HearthSim

Milestone 3

Stephanie Li - 6422659
Erik Stodola - 6085210
Geoffrey Liu - 6586759
Robert Wen - 6461689
Georges Katsaros - 5979889

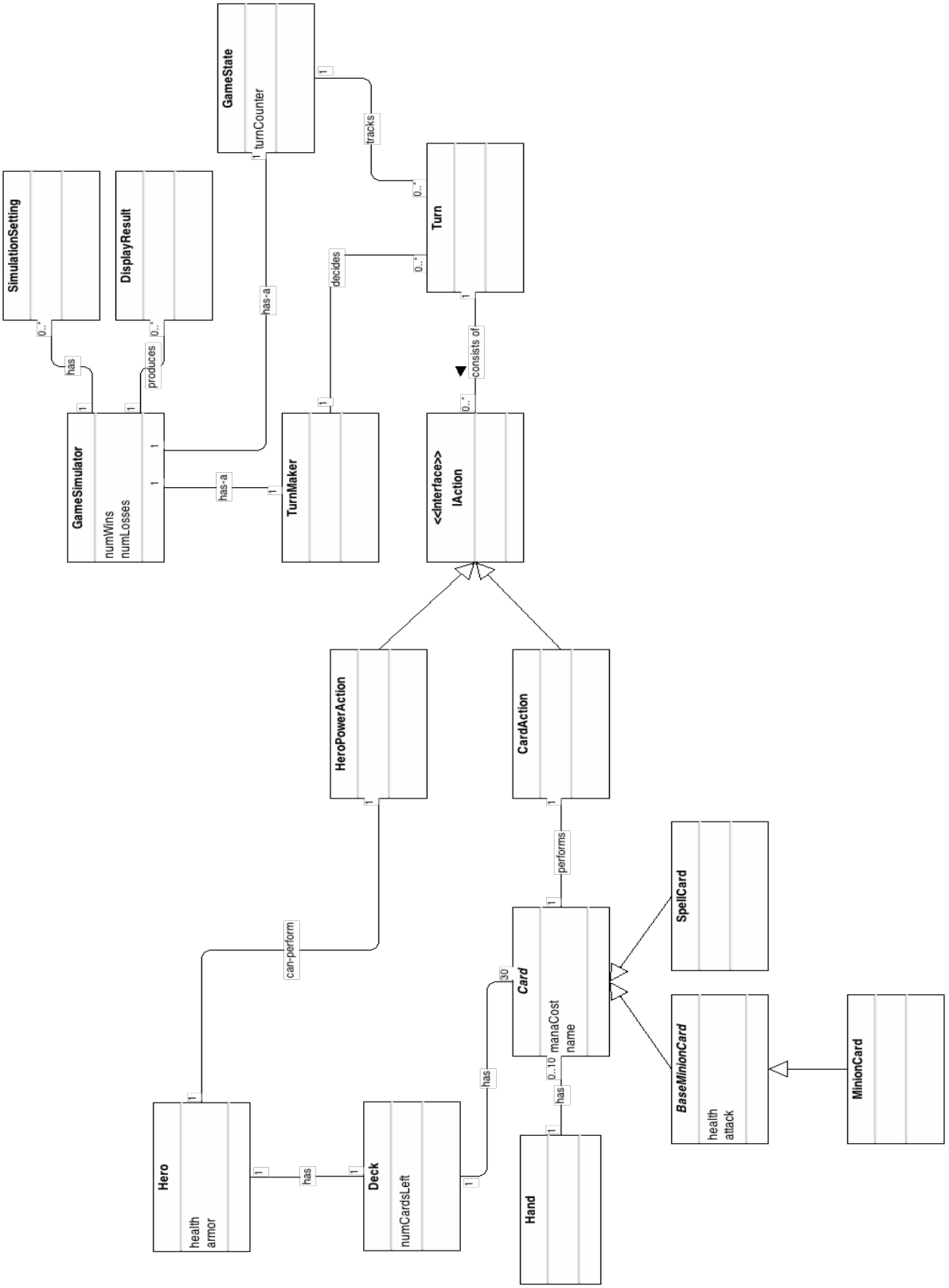
Concordia University
SOEN 343: Software Architecture and Design I
Prof. Peter C. Rigby, PhD

Summary of Project

The main goal of the simulation tool HearthSim is to provide data on game strategy for the card-based video game *Hearthstone*. *Hearthstone* gameplay largely involves building and playing a deck of cards against an opponent. Thus, the optimization of a deck's composition is an important part of the *Hearthstone* player's strategy. Usually, a player decides the composition of a winning deck by playing many *Hearthstone* games and observing the deck's win rates. HearthSim informs this deck-building process in a faster and more efficient manner by allowing the simulation of *Hearthstone* games using custom built decks. HearthSim's game simulator is able to run hundreds of AI versus AI games in a short time span and outputs the resulting win rate (and other data) of the user's custom built deck. In this manner, HearthSim provides a player useful insight into deck building much faster than he/she could obtain by playing real *Hearthstone* games and ultimately increases the chances that a player can win more matches.

