We Showed Up

Prevent, Mitigate, and Recover (PMR) Insight Collective Knowledge System (PICK) Software Design Document

2.1

03/21/20

<team>

Document Control

Approval

The Guidance Team and the customer shall approve this document.

Document Change Control

Initial Release:	1.0
Current Release:	2.1
Indicator of Last Page in Document:	*
Date of Last Review:	03/06/2020
Date of Next Review:	03/31/2020
Target Date for Next Update:	03/31/2020

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. Roach
- Jake Lasley

Customer:
- Dr. Oscar Perez
- Vincent Fonseca
- Herandy Denisse Vazquez
- Baltazar Santaella
- Florencia Larsen
- Erick De Nava

Software Team Members:
- Daniela Garcia
- Ricardo Alvarez
- Diego Rincon
- Mathew Iglesias
- Jessica Redekop

Change Summary

The following table details changes made between versions of this document

Version	Date	Modifier	Description
1.0	03/06/2020	Jessica Redekop	Section 1: Introduction
1.1	03/08/2020	Jessica Redekop	Collaboration Diagrams
1.2	03/08/2020	Matthew Iglesias	Section 4: Database Schema Diagram
1.3	03/08/2020	Ricardo Alvarez	Section 3, Initial description of classes
1.4	03/08/2020	Diego Rincon	Section 2.2 & 2.3
1.5	03/09/2020	Daniela Garcia	CRC cards and Section 3 tables
1.6	03/09/2020	Ricardo Alvarez	Initial version of contract #1
1.7	03/09/2020	Jessica Redekop	Collaboration Graphs
1.8	03/29/2020	Jessica Redekop	Section 3.1: Contracts for Ingestion Subsystem
1.9	03/30/2020	Ricardo Alvarez	Section 3.1: Contracts pt.2 and Subsystem

			UML
2.0	03/30/2020	Matthew Iglesias	Section 3.3
2.1	03/30/2020	Daniela Garcia	Section 3.2: contracts for Graphing subsystem

Table of Contents

1. Introduction

1.1. Purpose and Intended Audience

The purpose of creating the Software Design Document (SDD) is to give the customer a clean and precise description of the system design of the Prevent, Mitigate, and Recover (PMR) Insight Collective Knowledge System (PICK). The SDD divides the system design specifications into the following three parts: The decomposition of the system, a collaboration description, and the detailed design of the system's subcomponents.

The intended audience of the SDD is Dr. Oscar Perez, Mr. Vincent Fonseca, Ms. Herandy Vazquez, Mr. Baltazar Santaella, Ms. Florencia Larsen, and the Software Engineering teams. This document serves as an agreement between both parties regarding the system to be developed.

1.2. Scope of Product

The Lethality, Survivability, and HSI Directorate (LSH) recognizes the complexity and the time it takes to analyze the applicable logs, observation notes, and other artifacts gathered from an adversarial assessment from the red, blue, and white teams and generate a report that presents the events that took place during the adversarial assessment. They want a system that would aid their analysts in correlating red team's activities to blue team's responses and represent the events that took place during an adversarial assessment graphically.

The University of Texas at El Paso (UTEP) and LSH are collaborating to develop Prevent, Mitigate, and Recover (PMR) Insight Collective Knowledge System (PICK) that will provide the ability to correlate red team's activities to blue team's responses and graphically represent the events that took place during an adversarial assessment. [1]

1.3. References

[1] R. Alvarez, D. Garcia, M. Iglesias, J. Redekop, D. Rincon, "PICK Responsibilities, and Collaborations."

[2] S. Roach, and E. T. Ramirez, "PICK Software Requirements and Specification."

1.4. Definitions, Acronyms, and Abbreviations

1.4.1. Definitions

The definitions in this section are given in the context of the product being developed. This intention is to assist the user in their understanding of the document.

Table 1: Definitions

TERM	DEFINITION
Cleansing	The removal of unwanted characters from uncleansed TMUX log files, blank rows from uncleansed excel log files, and blank lines from uncleansed log files.
Log Entry	Log files feed into and normalized through SPLUNK. Log entries can be filtered and edited by users of the system.
Timestamp	Denotes date, time, and section in the following format: month:date:year hours:minutes, and section in am/pm.

1.4.2. Acronyms

This section lists the acronyms used in this document and their associated definitions.

Table 2: Acronyms

TERM	DEFINITION
SDD	Software Design Documentation
SRS	System Requirements Specification
UTEP	The University of Texas at El Paso
PICK	Prevent, Mitigate, and Recover (PMR) Insight
LSH	Lethality, Survivability, and HSI Directorate
PMR	Prevent, Mitigate, and Recover
IP	Internet Address
AA	Adversarial Assessment
UI	User Interface
VCS	Version Control Subsystem

1.4.3. Abbreviations

This section provides a list of used abbreviations and their associated definitions.

Table 3: Abbreviations

TERM	DEFINITION

1.5. Overview

The SDD is divided into four major sections: Introduction (Section 1), Decomposition Description (Section 2), Dependency Description (Section 3), and Detailed Design (Section 4).

Section 1 includes five sections. Section 1.1 describes the purpose and intended audience. Section 1.2 describes the scope of the system. Section 1.3 contains the references used throughout the document. Section 1.4 defines the definitions, acronyms, and abbreviations used throughout the document. Section 1.5 is this overview of the SDD.

Section 2 includes four sections. Section 2.1 defines the scope of the decomposition descriptions of the system. Section 2.2 describes the use of the decompositions of the system. Section 2.3 has the subsystem description. Section 2.4 contains the hierarchy graphs of the system.

Section 3 includes three sections. Sections 3.1 describes the scope of the dependency descriptions of the system. Section 3.2 describes the use of the dependencies of the system. Section 3.3 contains the collaboration descriptions between the system.

Section 4 includes four sections. Section 4.1 defines the scope of the detailed design of the system components. Section 4.2 describes the use of the detailed design for the system. Section 4.3 describes the components of the system. Section 4.4 contains the database schemas for the system.

2. Decomposition Description

The following section depicts the system component collaboration diagram, identifies one subsystem and describes its components to identify which entity is responsible for specific functions, and finally describes how the component dependencies will impact development.

