*Elite Lu*

*lue13*

*Spy Network Assignment*

Written Answers for the Assignment

Main Information

# UML Diagrams

The UML diagrams are in the folder. There are two of them, which are UML Diagram Q1 and UML Diagram Q2.

# Assignment Assumptions and Additional Information:

1. I will assume that dead spies cannot send or receive messages. This is because they are dead.
2. The remote bases will be connected to the home base by default.
3. For the UML diagram, the <<>> for the interfaces have been replaced with symbols containing ‘I’, ‘C’, and ‘A’ along with others to represent interface, class, abstract class, etc. I am unable to change it. I changed it for the + - # for the variables and methods.
4. Additional formatting includes the following
   1. The correct arrows for implementation, as the arrows are done as such because of how it is implemented
   2. The return variable type is specified for all the functions EXCEPT for the constructor. This is done to prevent any miscommunication
   3. Underlining is for static variables. This is mainly seen with the home base
   4. Variables with full caps as a name represents final variables
5. The main class is the client. It will be used for testing. It will NOT be included in the UML diagrams (I asked Dr. Maccio about this).

# Changes from Part 1 to Part 2

Going from part 1 to part 2, I added the decorator class, which is called EncryptionDecorator, to the UML diagram. This is because by adding the decorator, I was able to stack on different behaviours onto the decoding method. In order to not change a single thing, I made two decorators and a decorator abstract class. These are called CaesarDecorator and AlternatingDecorator. These are used specifically for the already premade ciphers CaesarCipher and AlternatingCipher. What these decorators will do is that it will call on these pre-existing ciphers and will encode and decode using them.

When using the main class, I can just stack on the different ciphers by making an object with type DecodeInterface. If it is the base (the first encoding method and the last decode method based on order), then I will call the Cipher classes rather than the decorator. In order to stack more of these encryption/decryption methods onto each other, I will make the variable equal to the decorator. Since the decorators take a DecodeInterface as a variable when instantiating, I can keep stacking it on. Here is an example:

Graphical user interface, text, application, chat or text message

Description automatically generated

As you can see, I first instantiate with a base. After that, I keep stacking/applying different encoding and decoding schemes. Changing the scheme works the same as before, as I call a method in the HomeBase class that will notify and update every single entity that is still connected to the HomeBase.

If I was able to change the previous encryption classes, I would be able to reduce a lot of clutter since I would not need to copy and paste the original encryption/decryption methods. I would have changed the original class line to extend from the EncryptionDecorator. The base encryption would be just returning the original base string and I would just do the same thing as before for changing the scheme.

Design Patterns

Here is a list of all the design patterns I used shown below:

* Singleton
* Observer
* Decorator
* Strategy

# Singleton

For singleton, this pattern is used when I want to ensure that there is only one object instance in the whole project. Because of this, this is extremely important for the home base. This is because the assignment states that there can be only one home base. Because of this, the singleton pattern is used to ensure this. It will check whether an instance of the home base class exists. If it does, then it will return the home base. Otherwise, a new home base object will be instantiated. This ensures that there is only one object because if the class is not static, then multiple home bases objects can be created. By doing this, I can ensure only one instance of the home base exists.

# Observer

This will be used for the field base objects. For context, the observer pattern is used when a subscription service esc mechanism is needed where the subscribers are notified and can freely subscribe or unsubscribe from the main publisher class. This was one of the main patterns I used. The first occurrence is the relation between the home base and remote base. Even though there is only one type of remote base specified in the question, using an interface to represent the main functions of the remote base and allows for me to incorporate the open closed principle, as I can implement the interface instead of editing classes. By using this design pattern, the remote bases can freely subscribe and unsubscribe and get updates if they are subscribed. In this scenario, the remote bases are automatically subscribed when they are created to the home base, which is the publisher. This is called reconnect and baseConnect in the different classes, as remote bases are connected to the home base (based on my original assumption that remote bases on creation are connected). The unsubscribe function would be the baseRemove, which disconnects the remote and home base. However, the remote base can always resubscribe/reconnect with the home base.

