CS205 - Final Project - Moose in the House
DELIVERABLE II

Group #2 (Rachel Temple, Joey Palchak, Jonathan Carter, Ian Benson, Cj Zhang)

The following data is a mixture of testing data, code snippets and screen shots from V0.1 of our groups project *Moose In The House*. We have designed almost all the classes from our aforementioned specifications. Each class is specified below along with their testing data.

DESCRIPTIONS & TEST RESULTS

MITH_Card

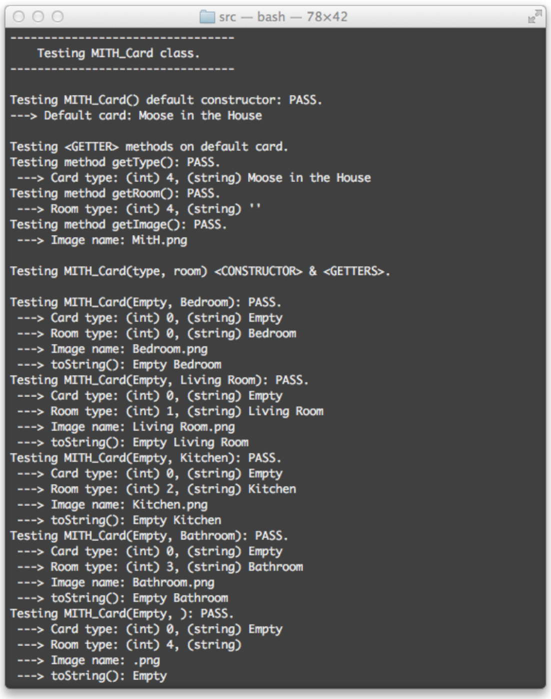
Class MITH_Card was the first of the primary classes developed in the initial design phase; perhaps one of the most crucial class objects required to run the game, while also possibly the simplest to implement.

Testing for the Card class was accomplished through the implementation of a main method within the class file, which would successivelycall each method built within the class. The test method would run through every possible iteration of a card object, examining both positive and negative outcomes and determine if the class was capable of properly constructing every card required to play a game of Moose in the House.

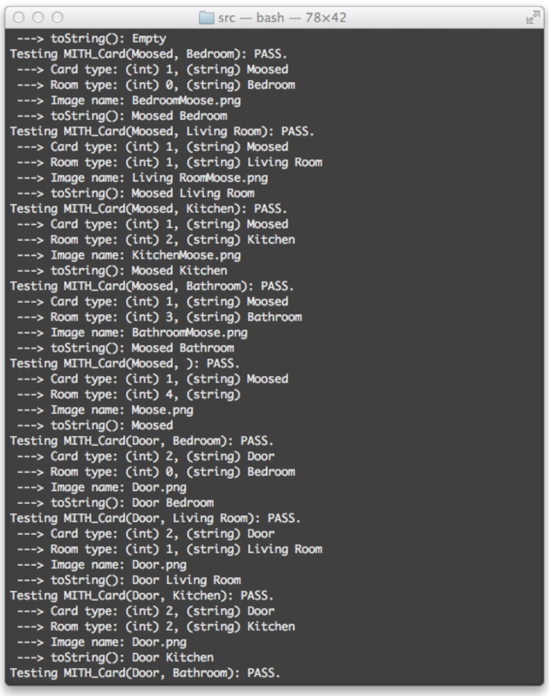
A card object itself is represented by a collective of three variables: an integer representing the card type (Empty Room, Occupied Room, Moose in the House, Trap, or Door), an integer representing room type / card value (Bedroom, Bathroom, Kitchen, Living Room, or non-room), and a string which represents the name of the card's image (necessary for GUI implementation). These card values can be translated into string values by calling upon public methods within the class.

The construction of Card objects is accomplished entirely within the framework of the MITH_Deck class; during the process of creating the 58 card deck, a desired card is produced by calling the MITH_Card constructor method and specifying the card's exact values.

It was noticed, however, during the testing phase of MITH_Card that if a card object was called with illegal values (that is, if a card was called for which did not exist within the context of the game) the card would still be constructed. For instance, if a Trap card was called with a room type of Bathroom, a "Trap Bathroom" card would indeed be created - as opposed to the proper “Trap” card with an empty room type value. An example of this can be seen in the testing output shown below.

