TextGameEngine Design Document

Phase 2

Specification

Description

An all-in-one text-based game editor AND player. A user can create a text-based game with the editor and play that same game in the player, with lots of features.

When a user starts the program, they are shown the title screen which prompts them to launch the editor or the game player (or quit). Launching the editor will bring the player to a blank canvas consisting of 5 buttons, add slide, save, load, playtest, and change theme. The save and load are simple, save the current state of the editor and load it back in. A user can add a slide via the "add slide" button and can edit this slide to have as many decisions or as long a dialogue as needed. Decisions point to other slides that are added, and an entire game is able to be made.

Slides have a "set first" option, which will tell the game player which slide is the starting slide. Each decision has an edit menu which contains the dialogue, the conditionals, as well as an item box. The item box allows the user to choose which item to give when this decision is chosen. The conditionals let the user decide which items or chosen decisions are needed to choose this decision option. Each decision has input and output anchors so the user can drag these and assign the decision to other slides.

The "Save" button saves the editor state to a TTL file using RDF ontological terms. Each value from the editor (the decisions, decision attributes, slides, slide locations etc) is saved into this file so that it can be properly loaded back in at any time.

The "playtest" button calls the gameplayer, which runs the current state of the editor as a game! The gameplayer is also accessed by clicking "Launch Player" from the main start screen. Themes can be changed and the speed at which the dialogue shows up can be changed in the settings. In the player, the user can load a TTL file that was saved from the editor. At the top is the dialogue of the slide and the bottom contains the decision dialogues. When a user clicks one of the decision dialogues, the outgoing slide attached to that decision is now displayed. There is also an inventory at the bottom to display the items collected.

Our specification from phase 0 and 1 has kept the same spirit of the program, but has changed drastically. Things aren't launched in the same order, slides and decisions work a bit differently, and the player now has its own title screen! We were sure to add our new classes and methods to follow clean architecture and SOLID principles as highlighted in the below document. We're very proud of how this turned out.

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.

On top of the major design changes from phase 1, we have definitely made some big new ones. After phase 1, we had our shell program basically done and all that was needed was some extra features, and there were a lot in lots of aspects.

Firstly, we decided to have a new way to run the program. Instead of having two main functions that run the player and the editor separately, we implemented a main title screen that allows the user to choose. This is just good program design in general, why have two mains?

Secondly, we revamped the entirety of the gameplayer. It used to be in Swing, but our team did an entire rework and recreated it in JavaFX. It was only necessary, it improved our ability to have contact between the editor and the gameplayer, as well as share some classes (like the themes). Because of this, we were able to add the main menu!

Thirdly, the entire look of our program changed. Slides and decisions are now readable and far more interactable. Before, to edit a slide, one would have to open an entirely new window. This new design removes the need for a window altogether. There is no longer a “Set slide dialogue” button since the dialogue updates with every keypress. Same goes for decisions. Themes were added and are communicated across the player and the editor.

Lastly, two large features were added. Conditional decisions and items. This is revolutionary to our program because the game is no longer just a tree-traversal type game. Now, a player may need 5 coins to be able to progress through the desired path, or the player must have taken a particular route to get a secret ending! Of course, this is all reflected in the RDF as well.

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 and 2 allowed us to adhere to the SRP closely. In addition we increased the types of GUI classes as well. This meant that instead of having a small number of handler classes or a small number of GUI classes with too many functions we could have a large number of classes with a more defined role. Our Phase 1 and 2 refactoring made this goal a reality.

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.

The themes make great use of the Open/Closed principle. For all locations in the code that use the themes, it all relies on the same instance of a class. This theme class can then add another 500 themes, and all themes will be reflected in the theme pickers immediately.

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. Since all examples of parent classes and interfaces follow the Liskov Substitution Principle our entire project follows this 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. Our use of interfaces throughout the code, namely PlayDisplayer, allows us 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.

Not much has changed since phase 1. It worked so well for us there wasn't much else needed to change. We still persisted with the separate branches and pull requests and the merge assistant. Discord was still the primary use of our issues and other communication.

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.

