Observer Pattern

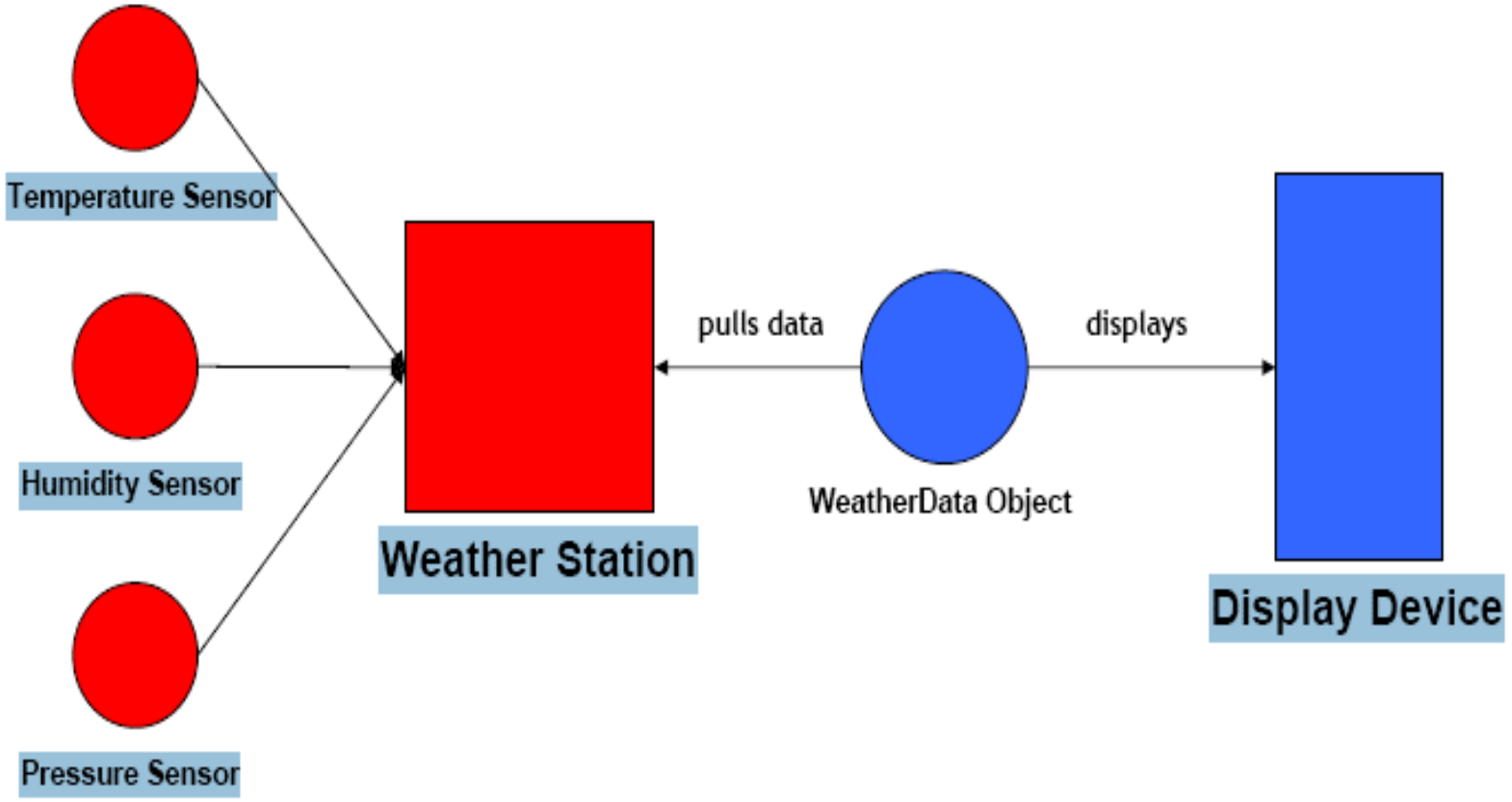
Pretest : create class diagram

Weather Monitoring Station Application

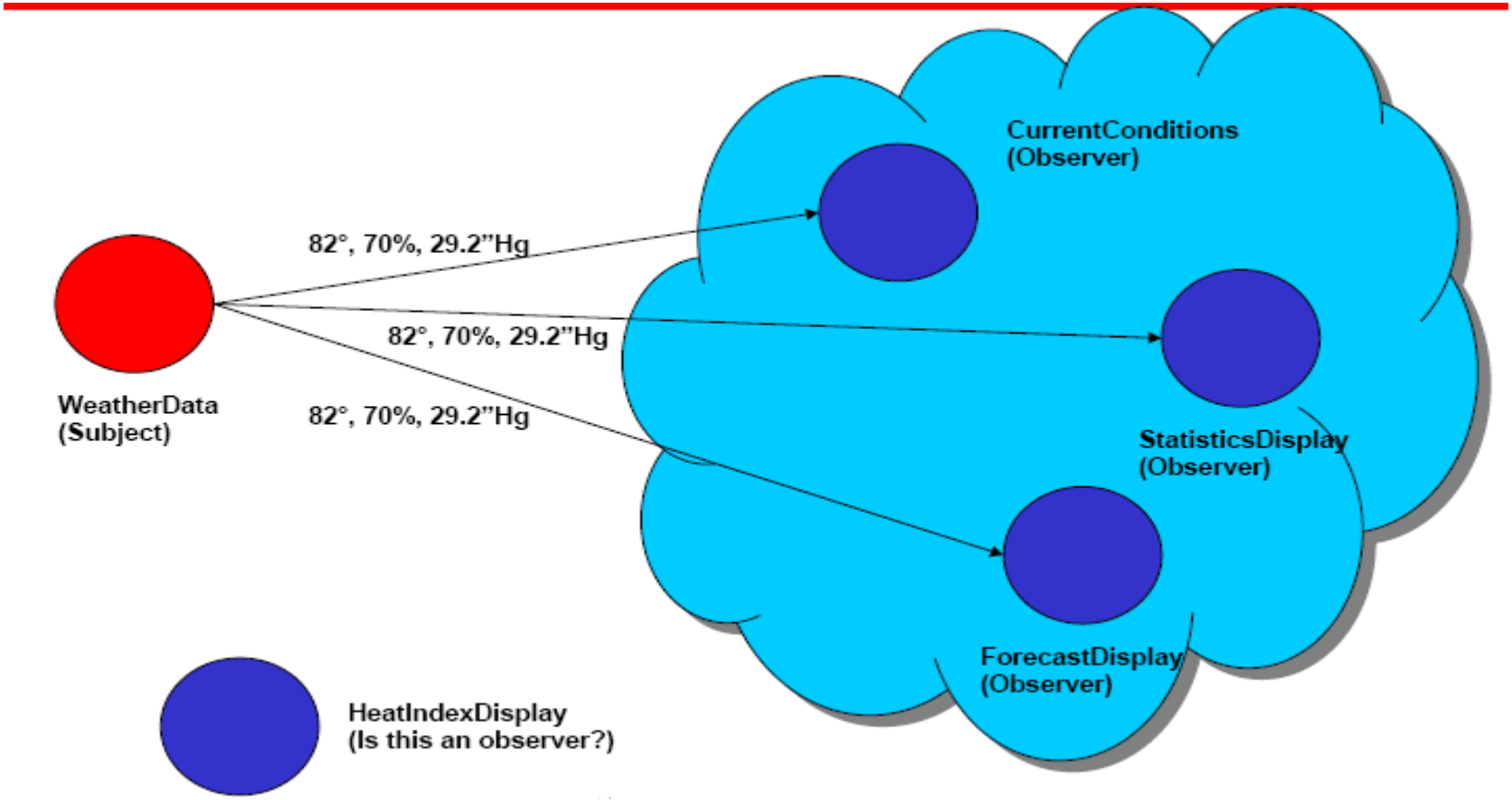
- **Objective:**

– **Design an internet-based Weather Monitoring Station application that pulls weather-related data from a weather station and displays it onto a device–Temperature, humidity, and barometric pressure data is sent to the weather station**

