Group 14 — Assignment 3
Raccoon Simulator

Alex Impert, Daniel Todd,
Ryan Wittkopf

April 2022

1 Introduction

The purpose of this assignment is to refactor some of our code adhering to appropriate design principles.

2 Findings / Bad Smells

With the new information learned about refactoring and bad smells from the lectures in CMPT 276, we were able to spot several bad smells within our code that required refactoring. There were several long methods, lack of abstraction, and inefficient creation of objects to name a few.

1. **KeyHandler**

   keyPressed() keyPressed() was a method we had within our KeyHandler method to perform various tasks and set various attributes dependent upon the current gameState (whether the game was at the title screen, currently being played, etc.) This method was overly large so we split it up into natural submethods dependent upon the current gameState of the game: keyPressedTitle(), keyPressedPause(), keyPressedEnd(), and keyPressedPlay(). The keyPressed() method now calls one of its respective submethods based on the current gameState of the game. Reducing coupling and increasing modularity.

2. **Subject Position**

   subject.x & subject.y Both our player and enemy classes inherit from our subject class and use a (x, y) coordinate system to determine their position within the game. While going through our code together we noticed that many other classes (CollisionHandler and others) were accesses the player and/or enemies x and y values directly which is improper OOP programming and results in high coupling. We fixed this by adding getters and setters for subject's x and y attributes to remove this direct access and we set x and y to be protected data types.

3. **RedRaccoon**

   drawObject() drawObject() was an overridden method from the drawableObject() class which was specialized for the randomly spawning RedRaccoon reward item in our game. It incremented and checked a timer variable and performs various tasks based on the value of the timer. This was a long method. We split it up into several natural helper methods: generateSpawnLocation(), spawnRedRaccoon(), removeRedRaccoon(). drawObject() is now responsible for increment the timer and calling these various self-explanatory methods based on the value of the timer. This reduced coupling and make the class more modular.

4. **Graphmaker**

   GraphMaker.Node Object types had very similar variables to Node types causing low cohesion by data clumps. I fixed this by creating an Extract Class, GeneralObject class, increasing cohesion

within the code. I then had Node inherit from this, increasing our cohesion. I then found that I had high coupling, and I reduced this by making GeneralCollectableObject which inherits from GeneralObject, allowing for more efficient data flow.

5. **MapBlock**

   MapBlock type GeneralObject types had similar parameters to the MapBlock type causing low cohesion by data clumps. So I renamed MapBlock to GeneralDrawableObject, and now GeneralDrawableObject inherits from GeneralObject, increasing cohesion.

6. **MapManager**

   Created extract class MapManager was too large and, as such promoted high coupling. I fixed this by creating Extract Class BlockList, and renaming MapManager to BlockManager. Now, BlockList holds all the DrawableObject instantiations and allows for utilisation of DrawableObject's draw method.

7. **Project Structure**

   Package and class layout In general the structure of our project, particularly our classes and packages, seemed to be completely random and made no sense at all. I fixed this by creating the following packages: Factory, Enemy, and CollectableObject. I also renamed some classes and old packages to allow for a more straightforward directory