CS 247

Software Engineering Principles

Chapter 10

The Observer Pattern

Victoria Sakhnini

Table of Contents

The Problem: Weather-O-Rama

In the previous chapter, we argued that good designs name what varies, program to interfaces, and prefer composition over inheritance. Those are principles. This chapter takes those principles and lifts them off the page: we will apply them to a single, concrete problem and watch them crystallize into one of the most widely used patterns in object-oriented software, **the Observer Pattern**.

We will use the running example introduced in the lecture deck and borrowed from the Head First Design Patterns book. A start-up named *Weather-O-Rama, Inc.* wants to build its next-generation, Internet-based weather monitoring station. There are three players in their system:

- **(1) The weather station itself**, a physical device sitting outside, with thermometers, hygrometers, and barometers, producing raw measurements.
- **(2) A** `WeatherData` object, a piece of software that reads the device, holds the latest temperature, humidity, and pressure, and exposes these to the rest of the program through getters.
- **(3) A set of display elements**, small UI widgets that have to redraw themselves every time the weather data updates.

Right now, Weather-O-Rama has three displays in mind:

- **Current Conditions Display** shows the temperature and humidity right now.
- **Statistics Display** shows the minimum, maximum, and average temperature observed so far.
- **Forecast Display**, uses the barometric pressure to make a crude forecast (rising pressure → improving, falling → stormy, steady → more of the same).

But this is the part that matters: Weather-O-Rama wants the system to be *expandable*. They expect third-party developers to write their own display elements and plug them in. They expect end users to freely add or remove display elements. They cannot enumerate the set of displays in advance, because they do not know who will be writing them or when. The first version of the system will ship with three displays; the tenth version might ship with thirty.

That word "expandable" is the centre of gravity for this entire chapter. If Weather-O-Rama only ever needed three displays, and they were always going to be the same three displays, written by the same person, with no third-party plug-ins, no end-user customization, no future evolution, then almost any design would work. But the moment we admit that the set of displays will change, we have admitted that *the things using the data are not under our control*. And once that is true, we must design the `WeatherData` object to notify a set of receivers that *it does not know in advance*.

That problem, "one object needs to keep an unknown number of unknown other objects in sync with its state", is the problem the Observer Pattern solves. Before we name the pattern, let us first try to solve it badly. The badness will make the pattern's elegance visible.

> **? Question**
> Before reading on, sketch a solution yourself. How would you let a `WeatherData` object update a `CurrentConditionsDisplay`, a `StatisticsDisplay`, and a `ForecastDisplay` whenever the measurements change, without forcing `WeatherData` to know in advance which displays exist?

A First Attempt Without the Pattern

The most natural way to wire up the system, if you have never met the Observer Pattern, is to give `WeatherData` a method called `measurementsChanged()` and have it call the displays directly. Something like this:

```
// weatherdata.cpp  --  NAIVE VERSION (do not ship this)
void WeatherData::measurementsChanged() {
    float t = getTemperature();
    float h = getHumidity();
    float p = getPressure();

    currentConditionsDisplay.update(t, h, p);
    statisticsDisplay.update(t, h, p);
    forecastDisplay.update(t, h, p);
}
```

This works. If you compile and run it, you will see all three displays redraw whenever the measurements update. So what is wrong with it?

What is wrong, in plain English

Look at the code one more time and ask: which of the three design principles from the previous chapter is this code violating?

- **It is hard-coded against concrete types.** `WeatherData` calls `currentConditionsDisplay.update(...)`, `statisticsDisplay.update(...)`, and so on. It knows the exact class names of every display element. This violates *"program to an interface, not an implementation"*.
- **It will need to be modified whenever a display is added or removed.** Want a fourth display? Edit `measurementsChanged()` and add a fourth line. Want to remove one at runtime because the user closed that window? You can't; the call is baked into the source. This violates *"encapsulate what varies"*, because the thing that is varying, the list of displays, is sitting in the middle of a function instead of being a first-class data structure.

- **It has no notion of dynamic registration.** Third-party developers cannot add their displays without recompiling WeatherData, which means they cannot add them at all unless we ship them our source.
- **It is brittle across releases.** Every new display is a new compile-time dependency from `WeatherData.cpp` to a new header. The class that used to depend on three files now depends on thirty.

All four problems stem from the same underlying mistake: `WeatherData` *knows* what its consumers are. We want a design where it does not.

What we actually want

We want `WeatherData` to hold a *list of receivers*, to have no knowledge of the receivers' concrete types, and to expose two operations to the outside world: `registerObserver` (add me to the list) and `removeObserver` (take me off). When its measurements change, `WeatherData` walks the list and tells each receiver "something changed", through a common interface that every receiver promises to implement.

That is the entire Observer Pattern in one sentence. The remainder of the chapter unpacks that sentence: what the interface looks like, what the registration mechanism looks like, what to call the participants, where the data flows, what can go wrong, and what the concrete C++ code looks like.

> **Tip**
>
> **Smell test for the naive approach.** Whenever you find yourself writing a function whose body is a *list of method calls on hard-coded names*, and you can imagine that list growing or shrinking in the future, you almost certainly want some kind of subscriber list and some kind of broadcast call. The Observer Pattern is the canonical way to spell that intuition.

Definition: The Observer Pattern

Here is the official definition as it appears in *Design Patterns: Elements of Reusable Object-Oriented Software*, the 1994 book by Gamma, Helm, Johnson, and Vlissides (the "Gang of Four"):

> **Tricky**
>
> **Observer Pattern.** Define a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Read that sentence three times. Every word is doing work. "One-to-many" is the multiplicity, a single subject can have many observers. "Dependency" is the structural relationship: observers depend on the subject for its state. "Changes state" is the trigger; notification happens when something interesting changes, not on every method call. "Notified and updated automatically" is the behavioural promise; the subject does the calling; observers do not have to poll.

