Keikaku 企画

PMR Insight Collective Knowledge (PICK)
System Design Document

Version 2.5
3/29/2020

Document Control

Approval

The Guidance Team and the customer shall approve this document.

Document Change Control

Initial Release:	1.0
Current Release:	2.5
Indicator of Last Page in Document:	[END]
Date of Last Review:	03/29/2020
Date of Next Review:	03/29/2020
Target Date for Next Update:	04/01/2020

Distribution List

This following list of people shall receive a copy of this document every time a new version of this document becomes available:

Guidance Team Members:
Dr. Ann Gates, Dr. Steve Roach, Mr. Jake Lasley

Clients:
Mr. Vincent Fonseca, Mr. Baltazar Santaella, Ms. Herandy Vazquez, and Mr. Erick De Nava

Software Team Members:
Mr. Anthony Desarmier, Mr. Angel Villalpando, Mr. Mario Delgado, Mr. David Rayner,
Mr. Valentin Becerra, Mr. Jorge Garcia

Change Summary

The following table details changes made between versions of this document

Version	Date	Modifier	Description
0.1	02/26/2020	Jorge Garcia	Added Template
1.0	03/08/2020	Anthony DesArmier	Wrote class descriptions, contracts, responsibilities; wrote protocols for Event, wrote sections 2, 2.2, 2.3
1.0	03/08/2020	Jorge Garcia	Added class descriptions from CRC, wrote section 1.5 and updated references.
1.0	03/09/2020	David Rayner	Completed section 1.1 Completed Component Diagram section 2.1.
2.0	03/13/2020	David Rayner	Refactored contracts changed collaboration diagram and updated section 2.2 with new contracts.
2.1	03/25/2020	David Rayner	Completed protocols for Vector, ActiveVector, IDDict, SplunkManager and EAR.
2.2	03/24/2020	Valentin Becerra	Completed protocols for ExportGraph
2.3	03/25/2020	Valentin Becerra	Completed protocols for Relationship
2.4	03/26/2020	Valentin Becerra	Complete protocols for Node and Graph

2.4	03/26/2020	Jorge Garcia	Complete protocols for ExportTable and ProjectMerge
2.4	03/27/2020	Angel Villalpando	Completed protocols for all contracts in History and Sync classes and completed the protocols for contracts 11, 12, and 13 in Settings class.
2.5	03/29/2020	Anthony DesArmier	Reviewed all protocols, updated section 2, formatting

Table of Contents

1. Introduction

1.1. Purpose and Intended Audience

The purpose of the Software Design Document to guide the software team in the construction of the PICK system. It is a means of construction such that the client's requirements are to be satisfied. The design or architectural overview of the system is to model the components that interact within the system. The Software Design Document is also intended for the clients to further establish or maintain the system in the future. Thus, the document must be detailed enough to provide both the software team and clients with enough information regarding the system design as possible.

1.2. Scope of Product

The PMR Insight Collective Knowledge (PICK) is the software system for which this SDD is written for. PICK is a software system to help Prevent, Mitigate, and Recover Analysts analyze vast amounts of data collected during an Adversarial Assessment (AA) by allowing them to quickly search through, view, correlate, and build visual documents which help explain the AA itself to uninvolved personnel. The customers - in this case PMR Analysts - currently must sift through the vast amounts of generated data from the AA by hand which severely hinders their workflow and efficiency in developing a report with visual aids for which to explain the nature of the AA to other personnel.

PICK will allow the customers to insert all the data generated from an AA into its system and display an organized, searchable database of that information. The customers can then quickly and efficiently find and correlate relevant data events together and help craft timelines which describe the significant events and their relations to one another during the AA. PICK will then assist the customers in crafting a visual representation of these series of events as attack graphs in order to help visualize the timeline of the AA. This assistance of analyzing the data generated by the AA and constructing visual representations of significant events will substantially reduce the time and work hours needed by the customers to understand and construct a report on the results of the AA to deliver to other personnel.

1.3. Definitions, Acronyms, and Abbreviations

1.3.1. Definitions

Table I.
Definitions of Complex Terms Found Within this Document

TERM	**DEFINITION**
Adversarial Assessment	An *Adversarial Assessment* gauges the ability of a system to support its mission(s) while withstanding validated and representative cyber threat activity [1]-[4].
Analyst	An Analyst is the end user who is evaluating log entries and associating them to their respective vectors. Their sole purpose is to generate a picture of the events taken place during an AA [1]-[4].
Log	A log is a series of noted occurrences or events that describe the events that have taken place during an AA [1]-[4].
Vector	A vector is a series of events an adversary attempts to execute in order to achieve an objective or series of objectives. In this context, Red team attack initiatives and Blue team defenses [4].
Vector Table	A vector table is a representative tabular view of vector properties this includes, vector name, vector description, and the vector’s nodes [4].
Vector Element	A vector element is either a Node or a Relationship that can be added to a Vector [4].
Validator	A Validator performs the validation of the raw log files based on the criteria provided by the PMR Analyst [4].
Graphical View	A vector visualization or graphical view of the respective vector. This includes the circular nodes with image icons and their connections denoted with directional lines [4].
Event	An event is a log entry consisting of a timestamp and a textual description or message of what took place [2], [3].
Node	A node is representative of a significant event, that is an event an analyst assigns to a vector and is marked as visible on a Vector table [1]-[4].
Relationship	A relationship is a connection between a parent node to their child nodes. It is how the nodes/events are associated [1]-[4].
Attack Graph	An attack graph is a succinct representation of data structures that model all possible avenues of attacking a network [1]-[4].
White Team	White team represents the third party LSH members or analysts that observe the AA events [1]-[3].
Red Team	Red team represents the team that perform cyber-attacks on a combat system [1]-[4].
Blue Team	Blue team represents the defense team, or the team that responds to the Red team cyber-attacks [1]-[4].
Constraint	A system limitation, or the bounds to which the system can operate.
Configuration	Parameters that are defined by the Analysts regarding the AA’s details [4].
Regular Expression	A search pattern consisting of a sequence of characters [1]-[3].
Property	An attribute or characteristic that belongs to an object [4].
Ingest	The intake of files [1]-[3].
Parse	The separation of text to smaller more manageable parts.
Interactive	User-system interaction, system responses to user inputs.
Prompt	Pop-up window used to display an alert or dialog of information

Enforcement Action Report	Report that outlines various errors found in validation and solicits the user for an enforcement action.
Push	The sending of committed changes to a remote repository.
Pull	The fetching and downloading of content from a remote repository and immediately updating the local repository to match that content.

1.3.2. Acronyms

Table II.
Definitions of Acronyms Found Within this Document

TERM	DEFINITION
SRS	Software Requirements Specification
PMR	Prevent Mitigate Recover [1]-[4]
PICK	PMR Insight Collective Knowledge [1]-[4]
AA	Adversarial Assessment [1]-[4]
EAR	Enforcement Action Report

1.3.3 Abbreviations

Table III.
Definitions of Abbreviations Found Within this Document

TERM	DEFINITION
e.g.	For example
i.e.	In other words
etc.	Et cetera “and” “the rest”

1.4. References

[1] V. Becerra, A. DesArmier, J. Garcia, D. Rayner and A. Villalpando, "PMR Insight Collective Knowledge (PICK) Interview Report," El Paso, 2019.

[2] Lethality, Survivability and HSI Directorate (LSH), "Software Engineering I - Fall 2019 PMR Insight Collective Knowledge (PICK)," Ramirez, Tai Elsa, El Paso, 2019.

[3] H. Vasquez , F. Larsen, F. Vicent , B. Santaella and E. De Nava, "U.S. ARMY Combat capabilities development command - Data and Analysis Center PMR Insight Collective Knowledge (PICK)," Ramirez, Tai Elsa, El Paso, 2019.

[4] Lethality, Surivability and HSI Directorate , "PICK Needs 11-6 - Updated," Ramirez, Tai Elsa, El Paso, 2019.

[5] V. Becerra, A. DesArmier, J. Garcia, D. Rayner, A. Villalpando and Mario Delgado, “Class-responsibility-collaboration Model,” El Paso, 2020.