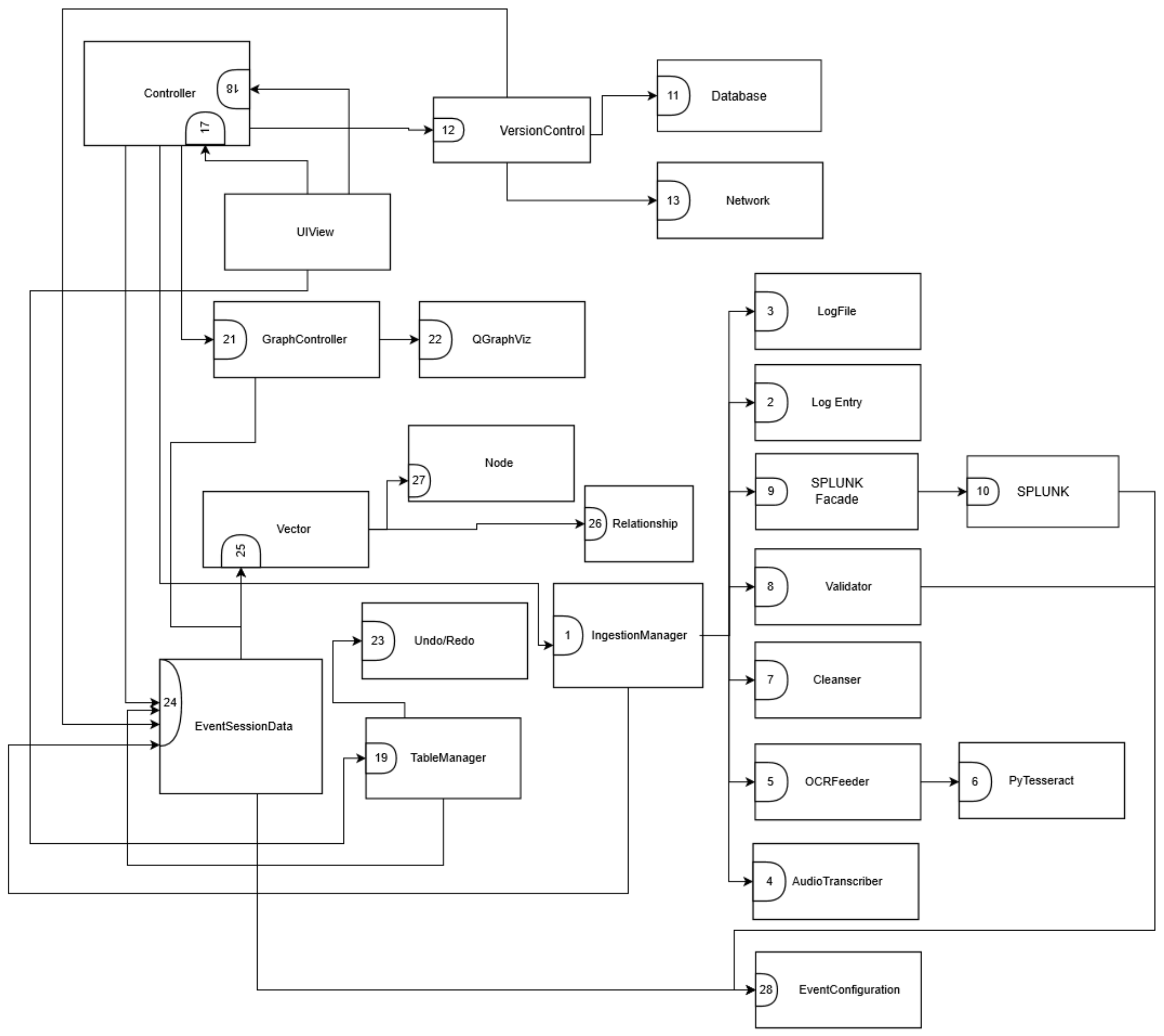
2.1. System Collaboration Diagram

<< Provide a UML Component Diagram or a Wirffs-Brock Collaboration Diagram. If this is a subsystem or part of a larger system, show the collaboration or component diagram for the entire system in a separate diagram first. Show the major components or subsystems in this system and indicate collaborations between components. If useful, show the UML class diagram that indicates class hierarchies.>>

<< Provide a description of the way the system has been structured and the major divisions between the design entities. Subsystems and classes are referred as design entities >>
The system contains all user interface (UI) items

2.1.1. UML System Component Diagram

Diagram 1: UML System Component Diagram



2.2. Subsystem and Component Descriptions

2.2.1. Ingestion

The ingestion subsystem cleanses and validates the log files before normalization, transcribes audio and video into text files, and feeds the validated log files into SPLUNK to convert them into log entries. The subsystem is composed of the classes IngestionManager, Validator, Cleanser, SPLUNK Façade, SPLUNK, AudioTranscriber, OCRFeeder, LogFile, PyTesseract and LogEntry. This subsystem consists of the following contracts:

- Contract 1: Gather files from Directories
- Contract 2: Transcribe Audio Files
- Contract 3: Transcribe Image Files
- Contract 4: Cleanse Files
- Contract 5: Validate Files
- Contract 6: Ingest Files
- Contract 7: Provide Data of an Occurrence
- Contract 8: Transcribe Image Log Files
- Contract 9: Send Log Files to SPLUNK

- Contract 10: Provide Log Entries from SPLUNK

A detailed description of the ingestion subsystem will be discussed in section 3.

2.2.2. VCS

The version control subsystem (VCS) tracks the changes in the vectors in the analysts' vector database, allows analysts to push and pull vector changes into the database, and allows the lead analyst to control the changes pushed to the database. The subsystem is composed of the classes VersionControl, Database, and Network. This subsystem consists of the following contracts:

- Contract 11: Establish database connection
- Contract 12: Track changes in database
- Contract 13: Store events in database
- Contract 14: Store vectors in database
- Contract 15: Store nodes in database
- Contract 16: Store relationships in database

2.2.3. User Interaction

The user interaction subsystem updates information displayed and embedded within the user interface, allowing for updating tables and graphs. The subsystem is composed of the classes Controller, MainUI, UndoRedoManager, TableManager, QGraphViz and GraphController. This subsystem consists of the following contracts:

- Contract 17: Update View
- Contract 18: Update Models Given User Input
- Contract 19: Update UI Tables
- Contract 20: Send Table Changes to Data Models
- Contract 21: Display Graph
- Contract 22: Create Graph
- Contract 23: Undo/Redo

2.2.4. User Data

The user data subsystem serves the users input data for the creation of events, nodes, vectors and relationships. The subsystem is composed of the classes EventSessionData, Node, Vector, Relationship and Event Configuration. This subsystem consists of the following contracts:

- Contract 24: Provide Event Data
- Contract 25: Provide Vectors
- Contract 26: Add Relationship
- Contract 27: Add Node
- Contract 28: Provides Event Configuration Data

2.3. Dependencies

The following section describes how the component dependencies will impact development.

1. **PyQT5:** This library serves as the main building block for the user interface, signaling and graph creation.
2. **QGraphViz:** This library serves as a bridge between the Qt objects and the code in order to create a movable/draggable graph.
3. **PyTesseract:** This library will be used to transcribe image files into text log files in order to be ingested.
4. **PocketSphinx:** This library will be used to transcribe audio files into text log files in order to be ingested.
5. **Splunk:** The Splunk service will be used to parse, store, retrieve and filter the log entries gathered from the log files.

6. **MongoDB:** The MongoDB service will be used as the main database to store the event data of the PICK Tool.

