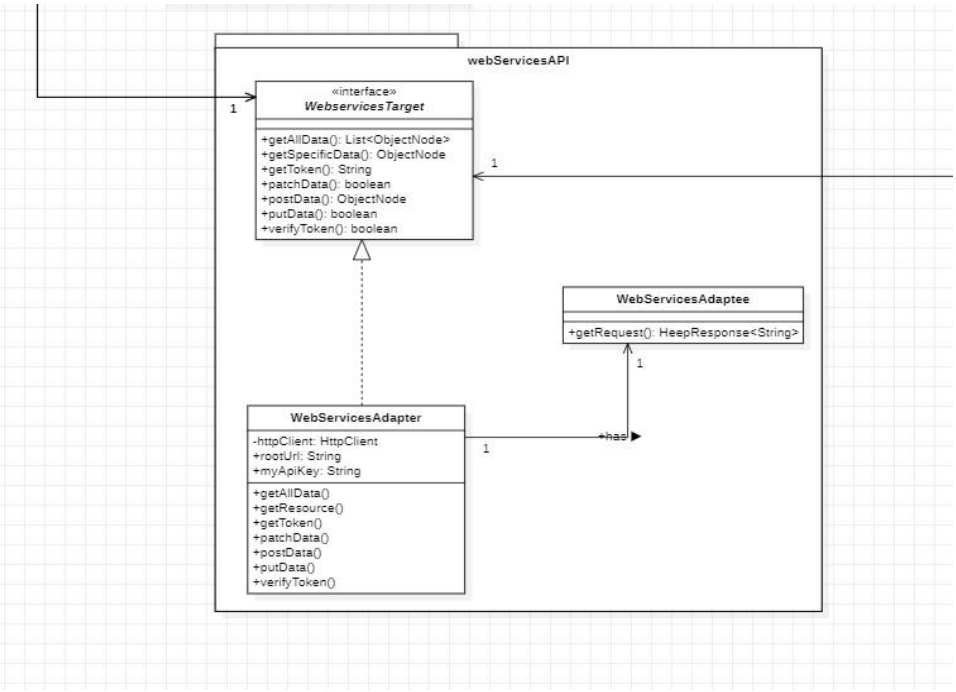
The Adapter Pattern

In our system, we need to integrate the Http Class (3rd party). We can't use this class directly because it has an incompatible interface. So, we implement the **Adapter Pattern** to create an adapter. The **adapte**r can convert data from different formats and convert the interface of a class into another interface the clients expect to work together with other classes.

In our code, the WebServicesAdapter (Adapter) class can interact with both WebServicesTarget (client interface) and WebServicesAdaptee (HTTP class). It encapsulates the HTTP object while implementing the client interface. The adapter accepts calls made by the client through the interface and converts them to calls that apply to the encapsulated object.



Using the **Adapter Pattern**, the pros are:

1. **Single Responsibility Principle (SRP)**. You can separate the interface or data conversion code from the primary business logic of the program.

2. **Open Closed Principle (OCP)**. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface.

The con is that the overall complexity of the code increases because we need to introduce a set of new interfaces and classes.

The operations(GET, PUT, POST, PATH, DELETE) are very clear in our code, it is impossible that we will introduce too many new classes to increase the complexity of the code. So we decided to use the **Adapter Pattern** here.