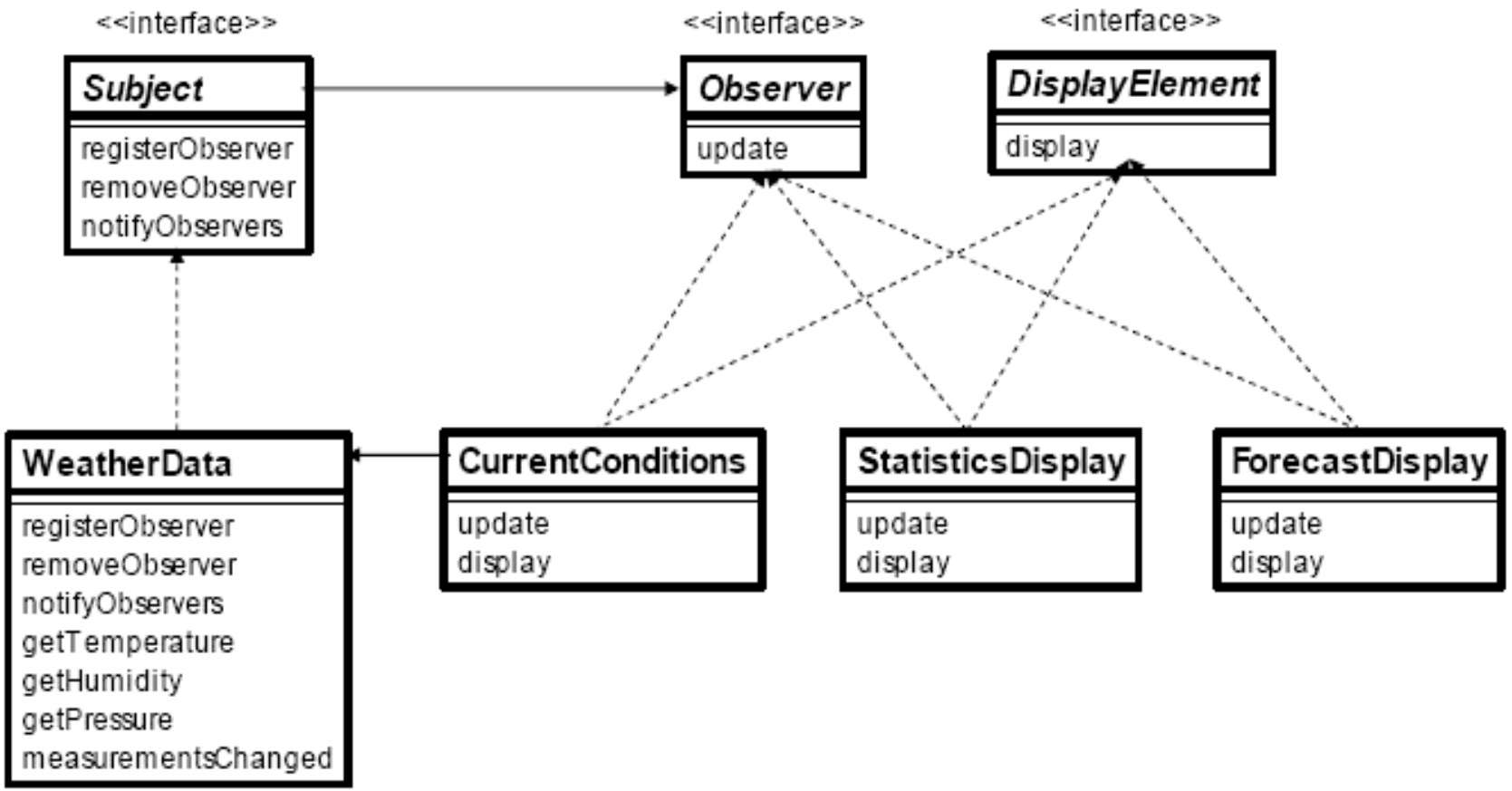
Weather Monitoring Station Application



Weather Monitoring Station Application



Solution


<<interface>>
Subject
registerObserver
removeObserver
notifyObservers
<<interface>>
Observer
update
<<interface>>
DisplayElement
display
WeatherData
registerObserver
removeObserver
notifyObservers
getTemperature
getHumidity
getPressure
measurementsChanged
CurrentConditions
update
display
StatisticsDisplay
update
display
ForecastDisplay
update
display

Observer

Intent

- **Defines a one-to-many dependency among objects so that when one object changes state, all its dependents are notified and updated automatically**

- **A way of notifying change to a number of classes**

Observer

- **Also known as**
  - **Dependents**
  - **Publish-Subscribe**
- **Motivation**
  - **To avoid making classes tightly coupled that would reduce their reusability**

