SE Lab – **Final Project Description**

- Individual Phase -

October 21, 2022

Vianney Jerry Tchouwa Takou (Matriculation Number: 7013919)

Functionality

My server is based on the reference implementation. I tried to retain the MVC architecture as much as possible. To achieve this, I used a command pattern for the handling of the casting of spells. These spells use the same return type as the commands for indicating how the state machine should continue after casting a spell. They will be casted in the combat phase if they are triggered by the player. The cast method nearly all return a **PROCEED** enumerator of **ActionResult**. The only difference is the structural spell which returns **ENDGAME** if a tile is conquered. Helper functions and fields have been added to the **Player**, **Dungeon**, **TunnelGraph**, **Model** and **Adventurer** to make the casting of spells possible and easy to handle. Important helper functions or fields will be explained below.

Concerning the configuration parser, a method, **parseSpells ()** has been added which will parse a JSONObject of a spell and pass it to one of the builder methods (**addResourceSpell (), addBuffSpell (), addRoomSpell (), addBiddingSpell (), addStructuralSpell** ()), which have also been added in the **modelBuilderInterface**, for validation and building. The Boolean parameter, other, which is found in each parameter simply specifies if an attribute that a spell doesn't normally contain is present. If it is the case, these overridden methods in the **modelValidator** will throw an exception. A **magicPoint** parameter has also been added to the **addAdventurer ()** method.

Regarding the extension of command, I decided that the **castCounterSpellCommand ()** will extend from the **playerCommand ()** class since it is a command that a player may only submit if it is his/her turn. Moreover, a **findCounterSpell ()** method has been added to the **DigTunnelCommand ()** to calculate the probability of getting a counter spell. If a spell has been found, it is then added the field counterSpell, which is present in the player class and has a type int. When the player wants to use a spell in the combat round, the useCounterSpell () method is called which decreases the number of counterSpells he has.

A good worry now might be how I'm going to deal with spells. Well, the Player will basically be assigned a list of spells to withstand, which is represented by a hash map of season to lists of spells. The triggering of spells is done before the evaluation of the bid, since at this time, all players bids are placed. This is made possible through a spell decorator which decorates the bid with the specific spell. Here's the procedure: The model will reveal if there is an available bid that matches the current **bidType ()** and the slot of the bid to be evaluated. If it's the case, the bid is decorated with a **spellDecorator ()** to a spell bid. The **spellbid ()**

assigns the spell to the lord before calling the evaluation method of that the specific bid, which it has as field.

Additionally, the representation of the archmage Linus is done with a simple Boolean in the dungeon, which specifies if he is present or not. If he is present, details associated to his presence will be handle accordingly. That is, the fatigue points, the fact that when, it turns to true, a monster shall be removed, the addition of magic points when the Boolean is true.

Let me now try to give a brief summary of the model and adventurer changes. Important fields like a list of spells and a list available spells have been added to the model to handle the spell data, **ShuffleCards ()** has been modified because of the additional spell cards and **drawSpell()** has been added to handle the drawing of cards from the stack to the available once. **GetAvailableSpell ()** has been added to return the list of spells that might be triggered at a specific **bidType** and slot. Concerning the adventurers, the field **magicPoints** has been added to the adventurer. It is going to be use in the combat phase to calculate the total number of magic points. Fields, that describe the original value of the adventurer have been added, to handle the restoring to the original value, if they surpass the maximum. The helper method that was added for this purpose is the **restoreOriginalValue()** which will restore fields to their original values if they exceed the maximum because of the buff spell.

Let's come to the changes in the player. I already introduced the fields **counterSpells** and **spellMap**, which are basically the number of counter spells the lord has and the mapping of season to list of spells, which the lord must withstand in the combat. The fields **noActivatingSeason** and **blockedOptionBySpell** are basically a list of seasons, in which the lord can't activate rooms and a mapping of season to list of blocked bids. They are going to be use in the room spell casting and in the bidding spell casting methods respectively. The the most important functions added are: the **addSpell()** method, which adds a spell to the **spellMap** and the **blockOptionBySpell()** method, which adds a blocked bid to the **blockedOptionBySpell** map.

