**Spike:** Task 31

**Title:** Custom Project

**Author:** Sam Huffer, 101633177

# Custom Project Plan

This year as part of the capstone units for my BA. Games and Interactivity, along with the rest of my teammates in Under Ctrl, I have contributed to the development of the game *Get the Fog Out*, a small-scale RTS where you seek to repair your ship and escape a desolate planet before a hostile fog can kill you. I was one of the programmers for it, programming the fog, tutorial, and dialogue system.

For my custom project, I planned to dissect scripts related to those areas in a post-mortem of my contributions to *Get the Fog Out*. Referring to data structures and software patterns mentioned in the lectures, I planned to go through those scripts and identify where particular structures and patterns had been used currently or in the past, how they were implemented and why, and - knowing what I do now thanks to this unit - whether those choices of structures, patterns and their implementation were optimal, why or why not, and any improvements I would make were we to recreate *Get the Fog Out* again from scratch or further develop it after submission.

# Intended Learning Outcomes

1. **Design**: Discuss game engine components including architectures of components, selection of components for a particular game specification, the role and purpose of specific game engine components, and the relationship of components with underlying technologies.
2. **Implementation**: Create games that utilise and demonstrate game engine component functionality, including the implementation of components that encapsulate specific low-level APIs.
3. **Performance**: Explain and illustrate the role of data structures and patterns in game programming, and rationalise the selection of these for the development of a specified game scenario.
4. **Maintenance**: Explain and illustrate the role of data structures and patterns in game programming, and rationalise the selection of these for the development of a specified game scenario.

# Links

* *Get the Fog Out*’s Itch.io page, with further information and a build available for download: <https://underctrlgames.itch.io/get-the-fog-out>
* *Get the Fog Out* Build: <https://drive.google.com/open?id=1g-kbjxa0kVx5RDpjsmL9vFlKOIz0-2NH>
* *Get the Fog Out* Source Code: <https://drive.google.com/open?id=1RWaMqF8KwiObDIKwVbxbsrx3JdWmEQL7>

