Clue Game – Players

Test Driven Development

(note, if you see where this document can be improved, let me know).

Have you played Clue yet?

- We're getting into the "meat" of the game… be sure you've played the game (there's a sample from a prior semester in Files/Clue in Canvas). May be a few small changes, but it will definitely give you an idea. Or find someone who has the game and play it ☺.
- You may also want to re-read the specs in Clue Rules. It's not too long but it has a lot of important information. And its short compared to specs for a real system! You can find it in Files/Clue in Canvas.

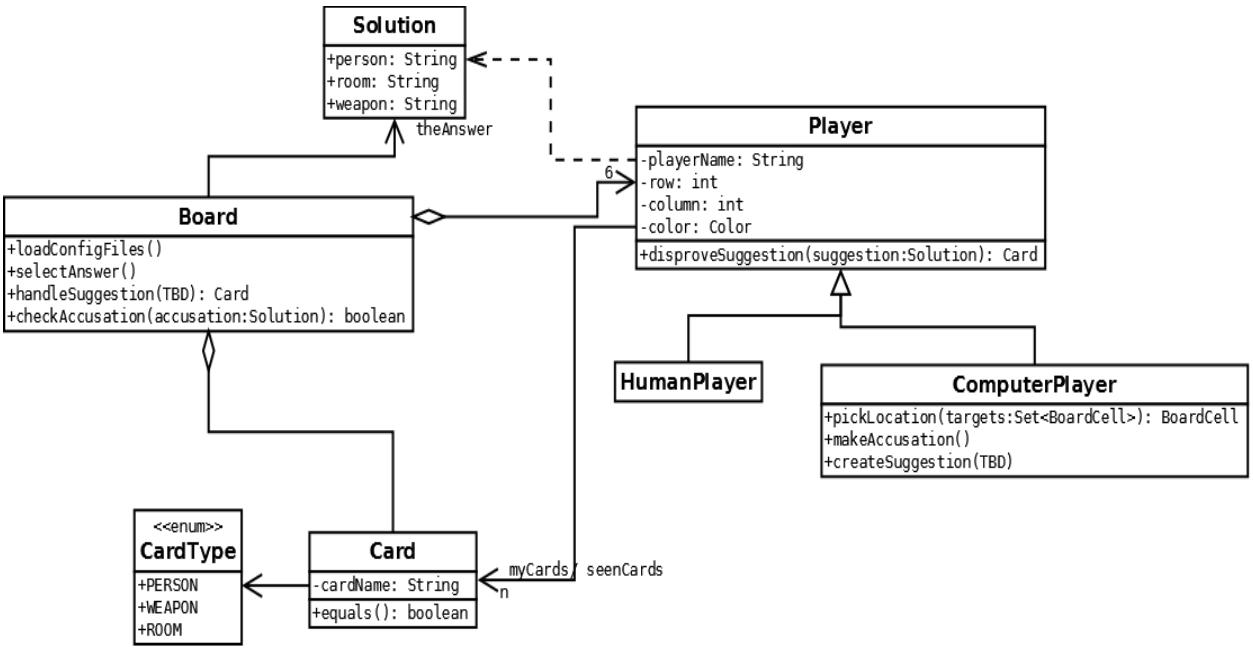
Failing then Passing Tests

- Failing tests should be written first
- BUT, creating a lot of failing tests, then implementing them all at once, is not really good practice
- **Typical practice:** write a failing test, make it pass, write the next test, make it pass, etc.
- SO, you will only be submitting passing tests. BUT, we will use git logs to show your failing/passing process. Commit frequently!

TDD (Test Driven Design) is a design process

- Think about what data/methods you need
  - *which class implements the behavior*
  - *what is a meaningful name for the method*
  - *what data type(s) would be effective*
  - *some of these decisions specified, some you will decide*
- Think about what to test, how to set up conditions
  - *test "true" conditions*
  - *test "false" conditions*
  - *test behavior deterministically*
  - *test random behavior*
- Some design decisions I've made for you…
- We'll have a Player class with two children, ComputerPlayer and HumanPlayer. Must have a list of cards. Must have a name. Must have a location.
- We'll have a Card class to hold cards. We'll use an enumerated type for card type. Cards also need a name.

- UML has more suggestions. These classes will get additional fields/methods when we do the GUI.



Configuration Files

- Rooms (done)
- Board (done)
- Players – to do, **you determine format**
- Weapons – to do, **you determine format**

Game Setup Tests - Things that happen before game play begins

Initializing Data Structures

Game setup includes:

- Loading the people
- Load/create the complete deck of cards
- Dealing the cards

5-minute Quick Exercise with your partner:

- What are the requirements for a correct deal?
- How can you test?

Test loading people

You've dealt with config files before.