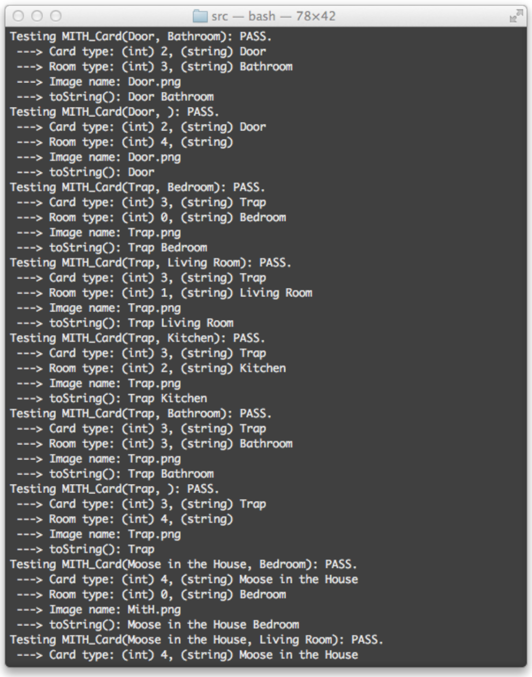


```
---------------------------------
    Testing MITH_Card class.
---------------------------------

Testing MITH_Card() default constructor: PASS.
---> Default card: Moose in the House

Testing <GETTER> methods on default card.
Testing method getType(): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
Testing method getRoom(): PASS.
 ---> Room type: (int) 4, (string) ''
Testing method getImage(): PASS.
 ---> Image name: MitH.png

Testing MITH_Card(type, room) <CONSTRUCTOR> & <GETTERS>.

Testing MITH_Card(Empty, Bedroom): PASS.
 ---> Card type: (int) 0, (string) Empty
 ---> Room type: (int) 0, (string) Bedroom
 ---> Image name: Bedroom.png
 ---> toString(): Empty Bedroom
Testing MITH_Card(Empty, Living Room): PASS.
 ---> Card type: (int) 0, (string) Empty
 ---> Room type: (int) 1, (string) Living Room
 ---> Image name: Living Room.png
 ---> toString(): Empty Living Room
Testing MITH_Card(Empty, Kitchen): PASS.
 ---> Card type: (int) 0, (string) Empty
 ---> Room type: (int) 2, (string) Kitchen
 ---> Image name: Kitchen.png
 ---> toString(): Empty Kitchen
Testing MITH_Card(Empty, Bathroom): PASS.
 ---> Card type: (int) 0, (string) Empty
 ---> Room type: (int) 3, (string) Bathroom
 ---> Image name: Bathroom.png
 ---> toString(): Empty Bathroom
Testing MITH_Card(Empty, ): PASS.
 ---> Card type: (int) 0, (string) Empty
 ---> Room type: (int) 4, (string)
 ---> Image name: .png
 ---> toString(): Empty
```



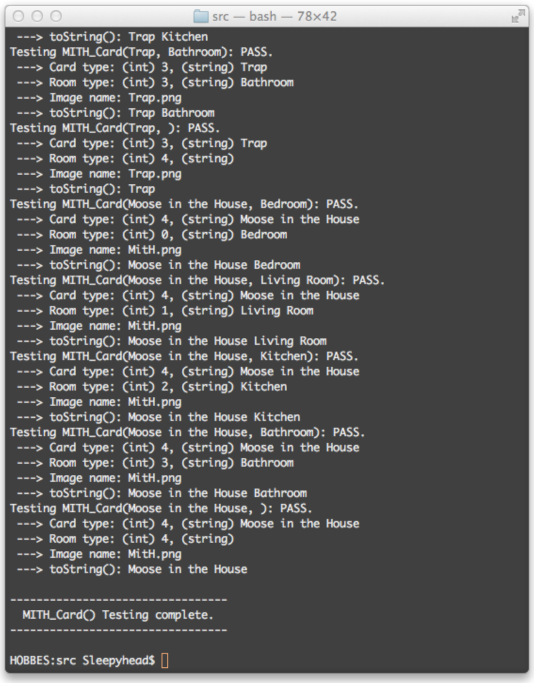
```
 ---> toString(): Empty
Testing MITH_Card(Moosed, Bedroom): PASS.
 ---> Card type: (int) 1, (string) Moosed
 ---> Room type: (int) 0, (string) Bedroom
 ---> Image name: BedroomMoose.png
 ---> toString(): Moosed Bedroom
Testing MITH_Card(Moosed, Living Room): PASS.
 ---> Card type: (int) 1, (string) Moosed
 ---> Room type: (int) 1, (string) Living Room
 ---> Image name: Living RoomMoose.png
 ---> toString(): Moosed Living Room
Testing MITH_Card(Moosed, Kitchen): PASS.
 ---> Card type: (int) 1, (string) Moosed
 ---> Room type: (int) 2, (string) Kitchen
 ---> Image name: KitchenMoose.png
 ---> toString(): Moosed Kitchen
Testing MITH_Card(Moosed, Bathroom): PASS.
 ---> Card type: (int) 1, (string) Moosed
 ---> Room type: (int) 3, (string) Bathroom
 ---> Image name: BathroomMoose.png
 ---> toString(): Moosed Bathroom
Testing MITH_Card(Moosed, ): PASS.
 ---> Card type: (int) 1, (string) Moosed
 ---> Room type: (int) 4, (string)
 ---> Image name: Moose.png
 ---> toString(): Moosed
Testing MITH_Card(Door, Bedroom): PASS.
 ---> Card type: (int) 2, (string) Door
 ---> Room type: (int) 0, (string) Bedroom
 ---> Image name: Door.png
 ---> toString(): Door Bedroom
Testing MITH_Card(Door, Living Room): PASS.
 ---> Card type: (int) 2, (string) Door
 ---> Room type: (int) 1, (string) Living Room
 ---> Image name: Door.png
 ---> toString(): Door Living Room
Testing MITH_Card(Door, Kitchen): PASS.
 ---> Card type: (int) 2, (string) Door
 ---> Room type: (int) 2, (string) Kitchen
 ---> Image name: Door.png
 ---> toString(): Door Kitchen
Testing MITH_Card(Door, Bathroom): PASS.
```

The default constructor passed, unsurprisingly, as its hardcoded values produce a simple Moose in the House card. However, after observing the leniency of the primary constructor, as shown, multiple if-statements were included which helped to alleviate the issue.



