Software Architecture Design

Jenny Hansdotter, Dina Jawad & Felicia Rosell

Software Architecture

This section aims to give an overview of the software architecture. Figure 1 below illustrates the core functionality classes of the game and how they utilize each other. For example, **AttackHandler** uses **DamageHandler** to deal damage. Most of the game logic is handled in these classes. All card specific game logic is contained in the class **BuffMethods**.

Figure 2 shows both the classes contained in Figure 1, and the classes involved in the Observer pattern communication.

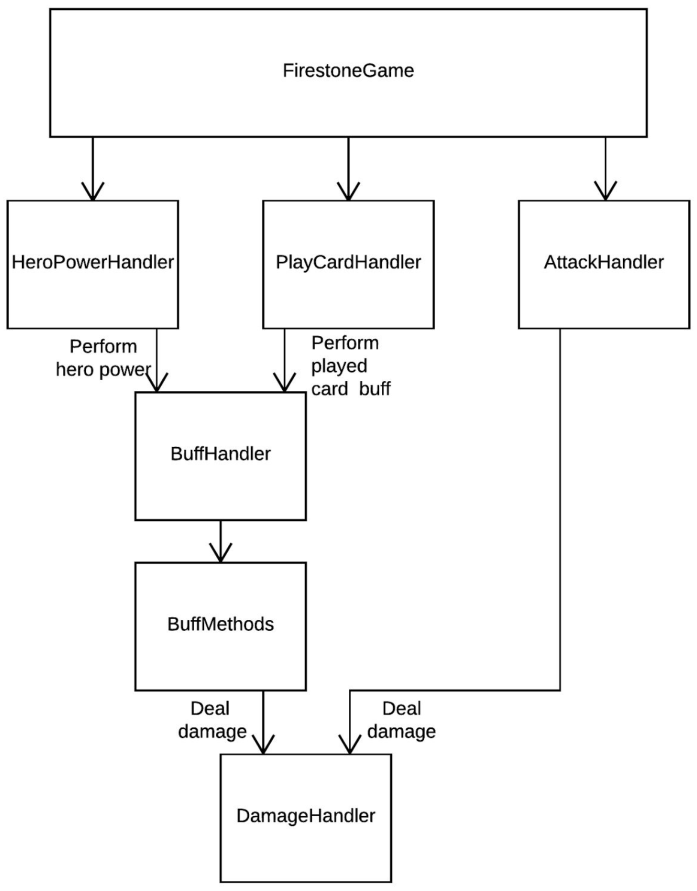

Figure 1. Illustrates the most important classes and how they utilize each other (with the exception of Observer pattern functionality)

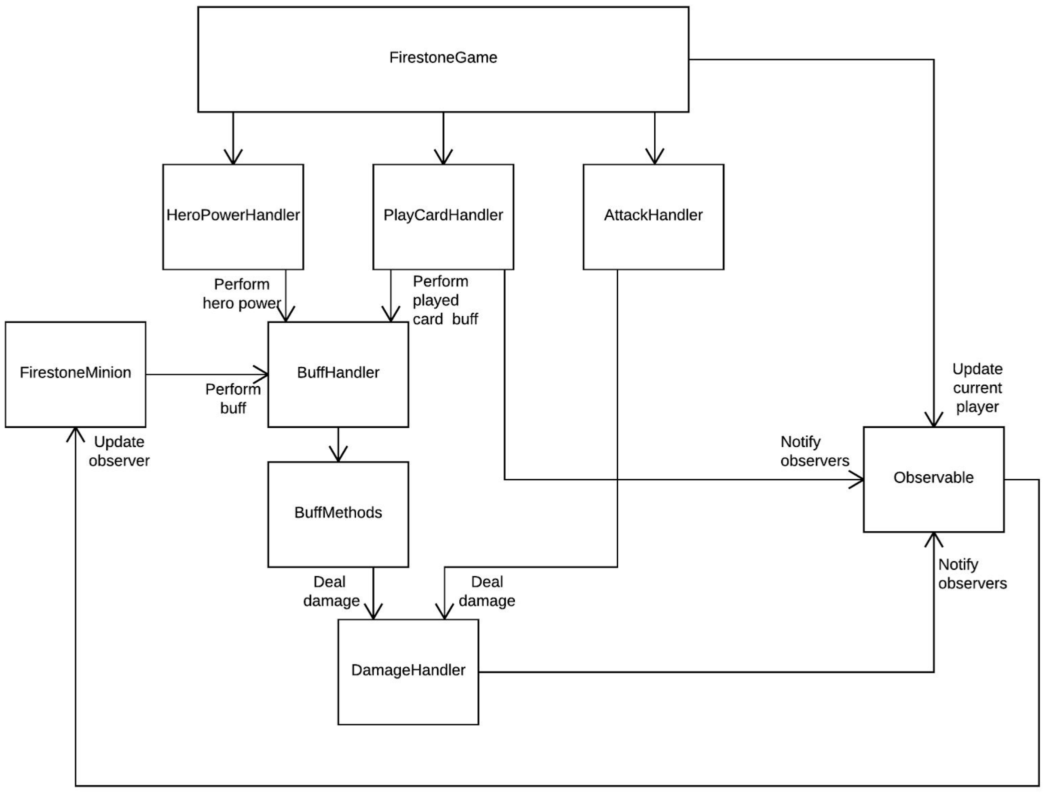

Figure 2. Illustrates the most important classes and how they utilize each other, including the Observer pattern functionality

Adding New Cards

This section is a guide that explains the steps needed to implement new cards in the game. Each relevant class used to create cards is also listed in Table 1 and explained to give an overview of the functionality. It is advised to have a look that Table 1 first.