Link to forked project on Github:
https://github.com/sli-fox/HearthSim

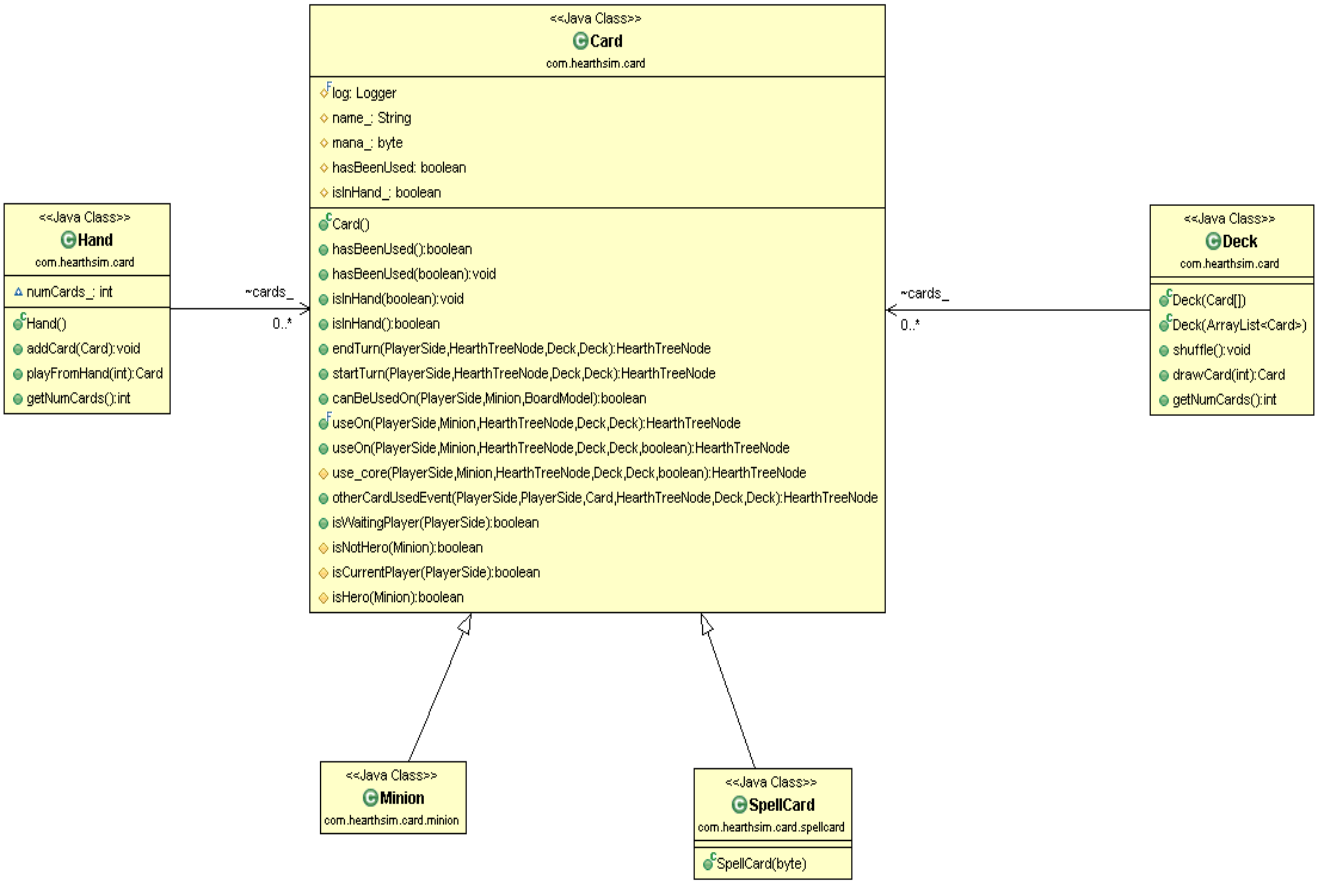
Part 1: Class Diagram of Ideal System



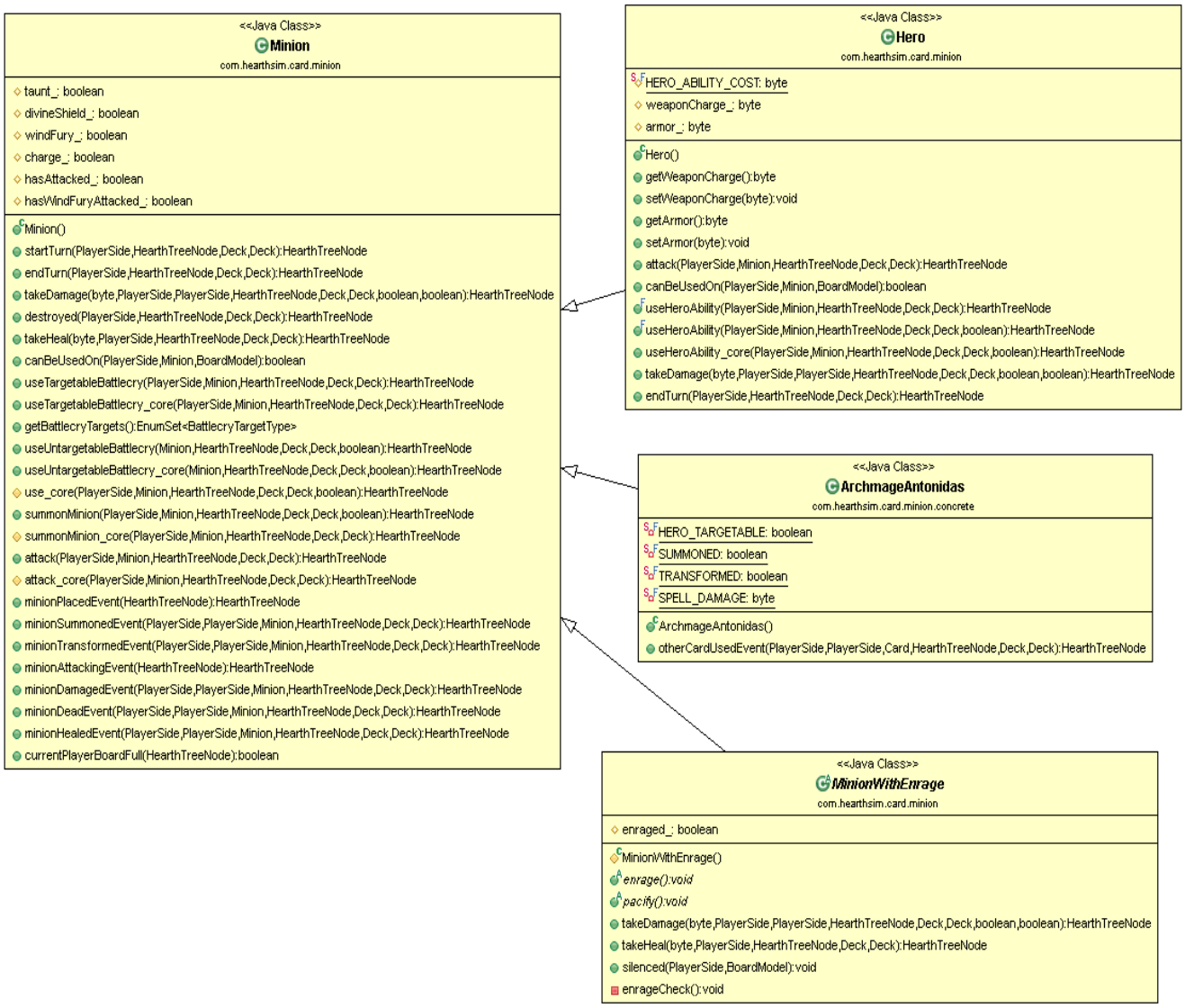
Class Diagram of Actual System

Card

Note: Card acts as a superclass for Minion. Minion is depicted in more detail in the Minion class diagram.



