Assignment 3

Group 19
DD2480 Software Engineering Fundamentals
February 2024

Project

Name: geometry-api-java
URL: https://github.com/Esri/geometry-api-java

The Esri Geometry API for Java enables developers to write custom applications for the analysis of spatial data. This API is used in the Esri GIS Tools for Hadoop and other 3rd-party data processing solutions.

Onboarding experience

Our first attempt at find a suitable project was the JSON iterator project in Java. The documentation and onboarding were both out of date and building the project locally turned out to be almost impossible. This was caused by the ancient Java version used. When we ultimately managed to build the project, we instead saw that hundreds of tests failed, causing us to decide to abandon the project and move on to geometry-api-java.

The onboarding for geometry-api-java was relatively pain-free. It uses Maven and is easy to build locally. There were some configuration issues relating to the relatively old 1.7 Java version in use, but those were related to IntelliJ rather than the project itself.

Complexity

As suggested, we used lizard (v. 1.17.10) to find five functions with high cyclomatic complexity. The five we initially decided upon were all methods of the Clipper class, which concerns calculations of clipping with geometric objects in 2D space. The output for the functions in question from lizard was as follows:

Function name	CCN	LOC	Tokens	Parameters
Clipper::clipPolygon2_	59	189	1305	3
Clipper::clipPolyline_	36	143	904	2
Clipper::splitSegments_	32	146	1181	2
Clipper::fixPaths_	17	99	721	0
Clipper::clip	23	86	589	4

They all have high cyclomatic complexity (≥ 17). In most cases, we got the same CCN when manually counting (see more detailed accounts below).

Generally, the methods are both highly complicated and long. They involve multi-layered nested decision statements and are difficult to understand, owing both to a critical lack of

documentation, confusing naming schemes, and the complexity of the code base overall. There are many classes that in turn inherit from other custom classes, and so on.

Below is a more detailed description of the methods and their cyclomatic complexity when counting manually compared to lizard.

Clipper::clipPolygon2_

Manual CCN: 59
When counting the first time we got drastically different results. After recounting we got the same result, which turned out to also be the same that lizard gave. This was the longest function we tested, and also the one with the highest complexity. The purpose is to modify polygons in ways that involve resizing them or splitting them into multiple ones. The method lacked documentation for the different possible outcomes. Asserts are taken into consideration, exceptions are not present in the method.

Clipper::clipPolyLine_

Manual CCN: 36
Both manual counters and lizard got the same answer. Being 143 LOC, the function is complex and long. Its purpose is to calculate the resulting polyline from clipping an existing polyline with a 2D rectangle. The complexity mainly stems from the extremely detailed and fundamental implementation of geometry in the ESRI Java library, as well as deeply nested decision statements. Complexity could be reduced by separating logic into other methods (which would also increase readability), but the total cyclomatic complexity would not necessarily decrease. The documentation is almost non-existent.

Clipper::splitSegments_

Manual CCN: 30
When using Lizard to count the cyclomatic complexity we instead got 32. The difference in complexity is probably based on the unclear definition of cyclomatic complexity. The high complexity is probably due to the long code, around 180 lines. The purpose of this function is a bit unclear as it has no documentation (except very few and obscure comments in the body) but the main purpose is to split a line segment along a given axis. Asserts are taken into consideration, exceptions are not present in the method.

Clipper::fixPaths_

Manual CCN: 15
The cyclomatic complexity achieved when counting manually differed from the one from lizard by 2, lizard giving a CCN of 17. The difference probably occurred due to a different interpretation/understanding of what CCN is. For example, if for-loops should be counted or not. The method is quite long at 89 lines of uncommented code and whilst the operations performed aren’t that complex in and of themselves they are deeply reliant on other parts of the code and their own data structures which makes it difficult to quickly understand their purpose.

The cyclomatic complexity mainly comes from the total length of the method and not necessarily its functional complexity.

Clipper::clip

Manual CCN: 22
When counting the first time, we got the same amount of edges before the case statement as well as nodes. However, after the case statement, one member had 7 nodes and one member had 8 nodes. The difference in edges was 2, which was later resolved to be 17 edges. The differences occurred due to exit and start of the program. Also, both members used the wikipedia formula instead of the one used in the lecture, which is also a reason that the resolution was an easy fix. The manual CCN is very close to the one given by lizard which was 23.

Coverage

DIY branch coverage

The DIY branch coverage for the five methods can be found in the `branch-coverage` branch. Link

The implementation can be seen by running: `git diff master 31-DIY-branch-coverage`

1. What is the quality of your own coverage measurement? Does it take into account ternary operators (condition ? yes : no) and exceptions, if available in your language?
   We manually defined the separate branches in the methods by identifying each decision statement (if/&&//while etc). We then create a hashmap object for each method. The numbered of covered branches is printed to stdout after each test case has been run for each function where the DIY coverage has been implemented. These reached branches can then be combined to see the overall coverage for all the tests. It does not take into account ternary operators and exceptions behave as if the code had terminated at that point.
2. What are the limitations of your tool? How would the instrumentation change if you modify the program?
   Combining the results is a manual process, as is the addition of new branches. This means that every time you change the code, you would have to add the relevant branches in the code and re-run the coverage testing as well, which likely becomes very tedious. Having more automated tools would probably be beneficial in the long term, and reduce the risk of making mistakes when determining what is a new branch and what is not.
3. If you have an automated tool, are your results consistent with the ones produced by existing tool(s)?
   Closely, but not exactly. There are some differences between IntelliJ's reporting of which lines have been executed and what must have been executed to provide the results in the more manual tests. There is however not any branch that has been reported as executed by one of the coverage tools but not by the other. This points to

the fact that any issue might be related to defining and separating branches, rather than the logic used to document coverage.

