Week 8 Status Report

Christopher O’Hara, Spencer Murphy, Josh Harbison

1. Progress made:
   1. Added more features to player windows & market windows
   2. More internationalization bug fixes
   3. Began tech
   4. Added panel details for other civilizations
2. Next week progress plan
   1. Implement tech, government
   2. More code analysis/refactoring to boost coverage
3. Code coverage is at 78.5%
4. Approximate lines of code per team member:
   1. Spencer: 176
   2. Chris: 529
   3. Josh: 797

Fun Metrics Overview:

For our first fun metric, we would measure the number of times the game is played per month. This is valuable for a couple reasons, but the interpretation of it is based on other metrics as well. First, this metric would tell us how popular the game is and how much the player base is growing. If the number is high, then we must be doing several things right, and the players are enjoying the game overall. If the number is low, then perhaps the game has some bugs that need fixed or the game is too long to play to be enjoyable. It could also correlate to the amount of advertising we’ve used to market the game. With not very much advertising, not many people would know about the game or would be drawn toward it. This is a useful fun metric to have, but the most accurate interpretation of it would require additional metrics.

Our second fun metric is the number of games that are completed in full. If our game does not have very many major bugs, then this number will tell us how enjoyable the game is to the users. If the number is low, it could be that the users aren’t feeling that it is worth their time, or that the game is just too long to play. We would probably have to revise the game if this were the case.

The third fun metric would measure the number of ways to win and how many wins each of the ways has. This is important because it tells us what features the players prefer and what features they tend not to use or enjoy. If the wins are evenly distributed, then there’s a good mix of players that enjoy winning each of the different ways. If the results are lopsided, then one or more of the ways to win are not being used as much as expected. If this is the case, then either the way that has the most amount of wins needs revised (perhaps it is too powerful or easy to win), or the less-popular way to win should be removed from the game so as to make the game simpler and to please the players.

Our fourth fun metric is the victory progress for all the conditions. Eventually, each player makes a choice of what journey they want to take in order to win the game, whether this is by combat, culture, etc. This metric would measure each player’s progress in their journey choice, so we can see if the game is still too lopsided towards one way to win or if each of the ways to win are pretty even. We would want them to be relatively even. This metric would only provide useful information after the players have played the game for a decent amount of time (they won’t make their choices until later in the game).

Lastly, our final fun metric would be the number of battles per game compared to the overall victory condition. Combat is a big portion of the game (especially later in the game), so we want to check to see how many battles there are, and what the final victory condition was. If the combat never influences the victory result, then we probably need to boost the effect that winning battles has. If the relation is split up among the various victory conditions, then it is behaving like we want it to. Also, we don’t want combat to be too powerful, so if all the victories are based on battle victories, then we should consider toning down the effect that combat has on the overall game. We are going for the good balance.

**Code Quality Metrics**

In order to ensure that our code is of the highest quality, we decided to track several metrics. These metrics should keep our code cleaner and not as poorly written. As we are developing in Java, we decided to use the metrics tool discussed in Lab 10 to track our metrics. This metrics tool provides lots of different metrics that we could track, but we decided on just the five that were the most meaningful to our project. Below is a table containing the metrics that we chose to track, the values we based them on, and the actual values that our current program produces. Each metric will be discussed after the table.

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Max Allotted Value | Actual Max Value | Avg. Value |
| Nesting Depth | 6 | 10 | 1.29 |
| Method Lines of Code | 100 | 207 | 9.724 |
| Total lines of code per class | 1500 | 1130 | 149.3 |
| Number of Parameters | 5 | 9 | 0.588 |
| Number of Attributes | 10 | 21 | 2.278 |

One metric we tracked was Method Lines of Code. We wanted to track this to ensure that no methods were doing too much at one time. Making smaller methods out of our long methods also increases the overall readability of the code. We chose a kind of high max of 100 lines so that we could have longer methods, but nothing crazy. We ended up with four methods that were above our threshold of 100 lines. Our average, however, was only about 10. This means that as a whole, we did an ok job of keeping our method sizes down, with a few problem methods are way too long. To fix this, we should extract some of the code in these methods into smaller methods or try to make the code slicker.

Another metric that was over the max allotted value was number of parameters. This was only in the tile class constructor where lots of information is transferred from a file to create the object. This could be fixed, but overall it is not a huge problem. In order to fix this, we could have separate setters to split up the constructor.

One final metric that was way over the max value was number of attributes. We set a max at 10, but 6 of our classes were well over the max that we set. This may have been an underestimate of a good value on our part, but it also shows that we indeed use lots of attributes in our classes. Some of these could be fixed by breaking up some of their classes into smaller more specialized classes.