***Game Systems Architecture Detail  
Initial Ideation/Proposal***16.02.25 [Week 1] First Draft

**Proposal considerations:**

* **Very rough around the edges**
* Component-Based and Event-Driven architecture hybrid: this reduces the amount of repetitive programming that we will have to do, and components can interact with each other using a Global Event management system – logic is handled internally by each component
* This implementation is compatible with an object oriented paradigm: class-attribute-method approach, encapsulation of internal logic and attributes, polymorphic overwriting of generalised member methods/functions if necessary
* In terms of implementation of assets in-editor by Designers, prefabs will only need a custom programmed logic unit and “Attachable modules” (see that section) and they should “just work” (\*\*\*in theory\*\*\*)

**Carrying over to next week (post-Friday):**

* After communicating proposal, elicit any feedback, and implement before finalising it for approval

**\*\*\***

**Gamestate Manager (one instance)**

* Stores data external to the current scene, like:
  + *previous scene*
  + *save loaded or not*
  + *player profile data like input device, control mapping, profile details*

and generates events based on those details and passes them to the **Global Event Manager**

* Responsible for managing the gamestate and menus/UI and their navigation based on events (game state change events and input events) received from and sent to **Global Event Manager**, perhaps like:
  + *game paused*
  + *game over*
  + *inventory*
  + *options*
  + *tutorial overlays (?)*
* Manages the top layer rendering effects (since they will most often be dictated by the menu contexts) based on events from the **Global Event Manager**, like:
  + *sepia, greyscale, black-point manipulation (brightness/darkness/contrast/exposure), hue, saturation, depth of field, field of view etc*

(but might also be more appropriate as its own small module like **Rendering Manager**  that interfaces directly with the **Global Event Manager** depending on whether we want to manipulate the rendering effects during gameplay)

**Global Event Manager (one instance)**

* Manages a queue of events by evaluating each event in the queue and broadcasting it to the appropriate modules based the event type, potentially like:
  + *EVENT\_MENU*
  + *EVENT\_INPUT*
  + *Etc*

the type of event itself being an attribute of some generalised GlobalEvent struct or class that encapsulates the nature of the event and any other important data, like:

* + *int (enum) mEventType; int (enum) mEvent; list/map/array mData (could contain i.e. pos of mouse-click, pos of gameboject, string of error message, amount of HP to deduct, endlessly etc).*
* Queue should be private and encapsulated
* Any GameObject can propend an Event to the queue using a setter method
* Global Event Manager grabs Event at first index and handles it, broadcasting it discriminately only to those GameObjects to whom the event pertains, and once the event is handled it’s popped from the queue
* Global Event Manager should have peek functionality to ascertain whether there exists an event in the queue behind the next event to be handled: important if there are many events whose outcome is voided by a later event (i.e., no point handling anything related to in-game behaviour if the player has asked to quit the level or the win-condition was met and the next level should be loaded, or another scenario: where a boss has died and there are still events in the queue that reference it which dictate its behaviour or internal state)
* **Globa Event Manager** should itself be described diagrammatically before programming in order to model the flow of events and prevent any instances of references to GameObjects that have been nulled

**Savegame Manager (one instance)**

* Loads or saves the gamestate based on received events from the **Global Event Manger**.
* Probably needs data stored in the **Gamestate Manager** such as the file path, to load or not to load on start, etc – probably this should be sent by some kind of “initialising” Event from the **Gamestate Manager** on Awake. This needs to be explored as this might not be the sensible/appropriate approach.
* Read/write from .json file, coule possibly parse to/from binary and/or encryption which could be managed using public/private cryptography to prevent users manipulating raw .json to edit the save state, however since ecryption is probably not performant, we might need to enlist the Unity **JobScheduler** to “multi-thread” the save/load while some visual representation is exposed to the player (change cursor, load bar, save bar), which monitors the progress of the save operation and updates concurrently.
* **Savegame Manager** needs to be described diagrammatically before programming in order to model data flow.

