Danaus Manual

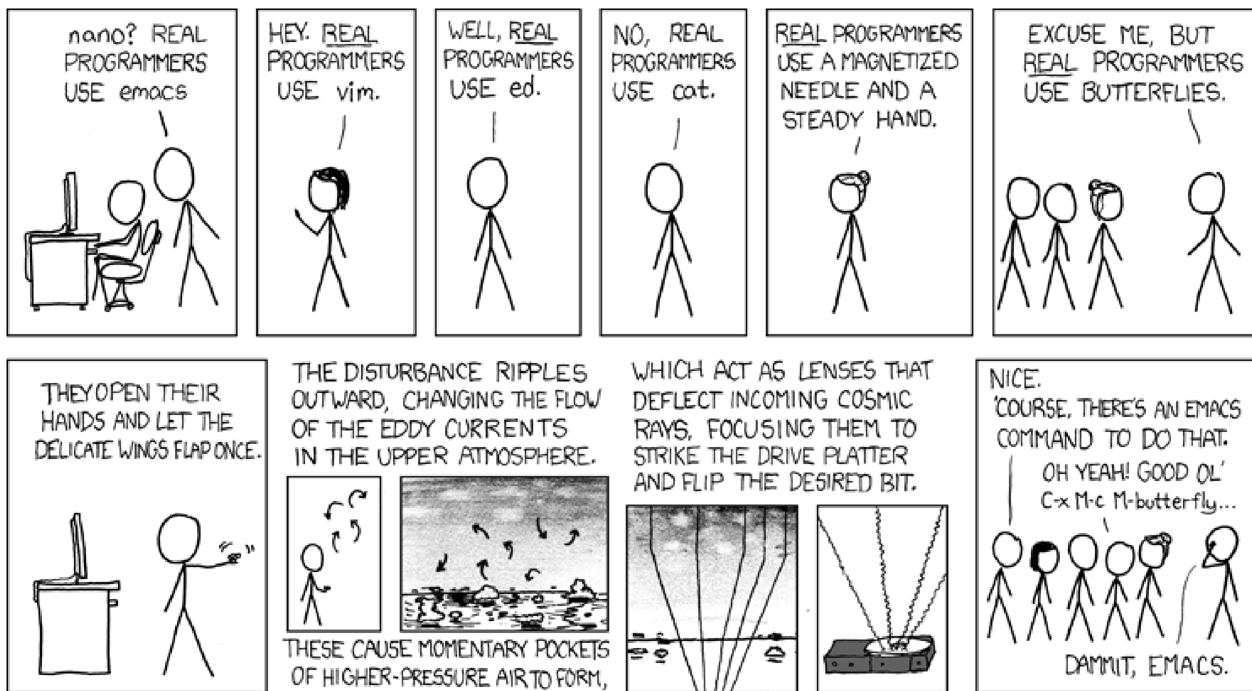


Figure 1: Evidence you are all **real** programmers.

Contents

1 Overview **4**
1.1 Course Overview 4
1.2 Danaus Overview 4

2 Simulation Basics **4**
2.1 Parks, Maps, and Tiles 4
2.2 Butterflies 5
2.3 Power 5
2.4 Direction 5
2.5 Speed 6

3 Maps **6**
3.1 Tile Types 6
3.1.1 Land 6
3.1.2 Forest 6
3.1.3 Cliff 6
3.1.4 Water 6
3.2 Connectivity 6
3.3 Tile State 7
3.3.1 Location 7
3.3.2 Light 7
3.3.3 Wind 7
3.3.4 Flowers 8
3.3.5 Aromas 9

4 Butterfly API **9**
4.1 void fly(Direction heading, Speed s) 9
4.2 void land() 9
4.3 boolean collect(Flower f) 9
4.4 void refreshState() 10
4.5 Power getPower() 10
4.6 int getMapWidth() 10
4.7 int getMapHeight() 10

5 Learning and Running **10**
5.1 Learning 10
5.2 Running 11

6 Command Line Options **11**

7 Credits **12**

1 Overview

1.1 Course Overview

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

The multi-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`.

- `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 butterfly. Not to be confused with the Greek god of the same name.

- 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.

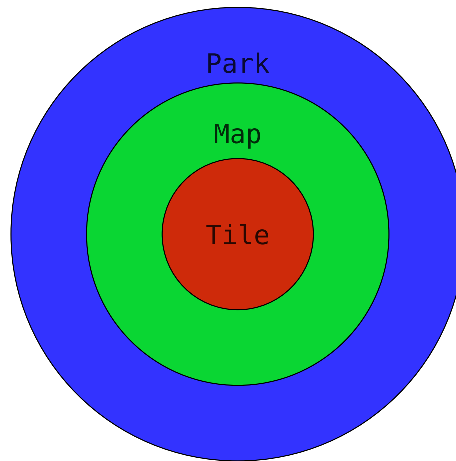


Figure 2: The hierarchy of Danaus structures.

2.2 Butterflies

During a simulation, a butterfly both learns and runs on a map. When learning, a butterfly exhaustively searches a map, collecting as much information about the map as possible. When running, a butterfly collects a set of flowers in as few turns as possible.

2.3 Power

This feature has been removed.

2.4 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.

2.5 Speed

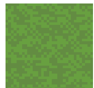
This feature has been removed.

3 Maps

3.1 Tile Types

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

3.1.1 Land



`Land`, the most basic type of `Tile`, is flat and flyable.

3.1.2 Forest

The tall and dense trees of a `Forest` tile are flyable.

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 Connectivity

All maps are guaranteed to be connected. That is, there exists a path between all pairs of flyable tiles. This implies that it is always possible for a butterfly to explore every flyable tile in a map.

3.3 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`.

3.3.1 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.

3.3.2 Light

This feature has been removed.

3.3.3 Wind

This feature has been removed.

3.3.4 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[2]. 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 3×3 map with a flower planted in the center as shown in Figure 3 and Figure 4.

1	1	1
1	0	1
1	1	1

Table 3: Distances from the center of the map.

250000	250000	250000
250000	1000000	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

[2]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. **Each instantiated flower is unique**. Even if two flowers use the same image and occupy the same tile, they are distinct. A corollary of this invariant is that **no two `Flowers` are equal as determined by `Flower.equals()`**.
2. 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`.

3.3.5 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 land()

This feature has been removed.

4.3 boolean collect(Flower f)

During the `run()` phase, the goal of a butterfly is to collect a set of flowers. Method `collect` attempts to collect `Flower f` in the tile the butterfly is currently on. If `f` is planted on the tile, it is collected and `true` is returned; if `f` is not planted on the tile, `false` is returned.

4.4 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`.

4.5 Power getPower()

This feature has been removed.

4.6 int getMapWidth()

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

4.7 int getMapHeight()

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

5 Learning and Running

Danaus simulations 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.

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 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.

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 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.