Assignment 2: Refactoring the Bad Boids

Rianna Kelly

Student ID: 16100406

Course Code: MPHYG001

February 21, 2017

Abstract

This assignment takes the bad-boids code written by Dr. James Hetherington (sourced on GitHub[1]) and safely refactors it into a clean implementation of the flocking code, finishing with an appropriate object oriented design. This report documents the code smells identified and the associated refactorings, provides a UML diagram of the final class structure and discusses problems encountered during the assignment. It also describes the advantages of a refactoring approach to improving code.

1 Introduction

The bad-boids code [1] is an implementation of the Boids program designed by Craig Reynolds to simulate the aggregate motion of a flock, swarm, herd or schools of animals [2]. The submitted code, accessible from `https://github.com/RiannaK/Coursework2`, is broken down into functions, classes, methods and modules. The code is readable with clear variable, function, class and method names, and includes comments where appropriate. Git version control was used and relevant commit messages were submitted. Automated tests are in place for each method and class, and the build status (passing) and test coverage (100%) can be seen on the Git repository homepage. The packaged code is suitable to be installed either directly through python setup.py or via pip installation, with usage and install instructions clearly defined within the README.md file of the Git repository. The use of the configuration file is also explained within the README.md. Further files are included in the Git repository relating to the licence for further use and the citation requirements.

2 Entry Point and Configuration File

The package can be installed on the user's computer either by manually downloading from GitHub and installing, or by installing directly from GitHub using the standard pip command.

Once installed, the user can choose to run badboids using the default settings. In this case, the badboids script can be called from the command line using the following syntax:

```
badboids
```

If desired, the user can specify their own simulation parameters and setup parameters within a configuration (.yaml) file. The required parameters and an example configuration file can be easily found within the README.md file of the Git repository.

In this case, once the configuration file is updated, the badboids script can be called from the command line using the following syntax:

```
badboids --config "insert\full\file\path\to\config.yaml"
```

3 Refactoring approach

3.1 Implementation

A key objective in this assignment was to demonstrate a methodology that allows for the safe refactoring of the original code. To achieve this, a systematic approach of testing, making incremental changes and retesting was undertaken as follows:

1. Regression testing – As soon as the code was compiling, a regression test was set up. A regression test distinguishes itself from a unit test in that it aims to test large portions (in this case all) of the code base as opposed to isolating any particular unit of code. The inputs to the test were carefully selected to ensure that all of the boids logic was covered by the test (i.e. no branching logic was left uncovered), and this was verified using Coverall's line coverage feature.

2. Continuous integration – Having established the regression testing framework, both a continuous integration build server (Travis CI) and an automated testing system (Coveralls) were put in place. The purpose of this was to ensure that feedback on the status of the tests was constantly provided on each push to GitHub. In this way, any failures in the regression tests – indicating a possibly incorrect refactoring – would be picked up at the first possible instance.

3. Code smells – Code smells were addressed and each code smell was mentioned in its own Git commit. A full list of the code smells identified is provided in the next section.

4. Unit tests – While regression tests allow one to quickly achieve a high degree of code coverage, they do not provide as much granularity or clarity as a unit test. As individual units of code emerged (via the introduction of methods), further tests were written in the form of unit tests.

5. Design Architecture – Only after a suite of unit tests was established, were major architectural changes made. The code was further refactored from procedural code to an application that utilises classes and established design patterns such as the builder pattern and Model-View-Controller (MVC).

3.2 Code Smells

The following code smells were identified and addressed.

- Code smell: repeated for loops. Removed repeated for loops.
  Commit: 80709a115a088c1db98e2b242b9d3bbace86507d

- Code smell: repeated code. Removed repeated code len(xs) with a named variable.
  Commit: d1df994054026e47108f0da5c54cdca08840724b

- Code smell: magic numbers. Replaced magic numbers with constants to provide clarity on what the numbers mean.
  Commit: 7265b9f4c8317dd666957a22dda1de953e4406b4

- Code smell: hand-written code. Replaced hand-written code with library code. Used numpy to replace for loop with "vectorised" code.
  Commit: ebd53a993cf35fb3720cc03eef81a1ebcaa5c4d8

- Code smell: change of variable name. Changed badly named variables for clarity.
  Commit: ac317fa24611491b2fda19431ad34b42f6318713

