Read the README.md file for a better experience.

Part A

1. Unit Testing

1:

this does not fulfill the requirments, they requirments state: "There are two players, either one local and one remote, or two remote players both connecting to a server" but we can see in Server.java:57 that it allways adds a local player.
From my knowlage this is not testable in JUnit, since we cant mock the sockets (but is possible to test in som VMs). We could however write a test that will fail since if we start it with withBot == True, it should fail since then we have 1 local player and bot, which was not an acceptable iteration of the requirmnet.

2:

a: from my readthrough of the json, and code it apears to load all the cards into the vectors. This is hard to test by hand but would be a good fit for unit tests. JUnit can test this by checking the quantity of each of its vecktors.
b: This apears to be working, and can be tested by JUnit, by looking at the regionDice (Server.java:139) after the init fucntion, and looking at what card is stored at each position.
c-f: apears to work and can be tested by JUnit in similar ways.

3:

It technically fulfills the requirments for this rule, since it replenishes the hand by 3, but according to the actual game rules it does so incorrectly, since the game rules state
"draws the top 3 cards from one of the 4 draw stacks", and we can se in Server.java:1248-1266 that it lets the user pick one card each loop from a chosen stack, instad of picking one draw stack and then drawing 3.
This can be tested JUnit by comparing the player hands before and after they have called replenish the first time (it is called in the start function so it would be very easy to test by setting up the game and calling start, then looking at the amount of elements for example and se if they are a Card).

4:

The game requirments state: "(random player starts the game)"
but the actual games rules state: "Each player rolls the dice. The high roller is the starting player"

It follows the requirment correctly, by doing a random and picking the player from it. Though this cant be tested in JUnit since we cant test that it is random.

a:

It seems to be doing this correctly, and it can be tested by checking what function we call first, ????

i:

I am reading the requirment as it a a 4 side die 0 to and 4 (0..3), and if thta is the case it does it incorrectly since it "rolls" a d6, it does that correctly, but I blame that on poorly written requirments.
We can do this by mocking the state and testing iterations of it, but it is poorly written for testing since we cant inject the rnadom value.
ii:

5:

This is impllemented, we can test it in JUnit, by giving the active player 7 victory points
and then calling the fucntion.but is hard due to the maount of private varables that needs to be initiated correctly.

(I ran out of text space on my A4 page :/)

2a:

for some shortcoming in relation to SOLID and booch metrics.
for the Single Responsibility Principle we can see that run() in Server.java:217 contains the whole game loop,
it has devided some stuff up in other functions but it does to many things even if we allowed it to have the whole game phase loop in a while true statemnt.
we can see in Server.java:55 in the start function that it is not easy to modyfiy diffrent ways to play the game without changing the code failing the Open-Closed Principle.
and for noting some booch metrics, Primitiveness is not fufilled since we have some monter functions that dont serve a single perpuse but multiple. Sufficiency is not good since the current code does done support implementing all features, for examples having a victory_points_to_win be different depending on which expantion you are playing.

My Implementation

saken

varför har jag saken?
SOLID?
BOOCH?
quality attributes?
Modifiability, Extensibility, Testability?

Program structure

The game is based around an event queue, where all things that change the game_state
has to happen during events
The current code is not the complete vertion and is more a demo for the design architecture.

Events

Evetns are used to to modify game_state and is the only thing that is allowed to change it (according to my design). The Events follow a strategy pattern where all events is a separete strategy.
It was implemented this way in order to have a easily modifyable and testable design for the game.
The structure allows the events (concrete strategies) to focus on one thing following the Single responsibility pattern.
The strategy pattern is also a good option for when we need flexibility and extesnibility in the program, following th open closed principle it is easy to add new strategies thet modify the game in their way, while being closed to modification.
to handle the queue we define an event_queue: List[Event], where the event_queue is dependant on the abstract base class instead of the concrete strategies, following the Dependency Inversion Principle.

These events are very easy to create tests for since we can easly enque events into the eventque
then we can step and the in a test compare the game_state we have now with the desired game state.

Modifiers

These modifiers build upon the events like wrappers following the decorater pattern.
This happens in the game by iterating over the wrappers in the modifierengine when we enque a event into the event queue.
This was nessesary since we have ceratin things in the game that happen differently depending on
if for example a brick facotry is next to a hills card and the production die is equal to the number on the hills card, Since if there is just a hills card there it will add one

brick to the card (if there is enough space) but if there was a brick factory next to the card it would instead add two bricks.