src — bash — 78×42

```
Testing MITH_Card(Door, Bathroom): PASS.
 ---> Card type: (int) 2, (string) Door
 ---> Room type: (int) 3, (string) Bathroom
 ---> Image name: Door.png
 ---> toString(): Door Bathroom
Testing MITH_Card(Door, ): PASS.
 ---> Card type: (int) 2, (string) Door
 ---> Room type: (int) 4, (string)
 ---> Image name: Door.png
 ---> toString(): Door
Testing MITH_Card(Trap, Bedroom): PASS.
 ---> Card type: (int) 3, (string) Trap
 ---> Room type: (int) 0, (string) Bedroom
 ---> Image name: Trap.png
 ---> toString(): Trap Bedroom
Testing MITH_Card(Trap, Living Room): PASS.
 ---> Card type: (int) 3, (string) Trap
 ---> Room type: (int) 1, (string) Living Room
 ---> Image name: Trap.png
 ---> toString(): Trap Living Room
Testing MITH_Card(Trap, Kitchen): PASS.
 ---> Card type: (int) 3, (string) Trap
 ---> Room type: (int) 2, (string) Kitchen
 ---> Image name: Trap.png
 ---> toString(): Trap Kitchen
Testing MITH_Card(Trap, Bathroom): PASS.
 ---> Card type: (int) 3, (string) Trap
 ---> Room type: (int) 3, (string) Bathroom
 ---> Image name: Trap.png
 ---> toString(): Trap Bathroom
Testing MITH_Card(Trap, ): PASS.
 ---> Card type: (int) 3, (string) Trap
 ---> Room type: (int) 4, (string)
 ---> Image name: Trap.png
 ---> toString(): Trap
Testing MITH_Card(Moose in the House, Bedroom): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
 ---> Room type: (int) 0, (string) Bedroom
 ---> Image name: MitH.png
 ---> toString(): Moose in the House Bedroom
Testing MITH_Card(Moose in the House, Living Room): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
```



src — bash — 78×42

```
 ---> toString(): Trap Kitchen
Testing MITH_Card(Trap, Bathroom): PASS.
 ---> Card type: (int) 3, (string) Trap
 ---> Room type: (int) 3, (string) Bathroom
 ---> Image name: Trap.png
 ---> toString(): Trap Bathroom
Testing MITH_Card(Trap, ): PASS.
 ---> Card type: (int) 3, (string) Trap
 ---> Room type: (int) 4, (string)
 ---> Image name: Trap.png
 ---> toString(): Trap
Testing MITH_Card(Moose in the House, Bedroom): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
 ---> Room type: (int) 0, (string) Bedroom
 ---> Image name: MitH.png
 ---> toString(): Moose in the House Bedroom
Testing MITH_Card(Moose in the House, Living Room): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
 ---> Room type: (int) 1, (string) Living Room
 ---> Image name: MitH.png
 ---> toString(): Moose in the House Living Room
Testing MITH_Card(Moose in the House, Kitchen): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
 ---> Room type: (int) 2, (string) Kitchen
 ---> Image name: MitH.png
 ---> toString(): Moose in the House Kitchen
Testing MITH_Card(Moose in the House, Bathroom): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
 ---> Room type: (int) 3, (string) Bathroom
 ---> Image name: MitH.png
 ---> toString(): Moose in the House Bathroom
Testing MITH_Card(Moose in the House, ): PASS.
 ---> Card type: (int) 4, (string) Moose in the House
 ---> Room type: (int) 4, (string)
 ---> Image name: MitH.png
 ---> toString(): Moose in the House

--------------------------------
  MITH_Card() Testing complete.
--------------------------------

HOBBES:src Sleepyhead$ 
```

MITH_Deck

Class MITH_Deck() creates and represents the entire 58 card deck that is used to play the game. The class contains various functions that are necessary for the game to play properly. Creates 40 room cards, 20 occupied and 20 non-occupied rooms, 10 Moose in the House cards, 5 doors and 3 traps.

The class was tested through a main function that was written with the interior code and logic in mind. The drawCard() allows the user to pick the top card from the deck and then remove it. This function was tested by displaying the entire deck first. Then the function is called and pulls the top card and compares it to the card that is listed as the top card. If the two cards match then the function is considered working.

The shuffle method is tested by printing out a totally new random 58 card deck. This is considered a successful method if the new shuffled deck does not match the previous one. This was very easy to test and implement.

A method called numCard was created to count the number of cards in the deck, and was tested by counting the number of cards. If there were 58 cards in the deck then the function was labeled as passing.

While playing the actual game, we discovered that once all the players have used the "Moose in the House" card the remaining ones in the deck are pointless. So the method removeMITH() was created. This method removes all the remaining “Moose in the House” cards from the deck once every player has one at least in their hand.

We also decided that there should be a method that lets you know when the deck is empty. So the method isEmpty() was implemented. This was easily tested by checking to see if the current deck contained zero cards. If it was then the method passed as true. Otherwise the deck was not empty and drawing a new card would still be possible.

MITH_Hand

The hand class was designed to keep track of all the cards in a players hand during the game. It uses objects from MITH_Deck and MITH_Card.
It takes the array of cards that MITH_Card created and uses that to fill the hand.

