Final Project Report

Software Reengineering

Van Muylder Ben & Geeraert Lander

May 29, 2019

0 Introduction

0.1 Context

This project is part of the course 'Software Reengineering' at the University of Antwerp. It serves (to quote the assignment) "to demonstrate that you indeed acquired the range of principles, techniques, and skills that are currently being used for reengineering", and such we will need to (and we'll again quote the assignment) "restructure an existing software system to prepare for a given suite of new requirements".

0.2 Do reengineer, do not implement

As you can read in the above part, we are required to prepare the software system 'for a given suite of new requirements', hence we are not required to implement these features, as is confirmed by yet another excerpt from the assignment: 'the extra functionality as such needs not be implemented; but [...] the existing design must be adjusted in such a way that adding the new functionality becomes a [...] piece of cake".

We want emphasize that we will **not** be implementing the features as they were noted in the assignment (those being: having the functionality to have separate shapes for data points, and having the functionality to read shapes for data point from a database).

0.3 Problem at hand

The problem which we should try to grasp is the following: a hypothetical customer wants us to refactor the existing software library 'JFreeChart'. To be more precise, he wants us to refactor the library as to allow for functionality which the library currently does not have. This functionality (as mentioned before) is a) to allow for separate shapes for data points and b) to allow for the loading of shapes from a database.

1 Preparation

1.1 Scope

The assignment is directed at a very specific part of the library. While there is no functionality to explicitly give each separate datapoint a specific shape, there is already functionality for datapoints to have shapes (which was to be expected). While there are very much aspects to this library, many of which could perhaps require their own refactoring, we will only focus on the functionality concerning shapes. Throughout the report we might refer back to this limit as our scope.

2 Design Recovery

To understand what should be changed in the project, we must first understand how the project works. Since our scope is the functionality of shapes, we must thus try to work out what the role of these shapes is, and how precisely they work.

To try and understand the working of the project, we used two methods: simply digging through code and using project visualizers. One such visualizer is Gource. Gource in general can be used to give a visual representation of how a repository has evolved through its development, but the final image (which can be interacted with) shows us a general overview of the project structure (see Figure 1). Such we can also see that there is a clean symmetry between the upper branch and the right branch of the project, which represent the source and the test parts of the project. For most classes in the source, a representative class could be found in the tests.

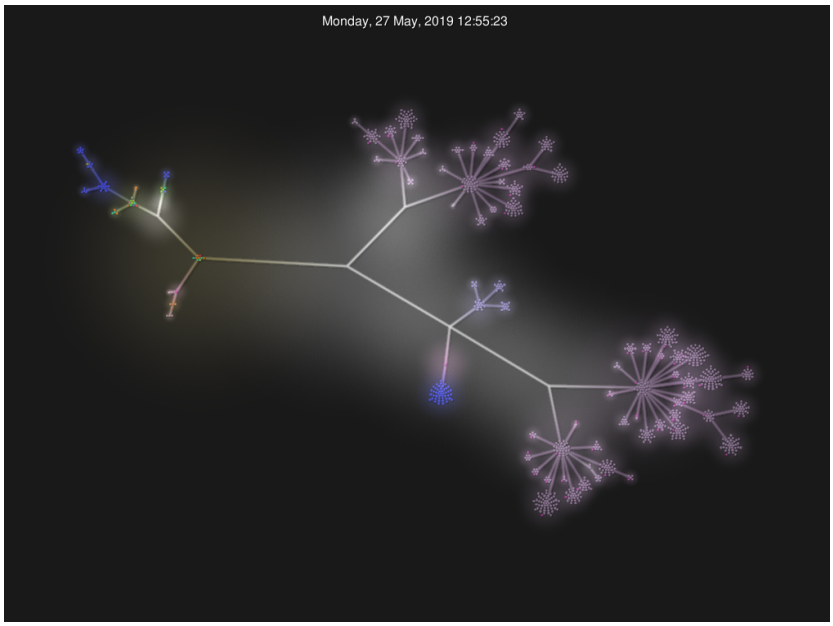

Figure 1: AbstractCategoryItemRenderer before any refactorings

2.1 Analysis of the current design

We'll be honest and upfront in saying that we won't have a whole explanation of how this library works, but rather a simplified view of happens in general, concerning to our scope (with possible deeper insights in those aspects which need more explanation).

