Andrew Goetz (goetza3)
Ryan Spurgeon (spurgeor)

Assignment 2

COMPSCI 2ME3
November 18, 2022

1 The Problem - Part 1

1.1 UML Diagram

To make it easier to understand, we have separated the structure diagram (see Figure 1) from the details of each class (i.e. their attributes and methods) which are in a table below. We did this so that our diagram may help the reader understand the overall structure of our code.

Name	Attributes	Methods
«iMessage»	+data	
«MessagingMachine»	-scheme, -key	+encrypt(iMessage)→iMessage, +decrypt(iMessage)→iMessage, +updateScheme(Scheme)→void, +send(iMessage)→void, +receive(iMessage)→void, +getScheme()→Scheme
«iContactList<T>»		+add(T)→void, +remove(T)→void, +getAt(int)→T, +length()→int
«Unit»		+send(String, Unit)→void, +receive(iMessage)→void
«Field»		+getUpdate(Scheme)→void, +goDark()→void, +sub()→void, +regSpy(SpyObserver)→boolean, +killSpy(SpyObserver)→void, +getScheme()→Scheme, +updateAll()→void, +sendUpdate(SpyObserver)→void
«Scheme»		+encrypt(iMessage, int)→iMessage, +decrypt(iMessage, int)→iMessage, +getKey()→int, +setKey(int)→void
«SpyObserver»		+die()→void, +getIsDead()→boolean, +getIsReg()→boolean, +register()→void, +getUpdate(Scheme)→void, +send(String, Unit)→void, +receive(iMessage)→void
«HomeSubject»		+updateAll()→void, +sendUpdate(Field)→void, +setScheme(Scheme)→void, +unsub(Field)→void, +subscribe(Field)→void, +getScheme()→Scheme
HomeShell	-home	+subscribe(Field)→void, +unsub(Field)→void, +getScheme()→Scheme
Spy	-isDead, -field, -isReg, -messenger	+send(String, Unit)→void, +receive(iMessage)→void, +die()→void, +getIsDead()→boolean, +getIsReg()→boolean, +register()→void, +getUpdate(Scheme)→void

Name	Attributes	Methods
Message	+data	
Contacts<T>	-contacts	+add(T)→void, +remove(T)→void, +getAt(int)→T, +length()→int
FieldShell	-field	+getScheme()→Scheme, +killSpy(SpyObserver)→void, +regSpy(SpyObserver)→boolean
Home	-homeBase, -fieldBases, -messenger	+getInstance()→Home, +send(String, Unit)→void, +receive(iMessage)→void, +subscribe(Field)→void, +unsub(Field)→void, +updateAll()→void, +sendUpdate(Field)→void, +setScheme(Scheme)→void, +getScheme()→Scheme
AField	-homeShell, -goneDark, -messenger, -spies	+getUpdate(Scheme)→void, +send(String, Unit)→void, +receive(iMessage)→void, +goDark()→void, +sub()→void, +regSpy(SpyObserver)→boolean, +killSpy(SpyObserver)→void, +getScheme()→Scheme, +updateAll()→void, +sendUpdate(SpyObserver)→void
AScheme	-key	+encrypt(iMessage, int)→iMessage, +decrypt(iMessage, int)→iMessage, +getKey()→int, +setKey(int)→void
BScheme	-key	+encrypt(iMessage, int)→iMessage, +decrypt(iMessage, int)→iMessage, +getKey()→int, +setKey(int)→void

1.2 Design Patterns

- *Singleton* - was used for the `Home` class so that there can only be one instance of it at any time
- *Observer (push)* - was used in two different contexts. We first created a subject-observer relationship between `Home` and `Field` objects where `Home` acts as a subject that holds information like the encryption scheme and `Field` is an observer that is updated whenever the encryption scheme changes. The other context in which we used it was for the relationship between the `Field` and `Spy` classes where `Field` keeps track of `Spy` contacts and updates them whenever there is a change to the scheme.
- *Strategy* - was used for the relationship between `MessagingMachine` and `Scheme`. What we did with this is made an external object that every "unit" (home, field-base, or spy) will create for them-self upon instantiation that will handle all of the sending, receiving, and en-/de-crypting of message. The scheme it uses for encryption depends on what `Scheme` object it has. Every `Scheme` object contains the key it will use as well which can

be modified by the `MessagingMachine`. We chose to do it this way because depending on the scheme, the format of the key may be different but we still want to keep the key dynamic and easily changeable by the unit who is using the messaging device.