Observer

- **Design Principle**

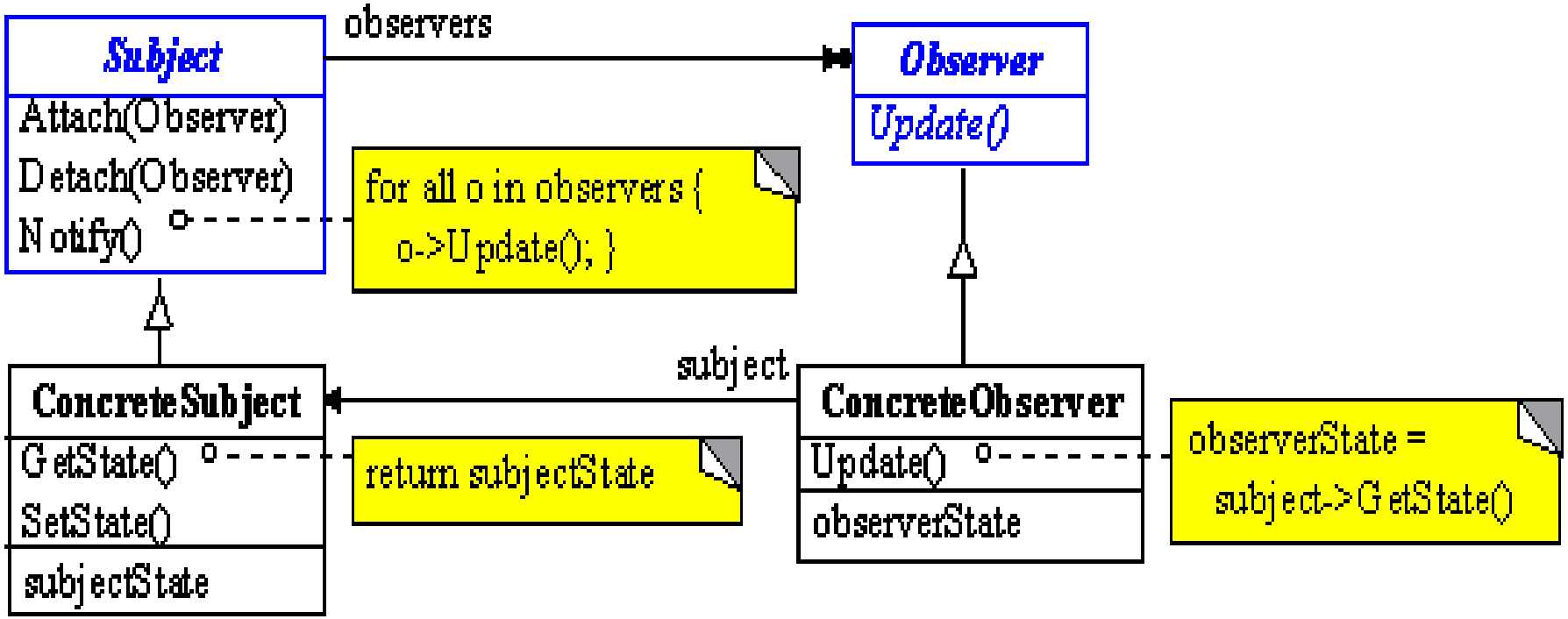
  – Strive for loosely coupled designs among objects that interact

- **• Use this pattern when:**

  –A change to one object requires changing others, and the number of objects to be changed is unknown

  – An object should be able to notify other objects without making assumptions about who these objects are Avoids having these objects tightly coupled

Observer (Continued)



Class Structure

Subject
Knows it observers
Has any number of observer
Provides an interface to attach and detaching observer object at run time

Observer
Provides an update interface to receive signal from subject

ConcreteSubject
Store subject state interested by observer
Send notification to it's observer

ConcreteObserver
Maintain reference to a ConcreteSubject object
Maintain observer state
Implement update operation

Collaborations

- ConcreteSubject notifies its observers whenever a change that could make it's state inconsistent with observers.

- After a ConcreteObserver be notified, it queries the subject state by using the **GetState** function. ConcreteObserver uses this information to change it's internal state

Implementation Issues

- **Mapping subjects to their observers**. A subject can keep track it's list of observers as observer reference or in a hash table.
- **Observing more than one subject**. It might make sense to implement many-to-many relationship between subject and observer. The Update interface in observer has to know which subject is sending the notification. One of the implement is that subject can pass itself as a parameter in the Update operation.
- **Who triggers the update (Notify operation in Subject)**.
  State setting operation in subject to trigger Notify.
  Observer to trigger Notify.
- **Push model**: subject sends details change information to observer.

Implementation Issues(Continued)

- **Dangling references to deleted subjects**. Deleting a subject or a observer should not produce dangling references.
- **Making sure subject state is self-consistent before notification**. Otherwise, an observer can query subject's intermediate state through GetState operation.
- **Avoiding observer-specific update protocols: push and pull models.**

Implementation Issues(Continued)

- **Poll model**: subject sends minimum change information to observer and observer query for the rest of the information.

