Practical Assessment Task

Phase 2 – Design Document

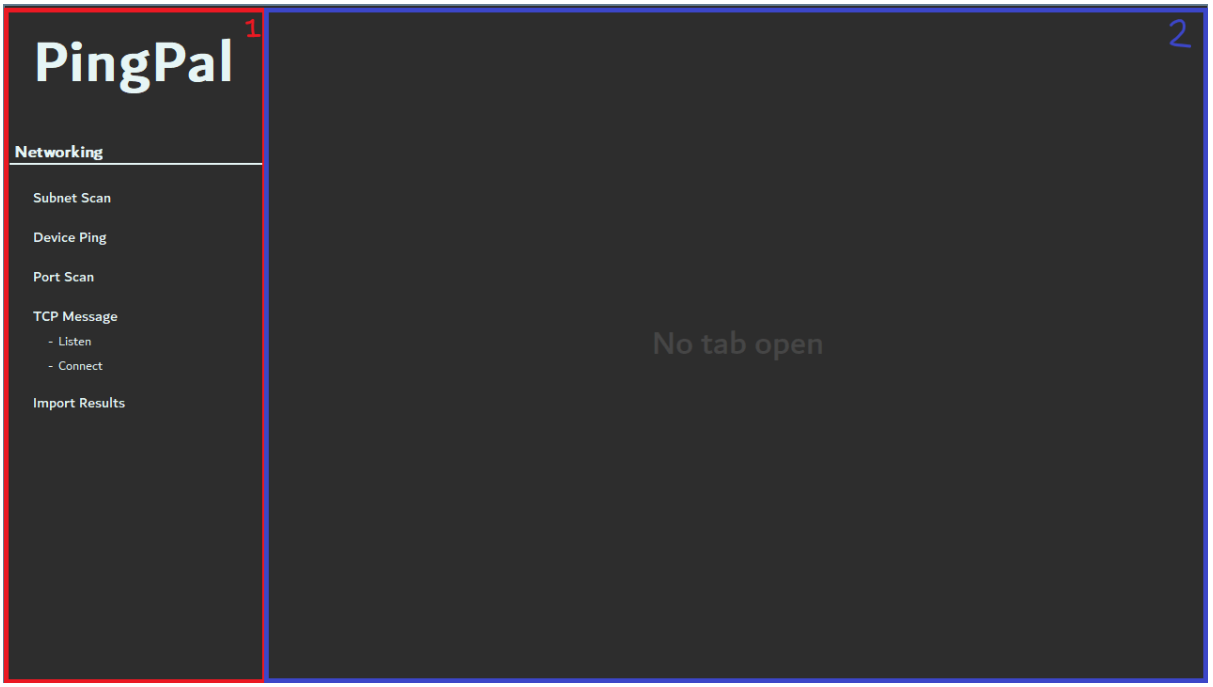
Igor Karbowy

Table of Contents

2.1 — User Interface Design

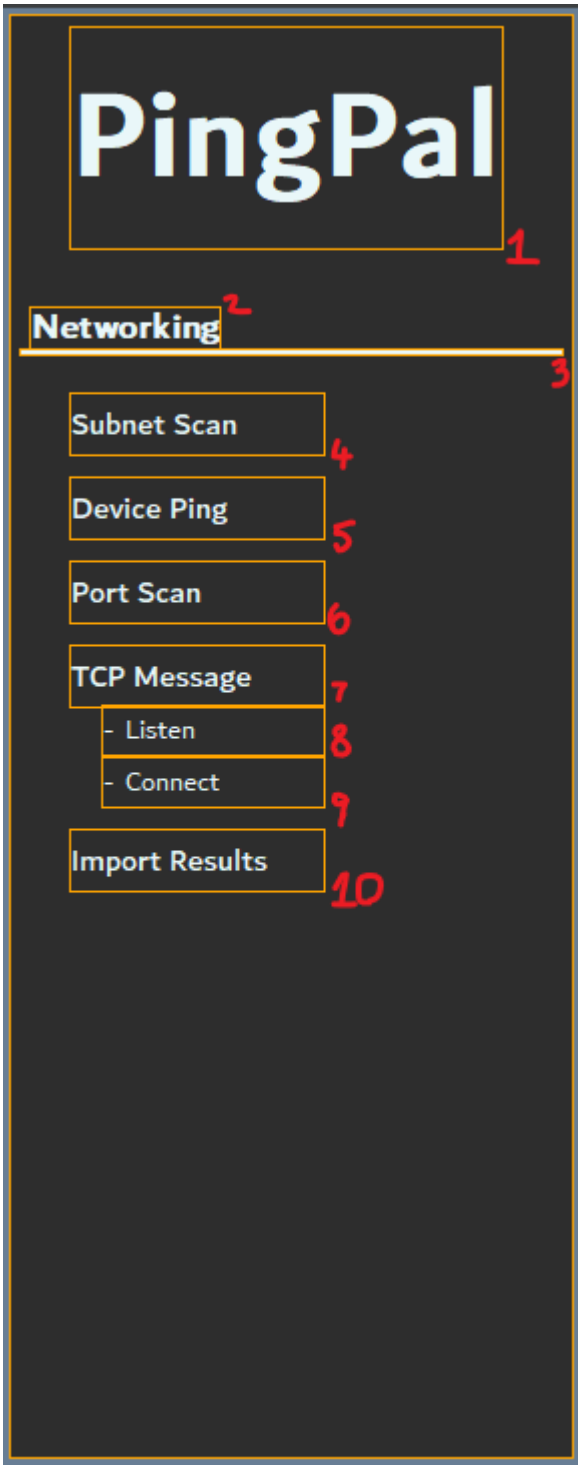
The screens of the graphical user interface (GUI) of *PingPal* include—

A. **Home Page—**



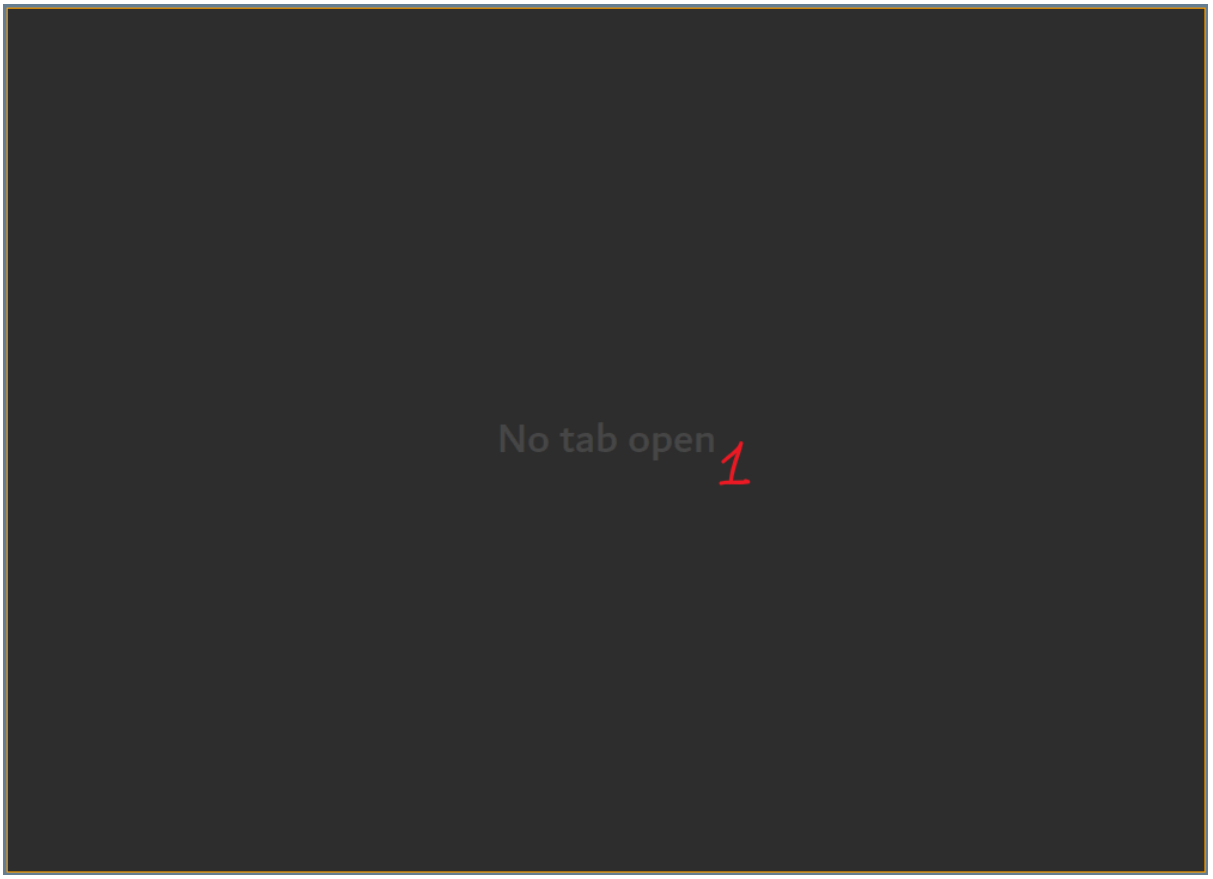
The *Home Page* is split into 2 panels. These panels are—

a. Sidebar Panel (red)—



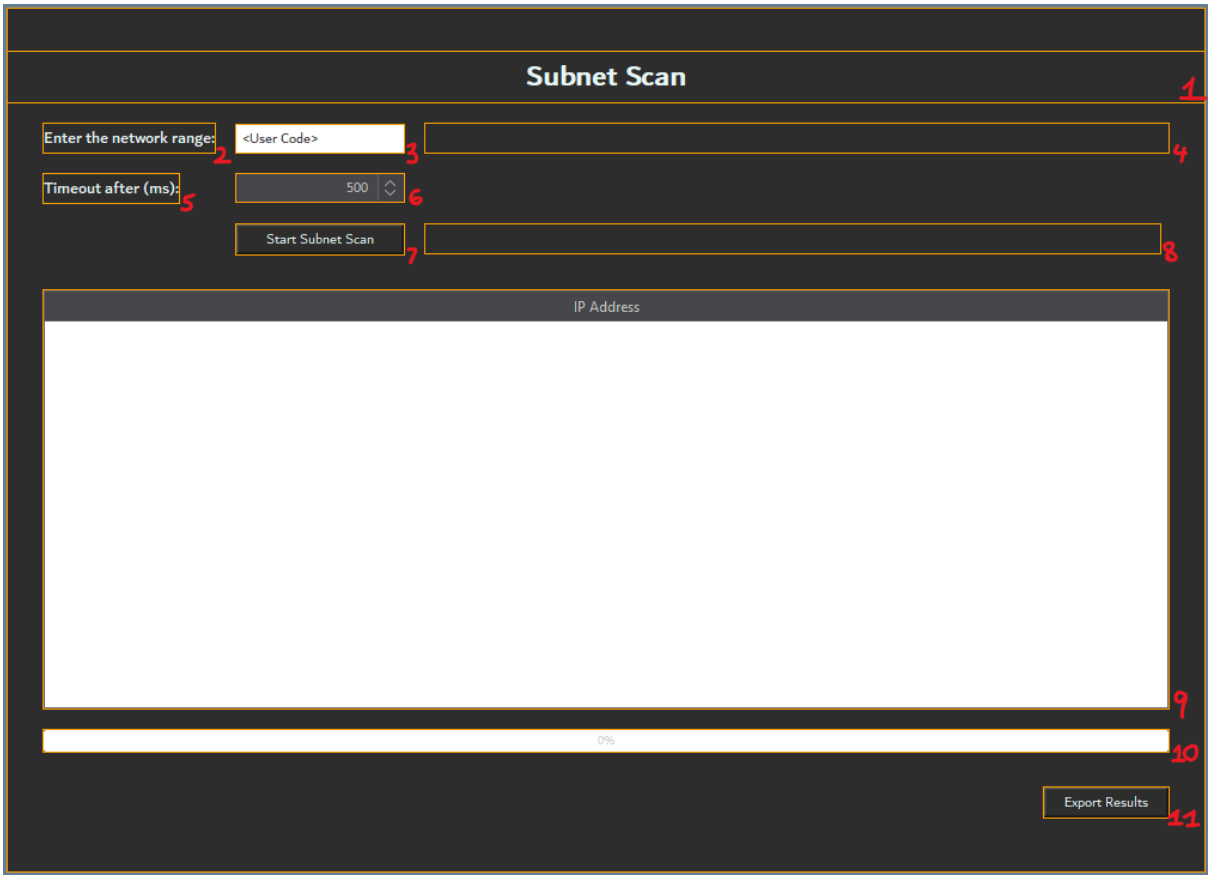
Component No.	Component Type	Component Name	Component Function
1	Label	lblTitle	Displays the name of the application.
2	Label	lblNetworking	Displays the heading for the *Networking* section of the menu.
3	Panel	pnlLine	Used to divide the heading and the list of functions found underneath the heading.
4	Button	btnSubnetScan	Opens and displays the *Subnet Scan* card.
5	Button	btnDevicePing	Opens and displays the *Device Ping* card.
6	Button	btnPortScan	Opens and displays the *Port Scan* card.
7	Button	btnTCPMessage	Initially, opens and displays the *TCP Message Listen* card. Afterwards, it is used to cycle between the *TCP Message Listen* and *TCP Message Connect* cards. Additionally, when a *TCP Message* card loses focus, pressing this button returns the user to the last opened one.
8	Button	btnTCPMessageListen	Opens and displays the *TCP Message Listen* card.
9	Button	btnTCPMessageConnect	Opens and displays the *TCP Message Connect* card.
10	Button	btnImportResults	Opens and displays the *Import Results* pop-up.