Minion


<<Java Class>>
Minion
com.hearthsim.card.minion
taunt_: boolean
divineShield_: boolean
windFury_: boolean
charge_: boolean
hasAttacked_: boolean
hasWindFuryAttacked_: boolean
Minion()
startTurn(PlayerSide,HearthTreeNode,Deck,Deck):HearthTreeNode
endTurn(PlayerSide,HearthTreeNode,Deck,Deck):HearthTreeNode
takeDamage(byte,PlayerSide,PlayerSide,HearthTreeNode,Deck,Deck,boolean,boolean):HearthTreeNode
destroyed(PlayerSide,HearthTreeNode,Deck,Deck):HearthTreeNode
takeHeal(byte,PlayerSide,HearthTreeNode,Deck,Deck):HearthTreeNode
canBeUsedOn(PlayerSide,Minion,BoardModel):boolean
useTargetableBattlecry(PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
useTargetableBattlecry_core(PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
getBattlecryTargets():EnumSet<BattlecryTargetType>
useUntargetableBattlecry(Minion,HearthTreeNode,Deck,Deck,boolean):HearthTreeNode
useUntargetableBattlecry_core(Minion,HearthTreeNode,Deck,Deck,boolean):HearthTreeNode
use_core(PlayerSide,Minion,HearthTreeNode,Deck,Deck,boolean):HearthTreeNode
summonMinion(PlayerSide,Minion,HearthTreeNode,Deck,Deck,boolean):HearthTreeNode
summonMinion_core(PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
attack(PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
attack_core(PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
minionPlacedEvent(HearthTreeNode):HearthTreeNode
minionSummonedEvent(PlayerSide,PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
minionTransformedEvent(PlayerSide,PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
minionAttackingEvent(HearthTreeNode):HearthTreeNode
minionDamagedEvent(PlayerSide,PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
minionDeadEvent(PlayerSide,PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
minionHealedEvent(PlayerSide,PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
currentPlayerBoardFull(HearthTreeNode):boolean
<<Java Class>>
Hero
com.hearthsim.card.minion
HERO_ABILITY_COST: byte
weaponCharge_: byte
armor_: byte
Hero()
getWeaponCharge():byte
setWeaponCharge(byte):void
getArmor():byte
setArmor(byte):void
attack(PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
canBeUsedOn(PlayerSide,Minion,BoardModel):boolean
useHeroAbility(PlayerSide,Minion,HearthTreeNode,Deck,Deck):HearthTreeNode
useHeroAbility(PlayerSide,Minion,HearthTreeNode,Deck,Deck,boolean):HearthTreeNode
useHeroAbility_core(PlayerSide,Minion,HearthTreeNode,Deck,Deck,boolean):HearthTreeNode
takeDamage(byte,PlayerSide,PlayerSide,HearthTreeNode,Deck,Deck,boolean,boolean):HearthTreeNode
endTurn(PlayerSide,HearthTreeNode,Deck,Deck):HearthTreeNode
<<Java Class>>
ArchmageAntonidas
com.hearthsim.card.minion.concrete
HERO_TARGETABLE: boolean
SUMMONED: boolean
TRANSFORMED: boolean
SPELL_DAMAGE: byte
ArchmageAntonidas()
otherCardUsedEvent(PlayerSide,PlayerSide,Card,HearthTreeNode,Deck,Deck):HearthTreeNode
<<Java Class>>
MinionWithEnrage
com.hearthsim.card.minion
enraged_: boolean
MinionWithEnrage()
enrage():void
pacify():void
takeDamage(byte,PlayerSide,PlayerSide,HearthTreeNode,Deck,Deck,boolean,boolean):HearthTreeNode
takeHeal(byte,PlayerSide,HearthTreeNode,Deck,Deck):HearthTreeNode
silenced(PlayerSide,BoardModel):void
enrageCheck():void