Two names, same actors

The pattern's literature is unfortunately split between two vocabularies, and you will see both in production code and textbooks. We will use the GoF names throughout this chapter, but you should recognize the synonyms instantly:

GoF name (this chapter)	Java / .NET name	What it does
Subject	Observable	Holds the state that observers care about; manages the list of observers; notifies them.
Observer	Observer	Receives notifications; reads (some of) the subject's state in response.
ConcreteSubject	Concrete Observable	A real class that implements Subject, e.g., WeatherData.
ConcreteObserver	Concrete Observer	A real class that implements Observer, e.g., CurrentConditionsDisplay.

Java's standard library historically had a `java.util.Observable` class and a `java.util.Observer` interface (both deprecated in Java 9), which is why you will hear Java programmers say "Observable" instead of "Subject". C++ has no equivalent in its standard library; there is no `std::observable`, so we will roll our own. That is the subject of the implementation section.

The minimal API

Every implementation of the pattern, regardless of language, exposes the same three operations on the Subject side:

- `registerObserver(observer)`, adds an observer to the subject's list. Some literature calls this `"attach" or "subscribe"`.
- `removeObserver(observer)`, takes an observer off the list. Synonyms: `detach`, `unsubscribe`.
- `notifyObservers()`, walks the list and calls `update()` on every entry. Synonyms: `broadcast`, `publish`.

And on the Observer side, a single operation:

- `update()`, called by the subject when something has changed. Inside this method, the observer reacts: it might read the subject's current state, re-render itself, log an event, send an email, ring a bell.

That's the whole interface. Four methods. Everything else in the pattern is convention, naming, and good taste about who calls whom.

Anatomy of the Pattern

The canonical UML class diagram for the Observer Pattern looks like this:

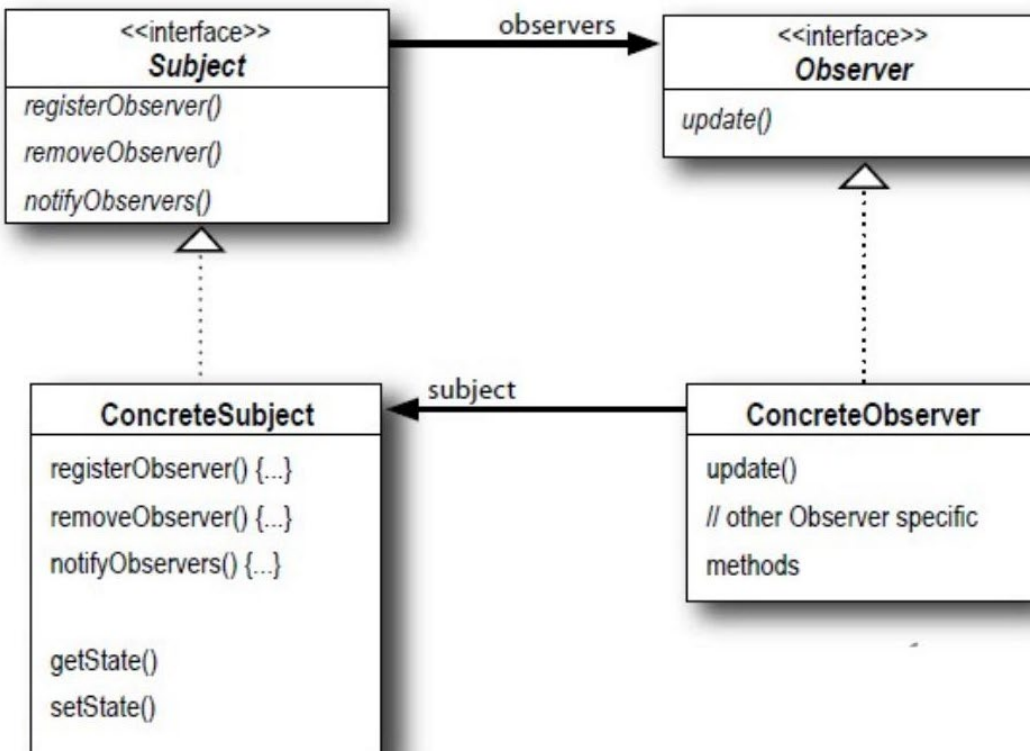

Figure 4.1: The Observer Pattern. A Subject manages a collection of Observers. ConcreteSubject and ConcreteObserver are application-specific classes that implement the interfaces.

Spend a moment reading the diagram before reading on. There are four classes (well, two of them are interfaces), three associations, and two generalizations. Let us walk through each piece:

The Subject interface

Top-left of the diagram. The **Subject** is an abstract interface declaring `registerObserver`, `removeObserver`, and `notifyObservers`. It is what observers "attach to". Any concrete class that

wants to be observable, a stock ticker, a weather station, a chat room, or a button implements this interface.

Notice the heavy arrow labelled Observers, pointing from Subject to Observer with no diamond. This is a plain association with multiplicity many: a single Subject holds (and can navigate to) many Observers. In implementation terms, the concrete Subject will store a std::vector<Observer*>, a list of pointers to objects implementing the Observer interface.

The Observer interface

Top-right of the diagram. The **Observer** is an abstract interface with one method: `update()`. This is the only obligation an observer takes on. It does not have to declare what it does with the update, what state it stores, what the user sees, or how it logs the event. The contract is one method, with no return value.

Because the contract is so narrow, *any* class in the system can become an observer simply by implementing one method. That narrowness is not an accident; it is the source of the pattern's flexibility, as we will explore in the next section.

ConcreteSubject

Bottom-left of the diagram. **ConcreteSubject** is the actual class doing something interesting, in our running example, `WeatherData`. It inherits from Subject (so it must implement the three registration/notification methods) and adds its own state, plus methods to read and modify it. The empty braces `{...}` in the diagram mean "implementation supplied here, in this class."