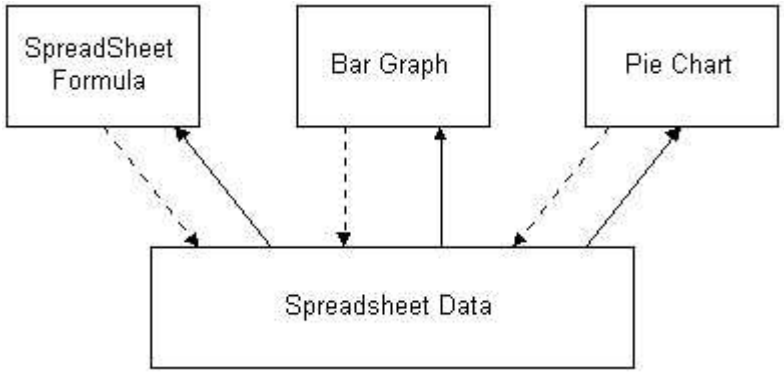
- **Specifying modifications of interest explicitly**. One can register observer for only specific events. This can improve update efficiency.
- **Encapsulating complex update semantics**. For any complex set of subject and observer relationships, one can implement Change Manage to handle their Update operation. For example, if multiple subjects have to change state before any of their observers can update. Change Manager can handle change and update sequence for the operation.

Example Usage - Simple

An example of using the observer pattern is the graphical interface toolkit which **separates the presentational aspect with application data**. The presentation aspect is the **observer** part and the application data aspect is the **subject** part.

In a spreadsheet program, the Observer pattern can be applied as in the following diagram. Each rectangular box in the diagram in an object. **SpreadSheetFormula**, **BarGraph**, and **PieChart** are the observer objects. **SpreadsheetData** is the subject object. The SpreadsheetData object notifies its observers whenever a data changes that could make it's state inconsistent with the observers.

Example Usage - Simple



Subject	• Spreadsheet Data	Send notify signal to observer object whenever data changes
Observer	• Spreadsheet Formula • Bar Graph • Pie Char	Request subject for change information in order to update itself accordingly

Applicability

Use the observer pattern in any of the following situations:

- When the abstraction has **two aspects with one dependent on the other**. Encapsulating these aspects in separate objects will increase the chance to reuse them independently.

- When the subject object doesn't know exactly how many observer objects it has.

- When the subject object should be able to notify it's observer objects without knowing who these objects are.

Consequences

- Further benefit and drawback of Observe pattern include:
  - Abstract coupling between subject and observer, each can be extended and reused individually.
  - Dynamic relationship between subject and observer, such relationship can be established at run time. This gives a lot more programming flexibility.
  - Support for broadcast communication. The notification is broadcast automatically to all interested objects that subscribed to it.
  - Unexpected updates. Observes have no knowledge of each other and blind to the cost of changing in subject. With the dynamic relationship between subject and observers, the update dependency can be hard to track down

A complete example

A Silly Text Processor:

Counts the number of words that start with an uppercase letter

Save the lines to a file

Shows the progress (e.g., then number of lines processed)

A complete example (Cont'd)

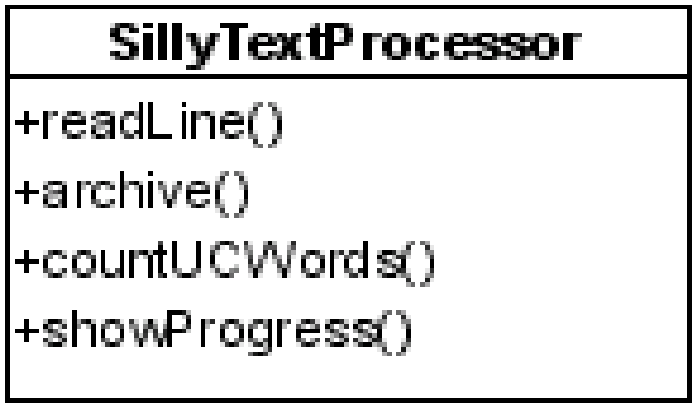
Some Observations:

This is not going to make us any money

We can use it to explore different designs

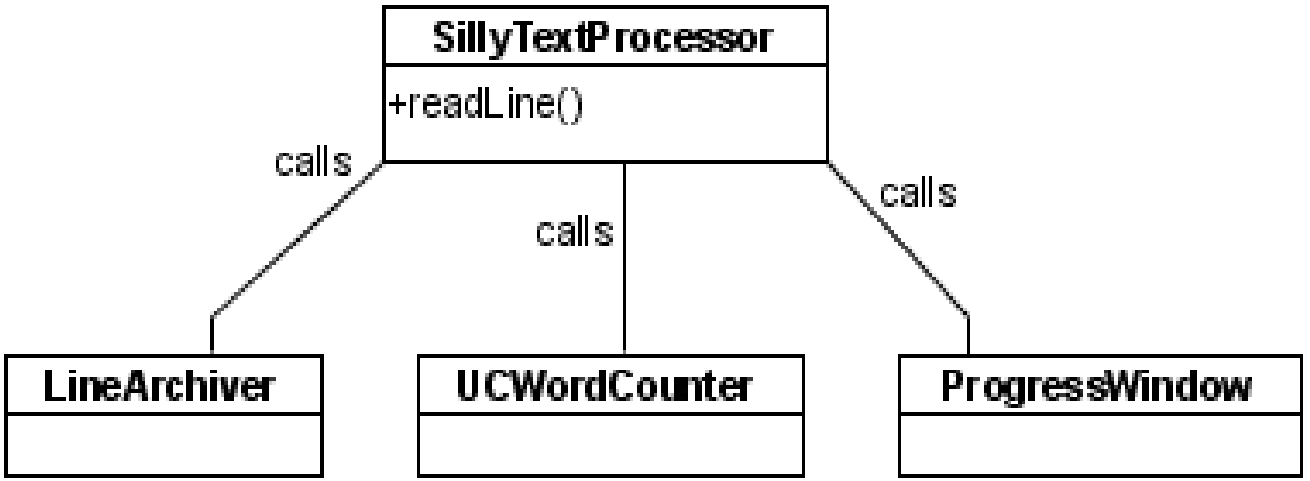
A complete example (Cont'd)

A Bad Design:



A complete example (Cont'd)

A Better Design:



A complete example (Cont'd)

A Better Design:

This design is better.

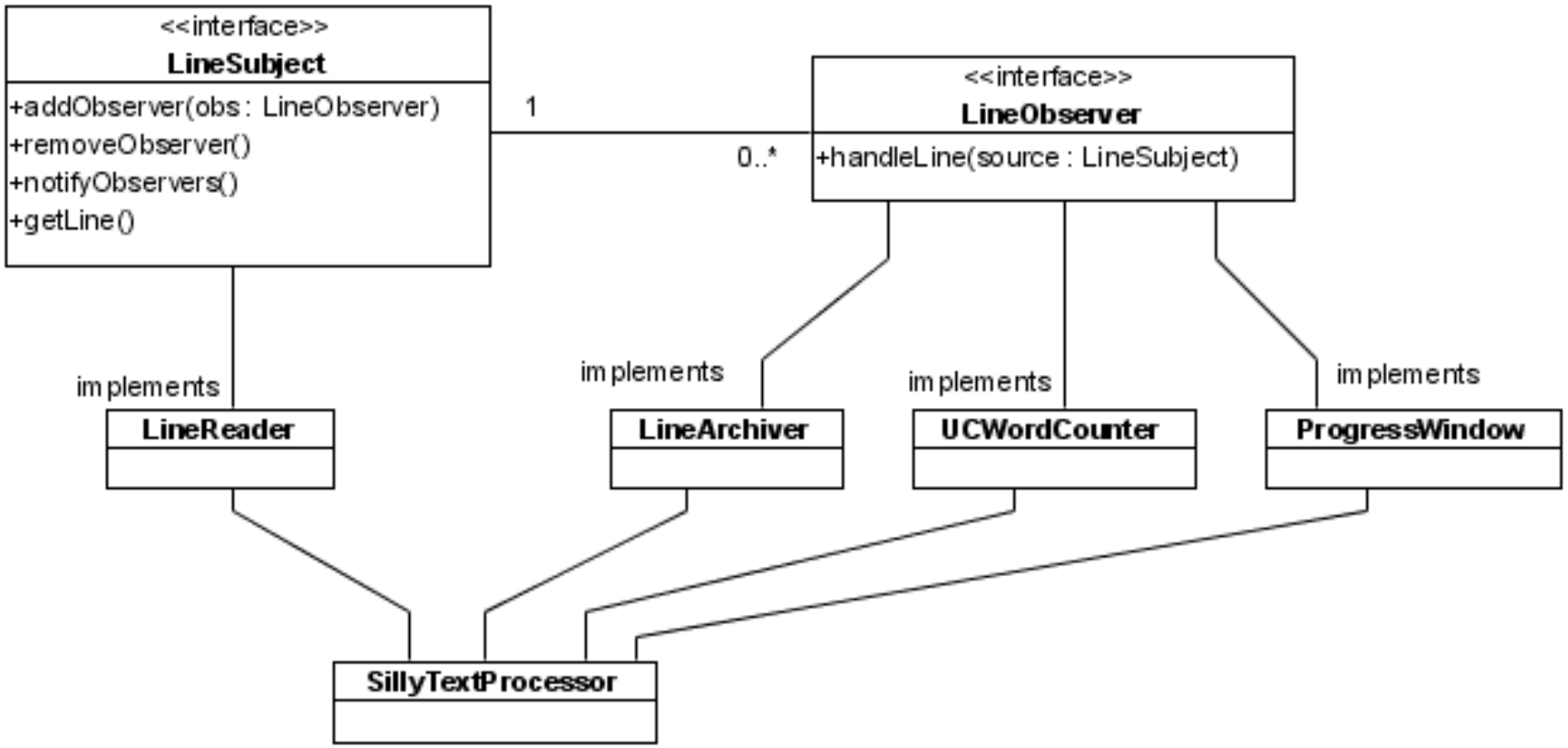
It is, however, too tightly coupled.