A shape can be used for many things, but with respect to the assignment; it's used to represent a datapoint on a plot. A datapoint can be a point of its own, or it can be part of a set of datapoints (which we mostly refer to as a serie). This serie might represent a lot of things such as just simply random points, points of data which have been imported, points which represent a (mathematical) function...

However, purely in data, a shape does not have not much meaning. The shape itself is meant so we, the user of the library, can see these points on a screen. And when does this happen? When we export our plot (and data) to an image (for example).

This is where our overview of the workings begin. When we want to export to an image, we'll want to output some data to a buffer. The output comes from a draw function which is a central part of the core functionality, which in itself calls the draw functionality of the Plot interface (which is implemented by XYPlot and CategoryPlot among others...). These draw funtions do a lot of things, but one thing in particular: they render data items, or at least, set this process into motion. XYPlot for example has its own render function, which then in itself calls to the drawItem function of the renderer for that class instance (which would be an XYItemRenderer, which is in itself an interface, but is implemented in a lot of derived classes such as XYLineAndShapeRenderer). The drawItem functionality of XYLineAndShapeRender works with different passes: a first pass which draws the background, and a second pass which draws effective data items.

We have now reached the point where we will effectively have interaction with shapes. For simplicity we'll say that the draw function of these classes effectively *draws* the shapes somewhere. But to do this, the function needs a shape to draw. They do this by calling the getItemShape function of AbstractXYItemRenderer or AbstractCategoryItemRenderer (which both inherit this function from AbstractRenderer, and which both are extended by lots of other classes, such as XYLineAndShapeRenderer). This function provides and effective shape which the draw function then can use to *draw*.

getItemShape is special in 2 ways. The first way is in that it will always return the same shape for data items of the same serie (because of the deeper implementation of this function). The second way is that it is called in many other classes (which are thus extensions of AbstractXYItemRenderer and AbstractCategoryItemRenderer); unlike the function setSeriesShape (and setItemShape, which is a function which doesn't exist (yet)).

The function to set a specific shape for a serie (also part of AbstractRenderer), is never used in the whole project. But plots still need shapes for their data points? This is where default shapes come in play. Classes can set a shape by either changing the value for the default shape in the constructor, or by setting a shape for the series legend and then later on requesting the shape of the data point via the shape of the legend of that serie.

3 Design

3.1 The initial idea

When figuring out how the current implementation of the library works, we noticed one thing in particular. While the library does not have the functionality which is requested by our metaphorical customer, it still possible without the need for grand refactors and restructurings of the project.

For example: in the above part we spoke specifically of the function getItemShape, and not of the function getSeriesShape. This is because getItemShape is meant to return a specific shape for each separate datapoint, while getSeriesShape is meant to return a shape for each separate serie. In practice, the second parameter of getItemShape (which would denote the specific point of a serie for which we want to get a shape) is completely ignored, after which getSeriesShape is called.

getItemShape in this case serves purely as a passthrough function, where it is explicitly mentioned in the documentation for the function that if other behaviour is required, the users should override the existing function. This however, doesn't seem to happen anywhere in the project and thus we can safely say that getItemShape effectively is the same as lookupSerieShape (which in itself calls getSeriesShape).

Because of this, in our opinion, the project does not need refactoring to allow for the requested features, not even to make the implementation easier, as implementing it would only require the getItemShape function to be overridden correctly. One could even write an implementation for getItemShape where they perform a specific database query (based on the row and column parameter of getItemShape) to request a specific shape from a database.

Nevertheless, we'll try and find more opportunities to restructure the project to allow for more ease of use.

3.1.1 Pulling away the shape functionality

While we think that the shape functionality itself does not need changing to allow for the requested features, we do think that it is not easy for people who are not familiar with the project to effectively find where the functionality of the shapes resides (which is in the AbstractRenderer class).