b. Main Panel (blue)—



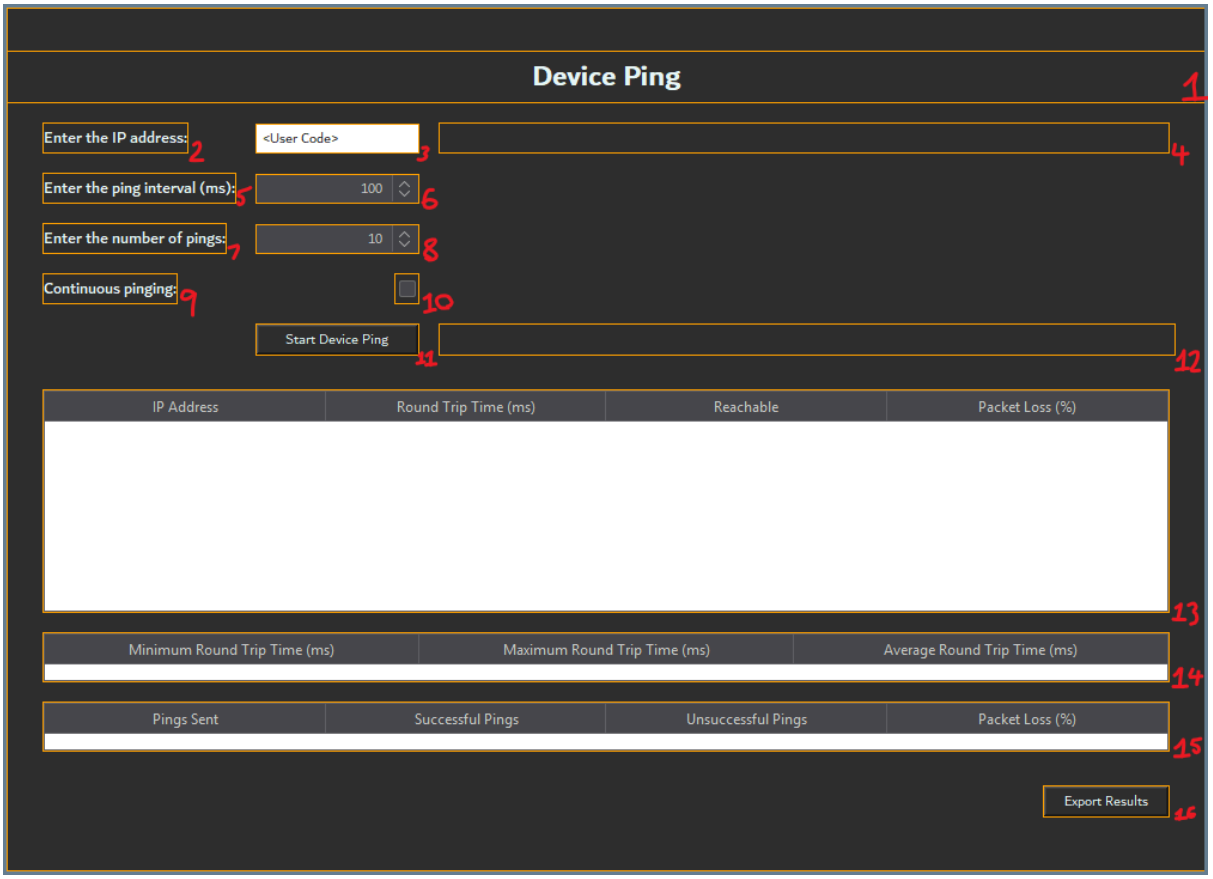
Component No.	Component Type	Component Name	Component Function
1	Label	lblNoTabOpen	Text to inform the user that no tab is open and that no card is being displayed.

B. **Subnet Scan Card—**



Component No.	Component Type	Component Name	Component Function
1	Label	lblSubnetScan	Displays the title of the card (*Subnet Scan*).
2	Label	lblNetworkRange	Text indicating what information (the network range) needs to be inputted by the user into the corresponding text field.
3	Text Field	txfNetworkRange	Field where the user can input network range to scan.
4	Label	lblNetworkRange–Error	Text displayed when the user tries running a subnet scan when the entered network range is invalid. Different errors display different text.
5	Label	lblTimeoutSubnet–Scan	Text indicating what information (the timeout) needs to be inputted by the user into the corresponding spinner.
6	Spinner	spnTimeoutSubnet–Scan	Used to select how long the program should wait before considering an IP address offline, and moving onto scanning the next one.
7	Button	btnStartSubnetScan	Starts the subnet scan, and is used to interrupt the subnet scan once it is running.
8	Label	lblSubnetScanIn–Progress	Text displaying the current status of the subnet scan—i.e. if it is finished, if it is ongoing, if it was interrupted, etc.
9	Table	tblSubnetScan	Table that displays the results of the scan. The data displayed includes the IP addresses of the devices found by the scan. This table is updated immediately when a device is found.
10	Progress Bar	prgSubnetScan	Displays how far along the subnet scan is—i.e. what percentage of devices from the given range have been scanned.
11	Button	btnExportResults–SubnetScan	Exports the results from the scan into a JSON file.

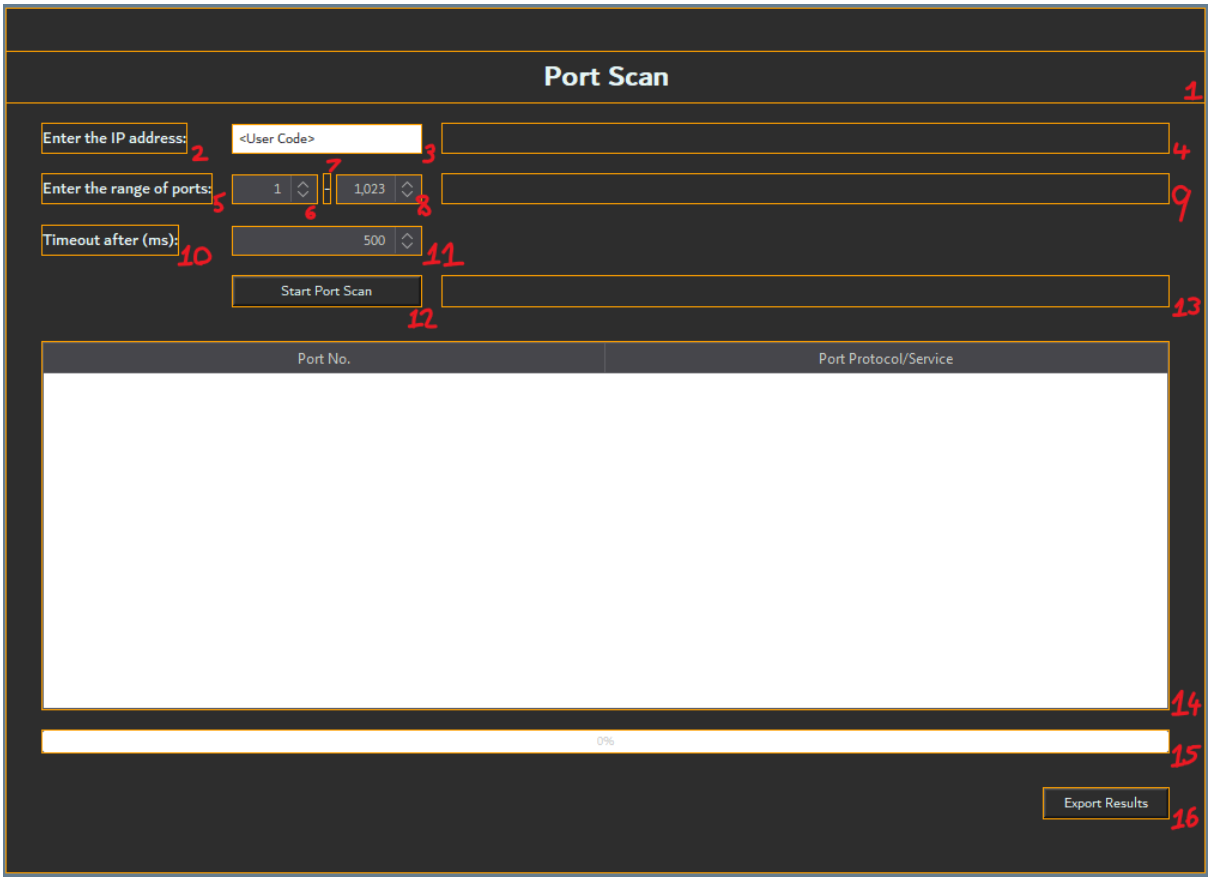
C. **Device Ping Card—**



Component No.	Component Type	Component Name	Component Function
1	Label	lblDevicePing	Displays the title of the card (*Device Ping*).
2	Label	lblIPAddressDevice–Ping	Text indicating what information (the IP address) needs to be inputted by the user into the text field.
3	Text Field	txfIPAddressDevice–Ping	Field where the user can input IP address to ping.
4	Label	lblIPAddressError–DevicePing	Text displayed when the user tries running a device ping when the entered IP address is invalid. Different errors display different text.
5	Label	lblPingInterval	Text indicating what information (the ping interval) needs to be inputted by the user into the spinner.
6	Spinner	spnPingInterval	Used to select the ping interval between each consecutive ping in milliseconds.
7	Label	lblNumberOfPings	Text indicating what information (the number of pings) needs to be inputted by the user into the spinner.
8	Spinner	spnNumberOfPings	Used to select the number of times the device should be pinged.
9	Label	lblContinuous–Pinging	Text indicating that the adjacent checkbox determines whether the pings are continuous or performed for a defined amount of times.
10	Checkbox	chkContinuous–Pinging	Checkbox that determines whether the pings are continuous or performed for a defined amount of times.
11	Button	btnStartDevicePing	Starts the device ping test, and is used to interrupt the device ping once it is running.
12	Label	lblDevicePingIn–Progress	Text displaying the current status of the device ping—i.e. if it is finished, if it is ongoing, if it was interrupted, etc.

13	Table	tblDevicePing	Table that displays the results of the scan as it is ongoing. The data displayed includes the IP address of the device being pinged, the round trip time in milliseconds, whether the device was reached during that ping, and the packet loss percentage.
14	Table	tblDevicePing–ResponsesResults	Table that displays the results of the scan once it has finished/been interrupted. The data displayed includes the minimum round trip time in milliseconds, the maximum round trip time in milliseconds, and the average round trip time in milliseconds.
15	Table	tblDevicePing–PacketResults	Table that displays the results of the scan once it has finished/been interrupted. The data displayed includes the amount of pings that were sent, the amount of successful pings, the amount of unsuccessful pings, and the final packet loss percentage measured.
16	Button	btnExportResults–DevicePing	Exports the results from the scan into a JSON file.

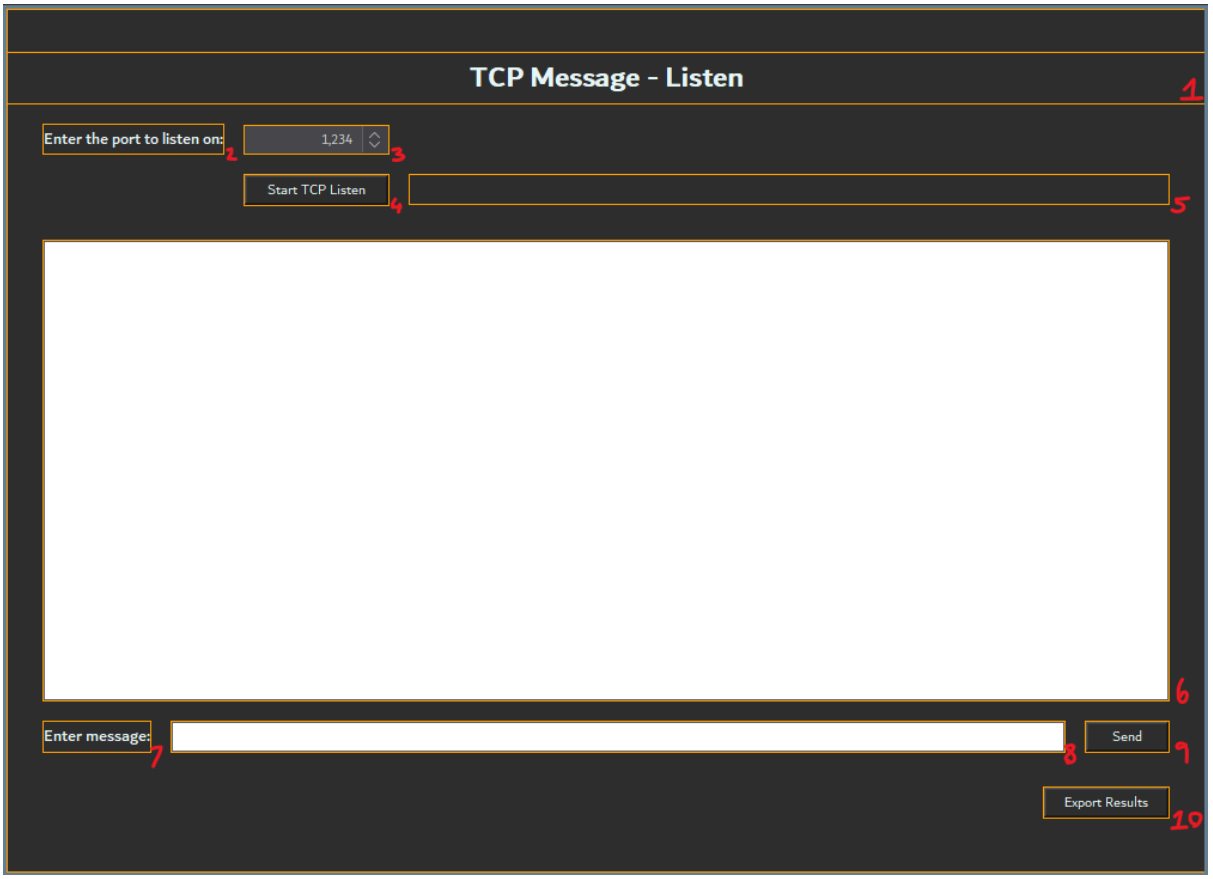
D. **Port Scan Card—**



Component No.	Component Type	Component Name	Component Description
1	Label	lblPortScan	Displays the title of the card (*Port Scan*).
2	Label	lblIPAddressPort–Scan	Text indicating what information (the IP address) needs to be inputted by the user into the text field.
3	Text Field	txfIPAddressPort–Scan	Field where the user can input IP address to scan.
4	Label	lblIPAddressError–PortScan	Text displayed when the user tries running a port scan when the entered IP address is invalid. Different errors display different text.
5	Label	lblPortRange	Text indicating what information (the range of ports) needs to be inputted by the user into the adjacent spinners.
6	Spinner	spnBottomRange–Port	Used to select the port for the bottom of the range—i.e. the port from which the program should start scanning from.
7	Label	lblDash	Text used to create a divider between the two spinners.
8	Spinner	spnTopRangePort	Used to select the port for the top of the range—i.e. the port which the program should scan until.
9	Label	lblPortRangeError	Text displayed when the user tries running a port scan when the entered port range is invalid.
10	Label	lblTimeoutPortScan	Text indicating what information (the timeout) needs to be inputted by the user into the corresponding spinner.
11	Spinner	spnTimeoutPortScan	Used to select how long the program should wait before considering a port closed, and moving onto scanning the next one.
12	Button	btnStartPortScan	Starts the port scan, and is used to interrupt the port scan once it is running.