BoardModel


<<Java Class>>
BoardModel
com.hearthsim.model
p0_deckPos_: int
p1_deckPos_: int
p0_fatigueDamage_: byte
p1_fatigueDamage_: byte
allMinionsFIFOList_: IdentityLinkedList<MinionPlayerPair>
BoardModel()
buildModel():void
getMinions(PlayerSide):MinionList
modelForSide(PlayerSide):PlayerModel
sideForModel(PlayerModel):PlayerSide
getMinion(PlayerSide,int):Minion
getCard_hand(PlayerSide,int):Card
getCurrentPlayerCardHand(int):Card
getWaitingPlayerCardHand(int):Card
getCharacter(PlayerSide,int):Minion
getCurrentPlayerCharacter(int):Minion
getWaitingPlayerCharacter(int):Minion
getMinionForCharacter(PlayerSide,int):Minion
placeMinion(PlayerSide,Minion,int):void
placeMinion(PlayerSide,Minion):void
placeCard_hand(PlayerSide,Card):void
placeCardHandCurrentPlayer(int):void
placeCardHandCurrentPlayer(Card):void
placeCardHandWaitingPlayer(int):void
placeCardHandWaitingPlayer(Card):void
removeCard_hand(Card):void
placeCardHand(PlayerSide,Card):void
removeCardFromHand(Card,PlayerSide):void
getNumCards_hand():int
getNumCards_hand(PlayerSide):int
getNumCardsHandCurrentPlayer():int
getNumCardsHandWaitingPlayer():int
getHero(PlayerSide):Hero
drawCardFromWaitingPlayerDeck(Deck,int):void
removeMinion(MinionPlayerPair):boolean
removeMinion(Minion):boolean
removeMinion(PlayerSide,int):boolean
addOverload(PlayerSide,byte):void
isAlive(PlayerSide):boolean
getAttackableMinions():ArrayList<Integer>
<<Java Class>>
HearthTreeNode
com.hearthsim.util.tree
depth_: byte
score_: double
numNodesTried_: int
HearthTreeNode(BoardModel)
HearthTreeNode(BoardModel,double)
HearthTreeNode(BoardModel,double,byte)
HearthTreeNode(BoardModel,double,byte,List<HearthTreeNode>)
HearthTreeNode(BoardModel,double,byte,List<HearthTreeNode>,int)
getNumNodesTried():int
setNumNodesTries(int):void
getDepth():byte
setDepth(byte):void
getScore():double
setScore(double):void
addChild(HearthTreeNode):HearthTreeNode
clearChildren():void
getChildren():List<HearthTreeNode>
setChildren(List<HearthTreeNode>):void
numChildren():int
isLeaf():boolean
findMaxOfFunc(ArtificialPlayer):HearthTreeNode
findMaxOfFuncImpl(ArtificialPlayer):NodeValPair
numLeaves():int
+data_
0..1
~children_
0..*
<<Java Class>>
PlayerModel
com.hearthsim.model
name: String
playerId: int
hero: Hero
deck: Deck
mana: int
maxMana: int
minions: MinionList
spellDamage: byte
hand: IdentityLinkedList<Card>
overload: byte
PlayerModel(int,String,Hero,Deck)
drawFromDeck(int):Card
getMinions():MinionList
setMinions(MinionList):void
getSpellDamage():byte
setSpellDamage(byte):void
addSpellDamage(byte):void
subtractSpellDamage(byte):void
getHand():IdentityLinkedList<Card>
setHand(IdentityLinkedList<Card>):void
getOverload():byte
setOverload(byte):void
placeCardHand(Card):void
placeCardHand(int):void
getPlayerId():int
resetMana():void
-currentPlayer
0..1
-waitingPlayer
0..1
~playerGoingFirst
0..1
~playerGoingSecond
0..1
<<Java Class>>
ArtificialPlayer
com.hearthsim.player.playercontroller
log: Logger
MAX_THINK_TIME: int
myAttackWeight: double
myHealthWeight: double
enemyAttackWeight: double
enemyHealthWeight: double
myHeroHealthWeight: double
enemyHeroHealthWeight: double
tauntWeight: double
manaWeight: double
myNumMinionsWeight: double
enemyNumMinionsWeight: double
spellDamageAddWeight: double
spellDamageMultiplierWeight: double
myDivineShieldWeight: double
enemyDivineShieldWeight: double
myWeaponWeight: double
enemyWeaponWeight: double
myChargeWeight: double
useSparseBoardStateFactory_: boolean
ArtificialPlayer()
buildStandardAI2():ArtificialPlayer
buildStandardAI1():ArtificialPlayer
ArtificialPlayer(Path)
getUseSparseBoardStateFactory():boolean
setUseSparseBoardStateFactory(boolean):void
cardInHandScore(Card):double
heroHealthScore_p0(double,double):double
heroHealthScore_p1(double,double):double
boardScore(BoardModel):double
playTurn(int,BoardModel):BoardModel
playTurn(int,BoardModel,int):BoardModel
~boardModel_
0..1
<<Java Class>>
Game
com.hearthsim
log: Logger
maxTurns_: int
curTurn_: int
Game(PlayerModel,PlayerModel,ArtificialPlayer,ArtificialPlayer)
Game(PlayerModel,PlayerModel,ArtificialPlayer,ArtificialPlayer,boolean)
runGame():GameResult
playTurn(int,GameRecord,ArtificialPlayer):GameResult
checkGameOver(int,GameRecord):GameResult
beginTurn(int,BoardModel):BoardModel
playAITurn(int,BoardModel,ArtificialPlayer):BoardModel
endTurn(BoardModel):BoardModel
~aiForPlayerGoingSecond
0..1
~aiForPlayerGoingFirst
0..1

Diagram Description

The Card class acts as a base class for all Card types in the HearthSim application. Aside from the AI that simulates the game between two players, the Card class is the most important part of the the system. Many classes either inherit or make use of this class. The Hand class contains information about the Cards in a Player's hand, while the Deck class holds information about the number of Cards remaining in a Player's deck. Minion and SpellCard are both subclasses of Card and inherit all of its properties and methods.

The Minion class contains information about generalized Minions, and all Minion subclasses inherit this class' properties. Subclasses of Minion include specific concrete Minion cards (like Archmage Antonidas) and Minion types (like Beast - not depicted in the diagram). The Hero class also inherits from Minion (despite the fact that this does not make logical sense as a Hero is not a minion and can actually be targeted by Minions and Spells). Concrete Heroes (not depicted) inherit from the Hero class. The system's architecture also contains a subclass of Minion called MinionWithEnrage. This class is important to list in the architecture, not because it is useful, but because it should not be implemented in this manner. This problem is discussed below in Part 3.

The BoardModel class is responsible for handling and monitoring the state of the board at all times. This class contains a large amount of methods (many of which are not included in the diagram) that allow it to access information regarding each player's Hand, Deck, and Minions on the board. It obtains player-related information from the PlayerModel class. The Game class contains an instance of the BoardModel, which it passes to the ArtificialPlayer during a turn in the simulated game. HearthTreeNode is a tree that that keeps track of possible BoardModel states. The AI then uses the BoardModel to calculate the best move for the turn based on all possible states provided by HearthTreeNodes through the use of an algorithm that assigns a weight to each possibility.

Many of the classes in the conceptual diagram can be mapped to the actual system's architecture. Much like the original developers' depiction of the system, the conceptual diagram includes a Card superclass that Minion and Spell inherit from. The Deck and Hand classes also consist of Card objects. Furthermore, the conceptual diagram has a BaseMinionCard that acts as a superclass for concrete Minions and Minion types, much like the actual system. An odd thing to note is that the actual system implements the Hero class as a subclass of Minion, when it should actually be an entity on its own that does not inherit from Minion.