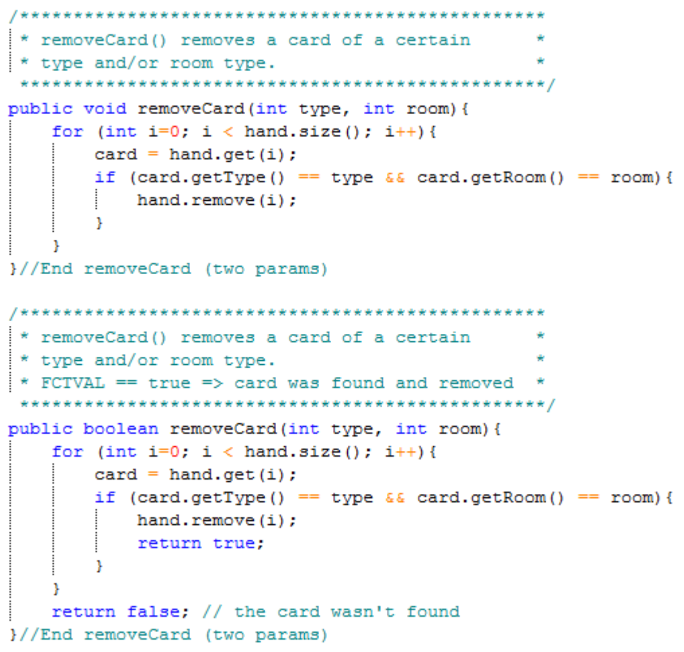
This was an important class to develop as it is one of the primary gameplay functions. It was fairly easy to code as it is made up of a only a few methods.

Testing for this class was done by implementing a main function within the file. This function tested each of the methods that MITH_Hand uses for gameplay. First a new hand is created, and the size is checked. If the size of the hand is zero then a card is added to the hand. This was how the addCard() method was tested.

The isEmpty() function was one of the easier functions to test simply because it checks the size of the hand and if the size is equal to zero then the function is considered successful.

In order to test the removeCard function the hand could not be empty. So if the hand size was equal to zero a card was added, a statement printed out displaying the new hand size and then the card was removed and the new hand size was printed.

While testing Hand a bug was revealed in the removeCard() function. The method was to look for a card in the hand and, if found, remove it. However the way it was initially written, it removed all instances of the given card. The function, with it’s fixed version is pictured below. Note that the return type was changed to allow feedback to the caller about whether or not the operation was successful. This required updating other classes and their tests.



```
/**************************************************
 * removeCard() removes a card of a certain      *
 * type and/or room type.                        *
 **************************************************/
public void removeCard(int type, int room){
    for (int i=0; i < hand.size(); i++){
        card = hand.get(i);
        if (card.getType() == type && card.getRoom() == room){
            hand.remove(i);
        }
    }
}//End removeCard (two params)

/**************************************************
 * removeCard() removes a card of a certain      *
 * type and/or room type.                        *
 * FCTVAL == true => card was found and removed  *
 **************************************************/
public boolean removeCard(int type, int room){
    for (int i=0; i < hand.size(); i++){
        card = hand.get(i);
        if (card.getType() == type && card.getRoom() == room){
            hand.remove(i);
            return true;
        }
    }
    return false; // the card wasn't found
}//End removeCard (two params)
```

MITH_House()

The class MITH_House was added to the system after the initial design phase when it was realized while designing the implementation of the Player and Game classes that the complex interactions between the game manager, players and the house cards on the table would be simplified if there was a common object for them to interact with. It was decided that development of the House class was crucial to having meaningful interaction with the Player class and so it was developed first.

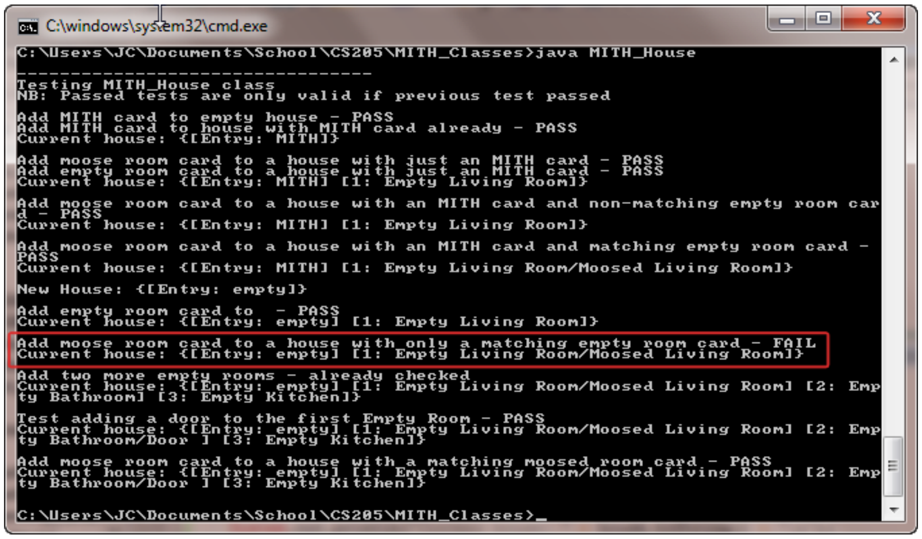
Testing for the house class was accomplished through a main method within the class which exercises each method of the class, examining both positive and negative actions and determining if the desired behavior occurred. An example of the test code is given below. The state of the house is represented by a string showing the Entry as either empty or having a Moose in the House card followed by bracketed and enumerated card lists that should either be single (in the event of an empty room) or in pairs (if a moosed room is played on top of an empty room). Clearly more testing will be required when the class is mated to the GUI and cards' images are shown. Most of that testing will be within the GUI and Card classes to make sure that the correct images are exchanged.