3. Detailed Descriptions

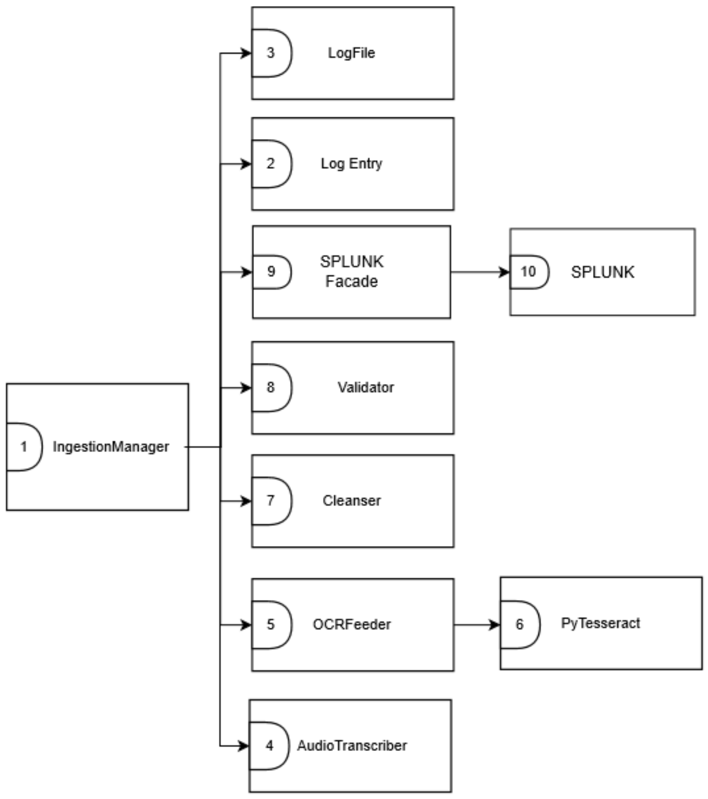
This section provides a detailed description of the identified subsystems of the PICK Tool, including their classes and contracts.

Ingestion Subsystem

This subsystem encloses all the actions taken by the PICK Tool in order to gather the logs from the root directory, cleanse, validate, upload to SPLUNK, and signal any errors pertaining to the log files. Due to these responsibilities, the subsystem includes the following classes:

1. IngestionManager
2. LogEntry
3. LogFile
4. AudioTranscriber
5. OCRFeeder
6. Pytesseract
7. Cleanser
8. Validator
9. SPLUNKFaçade
10. SPLUNK

Diagram 2: UML System Diagram of the Ingestion Subsystem



3.1.1. Class Descriptions for Ingestion Subsystem

3.1.1.1. Class Description IngestionManager

Class Name: IngestionManager	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Manage the ingestion process • Schedule cleansing, validation and uploading of log files to SPLUNK • Sets properties for directory once structural check operation is done	
Contract: 1. Gather files from directories	
Responsibilities	**Collaborations**
1. Retrieve files from each directory	EventSession
Contract: 2. Transcribe Audio Files	
Responsibilities	**Collaborations**
1. Transcribe audio files to text files	AudioTranscriber
Contract: 3. Transcribe Image Files	
Responsibilities	**Collaborations**
• Transcribe image files to text files	OCRFeeder
Contract: 4. Cleanse Files	
Responsibilities	**Collaborations**
1. Retrieve cleansed files from the text files	Cleanser
Contract: 5. Validate Files	
Responsibilities	**Collaborations**
1. Validate files by marking validation status and generating a list of errors if applicable.	Validator
Contract: 6. Ingest Files	
Responsibilities	**Collaborations**
2. Ingest processed files into splunk	SPLUNKFacade

3.1.1.2. Class Description LogFile

Class Name: LogFile
Superclass: N/A
Subclasses: N/A
Private Responsibilities • Know log file sourcetype • Know log file source • Know log file content • Know log file data type • Know log file path

3.1.1.2.1. Contract #2: Provide ingestion status of log files

Contract Description: Provides the ingestion status of each LogFile after each step in the ingestion process
Protocols:

1. **Provide Ingestion Status**
   - **Method name:** get_status()
   - **Pre-condition:**
     - The list of LogFile is initialized

- **Post-condition:**
    - N/A

3.1.1.2.2. Contract #8: Provide data of a Log File

Contract Description: Appends to the list of log files if any are found.

Protocols:

1. Get File Path

- **Method name:** add_errors(invalid_errors)
- **Pre-condition:**
    - An event is created.
- **Post-condition:**
    - The LogFile is returned with an appended error list. (If applicable)

3.1.1.3. Class Description LogEntry

Class Name: LogEntry	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • N/A	
Contract: 7. Provide data of an occurrence	
Responsibilities	**Collaborations**
1. Know log entry ID 2. Know timestamp 3. Know source 4. Know sourcetype 5. Know host 6. Know content description	

3.1.1.4. Class Description Audio Transcriber

Class Name: Audio Transcriber
Superclass: N/A
Subclasses: N/A
Private Responsibilities: Converts audio into a text file.

3.1.1.4.1. Contract #2: Transcribe Audio Files

Contract Description: The contract is between the IngestionManager and AudioTranscriber classes to transcribe an audio file and store the extracted text.

Protocols:

1. Transcribe Audio

- **Method name:** audio_transcribe(folder_path, new_path, filename)
- **Pre-condition:**
    - A list of LogFile is initialized
- **Post-condition:**
    - The file is transcribed and saved in the provided directory.

3.1.1.5. Class Description OCRFeeder

Class Name: TesseractOCR
Superclass: N/A

Subclasses: N/A	
Private Responsibilities • Transcribes an image file into text into a specified directory	
Contract: 8. Transcribe Image Log Files	
Responsibilities	**Collaborations**
1. Transcribe image files	Pytesseract

3.1.1.5.1. Contract #3: Transcribe Image Files

Contract Description: The contract is between the IngestionManager and AudioTranscriber classes to transcribe an image file and store the extracted text.
Protocols:
1. **Transcribe Audio**
    - **Method name:** def OCR_transcription(folder_path, new_path, f)
    - **Pre-condition:**
        - A list of LogFile is initialized
    - **Post-condition:**
        - The file is transcribed and saved in the provided directory.

3.1.1.6. Class Description Pytesseract

Class Name: Pytesseract
Superclass: N/A
Subclasses: N/A
Private Responsibilities • Converts image files to a text file.

3.1.1.6.1. Contract #8: Transcribe Image Log Files

Contract Description: Converts image log files into plain text.

Protocols:
1. **Convert Image Log Files**
    - **Method name:** image_to_string(image_file, lang="eng")
    - **Pre-condition:**
        - The event is created.
    - **Post-condition:**
        - The image is transcribed into text and stored in the specified directory.

3.1.1.7. Class Description Cleanser

Class Name: Cleanser
Superclass: N/A
Subclasses: N/A
Private Responsibilities • Cleanses empty rows of csv files. • Cleanses unwanted characters of any file.

3.1.1.7.1. Contract #4: Cleanse Files

Contract Description: Cleanses file from unwanted characters.
Protocols:
1. **Cleanse Log File**
    - **Method name:** def reader(filename, filepath)

- **Pre-condition:**
  - The list of LogFile is initialized
- **Post-condition:**
  - The file is cleansed from unwanted characters.

3.1.1.8. Class Description Validator

Class Name: Validator	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Knows interaction parameters for validation script calling • Knows the timestamp formats • Modifies validation status of the LogFile being validated • Appends to a list of errors of the LogFile if any are found	
Contract: 9. Append Errors	
Responsibilities	**Collaborations**
1. Appends to the list of log files if any are found	LogFile

3.1.1.8.1. Contract #5: Validate Files

Contract Description: Validates the LogFile and generates a list of error messages describing the errors in each line of the file.
Protocols:
1. Validate Log File
- **Method name:** validate_file(log_file):
- **Pre-condition:**
  - The LogFile is cleansed.
- **Post-condition:**
  - The LogFile has a valid or invalid status.
  - The invalid files are send to the Enforcement Action Report.