Similarly, the AI components of the conceptual system can be mapped to the actual system's classes. The actual system's BoardModel and Artificial classes are equivalent to the conceptual diagram's GameState and TurnMaker classes, respectively. Much like how the actual system's instance of the Game provides an instance of the BoardModel to the ArtificialPlayer, the conceptual model's GameSimulator class provides the TurnMaker a reference to the GameState. Finally, in the conceptual model, the TurnMaker uses the Turn class to keep track of

all the Turns and Turn possibilities, similar to how the HearthTreeNodes track different game states and are used to determine the best course of action for a given turn in the actual system.

One major difference between the actual system and the conceptual model is that the concept of an Action does not exist in the actual system. Instead, the system's Card class implements methods (such as useOn()) which mimics the functionality of an Action. This is also undesirable as having an Action interface and different action types for the different types of entities would reduce the amount of coupling in the system while increasing overall cohesion.

ObjectAid UML Explorer, an Eclipse plugin that allows the visualization of code, was used to obtain and view the architecture of the system. Class diagrams can be easily made by dragging and dropping classes of interest into a diagram. Relationships between classes are drawn automatically. Because many of the classes had a large amount of attributes and methods, only the more useful fields and functions were depicted in the diagrams, while the rest were commented out for the sake of clarity.

Relationships Between Two Classes

HearthSim / src / main / java / com / hearthsim / card / Card.java [Card Class]

HearthSim / src / main / java / com / hearthsim / card / minion / Minion.java [Minion Class]

Source Code from Card Class

```
public Card(String name, byte mana, boolean hasBeenUsed, boolean isInHand)

protected String name_;

protected byte mana_;

protected boolean hasBeenUsed;

protected boolean isInHand_;
```

- The attributes above are encapsulated within the card class, and are accompanied by their appropriate getters and setter methods. All cards played within the Hearthstone game have at least a name, a mana amount, has the card been used, and is the card in your hand. This creates a base class that more specialized cards will branch from allowing for increased cohesion, though consequently an increase in coupling.

Source Code from Minion Class

```
import com.hearthsim.card.Card;
```

- The minion java file imports the card java file so that it may have accessibility to its classes, methods and attributes.

```
public class Minion extends Card
```

- The entire minion class extends the card class, which creates a parent-child inheritance relationship with with the card being the parent and the minion being the child. So all the functionality of the card class like its methods and attributes are usable now by minion. Polymorphism is now able to be implemented to override any methods in card class, that occur in this minion class.

```
public Minion()

super();

mana_ = (byte) implementedCard.mana_;

name_ = implementedCard.name_;

isInHand_ = true;
```

- The super is called first which is the constructor of the card class, then the constructor of the minion class follows. By having access to the attributes of the generic card class, we are able to create control statements and adjust attribute values accordingly.

```
if (hasBeenUsed){return null;}
```

Part 3: Code Smells and System Level Refactorings

The Card class: In this class, there are many functions that represent how a Card object can be used. For example, some functions found within the Card class are useOn(), canbeUsedOn(), endTurn(), startTurn() etc., that indicate the behavior of performing an action with a Card. In this scenario, the Card class acts as a god class that does everything that is related to a card. Instead of making the Card class execute this behavior, a proposed refactoring is to create a new class called CardAction (that implements a generic Action interface) in which we will move the methods in the Card class such as useOn, canBeUsedOn, endTurn, and all other methods that indicate an action that a card can undertake or perform. By doing so, we can increase the cohesion within Card by delegating the actions that a card can be used to the CardAction class.

Similarly, in the same functions mentioned above, Deck and PlayerSide are being passed as well as the board state (a refactoring for the board state class is discussed in detail in Part 4). This creates undesirable coupling between the Card and Deck classes, as well as the Card and PlayerSide classes. The functions in the card class do not need to know about the state of both players' decks or which side is currently playing while performing a desired action. The refactoring above will implement a CardAction class that will be handling the Card's actions. Therefore, the CardAction class should be able to update the Deck of a player and have a notion of the current player. An Observer pattern can be used to implement this refactoring. The Deck and PlayerSide classes can be listeners to the CardAction class. This will decouple the Deck, PlayerSide and Card classes as any change to a player's deck will be pushed to the listening class.

The Minion class: The state of a minion is declared by a boolean. In Hearthstone, a minion can be under the effects of multiple states, and each state represents effects on the minion card. For example, a frozen minion will lose its next attack. Thus the frozen minions are unable to perform any actions until they are unfrozen. In the code, the developers represent each possible state that a minion can be in as a booleans. Due to this implementation, the Minion class is bloated with attributes which constitutes a code smell. Another side effect of this code smell is that it adds to the already long parameter list in the constructor. A suggested refactoring that can be implemented is to create a card state Interface called MinionState, which will be a base class for each unique state that exists in the game (TauntState, or FrozenState, DivineShieldState, etc.) Our motivation by doing so will remove the bloating components and it will substantially increase the independence of the states. This allows for an increase in cohesions however consequentially it increases the coupling, though the surge in cohesion is more ultimately more favorable.

Furthermore, minions come in several different types such as beast, demon, etc. In the code, all these different types are classes that extend the minion class but they all have the exact same

methods and code. A proposed solution will be to store all the minion types in an enum, greatly increase cohesion amongst the Minion class and its different types.

The MinionWithEnrage class: This class represents a minion that is under an enraged state and extends the minion class. The difference between enrage and other statuses (e.g. frozen, taunt, etc.) is that only a small subset of the entire collection of minions have enrage abilities, and these abilities vary from minion to minion. For example, if Minion A has taunt and Minion B has taunt, the effect is the same; other minions will be forced to target either of these minions before anything else. On the other hand, if both Minion A and Minion B have enrage, the effects of the enrage are completely independent of each other and are properties of those specific minions. This kind of implementation can be dealt with using the Strategy pattern because different minions with enrage require different backend logic. Thus, MinionWithEnrage should act as an interface for all minions that have enrage effects, and a new class, ConcreteMinionWithEnrage, should act as a specific minion with a specific enrage implementation.