The result of any one test relies on the correctness of behavior of the system for all previous tests. This is because the tests independently assume the state of the sub-system, the Hand, at the beginning of each test so a previously failed test could leave the state of the Hand in an illegal state or an unexpected legal state. For example, if the test to add a moose to an empty room when there is no Moose in the House card present fails and the moose is added anyway, then the following test expecting that room to remain empty will not have the proper initial conditions. This requirement that all previous tests pass to validate a current test is stated at the beginning of the test results output.

The test driver was developed along with the code of the House class. Each test consists of an attempt to perform an action on the House and then an if/else construct is used to print the corresponding result. While this is essentially white-box testing because it belongs to the class it is testing, the tests are written to only exercise public methods and make no direct assumptions of the state of private members of the class. An attempt was made to make sure tests were written before the code for that interaction was implemented so that tests would change from failed to passed as the code was developed.

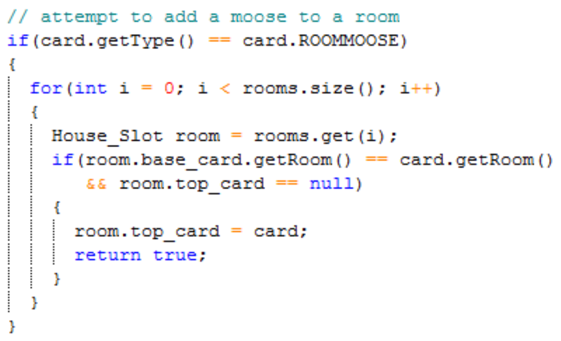
Sometimes coding inertia flip-flopped the process and tests had to be written for code that already existed. Because the class was developed independently of all other classes except the Card class, testing and development proceeded in a mostly linear fashion. Each requirement was picked, a test for it was written, the code was shown to fail, and then that requirement was implemented. There were a few cases where this didn't happen and the testing was able to expose the error. Here is an illustration of one such case.

The screenshot of test output shows that the test for adding a moose to an empty room in the house when no Moose in the House card is present failed meaning that the class allowed the addition to happen.



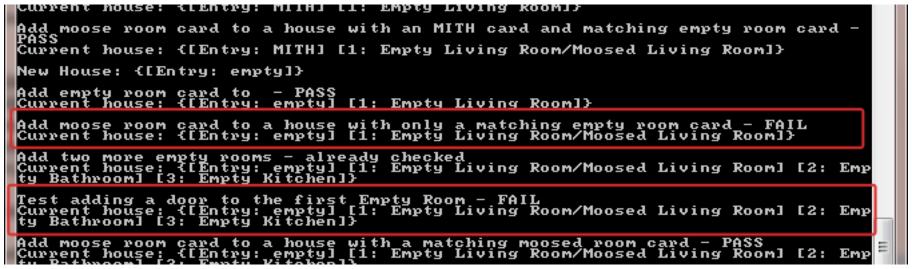
```
C:\Users\JC\Documents\School\CS205\MITH_Classes>java MITH_House
---------------------------------
Testing MITH_House class
NB: Passed tests are only valid if previous test passed

Add MITH card to empty house - PASS
Add MITH card to house with MITH card already - PASS
Current house: {[Entry: MITH]}

Add moose room card to a house with just an MITH card - PASS
Add empty room card to a house with just an MITH card - PASS
Current house: {[Entry: MITH] [1: Empty Living Room]}

Add moose room card to a house with an MITH card and non-matching empty room car
d - PASS
Current house: {[Entry: MITH] [1: Empty Living Room]}

Add moose room card to a house with an MITH card and matching empty room card -
PASS
Current house: {[Entry: MITH] [1: Empty Living Room/Moosed Living Room]}

New House: {[Entry: empty]}

Add empty room card to  - PASS
Current house: {[Entry: empty] [1: Empty Living Room]}

Add moose room card to a house with only a matching empty room card - FAIL
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room]}

Add two more empty rooms - already checked
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room] [2: Emp
ty Bathroom] [3: Empty Kitchen]}

Test adding a door to the first Empty Room - PASS
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room] [2: Emp
ty Bathroom/Door ] [3: Empty Kitchen]}

Add moose room card to a house with a matching moosed room card - PASS
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room] [2: Emp
ty Bathroom/Door ] [3: Empty Kitchen]}


C:\Users\JC\Documents\School\CS205\MITH_Classes>_
```

Examining the code quickly reveals that the test for presence of a Moose in the House card was omitted.



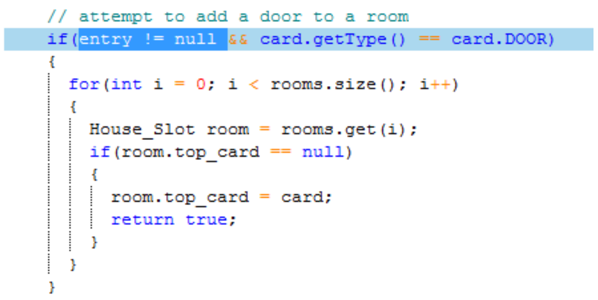
```java
// attempt to add a moose to a room
if(card.getType() == card.ROOMMOOSE)
{
  for(int i = 0; i < rooms.size(); i++)
  {
    House_Slot room = rooms.get(i);
    if(room.base_card.getRoom() == card.getRoom()
       && room.top_card == null)
    {
      room.top_card = card;
      return true;
    }
  }
}
```

The change is made and the test are run again. The results of the next round of testing show that this test still fails. It was then noticed that the test for adding a door to an empty room also failed - even though that test had passed in the previous round.



