COMP1202 Coursework 2013/14

Key Information

Deadline: **Wednesday 11th December 2013 at 17.00** via https://handin.ecs.soton.ac.uk

Lecturers: David Millard, Mark Weal, Rikki Prince

Value: 30% of the marks available for this module.

This coursework is to be completed individually.

Please note: Any alterations to the coursework and answers to frequently asked questions will be submitted to the course wiki:
https://secure.ecs.soton.ac.uk/student/wiki/w/COMP1202/Coursework

Coursework is the continuously assessed part of the examination, and is a required part of the degree assessment. Assessed work must be submitted as specified below. You are reminded to read all of the following instructions carefully.

Coursework Aims

This coursework allows you to demonstrate that you:

- Understand how to construct simple classes and implement methods.
- Are able to take simple pseudo-code descriptions and construct working Java code from them.
- Can write a working program to perform a complex task.
- Have an understanding of object oriented programming.
- Can correctly use polymorphism, exceptions and I/O.
- Can write code that it is understandable and confirms to good coding practice.

Contacts

General programming queries related to the coursework should be made to your demonstrator in the timetabled labs.

Queries about the coursework specification should be made to Mark Weal (mjw@ecs.soton.ac.uk).

Any issues that may affect your ability to complete the coursework should be made known to Mark Weal (mjw@ecs.soton.ac.uk) or David Millard (dem@ecs.soton.ac.uk), ideally before the submission deadline.

ECS Beekeeping

Specification

The aim of this coursework is to construct a simple simulation of a beehive in a garden. This beehive will consist of a colony of bees that perform different tasks in the hive and garden.

You are not required to have any knowledge of beekeeping in order to complete this coursework and no scientific accuracy is claimed in the representation of bees and flowers presented here.

The Beehive may bear some similarities to a real beehive but is grossly simplified and in some cases likely to be quite different to how a real Beehive might work.

Your task is to implement the specification as written.

No marks will be awarded for deviating from the specification in order to increase the realism of the Beehive in fact it may well cost you marks and make it much harder to use the test harnesses provided.

How ECS Beekeeping Works

For this coursework you are expected to follow the specification of the hive and garden as set out below. This will not correspond exactly to a real hive or garden in reality but we have chosen particular aspects for you to model **that help you to demonstrate your Java Programming**.

There are a number of animals, plants and objects that contribute to this simulation. For our purposes these include:

Bees**:** In modelling the hive you will create a number of classes representing the different types of bee (queen, worker, drone) and their various developmental stages (egg, larvae, pupa.) You will also be modelling the tasks they perform (laying eggs, collecting pollen, making Royal Jelly, etc.)

a **Hive:** The bees live in a Hive, which has a number of cells for each bee and supplies of key foods such as Honey, Royal Jelly and Pollen.

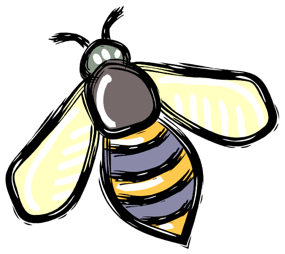
a **Garden:** Your Hive will be situated in a Garden. This garden will contain a number of flowers.

Flowers: Flowers supply the bees with Pollen.

a **Beekeeper:** The beekeeper is responsible for setting the Hive up and making sure the garden has flowers in it.

The next sections will take you through the construction of the various animals, plants, objects and people. You are recommended to follow this sequence of development as it will allow you to slowly add more functionality and complexity to your simulation. The first steps will provide more detailed information to get you started. It is important that you confirm to the specification given. If methods are written in red then they are expected to appear that way for the test harness and our marking harnesses to work properly.

Part 1 – Modelling Bees



To model the various types of bee you will need for your *Hive* (*Queen, Worker, Drone, Egg, Larvae and Pupa*) you will need to use inheritance.

The following simple diagram shows you how the seven classes that you will create are related to each other.

