Distinction Report Key Points

# Fog

## 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.

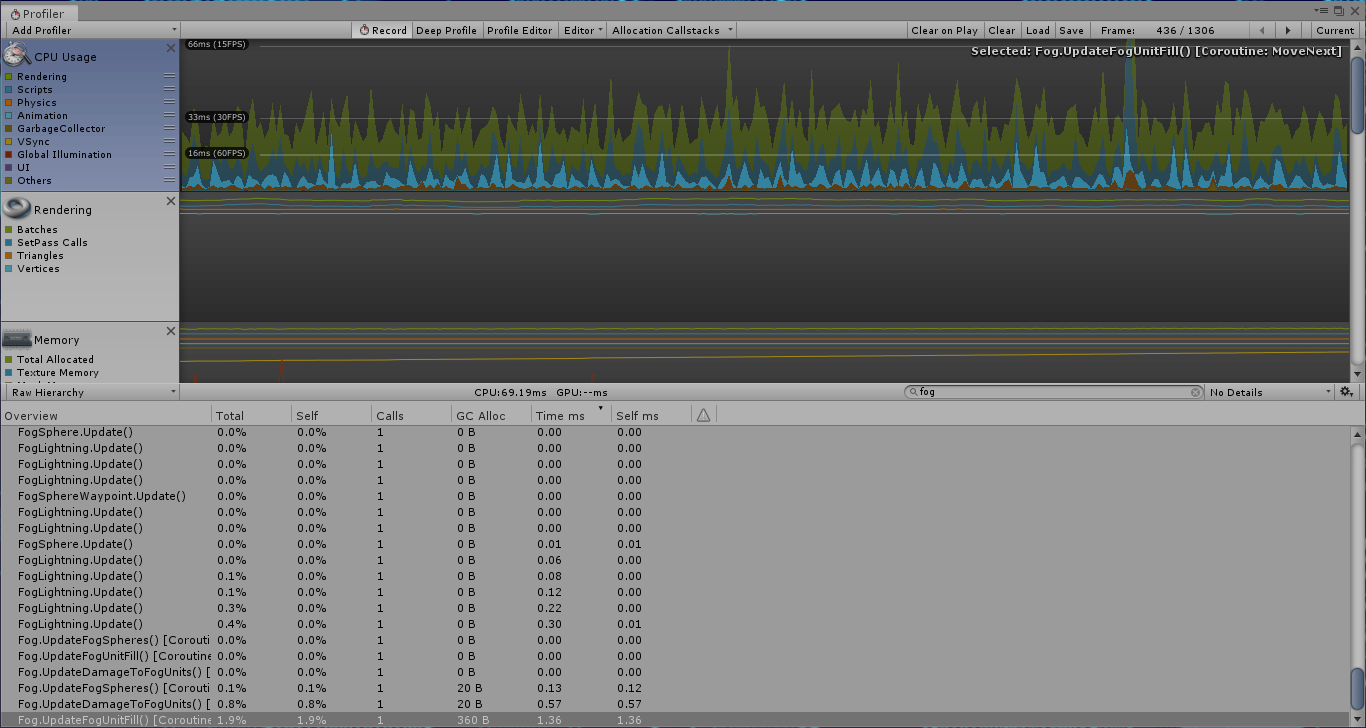
* I originally used a pooling system with 2601 FogUnits split between two lists and allocated to TileDatas at runtime from the pool. It wasn’t very fast (fig. 2).
* Running “update” methods less than once per frame helped, but still left noticeable split-second pauses, particularly after the map went from 51x51 to 71x71 (with 5041 FogUnits).
* I tried the combination of: storing FogUnits in a 2D array, with their positions matched to their TileData’s in their 2D array and on the map, for accessing when the FogUnit’s position is known; keeping the in-play list for for and foreach loop processing but scrapping the pooling list; and spreading FogUnits’ “updating” over all frames in the update interval rather than doing them all in a single frame. This resulted in a smoother framerate, with no noticeable performance drops attributable to the fog (fig. 3).

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

* FogSpheres used a two-list pooling system for their entire existence. When adding the FogUnits’ 2D array, having one here wasn’t appropriate considering FogSpheres weren’t stationary and there were only a handful of them, making List.Contains()’s linear searches[[1]](#footnote-1) for FogSpheres quick. 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

* FogSphere spawn points (i.e. FogSphereWaypoints with spawnPoint set to “true”) were stored in a list.
* Their list is only used as a base list for picking random waypoints for FogSpheres to spawn at, being copied into a new list rather than be manipulated itself. Given that it never changes, it could be made into an array that lacks lists’ extra functionality and should logically be smaller in memory, yet is as fast if not faster when accessed[[2]](#footnote-2). It would need to be implemented and tested against the current list’s performance.

