Submitted by:	
Marker:	

Notes	%mark	ratio	letter		
	0%	**100.00%**		**1**	**Part 1 : Commandprocessor and command processor adapter**
	0%	10.00%		**1.1**	**Knowledge/Correctness of Game Rules**
	0%	0.00%	X	1.1.1	Students are fully aware of the correct Warzone game rules to implement during the presentation
	0%	0.00%	X	1.1.2	Code is implementing game mechanics that is fully according the Warzone game
	0%	60.00%		**1.2**	**Compliance of solution with Stated Problem**
	0%	0.00%	X	1.2.1	Upon starting the application, a command line option is set to either read commands from the console or from a given file.
	0%	0.00%	X	1.2.2	Commands can be read from the console
	0%	0.00%	X	1.2.3	Commands can be read from a file
	0%	0.00%	X	1.2.4	Commands are save in a list of commands in the command processor
	0%	0.00%	X	1.2.5	Commands can be validated by the command processor
	0%	0.00%	X	1.2.6	All data members of user-defined class type are of pointer type.
	0%	0.00%	X	1.2.7	Classes declared in header file. Functions implemented in cpp file. Absence of inline functions
	0%	0.00%	X	1.2.8	All classes implement a correct copy constructor, assignment operator, and stream insertion operator.
	0%	0.00%	X	1.2.9	Absence of memory leaks.
	0%	0.00%	X	1.2.10	Driver demonstrates that commands can be read from the console using the **CommandProcessor** class.
	0%	0.00%	X	1.2.11	Driver demonstrates that commands can be read from a saved text file using the **FileCommandProcessorAdapter** class.
	0%	0.00%	X	1.2.12	Driver demonstrates that commands that commands that are invalid in the current game state are rejected and valid commands result in the correct effect and state change.
	0%	10.00%		**1.3**	**Modularity of Solution**
	0%	0.00%	x	1.3.1	All is implemented in the file duo named **CommandProcessing.cpp/CommandProcessing.h,** and no other files.
	0%	0.00%	x	1.3.2	Presence of classes named **CommandProcessor**, **Command**, **FileCommandProcessorAdapter**, and **FileLineReader** organized according to the adapter design pattern as depicted on the assignment handout
	0%	0.00%	x	1.3.3	The public **CommandProcessor::getCommand()** method can be used by the GameEngine to read and save a command when needed.
	0%	0.00%	x	1.3.4	The protected **CommandProcessor::readCommand()** method reads a string from the console.
	0%	0.00%	x	1.3.5	The protected **CommandProcessor::saveCommand()** method saves the command in the list of commands.
	0%	0.00%	x	1.3.6	The **Command::saveEffect()** method can be used to save the effect of the command as a string.
	0%	0.00%	x	1.3.7	The **CommandProcessor::validate()** command can be used to validate if a given command is valid in the current game state.
	0%	10.00%		**1.4**	**Mastery of Language/Tools/Libraries**
	0%	0.00%	X	1.4.1	The program never crashed during the demonstration or code review
	0%	0.00%	X	1.4.2	Students were very clear in technical discussions during the demonstration
	0%	10.00%		**1.5**	**Code readability: name conventions, clarity of code, use of comments**
	0%	0.00%	x	1.5.1	All user-defined classes, methods, free functions, and operators are documented
	0%	0.00%	x	1.5.2	Absence of commented-out code
	0%	**100.00%**		**2**	**Part 2 : Game startup phase**
	0%	10.00%		**2.1**	**Knowledge/Correctness of Game Rules**
	0%	0.00%	X	2.1.1	Students are fully aware of the correct Warzone game rules to implement during the presentation
	0%	0.00%	X	2.1.2	Code is implementing game mechanics that is fully according the Warzone game
	0%	60.00%		**2.2**	**Compliance of solution with Stated Problem**
	0%	0.00%	X	2.2.1	In the start state, the loadmap command results in successfully loading a readable map, transitioning to the maploaded state
	0%	0.00%	X	2.2.2	In the maploaded state, the validatemap command is used to validate the map. If successful, the game transitions to the mapValidated state
	0%	0.00%	X	2.2.3	In the mapValidated state, the addplayer command can be used to create new players and insert them in the game (2-6 players).