3.1.1.9. Class Description SPLUNKFacade

Class Name: SPLUNKFacade	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • N/A	
Contract: 9. Send log files to SPLUNK	
Responsibilities	**Collaborations**
1. Upload log files to be parsed and stored in SPLUNK	SPLUNK
Contract: 10. Provide SPLUNK log entries	
Responsibilities	**Collaborations**
2. Forward the SPLUNK log entries	SPLUNK

3.1.1.9.1. Contract #6: Ingest Files

Contract Description: Ingests the files through SPLUNK from the list of LogFile stored in IngestionManager into a specified event.
Protocols:

1. **Ingest Log Files**
    - **Method name:** add_file_to_index(log_file_path, index)
    - **Pre-condition:**
        - The LogFile has a validated ingestion status.
    - **Post-condition:**
        - The LogFile is ingested into the event.

3.1.1.10. Class Description for SPLUNK

Class Name: SPLUNK
Superclass: N/A
Subclasses: N/A
Private Responsibilities • Interface with SPLUNK through the python sdk

3.1.1.10.1. Contract #9: Send Log Files to SPLUNK

Contract Description: Sends the log files to a specific index in SPLUNK.

Protocols:

1. **Import Log File to Splunk**
    - **Method name:** import_log_file_to_splunk(log_file)
    - **Pre-condition:**
        - The log file has been cleansed.
        - The log file has been validated or the user has purposely skipped validation of it.
        - The event has been initialized.
        - The index pertaining to the event has been initialized.
        - The SPLUNK service has been initialized.
    - **Post-condition:**
        - The log entries inside the log file has been parsed and stored in the SPLUNK index.

3.1.1.10.2. Contract #10: Provide Log Entries from SPLUNK

Contract Description: Gets the log entries from the SPLUNK index pertaining to the event.

Protocols:

1. **Get SPLUNK Log Entries**
    - **Method name:** get_splunk_log_entries()
      Collaborators: EventConfig
    - **Pre-condition:**
        - The event has been initialized
        - The SPLUNK index of the event has been initialized
        - The SPLUNK service has been initialized.
    - **Post-condition:**
        - The log entries in the SPLUNK index of the event are gathered in a list of type LogEntry.

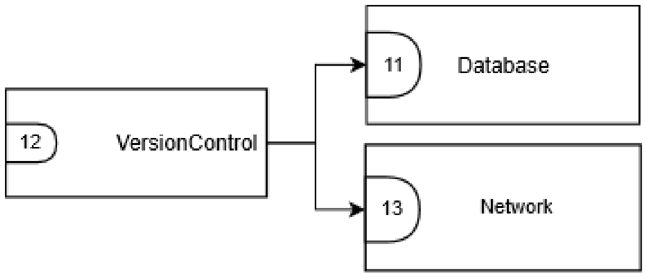
3.2. Version Control Subsystem

The version control subsystem dedicates itself to connecting database and network components to the program. These include storing the necessary data for retrieval and connect the lead with a team of analysts via offline platforms. This subsystem's responsibilities includes the following classes:

1. Version Control

2. Database
3. Network

Diagram #3: UML Diagram for Version Control Subsystem



3.2.1. Class Descriptions for Version Control Subsystem

3.2.1.1. Class Description for Version Control

Class Name: Version Control	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Keep track of table changes • Keep track of database changes	
Contract: 12. Track changes in Database	
Responsibilities	**Collaborations**
1. Knows and tracks changes in the vector of the host Database 2. Knows changes that need approval from lead analyst 3. Knows different versions of the vector 4. Pull changes from database 5. Push changes from database	EventSession

Class Description for Database

Class Name: Database	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Store Events to Database • Stores Vectors to Database • Stores Nodes to Database • Stores Relationships to Database • Serializes/Deserializes data stored in Database • Knows ObjectID's of Event in Database • Knows ObjectID's of Vectors' in Database • Knows ObjectID's of Nodes' in Database • Knows ObjectID's of relationships' in Database	
Contract: 1 Store vectors of Event	
Responsibilities	**Collaborations**
1. Knows vectors of the event	Vector
Contract: 2. Update vectors of Event	
Responsibilities	**Collaborations**
1. Knows vectors of the event	Vector

Contract: 3. Store Nodes of Event	
Responsibilities	**Collaborations**
1. Knows nodes of the event	Nodes
Contract: 4. Update Nodes of Event	
Responsibilities	**Collaborations**
1. Knows nodes of the event	Nodes
Contract: 5. Store relationships of Event	
Responsibilities	**Collaborations**
1. Knows relationships of the Nodes	Relationships
Contract: 6. Update relationships of Event	
Responsibilities	**Collaborations**
1. Knows relationships of the Nodes	Relationships
Contract: 7. Stores Event to Database	
Responsibilities	**Collaborations**
1. Knows Event name 2. Knows nodes of the event 3. Knows vectors of the event 4. Knows relationships of the event	EventConfiguration Vector Nodes Relationships
Contract: 8. Update Event in Database	
Responsibilities	**Collaborations**
1. Knows Event name 2. Knows nodes of the event 3. Knows vectors of the event 4. Knows relationships of the event	EventConfiguration Vector Nodes Relationships

3.2.1.1.1. Contract #11: Establish Connection to Database

1. **Establish connection to Database**
   - **Method name:** connect_analyst_to_lead(self,lead_ip)
   - **Pre-condition:**
     - The IP address must be different from the lead
     - The local database must be an analyst; it cannot be the lead
     -
   - **Post-condition:**
     - The local database is connected to the lead database

3.2.1.1.2. Contract #12: Track Changes in Database

1. **Track changes in Database**
   - **Method name:** check_deltas()
   - **Pre-condition:**
     - N/A
   - **Post-condition:**
     - Vectors in the host database are updated if changes were found; otherwise, vectors remain the same

2. **Pull changes from Database**
   - **Method name:** pull_from_database()
   - **Pre-condition:**
     - N/A
   - **Post-condition:**
     - Changes in host database are pulled from database and updates in local computer.

3. **Push changes to Database**
   - **Method name:** push_to _database()
   - **Pre-condition:**

  - Lead analyst must approve of analyst changes to push
- **Post-condition:**
  - Changes made from lead or analyst updates the changes made to database

3.2.1.1.3. Contract #13: Store Events

1. **Updates Event data in Database**
   - **Method name:** save_event_data_to_database(self, event_config, vector_list)
   - **Pre-condition:**
     - Event must have a vector populated with nodes
   - **Post-condition:**
     - All data from specified Event are updated in MongoDB database

2. **Updates Event data in Database**
   - **Method name:** update_event_config(self,event_config,vector_id_list)
   - **Pre-condition:**
     - Updated data in specified Event must contain deltas from current stored Event
   - **Post-condition:**
     - All data from specified Event are updated in MongoDB database
     -

3.2.1.1.4. Contract #14: Store Vectors

1. **Store Vectors of Event**
   - **Method name:** save_vector_to_database(self,vector)
   - **Pre-condition:**
     - Event must have a vector populated with nodes
   - **Post-condition:**
     - Vector from specified Event is stored in MongoDB database

2. **Update Vectors of Event**
   - **Method name:** update_vector(self,vector)
   - **Pre-condition:**
     - Updated vectors must contain deltas from currently stored vectors
   - **Post-condition:**
     - Vector from specified Event is stored in MongoDB database

3.2.1.1.5. Contract #15: Store Nodes

1. **Store Nodes of Event**
   - **Method name:** save_node_to_database(self,node)
   - **Pre-condition:**
     - Event must have a vector populated with nodes
   - **Post-condition:**
     - Nodes from specified Event is stored in MongoDB database.