## Design Patterns

### State Pattern

Figure 4: the informal “stages” of the fog.

* FogUnits included an informal implementation of the state pattern[[3]](#footnote-3) (fig. 4), with Fog featuring an Intensity public property updating the fog’s behavioural parameters. This worked simply and cleanly, and should be kept.

Figure 6: FogSphere’s states.

* More formally, FogSpheres have distinct behavioural states, indicated by the FogSphereState enum, which is checked in Fog.UpdateFogSpheres() and FogSpheres’ appropriate methods called for each state, with the methods executing their behaviour and checking conditions regarding state changes. This worked okay, but does leave room for improvement (see below).

### Singleton Pattern

* Fog, like other manager-type classes, implemented the singleton pattern[[4]](#footnote-4) to allow universal access and ensure its uniqueness.
* Its uniqueness ensures a duplicate Fog with an extra 5041 FogUnits wouldn’t be instantiated and kill *Get the Fog Out*’s (*GTFO*’s*)* performance. Therefore, I would not change this aspect.

### Factory Pattern

* Fog includes factory pattern-like[[5]](#footnote-5) methods for standardised creation of FogUnits and FogSpheres when the fog is initialised, and for instantiating and activating them when they are spawned.
* This helped keep Fog’s code clean and maintainable, as spawning only required method calls, not blocks of code.
* The methods’ inclusion in Fog rather than a dedicated FogFactory class resulted in Fog being longer and harder to search through, hindering maintainability.

### Component Pattern

* Unity’s GameObject class and its components implement the component pattern[[6]](#footnote-6) [[7]](#footnote-7). *GTFO* uses GameObjects, thereby including the component pattern by default.

## Usability Patterns

### State / Progress Patterns

* The fog informally implements the state and / or progress usability patterns[[8]](#footnote-8) thanks to its changes in colour through the game that match the intensity and aggressiveness of its behaviour, from a black-ish grey (asleep) to blue (awake) to red (the final stage of the game).

## Restructure

* To restructure the fog-related classes:
  + I would remove methods (e.g. FogSphere’s UpdateDamageToFogSphere(), Deal­DamageToFogSphere()) and enumeration values with related functionality (e.g. FogEx­pansionDirection.Orthogonal and its corresponding code in Fog.Expand­Fog()) that were obsolesced by design decisions and won’t be used again, which would result in (negligible) performance gains and shorter, more manageable, more maintainable code.
  + Fog is quite long, so I would break it up into manager classes for FogUnits, FogSpheres and FogLightning, and a factory class that could create each of those classes for their respective managers. FogFactory and each manager class would be singletons to ensure uniqueness and universal access.
  + I would dismantle FogUnit’s inheritance structure and use those separate classes as compo­nents of FogUnits. This would make Entity a Health component that FogSphere could also outsource code to, it’s health functionality being near identical to FogUnit’s. FogUnit and FogSphere’s common rendering members could also be made into a separate shared component class.
  + I would amend *GTFO*’s current implementation of the state pattern and split the methods associated with each of FogSphere’s states into separate classes that implement a base State class and operate on FogSpheres.
* This would create a more intricate network of which classes use each other, but each would be more singular in their purpose, shorter, more readable and maintainable, and more easily debugged, and any performance losses from increased public properties would be infinitesimal[[9]](#footnote-9) [[10]](#footnote-10).
* For further iterations, I would want to look into the FogEntityRenderer and Health components using the flyweight pattern of splitting classes into shared and unique functionality[[11]](#footnote-11), as a lot of their information is identical across all FogUnits and FogSpheres.

# Objectives (Tutorial / Main Game)

## Data Structures

### Lerp Targets Remaining: List

* TutorialController keeps a list of all Locatables to which upcoming tutorial stages will move the lerp target guiding players. The list affords painless appendment / removal. Any speed loss from not using an array (which is faster at searching) is minimised by the small number of Locatables and outweighed by the appendment / removal performance gains3.

