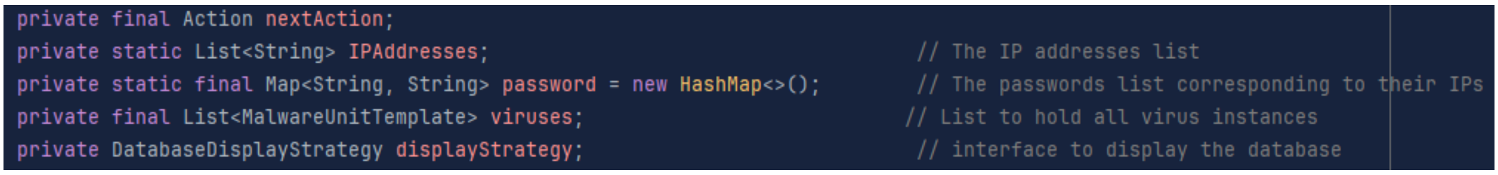
Virus Simulation

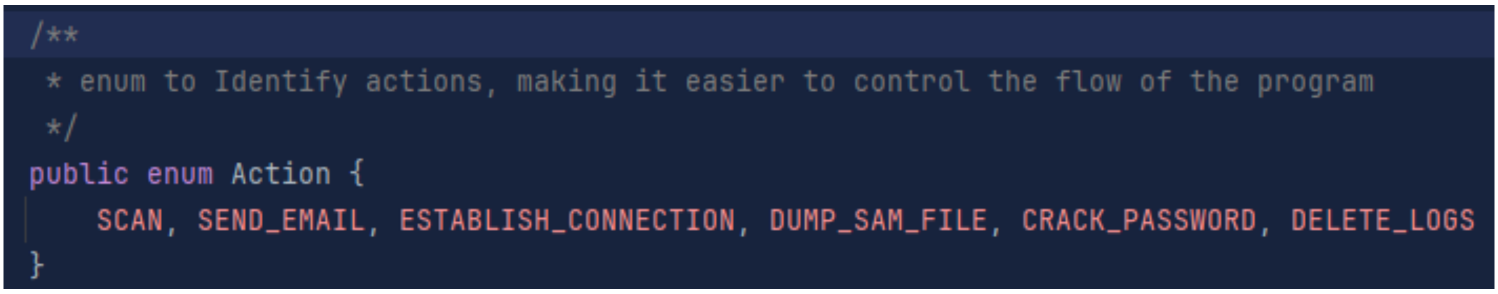
The Malware Simulation Program is designed to emulate various types of malware actions for educational purposes. It uses a structured approach to simulate actions such as scanning for hosts, sending phishing emails, establishing connections, dumping system files, cracking passwords, and deleting logs. The program is implemented in Java and extends a base class "Unit" to represent each type of malware as an object.

The program initilize the following:



```java
private final Action nextAction;
private static List<String> IPAddresses;                          // The IP addresses list
private static final Map<String, String> password = new HashMap<>();     // The passwords list corresponding to their IPs
private final List<MalwareUnitTemplate> viruses;                 // List to hold all virus instances
private DatabaseDisplayStrategy displayStrategy;                  // interface to display the database
```

The MalwareUnit has a method that helps enumerate through the actions and like scannig, sending emails, etc. Each MalwareUnit created has one role to do:



```java
/**
 * enum to Identify actions, making it easier to control the flow of the program
 */
public enum Action {
    SCAN, SEND_EMAIL, ESTABLISH_CONNECTION, DUMP_SAM_FILE, CRACK_PASSWORD, DELETE_LOGS
}
```

MalwareUnit class methods:

1. Constructor:
• **Purpose**: Initializes a new instance of with specific parameters.
• **Usage**: Sets up the malware unit with a name, simulation input parameters, the next action it should perform, and the strategy for displaying database information. It also initializes IP addresses and associated passwords if they are not provided in the input.

2. Method:
◇ **Purpose**: Ensures that IP addresses are shared and initialized correctly across multiple instances or parts of the simulation.
◇ **Usage**: This static method checks if the IP addresses are already initialized and if not, generates them using the . It ensures that all instances of use the same IP addresses, supporting consistency across the simulation environment.

3. Method:
◇ **Purpose**: Executes the action determined by the property.
◇ **Usage**: This method controls the workflow of the malware activities based on the specified action. It calls different methods associated with each action type (like scanning, sending emails, etc.), facilitating the simulation of various malware operations.

4. Method:

◇ **Purpose**: Submits and logs statistics related to malware actions.

◇ **Usage**: This method is crucial for tracking the number of actions performed by the malware unit, updating the statistical data which can be used for analysis and monitoring the effectiveness or intensity of the malware simulation.

5. Method:

◇ **Purpose**: Provides a mechanism to set or update the IP addresses being used in the simulation.

◇ **Usage**: This static method can be called to explicitly set the list of IP addresses if there is a need to refresh or change the IPs during the simulation.

6. Method:

◇ **Purpose**: Displays or logs the IP addresses and their associated passwords.

◇ **Usage**: This static method is used to print out a list of all IP addresses and their corresponding passwords, providing a clear view of the data being managed by the malware unit. This is useful for debugging and ensuring that IP address assignments are as expected.

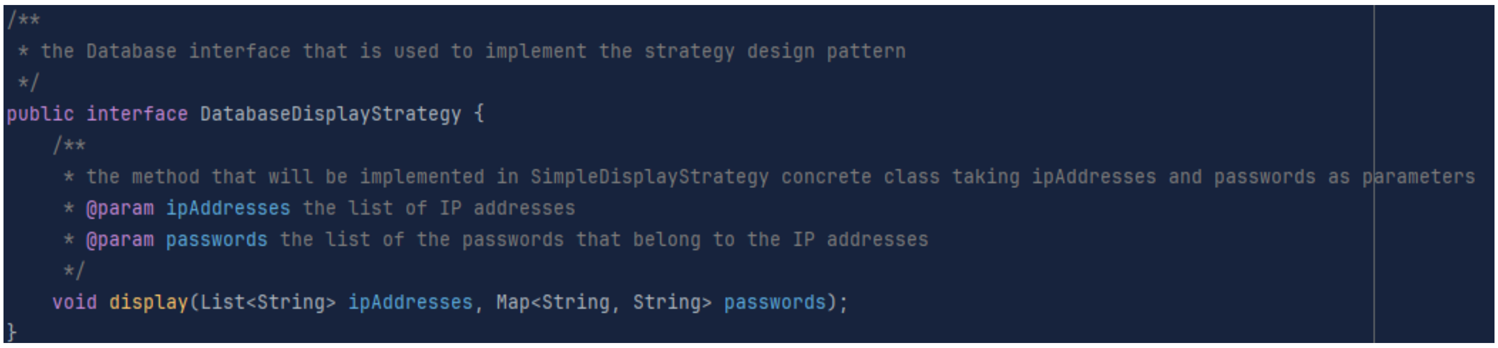
abstract class and interfaces

DatabaseDisplayStrategy (Interface)

• **Purpose**: This interface is designed to define a strategy for displaying data, specifically allowing for different ways to present malware-related data. It adheres to the Strategy Design Pattern, enabling the dynamic swapping of the database display algorithm at runtime depending on the context or specific requirements.

• **Methods**: ◇ : Abstract method intended to be implemented by classes that specify how the database should be displayed. Each implementation can display the database in a different format or style, such as verbose or minimalistic.

• **Classes that implemented this interface:** SimpleDisplayStrategy



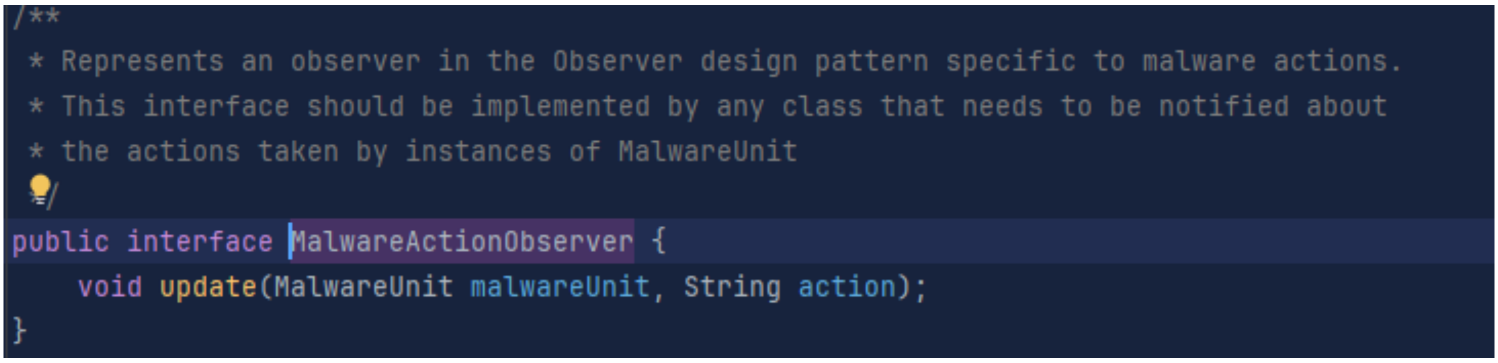
```java
/**
 * the Database interface that is used to implement the strategy design pattern
 */
public interface DatabaseDisplayStrategy {
    /**
     * the method that will be implemented in SimpleDisplayStrategy concrete class taking ipAddresses and passwords as parameters
     * @param ipAddresses the list of IP addresses
     * @param passwords the list of the passwords that belong to the IP addresses
     */
    void display(List<String> ipAddresses, Map<String, String> passwords);
}
```

MalwareActionObserver (Interface)

◇ **Purpose**: This interface is part of the Observer Design Pattern. It is used to define a standard for observers that need to react to changes or actions in a subject, in this case, malware actions.

