes overtime or more methods being added to classes.



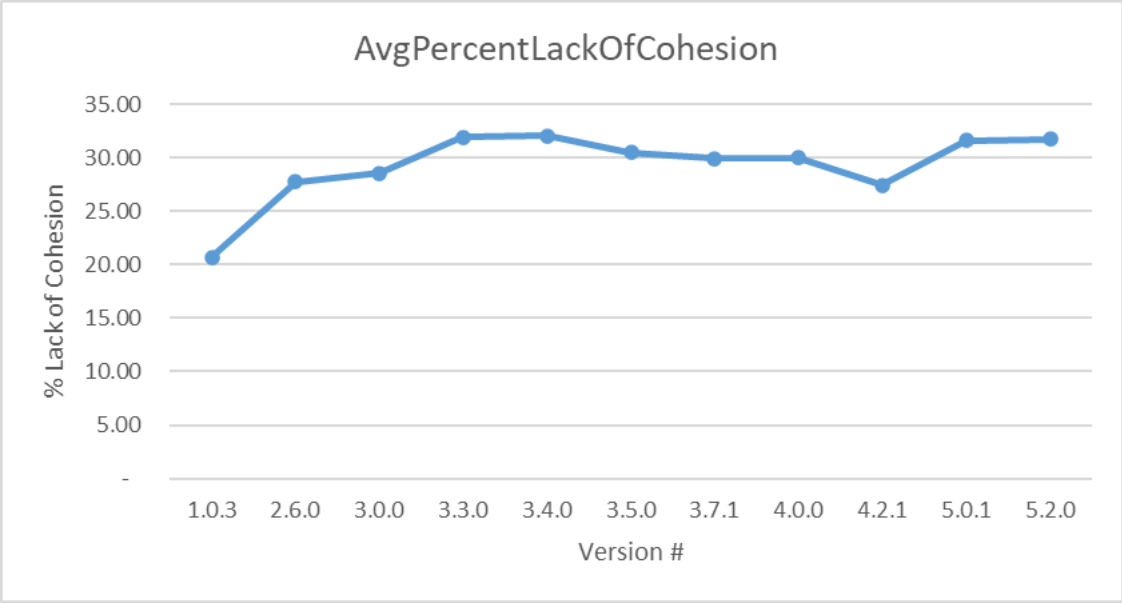
The average number of instance (non-static) variables for classes follows a similar pattern to the instance methods. The average is around 2.75 for the majority of the project's life. It is also unclear if this is due to the increase in classes overtime or more variables being added to classes.



The average number of non-inherited methods for this project is around 6. A dip in the average could mean that each class has more inherited methods, but that does not seem to be the case for this project.