### ButtonsNormallyAllowed: List

* ButtonsNormallyAllowed() compiles the list of buttons normally available on a given tile. The available buttons can vary, so the list needs to be built from scratch when needed, making lists, which are good with appendment, more suitable than fixed-size arrays that would waste space3.
* One could argue ButtonsNormallyAllowed() should be a member of TileData, such that anything that needs to check it can without going past TutorialController unless they need a list of what’s currently permissible. Doing it this way, I’d have it check if the list needs to be compiled, compiling and storing it if so, and then return the list. (Dismantling a Harvester on a depleted ResourceNode makes its tile a normal one, so a bool flag requesting recompilation would have to be set.) This would improve performance as the lists would only be compiled twice at most, rather than every time they’re accessed

### Local Data Structures of Methods: List

* TutorialController.DefencesOperable(), PositionFogExtenderLandmark() and GetBackupTar­get() also use lists, each compiling them to search or to com­pare with another list. Lists are most appropriate because they facilitate simple, flexible and efficient appendment, removal and searching3.

## Design Patterns

### State Pattern

* *GTFO* implements the state pattern4 for its tutorial and main-game objectives (fig. 11), with a switch checking an enum field and redirecting to methods for each stage, methods with a switch for substages, their functionality, and their conditions for progression.
* Switch-based stages were used because they were simple, straightforward and worked, but resulted in TutorialController and ObjectiveController being quite long and harder to maintain.
* Separating tutorial and main-game stages was an accidental by-product of how we were delegating work and who would do what, and led to unnecessary complexity and duplication.

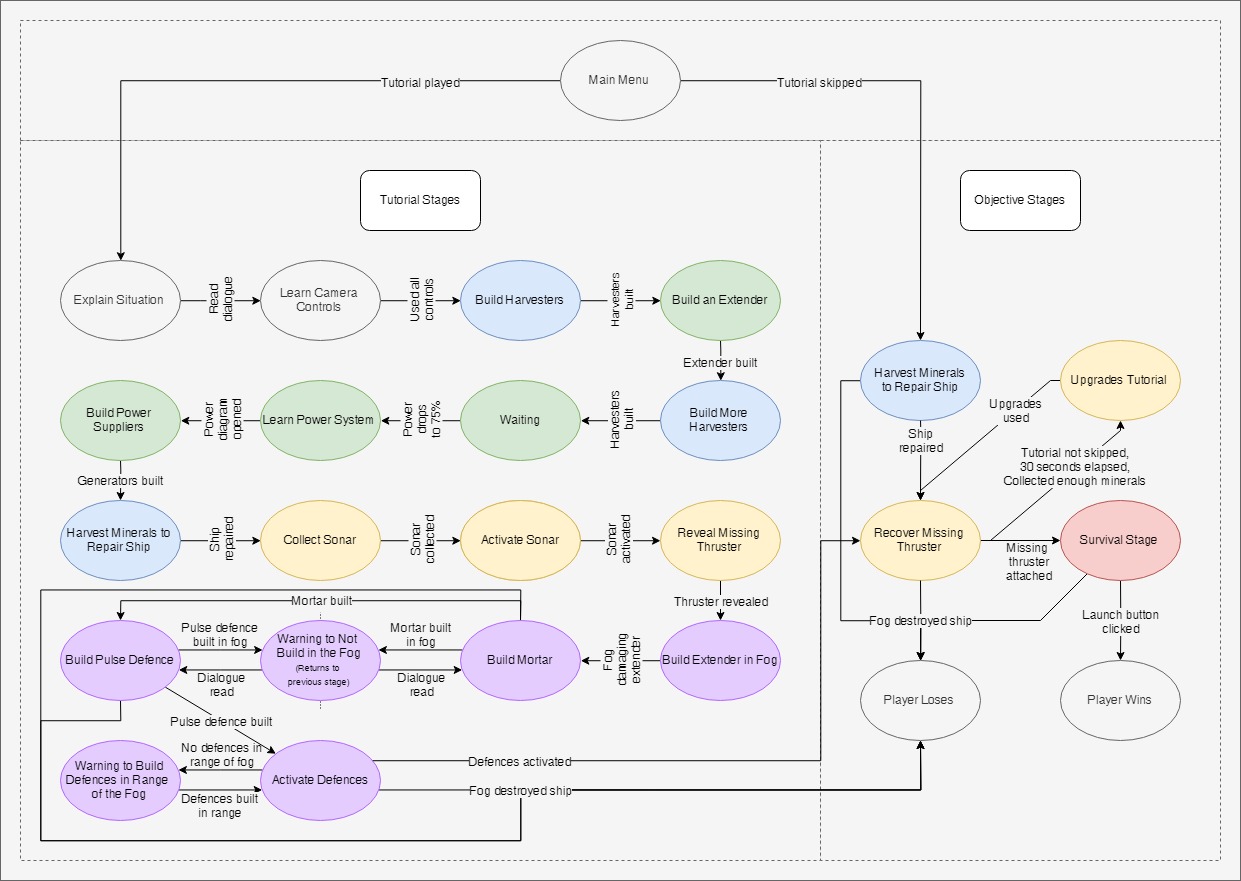


Figure 11: a state diagram for the tutorial and the main objectives of the game.