One of the largest refactors in phase 2 is changing the gamerenderer from Swing to JavaFX. This was done, as mentioned previously in the document, to be able to communicate better with the rest of the program. Themes were supported by javaFX and Swing in different ways, so this helped remove a theme class entirely.

Lots of classes had some theme specific refactoring. setTheme() methods appearing to remove some repeated code in child classes and other theme-changing calls. This helped us remove some code smells and other bad design elements.

RDFLoad went through a good deal of refactoring. In phase 1, it was fine and not very bloated, but during phase 2 with the new features, it became a mess. We decided to separate the methods of RDFLoad into different classes to avoid this hard to follow, foreign RDF code.

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!

The program remains fully functional as it was in phase 1, just with a whole lot of new features. Everything in the specification was updated to reflect what the program can do. As usual, a full tutorial will be shown to the TA in the presentation, showcasing all the new features!

Progress Report

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.

Work Since Phase 1

Kevin Portinga

Since phase 1 I have extended a number of features of our software. I have added an inventory system, allowing decisions to give items which can later enable specific future decisions. Additionally, I made decisions able to rely on other decisions, allowing a player's previous choices to be factored into what options they have available to them in the future. Finally, I added pop-up error boxes that appear if a user attempts to save or playtest an invalid game.

https://github.com/kdports/TextGameEngine/pull/40 <- This pull request was my addition of decision conditions to the project. This is our most significant feature extension in phase 2 and is extremely helpful for users. It built on pre-existing entity, use case, and GUI work.

Kevin Li

Refactored the code for game player from Swing to JavaFx alongside Jason you to keep the consistency of the GUI layer. Added an inventory button that allows you to check what items you have during the game. I also added the title screen for the game along with a settings button that allows you to choose which theme you want the game to use and what speed the animation for the text should have.

https://github.com/kdports/TextGameEngine/pull/46 <- This pull request was me and Jason You's refactoring of the game player from swing to JavaFx and it was also my addition of the title screen to the game player. It was combined with Greg's changes to the themes.

Jason You

Rewrote all of the code related to the game player to use JavaFx instead of Swing so that our code wasn't using 2 packages for the GUI. Also made the main title screen that either opens the game player or editor. Fixed various bugs and made many small qol changes.

https://github.com/kdports/TextGameEngine/pull/12
This pull request was my addition of a fully working basic game player. This is a fundamental aspect of our program as there isn't a point in making a game if you can't play it.

Jason Sastra

Added scrollbar into editor game so that it is possible to make large games. Helped with implementing the theme editor. Implemented TestFx to test game editor. Fixed various small bugs regarding the editor game.

https://github.com/kdports/TextGameEngine/pull/20

https://github.com/kdports/TextGameEngine/pull/21
The overall JavaFx game editor that is integrated to be able to load, delete, edit, connect, and change slides and decisions in the editor game. Both of these pull request does that

Greg Sherman

Revamped the entire look of the editor. All slides, decisions, and window changes were made to follow a nice new look. Added the theme capabilities for the editor (with the help of Jason Sastra on the slides and decisions) and for the player. The themes can be infinite due to the open/closed principle. Made the program much easier to use, can type directly into a box to have it save, no more window for the slide, removed a lot of redundant buttons and dialogue. Implemented RDFSave and RDFLoad with kaspar to support the new decision and item conditionals. Wrote multiple sections of this design document as well.

Pull request 55: https://github.com/kdports/TextGameEngine/pull/55
This is the updated RDF saving and loading, it's a crucial part of the game that was changed to support conditionals.

Pull request 46: https://github.com/kdports/TextGameEngine/pull/46
Added the theme ability and applied it to the new javaFX gameplayer and the editor.

Kaspar Poland

Worked on the RDF Ontology that outlines the metadata of the application, all stores in TGEO.java. Worked with Greg to implement the RDFSave and RDFLoad functionality with working inventory conditionals, working decision conditionals. Worked through creating new instances of an RDF list. Changed the storage of decisions from their in-memory toString() outputs to their RDF URIs, making our .trig output file entirely self-sufficient (could be imported anywhere and understood).