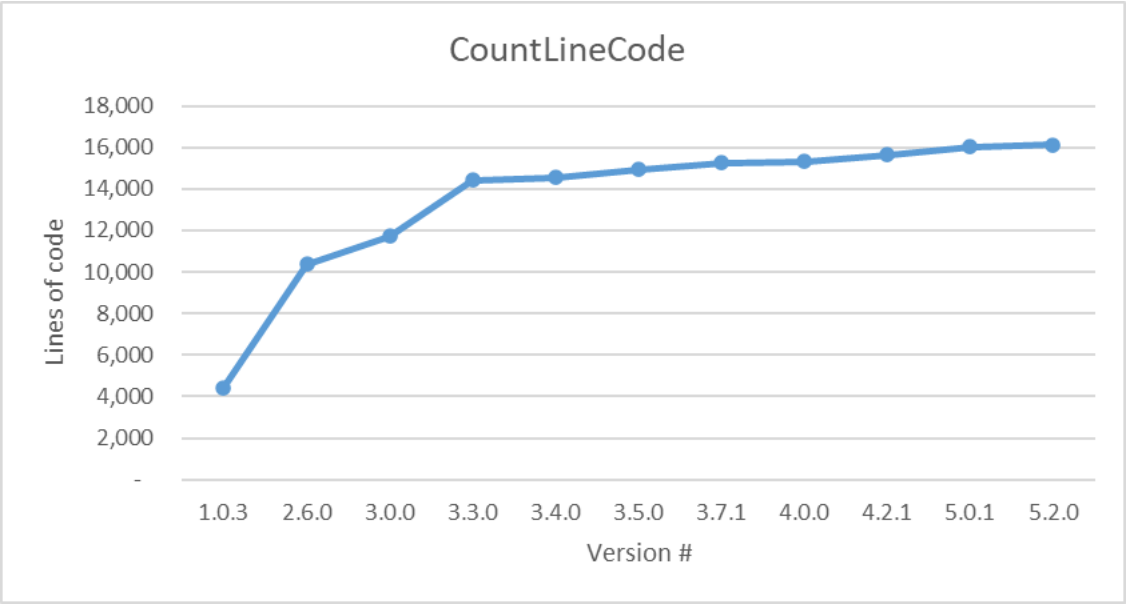


The average number of classes coupled between a class is around 9 for the project. This means that on average, a class depends on 9 other classes. For this specific project, 9 as the average dependency on classes might be acceptable. In general, the lower the number of classes coupled to one another the better. It is nice that the coupling has remained relatively flat throughout the project.

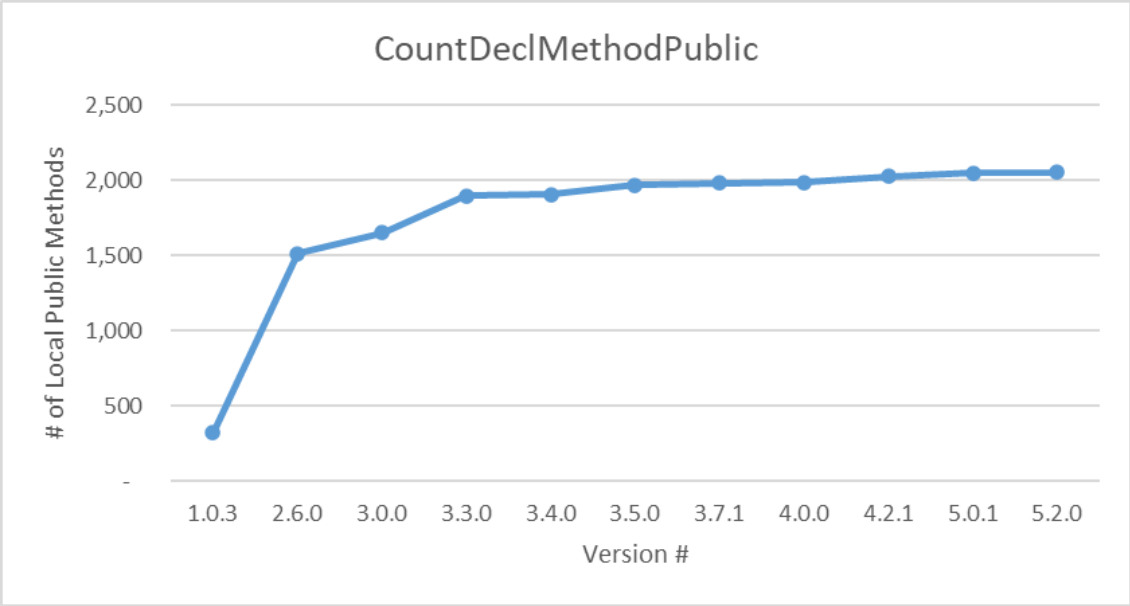


The average percentage of lack of cohesion started off at 20%, increased to a little over 30% and steadily decreased until version 5.0.1 where the % increased back to over 30%. The lower the percentage the more cohesion a class has according to this metric. The sudden decrease in cohesion from version 4.2.1 to 5.0.1 seems to follow the shape of the CounDeclInstanceMethod and CountDeclInstanceVariable shape which suggest 5.0.1 was a major update.