### Singleton Pattern

* The stage controllers both implement the singleton pat­tern5 to ensure uniqueness and universal access. They implement it separately rather than via DialogueBoxController to avoid any complications that might arise from their both being its children, and to avoid requiring casting every time they’re accessed.

### Observer Pattern

* btnTutorial and TutorialController interact inversely to the observer pattern[[12]](#footnote-12), btnTutorial checking every update with TutorialController if it’s button should be interactable, rather than subscribing to have TutorialController tell it when it should or shouldn’t be interactable. This was used because it was simple and it worked, but it’s less efficient than a proper observer-pattern implementation.

## Usability Patterns

### Focus Pattern

* The tutorial extensively uses the fo­cus pattern9 to teach players *GTFO*, using lerp targets in the UI and on the map and interaction-appropriate diagrams to draw players’ attention.
* The tutorial also restricts player input to what it wants players to do, forcing them to participate to progress, but ensuring they learn from experience rather than theory.

### Magnetism Pattern

* The tutorial features several manifestations of the magnetism pattern9:
  + Lerp targets getting players to build on specific tiles and UI lerp targets getting players to use particular buildings, so as to reinforce what can be built where and prevent illegal moves during the tutorial.
  + If a lerp target would be on an already occupied tile, it moves to a nearby tile that’s not occupied or earmarked, preventing the tutorial requesting illegal moves of the player.
  + The lerp target for the extender in the fog being placed where a player is already extending around the rocks between the ship and the missing thruster, or where they can first extend out to the fog if they haven’t done so yet. The defensive buildings have their lerp targets placed in the same area so that players expand outwards and start clearing fog in the same direction.

### State Pattern

* The stage controllers feature several manifestations of the state usability pattern9 to indicate changes in options or status:
  + The fog changing colour as it becomes more aggressive, conveying its behavioural state.
  + btnTutorial making buttons opaque and interactable or faded and uninteractable as the current stage requires, matching their appearance to their intractability, and conveying what they should build during the tutorial.
  + The objective window tweens onto the screen with updated text when the objective changes, and pulsating buttons appear for players to complete major objectives. (The associated methods are UIController’s members, but are called by the stage controllers.)

### Automatic Mode Cancellation Pattern

*GTFO*’s build menu (fig. 12c) includes an ease of use feature where closes automatically when the player clicks on a tile that is unusable, thereby exemplifying the automatic mode cancellation pattern9. Most often this is a tile that is not powered, but whenever a lerp target is active during the tutorial, it also includes tiles outside the bounds of the lerp target. Most of this functionality is managed by other classes, however TutorialController contributes through its method TileAllowed(), which returns whether a selected tile can be built upon according to the criteria restricting building to within the lerp target. Besides providing an easy, conventional way to close the build menu, it also implies which tiles are available to build upon or not in case the lerp target wasn’t clear enough.

### Progress Pattern

The tutorial also exhibits the progress pattern9 by having a progress bar at the bottom of the objective win­dow during the tutorial (fig. 13). At significant points in the tutorial, the bar’s progress is incremented, giving the player an indication of how far along in the tutorial they are, and how much they have left to go. It was added after feedback that often players were waiting for the tutorial to finish, but had no idea how long they had left to complete it and were left wondering if they had to continue for another dozen stages or if they were near the end and about to begin playing the main game.

## Restructure

As discussed, there are a number of changes I would make, such as making TutorialControl­ler.ButtonsNormallyAllowed() a member of TileData as it more naturally belongs there and could from there be tweaked to be more efficient, and implementing a proper observer pattern between btnTutorial and Tu­torialController to improve efficiency. There are also various enum values and field and method names whose names are out of date compared to what they are being called from the player’s perspective in the current game; those I would update to reflect their current names, the fields and enum values a few at a time to make sure I remember to reconnect anything that breaks.