1.5. Overview

The SDD describes the system at an architectural level providing a high-level description of subsystems, data management and other system components. Such descriptions have been provided in the following sections.

Section 2: Decomposition Description includes a System Collaboration Diagram in the form of a UML Component Diagram. This provides the system’s relationships with its subsystems. This also includes a list of all subsystem descriptions and the contracts their components fulfill.

Section 3 provides a detailed description of the User Interface subsystem. This subsystem is responsible for accepting and handling all user input. Additionally, it is also responsible for displaying all the internal system data.

Section 4 provides a detailed description of the Adversarial Assessment Data Processing subsystem. This subsystem is responsible for ingesting user provided files, relaying the files to Splunk and retrieving the data generated from said files.

Section 5 provides detailed information on the Internal Data Store subsystem. This subsystem is responsible for all the data persistence it is also responsible for all the maintenance of the configurable items in the system.

Section 6 provides detailed information on the Graph subsystem. This subsystem is responsible for the displaying and the manipulation of the node and relationship data found within a vector.

2. Decomposition Description

The decomposition description can be used by designers and maintainers to identify the major design entities of the system for purposes such as determining which entity is responsible for performing specific functions and tracing requirements to design entities. Design entities can be grouped into major classes to assist in locating information and to assist in reviewing the decomposition for completeness.

The information in the decomposition description can be used by project management for planning, monitoring, and control of a software project. They can identify each software component, its purpose, and basic functionality. This design information together with other project information can be used in estimating cost, staff, and schedule for the development effort.

2.1. System Collaboration Diagram

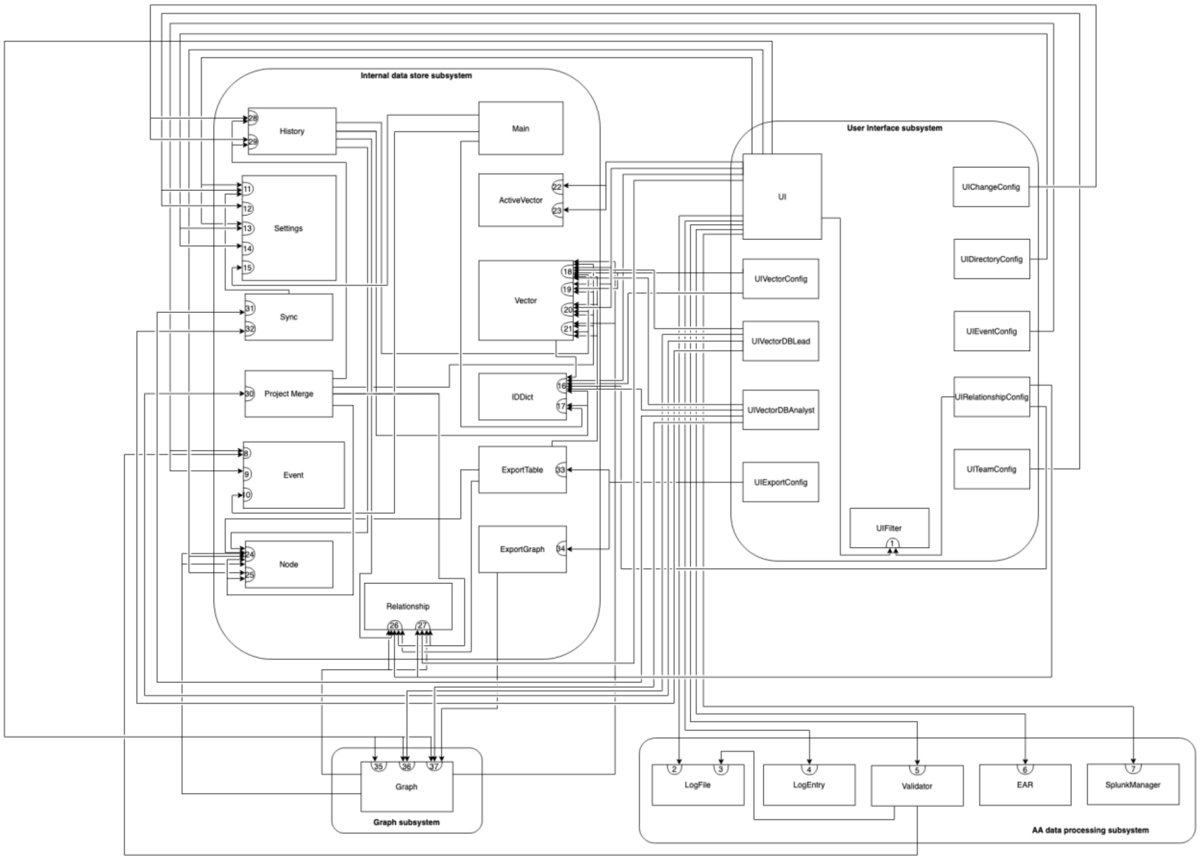

Fig. 1. UML Collaboration Diagram for the PICK System, 2020.

The system has been structured loosely on the model-view design pattern with further separation within the model component to separate distinct and independent functionality and purpose. The User Interface subsystem serves as both the view and controller components for the other subsystems. The Adversarial Assessment Data Processing, Internal Data Store, and Graph subsystems each represent a distinct and separate core function and service to the system.

2.2. Subsystem and Component Descriptions

The User Interface subsystem accepts user input for control of the other subsystems and displays internal system data. This subsystem is comprised of the following components:

1. UI: The main window which provides the bulk of the system's interface. (3.2)
   No Contracts
2. UiDirectoryConfig: The directory window which handles the setting of file directory paths. (3.3)
   No Contracts
3. UiEventConfig: The event window which handles the setting of project event information. (3.4)
   No Contracts
4. UiExportConfig: The export window which handles exporting vector and entry data to an external file. (3.5)
   No Contracts
5. UiFilterConfig: The filter window which handles the creation and application of a search filter. (3.6)
   1. Provide filter.
6. UiTeamConfig: The team window which handles the system role and connections to the host when running as a client. (3.7)
   No Contracts
7. UiVectorConfig: The vector window which handles the adding, editing, and deleting of vectors and their descriptions. (3.8)
   No Contracts
8. UiRelationshipConfig: The relationship window which handles the relationship table for the active vector. (3.9)
   No Contracts
9. UiVectorDBAnalyst: The analyst vector DB window which handles the pulling and pushing of vector changes between the analyst and host systems. (3.10)
   No Contracts
10. UiVectorDBLead: The lead vector DB window which handles the accepting of queued analyst push requests. (3.11)
    No Contracts
11. UiChangeConfig: The commit window which handles the committing of log entry and vector changes. (3.12)
    No Contracts

The Adversarial Assessment Data Processing subsystem is responsible for ingesting user-provided files, validating them, sending them to Splunk for processing, and retrieving the Splunk-generated log entries for analyst processing and manipulation in the PICK system. This subsystem is comprised of the following components:

1. LogFile: responsible for reading the user-provided directory path and importing the contents of the file. (4.2)
    2. Provide file status.
    3. Save file status.
2. LogEntry: responsible for storing the indexed entries in a log file. (4.3)
    4. Provide entry details.
3. Validator: responsible for validating and cleansing the content of user-provided files. (4.4)
    5. Cleanse and validate log files.
4. EnforcementActionReport: responsible for a report that outlines various errors found in validation. (4.5)
    6. Provide enforcement action report.
5. SplunkManager: responsible for the interface between Splunk and the system to submit log files and retrieve log entries. (4.6)
    7. Generate log entries from Splunk.