**Music/Ambience Scheduler**

* Accepts and receives events from the Game Event manager and manages play of appropriate music/ambience sound. A simple module to understand but programming the scheduling of timed music (i.e. multiple layers/stems of a musical piece could fade in and out, scenarios like: more percussion fades into the music if an enemy appears, and percussion fades out when they died) can be fiddly; there are questions around calculating current location in an audio file and translating that to passed time and comparing it with in-game passed time, making sure they are not discordant, and essentially keeping them in time with each other.
* Should have an audio source component (for non-spatial audio like menu, music)
* Scheduler should manage AudioMixer components; external global events should make in-game Game Objects with audio sources subscribe to the correct AudioMixer (i.e. example; tell them all to play on the “delay/reverb” channel when in a huge internal space) – instinctively it makes sense to attach them to the Scheduler but would they be more appropriate attached to another module? The Scheduler will never have to interact with them if they are all preconfigured

**Audio Listener**

* Audio Listener could be attached to: scheduler, main camera, player. What are the considerations?

**Input Manager (one instance)**

* Converts raw Unity input events into custom input events.
* The separation of an **Input Manager** and player controller functionality is important for gracefully handling navigation of menus without consulting the player object (to me, consulting the player object to detect input to navigate menus would be abusive and violate the scope of the player object).
* The separation of the **Input Manager** and the **Global Event Manager** is important in order to allow for raw input from differing devices to be handled and converted to a uniform event that the game always handles in the same way. This will mean (perhaps will not be present in the vertical slice, but important to consider in terms of market research) multiple popular methods of input for players can be handled i.e. controllers (which?), keyboard/mouse, etc. It will also mean that multiple platforms can be considered. Again, probably out of our scope to actually implement, but our architecture should be scalable and should consider that this is what some hypothetical full game might require.
* Probably also needs data stored in the **Gamestate Manager** (i.e. player preferences, control mapping), so handle this in the same way as the **Savegame Manager** will (yet to be decided, perhaps an initialising event?).

**Main Camera (one instance)**

* Reads player position each frame and convert it into a target. Does this violate encapsulation?
* Receives input events (i.e. so the player can control the camera if that is what the game design dictates, like zoom in/out, rotate, or push it around to “look around”) and other game-generated events (to manage the cinematic side, i.e. succumb to other cross-shoulder cameras for NPC conversations, tightening or squeezing or relaxing of viewpoints based on size of current area or in other words get closer if we are in a tiny cave and pull out if we are in a large wasteland) from the **Global Event Manager**

**GameObjects** (many instances)

* GameObjects of each type have their own internal attributes (position, etc).
* (Dynamic) GameObjects should all have some kind of **state machine/local event manager** that accepts global events from the **Global Event Manager** and converts them into local events which propagate throughout some kind of modular framework to update the GameObject in various ways. This **state machine/local event manager** inherits from a generic class and will polymorphically overwrite the generic functions with custom logic depending on the needs of the type of GameObject and should be the only customisable module in terms of the wider architecture. GameObject **state machine/local event managers** should be attached to prefabs of the types of GameObjects in order to reduce the propagation of bugs, bad logic or other mistakes throughout a very large collection of GameObjects.
* All Saveable objects should have a **serializable** copy-class instance which maintains attributes of the GameObject using serializable data types, which will be the structure that is passed to the **Savegame Manager** for serialisation into .json format when called. The class will also be necessary to deserialize save-game data and populate the GameObject’s attributes on load.