More substantially, TutorialController is unnecessarily long, which makes it harder to maintain, and TutorialController and ObjectiveController are separate classes, which has led to some duplication of func­tionality and unnecessary complexity. To address these issues, I would make the more extensive structural changes:

* Of combining DialogueBoxController, TutorialController and ObjectiveController into one StageCon­troller class that handles all the stages of the game. This would remove the unnecessary duplication and complexity of having fields and functionality split amongst and duplicated across its three pre­cursors.
* Of separating out all of the stages into separate classes that all inherit from an abstract Stage class. Each Stage would have additional fields and methods beyond the common Stage members as re­quired to do their jobs.
* Of separating out other members that don’t have anything to do with managing stages into other controller-type classes dedicated to those particular things. Identifiable across TutorialController and ObjectiveController are members that relate to in-world lerp target management, UI lerp target man­agement, and monitoring of power and whether it’s been overloaded. These members could be re­distributed amongst Target, UILerpController, and PowerOverloadMonitor classes.

Splitting up existing classes, TutorialController in particular, amongst this greater number of interconnected Stage and controller-type classes would increase *GTFO*’s programmatic complexity in terms of the number of interlocking classes and pieces. However, it would make each of those pieces so much more man­ageable, as you would know: which piece to open up to affect what; where to find it to do so, rather than having to rifle through a lengthy, omniscient and omnipotent master class; and that each piece would only be given access to the pieces it required, reducing the confusion of which fields in a long list of fields were used by which Stages of the game, and what other classes were interconnected to the StageController for what reason.

# Dialogue System

## Data Structures

### ColourTag

When text appeared in the dia­logue box, we wanted it to scroll onto the screen several charac­ters at a time. We also wanted it to be able to display text in differ­ent colours. When using HTML-like tags to achieve this, we found that part of the tag would be printed, before vanishing and then changing the colour of the text. To work around this, I cre­ated the ColourTag class to en­capsulate single-character open­ing and closing tags for changing text to specified colours without having HTML printed onto the screen. It needed to have its val­ues settable in the inspector and then readable by DialogueBox, but not later rewritable.

This could have been achieved with a class or struct; I chose to make it a class out of habit. A future iteration might consider converting it from a class to a struct; it seems ColourTag would be just about appropriate[[13]](#footnote-13), and being a struct might help the game run faster or consume less memory. This would need to be tested, however. Similarly, a future iteration might consider doing away with using public properties by making the fields that never have their values changed (openingTag, closingTag and colour) public but read-only, such that they could be accessed directly rather than via their corresponding public property, thereby decreasing the num­ber of steps taken to execute the code. However, the performance gains from doing so would be negligible11 11, particularly since there would only be a handful of ColourTag objects stored in DialogueBox, and it would render those fields that need to be viewable in the inspector unviewable[[14]](#footnote-14).

The ColourName public property breaks the pattern of being a get-only property, having a setter accessed during DialogueBox.Awake(). Looking at it now, that could be made redundant and made into a get-only public property by giving ColourTag an Awake() method that does the setting of colourName rather than DialogueBox.Awake(), or making the public property check if colourName is empty, retrieve and store it if it is, and then return its value.

### ColourTags: List

DialogueBox stores its ColourTags in a List, which is visible in the inspector. A List was chosen for this as I knew it would be visible in the inspector and elements could be added or removed as necessary. Looking through the code, once the List is defined, it never changes; the code either goes through all List elements or accesses specific elements. Consequently, an argument could be made for swapping the List for an array, which can also be viewed in the inspector15 and doesn’t have the extra functionality of a List and therefore logically should be smaller in memory, but is as fast if not faster when it comes to sequential and random access3. Whether or not a future iteration converts to using an array would have to be informed by perfor­mance tests to assess performance changes.

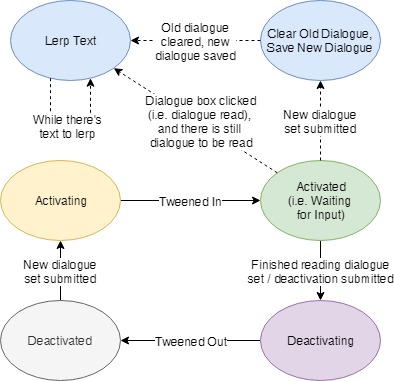
### ExpressionDialoguePair

We wanted the on-screen dialogue box to display lines of dialogue and matching facial expressions of the AI assisting the player. I wanted them to be grouped together in the inspector, and locked 1:1. Rather than have a List of strings for dialogue and a List of AI expressions and have to track if they were the same length, I put together the ExpressionDialoguePair class to store each line with its matching expression. The only change that might be worth making would be to make the class a struct, following the reasoning discussed regarding the same changes for Colour­Tag. Again, each of these changes would need to be tested for their performance benefit.