The second occurrence of the observer pattern would be the relation between the remote bases and spies. This relationship is a modified form of observer since the spies are unable to freely unsubscribe and resubscribe. This is reflected in how I coded it, as I check for whether the conditions are met for unsubscribing, which is death. Like the remote bases and home base, the spy is always subscribed and updated when connected to the remote base when alive. There is no actual way for a spy to resubscribe, not because I am not following the design pattern, but since when a spy dies, they cannot join any remote base as they are dead.

# Decorator

The decorator design pattern is used when new behaviours are to be attached to the object. This is done to prevent the programmer from changing previous code and having to go through it to implement additional features. This was mainly done with the different types of encoding and decoding. Based on the specifications, the second part states that I need to add multiple new traits (new coding schemes) and keep stacking them. Because I was unable to change the rest of my code, I resorted to this as I could stack encoding schemes onto the previous one.

However, there is a small issue. With the implementation of this, I would have to be decorating a decorator. What I mean by the fact that the object I am decorating is the same object that will be used as my base. Because of this issue, I created two classes

# Strategy

The strategy pattern is used when a class is specified and there are multiple different algorithms that need to be swapped on runtime. This was used for the different types of algorithms. When the user wants to change the encryption method, the context class is notified and changes the key and the algorithm. This was used specifically for the encryption/decryption schemes. I implemented this by making a method in the HomeBase entity that takes in a DecodeInterface variable. It will then proceed to change the method of encoding and decoding. Along with the observer pattern, it will notify all subscribers and also change their schemes and keys.

Design Principles

Some key design principles I have followed include some of the letters from the acronym SOLID along with other principles. I will first start off with SOLID and some of the principles from the acronym:

# SOLID

## Single Responsibility Principle

The single responsibility states that each function or module should be only responsible for doing a singular task. This prevents any unwanted consequences. For example, if I call a function to return the value of a variable, that variable should not be changed.

This can be seen throughout the assignment, as I used the principle in every method and class. For example, the getters and setters for some of the variables such as the key only do one thing. This allows for easy modification if I ever need to fix it.

## Open-closed principle

This means open for extension and closed for modification. This principle can be seen with the usage of the observer pattern. The main question that gets asked with this assignment is that why make an interface when there is only one type of remote base and spy. By doing this, if I ever wanted to incorporate different types of spies such as ultimate spies or overseas remote bases that perform differently with the same functions, I have a template with the interfaces and I can just implement them without having to modify all my code.

## Liskov Substitution

Note: We did not cover this in class, but because it was one of the only letters in SOLID not covered, I wanted to do so.

Liskov substitution is where all subclasses can replace their base classes. One of the most popular examples would be the rectangle and the square extending from the rectangle. In this assignment, I used it for the MessageSender since I extended it in many other classes. Whenever I made an object a MessageSender, I would be able to replace it with specific classes such as the normal remote base and spy.

## Interface Segregation

The interface segregation principle states that the classes or clients should not be dependent on interfaces they do not use. What this means is that I should not have methods that the classes or objects that implement it do not use.

This is seen in the interface used for designing all the entities in the system. I made it in such a way that the interfaces used for the remote base and spy only have the functions that they need, and I separated the functionality for the message senders in an abstract class. For example, I prevented entities such as spies from using methods such as going black, a method reserved solely for the remote bases. In addition to this, remote bases are not able to access some functions such as died, which checks if a spy has died or kill, which is a function that kills spies.

Another example I did for this was the interface EntityObject. This interface only has two functions, which include getName and update. This interface is implemented in the abstract class MessageSender and extended in the interfaces SpyInterface and RemoteBaseInterface. This is because all these entities need to use the functions getName, which returns a name, and the update function, which updates the entities connected and itself. If I did not do this, all the classes and interfaces would have to either have different methods with the same functionality or overriding the method if those corresponding methods had the same name.