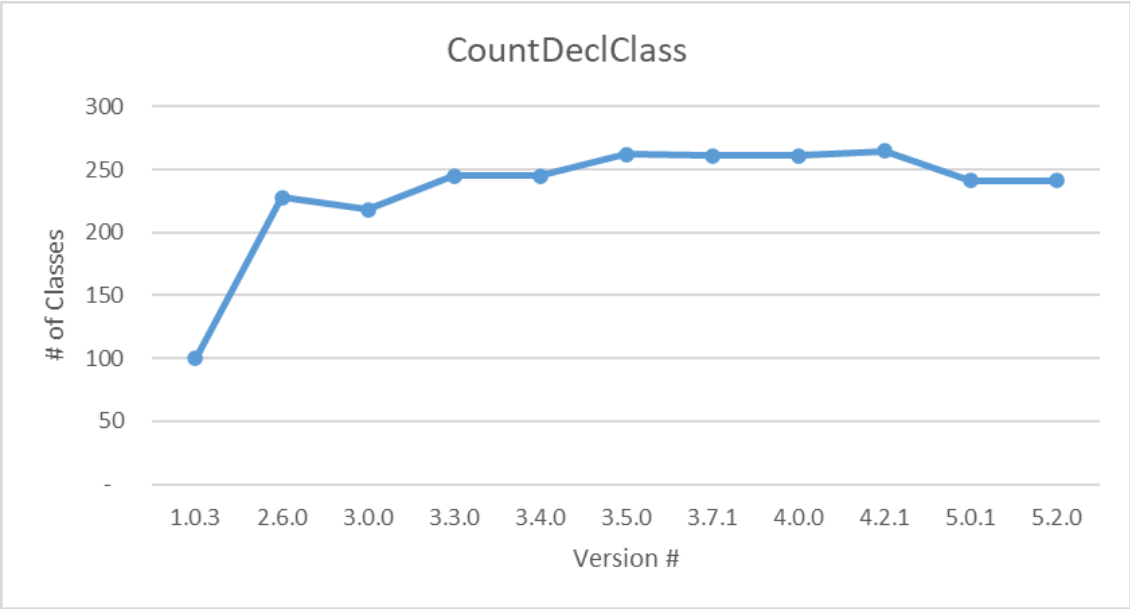
Count Metrics



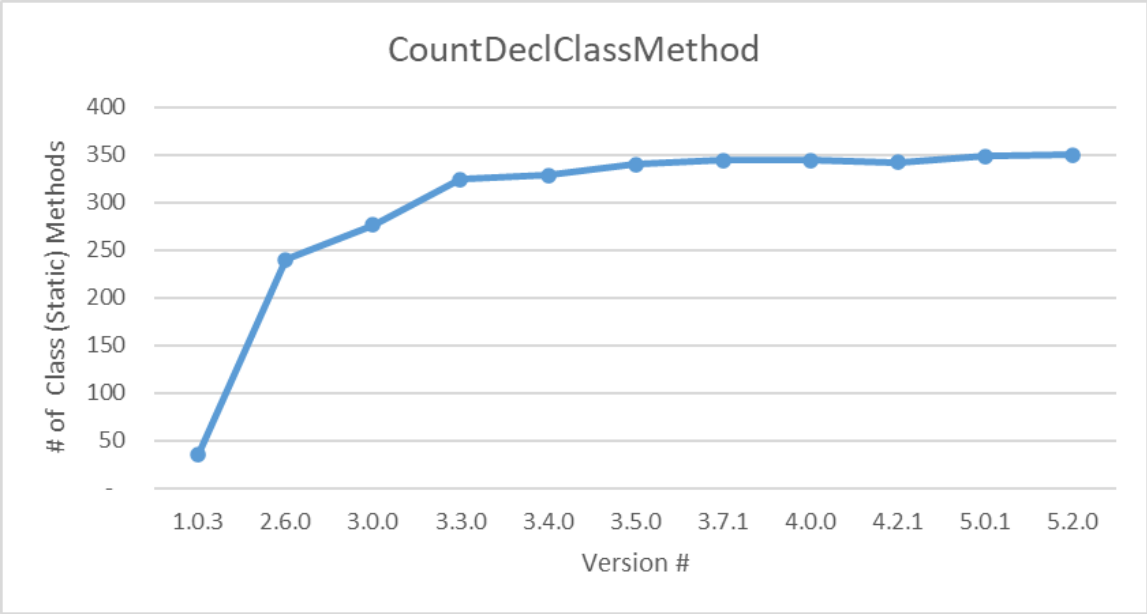
This graph shows that as the project evolved, more lines of code were added which makes sense as a project evolves and more features are added. The leveling off after version 3.3.0 suggests that around version 3.3.0 is when the code base stabilized and the core of the project was implemented.