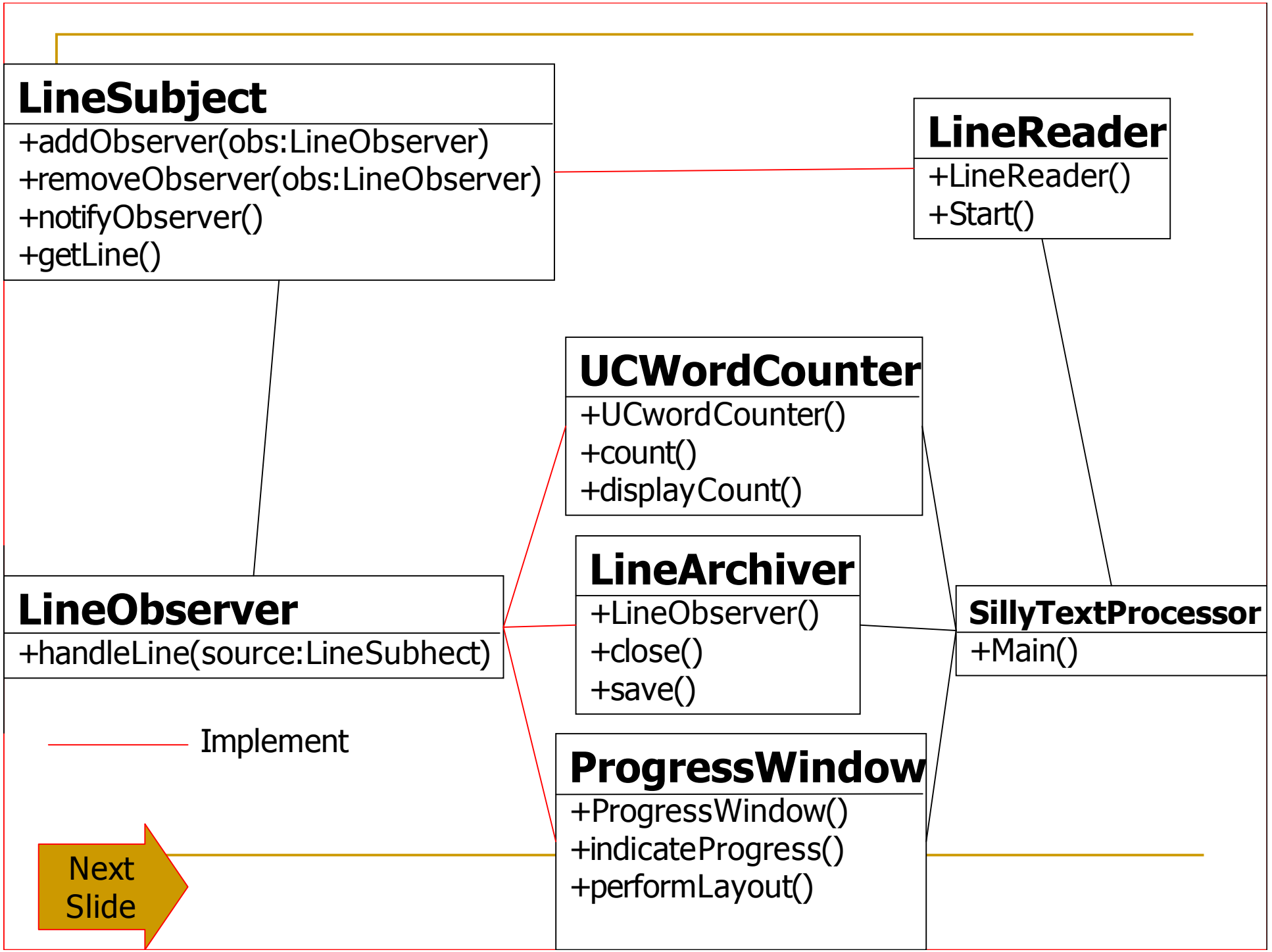
A complete example (Cont'd)

