Actual refactoring code				
	Previous	Later	Description/Need of refactoring	Testing class
1	@Override public boolean validateCommand(){ Player l_Player = getOrderInfo().getPlayer(); Country l_TargetCountry = getOrderInfo(). getTargetCountry(); // Check for valid player if(l_Player==null){ Constants.printValidationOfValidateCommand ("Invalid Player"); d_GameEventLogger.logEvent("The Player is invalid"); return false; } // Check for if the player has the card if(!l_Player.checkIfCardAvailable(CardType. BOMB)){ Constants.printValidationOfValidateCommand ("Invalid BOMB Card"); d_GameEventLogger.logEvent("The BOMB card is invalid"); return false; } // check if the target country belongs to the player if(l_Player.getCapturedCountries().contains (l_TargetCountry)){ Constants.printValidationOfValidateCommand ("The Player cannot bomb its own country"); d_GameEventLogger.logEvent("The Player cannot bomb its own country"); return false; } // Check if diplomacy is there or not if(l_Player.getNeutralPlayers().contains (l_TargetCountry.getPlayer())){ System.out.printf("There is diplomacy between %s and %s\n", l_Player.getName(), l_TargetCountry. getPlayer().getName()); d_GameEventLogger.logEvent("There is diplomacy between the countries"); l_Player.getNeutralPlayers().remove (l_TargetCountry.getPlayer()); l_TargetCountry.getPlayer(). getNeutralPlayers().remove(l_Player); return false; } // Validate if the target country is a neighbor of the player owned country boolean l_adjacentCountry = false; for(Country l_OwnCountry:l_Player. getCapturedCountries()){ HashMap<String, Country> Neighbors = l_OwnCountry.getNeighbours(); if (Neighbors.containsKey(l_TargetCountry. getCountryId().toLowerCase())) { l_adjacentCountry = true; break; } } if (!l_adjacentCountry){ Constants.printValidationOfValidateCommand ("The target country is not a neighbor of player owned country"); d_GameEventLogger.logEvent("The target country is not a neighbor of player owned country"); return false; } return true; }	public boolean validateCommand() { Player l_Player = getOrderInfo().getPlayer(); Country l_TargetCountry = getOrderInfo().getTargetCountry(); if (l_Player == null) { System.err.println("The Player is not valid."); d_Logger.log("The Player is not valid."); return false; } // validate that the player has the bomb card if (!l_Player.checkIfCardAvailable(CardType.BOMB)) { System.err.println("Player doesn't have Bomb Card."); d_Logger.log("Player doesn't have Bomb Card."); return false; } //check whether the target country belongs to the player if (l_Player.getCapturedCountries().contains(l_TargetCountry)) { System.err.println("The player cannot destroy armies in his own country."); d_Logger.log("The player cannot destroy armies in his own country."); return false; } // validate that the country is adjacent to one of the neighbors of the current player Boolean l_Adjacent = false; for (Country l_PlayerCountry : l_Player. getCapturedCountries()) { for (Country l_NeighbourCountry : l_PlayerCountry. getNeighbors()) { if (l_NeighbourCountry.getName().equals (l_TargetCountry.getName())) { l_Adjacent = true; break; } } } if (!l_Adjacent) { System.err.println("The target country is not adjacent to one of the countries that belong to the player."); d_Logger.log("The target country is not adjacent to one of the countries that belong to the player."); return false; } //Check diplomacy if (l_Player.getNeutralPlayers().contains(l_TargetCountry. getPlayer())) { System.err.printf("Truce between %s and %s\n", l_Player. getName(), l_TargetCountry.getPlayer().getName()); d_Logger.log("Truce between" + l_Player.getName() + "and " + l_TargetCountry.getPlayer().getName()); l_Player.getNeutralPlayers().remove(l_TargetCountry. getPlayer()); l_TargetCountry.getPlayer().getNeutralPlayers().remove (l_Player); return false; } return true; }	Added log infos and more validation in the validate and execute command functions for all the orders.	BlockadeOrderTest.java

