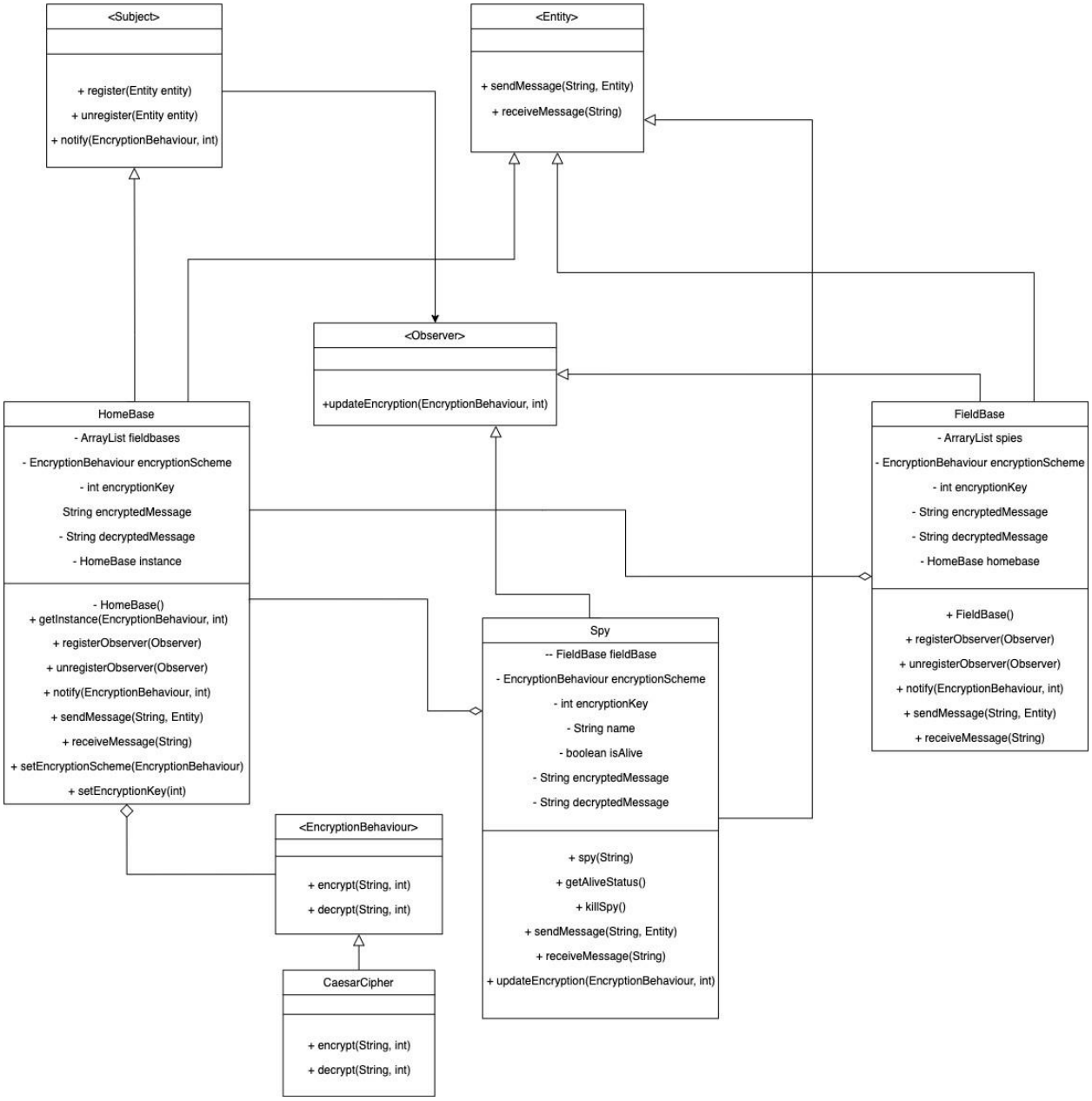
UML Class Diagram



Design Patterns Used:

We use the **Singleton design pattern** for instantiating a HomeBase object since the information provided in the first bullet point tells us that there will only be one home base. We achieve a

unique, single instance of a home base in our HomeBase class by following the singleton pattern's way of constructing things with a private constructor and a getInstance() method. The getInstance() method ensures that the home base object is only instantiated once and never again. In this method, we return a home base object if the instance variable in the class is not instantiated yet (is null). Therefore, all future attempts in instantiating a concrete home base object will be voided as we will be returning the object that was previously instantiated (since the instance in the class is not null). Now, when we create a home base object in our runner for the first time, we will be using that created object for the rest of the other code in our simulation. When one attempts to instantiate a new concrete home base, the original home base that was created is the only concrete class that will be returned (not re-instantiated, but reused).