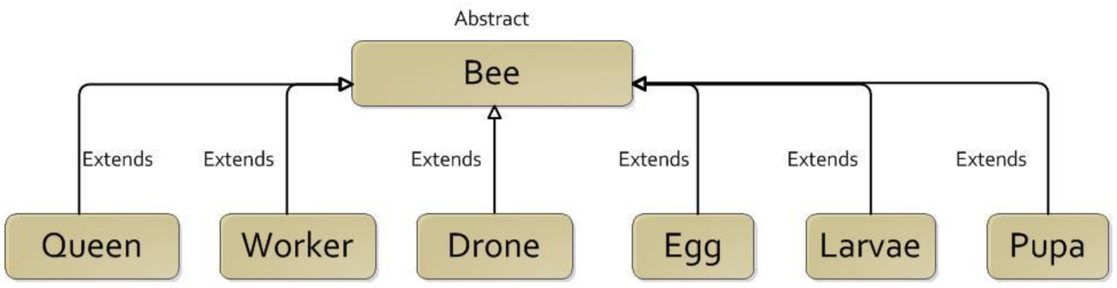


Fig 1: Class diagram for the Bee classes.

The first class to create is the *Bee* class. This is an abstract class that defines the basic properties and methods that all the different bee classes will use. The properties that you will need to define are:

- `type` – the type of the bee (Type IDs can be seen in Table 1 below.)
- `age` – To help model the developmental lifecycle it is important to know the age of the bee. This represents the number of days since the bee was laid as an egg.
- `health` – Some of the types of bee need to eat to stay healthy. A healthy bee has a health of 3. It loses one health per day unless it eats the right amount of the right food. If a bees health reaches zero, it sadly dies.

The abstract *Bee* class will also need to define some methods for use by the *Hive* and *beekeeper*. These methods can be overridden by the other bee sub-classes. The methods you need to create are *getType()*, *setAge(`int age`)*, *getAge()* and the abstract methods *eat()* and *anotherDay()*, both of which have a return type of `boolean`. *anotherDay()* will be called on each Bee in the hive once each day and will contain all the code that enables the bee to carry out its various tasks. *eat()* will be called at an appropriate time by anotherDay() to allow the bee to try and eat the required food and keep healthy.

You should now have a *Bee* class. As we progress you may choose to add additional methods to your abstract class.

Bee	ID number	Days old	Eats	Work
Queen	1	11+	Honey x 2	Lays an egg every three days
Worker	2	11+	Honey	Collects pollen Makes honey Makes royal jelly
Drone	3	11+	Honey	Does nothing
Egg	4	1-3	nothing (yolk)	Sits there
Larvae	5	4-6	Royal Jelly	Wriggles a bit
Pupa	6	7-10	nothing	Hides in cocoon

Table 1: Types of bee and relevant information

Part 2 – Modelling the Hive

Our *Hive* class is where all our *Bees* will live. You should use an `ArrayList` to represent the cells in the *Hive*. For the purposes of our simulation we will allow our Hive to have no more than 100 cells in it. The *Hive* also has stores of different types of food (we'll use integers amounts to represent the stores. The different types of food are honey, royal jelly and pollen. You can use a counter to record how much of each the Hive has. You might want to start your Hive with some food in it already.

The *Hive* needs a number of access methods. These should include:

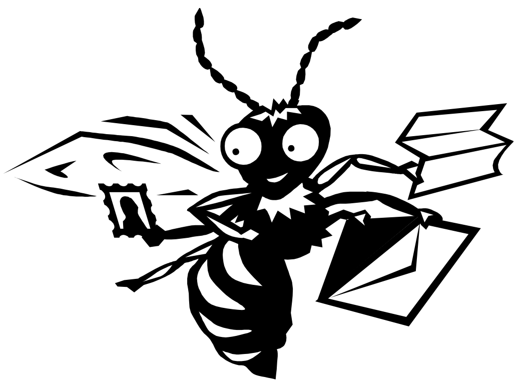
- *addBee(`Bee`)* – this adds a new *Bee* into the next available cell.
- *getBee(`int`)* – this returns the *Bee* in cell n, or null if there is no cell n.
- *size()* – this returns the number of *Bee*s in the *Hive*.
- *addHoney(`int`)*, *addRoyalJelly(`int`)*, *addPollen(`int`)* – are used to add food to the *Hive*.
- When extracting food, the following methods are used. Each takes as a parameter the amount of food required and either returns that amount if it is available or returns 0 if there is no food of that type to be had. *takeHoney(`int`)*, *takeRoyalJelly(`int`)*, *takePollen(`int`)*.
- *anotherDay()* – You'll fill this in later, this method is called as part of the simulation to trigger a new day in the *Hive*.