IntelliJ Coverage

It was quite easy to use the built-in coverage tool in IntelliJ. The entire team was already using IntelliJ for the previous assignments, which made the barrier to using the coverage tool close to zero. Since it is built-in into the IDE it is well integrated into the build environment and has the same documentation pages as the rest of it. One limitation is the difficulty in knowing exactly which methods cause what coverage. Also, displaying the coverage report per method (as opposed to by class) does not appear to be possible.

Coverage improvement

Because of the existing high coverage and the very complicated nature of the Clipper methods, we partially opted for improving coverage for a set of other methods that also had high cyclomatic complexity, rather than the five studied until this point.

We recorded the old coverage by running every test class in the project and then creating a report using IntelliJ. As stated previously, the coverage could not easily be filtered by method, but the process was otherwise very convenient. The old coverage report in HTML can be found in the repository on the old-coverage-report branch [Link].

Report of new coverage: Link

Added test cases can be found by running `git diff master 23-tests-for-improved-coverage`

Detailed descriptions of coverage improvement by method can be found below.

Line::_intersectLineLineExact

Number of test cases: 6. Old coverage: 0%. New coverage: 100%.

The method _intersectLineLineExact takes as argument two lines in 2D space and returns the number of exact intersections of their endpoints (0, 1, 2). It had a high CCN of 29 and zero test coverage.

Tests were written to assert the expected results (i.e. the returned integer value of the number of intersections), but also to assert the additional parameters. This was further used when refactoring code to ensure that functionality remained.

By creating a separate TestIntersectLineLineExact test class and adding six test cases for different setups, line coverage went from 0% to 100%, according to IntelliJ.

PointInPolygonHelper::_isPointInPolygon

_isPointInPolygon is completely untested. Tests could be to have polygon without points, a polygon with a point outside and a polygon with a point inside.

Clipper::_splitSegments

To improve branch coverage for splitSegments_ test cases involving a normalized distance between two points is 0 and 1 respectively.

JsonStringWriter::appendQuote_

The coverage for the appendQuote_ function was improved by implementing tests for more of the potentially problematic chars which are escaped by the function, but were not tested in the upstream branch.

SimpleRasterizer::addEdge

The coverage for the addEdge() function was improved. This was achieved by creating tests that specified certain edge cases. One of which being when both x1 and x2 are outside of a specified width. I.e. if the width is 1 and x1 and x2 were both greater than 1, say 2 and 3, then they would be set to the width. This error handling was done in the code but never met during any of the test cases, as such one was added.

Another case was when x1 was a very small number. (x1 < -0×7fffff) Similarly to the aforementioned test, another one was added that tried to add an edge where x1 < -0×7fffff, thus triggering the if-statement.

Clipper::Clip

Number of test cases: 4.
Old coverage: 49%
Coverage after 2 tests: 54%
Coverage after 4 tests: 62%
⇒
New coverage: 62%
The coverage was improved by these test functions:

- testClipEmptyGeometry: This test asserts that an empty geometry remains empty after clipping with a valid extent. This checks the clip function's handling of empty inputs.
- testClipWithEmptyExtent: This asserts that clipping a non-empty geometry with an empty clipping extent results in an empty geometry. This confirms the function's behavior when the clipping boundary is invalid or undefined.
- testGeometryFullyInsideExtent: This one asserts that a geometry fully within the clipping extent remains non-empty after clipping. This makes sure that the clip function in a right way identifies and preserves geometries in full within the boundary.
- testEnvelopePartiallyIntersectingExtent: Asserts that an envelope partially intersecting the clipping extent results in a non-empty geometry. This is a scenario that tests that if we clip a geometry that is only partially in the clipping boundary, the intersecting portion is not removed.

The clip method is designed to clip a geometry object with a specified envelope (a rectangular area), so only a portion of the geometry (a certain area) is interesting. The method handles different types of geometries, for example, points, envelopes, multi-points, polygons, and polylines, where each of these requires specific logic to determine the clipped result.

By creating a separate test class and adding four test cases for different setups, line coverage went from 48.65% to 62.16%.

Plans for refactoring

Line::_intersectLineLineExact

_intersectLineLineExact was refactored primarily by extracting the logic for updating the line parameters and the array containing the 2D points where the lines intersect into a separate method. This logic was unnecessarily repeated multiple times. The cyclomatic complexity was reduced from 29 to a total sum of 17, split over three methods. A reduction of ~41%

Clipper::splitSegments_

The plan is to split out the eg `n_1 == 1` and `n_2 == 1`, many of the steps are also the same so code duplication could be avoided. Additionally, the adding of vertices (604–629) could also be extracted. This will lead to lower CC but will probably also make the function easier to understand as the method call can be seen as documentation. However, the methods are too complex which means that they will not be pure. This refactoring was done and led to a 50% decrease of complexity, 16 instead of 32.