# Fog

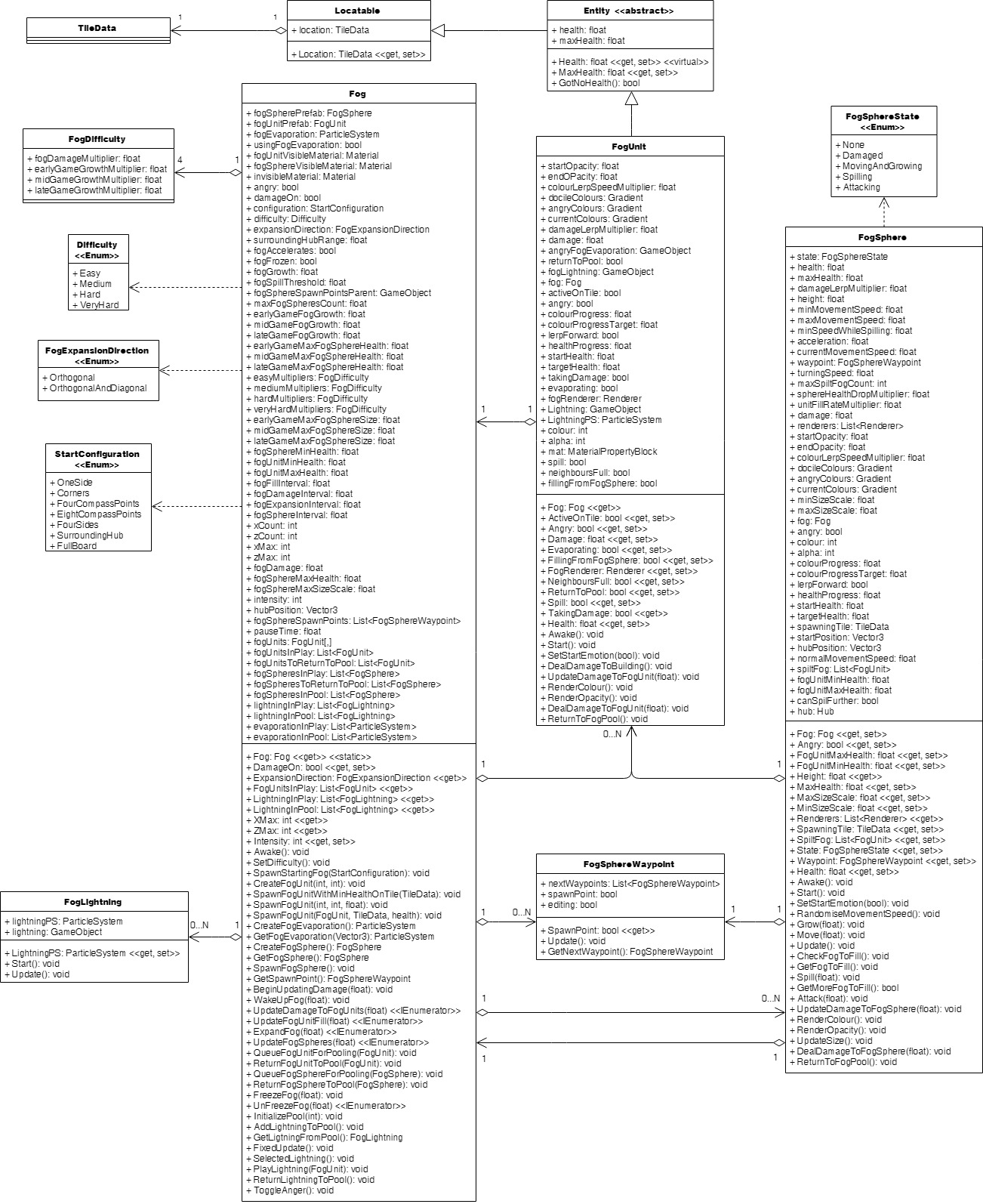


Figure 1: the current structure of the fog-related classes.

## Data Structures

### FogUnits: List, 2D Array

Figure 2: Fog.UpdateFogUnitFill() under the profiler while all FogUnits were being updated every time the method was called.

Originally, about 2500 FogUnits in Fog were split between two lists, one of the FogUnits currently in play, and the other of FogUnits that had been pooled, with FogUnits being allocated to TileDatas at runtime and drawn

