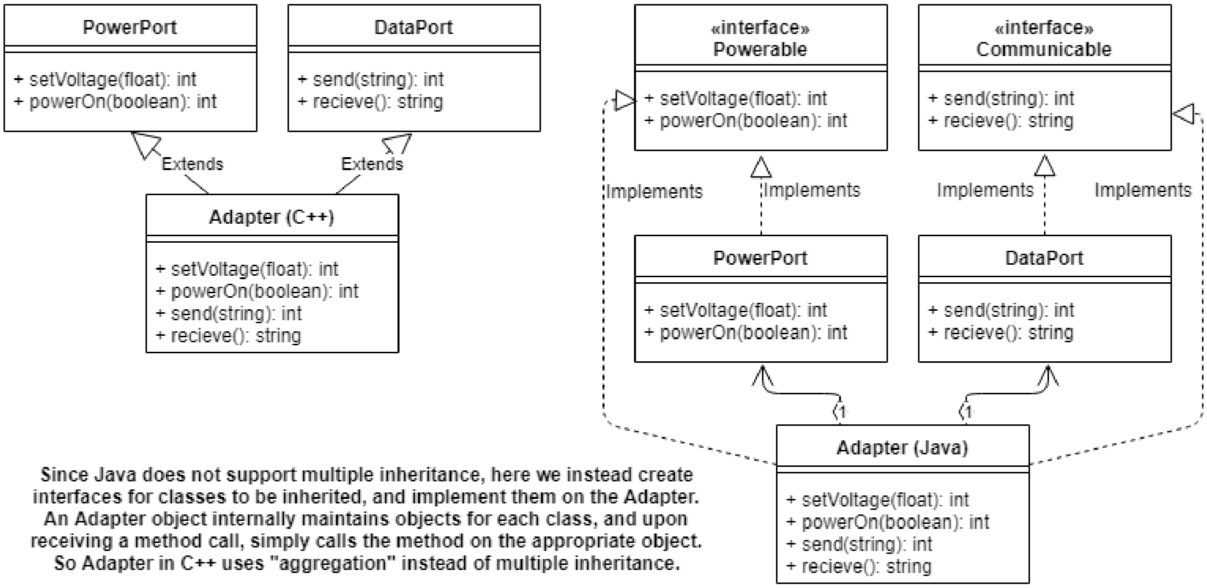
1. Does pair programming reduce the need for refactoring? Support your argument with appropriate evidence.

2. In “No Silver Bullet --Essence and Accidents of Software Engineering” by Fred Brooks, the author claims “there is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity”. List arguments to support/contradict his claim.

3. The C++ implementation of class Adapter specifies the use of multiple inheritance.Using a diagram show how this could be implemented in Java? Be specific about the Java features that you are using.



Since Java does not support multiple inheritance, here we instead create interfaces for classes to be inherited, and implement them on the Adapter. An Adapter object internally maintains objects for each class, and upon receiving a method call, simply calls the method on the appropriate object. So Adapter in C++ uses "aggregation" instead of multiple inheritance.

4. One method of classifying iterators does so along two dimensions.The first indicates the location of control of the iteration (internal to the iterator or external client control), and the second which indicates the location of the definition of the iteration logic (embedded as part of the collection objects or in objects separate from the collections).Considering each dimension separately, what are the positive and/or negative aspects of iterators of each type?

**Location of control of Iteration:**

- **Internal to the iterator:** This means that the iterator can only iterate over elements of a collection in a linear fashion only. This makes it possible to define a common “Iterable” interface for various datatypes and even “Generator functions”. All algorithms can use this common Iterable interface, but algorithms which need “client control” will not be able to make use of iterators.

- **External client control:** This means that the iterator can be controlled by the client in possible multiple ways, such as moving to the “previous” element, or moving in “steps”. Since not all collections might be able to support such special functions, there would need to be multiple “iterable” interfaces instead for each type of client control offered. In such a case however, algorithms which need “client control” will be able to use the specialized Iterable they need.