- Code smell: repeated code. Replaced code around position/velocity difference with named variables.
  Commit: ccd56024149fbe791d7d43b01ec9849e3bdde50b

- Code smell: Move hardcoded constants to config file. Adding a simulation parameters loader class.
  Commit: e77e2339fb315d86471a021027e1220f5c629c67
- Code Smell: Hardcoded constants moved to config file. Renaming simulator to simulator_model (MVC pattern used).
  Commit: 9230324b85abb96c62641499faf36cd220c9fa2a

4 Design Architecture

The original implementation was transformed from procedural code to a lightweight script that leverages off individual classes. Design patterns were used where appropriate. For instance, the builder pattern was used to create the Boids class; an operation which would otherwise have been complex and potentially dangerous can now be performed safely.

Classes were written with best practices in mind. No single class does a large amount of work but the sum of all classes achieves a considerable amount. Each class, having a single, well-defined role, demonstrates a guiding principle known as the 'Single Responsibility Principle' [3].

The code follows the MVC design pattern to address 'separation of concerns'. Here, the *SimulatorModel* acts as the model, *BoidsView* is the view responsible for charting and *BoidsController* ties the model and view together. The advantage of such an approach is twofold: business logic is kept clear from display logic improving readability/maintainability and, secondly, the re-usability is increased as the Model code could easily be used in another project entirely independent of the view.

The final class structure can be seen in the UML diagram given below in Figure 1.

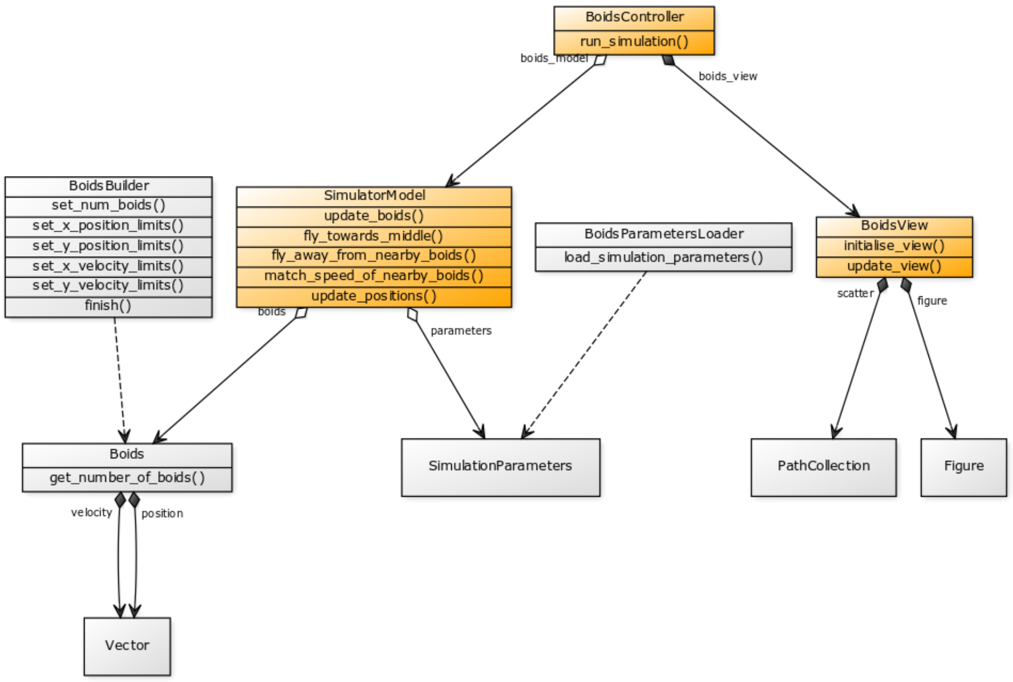

Figure 1: *UML diagram showing the relationships between the various classes.*

5 Problems Encountered

The most obvious difficulty encountered in this assignment was in refactoring the original implementation to use ‘vectorised code’. The program uses first-order numerical integration – Euler’s method – to calculate the new positions of the boids based off their current positions and velocities. However, the original code is unorthodox in that, as the new velocities are being computed within the update_boids method, they are immediately used to calculate the new velocities of other boids within the same iteration of the integration. This leads to the undesirable feature that the resultant positions are dependent on the order in which the program evaluates the boids; the initial implementation is arguably incorrect.

