We tried to keep each class highly cohesive and tried to reduce coupling when possible. Overall our model is very cohesive in nature but is more highly coupled as it relies on smaller classes to reduce complexity in each individual class. A good example would be how instead of storing 4 individual integers to represent the area of a class, we created our own Area class to simplify all the classes that utilize it. The tradeoff being that many classes are dependent on the Area class to function correctly. As far as coupling we tried to use internal class functions when possible, or generalize it into a “manager” class that could perform functions that encapsulate all members of a type, such as Deck or Board.

Cohesion

Functional:

- Area - only stores area parameters
- Bank - only deals with changing player currency amounts
- Take - only deals with take parameters
- Upgrade - only stores information on upgrade levels
- Part - only stores information on player parts/roles
- Dice - only has one method that returns dice rolls
- BoardLocation - only stores information about the location on the board it represents
- Card - only stores information on the given card
- Board - only stores BoardLocation objects and methods for fetching
- Deadwood - only has initialization and flow - no display or logic
- Deck - more borderline than the rest, stores all cards and facilitates shuffling/dealing cards, but still feel that functional is appropriate as otherwise the class would just be an arraylist of cards
- PlayerManager - only performs actions that would normally be taken by the physical player in deadwood (move, act, rehearse, etc.) (manipulating player)
- Model - only stores model information, just getters and setters
- Player - only stores player data
- CLIView - displays user information, does some output cleaning but no game logic
- Controller - only handles user input and then using that input to push data to the view

Logical

- Admin - Admin performs tasks that would normally be handled by the physical administrator during the game

Temporal

- XMLParser - reads data from XML and creates objects in one class/method - may have been better to have a buildFromXML option in Deck, UpgradeManager and Board

Coupling

Data

- Area - only setters/getters
- Bank - no data stored, all parameters are passed
- Take - only setters/getters
- Upgrade - only setters/getters

- Part - only setters/getters
- Dice - only one parameter, clearly labeled/important to function
- BoardLocation - only setters/getters/operations on internal data
- Card - only setters/getters
- Board - only stores BoardLocation objects and methods for fetching
- Deadwood - isn't inherently coupled to anything, main method
- Deck - only dependencies are the datatypes that it manages
- PlayerManager - all parameters passed/only operations on internal data
- Player - only stores player data

Procedural

- CLIView - Accepts user input/displays user information, does some i/o cleaning but no game logic. Could have created a separate class but seemed more straightforward to do some of those functions in the view class as the input will not be the same for GUIView

Stamp

- Admin - subclass of model
- Model - linked with subclass admin - These two are linked, less so because they need to be separate, more so to distinguish between administrative functions and model functions. Almost exclusively works with model data, it made sense to make admin a subclass.

External

- XMLParser - reads data from XML - only external facing system, isolated to one class. Necessary to get data to run program.

Design Statement Paragraph:

We chose to use an MVC architecture as we knew that we would eventually be implementing a GUI and having a detachable view would make that process easier. When it comes to building the model, we chose to make a larger number of smaller classes that specialize in a type of data to make our code more readable and reduce complexity in each individual class. Even with this philosophy, some classes had upwards of 5 attributes. To resolve this we made prolific use of the builder pattern, specifically with fluent setters for readability and flexibility. For the controller we tried to keep it purely input handling and passing data to the view, primarily using switch statements for readability. Obviously the view just uses pretty rudimentary command line printing to display, nothing all that special from a design perspective.