How to test loading people from file

I check at least one human and computer, as well as 1st, 3rd and last player (the first and last in my file) to ensure that they have the correct:

- Name
- Color
- Human/Computer (note, we may opt later to assign this in game instead of config).
- Starting location
- Remember that **you** determine the format for the file

You will need to read the player's color from file. But in the code, you'll need a Java Color object. Code to convert from a string to a Color, from:

- http://stackoverflow.com/questions/2854043/converting-a-string-to-color-in-java

```
import java.awt.Color;  // BE SURE TO USE THIS IMPORT
                                // not the one Eclipse suggests
import java.lang.reflect.Field;

// Be sure to trim the color, we don't want spaces around the name
public Color convertColor(String strColor) {
    Color color;
    try {
        // We can use reflection to convert the string to a color
        Field field = Class.forName("java.awt.Color").getField(strColor.trim());
        color = (Color)field.get(null);
  } catch (Exception e) {
        color = null; // Not defined
  }
  return color;
}
```

Test loading deck of cards

People, Weapons, Rooms

My strategy:

- Ensure the deck contains the correct total number of cards
- Ensure the deck contains the correct number of each type of card (room/weapon/person)
- I choose one room, one weapon, and one person, and ensure the deck contains each of those (to test loading the names).

Test dealing cards

- How will you know if the deal is correct?
    - All cards should be dealt
    - All players should have *roughly* the same number of cards
        - Deck size could easily change (e.g., we could add more weapons). So try to avoid hard-coding # of cards/player.
    - The same card should not be given to >1 player
- THINK: do computer processes need to exactly mimic human processes? Not really. So you *may* decide to remove cards from the deck as they are dealt. But is not strictly needed (and may not be the most efficient solution).

Game actions

Do players behave as desired?

- Test selecting a target location (ComputerPlayer)
- Test checking an accusation (Board)
- Test creating a suggestion (ComputerPlayer)
- Test disproving a suggestion (Player)
- Test asking players in order to disprove suggestion (Board)

Reminder:

- A ***suggestion*** is a guess that's used in the process of elimination… if I'm in the Ballroom and I say "Mrs. Peacock with the Knife" then other players will try to "disprove" the suggestion by showing one card of those 3 cards (if someone has Mrs. Peacock, she clearly is not the murderer).
- An ***accusation*** is done by a player who thinks they know the answer. In this case, the Clue Game says yes or no (but if "no" the player can continue)

Test selecting a target – Computer Player

- Should only choose a valid target (calculating targets already tested –yay)
- If no rooms in list, choose a target randomly
- If the list includes a room, it should be chosen – *unless* player was just in that room. In that case, the room has the **same chance of being chosen** as any other target (i.e., player might go into room, or might not).

- If your board is designed so players can bounce between 2 rooms, you'll need to handle appropriately.

This is not a trivial test!

- Issue: how to ensure behavior happens consistently (e.g., ensure if a room is in the list, it will be chosen)
  - Approach: create a target list that includes a room. Run the method multiple times, ensure that ***only*** the desired behavior (i.e., picking the room) is seen.
- Issue: how to test for random behavior (e.g., ensure target is chosen randomly).
  - Approach: create a list of targets with no rooms, run the method multiple times, ensure that ***all*** desired behaviors are seen. See code next slide.

Example code for random selection

```
@Test
public void testTargetRandomSelection() {
    ComputerPlayer player = new ComputerPlayer();
    // Pick a location with no rooms in target, just three targets
    board.calcTargets(14, 0, 2);
    boolean loc_12_0 = false;
    boolean loc_14_2 = false;
    boolean loc_15_1 = false;

    // Run the test a large number of times
    for (int i=0; i<100; i++) {
        BoardCell selected = player.pickLocation(board.getTargets());
        if (selected == board.getCellAt(12, 0))
            loc_12_0 = true;
        else if (selected == board.getCellAt(14, 2))
            loc_14_2 = true;
        else if (selected == board.getCellAt(15, 1))
            loc_15_1 = true;
        else
            fail("Invalid target selected");
    }
    // Ensure each target was selected at least once
    assertTrue(loc_12_0);
    assertTrue(loc_14_2);
    assertTrue(loc_15_1);
}
```

I have three test methods for testing selection

- One just does a random test as shown above.
- One ensures that the room is always selected if it isn't the last visited
- One ensures that if the room is the last visited, a random choice is made.

Test selecting target – design

Design decisions for you to make:

- What data to pass in to pickLocation. Board cells? Integers? Pass in the Board and call calcTargets inside pickLocation? My UML suggests passing in the list of targets, but you may do this differently.
- How will you ensure the player doesn't just keep returning to the same room? (this is not in UML, something for you to decide with your partner)
- These tests are a lot of effort, ***are they worth it***? Consider the alternative: playing the game over and over to see what the computer player is doing.