Note the optional methods `getState()` and `setState()`, these stand in for whatever business operations the concrete subject offers. For `WeatherData`, these will be `getTemperature()`, `getHumidity()`, `getPressure()`, and `setMeasurements(...)`.

ConcreteObserver

Bottom-right of the diagram. **ConcreteObserver** is an actual class that wants to be told when the subject changes, in our running example, the three (and later four) display classes. It inherits from Observer and provides a real implementation of `update()`.

The thin arrow from ConcreteObserver back to ConcreteSubject, labelled `subject`, indicates that the observer typically holds a reference back to the subject. This is not strictly required by the pattern; there are "push" variants where the subject sends all the data with the notification, but it is the more common arrangement and the one we will use. We will explore the trade-off later in this chapter.

> **Question**
>
> Look at Figure 4.1. The arrow from Subject to Observer is a *plain association* (no diamond), while the arrow from ConcreteSubject to Subject is a *generalization* (open triangle). What is the relationship between ConcreteSubject and Observer? Trace through the diagram to find out.
>
> **Answer.** ConcreteSubject doesn't have a direct relationship with Observer in the diagram, but it inherits the `observers` association from Subject. So a ConcreteSubject *can* reach many Observers, it has an `observers` field, but it only knows them as "things that implement the Observer interface". This indirection is the entire point of the pattern.

Loose Coupling: Why the Pattern Works

> **Tricky**
>
> **Design Principle.** Strive for loosely coupled designs among interacting objects.

"Loose coupling" is one of those phrases that gets thrown around in software engineering until it stops meaning anything. Let us pin it down. Two classes are *tightly coupled* if changes to one of them force changes to the other. They are *loosely coupled* if you can change one without touching the other, as long as both still honour their public agreements.

By that definition, the naive design in the second section was about as tightly coupled as design gets. `WeatherData` knew the concrete class of every display; the displays knew the concrete class of `WeatherData`; a change to either side rippled outward. The Observer Pattern fixes this by introducing two interfaces in the middle. Let us spell out exactly what we gain.

The Subject doesn't know the concrete class of any observer

The only thing the subject knows about an observer is that it implements the **Observer** interface, that is, that it has an `update()` method. It does not know the concrete class, what the observer does in response to `update()`, how the observer is constructed, whether it draws to the screen or sends an email, or even what file it lives in.

In implementation terms, the subject holds a `std::vector<Observer*>`. The static type of every element of that vector is `Observer*`. The dynamic type can be anything that inherits from Observer.

This is the standard C++ trick for polymorphic collections: store base-class pointers, let virtual dispatch do the work.

We can add new observers at any time

Because the subject depends only on the Observer interface, we can add new observers at compile time, link time, or runtime through dynamic registration. We can write a new `RainfallAlarmDisplay` class six months after `WeatherData` was finalized, ship it as a separate library, and have it register itself with an existing `WeatherData` instance, with no changes to WeatherData whatsoever.

Note especially: this is true even at runtime. We can replace one observer with another while the program is running. We can remove an observer when the user closes a window and add it back when they reopen it. The subject doesn't notice; it iterates its list and calls `update()` on whatever it finds there.

We never need to modify the subject to add new types of observers

This is the principle that distinguishes a closed product from an extensible platform. The naive design from Section 2 had to be modified for every new display. The Observer Pattern lets us add new display types, entire new classes, in new files, with new behaviours, without editing `WeatherData.cpp` at all. The Open–Closed Principle from earlier in the course says we should design classes that are "open for extension, closed for modification". This is what that looks like in practice.

We can reuse subjects or observers independently of each other

Because the two sides only know about interfaces, you can take a subject out of one application and drop it into another, as long as you supply the observers it needs. Likewise, an observer designed for one subject can be reused with a different subject that publishes the same kind of state. Loose coupling is what makes parts genuinely reusable; tight coupling is what makes "reusable parts" turn into ten-page integration guides.

Changes to either side do not affect the other

This is the cumulative payoff of all the previous points. You can rewrite `WeatherData`, change how it reads the sensors, what units it stores, how it caches values, and as long as it still calls `update()` on its observers, every observer continues to work without modification. Conversely, you can rewrite a display, change its layout, add animations, and log to a file; as long as it still implements `update()`, the subject does not notice.

> **Tip**
>
> **How to feel coupling.** Imagine you are about to delete a class from your project. How many other files will not compile until you also edit them? The answer is a quantitative measure of how coupled that class is to its neighbours. A well-applied Observer Pattern keeps that number very small on both sides.

Designing the Weather Station

With the pattern in hand, let us return to Weather-O-Rama. The lecture instructs us to "get inspired" by the canonical UML in Figure 4.1 and apply it to our concrete classes. Here is the result:

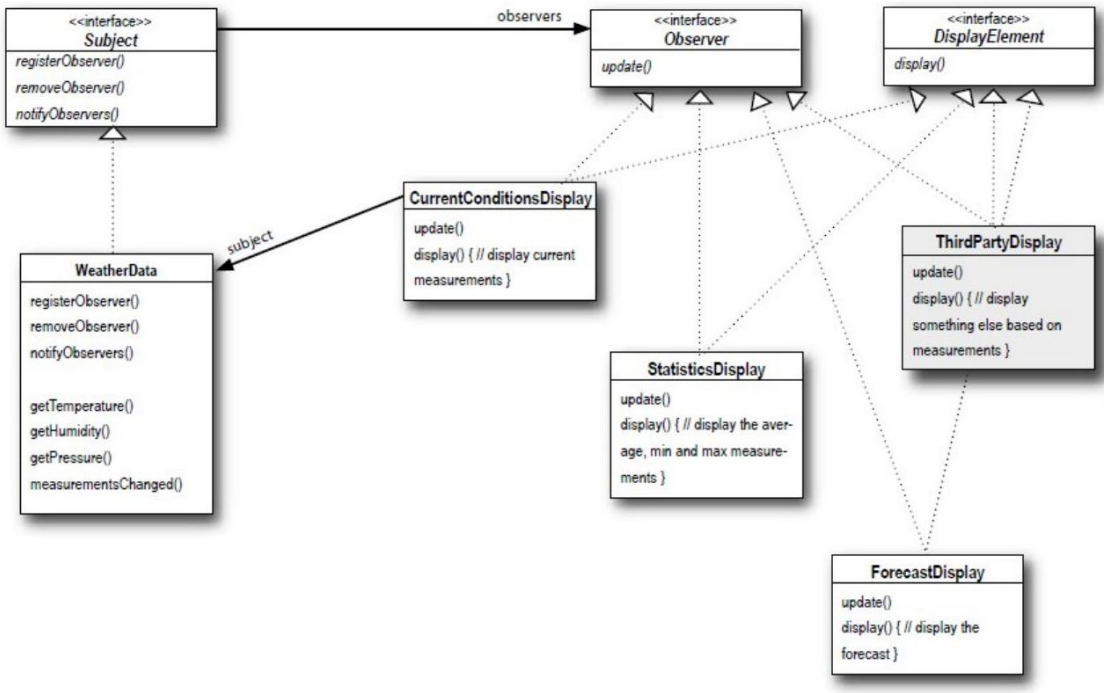

Figure 6.1: The Weather Station design. WeatherData is the ConcreteSubject. The four displays are ConcreteObservers that also implement DisplayElement, a second interface that promises a display() method.

There are six things worth pointing out in this diagram.

- **(1) WeatherData implements Subject.** It inherits the registration and notification interface and adds its own state: temperature, humidity, pressure, plus getters and a `measurementsChanged()` method that the sensor-reading code will call.

- **(2) All four displays implement Observer.** This gives them a uniform `update()` method that the subject can call without knowing their concrete types.
- **(3) There is a second interface, DisplayElement, with a single** `display()` method. This is a separate concern from Observer. Observer says "I want to know when something changes." DisplayElement says, "I am something that can render itself to the screen." The two are orthogonal, and the displays implement both.
- **(4) ThirdPartyDisplay is in the diagram.** We have not written it yet, but we are designing for it. Its existence in the diagram is a reminder that the architecture must accommodate observers that we did not foresee.
- **(5) Every display has a** `subject` pointer back to WeatherData. In the diagram, this is only drawn once (from CurrentConditionsDisplay) for readability, but conceptually every display has one. The reason is that `update()` carries no parameters; when notified, the observer has to ask the subject, "So, what changed?" We will discuss this design choice in Section 8.
- **(6) Composition over inheritance, twice.** WeatherData does not *inherit from* its observers; it contains them as a list. The displays do not inherit from WeatherData; they hold a pointer to it. Both directions of communication use composition, which keeps the inheritance hierarchy shallow and the system flexible.

Why two separate interfaces?

A common student question at this stage: Why are Observer and DisplayElement separate interfaces? Why not merge them into a single interface, UpdatableDisplay, with both an `update()` and a `display()` method?

The answer is the **Single Responsibility Principle**. The two methods serve different concerns:

- `update()` is about *reactive behaviour*: something changed, do whatever you need to do internally to absorb that change. An observer that logs to a file would perform its file write inside `update()`, and that observer is not a "display" at all.
- `display()` is about *rendering*: draw yourself, show the user what you know. A display that also has a refresh button calls `display()` without `update()`; it has no new data, but the user wants to see what it has.

Keeping the two interfaces separate means a class can be an observer *without* being a display (e.g., a logger), or a display without being an observer (e.g., a static splash screen). The classes we are building happen to be both, and they implement both interfaces accordingly, but that is a property of the specific classes, not of the architecture.

Implementing the Weather Station in C++

Time to write code. We will build each class in the diagram, one file at a time, in the order a designer typically picks them up: the two interfaces first, then the concrete subject, then the concrete observers, then `main.cpp` to glue everything together. Eight C++ source files in total. We will use raw pointers throughout. Later on, in this chapter, we will discuss why the smart-pointer version would change and what it would change.

The Subject interface

File `subject.h`:

```
#ifndef SUBJECT_H
#define SUBJECT_H

class Observer;

class Subject {
public:
    virtual ~Subject() = default;
    virtual void registerObserver(Observer* o) = 0;
    virtual void removeObserver(Observer* o)   = 0;
    virtual void notifyObservers()             = 0;
};

#endif // SUBJECT_H
```

Three things to notice in those fourteen lines:

- **Forward declaration of Observer.** Line 4 says `class Observer;` without including `observer.h`. This is sufficient because the only thing Subject does with Observer is take a pointer to it, and the compiler can reason about a pointer-to-X without knowing what X looks like. Forward-declaring breaks the circular dependency between subject.h and observer.h.
- **Virtual destructor.** Line 8: `virtual ~Subject() = default;`. Any class meant to be used polymorphically must have a virtual destructor, otherwise `delete subjectPtr` through a base pointer skips the derived-class destructor. We default it because there's nothing to clean up in the interface itself.
- **Three pure-virtual methods.** The `= 0` suffix says "no implementation here; subclasses must provide one". This makes Subject an abstract class, you cannot instantiate it directly. Trying to write `Subject s;` is a compile error, which is exactly what we want from an interface.

> **Tricky**
>
> **The `= 0` idiom.** In C++, the syntax `virtual T method() = 0;` makes `method` a *pure virtual function*. It has no body in the base class, and any concrete (non-abstract) descendant must provide one. This is C++'s equivalent of Java's `abstract` keyword. A class with at least one pure virtual function cannot be instantiated.

The Observer interface

File `observer.h`, even shorter:

```
#ifndef OBSERVER_H
#define OBSERVER_H

class Observer {
public:
    virtual ~Observer() = default;
    virtual void update() = 0;
};

#endif // OBSERVER_H
```

One pure virtual method, `update()`, with no parameters. The lack of parameters is significant; the lecture deck calls this the "pull" style, and we will discuss why it is considered more correct than the alternative "push" style in Section 8. For now, when the subject changes, observers are told "something changed," and they must go back to the subject to find out what it is.

