TextGameEngine Design Document

Phase 1

Specification

Description

A graphical game studio for quickly creating a text-based adventure game with decisions at each scene. This is done through a GUI consisting of boxes and arrows between them to represent scenes and player decisions.

When a user starts the program, they are shown an empty dark canvas with a button that has one option: create slide. When the ‘create slide’ button is clicked, the user drops a box on the canvas somewhere. They can drop multiple boxes and connect them with decisions.

A user can click a slide box to bring up a text box in which the dialog/prompt for the slide is entered. The user can click a decision arrow to bring up a text box in which the decision text is entered.

There are OUTPUT anchor points on the right side of each scene box, 1 more than the current number of decisions leading FROM that scene.

There are INPUT anchor points for each box, 1 more than the current number of decisions leading INTO that scene. In addition, we added a new specification for phase 1 where a decision can select what scene it leads into on its own.

The user can create decision arrows by clicking 'add decision' on the slide box. When the user drops the arrow, that arrow is coloured red when unconnected to a scene box. The arrow is colored yellow if there is no decision text yet. The arrow is coloured white when connected and has decision text.

The user is given an option to “export” to an RDF string via the 'Save' button. They can test the game through the GameRenderer using the Test button (which calls a TestHandler class).

The other component of the specification is the GameRenderer. A GameRenderer is a mostly separate part of the project that has some limited interaction with the

game studio. A GameRenderer creates and uses a Player controller to draw the Slides and Decisions of a Game entity. GameRenderer is part of the PlayDisplayer interface, which is called by TestHandler. Since it is an interface this does not violate clean architecture.

While much of our overall specification remains the same we have added a number of new handlers and buttons. We have added a CreateNewDecisionHandler and CreateNewSlideHandler to be called from the GUI and separated them from SlideHandler and DecisionHandler.

Diagram

https://app.diagrams.net/#G1o9mZ4J0Z7chtzJ1j1qkbVmvthBuS5CEK

Major Design Decisions

We have decided how Slides and Decisions should interact with each other. This was a major point of contention in phase 0 on whether decisions should know the slide they come from and the slide they are going to. However, as phase 1 was worked on we refactored and solidified the plan that Slides will keep track of the decisions that will show up and decisions will keep track of the slides they go to. The clearest reason for this is because users may wish to repeat the decisions at multiple points in their game (for example, multiple rooms in a house may have a decision to enter the bedroom).

Although our GUI in studio works similarly to how it did in phase 0 from a user's perspective we significantly refactored how we dealt with our GUI in order to maintain adherence to Clean Architecture. Namely, additional handlers and use cases (such as CreateNewSlideHandler, CreateNewDecisionHandler, EditorGame, BaseHandler) cleaned up the relationship between our GUI and entities which had been questionable in our earlier work. While this design decision cost us a lot of time and wasn't visible from the user perspective we decided to go ahead because it gave us more confidence in our ability to extend our work going forward. Our previous design made it difficult for us to add new features cleanly, while our decision to refactor will pay dividends going forward.

Clean Architecture

We are confident we adhere to Clean Architecture even after considering the complexities of our project. We worked hard this phase to create clear lines of dependency between layers that adhere to Clean Architecture standards, as well as reduce coupling whenever possible. Additionally, our use of Dependency Injection with PlayDisplayer allows us to be confident that we are avoiding Clean Architecture violations with how we render information from our underlying entities.

SOLID Principles

Single Responsibility Principle

Our increase in handler classes during Phase 1 allowed us to adhere to the SRP closely. In addition we increased the types of GUI classes as well.

Open/Closed Principle

Our use of the PlayDisplayer interface allows our program to be open for extension while remaining closed for modification. In addition, our implementation of an RDFLoad parent class would allow future developers to use our saves in currently unplanned or unexpected ways without needing to do extensive modifications.

Liskov Substitution Principle

We have multiple instances of parent classes, subclasses, and interfaces in our code. Namely the RDFLoad class and its children, PlayDisplayer and GameRenderer that implements it, and RenderableObject with its two children. All follow the Liskov Substitution Principle.

Interface Segregation Principle

All of our interfaces have a slim amount of methods (two) that are directly related to a sole purpose of the interface. Users of our code will therefore not have to use unwanted methods when they implement our interfaces.

Dependency Injection Principle

The PlayDisplayer interface is used as a dependency injection in order to bypass potential violations of the clean architecture. In general, our code is structured closely to clean so as to follow the Dependency Injection Principle.

Packaging Strategy

Our packaging strategy is packaging by layer of architecture. We have a handlers folder, entities folder, buttons folder, and client (GUI) folder. However we also have incorporated some aspects of packaging by shared features. For example we have an rdf folder for our saving and loading of games since they have a large number of shared features and functionality. We also have an interfaces package to denote high-level objects and interfaces that have similarities outside of our other GUI classes.

While our approach mixes strategies it is effective and fulfills our design well. It prevents any packages from being too clustered and also uses clear and self-explanatory naming conventions. It may not be clear to users where rdf files would be packaged if they did not have their own folder, but it would seem natural to users looking to extend the GUI to look right away in the clients folder, or users looking to add a new type of slide to go straight to the entities folder.

Design Patterns

We implemented an Observer design pattern with our Slide/Decision, GUISlide/GUIDecision and EditorGame classes. We use the ObservableMap class from javafx in EditorGame in order to link particular Slides and GUISlides as well as particular Decisions and GUIDecisions together. This allows us to have edits a user makes to GUISlides or GUIDecisions be observed by a respective Slide or Decision stored in a game entity.