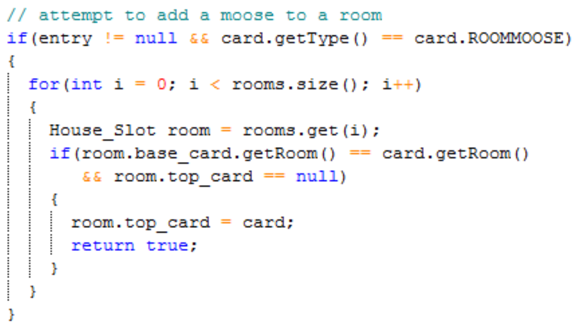
```
Current house: {[Entry: MITH] [1: Empty Living Room]}
Add moose room card to a house with an MITH card and matching empty room card -
PASS
Current house: {[Entry: MITH] [1: Empty Living Room/Moosed Living Room]}
New House: {[Entry: empty]}
Add empty room card to  - PASS
Current house: {[Entry: empty] [1: Empty Living Room]}
Add moose room card to a house with only a matching empty room card - FAIL
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room]}
Add two more empty rooms - already checked
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room] [2: Emp
ty Bathroom] [3: Empty Kitchen]}
Test adding a door to the first Empty Room - FAIL
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room] [2: Emp
ty Bathroom] [3: Empty Kitchen]}
Add moose room card to a house with a matching moosed room card - PASS
Current house: {[Entry: empty] [1: Empty Living Room/Moosed Living Room] [2: Emp
```

Using this information it was quickly recognized that due to the similarities in the code, the fix was added to the wrong section of code.



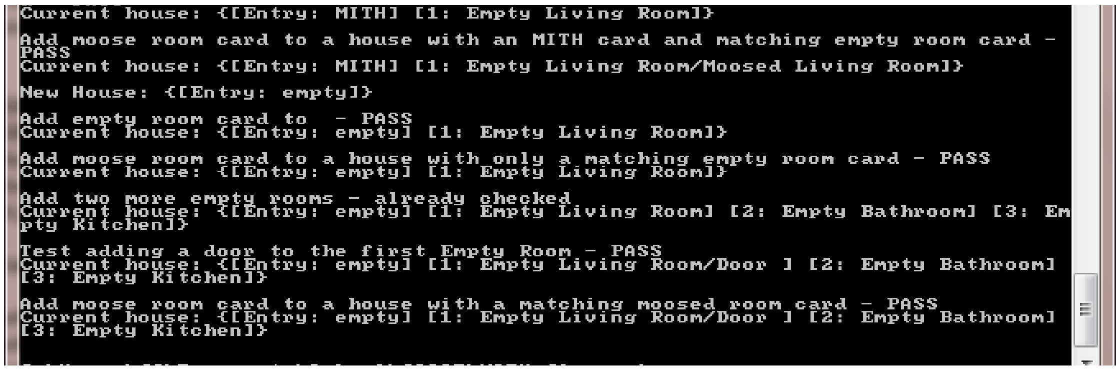
```java
// attempt to add a door to a room
if(entry != null && card.getType() == card.DOOR)
{
  for(int i = 0; i < rooms.size(); i++)
  {
    House_Slot room = rooms.get(i);
    if(room.top_card == null)
    {
      room.top_card = card;
      return true;
    }
  }
}
```

The incorrect change was removed and the correction was added to the correct place in the code.



```java
// attempt to add a moose to a room
if(entry != null && card.getType() == card.ROOMMOOSE)
{
  for(int i = 0; i < rooms.size(); i++)
  {
    House_Slot room = rooms.get(i);
    if(room.base_card.getRoom() == card.getRoom()
       && room.top_card == null)
    {
      room.top_card = card;
      return true;
    }
  }
}
```

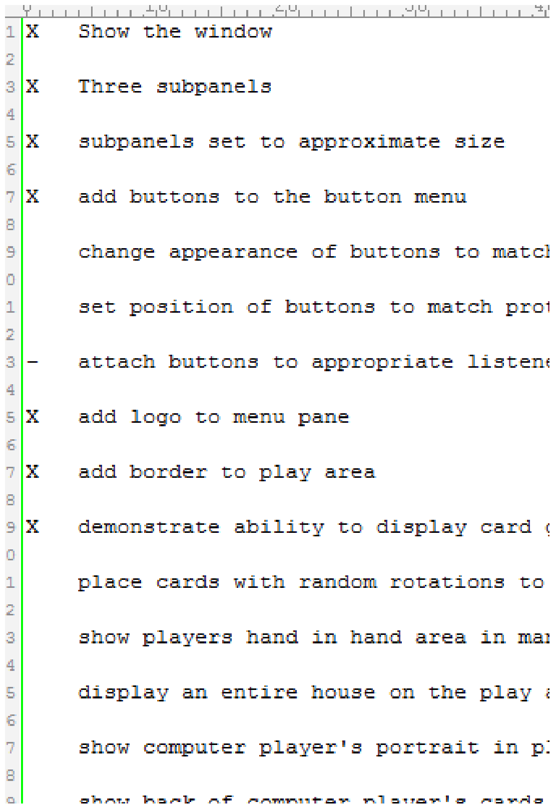
The next time the test suite was run all tests passed.



