Danaus Manual

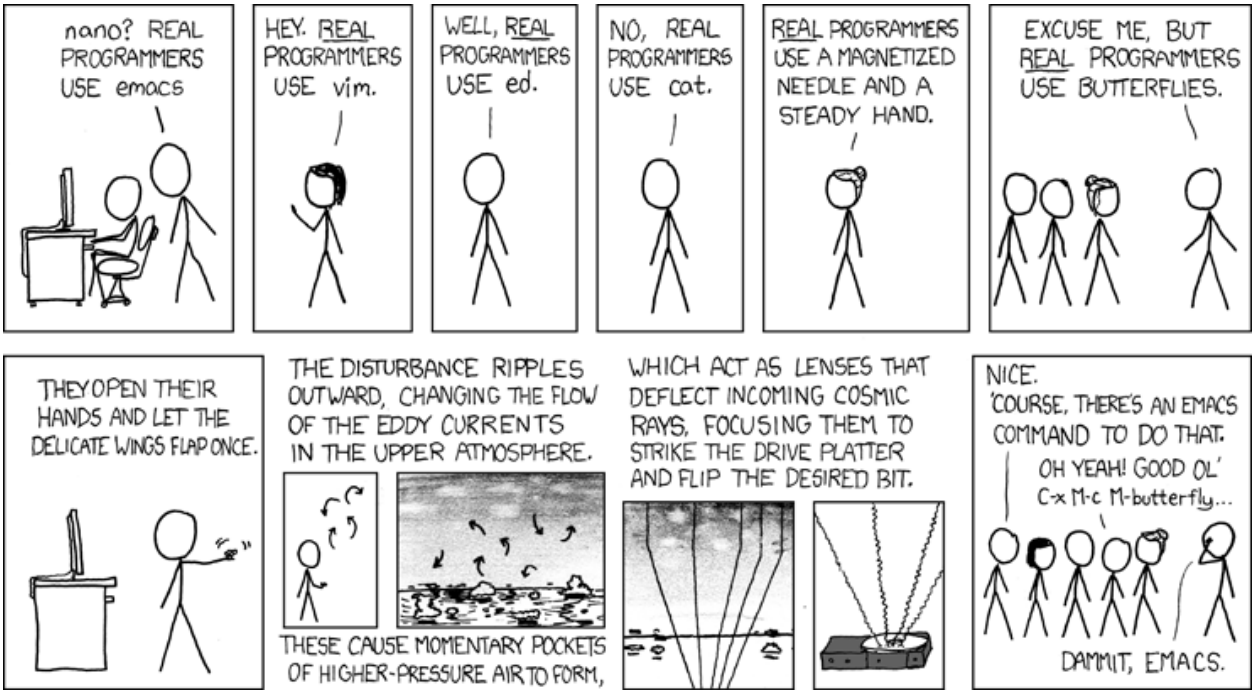

Figure 1: Evidence you are all **real** programmers.

Contents

1 Overview

1.1 Course Overview

Throughout the semester, you will be completing a large, three-part programming assignment in addition to several smaller, independent programming assignments.

The three-part programming series, known colloquially as Danaus[1], will expose you to a holistic, integrated form of object-oriented programming. For each assignment in the three part series, you will be adding to the code you have previously written to solve new challenges. Danaus will cover topics ranging from project design and object orientation to graph exploration and traversal to machine learning.

The smaller independent assignments will tax your skills with the fine-grained details of object oriented programming.

1.2 Danaus Overview

In a nutshell, Danaus is a butterfly flight simulation engine. We have provided a framework to generate pseudo-random maps, animate butterflies, and assess their performance. You will be designing and programming the logic behind the butterfly.

More specifically, yet with many details still elided, a simulation consists mainly of a map, a butterfly, and some information on the state of the simulation. A map is a toroidal grid of tiles. Each tile has a set of attributes: light, wind, etc. A butterfly is placed within a map and can explore the map by flying and landing. The goal of the butterfly is to collect a set of flowers, distributed randomly throughout the map, as quickly and as efficiently as possible. Throughout the simulation, various performance metrics are recorded, such as total turns and time taken.