Because of this we propose to pull away this functionality from the AbstractRenderer class, and put it inside of it's own class (which we will call ShapeManager) and make AbstractRenderer extend from it, simply to provide more clarity and to make the functionality easier to find. As such, the following are diagrams of how the current implementation of AbstractXYItemRenderer

is and how the current implementation of AbstractCategoryItemRenderer is.

We would also pull away all functionality which was needed by shape functionality, mostly function relevant to listeners., as well as the clone and equals function (although their functionality had to be changed a little bit as they would not need to copy/compare the attributes which had not been pulled away from AbstractRenderer). This however was only decided after our initial feature extraction had been performed and served mostly to avoid extra duplication.

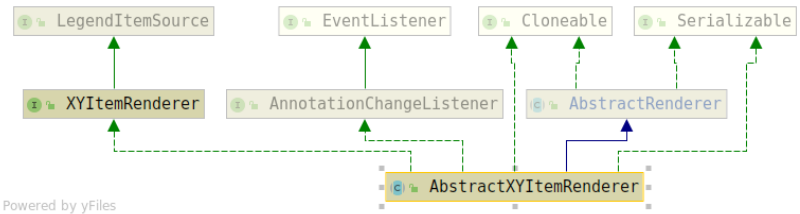

Figure 2: AbstractXYItemRenderer before any refactorings

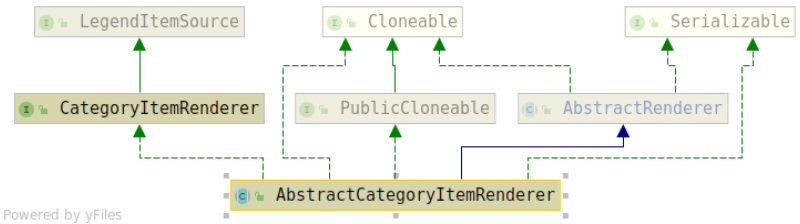

Figure 3: AbstractCategoryItemRenderer before any refactorings

What is important to note from these figures, is that AbstractRenderer (the class where the shape functionality resides) has not further dependencies. This is what will change in our proposed design. As such, the following are diagrams of how our proposed implementation is for AbstractXYItemRenderer and AbstractCategoryRenderer.

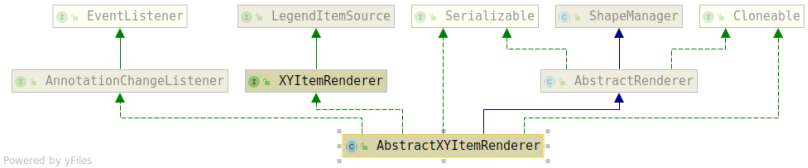

Figure 4: AbstractXYItemRenderer after refactoring

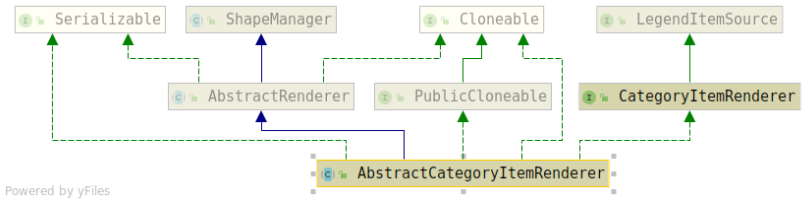

Figure 5: AbstractCategoryItemRenderer after refactoring

To reiterate; we want to perform this refactor purely for the sake of clarity, to help our (hypothetical) customer with figuring out where he should start with the implementation of his new functionality.

3.1.2 Risks

Since we simply want to pull away the functionality, and put it in a class from which the original class will extend, we effectively make sure that our original class doesn't change (since a compiled version will use flattened versions for classes, having the original class extend from the class with the functionality, makes it so that our original class (which is AbstractRenderer) keeps the functionality which we pulled away).

