Assignment Completed by Hilary He, Timothy Leung
MACID: leungt30
MACID: hez88

Part 1:

Design Patterns Applied:

- Observer Patterns (HomeBase, FieldBase, and Spy)
- Singleton Pattern (HomeBase)
- Strategy Pattern (EncryptionScheme)

Observer Patterns: One of the requirements specified in the assignment is: “Field bases are associated/registered with the home base, and spies are associated/registered with a single field base”. Thus we created two observer patterns, where field bases are observers to the home base, and spies are observers to the field base. The observer pattern is able to smartly avoid cycles between the observers and the subject.

Singleton Pattern: The assignment specified that there is only a single home base, thus we applied the singleton pattern to the home base. It allows the program only to initialize the home base once, and no more home bases will occur inside the program.

Strategy Pattern: There can be multiple different algorithms to create the encryption scheme, thus a strategy pattern is used to implement encryption methods of the message. It encapsulates all the algorithms which satisfies the open-closed principle, which makes adding another algorithm easier.

Design Principles Enforced:

- Single responsibility principle/information hiding
- Open-closed principle
- Interface segregation principle
- Dependency inversion principle

Single responsibility principle: The Data class encapsulates all the arraylist implementations used in the program. Hiding the data structure that is most likely to be changed, so that further iterable objects, such as stacks, queues, arrays can all be implemented with only modifying the Data class.

Open-closed principle: Using a strategy design pattern to implement different encryption algorithms allows new algorithms to be included with no change to any other coding parts in the program. The Data encapsulation to the arraylist data structure. The home base, field bases and spy are all based on abstract interfaces, thus further extension can be performed. Having a <**FieldBaseInterface**> that includes every functionality a field base needs allows extending further changes to a concrete field base to be simply implementing that interface.

Interface segregation principle: The concrete field base implements two interfaces, we separated methods that are available to the home base and the spies, as a result, they are not exposed to methods that they should not use.

- <**FieldBaseHOMEInterface**> - only has methods **update()**, which allows the home base to call it when the scheme and key is updated. **unregister()**, which allows a field base to "go dark"; **register()**, allows a field base to register itself with the home base. Since the home base class is only aware of the <**FieldBaseHOMEInterface**>, it only has access to these three methods, which are intended for the home base.
- <**FieldBaseInterfaceSPY**> - only has methods, **registerSpy(SpyInterface s)**, **getScheme()**, **getKey()**: allows the spy to update their scheme and key once the home base notifies all field bases. **deadSignal(Spy s)**, which sends the dead signal to the field base once a spy is dead. The spy class only has access to this part of the field base, which is intended.

Some methods are associated solely with the concrete class, consequently, these methods are hidden from other classes that use/have the class, because other classes initialize the class with its interface, instead of the concrete class. (also considered as the single responsibility/information hiding) These methods include:

HomeBase:

- **update() -** calls notifySubject() in the home base class, no other classes should have access to this method, thus private
- **notifySubject() -** calls the update method associated with the field bases registered with the home base, no other classes should have access to this method, thus private

- **setScheme(EncryptionScheme scheme)** - the home base sets the scheme, then calls **update()**
- **setKey(int key)** - the home base sets the key and then calls **update()**

Field bases registered with the home base should not have access to these methods, however, the runner need **setScheme(EncryptionScheme scheme)** and **setKey(int key)**, thus they are associated with the class itself, and the runner should initialize the concrete home base class instead of the interface.

ConcreteFieldBase:

- **notifySpies()** - calls the update method associated with spies registered with the field base. It is private because both the spy and the home base should not have access to
- **unregisterSpy(SpyInterface s)** - unregister dead spies, both the home base and the spy should not have access to it, thus it is private

Spy

- **isDead()** - checks the status of a certain spy, the runner probably wants to use it. Thus is it associated with the concrete class
- **die()** - changes the status of a certain spy, the runner probably wants to use it to modify the live status of spies to trigger the dead signal and unregister spies
- **getFieldBase()** - gets the field base that a spy is registered with. Again, probably useful for the runner

All of the above methods are not necessary for the field base that a spy is registered with. Thus they are associated with the class

Dependency inversion principle: All three entities depend on abstract interfaces rather than concrete classes, meanwhile, they all depend on interfaces to send and receive messages, namely, <**messageSender>** and <**messageReceiver**>. All scheme algorithms depend on a Scheme interface as well.

Part 2:

Design Pattern Applied:

- Decorator pattern

Decorator pattern: To solve this portion of the assignment, we decided to use the decorator pattern. Firstly, create a new abstract class called **EncryptionDecorator**, “is a” and “has a” relationship to the <**EncryptionScheme**>, then concrete decorators/encryption wrappers can extend the **EncryptionDecorator** class. A decorator pattern is used primarily because there is no internal logic between wrappers for encrypting messages, and they are all symmetric encryption which allows decryption to simply “unwrap” the encrypted message. **Scheme1** and **Scheme2** are treated as default schemes to use.

This way of implementing part 2 also satisfies the open-closed principle. Adding a new decorator does not affect any other part of the code. The fact that we can design to incorporate this add-on to our previous implementation without modifying any existing code is a demonstration of the open-closed principle. Moreover, it also satisfies the dependency inversion principle, all concrete decorators are dependent on an abstract decorator class instead of concrete classes.