You should now have a *Hive* class that you can add Bees to.

We should now move on and create some actual *Bee*s to add to our *Hive*.

Part 3 – Modelling the Queen

Your *Queen* is an extension of the *Bee* class.



The *Queen* stays in the Hive and lays eggs. In reality, a well-fed queen might lay 2000 eggs in a day, but we'll keep things very simple for our simulation and say that a *Queen* will lay an egg every three days. Queen bees eat honey, and for our simulation the queen will need to eat 2 units of honey a day to stay healthy (two units of honey gives one health point back to the Queen.)

Create a class called *Queen* that extends *Bee* and is of type 1.

The *Queen* will need to know about her *Hive*, so add a constructor to the *Queen* class that takes a *Hive* as a parameter and stores it locally. The *Queen* can use this to call methods in the *Hive*.

Override the *eat()* method in your *Queen* class. Inside the class the *Queen* needs to take 2 units of honey from the *Hive* using the *takeHoney()* method. If this succeeds the Queen's health increases by 1, unless it is already at a maximum of 3, if it fails (there is no honey) then the Queen's health is reduced by 1.

Before working on the *anotherDay()* method for the *Queen* we will need to create the *Egg* class. The *Egg* class extends the *Bee* class and is of type 4. The *Egg* has little knowledge of its environment and so doesn't need to know about the *Hive*. It also doesn't need to eat (it just eats the yolk it was born in) so the *eat()* method should just return `true`;

The *anotherDay()* method for *Egg* needs to be overridden to increase the age of the *Egg* by one day before returning `true`. That's it for the *Egg*, they're very simple things.

Back with the *Queen*, we now need to override the *anotherDay()* method and carry out the duties of the *Queen*. Inside the anotherDay() method you will need to perform the follow actions, represented by the pseudo-code snippit below:

```
Add 1 day to the age of the Queen
Try and eat.
If it is three days since the last Egg
    Make a new Egg and add it to the Hive
```

You can use a counter to represent time since the last egg, or alternatively use the age of the *Queen* as a guide.

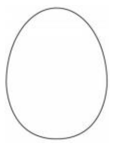
The final change you will need to make to turn this into a simulation is to add code to the *anotherDay()* method in your hive to cycle round all the cells in the *Hive* calling the *anotherDay()* method on the *Bee* inside it.

By this stage you should have a working *Hive*, which can take a *Queen* and that *Queen* can lay *Egg*s in the *Hive*.

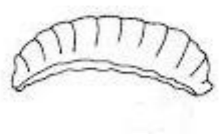
You can test your *Queen*, *Egg* and *Hive* using the test harness we have provided called TestBasicHive.java.

Part 4 – Modelling the Reproductive Lifecycle

As shown in Table 1. The *Queen* lays an *Egg*. The *Egg* turns into a *Larvae*, the *Larvae* turns into a *Pupa* and the *Pupa* turns into a *Worker*, *Drone* or *Queen*.