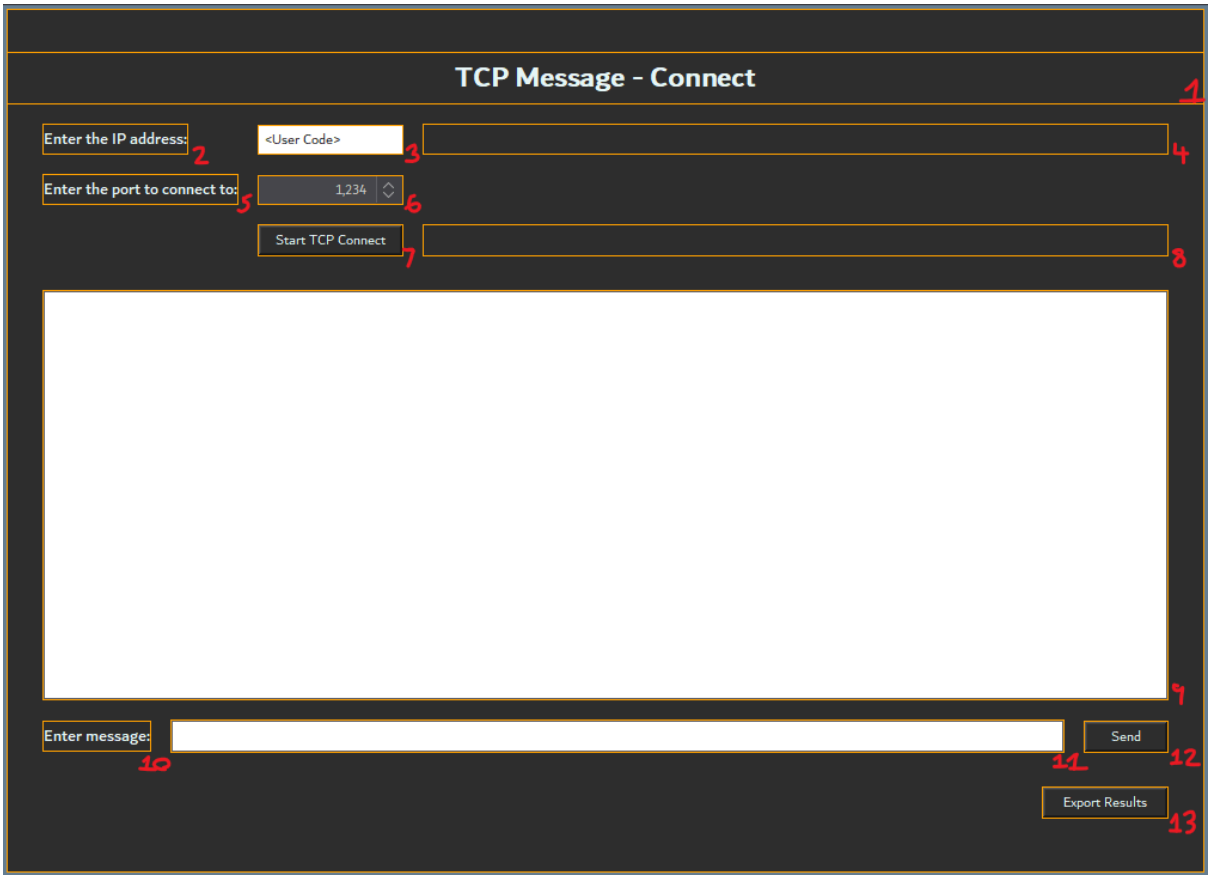
13	Label	lblPortScanIn–Progress	Text displaying the current status of the port scan—i.e. if it is finished, if it is ongoing, if it was interrupted, etc.
14	Table	tblPortScan	Table that displays the results of the scan. The data displayed includes the number of the open port found by the scan, and the associated protocol with that port number. This table is updated immediately when an open port is found.
15	Progress Bar	prgPortScan	Displays how far along the port scan is—i.e. what percentage of ports from the given range have been scanned.
16	Button	btnExportResults–PortScan	Exports the results from the scan into a JSON file.

E. **TCP Message Listen Card—**



Component No.	Component Type	Component Name	Component Description
1	Label	lblTCPMessageListen	Displays the title of the card (*TCP Message - Listen*).
2	Label	lblPortTCPMessage–Listen	Text indicating what information (the port to listen on) needs to be inputted by the user into the spinner.
3	Spinner	spnPortTCPMessage–Listen	Used to select the port to open the connection on—i.e. the port on which the program will connect to another device, receive messages, and send messages.
4	Button	btnStartTCPMessage–Listen	Starts the messaging session, and is used to interrupt the session once it is running.
5	Label	lblTCPMessageListen–InProgress	Text displaying the current status of the messaging session—i.e. if it is finished, if it is ongoing, if it was interrupted, etc.
6	Text Pane	txpTCPMessageListen	Text pane that displays the results of the messaging session. The data displayed includes system messages about the connection, and the messages exchanged during the session.
7	Label	lblEnterMessageTCP–MessageListen	Text indicating what information (the message to be sent) needs to be inputted by the user into the text field.
8	Text Field	txfMessageTCP–MessageListen	Field where the user can input the message to be sent.
9	Button	btnSendTCPMessage–Listen	Sends the message to the connected device, and appends the message to the text pane.
10	Button	btnExportResultsTCP–MessageListen	Exports the results from the messaging session into a text file.

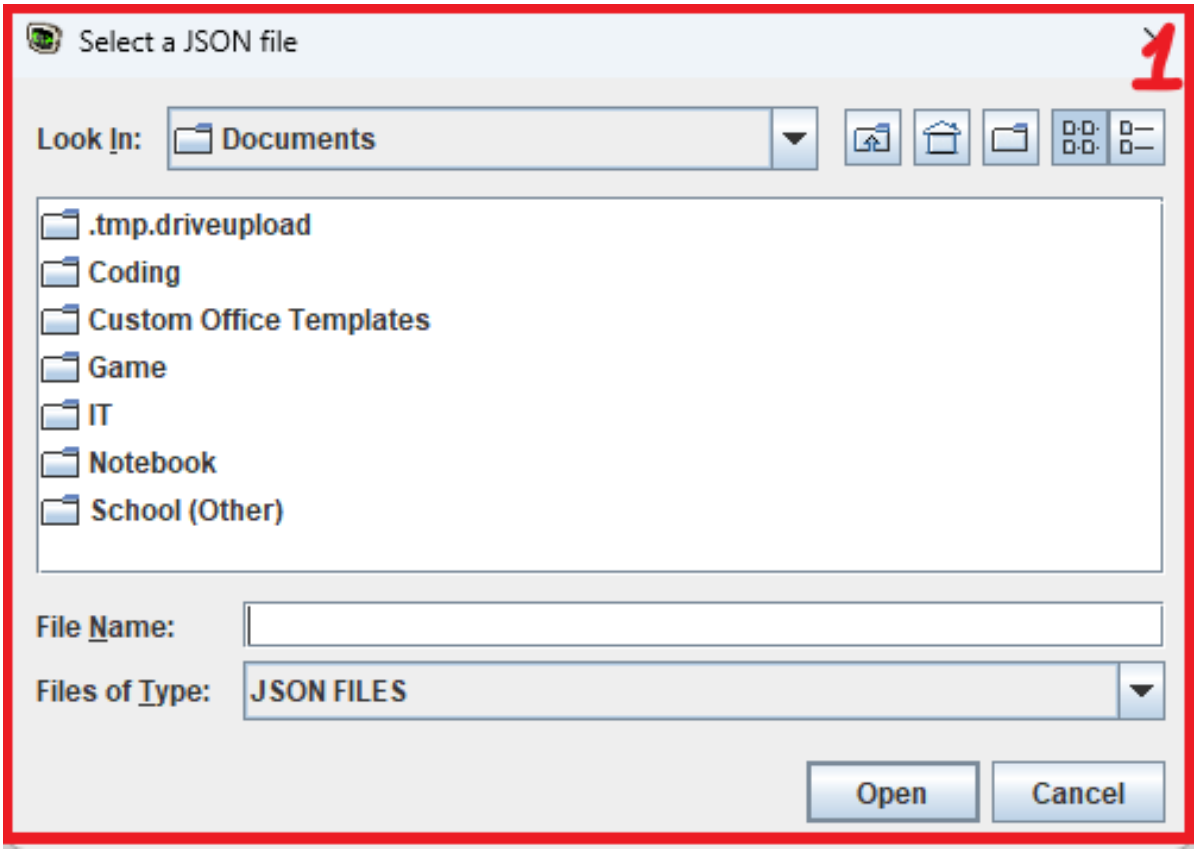
F. **TCP Message Connect Card—**



Component No.	Component Type	Component Name	Component Description
1	Label	lblTCPMessage–Connect	Displays the title of the card (*TCP Message - Connect*).
2	Label	lblIPAddressTCP–MessageConnect	Text indicating what information (the IP address) needs to be inputted by the user into the text field.
3	Text Field	txfIPAddressTCP–MessageConnect	Field where the user can input IP address to attempt to connect to.
4	Label	lblIPAddressError–TCPMessageConnect	Text displayed when the user tries establishing a connection when the entered IP address is invalid. Different errors display different text.
5	Label	lblPortTCPMessage–Connect	Text indicating what information (the port to connect to) needs to be inputted by the user into the spinner.
6	Spinner	spnPortTCPMessage–Connect	Used to select the port to try to establish the connection on—i.e. the port on which the program will connect to another device, receive messages, and send messages.
7	Button	btnStartTCPMessage–Connect	Starts the messaging session, and is used to interrupt the session once it is running.
8	Label	lblTCPMessage–ConnectInProgress	Text displaying the current status of the messaging session—i.e. if it is finished, if it is ongoing, if it was interrupted, etc.
9	Text Pane	txpTCPMessage–Connect	Text pane that displays the results of the messaging session. The data displayed includes system messages about the connection, and the messages exchanged during the session.
10	Label	lblEnterMessage–TCPMessageConnect	Text indicating what information (the message to be sent) needs to be inputted by the user into the text field.

11	Text Field	txfMessageTCP–MessageConnect	Field where the user can input the message to be sent.
12	Button	btnSendTCPMessage–Connect	Sends the message to the connected device, and appends the message to the text pane.
13	Button	btnExportResults–TCPMessageConnect	Exports the results from the messaging session into a text file.

G. **Import Results Pop-Up—**



Component No.	Component Type	Component Name	Component Function
1	File Chooser	fchFileChooser	Allows the user to select the JSON file to parse and read—i.e. import. The file chooser is configured only to accept JSON files.

2.2 — Program Flow Diagram

The flow of the entire program would be too large to fit on a single A4 page. For that reason, the program flow diagram has been broken down into smaller diagrams, namely—