The DisplayElement interface

File `displayelement.h`:

```
#ifndef DISPLAYELEMENT_H
#define DISPLAYELEMENT_H

class DisplayElement {
public:
    virtual void display() = 0;
};

#endif // DISPLAYELEMENT_H
```

Smaller still. Note that this interface omits the virtual destructor; that's because in practice no one will ever delete a DisplayElement through a DisplayElement-pointer (you always delete through Observer or the concrete type). In a production codebase, you would still add `virtual ~DisplayElement() = default;` defensively; we are matching the lecture's minimal version here.

WeatherData: the ConcreteSubject

Now the concrete subject. The header file `weatherdata.h` declares the class and its members:

```
#ifndef WEATHERDATA_H
#define WEATHERDATA_H

#include "subject.h"
#include "observer.h"
#include <vector>

class WeatherData : public Subject {
private:
    std::vector<Observer*> observers;
    float temperature{0.0f};
    float humidity{0.0f};
    float pressure{0.0f};
public:
    void registerObserver(Observer* o) override;
    void removeObserver(Observer* o)   override;
    void notifyObservers()             override;
    float getTemperature() const { return temperature; }
    float getHumidity()    const { return humidity; }
    float getPressure()    const { return pressure; }
    void measurementsChanged();
    void setMeasurements(float temperature, float humidity, float
pressure);
};

#endif // WEATHERDATA_H
```

Several details to absorb:

- **Inheritance.** `class WeatherData: public Subject`, public single inheritance from the abstract Subject interface. This is the C++ idiom for "WeatherData IS-A Subject".
- **The observer list.** `std::vector<Observer*> observers`, a vector of base-class pointers. The dynamic type of each entry can be any descendant of Observer. This is where the polymorphism lives.
- **Default member initializers.** `float temperature{0.0f},` initialize to a known value at declaration. Means we cannot accidentally read uninitialized garbage on construction.

- **override on every virtual.** The `override` keyword after each base-method signature is not strictly required, but it tells the compiler "this function is supposed to override a virtual function from the base class", and the compiler errors if it doesn't actually match a base method. This catches typos such as registerObserver vs. `registerObserver` at compile time.
- **Inline getters.** The `getTemperature() const { return temperature; }` trio is declared `const` because reading state does not mutate the object, and defined inline because the body is one line. The `const` lets us call these from inside an observer's update() method, which holds the subject by pointer-to-const.

Now the implementation file is `weatherdata.cpp`:

```
#include "weatherdata.h"
#include <algorithm>

void WeatherData::registerObserver(Observer* o) {
    if (o) { observers.push_back(o); }
}

void WeatherData::removeObserver(Observer* o) {
    if (auto it = std::find(observers.begin(), observers.end(), o);
        it != observers.end()) {
        observers.erase(it);
    }
}

void WeatherData::notifyObservers() {
    for (auto* observer : observers) {
        observer->update();
    }
}

void WeatherData::measurementsChanged() {
    notifyObservers();
}

void WeatherData::setMeasurements(float temperature, float humidity, float
pressure) {
    this->temperature = temperature;
    this->humidity    = humidity;
    this->pressure    = pressure;
    measurementsChanged();
}
```

Walk through this with care; the entire pattern fits within 25 lines of code, and every line does something important.

- **registerObserver.** Defensively rejects `nullptr`, then appends. We are storing raw pointers; the caller retains ownership of the pointee's lifetime. We will revisit this in Section 9.
- **removeObserver.** Uses the C++17 "if with initializer" syntax: `if (auto it = std::find(...); it != end)`. This declares `it` inside the if's scope, runs the find, and tests against `end()`. If found, `erase` removes the entry from the vector. If not found, do nothing; calling `removeObserver` on a non-registered observer is a no-op, not an error.
- **notifyObservers.** A range-based for loop walking the vector and calling `update()` on each. Because `update()` is virtual, dynamic dispatch kicks in, and the concrete observer's override runs, without `WeatherData` knowing what type that observer is.
- **measurementsChanged.** A trivial wrapper around `notifyObservers`, but a useful indirection, it gives the sensor-reading code a natural-sounding API and lets us add filtering or rate-limiting in one place if we ever want to. (For example: "only notify if temperature has changed by at least 0.1 degrees.")
- **setMeasurements.** The realistic entry point: the sensor code calls this with three new readings, the method stores them and triggers a notification. In a real product, this would be called by a thread reading the serial port from the weather station hardware.

> ⚠ **Warning**
>
> **Self-destruction during notification.** Look at `notifyObservers` carefully. We are iterating over `observers` while calling code (the observers' `update()` methods) that might, in principle, call `removeObserver` on us, including possibly removing themselves. If that happens during iteration, we have just invalidated our own iterator. This is a real-world bug in observer implementations.
>
> The lecture deck does not address this; the simple version of the pattern assumes observers do not unregister during notification. Production implementations either (a) copy the observer list before iterating, (b) defer removals until the notification finishes, or (c) use a list rather than a vector, where erase invalidates only the erased iterator.