1.3 Design Principles

1.3.1 Single Responsibility

The single responsibility principle is diligently enforced throughout the design. In short, the single responsibility principle dictates that a class should have only one reason to change. The most obvious example of this in our design is the messaging system. Although each class is forced to implement send() and receive() methods, they deal with a MessagingMachine class that handles abstract iMessage and Message types. This enforces single responsibility as although each unit that invokes the Messaging classes instead of handling messaging themselves.

Some additional examples of our design enforcing Single-Responsibility:

- Scheme/encryption: Each unit has a MessagingMachine object which has a scheme which handles encryption and decryption for it – notice the responsibility of MessagingMachine object is greatly limited by abstracting encryption into a scheme interface. Also scheme classes only encrypt/decrypt Messages.
- Observer's iContactList: Each subject maintains a list of observers type iContactList implemented by the Contacts class. The contact list has only one job, and that job is abstracted out of the Subject classes.
- Each `Unit` implementation (Home, Field, Spy) is only directly responsible for maintaining the network through the observer pattern. That is sending/receiving up to date scheme information. Thus, the only reason it must change would be a change in this aspect.
- The Unit interface only ensures that all Field Bases, Spies, the Home instance have a way of sending messages. Though, it is not the responsibility of Units to create messages – these details are abstracted.

The single responsibility principle is a driving force behind our design as all main functionality is divided across several classes, each performing a single duty i.e. message sending (MessagingMachine), encryption (Scheme), message representation (iMessage), network updates (Observer/Subject implementations like Spy).

1.3.2 Open-Closed Principle

The Open-Closed principle dictates that files/classes should be open to extension but closed to modification. One way our design maintains this principle is through the use of interfaces

for abstracting Message, Contact List, and Scheme types:

- Unit Classes like Home, Field Base, and Spy, use the iMessage interface to denote the message type. Under the hood it is just a string; should that ever change, the classes that depend on Message types would not need to change.
- In the same spirit, Subject classes like Home and AField use types of the iContactList interface (to maintain list of observers), whose internal logic is hidden.
- Furthermore, Schemes are abstract, Messaging systems that rely on encryption schemes are unaware of its internal logic, and unaffected by changes since they implement an interface.
- Another reason why the design upholds open-closed is because all observer units (Spy and Field) implement an interface that allows for extendibility. We can make a new implementation of «Field»or «Spy»by extending interfaces; without changing their subjects
- Finally, the entire network of units that can send messages to each other is extendable simply by implementing another Unit type. The existing units (like Spy, Home, Field Base) only demand an object type Unit when sending messages.

1.3.3 Design for Interfaces (Dependency Inversion - Strategy)

The design for interfaces aka dependency inversion principle is maintained in the design by using interfaces which create a barrier between high and low-level implementations. This is most evident in the MessagingMachine class.

- This class employs the Strategy pattern where each concrete Scheme (encryption scheme) is a strategy which implements the Scheme interface (Strategy interface). The high-level implementations being the Context (MessagingMachine), and its clients (like Spy, Field Base) are unaffected by changes made to these implementations. New Strategies (encryption schemes) can be created, while the high-level implementations are only concerned with the interface.
- Another example of high-level modules not depending on low-level modules both depending on abstractions is MessagingMachine objects depend on iMessage (abstraction) to represent messages while concrete messages also only interact with the interface (creating an interface layer between levels). This is also, in part, the motivation behind having the interface iContactList between the Subject classes (like Home) and the concrete contacts (strategies).

1.3.4 Interface segregation (Observer-Shell Pattern)

The principle of interface segregation is reinforced using shell subjects in the observer pattern used to implement the network. This is because HomeShell and FieldShell keep only the methods which are the concern of the observers, and observers only keep a shell instance of the subject. This is further reinforced by placing a shell subject with a private instance of the full subject in the same package as the observer. The shell subject can choose which subject methods to implement and thereby give the observer access to. Subject Methods like update and setters are not contained in the shell thus preventing observer from knowing about/using them.

1.3.5 Information hiding

Many of the above instances of design patterns are heavily correlated with information hiding. However, some specific reasons why the design upholds this principle are:

- Units (Spy, Home, Field Base) are not aware of the internal logic of encryption, message sending/receiving, or message representation (iMessage type).
- Subject classes (Home, Field) are unaware of the internal logic of how their Observer list (iContactList) is implemented; this is hidden as all subjects use the interface.
- Using shell subjects (interface segregation), Observers do not have access to (know about) subject methods like update or setScheme. Observers (like Field and Spy) only know about the subject methods that are necessary.
- Although units can send messages between each other – they do not have access to each other's fields (this is implemented by hierarchical design and package manipulation). For example, a spy can message home, but can't call home's update method.

