D&D Companion

Project Documentation

Development Process

Technologies Used

Sections

Gabriel Rosa da Silva

06/11/2024

Summary

[1. Project Idea 2](#_Toc183256767)

[2. API 3](#_Toc183256768)

[3. User Interface 6](#_Toc183256769)

[4. Game Tool Kit 7](#_Toc183256770)

[5. Archived System – MySQL Database 8](#_Toc183256771)

[5.1. Naming Rules: 8](#_Toc183256772)

[5.1.1. Category\_Case 8](#_Toc183256773)

[5.1.2. Plurality 9](#_Toc183256774)

[5.1.3. Object Tables 9](#_Toc183256775)

[5.1.4. Relational Tables 9](#_Toc183256776)

[6. Archived System – Java Data Structure 10](#_Toc183256777)

[6.1. Concepts and Implementations 10](#_Toc183256778)

[6.1.1. Atomic 10](#_Toc183256779)

[6.1.2. Element 10](#_Toc183256780)

[6.1.3. Group 11](#_Toc183256781)

[6.1.4. Bundle 11](#_Toc183256782)

[6.1.5. Measure 11](#_Toc183256783)

[6.1.6. Technique 11](#_Toc183256784)

[6.1.7. ElementCluster 11](#_Toc183256785)

[6.1.8. GroupManager 12](#_Toc183256786)

[6.1.9. Convertible 12](#_Toc183256787)

[6.1.10. Example 13](#_Toc183256788)

1. Project Idea

This project main goal is to train the authors’ software development skills, togheter with incetivizing studying and research.

It was broght up by the author Gabriel Rosa da Silva, which is a fan of the tabletop rpg system D&D 5e. The final product is supposed to be like a game-engine, therefore, in our conceptualization, a system which automatically deals with anything that doesn’t require human choice.

The necessity came from the fact that, eventhough there are multiple similar applications in the market, most (if not all) don’t fully satisfy a player’s needs.

First of all, they are *freemium*, a generalized term implying the service offers basic functions for free, but it’s full potential can only be unlocked through some type of payment.

Second, their implementations are more akin to canvas than to a game-engine. They allow the creation of custom components, but don’t offer programmed actions between those components. Using them resembles more documenting the game you are playing than in fact helping play the game.

1. Local Storage

Our main storage system will consist of a local directory named *data*, in which all the information will be stored in json objects. The reason for doing so is the fact that we want to make the data as accessible and flexible as possible. Honestly, our local storage is prety much a NoSQL Database, but more strongly coupled to the source code.

Doing things this way allows any user to easily add data to the application just by creating a new json file. Obviously, for the system to work correctly, the data needs to be in a particular shape, so data validation is key for correct functioning.

* 1. Directory Hierarchy

Inside the folder *data* there are more folders in which we’ll call *collections*. They will classify the information in order to define the shape of the collection’s data. For example, D&D armors will have a specific collection called *armors* and all of its objects will have the same json structure. All collections are sometimes called *groups* and they are declared in the section **Data Collections**.

* + 1. Data Collections
* armors
* weapons
* items
* options
* campaigns
* monsters
* characters
* NPCs (non-playable characters
  1. Code Description

Everything related to the local storage system is mediated through a Java Maven application. The json manipulation is made using the Jackson library and the local files are manipulated with Java’a io and nio libraries. The application is subdivided into smaller packages as follows: *data*, *data build*, *data handling*, *data validation*, *storage build* and *system components*.

* + 1. Data Package

This package works to define and manipulate all necessary data objects. It is subdivided into two smaller packages: *structure* and *manipulation*.

The *data structure* package defines the shape of each stored object, which attributes each element has and how they relate to each other. They are all defined through Java Records, since they don’t have any actions imbued to them, only states.

The *data manipulation* package implements functional classes that defines how each data object is treated. Some classes need

1. API

The system’s API will consist in a Spring Application.

1. User Interface
2. Game Tool Kit
3. Archived System – MySQL Database

Our storage system will consist in a MySQL bank, which will define three types of tables in a hierarchy ordering. Each table can be either an object table, relational table or a option table.

An object table is simply the database implementation of a singular and (mostly) independent object.

A relational table is one that references objects interchangeably, defines the relational classifications which includes multiple options (1-n or n-n). It may also add primitive fields intrinsics to the relation. For 1-1 relations, a simple reference is made through the implementation of an Foreign Key Column.

An option table is a subcategory of object tables, in which it defines not the object itself, but labels about the object. They are used in cases where the option-object only implements an attribute type that has fixed possibilities, like an Enumerator or a data type. As an example, the table for *ability scores* is labeled as an option table since the *ability scores* themselves aren’t a singular object, but a type for a general ability score object, being only turned concrete at the object *player*’s definition.

The relational map between tables defines a tree-shaped hierachy. Objects at the root of the tree don’t reference any other object and sets a base for every other referencing object. Objects at the leaves of the tree don’t have any references being made to them, mostly defining actors with lots of states and actions in other systems.

Since “root” objects don’t reference any other database object, they are called Atomic, resembling the idea that there’s no database object smaller than them. As for “leaf” objects, they are called Final, since they are literally at the end of a branch.

* 1. Naming Rules:
     1. Category\_Case

All names uses a combination of snake\_case and PascalCase, which we’ll call Category\_Case. In Category\_Case, the defined name has subcategories inside, such as prefixes or sufixes. In the category’s part of the name, PascalCase is used. Between categories, there must be an underline, like snake\_case.