As such it fairly clear that there aren't much risks involved with performing this refactor. Nevertheless, we should at least a) check with an IDE whether we have created errors and b) rerun the existing test suite. If the test suite fails, there is a problem. However, if it succeeds this can mean either that everything still works, or that the existing test suite is not capable and produced a false positive. Nevertheless, we'll assume that we started with a stable version of the library, which has a capable test suite.

Then there is also the fact that we would literally copy some functions and with that effectively caused some duplication (as we will show later on as well). After evaluating that we our suspicions were true, and that we would indeed create extra duplicates (see further), we assessed if it were possible to extract those functions which caused the duplication. This seemed to be a viable option, thus we planned to remove some more functionality from AbstractRenderer and put it in our ShapeManager.

3.2 Other possibilities

Now that we know that we don't need to refactor the project **to allow for the requested funtionality**, we can go looking at what can be optimized in the general sense of refactoring. But this raises another problem. Since we localized the current functionality (and the place where the future functionality would or could be placed) of the project to being a very precise spot, the effective code

scope of this assignment has been greatly reduced. Either we try to refactor the functionality which we plan to extract from AbstractRenderer, or we expand our scope.

After some hefty discussing, we agreed that expanding our scope, would make the scope drift too far from the assignment. Thus, we'll initially extract the shape functionality from the AbstractRenderer, after which we look how this extracted functionality can be further refactored.

4 Management

Due to the nature of our workflow, you might already have guessed that there is not much planning and managing to do on with our current plan (which turned out to be true). The greater part of what we did was mostly discussing whether what we were doing was the right thing to do.

In other cases using a Ghantt or Pert chart might be a useful tool, but in our case the work which has to be done should only take a mere minutes, since it's mostly cut and paste work. Nevertheless this doesn't mean that we shouldn't have to check the changes that we made.

As such, we would also have to asses the existing tests for the system and decide whether or not we would require new/separate tests for our changed components. Initially we discussed that there wouldn't need to be additional tests because of the nature of our refactor, but we nevertheless planned to write (or at least create) a separate test class which would test some specific features of the ShapeManager. Yet once again, we decided that this would need no specific planning as we thought the work would only be a matter of copying and pasting (and some slight additions), which it indeed turned out to be.

5 Restructuring

5.1 Refactoring, pass1

Our plan (as is described above) is exactly what we effectively performed during pass 1 of our refactor (we'll come back to why there are multiple passes in pass 2): we took all the relevant functionality concerning shapes from the class AbstractRenderer, and extracted this functionality into it's own class ShapeManager. We then let AbstractRenderer extend from ShapeManager.

5.1.1 Results - Duplicate Code

We initially used 2 tools to look for code duplication, Intelij and Iclones. Both had confusing results.

First we ran the tools on a version of Jfreechart without the refactor to have a baseline to compare to. We saw that Iclones found a lot of duplicates. These can be found in the clonereportbefore.txt file. These duplicate results to increase afther the functionality extraction, since some functions were literally copied for stability, thus effectively introducing new duplicates.

After running Iclones on the recent version of JfreeChart (one where the extraction was done) we saw a surprising result. The amount of duplicates found had drastically been reduced. Which we found odd because of the aforementioned duplicates we had introduced.

To try clarify these results we ran also duplicate detection with Intelij. This gave similar results but was easier to navigate and analyze. Here we discovered that some duplicates didn't show up because of a low complexity or severity.

We theorize that thanks to the refactor. A lot of duplicates get less severe or complex. That in turn makes them not show up in the results of duplicate analysis. Another clarification could be that the heuristic used during duplicate analysis somehow doesn't recognize the same duplicates anymore.

In any case, we would need to find some way of having more clarity. Thus we moved to use a third tool (which did a lot more than just duplicate detection), SonarCloud. This seemed to be a better option, since it was a) easier to navigate and use than both Iclones and IntelliJ, b) showed us what we expected; the copied code was detected as duplicates and c) it generally gave us better duplication information about the project.

Now that we had confirmation that we indeed created some more duplication, we tried to pull away the functionality which caused the duplication (as we mentioned above).