A last important change is the one done in the **tunnelGraph**. The function **closestRoomToAdventurer ()** returns an optional Tunnel which contains the closest room to the battleground. If the tunnel is present, then the room is also present, the destruction is done in the structural spell cast method.

Design Decisions

My changes make use of the two different design patterns: Command and Decorator pattern.

- **Command Pattern**: I use the command pattern to model the casting of spells since every spell has a specific or special way of casting. The command pattern ensures the encapsulation of the logic of casting from the rest of the game and make things flexible for extension.
- **Decorator Pattern**: I needed a way to add a functionality to the evaluation of bids if it is associated with the triggering of a spell. A decorator has been proved to be good in this kind of situation. It ensures more flexibility and is easy to add.

- **Archmage Linus:** I described the Linus in my design with a Boolean in the dungeon. Although I'm modifying already existing code, it makes the handling of Linus easy. The presence or absence of Linus will be done by a simple setter. While setting the field to true, a monster is automatically removed from the lord and every other logic associated with the Linus will happen depending on the Boolean.
- **Counter spell:** The counter spells are represented with just an int because they don't have a special logic.
- **Blocked options by biding spell:** They are basically represented with a hashMap of integer to list of bids because I need to keep track of the season in which the bids are going to be blocked. This is the same design decision I made with the **number of spells that a player can trigger.**

Challenges

The most challenging part of the extension was to find how to deal with the spells. Especially when it came to how the spells were going to be assigned to the lords and in which season, they were going to be used. I solved this with the introduction of a hashMap which binds the seasons to the specific list of spells and the decorator pattern which helps for the assignation of spells before evaluation.

Update

- ### Spell class and sub classes changes
  The difference with the new implementation is the introduction of another cast method which is **abstract, returns void and only visible to subclasses**. This is the cast method that is overridden by all subclasses of spells. The cast method that returns action result calls before everything another function **castAndCounterSpell(),** which sends the spell cast event and checks if the player can counter the spell, if it's the case, it sends the **the sendCounterSpell** event and asserts for a command.
  I made this change because I noticed that all these spells were doing **the same thing** at the beginning so it's a good idea to factor it out and encapsulate it somewhere.

- **Configuration Parser change**
  There is no big change here but to my mind it makes sense to explain that in the config parser, the class Number is used to determine if an attribute is present or not. Knowing this I can do checks that are going to help in the model validator.

- **State. Phase change and relation to castCounterSpellAction**
  The **state SPELL** has been added to the state, because the **castCounterSpellAction** can only be received in a single state which is of course not the combat.

- **Dungeon changes and lord changes**
  The field **penguinVisits** has been added to the game, to have the number of times the archmage linus came into the dungeon. In the lord the functions **canBlockBid()** and

buildingEnd() have been added. The **canBlockBid()** checks if it’s possible to block a bid type and **buildingEnd()** remove all bids that have been blocked by spells and clear the list of no room activation season. It is called at the end of each building phase. More on the player, to get the list of spells to withstand in a round, the **spellToWithStand ()** function was added. It has as parameter the round in which the spell has to be withstood and returns a list of spells, which correspond to the spells, the lord must withstand in that particular round.

- **Building state changes**
  For the purposes of complexity, I had to divide the evaluation of option in the bidding state. I added a method **evaluateBid,** which evaluates the bid and decorate the bid with a spell bid if the slot is a slot in which a spell must be triggered. If it returns false, the game must end.

- **Combat state changes**
  The method **dealWithSpells (),** has been added to the combat, which deals with the casting of spell logic. It checks if there’s enough magic points and calls the cast method if it’s the case. The return value determines how the game should continue.

- **System Test design**
  I would like to shortly introduce the system test design I did. I implemented it with the help of two helper classes (**SystemTestWrapper)** and **SystemTestBroadcasts**. These two classes provide functions that are helping for bidding, evaluating, and broadcasting events. The big Thing here is the **SystemTestTemplate** which is a template of the game! All scenarios inherit from this template and override functions that are specific to them.
  I made the choice of a template pattern here because all the scenarios have the same blueprint but differ in specific functions like evaluations and rounds of combat. The template provides a blueprint for testing 1 or 2 players.