This document elaborates on the basics of a simulation, the attributes of a map, the functionality of the butterfly API, etc. It is imperative that you read and understand this document in full before you begin to design and implement your butterfly.

2 Simulation Basics

This section provides an overview of Danaus' basics to help you form a big-picture understanding of Danaus. The basics discussed here are explained in greater detail in later sections.

2.1 Parks, Maps, and Tiles

Three important classes within Danaus are `Tile`, `Map`, and `Park`[2].

- `Tiles`, such as `Land` and `Water`, are the basic building blocks of Danaus. Butterflies travel to and from `Tiles`; `Aroma` and `Wind` are spread about `Tiles`; etc.

[1] The genus of a butterly. Not to be confused with the Greek god of the same name.

[2] `Maps` and `Tiles` are explained in much greater detail in Section 3.

- A `Map` consists of an array of `Tiles` and some other bookkeeping information. A `Butterfly` interfaces most heavily with a `Map`.
- A `Park` consists of a `Map` and some information about the state of a simulation. Butterflies do not directly interface with a `Park`, but a `Park` will record simulation statistics and performance metrics of the `Butterfly`.

The hierarchical nature of `Tiles`, `Maps`, and `Parks` is illustrated in Figure 2.

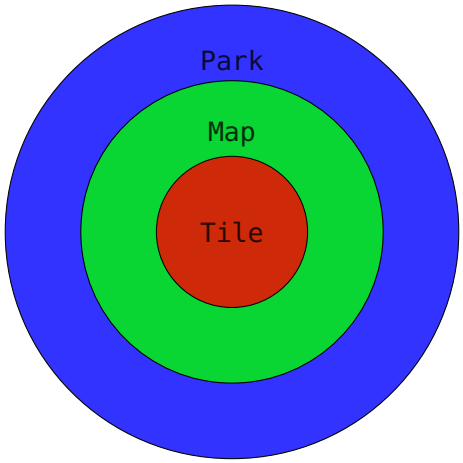

Figure 2: The hierarchy of Danaus structures.

2.2 Butterflies

A butterfly flies about a map collecting a prescribed set of flowers as efficiently as possible and in any order. A detailed description of flower collection is provided in Section 5.

2.3 Direction

Directions, of enum `Direction`, are the eight basic cardinal directions: N, NE, E, SE, S, SW, W, and NW. Each direction corresponds to one of a tile's eight neighbors, as shown in Table 1.

NW	N	NE
W		E
SW	S	SE

Table 1: The relationship between cardinal directions and tile neighbors.

3 Maps

3.1 Tile Types

A map is a toroidal grid of `Tiles`, each of which can be one of four subclasses.

3.1.1 Land

`Land`, the most basic type of `Tile`, is flat and flyable. Butterflies incur no cost when flying over land.

3.1.2 Forest

The tall and dense trees of a `Forest` tile are flyable but difficult to fly over. When a butterfly flies over a `Forest`, `slowSteps` is incremented by 1.

3.1.3 Cliff

A `Cliff` is towering and unflyable. When a butterfly tries to fly over a `Cliff`, a `CliffCollisionException` is thrown, and the butterfly's location does not change.

3.1.4 Water

`Water` is treacherous and unflyable. When a butterfly tries to fly over `Water`, a `WaterCollisionException` is thrown, and the butterfly's location does not change.

3.2 Tile State

Each tile also has a unique set of characteristics, as defined in class `TileState`. Each `Tile` contains a `TileState` instance. The following subsections explain the key fields of `TileState`.

3.2.1 Location

Each tile is identified by a `[row, col]` coordinate pair in an object of class `Location`. The coordinate system of `Locations` is identical to the indexing system of Java arrays. The top left `Tile` is `[0,0]`. Travel east and `col` increases. Travel south and `row` increases.