The Internal Data Store subsystem is responsible for the maintenance and persistence of user-provided, configurable items. Such items can include customized event names, vector names, vector components such as nodes and relationships, and any other labels or data that the system allows the user to personalize. This subsystem is comprised of the following components:

1. Main: serves as an entry and exit point to the system. (5.2)
    No Contracts
2. Event: responsible for maintaining the information and attributes that describe the specific event for an analysis/session. (5.3)
    8. Provide event details.
    9. Save event details.
    10. Save-load event object.
3. Settings: responsible for housing global variables of a given event/session. (5.4)
    11. Provide networking details.
    12. Save networking details.
    13. Provide directory details.
    14. Save directory details.
    15. Save-load setting object.
4. IDDict: responsible for storing a map of generic objects and their unique id. (5.5)
    16. Modify dictionary.
    17. Save-load dictionary object.
5. Vector: responsible for storing the data of nodes and relationships that compose a vector. (0)
    18. Provide vector details.
    19. Save vector details.
    20. Modify node dictionary.
    21. Modify relationship dictionary.
6. ActiveVector: responsible for storing the currently selected vector and its unique id. (5.7)
    22. Provide active vector details.
    23. Save active vector details.
7. Node: responsible for the storage of log entry properties. (5.8)
    24. Provide node details.
    25. Save node details.

8. Relationship: responsible for storing the child and parent node IDs that constitute a given relationship in the system. (5.9)
    26. Provide parent-child node relationship details.
    27. Save parent-child node relationship details.
9. History: responsible for storing the vectors’, nodes’ and relationships’ history of changes and saving them. (5.10)
    28. Display history of changes.
    29. Save changes.
10. ProjectMerge: responsible for the merging of analyst and lead history changes.
    30. Merge data. (5.11)
11. Sync: responsible for handling data transferal from lead and client. (5.12)
    31. Provides push/pull request changes to lead.
    32. Modify push request queue.
12. ExportTable: responsible for the formatting of the log entry tables that compose the vectors of a given event into an exportable file. (5.13)
    33. Save tables to file.
13. ExportGraph: responsible for the formatting of the final graph with its corresponding vector into an exportable image file. (5.14)
    34. Save graph to image file.

The Graph subsystem is responsible for the displaying and the manipulation of the node and relationship data found within a vector. This subsystem is comprised of the following component:

1. Graph: responsible for rendering nodes, relationships, their relative positions, and their properties for a given vector. (6.2)
    35. Provide graph details.
    36. Save graph details.
    37. Render-move graphical elements.

2.3. Dependencies

Since the User Interface subsystem displays and controls all other subsystems, this subsystem needs to be developed first. The Adversarial Assessment Data Processing, Internal Data Store, and Graph subsystem are patterned on the specific workflow sequence of the system and thus each subsystem is generally dependent on the previous subsystem to perform any meaningful functions. Therefore, the Adversarial Assessment Data Processing, Internal Data Store, and Graph subsystem should be developed in sequence.

3. User Interface: Detailed Description

3.1. Component Description

The User Interface subsystem accepts user input for control of the other subsystems and displays internal system data. This subsystem is composed of the Ui, UiDirectoryConfig, UiEventConfig, UiExportConfig, UiFilterConfig, UiTeamConfig, UiVectorConfig, UiRelationshipConfig, UiVectorDBAnalyst, UiVectorDBLead, and the UiChangeConfig classes.

3.2. Class Description: Ui

The main window which provides the bulk of the system's interface.

Class Name: Ui	
Superclass: QMainWindow	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Launches UiChangeConfig.	
• Launches UiDirectoryConfig.	
• Launches UiEventConfig.	
• Launches UiExportConfig.	
• Launches UiFilterConfig.	
• Launches UiTeamConfig.	
• Launches UiVectorConfig.	
• Launches UiRelatonshipConfig.	
• Launches UiVectorDBAnalyst when not lead.	Settings 1
• Launches UiVectorDBLead when lead.	Settings 1
• Knows the vector IDDict.	
• Knows the log file IDDict.	
• Knows the log entry IDDict.	
• Discovers log files.	Settings 7-10; IDDict 1
• Displays log file table.	IDDict 4; LogFile 1-6
• Displays Enforcement Action Report table.	EnforcementActionReport 1
• Selects an Enforcement Action Report to display.	LogFile 6
• Cleanses log files.	Validator 1
• Validates log files.	Validator 2
• Ingests log files.	SplunkManager 1
• Retrieves indexed log entries.	SplunkManager 2; IDDict 1
• Displays log entry table.	IDDict 4; LogEntry 1-4; Filter 1-4
• Assigns log entries to vectors.	Vector 1
• Sets active vector.	IDDict 3; ActiveVector 3-4
• Displays active vector name.	ActiveVector 1; Vector 1
• Displays active vector description.	ActiveVector 1; Vector 2
• Displays node table.	ActiveVector 1; Vector 26; Filter 1-4
• Adds nodes.	ActiveVector 1; Vector 23
• Edits node properties.	ActiveVector 1; Vector 9; 11-19
• Knows node visibility.	ActiveVector 1; Vector 25; Node 9
• Sets node visibility.	ActiveVector 1; Vector 25; Node 19

• Deletes nodes.	ActiveVector 1; Vector 24
• Displays node property visibilities.	ActiveVector 1; Vector 3-11
• Selects node property visibilities.	ActiveVector 1; Vector 14-22
• Displays vector graphical view.	ActiveVector 1; Graph 1-2, 5-6
• Zooms in/out on the graphical view.	
• Knows the timeline orientation.	Graph 1
• Knows the timeline interval.	Graph 2
• Sets the timeline orientation.	Graph 3
• Sets the timeline interval.	Graph 4
• Moves nodes on graph.	Graph 7
• Moves relationships on graph.	Graph 8
• Adds a new relationship.	ActiveVector 1; IDDict 1
• Deletes a relationship.	ActiveVector 1; IDDict 2
• Edits existing relationship attributes.	ActiveVector 1; IDDict 3; Relationship 5-7
No Contracts	

3.3. Class Description: UiDirectoryConfig

The directory window which handles the setting of file directory paths.

Class Name: UiDirectoryConfig	
Superclass: QDialog	
Subclasses: None	
Private Responsibilities	**Collaborations**
• Displays user provided directory path for root ingestion.	Settings 10
• Displays red team directory path.	Settings 7
• Displays blue team directory path.	Settings 8
• Displays white team directory path.	Settings 9
• Collects user provided directory path for root ingestion.	Settings 15
• Collects red team directory path.	Settings 12
• Collects blue team directory path.	Settings 13
• Collects white team directory path.	Settings 14
No Contracts	

3.4. Class Description: UiEventConfig

The event window which handles the setting of project event information.

Class Name: UiEventConfig	
Superclass: QDialog	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Displays event name.	Event 1
• Displays event description.	Event 2
• Displays event start timestamp.	Event 3
• Displays event end timestamp.	Event 4
• Collects event name.	Event 5
• Collects event description.	Event 6
• Collects event start timestamp.	Event 7
• Collects event end timestamp.	Event 8

• Saves event information to file.	Event 9
No Contracts	

3.5. Class Description: UiExportConfig

The export window which handles exporting vector and entry data to an external file.

Class Name: UiExportConfig	
Superclass: QDialog	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Collects specified directory. • Collects directory file format. • Collects name for exported file. • Creates an image file of vector graph. • Creates a csv file of vector table.	ExportGraph 1 ExportTable 1
No Contracts	

3.6. Class Description: UiFilterConfig

The filter window which handles the creation and application of a search filter.

Class Name: UiFilterConfig	
Superclass: QDialog	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Collects keywords to search for in generated log entries, nodes, and relationships. • Collects the start/end timestamp search-criteria to apply to generated log entries and nodes. • Collects the “creator” (red, white, or blue) search criteria for generated log entries and nodes. • Collects the “event type” (red, white, or blue) search-criteria for generated log entries and nodes.	
Contract: 1. Provide filter.	
Responsibilities	**Collaborations**
1. Knows keywords. *def get_keywords() -> List:* *returns: List containing keyword strings.* *Retrieves the keywords from the filter.* *Pre: None* *Post: Returned list is non-null.* 2. Knows timestamps. *def get_timestamps() -> List:* *returns: List containing QDateTime stamps.* *Retrieves the timstamp tamps from the filter.* *Pre: None* *Post: Returned list contains exactly one start datetime stamp and end datetime stamp.* 3. Knows “creators”.	

def get_creators() -> List: *returns: List containing selected creators.* *Retrieves selected creators from the filter.* *Pre: None* *Post: Returned list is non-null.* 4. Knows “event type”. *def get_event_types() -> List:* *returns: List containing selected event types.* *Retrieves selected event types from the filter.* *Pre: None* *Post: Returned list is non-null.*	

3.7. Class Description: UiTeamConfig

The team window which handles the system role and connections to the host when running as a client.