The Singleton Pattern

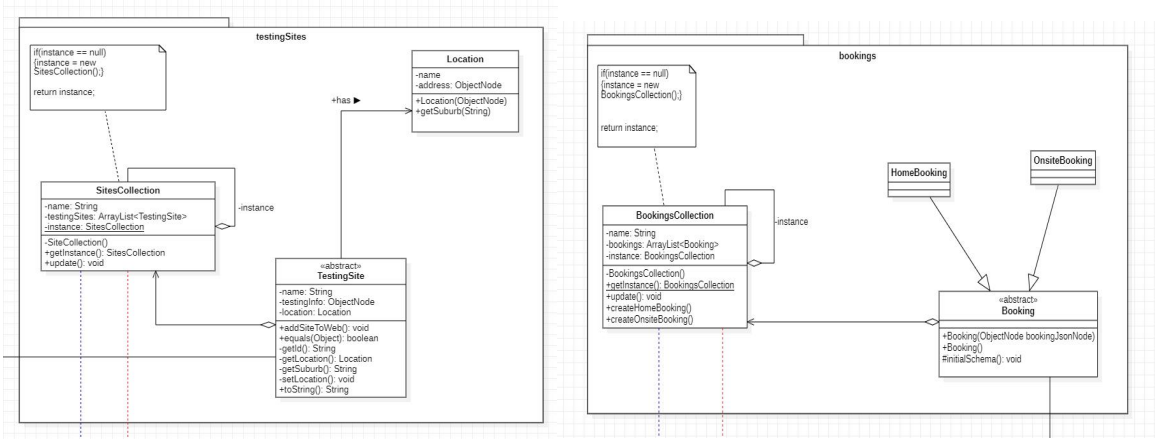
We use the **Singleton Pattern** for SitesCollection class, BookingsCollection and UsersCollection classes. In terms of the reasons for using this pattern:

The overall idea of our code is like MVC, which is to fetch data from the remote and store it locally, so that users don't have to send HTTP requests multiple times. In doing so, we can reduce the network load, upload all updates to the remote,and update the relevant data to maintain consistency(it is

necessary to be consistent because the system shares the same data). This is also why the three collections use the **Singleton Pattern**.

SitesCollection class, BookingsCollection and UsersCollection classes in our program are database objects shared by different parts of the program. So, they should have just a single instance available to all clients. And what's more, we can provide a global access point to the instance. Normally, we use the global variables to store some essential objects. While they're handy, they are also very unsafe since any code can potentially overwrite the contents of those variables and crash the app. The **Singleton pattern** not only lets us access some objects from anywhere in the program, but also protects that instance from being overwritten by other code.

Here, SitesCollection class, BookingsCollection and UsersCollection classes only have a single instance available. We can't create objects of the SitesCollection class, BookingsCollection and UsersCollection classes except for the creation method(getInstance()). This method either creates a new object or returns an existing one if it has already been created. (For reference, we put the pictures about SitesCollection and BookingCollection classes)



Using this pattern, the pros are:

1. We can be sure that SitesCollection, BookingsCollection and UsersCollection classes have only a single instance

2. We can gain a global access point to the instance of SitesCollection, BookingsCollection and UsersCollection classes.

3. The SitesCollection, BookingsCollection and UsersCollection objects are initialized only when they're requested for the first time.

But the cons about this pattern are:

1. The **Singleton pattern** can mask bad design, for instance, when the components of the program know too much about each other.

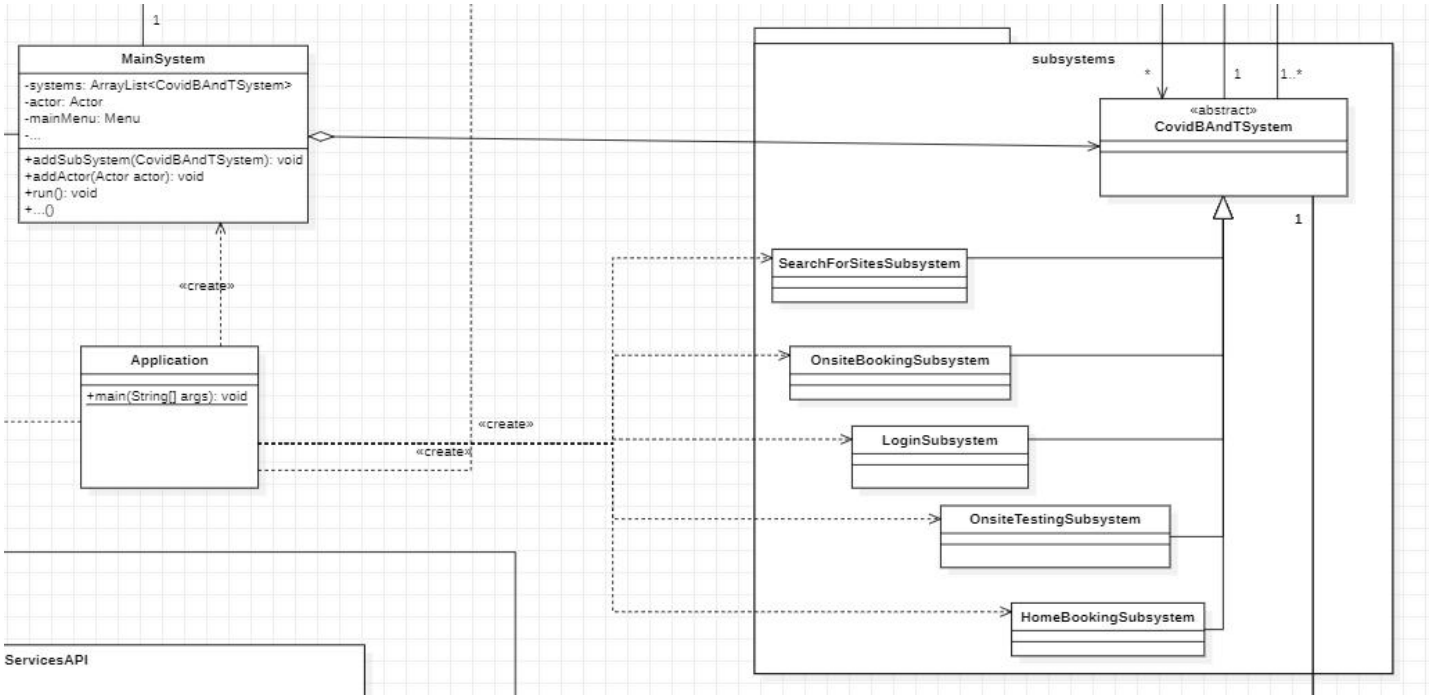
2. It may be difficult to unit test the client code of the singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, we have to think of a creative way to mock the singleton.