[0,0]	[0,1]	[0,2]
[1,0]	[1,1]	[1,2]
[2,0]	[2,1]	[2,2]

Table 2: A small map with annotated locations.

Though maps appear to be planar, they behave like a torus. Thus, a butterfly traveling off the east border of a map appears on the west border. Similarly, a butterfly traveling off the north border of a map appears on the south border. Figure 3 shows a toroidal Earth.

Figure 3: Toroidal Earth. Note that the Earth is not toroidal, and spheres cannot be arbitrarily converted to toruses. This is an illustration of what the Earth would look like *if it were* a torus.

3.2.2 Light

This feature has been removed.

3.2.3 Wind

This feature has been removed.

3.2.4 Flowers

Each tile has zero or more flowers (of class `Flower`), which can be accessed via `TileState`'s method `getFlowers()`. A flower has a unique long identifier, `flowerId`, and an initial aroma intensity of 1×10^6, and each flower radiates its aroma about the map[3]. The aroma at a distance d from a flower can be calculated as follows.

$$\text{aroma}_d = \frac{\text{aroma}_{\text{initial}}}{(1+d)^2}.$$

Here, d is the shortest distance from a tile to a flower. For example, consider the 3x3 map with no wind and with a flower planted in the center with an initial aroma intensity of 16.0 in Table 3 and Table!4.

[3]Really, a flower will radiate its aroma only up to `Integer.MAX_VALUE` steps away. We will never test you on maps of such a size, so you can assume that it radiates its aroma to all `Tiles`.

1	1	1
1		1
1	1	1

Table 3: Distances from the center of the map.

250000	250000	250000
250000		250000
250000	250000	250000

Table 4: Sample spreading of aroma without obstacles or map wrapping.

Figure 4 shows aroma propagation on a simple map without obstacles or wrapping. However, including these obstacles and wrapping does not complicate the propagation. The distance d for each tile is still its shortest distance to the flower.

Flower Invariants

1. **Each instantiated flower is unique**. Even if two flowers use the same image and occupy the same tile, they are distinct.
2. At most one instance of any given flower is on a map: no two flowers are the same.
3. Flowers may bloom (1) before `learn` is invoked or (2) after `learn` terminates and before `run` is invoked. **Once a flower blooms, it continues to bloom** at the same location for the duration of the simulation. Flowers that bloomed in `learn` are there during `run`.

3.2.5 Aromas

Each tile has zero or more aromas (of class `Aroma`), which can be accessed using `TileState` method `getAromas()`. An aroma has an intensity (in public field `intensity`) and an associated `flowerId`, a long that will match a `Flower` object, once that `Flower` is found.

Each aroma is associated with one specific flower. Aromas of different flowers are completely independent and are completely separate instantiations of `Aroma`.

4 Butterfly API

In order to implement your butterfly, you must first create a subclass of `AbstractButterfly`, which defines various public and protected methods that you use to control your butterfly. For more information on the butterfly API, refer to the javadoc specifications of the fields and methods.

4.1 void fly(Direction heading, Speed s)

Method `fly` attempts to fly your butterfly in `Direction heading` with `Speed s`. A butterfly can attempt to fly to any of its neighboring tiles using any of the eight cardinal directions.

A flight attempt can fail for several reasons. First, if a butterfly attempts to fly into a `Cliff` or over `Water`, an `ObstacleCollisionException` exception is thrown. The `Location` of the `Butterfly` is unchanged.

4.2 void refreshState()

Every instance of a subclass of `AbstractButterfly` has a `TileState` field named `state`. When `refreshState` is called, `state` is updated with the `TileState` of the tile the butterfly is currently on.

Note that if a butterfly flies to a tile, the butterfly's state is **not** automatically updated. For example, `state.location` will contain the wrong information. It is up to you to call `refreshState()` if you want access to the current tile's `TileState`.

