Observer Pattern

Software Design Pattern

- In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

- It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

- Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Extracted from https://en.m.wikipedia.org/wiki/Software_design_pattern

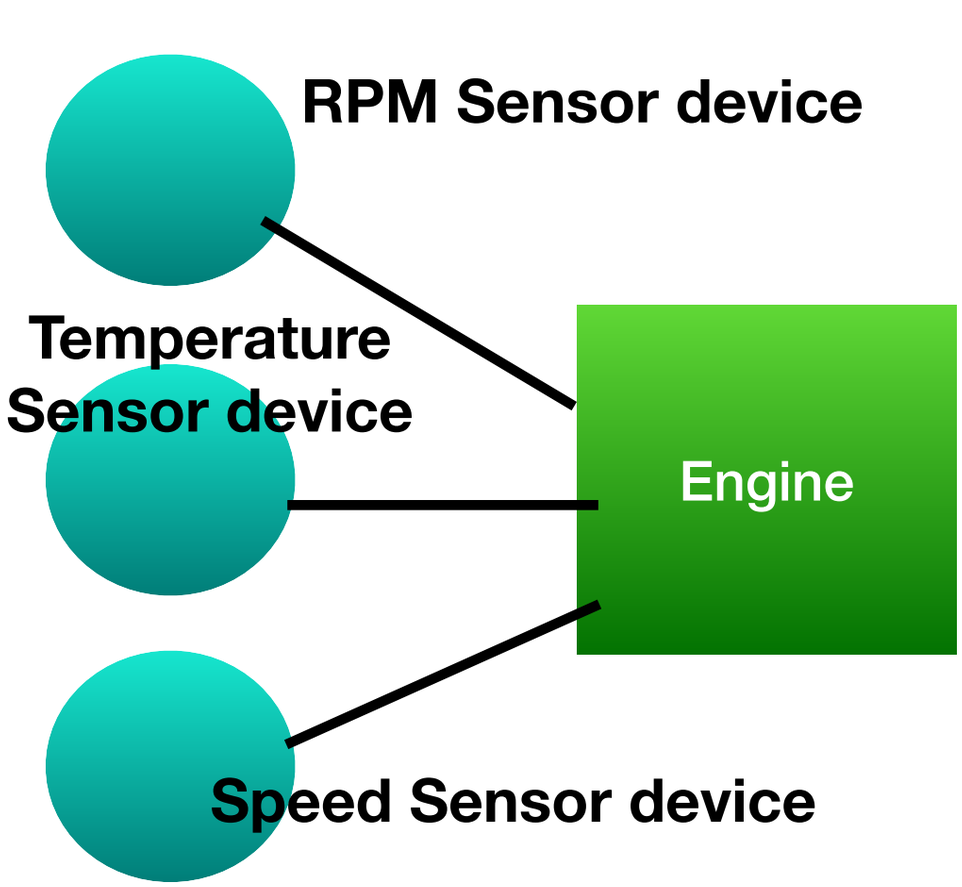
Observer Pattern

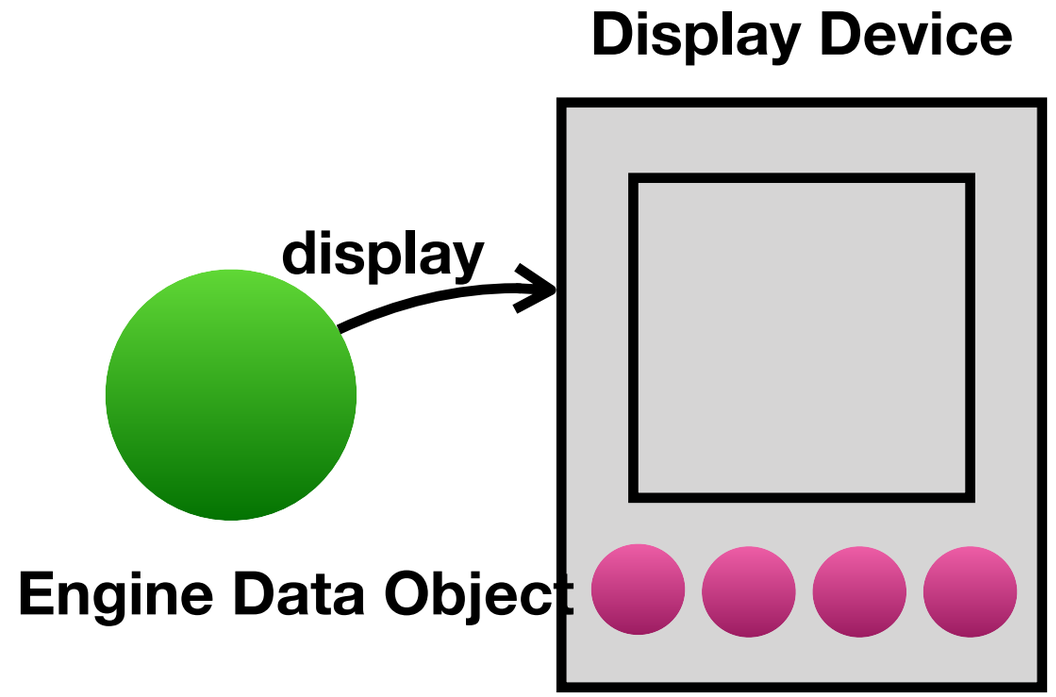
- The observer pattern is useful for state monitoring and event handling situations. This pattern allows a given object to be monitored by an unknown and dynamic group of "observer" objects.

- The Observer Pattern keeps your objects informed when something they might care about happens. Objects can even decide at run time whether they want to be kept informed. The Observer pattern is one of the most heavily used patterns in the JDK or .NET FCL, and it is very useful. We will also look at one to many relationships and loose coupling .

Case Study - The Car Dashboard Application

- The three players in the system are the Car Dashboard that acquires the actual engine data, the EngineData object that tracks the data coming from the Engine and updates the displays, and the display that shows users the current conditions.
- The EngineData object knows how to talk to the physical Engine, to get updated data. The EngineData object then updates its displays for the three different display elements: Analog Display (shows temperature, rpm, and speed), Digital Display, and a simple gauge.





Project Requirements

1. Our job is to create an app that uses the EngineData object to update the three displays for analog display, digital display, and a gauge. These displays must be updated each time EngineData has new measurements.
2. The EngineData class already exists, it has getter methods for three measurement values: temperature. speed and rpm. And we need to implement MeasurementChanged() so that it updates the three displays for analog display, digital display, and gauge.
3. The measurementsChanged() method is called any time new engine measurement data is available.
4. The system must be expandable - other developers can create new custom displays and user can add or remove as many displays as they want to the application.

The following code has design problems

- By coding to concrete implementations we have no way to add or remove other displays without making changes to program