You will model this process inside the *Hive* by using the polymorphism of the *Bee* class. Your current *anotherDay()* method inside the *Bee* class returns a boolean to indicate whether the *Bee* is still alive or not. We need to modify the method so that instead of returning a boolean it will return a *Bee* object. If an



Egg is over 3 days old, it will need to hatch into a *Larvae*. You will need to create a new *Larvae* and replace the *Egg* in the cell with this *Larvae*. To do this, when *anotherDay()* is called on the *Egg*, if it is over three days old it should create a new Bee of type Larvae, and return this. The *Hive* can then replace the *Egg* in that cell with the new *Larvae*. If the *Egg* is less than three days old it should simply return itself (i.e. return *this*). Similarly, if a *Larvae* is over 6 days old it will return a new *Pupa* for it to be replaced with. When the *Bee* emerges from its *Pupa* cocoon on day 11, it would be a *Queen*, *Worker* or *Drone*. To keep things simple, our *Pupae* won't turn into *Queens*, they will either be a *Worker* or a *Drone*, with only a 20% chance of it being a drone. You can use some of the random number code you have encountered in the labs to decide whether the *Pupa* turns into a *Worker* or a *Drone*.

In order to model this you will need to create new extended *Bee* classes for *Larvae*, *Pupa*, *Worker* and *Drone*. Table 1. Gives details of what these types of *Bee* eat and what activities they get up to. We will implement the work of the *Worker* in a later Part so for now you just need to worry about it eating.

You should now have a working *Hive* class that can run a simple simulation with *Bee*s being born and evolving within the *Hive*.

In your Hive main() method you can loop round all the bees calling anotherDay() on them. You can use print statements to record what is happening in your simulation. To slow things down a little, the following code will allow you to pause for half a second between days if you choose to include it.

```
try
{
      Thread.sleep(500);
}
catch (InterruptedException e)
{
}
```

Your *Bees* won't last long though without more food, and to do that we need to put our Hive in a *Garden*.

Part 5 – Modelling a Garden.

Your *Garden* class contains a *Hive* and a collection of *Flowers*. The *Flowers* in the *Garden* can be held in an ArrayList.

There are a number of different types of *Flower* in the *Garden* and so we will model *Flower* as an abstract class again, with different *Flower*s as extensions of this. This will then give us the ability to have them *grow()* differently. Figure 2. Shows the class hierarchy we would like you to use for *Flowers*.

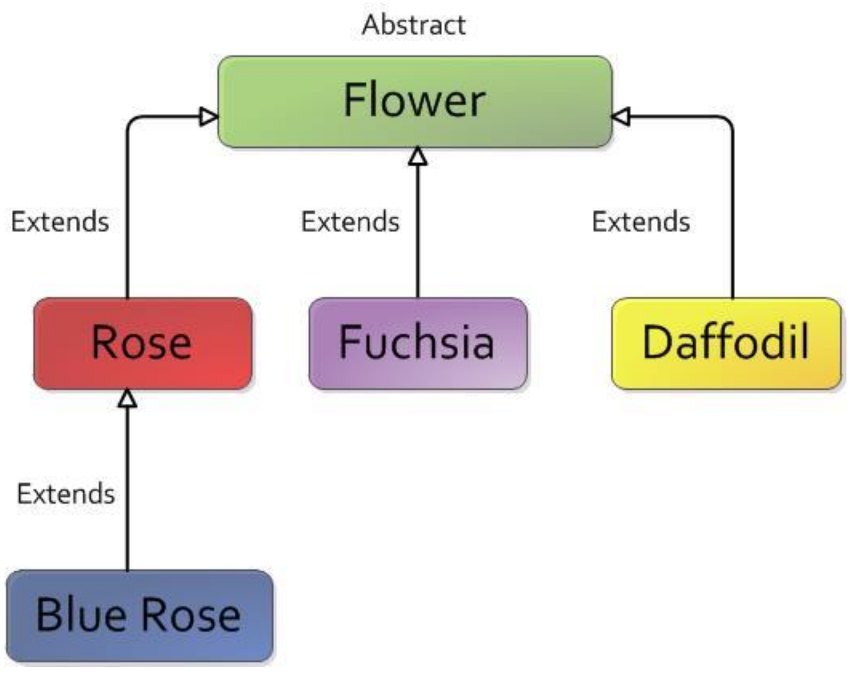


Fig 2: Class diagram for the Flower classes.