CurrentConditionsDisplay

First concrete observer. The header `currentconditionsdisplay.h`:

```
#ifndef CURRENTCONDITIONSDISPLAY_H
#define CURRENTCONDITIONSDISPLAY_H

#include "observer.h"
#include "displayelement.h"
```

```
class WeatherData;

class CurrentConditionsDisplay : public Observer, public DisplayElement {
private:
    float temperature{0.0f};
    float humidity{0.0f};
    WeatherData* weatherData;
public:
    explicit CurrentConditionsDisplay(WeatherData* weatherData);
    void update()  override;
    void display() override;
};

#endif // CURRENTCONDITIONSDISPLAY_H
```

- **Multiple inheritance.** `class CurrentConditionsDisplay: public Observer, public DisplayElement`, yes, C++ allows this, and yes, it is the right tool here. Both Observer and DisplayElement are pure interfaces with no state, so inheriting from both is free of the diamond-of-doom complications you read about in books. The class IS-A Observer and IS-A DisplayElement at the same time.
- **Pointer back to the subject.** `WeatherData* weatherData`, used inside `update()` to pull the latest values when notified. Note: forward-declared `class WeatherData;` on line 7 instead of `#include "weatherdata.h"`, again, we only need to know about the type, not its layout, to declare a pointer.
- **explicit single-argument constructor.** Prevents implicit conversions like `CurrentConditionsDisplay d = somePointer;`, you must write the constructor name explicitly. This is good defensive coding for any constructor that takes one argument.

The implementation `currentconditionsdisplay.cpp`:

```
#include "currentconditionsdisplay.h"
#include "weatherdata.h"
#include <iostream>

CurrentConditionsDisplay::CurrentConditionsDisplay(WeatherData*
weatherData)
    : weatherData(weatherData) {
    weatherData->registerObserver(this);
}

void CurrentConditionsDisplay::update() {
    this->temperature = weatherData->getTemperature();
    this->humidity    = weatherData->getHumidity();
    display();
}
```

```
void CurrentConditionsDisplay::display() {
    std::cout << "Current conditions: " << temperature
              << "F degrees and " << humidity << "% humidity\n";
}
```

The constructor does two things: it stores the pointer to the subject, and, crucially, it *registers itself* with that subject. This is the convention used throughout this chapter: an observer registers itself in its constructor. It is not the only option (the caller could register manually after construction), but it has the very nice property that you cannot accidentally construct an observer and forget to wire it up.

Inside `update()`, we see the pull style in action: when notified, we reach back to `weatherData` and ask for the values we care about, just temperature and humidity, not pressure. This is the second great strength of the pull style: each observer takes exactly the data it needs, no more, no less. Then `display()` prints to the console.

ForecastDisplay

The forecast display uses two pressure snapshots to infer the trend. Header `forecastdisplay.h`:

```
#ifndef FORECASTDISPLAY_H
#define FORECASTDISPLAY_H

#include "observer.h"
#include "displayelement.h"

class WeatherData;

class ForecastDisplay : public Observer, public DisplayElement {
private:
    float currentPressure{29.92f};
    float lastPressure{29.92f};
    WeatherData* weatherData;
public:
    explicit ForecastDisplay(WeatherData* weatherData);
    void update()  override;
    void display() override;
};

#endif // FORECASTDISPLAY_H
```

Note the initial value `29.92f`, that's standard atmospheric pressure in inches of mercury, the U.S. unit for barometric pressure. The two pressure members let us compare "what pressure was last time we were told" against "what it is now". Implementation `forecastdisplay.cpp`:

```
#include "forecastdisplay.h"
#include "weatherdata.h"
#include <iostream>

ForecastDisplay::ForecastDisplay(WeatherData* weatherData)
    : weatherData(weatherData) {
    weatherData->registerObserver(this);
}

void ForecastDisplay::update() {
    lastPressure    = currentPressure;
    currentPressure = weatherData->getPressure();
    display();
}

void ForecastDisplay::display() {
    std::cout << "Forecast: ";
    if (currentPressure > lastPressure) {
        std::cout << "Improving weather on the way!\n";
    } else if (currentPressure == lastPressure) {
        std::cout << "More of the same\n";
    } else {
        std::cout << "Watch out for cooler, rainy weather\n";
    }
}
```

The pattern is identical to `CurrentConditionsDisplay`: constructor registers, `update()` pulls the relevant data and shuffles internal state, `display()` prints. Note that this observer reads only `getPressure()`; it doesn't care about temperature or humidity. Different observers, different data needs.

StatisticsDisplay

The statistics display is mildly more interesting: it accumulates a history of temperatures and reports running min/max / average. Header `statisticsdisplay.h`:

```
#ifndef STATISTICSDISPLAY_H
#define STATISTICSDISPLAY_H

#include "observer.h"
#include "displayelement.h"
#include <vector>

class WeatherData;
```

```
class StatisticsDisplay : public Observer, public DisplayElement {
private:
    std::vector<float> temperatureHistory;
    WeatherData* weatherData;

    float calculateAverage() const;
    float calculateMin()     const;
    float calculateMax()     const;
public:
    explicit StatisticsDisplay(WeatherData* weatherData);
    void update()  override;
    void display() override;
};
#endif // STATISTICSDISPLAY_H
```

The three calculator helpers (`calculateAverage`, `calculateMin`, `calculateMax`) are private and marked `const`. They are implementation details, not part of the public API. Implementation `statisticsdisplay.cpp`:

```
#include "statisticsdisplay.h"
#include "weatherdata.h"
#include <iostream>
#include <numeric>
#include <algorithm>

StatisticsDisplay::StatisticsDisplay(WeatherData* weatherData)
    : weatherData(weatherData) {
    weatherData->registerObserver(this);
}

void StatisticsDisplay::update() {
    temperatureHistory.push_back(weatherData->getTemperature());
    display();
}

float StatisticsDisplay::calculateAverage() const {
    if (temperatureHistory.empty()) return 0.0f;
    return std::accumulate(temperatureHistory.begin(),
                           temperatureHistory.end(),
                           0.0f) / temperatureHistory.size();
}

float StatisticsDisplay::calculateMin() const {
    if (temperatureHistory.empty()) return 0.0f;
    return *std::min_element(temperatureHistory.begin(),
                             temperatureHistory.end());
}
```

```
float StatisticsDisplay::calculateMax() const {
    if (temperatureHistory.empty()) return 0.0f;
    return *std::max_element(temperatureHistory.begin(),
                             temperatureHistory.end());
}

void StatisticsDisplay::display() {
    std::cout << "Avg/Max/Min temperature = "
              << calculateAverage() << "/"
              << calculateMax()     << "/"
              << calculateMin()     << "\n";
}
```

This is the chapter's first interaction with the STL algorithms from earlier in the course: `std::accumulate` sums the history (and we divide by size to get the mean), `std::min_element` and `std::max_element` each return an iterator to the extreme element (which we dereference with `*` to get the float). Each guard against the empty-history case with an `if (empty()) return 0.0f`, otherwise dividing by zero or dereferencing `end()` would be undefined behaviour.

