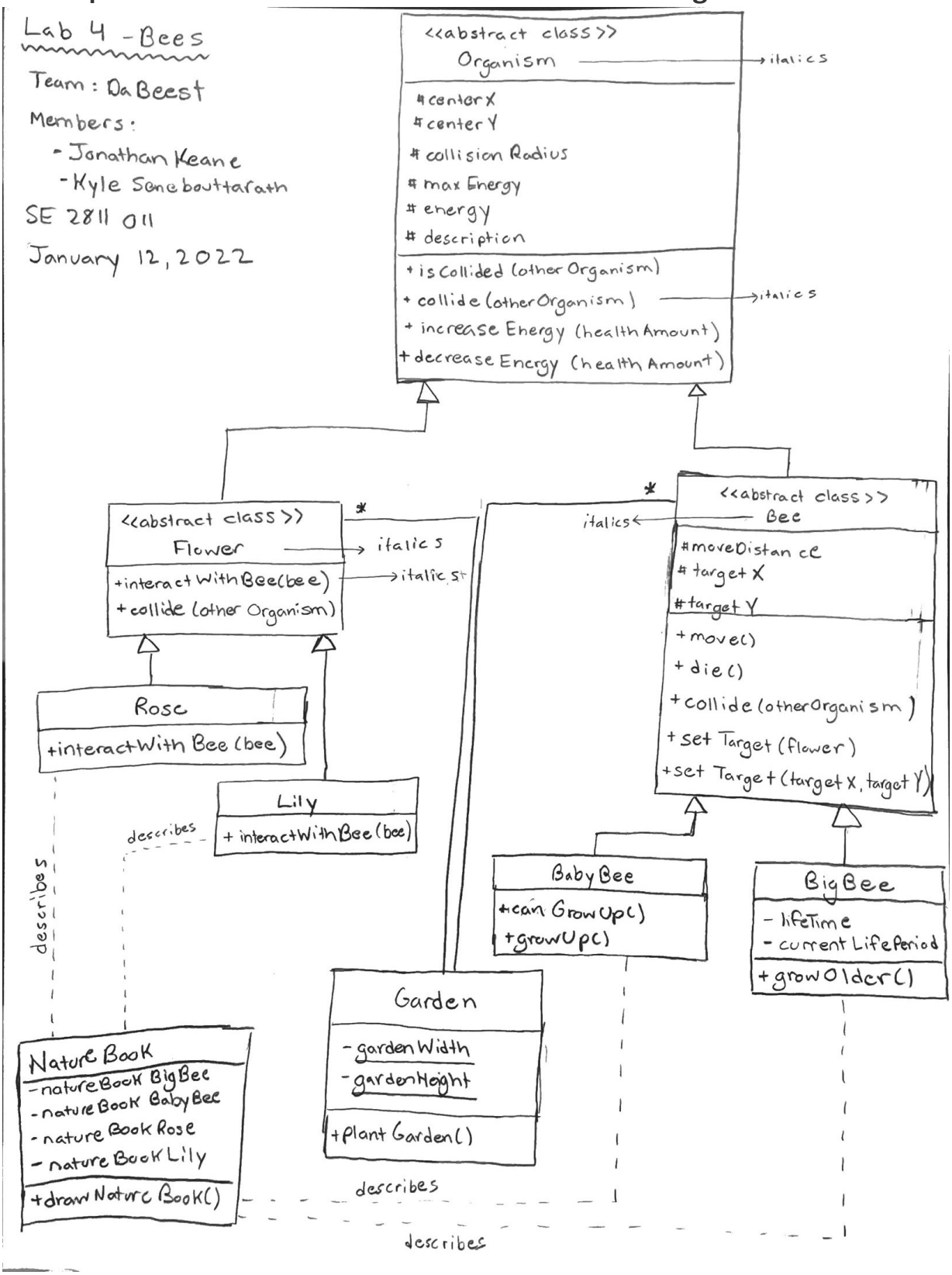
Jonathan Keane
Kyle Senebouttarath
1/12/2022
SE2811 011
Lab 4 – Bees Continued

The following PDF contains these elements on the next pages:

1. An updated hand-drawn version of the domain-level UML diagram, correcting any issues provided in the Lab 3 feedback.
2. Updated responsibilities of the domain classes we had. This is not required, but we included them because we added some responsibilities to some of the classes (mainly the bees and flowers) towards the end of the lab.
3. A paragraph describing any significant changes to the design such as new classes. Recall that you do not have to introduce design patterns into this lab.
4. A paragraph giving advice to next year’s class on how to use the design requirements for the lab to make the coding easier.
5. A paragraph giving advice to next year’s class on where to cut back on the design without hurting your coding efforts later.
6. A paragraph describing who implemented which portions of the system.
7. A discussion of any known problems, either through implementation or the design. Typically, documenting known problems reduces the penalty.

1. Updated hand-drawn domain-level UML diagram:



2. Updated Domain Class Responsibilities

Organism – An organism responsible for recognizing and handling collisions, managing health, and managing positioning for an organism in the garden. A Bee or Flower can be an Organism in the garden.

Bee – A Bee is responsible for representing bees within the garden. Bees are moving organisms that follow a movement pattern to move around the garden. When bees hit a flower, they will attempt to take energy from the flower for themselves. If a bee hits another bee, they will crash, and both lose some energy. When bees run out of energy, they die. Bees lose energy as they move.

BabyBee – The BabyBee is a type of bee that is a small bee. It has a small amount of energy, but it’s faster and has a smaller collision area. It has the same responsibilities as a bee, but it has defined properties and the additional responsibility of handling the bee's ability to grow up if it is reaches maximum health.

BigBee – The BigBee is a type of bee that is a large bee. Because of its size, it will be more likely to run into other organisms. However, it has more energy. It has the same responsibilities of a normal bee, such as handling movement and dying, but it is has set properties to make it larger and slower. Big bees grow older and slower over time.

Flower – A flower is responsible for representing a flower within the garden and interacting with bees that visit them. Flowers are stationary organisms that attract bees. They either give bees’ health when interacted with or drain bees’ energy if it’s a bad flower. Flowers only have a limited amount of pollen (their "energy") that slowly recharges over time. Flowers cannot die.

Lily – A Lily is a type of flower that is responsible for giving energy to a bee with it collides with one. If the Lily has enough energy, it will give the bee energy and will lose energy in doing so. It regenerates pollen (it's "energy") over time.

Rose – A type of flower that has the responsibility of taking away the energy from a bee when it collides with one (either has enough energy to take from the bee or not). The Rose will lose energy itself when it drains a bee of its energy. It can only drain energy when it has enough itself. It will drain a set amount of energy only. It regenerates energy over time.

Garden – The Garden is responsible for holding a collection of different organisms that exist inside it. These organisms will be either bees or flowers. The size of the garden should not overlap with the NatureBook. When the Garden is initially created, it will have a set number of bees and flowers in it.

Nature Book - The NatureBook class has descriptions and pictures of all the natural things that can be found within the garden! This includes all flowers and bees and their descriptive information.

3. What are the significant changes to the design? Are there any new classes or interfaces?

The biggest significant change we added to our final code was the implementation of movement patterns for our bees. Using the strategy pattern for the movement of bees, we created three new classes. There is RandomLiveMovement.java and FlowerOrientedMovement.java, which both extend off BeeMovementPattern.java. Every bee now has a reference to a BeeMovementPattern class, which it then uses to determine the target x-coordinate and y-coordinate the bee should fly to.

In the FlowerOrientedMovement class, the bee calculates its target x-coordinate and y-coordinate to a random flower in the garden. In the RandomLineMovement class, the bee calculates its target x-coordinate and y-coordinate to a random location in the 2D plane of the garden. We made these separate movement patterns so the bees wouldn’t behave all the same way, and to throw in some randomness into their movements. In the CircularMovement

class, the bee moves in a circular motion and its radius and increases to a maximum and then decreases to a minimum over time.

4. **What is some advice to give to next year's class on how to use the design requirements for the lab to make the coding easier?**

   Strongly focus on classes that pertain to the domain during the design phase. Don't think about any solution level classes until your domain design and UML is completed. Your domain design should already have enough classes for you to be able to code a solution for this lab because it's smaller and not a large-scale project.

