**handleSuggestion**

**public** Card handleSuggestion(Solution suggestion, String accusingPlayer) {

// Find the accusing player object

Player accuser = **null**;

**boolean** accuserIsHuman = **false**;

**if** (humanPlayer.getName() == accusingPlayer) {

accuser = humanPlayer;

accuserIsHuman = **true**;

}

**for** (Player player : compPlayers) {

**if** (player.getName() == accusingPlayer) {

accuser = player;

}

}

**if** (accuser == **null**) **return** **null**;

// Query players that come after the accuser in the cycle

Card disprovingCard;

**boolean** startedCycle = **false**;

**if** (accuserIsHuman) startedCycle = **true**;

**for** (ComputerPlayer player : compPlayers) {

**if** (startedCycle) {

disprovingCard = player.disproveSuggestion(suggestion);

**if** (disprovingCard != **null**) {

**return** disprovingCard;

}

}

**if** (player.getName() == accuser.getName()) {

startedCycle = **true**;

}

}

// Query players that come before the accuser in the cycle

**if** (accuserIsHuman) **return** **null**;

disprovingCard = humanPlayer.disproveSuggestion(suggestion);

**if** (disprovingCard != **null**) {

**return** disprovingCard;

}

**for** (ComputerPlayer player : compPlayers) {

**if** (player.getName() == accuser.getName()) {

**break**;

}

disprovingCard = player.disproveSuggestion(suggestion);

**if** (disprovingCard != **null**) {

**return** disprovingCard;

}

}

// Return null if none of the players could disprove

**return** **null**;

}

**handleSuggestion Tests**

// Test that the handleSuggestion function works correctly, and that all

// players are queried. A human player and set of comp players are created

// to test the function with.

HumanPlayer hp = **new** HumanPlayer();

hp.giveCard(gunCard);

hp.giveCard(swordCard);

hp.setName("Human Player");

ArrayList<ComputerPlayer> cps = **new** ArrayList<ComputerPlayer>();

ComputerPlayer tempCP = **new** ComputerPlayer();

tempCP.giveCard(smithCard);

tempCP.giveCard(brownCard);

tempCP.setName("Comp 1");

cps.add(tempCP);

tempCP = **new** ComputerPlayer();

tempCP.giveCard(poolCard);

tempCP.giveCard(kitchenCard);

tempCP.setName("Comp 2");

cps.add(tempCP);

tempCP = **new** ComputerPlayer();

tempCP.giveCard(bowCard);

tempCP.giveCard(maceCard);

tempCP.setName("Comp 3");

cps.add(tempCP);

board.setHumanPlayer(hp);

board.setCompPlayer(cps);

// Test a suggestion that no players can disprove

*assertEquals*(board.handleSuggestion(**new** Solution("Grant Jones", "Balcony", "Axe"), hp.getName()), **null**);

// Test a suggestion that only the human can disprove

*assertEquals*(board.handleSuggestion(**new** Solution("Grant Jones", "Balcony", "Gun"), cps.get(0).getName()).getCardName(), "Gun");

// Test a suggestion that only the accuser can disprove (human)

*assertEquals*(board.handleSuggestion(**new** Solution("Grant Jones", "Balcony", "Sword"), hp.getName()), **null**);

// Test a suggestion that only the accuser can disprove (comp 3)

*assertEquals*(board.handleSuggestion(**new** Solution("Grant Jones", "Balcony", "Mace"), cps.get(2).getName()), **null**);

// Make sure the players are queried in the proper order. The proper order/cycle is:

// 1. The human player

// 2. The computer players in order of their name

// The computer players go in order of their name because they are stored in a TreeSet.

hp = **new** HumanPlayer();

hp.giveCard(gunCard);

hp.setName("CCD");

cps = **new** ArrayList<ComputerPlayer>();

tempCP = **new** ComputerPlayer();

tempCP.giveCard(smithCard);

tempCP.setName("CDD");

cps.add(tempCP);

tempCP = **new** ComputerPlayer();

tempCP.giveCard(poolCard);

tempCP.setName("CCC");

cps.add(tempCP);

tempCP = **new** ComputerPlayer();

tempCP.giveCard(bowCard);

tempCP.setName("ABB");

cps.add(tempCP);

board.setHumanPlayer(hp);

board.setCompPlayer(cps);

// Test suggestion that multiple players can disprove. Ensure first alphabetical

// player is the one to disprove

*assertEquals*(board.handleSuggestion(**new** Solution("John Smith", "Pool", "Crossbow"), hp.getName()).getCardName(), "Crossbow");

// Ensure the entire "cycle" of players are queried by making the

// last computer player the accuser, and the second to last computer player

// the disprover.