Class Name: UiTeamConfig	
Superclass: QDialog	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Displays lead status. • Displays host machine IP address. • Displays lead machine IP address. • Collects lead status. • Collects host machine IP address. • Collects lead machine IP address. • Displays number of connections to the lead machine IP address.	Settings 1 Settings 3 Settings 2 Settings 4 Settings 6 Settings 5
No Contracts	

3.8. Class Description: UiVectorConfig

The vector window which handles the adding, editing, and deleting of vectors and their descriptions.

Class Name: UiVectorConfig	
Superclass: QFrame	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Displays vector name. • Displays vector description. • Collects vector name. • Collects vector description. • Adds a new vector. • Deletes a vector. • Edits existing vector attributes.	IDDict 4; Vector 1 IDDict 4; Vector 2 IDDict 1 IDDict 2 IDDict 3; Vector 12-13

No Contracts

3.9. Class Description: UiRelationshipConfig

The relationship window which handles the relationship table for the active vector.

Class Name: UiRelationshipConfig	
Superclass: QFrame	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Display relationship id.	IDDict 4; Filter 1-4
• Display parent id.	IDDict 4; Relationship 1; Filter 1-4
• Display child id.	IDDict 4; Relationship 2; Filter 1-4
• Display label.	IDDict 4; Relationship 3; Filter 1-4
• Collects parent id.	
• Collects child id.	
• Collects label description.	
• Adds a new relationship.	IDDict 1
• Deletes a relationship.	IDDict 2
• Edits existing relationship attributes.	IDDict 3; Relationship 5-7
No Contracts	

3.10. Class Description: UiVectorDBAnalyst

The analyst vector DB window which handles the pulling and pushing of vector changes between the analyst and host systems.

Class Name: UiVectorDBAnalyst	
Superclass: QFrame	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Displays lead's pull table.	Sync 2; IDDict 3; Vector 1-2; Graph 5-6
• Displays client's push table.	Vector 1-2; Graph 5-6
• Initiates pull request from lead's table.	Sync 2; IDDict 3; Vector 1-2
• Initiates push request from client's table.	Sync 1; IDDict 3; Vector 1-2
No Contracts	

3.11. Class Description: UiVectorDBLead

The lead vector DB window which handles the accepting of queued analyst push requests.

Class Name: UiVectorDBLead	
Superclass: QFrame	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Knows pending push requests from clients.	Sync 3
• Accept push requests to the database.	ProjectMerge 1, Sync 4
• Declines push requests to the database.	Sync 4
• Display information to be pushed to the database.	Vector 1-2; Graph 5-6

No Contracts

3.12. Class Description: UiChangeConfig

The commit window which handles the committing of log entry and vector changes.

Class Name: UiChangeConfig	
Superclass: QDialog	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Display vector changes.	History 1
• Display node changes.	History 1
• Display relationship changes.	History 1
• Saves vector changes.	History 2
• Saves node changes.	History 2
• Saves relationship changes.	History 2
No Contracts	

4. Adversarial Assessment Data Processing: Detailed Description

4.1. Component Description

The Adversarial Assessment Data Processing subsystem is responsible for ingesting user-provided files, validating them, sending them to Splunk for processing, and retrieving the Splunk-generated log entries for analyst processing and manipulation in the PICK system. This subsystem includes the LogFile, LogEntry, Validator, EnforcementActionReport, and SplunkManager classes.

4.2. Class Description: LogFile

The LogFile class is responsible for reading the user-provided directory path and importing the contents of the file.

Class Name: LogFile	
Superclass: None	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Creates copy of cleansed file. • Records ingestion status. • Records log file name. • Records file path.	
Contract: 2. Provide file status.	
Responsibilities	**Collaborations**
1. Knows cleansing status. *def get_cleansing_status() -> bool:* *returns: True if file was cleansed or required no cleansing, False if cleansing failed or has not been cleansed yet.* *Retrieves the cleansing status of the logfile.* *Pre: None* *Post: Returned value is either True or False.* 2. Knows validation status. *def get_validation_status() -> bool:* *returns: True if file passed validation, False if file failed validation or has not been validated yet.* *Retrieves the validation status of the logfile.* *Pre: None* *Post: Returned value is either True or False.* 3. Knows ingestion status. *def get_ingestion_status() -> bool:* *returns: True if file has been sent for ingestion, False if file has not been ingested yet.* *Retrieves the ingestion status of the logfile.* *Pre: None* *Post: Returned value is either True or False.*	

4. Knows log file name. *def get_name() -> str:* *returns: String representing the logfile name.* *Retrieves the name from this logfile.* *Pre: None* *Post: Returned string is non-null.* 5. Knows file path. *def get_file_path() -> str:* *returns: String representing the logfile filepath.* *Retrieves the filepath from this logfile.* *Pre: None* *Post: Returned string is non-null.* 6. Knows enforcement action report. *def get_ear() -> EnforcementActionReport:* *returns: EnforcementActionReport for this logfile.* *Retrieves the EnforcementActionReport from this logfile.* *Pre: None* *Post: Returned EnforcementActionReport is non-null.*	
Contract: 3. Save file status.	
Responsibilities	**Collaborations**
7. Records cleaning status. *def set_cleansing_status (status: bool):* *status: Boolean representing the status of cleansing.* *Sets the cleansing status of the logfile.* *Pre: status is non-null.* *Post: Logfile attribute cleansed is set to status.* 8. Records validation status. *def set_validation_status (status: bool):* *status: Boolean representing the status of validation.* *Sets the validation status of the logfile.* *Pre: status is non-null.* *Post: Logfile attribute validated is set to status.* 9. Records enforcement action report. *def set_ear(ear: EnforcementActionReport):* *name: String representing the event name.* *Sets the Enforcement Action Report of the logfile.* *Pre: ear is non-null.* *Post: Logfile attribute ear is set to ear.*	

4.3. Class Description: LogEntry

The LogEntry class is responsible for storing the indexed entries in a log file.

Class Name: LogEntry
Superclass: None
Subclasses: None
Private Responsibilities: None
Contract: 4. Provide entry details.

Responsibilities	**Collaborations**
1. Knows entry timestamp. *def get_timestamp() -> str:* *returns: String representing the entry timestamp.* *Retrieves the timestamp for this entry.* *Pre: None* *Post: Returned string is non-null.* 2. Knows entry description. *def get_description() -> str:* *returns: String representing the entry description.* *Retrieves the description from this entry.* *Pre: None* *Post: Returned string is non-null.* 3. Knows entry line number. *def get_line_number() -> int:* *returns: Integer representing the line number.* *Retrieves the line number from this entry.* *Pre: None* *Post: Returned integer is non-negative.* 4. Knows log file path. *def get_log_file_path() -> str:* *returns: String representing the entry log file path.* *Retrieves the log file path from this entry.* *Pre: None* *Post: Returned string is non-null.*	

4.4. Class Description: Validator

The Validator class is responsible for validating and cleansing the content of user-provided files.

Class Name: Validator	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 5. Cleanse and validate log files.	
Responsibilities	**Collaborations**
1. Cleanses log files of empty lines/rows and unwanted TMUX characters. *def cleanse(logfile: LogFile):* *Cleanses logfile of empty rows and unwanted TMUX characters.* *Pre: logfile is non-null.* *Post: logfile is non-null and does not contain any empty rows or unwanted TMUX characters if applicable. An Enforcement Action Report is generated if the process could not be completed. The logfile's cleansing status is set.*	LogFile 7, 9
2. Validates log files' timestamps. *def validate(logfile: LogFile):*	LogFile 8-9; Event 3-4

Validates the logfile against the start and end timestamps. *Pre: logfile is non-null.* *Post: An Enforcement Action Report is generated if the log file failed validation. The logfile's validation status is set.*	

4.5. Class Description: EnforcementActionReport

The EnforcementActonReport is responsible for a report that outlines various errors found in validation.