A. **General Overview**

B. **Subnet Scan**

C. **Device Ping**

D. **Port Scan**

E. **TCP Message Listen**

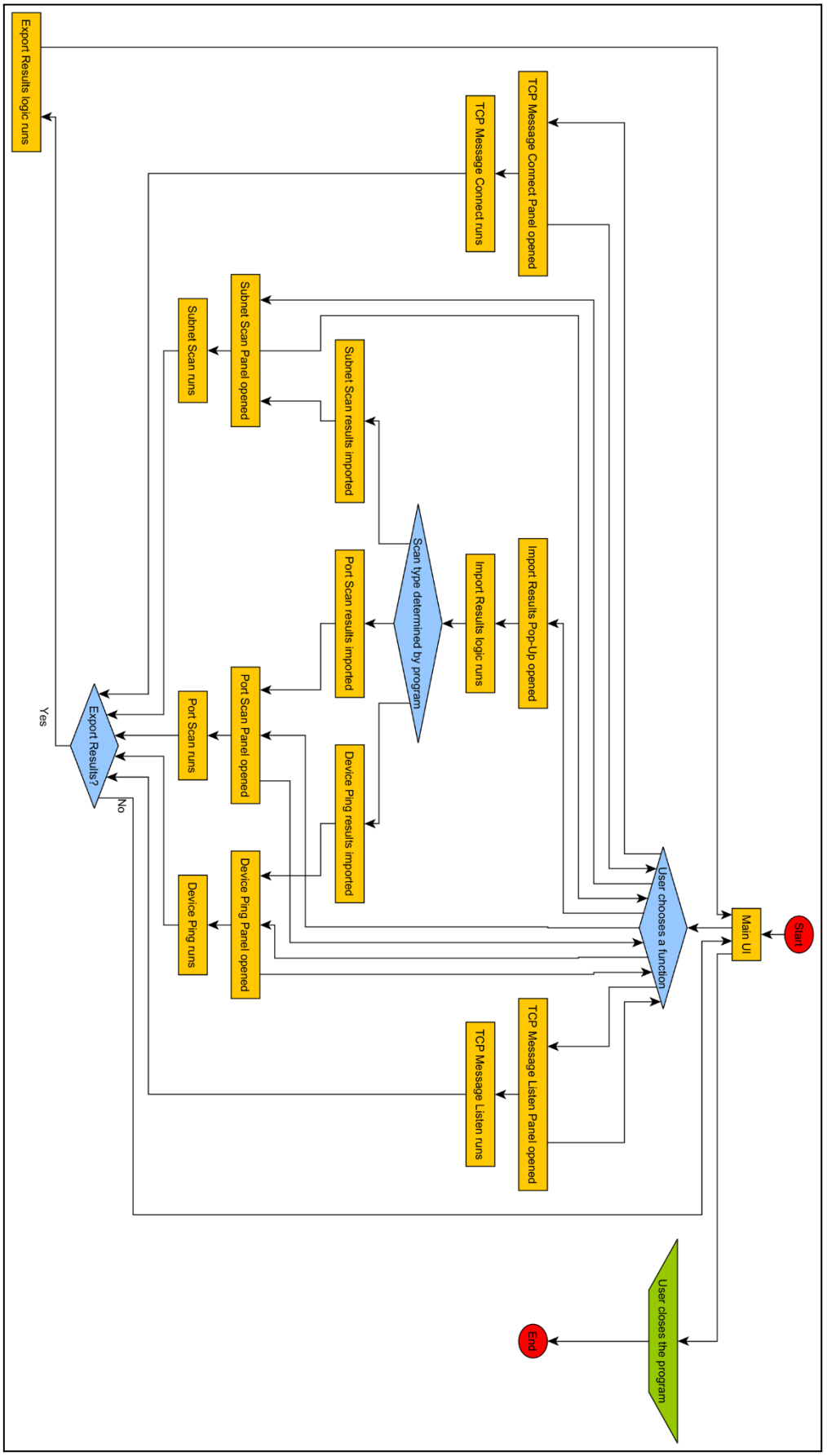
F. **TCP Message Connect**

G. **Export Results**

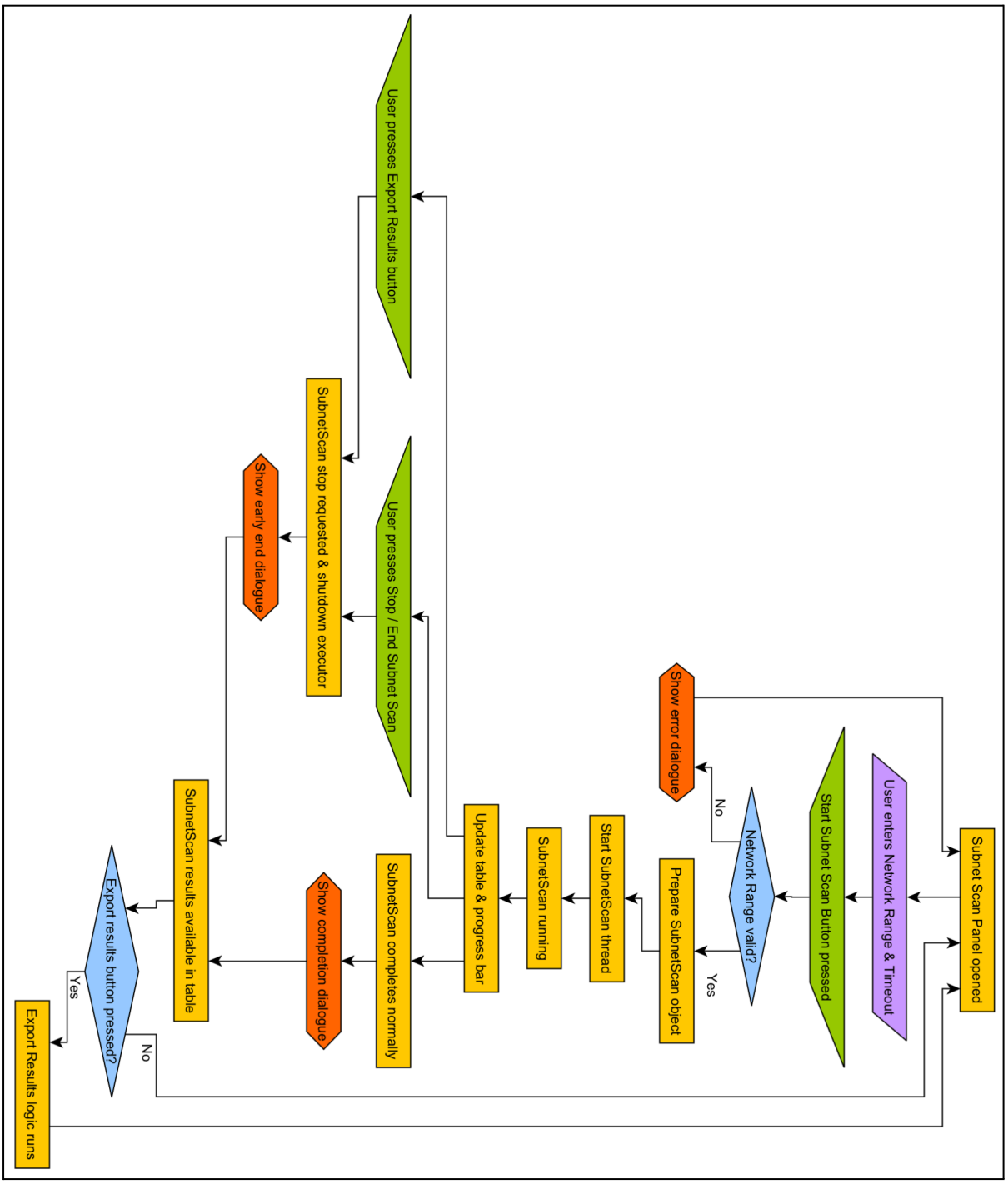
H. **Import Results**

The diagrams can be found on the following pages—

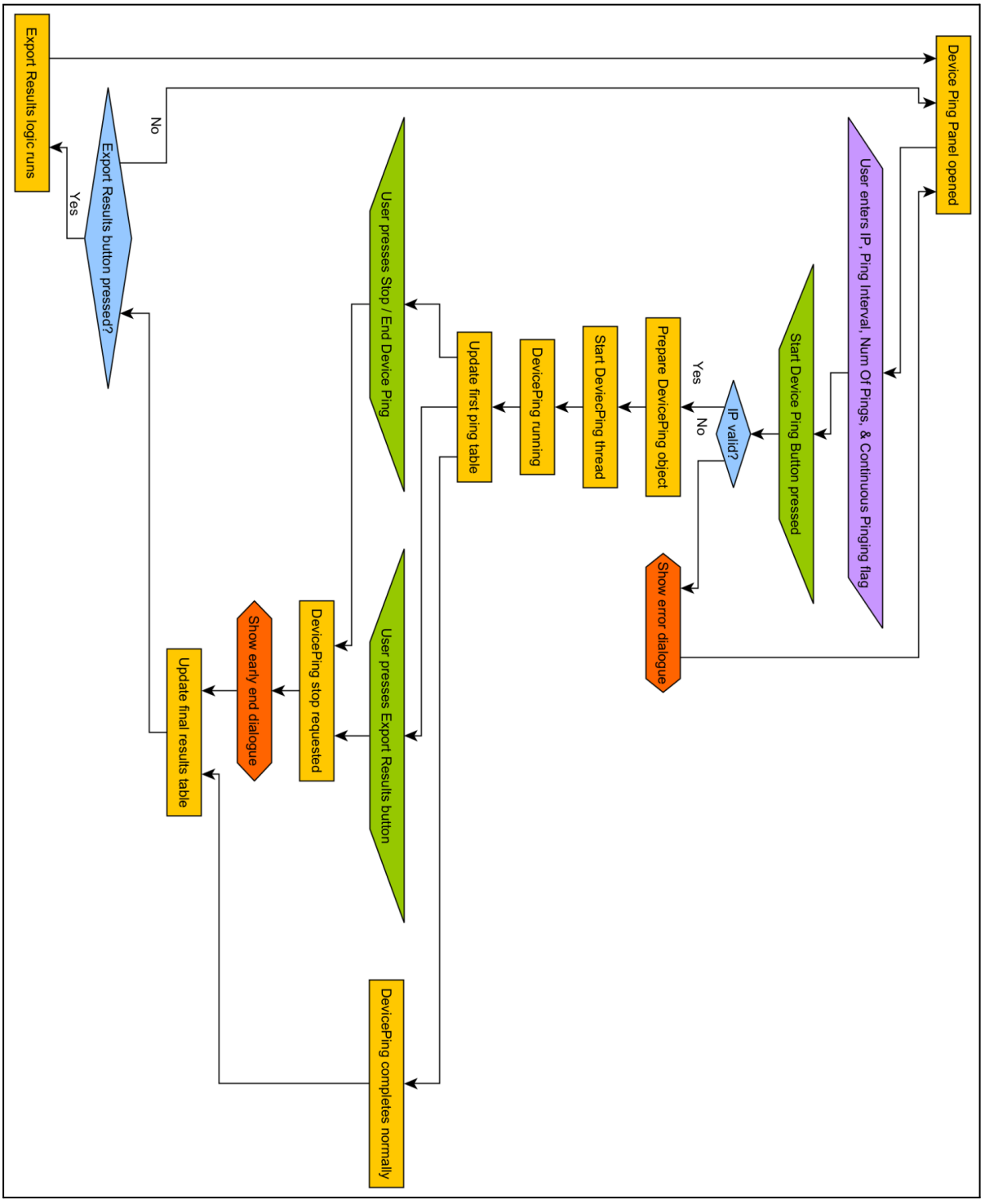
A. **General Overview—**



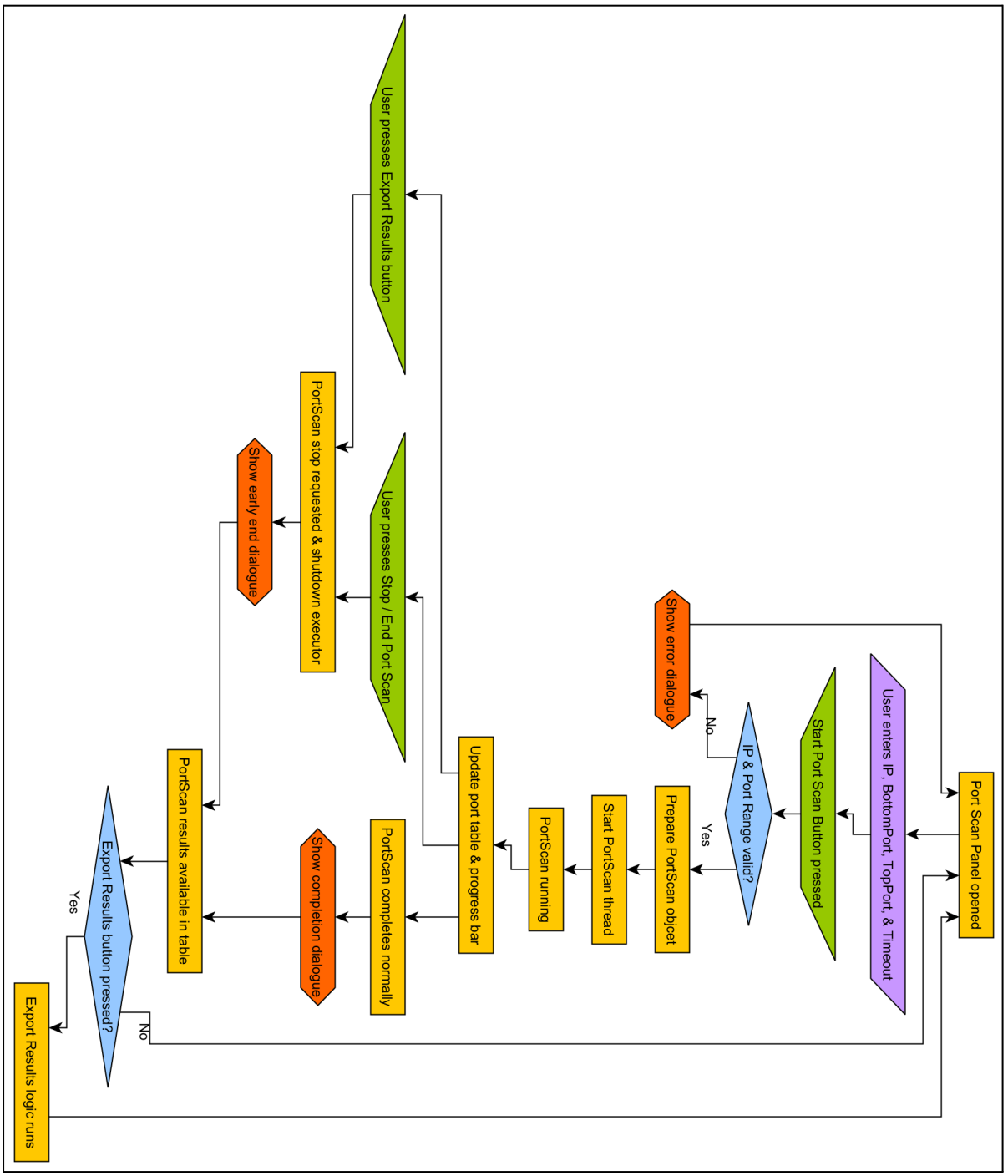
B. **Subnet Scan—**



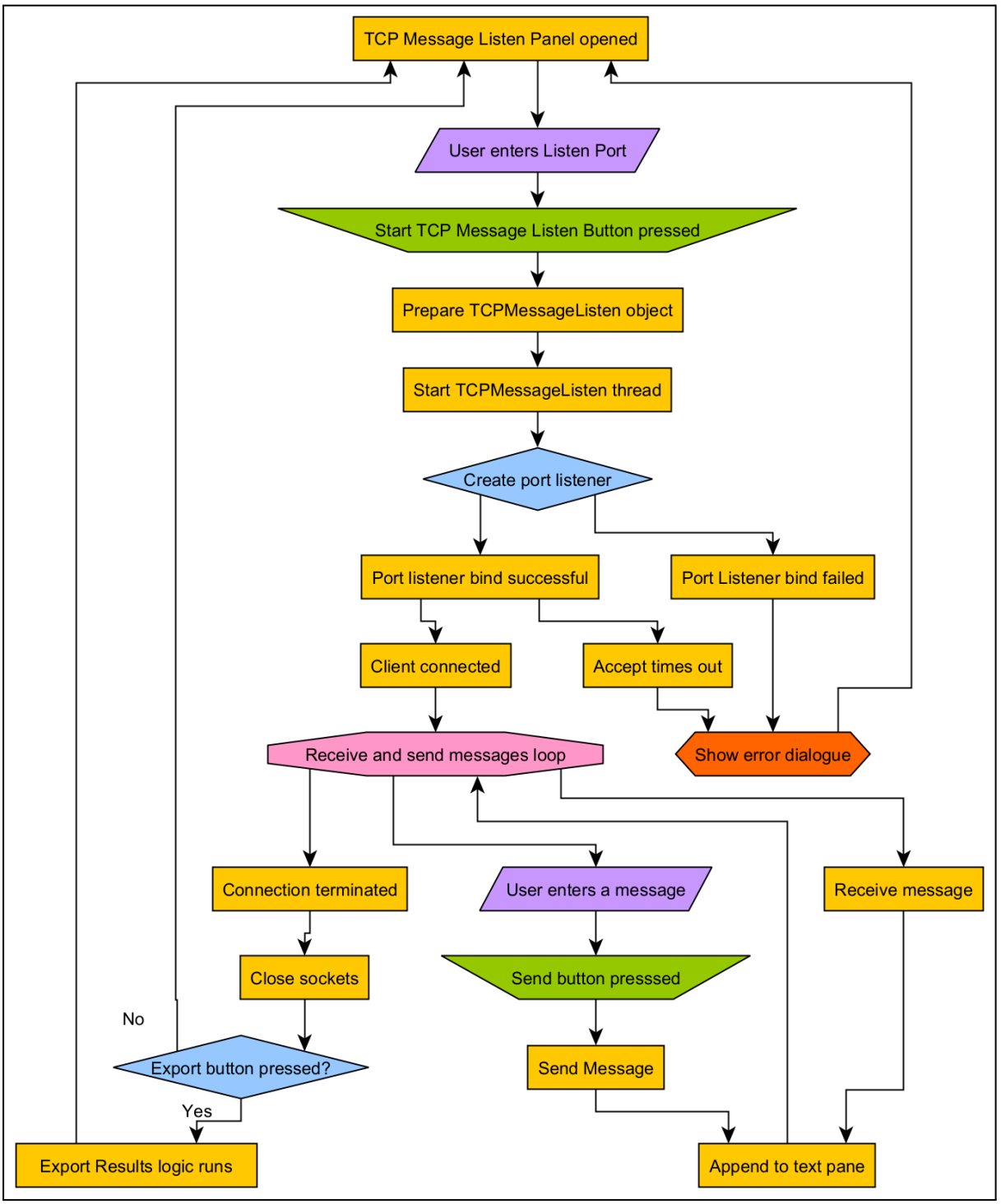
C. **Device Ping—**



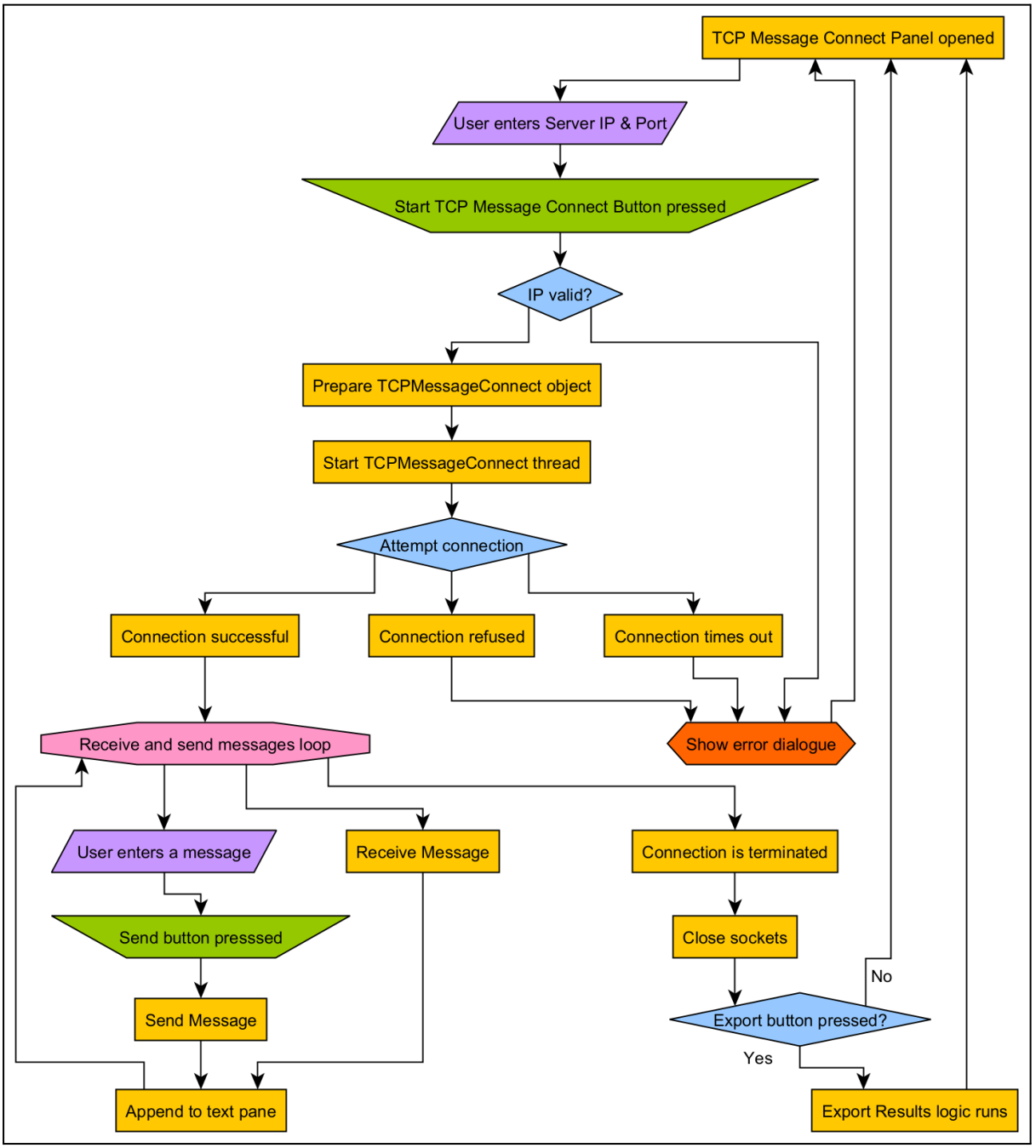
D. **Port Scan—**


Port Scan Panel opened
User enters IP, BottomPort, TopPort, & Timeout
Start Port Scan Button pressed
IP & Port Range valid?
No
Show error dialogue
Yes
Prepare PortScan object
Start PortScan thread
PortScan running
