ARGO with Godot Proposal

Motivation

In Favour of Godot

The Godot game engine provides an easy to understand interface through which to create games. Godot allows easy portability to multiple different devices without changing any code or project settings. Games can be very quickly prototyped and developed using the engine’s node system, of which each node provides an array of useful functionality.

Godot projects are very small in size as they save only the information they must, and use very simple file formats, making collaboration through git much easier, as merge errors pose far less of a risk.

Godot has been gaining traction massively in the past few years, and with the imminent release of 4.0, it should be on par with the main commercial game engines, such as Unity or Unreal, in regards to the functionality it provides.

With Godot receiving large donations from Meta, Microsoft, and other big names, the engine is likely to be a hot new skill in the coming years, providing many new job prospects.

Variance with Published Rubric

For ARGO, the rubric for each module has been flexible enough to support nearly any game engine, except Games Engineering, as the module is focused around the game’s architecture.

While some of the rubric conflicts with the engine architecture, alternative design patterns with a similar level of technical achievement could be found.

The patterns which cannot be implemented or are already implement patterns are:

- **ECS** - The engine would have to be partially rewritten to support ECS, and doing so would remove the benefits of the engine and its node-based system.

- **Game Update Loop** - while the game will be developed making use of this pattern, the pattern itself is already built into the engine. Godot’s _process(delta) method is equivalent to an update() method, and is used the same way.

- **Observer Pattern** - Godot provides an incredibly flexible and versatile observer pattern built into the engine in the form of "signals". Signals will inevitably be used all through the project, though the implementation of them is already handled.

Below, in the next section, is a list of the proposed patterns we plan to use in the project, including details of where we plan to use them, the benefits of them, and their impact on further development.

Proposed Design Patterns

Overview

As the game is focused around many interlocking systems made up of objects, the project will heavily use the Flyweight and Type Object patterns for storing data. The Factory pattern will be used to create this data, and the Visitor pattern will be used to save the current state of the data.

The second large area of the game is *interaction*. Actors in the game (players and AI) will have numerous actions to choose from at any one time. These actions will be structured as Commands, following the structure laid out in the Command pattern.

To navigate the decision space of the game, the AI will use a decision/behaviour tree, implemented as a multilayered Finite State Machine to make its decisions and play strategically.

The game will have many sounds and visual effects throughout the game, to avoid the performance hits of constantly instantiating new objects for these, or the memory trade-off of allowing each object multiple audio players and particle systems, an Audio Manager and Particle Manager will be implemented as Object Pools to manage the shared use of these resources.

Finally, to thoroughly test the game, the Bridge pattern will be used throughout to separate the dependencies of the game and allow classes to be switched out for ‘mock’ versions during testing.

Factory

Jack has planned to implement this pattern.

An existing requirement on the rubric.

For constructing factories and tile data (when harvested or polluted). As types within the game will be implemented as Type Objects, they won't have their own constructor, and factories will be used to instantiate them with the correct values.

Quality Measure

The creation of Game Objects should be managed and optimised for the title.

Command

The entire team will implement parts of this pattern.

An existing requirement on the rubric.

To abstract player and AI control over the world. All actions that can be taken on the world (purchasing land, building factories, researching new tech, etc.) will be implemented as commands to allow both the players and AI to perform them without duplicating code.

Quality Measure

Input mechanisms should feel natural and be independent of the Human-Computer Input mechanism.

Finite State Machine

Michael may implement this pattern.

The first of two optional patterns listed on the rubric.

Used for AI decision/behaviour trees to handle strategy.

Quality Measure

AI decision logic is carefully split into states, each of which encapsulates all data and functionality necessary only for that state.

Flyweight

Emma may implement this pattern.

The second of two optional patterns listed on the rubric.

Used for tile and factory information for sharing data.
This is used alongside the Type Object pattern, more about this below.

Quality Measure

The creation of runtime objects should be managed and optimised for the title.

Type Object

Emma and Michael may implement this pattern.

This pattern will be used alongside flyweight to allow each building and tile type to have their data read in from a data file at runtime.

Flyweight will be responsible for the sharing of data, while this pattern will be used to allow for the easy addition of new tile and building types without adding new hardcoded classes.

Quality Measure

Objects that contain many types should utilise this pattern to reduce the amount of redundant code and simplify the process of creating more.

Object Pool

Szymon may implement this pattern.

The optimisation technique - for memory and performance.

Used for sound FX and potentially VFX.

A singleton *audio manager* class will be used to play audio without constantly instantiating new instances of audio players by encapsulating an object pool of audio players.

Similar may be done for particle effects and such.

Quality Measure

Objects used often by multiple different classes are pooled and called for when needed opposed to instantiated and removed during runtime.

Visitor

Szymon may implement this pattern.

Used for saving game data to a file. Most classes need the ability to be saved to a file so games can be paused and continued at a later time.

Quality Measure

Classes containing data related to the current game state have the matching visitor methods to save them to file.

Bridge

Jack may implement this pattern.

Used to allow the codebase to be easily testable, as complex systems that have multiple dependencies can use a ‘mock’ version of the dependency. This will allow tests to run faster and be more granular.

Quality Measure

The core systems are broken into regions, with dependencies between them passing through a bridge which can be swapped out for a mock.

Optimisation techniques

The Object Pool pattern will be used to better optimise the speed of the program, while the Flyweight pattern will be used to lower the memory costs.

The Type Object pattern will allow us to create new content for the game with great efficiency and speed.

Project Goals

Minimum Goals	**Peer Demonstration Goals**	**Showcase Goals**
The above patterns are implemented into the game. The pattern code is commented and structured well. The pattern implementation has been researched and implemented effectively.	The patterns are used appropriately and aid the game immersion, directly or indirectly. The pattern code is commented and structured well. The core systems have been adequately tested and no bugs have been identified. The game systems are enjoyable and have some novelty or replayability.	The design patterns were well used and aided the overall immersion. The pattern code is commented and structured well. The core systems and vital functionality of the game have been tested sufficiently and no bugs have been identified. The overall game is immersive and replayable.