2. **Update Nodes of Event**
   - **Method name:** update_node(self,node)
   - **Pre-condition:**
     - Updated nodes must contain deltas from currently stored nodes
   - **Post-condition:**

- Nodes from specified Event is stored in MongoDB database.

3.2.1.1.6. Contract #16: Store Relationships

1. Store relationships of Event

- **Method name:** save_relationships_to_database(self,relationship)
- **Pre-condition:**
  - Event must have a vector populated with nodes
  - Event must have a vector populated with nodes containing relationships
- **Post-condition:**
  - Relationships from specified Event are stored in MongoDB database

1. Store relationships of Event

- **Method name:** update_relationship(self,relationship)
- **Pre-condition:**
  - Updated relationships must contain deltas from currently stored relationships
- **Post-condition:**
  - Relationships from specified Event are updated in MongoDB database

3.2.1.2. Class Description for Network

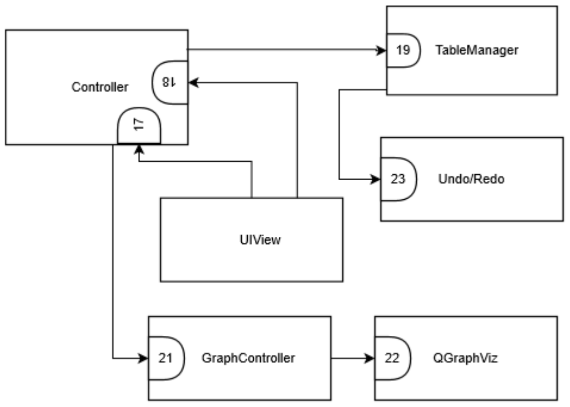
Class Name: Network	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • N/A	
Contract: 11. Establish connection to Database	
Responsibilities	**Collaborations**
1. Connect analyst and host Database 2. Send analyst commits to lead 3. Send updated vectors to analyst	EventConfiguration

3.3. User Interaction Subsystem

This subsystem is the most fluent when it comes to sustaining the system’s functionality. Included are the classes Controller, MainUI, TableManager, UndoRedoManager, GraphController and QGraphViz. In this case, the primary focus is the Controller.

- Contract 17: Update View
- Contract 18: Update Models Given User Input
- Contract 19: Update UI Tables
- Contract 20: Send Table Changes to Data Models
- Contract 21: Display Graph
- Contract 22: Create Graph
- Contract 23: Undo/Redo

Diagram 4: UML Table User Interface Sub-System Component Diagram



3.3.1. Class Descriptions for User Interaction Subsystem

3.3.1.1. Controller

Class Name: Controller	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • N/A	
Contract: 17. Update view	
Responsibilities	**Collaborations**
1. Gather data updates from the models. 2. Update the view.	TableManager EventSession GraphController
Contract: 18. Update models given user input	
Responsibilities	**Collaborations**
1. Gather user input. 2. Update the models. 3. Direct models to take some action based on the user input.	IngestionController SplunkFacade Database EventSession GraphController TableManager

3.3.1.1.1. Contract 17: Update View

Contract Description: Updates the view to represent the current data of the models.
Protocols:

1. Update Node Table
   a. **Signature:** update_node_table()
   b. **Description:** Forwards the nodes to the table manager to update the node table.
   c. **Pre-Condition(s):**
      i. The nodes list has been initialized (there shouldn’t necessarily be nodes in it)
   d. **Post-Condition(s):**
      i. The widget pertaining to the node table is populated with the relevant information of the nodes.

2. Update Relationship Table
   a. **Signature:** update_relationship_table()
   b. **Description:** Forwards the relationship objects to the table manager to update the relationship table.

c. **Pre-Condition(s):**
   i. The relationships list has been initialized (there shouldn't necessarily be relationships in it)

d. **Post-Condition(s):**
   i. The widget pertaining to the relationship table is populated with the relevant information of the relationships.

3. Update Log Entry Table
   a. **Signature:** update_log_entry_table()
   b. **Description:** Forwards the log entries to the table manager to update the log entry table.
   c. **Pre-Condition(s):**
      i. The log entries list has been initialized (there shouldn't necessarily be log entries in it)
   d. **Post-Condition(s):**
      i. The widget pertaining to the log entry table is populated with the relevant information of the log entries.

4. Update Vector Tables
   a. **Signature:** update_vector_tables()
   b. **Description:** Forwards the vector objects to the table manager to update the vector configuration and the add-to-vector tables.
   c. **Pre-Condition(s):**
      i. The vector list has been initialized (there shouldn't necessarily be vectors in it)
   d. **Post-Condition(s):**
      i. The widgets pertaining to the vector configuration and add-to-vector tables are populated with the relevant information of the vectors.

5. Update Vector Drop-Down
   a. **Signature:** update_vector_dropdown()
   b. **Description:** Forwards the vector objects to the table manager to update vector drop-down.
   c. **Pre-Condition(s):**
      i. The vector list has been initialized (there shouldn't necessarily be vectors in it)
   d. **Post-Condition(s):**
      i. The widget pertaining to the vector drop-down is populated with the relevant information of the vectors.

6. Update Node Drop-Down
   a. **Signature:** update_node_dropdown(combo_box)
   b. **Description:** Forwards the vector objects to the table manager to update given node drop-down (combo-box).
   c. **Pre-Condition(s):**
      i. The nodes list of the selected vector has been initialized.
      ii. At least two nodes shall be in the list.
   d. **Post-Condition(s):**
      i. The combo-box widget from the parameter is populated with the relevant information of the vectors.

3.3.1.1.2. Contract 18: Update Models

Contract Description: Updates the back-end given some user input or instruction.
Protocols:

7. Update Event Data
   a. **Signature:** update_event_data()
   b. **Description:** Updates the event data from the database according to the object id from the event.
   c. **Pre-Condition(s):**

        i. MongoDB is running
        ii. An event with the corresponding object id exists in the database.
    d. **Post-Condition(s):**
        i. The event data is updated from the version stored in the database.

8. Update Event DB
    a. **Signature:** update_event_db ()
    b. **Description:** Updates the data in the database corresponding to the event with the current information of the event.
    c. **Pre-Condition(s):**
        i. MongoDB is running
        ii. The event has been initialized.
        iii. An event with the corresponding object id of the current event exists in the database.
    Post-Condition(s):
        i. The event data stored in the database is replaced with the event data from the current session of the user.

9. Update Splunk Filter
    a. **Signature:** update_splunk_filter (keyword, start_time, end_time)
    b. **Description:** Updates the arguments used to filter in Splunk and calls the filtering to start.
    c. **Pre-Condition(s):**
        i. The Splunk service is running.
        ii. The event has been initialized (therefore, the index corresponding to it has been initialized too).
        iii. The user has been connected to Splunk.
    Post-Condition(s):
        i. The filtering parameters for filtering through the Splunk index have been set and a request to filter has been initiated.