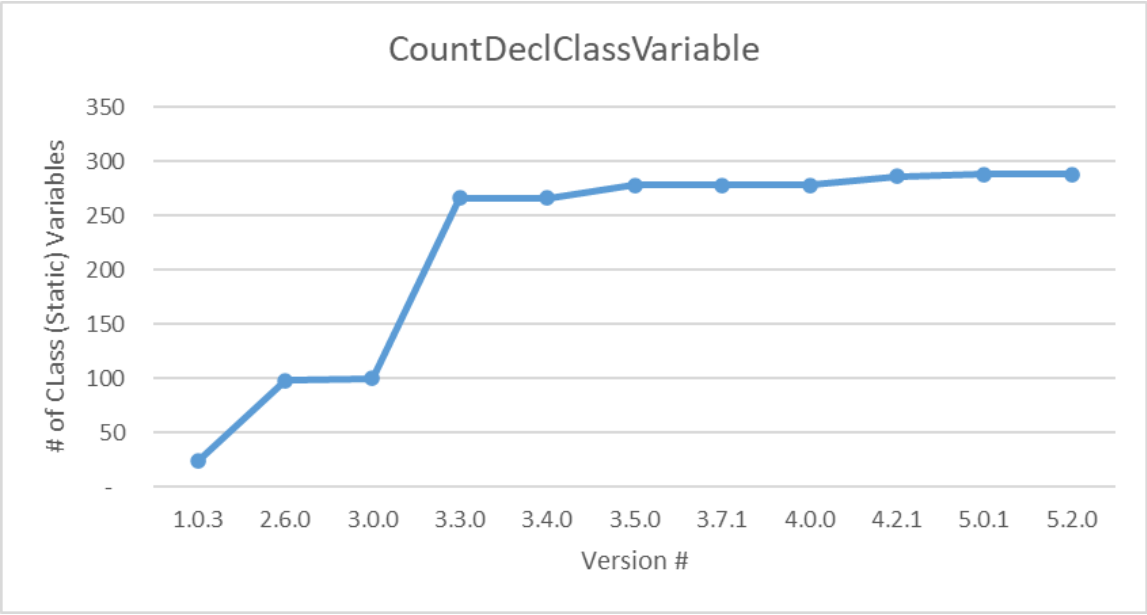
This graph shows that as the project evolved, more non-inherited public methods were added to the project and follows a similar shape to the CountLineCode graph. The number of public methods leveled off around 2,000 starting version 3.3.0 which again suggests that the code base began to stabilize around version 3.3.0 and the core of the project was implemented.



The graph shows the number of classes that the project has. After version 1.0.3, the number of classes decreased from version 2.6.0 to 3.0.0, increased, and then decreased again from version 4.2.1 to 5.0.1. This indicates that some refactoring might have been done from version 2.6.0 to 3.0.0 and again from version 4.2.1 to 5.0.1.