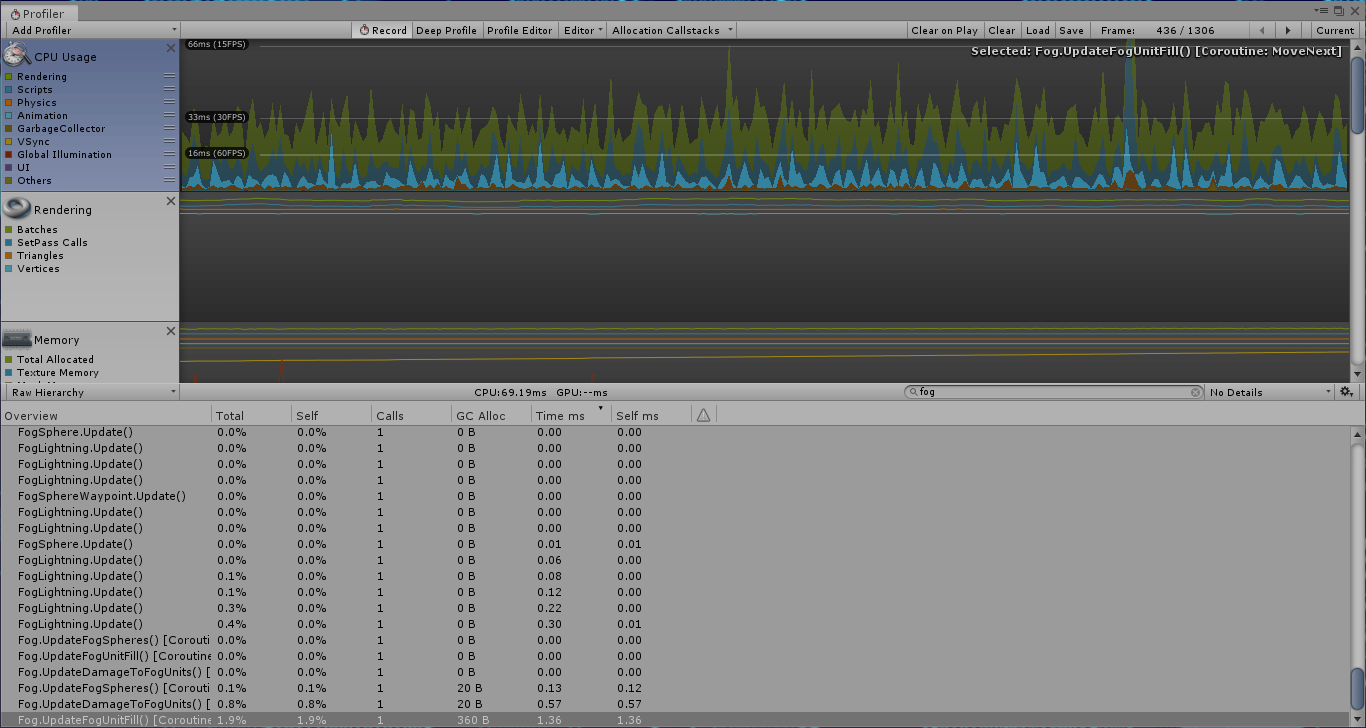
from the FogUnit pool, and put back there when destroyed. The result: a very slow game. Even running update methods less often than every frame (e.g. every 0.25 seconds for UpdateFogFill()) still had regular and noticeable drops in performance (fig. 2), which became more prominent after we increased the map size from 51x51 (2601 FogUnits) to 71x71 (5041 FogUnits). Eventually, we tried a combination of a) storing all FogUnits in a 2D array, with FogUnits’ positions in the array matching their corresponding TileData’s position in the 2D array of TileDatas, and accessing the array whenever interacting with a specific FogUnit of a known position (e.g. to check if it was active), b) removing the fogUnitsInPool List but still keeping all active FogUnits in the List fogUnitsInPlay for processing all active FogUnits one after the other, and c) spreading out the updating of FogUnits as part of UpdateFogUnitFill() over all frames in the 0.25 second interval, rather than updating all of them within the one frame. The combined results of these decisions led to a much smoother framerate, with minimal spikes in resources usage and no noticeable drops in frame rate attributable to FogUnits (fig. 3). If I were to do this project again, I would want to use this setup right from the start, especially if I knew there would be thousands of FogUnits. If the number was much lower, a List might be okay performance-wise, but this would still be preferable.

Figure 3: UpdateFogUnitFill() as an IEnumerator updating chunks of FogUnits each frame over the whole 0.25 second interval, rather than updating all of them within the one frame.

### Fog Spheres: List

From their addition until the final submission of *Get the Fog Out*, FogSpheres were stored between an in-play List and a pooled List, with FogSpheres swapped between them at runtime as they were brought into play or died. When I was adding the 2D array for the FogUnits, I didn’t feel it would be helpful or necessary for the FogSpheres to be put in a 2D array since a) FogSpheres weren’t tied to particular tiles or positions and b) there were only a dozen or so of them, so searching through lists to check if they were active or not wouldn’t be particularly time consuming. (If there were going to be hundreds or more, I’d only add a bool flag to FogSpheres, as concern a) would still be a consideration.) Managing the FogSpheres between these two lists proved simple and elegant, with negligible effects on performance thanks to their low number.

### Difficulty Modifiers: Struct

I stored the modifiers for various floats at a given level of difficulty in a struct with appropriate fields. I found this solution to be clean and the structs (once created) to be interchangeable, as the appropriate struct merely needed to be accessed to determine values for that difficulty when the difficulty would be set.

### FogSphere Spawn Points: List

The FogSphere spawn points I stored in a List of FogSphereWaypoints (as the spawn points were just FogSphereWaypoints with the “spawnPoint” bool checked in the inspector), allowing for flexible selection of any FogSphereWaypoint from the List, seeing as they were only used when picking a random FogSphereWaypoint for a FogSphere to spawn at.

## Design Patterns

### State Pattern

Figure 4: the informal “stages” of Fog and FogUnit.