◇ **Methods**:▪ `update(MalwareUnit malwareUnit, String action)`: Method that gets called to notify the observer about an action performed by a `MalwareUnit`. This allows the observer to perform specific reactions to different types of actions (like logging, altering configurations, etc.).

◇ **Classes that implemented this interface:** LoggingObserver, MonitoringObserver

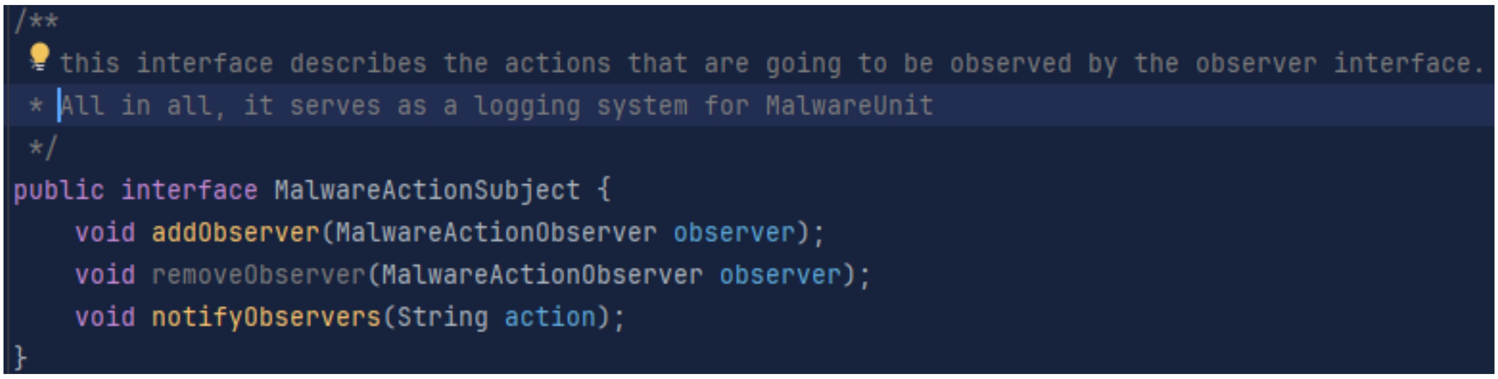

```java
/**
 * Represents an observer in the Observer design pattern specific to malware actions.
 * This interface should be implemented by any class that needs to be notified about
 * the actions taken by instances of MalwareUnit
 */
public interface MalwareActionObserver {
    void update(MalwareUnit malwareUnit, String action);
}
```

MalwareActionSubject (Interface)

◇ **Purpose**: Complements the `MalwareActionObserver` by providing an interface for subjects. It allows subjects to manage (add, remove) observers and notify them of changes or actions.

◇ **Methods**:▪ `addObserver(MalwareActionObserver observer)`: Adds an observer to the subject's list of observers.
▪ `removeObserver(MalwareActionObserver observer)`: Removes an observer from the subject's list of observers.
▪ `notifyObservers(String action)`: Notifies all registered observers of an action, triggering their update method.

◇ **Classes that implemented this interface:** MalwareUnit

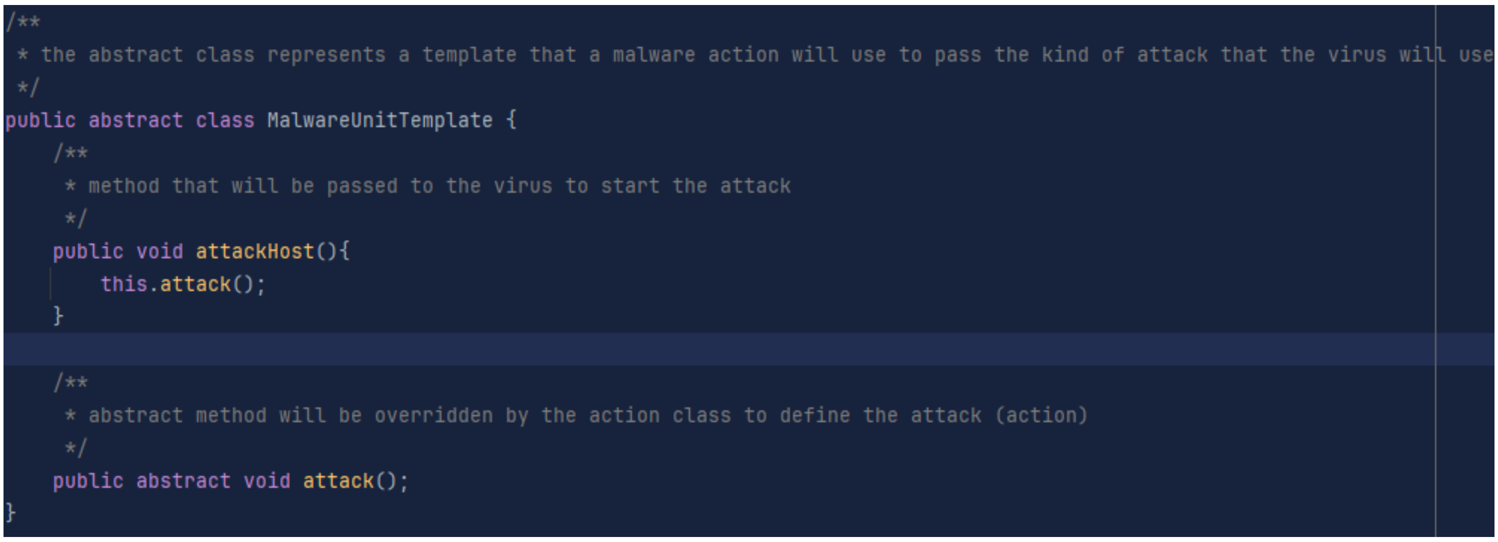

```java
/**
 this interface describes the actions that are going to be observed by the observer interface.
 * All in all, it serves as a logging system for MalwareUnit
 */
public interface MalwareActionSubject {
    void addObserver(MalwareActionObserver observer);
    void removeObserver(MalwareActionObserver observer);
    void notifyObservers(String action);
}
```

MalwareUnitTemplate (Abstract Class)

◇ **Purpose**: This abstract class serves as a template for creating different types of malware units. It utilizes the Template Method Design Pattern to define the skeleton of an algorithm, in this case, the steps involved in malware operations, while allowing its subclasses to redefine certain steps without changing the algorithm's structure.

◇ **Methods**:▪ `attack()`: An abstract method that must be implemented by subclasses to define specific attack behaviors of different types of malware.
▪ `attackHost()`: A concrete method that calls the `attack()` method. This method provides a fixed algorithm structure for executing an attack while the specific details of the attack are defined in `attack()` by subclasses.

◇ **Classes that extended this class:** Scan, SendingEmail, EstablishConnection, DumpSAMFile, CrackPassword, and DeleteLogs



```java
/**
 * the abstract class represents a template that a malware action will use to pass the kind of attack that the virus will use
 */
public abstract class MalwareUnitTemplate {
    /**
     * method that will be passed to the virus to start the attack
     */
    public void attackHost(){
        this.attack();
    }

    /**
     * abstract method will be overridden by the action class to define the attack (action)
     */
    public abstract void attack();
}
```