   Understand the is-a/has-a relationships at a domain level because this will clarify where you want to write your methods and will help you avoid redundant code if you have is-a relationships that allow many classes to share functionality. For example, bees and flowers both have to update and draw on every click, so by making these both has an is-a relationship with something like an Organism class can help you meet multiple requirements with the same code.

5. **What is some advice to give to next year's class on where to cut back on the design without hurting your coding efforts later?**

   Make sure to stay focused on the domain-level classes only during the design process. Do not start thinking about classes or interfaces that may be used on a solution level (such as a Drawable interface or Update interface). In general, your domain-level classes should be able to sustain these solution-level ideas; they should also help you keep within scope of the lab, because it can be very easy to get carried away with adding loads of more classes that don't particularly focus on the domain during the design phase.

Another point is to be wary of giving synonym names for solution level classes to force them into a domain design. In our case, we renamed our GUI key, which seems like a solution-based class, into a “Nature Book” class. This is a light example, but it can quickly get egregious if not monitored (for example, changing a “description” attribute to a “life purpose” attribute). The domain gets you off the ground, but the domain isn’t going to perfectly accommodate your solution.

Make sure that you get the main points of the lab first. There are many ways that you can add more challenges to the lab if you finish with time, but you should try to get your movement and collision detection done first because this allows your more elaborate ideas to come to life if you want to. If not, you already have the bulk of the lab completed.

It may seem beneficial to use interfaces liberally to give classes with similar functionality the same methods, so they can be called in bulk on a collection of that interface (ex: Drawable interface with a draw method). While this is a great practice for normal coding because it allows classes to easily pick and choose the functionality they need, we were able to complete all the requirements without these extra interfaces, so sticking to your domain classes instead of branching out to many new solution-level interfaces could save you time in coding without costing you functionality.

6. What portions of the system did everyone implement?

Although we both were assigned our own classes that we should have solely done, we ended up hopping over each other’s classes to fix/update parts of the other’s implementations. Kyle mainly focused on the getting the Organism class functioning, as well as implementing the Bees and the variants of Bees. There were some moments where Kyle had to jump to either edit portions of the Garden or the Flower classes to make sure they were up to date with the Organism class’s implementation. The core handling of the Bees’ movement to target points was also done by Kyle.

Jonny focused on the Flower classes and their variants, the Garden, and NatureBook classes. There were some moments where he as well had to jump over to other classes, such as Organism to hide health bar displays in the NatureBook organisms. Jonny added some simple GUI functionality to regenerate the garden's organisms in the Garden class. Jonny also went ahead and expanded the Bees' movement into the Strategy Pattern, so that now Bees would only fly to target flowers (default movement implemented earlier) or fly in a circular pattern that changed its radius (CircularMovement). He also implemented a pattern to fly to random points in the garden, but this is currently unused because we reread lab requirements and it said to not use this. Because we were only a two-person team and previously worked together in the past in labs outside of the class, we didn't run into any issues with code or merges, and strong communication was maintained about who was editing what.

7. **What are some of the known problems with either the implementation or the design?**

There were some troubles with writing the implementation for BabyBees growing up into BigBees, and it mainly boiled down to how the JavaFX solution should be integrated with the domain-driven design we had. Initially, we had our Garden have a Pane which would then hold all the other Panes for the organisms in the garden; the Garden also had a list of Flower and Bee objects as well. When adding a new Bee or Flower into the garden, we would thus have to add the organism's Pane into the garden Pane, and then also add the new Organism object into its respective list. This became troublesome when trying to implement a growUp() method in our BabyBee class, as the BabyBee would've needed a reference to the garden it was in, but we also didn't want to make BabyBee the only exception to this rule.

Instead, we opted to create a canGrowUp() method instead, which returns a true or false depending on if the BabyBee meets the requirements for growing up. That way, when the garden detects that a BabyBee canGrowUp(), the growUp() method we implemented would return to the garden a reference to a BigBee representing

the grown form of the BabyBee. From there, the garden would do all the work for removing the BabyBee object and adding the new BigBee object.

There are some rare cases where the bee death color adjustment (graying out and darkening) did not seem to work completely. We could not locate the source of this problem and it was somewhat infrequent (possibly some infrequent problem with JavaFX), so we don’t have a great way of replicating the problem to see what is going wrong.