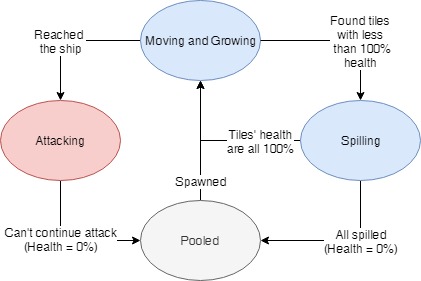
The state pattern was informally used with the Fog and FogUnits, but more explicitly implemented in FogSpheres. In the case of the informal implementation, Fog had an integer Intensity property (fig. 4) that when updated would update the rate of growth for FogUnits and the maximum size and health of FogSpheres (fig. 5). As all the values updated were stored in Fog itself, this seems a clean enough implementation to retain in future barring other extensive restructures.

Figure 5: an excerpt from Fog.Intensity.

More formally, the FogSpheres had distinct states of behaviour corresponding to values of the FogSphereState enumeration; during UpdateFogSpheres(), Fog would run the FogSphere’s state through a switch and call FogSphere’s methods as appropriate for that state. Each would execute its behaviour and check if conditions were met to update the FogSphere’s state (fig. 6). It worked cleanly enough, but leaves room for improvement that I will discuss below.

Figure 6: FogSphere’s states.

### Singleton Pattern

The singleton pattern was used extensively throughout *Get the Fog Out* to allow access to various controller or manager classes from anywhere that needed it without having to have it be a field of the class. Fog was one of those manager / controller classes, controlling the FogUnits and FogSpheres, so its Instance property and Awake() method were programmed to implement the singleton pattern as would best fit implementation in Unity. The other effect of ensuring only one instance of Fog was running at a time was also important; given the high number of FogUnits, a duplicate Fog with its own FogUnits would cause performance to drop unacceptably. As such, this is an aspect of Fog that I would not alter in any future updates to *Get the Fog Out*.

### Factory Pattern

Though not split into a separate class, the Fog class featured factory pattern methods for standardised creation of FogUnits and FogSpheres when the fog was initialised, and for instantiation / activation of FogUnits and FogSpheres as they were spawned. Programmatically this helped keep Fog’s code clean when it came to creating or spawning bits of fog, as only one of a handful of methods would need to be called. However, the inclusion of those methods in Fog rather than a dedicated FogFactory class wasn’t an ideal choice, as it resulted in the class being much longer (and that much harder to search through) than it could have been.

### Component Pattern

While I did not implement the component pattern with the Fog, FogUnits and FogSpheres, Unity’s GameObjects do make use of it with components like Transforms and Renderers. Consequently, it was a part of how they worked by default.

## Usability Patterns

### State / Progress Patterns

Alongside the informal programmatic states of the fog discussed above, the state and / or progress usability pattern was also informally implemented through the changes in colour applied to FogSpheres and FogUnits at different points (fig. 7): when the Fog was woken up, it changed colour from grey to blue and grey, and again from blue and grey to red and grey when the player attached their ship’s missing wing to the rest of their ship. These changes in colour corresponded to and conveyed to the player changes in the Intensity of the Fog’s behaviour due to the player’s progress through the game: from asleep to awake and approaching, and from approaching slowly to more aggressively.

## Restructure

Were I to restructure, update or redo the fog classes, there are a number of changes that I would make. There are a number of methods and options that were made available but never made use of (e.g. FogSphere’s UpdateDamageToFogSphere() and DealDamageToFogSphere() methods, and the orthogonal-only FogExpansionDirection value and the corresponding code in Fog.ExpandFog(). Where they would not be required or were obsolesced by design decisions made for *Get the Fog Out* and would not be made use of again, I would remove them. Though the savings in compilation time, speed at runtime, and file size would be negligible, they would be savings nonetheless. More substantially however, removing that unnecessary content would reduce the methods, lines of code, etc. that developers would have to search through to while fixing bugs and updating code.

On that same note, the Fog, FogUnit and FogSphere classes are rather long, especially the former. Fog and FogSphere have members that could be removed and turned into their own classes, while FogUnit and FogSphere feature methods that are near identical yet they don’t share a common ancestor in the inheritance hierarchy or make use of common components, resulting in duplicated code. To address these issues in a further iteration of *Get the Fog Out*:

Figure 7: the progression of the colour of the fog, from a black-ish grey (asleep) to blue (awake) to red (the final stage of the game).