### DialogueSet

When the on-screen dialogue box presented dialogue to the player, we wanted players to be able to go through it a chunk at a time rather than facing a wall of text, but for that whole block of text to still be submitted for displaying all at once rather than small chunk by small chunk. Consequently, I put together the DialogueSet class, which stores a string identifier and a List of ExpressionDialoguePairs that represent one or two lines of dialogue. Ideally, this information would have been represented in the inspector using a diction­ary, with strings as the keys and Lists of ExpressionDialoguePairs as the values. However, Unity doesn’t dis­play Dictionaries in the inspector, which is where the string identifiers and ExpressionDialoguePairs get set. Therefore, this workaround had to be implemented to create Dictionary-like representation suitable for set­ting in the inspector. And again, there are cases to be made for making the class into a struct and the List of ExpressionDialoguePairs into an array, cases that would need to be supported by test results.

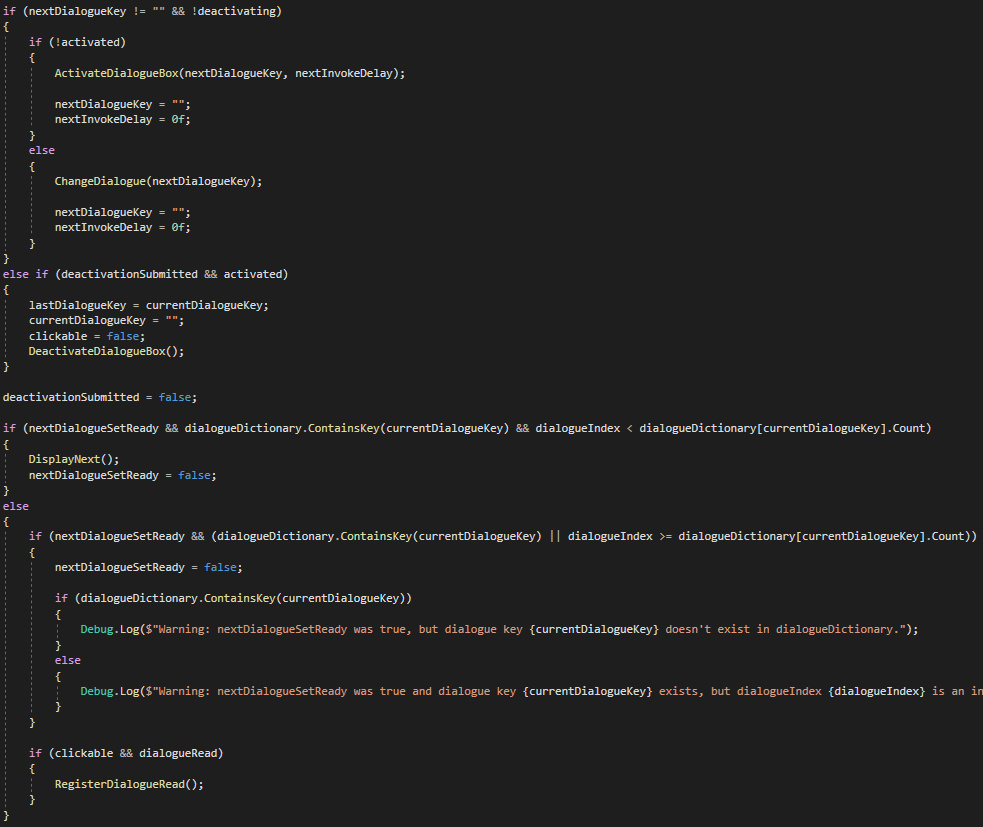
### Dialogue: List, Dictionary

Since we wanted to display many batches of dialogue to the player at different points in the game, we needed to be able to define multiple DialogueSets in the inspector, necessitating they be stored in a List exposed to the inspector. (Yes, again, List vs array testing could be justified as the List doesn’t change once defined.) As discussed just above, I didn’t make it a Dictionary of string-ExpressionDialoguePair key-value pairs since it needed to be visible in the inspector, something that Dictionaries aren’t. However, for use once the program was running and the DialogueSets were defined, I did find that using Dictionaries was easier, as I could just use the string key to directly access the associated ExpressionDialoguePairs rather than have to loop through a List and manually check each string key, which would have resulted in longer and slightly less maintainable code. As such, I kept the List for setting values in the inspector, but in the Awake() method added a loop to take each DialogueSet’s string and List of ExpressionDialoguePairs and create Dictionary entries where they were the keys and values re­spectively, and had the code after that point access dialogue content from the Dictionary rather than the List. The only change I would make here would be to use the Dictionary right from the start and do away with the DialogueSet class to remove code and thereby shorten and simplify DialogueBox.Awake() and DialogueBox.cs, but that is a non-viable change to make until Unity allows Dictionaries in the inspectors like Lists or I learn of a way to work around or bypass that limitation.