**Player**

* Special instance of a GameObject; the only one that interfaces with modules other than the **Global Event Manager** (the camera) – questions over whether this violates encapsulation, but because this needs to be crazy fast / accurate, probably it’s the only option, since passing player location events to the **Global Event Manager** will flood the queue, give less performant results, and probably is an abuse of the purpose of the **Global Event Manager** system

**Secondary Player/Companion**

* Will the location of the companion affect the camera? Do we need to triangulate the distance between them, target the result, zoom in/out the camera so that they are both always on screen? If so, interfacing with the **Main Camera** like the **Player** does is a possibility. See Player and Main Camera for thoughts about encapsulation. Or, possibly more appropriate: some intermediary **Camera Target** module can take the **Player**, **Companion** positions and triangulate them and also take into account other cinematic events, and the Main Camera instead just reads from the **Camera Target** module. In other words, computation of camera target would be separated from overall automatic and manual control of the **Main Camera**?
* Likely the companion will be fast moving. We might need some clamping/dampening functions so that the camera speed does not increase so as to become distracting; i.e. if companion is “thrown and boomerangs back”

**Attachable Modules** (many instances)

* Attachable modules can be attached to the prefabs of the GameObjects in order to handle modular behaviour. All of them are optional and work independently of other modules and communicate (if necessary) using the GameObject **state machine/local event manager.** Modules proposed so far are as follows:
  + Animation Controller
    - Manages the play, speed, transition of the GameObject’s animations
  + Savegame Controller
    - Manages a unique ID in-editor which is assigned on instantiation, serialised, and stored in a Scriptable Object component of the Savegame Manager to be used at runtime to reference GameObjects – dataflow should be modelled diagrammatically to ensure that in-editor IDs are unique, correct, and cannot diverge from the data stored in the Savegame Manager, and to throw / manage errors when necessary in-editor and at run-time
  + Trigger Controller
    - A module that can imbue any GameObject with trigger functionality (i.e. volume triggers). Should be scripted to accept conditions in the inspector window, to accept trigger nature (one-shot, one-shot with reset condition/multi-shot, continuously persistent or discretely persistent etc) as well as to accept outcomes – this is the most Designer-facing module and its implementation will have a big impact on the Level Designers, we must elicit feedback from designers about what they might need from triggers, there’s potentially a blind-spot here because the programmers haven’t done any level design modules
    - Unity has some trigger management but our functionality should be wrapped in its own module, since triggers should accept global Events in order that triggers can be voided, or reset, or timed, for instance: deactivate trigger for a trap if some switch is pulled, or reset a trigger if some fail condition of a puzzle is met
  + Audio Controller
    - Has audio source
    - Spatial or game-wide pan (i.e. enemy die sound or menu click sound)
    - On instantiation is passed which channel to play? Channel management needs consideration in terms of **Global Event Manager** and **Audio Scheduler**
  + Physics Controller
    - Lightweight physics simulation, either manual or using Unity’s rigid-body script methods. Requires experimentation to find most appropriate method of handling physics, or whether hybrid approach is necessary.

**Emitters/Factories/Object Pool**

* At times we might need to generate prefabs that aren’t instantiated in the editor. (Think bullets, arrows, ranged weapons like grenades, environmental debris from a destructible object). One way of handling this is to create an **Emitter/Factory** module that can spit out an instance of a passed GameObject prefab class at some xyz position vector, accel, velocity and potentially more. However, this is completely up in the air.

Arguments for:

* + Centralised information for save/load about runtime generated objects
  + Emitter module need only be programmed once
* A way to optimise this approach is to design an object pool from which GameObjects can be borrowed, reinitialised, used and then returned to the object pool deactivated – this way, the garbage collector has a seriously reduced amount of work to do when clearing up after destroyed objects from the memory (case study where FPS was increased from 66 to nearly 100 just by implementing an object pool)
* There is a serious technical consideration: how should the Savegame Manager handle the object pool?

**Utilities**

* We’ll probably need several little class utilities like a little timer, etc that we can instantiate in the GameObjects as and when we need them (i.e. player was hit, cannot be hit again for another 3 seconds, flashes for 3 seconds, whatever).
* In-editor utilities like error handling and resource mis-management