To sum up, although the **Singleton Pattern** has certain limitations. However, the SitesCollection, BookingsCollection and UsersCollection data are shared across the system and must be consistent, so we must create only one instance for them, which only the **Singleton Pattern** can achieve.

The Facade Pattern

We use the **Facade pattern** to manage the multiple subsystems.

We must make our code work with a broad set of subsystem objects. Ordinarily, we need to initialize all those subsystem objects and keep track of dependencies, execute methods in the correct order and so on. As subsystems get more complex over time, the amount of configuration and boilerplate code it demands from a client grows ever larger. It will be very hard to comprehend and maintain. So, we implement the **Façade Pattern** to fix this problem by providing a shortcut to the most-used features of these subsystems which fit most client requirements.



Here, the MainSystem class is a **façade** which provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts. And the Complex Subsystem consists of various subsystem objects(SearchForSitesSubsystem, OnsiteBookingSubsystem, LoginSubsystem, OnsiteTestingSubsystem, HomeBookingSubsystem). These subsystems classes aren't aware of the **façade**'s existence. They operate within the system and work with each other directly. And client will use the **facade** instead of calling the subsystem objects directly. Here we just only need deploy MainSystem in Application.

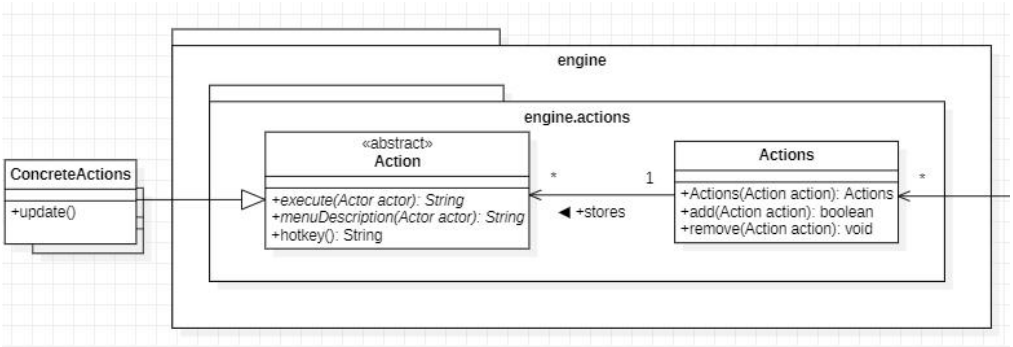
Instead of making the code work with dozens of subsystem classes directly, the **façade** class which encapsulates that functionality and hides it from the rest of the code. The structure also helps us to minimize the effort of upgrading to future versions of the framework or replacing it with another one. The only thing we need to change in our code would be the implementation of the **façade**'s method.(and add new subsystem to the MainSystem)

The good thing about this pattern is that we can isolate our code from the complexity of a subsystem and the **façade** (Application class) creates a balance between **Common Closure Principle (CCP)** and **Common Reuse Principle (CRP)**. The issue is that the **façade** (Application class) can become a god object coupled to all classes of an app.