10. Add Vector
    a. **Signature:** add_vector()
    b. **Description:** Adds a vector to the list of the event session.
    c. **Pre-Condition(s):**
        i. The vector list has been initialized
    Post-Condition(s):
        i. A new vector is appended into the vector list of the event session.

11. Delete Vector
    a. **Signature:** delete_vector()
    b. **Description:** Deletes the vector(s) marked from the vector configuration table from the vector list of the event session.
    c. **Pre-Condition(s):**
        i. The vector list has been initialized.
        ii. The selected vector(s) exist in the list.
    Post-Condition(s):
        i. The selected vectors are removed from the list.
12. Create Node
    a. **Signature:** create_vector()
    b. **Description:** Adds a node into the selected vector.
    c. **Pre-Condition(s):**
        i. The node list of the selected vector has been initialized.
    Post-Condition(s):
        i. A blank node has been appended into the node list of the selected vector.

13. Create Relationship
    a. **Signature:** create_relationship(parent_id, child_id, name)

   b. **Description:** Adds a relationship linked to the parent and child nodes into the selected vector.
   c. **Pre-Condition(s):**
      i. The relationship list of the selected vector has been initialized.
      ii. The node list of the selected vector has been initialized.
      iii. A node corresponding to the parent id exists in the node list.
      iv. A node corresponding to the child id exists in the node list.

   Post-Condition(s):
      i. A relationship with the corresponding name and linked to the parent and child nodes specified in the parameters is created in the selected vector.

14. Create Event
   a. **Signature:** create_event(name, description, start_time, end_time)
   b. **Description:** Setup the event data corresponding to a new event and creates an index in Splunk.
   c. **Pre-Condition(s):**
      i. Splunk is initialized.
      ii. The user has been connected to the Splunk service.
      iii. The start time is earlier than the end time.
      iv. The isn't an index with the same name in the Splunk db.

   Post-Condition(s):
      i. An index with the name of the event is created.
      ii. An event config with the corresponding information is created and set as the one to be used during the event session.
15. Connect to Splunk
   a. **Signature:** connect_to_splunk(username, password)
   b. **Description:** Connects the user given their credentials to the Splunk service.
   c. **Pre-Condition(s):**
      i. Splunk is initialized.

   Post-Condition(s):
      i. If the username and password combination is valid the user is connected and returns a True value.
      ii. If the username and password combination in invalid the user is not connected to Splunk and returns a False value.

16. Add Log Entries to Vector
   a. **Signature:** add_log_entry_to_vector(selected_log_entries, selected_vectors)
   b. **Description:** Adds the selected log entries as nodes inside the selected vectors.
   c. **Pre-Condition(s):**
      i. The node list of the selected vector(s) has been initialized.
      ii. The log entry list of the event session has been initialized.

   Post-Condition(s):
      i. New nodes corresponding to the selected log entries are added to the nodes list of the selected vectors.

17. Validate Anyway
   a. **Signature:** validate_anyway()
   b. **Description:** Marks the selected log files as validated and forwards them to be ingested.
   c. **Pre-Condition(s):**
      i. The selected log files have been marked as invalid.

   Post-Condition(s):
      i. The selected log files are marked as valid.
      ii. The selected log files are forwarded to be ingested.

18. Validate Anyway
   a. **Signature:** validate_anyway()
   b. **Description:** Marks the selected log files as validated and forwards them to be ingested.

c. **Pre-Condition(s):**
    i. The selected log files have been marked as invalid.

   Post-Condition(s):
    i. The selected log files are marked as valid.
    ii. The selected log files are forwarded to be ingested.

19. Export Log Entry Table
    a. **Signature:** export_log_entry_table(filename)
    b. **Description:** Exports a CSV of the items presented in the log entry table with the given filename.
    c. **Pre-Condition(s):**
        i. The log entry list is initialized.

       Post-Condition(s):
        i. A CSV with the corresponding filename is created containing the information of the log entry table.

20. Export Node Table
    a. **Signature:** export_vector_table(filename)
    b. **Description:** Exports a CSV of the items presented in the node table with the given filename.
    c. **Pre-Condition(s):**
        i. The node list of the selected vector has been initialized.

       Post-Condition(s):
        i. A CSV with the corresponding filename is created containing the information of the node table.

21. Export Vector Configuration Table
    a. **Signature:** export_vector_configuration_table(filename)
    b. **Description:** Exports a CSV of the items presented in the vector configuration table with the given filename.
    c. **Pre-Condition(s):**
        i. The vector list of the event session has been initialized.

       Post-Condition(s):
        i. A CSV with the corresponding filename is created containing the information of the vector configuration table.

3.3.1.2. TableManager

Class Name: TableManager	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Knows the structure of the Vector class. • Knows the structure of the Node class. • Knows the structure of the LogEntry class. • Knows the structure of the LogFile class. • Knows the structure of the Relationship class. • Knows the structure of the QTableWidgets.	
Contract: 19. Update UI Tables	
Responsibilities	**Collaborations**
1. Update the QTableWidgetItems inside the QTableWidgets to match the data.	
Contract: 20. Send Table Changes to Data Models	
Responsibilities	**Collaborations**

1. Updates the relevant data models given the changes in the data.	EventSession

3.3.1.2.1. Contract 19: Update UI Tables

Contract Description: Updates the tables from the UI given the relevant data.

Protocols:

1. Populate Log Entry Table
    a. **Signature:** populate_log_entry_table(log_entries)
    b. **Description:** Populates the log entry table with the data of the log entries.
    c. **Pre-Condition(s):**
        i. The log entry table has been defined.
        ii. The log entries follow the format specified of the LogEntry class (May 9th, 2020).
    d. **Post-Condition(s):**
        i. The log entry table is populated with the data of the log entries.

2. Populate Node Table
    a. **Signature:** populate_node_table(nodes)
    b. **Description:** Populates the node table with the data of the nodes.
    c. **Pre-Condition(s):**
        i. The node table has been defined.
        ii. The nodes follow the format specified of the Node class (May 9th, 2020).
    d. **Post-Condition(s):**
        i. The node table is populated with the data of the nodes.

3. Populate Relationship Table
    a. **Signature:** populate_relationship_table(relationships)
    b. **Description:** Populates the relationship table with the data of the relationship.
    c. **Pre-Condition(s):**
        i. The relationship table has been defined.
        ii. The relationships follow the format specified of the Relationship class (May 9th, 2020).
    d. **Post-Condition(s):**
        i. The relationship table is populated with the data of the relationships.

4. Populate Vector Configuration Table
    a. **Signature:** populate_vector_configuration_table(vectors)
    b. **Description:** Populates the vector configuration table with the data of the nodes.
    c. **Pre-Condition(s):**
        i. The vector configuration table has been defined.
        ii. The vectors follow the format specified of the Vector class (May 9th, 2020).
    d. **Post-Condition(s):**
        i. The vector configuration table is populated with the data of the vectors.

5. Populate Log File Table
    a. **Signature:** populate_log_file_table(log_files)
    b. **Description:** Populates the log file table with the data of the log file objects.
    c. **Pre-Condition(s):**
        i. The log file table has been defined.
        ii. The log file objects follow the format specified of the LogFile class (May 9th, 2020).
    d. **Post-Condition(s):**
        i. The log file table is populated with the data of the log file objects.