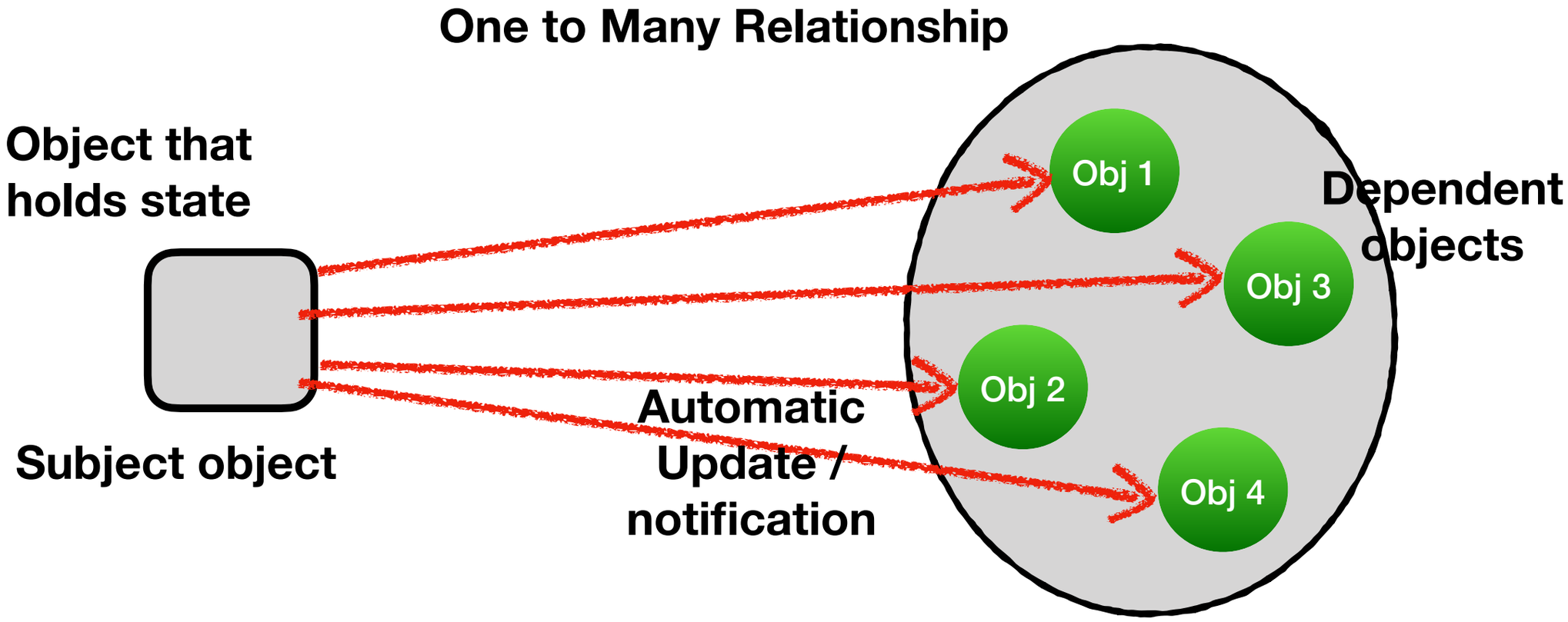
- The displays seem to have a common interface that has an update() method takes the temp, humidity, and pressure values.

```
class EngineData:
    def measurementsChanged(self):
        temp = getTemperature()
        rpm = getRPM()
        speed = getSpeed()

        analogDisplay.update(temp, rpm, speed)
        digitalDisplay.update(temp, rpm, speed)
        gaugeDisplay.update(temp, rpm, speed)
```

The Observer Pattern

- Defines a one to many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- **Observer Pattern = Publisher (Subject) + Subscriber (Observer)**
  - The **SUBJECT** likes a newspaper publisher begins publishing its newspaper.
  - The **OBSERVER** will subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as your remain a subscriber.
  - You unsubscribe when you don't want newspaper anymore, and they stop being delivered.



Typical Class design for observer pattern

- There are few different ways to implement the Observer Pattern but most revolve around a class design that includes Subject and Observer Interfaces.
    1. Subject interface : objects use this interface to register as observers and also to remove themselves from being observers.
    2. Observer interface : all potential observers need to implement the observer interface. This interface just has one method, update() that gets called when the subject's state changes.
    3. Concrete subject ; implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.
    4. The concrete subject may also have methods for setting and getting its state.
    5. Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.
- Advantage
    - The subject is the sole owner of the data, the observers are dependent on the subject to update them when the data changes. This leads to a good OO design than allowing many objects to control the same data.