ThirdPartyDisplay

And here is the entire point of the architecture, a display written by someone other than the original authors of Weather-O-Rama. Header `thirdpartydisplay.h`:

```
#ifndef THIRDPARTYDISPLAY_H
#define THIRDPARTYDISPLAY_H

#include "observer.h"
#include "displayelement.h"

class WeatherData;

class ThirdPartyDisplay : public Observer, public DisplayElement {
private:
    float temperature{0.0f};
    float humidity{0.0f};
    float pressure{0.0f};
    WeatherData* weatherData;
public:
    explicit ThirdPartyDisplay(WeatherData* weatherData);
    void update()  override;
    void display() override;
};

#endif // THIRDPARTYDISPLAY_H
```

Implementation `thirdpartydisplay.cpp`:

```
#include "thirdpartydisplay.h"
#include "weatherdata.h"
#include <iostream>

ThirdPartyDisplay::ThirdPartyDisplay(WeatherData* weatherData)
    : weatherData(weatherData) {
    weatherData->registerObserver(this);
}

void ThirdPartyDisplay::update() {
    this->temperature = weatherData->getTemperature();
    this->humidity    = weatherData->getHumidity();
    this->pressure    = weatherData->getPressure();
    display();
}

void ThirdPartyDisplay::display() {
    std::cout << "Third Party Display - Temp: " << temperature
              << "F, Humidity: " << humidity
              << "%, Pressure: " << pressure << " in\n";
}
```

Notice what is missing from this file: any modification to `WeatherData.h`, `WeatherData.cpp`, or any of the other displays. ThirdPartyDisplay sits beside the original code without disturbing it. This is what the Open–Closed Principle looks like when it works.

The driver

Finally, `main.cpp` wires up the system and drives a few measurements through it:

```
#include "weatherdata.h"
#include "currentconditionsdisplay.h"
#include "statisticsdisplay.h"
#include "forecastdisplay.h"
#include "thirdpartydisplay.h"

int main() {
    WeatherData weatherData;

    CurrentConditionsDisplay currentDisplay(&weatherData);
    StatisticsDisplay        statisticsDisplay(&weatherData);
    ForecastDisplay          forecastDisplay(&weatherData);
    ThirdPartyDisplay        thirdPartyDisplay(&weatherData);
```

```
    weatherData.setMeasurements(80, 65, 30.4f);
    weatherData.setMeasurements(82, 70, 29.2f);
    weatherData.setMeasurements(78, 90, 29.2f);

    return 0;
}
```

Each display's constructor registers it with `weatherData`. By the time the three `setMeasurements` calls fire, the subject has all four observers on its list. Each call to `setMeasurements` triggers a notification cascade: every observer's `update()` fires, which in turn calls its `display()`, which prints to the console.

Compile and run

From a command line:

```
$ g++ -std=c++17 weatherdata.cpp \
              currentconditionsdisplay.cpp \
              statisticsdisplay.cpp \
              forecastdisplay.cpp \
              thirdpartydisplay.cpp \
              main.cpp
$ ./a.out
Current conditions: 80F degrees and 65% humidity
Avg/Max/Min temperature = 80/80/80
Forecast: Improving weather on the way!
Third Party Display - Temp: 80F, Humidity: 65%, Pressure: 30.4 in
Current conditions: 82F degrees and 70% humidity
Avg/Max/Min temperature = 81/82/80
Forecast: Watch out for cooler, rainy weather
Third Party Display - Temp: 82F, Humidity: 70%, Pressure: 29.2 in
Current conditions: 78F degrees and 90% humidity
Avg/Max/Min temperature = 80/82/78
Forecast: More of the same
Third Party Display - Temp: 78F, Humidity: 90%, Pressure: 29.2 in
```

Three measurement sets, four observer reactions per set, twelve lines of output. Read down the first "set" of four lines: current conditions print first, then statistics, then forecast, then third-party, in the order the observers were constructed and registered. This is not guaranteed by the pattern; it is an artifact of the order in our particular `main.cpp`. Section 9 covers why you should never rely on it.

Push vs. Pull

Our `update()` method took no parameters. When notified, each observer reached back to the subject through the `weatherData` pointer and called the getters it needed. The lecture deck calls this the **pull** style: observers pull data from the subject when asked.

The alternative is to make `update()` take parameters carrying the changed data. Something like:

```
// push-style update
virtual void update(float temperature,
                    float humidity,
                    float pressure) = 0;

// and on the subject side
void WeatherData::notifyObservers() {
    for (auto* observer : observers) {
        observer->update(temperature, humidity, pressure);
    }
}
```

This is the **push** style, the subject pushes the changed data into every observer's `update()`. It has its appeal: there is no need for the observer to hold a pointer back to the subject, the data is right there, and one might argue the code is simpler.

So which is correct? The GoF book and the lecture both sides with pull, and with reason. Let us enumerate the trade-off.

Aspect	Push	Pull
Update signature	Carries the changed data as parameters	Empty: update()
Observer holds subject?	Not necessarily	Yes (pointer back)
What data observer gets	Whatever subject sends	Whatever it asks for
Adding a new state	Breaks all observer signatures	Observer just calls the new getter
Sub-setting data	Observer gets data, it ignores	Observer asks for what it needs
Does the subject know the observer's needs?	Yes (must know what to send)	No (observer chooses)

Aspect	Push	Pull
Best for	Small, fixed data set; no evolution	Most real-world scenarios

The third row is the killer. If WeatherData adds a fourth measurement next year, say, wind speed, every push-style observer's `update()` signature must change to take that fourth argument, even if the observer doesn't care about wind. With pull, observers that don't care about wind don't change; observers that do care add a `getWindSpeed()` call inside their existing `update()`.