6. Populate Enforcement Action Report Table
    a. **Signature:** populate_log_file_table(log_file)

   b. **Description:** Populates the enforcement action table with the data of the error list in the log file object.
   c. **Pre-Condition(s):**
      i. The log file table has been defined.
      ii. The log file object follows the format specified of the LogFile class (May 9th, 2020).
   d. **Post-Condition(s):**
      i. The enforcement action report table is populated with the data of the error list in the log file object.

7. Populate Vector Drop-Down
   a. **Signature:** populate_vector_dropdown(vectors)
   b. **Description:** Populates the vector drop-down with name of the vectors.
   c. **Pre-Condition(s):**
      i. The vector drop-down combo-box has been defined.
      ii. The vectors follow the format specified of the Vector class (May 9th, 2020).
   d. **Post-Condition(s):**
      i. The vector drop-down is populated with the name of the vectors.

8. Populate Node Drop-Down
   a. **Signature:** populate_node_dropdown(nodes, combo-box)
   b. **Description:** Populates the drop-down combo-box with the name of the nodes.
   c. **Pre-Condition(s):**
      i. The nodes follow the format specified of the Node class (May 9th, 2020).
   d. **Post-Condition(s):**
      i. The drop-down is populated with the name of the nodes.

3.3.1.2.2. Contract 20: Send Table Changes to Data Models

Contract Description: Updates data models.
Protocols:

1. Edit Node Table
   a. **Signature:** edit_node_table(row, column, value, nodes)
   b. **Description:** Edits the node displayed in the selected row of the and edit the value according to the column position.
   c. **Pre-Condition(s):**
      i. The node table has been defined.
      ii. The nodes follow the format specified of the Vector class (May 9th, 2020).
   d. **Post-Condition(s):**
      i. The pertaining field of the node has been updated.
      ii. If the updating of the node field triggers an extra change, that change is reflected in the table.

2. Edit Vector Configuration Table
   a. **Signature:** edit_vector_table(row, column, value, vector)
   b. **Description:** Edits the vector displayed in the selected row of the and edit the value according to the column position.
   c. **Pre-Condition(s):**
      i. The vector configuration table has been defined.
      ii. The vectors follow the format specified of the Vector class (May 9th, 2020).
   d. **Post-Condition(s):**
      i. The pertaining field of the vector has been updated.

3.3.1.3. GraphController

Class Name: GraphController

Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Knows how to interact with QGraphViz.	
Contract: 21. Display Graph	
Responsibilities	**Collaborations**
1. Uses QGraphViz to display the graphical representation of the vector. 2. Keep track of the node positions.	QGraphViz
Contract: 24. Provide Event Data	
Responsibilities	**Collaborations**
1. Gets all data necessary from event session	Event Session

3.3.1.3.1. Contract 21: Display Graph

Contract Description: Displays the graphical representation of a vector.
Protocols:

1. Read Vector Table
    a. **Signature:** read_vector_table(vector)
    b. **Description:** Displays a new graph corresponding to the vector information.
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
    d. **Post-Condition(s):**
        i. If the vector is empty then a graph with an empty prompt is displayed, otherwise it displays the graphical representation of the vector.

2. Update Graph
    a. **Signature:** update_graph(vector)
    b. **Description:** Refreshes the existent graph with the information of a vector.
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
        ii. The graph has been initialized.
    d. **Post-Condition(s):**
        i. The widget displays the graphical representation of the vector.

3. Save Node Positions
    a. **Signature:** save_node_positions(vector)
    b. **Description:** Stores the coordinates of the represented nodes.
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
        ii. The graph has been initialized.
    d. **Post-Condition(s):**
        i. The x and y parameters of the nodes inside the vector are updated to reflect their current position in the graph.

4. Export Graph
    a. **Signature:** export_graph(filename)
    b. **Description:** Exports a PNG file of the displayed graph.
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
        ii. The graph has been initialized.
    d. **Post-Condition(s):**
        i. A PNG file with the given filename is exported containing a representation of the graph.

3.3.1.4. QGraphViz

Class Name: GraphController	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Knows how to setup widgets to create a graphical representation.	
Contract: 22. Create Graph	
Responsibilities	**Collaborations**
1. Add nodes 2. Add edgees	N/A

3.3.1.4.1. Contract 22: Create Graph

Contract Description: Creates a graphical representation given some nodes and edges.
Protocols:

1. Build
    a. **Signature:** build()
    b. **Description:** Builds the graph.
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
    d. **Post-Condition(s):**
        i. The graph is built inside the defined widget.

2. Add Node
    a. **Signature:** add_node()
    b. **Description:** Adds a node to the graph
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
    d. **Post-Condition(s):**
        i. A new node is added into the graph structure.

3. Remove Node
    a. **Signature:** remove_node()
    b. **Description:** Removes a node from set in the graph
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
    d. **Post-Condition(s):**
        i. The node is removed from the graph structure.

4. Add Edge
    a. **Signature:** add_edge()
    b. **Description:** Adds an edge to the graph.
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
        ii. Two nodes are selected in the graph.
    d. **Post-Condition(s):**
        i. A new edge is added between the selected nodes into the graph structure.

5. Remove Edge
    a. **Signature:** remove_edge()
    b. **Description:** Removes the edge from the selected nodes from the graph.
    c. **Pre-Condition(s):**
        i. The graph widget has been defined.
        ii. Two nodes are selected in the graph.
    d. **Post-Condition(s):**

i. The edge between the selected nodes is removed.

3.3.1.5. UndoRedoManager

Class Name: UndoRedoManager	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Keep track of the user actions in the node table. • Knows the allowed actions in the table.	
Contract: 23. Undo/Redo	
Responsibilities	**Collaborations**
1. Undo the latest action of the user. 2. Redo the lates undo of the user.	TableManager EventSession

3.3.1.5.1. Contract 23: Undo/Redo

Contract Description: Undo or redo the actions of the user.
Protocols:

1. Undo
    a. **Signature:** undo()
    b. **Description:** Reverts to the last action of the user recorded in the undo stack.
    c. **Pre-Condition(s):**
        i. The command switcher has been initialized.
        ii. The event session has a selected vector.
        iii. The node list of the selected vector has been initialized.
    d. **Post-Condition(s):**
        i. The latest action of the user in the node table has been undone (i.e. if the user renamed a node from “a” to “b” then the node is renamed back to “a”).
        ii. The corresponding change is also reflected in the graph.
        iii. The undo action is tracked in the redo stack.

2. Redo
    a. **Signature:** redo()
    b. **Description:** Reverts to the last undo action of the user recorded in the redo stack.
    c. **Pre-Condition(s):**
        i. The command switcher has been initialized.
        ii. The event session has a selected vector.
        iii. The node list of the selected vector has been initialized.
    d. **Post-Condition(s):**
        i. The latest undo action of the user in the node table has been redone.
        ii. The corresponding change is also reflected in the graph.
        iii. The redo action is tracked in the redo stack.