All the above allow for a segregation of internal logic (hiding from other classes). This allows for anticipation of change as it protects classes that know very little from needing to be modified; for instance, a change to contacts (observer arraylist) is unlikely to affect the subject.

1.4 How the Design is intended to be Used the overall application

Each design pattern used helps reinforce the design principles in question three – they will be summarized again here. We will also describe how our application works with the design

Observer: The Observer pattern (push) is used to send updated schemes through the network. A push model was ideal since units always require the most up-to-date scheme information. Observers: Field Spy, Subjects: Home and Field.

The Observer pattern reinforces the open-closed principle. This is because new observers can be created without having to change the subject classes and vice versa as there is a subject interface. For example, one could create a new implementation of SpyObserver, without changing Field.

This also allows for greater dependency inversion (design for interfaces) as the subject classes do not rely on the implementation of subjects, rather they rely on the abstract interface. For instance, Field relies on abstract SpyObserver not concrete Spy.

Also, the network observer pattern adheres to the principle of interface segregation as, by definition, our shell-subjects only give the observers access to the minimum amount of functionality while hiding what they do not need. For instance, update is not in HomeShell, as AField doesn't need it.

Strategy:

The Strategy pattern is used to allow encryption schemes (strategies) to be interchangeable at runtime. The home base can change between schemes using a MessagingMachine type; observers are automatically updated.

This enforces open-closed since new strategies can be introduced at runtime without having to change the client (Home) or the context (Messaging Machine).

Also, dependency inversion for similar reasons to the above, since no classes depends on a concrete scheme implementation, but instead an abstraction (scheme interface)

Singleton:

The singleton pattern was especially useful in ensuring that there was only one home base. Units call getInstacnce() of home rather than Home itself.

1.4.1 Summary of Application

As part of task 4 of part 1, please reference this section if any of the design, or explanation is not clear. The explanation will work down the hierarchy:

- Unit interface ensure that all concrete classes that fall under Home Base, Field Bases, or Spies, implement a messaging system. Let unit refer to a Field Base, Spy, or the Home Base in the application explanation.
- The network uses an observer pattern to send updates. Home and Spies are strictly Subjects and Observers, respectively. Field plays the role of both a subject and observer, receiving and sending updated scheme information.
- Shell-Subjects use the full subject but are placed in a different package with observers to prevent observers from accessing full subject functionality.
- Field Bases "go dark" by unsubscribing to home, and re-register at will (but will not receive updates until they r=do so).

- Spies has attributes isdead and isreg to ensure it does not get registered when it shouldn't. The die() acts as their device – this method invokes Field to unregister them. A dead spy can't re-register.
- All Units use a private MessagingMachine which employs a strategy pattern to swap between encryption schemes. Only when home changes its scheme is the rest of the registered network notified.
- Messages are represented by an iMessage interface, implemented by a Message class.
- Subjects like Home and AField use an abstract arrayList called iContactList to store their observers.

1.5 Expected Output

```
Spy registered to Field Base.

Spy registered to Field Base.

Spy registered to Field Base.

Spy registered to Field Base.

Message sent.

Message Received: [Decrypted with AScheme(12): [Encrypted with AScheme(12):
First Message]]

Message sent.

Message Received: [Decrypted with AScheme(12): [Encrypted with AScheme(12):
Second Message]]

Message sent.

Message Received: [Decrypted with BScheme(54): [Encrypted with BScheme(54):
Third Message]]

Message sent.

Message Received: [Decrypted with BScheme(0): [Encrypted with BScheme(54):
Fourth Message]]
```

```
Message sent.

Message Received: [Decrypted with BScheme(0): [Encrypted with BScheme(0):
Fifth Message]]

Message sent.

Message sent.

Message Received: [Decrypted with BScheme(0): [Encrypted with BScheme(0):
Final Message, Part 1.]]
```

2 The Problem - Part 2

2.1 Updated UML Diagram

Please see Figure 2.

2.2 Explanation

To accomplish part two, we decided to expand the existing Scheme abstract class into a decorator pattern. None of the previous code needs to be modified since we are extending the functionality of the Scheme interface (technically an abstract class).