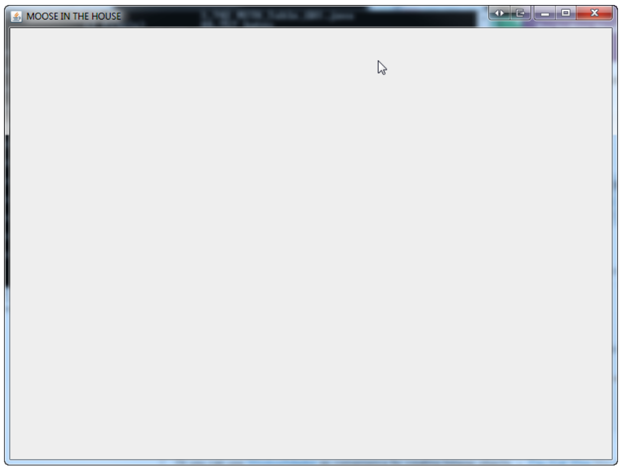
```
Current house: {[Entry: MITH] [1: Empty Living Room]}

Add moose room card to a house with an MITH card and matching empty room card -
PASS
Current house: {[Entry: MITH] [1: Empty Living Room/Moosed Living Room]}

New House: {[Entry: empty]}

Add empty room card to  - PASS
Current house: {[Entry: empty] [1: Empty Living Room]}

Add moose room card to a house with only a matching empty room card - PASS
Current house: {[Entry: empty] [1: Empty Living Room]}

Add two more empty rooms - already checked
Current house: {[Entry: empty] [1: Empty Living Room] [2: Empty Bathroom] [3: Em
pty Kitchen]}

Test adding a door to the first Empty Room - PASS
Current house: {[Entry: empty] [1: Empty Living Room/Door ] [2: Empty Bathroom]
[3: Empty Kitchen]}

Add moose room card to a house with a matching moosed room card - PASS
Current house: {[Entry: empty] [1: Empty Living Room/Door ] [2: Empty Bathroom]
[3: Empty Kitchen]}
```

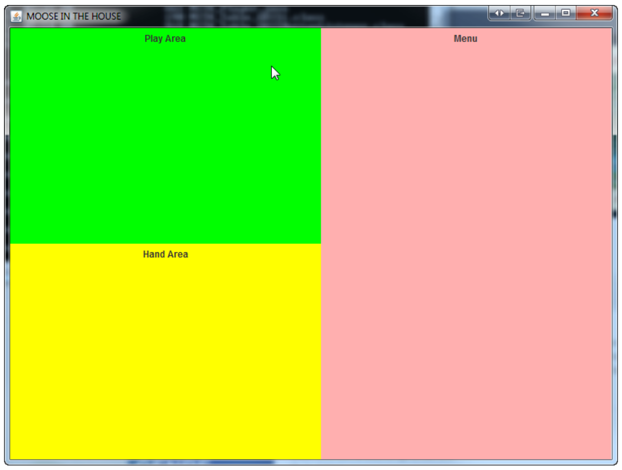
MITH_Table_GUI()

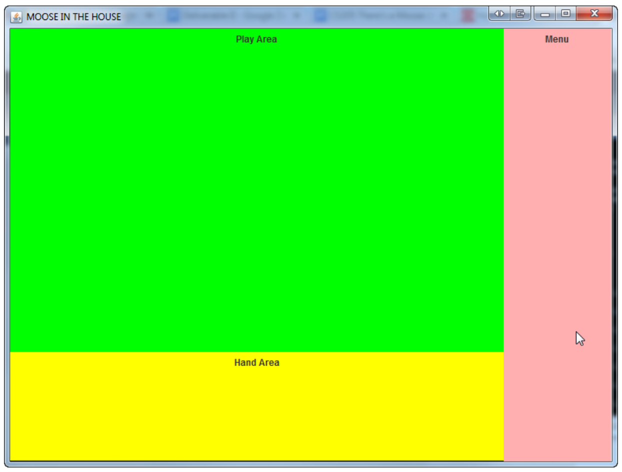
The table GUI class was tested in a very different fashion. GUI development does not lend itself to automatic testing. For this class the test harness consisted of a checklist of desired features of the GUI which incrementally brought the GUI closer to the mockup in the original specification document. There was an initial problem getting anything to appear. The window would only show an overall gray color even though it was intended to show three subpanels, each with a different background color. When they wouldn't appear adding a label to each panel was tried in case an empty panel wouldn't show. Finally it was noticed that the panel holding the other three panels wasn't added to the parent class. Once that happened development proceeded in a smoother manner. The following screenshots show steps in the evolution of the GUI.