Clipper::clip

The clipping operation can be made into different methods based on the geometry type, and the main clip function can be made to route
instead of implementing all the logic for these aswell.

The refactoring would be to make each geometry type (Point, Envelope, MultiPoint, Polygon, Polyline) into different methods, and each method would handle the clipping logic specific to that geometry type.

Many of the conditional blocks can be made into separate methods which would make the main logic more straightforward.

The refactoring would be to take blocks that check the geometry attributes ex if geometry is empty or extent is empty, and then instead create methods for them ex. isGeometryEmpty(Geometry geometry) and isExtentEmpty(Envelope2D extent).

For operations that are repeated across different geometry types, like checking intersection with the extent, we can create methods that can be reused.

JsonStringWriter::appendQuote_

While the function is being reported by lizard as having high CC, in reality it is quite readable. Since its body is mostly a switch-case, one of the best ways to reduce the CC would be to move the switch-case to its own function. This would make the function somewhat easier to test. To reduce it even further one could move all cases into their own functions, but that would likely not improve neither performance nor readability.

SimpleRasterizer::addEdge

The method is in reality very well formatted even though it has a high complexity. The code is very readable and simple, not relying on many outstanding variables or data structures. One thing that could be simplified to reduce the complexity of the function would be to merge a few of their edge case handlers into a function and apply it to an Array or similar made from the variables.

More concrete there are four test cases where the check if (x1 < -0×7fffff), (x1 > 0×7fffff), (x2 < -0×7fffff), (x2 > 0×7fffff).

So by making a function that checks if a number is between -0×7fffff and 0×7fffff and performs corrections and then applying it to the array [x1, x2].

Self-assessment: Way of working

The team is in the state of “In Use” according to the Essence standard, and is on a good path to the next stage, which is the “In Place” state.

The conclusion can be drawn because the team is working well together as one unit, with 2-3 hour coworking sessions 2-3 times a week, as well as using the tools and standards set in place regarding commits and pull requests.

The group members are also open about their issues and what they need help with,

as well as directing each other and assessing work with pull requests.

The team has also grown well and through many discussions on Discord, they have become closer.

The team is on a good track to the next stage, and the following points are reached:

- The team consistently meets its commitments.
- The team continuously adapts to the changing context.
- The team identifies and addresses problems without outside help.
- Wasted work and the potential for wasted work are continuously identified and eliminated, through pull request feedback and constructive feedback.

However, to fully reach the next stage, the team would need to do the following:

- Effective progress is being achieved with minimal avoidable backtracking and reworking, by pair programming more and collaborating.

Some group members have utilized pair programming, but we are still backtracking a bit and reworking solutions. We will most likely reach the next stage in the next programming assignment.

The group is good at splitting issues into smaller parts and working on larger problems together. However, some parts feel like “small” issues, for example, writing tests for a method each, but these tests took each person many hours to write because the existing code was confusing. By working together, the team would be much more likely to solve this issue faster. The idea would be to start on issues together, and when the foundation is laid go and split up the work.

Overall experience

What are your main takeaways from this project? What did you learn?

While cyclomatic complexity is an indication of whether a function is needlessly complicated, it does not provide the whole picture. In some cases the actions performed by a function are dependent on each other in ways that make refactoring difficult. (This could also indicate that the function should have been refactored earlier, but it varies from case to case). It becomes especially difficult in open-source projects that have limited documentation. Continuously identifying large functions and refactoring them helps keep the code maintainable, which combined with using javadoc or similar tools makes the project much more approachable for new contributors.

We learned a lot about taking on a completely new, large project and dealing with an unfamiliar code base.

Statement of contributions

- **Everyone**: Writing the report. Creating issues, choosing the project, planning. Complexity and coverage reports.
- **Björn Thiberg**: Manual CCN counting for clipPolyLine *and clipPolygon2*. DIY coverage for clipPolyLine_. Improved branch coverage and refactoring for Line::_intersectLineLineExact.
- **Eric Johansson**: Wrote the DIY test coverage for the fixPaths_ method and improved the coverage/wrote the refactor plan for the addEdge function.
- **Sam Khosravi:** Wrote tests, DIY complexity and refactoring plan for clip method. Counted complexity for Clipper::splitSegments_ and Clipper::clip.
- **Sam Shahriari**: Counted CC for SplitSegments and ClipPolyline. Wrote DIY branch coverage for SplitSegments. Wrote test cases for splitSegments *and isPointInPolygon (in total 4 test cases). Wrote and carried out refactoring for splitSegments* which halved CC.
- **Tore Nylén**: Counted CC for clipPolygon2 *and Clip, implemented DIY coverage for clipPolygon2*. Wrote test cases and refactoring plan for JsonStringWriter::appendQuote_.