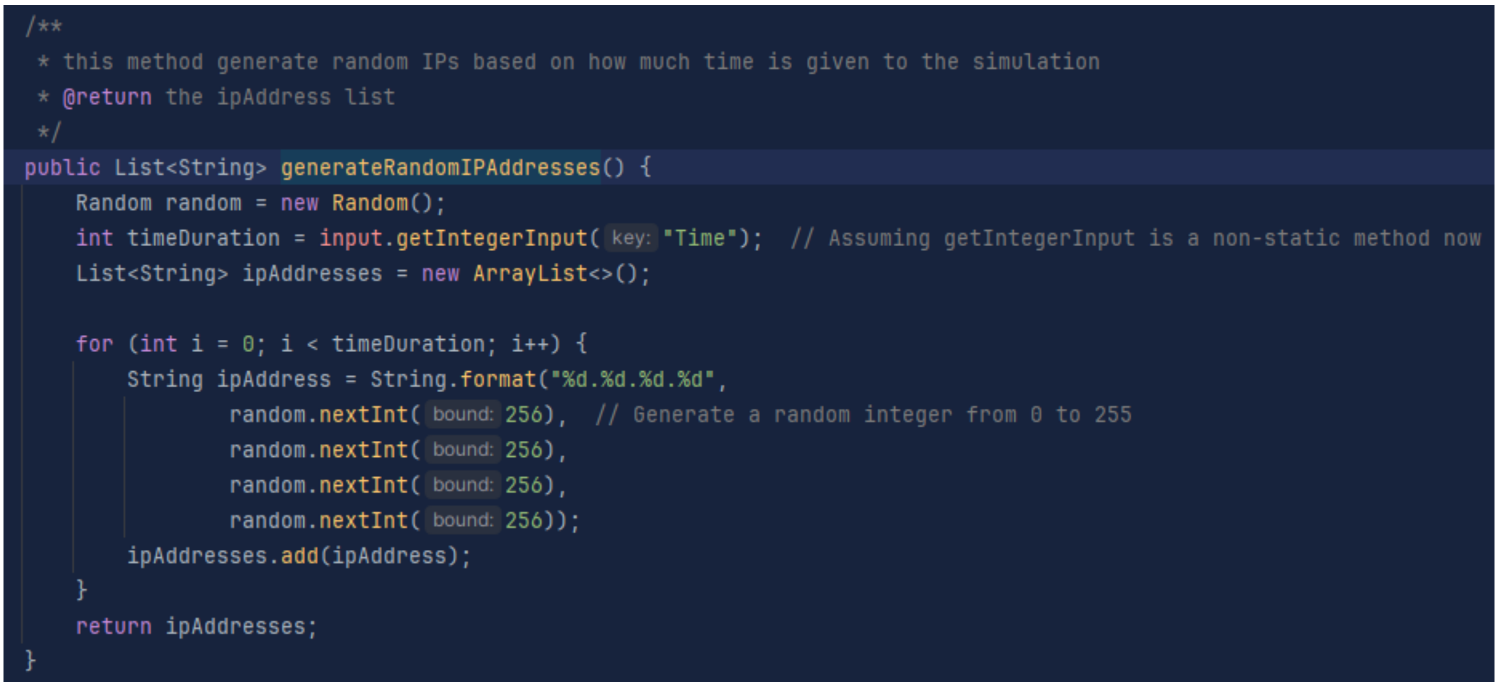
Other Classes

IPAddressGenerator Class

The class is responsible for dynamically generating IP addresses when no specific IPs are provided to the system. This functionality is critical in simulations where the user does not specify target IP addresses, allowing the system to create a realistic network environment by simulating interactions with various generated IPs.

Methods:

generateRandomIPAddresses(): This is the primary method of the class. It generates a list of IP addresses based on the parameters defined within the method or passed through the constructor. The method typically involves random generation techniques to ensure a diverse set of IPs, mimicking real-world scenarios where a malware might interact with different systems across the internet.



```java
/**
 * this method generate random IPs based on how much time is given to the simulation
 * @return the ipAddress list
 */
public List<String> generateRandomIPAddresses() {
    Random random = new Random();
    int timeDuration = input.getIntegerInput( key: "Time");  // Assuming getIntegerInput is a non-static method now
    List<String> ipAddresses = new ArrayList<>();

    for (int i = 0; i < timeDuration; i++) {
        String ipAddress = String.format("%d.%d.%d.%d",
                random.nextInt( bound: 256),  // Generate a random integer from 0 to 255
                random.nextInt( bound: 256),
                random.nextInt( bound: 256),
                random.nextInt( bound: 256));
        ipAddresses.add(ipAddress);
    }
    return ipAddresses;
}
```

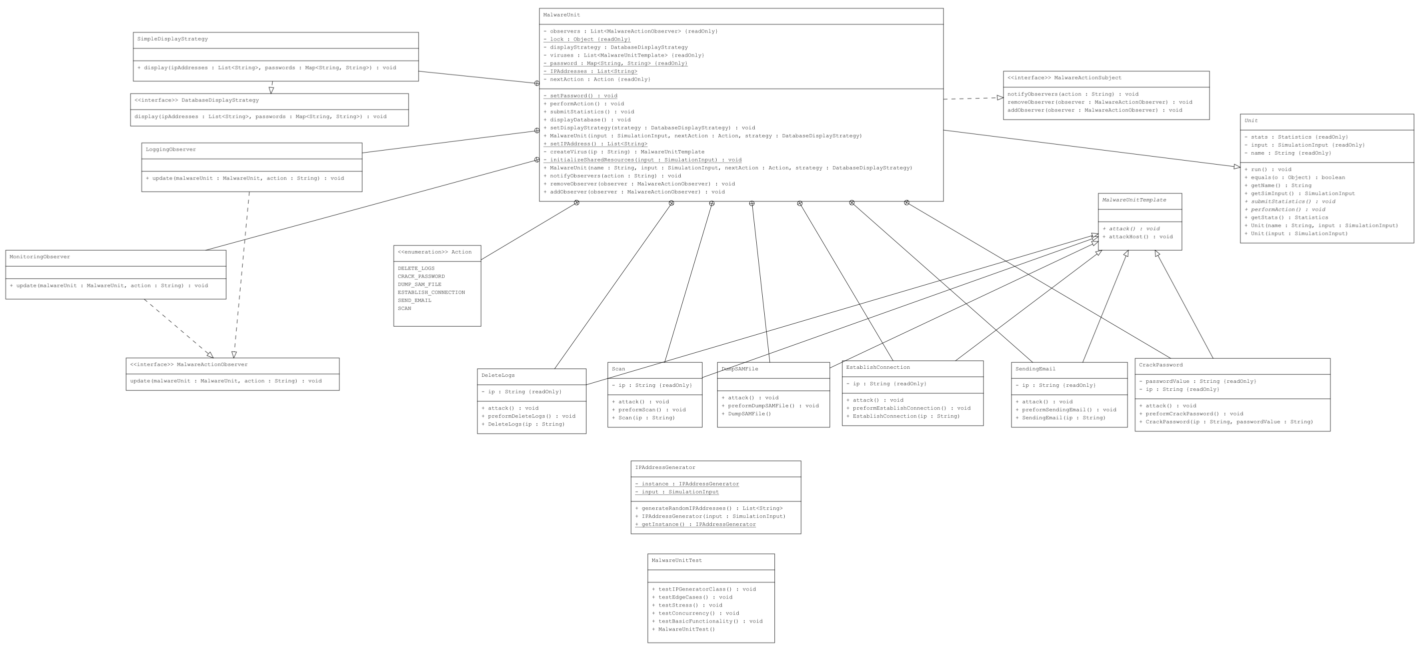
MalwareUnitTest Class

The MalwareUnitTest class is designed to conduct automated tests on the MalwareUnit class. It evaluates the functionality of malware simulations under controlled conditions to ensure that the malware behaves as expected across different scenarios.

Methods:

1. **testBasicFunctionality()**: Tests the basic operations of the MalwareUnit such as executing actions like scanning, sending emails, etc. It ensures that the unit can perform its fundamental tasks.
2. **testConcurrency()**: This test checks the MalwareUnit's ability to handle simultaneous actions on multiple threads, simulating a real-world scenario where a malware might have to interact with multiple systems concurrently.
3. **testStress()**: Pushes the MalwareUnit to its limits by requiring it to perform a high number of actions in a short amount of time. This tests the robustness and efficiency of the malware under stress conditions.
4. **testEdgeCases()**: Focuses on how the MalwareUnit handles boundary or unusual conditions, such as zero time to perform actions or an empty list of target IPs.
5. **testIPGeneratorClass()**: Focuses on testing the IPAddressGenrator class to see by giving an empty list with a time to preform the simulation.