	0%	0.00%	X	2.2.4	In the mapValidated state, the gamestart command results in fairly distributing the territories among all players
	0%	0.00%	X	2.2.5	In the mapValidated state, the gamestart command results in randomly determine the order of play of the players in the game
	0%	0.00%	X	2.2.6	In the mapValidated state, the gamestart command results in giving 50 initial armies to each player
	0%	0.00%	X	2.2.7	In the mapValidated state, the gamestart command results in each player to draw 2 cards each from the deck
	0%	0.00%	X	2.2.8	In the mapValidated state, the gamestart command results transiting to the play phase
	0%	0.00%	X	2.2.9	Invalid commands for the current state are rejected.
	0%	0.00%	X	2.2.10	All data members of user-defined class type are of pointer type.
	0%	0.00%	X	2.2.11	Classes declared in header file. Functions implemented in cpp file. Absence of inline functions
	0%	0.00%	X	2.2.12	All classes implement a correct copy constructor, assignment operator, and stream insertion operator.
	0%	0.00%	X	2.2.13	Absence of memory leaks.
	0%	0.00%	X	2.2.14	Driver clearly demonstrates the effect of the loadmap command (see 2.2.1)
	0%	0.00%	X	2.2.15	Driver clearly demonstrates the effect of the validatemap command (see 2.2.2)
	0%	0.00%	X	2.2.16	Driver clearly demonstrates the effect of the addplayer command (see 2.2.3)
	0%	0.00%	X	2.2.17	Driver clearly demonstrates the effect of the gamestart command (2.2.4 to 2.2.8)
	0%	10.00%		**2.3**	**Modularity of Solution**
	0%	0.00%	x	2.3.1	All is implemented in the file duo named **GameEngine.cpp/GameEngine.h,** and no other files.
	0%	0.00%	x	2.3.2	Presence of a **GameEngine::startupPhase()** that implements the whole startup phase as described in the assignment handout
	0%	10.00%		**2.4**	**Mastery of Language/Tools/Libraries**
	0%	0.00%	X	2.4.1	The program never crashed during the demonstration or code review
	0%	0.00%	X	2.4.2	Students were very clear in technical discussions during the demonstration
	0%	10.00%		**2.5**	**Code readability: name conventions, clarity of code, use of comments**
	0%	0.00%	x	2.5.1	All user-defined classes, methods, free functions, and operators are documented
	0%	0.00%	x	2.5.2	Absence of commented-out code
	0%	**100.00%**		**3**	**Part 3 : Game play: main game loop**
	0%	10.00%		**3.1**	**Knowledge/Correctness of Game Rules**
	0%	0.00%	X	3.1.1	Students are fully aware of the correct Warzone game rules to implement during the presentation
	0%	0.00%	X	3.1.2	Code is implementing game mechanics that is fully according the Warzone game
	0%	60.00%		**3.2**	**Compliance of solution with Stated Problem**
	0%	0.00%	X	3.2.1	Players get the correct amount of reinforcement during te reinforcement phase during game play, after which the issue orders phase starts.
	0%	0.00%	X	3.2.2	Each player's issueOrder() method is called in round-robin fachion during the issue orders phase.
	0%	0.00%	X	3.2.3	After all players have signified that they dont have any more orders to isse, the orders execution phases starts.
	0%	0.00%	X	3.2.4	A player can create any kind of order, inlcuding those that can only be created using cards.
	0%	0.00%	X	3.2.5	The game engine gets the top order from the list of each players orders list in a round-robin fashion and executes them one by one.
	0%	0.00%	X	3.2.6	Once all orders have been executed, the game engine goes back to the reinforcement phase.
	0%	0.00%	X	3.2.7	If during order execution one player controls all territories, the game goes to the win state, after which the replay command would put the game back into the start state, or the quit command to stop the application.
	0%	0.00%	X	3.2.8	All data members of user-defined class type are of pointer type.
	0%	0.00%	X	3.2.9	Classes declared in header file. Functions implemented in cpp file. Absence of inline functions
	0%	0.00%	X	3.2.10	All classes implement a correct copy constructor, assignment operator, and stream insertion operator.
	0%	0.00%	X	3.2.11	Absence of memory leaks.
