Project Valence

Code Quality Review

April 5th, 2024

Nicolas Ansell, Julian Gonzales, Michael Osachoff, Cameron Wilson

Table of Contents

Table of Figures

Introduction

Code quality is essential for software development projects to ensure the long-term success of a project. Writing and maintaining a solid foundation allows projects to further extend their feature set, reach more audiences, and resolve issues in haste. Principles such as Object-Oriented Design (OOD), Don't Repeat Yourself (DRY), and following industry standard design patterns are a few ways a project can ensure success. Project Valence will be compared to OOD, DRY, and industry standard design patterns to evaluate the quality of code then give recommendations for improvement before further additions are made to the game.

Godot

Godot is an open-source, lightweight, cross-platform, all-in-one game engine and the foundation of Project Valence (Godot Engine, 2024). Godot has two major design patterns that are core to the engine, object-oriented design, and composition (*Godot's design philosophy*). As a team, we elected to stick to composition as our main source of maintaining the Don't Repeat Yourself (DRY) methodology. For example, in our code, machines are composed of a sprite, machine logic node, interaction area, a machine timer, and other supporting animation nodes.

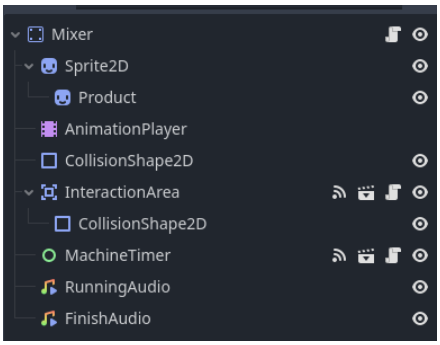

Figure 1 Mixer Nodes

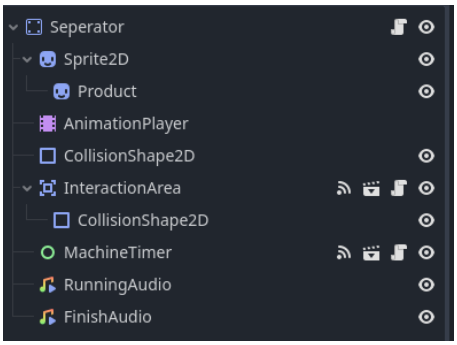

Figure 2 Separator Nodes

Design Patterns in Project Valence

Items

Items in Project Valence are objects that can be placed within a user's inventory, supporting crafting using machines and cleaning up hazards. Items are a custom node that inherits from Area2D and stores all the information required by the items. This includes the item's name, description, texture, id, the machine required to craft, items required for crafting, regeneration time, crafting time, the potential hazards, and a flag if it can be deposited into a machine. When building a level from a contract, the items create trees with its leaves representing what items must spawn into the level. Items are responsible for all their own information, leaning into both the DRY principle and the single responsibility principle, as items are responsible for their information and its management only.

```gdscript
extends Area2D

class_name Item

@export var item_name : String = ""
@export var description : String = ""
@export var texture: Texture = preload(Globals.
@export var _id : String = ""
@export var machine_required : String
@export var items_required : Array[String]
@export var regeneration_time : float = 1.0
@export var time_to_craft : int = 0
@export var hazard_depo_required : String
@export var can_deposit : bool = true

```

Figure 3 Item Export Variables

The Fridge Factory

Originally items were retrieved in the levels from literal fridges. The naming of these nodes internally became Fridge because of their original visualization. They were refactored during development to match the factory pattern and created the Fridge Factory object. Following a well-documented and known design pattern allowed us to have faster development, more streamlined access to level creation, and a single place to change any creation and configuration of the Fridge class.

```gdscript
extends Node

## Creates instances of fridges for the player to interact with
##
## Used to generate fridges given a specific item
##
class_name FridgeFactory

var _base_fridge_path = preload("res://Machines/GenericPickupFridge.tscn")

## Pass in an item object and optionally a time to create a new pickup fridge.
## default regeneration time is 1.0 seconds
func create_fridge(item : Item, texture : Texture2D = null ,regeneration_time_in_seconds : float = 1.0):
    var new_fridge : Fridge = _base_fridge_path.instantiate()
    new_fridge.set_item(item)
    if(texture != null):
        new_fridge.set_texture(texture)
    return new_fridge
```

Figure 4 Fridge Factory Class

The Managers

In Project Valence we make extensive use of the singleton pattern marked with the Manager title. The best example of a manager is the Interaction Manager scene that manages interactions from its assigned player to nodes with the interaction area node. Managing interactions in this way allows the game to prevent overlapping areas from all being triggered at once. The manager has a user interface built into it that retrieves information about the node a user is attempting to interact with it. Making the interaction manager responsible for the UI allows a clean synchronization to the object a user will interact with and the UI representing it.

The Observer Pattern

Godot makes intense use of the Observer Pattern with its built-in signals, allowing easy communication from one node to many in the scene. One of the simple implementations of the observer pattern is the Contact Manager class, a singleton-observer class that communicates by having other nodes subscribe to its signals. An example of this is shown from the editor in the figures five and six below, with red being a signal owned by the contract manager and green being subscribers.