As an example, let’s use *FRK\_Class\_Id\_REF*, where *FRK* represents a Foreign Key label prefix, *Class* represents the object, *Id* represents the attribute and *REF* represents a Referencing label sufix.

* + 1. Plurality

Whenever there needs to have plurality in a name, it can be switched to a collective noun, but only if its meaning is clear. Otherwise, it must be noted down in the documentation and the code.

* + 1. Object Tables

All object tables must have the prefix OBJ and mention said object in plural in their names.

* + 1. Relational Tables

All relational tables must have the prefix REL and all objects of the relation in their names. The relations are declared in the plurality of the objects in the table name. As an example, an 1-n relational table must have its first object in its name in singular form, while the second one must be in plural form.

1. Archived System – Java Data Structure

The game data structure will be define in a Java Application and it’s main purpose is defing pre-determined objects and their actions. Since D&D 5e has lots of different objects, each with their own rules, it’s important to set appart the main application from the objects defined for the program. Otherwise, it will give too much responsibility for the game application, leaving the development tree a giant mess and hindering performance.

* 1. Concepts and Implementations

We’re using *component* as a generic therm for groups of code built inside the system with states and actions. It doesn’t take into account it’s implementation, only the concept, states and actions.

Below all definitions, we’ll folow trough an example for building components and how to implement each component type.

* + 1. Atomic

Components deamed *atomic* are the smallest data structure possible for objects. They are divided in some categories and sub-components, to help guide their implementation and usage. They are as such:

* + 1. Element

An Element component is the system’s representation of D&D’s objects. It is the smallest part of an Atomic component, representing only the data and its data-types for every single object defined in the rpg system. It’s divided in two types: a standard element and a custom element. Being standard means it is pre-built inside the code where as being custom means it needs to have information inserted externally.

You may relate standard element to a built-in database, where every attribute for the element is define whitin the component’s definition. For custom elements, you may relate them to a casting mold, where only the general shape is defined and the data is given by something outside the component.

The Element component is an interface which defines only an action for extracting a *name* state. It’s more of a abstraction component than actually a contract. The reason for being an interface, rather than a abstract class, is the way it’s implemented. For the purpose of easy usage, standard element components are implemented as enums (since it’s meant to be class with constant intances) meaning it can’t extend from another class, only implement interfaces.

An element can be further categorized into a NormalizeableElement. Such component extends from Element and implements another semi-state. For elements that can be converted into each other through the use of a multiplicative process, we implemented a normalizing factor. This factor represents the relation between each element and it is the constant by which the multiplicative conversion occurs.

* + 1. Group

A Group component is the smallest functional part of an Atomic component. Elements can’t do anything by themselves, they are only data, so it’s the group component’s resposability to define individual actions for each element component, along side with extra states not defined in a element.

Some Groups can be further categorized into Bundle, Measure and Technique, each of which extends Group and defines some basic actions and states for elements. Bundle and Measure components can use normalizeable elements in their construction, since they allow a multiplicative conversion between groups.

* + 1. Bundle

A Bundle component is a Group which has an integer amount of its element. It defines the state *amount* and actions to increase or decrease said amount.

* + 1. Measure

A Measure component is a Group which gives a decimal value for its element. It defines the state *value* and actions to increase or decrease said value.

* + 1. Technique

A Technique component is a Group which applies inside it the Proficiency component, which will be better defined later. Generally, tehcnique components represents D&D actions, not system ones, and the proficiency component defines how well an actor can realize such actions.

As its concept suggests, a technique component differs from a normal group component in a unique way: it depends on it’s actor proficiency, not only the elements implemented.

* + 1. ElementCluster

An ElementCluster component is a collection of Elements that allows easy implementation for large amounts of elements instances. It defines a *values* state responsible for storing every element related to the cluster, along with four actions: *to\_array*, *remove*, *gather* and *merge*.

The state *values* is a Java’s Set collection that has the cluster’s elements as entry. To access an entry, you must iterate over all entries and filter the chosen one.

The action *to\_array* returns a simpler collection of element components, an array, since some Java actions work better with arrays.

The action *remove* takes away some chosen collection of element components, shrinking the cluster.

The action *gather* adds some chosen collection of element components to the cluster.

The action *merge* joins togheter different clusters into a single one.

* + 1. GroupManager

A GroupManager component is a collection of Groups that allows easy implementation for large amounts of group components. It defines a *mapping* state responsible for storing every group related to the manager, along with four actions: *to\_array*, *remove*, *gather* and *merge*.

// Refactor below text

The state *mapping* is a Java’s Map collection that has a generic key and the manager’s group component as entry. The map key is generic because it’s the way to access its Group and how this access is implemented may be different between managers, for easier usage.

The action *to\_array* returns a simpler collection of group components, an array, since some Java actions work better with arrays.

The action *remove* takes away some chosen collection of group components, shrinking the manager.

The action *gather* adds some chosen collection of group components to the manager.

The action *merge* joins togheter different managers into a single one.

* + 1. Convertible

Convertible is a type of Group that allows conversion between group components. It’s an interface that implements two actions: *convert\_to* and *convert\_new*. The latter converts the group’s element component to a different element. The former, creates a new group component based on the original component but with its element converted into another one.

* + 1. Example

As an example, we’ll conceptualize the data-structure of a simple yet very important part of D&D, weight measures.

First, we create a interface Dice extending from Element, defining getters for important attributes related to an n-sided dice. The implemented getters will serve as abstracted attributes for the element. Third, we define two additional components that implements from Dice: an enum StandardDice and a normal class CustomDice.