The **Observer design pattern** is used due to the bullet point that tells us "Field bases are associated/registered with the home base, and spies are associated/registered with a single field base." We associate the relationship between association/registration with the observer to subject relationship in the observer pattern. Furthermore, we are told that the home base can change the encryption scheme and key and send the updated information to the field base; this is exactly like a subject using a "notify" method to allow the observers to "update" with the new information the subject provides. We are further told that the home base sends the updated encryption scheme and/or key to the field bases and the field bases send the updated encryption scheme and/or key to the spies. As a consequence from this information, we use the observer pattern twice, where each observer (field base and spies) implement the observer interface and each subject (home base and field base) implement the subject interface. This is how we achieve the observer pattern with field bases (observer) observing home bases (subject) and the spies (observer) observing the field bases (subject). Note that in the grand scheme of the UML diagram, the field base is both an observer and a subject but in two different instances of the same pattern.
We use the **Strategy design pattern** to create a system of achieving an encryption with the option of using multiple different encryption algorithms/methods, such as the given Caesar Cipher example in the assignment. We use the strategy design pattern for this because the fifth bullet point tells us "The scheme and key are both determined by the home base. When the home base changes the scheme or the key [...]". This implies that the homebase can change the current

encryption scheme to a different encryption scheme at will. The client, homebase, has an EncryptionBehaviour, where EncryptionBehaviour is an interface in which encryption behaviour is encapsulated. The EncryptionBehaviour has encrypt and decrypt methods which are implemented by different concrete "strategy" classes. For example, CaesarCipher would be a concrete class that implements EncryptionBehaviour's encrypt and decrypt functions. A homebase then has methods to set an encryption algorithm and perform the encrypt and decrypt methods of the encryption algorithm (since a homebase has an EncryptionBehaviour). This pattern is useful, since we can then create multiple independent concrete symmetric cipher classes that implement EncryptionBehaviour and then have a homebase simply set its EncryptionBehaviour algorithm at runtime to be any one of those.

Design Principles Enforced:

The **Encapsulation** design principle refers to the bundling of data with the methods that operate on that data or the restricting of direct access to some of the components of an object. Throughout this project, encapsulation is enforced through both our usages of the strategy design pattern and observer design pattern.

We use the strategy pattern to encapsulate encryption behaviour into an interface called EncryptionBehaviour. This EncryptionBehaviour interface has methods that operate on data (encrypt and decrypt functions encrypt and decrypt messages given a key). The EncryptionBehaviour interface is implemented by concrete symmetric cipher classes such as CaesarCipher. Encapsulation is effectively enforced in this manner, as a family of symmetric ciphers are encapsulated into their own concrete classes.

We use the observer pattern to restrict access and encapsulate class fields from other given classes. The objects (like the HomeBase object and FieldBase objects) take care of notifying its observer classes and can only notify them when a new state is available. They do not have the power to physically change the state of the observers because their data is encapsulated away. Rather, the observers will call update() themselves and change their own states that way.

The **Single Responsibility** design principle refers to the idea that each class should only have one responsibility to carry out. Concrete symmetric cipher classes that implement EncryptionBehaviour encrypt and decrypt methods in our project abide by this principle as they are each independent of each other and only have a single responsibility to carry out, that is, their own implementations of the encrypt and decrypt methods. The FieldBase class will abide by this principle in the sense that the FieldBase will not implement encrypt and decrypt, but instead focus on its relationships with its homebase and spies, i.e., sending/receiving messages to/from its homebase and spies, registering/unregistering its spies, and making sure its spies encryption scheme and key is the most recent. This behaviour is similar for the Spy class and the HomeBase. (Note that the HomeBase class does not exactly follow the single responsibility principle since it is implemented using the Singleton design pattern, meaning that HomeBase has two responsibilities: implementing its primary behaviour, and making sure that only one instance of it can be created).