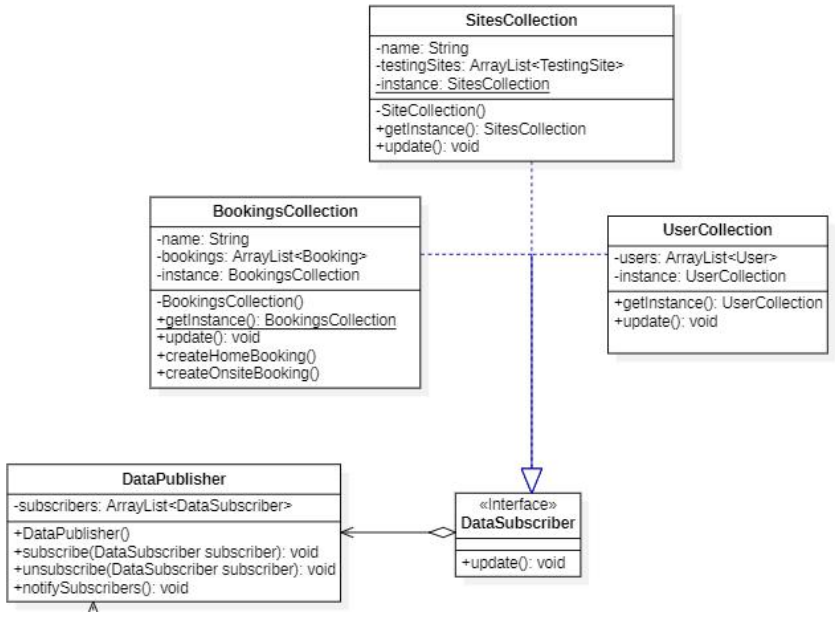
The Observer Pattern

In our code, the ConcreteActions classes and (SitesCollection, BookingsCollection, UsersCollection classes) want to track changes to the Actions class and DataPublisher Class. And the Actions class and DataPublisher are going to notify the ConcreteActions classes and (SitesCollection, BookingsCollection, UsersCollection classes) about changes to its state. So the ConcreteActions classes and (SitesCollection, BookingsCollection, UsersCollection classes) are like subscribers. And the Actions class and DataPublisher Class are like publishers. So we need to implement the **Observer Pattern** to define a subscription mechanism to notify multiple subscriber objects about any events that happen to the publisher object they're observing.

In our code, we use the **Observer Pattern** in two places, as shown below,



The Actions class issues events of interest to other ConcreteActions classes. These events occur when the Actions class changes its state or executes some behaviors. The **Observer Pattern** suggests that we add a subscription mechanism to the Actions class so individual objects can subscribe to(add(Action action)) or unsubscribe(remove(Action action)) from a stream of events from the Actions class. And the ConcreteActions classes will perform some actions in response to notification issued by the Actions class. All of these classes must implement the same abstract Action class so the Actions class isn't coupled to concrete actions classes.



Same logic here, the DataPublisher is the publisher to notify its concrete subscribers(SitesCollections, BookingsCollection, UsersCollections) everytime the state changes or behavior is executed. The DataPublisher class also contains a subscription infrastructure that notifies the subscribers(notifySubscribers()) and lets new subscribers join(subscribe(DataSubscriber subscriber)) or current subscribers leave the list(unSubscribe(DataSubscriber subscriber)). And the DataSubscriber interface declares the notification interface. It contains a single update method. The method will let the

DataPublisher class pass some event details along with the update. These subscriber classes implement the same interface so the DataPublisher isn't coupled to concrete subscriber classes

The pros about using the **Observer Pattern**:

1**.Open Closed Principle (OCP)**. We can introduce new subscriber classes(ConcreteAction and DataCollection) without having to change the code of the publisher(Actions and DataPublisher)