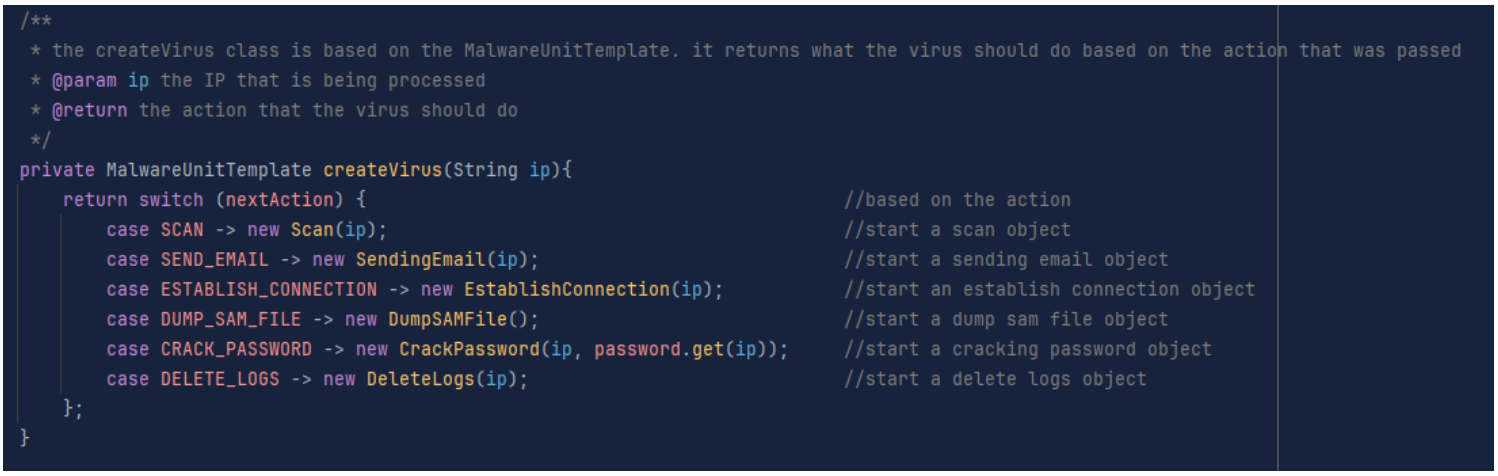
UML final



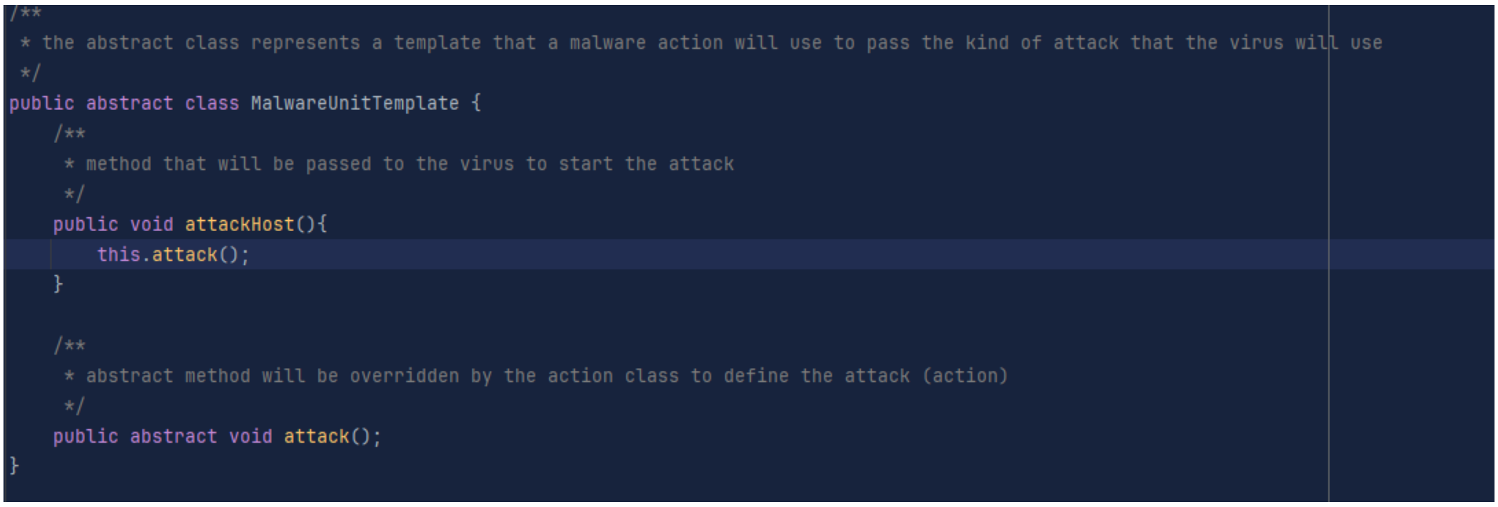
SimpleDisplayStrategy
+ display(ipAddresses : List<String>, passwords : Map<String, String>) : void
<<interface>> DatabaseDisplayStrategy
display(ipAddresses : List<String>, passwords : Map<String, String>) : void
LoggingObserver
+ update(malwareUnit : MalwareUnit, action : String) : void
MonitoringObserver
+ update(malwareUnit : MalwareUnit, action : String) : void
<<interface>> MalwareActionObserver
update(malwareUnit : MalwareUnit, action : String) : void
MalwareUnit
- observers : List<MalwareActionObserver> {readOnly}
- lock : Object {readOnly}
- displayStrategy : DatabaseDisplayStrategy
- viruses : List<MalwareUnitTemplate> {readOnly}
- password : Map<String, String> {readOnly}
- IPAddresses : List<String>
- nextAction : Action {readOnly}
- setPassword() : void
+ performAction() : void
+ submitStatistics() : void
+ displayDatabase() : void
+ setDisplayStrategy(strategy : DatabaseDisplayStrategy) : void
+ MalwareUnit(input : SimulationInput, nextAction : Action, strategy : DatabaseDisplayStrategy)
+ setIPAddress() : List<String>
- createVirus(ip : String) : MalwareUnitTemplate
- initializeSharedResources(input : SimulationInput) : void
+ MalwareUnit(name : String, input : SimulationInput, nextAction : Action, strategy : DatabaseDisplayStrategy)
+ notifyObservers(action : String) : void
+ removeObserver(observer : MalwareActionObserver) : void
+ addObserver(observer : MalwareActionObserver) : void
<<interface>> MalwareActionSubject
notifyObservers(action : String) : void
removeObserver(observer : MalwareActionObserver) : void
addObserver(observer : MalwareActionObserver) : void
Unit
- stats : Statistics {readOnly}
- input : SimulationInput {readOnly}
- name : String {readOnly}
+ run() : void
+ equals(o : Object) : boolean
+ getName() : String
+ getSimInput() : SimulationInput
+ submitStatistics() : void
+ performAction() : void
+ getStats() : Statistics
+ Unit(name : String, input : SimulationInput)
+ Unit(input : SimulationInput)
MalwareUnitTemplate
+ attack() : void
+ attackHost() : void
<<enumeration>> Action
DELETE_LOGS
CRACK_PASSWORD
DUMP_SAM_FILE
ESTABLISH_CONNECTION
SEND_EMAIL
SCAN
DeleteLogs
- ip : String {readOnly}
+ attack() : void
+ preformDeleteLogs() : void
+ DeleteLogs(ip : String)
Scan
- ip : String {readOnly}
+ attack() : void
+ preformScan() : void
+ Scan(ip : String)
DumpSAMFile
+ attack() : void
+ preformDumpSAMFile() : void
+ DumpSAMFile()
EstablishConnection
- ip : String {readOnly}
+ attack() : void
+ preformEstablishConnection() : void
+ EstablishConnection(ip : String)
SendingEmail
- ip : String {readOnly}
+ attack() : void
+ preformSendingEmail() : void
+ SendingEmail(ip : String)
CrackPassword
- passwordValue : String {readOnly}
- ip : String {readOnly}
+ attack() : void
+ preformCrackPassword() : void
+ CrackPassword(ip : String, passwordValue : String)
IPAddressGenerator
- instance : IPAddressGenerator
- input : SimulationInput
+ generateRandomIPAddresses() : List<String>
+ IPAddressGenerator(input : SimulationInput)
+ getInstance() : IPAddressGenerator
MalwareUnitTest
+ testIPGeneratorClass() : void
+ testEdgeCases() : void
+ testStress() : void
+ testConcurrency() : void
+ testBasicFunctionality() : void
+ MalwareUnitTest()

Design and Implementations decisions

The program is designed to create a new MalwareUnit object for each IP address that is being passed and a virus for each type of action. The viruses are created based on a switch that takes nextAction variable as the indication of what the virus should do:

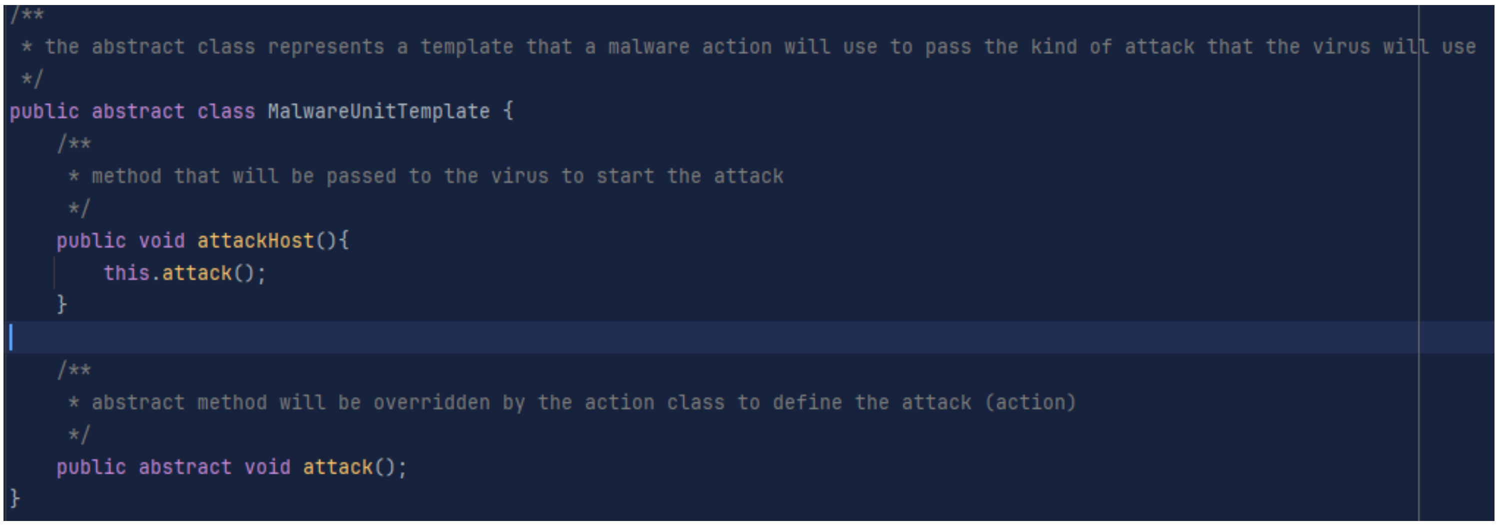

```java
/**
 * the createVirus class is based on the MalwareUnitTemplate. it returns what the virus should do based on the action that was passed
 * @param ip the IP that is being processed
 * @return the action that the virus should do
 */
private MalwareUnitTemplate createVirus(String ip){
    return switch (nextAction) {                                              //based on the action
        case SCAN -> new Scan(ip);                                            //start a scan object
        case SEND_EMAIL -> new SendingEmail(ip);                              //start a sending email object
        case ESTABLISH_CONNECTION -> new EstablishConnection(ip);             //start an establish connection object
        case DUMP_SAM_FILE -> new DumpSAMFile();                              //start a dump sam file object
        case CRACK_PASSWORD -> new CrackPassword(ip, password.get(ip));       //start a cracking password object
        case DELETE_LOGS -> new DeleteLogs(ip);                               //start a delete logs object
    };
}
```

Statistics are being submitted based on the Actions that the Malware unit preforms and how many active units are being created for the list of IPs that was given. For the actions that the virus preformes, each action is a separate class that extends its functionality from MalwareUnitTemplate which is an abstract class. The reason why I chose to make a template is to give me more flexibility with accessing the attacks, in addition to not have to initilize a new object every time.