**Location of the definition of the iteration logic:**

- **Embedded as part of collection:** In this case each datatype has the freedom to define its own custom iteration logic, and hence various type of collections will behave as iterables. The client need not be aware of the logic used for any type of collection.

- **In objects separate from the collection:** Defining common iteration logic for similar data structures would improve code reuse, but this would come at the cost of lack of generality since the client would need to be aware of which algorithm to use got which type of collection.

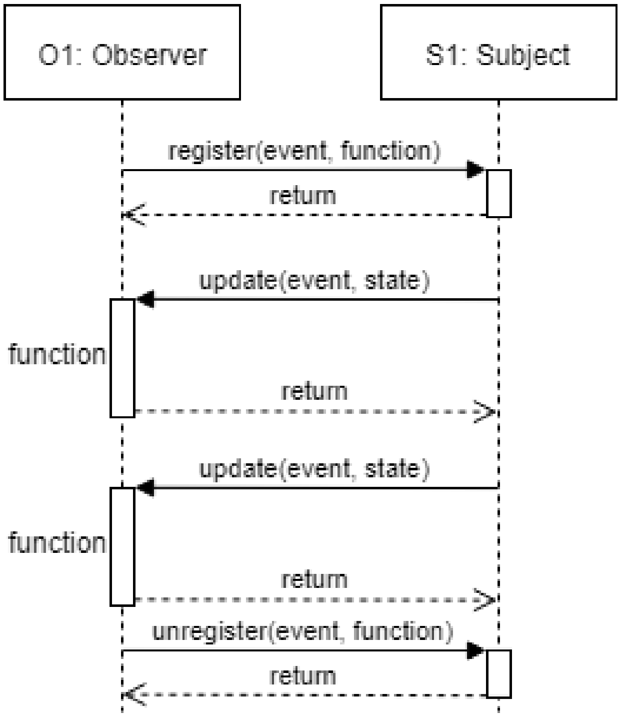
5. What language constructs would you use to give iterators privileged access in Java and in C++? How will your answer depend on the classifications for iterators given above?

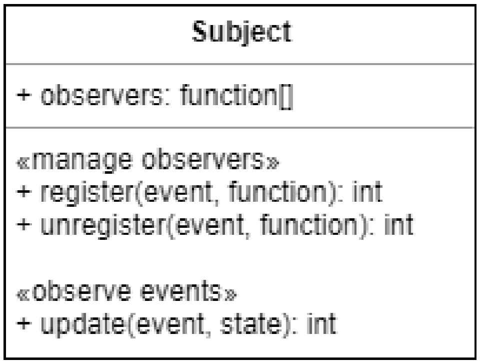
This will depend upon location of the definition of the iteration logic:

- **Embedded as part of collection:** In this case each datatype can be defined to have an “iterator()” function which returns appropriate object with Iterable interface. The desired “access specifier” can be specified on this function in both C++/Java.

- **In objects separate from the collection:** Each overload of iteration logic can specify the desired “access specifier” based on the datatype(s) it operates on in both C++/Java.

6. Draw sequence diagrams for the registration and update cycles of an Observer pattern implemented using appropriate Java classes. Be sure to label the object with the Java class and its role in the Observer pattern using Stereotypes.




Subject
+ observers: function[]
«manage observers»
+ register(event, function): int
+ unregister(event, function): int
«observe events»
+ update(event, state): int

7. You have seen that it is very common to use the Observer pattern within GUI frameworks, such as Java's Swing framework. Why do you need to be concerned about how long it takes to process a notification by a GUI element? Should this be the application designer's concern or be dealt with by the framework itself? How can the concern be eliminated?

8. Argue both for and against including the functions to handle the Composite's children in the Component interface. What are the implications for implementation of the pattern and on the intention of the pattern?