* I would break Fog up into manager classes for FogUnits, FogSpheres and FogLightning, as well as a factory class that could instantiate and do basic configuration of each of those parts of the fog for their respective manager classes. Each of the manager classes and the FogFactory class would be a singleton to ensure their uniqueness and universal access to them.
* I would dismantle FogUnit’s inheritance structure (FogUnit 🡪 Entity 🡪Locatable) and instead use those separate classes as components of FogUnits. In the case of Entity, this would boil it down to just a Health component that FogSphere could also use and remove some of FogSphere’s methods that a Health component could take care of. FogUnit and FogSphere’s common rendering members could also be extracted and made into a separate class for use as a component of FogUnit and FogSphere.
* I would amend *Get the Fog Out*’s current implementation of the state pattern and go for a more modular state / strategy pattern approach, splitting the methods associated with each of FogSphere’s states into separate classes that operate on FogSpheres and inherit from a common base state class, with instantiated objects of those child classes able to be plugged in interchangeably as FogSpheres’ current states.

While breaking up these classes like this would create a more intricate network of which class uses what and who contains copies of who, and the more prevalent use of public properties this would necessitate might slow down the game (albeit negligibly), it would also result in smaller versions of the existing classes that have more clearly defined and singular purposes, those classes being more readable and more easily searchable during debugging, and bugs more pinpoint-able to particular classes rather than somewhere in a mega-class, therefore making them more manageable and easier to maintain.