2	private String getCommandFromPlayer() {
	String l_Command;
	System.out.println(Constants.
	ISSUE_COMMAND_MESSAGE);
	Constants.showIssueOrderCommand();
	l_Command = d_Scanner.nextLine();
	//Todo add validation
	if(Objects.equals(l_Command.split(" ")[0],
	Constants.SHOW_MAP)){
	new ShowMapController(d_GameMap).
	show();
	return getCommandFromPlayer();
	}
	//Todo add validation
	return l_Command;
	}
		public void showPlayerStatusAndCommands(Player p_Player) {
		d_Logger.log(Constants.EQUAL_SEPARATOR);
		d_Logger.log("List of game loop commands");
		d_Logger.log("To deploy the armies : deploy countryID
		numarmies");
		d_Logger.log("To advance/attack the armies : advance
		countrynamefrom countynameto numarmies");
		d_Logger.log("To airlift the armies : airlift sourcecountryID
		targetcountryID numarmies");
		d_Logger.log("To blockade the armies : blockade countryID");
		d_Logger.log("To negotiate with player : negotiate playerID");
		d_Logger.log("To bomb the country : bomb countryID");
		d_Logger.log("To skip: pass");
		d_Logger.log(Constants.EQUAL_SEPARATOR);
		String l_Table = "|%-15s|%-19s|%-22s|%n";
		System.out.format
		("|===============|======================|===========
		==========|%n");
		System.out.format("| Current Player | Initial Assigned | Left
		Armies | %n");
		System.out.format
		("|===============|======================|===========
		==========|%n");
		System.out.format(l_Table, p_Player.getName(), p_Player.
		getReinforcementArmies(), p_Player.getIssuedArmies());
		System.out.format
		("|===============|=======================|==========
		=========|%n");
		
		d_Logger.log(Constants.ASSIGNED_COUNTRIES);
		System.out.format
		("|==============|=======================|===========
		=======|=========|%n");
		
		System.out.format(
		"|Country name |Country Armies | Neighbors countries
		|%n");
		System.out.format(
		
		"|==============|=======================|============
		======|=========|%n");
		for (Country l_Country : p_Player.getCapturedCountries()) {
		String l_TableCountry = "|%-15s|%-15s|%-35s|%n";
		String l_NeighborList = "";
		for (Country l_Neighbor : l_Country.getNeighbors()) {
		l_NeighborList += l_Neighbor.getName() + "-";
		}
		System.out.format(l_TableCountry, l_Country.getName(),
		l_Country.getArmies(), l_Country.createANeighborList(l_Country.
		getNeighbors()));
		}
		System.out.format
		("|==============|=======================|===========
		=======|=========|\n");
		
		d_Logger.log(Constants.CARDS_OF_PLAYER);
		if (!p_Player.getPlayerCards().isEmpty()) {
		for (Card l_Card : p_Player.getPlayerCards()) {
		d_Logger.log(l_Card.getCardType().toString());
		}
		}
		if (!p_Player.getOrders().isEmpty()) {
		d_Logger.log("The Orders issued by the Player " +
		p_Player.getName() + " are:");
		for (Order l_Order : p_Player.getOrders()) {
		d_Logger.log(l_Order.getOrderInfo().getCommand());
		}
		}
		}
			Improvised the data for the showing to the user since previously when any user play a game then it was not allowing the user to see neighbour countries, total armies to see complete status of the map, game and his/her current position.	IssueOrderTest.java

3	@Override
	public void printOrderCommand() {
	System.out.println("Advanced " + getOrderInfo().
	getNumberOfArmy() + " armies " + " from " +
	getOrderInfo().getDeparture().getCountryId() + " to " +
	getOrderInfo().getDestination().getCountryId() + ".");
	System.out.println(Constants.SEPERATER);
	d_GameEventLogger.logEvent("Advanced " +
	getOrderInfo().getNumberOfArmy() + " armies " + "
	from " + getOrderInfo().getDeparture().getCountryId()
	+ " to " + getOrderInfo().getDestination().
	getCountryId() + ".");
	}
		public class GameConsoleWriter implements Observer,
		Serializable {
		
		/**
		* Updates the console with the provided message.
		*
		* @param p_s The message to be displayed on the console.
		*/
		@Override
		public void update(String p_s) {
		System.out.println(p_s);
		}
		
		/**
		* Clears the console logs by resetting the console screen.
		*/
		@Override
		public void clearGameLogs() {
		System.out.print("\033[H\033[2J"); // ANSI escape sequence
		to clear console screen
		}
		}
		
		@Override
		public void printOrderCommand() {
		d_Logger.log("Order Info: Advance " + getOrderInfo().
		getNumberOfArmy() + " armies " + " from " + getOrderInfo().
		getDeparture().getName() + " to " + getOrderInfo().getDestination().
		getName() + ".");
		d_Logger.log(Constants.EQUAL_SEPARATOR);
		}
			Refactored the code for the printOrderCommand to make it work with the console write. This was done to print/save logs with observer pattern and it will be responsible to show and save the logs which will be seen in CMD.	