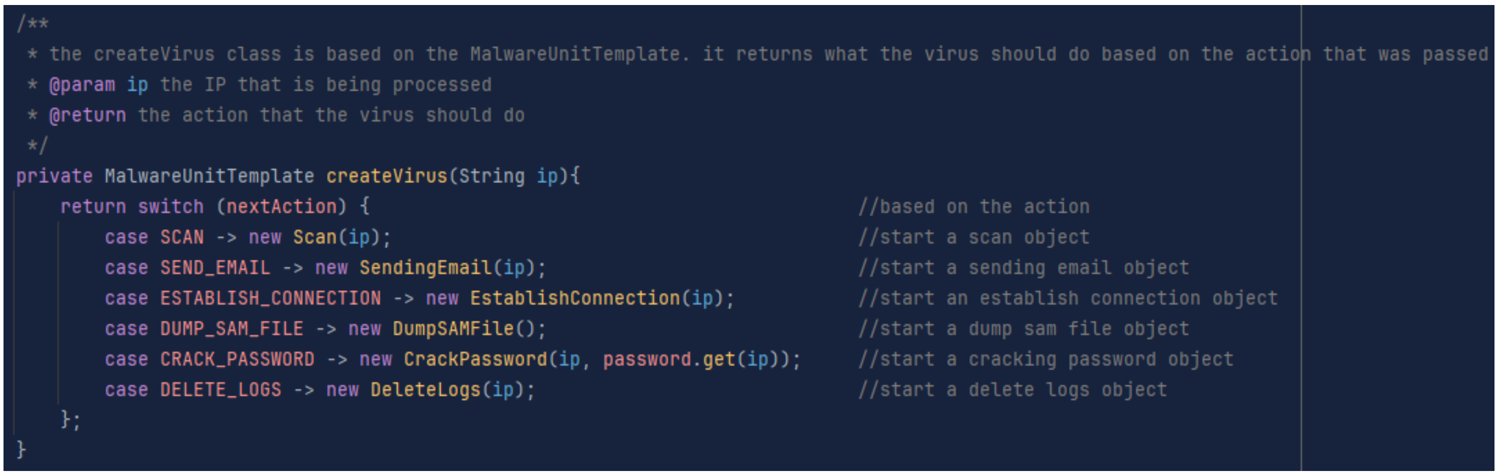

```java
/**
 * the abstract class represents a template that a malware action will use to pass the kind of attack that the virus will use
 */
public abstract class MalwareUnitTemplate {
    /**
     * method that will be passed to the virus to start the attack
     */
    public void attackHost(){
        this.attack();
    }

    /**
     * abstract method will be overridden by the action class to define the attack (action)
     */
    public abstract void attack();
}
```

Design Patterns that are used:

• **Template Pattern:** The class uses the template method pattern via its abstract superclass , where the method is overridden to execute different malware actions based on the malware unit's current state. Also, it is used with the attacks if the viruses. It is shown in the abstract class MalwareUnitTemplate and every action class extends from that class and override the method attack(). The reason why I chose the template method for the viruses actions because most of the actions has the same algorithem with some minor differences.

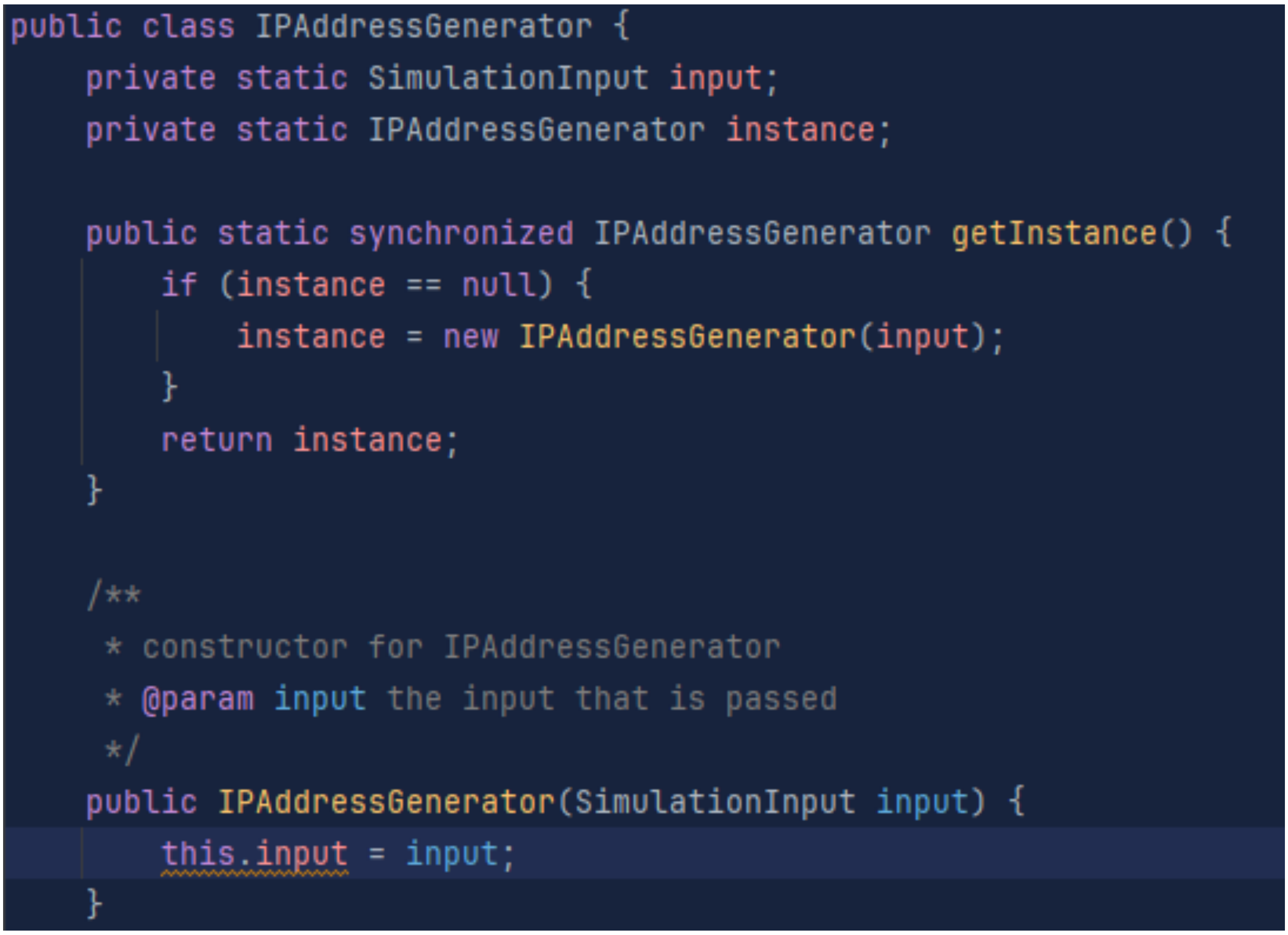


```java
/**
 * the abstract class represents a template that a malware action will use to pass the kind of attack that the virus will use
 */
public abstract class MalwareUnitTemplate {
    /**
     * method that will be passed to the virus to start the attack
     */
    public void attackHost(){
        this.attack();
    }

    /**
     * abstract method will be overridden by the action class to define the attack (action)
     */
    public abstract void attack();
}
```