5.1.2 Results - Coverage

We ran coverage reports before and after we performed our shape functionality extraction (moved them to our ShapeManager class), which gave us some surprising results. While we didn't expect much of the coverage to change (since in theory almost everything stays the same) except for some minor details concerning the classes/packages which were directly linked to AbstractRenderer, but the results show us some strange changes.

For example; some other packages seem to have gained some classes (chart.axis) while other packages seem to have lost some classes (data.json, data.json.impl, chart.text).

5.2 Testing

5.2.1 Results - Coverage

After we completed our additional test classes, we reran our coverage analysis which greeted us with some promising results. Line coverage had improved from 62% to 68% and branch coverage had improved from 49% to 52% in the whole renderer package (and not just in our ShapeManager class).

5.2.2 Preparing for the future

While we didn't actually expand the testing suite all that much, we can still suggest future testing scenarios which could help the person implementing the new functionality with testing the old and new functionality. These are scenario's which could occur when you effectively would implement the requested features in the way the current code suggests that you do it (overriding the getItemShape function, which can happend at different 'depth levels' of the code).

Each scenario notes where it's applicable and what needs to be tested. The following steps are the same over all scenario's:

1. export a chart using the specified *Renderer*.
2. verify if the chart has the desired shapes.

SC1: The getShape function is overwritten in abstractRenderer
SC1.1:

- Use: *all classes that inherit from abstractXYItemRenderer*
- Result: the chart should use the changed shape functionality

SC1.2:

- Use: *all classes that inherit from abstractCategoryRenderer*
- Result: the chart should use the changed shape functionality

SC1.3:

- Use: *all subclasses of classes that inherit from abstractXYItemRenderer*
- Result: the chart should use the changed shape functionality

SC1.4:

- Use: *all subclasses of classes that inherit from abstractCategoryRenderer*
- Result: the chart should use the changed shape functionality

SC2: The getShape function is overwritten in AbstractXYItemRenderer
SC2.1:

- Use: *all classes that inherit from abstractXYItemRenderer*
- Result: the chart should use the changed shape functionality

SC2.2:

- Use: *all subclasses of classes that inherit from abstractXYItemRenderer*
- Result: the chart should use the changed shape functionality

SC2.3:

- Use: *all classes that inherit from abstractCategoryRenderer*
- Result: the chart should use the default shape functionality

SC2.4:

- Use: *all subclasses of classes that inherit from abstractCategoryRenderer*
- Result: chart should use the default shape functionality

SC3: The getShape function is overwritten in AbstractCategoryRenderer.
SC3.1:

- Use: *all classes that inherit from abstractXYItemRenderer*
- Result: the chart should use the default shape functionality

SC3.2:

- Use: *all subclasses of classes that inherit from abstractXYItemRenderer*
- Result: the chart should use the default shape functionality

SC3.3:

- Use: *all classes that inherit from abstractCategoryRenderer*
- Result: the chart should use the changed shape functionality

SC3.4:

- Use: *all subclasses of classes that inherit from abstractCategoryRenderer*
- Result: the chart should use the changed shape functionality

SC4: The getShape function is overwritten in a inherited Renderer R of either abstractXYItemRenderer or abstractCategoryRenderer.
SC4.1:

- Use: *all classes inherited of abstractXYItemRenderer or abstractCategoryRenderer other than R*
- Result: the chart should use the default shape functionality

SC4.2:

- Use: *all subclasses of R*
- Result: the chart should use the changed shape functionality

SC4.3:

- Use: *all subclasses of a inherited class of abstractXYItemRenderer or abstractCategoryRenderer that does not inherits from R*
- Result: the chart should use the default shape functionality

SC5: The getShape function is overwritten in a subclass of a inherited Renderer R of either abstractXYItemRenderer or abstractCategoryRenderer.
SC5.1:

- Use: *all subclasses of a inherited Renderer of abstractXYItemRenderer or abstractCategoryRenderer other than R*
- Result: the chart should use the default shape functionality

SC5.2:

- Use: *all inherited Renderers from where R is a subclass*
- Result: the chart should use the default shape functionality

SC6: the getShape function is implemented in the ShapeManager.
SC6.1:

- Use: *all subclass of a inherited Renderer of abstractCategoryRenderer or abstractXYItemRenderer*
- Result: the chart should use the changed shape functionality

SC6.2:

- Use: *all inherited Renderers of abstractCategoryRenderer or abstractXYItemRenderer*
- Result: the chart should use the changed shape functionality

5.3 Refactoring, pass 2

After our first pass, we saw that there was some more functionality which could have been extracted, as well as some duplication caused by our original extraction. Because of this we wanted to do a second refactor pass to improve upon the things we did in the first pass. Our second pass then effectively was extracting additional functionality as well as removing redundant functionality, or functionality which caused duplication.

5.3.1 Results - Duplicate Code

Now that we made some additional changes to the project, we had to rerun our duplication analysis to get confirmation whether or not the duplication which we had caused in the first place had indeed been removed by our subsequent action. The results of our analysis did not make it immediately clear that our duplication had been removed (this probably because the caused duplication had such a small impact on the total amount of duplication present in the project), but upon further inspection, we could indeed confirm that our duplication had been removed.

5.3.2 Results - Coverage

Even more so, after having our duplication removed, we reran our coverage analysis once more and could conclude that our results had improved even more; line coverage had increased with 1% and branch coverage had even increased with 2% (once again this is for the whole renderer package, and not just the ShapeManager).