	0%	0.00%	X	3.2.12	Driver clearly demonstrates that a player receives the correct number of armies in the reinforcement phase (showing different cases)
	0%	0.00%	X	3.2.13	Driver clearly demonstrates that a player will only issue deploy orders and no other kind of orders if they stil have armies in their reinforcement pool
	0%	0.00%	X	3.2.14	Driver clearly demonstrates that a player can issue advance orders to either defend or attack, based on the toAttack() and toDefend() lists
	0%	0.00%	X	3.2.15	Driver clearly demonstrates that a player can play cards to issue orders
	0%	0.00%	X	3.2.16	Driver clearly demonstrates that a player that does not control any territory is removed from the game
	0%	0.00%	X	3.2.17	Driver clearly demonstrates that the game ends when a single player controls all the territories
	0%	10.00%		**3.3**	**Modularity of Solution**
	0%	0.00%	x	3.3.1	All is implemented in the file duo named **GameEngine.cpp/GameEngine.h,** and no other files.
	0%	0.00%	x	3.3.2	Presence of a **GameEngine::reinforcementPhase()** that implements the whole reinfrocement phase as described in the assignment handout
	0%	0.00%	x	3.3.3	Presence of a **GameEngine::issueOrdersPhase()** that implements the whole order issuing phase as described in the assignment handout
	0%	0.00%	x	3.3.4	Presence of a **GameEngine::executeOrdersPhase()** that implements the entire orderexecution phase as described in the assignment handout
	0%	10.00%		**3.4**	**Mastery of Language/Tools/Libraries**
	0%	0.00%	X	3.4.1	The program never crashed during the demonstration or code review
	0%	0.00%	X	3.4.2	Students were very clear in technical discussions during the demonstration
	0%	10.00%		**3.5**	**Code readability: name conventions, clarity of code, use of comments**
	0%	0.00%	x	3.5.1	All user-defined classes, methods, free functions, and operators are documented
	0%	0.00%	x	3.5.2	Absence of commented-out code
	0%	**100.00%**		**4**	**Part 4 : Order execution imlementation**
	0%	10.00%		**4.1**	**Knowledge/Correctness of Game Rules**
	0%	0.00%	X	4.1.1	Students are fully aware of the correct Warzone game rules to implement during the presentation
	0%	0.00%	X	4.1.2	Code is implementing game mechanics that is fully according the Warzone game
	0%	60.00%		**4.2**	**Compliance of solution with Stated Problem**
	0%	0.00%	X	4.2.1	All orders are implemented.
	0%	0.00%	X	4.2.2	All orders are properly validated.
	0%	0.00%	X	4.2.3	All orders execution result in the correct effect.
	0%	0.00%	X	4.2.4	All data meimbers of user-defined class type are of pointer type.
	0%	0.00%	X	4.2.5	Classes declared in header file. Functions implemented in cpp file. Absence of inline functions
	0%	0.00%	X	4.2.6	All classes implement a correct copy constructor, assignment operator, and stream insertion operator.
	0%	0.00%	X	4.2.7	Absence of memory leaks.
	0%	0.00%	X	4.2.8	Driver clearly demonstrates that each order is validated before being executed.
	0%	0.00%	X	4.2.9	Driver clearly demonstrates that ownership of a territory is transferred to the attacking player if a territory is conquered as a result of an advance order.
	0%	0.00%	X	4.2.10	Driver clearly demonstrates that one card is given to a player if they conquer at least one territory in a turn (not more than one card per turn).
	0%	0.00%	X	4.2.11	Driver clearly demonstrates that the negotiate order prevents attacks between the two players involved
	0%	0.00%	X	4.2.12	Driver clearly demonstrates that the blockade order transfers ownership to the Neutral player
	0%	0.00%	X	4.2.13	Driver clearly demonstrates that all the orders described above can be issued by a player and executed by the game engine.
	0%	10.00%		**4.3**	**Modularity of Solution**
	0%	0.00%	X	4.3.1	All is implemented in the file duo **Orderch/Orders.cpp**
	0%	0.00%	X	4.3.2	Presence of a **Order::execute()** pure virtual method, and an overridden **execute()** method in every Order subclass (**Deploy, Advance, Airlift, Bomb, Blockade, Negotiate**) that contains all the necessary information to execute the order.