In transitioning to vectorised code, Numpy automatically takes care of handling the integration in the expected manner. However, it did cause a number of tests to fail given the unconventional original implementation. As the mismatches were small, all less than 1%, and I was comfortable with the differences, I handled this by updating the tests to reflect the new numbers. The mismatches are shown below:

```
Arrays are not almost equal to 6 decimals
E
E       (mismatch 100.0%)
E        x: array([-48.660374, -38.401894, -32.559609, -26.718649, -20.879062,
E               -15.040899,  -9.204209,  -3.369049,   2.464528,  12.698783])
E        y: array([-48.615625, -38.409375, -32.578125, -26.746875, -20.915625,
E               -15.084375,  -9.253125,  -3.421875,   2.409375,  12.615625])
```

I also encountered a problem owing to the brittleness of the initial regression test. The open/closed principle states that code should be ‘open to extension, but closed to modification’ [4]. Code that is closed to modification has the clear advantage (amongst others) that tests do not need constant maintenance. In refactoring the code to an Object Oriented design, some breakage of the (regression) tests was inevitable. One must always be careful when updating tests to ensure that bugs do not inadvertently creep in. To ensure this did not happen, two precautions were taken:

1. granular unit tests were written for new code so that a high degree of confidence could be placed in the code before it was integrated into the code base
2. regression tests were not simply regenerated whenever the public API was broken. Instead, every effort was made to marshal the inputs of the existing regression test so that it could still run without needing to be regenerated. By recycling the original test numbers rather than creating the test numbers afresh, I had confidence that the numbers themselves were not changing throughout the refactoring process. Only after the regression test was shown to pass under these conditions was the regression test itself regenerated (as required).

6 Advantages of a refactoring approach

A refactoring approach to improving code involves making incremental changes to the code in order to improve its structure or performance, whilst maintaining the original functionality. There are a number of advantages to this method.

Making incremental changes is more manageable and reversible in the instance that the refactoring introduces a break or bug. By separating each refactor into individual

Git commits, it is easy to isolate individual changes and understand the intent which is useful when working within a community of developers. Ensuring there is a robust regression test in place and creating unit tests as 'units' emerge allows the developer to refactor with confidence. For this assignment, I chose the 'Arrange, Act, Assert' pattern for formatting and arranging the code within my unit tests [5]. I also included the 'system under test' (sut) pattern to clearly distinguish the tested method from the setup [6].

More obviously, the process of refactoring introduces a number of benefits. By breaking the code into testable units, readability improves and complexity decreases which reduces the learning curve of anyone wishing to contribute to the code base. The addition of well written unit tests is certainly a benefit in itself; unit tests are a useful way to document the code base's functionality and allow subsequent refactorings to be done in a safe and controlled manner.

In this case, the refactoring exercise also involved the addition of an object orientated design. Moving from procedural code to an object orientated approach has many advantages. A class library in which classes have well defined roles promotes re-usability as users have flexibility to choose the individual classes relevant to their project. This would be impossible in a large procedural script where many concerns were tightly coupled. Moreover, object orientated code is intrinsically more understandable. Implementation details can be encapsulated via the use of private methods allowing for a clear API to be designed. Design patterns play a part here too – a Boids class and a BoidsBuilder make it perfectly explicit what is being represented and how it should be created.

Finally, Git complements the refactoring process by providing support for branching. For this assignment, I used a simplifed version of the referenced branching model [7]. A master branch was established and a develop branch was created by branching from master. The refactoring process took place on develop and, once complete, was merged back into the master branch. A branching strategy has the advantage that factoring can take place on a separate branch without fear of disrupting the code base. Once the feature is complete, it can be merged into the main code base. Within a community, this can be easily managed by using pull requests to merge development streams as and when they are ready.

References

[1] Dr James Hetherington bad-boids GitHub repository
`https://github.com/jamespjh/bad-boids`

[2] Craig Reynold Boids
`http://dl.acm.org/citation.cfm?doid=37401.37406`

[3] Single Responsibility Principle
`http://www.oodesign.com/single-responsibility-principle.html`

[4] Open-Close Principle
`http://www.oodesign.com/open-close-principle.html`

[5] Unit Test Basics
`https://msdn.microsoft.com/en-GB/library/hh694602.aspx`

[6] SUT at XUnitPatterns.com
`http://xunitpatterns.com/SUT.html`

[7] A successful Git branching mode
`http://nvie.com/posts/a-successful-git-branching-model/`