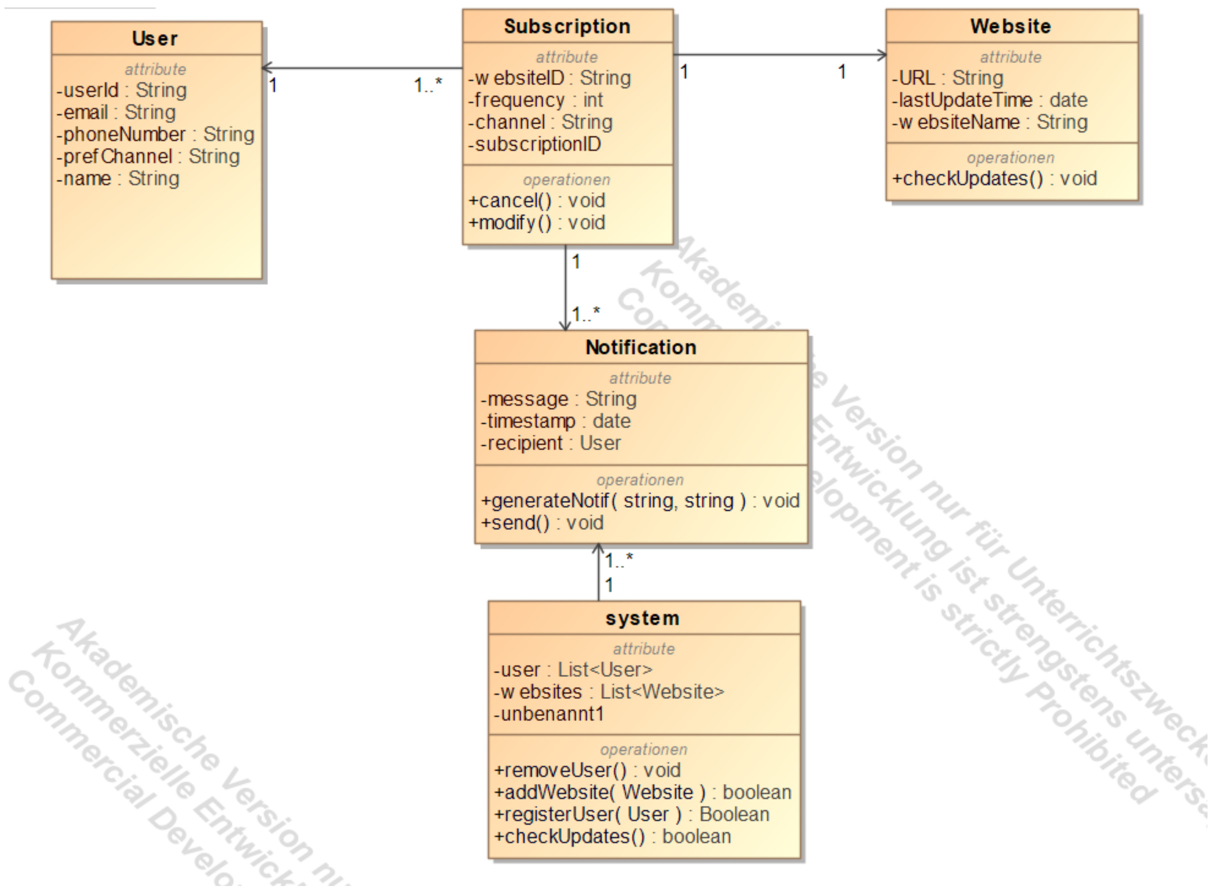
1. Implement the system from Exercise 3 task 1 in Java.



2. Calculate the metrics afferent coupling, efferent coupling and instability for each class.

Understand what each of these metrics represents:

- **Afferent Coupling (Ca)**: This measures how many other classes depend on a given class. It's a measure of how many classes would be affected if the given class were to change
- **Efferent Coupling (Ce)**: This measures how many classes a given class depends on. It's a measure of how many classes could affect the given class if they were to change .
- **Instability (I)**: This is the ratio of efferent coupling (Ce) to total coupling (Ce + Ca). It's a measure of the class's resilience to change. A value of 0 indicates a completely stable class, and a value of 1 indicates a completely unstable class.

Analyzing Dependencies

1. **Class `User`**
    - **Ca (Afferent Coupling):**
        - `System`: Depends on `User` (registerUser, removeUser, printUsers)
        - Total Ca = 1
    - **Ce (Efferent Coupling):**
        - None
        - Total Ce = 0
    - **Instability (I):**
        -

I = Ce/Ca + Ce = 0/1+ 0 = 0

2. **Class `Website`**
    - **Ca (Afferent Coupling):**
        - `System`: Depends on `Website` (addWebsite, checkUpdates)
        - Total Ca = 1
    - **Ce (Efferent Coupling):**
        - None
        - Total Ce = 0
    - **Instability (I):**
        -

I = Ce/Ca + Ce = 0/1+ 0 = 0

3. **Class `Subscription`**
    - **Ca (Afferent Coupling):**
        - None
        - Total Ca = 0
    - **Ce (Efferent Coupling):**
        - None
        - Total Ce = 0
    - **Instability (I):**
        -

I = Ce/Ca + Ce = 0/0 + 0 = 0

4. **Class `Notification`**
    - **Ca (Afferent Coupling):**
        - None
        - Total Ca = 0
    - **Ce (Efferent Coupling):**
        - None

        - Total Ce = 0
    - **Instability (I):** I = Ce/Ca + Ce = 0/0 + 0 = 0
5. **Class `System`**
    - **Ca (Afferent Coupling):**
        - None (Assuming `System` is the top-level controller)
        - Total Ca = 0
    - **Ce (Efferent Coupling):**
        - `User`: Depends on `User` (registerUser, removeUser, printUsers)
        - `Website`: Depends on `Website` (addWebsite, checkUpdates)
        - Total Ce = 2
    - **Instability (I):** I = Ce/Ca + Ce = 2/0 + 2 = 1

`Subscription` and `Notification` have undefined instability metrics because they have no dependencies at all (Ca + Ce = 0), so we typically treat these as 0 in this context.

3. Suggest a package structure for your implementation.

com.myproject.users: This package would contain the `User` class and any other classes related to users.
com.myproject.websites: This package would contain the `Website` class and any other classes related to websites.
com.myproject.system: This package would contain the `System` class, which seems to be a top-level controller in your application.
com.myproject.subscriptions: This package would contain the `Subscription` class and any other classes related to subscriptions.
com.myproject.notifications: This package would contain the `Notification` class and any other classes related to notifications.

4. Commit your software to a new github.com repository.

5. Name options to reduce coupling between your packages.

- **Encapsulation**: Encapsulation can help reduce content coupling. By hiding the internal details of a package and exposing only what's necessary through its interface, you can prevent other packages from depending on those internal details.
- **Package by Feature**
  Organize packages by feature rather than by layer. Each package contains all classes related to a specific feature, reducing cross-package dependencies.

- **Law of Demeter**: This principle, also known as the principle of least knowledge, suggests that an object should only communicate with its immediate neighbors and should not have knowledge about the inner workings of other objects. This can help reduce structural coupling.
- **Use Interfaces and Abstract Classes**
  Define common interfaces or abstract classes that your classes can implement or extend. This allows your system to depend on abstractions rather than concrete implementations, reducing direct dependencies.
-