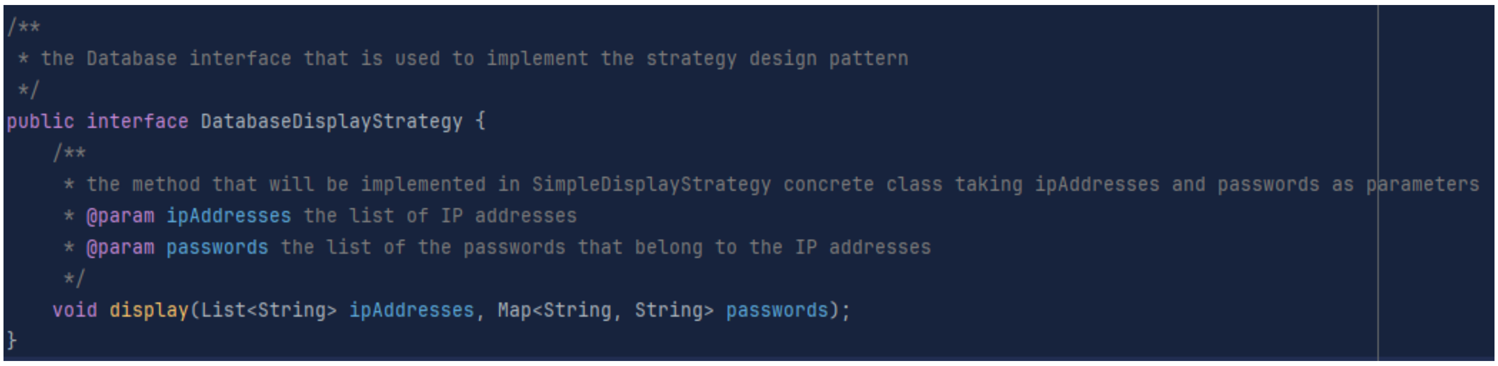
```java
/**
 * the createVirus class is based on the MalwareUnitTemplate. it returns what the virus should do based on the action that was passed
 * @param ip the IP that is being processed
 * @return the action that the virus should do
 */
private MalwareUnitTemplate createVirus(String ip){
    return switch (nextAction) {                                           //based on the action
        case SCAN -> new Scan(ip);                                         //start a scan object
        case SEND_EMAIL -> new SendingEmail(ip);                           //start a sending email object
        case ESTABLISH_CONNECTION -> new EstablishConnection(ip);          //start an establish connection object
        case DUMP_SAM_FILE -> new DumpSAMFile();                           //start a dump sam file object
        case CRACK_PASSWORD -> new CrackPassword(ip, password.get(ip));    //start a cracking password object
        case DELETE_LOGS -> new DeleteLogs(ip);                            //start a delete logs object
    };
}
```

• **Singleton Pattern:** Used in classes like `IPAddressGenerator` to ensure that there is a single instance of the IP addresses is generated for the viruses in case there is no input from the user. The reason why I chose Singleto is because all the viruses should work on the same IP addresses. therefore it is important that the IP generator has only one instance.



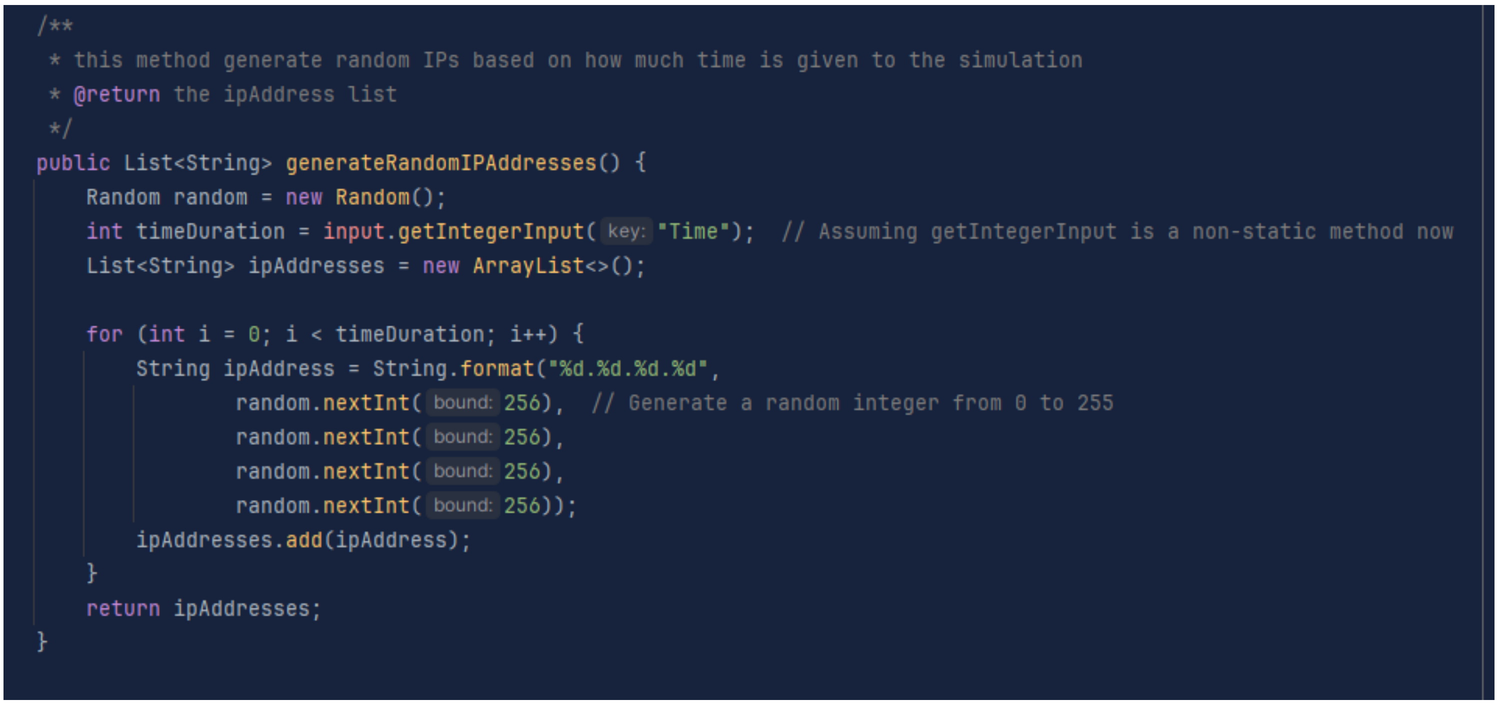
```java
public class IPAddressGenerator {
    private static SimulationInput input;
    private static IPAddressGenerator instance;

    public static synchronized IPAddressGenerator getInstance() {
        if (instance == null) {
            instance = new IPAddressGenerator(input);
        }
        return instance;
    }

    /**
     * constructor for IPAddressGenerator
     * @param input the input that is passed
     */
    public IPAddressGenerator(SimulationInput input) {
        this.input = input;
    }
```

• **Strategy Pattern:** The program utilizes the strategy pattern to change the display behavior of the simulation output through the `DatabaseDisplayStrategy` interface. This allows changing the output format without altering the virus objects. I chose the Strategy pattern because it allows us the flexibility to implement more classes that are more specialized with the representation of the data. In this program, this interface has been used to show the info that the viruses have like the IPs and their passwords. We could create another class that could display a more specified info about the target for example.

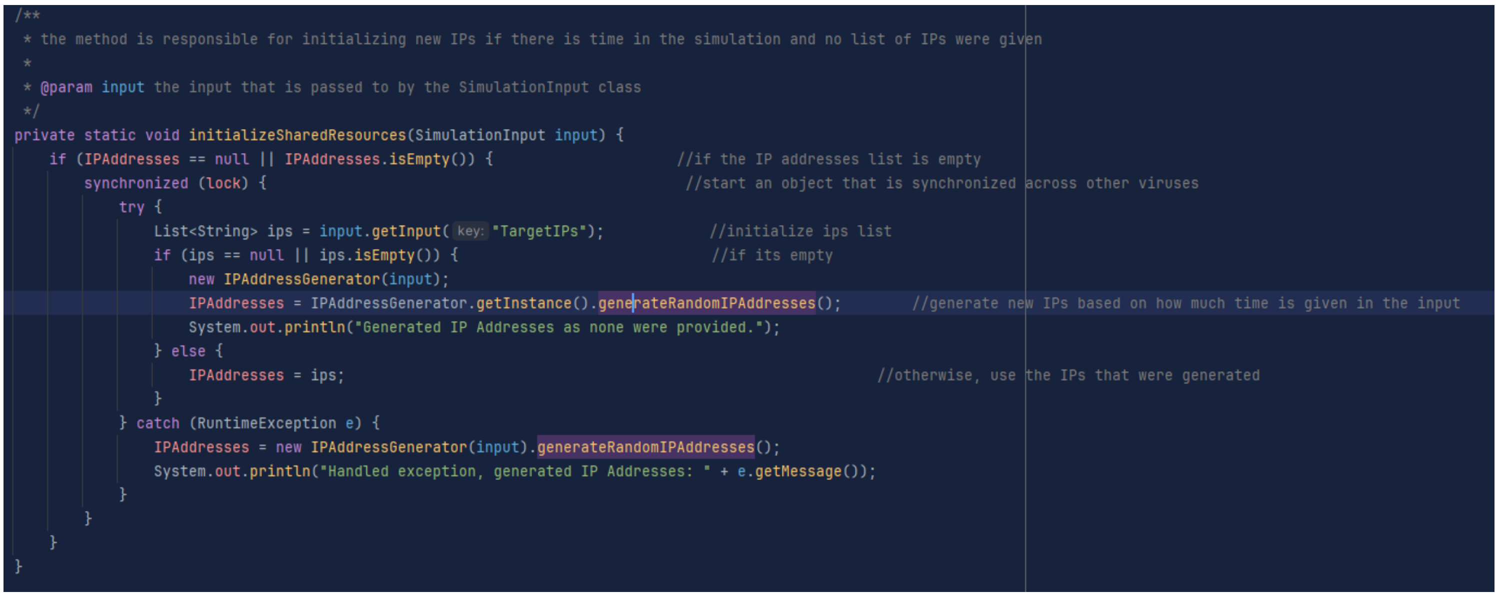


```java
/**
 * the Database interface that is used to implement the strategy design pattern
 */
public interface DatabaseDisplayStrategy {
    /**
     * the method that will be implemented in SimpleDisplayStrategy concrete class taking ipAddresses and passwords as parameters
     * @param ipAddresses the list of IP addresses
     * @param passwords the list of the passwords that belong to the IP addresses
     */
    void display(List<String> ipAddresses, Map<String, String> passwords);
}
```