Update port table & progress bar
PortScan completes normally
Show completion dialogue
PortScan results available in table
Export Results button pressed?
Yes
No
Export Results logic runs
User presses Stop / End Port Scan
PortScan stop requested & shutdown executor
Show early end dialogue
User presses Export Results button

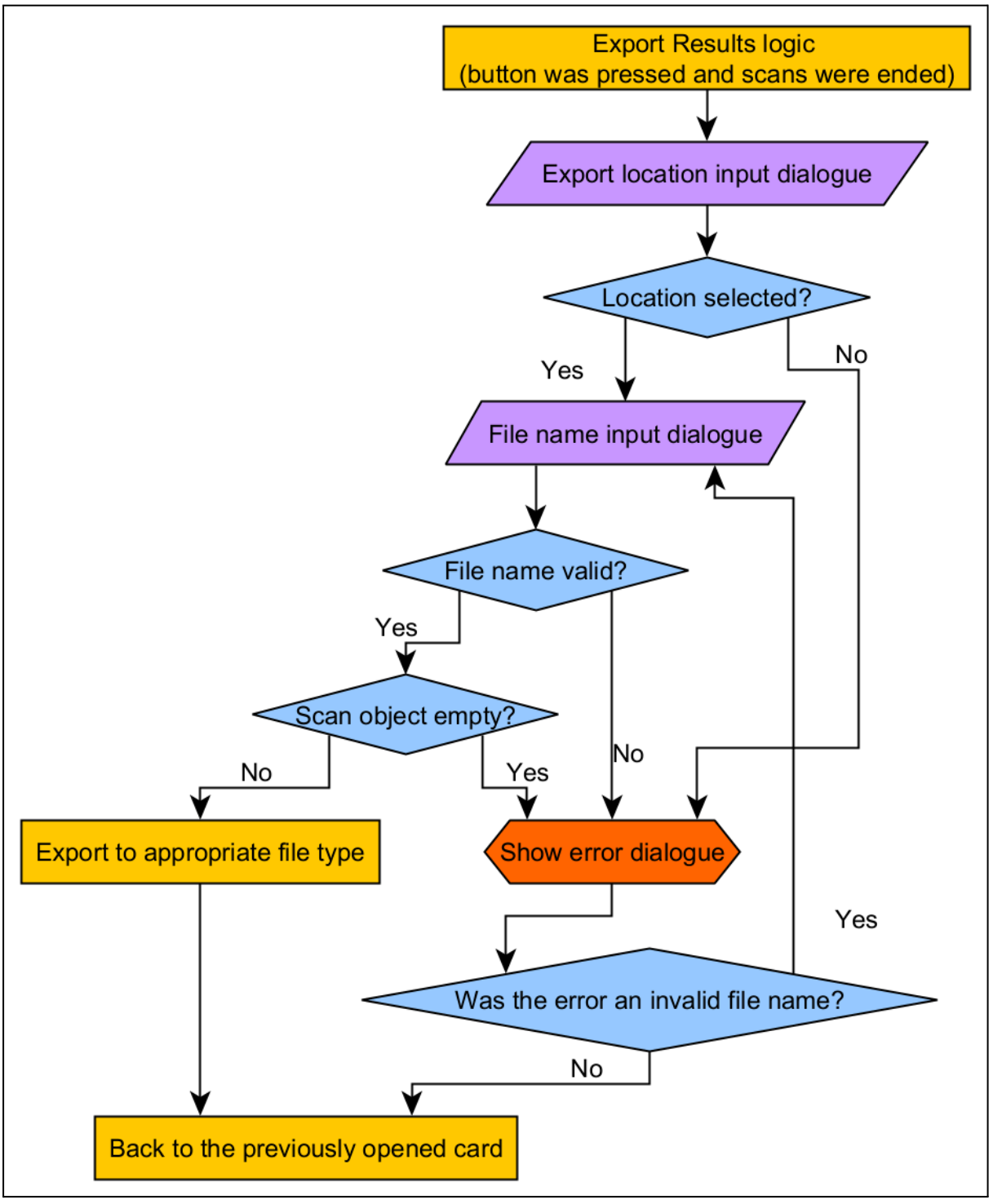
E. **TCP Message Listen—**



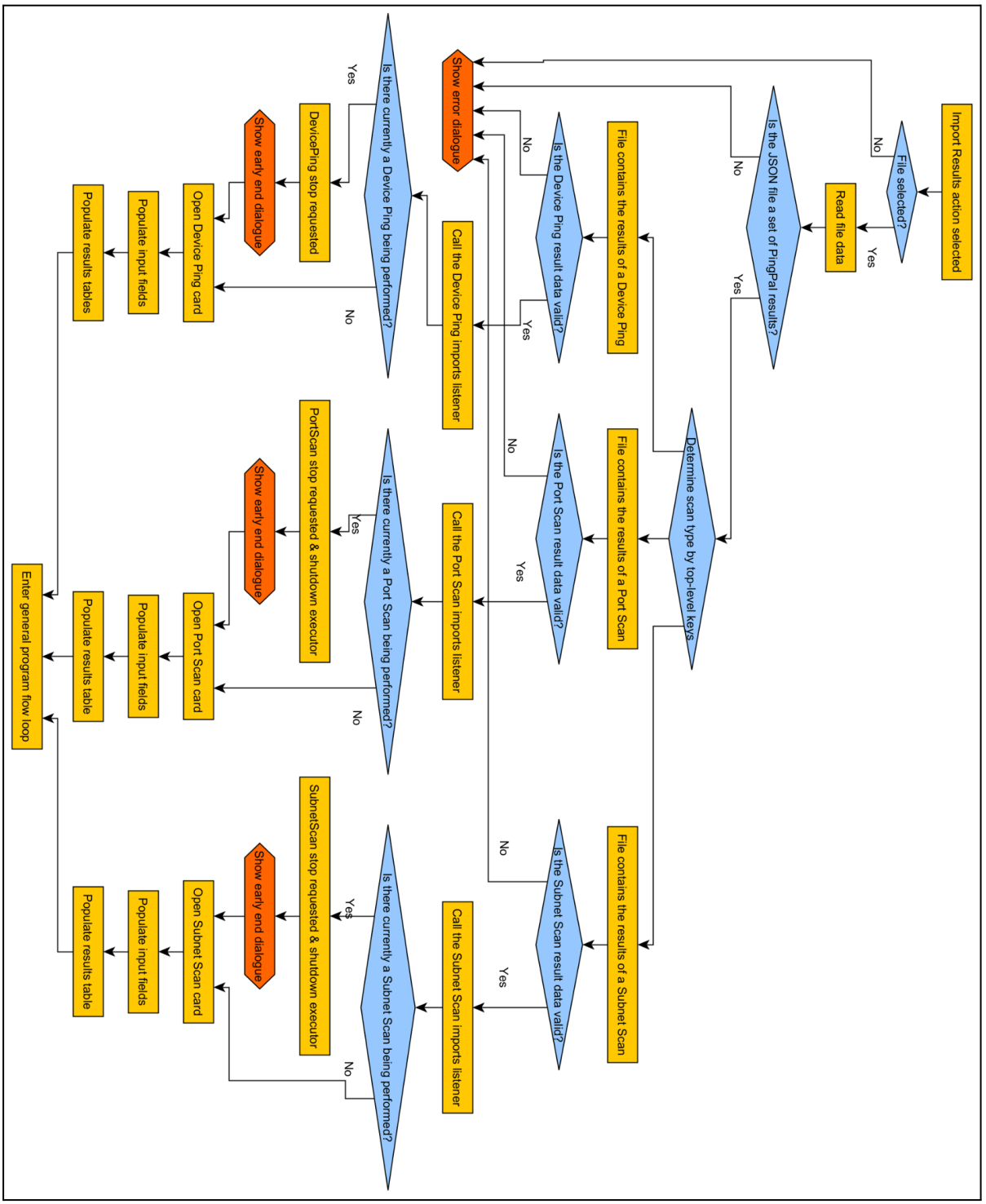
F. **TCP Message Connect—**



G. **Export Results—**

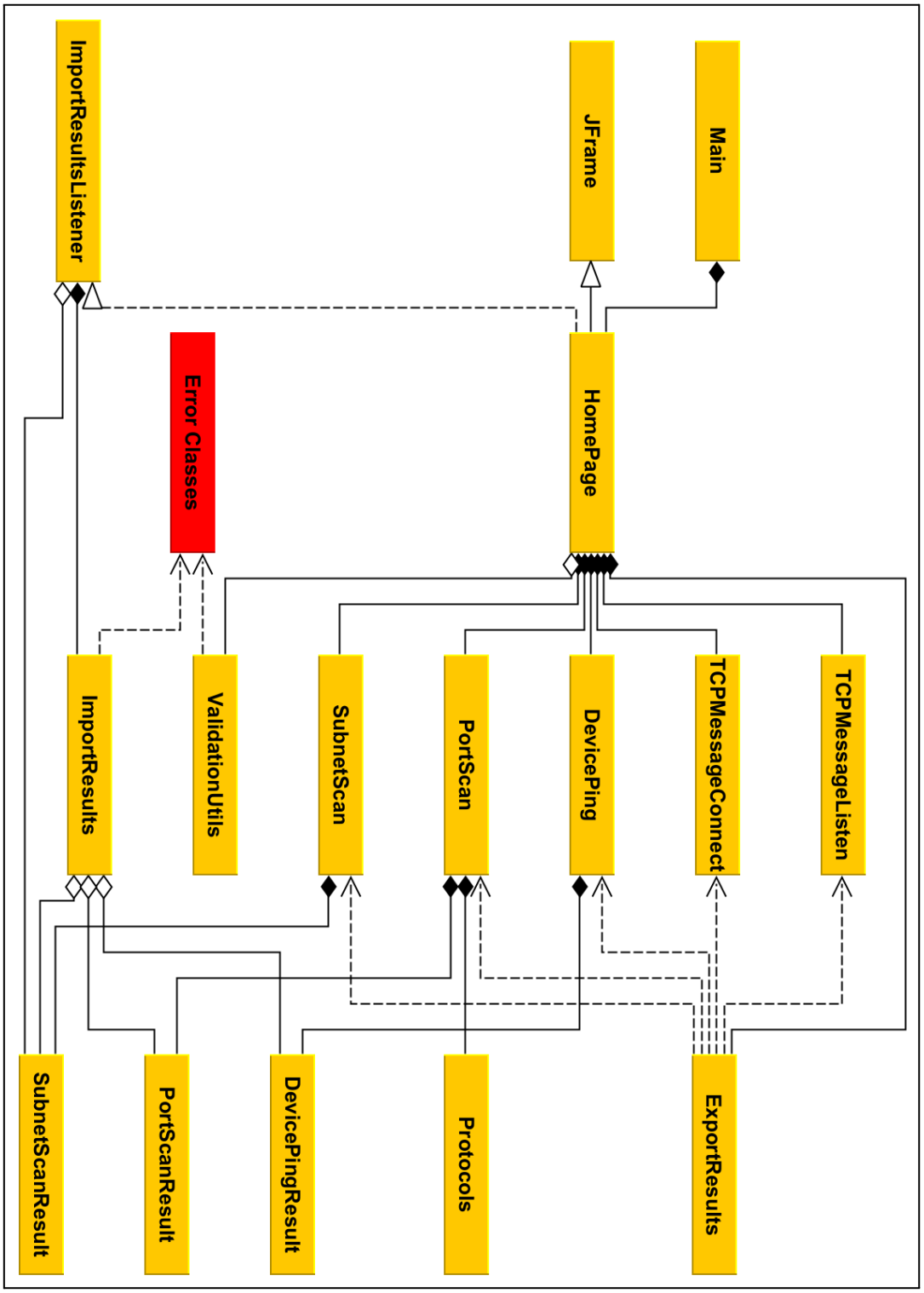


H. **Import Results—**



2.3 — Class Design and OOP Principles

The class hierarchy summary is as follows—



**note the error classes have been grouped into one box for the sake of simplicity, understanding, and fitting everything into one page*

