Cohesion and Coupling in our design

This project started out with good cohesion and coupling, but by the end there were several cases where our implementation had bad cohesion and coupling.

Bad cases of Coupling and Cohesion

- RuleManager.tryMove(), .tryAct(), etc. - When the Controller finds an input it thinks is an action command, it passes related information to the RuleManager by calling a Rulemanaget.tryAction() method. Some of the implementations of these methods include behavior that should have been in the Action classes. This case is an example of poor cohesion.
- Messages to the Controller - When something happens (i.e. and action is successful or a new day starts) a message is displayed to the console. With the exception of some error messages, the messages are always printed from Controller, which is good. However, the construction of the messages is inconsistent. Sometimes the controller constructs the messages on its own. Other times it asks the RuleManager for a message. A more cohesive design would have clearly separated RuleManager messages and Controller messages.
- Controller has RuleManager responsibilities - Methods in Controller like dayUpdate() and turnUpdate() would have made more sense in the RuleManager class. However, most of the actual logic of updating is left to RuleManager, so this case is still partially cohesive.
- Coupling between Player and Area - Originally we planned on having a Location class to store Player and Area locations. This way Player and Area would not have to depend on each other. Later, we decided to remove the Player's location and store its current Area instead. While this eliminated an unnecessary class, it also increased coupling between Player and Area. However, the Player never uses Area except to get and set its location, so the coupling is still acceptable.
- RuleManager.getValidActions() - This method gets actions the player can generally do without checking the specifics. For example, it can check whether roles exist, but it does not check whether any roles of the correct rank exist. This method should be split up and added to the Action class to improve cohesion.

Good examples of Coupling and Cohesion

- Actions - Including separate classes for actions increases cohesion and reduces coupling because Action behaviors are contained within the class and the RuleManager only needs to call Action.isValid() and Action.execute() to perform an action.
- Controller only sees RuleManager - Whenever the Controller parses a command or updates, it calls the related method in RuleManager. The Controller never sees any other classes directly. This is an example of reduced coupling.
- Board Objects - The board is structured like a tree where lower components do not know about higher components. For example, Areas do not know about the board, Cards do not know about Areas, and Roles do not know about Cards or Areas. This is an example

of reduced coupling. The classes are also cohesive because they handle their own behaviors. Each class in the BoardModel is also decoupled from Player. In the future, we plan to decouple BoardModel from Player as well.
- Player as a data-storage class - Since all actions belong to their own class, there is not much behavior needed in the Player class. We decided to limit the Player's behavior to getting and setting data about itself. This includes incrementing rehearsals and calculating the score. This makes the Player class cohesive and reduces coupling.

Rationale for our design

While we were designing our implementation of Deadwood we focused on “is a” and “has a” relationships.

“Is a” relationships
There were two cases where “is a” relationships made sense.
- Area - Set, Trailer, and Casting Office are all types of areas, so it made sense to have them inherit from the Area class. We decided on inheritance because common behavior between areas exists.
- Action - Upgrade, Rehearse, Act, TakeRole, and Move are all types of actions, so it made sense to have them implement the Action interface. We decided on an interface because although actions should have a validity check and execution, each type of action has a different implementation.

“Has a” relationships
Although “has a” relationships can be used to describe any type of association, we used it to decide when to use composition and aggregation.
- Player - Player has a role and a location(area) but neither of those need to have a player.
- Set - Set has a Card and Roles and a Card has Roles, but Roles do not have Sets or Cards, and Cards do not need to know about the set they are on.
- BoardModel - originally we wanted the board to contain the players, but we decided to only “compose” the board of Areas, Cards, and Roles.
- RuleManager - the manager has all the players, all the actions, and the board.
- Controller - The controller has the manager and the view. Currently the view is not being used.

Regular Association
For relationships that could not be described as “has a” or “is a”, we tried to minimize association. For example, the Controller always has to access data through the manager.

A note on Actions

Usually components with names that are verbs are implemented as methods in a class, but with the actions we decided to make an exceptions. Each action is unique, so including all actions within the Player class would have made Player bloated. Also, having instances of Actions reduced coupling because the RuleManager can handle actions without going through the Player.