These wrappers, follow the single responsibility patter, since their only job is to wrap themselfs onto suitable events. though you could argue that the current vertion of the examples in midifiers_examples could be split further down, for example having a function that checks if it is aplicable and one that wraps and returns. The follow the Open Closed Principle since the wrappers are closed for modification but you could add more wrappers, meanining it is open to for extention.
it also follows Interface Segregation Principle but it is not hard seeing how the Modifier abstract base class (ABC) only has one fucntion.
like the event_queue, the modfierengine depends on the ABC and not the concrete wrappers themselfs.

These wrappers also open up to extensibility and modifyability in future expantions since the events themselfs dont know that they are getting wrapped, and we can the in later exapntions add cards that interact with the base games logic without modfying it. Similarly to the events we can add the modifiers to the modifierengine, and then pass in some events step()
and then compare states to see if they wrapped correctly.

Phases

My game has implemented a sate pattern called phases that is called phases, mimicing the name from the game. These phases break some desing principles since they modify the game which also owns the phase (in the phase variable), but this is a tradeoff i decided to make since it keeps the Events more clean allows them to be used freely, instead of forcing them to either have a reference to the next event that should be called or liek in the given code having a big while loop that calls the fucntions.

It is is a good way to keep the srp and ocp since the phaes only do "one" thing and then it is easy to write new ones. Though with the current implemetations of the code I would need to rewrite all of them since they are usning hard coded values, but that is an easy fix.

Players

Player is a class that hold some datate about the player and the its strategy

Player Strategies

Each player implements player strategy, another strategy pattern which would allow "players" to have unique way to interact with the game.

Mediator

Communication betwen the game and players happen through the mediator (follwing the mediator pattern) Allowing the game to comunicate to the player through the mediaor. This was added so that all channels of comunication for with a player can happen with through the mediator, instead of having high coupeling between a lot of events, modifiers and the comunication with the player.

<!--
right now the only player strategy that is implemented is the human input strategy where it asks the user for terminal inputs (the networking for that is not complete). But there is also an example of how a -->

Observer

Right now there is an observer pattern used,
it was meant for debugging during development and manual tests.
The future use for it was mean for having a way for players to view back the actions that had happned in the game sort of like the history view in hearthstone.

It is not needed for the game and is not a rquirment.

Booch metrics

I would say that my design has a okay level of coupeling, Events and ModifierEngine depend on ABCs
rather than concrete implementations, leading to lower coupeling.
But there is a circular ownership between game and phase that creates runtime coupeling.
Modifiers wrap events at enque time, which is good for separation but increases coupeling.

The architecture supports all game requirments but the demo code lacks ceratin features that would make the whole design sufficient, for example phases being hard coded. A way to fix this would be to give the phases their next phase allowing for more modualrity and open up more for extentions and testing.

The game is lacking in completness while the architecture support all game requirmnets the current code is missing ceratin features, for example; right now it is hard coded to two local players,
helper functions are missing and so on.

The design has good primitivness, with small events and modifiers; composition over monoliths.
It is not perfect hower since for example as prevoiusly mentioned; modifier examples combine applicability checks and wrapping.

Software Quality Attributes

Availability; single process, in memory game state and event que makes servise easy to run but fragile. Though you only need a very minial amount of the architerture for the game to "work"
If you could for example simulate the while game, without using the player strategies and phases by using injecting states and events into the game.

The design has high modifyability. Strategy patterns for Events and PlayerStrategies, decorater pattern for modifiers, and dependancy on ABCs. Though it is lacking in some areas as prevously mentioned phases are hard coded at the moment.

The design has okay Integrebility, the mediator integration for the player allows for extentions UI and networking. Though there is no defined API for comunicationg with the program, and the game phase circular ownership makes it harder for the program to function cohesively.

The game is designed around being easy to test, events are single pourpose and determenistic, and the game can easily be tested in ways prevously mentioned. One way to make it more testable for the random events is to switch to using determenistic RNG (by using seeds).
Allowing the the random parts of the program to be tested.

The performance is sufficiant for the game, and since it is not a requirment it was not a focus in development, for example it is in python, using busy wait for user responses we iterate over the modifier lsit every time we enque, single threaded design. But even with all this since it is a slow turn based game it is performant enough for almost all senarios.

what is currently not implemented and how i would implement it

My code was mainly used as a skeleton for the architecture, these are some concrete implementations that were used to test the functionality of the system, hower there remains some code quality isses like concrete implementeations and interfaces in the same folder and so on.
This section will feature some insight into how the system could be tested.