The UML class diagrams of the back end classes are—

A. **SubnetScan—**

SubnetScan	
Fields:	
– subnetScanResults : ArrayList<SubnetScanResult>	List of subnet scan results
– networkRange : string	The network range to scan
– timeout: integer	Timeout in milliseconds for checking if an IP is reachable
– numOfIPs : integer	Total number of IPs to scan (calculated from the network range)
– model : DefaultTableModel	Table model that will be updated with the reachable IP addresses
– prgSubnetScan : JProgressBar	Progress bar to display the scanning progress
– THREAD_COUNT : integer = *available processors** × 32	Number of threads used for scanning
– executorService : ExecutorService = *new fixed thread pool with* THREAD_COUNT *number of threads***	Executor service for running scan tasks concurrently
– stopRequested : boolean = false	Flag to indicate if a stop has been requested for the scan
Methods:	
+ Constructor(networkRange : string, timeout : integer, tbl : JTable, prgSubnetScan : JProgressBar)	Constructs a new SubnetScan instance with the specified network range, timeout, table for results, and progress bar
+ start()	Starts the subnet scan
– setNumOfIPs()	Calculates and sets the number of IP addresses to scan
– generateIP(ipNum : integer) : string	Generates an IP address from the subnet starting IP based on the given index
– scanIP(inAddress : InetAddress)	Attempts to ping the single given IP address
– updateProgressBar(ipNum : integer)	Updates the progress bar based on the

	number of IPs scanned
+ getSubnetScanResults() : ArrayList<SubnetScanResult>	Retrieves the list of subnet scan results
+ setSubnetScanResutls(subnetScanResults : ArrayList<SubnetScanResult>)	Sets the subnet scan results
+ getNetworkRange() : string	Returns the network range that was scanned
+ getTimeout() : integer	Returns the timeout used for each IP scan
+ shutDownExecutorService()	Forcefully shuts down the executor service, stopping any running tasks
+ requestStop()	Requests the subnet scan to stop
+ isStopRequested() : boolean	Checks whether a stop has been requested for the scan

**Refers to the number of logical processors found in the CPU of the device which is running the program. It is a number variable to each computer, and is found by calling the* `.getRuntime().availableProcessors()` *method of the java* `Runtime` *class.*

***Refers to a new fixed thread pool created by calling the* `.newFixedThreadPool()` *method of the java* `Executors` *class. The number of threads in the pool is dependent on the* THREAD_COUNT *value.*

B. **SubnetScanResult—**

SubnetScanResult	
Fields:	
– ipAddress: string	The IP address that was found to be reachable
Methods:	
+ Constructor(ipAddress: string)	Constructs a new SubnetScanResult instance with the specified IP address
+ getIPAddress(): string	Returns the reachable IP address stored in this result

C. **DevicePing—**

DevicePing	
Fields:	
– devicePingResults : ArrayList<DevicePingResult>	List of results from each ping operation
– ipAddress : string	The IP address to ping
– pingInterval : integer	The interval (ms) between successive pings
– numOfPings : integer	The total number of pings to attempt if not in continuous mode
– continuousPinging: boolean	If true, the ping operations continue indefinitely
– pingCount : integer	Counter for the total number of pings attempted
– successfulPings : integer	Counter for the number of successful pings
– devicePingTableModel : DefaultTableModel	Table model for displaying individual ping results
– devicePingResponseResultsTableModel : DefaultTableModel	Table model for displaying ping response summaries (min, max, avg RTT)
– devicePingPacketResultsTableModel : DefaultTableModel	Table model for displaying packet loss summary results
– stopRequested : boolean = false	Flag indicating if a stop has been requested
Methods:	
+ Constructor(ipAddress : string, pingInterval : integer, numOfPings : integer, continuousPinging : boolean, tblDevicePing : JTable, tblDevicePingResponseResults : JTable, tblDevicePingPacketResults : JTable)	Constructs a new DevicePing instance with the specified target IP, interval, count, mode, and result tables
+ start()	Starts the pinging process, performing repeated pings and then populating summary tables
– pingIP()	Performs a single ping attempt, records RTT or failure, and updates the individual results table

+ populateResultsTables()	Populates the response and packet-loss summary tables with aggregated statistics
– getPacketLoss() : real	Calculates and returns the packet loss percentage, rounded to two decimals
– getMinimumRoundTripTime() : integer	Computes and returns the minimum RTT among all ping results
– getMaximumRoundTripTime() : integer	Computes and returns the maximum RTT among all ping results
– getAverageRoundTripTime() : real	Computes and returns the average RTT among successful pings, rounded to two decimals
+ requestStop()	Requests the ping loop to stop by setting the stop flag
+ isStopRequested() : boolean	Returns true if a stop request has been made
+ getDevicePingResults() : ArrayList<DevicePingResult>	Returns the list of individual ping results
+ setDevicePingResults(devicePingResults : ArrayList<DevicePingResult>)	Sets the list of ping results (used when importing results)
+ getIpAddress() : string	Returns the target IP address being pinged
+ getPingInterval() : integer	Returns the ping interval in milliseconds
+ getNumOfPings() : integer	Returns the total number of pings configured
+ isContinuousPinging() : boolean	Returns true if the ping process is set to run continuously
+ getSuccessfulPings() : integer	Returns the number of successful pings recorded
+ setSuccessfulPings()	Recalculates the count of successful pings from the results list
+ setPingCount()	Updates the pingCount to match the number of entries in the results list

D. **DevicePingResult—**

DevicePingResult	
Fields:	
– roundTripTime: integer	The IP address that was found to be reachable
– successfulPing : boolean	Flag which indicates whether the ping was successful
– packetLoss : real	The percentage of packet loss measured during the ping
Methods:	
+ Constructor(ipAddress: string)	Constructs a new DevicePingResult instance
+ getRoundTripTime() : integer	Returns the round-trip time (RTT) for the ping operation
+ isSuccessfulPing() : boolean	Returns whether the ping was successful
+ getPacketLoss() : real	Returns the packet loss percentage observed during the ping

E. **PortScan—**

PortScan	
Fields:	
– portScanResults : ArrayList<PortScanResult>	A list of port scan results
– ipAddress : string	The target IP address to scan
– bottomRangePort : integer	The starting port number of the scan range
– topRangePort : integer	The ending port number of the scan range
– timeout : integer	The timeout in milliseconds for attempting to connect to each port
– model : DefaultTableModel	The table model used to update the UI with the scan results
– prgPortScan : JProgressBar	The progress bar used to display scan progress
– THREAD_COUNT : int = *available processors** × 32	Number of threads used for scanning
– executorService : ExecutorService = *new fixed thread pool with* THREAD_COUNT *number of threads***	Executor service for running scan tasks concurrently
– stopRequested : boolean = false	A flag indicating whether a stop has been requested
Methods:	
+ Constructor(ipAddress : string, bottomRangePort : integer, topRangePort : integer, timeout : integer, tbl : JTable, prgPortScan : JProgressBar)	Constructs a new PortScan instance with the specified IP address, bottom range port, top range port, timeout, table for results, and progress bar
– scanPort(port : integer)	Attempts to scan a single port on the target IP address
+ start()	Starts the port scan
– updateProgressBar(portNum : integer)	Updates the progress bar based on the number of ports scanned
+ shutDownExecutorService()	Forcefully shuts down the executor service, stopping any running tasks
+ requestStop()	Requests the port scan to stop

+ isStopRequested() : boolean	Checks whether a stop has been requested for the scan
+ getPortScanResults() : ArrayList<PortScanResult>	Retrieves the list of port scan results
+ setPortScanResults(portScanResults : ArrayList<PortScanResult>)	Sets the port scan results
+ getIpAddress() : string	Returns the target IP address
+ getBottomRangePort() : integer	Returns the starting port number of the scan range
+ getTopRangePort() : integer	Returns the ending port number of the scan range
+ getTimeout() : integer	Returns the timeout used for each port connection attempt

**Refers to the number of logical processors found in the CPU of the device which is running the program. It is a number variable to each computer, and is found by calling the* `.getRuntime().availableProcessors()` *method of the java* `Runtime` *class.*

***Refers to a new fixed thread pool created by calling the* `.newFixedThreadPool()` *method of the java* `Executors` *class. The number of threads in the pool is dependent on the* THREAD_COUNT *value.*

F. **PortScanResult—**

PortScanResult	
Fields:	
– portNumber : integer	The port number that was found to be open
– protocol : string	The protocol associated with the port number
Methods:	
+ Constructor(portNumber : integer, protocol : string)	Constructs a new PortScanResult with the specified port number and protocol
+ getPortNumber() : integer	Returns the open port number associated with this scan result
+ getProtocol() : string	Returns the protocol name associated with this port scan result

G. **Protocols—**

Protocols	
Fields:	
– portProtocolMap : Map<integer, string>	A mapping from port numbers to protocol names loaded from CSV
– RESOURCE_PATH : string = "/com/pingpal/resources/databases/port_list.csv"	The path to the file containing the port-protocol relationship
Methods:	
– loadFromClasspath()	Reads CSV file and populates portProtocolMap skipping header and shows error dialogue if file not found
<<static initialiser>>	Calls the method to load the data from the class path.
– Constructor()	Private constructor to prevent instantiation
+ getProtocolForPort(portNumber : integer) : string	Retrieves the protocol name for the specified port or returns default message if not mapped

H. **TCPMessageListen—**

TCPMessageListen	
Fields:	
– SUCCESS_COLOR : Color = Color(0, 204, 0)	Colour for successful messages
– ERROR_COLOR : Color = Color(255, 51, 0)	Colour for error messages
– MESSAGE_COLOR : Color = Color(45, 45, 45)	Colour for normal messages
– DATE_TIME_COLOR : Color = Color(26, 39, 107)	Colour for date and time text
– HOSTNAME_COLOR : Color = Color(113, 89, 138)	Colour for hostname text
– dateTimeStyle : Style	Style for date and time formatting
– hostnameStyle : Style	Style for hostname formatting
– messageStyle : Style	Style for message text formatting
– errorStyle : Style	Style for error message formatting
– successStyle : Style	Style for success message formatting
– txpTCPMessageListen : JTextPane	Text pane used to display messages
– doc : StyledDocument	Styled document extracted from text pane
– stopRequested : boolean = false	Flag indicating whether stop has been requested
– serverSocket : ServerSocket	Server socket listening for client connections
– clientSocket : Socket	Socket for connected client
– in : BufferedReader	Reader for incoming messages
– out : PrintWriter	Writer for outgoing messages
– port : integer	Port on which the server listens
Methods:	
+ Constructor(port : integer, txpTCPMessageListen : JTextPane)	Initialises fields, extracts document and sets up styles

+ start()	Starts server socket, waits for client and begins receive loop
+ sendMessage(message : string)	Sends formatted message to client and appends it locally
+ requestStop()	Sets stopRequested closes sockets and logs exit
+ isDeviceConnected() : boolean	Returns whether a client is currently connected
+ getTextPaneContents() : string	Returns full contents of the text pane
– receiveMessages()	Continuously reads incoming messages formats and displays them
– updateTextPane(message : string, style : Style)	Appends styled message to text pane on EDT
– setStyles()	Defines and registers text styles for message formatting

I. **TCPMessageConnect—**

TCPMessageConnect	
Fields:	
– SUCCESS_COLOR : Color = Color(0, 204, 0)	Colour for successful messages
– ERROR_COLOR : Color = Color(255, 51, 0)	Colour for error messages
– MESSAGE_COLOR : Color = Color(45, 45, 45)	Colour for normal messages
– DATE_TIME_COLOR : Color = Color(26, 39, 107)	Colour for date and time text
– HOSTNAME_COLOR : Color = Color(113, 89, 138)	Colour for hostname text
– dateTimeStyle : Style	Style for date and time formatting
– hostnameStyle : Style	Style for hostname formatting
– messageStyle : Style	Style for message text formatting
– errorStyle : Style	Style for error message formatting
– successStyle : Style	Style for success message formatting
– txpTCPMessageConnect : JTextPane	Text pane used to display messages
– doc : StyledDocument	Styled document extracted from text pane
– stopRequested : boolean = false	Flag indicating whether stop has been requested
– socket : Socket	Socket for server connection
– in : BufferedReader	Reader for incoming messages
– out : PrintWriter	Writer for outgoing messages
– ipAddress : string	Server IP address to connect to
– port : integer	Server port to connect to
Methods:	
+ Constructor(ipAddress : string, port : integer, txpTCPMessageConnect : JTextPane)	Initialises fields, extracts document and sets up styles

+ start()	Attempts connection and begins receive loop
+ sendMessage(message : string)	Sends formatted message to server and echoes locally
+ requestStop()	Sets stopRequested closes socket and logs exit
+ isDeviceConnected() : boolean	Returns whether socket is currently connected
+ getTextPaneContents() : string	Returns full contents of the text pane
– receiveMessages()	Continuously reads incoming messages formats and displays them
– updateTextPane(message : string, style : Style)	Appends styled message to text pane on EDT
– setStyles()	Defines and registers text styles for message formatting

J. **InvalidNumOfPingsException—**

InvalidNumOfPingsException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

K. **InvalidPacketLossRangeException—**

InvalidPacketLossRangeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

L. **InvalidPingIntervalRangeException—**

InvalidPingIntervalRangeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

M. **InvalidPortNumberRangeException—**

InvalidPortNumberRangeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

N. **InvalidPortProtocolRelationshipException—**

InvalidPortProtocolRelationshipException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

O. **InvalidRoundTripTimeException—**

InvalidRoundTripTimeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

P. **InvalidScanTypeException—**

InvalidScanTypeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

Q. **InvalidSuccessfulPingException—**

InvalidSuccessfulPingException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

R. **InvalidTimeoutRangeException—**

InvalidTimeoutRangeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

S. **InvalidVariableInstanceException—**

InvalidVariableInstanceException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

T. **MissingRequiredKeysException—**

MissingRequiredKeysException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

U. **BlankFieldException—**

BlankFieldException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

V. **InvalidIPAddressException—**

InvalidIPAddressException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

W. **InvalidNetworkRangeException—**

InvalidNetworkRangeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

X. **InvalidPortRangeException—**

InvalidPortRangeException	
Fields:	
(none)	(none)
Methods:	
+ Constructor(message : string)	Constructs with specified detail message explaining the reason for the exception
+ Constructor(message : string, cause : Throwable)	Constructs with specified detail message and cause for later retrieval