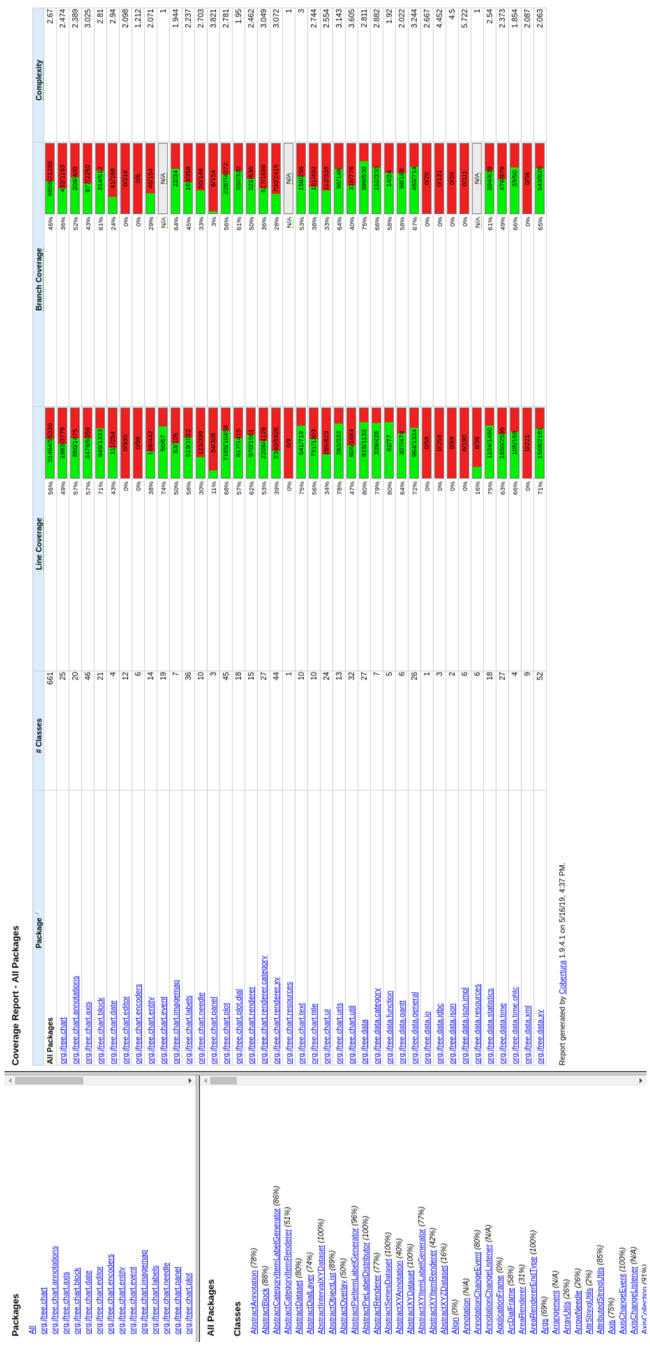

Figure 6: coverage before extraction

Figure 7: coverage after extraction

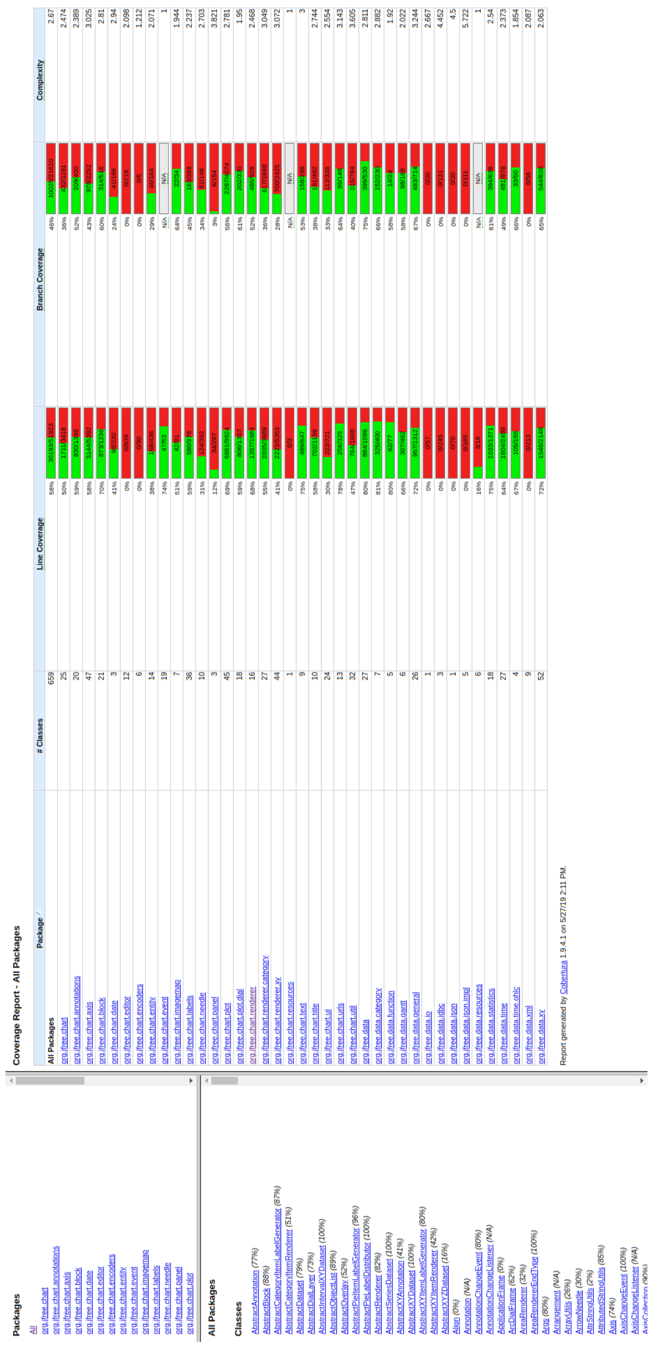

Figure 8: coverage after expanding test suite

Figure 9: coverage after expanding test suite and second refactor pass

5.3.3 Results - Remarks

We would like to note that even though we use the coverage of the system to prove that some aspects of the system are improved, we would still like to say that we know that coverage is just a metric and not absolute proof. It can be used to aide your findings and workflow, but it gives no guarantee of a perfect (testing) system (as we have clearly seen in the Software Testing lecture). Nevertheless, we wanted wanted to use coverage as it is a indicator for improvement for our specific changes.