The first step when adding a new card is to add its data into **GameData**. Once that is done, the card now exists in the game and can be used. However, if the card has a buff, it needs to be implemented. This is done in **BuffMethods**. There are three separate sections in **BuffMethods**, one for implementing minion cards, another for spells and finally one for hero powers. (**GameData** is structured in a similar manner.)

Create a new method in **BuffMethods** and have a look at the structure of the other methods already implemented there, as well as their documentation. The method may need to make use of the *performBuff* variable (see Table 1) as methods implemented in this class also are called to check if a buff can be performed without actually performing the buff.

After implementing the method in BuffMethods, add a test method in BuffMethodsTest for testing the logic in the method.

Finally, add the name of the new method into *methodMap* found in the **BuffHandler**.

If the new card's buff requires additional information that is not available in **Action** at the moment (see Table 1), it can be added to **Action**'s constructor. Doing so will require an update of all **Action** objects created in the project. Currently, **Action**'s are created in **PlayCardHandler**, **DamageHandler** and **HeroPowerHandler**, including some test classes.

Table 1. Important classes for implementing cards

CLASS	USAGE
BuffMethods	This is where card buffs are implemented. Each method in **BuffMethods** must have the following three parameters: **Action** *action*, **Minion** *minion*, boolean *performBuff*. This class is used to perform buffs and to check if it is possible to perform them, without actually doing so.
BuffHandler	This class handles invocation of buffs. A map called *methodMap* is stored in this class that maps buffs on the cards to their respective method in **BuffMethods**.
GameData	This class stores a HashMap with card data. Minion and spell cards, and hero powers have the keys: *name, health, attack, mana, race, state, buff, type*. Hero powers also have the key *class*.
Action	This class stores the event that just took place. Methods in **BuffMethods** can use an instance of **Action** to get information about the state of the game. **Action** stores the following data: *players,*

	currentPlayerId, playedCardId, minionCreatedId, position, targetId, damagedCharacterId, actionType. Some of these values may be *null* for some cards when those values are not relevant the action.
DamageHandler	This class is used in **BuffMethods** to deal damage to minions and heroes. It also removes dead minions from the board.

Performing buffs

This section describes how the buffs of the cards or minions are performed during the game. Buffs happen in one of two stages, either when a card is being played or when a character has a buff which is triggered by a card being played or damage being taken. These two situations will be described below.

Relevant classes for invoking buffs are **BuffMethods**, **BuffHandler**, **Action** and **DamageHandler** which are described in Table 1, and **PlayCardHandler**, **FirestoneObservable** and **FirestoneGame** described in Table 2.

Table 2. Important classes for handling of buffs

CLASS	**USAGE**
PlayCardHandler	This class is used by **FirestoneGame** to play cards. It removes the card from the player, performs the buff of the card and, if it is a minion card, creates a minion and places it on the board.
FirestoneObservable	Extends **Observable** class. Contains a list of the observers and methods to operate on the list. It also has knowledge of the players and who the current player is. This is used by **PlayCardHandler** and **DamageHandler** to allow them to have access to the same list of

	observers.
FirestoneGame	Modifies the current player in **FirestoneObservable**.

The first situation when a buff needs to be invoked is when a card is played, and the card is a spell card or a minions card with a battlecry or combo ability. This is done by the **PlayCardHandler** creating an **Action** object (see Table 1) with the information about the played card, which is needed in the **BuffHandler**. The **BuffHandler** then uses this information to find the right buff, see if the buff should be invoked or not and then **BuffMethods** is called.

The second situation when a buff of a minion needs to be invoked is when a minion is triggered by a certain card being played or damage being taken. This means the minion needs to be notified of what has happened so the buff can be invoked if the circumstances are right. For this reason, minions are set to be observers of both the **PlayCardHandler** and **DamageHandler**.

When a card is played or damage is dealt an **Action** object is created with information of the played card (what type of card, if minions are involved and which they are) or a damage incident (who got damaged). This is then used as the argument when notifying the observers and the observers can then call the **BuffHandler** to see if their buff should be invoked. For **PlayCardHandler** or **DamageHandler** to create an **Action** they need to know which player is in turn. This knowledge is contained in **FirestoneGame**, which therefore communicates this to **FirestoneObservable** each time a turn is ended. The information can then be acquired from **FirestoneObservable**.