Y. **ValidationUtils—**

ValidationUtils	
Fields:	
+ ERROR_COLOR : Color = Color(250, 200, 200)	Colour constant used for validation feedback
+ GRAYED_OUT_COLOR : Color = Color(184, 184, 184)	Colour constant used for validation feedback
+ NORMAL_TEXT_COLOR : Color = Color(233, 247, 249)	Colour constant used for validation feedback
+ SUCCESSFUL_SCAN_COLOR : Color = Color(0, 204, 0)	Colour constant used for validation feedback
+ INTERRUPTED_SCAN_COLOR : Color = Color(255, 51, 0)	Color constant used for validation feedback
+ NETWORK_RANGE_PATTERN : string = "^(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\\/(?:[1-9]|[12]\\d|3[0-2])$"	Regex pattern for validating network ranges
+ IP_ADDRESS_PATTERN : string = "^(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])$"	Regex pattern for validating IP addresses
+ MIN_TIMEOUT : integer = 100	Minimum acceptable timeout value
+ MAX_TIMEOUT : integer = 10000	Maximum acceptable timeout value
+ MIN_PING_INTERVAL : integer = 100	Minimum acceptable ping interval value
+ MAX_PING_INTERVAL : integer = 10000	Maximum acceptable ping interval value
+ MIN_PINGS : integer = 1	Minimum acceptable number of pings
+ MAX_PINGS : integer = 100	Maximum acceptable number of pings
+ MIN_PORT : integer = 1	Minimum acceptable port number
+ MAX_PORT : integer = 65535	Maximum acceptable port number
Methods:	
– Constructor()	Private constructor to prevent instantiation
+ validateFieldPresence(obj : Object)	Validates that obj is non-null and non-blank if it is a string

+ validateNetworkRange(networkRange : string)	Validates that networkRange matches required format
+ validateIPAddress(ipAddress : string)	Validates that ipAddress matches required format
+ validatePortRange(bottomRange : integer, topRange : integer)	Validates that bottomRange is not greater than topRange
+ validateRequiredKeys(topLevelKeys : string[], fileData : JSONObject)	Validates that fileData contains all topLevelKeys
+ validateInstanceString(obj : Object)	Validates that obj is an instance of String
+ validateInstanceInteger(obj : Object)	Validates that obj is an instance of Integer
+ validateInstanceNumber(obj : Object)	Validates that obj is an instance of Number
+ validateInstanceBoolean(obj : Object)	Validates that obj is an instance of Boolean
+ validateInstanceJSONArray(obj : Object)	Validates that obj is an instance of JSONArray
+ validateTimeoutRange(timeout : integer)	Validates that timeout falls within acceptable range
+ validatePingInterval(pingInterval : integer)	Validates that pingInterval falls within acceptable range
+ validateNumOfPingsRange(numOfPings : integer)	Validates that numOfPings falls within acceptable range
+ validateRoundTripTime(roundTripTime : integer, pingInterval : integer)	Validates that roundTripTime is within acceptable bounds
+ validateSuccessfulPingLogic(successfulPing : boolean, roundTripTime : integer, pingInterval : integer)	Validates consistency between successfulPing and roundTripTime
+ validatePacketLossRange(packetLoss : real)	Validates that packetLoss falls within acceptable range
+ validatePortNumberRange(portNumber : integer)	Validates that portNumber falls within acceptable range
+ validatePortCorrespondsToProtocol(portNumber : integer, protocol : string)	Validates that protocol corresponds to portNumber

Z. **ExportResults—**

ExportResults	
Fields:	
– panel : JPanel	Parent UI panel for dialogues
– fchDirectoryChooser : JFileChooser	File chooser for directory selection
– exportResultsPath : Path	Path of the directory selected for export
– fileName : string	Base name for the output file (no extension)
Methods:	
+ Constructor(panel : JPanel)	Constructs instance and prompts for directory then file name
– setExportResultsPath()	Shows directory chooser and stores chosen path
– setFileName()	Prompts repeatedly for valid file name (no blanks periods or slashes)
– writeToJSONFile(output : JSONObject)	Writes JSON object to .json file with success or error dialogue
– writeToTextFile(txt : string)	Writes plain text to .txt file with success or error dialogue
+ exportResults(subnetScan : SubnetScan)	Exports SubnetScan results to JSON
+ exportResults(devicePing : DevicePing)	Exports DevicePing results to JSON
+ exportResults(portScan : PortScan)	Exports PortScan results to JSON
+ exportResults(tcpMessageListen : TCPMessageListen)	Exports TCPMessageListen results to text file
+ exportResults(tcpMessageConnect : TCPMessageConnect)	Exports TCPMessageConnect results to text file

AA. **ImportResults—**

ImportResults	
Fields:	
– panel : JPanel	Parent UI panel for dialogues
– fchFileChooser : JFileChooser	File chooser configured to select only JSON files
– importResultsFile : File	File containing the imported JSON scan results
– fileData : JSONObject	JSONObject parsed from the selected file
– listener : ImportResultsListener	Listener notified when scan results have been successfully imported
Methods:	
+ Constructor(panel : JPanel, listener : ImportResultsListener)	Constructs a new ImportResults instance with the specified panel and listener
+ setImportResultsPath()	Opens a file chooser dialogue for the user to select a JSON file
+ determineScanType()	Determines the type of scan results contained in the imported JSON file, calls helper methods to validate the data, then calls the appropriate import method
– readFileData() : JSONObject	Reads and parses JSON data from the selected file
– validateSubnetScanData() : boolean	Validates that the JSON data for a subnet scan contains the required fields and that their types and values are correct
– validateDevicePingData() : boolean	Validates that the JSON data for a device ping scan contains required fields and that their values are valid
– validatePortScanData() : boolean	Validates that the JSON data for a port scan contains the required fields and that each field is valid
– parseSubnetScanResultsArray() : ArrayList<SubnetScanResult>	Parses subnet scan results from the JSON array and converts them into a list of SubnetScanResult objects
– importSubnetScanData()	Imports subnet scan results by extracting the

	data from the JSON object, then passing it to the listener
– parseDevicePingResultsArray() : ArrayList<DevicePingResult>	Parses device ping results from the JSON array and converts them into a list of DevicePingResult objects
– importDevicePingData()	Imports device ping results by extracting the data from the JSON object, then passing it to the listener
– parsePortScanResultsArray() : ArrayList<PortScanResult>	Parses port scan results from the JSON array and converts them into a list of PortScanResult objects
– importPortScanData()	Imports port scan results by extracting the data from the JSON object, then passing it to the listener

BB. **ImportResultsListener—**

<<interface>> **ImportResultsListener**	
Fields:	
(none)	(none)
Methods:	
onSubnetScanResultsImported(networkRange : string, timeout : integer, subnetScanResults : ArrayList)	Callback invoked when subnet scan results are imported, providing network range, timeout, and list of reachable IP addresses
onDevicePingResultsImported(ipAddress : string, pingInterval : integer, numOfPings : integer, continuousPinging : boolean, devicePingResults : ArrayList)	Callback invoked when device ping results are imported, providing IP address ping parameters and list of ping results
onPortScanResultsImported(ipAddress : string, bottomRangePort : integer, topRangePort : integer, timeout : integer, portScanResults : ArrayList)	Callback invoked when port scan results are imported, providing IP address, port range, timeout, and list of port scan results

2.4 — Secondary Storage Design

PingPal makes use of three types of secondary storage to permanently store data from six distinct program functions. They are—

A. **JSON files—**

JSON files are used in three program functions. They are—

a. **Subnet Scan**—

The results of a subnet scan can be extracted to a JSON file. This allows for the user to be able to import the file later and not lose any of the data.

In *PingPal*, the *SubnetScan* class holds in-memory data consisting of:

1. *networkRange* (string)—e.g. “192.168.0.0/24”;

2. *timeout* (integer)—e.g. 500 (ms);

3. *subnetScanResults* (ArrayList<SubnetScanResult>)—each *SubnetScanResult* wraps a reachable IP address

To persist these results, they are saved to a JSON file. The JSON structure mirrors the in-memory model, grouping metadata and results into a single object.

The JSON file top-level property structure for the *SubnetScan* class is as follows—

Property	**Type**	**Description**
networkRange	string	Network range scanned in CIDR notation
timeout	integer	Timeout used for each individual ping attempt in ms
subnetScanResults	ArrayList	List of reachability results

Each *SubnetScanResult* object in the *subnetScanResults* array contains—

Field	Type	Description
ipAddress	string	The reachable IP address in dot-decimal notation

The sample data for the *SubnetScan* class is as follows—


```
{
    "networkRange": "150.0.2.0/24",
    "subnetScanResults": [
        {"ipAddress": "150.0.2.97"},
        {"ipAddress": "150.0.2.3"},
        {"ipAddress": "150.0.2.22"},
        {"ipAddress": "150.0.2.42"},
        {"ipAddress": "150.0.2.46"},
        {"ipAddress": "150.0.2.45"},
        {"ipAddress": "150.0.2.15"},
        {"ipAddress": "150.0.2.53"},
        {"ipAddress": "150.0.2.72"},
        {"ipAddress": "150.0.2.65"},
        {"ipAddress": "150.0.2.101"},
        {"ipAddress": "150.0.2.109"},
        {"ipAddress": "150.0.2.93"},
        {"ipAddress": "150.0.2.113"},
        {"ipAddress": "150.0.2.111"},
        {"ipAddress": "150.0.2.127"},
        {"ipAddress": "150.0.2.52"},
        {"ipAddress": "150.0.2.31"},
        {"ipAddress": "150.0.2.131"},
        {"ipAddress": "150.0.2.172"},
        {"ipAddress": "150.0.2.149"},
        {"ipAddress": "150.0.2.156"},
        {"ipAddress": "150.0.2.75"},
        {"ipAddress": "150.0.2.154"},
        {"ipAddress": "150.0.2.5"},
        {"ipAddress": "150.0.2.44"},
        {"ipAddress": "150.0.2.76"},
        {"ipAddress": "150.0.2.110"},
        {"ipAddress": "150.0.2.136"},
        {"ipAddress": "150.0.2.92"},
        {"ipAddress": "150.0.2.28"},
        {"ipAddress": "150.0.2.185"},
        {"ipAddress": "150.0.2.188"},
        {"ipAddress": "150.0.2.174"},
        {"ipAddress": "150.0.2.99"},
        {"ipAddress": "150.0.2.181"},
        {"ipAddress": "150.0.2.23"},
        {"ipAddress": "150.0.2.103"},
        {"ipAddress": "150.0.2.195"}
    ],
    "timeout": 500
}
```

b. **Device Ping—**

The results of a device ping can be extracted to a JSON file. This allows for the user to be able to import the file later and not lose any of the data.

In *PingPal*, the *DevicePing* class holds in-memory data consisting of:

1. *ipAddress* (string)—e.g. “192.168.0.1”;
2. *pingInterval* (integer)—e.g. 500 (ms);
3. *numOfPings* (integer)—e.g. 10 pings;
4. *continuousPinging* (boolean)—e.g. true;
5. *devicePingResults* (ArrayList<DevicePingResult>)—each *DevicePingResult* wraps the information regarding the ping sent to the target IP

To persist these results, they are saved to a JSON file. The JSON structure mirrors the in-memory model, grouping metadata and results into a single object.

The JSON file top-level property structure for the *DevicePing* class is as follows—

Property	**Type**	**Description**
ipAddress	string	The target IP address that was pinged
pingInterval	integer	The interval (in milliseconds) between successive pings
numOfPings	integer	The total number of pings that was set to be attempted
continuousPinging	boolean	*true* if pings continued indefinitely, *false* otherwise
devicePingResults	ArrayList	A list of individual ping result objects

Each element of the *devicePingResults* array is an object with the following fields—

Field	Type	Description
roundTripTime	integer	The measured round-trip time for that ping (ms)
successfulPing	boolean	*true* if that ping succeeded, *false* otherwise
packetLoss	real	The packet-loss percentage at that point (0.00–100.00)

The sample data for the *DevicePing* class is as follows—


```
{
    "pingInterval": 100,
    "devicePingResults": [
        {
            "packetLoss": 0,
            "roundTripTime": 19,
            "successfulPing": true
        },
        {
            "packetLoss": 0,
            "roundTripTime": 12,
            "successfulPing": true
        },
        {
            "packetLoss": 0,
            "roundTripTime": 10,
            "successfulPing": true
        },
        {
            "packetLoss": 0,
            "roundTripTime": 36,
            "successfulPing": true
        },
        {
            "packetLoss": 0,
            "roundTripTime": 13,
            "successfulPing": true
        },
        {
            "packetLoss": 0,
            "roundTripTime": 13,
            "successfulPing": true
        }
    ],
    "continuousPinging": true,
    "ipAddress": "150.0.2.36",
    "numOfPings": 10
}
```

c. **Port Scan—**

The results of a port scan can be extracted to a JSON file. This allows for the user to be able to import the file later and not lose any of the data.

In *PingPal*, the *PortScan* class holds in-memory data consisting of:

1. *ipAddress* (string)—e.g. “192.168.0.1”;
2. *bottomRangePort* (integer)—e.g. 1;
3. *topRangePort* (integer)—e.g. 1023;
4. *timeout* (integer)—e.g. 500 (ms);
5. *portScanResults* (ArrayList<PortScanResult>)—each *PortScanResult* wraps an open port and its associated protocol

To persist these results, they are saved to a JSON file. The JSON structure mirrors the in-memory model, grouping metadata and results into a single object.

The JSON file top-level property structure for the *PortScan* class is as follows—

Property	**Type**	**Description**
ipAddress	string	The target IP address that was scanned
bottomRangePort	integer	The starting port number of the scan range
topRangePort	integer	The ending port number of the scan range
timeout	integer	The timeout in milliseconds used per port probe
portScanResults	ArrayList	A list of results, one for each port that was successfully scanned and determined open

Each element of the *portScanResults* array is an object with the following fields—

Field	Type	Description
portNumber	integer	The number of the open port that was detected
protocol	string	The protocol associated with the open port (e.g. “http”)

The sample data for the *PortScan* class is as follows—


```
{
    "topRangePort": 65535,
    "ipAddress": "150.0.2.93",
    "portScanResults": [
        {
            "protocol": "daap",
            "portNumber": 3689
        },
        {
            "protocol": "commplex-main",
            "portNumber": 5000
        },
        {
            "protocol": "afs3-fileserver",
            "portNumber": 7000
        },
        {
            "protocol": "font-service",
            "portNumber": 7100
        },
        {
            "protocol": "No specific protocol associated with
this port.",
            "portNumber": 62078
        }
    ],
    "bottomRangePort": 1,
    "timeout": 1000
}
```

B. **Text files—**

Text files are used in two program functions. They are—

a. **TCP Message Listen—**

The messages exchanged between two devices can be extracted to a text file. This allows for the user to be able to not lose any of the messages.