*assertEquals*(board.handleSuggestion(**new** Solution("Connor Davis", "Pool", "Sword"), cps.get(0).getName()).getCardName(), "Pool");

}

**pickLocation**

// This function simply selects a target and keeps track of the last

// room chosen. it does not actually MOVE the player to the chosen

// location.

**public** BoardCell pickLocation(Set<BoardCell> targets) {

// Look for room targets

**for** (BoardCell cell : targets) {

**if** (cell.isRoom() && cell.getInitial() != lastRoomChosen) {

lastRoomChosen = cell.getInitial();

**return** cell;

}

}

// Room target not found -> select target randomly

ArrayList<BoardCell> shuffledTargets = **new** ArrayList<BoardCell>();

shuffledTargets.addAll(targets);

Collections.*shuffle*(shuffledTargets);

**return** shuffledTargets.get(0);

}

// Returns either the accusation, or null if an accusation couldn't be made.

**public** Solution makeAccusation() {

Set<Card> cardsNotSeen = Board.*getDeck*();

cardsNotSeen.removeAll(seenCards);

// The computer doesn't make an accusation if it doesn't have enough information.

// The computer will make an accusation once there is only one of each card type

// left that it has not seen.

**int** peopleLeft = 0;

**int** roomsLeft = 0;

**int** weaponsLeft = 0;

Solution accusation = **new** Solution();

**for** (Card card : cardsNotSeen) {

**if** (card.getCardType() == CardType.***PERSON***) {

accusation.person = card.getCardName();

peopleLeft++;

} **else** **if** (card.getCardType() == CardType.***ROOM***) {

accusation.room = card.getCardName();

roomsLeft++;

} **else** {

accusation.weapon = card.getCardName();

weaponsLeft++;

}

}

// Computer couldn't make accusation

**if** (peopleLeft > 1 || roomsLeft > 1 || weaponsLeft > 1) {

**return** **null**;

}

**return** accusation;

}

**pickLocation Tests**

@Test

// This test ensures that:

// - If a room is a target it is chosen.

// - If a room is a target that we visited recently, a target is chosen randomly.

// - That random selection is in fact random.

**public** **void** testTargetSelection() {

// Ensure that a room is chosen if it is in the list of targets and

// was not the last visited room. The following test places the player

// close to a room, and makes sure that the player enters the room.

board.calcTargets(17, 6, 3);

ComputerPlayer testPlayer = **new** ComputerPlayer();

**for** (**int** i=0; i<20; i++) { // Make sure the room was not entered by luck.

testPlayer = **new** ComputerPlayer();

testPlayer.setRow(17);

testPlayer.setCol(6);

*assertTrue*(testPlayer.pickLocation(board.getTargets()).getInitial() == 'S');

}

// Now that this room been visited, repeat the same test as above,

// but make sure that ALL targets are chosen randomly.

**int**[] timesHitEachTarget = **new** **int**[5];

**for** (**int** i=0; i<5; i++) {

timesHitEachTarget[i] = 0;

}

**for** (**int** i=0; i<500; i++) {

testPlayer.setRow(17);

testPlayer.setCol(6);

BoardCell chosenTarget = testPlayer.pickLocation(board.getTargets());

// Test 5 of the possible targets to make sure they are all visited.

**if** (chosenTarget.getRow() == 14 && chosenTarget.getCol() == 6) {

timesHitEachTarget[0]++;

}

**if** (chosenTarget.getRow() == 18 && chosenTarget.getCol() == 8) {

timesHitEachTarget[1]++;

}

**if** (chosenTarget.getRow() == 17 && chosenTarget.getCol() == 4) {

timesHitEachTarget[2]++;

}

**if** (chosenTarget.getRow() == 17 && chosenTarget.getCol() == 7) {

timesHitEachTarget[3]++;

}

**if** (chosenTarget.getRow() == 19 && chosenTarget.getCol() == 5) {

timesHitEachTarget[4]++;

}

}

// Make sure each target was hit at least 5 times.