Class Name: EnforcementActionReport	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 6. Provide enforcement action report.	
Responsibilities	**Collaborations**
1. Knows errors found within log file. *def generate_ear(file_path: str) -> dict:* *returns: Dictionary of line number : error message found within file.* *Retrieves the list of errors.* *Pre: None* *Post: If no errors found returned list is null,* *otherwise non-null list is returned.*	

4.6. Class Description: SplunkManager

The SplunkManager class is responsible for the interface between Splunk and the system to submit log files and retrieve log entries.

Class Name: SplunkManager	
Superclass: None	
Subclasses: None	
Private Responsibilities: • Know configuration details • Connect to Splunk • Create index for log entry storage	
Contract: 7. Generate log entries from Splunk.	
Responsibilities	**Collaborations**
1. Sends log file to Splunk. *def add_file(file_path: str, index_name: str):* *Add file to Splunk index.* *Pre: file_path is valid.* *index_name is valid.* *Post: Log file log entries are stored on Splunk index.* 2. Retrieves parsed log entries. *def search(query: str) -> dict:* *dict: Dictionary representing the query results.* *Retrieves the log entries from Splunk.* *Pre: Query is valid.* *Post: Returned dictionary is non-null.*	

5. Internal Data Store: Detailed Description

5.1. Component Description

The Internal Data Store subsystem is responsible for the maintenance and persistence of user-provided, configurable items. Such items can include customized event names, vector names, and any other labels or data that the system allows the user to personalize. This subsystem includes the Main, Event, Settings, Dictionary, QObject, Vector, ActiveVector, Node, Relationship, History, ProjectMerge, ExportTable, and ExportGraph classes.

5.2. Class Description: Main

Serves as an entry and exit point to the system.

Class Name: Main	
Superclass: None	
Subclasses: None	
Private Responsibilities:	**Collaborations**
• Loads the event object from file.	Event 10
• Loads the settings object from file.	Settings 18
• Saves the settings object to file.	Settings 17
• Loads the vector IDDict from file.	IDDict 6
• Loads the log file IDDict from file.	IDDict 6
• Loads the log entry IDDict from file.	IDDict 6
• Saves the log file IDDict to file.	IDDict 5
• Saves the log entry IDDict to file.	IDDict 5
• Launches UI.	
No Contracts	

5.3. Class Description: Event

The Event class is responsible for maintaining the information and attributes that describe the specific event for an analysis/session.

Class Name: Event	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 8. Provide event details.	
Responsibilities	**Collaborations**
1. Knows user-provided event name. *def get_name() -> str:* *returns: String representing the event name.* *Retrieves the name from this event.* *Pre: None* *Post: Returned string is non-null.* 2. Know user-provided event description. *def get_desc() -> str:* *returns: String representing the event description.* *Retrieves the description from this event.*	

<table>
<tr><td>*Pre: None*
Post: Returned string is non-null.

3. Knows user-provided start timestamp.
def get_start_time() -> QDateTime:
returns: QDateTime representing the start timestamp.
Retrieves the start timestamp from this event.
Pre: None
Post: Returned QDateTime is non-null.

4. Knows user-provided end timestamp.
def get_end_time() -> QDateTime:
returns: QDateTime representing the end timestamp.
Retrieves the end timestamp from this event.
Pre: None
Post: Returned QDateTime is non-null.</td><td></td></tr>
<tr><td colspan="2">**Contract**: 9. Save event details.</td></tr>
<tr><td>**Responsibilities**</td><td>**Collaborations**</td></tr>
<tr><td>5. Records user-provided event name.
def set_name(name: str):
name: String representing the event name.
Sets the name of this event.
Pre: name is non-null.
Post: name is set to event’s attribute name.

6. Records user-provided event description.
def set_desc(desc: str):
desc: String representing the event description.
Sets the description of this event.
Pre: desc is non-null.
Post: desc is set to event’s attribute desc.

7. Records user-provided start timestamp.
def set_start_time(start: QDateTime):
start: QDateTime representing the start timestamp.
Sets the start timestamp of this event.
Pre: start is non-null.
Post: If start is set before attribute end_timestamp, start is set to attribute start_timestamp; otherwise, start_timestamp is set to end_timestamp.

8. Records user-provided end timestamp.
def set_end_time(end: QDateTime):
end: QDateTime representing the end timestamp.
Sets the end timestamp of this event.
Pre: end is non-null.
Post: If end is set after attribute start_timestamp, end is set to attribute end_timestamp; otherwise, end_timestamp is set to start_timestamp.</td><td></td></tr>
<tr><td colspan="2">**Contract**: 10. Save-load event object.</td></tr>
</table>

Responsibilities	**Collaborations**
9. Saves itself to file. *def save():* *Saves this event object to a file.* *Pre: None* *Post: Settings object binary is written to a file in ./cache/event.pk; throws IOException.* 10. Loads itself from file. *def load():* *Loads an event object from a file.* *Pre: None* *Post: If file './cache/event.pk' exists, settings object is initialized from the file; if file does not exist, nothing happens; throws IOException.*	

5.4. Class Description: Settings

The Settings class is responsible for housing global variables of a given event/session.

Class Name: Settings	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 11. Provide networking details.	
Responsibilities	**Collaborations**
1. Knows lead status check. *def get_lead_status() -> bool:* *returns: True if the lead checkbox is checked, false otherwise.* *Checks whether the lead IP address check box is checked.* *Pre: None* *Post: Returned boolean is non-null.* 2. Knows lead machine IP address. *def get_lead_IP() -> str:* *returns: String representing lead machine's IP address.* *Retrieves the recorded, lead IP address* *Pre: None* *Post: Returned string is non-null.* 3. Knows host machine IP address. *def get_host_IP () -> str:* *returns: String representing the host machine's IP address.* *Retrieves the current host's IP address.* *Pre: None* *Post: Returned string is non-null.*	
Contract: 12. Save networking details.	
Responsibilities	**Collaborations**
4. Records lead status check. *def set_lead_status(status: bool):*	

status: Boolean condition which sets the lead's status either true or false. *Sets the lead status for the session.* *Pre: status is non-null.* *Post: status is set to settings attribute lead_status.* 5. Records lead machine IP address. *def set_lead_IP(lead_IP: str):* *lead_IP: String representing the lead's IP address.* *Records the IP address of the lead for a given session.* *Pre: lead_IP is non-null.* *Post: lead_IP is set to settings attribute lead_IP_address.* 6. Records host machine IP address. *def set_host_IP(host_IP: str):* *host_IP: String representing the host's IP address.* *Sets the host machine's IP address.* *Pre: host_IP is non-null.* *Post: host_IP is set to settings attribute host_IP_address.*	
Contract: 13. Provide directory details.	
Responsibilities	**Collaborations**
7. Knows red team directory. *def get_red_directory() -> str:* *returns: String representing the red team's directory path.* *Retrieves the red team's directory path.* *Pre: None* *Post: Returned string is non-null.* 8. Knows blue team directory. *def get_blue_directory() -> str:* *returns: String representing the blue team's directory path.* *Retrieves the blue team's directory path.* *Pre: None* *Post: Returned string is non-null.* 9. Knows white team directory. *def get_white_directory() -> str:* *returns: String representing the white team's directory path.* *Retrieves the white team's directory path.* *Pre: None* *Post: Returned string is non-null.* 10. Knows root directory. *def get_root_directory() -> str:* *returns: String representing the root directory path.* *Retrieves the root directory's path.* *Pre: None* *Post: Returned string is non-null.*	