This design pattern was extremely useful for our project because it gave us a simple way to solve an issue we had faced since the beginning of our work. By having the two values in the ObservableMap observe each other the process of rendering GUI and saving changes to the GUI became simple, enabling an essential feature in our program.

Use of GitHub Features

Our group made very good use of the GitHub features for the duration of our project. All of us did plenty of work on specific branches and created pull requests mostly when large changes were done. The pull requests require at least one review so we do not accidentally make mistakes in our main branch, and our commits let the reviewer know exactly what was changed so we could easily confirm the merge.

We had also utilized the issues feature. Though they are mostly announced and communicated with the team via discord, the issue page on GitHub is a reminder of what to fix.

Code Style and Documentation

All of our classes and methods have headers that accurately describe what each class/method does. Not only that, but inside methods with non-obvious code, blocks of code are also commented to highlight their specific purpose as well. Any warnings we see are generally fixed automatically with IntelliJ's corrector and any errors are immediately fixed before ever moving on from that block of code.

Since many of us are working on different areas, we don't always know how some parts of the program are constructed. Because of our documentation with readable code, those who were working in different areas are able to understand the code from the other areas.

Testing

All entities have unit tests that investigate possible issues and unexpected outcomes from edge cases. Additionally, we have tested all of the Handler, Use Case, and GUI components on both sides of our project.

Many of the GUI classes posed a challenge to test extensively. Since much of the GUI's proper function had to do with its display output it was difficult to test it from anything other than a human tester. Unit tests were nearly impossible to write, since the only way to confirm that the width, height, and theme had been set correctly was with human observation (this is as much of a design decision of the creators of javafx than it is of our team).

However, our design decisions did make it more difficult to test Handler classes. Due to our adherence to Clean Architecture we had to run the entire program in order to test particular Handlers.

Refactoring

We have certainly done a fair amount of refactoring. For example, every button in the editor had their construction and event handler in the same SidebarButtons class. We changed this to give each button their own class and made them a subclass of a main constructor to remove the very repetitive code. This was done in pull request #25 commit 4eeacca.

Pull request #29 also had a large amount of refactoring. The refactoring there was done for the purposes of a more reactive front-end. We were storing the data for the application in a data structure, EditorGame, and the data that the user saw in the GUI, and were storing those separately. The problem with that was that there was the potential of changing the underlying data without changing the visuals, or vice versa, which could cause synchronization issues. The refactor allowed the GUI to “listen” to the data structure, and store none of the data itself.

Code Organization

Our code is organized by level of Clean Architecture with certain exceptions made in instances where the package structure would be more logical and intuitive. Our biggest packages, entities, client (GUI), and handlers are all separated based on their location in the Clean Architecture hierarchy.

However, there are certain exceptions made. Buttons is an example of a notable package excluded from the client package. The reason for this decision is buttons' notable role in the game studio and their independence compared to the other client classes.

Additionally, the rdf package separates classes related to RDF (which is how we save and load game files) into their own distinct category. While most of the classes in that package would likely fall into the handlers package they serve a distinct role that should be immediately accessible to a programmer, and therefore be clearly separated from the other handler classes. While this may violate the general tradition of separating by layer, it makes it easier for future users to find and is a logical structure of organization for such a unique part of our software.

Functionality

Our program is indeed fully functional! The user can do anything specified in the specifications. Adding a slide, adding decisions to that slide, editing the text of both, and playtesting the game from a playable state. Not only that, but we are able to save and load states of an editor. We utilized RDF to convert our program to a

saveable state which we save as a .ttl file in any location specified by the user. Loading this file back into the editor allows the user to continue a game they were working on and playtest it!

A full tutorial will be shown to the TA to showcase the functionality of the program and how it works.

Progress Report

Open Questions

Our biggest success with our design so far has been our ability to maintain separation between different aspects of our project. Since our GUI is not reliant on our GameRenderer (and vice versa) we have been able to make edits to and improve one without needing to modify the other. In addition we are reaching a level of design where we can extend our code easily without the need for modification of other features or functionality.

Current and Future Work

Kevin Portinga

Helping Kaspar refactor the code to better adhere to clean architecture, looking over and commenting on/approving pull requests, writing the design document, writing unit tests, and creating the diagrams. Will work on feature extension and further documentation.

Kevin Li

Worked with Jason You on improving the visuals of GameRenderer and refactoring code for GameRenderer. Will implement additional features for the game and ensure code follows the clean architecture principles.

Jason You

Worked on GameRenderer by improving visuals and customizability. Will continue implementing features into the game player portion as new features are added in the game engine

Jason Sastra

Worked on Game Editor GUI, adding connectable slides and decisions that updates along with the backend. Along with helping on some bugs regarding visual loading. Will continue to make various quality of life changes for GUI

Greg Sherman

Worked with Kaspar on saving the editor version of the game into a ttl file that can be loaded into the gameplayer. Also worked on loading that same file back into the editor as well as the playtest button, which allows one to run the game directly from the editor. Greg also worked on the design document for more elaboration.

Kaspar Poland

Refactoring code to adhere to clean architecture, communicating with TA on problems and implementing solutions, managing overall team direction. Worked with Greg to implement all of the RDF classes needed for saving, loading, and playtesting.