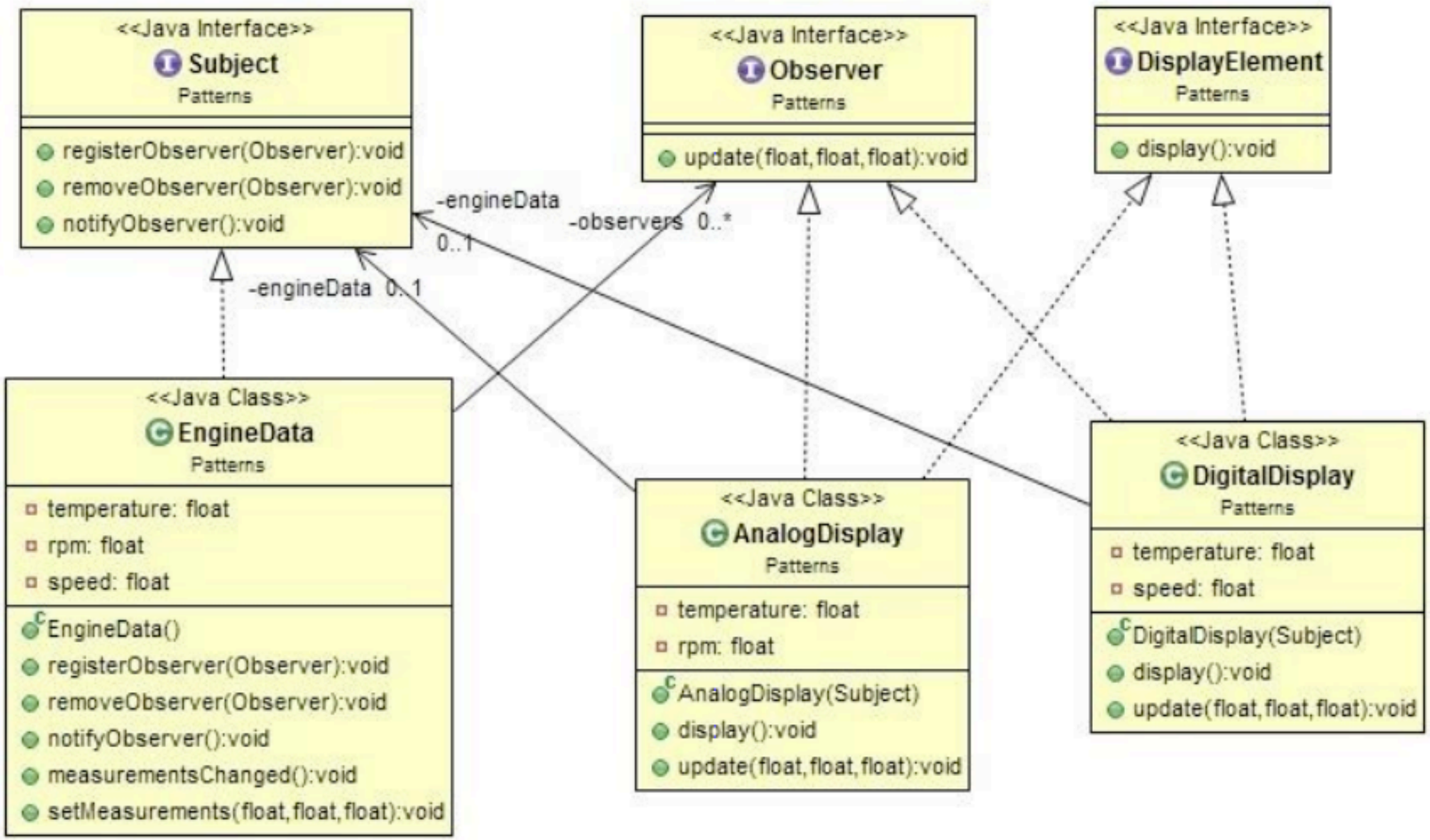
Loose Coupling

1. When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
2. The Observer Pattern provides an object design where subjects and observers are loosely coupled.
3. The Subject knows nothing about the observer except the Observer interface.
4. We can add new observers at any time. Likewise, we can remove observers at any time.
5. We never need to modify the subject to add new types of observers.
6. We can reuse subjects or observers independently of each because they are not tightly coupled.
7. Changes to either the subject or an observer will not affect the other.

- Advantages of Loose Coupling
  - Allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Designing the Dashboard

- Implement the Car Dashboard App, the EngineData class and its display elements using Class Diagrams.



Exercise: Academic Dashboard

- Implement the following class diagram in Python.