Test Accusation

Fairly trivial test

- Accusation = person + weapon + room
- Need to ensure that all 3 are correct
- Issue: how do we know the answer?
    - Easy option: just set to known values
- Test that returns true if person/weapon/room correct
- Test that returns false if wrong person OR wrong weapon OR wrong room (3 slightly different test scenarios)
- Design decisions: (see UML for suggestions)
    - How is the solution represented?
    - What do you call the method?
    - Will you pass in cards or strings?

Create Suggestion – Computer Player

- When the computer player enters a room, it must make a suggestion (human players do too, but human will select from dialog box)
- Goal: ensure the computer player learns something
- Therefore: don't ask for card in hand or already disproved (i.e., suggestion should choose randomly from the "unseen" cards)
- Test must ensure:
    - Weapon is chosen randomly from those not seen
    - Person is chosen randomly from those not seen

- Issue: this test relies on a player that has a hand and a list of "seen" cards. What should we do?
    - Create methods to add cards to the hand and to a list of "seen" cards
    - Call those methods to set up your test scenario(s)
    - Design decision. Will you store Cards or Strings in the "hand" and "seen" lists?
- Issue: suggestion must be based on room the player is in. How do we handle?
    - Create a setter so you can specify.
    - Design decision. UML specifies that Player knows location (row, column). How will you determine what room the player is in?
- Issue: method must return the suggestion
    - Design decision. How will you represent a "suggestion" – as three strings? Maybe as a class with 3 public strings… sort of like a C++ struct?
    - NOTE: for this type of usage, we can break the "private only" policy. Why is that OK?
    - My UML has a possible option
- You will probably want to use the same data structure for both checking an accusation and returning/disproving a suggestion

Test Disproving a Suggestion

How disproving works in the board game

- Person is in room, makes a *suggestion*
- Disproving starts with person on the left.
    - If that person has a card, they show it just to the player who made the suggestion
        - We have dumb computer players, so we will "show" the card to all players. That's in a future lab.
    - If the person has multiple cards, they choose just one to show (as humans we try to use strategy here… our players are not that smart and will just randomly choose)
        - We don't have a "left" player, so we'll start at the next player in the list. Must remember to include human.
- If that person can't disprove, it goes to the next person
- This continues until either a) a card has been shown or b) all players have been queried and none can disprove
    - We return null if no one can disprove

Does null return == solution?

- The fact that no one returned a card doesn't necessarily mean that suggestion is the final solution! Remember that a player can suggest cards in his/her hand, in order to look for one specific card. Humans will do this; our computer players will not be that smart ☺

Design decisions for suggestions

- How is this task assigned to the classes?
    - THINK: we don't want to "reach into the pocket" of another class… so the **Player** should search its list of cards. Is this different for human than computer? No – our human player doesn't decide what card to show. So **Player** has a disproveSuggestion method.
    - Querying the players in order is *not* done by the Player class… this should be a method of… the board? the game? Pros and cons either way, but let's put it in the **Board** class and call it handleSuggestion.
    - Design decision: What parameter(s) should be sent to disproveSuggestion? To handleSuggestion? You've already given this some thought when creating a suggestion.
- Design decision: How will we know which player is the accusing player?

Disprove Suggestion – Computer Player

First test the behavior of a single player

- If a player has one matching card, return it. Simple way to test: create a player with known cards, call the disprove method, ensure desired card is returned. I test returning a room, a weapon and a person, but this may be overkill (does ensure matches happen regardless of card type).
- If a player has no matching cards, return null. Test as above, but obviously with no matching card.
- If the player has multiple cards that match, return should be random. How to test? Set up player, use a loop, ensure that each matching card is returned at least once.

Handle suggestion – Board

Testing the overall behavior to disprove a suggestion is more complex.

- Setup: Create a small number of players with known cards (simulated deal. But it's not necessary to distribute all cards).
- Be sure to include the human.
- Players are queried in order. Some ways to test:
    - Do a query that no players can disprove, ensure null returned.
    - Do a query that only the *accusing player* can disprove, ensure null.
    - Do a query that only the *human* can disprove, human is *not* accuser, verify return card from human's hand.

- Do a query that only the human can disprove, human ***is*** the accuser, ensure null returned.
- Players are queried in order. Do several queries with player 0 as accuser. Do a query that player 1 and 2 can disprove, ensure player 1 disproves (ensures players are not asked after one can disprove).
- Do a query that human and another player can disprove, ensure correct response.

- Hints
  - I have @BeforeClass method creates a number of cards that I can use to create specific test conditions. Cards are static (only one copy needed, set up in @BeforeClass which is static)

    ```
    mustardCard = new Card("Colonel Mustard",  Card.CardType.PERSON);
    ```

  - The scenarios I describe should test the majority of paths through your code. Feel free to add others. You could also try using Emma (code coverage).