4.3 int getMapWidth()

Method `getMapWidth` returns the number of columns of the map the butterfly is on.

4.4 int getMapHeight()

Method `getMapHeight` returns the number of rows of the map the butterfly is on.

5 Learning and Running

Machine learning[4] is a branch of artificial intelligence in which a programs can learn from a given set of data. Formally, a program is said to learn if its performance as some task **T** improves with some experience **E** as measured by some performance metric **P**[5]. In danaus, E, T, and P can be described informally as follows.

[4] For more information on machine learning, refer to the online resources of Cornell's Introduction to Machine Learning course: CS 4780

[5] Machine Learning, Tom Mitchell

- **T** Collect an ordered set of flowers on an unfamiliar map.
- **E** Exploring maps populated with flowers.
- **P** Danaus includes a multitude of performance metrics.

Many machine learning algorithms consists of two phases, which we will refer to as learning and running. During learning, the algorithm receives and processes sets of training data. This experience is associated with E. During running, the algorithm attempts to perform well at task T as measured by P.

As a machine learning framework, Danaus simulations also have two phases: learning and running. AbstractButterfly declares these phases as two methods: `TileState[][] learn()` and `void run(List<long> flowerIds)`. When Danaus is started, it sets up the required infrastructure to execute a simulation. Once this is complete, it calls your method learn(). After some computation and map modifications, it call method run(). The details of these two methods and the flow of a simulation are provided below.

These two phases are the two key components of a simulation. Once you understand these two phases, you will understand what exactly your butterfly must do in order to successfully complete a simulation.

5.1 Learning

Before `learn()` is invoked, Danaus sets up the appropriate framework. First, it parses arguments to method `main`. It decides whether to instantiate a GUI. It randomly generates a map or parses a map from a map file. It instantiates an instance of your butterfly and places it on the map. It begins a timer and invokes your implementation of learn().

The purpose of `learn()` is for a butterfly to explore and save the flyable tiles of a map. It does so by generating and returning a two-dimensional array of `TileStates`, whose flyable tiles must match the map's `TileStates`. Danaus then determines how well the map and the returned array match. `TileStates` do not have to be provided for `Cliff` or `Water` tiles. Any `TileState`, or null, can be provided and will be counted as a correct.

During the learning phase of a simulation, you **should not and cannot** collect flowers. An attempt to collect a flower during `learn()` will throw a `PrematureCollectionException`, and no flower will be collected. Of course, the list of flowers can be constructed, as in A5.

Instead, focus on collecting the required information about the map as efficiently as possible. You are required to return a two-dimensional array of `TileStates` that contains the information in all flyable states of the map. It is up to you to determine how to explore the map to do this.

5.2 Running

After the `learn()` phase, Danaus shifts to the running phase of the simulation. First, Danaus randomly adds more flowers to the map and spreads their aromas. (Other than the new flowers and aromas, the map does not change. The location of the butterfly and all the information it has saved up to this point is also maintained.)

Next, Danaus randomly generates a list of flower id's and passes this list to your method `run()`. The list may contain some flower id's belonging to flowers that were initially on the map as well as flowers that were added after `learn()` terminated. The goal is to collect the associated flowers as efficiently as possible — **the order in which they are collected does not matter.**

Before having the butterfly fly around and collect flowers, your method `run()` should use the two-dimensional `TileState` array that was constructed during the `learn()` phase to figure out how to fly around and collect flowers as efficiently as possible. There are many ways to do this. Coding a correct, readable, well-documented, and efficient butterfly during the `run` phase is the core challenge and fun of Danaus.

Once `run` terminates, Danaus checks the collected flowers against the list of flowers given to `run`. If your butterfly collected every flower **(and only those) in any order**, you successfully complete the simulation.

6 Command Line Options

Danaus has various command-line options that affect execution of a simulation, and you may want to alter the execution by providing some of them. Most students will be running using Eclipse, and we now explain how to give command-line arguments in Eclipse.