4	private String getCommandFromPlayer() {
	        String l_Command;
	        System.out.println(Constants.
	ISSUE_COMMAND_MESSAGE);
	        Constants.showIssueOrderCommand();
	        l_Command = d_Scanner.nextLine();
	        //Todo add validation
	        if(Objects.equals(l_Command.split(" ")[0],
	Constants.SHOW_MAP)){
	            new ShowMapController(d_GameMap).
	show();
	            return getCommandFromPlayer();
	        }
	        //Todo add validation
	        return l_Command;
	    }
		d_Commands = l_Player.readFromPlayer();
		
		 public String readFromPlayer() {
		        return this.d_PlayerStrategy.createCommand();
		    }
		
		//Human strategy
		@Override
		    public String createCommand() {
		        return SCANNER.nextLine();
		    }
		
		//Aggresive strategy
		public String createCommand() {
		        d_Player = GameMap.getInstance().getCurrentPlayer();
		        d_Logger.log("Issuing Orders for the Aggressive Player - " +
		d_Player.getName());
		        if (d_Player.getCapturedCountries().size() > 0) {
		            createAndOrderCountryList();
		            deployCommand();
		            if (bombOrAttack()) {
		                return Constants.PASS_COMMAND;
		            }
		            moveToSelf();
		        }
		        return Constants.PASS_COMMAND;
		    }
			First the commands were directly being fetched from CMD but as we have different player stratagies then according to the strategy the commands getter logic will be changed. So due to that we have changed the logic of the getting the input from the user according to the strategy.	TournamentModeTest.java

5	public boolean saveMap(){
	if(!new MapValidator().validateMapObject(this.
	d_GameMap)){
	System.out.println("This Map Format is not
	valid");
	return false;
	}else{
	try {
	BufferedWriter l_WriterPointer = new
	BufferedWriter(new FileWriter
	("src/main/resources/maps/"+this.d_FileName +".
	map"));
	int l_Continent_idx = 1;
	int l_Country_idx = 1;
	//These Hashmaps are for creating the
	border indexes
	HashMap<Integer, String>
	l_IndexToCountry = new HashMap<>();
	HashMap<String, Integer>
	l_CountryToIndex = new HashMap<>();
	
	//write basic information
	l_WriterPointer.write("name " + this.
	d_FileName + " Map");
	l_WriterPointer.newLine();
	l_WriterPointer.newLine();
	l_WriterPointer.write("[files]");
	l_WriterPointer.newLine();
	l_WriterPointer.newLine();
	l_WriterPointer.flush();
	
	// Write Continents
	l_WriterPointer.write("[continents]");
	l_WriterPointer.newLine();
	
	try {
	for (Continent l_Continent : this.
	d_GameMap.getContinents().values()) {
	l_WriterPointer.write(l_Continent.
	getContinentId() + " " + l_Continent.
	getControlValue());
	l_WriterPointer.newLine();
	l_WriterPointer.flush();
	l_Continent.setContinentFileIndex
	(String.valueOf(l_Continent_idx));
	l_Continent_idx++;
	}
	l_WriterPointer.newLine();
	}catch (Exception e) {
	System.out.println(e.getMessage());
	}
	// Write Countries
	l_WriterPointer.write("[countries]");
	l_WriterPointer.newLine();
	try {
	for (Country l_Country : this.d_GameMap.
	getCountries().values()) {
	l_WriterPointer.write(l_Country_idx + "
	" + l_Country.getCountryId() + " " + this.d_GameMap.
	getContinents().get(l_Country.getParentContinent().
	toLowerCase()).getContinentFileIndex() + " " + "0" + "
	" + "0");
	l_WriterPointer.newLine();
	l_WriterPointer.flush();
	l_IndexToCountry.put(l_Country_idx,
	l_Country.getCountryId().toLowerCase());
	l_CountryToIndex.put(l_Country.
	getCountryId().toLowerCase(), l_Country_idx);
	l_Country_idx++;
	}
	l_WriterPointer.newLine();
	}catch (Exception e) {
	System.out.println(e.getMessage());
	}
	