## Dependency Inversion/Design for Interfaces

The dependency inversion principle is where modules, regardless of level (high-level or low-level), are not dependent on modules lower on the hierarchy. What this means is that high-level modules are not dependent on low-level modules. Instead, all of them are dependent on abstractions. In this circumstance, that would mean that I would use interfaces and abstract classes, which would be abstractions of the classes. This is most apparent in the interfaces for the spy and remote bases. I implemented these interfaces for the spies and normal remote bases and these interfaces define some set functions that each class should have. Even though there is only one type of remote base called normal remote base and one type of spy called spy, having the interface allows for future classes the implement spy interface and remote base interface without having to modify any previous legacy code.

Code Functions

To start off, I will talk about each part of the code, which includes the home base, the remote bases, the spies, and the encoding/decoding methods. In addition to this, I will mention every function that is used and a brief description of what they do along with some parameters. Note that I will not be repeating functions. What I mean is that if an interface has a function that is used in a class, I will not reexplain what it does. If I did that, this document would be even longer. This does not mean I will not include the functions with the same name from interfaces and abstract classes, which include getName and update.

# Interfaces

## RemoteBaseInterface

This interface is used for creating different types of interfaces. This interface extends from EntityObject. This interface has the following functions with their descriptions of their purpose when implemented:

* public void addSpy(SpyInterface s): Adds a spy to the
* public void removeSpy(SpyInterface s): Deletes the
* public void printSpies(): Prints out every spy connected to the remote base
* public void goDark(): Disconnects from the home base.
* public void reconnect(): Reconnects to the home base.

## SpyInterface

The SpyInterface is used for making spies. This interface extends from EntityObject. This interface has self-explanatory functions which are the following:

* public void kill(): Kills the spy and removes/unsubscribes the spy from the remote base.
* public boolean died(): Determines whether the spy is dead and returns a boolean stating that.

## EntityObject

This interface is used for the functions getName and update. This is done to prevent from two interfaces having to use the same method (Originally, SpyInterface, RemoteBaseInterface, and the abstract class MessageSender need to update and get the name.)

* public String getName(): Returns the name of the entity
* public void update(int key, CipherContext c): Updates the entity’s key value and CipherContext.

## DecodeInterface

This is the interface for the encoding and decoding schemes. This gets used specifically for the schemes Caesar Cipher and the Alternating Cipher. In part 2, the decorator abstract class implements this interface. The main functions in here are encrypt and decrypt, which both take in a string and an int. Here are the functions shown below:

* public String encrypt(String sentence, int key): Encrypts the message.
* public String decrypt(String sentence, int key): Decrypts the message.

# Abstract Classes

## MessageSender

This abstract class is used for the base entities and the spy entities. This interface implements from EntityObject. This includes the home base, NormalRemoteBase, and Spy.

* private int key: Variable for the key
* private DecodeInterface c: Variable for the DecodeInterface
* private String message: Variable for the previous message that was received by the entity
* public void sendMessage(String message, MessageSender receiver): Sends a message to the receiver by first encrypting the message and printing it out.
* public void receiveMessage(String message): Receives the message and decrypts the message.
* private void storeMessage(String message): Store the message after the message has been decrypted. Only stores the last message that the entity receives.
* public void printMessage(): Prints the last message received by the entity.
* public abstract String getName(): Returns the name of the entity. Implementation is left for the entity that extends this class since it allows me to write a singular getName function in each entity to override depending on the context.
* public void update(int key, DecodeInterface c): Update function that updates the key and the cipher context in MessageSender. The classes that extend it will call it with super.