The existing implementations of Scheme remain unchanged (like Ascheme) as they are the concrete components of the decorator. We added a SchemeDecorator that both has a and is a Scheme, and concrete decorators like CSchemeDec and DSchemeDec, which wrap concrete schemes or even each other. As per symmetric encryption: encryption messages are decrypted in the reverse order they are encrypted, so the decorator's decrypt function passes a type iMessage into the super method, while the decorator's encrypt method encrypts the result of the super method itself.

2.3 Some notes on encryption

`[Encrypted with AScheme(12): Hello World]`

The above denotes a message, Hello World, that has been encrypted using method Ascheme with key 12. By Symmetric encryption the following would be valid as the text is being encrypted then decrypted with AScheme key 12:

`[Decrypted with AScheme(12): [Encrypted with AScheme(12): Hello World]] = Hello World`

The following would be invalid due to different Keys:

`[Decrypted with BScheme(0): [Encrypted with BScheme(54): Hello World]] = Hello World`

The following would be invalid due to different schemes:

`[Decrypted with BScheme(12): [Encrypted with AScheme(12): Hello World]] = Hello World`

2.4 Expected Output

```
Spy registered to Field Base.

Message sent.

Message Received: [Decrypted with BScheme(10): [Decrypted with DScheme(10):
[Encrypted with DScheme(10): [Encrypted with BScheme(10): Test Decorator]]]]

Message sent.

Message Received: [Decrypted with BScheme(10): [Decrypted with DScheme(10):
[Decrypted with CScheme(10):[Encrypted with CScheme(10): [Encrypted with
DScheme(10): [Encrypted with BScheme(10): One more layer...]]]]]]

Message sent.

Message Received:

[Decrypted with BScheme(10): [Decrypted with DScheme(10):[Decrypted with
CScheme(10):[Decrypted with DScheme(10):[Encrypted with DScheme(10): [Encrypted
with CScheme(10): [Encrypted with DScheme(10): [Encrypted with BScheme(10):
Now, this is EPIC!]]]]]]]]
```