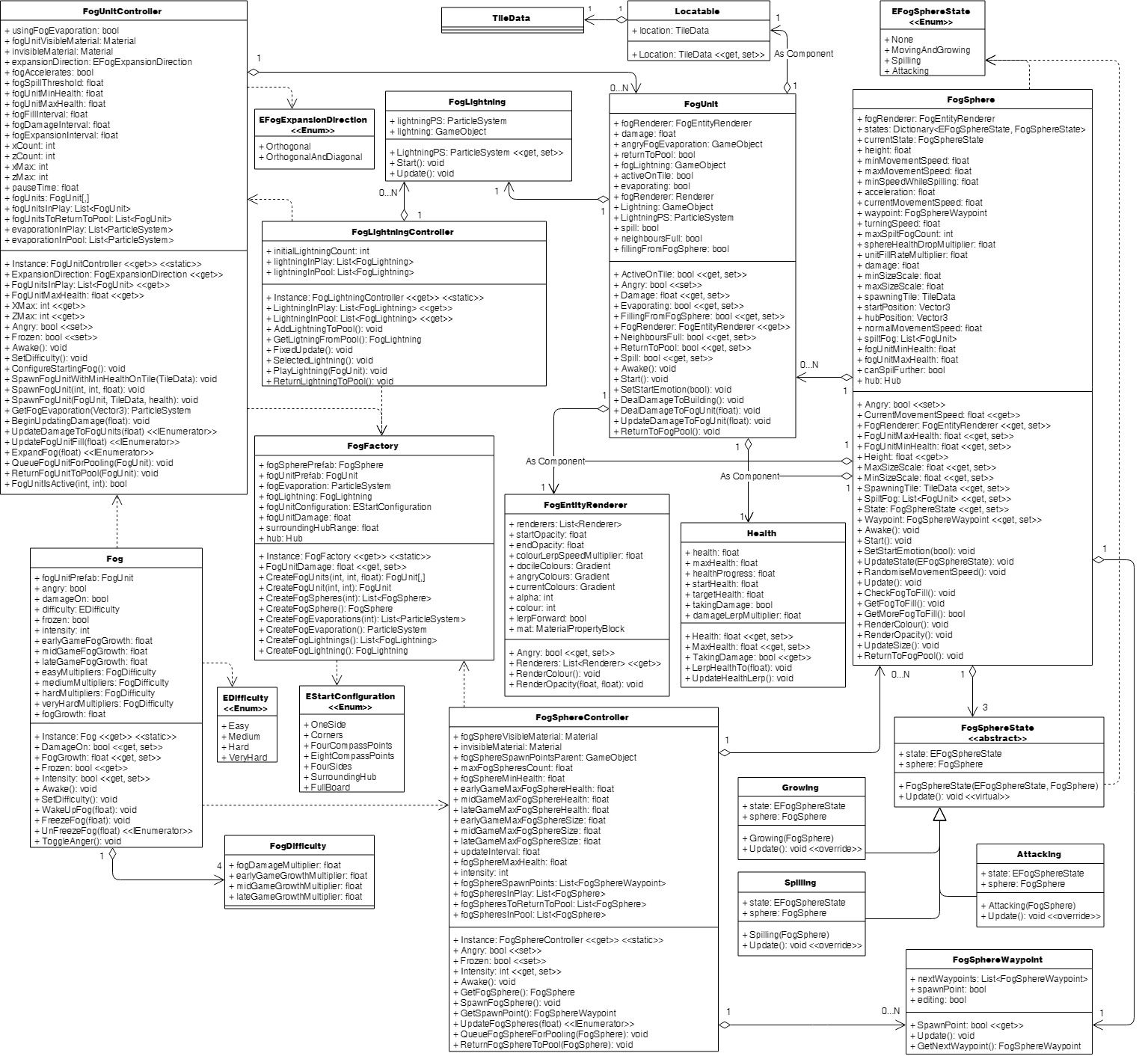


Figure 8: how I would restructure the fog classes to be more maintainable.

# Tutorial

Figure 9: the current structure of the tutorial-related classes.

## Data Structures

### Lerp Targets Remaining: List

Surprisingly, TutorialController only has one field that is a collection of items: lerpTargetsRemaining, the list of the Locatables to which upcoming tutorial stages will move the lerp target in order to prompt players to do something at that location (fig. 10). I stored them in a list to allow them to be added and removed quickly and painlessly. Searches of the list to check if a Locatable is still awaiting use will only have to run through a relatively small number of elements, so any loss of speed from using this over another data structure that would be more efficient is negligible, particularly compared to the ease of insertion, removal and searching. If a more efficient but equally or more easily usable data structure were available, it might be worth switching to that, but for the moment Lists are perfectly adequate, so I would leave this choice of data structure as is.

Figure 8: the activate sonar stage of the tutorial; the player is zoomed out and being prompted to activate the sonar (whose area of effect is yellow) within the lerp target (blue).

### ButtonsNormallyAllowed: List

The method ButtonsNormallyAllowed() compiles a List of buttons that would normally be operable when interacting with a particular tile. It compiles it as a List at runtime, and which buttons are usable on that tile aren’t guaranteed, so having an array of a fixed length would be less appropriate. Using Lists also affords the flexibility and ease of addition and removal I mentioned above. The choice of data structure, I wouldn’t change.

Structurally, however, it’s arguably more appropriate for this to be a member of the TileData checked rather than TutorialController. If it were, anything that needs to check what would normally be allowed would only have to access the TileData, not pass it to TutorialController and wait for it to do the processing. (Anything assessing what buttons would *currently be* allowed would still have to go past TutorialController.) Were it done this way, I’d have it check if it had previously compiled the List, compiling it and storing it if it hadn’t before returning the stored List. (To avoid tiles with no ResourceNodes only allowing Harvesters to be built after their ResourceNodes were fully mined and their corresponding Harvesters dismantled, adding a bool flag to indicate the button list is out of date would be necessary.) Compiling the list on request in the TileData itself wouldn’t increase the burden on the computer, as it’s already compiling these lists at runtime, and would keep the initial load time down as 5000+ TileDatas wouldn’t be all processing information at the start. Storing the list upon compilation and merely retrieving it upon subsequent requests would improve efficiency as the list wouldn’t be recompiled every time it gets requested, only being compiled once, twice at the most if that tile had a ResourceNode.

### Local Data Structures of Methods: List

## Design Patterns

## Restructure

* Renaming stuff to reflect more appropriate naming / current terminology that hasn’t been updated in-code.

# Dialogue System

## Data Structures

## Design Patterns

## Restructure

Screenshots and diagrams to back up

Make speculations on future design that relate to the four learning outcomes, back up with discussion of patterns to use and graphs and charts of the design

Email Tien with more complete draft on doubtfire, sort out feedback then

Fair game to list by tasks in LSR and address outcomes they address, rather than by outcome and listing the tasks they address