MPHYG001: Second Continuous Assessment: Refactoring Bad Boids

29th February 2016

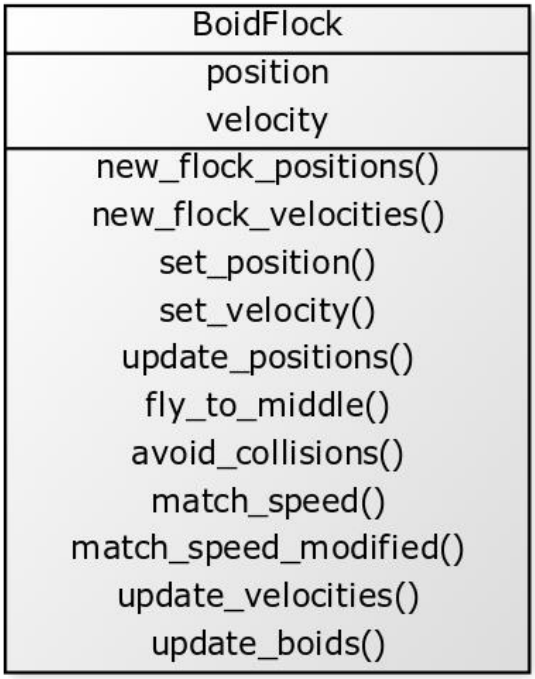
Ernest LO

SN:110006646

Git commits (newest first, only includes those that involve refactoring)	**Code smells identified and modifications made**
ad5b8b8 added packaging 2bd34e6 added folder structure e02df0d added fixtures and added more tests to testboids	The code is not neatly organised, and therefore split up and put into folders. With the new code, tests don't have to just test the full code but unit tests can be applied.
3bd55bd added command line file and config file f1548ff added functionality for a configuration file	A lot of the constants in the code, while defined with a suitable variable name, is still a magic number. We can input them using a configuration file so users can specify their own values without modifying the code. Added command line interface with usage guide to aid user
d4f9119 boids file is now the class file a3b3116 completed class structure 2a1fee0 updated class file c774582 added class file version of boids.py (incomplete)	The boid flock is a set of numpy arrays and loose variables. They can be codified into a Class for easier readability. This will take the code into an object oriented approach, which makes sense as the attributes are obviously position and velocity, and the methods that affect these will be the various modes of flight. Because of this new functionality, we could split the code up easily for tests, adding setter functions to input fixtures.
64a715a implemented match speed to a function 65a3625 implemented avoid collisions to a function 6bf7ffb traded for loops for function to fly towards middle	The three flight processes can be split up into functions, which allows them to be independently tested.
bca1913 added function to update velocity given the new position changes e9d9a7d added function to update positions given the new velocity changes	We identify that the main is group of many small algorithms that we can split up into functions for ease of testing and comprehension. Started with the update position and update velocity functions.
e2cd0da Fixed bugs, reconfigured code to work with Numpy arrays	Instead of using complicated lists structures, the positions and velocities can be easily replaced with numpy arrays, which allow array-wise operations.
6fa9e31 Replaced the 0.125/len(xs) with a variable 2fd9a70 Replaced the 0.01/len(xs) with a variable	Replaced the "magic number" proximity weights with a variable name

96b5264 Replaced handwritten initialisation of boid values with numpy function	The initialisation of the flock could be easily replaced with a numpy array which is quicker and more easily comprehendible.
210db56 Merged redundant loop	In the fly towards middle algorithm, the loops for y and x velocities are redundant, therefore they were merged.
130c7d5 Replaced proximity condition to match speed with function 3558f27 Replaced proximity condition to fly away with a function	Saw that the condition for the avoid collisions and match speed algorithms could be written neatly as a function, for ease of testing and manipulation
b10bb62 Replace xvs,yvs,xs,ys with appropriate variable names 89960f9 Replaced len(xs) with NBoids = len(xs)	Identified poor variable naming that displays little information. Changed variable names and identified that len(xs) is the size of the flock.
9e95d24 Changed rangelimit name to NBoids 3a3d5da Changed range limits to constant RangeLimit=50	Identified “magic number” of 50 as the limits of the ranges. Gave it a variable name, and later called it directly from the size of the flock (which is 50)

UML diagram of the class structure



Advantages of refactoring

The refactoring approach to improving code readability and robustness is widely successful because of its foolproof nature. While there can be subjective ideas about what is more readable and whether faster but less readable code is "better code", there are some well agreed upon modifications that generally improve maintainability. An obvious refactoring is to break up large methods or algorithms into simple classes and small functions, it becomes far easier to analyse and understand the logical structure as well as making tests a lot easier to implement.

Often when code needs to be maintained by multiple people or inherited by new programmers, such as commercial software and research code where it is often handed down to new students or employees, an abundance of code smells will propagate and make the code increasingly more difficult to understand, improve and maintain. This is particularly important for code that will likely be published.

There is the well observed programming concept of technical debt, where by taking the quick approach to code, writing something that works but is poorly written, will in the long run, produce propagating complications that must be rectified before any future work can be performed. " i.e. the debt must be repaid". While good coding practices are taught to all proficient software developers, code smells will inevitably be created because of time constraints. Therefore time dedicated to refactor working code is often seen as essential, and companies often have staff whose entire work revolves around making such changes.

Problems encountered

Identifying code smells is sometimes a subjective call. While there is general advice to reduce loops or replace it with iterators or array methods, often times it changes the logical structure of the code, and would be a code modification versus refactoring in its purest form. An example of this is the match speed algorithm for the boids. Unlike the flying towards middle and avoid collisions methods, the loop structure cannot be easily replaced without fundamentally changing the algorithm. While taking a numpy array method to the match speed algorithm may reduce looping and improve speed and readability, it is not a necessary improvement from a refactoring standpoint. Ultimately while I had written an alternate method for matching speed, I retained the old loop structure as it does not modify the logic of the original code.

Often when making large refactoring changes, such as writing new functions and classes, you will be writing a large amount of new code, there is no reason why this new code may not contain smells of its own, and require additional refactoring.

Lastly, even when there may be an obvious code smell, the way to refactor it may not be obvious, or there may be multiple vastly different refactoring solutions. This is where it may become subjective or heavily dependent on personal style and preferences. However because refactoring is meant to improve readability and maintainability for others, a biased approach may be counter-productive.