11. Knows icon directory. *def get_icon_directory() -> str:* *returns: String representing the icon directory path.* *Retrieves the icon directory's path.* *Pre: None* *Post: Returned string is non-null.*	
Contract: 14. Save directory details.	
Responsibilities	**Collaborations**
12. Records red team directory. *def set_red_directory(name: str):* *name: String representing the red team's directory name.* *Sets the red team's directory path.* *Pre: name is non-null.* *Post: name is set to settings attribute red_directory.* 13. Records blue team directory. *def set_blue_directory(name: str):* *name: String representing the blue team's directory name.* *Sets the blue team's directory path.* *Pre: name is non-null.* *Post: name is set to settings attribute blue_directory.* 14. Records white team directory. *def set_white_directory(name: str):* *name: String representing the white team's directory name.* *Sets the white team's directory path.* *Pre: name is non-null.* *Post: name is set to settings attribute white_directory.* 15. Records root directory. *def set_root_directory(name: str):* *name: String representing the root directory name.* *Sets the root team's directory path.* *Pre: name is non-null.* *Post: name is set to settings attribute root_directory.* 16. Records icon directory. *def set_icon_directory(name: str):* *name: String representing the icon directory name.* *Sets the icon team's directory path.* *Pre: name is non-null.* *Post: name is set to settings attribute icon_directory.*	
Contract: 15. Save-load settings object.	
Responsibilities	**Collaborations**
17. Saves itself to file. *def save():* *Saves this settings object to a file.* *Pre: None* *Post: Settings object binary is written to a file in ./cache/settings.pk; throws IOException.*	

18. Loads itself from file. *def load():* *Loads a settings object from a file.* *Pre: None* *Post: If file './cache/settings.pk' exists, settings object is initialized from the file; if file does not exist, nothing happens; throws IOException.*	

5.5. Class Description: IDDict

The IDDict class is responsible for storing a map of generic objects and their unique id.

Class Name: IDDict	
Superclass: QObject	
Subclasses: None	
Private Responsibilities: • Maps a generated uid for each entry.	
Contract: 16. Modify dictionary.	
Responsibilities	**Collaborations**
1. Adds an entry to the dictionary. *def add(obj: Any):* *obj: an object to insert.* *Adds new object to dictionary.* *Pre: object is non-null.* *Post: A new object has been inserted into dictionary.* *Add signal is emitted.* 2. Removes an entry from the dictionary. *def delete(uid: str):* *uid: the unique id of the object.* *Removes the object provided the uid.* *Pre: uid exists in dictionary.* *Post: Object is no longer in dictionary.* *Delete signal is emitted.* 3. Retrieves an entry from the dictionary. *def get(uid: str) -> Any:* *uid: the unique id of the object.* *returns: Object of the associated uid.* *Retrieves object given uid.* *Pre: uid exists in dictionary.* *Post: Returned object is non-null.* 4. Retrieves all entries from the dictionary. *def items() -> list:* *returns: List of key-value pair tuples (str: Any).* *Retrieves all the items in the dictionary.* *Pre: None.* *Post: Returned list is non-null.*	
Contract: 17. Save-load dictionary object.	
Responsibilities	**Collaborations**

5. Saves IDDict to file. *def save():* *Saves this dictionary object to a file.* *Pre: None* *Post: Dictionary object binary is written to a file in ./cache/dicitonary.pk; throws IOException.* 6. Loads IDDict from file. *def load():* *Loads a dictionary object from a file.* *Pre: None* *Post: If file './cache/dictionary.pk' exists, dictionary object is initialized from the file; if file does not exist, nothing happens; throws IOException.*	

5.6. Class Description: Vector

The vector class is responsible for storing the data of nodes and relationships that compose a vector.

Class Name: Vector	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 18. Provide vector details.	
Responsibilities	**Collaborations**
1. Knows vector name. *def get_name() -> str:* *returns: String representing the vector name.* *Retrieves the name from this vector.* *Pre: None* *Post: Returned string is non-null.* 2. Knows vector description. *def get_desc() -> str:* *returns: String representing the vector description.* *Retrieves the description from this vector.* *Pre: None* *Post: Returned string is non-null.* 3. Knows node id property visibility flag. *def get_node_id_visibility() -> bool:* *returns: True if set, false otherwise.* *Determines if node id visibility is set.* *Pre: None* *Post: Returned value is either True or False.*	

4. *Know node name* property visibility flag.
def get_node_name_visibility() -> bool:
returns: True if set, false otherwise.
Determines if node name visibility is set.
Pre: None
Post: Returned value is either True or False.

5. *Knows node timestamp* property visibility lag.
def get_node_time_visibility() -> bool:
returns: True if set, false otherwise.
Determines if node timestamp visibility is set.
Pre: None
Post: Returned value is either True or False.

6. *Knows node description* property visibility flag.
def get_node_desc_visibility() -> bool:
returns: True if set, false otherwise.
Determines if node description visibility is set.
Pre: None
Post: Returned value is either True or False.

7. *Knows node log entry reference* property visibility flag.
def get_log_entry_visibility() -> bool:
returns: True if set, false otherwise.
Determines if log entry reference visibility is set.
Pre: None
Post: Returned value is either True or False.

8. *Knows node log creator* property visibility flag.
def get_log_creator_visibility() -> bool:
returns: True if set, false otherwise.
Determines if log creator visibility is set.
Pre: None
Post: Returned boolean value in non-null.

9. *Knows node event type* property visibility flag.
def get_event_type_visibility() -> bool:
returns: True if set, false otherwise.
Determines if event type visibility is set.
Pre: None
Post: Returned value is either True or False.

10. *Knows node icon type* property visibility flag.
def get_icon_type_visibility() -> bool:
returns: True if set, false otherwise.
Determines if icon type visibility is set.
Pre: None
Post: Returned value is either True or False.

11. Knows node source property visibility flag. *def get_source_visibility() -> bool:* *returns: True if set, false otherwise.* *Determines if source visibility is set.* *Pre: None* *Post: Returned value is either True or False.*	
Contract: 19. Save vector details.	
Responsibilities	**Collaborations**
12. Records vector name. *def set_name(name: str):* *name: String representing the vector name.* *Sets the name of this vector.* *Pre: name is non-null.* *Post: name is set to vector attribute name.* 13. Records vector description. *def set_desc(desc: str):* *name: String representing the vector description.* *Sets the description of this vector.* *Pre: desc is non-null.* *Post: desc is set to vector attribute description.* 14. Records node id property visibility flag. *def toggle_node_id_visibility() -> bool:* *Toggles the node id visibility.* *Pre: None* *Post: Property visibility is updated.* *Property signal updated is emitted.* *15. Records node name* property visibility flag. *def toggle_node_name_visibility():* *Toggles the node name visibility.* *Pre: None* *Post: Property visibility is updated.* *Property signal updated is emitted.* *16. Records node timestamp* property visibility lag. *def toggle_node_time_visibility() -> bool:* *Toggles the node timestamp visibility.* *Pre: None* *Post: Property visibility is updated.* *Property updated signal is emitted.* *17. Records node description* property visibility flag. *def toggle_node_desc_visibility() -> bool:* *Toggles the node description visibility.* *Pre: None* *Post: Property visibility is updated.* *Property updated signal is emitted.*	

18. Records node log entry reference property visibility flag. *def toggle_log_entry_visibility()* *Toggles the log entry visibility.* *Pre: None* *Post: Property visibility is updated.* *Property updated signal is emitted.* *19. Records node log creator* property visibility flag. *def toggle_log_creator_visibility()* *Toggles the log creator visibility.* *Pre: None* *Post: Property visibility is updated.* *Property updated signal is emitted.* *20. Records node event type* property visibility flag. *def toggle_event_type_visibility():* *Toggles the event type visibility.* *Pre: None* *Post: Property visibility is updated.* *Property updated signal is emitted.* *21. Records node icon type* property visibility flag. *def toggle_icon_type_visibility():* *Toggles the icon type visibility.* *Pre: None* *Post: Property visibility is updated.* *Property updated signal is emitted.* *22. Records node source* property visibility flag. *def toggle_source_visibility():* *Toggles the source visibility.* *Pre: None* *Post: Property visibility is updated.* *Property updated signal is emitted.*	
Contract: 20. Modify node dictionary.	
Responsibilities	**Collaborations**
23. Add node to dictionary. *def add_node() -> str:* *returns: String representing the node's uid.* *Adds a new node to the dictionary.* *Pre: None* *Post: Returned string is non-null.* *Dictionary contains a new node.*	IDDict 1
24. Remove node from dictionary. *def delete_node(node_id: str):* *node_id: String representing the node's uid.* *Removes a node corresponding to node_id from the dictionary.* *Pre: node_id exists in the dictionary.*	IDDict 2