The Hero class: A hero in Hearthstone is its own entity. In the code, the hero class is currently inheriting from the Minion class. The problem with this coupling is that these two classes should have different functionalities and they should not have an inheritance relationship. A hero should not inherit from a Minion, despite the fact that they can share the same attributes. The simple fact is that a Hero is not a Minion. By creating a BaseEntity abstract class, it will remove the inheritance between the two classes and it will also remove the inappropriate intimacy by completely removing the inheritance between the Hero and Minion classes. The two classes will both extend the abstract Base Entity class, thus providing both classes access to the shared commonalities between them without any coupling involved.

The BoardModel class: The BoardModel class is the board state of the game. The state of the board comprises everything that is currently on the board, such as cards, players, decks, etc. The BoardModel class contains a lot of functions that are inappropriately implemented in the classl. For example, functions such as getSpellDamage(Player), getFatigueDamage(Player), and others pass in a Player as parameter. One refactoring solution is that instead of having the BoardModel pass a player each time, we can delegate the responsibilities to the Player class. The player will hold the information of his/her deck and what card he/she has played. The board should not be calculating the spell damage or knowing that the player is in “fatigue” (how much damage he/she takes when he/she runs out of cards). The player should be the one keeping track of the information on the player’s side and the BoardModel should just be a static view of what is currently played. Hence, by delegating these responsibilities it will increase the cohesion in the class and remove the clearly evident feature envy.

The WeaponCard and SpellCard classes: WeaponCard class is a base class which describes what kind of attribute a weapon can take. In Hearthstone, a weapon card only holds the cost, the number of charges left, the weapon attack, and the effect. In the source code, WeaponCard is implemented as a base class, although this class does not have much responsibilities, a way to refactor this problem is to turn WeaponCard into an abstract class.

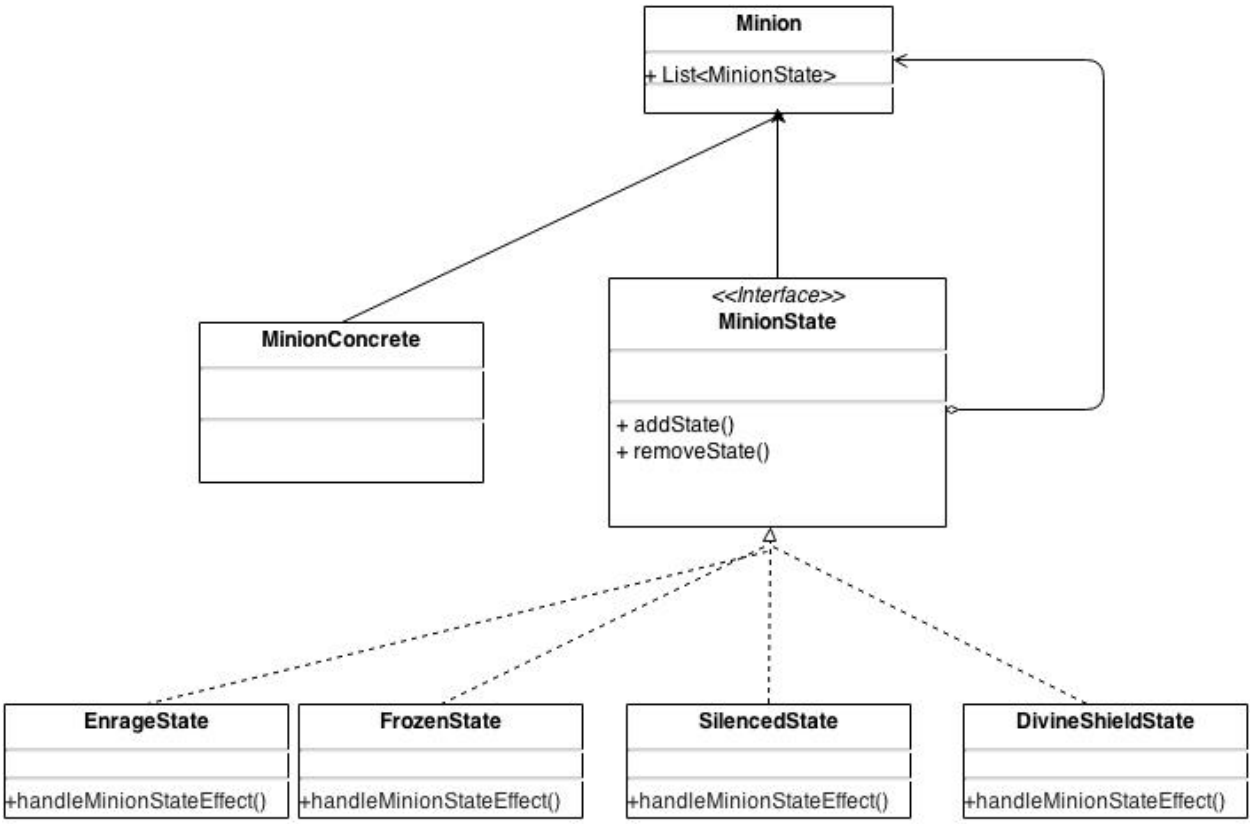
There are many classes, such as ArcaniteReaper (a weapon card), that extend the WeaponCard class in the code. By making the WeaponCard class abstract, we are ensuring that an instance of the base class cannot be made, with its limited functionalities. This also makes the code more adaptable for weapons that could be added to the game in the future.

The SpellCard class is structured identically to the WeaponCard class. Thus, a similar refactoring is suggested here: SpellCard should be transformed into an abstract base class and parent concrete SpellCard child classes.