(for the req. 1-7)

1: I currently only have support for local players but a way to achive this would be to implement a remote player strategy, since the game loop is agnostic to how the decitions are made, This approach keeps the game logic agnostic to how decisions are made, supporting both local and remote players, and is easy to test and extend. Then create a server around the whole game that hosts the games, then a user can choose which interation of players they want and go with that, it would also allow players to select AI opponents to play against.

2a: Is mostly implemented, with the current factory and json
you can write tests for it JUnit-style like this:

```py
assert len(game.game_state.draw_stacks['regions']) == 24
assert len(game.game_state.draw_stacks['settlements']) == 9
assert len(game.game_state.draw_stacks['cities']) == 7
assert len(game.game_state.draw_stacks['roads']) == 9
```

2b: is not fully implemeted, it does not enfore the hard layout depicted in the pictures, but that is something you could easily change in the json file

2c: Not really, since now they are all added into one draw stack but it is not "shuffled", but that could easily get added by using random order on the region card draw stack.

2d: Yes this is fulfilled, the cardFactory creates these stacks and since they dont need to be shuffled the draw stacks are in order.

2e: There is missing json for these cards but it would be implemednted similarly by adding all the cards into one temprary stack then shuffeling them, then adding them into the 4 draw stacks.

This test is only if they are the right size,

```py
# After implementing basic card logic
assert sum(len(game.game_state.draw_stacks[stack]) for stack in ['basic1', 'basic2', 'basic3', 'basic4']) == 36
# Optionally, check that each stack has 9 cards
for stack in ['basic1', 'basic2', 'basic3', 'basic4']:
    assert len(game.game_state.draw_stacks[stack]) == 9
```

2f: same as in 2e, but the yule card would be popped from the event draw stack then inserted into the its proper position.

```py
event_stack = game.game_state.draw_stacks['event']
assert event_stack[-4].template_id == 'yule'
```

3: I would add an initialphase that determines which player goes first (rolling one d6 and let the higher go first, or reroll on draw). Then enque two draw_three_cards, with the first player taking handeling the first event.

4a: the event_production_die Event is partially implemented, but only the production part of it. So I would need to create all the events and then extend the logic of the event_production_die for it to work properly.

4b: This is not implemented, and can currently not be tested.
I would need to create all the evntes then create a
choose action event in a phase, that enques the selected event.

4c: This is partially implemented, though I would need to create a replenish_hand event that work with the existing draw functionality for the cards, a test could look something liek this:

```py
# After replenish, player's hand should have more cards
before = len(player.card_hand)
game.replenish_hand(player)
after = len(player.card_hand)
assert after > before
```

4d: This is again creating a custom event, adding it to a phase then

5: for how the game is currently it is partially fufileld. we could create and event in a post_turn phase, called assert_game_completion
that can in a hard coded base case be jsut checking if the player has enough vicory points, but in my design that would be a get function since it depends on certain cards and points liek if a player has trade advantage.

```py
class CheckVictoryEvent(Event):
    def __init__(self, victory_condition_func):
        self.victory_condition_func = victory_condition_func
```

```
    def execute(self, game, game_state):
        if self.victory_condition_func(game_state):
            # Announce winner, end game, etc.
            pass
```

```py
def custom_victory_condition(gs):
    # logic here
    return True

game.enqueue_event(CheckVictoryEvent(custom_victory_condition))
```

6: The card system is data driven, though you would probably have to make some few tweaks to the card factory. then you would need to write the json for the expantions, it would be best to keep it in separate files ince the expantions use certain cards from one set but not others.
So it would be best to have separete json files for each expantions.
This also opens up for users to create their own game setup, by allowing users to create their own card setups.

The event system is built around extensibility and testability,
if you need more game logic you simple write more events without changing the existing code. Same applies to the Modifiers.

With the change prevoisly discussed about the new way to set up phases, it would be easy to set up new phase layaouts for future expantions.

The current way victory conditions is handled is not the best, it would be better to create an event that takes a funciton argument so that you can create completly custom win conditions, one simple is for the xpantions with hight victory point threshhold you simple create a new fucntion with the new threshhold. But you can also do completly new ones, for example the first player to end their turn with 10 gold. This is possible since you can simply pass the game_state to the funciton.

The player strategies and mediator allows the creation of new player types, like AI and differnet kinds of remote play.