3. Add Command
    a. **Signature:** add_command(command_key, args_list, target)
    b. **Description:** Adds a user action to be tracked into the target stack.
    c. **Pre-Condition(s):**
        i. The command switcher has been initialized.
    d. **Post-Condition(s):**
        i. The action is pushed into the target stack

3.3.1.6. MainUI

Class Name: MainUI

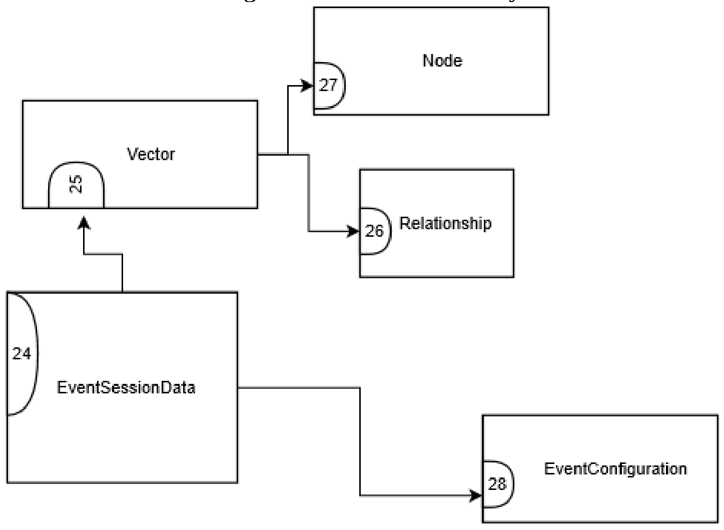
Superclass: UI_PICK
Subclasses: N/A
Private Responsibilities • Display the view to the user • Receive input from the user

3.4. User Data

This subsystem encloses all the actions taken by the PICK Tool in order to store and query.The following classes are:

1. EventSession
2. Node
3. Vector
4. Relationship
5. EventConfiguration

UML Diagram for User Data Subsystem



3.4.1. Class Description for User Data Subsystem

3.4.1.1. Class Description EventSession

Class Name: EventConfiguration
Superclass: N/A
Subclasses: N/A
Private Responsibilities • Knows event name • Knows event description • Knows event start date • Knows event end date • Knows event vectors

3.4.1.1.1. Contract #1: Provide path Directories

Contract Description: Gathers files from the directories and instantiates an object of LogFile for each file.

1. **Get Files from Root Directory**
   - **Method name:** get_root_path()
   - **Pre-condition:**An event has been created.
   - **Post-condition:**The Ingestion Manager has the directory to the root folder.

2. **Get Files from Red Team Directory**
   - **Method name:** get_red_team_path()
   - **Pre-condition(s):**An event has been created.
   - **Post-condition(s):**The Ingestion Manager has the directory to the red_team folder.

3. **Get Files from Blue Team Directory**
   - **Method name: Python Syntax**: get_blue_team_path()
   - **Pre-condition:**
     - An event has been created.
   - **Post-condition:**
     - The Ingestion Manager has the directory to the blue_team folder.

4. **Get Files from White Team Directory**
   - **Method name:** get_white_team_path()
   - **Pre-condition:**
     - An event has been created.
   - **Post-condition:**
     - The Ingestion Manager has the directory to the white_team folder.

3.4.1.1.2. Contract #24: Provide Event Data

Contract Description: Gathers files from the directories and instantiates an object of LogFile for each file.

1. **Get Files from Root Directory**
   - **Method name:** provide_event_data()
   - **Pre-condition:** An event has been created.
   - **Post-condition:** Event data is provided.

3.4.1.2. Class Description Node

Class Name: EventConfiguration
Superclass: N/A
Subclasses: N/A
Private Responsibilities • Knows node id • Knows node name • Knows node timestamp • Knows node description • Knows node log entry reference • Knows node log creator • Knows node event type • Knows node source • Knows node icon type • Knows node visibility • Knows node icon

• Knows node x position • Knows node y position • Knows own object id

3.4.1.2.1. Contract #26: Node

1. Create Node from Log Entry

Method name: node_from_log_entry(log_entry)

- **Pre-condition:**
  - An event has been created.
- **Post-condition:**
  - A node is created from a log entry.

3.4.1.3. Class Description Vector

Class Name: Vector	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Knows vector name • Knows vector description • Knows vector nodes • Knows vector relationships • Knows own object id	
Contract: 27. Add Nodes	
Responsibilities	**Collaborations**
Create a vector.	Node
Contract: 26. Add relationship	
Responsibilities	**Collaborations**
Add relationship between two nodes.	Relationship, Node

3.4.1.3.1. Contract #25: Provide Vectors of Event

1. Create Vector

Method name: create vector(name, description)

- **Pre-condition:**
  - An event has been created.
- **Post-condition:**
  - A vector is created

3.4.1.4. Class Description Relationship

Class Name: EventConfiguration
Superclass: N/A
Subclasses: N/A
Private Responsibilities • Knows event name • Knows event description • Knows event start date • Knows event end date • Knows event vectors

3.4.1.4.1. Contract #26: Add relationship

1. Add relationship

Method name: add_relationship(node, node)

- **Pre-condition:**
    - A vector is selected
- **Post-condition:**
    - A relationship is created between two nodes

3.4.1.5. Class Description EventConfiguration

Class Name: EventConfiguration	
Superclass: N/A	
Subclasses: N/A	
Private Responsibilities • Knows event name • Knows event description • Knows event start date • Knows event end date • Knows event vectors	
Contract: 25. Provide Vector	
Responsibilities	**Collaborations**
Create a vector.	Vector
Contract: 28. Provide Event Configuration Data	
Responsibilities	**Collaborations**
1. Provide Evet Configuration Data	

3.4.1.5.1. Contract #28: Provide Event Configuration Data

1. Provide Event Configuration Data

Method name: provide_data()

- **Pre-condition:**
    - An event has been created.
- **Post-condition:**
    - N/A

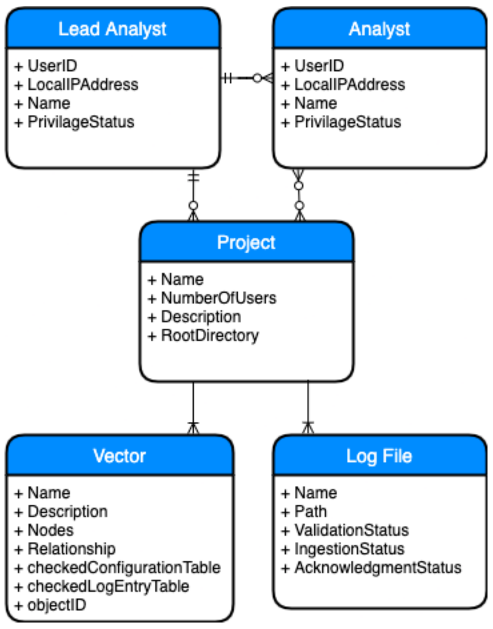
4. Database

This section describes the database aspect of the system used to associate with input and output data. Classes from each of the entities are carefully evaluated to determine whether itis requires database access to manipulate data. Included are the Database Schema and the determinable Entity-Relational Model.

4.1.Database Schema

The Database Schema presents the blueprint as to how data is to be collected and shared among classes in the database. This organization allows for the applications of tables and views to be applied in each of the classes.

Diagram 3: Database Schema



*