Since *TCPMessageListen* simply accumulates a styled transcript of all sent and received messages in its *JTextPane*, the export is a flat text file where each line represents one message entry. There is no ability to re-import into a *TCPMessageListen* object—this storage is write-only archival of the chat history.

No headers or metadata are written—the log begins immediately with the first message line. However, each file usually consists of—

Element	**Description**
Program messages regarding listening for connection	These are messages which indicate that the program is waiting for a connection, and in the case of a successful connection, indicate such.
Exchanged messages	This refers to the messages exchanged between the two devices. The format for this element is: <*dateTime*> [<*hostname*>] > <*message*>
Program messages regarding lost connection and error messages	These are messages which indicate that the connection has been closed or abruptly ended.

The sample data for the *TCPMessageListen* class is as follows—

```
Waiting for client connection on 169.254.123.33:1234.
Client connected: 169.254.123.33
08-08-25 10:21:45 [igors-laptop] > Hello!
08-08-25 10:21:57 [bobs-laptop] > Hi!
08-08-25 10:22:09 [igors-laptop] > How are you?
08-08-25 10:22:19 [bobs-laptop] > Good.
08-08-25 10:22:27 [igors-laptop] > That's good.
Exiting TCP Message and closing sockets.
```

b. **TCP Message Connect—**

The messages exchanged between two devices can be extracted to a text file. This allows for the user to be able to not lose any of the messages.

Since *TCPMessageConnect* simply accumulates a styled transcript of all sent and received messages in its *JTextPane*, the export is a flat text file where each line represents one message entry. There is no ability to re-import into a *TCPMessageConnect* object—this storage is write-only archival of the chat history.

No headers or metadata are written—the log begins immediately with the first message line. However, each file usually consists of—

Element	**Description**
Program messages regarding searching for a server to connect to	These are messages which indicate that the program is searching for a server to connect to, and in the case of a successful connection, indicate such.
Exchanged messages	This refers to the messages exchanged between the two devices. The format for this element is: <*dateTime*> [<*hostname*>] > <*message*>
Program messages regarding lost connection and error messages	These are messages which indicate that the connection has been closed or abruptly ended.

The sample data for the *TCPMessageListen* class is as follows—

```
Trying to establish a connection to 169.254.123.33:1234.
Connected to chat server at 169.254.123.33:1234.
08-08-25 10:21:45 [igors-laptop] > Hello!
08-08-25 10:21:57 [bobs-laptop] > Hi!
08-08-25 10:22:09 [igors-laptop] > How are you?
08-08-25 10:22:19 [bobs-laptop] > Good.
08-08-25 10:22:27 [igors-laptop] > That's good.
Exiting TCP Message and closing socket.
Server disconnected.
```

C. **CSV files—**

A CSV file is used in one program function. It is—

a. **Port-protocol relationship—**

The relationship between a port number and a protocol are permanently stored in a read-only CSV file. The data in this file is used to initialise the hash table in the *Protocols* class.

In *PingPal*, the *Protocols* class initialises permanently stored data into an in-memory HashMap consisting of:

1. *port_number* (integer)—e.g. 80;

2. *protocol* (string)—e.g. “http”

The CSV consists of two columns—

Column	**Description**
port_number	This refers to a port which is commonly used across devices.
protocol	This refers to the protocol generally associated with the linked port number.

The sample data from the port_list.csv file is as follows—

```
port_number,protocol
1,tcpmux
5,rje
7,echo
9,discard
11,systat
13,daytime
17,qotd
18,msp
19,chargen
20,ftp-data
21,ftp
22,ssh
...
45678,eba
45824,dai-shell
45825,qdb2service
45966,ssr-servermgr
46336,inedo
46998,spremotetablet
46999,mediabo
```

2.5 — Explanation of Secondary Storage Design

PingPal makes use of three types of secondary storage to permanently store data from six distinct program functions. They are—

A. **JSON files—**

I have chosen to use JSON files to persist the data from the different *PingPal* scans. The main reason why I decided on this format is that it makes use of object notation. The results of the scans make use of composition to represent each scan as an object in primary storage. Nested objects and arrays model the results naturally (metadata and list of result objects) without needing a relational schema for the current data size and usage pattern. JSON files maintain that class hierarchy, thereby making it most convenient and practical for persisting the data in secondary storage.

There are other advantages to using JSON in this case. Firstly, JSON is easily human-readable and editable. JSON is easily inspected or corrected by a user or reviewer without special tools. Secondly, JSON allows for native mapping to program data structures. The Java *org.json* library and the program's *ImportResults*/ *ExportResults* logic map cleanly between JSON objects/arrays and Java objects/ collections, reducing translation code complexity. Thirdly, JSON is extensible. Fields can be added in future releases without breaking parsers that ignore unknown fields. Fourthly, JSON is portable. JSON files can be shared across platforms, included in reports, or used by other tools for further analysis.

However, there are also some disadvantages to using JSON files. Firstly, there is no additional functionality for backing-up, restoring and archiving. Secondly, they are not secure. Anyone who has access to the JSON file can open and edit it.

B. **Text files—**

I have chosen to use plain text files to persist messages exchanged between devices during TCP Message sessions. The main reason why I have decided on this format is that the data from the text pane does not follow any format itself. The messages can be thought of plain text in the primary storage. Since transcripts are not designed to be re-imported, the overhead of structured formats proves to reduce simplicity and to be unnecessary. Text files maintain that simplicity, thereby making it most convenient and practical for persisting the messages in secondary storage.

There are other advantages to using text files in this case. Firstly, chat transcripts are primarily read by humans—plain text is the most straightforward, portable, and editable format for that purpose. Secondly, a simple timestamp, hostname, and message layout reads well in editors, logs, and when pasted into other documents. Thirdly, writing plain text requires minimal code and dependencies, simplifying export logic and reducing failure modes.

However, there are also some disadvantages to using text files. Firstly, there is no additional functionality for backing-up, restoring and archiving. Secondly, they are not secure. Anyone who has access to the text file can open and edit it.

C. **CSV files—**

I have chosen to use CSV files to persist the data of the port-protocol relationships. The main reason why I have decided on this format is that there is a one-to-one relationship between each port and protocol—for each port, there is only one protocol (theoretically, there can be multiple protocols associated with one port; however, cases thereof are stored as one value in the CSV file). Since the mapping is effectively static reference data, it does not require complex querying or concurrent updates. CSV files are used with one-to-one relationships in mind, thereby making it most convenient and practical for persisting port-protocol relationships in secondary storage.

There are other advantages to using CSV files in this case. Firstly, CSV is a minimal, human-readable format well suited for a small static lookup table. Secondly, developers can open and edit the file with a text editor or spreadsheet program without special tooling. Thirdly, reading a small CSV is fast, and no database dependency is required, which keeps the application lightweight.

However, there are also some disadvantages to using CSV files. Firstly, there is no additional functionality for backing-up, restoring and archiving. Secondly, they are not secure. Anyone who has access to the text file can open and edit it.

2.6 — Explanation of How Primary Data Structures Relate to Secondary Storage

The only classes which contain primary data that is persisted to secondary storage are—

A. **SubnetScan**

B. **DevicePing**

C. **PortScan**

D. **Protocols**

E. **TCPMessageListen**

F. **TCPMessageConnect**

Additionally, these classes contain arrays of objects from the following classes—

A. **SubnetScan** contains **SubnetScanResult** objects

B. **DevicePing** contains **DevicePingResult** objects

C. **PortScan** contains **PortScanResult** objects

However, I tried to make my code as modular as possible. This resulted in the implementation of other classes which deal with the writing to and reading from secondary storage of specific classes. These class are—

A. **ExportResults**

B. **ImportResults**

The following explains how each individual class listed: (a) if applicable, contains the data of another class; (b) relates the primary data structures to the secondary storage; (c) writes the data to secondary storage; (d) and reads data from secondary storage. The following classes are—

A. **SubnetScan—**

a. The *SubnetScan* class contains the data found in *SubnetScanResult* objects. This is done by storing an array of *SubnetScanResult* objects in the *SubnetScan* class—where each *SubnetScanResult* object represents a singe device on a given network. The data encompassed by a *SubnetScanResult* object contains only an IP address.

b. The primary data from the *SubnetScan* class which is persisted to the JSON files in secondary storage includes the following properties:

1. *networkRange*
2. *timeout*
3. *subnetScanResults*

Each property is stored in the JSON file as an individual top-level key. The name of each key corresponds to the name of the variable—each key shares the same name with the corresponding variable in the program (e.g. *networkRange* is stored as *networkRange* in the JSON file).

c. The *ExportResults* class handles writing the primary data to secondary storage. The export process consists of choosing the location to save the file to, choosing the name for the file, constructing the JSON object, populating the top-level fields, populating the JSON array, and finally, writing the data to the file. This design ensures that all in-memory subnet scan data is captured in a human-readable, structured JSON format suitable for local storage.

d. The *ImportResults* class handles reading the data from secondary storage into primary storage. The import process consists of choosing the location of the file to import, identifying what type of scan is being imported (if any), validating the data of the scan, then finally writing the data to primary storage and displaying the information in the relevant card. This design ensures that all secondary subnet scan data is in the correct format, and therefore can be read by the program.

B. **DevicePing—**

a. The *DevicePing* class contains the data found in *DevicePingResult* objects. This is done by storing an array of *DevicePingResult* objects in the *DevicePing* class—where each *DevicePingResult* object represents a result of a ping operation performed on a device. The data encompassed by a *DevicePingResult* object contains the round-trip time of the ping, whether the ping was successful, and the percentage of packet loss measured during that ping.

b. The primary data from the *DevicePing* class which is persisted to the JSON files in secondary storage includes the following properties:

1. *ipAddress*
2. *pingInterval*
3. *numOfPings*
4. *continuousPinging*
5. *devicePingResults*

Each property is stored in the JSON file as an individual top-level key. The name of each key corresponds to the name of the variable—each key shares

the same name with the corresponding variable in the program (e.g. *ipAddress* is stored as *ipAddress* in the JSON file).

c. The *ExportResults* class handles writing the primary data to secondary storage. The export process consists of choosing the location to save the file to, choosing the name for the file, constructing the JSON object, populating the top-level fields, populating the JSON array, and finally, writing the data to the file. This design ensures that all in-memory device ping data is captured in a human-readable, structured JSON format suitable for local storage.

d. The *ImportResults* class handles reading the data from secondary storage into primary storage. The import process consists of choosing the location of the file to import, identifying what type of scan is being imported (if any), validating the data of the scan, then finally writing the data to primary storage and displaying the information in the relevant card. This design ensures that all secondary device ping data is in the correct format, and therefore can be read by the program.

C. **PortScan—**

a. The *PortScan* class contains the data found in *PortScanResult* objects. This is done by storing an array of *PortScanResult* objects in the *PortScan* class—where each *PortScanResult* object represents a single open port found on a device. The data encompassed by a *PortScanResult* object contains the port number that was found to be open, and the associated protocol determined for that port.

b. The primary data from the *PortScan* class which is persisted to the JSON files in secondary storage includes the following properties:

    1. *ipAddress*

2. *bottomRangePort*
3. *topRangePort*
4. *timeout*
5. *portScanResults*

Each property is stored in the JSON file as an individual top-level key. The name of each key corresponds to the name of the variable—each key shares the same name with the corresponding variable in the program (e.g. *ipAddress* is stored as *ipAddress* in the JSON file).

c. The *ExportResults* class handles writing the primary data to secondary storage. The export process consists of choosing the location to save the file to, choosing the name for the file, constructing the JSON object, populating the top-level fields, populating the JSON array, and finally, writing the data to the file. This design ensures that all in-memory port scan data is captured in a human-readable, structured JSON format suitable for local storage.

d. The *ImportResults* class handles reading the data from secondary storage into primary storage. The import process consists of choosing the location of the file to import, identifying what type of scan is being imported (if any), validating the data of the scan, then finally writing the data to primary storage and displaying the information in the relevant card. This design ensures that all secondary port scan data is in the correct format, and therefore can be read by the program.

D. **Protocols—**

a. The *Protocols* class does not encapsulate primary data from a class other than itself—as in, there is no relationship where the *Protocols* class uses composition to store any other class within itself.

b. The primary data in the *Protocols* class relates to a CSV file in secondary storage. The *port_list.csv* file is used to populate the static lookup table within the class. The *Protocols* class loads the *port_list.csv* file at class load time and builds an in-memory *Map<Integer, String>* (named *portProtocolMap*) that maps a port number to a protocol name. The CSV file contains two columns, whereby the first column (*port_number*) relates to the first part of the lookup table (the *Integer* part), and the second column (*protocol*) relates to the second part of the lookup table (the *String* part).

c. The *port_list.csv* file is read-only, therefore, it is only ever read by the program and never written to.

d. The *Protocols* class handles its own imports and subsequent lookup table mapping. The class populates *portProtocolMap* in a static initialiser that runs once when the *Protocols* class is first loaded by the JVM. The import and mapping process consists of opening the *port_list.csv* file for reading, skipping the header, reading each line and splitting it into columns, and finally mapping each column from a single line to the lookup table. This design ensures that the data is not unnecessarily imported into the program multiple times, and that all secondary port scan data is in the correct format, and therefore can be read by the program.

E. **TCPMessageListen—**

a. The *TCPMessageListen* class does not encapsulate primary data from a class other than itself—as in, there is no relationship where the *TCPMessageListen* class uses composition to store any other class within itself.

b. The primary data in the *TCPMessageListen* class, which relates to secondary storage, consists of the raw text found within the text pane that is being

displayed to the user interface. The secondary storage which it relates to refers to a text file containing an exchange of messages between two devices on a network, which have been exported by the user. The text is simply stored in the text file, without any specific formatting.

c. The *ExportResults* class handles writing the primary data to secondary storage. The export process consists of choosing the location to save the file to, choosing the name for the file, retrieving the raw text data from the text pane, and finally, writing the data to the file. This design ensures that all in-memory message data is captured in the same way it was being displayed on the user interface, which is suitable for local storage.

d. The *TCPMessageListen* export files are write-only, therefore, they are only ever written to by the program and never read.

F. **TCPMessageConnect—**

a. The *TCPMessageConnect* class does not encapsulate primary data from a class other than itself—as in, there is no relationship where the *TCPMessageConnect* class uses composition to store any other class within itself.

b. The primary data in the *TCPMessageConnect* class, which relates to secondary storage, consists of the raw text found within the text pane that is being displayed to the user interface. The secondary storage which it relates to refers to a text file containing an exchange of messages between two devices on a network, which have been exported by the user. The text is simply stored in the text file, without any specific formatting.

c. The *ExportResults* class handles writing the primary data to secondary storage. The export process consists of choosing the location to save the file to,

choosing the name for the file, retrieving the raw text data from the text pane, and finally, writing the data to the file. This design ensures that all in-memory message data is captured in the same way it was being displayed on the user interface, which is suitable for local storage.

d. The *TCPMessageConnect* export files are write-only, therefore, they are only ever written to by the program and never read.