This graph shows that the number of static methods has been steadily increasing over the evolution of the project. Therefore the number of methods that do not need an object to be created from a class has been increasing. This might suggest that developers added utility methods over time.



The graph shows that the number of static variables have been increasing over the evolution of the project. There was a 2.5x increase from version 3.0.0 to version 3.3.0 before the number of static variables leveled off at a little under 300. This may be because in version 3.3.0 the developers made a decision to move more data into static variables. This means that objects of classes will be sharing the same data. This might make sense if the developers are declaring variables as “final” and creating constants.

Discussion

Patterns

Version 3.3.0 is around when the project matured

Based on the metrics *CountLineCode, RatioCommentToCode, CountDeclMethodPublic, CountDeclClass, CountDeclClassMethod,* and *CountDeclClassVariable*, I saw that the graphs leveled off around version 3.3.0. This to me indicates that the project has reached a mature state around this version and most of the core functionality that the library has been implemented. This makes sense as well since for almost 3 out of the 5 years this project has been in existence the version has been in the 3.0.0 version numbers.

Version 5.0.1 is when refactoring was done to reduce classes and move to methods and variables

Based on the metrics, *AvgCountDeclInstanceMethod, AvgCountDeclInstanceVariable, AvgCountDeclMethod, AvgPercentLackOfCohesion, AvgCountClassCoupled,* and *CountDeclClass,* I saw that these graphs had a similar shape, but the CountDeclClass decreased. This to me indicates that from version 4.2.1 to 5.0.1, refactoring was done to reduce the total number of classes and move some of the logic that the classes were trying to encapsulate into methods and variables of existing classes. This correlates with an increase in lack of cohesion and an increase in class coupling. Although the developers reduced the total number of classes, lack of cohesion and coupling increased which increases overall complexity and is something the developers should watch out for.

Mann-Whitney

	CountLineCode	Cyclomatic	RatioCommentToCode	MaxInheritanceTree	CountDeclInstanceMethod	CountDeclInstanceVariable
U-value	8632.5	425865	683344	10152	10132.5	10941.5
p-value	0.00001	0.851796	4.81E-24	0.00318	0.00288	0.0455
significant?	yes	no	yes	yes	yes	yes
Release	**Release Date**					
1.0.3	2/12/2017					
5.2.0	5/30/2022					

	CountDeclMethod	CountClassCoupled	PercentLackOfCohesion	CountDeclMethodPublic	CountDeclClass	CountDeclClassMethod	CountDeclClassVariable
U-value	8360.5	9934.5	10566	8191	9703	10836	10010.5
p-value	0.00001	0.00128	0.01428	0.00001	0.01865853	0.03318	0.0018
significant?	yes	yes	yes	yes	yes	yes	yes
	Release	**Release Date**					
	1.0.3	2/12/2017					
	5.2.0	5/30/2022					

The Mann-Whitney U test shows that all the metrics drastically evolved from version 1.0.3 to 5.2.0 over the span of 5 years except for cyclomatic complexity. Not a statistically significant difference for cyclomatic complexity. This test differs from the lines of code two tailed T-Test where the difference between each version is not statistically significant.

Conclusion

The Lottie library developed by Airbnb is a mature and widely used open source Java project. Metrics analyzed reveals that overall, the project has reached stability and change in complexity and other metrics is relatively minor. This stability began with version 3.x.x which makes sense because version 2.x.x were beta releases. The project evolution has been statistically significant since the early versions of the project with the exception of cyclomatic complexity which is neither good nor bad. With the 5.0.1 release refactoring has occurred and developers should be cautious that future refactors do not continue to increase complexity drastically. Future work could include analyzing test classes to see if they follow similar patterns as well as looking deeper into the source code for more insights.

References

[1] https://airbnb.io/lottie/#/
[2] https://github.com/airbnb/lottie-android
[3] https://www.scitools.com/features