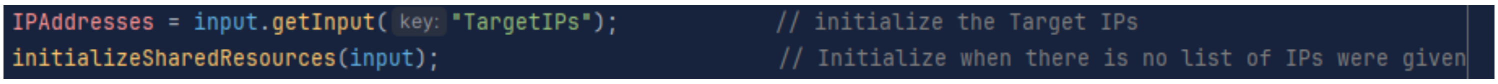
• **Factory Method:** The `IPAddressGenerator` class represents a factory method pattern that encapsulates the creation of IP address lists, abstracting the complexity of IP generation.



```java
/**
 * this method generate random IPs based on how much time is given to the simulation
 * @return the ipAddress list
 */
public List<String> generateRandomIPAddresses() {
    Random random = new Random();
    int timeDuration = input.getIntegerInput( key: "Time");  // Assuming getIntegerInput is a non-static method now
    List<String> ipAddresses = new ArrayList<>();

    for (int i = 0; i < timeDuration; i++) {
        String ipAddress = String.format("%d.%d.%d.%d",
                random.nextInt( bound: 256),  // Generate a random integer from 0 to 255
                random.nextInt( bound: 256),
                random.nextInt( bound: 256),
                random.nextInt( bound: 256));
        ipAddresses.add(ipAddress);
    }
    return ipAddresses;
}
```

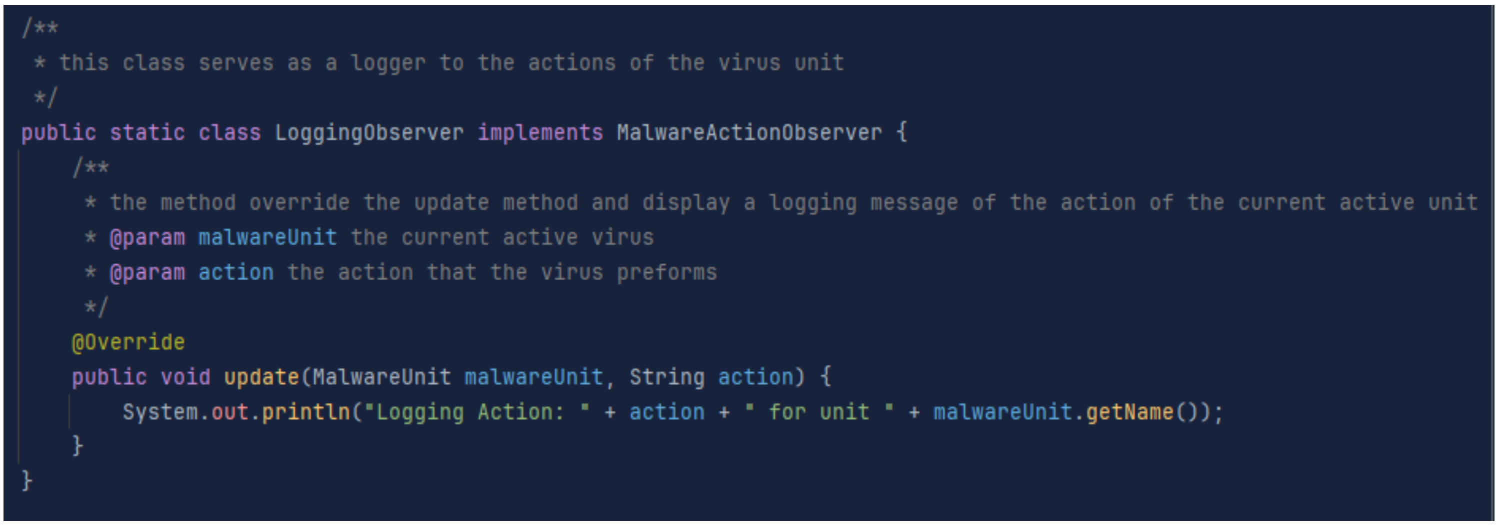


```java
/**
 * the method is responsible for initializing new IPs if there is time in the simulation and no list of IPs were given
 *
 * @param input the input that is passed to by the SimulationInput class
 */
private static void initializeSharedResources(SimulationInput input) {
    if (IPAddresses == null || IPAddresses.isEmpty()) {                    //if the IP addresses list is empty
        synchronized (lock) {                                               //start an object that is synchronized across other viruses
            try {
                List<String> ips = input.getInput( key: "TargetIPs");            //initialize ips list
                if (ips == null || ips.isEmpty()) {                          //if its empty
                    new IPAddressGenerator(input);
                    IPAddresses = IPAddressGenerator.getInstance().generateRandomIPAddresses();        //generate new IPs based on how much time is given in the input
                    System.out.println("Generated IP Addresses as none were provided.");
                } else {
                    IPAddresses = ips;                                                       //otherwise, use the IPs that were generated
                }
            } catch (RuntimeException e) {
                IPAddresses = new IPAddressGenerator(input).generateRandomIPAddresses();
                System.out.println("Handled exception, generated IP Addresses: " + e.getMessage());
            }
        }
    }
}
```

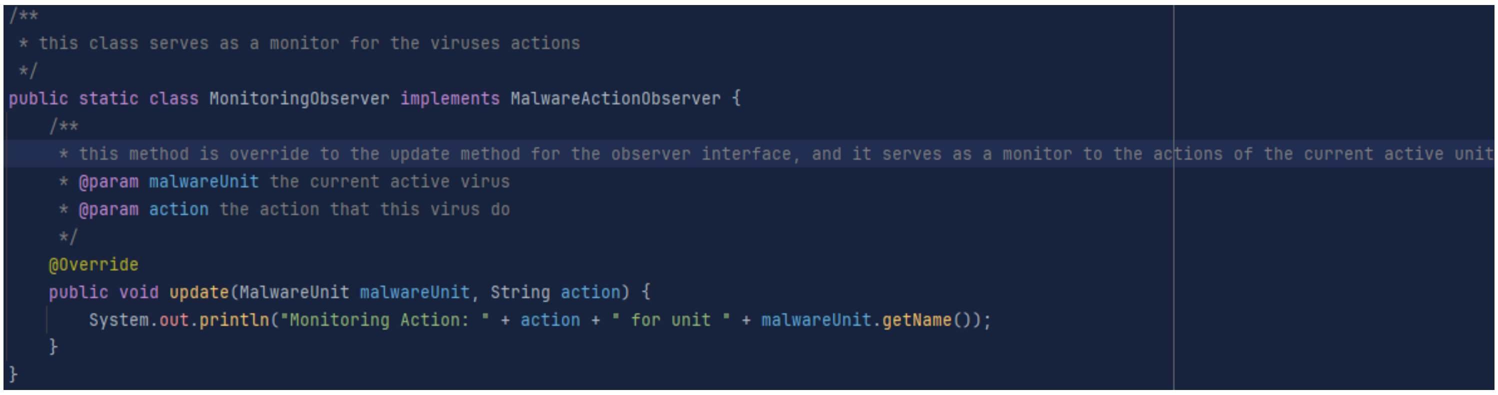


```java
IPAddresses = input.getInput( key: "TargetIPs");          // initialize the Target IPs
initializeSharedResources(input);                          // Initialize when there is no list of IPs were given
```

• **Observer Pattern:** I used the observer pattern to add the Logging and the Monitoring functionality to the MalwareUnit. I made 2 interfaces. MalwareActionObserver where this interface represents an observer in the Observer design pattern specific to malware actions. MalwareActionSubject is an interface where the actions like adding, removing, and notifying observer are being introduced. The update method are being overriden in the LoggingObserver and the MonitoringObserver classes where each class has its own functionality.



```java
/**
 * this class serves as a logger to the actions of the virus unit
 */
public static class LoggingObserver implements MalwareActionObserver {
    /**
     * the method override the update method and display a logging message of the action of the current active unit
     * @param malwareUnit the current active virus
     * @param action the action that the virus preforms
     */
    @Override
    public void update(MalwareUnit malwareUnit, String action) {
        System.out.println("Logging Action: " + action + " for unit " + malwareUnit.getName());
    }
}
```



```java
/**
 * this class serves as a monitor for the viruses actions
 */
public static class MonitoringObserver implements MalwareActionObserver {
    /**
     * this method is override to the update method for the observer interface, and it serves as a monitor to the actions of the current active unit
     * @param malwareUnit the current active virus
     * @param action the action that this virus do
     */
    @Override
    public void update(MalwareUnit malwareUnit, String action) {
        System.out.println("Monitoring Action: " + action + " for unit " + malwareUnit.getName());
    }
}
```

Choice of Logic and Data Structures:

• The program heavily relies on lists to manage IPs and concurrent threads to simulate real-time malware operations.
• is used to manage types of actions, ensuring a robust way of handling different operations by malware units.
• Hash Maps where used to store the passwords with its proper IP address.
• Switche was used because it was easier to us with enum to indicate the functionality of the virus than doing it with if else.
• Threads were used to start viruses instances.