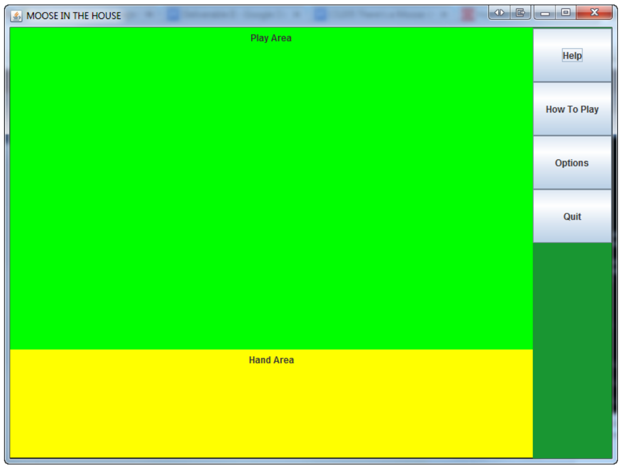


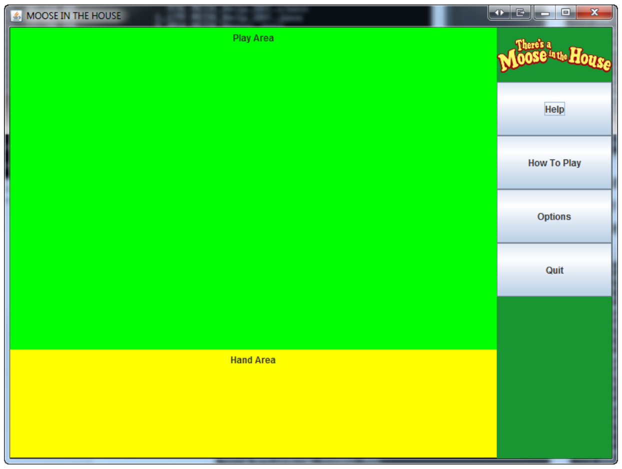
```
X   Show the window

X   Three subpanels

X   subpanels set to approximate size

X   add buttons to the button menu

    change appearance of buttons to matcl

    set position of buttons to match prot

-   attach buttons to appropriate listene

X   add logo to menu pane

X   add border to play area

X   demonstrate ability to display card (

    place cards with random rotations to

    show players hand in hand area in mai

    display an entire house on the play a

    show computer player's portrait in p]

    show back of computer player's cards
```

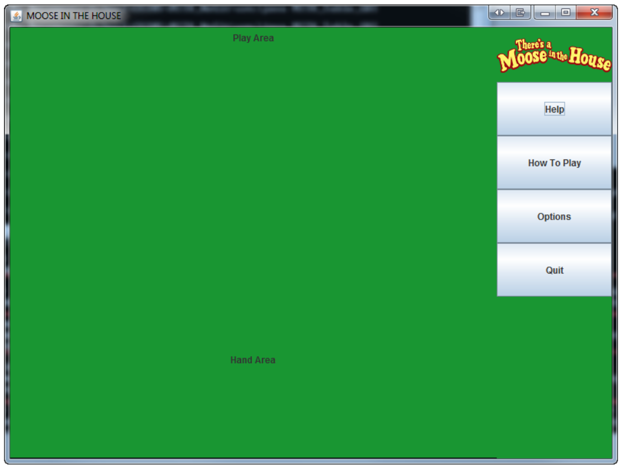


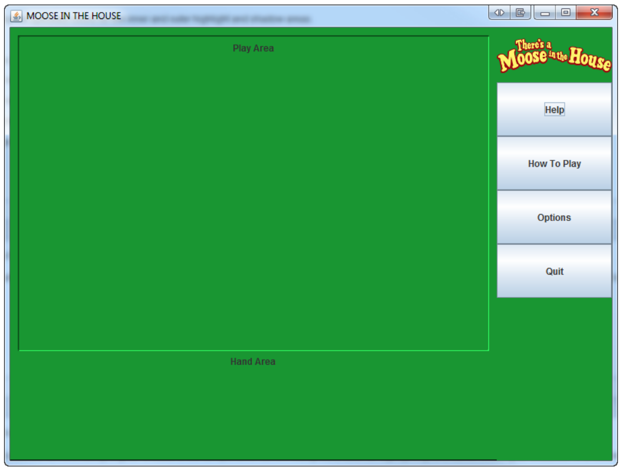


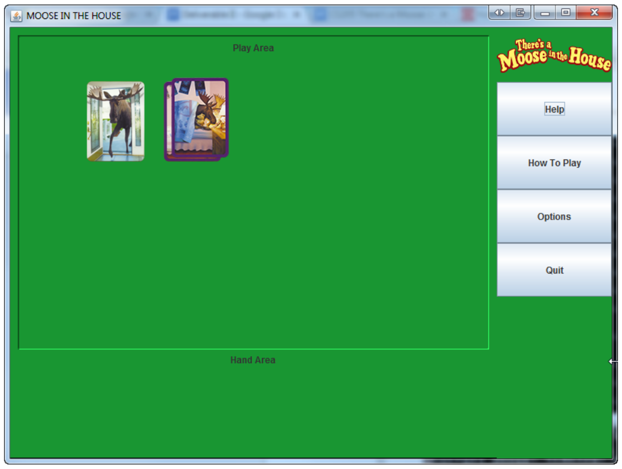



MOOSE IN THE HOUSE
Play Area
Hand Area
Help
How To Play
Options
Quit


MOOSE IN THE HOUSE
Play Area
Hand Area
There's a Moose in the House
Help
How To Play
Options
Quit


MOOSE IN THE HOUSE
Play Area
Hand Area
There's a Moose in the House
Help
How To Play
Options
Quit


MOOSE IN THE HOUSE
Play Area
Hand Area
There's a Moose in the House
Help
How To Play
Options
Quit


MOOSE IN THE HOUSE
Play Area
Hand Area
There's a Moose in the House
Help
How To Play
Options
Quit