UML Diagram of refactoring of the Minion Class:

Description:

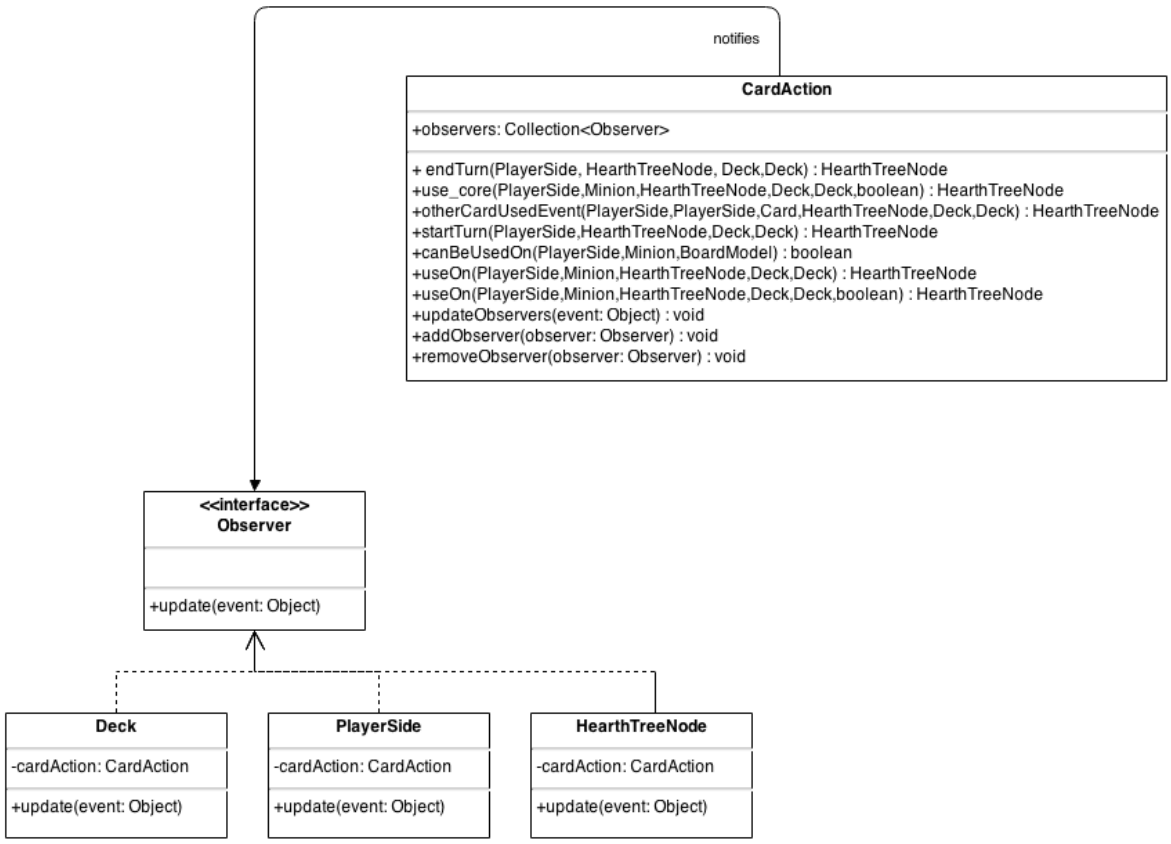
In order to eliminate the large amount of boolean variables found in the original Minion class, the possible states that a Minion can be found in are extracted into separate classes, with each class representing an individual state. An interface is implemented here in order to ensure that a Minion can be under the effect of multiple states.



UML Diagram of refactoring of the Card Class:

Description:

We moved the methods that were not supposed to be in the Card Class and we created a new class called CardAction which will be the one handling the actions of a card. Also, we use HearthTreeNode instead of the BoardModel class as our observer. BoardModel is only a static view of the board, and HearthTreeNode is the one that will update the BoardModel class, which is why we make the HearthTreeNode the observable.



Part 4: Specific Refactorings

To be implemented in Milestone 4

Refactoring 1: Create BaseEntity abstract class

An abstract base entity class is necessary in this case to decouple unrelated classes and to provide commonalities to be inherited by child entities in the this application.

In the current HearthSim project, two game entities (Hero and Minion) are coupled together inappropriately: the Hero class extends the Minion class. Conceptually, this is entirely incorrect as Heroes are not Minions and should not inherit from the Minion class. In a Hearthstone simulation, a player is represented by a hero and minions are creatures (represented by cards) on the game board that fight for their hero. Thus, Hero should not extend the Minion class.

The motivation behind Hero extending Minion is due to the fact that the two classes have similar attributes and functionalities. During a game, Heroes can use various means to attack and receive damage from other targets, exactly as a minion does. In fact, the Hero constructor explicitly calls the constructor of the super class, Minion, to set a lot of shared attributes for a Hero instance. Thus, this example is a good candidate for the implementation of an abstract base class that has the shared attributes between Hero and Minion.

The proposed refactoring will use the following procedure:

1. Create an abstract BaseEntity class with commonalities between all game entities, especially that of Hero and Minion.
2. Remove the inheritance between Minion and Hero.
3. Hero and Minion will now both extend the BaseEntity class and inherit the common class members from BaseEntity.

Refactoring 2: Create MinionState class interface and relevant concrete classes to replace current boolean system for capturing states of minion cards

In Hearthstone, minion cards in play on the game board can enter into different minion states as a result of actions that occur within the game. Some examples of these card states are “frozen”, “silenced”, and “stealthed”. These states change the power and role of the affected. For example, if a card is silenced, its abilities such as taunt (defensive strategy) are canceled as well as its buffs (increases in power) removed.

Currently, the minion state system within the Minion class is implemented using boolean values. Booleans named “silenced” and “frozen” are used as flags to capture whether or not a card has entered into each state. This is a code smell since these boolean minion states are used as

parameters in methods to control the flow of code execution. For example, the boolean “silenced” is use in a function called silenced() which has two control flows, one default and one that occurs is the boolean “silenced” is false. If “silenced” is false, then the board state is updated accordingly, otherwise, the default control flow of when “silenced” is true occurs -- this blocks the Minion from using taunt as well as debuffs it.

These “magic value” booleans can be vulnerable to bugs because these booleans can be easily set and reset in the code with heavy consequences. If “silenced’ is set to the wrong truth value by a piece of code in error, debugging the code for this source of error can be proven to be difficult. The use of many booleans is also a maintenance headache, as the boolean can be set and reset in multiple places and if changes are necessary, each of those booleans must be found and changed. If one of these booleans are missed during the course of changing the code, an annoying debugging problem arises.