6 Conclusion

We want to begin our conclusion with a well known saying: "If it aint broke, don't fix it." The problem with this quote is that it is often used as an excuse for not touching a working piece of software, which may still be broken in some way.[1]

The second thing we want to mention in our conclusion is yet again our scope. Simply said, we had to refactor the project as to allow the requested functionality to be implemented. So the first (big) question we asked ourselves was: what should be refactored to allow for this functionality to be implemented? When we finally figured out how the system worked, our first remark was: *this project doesn't need to be refactored to allow for the requested functionality.*

But this doesn't mean that everything in the system is optimal. So since our first step was to 'refactor the system as to allow for the requested behavior' (which eventually resulted in us extracting the shape functionality and adapting the testing suite), the second step would be 'check whether the system (within our working scope) has other optimization we can realize'. So after evaluating whether our scope had to be expanded (which we eventually concluded that it didn't), we began perfoming some analysis techniques on this extracted functionality. This resulted in what we called our 'second refactor pass'.

With our second pass, it became clear that in the process of extracting our functionality, we created some code duplication as well. After some discussing and assessing whether or not it was possible and feasible to extract more functionality to avoid this duplication, we eventually decided that it would cause no other trouble and would be beneficial overall.

6.1 Remarks

For us, this project consisted mostly of discussing what might be the right thing to do in this case. We came to a fairly quick conclusion that we thought the project should not be refactored to allow for the requested behaviour. Of course, this seemed a bit strange to us, so there was a lot of discussing whether or not that was the right thing to do. We opted to make the functionality concerning shapes more visible to anyone who would actually want to implement the requested functionality. But this was a mere cut and paste job for us. So once again we discussed whether this was enough, was this all we could do to try and achieve our goal.

We tried to stand in the shoes of the 'customer'. Whether he would be satisfied if we presented him our final result. It was a tricky situation, because from the customers point of view, he wants to receive the library in a state where it is possible for him (or someone else) to actually implement those features, but

we would be there to tell him: "It is already possible. It might not be that clear that it is already possible, so made it a bit clearer." So eventually we decided that our hypothetical customer would indeed be satisfied.

There has been a lot of discussing, about a lot of things, some of them being: what is our scope, when are we not doing enough, when are we going out of scope... But one thing in particular: What is the line between refactoring (preparing the project to allow for the implementation of the requested features) and implementing (actually readying the code for this implementation). We agreed that it was not our place to decide how a third person would or should implement those features, which led to being our biggest holdback in doing things during this project. Yes, we could have made a database interface to *portray that reading shapes from a database would be fairly easy*, or we could have effectively implemented some version of the getItemShape (and setItemShape), but other than serving as unfinished examples just to prove our point it would be useless. If our explanation cannot prove our point, than we probably are just explaining it wrong.

6.2 Notes

6.2.1 Tool Usage

In this section we'll list all the tools which we used to become our results for this assignment.

- Jetbrains IntelliJ
  usage: code editing, code inspection, feature location, duplicate detection, code smells, refactoring, project visualization, general analysis
- CodeScene
  usage: code smells, refactoring targets, general project analysis, repository visualization
  before refactor: `https://codescene.io/projects/4684/jobs/13060/results`
  after pass 1 `https://codescene.io/projects/4929/jobs/13798/results`
  after pass 2 `https://codescene.io/projects/4953/jobs/13862/results`
- iClones
  usage: duplicate detection
- SonarCloud
  usage: bug detection, duplicate detection, code smells, general analysis, coverage analysis
  results: `https://sonarcloud.io/dashboard?id=BeaverWalter_Softeneering`
- Cobertura (included with the project)
  usage: coverage analysis

- Gource
  usage: repository visualization
- Conqueror (Eclipse plugin)
  usage: feature location
- JaCoCo
  usage: coverage analysis
- Github
  usage: Version Control System
  results: `https://sonarcloud.io/dashboard?id=BeaverWalter_Softeneering`

References

[1] O. N. Serge Demeyer, Stphane Ducasse, Object-oriented reengineering patterns (2013).