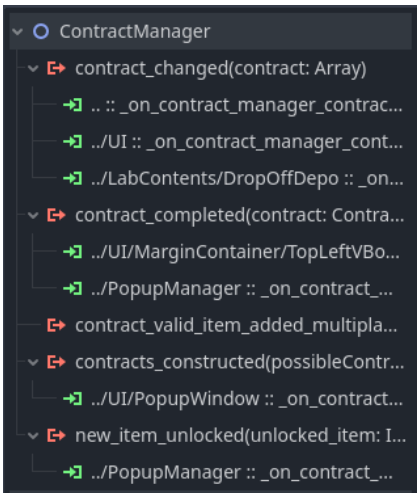

Figure 6 Contract Manager Signals

```
func _on_contract_manager_contract_changed(contract: Contract):
    var contract_items : Array[Item] = contract.get_products()
    var contract_counts : Array[int] = contract.get_contract_counts()
    for i in range(contract_items.size()):
        var new_item_display : ContractProgressDisplay = contract_item_ui_template.instantiate() as ContractProgressDisplay
        new_item_display.instantiate_contract(contract_items[i].item_name, contract_items[i].texture, contract_counts[i])
        new_item_display.size_flags_horizontal = Control.SIZE_SHRINK_BEGIN #sets horizontal alignment to shrink to beginning
        contract_container.add_child(new_item_display)
        new_item_display.add_to_group("progress_displays")
        var new_recipe_instruction_display : RecipeInstructionDisplay = recipe_instruction_display_template.instantiate() as
        new_recipe_instruction_display.set_product(contract_items[i])
        new_recipe_instruction_display.add_to_group(contract_items[i].item_name)
        recipe_box.add_child(new_recipe_instruction_display)
```

Figure 5 Example Subscription

Mediator Pattern – Multiplayer

Multiplayer in Project Valence is an example use where we employed the mediator pattern. The Multiplayer class acts as a mediator between the players and the server game state. The class is heavily coupled to the machines and general game level. This has led the Multiplayer and Game Level classes to become so called ‘God classes’ that could benefit from further refactoring.

Good Practices Followed

To keep the code base manageable, the team committed to a few best practices that were enforced with a code review process. Two major sections include our naming convention and the use of comments.

Code Commenting

When writing functions that may not make complete sense from the name, a header comment was used to explain what the function was for. A good example of this is the recipe match function in the code, where it is not clear the criteria for matching a recipe. The header comment allows a developer using this code to know the expected results without reading the function body.

```
## a recipe matches if the machine inventory and the recipe array contain the same items no matter the order.
func recipe_match(ingreds : Array) -> bool:
    var cache : Array = ingreds.duplicate()
    for item : Item in _machine_inventory.items:
        if !cache.has(item.item_name):
            return false
        else:
            cache.erase(item.item_name)
    return true
```

Figure 7 Function Comment

Code comments were not always implemented where they should have been, for example in the music player class there is no documentation to explain why there is an infinite recursive function in the class, nor does it include the optional class name.

```
1   extends AudioStreamPlayer
2
3
4   # Called when the node enters the scene tree for the first time.
5   func _ready():
6       _play_sound()
7
8   func _play_sound():
9       self.play()
10      await self.finished
11      _play_sound()
```

Figure 8 Music Player Class

Naming Conventions

In Project Valence we agreed on a simple naming convention that was stuck to for most of the code and file names. Directories, class names, and file names are pascal case with functions, variable names, and anything not listed was to be done in snake case. Since GDScript does not support encapsulation, anything private that is not meant to be used outside of a class/script should start with an underscore. We also made extensive use of the export variable type that allows you to set node properties from the editor with validation instead of setting variables in code.

Don't Repeat Yourself (DRY)

During the development of Project Valence there were multiple refactors to implement and maintain DRY principles. One of the main refactors we did was abstracting the machine logic to its own class. The class was used in the composition of the machines in the game allowing all five machines to function. Two of our other machines used a similar concept with a hazard logic class, containing all the logic for storing and allowing players to interact with the mop and fire extinguisher items.

```
func _on_start():
    if _machine_inventory.items.size() < 1:
        _player.play_error_sound()
        return
    for recipe in _recipes:
        var ingreds : Array = _recipes[recipe]
        if (ingreds.size() == _machine_inventory.items.size() && recipe_match(ingreds)):
            _craft_and_animate(recipe)
            crafted_item_multiplayer.emit(_parent_name, recipe)
            return
    _deal_with_crafting("")
    set_output(output_item)
    crafted_garbage_multiplayer.emit(_parent_name)
```

Figure 9 Example of general machine logic.

Areas For Improvement

A major area for improvement we could have in our code quality is to implement unit testing and seeded random values. Since most of our testing was done at the user level, we did not have guarantees that new modules did not have regressions from previous versions. As developers, we play tested each pull request to validate functionality however, this is still a large oversight for the project and should be corrected. Our level generation system also makes use of heavy random number generation, the number generation gives a different feeling to each level, but it also makes it hard to replicate bugs in generation. To change this, it would instead be wise to randomly generate a seed, and then use said seed in a predictable manner for level generation.

Our implementation of object-oriented fundamentals could also use some work. In the code base there are some instances of setters and getters where other times we do not use them at all. Using setters and getters in a more standard way when encapsulating code would strongly benefit our code base. Any instance of directly using a variable with an underscore prefix should be removed and, functions with this prefix should only be used within the class or for connecting node signals to internal code.

Conclusion

Project Valence overall has decent code quality. In most cases, code standards uphold DRY principles as well as the internal guidelines for variable naming. Key areas of improvement for code quality include the addition of unit testing. Unit testing is one method the industry uses to ensure code quality that we overlooked. Another key area of improvement would be increased commenting of functions within modules. Currently, we have sparse commenting and, despite our descriptive function names, a new developer to the project would have a hard time maintaining the code. We may also find that coming back to this code months from now we would no longer understand what certain functions did. In general improvements of OOD fundamentals and testing are areas the must be improved prior to the implementation of new features.

References

Godot Engine. (2024, April 2). *Free and open source 2D and 3D game engine*. https://godotengine.org/

Godot's design philosophy. Godot Engine documentation. (n.d.). https://docs.godotengine.org/en/stable/getting_started/introduction/godot_design_philosophy.html