These boolean parameters or flag arguments also lower method cohesion. In the above example of the usage of “silenced” in the functions silenced(), the control flow is split into two consequential actions, one where “silenced” is true and the other when it is false. This method is now accomplishing two different things and can be easily separated into two functions, one to handle if “silenced” is true and the other to handle if “silenced’ is false.

Our proposed refactoring is to encapsulate the knowledge of the minion state within an object that will be responsible for carrying out the effects and consequences of being in a state. A minion state object will be attached to each Minion card when the card enters into such a state. These states are also removed from the Minion card when necessary. In this way, the states are stored in one place for each Minion card and the states in which the card is in is easily maintainable. Furthermore, the minion state instances will directly handle the changes that arise when a minion enters into a state.

The proposed refactoring will use the following procedure:

1. Create MinionState interface with commonalities between different states
2. Create concrete MinionState classes that implement the MinionState interface
    2.1. These classes may be named SilencedState and FrozenState
    2.2. Each concrete MinionState properly handles the impact of entering such a state by a Minion
3. Update the Minion class with a new member variable of a data structure containing MinionState objects
4. Whenever a Minion enters into a new state, an appropriate MinionState object is added to the MinionState collection
    4.1. This attachment of a MinionState to a Minion also triggers the MinionState to accomplish the appropriate changes within Minion

Relevant source code declarations for Refactoring 2:

*Note that irrelevant parameters and instantiations are not shown

Constructor for Minion

```
   public Minion( String name, …, boolean frozen, boolean silenced, boolean stealthed, …
   ) {
        …,
        frozen_ = frozen;
        silenced_ = silenced;
        baseHealth_ = baseHealth;
        …
  }
```

Method called when a Minion enters into a silenced state

*Note that this method is used as an example for the other similarly implemented minion states

```
    public void silenced( PlayerSide thisPlayerSide, BoardModel boardState ) throws
       HSInvalidPlayerIndexException {
          if (!silenced_) {
               boardState.setSpellDamage(PlayerSide.CURRENT_PLAYER,
               (byte)(boardState.getSpellDamage(PlayerSide.CURRENT_PLAYER) -
spellDamage_));
           }
               divineShield_ = false;
               taunt_ = false;
               charge_ = false;
               frozen_ = false;
               windFury_ = false;
               silenced_ = true;
               deathrattleAction_ = null;
               stealthed_ = false;
               heroTargetable_ = true;

               //Reset the attack and health to base
               this.attack_ = this.baseAttack_;
               if (this.maxHealth_ > this.baseHealth_) {
                      this.maxHealth_ = this.baseHealth_;
                      if (this.health_ > this.maxHealth_)
                             this.health_ = this.maxHealth_;
               }
        }
```

Refactoring 3: Create CardAction delegate class (that implements an Action interface) and update board state objects using an Observer pattern

Since Hearthstone is a card-based game, a lot of the gameplay is based on the actions that playing a card will have on the state of the game board. Thus, the HearthSim application needs to be able to use the properties of a Card object to create effects on the game state.

Currently, card objects implement methods that take in a representation of the game board state, apply the actions associated with using the card in the game and then returning the board state representation. More specifically, the Card class has methods (called useOn(), for example) that performs the action of using the card on a given target. These functions take in a board state object, manipulate the object appropriately (i.e. subtract the cost of using the card from a player, removing the used card from a player's hand, create an instance of the Minion represented by the card, etc.), and then return the board state object.

The current implementation of card effects on the game board is problematic in a couple of ways. First of all, the passing of the board state object into methods of the Card class creates undesirably high coupling between the Card class and the class representing a board state (HearthTreeNode). Card objects should not have knowledge of board states -- they are just representations of a Card in the system. If for some reason, the board state is not initiated, the coupling between the Card and board state classes will create errors within the Card class. Thus, the Card class should be able to stand alone.

Also along the same conceptual lines, since Card objects are models of playable cards in a game simulation, the class should not have methods that perform actions that manipulate board states. The model itself represents the card and it should be used by other classes that perform changes in the game.

The goal in this refactoring is to decouple the Card and board state classes, thus preserving the concept that Card is purely a model of a Hearthstone card within the game. Our proposed refactoring will be create a delegate class called CardAction which performs the appropriate manipulations of using a card on the game board. The CardAction class will have functionalities similar to the problematic methods current found in the Card class. These new CardAction methods will take in a Card object and ultimately be able to update the game board state.

To create extensible code, the CardAction class will implement a general Action interface, in case other entities in the system will also act upon the board states (and other classes).

The second part of our proposed refactoring involves implementing an Observer pattern to help loosen the decoupling between the CardAction class and the board state class. Again, this works to preserve the integrity of the Card model. This will eliminate the need to have our CardAction methods take in and return board state objects, which relieves the strong coupling between the classes. In our case, the board state class will be the "observer" that subscribes to the notifications of our "subject", the CardAction class. Any relevant changes to the board state as a result of using a card via the CardAction class will be pushed to the board state.

Not only does this lessen the coupling between CardAction and the board state class, this pattern is also extendable to any other aspects of the game that may want to be updated of a card being played (the Deck of cards, or a player's Hand of cards come to mind). Essentially, the

usage of the Observer pattern here allows the CardAction class to announce its changes that result from using a card without being coupled to the class. For realistic purposes, our implemented refactoring in this project will only cover the Observer pattern being applied to the CardAction and board state classes.

The proposed refactoring will use the following procedure:

1. Create a delegate class called CardAction that is instantiated with an associated Card object
2. Create any necessary methods that handle the conversion of the Card's attributes into events that will impact the board state; return these events
3. CardAction will also be implemented as the "subject" of an Observer pattern
    a. CardAction will have a dynamically-sized collection of Observer objects in which the Observers will register themselves into
    b. CardAction will have an updateObservers() function that takes in the updated event caused by the usage of its Card object
        i. This function will iterate through the CardAction's collection of registered Observers, calling the update of the board state class and passing in the appropriate events
4. An Observer interface will be created with an update() function that handles the events pushed by the CardAction subject class
5. The board state class (HearthTreeNode) will be implemented as an "observer" of an Observer pattern and will implement the Observer interface
    a. The board state class will properly reflect changes in the game's board state as a result of the implemented update() function which receives event updates from the CardAction subject