With the Danaus project selected in the Package Explorer, click **"Run"** at the top of the screen. Then, click **"Run Configurations"**. A windowed menu pops up. You see a tab titled **"Arguments"** near the top of the screen next to the **"Main"** and **"JRE"** tabs. Click on the **"Arguments"** tab. The topmost text box is titled **"Program arguments:"**. This is where you enter your command line options.

Here is a list of the options Danaus provides. Arguments can appear in any order. Text in bold should be entered literally. While italicized text should be replaced with the appropriate arguments.

- **`--help`**
  Print a friendly help message and exit the program.
- **`-h, --headless`**
  Run Danaus without a GUI.
- **`-d, --debug`**
  Danaus keeps track of various debugging information. When this option is provided, the debugging information is printed to the screen.
- **`-w, --warning`**
  Danaus keeps track of various warning information. When this option is provided, the warning information is printed to the screen.
- **`-i, --infinite`**
  This feature has been removed.

- **`-s, --seed <seed>`**
  Danaus uses a single psuedo-random number generator to generate its randomness. A psuedo-random number generator creates a string of apparently random numbers from an initial seed. If you specify a seed, Danaus repeatedly produces the "random" map generated from that seed.

- **`-f, --file <mapfile>`**
  If a map file is provided, Danaus generates the map from the map file instead of randomly generating the map.

In addition to these command-line options, you can also provide command-line arguments. Unlike options, arguments do not require a "`-`" or "`--`".

The only command line arguments Danaus accepts are the names of the butterfly classes you want to run the simulation with. For example, if `CornellButterfly` is a subclass of `AbstractButterfly`, you can run Danaus as follows:

```
1  java danaus.Simulator student.CornellButterfly
```

Or, if running Danaus from Eclipse, simply put "`CornellButterfly`" in the **"Program arguments:"** tab, as described above. If several class names are provided, the first class is used. If no class names are provided, the name "`Butterfly`" is used by default.

For more information on Danaus' command line options, refer to Danaus' man page located in `/doc/man`. For more information on man pages or if you are having trouble using the Danaus command-line options, refer to Google, Piazza, or ask a professor, TA, consultant, or friend.

7 Some Advice From Me To You

Danaus can seem like a large and insurmountable challenge at first, especially for students learning how to program. A maelstrom of graph exploration, Danaus can be an intimidating beast. However, Danaus isn't as scary as it seems. Retrospectively, you will likely find the experience easy and fun. Here is some advice to help alleviate stress and ensure you learn a lot, have fun, and get a good grade.

- **Start early!** If you're a particularly adept programmer, you may be tempted to complete Danaus' assignments the night before they are due. This won't work.

- **Read, read, read.** Your implementation may be beautiful and efficient, but if it doesn't do what is required, you have little chance of success. Make sure you read through what we expect you to do and how we expect you to do it.

- **Plan.** Architects don't build buildings until they've drawn blueprints. Computer scientists should be no different. Think about your algorithms before you write them. Talk them over with your partner, if you have one.

- **Get help.** If you've read this document early and planned things out but are still confused, ask someone for help. You may have a friend that has a better understanding of what to

do. If not, ask a consultant, TA, or professor. Go to their office hours or send them an email. You have an army of staff ready, willing, and eager to help you out.

- **Adjust to your comfort.** If you are uneasy about your programming skills, write a simple solution. If you are more comfortable with programming, write an efficient solution. If you are an "1337 h4x0r", then optimize your code until the cows come home. No matter what level of expertise you are, adjust Danaus to meet your needs. You can get a good grade on the assignment no matter how adept you are.
- **Have fun and learn a lot!** CS 2110 is the foundation for your career as a computer scientist. Enjoy it and learn from it!

8 Credits

Many thanks to all those who contributed to the creation of Danaus and allowed us to use their work. Without them, Danaus would not be as complete or as elegant as it is with their contributions.

- Flower sprites were taken from neorice.
- Butterfly sprites were taken from David Nyari.