## Design Patterns

Figure 17: the identifiable states the dialogue box goes through.

### State Pattern

The on-screen dialogue box has a number of identifiable states that it transitions between, with the state of being active having several “sub-states” of things it might be doing (fig. 17). Programmatically, in terms of DialogueBox’s fields and the structure of its code, it does not look like it implements the state pattern4; rather, it uses a set of if statements to check if this condition is true or that thing is ready to be done (fig. 18).

This arguably makes it a lot more confusing and a lot less readable, even to the one who made it this way in the first place.

If how DialogueBox handles what it should be doing were to be restruc­tured, I would code it to have a more formal, switch-based implementa­tion of the state pattern similar to what Tutorial­Controller and Objective­Controller currently oper­ate on, having cases for each of an enum’s value that redirect Update() to state-specific methods that execute that state’s behav­iours and check the condi­tions to changing to a new state. This might lengthen DialogueBox’s code, but it would clean it up and simplify it, making identification of its current state and maintenance regarding those states easier; there wouldn’t be multiple checks for “Is it in this state?”, “Is it in this other state?” and “Or this one?”, but a single “Which state is it in?” “This one” “Do A, B and C” check. Such an implementation might also have some performance gains, as only what’s relevant for the current state would get checked for execution, as opposed to the current implementation where currently everything has to be checked at least partially to verify if it should be executed or not. Any such gains would need testing to verify, but would regardless be secondary to the organisation and readabil­ity gains. If a substantially greater number of stages were to be added to DialogueBox, I might instead con­sider a full class-based implementation of the state pattern as I have discussed for StageController above, but with the current limited number of states of DialogueBox, such an implementation seems like overkill for now.

Figure 18: a section of DialogueBox.Update() that handles what “state” DialogueBox is informally in, and what DialogueBox should therefore be doing.

## Usability Patterns

### State Pattern

As mentioned above, the on-screen dia­logue box has a number of visible states that it goes through (fig. 17): being not visible, tweening in, being visible with di­alogue, and tweening out again. The state of being active and ready to be interacted with is indicated by the dialogue box’s visibility, motionlessness, and yoyo-ing arrows. When it tweens in or out, or simply isn’t visible, that conveys to the player that it isn’t available to be interacted with or at all. These conveyances indicate to the player what they can and can’t do.

Figure 19: a dialogue box with a right-facing arrow, indicating that there is more dialogue to be read.

### Progress Pattern

As noted just above, the dialogue box features yoyo-ing arrows that help indicate the dialogue box’s intrac­tability. Their direction also provides some indication of the player’s progress through the dialogue set: when there are still more lines of dialogue to be displayed, they’ll point to the right and yoyo horizontally (fig. 19), but when the displayed dialogue is the last line for that dialogue set, they’ll point down and yoyo vertically (fig. 10), indicating that the next click will result in the dialogue box tweening out. This doesn’t convey the same level of detail as the tutorial’s progress bar, but provides some conveyance of progress nonetheless.

## Restructure

As noted in my discussion of data structures and patterns used in Dia­logueBox, there are a number of rel­a­tively minor adjustments that I would make in future iterations, in­cluding:

* Testing whether adding to ColourTags an Awake() method that sets its colour­Name field or configuring Col­ourName to retrieve col­our’s name if colourName is blank is more efficient, ei­ther one enabling the re­moval of the ColourName’s setter block.
* Testing if making ColourTag, ExpressionDialoguePair and DialogueSet structs rather than classes would be bene­fi­cial performance-wise, keep­ing the changes if they were, but otherwise revert­ing them.
* Testing if exchanging the Lists in those classes for ar­rays would affect per­for­mance, keeping them as ar­rays if it helped and reverting back to Lists if it didn’t.

More major changes that I would make to the structure of Dia­logueBox.Update() include:

* Splitting the contents of Update() into different methods that have dis­tinct purposes and are called from Update().
* Reorganising the transi­tioning between the cur­rent informal “states” to use a proper switch-based state pattern, with cases for values of an enum each redirecting to methods for handling execution during that state and checking the conditions for switching to a different state.