**for** (**int** i=0; i<5; i++) {

*assertTrue*(timesHitEachTarget[i] >= 5);

}

}

**makeSuggestion**

**public** Solution makeSuggestion(Board board, BoardCell location, Map<Character, String> rooms) {

ArrayList<Card> entireDeck = **new** ArrayList<Card>();

ArrayList<Card> cardsNotSeen = **new** ArrayList<Card>();

entireDeck.addAll(board.*getDeck*());

cardsNotSeen.addAll(entireDeck);

cardsNotSeen.removeAll(seenCards);

Collections.*shuffle*(entireDeck);

Collections.*shuffle*(cardsNotSeen);

// The player must suggest the room they are currently in

Solution suggestion = **new** Solution();

suggestion.room = rooms.get(location.getInitial());

// Randomly make suggestion from entire deck

**for** (Card card : entireDeck) {

**if** (card.getCardType() == CardType.***PERSON***) {

suggestion.person = card.getCardName();

}

**if** (card.getCardType() == CardType.***WEAPON***) {

suggestion.weapon = card.getCardName();

}

}

// Overwrite previous suggestion with cards that have not been seen.

// This ensures that if possible, the suggestion will contain cards

// that have not been seen, but if a player has seen all of one type

// of card, then they will suggest a card they have seen

**for** (Card card : cardsNotSeen) {

**if** (card.getCardType() == CardType.***PERSON***) {

suggestion.person = card.getCardName();

}

**if** (card.getCardType() == CardType.***WEAPON***) {

suggestion.weapon = card.getCardName();

}

}

**return** suggestion;

}

**makeSuggestion Tests**

@Test

// Test that the computer player makes an appropriate suggestion. An appropriate suggestion:

// - Has the room the player is currently in

// - Does not have a weapon or person that the player has already seen

**public** **void** testSuggestionMaking() {

// Put the player on the balcony, and show them a set of cards.

// Under these conditions, there are many possible suggestions

// that the player could make.

ComputerPlayer testPlayer = **new** ComputerPlayer();

testPlayer.showCard(**new** Card("Kitchen", CardType.***ROOM***));

testPlayer.showCard(**new** Card("Pool", CardType.***ROOM***));

testPlayer.showCard(**new** Card("Balcony", CardType.***ROOM***));

testPlayer.showCard(**new** Card("John Smith", CardType.***PERSON***));

testPlayer.showCard(**new** Card("Joe Brown", CardType.***PERSON***));

testPlayer.showCard(**new** Card("Bill Adams", CardType.***PERSON***));

testPlayer.showCard(**new** Card("Gun", CardType.***WEAPON***));

testPlayer.showCard(**new** Card("Sword", CardType.***WEAPON***));

testPlayer.showCard(**new** Card("Crossbow", CardType.***WEAPON***));

testPlayer.setRow(11); // Balcony

testPlayer.setCol(23);

// Because a suggestion is made randomly, we must check it many times

**for** (**int** i=0; i<500; i++) {

Solution suggestion = testPlayer.makeSuggestion(board, board.getCellAt(testPlayer.getRow(), testPlayer.getCol()), board.*getRooms*());

// Suggestion can only be made from the current room

*assertTrue*(suggestion.room.equals("Balcony"));

// Suggestion should only contain cards that have not been seen

**if** (suggestion.room.equals("Kitchen") ||

suggestion.room.equals("Pool") ||

suggestion.person.equals("John Smith") ||

suggestion.person.equals("Joe Brown") ||

suggestion.person.equals("Bill Adams") ||

suggestion.weapon.equals("Gun") ||

suggestion.weapon.equals("Sword") ||

suggestion.weapon.equals("Crossbow")) {

System.***out***.println(suggestion.person + " " + suggestion.weapon + " ");

*assertTrue*(**false**);

}

}

// Show the computer all but one person and all but one weapon, so that only one

// suggestion is possible. The player will NOT see: "Grant Jones", "Hammer".

// We need not show them all the rooms, because only one room is possible anyways.

testPlayer.showCard(**new** Card("Jake Williams", CardType.***PERSON***));

testPlayer.showCard(**new** Card("Connor Davis", CardType.***PERSON***));

testPlayer.showCard(**new** Card("Mace", CardType.***WEAPON***));

testPlayer.showCard(**new** Card("Axe", CardType.***WEAPON***));

// Because makeSuggestion is random, test multiple times

**for** (**int** i=0; i<100; i++) {

Solution suggestion = testPlayer.makeSuggestion(board, board.getCellAt(testPlayer.getRow(), testPlayer.getCol()), board.*getRooms*());

*assertTrue*(suggestion.room.equals("Balcony"));

*assertTrue*(suggestion.person.equals("Grant Jones"));

*assertTrue*(suggestion.weapon.equals("Hammer"));

}

}

}