The **Open-Closed** design principle refers to the idea that objects or entities should be open for extension but closed for modification. The way that we handle encryption algorithms in this project abides by this principle, since we do not modify the context (which is the homebase) directly if we want to add an encryption algorithm, but instead we extend our codebase by adding another concrete class that implements EncryptionBehaviour encrypt and decrypt methods. Moreover, if for example we wanted to add more observers to the FieldBase subject we would simply make more concrete classes that extend the observer interface. We do not go into any classes and change any code to make this new addition, but rather call functions that are previously defined, like registerObserver(), to handle the addition. Since we are extending interfaces (such as the observer interface), and not having to physically modify any class code in the interfaces (like Subject) or other concrete classes, we are adhering to the open-closed design principle.

The **Dependency-Inversion** principle refers to the idea that high-level modules should not import anything from low-level modules and instead both should depend on abstractions. Since our concrete classes implement interfaces, we are extending an abstraction of the actual

observer/subject. This way, a parent concrete subject, like HomeBase, does not rely on its children classes.

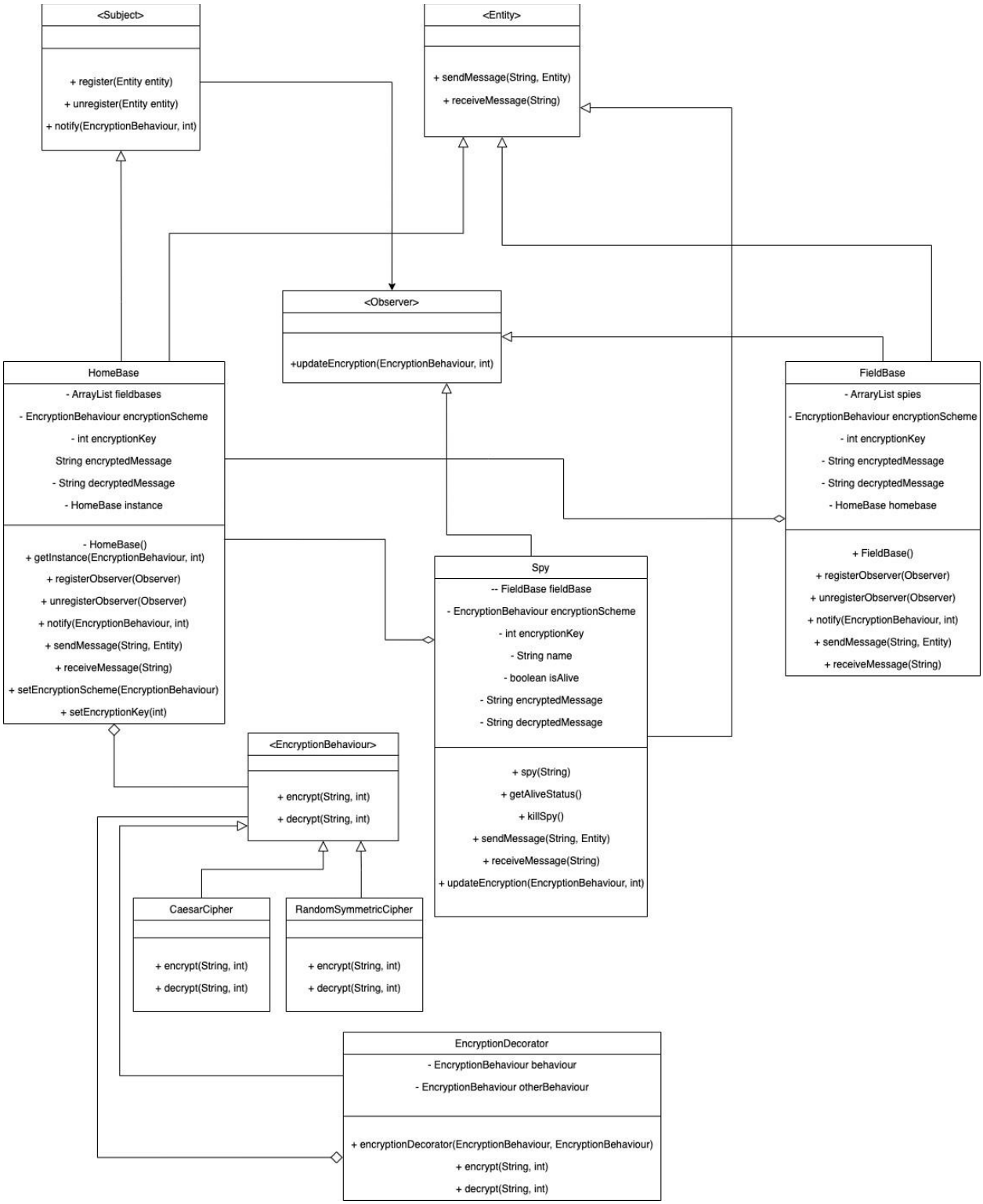
Explanation of Design and Use:

In this codebase we have an Entity interface which contains methods sendMessage() and receiveMessage(). The entities as described in the assignment are home bases, field bases, and spies. All of these entities implement Entity in order to send messages to each other and receive messages from each other. Furthermore, we have an Observer interface which contains the updateEncryption() method and we have a Subject interface which contains the registerObserver(), unregisterObserver(), and notify() methods. The observers will update themselves with the updated state once the Subject notifies them to (in accordance with the observer design pattern). In this project, a HomeBase is an Entity and a Subject, a FieldBase is an Entity, a Subject, and an Observer (it observes its HomeBase), and a Spy is an Entity, and an Observer (it observes its FieldBase). We also have an EncryptionBehaviour interface which has methods encrypt() and decrypt(). These methods are implemented by concrete classes that implement symmetric encryption encrypt() and decrypt() methods (for example a CaesarCipher class).

To use the code, we would begin by first creating an EncryptionBehaviour object that would be one of our concrete symmetric cipher classes that implement EncryptionBehaviour such as CaesarCipher for example. We would then create a unique instance of HomeBase using its getInstance method (due to it using Singleton design pattern), we would pass the EncryptionBehaviour we are using and the encryption key as arguments. We would then create FieldBase objects and register them to HomeBase and create Spy objects with names and register them to FieldBases that we want to register them to. Then, we can send messages between the three entities using the sendMessage method on them and passing a message string and recipient entity as arguments. Whenever we change the EncryptionBehaviour of the HomeBase or the encryption key, it will automatically notify its field bases, and those field bases will

automatically notify their spies. A field base has a goDark() method which when called, it will become unregistered from its HomeBase. A spy has a method killSpy(), which when called, will set its isAlive status to false, and become unregistered from its FieldBase. Once this happens, the FieldBase will remove the spy from its aliveSpies array and add it to its deadSpies array. A spy that has been put into the deadSpies array can never be registered to the FieldBase again and will never again be put into the aliveSpies array.

UML Class Diagram



Design Patterns Used

In order to achieve a composition of encryptions that our entities can utilize, we use the decorator design pattern to add functionality to existing encryptions. Our encryptionDecorator class “wraps” each encryption (such as the Caesar Cipher) with another encryption (like RandomSymmetricEncryption). This way, our RandomSymmetricEncryption will be composed with a Caesar Cipher encryption.

Design Principles Enforced:

The addition of the decorator design pattern into our general UML diagram exhibits all of the previously mentioned design principles.

We have continued our encapsulation of classes by making sure no fields from other classes can be accessed without a method. We are also following information hiding with our implementation as no class can directly modify another. Our encryption schemes are not modified, but instead wrapped in a new functionality, but on-top of the other one. The implementation of the old encryption scheme is not directly known, but it is used with the assumption that it works (which is exactly how the wrapping of encryptions works in this case).

The decorator design pattern holds the open-closed principle since everytime we would like to add functionality to an object (our encryption), we do not need to modify the encryption’s code (like CeasarCipher). Instead, we simply wrap our CeasarCipher with another cipher using our decorator. This fact is further proven since we can dynamically add functionality to our objects at runtime. This wouldn't be possible if we had to go in and physically modify code because otherwise we wouldn't be adding functionality at runtime.
Our decorator design pattern continues to hold the single-responsibility principle due to the fact that the decorators have one responsibility; to wrap each concrete component (our encryptions) with new functionality (a different encryption).

Explanation of Design and Use

We added to the codebase an EncryptionDecorator class. This EncryptionDecorator class implements EncryptionBehavour encrypt() and decrypt() methods and does so by taking in two EncryptionBehaviours. The encrypt() method of the EncryptionDecorator class is implemented by composing the encryption method from the first EncryptionBehaviour and the encryption method from the second EncryptionBehaviour respectively.