2. We can reuse these subscriber(ConcreteAction and DataCollection) classes independently of each other.

2. We can establish relations between these subscriber objects(ConcreteAction and DataCollection) at runtime.

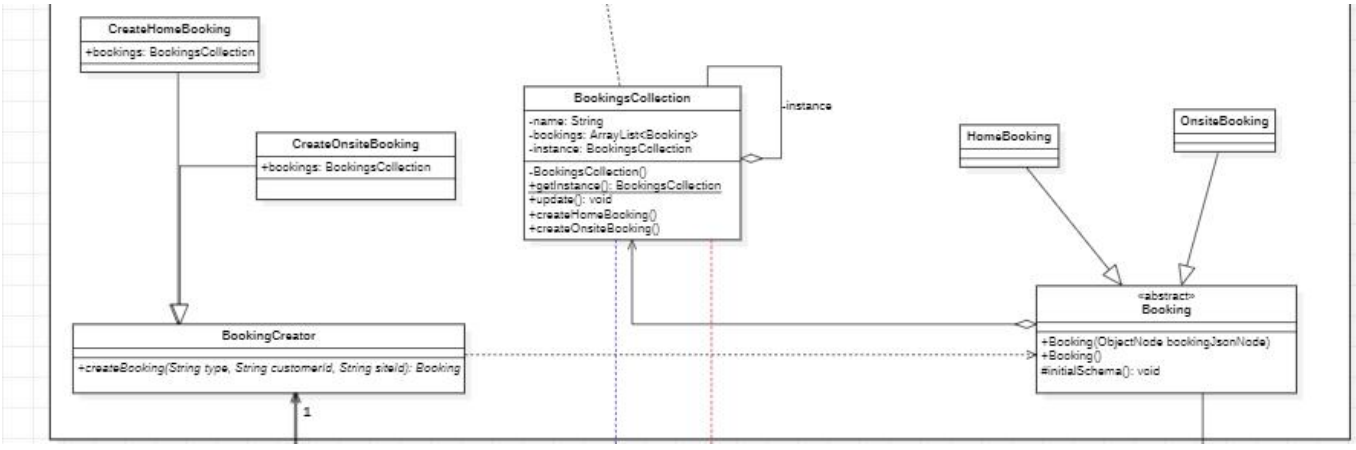
3. Changes to subscriber classes(ConcreteAction and DataCollection) will not affect the other.

But the con about this pattern is that the action classes are notified in random order.

It doesn't matter for our code about the notification order. So it will be great to use the **Observer Pattern** in these two places.

The Factory Pattern

We don't know beforehand the exact types and dependencies of the Booking object, and we want to keep the convenience to introduce the new type of bookings. So, we use the **Factory** Method here. On the one hand, the **Factory Pattern** can separate Booking construct code from the code that uses the Booking, therefore it is easier to extend the Booking construction code independently from the rest of the code. On the other hand, we put the code that constructs Booking across the framework into a single **Factory** method and let anyone override this method in addition to inheriting the booking itself.



According to the picture above, the HomeBooking class and OnsiteBooking class are different implementations of the abstract Booking class. The BookingCreator class declared the factory method that returns new booking objects. Instead of being a creator, the BookingCreator class includes the core business logic related to bookings. The **Factory** method helps to decouple this logic from the concrete booking classes(HomeBooking and OnsiteBooking). And the CreateHomeBooking and CreateOnsiteBooking classes override the base factory method so it returns a different type of the booking.

The benefits of using the **Factory pattern** are:

1.We can avoid tight coupling between the BookingCreator and the concrete bookings.

2.**Single Responsibility Principle(SRP)**. We can move the booking creation code into one place in the program to make the code easier to support

3.**Open Closed Principle(OCP)**. We can introduce new types of bookings into the program without breaking existing client code.

The issue of using the **Factory** method is that the code may become more complicated since we need to introduce a lot of new subclasses to implement it. But in our code about booking, we are sure that the situation that we need to introduce lots of new booking types won't happen. So we implement the **Factory** method here. But for other places, we don't decided to use this pattern because of the defect. For example, creating different type of tests is similar in code, we choose another better and simple way to deal with it.

Reference:

https://refactoring.guru/design-patterns

FIT3077 Lecture Slides