Usage of DRY and SOLID Principles:

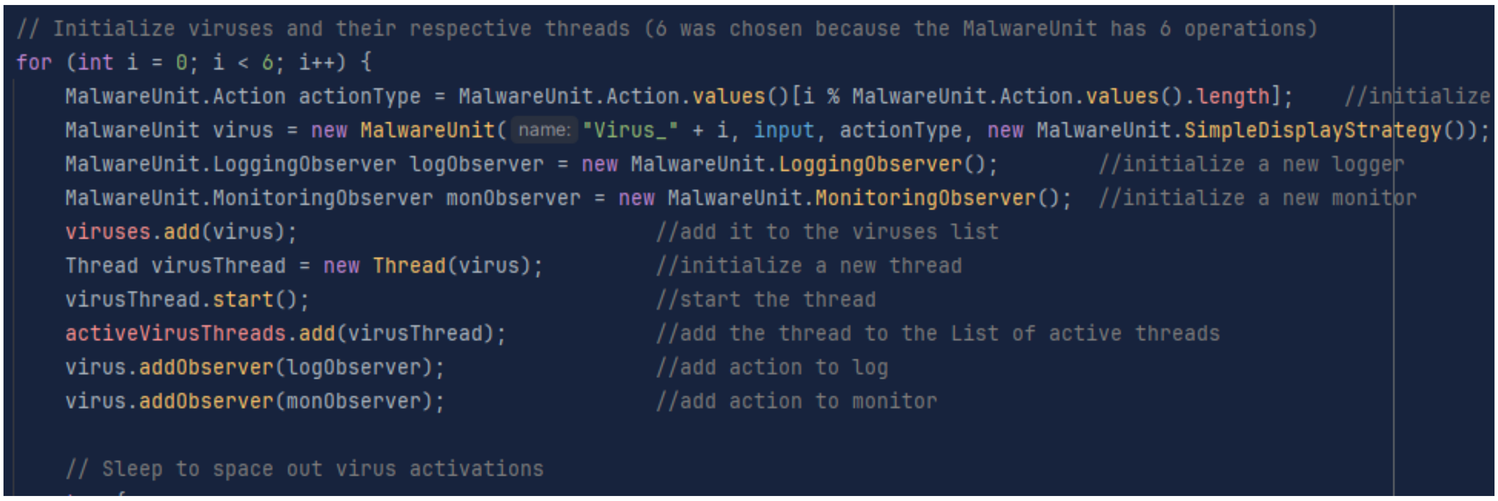
• **DRY (Don't Repeat Yourself):** Shared functionalities like IP generation and sleeping between actions are abstracted in separate methods or classes, reducing repetition.

• **SOLID:**
◇ **Single Responsibility Principle:** Each class has a single responsibility, e.g., only generates IPs.
◇ **Open/Closed Principle:** The system is open for extension but closed for modification, demonstrated by the ability to add new types of malware actions without altering existing code.
◇ **Liskov Substitution Principle:** Subclasses of like MalwareUnit class, can substitute for their parent without disrupting the expected behavior.
◇ **Interface Segregation and Dependency Inversion:** The use of interfaces in the program ensures that specific functionalities are not needlessly exposed to classes that do not require them. like the DatabaseDisplayStrategy interface where a class that implements it is only concerned about the specific logic that it wants to implement without altering the interface itself which achives the Interface Segregation. For Dependency Inversion, we can see this charcaristic is applied with all the

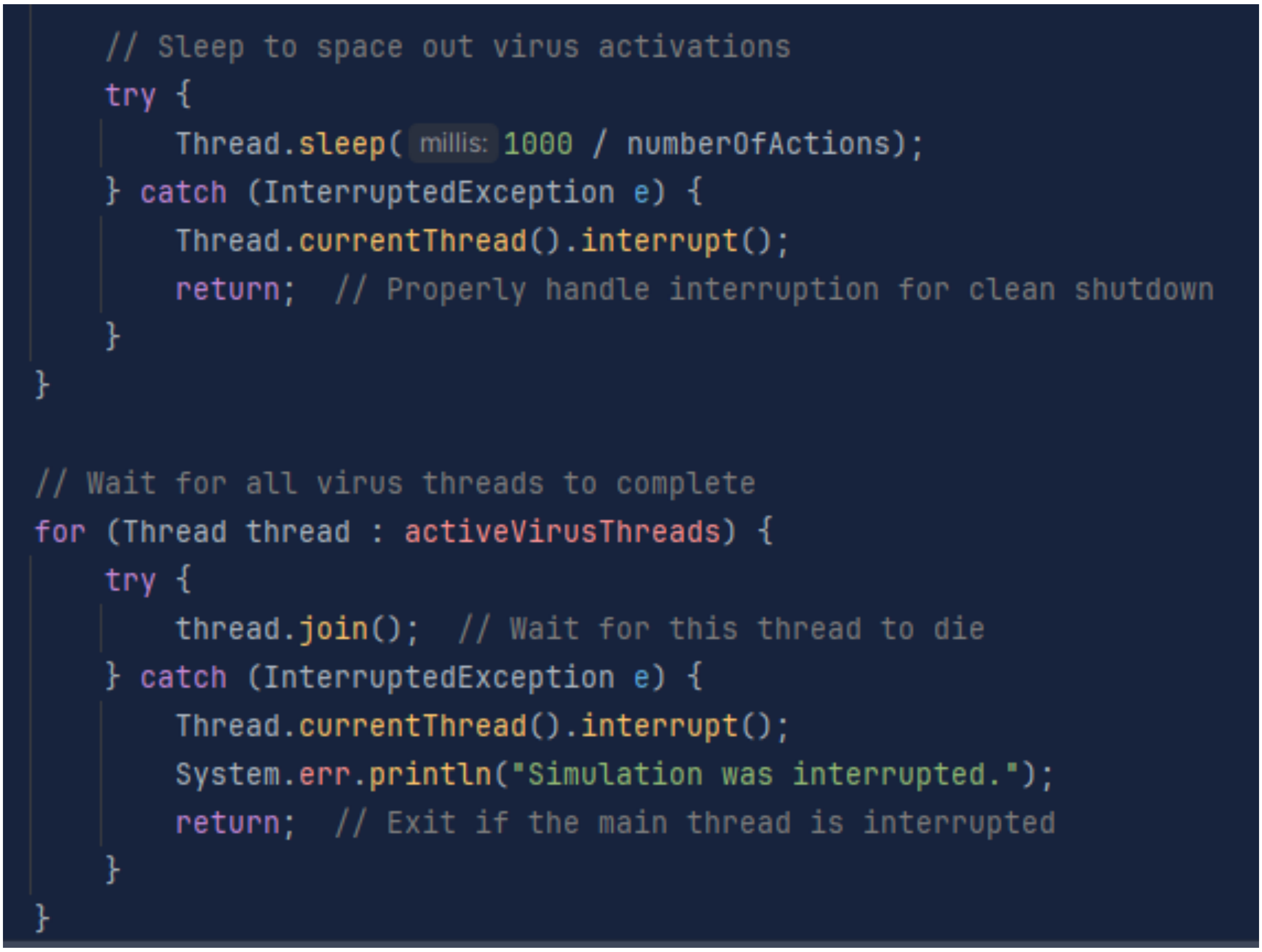
interfaces in the simulation, as well as, the abstract class MalwareUnitTemplate where we saw that abstraction like the template class does not depend on what the action is going to be, but the action classes like Scan depend on the abstract class.

Synchronization choices:

The one place that the Synchronization is presented is within the Matrix class where the viruses will be created as threads to run with the similation. The threads will be added to a list of threads so that each thread will have a turn and they would wait until the end of the previous thread to start:



```java
// Initialize viruses and their respective threads (6 was chosen because the MalwareUnit has 6 operations)
for (int i = 0; i < 6; i++) {
    MalwareUnit.Action actionType = MalwareUnit.Action.values()[i % MalwareUnit.Action.values().length];    //initialize
    MalwareUnit virus = new MalwareUnit( name: "Virus_" + i, input, actionType, new MalwareUnit.SimpleDisplayStrategy());
    MalwareUnit.LoggingObserver logObserver = new MalwareUnit.LoggingObserver();        //initialize a new logger
    MalwareUnit.MonitoringObserver monObserver = new MalwareUnit.MonitoringObserver();  //initialize a new monitor
    viruses.add(virus);                             //add it to the viruses list
    Thread virusThread = new Thread(virus);         //initialize a new thread
    virusThread.start();                            //start the thread
    activeVirusThreads.add(virusThread);            //add the thread to the List of active threads
    virus.addObserver(logObserver);                 //add action to log
    virus.addObserver(monObserver);                 //add action to monitor

    // Sleep to space out virus activations
```



```
    // Sleep to space out virus activations
    try {
        Thread.sleep(1000 / numberOfActions);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return;  // Properly handle interruption for clean shutdown
    }
}

// Wait for all virus threads to complete
for (Thread thread : activeVirusThreads) {
    try {
        thread.join();  // Wait for this thread to die
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        System.err.println("Simulation was interrupted.");
        return;  // Exit if the main thread is interrupted
    }
}
```