All Flowers have the following properties

- `pollen` – the amount of pollen on the flower.
- `type` – the type of the flower (1=Rose, 2=Fuchsia, 3=Daffodil, 4=B lue Rose.)
- `garden` – the garden they are growing in.
- `colour` – the colour of the flower.

Flower have the following methods

- *Grow()* This is called once a day and the flower will grow by increasing the amount of pollen it has.
- *extractPollen(*`int`*)* – This method is called by Bees to collect an amount of pollen. The return value indicates whether there is enough pollen for the Bee.
- And access methods *getPollen()*, *setPollen(*`int`*)* and *getType()*.

Different Flowers have different growth rates and colours as recorded in Table 2. below.

Flower	ID type	grows	Colour
Rose	1	+2 pollen per day	Red
Fuchsia	2	+1 pollen per day	Purple
Daffodil	3	+3 pollen per day	Yellow
Blue Rose	4	+1 pollen per day	Blue

Table 2: Types of flower and relevant information

With this initial information, the need to subclass Flower may not be readily apparent, but possible future extensions may require the over-riding of methods and so sub classing now will give flexibility later.

Your Garden will also require a number of methods. These will include

- *anotherDay()* – called to iterate over the *Flower*s to get them to *grow()*
- *addHive(`Hive`)* – used to place a *Hive* in the *Garden*
- *addFlower(`Flower`)* and *getFlower()*
- *findFlower()* – used to randomly return one of the *Flower*s in the *Garden*
- *size()* – returns the number of *Flower*s in the *Garden*.

Part 6 – Reading a simulation configuration file

A good simulation will allow you to set the starting conditions for your simulation and one way of doing this is for the simulation to read in a simple configuration file. For this step you will need to use your file handling methods as well as split strings into different component parts.

Our basic configuration file will look like the example below. You may chose to extend this for your extensions, but for testing purposes your code should accept and use configuration files in this form. Each line gives information about a bee in the hive or a flower in the garden.

```
hive:10,5,0
queen:3,15
worker:3,15
egg:3,1
rose:10
fuchsia:15
bluerose:12
```

The format for the *Hive* is

Hive:Honey,Royal Jelly,Pollen

for *Bee*s

Beename:health,days old

for *Flower*s is

Flowername:pollen

Some example simulation files of varying complexity will be found on the WIKI.

You should modify the main method in your *Garden* class so that it can take a file on the command line. This will enable you to start your garden by typing

```
java garden mygarden.txt
```

When the *Garden* receives the configuration it should read the file a line at a time. For each line the *Garden* will need to identify the Class, create a new class of this type, and set the appropriate parameters. If creating a new *Bee* it should add it to the *Hive*, if creating a new *Flower*, add it to the *Garden*. You may find you need to create specific methods or indeed a helper class, to parse the configuration file and extract the information that the *Garden* needs.

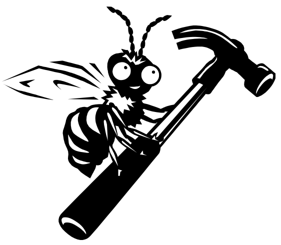
We are placing this code in the *Garden* to make it simple, so your simulation is of a *Garden* with a *Hive* in it. You could create a new Class *Simulator* or *Beekeeper* to perform this function if you wish, provided it is clear in the Readme.txt file you supply where the code can be found.

You should now have a simulator that runs with a *Garden* containing a *Hive* which has multiple *Bee*s in it.

Part 6 - Modelling the Drone and the Worker Bees

The final part of the Simulator, that you will need to create in order to keep the *Garden* and *Hive* growing and thriving is the *Worker*. The *Worker* is an extension of the *Bee* class and you will have created them earlier but left the *anotherDay()* method reasonably empty. You now need to fill this in.

First of all the *Drone*s, they're easy to simulate as the *Drone* bee does absolutely nothing except eat each day. You can find out how much they eat in Table 1. In reality the *Drone*s do serve a purpose in the *Hive* but we're not going to worry about that for the purposes of this coursework.