## EncryptionDecorator

This class implements DecodeInterface, which is the interface that is used for the schemes for encryption and decryption. Because of this, all the methods other than the constructor will be included since I explained what that method is used for in the interfaces section. The abstract class is used via extension by classes AlternatingDecorator and CaesarDecorator. The following variables and unique methods are shown below:

* private DecodeInterface b: Stores the schemes that are used for the encoding and decoding
* EncryptionDecorator(DecodeInterface b): Constructor for EncryptionDecorator

# Classes

For classes, I will be splitting it up into three sections. The first section will include the entities, which are NormalRemoteBase, HomeBase, and Spy. The second section include the encryption decryption scheme classes used in part 1 of the assignment. The final part will go over the decorators.

# Entities

## HomeBase

The HomeBase class is used for the home base. This class extends from MessageSender. This class is used for the singleton design pattern. Most of these are protected since I did not want any classes from outside the project to access them. Here are the functions and variables that are not from the abstract class:

* private final String NAME:
* private static HomeBase homeBase:
* private ArrayList<RemoteBaseInterface> connectBases: Stores a list of all the remote bases connected to the home base
* private int key: Stores the key. Not a duplicate since this variable is used to store the key that will be used to update other remote bases and spies.
* private DecodeInterface c: Variable to store the DecodeInterface, which has the encode and decode methods.
* private HomeBase(): Constructor for HomeBase. It is private since it will only be called if there is not an object instantiated for the variable named homeBase.
* protected static HomeBase getInstance(): This method either instantiates the class if there is no instantiation or will return the class object if there is already one.
* protected void baseConnect(RemoteBaseInterface r): Enrolls a remote base to the home base and adds the RemoteBaseInterface object to the ArrayList.
* protected void baseRemove(RemoteBaseInterface r): Removes the remote base from the ArrayList of RemoteBaseInterface.
* protected void changeKey(int value): Changes the key and then calls the update function to update any entities in the system with this change in key.
* protected void changeMethod(DecodeInterface method): Changes the method of encryption and decryption along with updating the DecodeInterface and the other entities in the system.

Note: When I say system, I mean all the remote bases that are still connected to home base and the spies that are connected to each remote base.

## NormalRemoteBase

This class extends from MessageSender and implements RemoteBaseInterface. This class is used to be just as its name implies, a normal remote base. The variables and the constructor are shown below, as all the methods are from the abstract class and multiple interfaces:

* private final String NAME:
* private ArrayList<SpyInterface> spyNetwork: Stores all the spies connected to the remote base.
* private HomeBase b: Stores the homebase. If the remote base and the homebase are disconnected, this variable is made null.
* protected NormalRemoteBase(String name, HomeBase b): Constructor for NormalRemoteBase.

## Spy

* private final String NAME: The name of the spy. Cannot be changed.
* private boolean isDead: Boolean that states if the spy is dead.
* private RemoteBaseInterface base:
* protected Spy(String name, RemoteBaseInterface base): The constructor of the Spy class.

# Encryption and Decryption Schemes

## CaesarCipher

In part 1 of the assignment, this is the original Caesar cipher. In part 2, it became the base of the decorators. The CaesarCipher implements the interface DecodeInterface. There are no variables and no additional methods, as the only two methods originate from DecodeInterface, which are encrypt and decrypt.

## AlternatingCipher

Like CaesarCipher, this was the original Alternating cipher class and in part 2, becomes another base for the decorators. AlternatingCipher implements the interface DecodeInterface. There are no variables but one additional method, which is an auxiliary function called alternator, which takes the string being encoded or decoded, a boolean, and the key.

For context, alternating cipher is a bit like CaesarCipher. What it does is that it will check if the character in each part of the string and will determine if it is a letter. If it is a letter, then the first letter will add the key to the ASCII value, the second letter would subtract the key, and will continue until the end. The decrypt would reverse this algorithm.

# Decorators

The two decorators, CaesarDecorator and AlternatingDecorator, are copied from the corresponding schemes. However, the difference is that the decorators implement the EncryptionDecorator. In addition to this, both decorators use the super line in order to access the stacked schemes. Finally, both decorators have a constructor that stores a DecodeInterface variable. This would be how these decorators access the rest of the schemes stacked.