How to design it better?

A complete example (Cont'd)

Using the Observer Pattern




LineSubject
+addObserver(obs:LineObserver)
+removeObserver(obs:LineObserver)
+notifyObserver()
+getLine()
LineReader
+LineReader()
+Start()
UCWordCounter
+UCwordCounter()
+count()
+displayCount()
LineArchiver
+LineObserver()
+close()
+save()
LineObserver
+handleLine(source:LineSubhect)
SillyTextProcessor
+Main()
Implement
ProgressWindow
+ProgressWindow()
+indicateProgress()
+performLayout()
Next Slide

Good things to know about the Observer Pattern

- Most heavily used
  (Compared to real life: Subscription to a newspaper or magazine)
- Incredibly useful
- Keeps objects in the know
- Give objects the maximal freedom
  (whether they want to be informed)

Known uses

- Smalltalk Model/View/Controller (MVC). User interface framework while Model is subject and View is observer.
- Smalltalk ET++, and the THINK class library provide the general Observer pattern.
- Other user interface toolkits such as InterViews, the Andrew Toolkit, and Unidraw.

Related Pattern

- **Mediator** : Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently
- **Singleton** : Ensure a class only has one instance, and provide a global point of access to it.

Design Principle Challenge

- Identify the aspect of your application that vary and separate them from what stays the same.
- Program to an interface, not implementation.
- Favor composition over inheritance.

  (Hint: think about how observers and subjects work together.)

Design Principle Challenge

- **Design Principle**

  Identify the aspects of your application that vary and separate them for what stays the same.

The thing that varies in the Observer Pattern is the state of the Subject and the number and the type of Observers. With this pattern, you can vary the objects that are dependent on the state of the subject, without having to change that Subject. That's called planning ahead!

Design Principle Challenge

- **Design Principle**

  Program to an interface, not an implementation.

Both the Subject and Observer use interfaces. The Subject keeps track of objects implementing the Observer interface, while the observers register with, get notified by, the Subject interface. As we've seen, this keeps things nice and loosely couples.

Design Principle Challenge

- **Design Principle**

  Favor composition over inheritance.

  This is a hard one, hint: think about how observers and subjects work together.

The Observer Pattern uses competition to compose any number of Observers with their Subjects. These relationship aren't set up by some kind of inheritance hierarchy. No, they are set up at runtime by composition!

A few questions...

1. One subject likes to talk to .... observers.
2. Observers are ..... on the Subject.
3. A Subject is similar to a .....
4. ..... can manage your observers for you
5. Observers like to be ..... when something new happens.
6. Java framework with lots of Observers.
7. You want to keep your coupling .....
8. Program to an ..... not an implementation.
9. The WeatherData class ..... the Subject interface.

A few questions... (Cont'd)

Solutions

0. One subject likes to talk to many observers.
1. Observers are dependent on the Subject.
2. A Subject is similar to a publisher.
3. Observable can manage your observers for you.
4. Observers like to be notified when something new happens.
5. Java framework with lots of Observers: Swing
6. You want to keep your coupling loose.
7. Program to an interface not an implementation.
8. The WeatherData class implements the Subject interface.

Conclusion

Conclusion

- **Design Principle**
  - **Strive for loosely coupled designs among objects that interact**

- **Use this pattern when:**

- A change to one object requires changing others, and thenumber of objects to be changed is unknown

- An object should be able to notify other objects without making assumptions about who these objects are

+ Avoids having these objects tightly coupled

Reference

- en.wikipedia.org/wiki/Observer_pattern
- Head First Design Patterns 2004 O'Reilly First Edition
- https://users.cs.jmu.edu/bernstdh/web/common/lectures/slides_observer_pattern.php
- http://www.hillside.net/