	//Write Borders
	l_WriterPointer.write("[borders]");
	l_WriterPointer.newLine();
	l_WriterPointer.flush();
	for(int i=1;i<l_Country_idx;i++) {
	String l_CountryId = l_IndexToCountry.
	get(i);
	try{
	Country l_Cd = this.d_GameMap.
	getCountries().get(l_CountryId.toLowerCase());
	l_WriterPointer.write(Integer.toString(i)
	+ " ");
	for (Country l_Neighbor : l_Cd.
	getNeighbours().values()) {
	l_WriterPointer.write(Integer.toString
	(l_CountryToIndex.get(l_Neighbor.getCountryId().
	toLowerCase())) + " ");
	l_WriterPointer.flush();
	}
	l_WriterPointer.newLine();
	}catch (Exception e) {
	System.out.println(e.getMessage());
	}
	}
	l_WriterPointer.close();
	}catch (IOException e){
	e.printStackTrace();
	return false;
	}
	}
	return true;
	}
		public void saveMap(boolean p_saveAsConquest) throws
		ValidationException, IOException {
		//Ask p_size for minimum number of countries based on
		player
		if (MapValidation.validateMap(d_GameMap, 0)) {
		DominationMap l_SaveMap = p_saveAsConquest ? new
		Adapter(new Adaptee()) : new DominationMap();
		boolean l_Bool = true;
		while (l_Bool) {
		d_GameMap.getName();
		if (Objects.isNull(d_GameMap.getName()) ||
		d_GameMap.getName().isEmpty()) {
		throw new ValidationException("Please enter the file
		name:");
		} else {
		if (l_SaveMap.saveMap(d_GameMap, d_GameMap.
		getName())) {
		d_Logger.log("The map has been validated and is
		saved.");
		} else {
		throw new ValidationException("Map name already
		exists, enter different name.");
		}
		l_Bool = false;
		}
		}
		} else {
		throw new ValidationException("Invalid Map, can not be
		saved.");
		}
		}
			The savemap feature has been completly changed as it is shifted to GameMap class so which ever instance of map is loaded then directly with the help of it's helper method it can be saved in the file. It also has adapter feature to work with both domination and conquest maps.	GameMapTest.java

6	`private boolean executeOrders() { while (!d_PlayerOrderList.isEmpty()) { Order l_PlayerOrder = Player.next_order(); if (!l_PlayerOrder.execute()) { return false; } } return true; }`	`private void executeOrders() { int l_Counter = 0; while (l_Counter < d_GameMap.getPlayers().size()) { l_Counter = 0; for (Player l_Player : d_GameMap.getPlayers().values()) { Order l_Order = l_Player.nextOrder(); if (l_Order == null) { l_Counter++; } else { if (l_Order.execute()) { l_Order.printOrderCommand(); } } } } }`	Increased Clarity and Readability: The refactored version appears to be more explicit in its intent. It's clearer how the orders are being processed and by whom. The loop structure separates the handling of orders for each player, making it easier to understand the flow of execution. Better Handling of Players: Instead of relying on a specific list of orders (d_PlayerOrderList), the refactored version appears to iterate over all players in the game (d_GameMap.getPlayers().values()). This makes the code more flexible, as it can accommodate varying numbers of players without modification. Printing Order Commands: The refactored version introduces a call to l_Order.printOrderCommand() after executing an order. This implies that the code now logs or prints information about the executed orders, which could be useful for debugging or logging purposes. Elimination of the isEmpty() Check: In the original code, the loop condition relies on checking whether d_PlayerOrderList is empty. In the refactored version, this check is eliminated. Instead, the loop continues until l_Counter equals the number of players, which effectively means that all players' orders have been processed. Consistency in Method Naming: In the original code, Player.next_order() is called to get the next order for a player, while in the refactored version, it's called l_Player.nextOrder(). This change may aim to enforce consistency in method naming conventions.	ExecuteOrderTest.java

Refactoring targets	
1	Add adapter pattern for loading the game for domination map and conquest map and refactor the code to minimise duplications.
2	Make all functions name in camel case.
3	Improvise state pattern for GamePhase.
4	Add strategy pattern for the Player's stratagey and refactor the code to minimise duplications.
5	Implement command pattern for All the orders and also include validation and showing what executed by commands in Models rather than in controllers.
6	Removing deadcode, add more understandable comments, change variable name so all variable names after "_" should be in capital. (e.g. d_logger changed to d_Logger)
7	Refactoring according to tournament mode and single player game.
8	Add more information in CMD for the players.
9	Refactoring test cases with Suit and also using singleton for map logic.
10	IssueOrderController was waiting for all the players to deploy their army but now it will be available until all the countries get captured.
11	Reduce if else statements and add more switch statements with ENUMS and use those ENUMS across the code
12	Added code for the Console log
13	Change data structures from Array to ArrayDeque for more improvised logic of Queue
14	Improvised observer pattern for logging, error handling and further exceptional handling.
15	Refactored showmap feature.