Restructuring DialogueBox.Update() in this way would improve readability and maintainability, making the code cleaner and easier to understand and manage.

# Summary

The data structures used in each class were chosen for their impact on flexibility and performance, and for their usability with Unity where necessary. The data patterns used were chosen for their suitability for the particular problems of each class. Some choices and implementations are fine as is, while others leave room for improvement. The longer classes each have ways that they could be rearranged or broken up into smaller pieces to be more readable and manageable, and therefore easier to maintain, as discussed in each section and visualised in the UML diagrams of the restructuring of each set of classes. Just as the design of *GTFO* could be iterated upon and improved, so too could its internal workings to make it more readable and more performant.

# Bibliography

Allen, J 2008, *When should I use a List vs a LinkedList*, Stack Exchange, viewed 15 November 2019, <https://stackoverflow.com/questions/169973/when-should-i-use-a-list-vs-a-linkedlist/29263914#29263914>.

Apoorv020 2010, *Performance overhead for properties in .NET*, Stack Exchange, viewed 14 November 2019, <https://stackoverflow.com/questions/3264833/performance-overhead-for-properties-in-net>.

Boppity Bop 2012, *Field vs Property. Optimisation of performance*, Stack Exchange, viewed 14 November 2019, <https://stackoverflow.com/questions/9842917/field-vs-property-optimisation-of-performance>.

Microsoft, *List<T>.Contains(T) Method*, Microsoft, viewed 14 November 2019, <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.contains?redirectedfrom=MSDN&view=netframework-4.8#System\_Collections\_Generic\_List\_1\_Contains\_\_0\_>.

Microsoft 2008, *Choosing Between Class and Struct*, Microsoft, viewed 15 November 2019, <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/choosing-between-class-and-struct>.

Nystrom, R, *Component*, GameProgrammingPatterns.com, viewed 15 November 2019, <http://gameprogrammingpatterns.com/component.html>.

SourceMaking.com, *Flyweight Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design\_patterns/flyweight>.

SourceMaking.com, *Object Pool Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design\_patterns/object\_pool>.

SourceMaking.com, *Singleton Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design\_patterns/singleton>.

SoruceMaking.com, *State Design Pattern*, SourceMaking.com, viewed 15 November 2019, <https://sourcemaking.com/design\_patterns/state>.

Tutorials Point, *Design Pattern – Factory Pattern*, Tutorials Point, viewed 15 November 2019, <https://www.tutorialspoint.com/design\_pattern/factory\_pattern.htm>.

Unity Technologies 2019, *GameObject*, Unity Technologies, viewed 14 November 2019, <https://docs.unity3d.com/ScriptReference/GameObject.html>.

Unity Technologies 2019, *Script Serialization*, Unity Technologies, viewed 16 November 2019, <https://docs.unity3d.com/Manual/script-Serialization.html>.

Woodard, C 2019, “Usability Patterns for Games”, *COS30031 Games Programming*, Learning Materials on Canvas, Swinburne University of Technology, 9 September, viewed 9 September 2019.

1. *List<T>.Contains(T) Method*, Microsoft. [↑](#footnote-ref-1)
2. *When should I use a List vs a LinkedList*, Allen 2008. [↑](#footnote-ref-2)
3. *State Design Pattern*, SourceMaking.com. [↑](#footnote-ref-3)
4. *Singleton Design Pattern*, SourceMaking.com. [↑](#footnote-ref-4)
5. *Design Pattern – Factory Pattern*, Tutorials Point. [↑](#footnote-ref-5)
6. *Component*, Robert Nystrom. [↑](#footnote-ref-6)
7. *GameObject*, Unity Technologies 2019. [↑](#footnote-ref-7)
8. *Usability Patterns in Games*, Woodard 2019. [↑](#footnote-ref-8)
9. *Performance overhead for properties in .NET*, Apoorv020 2010. [↑](#footnote-ref-9)
10. *Field vs Property. Optimisation of performance*, Boppity Bop 2012. [↑](#footnote-ref-10)
11. *Flyweight Design Pattern*, SourceMaking.com. [↑](#footnote-ref-11)
12. *Observer Pattern*, SourceMaking.com. [↑](#footnote-ref-12)
13. *Choosing Between Class and Struct*, Microsoft 2008 [↑](#footnote-ref-13)
14. *Script Serialization*, Unity Technologies 2019 [↑](#footnote-ref-14)