Post: Node with node_id is no longer in the dictionary.	
25. Retrieve node form dictionary. *def get_node(node_id: str) -> Node* *returns: Node object corresponding to node_id.* *node_id: String representing the node's uid.* *Retrieves a node corresponding to node_id from the dictionary.* *Pre: Node id exists in the dictionary.* *Post: Returned Node is non-null.*	IDDict 3
26. Retrieve all nodes from dictionary. *def node_items() -> List:* *returns: List of key-value tuples (str: Node).* *Retrieves all nodes from the dictionary.* *Pre: None* *Post: Returned list is non-null.*	IDDict 4
Contract: 21. Modify relationship dictionary.	
Responsibilities	**Collaborations**
27. Add relationship to dictionary. *def add_relationship() -> str:* *returns: String representing the relationsip's uid.* *Adds a new relationship to the dictionary.* *Pre: None* *Post: Returned string is non-null.* *Dictionary contains a new relationship.*	IDDict 1
28. Remove relationship from dictionary. *def delete_relationship(relationship_id: str):* *relationship_id: String representing the relationsip's uid.* *Removes relationship from the dictionary.* *Pre: relationship_id exists in the dictionary.* *Post: Relationship with relationship_id is no longer in the dictionary.*	IDDict 2
29. Retrieve relationship from dictionary. *def get_relationship(relationship_id: str) -> Relationship* *returns: Relationship object corresponding with relationship_id.* *relationship_id: String representing the relationsip's uid.* *Retrieves a relationsip corresponding to relationship_id from the dictionary.* *Pre: relationship_ id exists in the dictionary.* *Post: Returned Relationsip is non-null.*	IDDict 3
30. Retrieve all relationships from dictionary. *def relationship_items() -> List:* *returns: list of key-value tuples (relationship id: relationship).* *Retrieves all relationships from the dictionary.*	IDDict 4

Pre: None *Post: Returned List non-null.*	

5.7. Class Description: ActiveVector

The ActiveVector class is responsible for storing the currently selected vector and its unique id.

Class Name: ActiveVector	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 22. Provide active vector details.	
Responsibilities	**Collaborations**
1. Knows active vector. *def get_vector() -> Vector:* *returns: Vector of currently selected (active) vector.* *Retrieves the ActiveVector.* *Pre: None* *Post: Returned object is a vector if non-null.* 2. Knows active vector id. *def get_uid() -> str:* *returns: String representing the Active Vector id.* *Retrieves the uid from the Active Vector.* *Pre: None* *Post: Returned string is non-null.*	
Contract: 23. Save active vector details.	
Responsibilities	**Collaborations**
3. Records active vector. *def set_vector(vector: Vector):* *vector: Vector to be set as active.* *Sets the active vector.* *Pre: vector is non-null.* *Post: vector is set to ActiveVector attribute vector.* 4. Records active vector id. *def set_uid(uid: str):* *name: String representing the ActiveVector id.* *Sets the ActiveVector id.* *Pre: uid is non-null.* *Post: uid is set to ActiveVector attribute vector_id.*	

5.8. Class Description: Node

The Node class is responsible for the storage of log entry properties.

Class Name: Node	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 24. Provide node details.	
Responsibilities	**Collaborations**
1. Knows node name.	

def get_name() -> str:
returns: String representing the node name.
Retrieves the name from this node.
Pre: None
Post: Returned string is non-null.

2. Knows node timestamp.

def get_timestamp() -> QDateTime:
returns: QDateTime representing the node's timestamp.
Retrieves the timestamp from this node.
Pre: None
Post: Returned QDateTime is non-null.

3. Knows node description.

def get_description() -> str:
returns: String representing the node description.
Retrieves the description from this node.
Pre: None
Post: Returned string is non-null.

4. Knows node log entry reference.

def get_log_entry() -> str:
returns: String representing the log entry reference.
Retrieves the log_entry from this node.
Pre: None
Post: Returned string is non-null.

5. Knows node log creator.

def get_log_creator() -> str:
returns: String representing the person that created the log.
Retrieves the log_creator from this node.
Pre: None
Post: Returned string is non-null.

6. Knows node event type.

def get_event_type() -> str:
returns: String representing the event type.
Retrieves the event_type from this node.
Pre: None
Post: Returned string is non-null.

7. Knows node icon type.

def get_icon_type() -> str:
returns: String representing the type of icon the node will use.
Retrieves the icon_type from this node.
Pre: None
Post: Returned string is non-null.

8. Knows node source filename.

def get_source_file_name() -> str:

returns: String representing the source file name. *Retrieves the source_file_name from this event.* *Pre: None* *Post: Returned string is non-null.* 9. Knows node visibility. *def get_visibility() -> bool:* *returns: Boolean representing the node visibility.* *Retrieves whether this node shall be visible or not.* *Pre: None* *Post: Returned value is either True or False.* 10. Knows node coordinate position. *def get_coordinate_position() -> List:* *returns: Integer list representing the position of the node on the graph view.* *Retrieves the coordinate position from this node.* *Pre: None* *Post: Returned List is exactly 2 integers.*	
Contract: 25. Save node details.	
Responsibilities	**Collaborations**
11. Records node name. *def set_name(name: str):* *name: String representing the node name.* *Sets the name of this node.* *Pre: name is non-null.* *Post: name attribute is set to the given string.* 12. Records node timestamp. *def set_timestamp(timestamp: QDateTime):* *timestamp: datetime representing the node's time.* *Sets the timestamp of this node.* *Pre: timestamp is non-null.* *Post: timestamp attribute is set to the given QDateTime.* 13. Records node description. *def set_description(description: str):* *description: String representing the node's description.* *Sets the description of this node.* *Pre: description is non-null.* *Post: description attribute is set to the given string.* 14. Records node log entry reference. *def set_log_entry_reference(log_entry_reference: str):* *log_entry_reference: String representing the node's log entry reference.* *Sets the log_entry_reference of this node.* *Pre: log_entry_reference is non-null.* *Post: log_entry_reference attribute is set to the given string.*	

15. Records node log creator. *def set_log_creator(log_creator: str):* *log_creator: String representing the node's log creator.* *Sets the log_creator of this node.* *Pre: log_creator is non-null.* *Post: log_creator attribute is set to given string.* 16. Records node event type. *def set_event_type(event_type: str):* *event_type: String representing the node's event type.* *Sets the event_type of this node.* *Pre: event_type is non-null.* *Post: event_type attribute is set to the given string.* 17. Records node icon type. *def set_icon_type(icon_type: str):* *icon_type: String representing the node's icon type.* *Sets the icon_type of this event.* *Pre: icon_type is non-null.* *Post: icon_type attribute is set to the given string.* 18. Records node source filename. *def set_filename (filename: str):* *filename: String representing the node's filename.* *Sets the filename of this event.* *Pre: filename is non-null.* *Post: filename is set to settings attribute name.* 19. Records node visibility. *def set_visibility(name: bool):* *name: Bool representing the node's visibility.* *Sets the visibility of this node.* *Pre: visibility is non-null.* *Post: visibility attribute is set to the given boolean.* 20. Records node coordinate position. *def set_coordinate_position(coordinate_postition: List):* *coordinate_postition: Integer list representing the coordinate position of the node in the graph view.* *Sets the coordinate_postition of this event.* *Pre: coordinate_postition is non-null.* *Post: coordinate_position attribute is set to the given list.*	

5.9. Class Description: Relationship

The Relationship class is responsible for storing the child and parent node IDs that constitute a given relationship in the system.