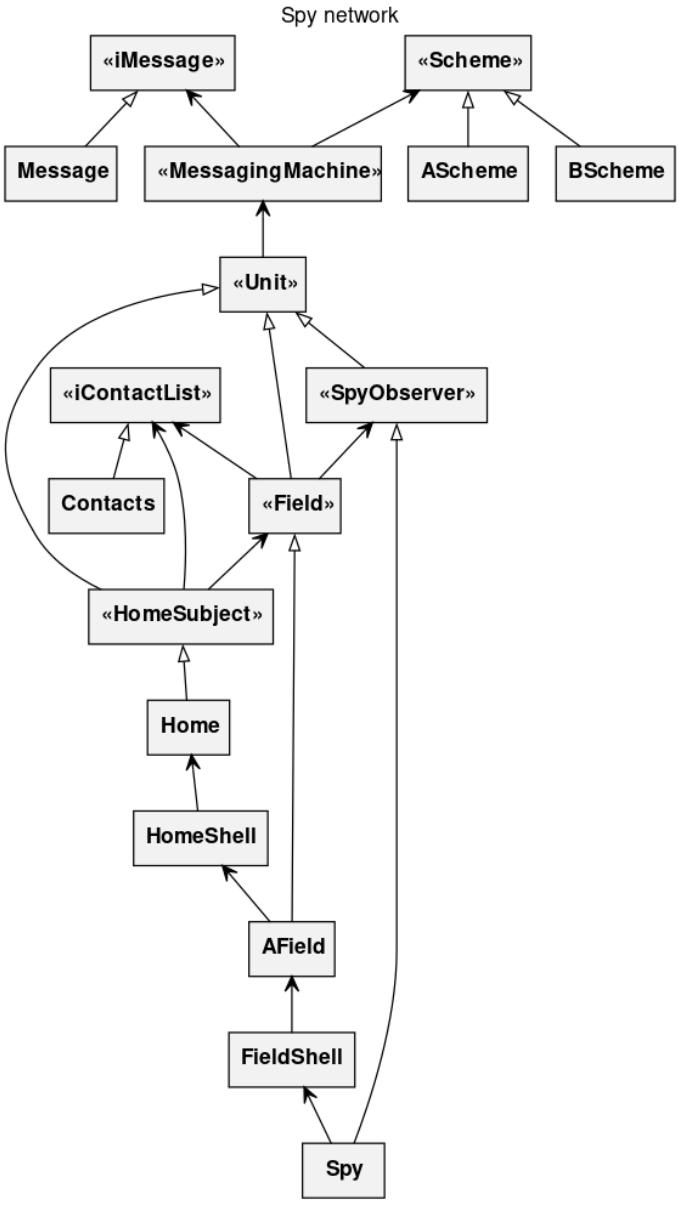

Figure 1: UML Diagram

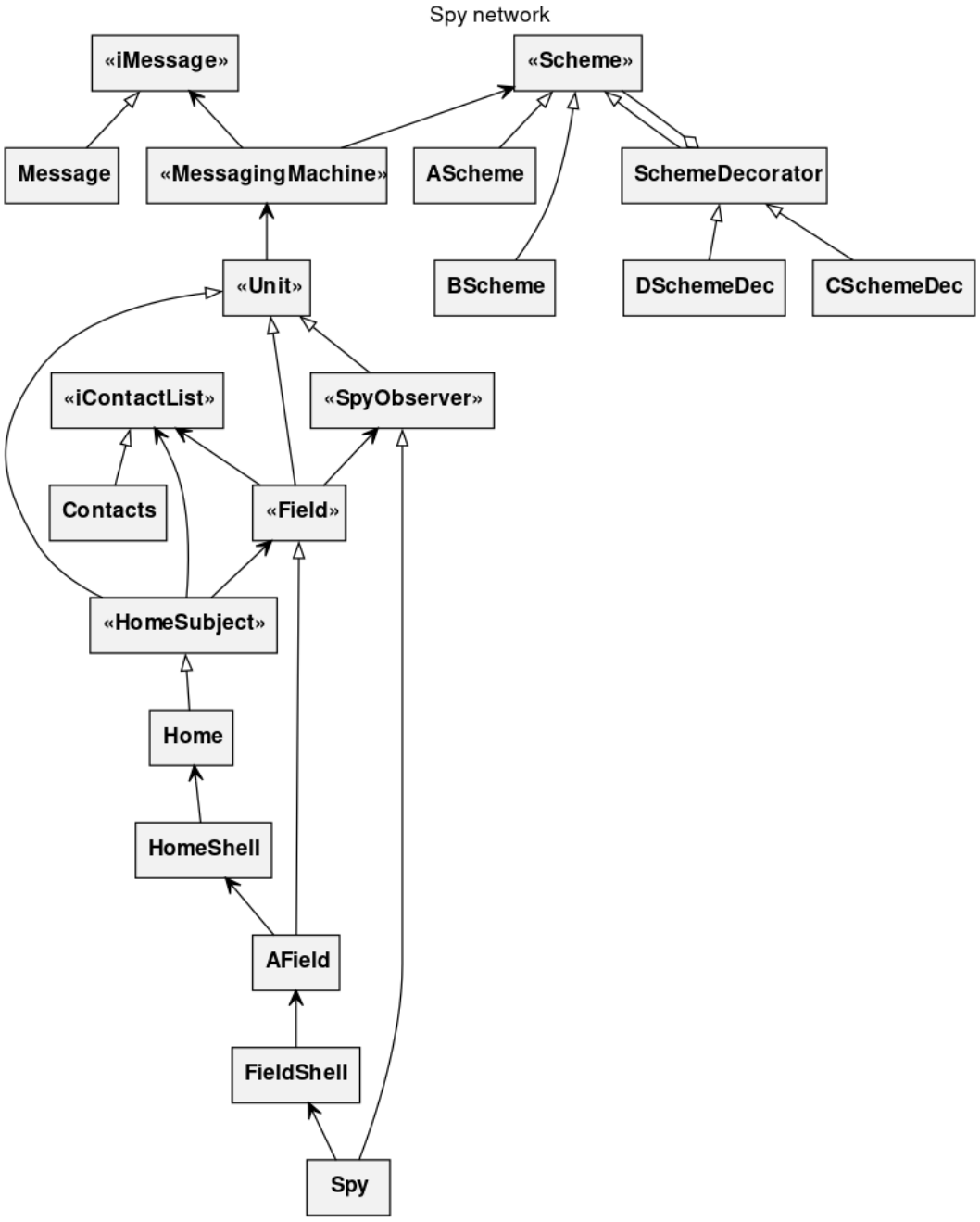

Figure 2: Updated UML Diagram