	0%	10.00%		**4.4**	**Mastery of Language/Tools/Libraries**
	0%	0.00%	X	4.4.1	The program never crashed during the demonstration or code review
	0%	0.00%	X	4.4.2	Students were very clear in technical discussions during the demonstration
	0%	10.00%		**4.5**	**Code readability: name conventions, clarity of code, use of comments**
	0%	0.00%	X	4.5.1	All user-defined classes, methods, free functions, or operators are documented
	0%	0.00%	X	4.5.2	Code is clear and there is zero presence of commented-out code
	0%	**100.00%**		**5**	**Part 5 : Game log observer: commands and orders**
	0%	10.00%		**5.1**	**Knowledge/Correctness of Game Rules**
	0%	0.00%	X	5.1.1	Students are fully aware of the correct Warzone game rules to implement during the presentation
	0%	0.00%	X	5.1.2	Code is implementing game mechanics that is fully according the Warzone game
	0%	60.00%		**5.2**	**Compliance of solution with Stated Problem**
	0%	0.00%	X	5.2.1	When a command is read, it is written in the log file. When a command is executed, its effect is written in the log file.
	0%	0.00%	X	5.2.2	When an order is inserted in a player's list of orders, the order is written into the log file.
	0%	0.00%	X	5.2.3	When an order is executed, its effect is written into the log file.
	0%	0.00%	X	5.2.4	When the game engine state changes, the new state is written into the log file.
	0%	0.00%	X	5.2.5	All data members of user-defined class type are of pointer type.
	0%	0.00%	X	5.2.6	Classes declared in header file. Functions implemented in cpp file. Absence of inline functions
	0%	0.00%	X	5.2.7	All classes implement a correct copy constructor, assignment operator, and stream insertion operator.
	0%	0.00%	X	5.2.8	Absence of memory leaks.
	0%	0.00%	X	5.2.9	Driver clearly demonstrates that the **Command**, **CommandProcessor**, **Order**, **OrderList**, and **GameEngine** classes are all a subclass of the **Subject** and **ILoggable** classes
	0%	0.00%	X	5.2.10	Driver clearly demonstrates that the **CommandProcessor::saveCommand()**, **Order::execute()**, **Command::saveEffect()**, **OrderList::addOrder()**, and **GameEngine::transition()** methods are effectively using the Observer patterns' Notify(Subject) method to trigger the writing of an entry in the log file
	0%	0.00%	X	5.2.11	Driver clearly demonstrates that when commands are entered on the console or read from a file, the commands are written to the gamelog.txt file, and their effect is written to the log file when the commands are executed
	0%	0.00%	X	5.2.12	Driver clearly demonstrates that when when an order is added to the order list of a player, the game log observer is notified which results in outputting the order to the log file
	0%	0.00%	X	5.2.13	Driver clearly demonstrates that when an order is executed, the game log observer is notified which results in outputting the effect of the order to the log file
	0%	0.00%	X	5.2.14	Driver clearly demonstrates that when the GameEngine changes its state, thegame log observer is notified and the new state is output to the log file.
	0%	10.00%		**5.3**	**Modularity of Solution**
	0%	0.00%	X	5.3.1	The classes **Subject**, **ILoggable**, **Observer**, and **LogObserver** are implemented in a file duo named **LoggingObserver.cpp/LoggingObserver.h**
	0%	0.00%	X	5.3.2	The classes **Order**, **OrderList**, **GameEngine**, **Command**, and **CommandProcessor** are subclasses of the **Subject** class and use the **Subject::notify()** method to signify the **LogObserver** of a state change
	0%	0.00%	x	5.3.3	Presence of GameEngine class
	0%	10.00%		**5.4**	**Mastery of Language/Tools/Libraries**
	0%	0.00%	X	5.4.1	The program never crashed during the demonstration or code review
	0%	0.00%	X	5.4.2	Students were very clear in technical discussions during the demonstration
	0%	10.00%		**5.5**	**Code readability: name conventions, clarity of code, use of comments**
	0%	0.00%	X	5.5.1	All user-defined classes, methods, free functions, or operators are documented
	0%	0.00%	X	5.5.2	Code is clear and there is zero presence of commented-out code