Class Name: Relationship
Superclass: None
Subclasses: None
Private Responsibilities: None

Contract: 26. Provide parent-child node relationship details.	
Responsibilities	**Collaborations**
1. Knows parent node identification number. *def get_parent() -> str:* *returns: String representing the parent node uid.* *Retrieves the uid from the parent Node.* *Pre: None* *Post: Returned string is non-null.* 2. Knows child node identification number. *def get_child() -> str:* *returns: String representing the child node uid.* *Retrieves the uid from the child Node.* *Pre: None* *Post: Returned string is non-null.* 3. Knows relationship label. *def get_label() -> str:* *returns: String representing the relationship label.* *Retrieves the name of the relationship label.* *Pre: None* *Post: Returned string is non-null.* 4. Knows relationship line coordinates. *def get_coordinates() -> List:* *returns: Integer list representing the endpoints of the relationship line.* *Retrieves a list of integers for relationship's line endpoints.* *Pre: None* *Post: Returned List is exactly 2 integers.*	
Contract: 27. Save parent-child node relationship details.	
Responsibilities	**Collaborations**
5. Records parent node identification number. *def set_parent(parent: str):* *parent: String representing the parent node of the relationship.* *Sets the string of the parent.* *Pre: parent is non-null.* *Post: parent attribute is set to the given string.* 6. Records child node identification number. *def set_child(child: str):* *child: String representing the child node of the relationship.* *Sets the string of the child.* *Pre: child is non-null.* *Post: child attribute is set to the given string.* 7. Records relationship label. *def set_label(label: str):* *label: String representing the relationship label.*	

Sets the name of this event. *Pre: label is non-null.* *Post: label attribute is set to the given string.* 8. Records relationship line coordinates. *def set_coordinates(coordinates: List):* *coordinates: Integer list representing the endpoints of the relationship line.* *Sets the integer list of this relationship.* *Pre: coordinates is non-null.* *Post: coordinates attribute is set to the given integer list.*	

5.10. Class Description: History

The History class is responsible for storing the vectors', nodes' and relationships' history of changes and saving them.

Class Name: History	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 28. Display history of changes.	
Responsibilities	**Collaborations**
1. Displays changes to vector, node, and relationship tables. *def get_changes() -> List:* *returns: A list which outlines the most recent changes to the vector, node, or relationship tables.* *Retrieves the detailed changes to either vector, node or relationship table.* *Pre: None* *Post: Returned list is non-null.*	IDDict 4; Vector 1-11, 26, 30; Node 1-10; Relationship 1-4
Contract: 29. Save changes.	
Responsibilities	**Collaborations**
2. Saves changes made to vector, node, and relationship tables. *def set_changes(changes: List):* *changes: List consisting of formatted message with changes to vector, node, or relationship tables.* *Pre: changes list is non-null.* *Post: Most recent change attribute is updated with changes list.*	IDDict 5

5.11. Class Description: ProjectMerge

The ProjectMerge class is responsible for the merging of analyst and lead history changes.

Class Name: ProjectMerge	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 30. Merge data.	
Responsibilities	**Collaborations**

1. Merges vector, node, and relationship data with stored vector, node, and relationship data. *def merge_data():* *Syncs retrieved data with database data.* *Pre: Data store must be established.* *Post: Data store is updated with retrieved data.*	Vector 1-22, 26, 30; Node 1-20; Relationship 1-8 History 1-2

5.12. Class Description: Sync

The Sync class is responsible for handling data transferal from lead and client.

Class Name: Sync	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 31. Provides push/pull request changes.	
Responsibilities	**Collaborations**
1. Pushes client changes to lead. *def push_client_changes(changes: List):* *changes: List of client's local changes*. *Provides detailed list to lead of client's local changes.* *Pre: List is non-null.* *Post: Lead now has new merge changes to accept/decline to push to DB.*	Settings 3
2. Pulls lead changes to client. *def pull_client_changes() -> List:* *returns: List providing itemized client changes.* *Pulls lead changes to client's local data store.* *Pre: None* *Post: Returned list is non-null.*	Settings 3
Contract: 32. Modify push request queue.	
Responsibilities	**Collaborations**
3. Queues push requests from clients. *def enqueue_push_request(push_request: List):* *push_request: A list that describes the various changes being pushed to the DB.* *Adds the push_request list to the end of the queue.* *Pre: push_request is non-null.* *Post: Client push request has been enqueued in chronological order*. 4. Removes push request from queue. *def pop_push_request() -> List:* *returns: List of the analyst changes that need to be pushed to DB.* *Provides the list of changes that need to be performed prior to sending changes to DB.* *Pre: None* *Post: Returned List is non-null.*	

5.13. Class Description: ExportTable

The ExportTable class is responsible for the formatting of the log entry tables that compose the vectors of a given event into an exportable file.

Class Name: ExportTable	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 33. Save tables to file.	
Responsibilities	**Collaborations**
1. Compiles the vector, node and relationship tables into a single, exportable file. *def table_export_file(file: str):* *file: String representing the filename and path in which to export to.* *Provides a single exportable file for the vector, node, and relationship tables.* *Pre: Tables must have information.* *Post: File is created and saved at the file path.*	Vector 1-2, 26, 30; Node 1-9; Relationship 1-3

5.14. Class Description: ExportGraph

The ExportGraph class is responsible for the formatting of the final graph with its corresponding vector into an exportable image file.

Class Name: ExportGraph	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 34. Save graph to image file.	
Responsibilities	**Collaborations**
1. Renders an exportable image file of the Graph. *def export_graph(vector: Vector, file: str):* *file: String representing the filename and path in which to export to.* *vector: The vector whose graph is to be rendered.* *Returns an image file of the vector's graph.* *Pre: Graph must be rendered.* *Post: File is created and saved at the file path.*	Graph 5-6

6. Graph: Detailed Description

6.1. Component Description

The Graph subsystem, which only includes one component Graph, is responsible for the displaying and the manipulation of the node and relationship data found within a vector.

6.2. Class Description: Graph

The Graph class is responsible for rendering nodes, relationships, their relative positions, and their properties for a given vector.

Class Name: Graph	
Superclass: None	
Subclasses: None	
Private Responsibilities: None	
Contract: 35. Provide graph details.	
Responsibilities	**Collaborations**
1. Knows graph orientation. *def get_orientation() -> int:* *returns: An integer encoding of the current orientation.* *Retrieves the orientation of this graph.* *Pre: None* *Post: Returned int is either 0 for horizontal, or 1 for vertical.* 2. Knows timeline interval. *def get_time_interval() -> Tuple:* *returns: A Tuple containing the interval unit and amount.* *Retrieves the timeline interval of this graph.* *Pre: None* *Post: Returned Tuple contains exactly one string and integer.*	
Contract: 36. Save graph details.	
Responsibilities	**Collaborations**
3. Records graph orientation. *def set_orientation(orientation: int):* *orientation: An integer encoding of the current orientation.* *Sets the orientation of this graph.* *Pre:orientation is either a 0 or 1.* *Post: orientation is set to graph attribute orientation.* 4. Records timeline interval. *def set_time_interval(time_interval: Tuple):* *time_interval: A Tuple containing the interval unit and amount.* *Sets the timeline interval of this graph.* *Pre: Tuple contains exactly one string and integer.* *Post: time_interval is set to graph attribute time_interval.*	
Contract: 37. Render-move graphical elements.	
Responsibilities	**Collaborations**

5. Renders nodes. *def render_nodes(vector: Vector):* *vector: Vector whose nodes are to be rendered.* *Renders nodes to be displayed in the graph view.* *Pre: None* *Post: Node rendering is serialized.*	Vector 1-11, 26; Node 1-10
6. Renders relationships. *def render_relationship(vector: Vector):* *vector: Vector whose relationships are to be rendered.* *Renders relationship to be displayed in the graph view.* *Pre: None* *Post: Relationship rendering is serialized.*	Vector 30; Relationship 1-4
7. Moves node. *def move_node(coord: List):* *coord: List of two integers representing the new coordinate position.* *Changes coordinates of the node. If the node has a relationship attached the coordinates of the relationship change.* *Pre: None* *Post: Node rendering is serialized.*	Vector 25; Node 20
8. Moves relationship. *def move_relationship(coord: List):* *coord: List of two integers representing the new coordinate position.* *Changes coordinates of the relationship to match the parent or child node.* *Pre: None* *Post: Relationship rendering is serialized.*	Vector 29; Relationship 8

[END]