Worker bees however, are very busy. They are going to go out into the *Garden* and get some pollen from the *Flower*s. They will use the *findFlower()* on the *Garden* to find two different *Flower*s and will call the *extractPollen()* method on the *Flower*s. Depending on the

*Flower*s and how much pollen they have, the *Worker* will return with an amount of pollen.

Having collected some pollen they now need to turn this into other types of food. If they have at least 2 units of pollen then they can create one unit of Royal Jelly and store that in the Hive. Any remaining pollen is converted to honey at and stored in the Hive. Each remaining unit of pollen becomes one unit of honey.

Having completed all their work the Worker bee can now *eat()*.

You should now have all the *Bee*s you need and a functioning *Garden* and *Hive*. Congratulations, you've built your very own Bee simulator.

Exceptions

You should be trying to use exceptions in the construction of your simulator where possible. You should be catching appropriate I/O exceptions but also might might consider the use of exceptions to correctly manage:

- The input of a configuration file that does not conform to the specified file format.
- Attempts to add more than one *Queen*.
- A *Bee*s health reaching 0 ☹
- *Flower*s having no pollen left for *Bee*s to collect.
- …

Extensions

You are free to extend your code beyond the basic simulator as described above. You are advised that your extensions should not alter the basic structures and methods described above as marking will be looking to see that you have met the specification as we have described it. If you are in any doubt about the alterations you are making please include your extended version in a separate directory.

Some extensions that we would heartily recommend include:

- Try modifying the simulator so that *Bee*s have a lifespan in days and after a certain lifespan then some Bees just don't make it back to the *Hive*. This is not too dissimilar to what happens in Hives normally. Similarly, perhaps not all *Egg*s hatch into *Larvae*, and any *Bee* that doesn't get enough food doesn't make it.
- You might decide to make your *Garden* more realistic by simulating the pollinating of *Flower*s. If a *Bee* takes pollen from two *Flower*s of the same type, then there would be a chance that the second flower gets pollinated and lays a seed. This will give you a new *Flower* of that type in your *Garden*. Different *Flower*s could pollinate with different levels of success and at different rates.
- Try writing code that lets you save out the current state of the simulation in a configuration file. This would allow you to restart your simulation at the point it had reached last time rather than having to restart from a fresh each time.
- You might want to extend the configuration file and classes so some of the parameters of the simulation can be set in the configuration file. This might include some of the details in Table 1 and Table 2, such as how much food different *Bee*s

eat, how much pollen different *Flower*s supply. You would then be able to experiment with these different parameters without having to recompile your code repeatedly.

15% of the marks are available for implementing an extension. Any of the above examples would be suitable, but feel free to come up with one of your own if you like. It is not necessary to attempt multiple extensions in order to gain these marks. Please describe your extension in the readme file that you submit with your code.

Space Cadets

You might want to add a GUI to your simulator so that you can visualise the state of the *Garden* and *Hive* at any given moment. **No marks are available for a GUI**, we put the suggestion forward simply for the challenge of it.

Marking

Marks are available for:

- Applications that compile and run
- Meeting the specification properly
- Successful completion of one or more extensions
- Good coding style
- Good comments in your source code

85% of the marks are available without attempting any extensions.

Submission

Submit **all Java source files** you have written for this coursework in a zip file **beehive.zip**. Do not submit class files. As a minimum this will be:

Bee.java, Queen.java, Hive.java, Egg.java, Larvae.java, Pupa.java, Worker.java, Drone.java, Garden.java, Flower.java, Rose.java + any further flowers you may have chosen to model.

Also include a text file **beehive.txt** that contains a brief listing of the parts of the coursework you have attempted, describes how to run your code including command line parameters, and finally a description of the extensions you have attempted if any. Submit your files by **Wednesday 11th December 2013 17:00** to: https://handin.ecs.soton.ac.uk

Originality of Work

Students' attention is drawn to Section IV of the University Calendar on Academic Integrity:

http://www.calendar.soton.ac.uk/sectionIV/part8a.html