That is why the deck says "pull is considered more correct". It scales with state evolution; push does not. There are corner cases where push is the better answer, for example, when notifications cross network boundaries and pulling would require an expensive callback, but for in-process observers, the default should be pull.

> **Tip**
>
> **Hybrid push-pull.** A common compromise is to push a *change-type tag* but not the data itself: `update(int eventType)`, where the type tells the observer *what kind* of change happened, and the observer pulls the actual data. This keeps the signature stable while letting observers skip notifications they don't care about. The deck doesn't cover this variant, but you will see it in production code.

Caveats and Pitfalls

The Observer Pattern is simple in concept, but it has a small zoo of pitfalls in practice. These are the ones the lecture deck calls out, plus a few you should know before shipping observer code into production.

Don't depend on notification order

Our implementation iterated `over observers in vector order, which happens to be insertion order, which in turn matches` the order constructors ran in `main.cpp`. You should regard that ordering as an *incidental* property of the implementation, not a contract.

If observer A's correct behaviour depends on running before observer B, you have introduced a coupling between A and B *through* the subject. That coupling is invisible from the code; neither A nor B mentions the other, but it is real, and it is fragile. If a future refactor changes the underlying container from

`std::vector` to `std::unordered_set`, or notifies in parallel on multiple threads, your code will break in ways that look like a heisenbug.

If you need ordering between observers, model it explicitly: have A notify B directly, or use a priority value in the registration, or use two separate notification phases. Do not rely on insertion order.

Object lifetime with raw pointers

We used `std::vector<Observer*>`, raw pointers. The subject does not own the observers; it just holds references to them. This means:

- If an observer is destroyed without being unregistered, the subject has a *dangling pointer* in its list. The next `notifyObservers` will call `update()` through a pointer to freed memory, undefined behaviour, usually a segfault.
- Conversely, if an observer outlives the subject and tries to read from it inside `update()` (e.g., because some other code called `update()` directly), it dereferences a dangling pointer.

The lecture's solution to this is discipline: every observer *must* unregister itself in its destructor. The robust solution is smart pointers. Three options:

- **std::weak_ptr<Observer>.** The subject holds weak pointers; before each `update()` call, it `lock()`s the weak pointer. If the lock fails, the observer is gone, skip it (and, conveniently, prune it from the list).
- **std::shared_ptr<Observer>.** The subject holds shared pointers, so it co-owns each observer. Simpler, but can create cycles, since most observers hold a pointer back to the subject.
- **RAII registration handle.** The `registerObserver` function returns a small handle object whose destructor calls `removeObserver`. Observers hold the handle for as long as they want to be subscribed; letting it go out of scope automatically unsubscribes.

> ⚠ **Warning**
>
> **Why does the lecture use raw pointers anyway?** Smart-pointer machinery would have added a page of complexity to an introductory exposition of the pattern. In a real codebase, especially one with non-trivial observer lifetimes, you should reach for one of the three smart-pointer options above. The pattern's *structure* doesn't change; only the ownership semantics do.

C++ has no built-in Observable

Java has `java.util.Observable` (deprecated since Java 9) and PropertyChangeSupport. C# has events and delegates. JavaScript has DOM events and listeners. C++ has nothing in the standard library. You roll your own, exactly as we did in this chapter.

In a Qt or Boost-using codebase, you may find ready-made implementations: Qt has its signal/slot mechanism (which is a thread-aware Observer with type-checked signatures), and Boost.Signals2 provides a generic version. For straight standard C++, the 25-line implementation we wrote is the canonical starting point.

Reentrant notification

We touched on this in Section 7's warning callout. If an observer's `update()` method, directly or transitively, calls `registerObserver` or `removeObserver` on the subject, you have re-entered the subject while its notification loop is still running. Iterators get invalidated, sets get mutated, and undefined behaviour ensues.

The simplest defensive pattern is to copy the observer list before iterating. This costs an extra heap allocation per notification but is safe even if observers register or unregister mid-notify. A common idiom:

```
void WeatherData::notifyObservers() {
    auto snapshot = observers;          // copy
    for (auto* o : snapshot) {
        o->update();                    // safe even if observers list mutates
    }
}
```

Thread safety

Our implementation is not thread-safe. If one thread is iterating `observers` inside `notifyObservers` while another thread calls `registerObserver`, the vector is mutated mid-iteration, and the program will crash or corrupt data. A multi-threaded weather station, where the sensor-reading thread calls `setMeasurements` while the UI thread registers and removes displays, must add a mutex around the registration and iteration code.

Thread-safe observer implementations also have to think about which thread the observer's `update()` runs on. If the subject's measurements thread calls it directly, the observer must be prepared to perform UI work from a non-UI thread (which is often forbidden by GUI frameworks). The typical approach is to dispatch notifications via a thread-safe queue and have observers pick them up on their preferred thread. This is well outside the scope of an introductory pattern, but it is the thing that bites first when you take Observer to production.

Observers in the Wild

One way to convince yourself that the Observer Pattern is fundamental is to spot it in software you already use.

Notifications and feeds

- **Push notifications.** Your phone is a giant observer attached to a server-side subject. When the server has news for you, every device that registered for that news gets pinged.
- **Twitter / X follow.** "Following" a user is the act of `registerObserver(me)` on that user's tweet stream. Unfollowing is `removeObserver(me)`. Every tweet they post is a `notifyObservers()`.
- **Facebook News Feed.** Your friends are subjects; you are the observer. The aggregator (the feed) combines all your subscriptions into a single timeline.
- **YouTube subscriptions.** Same pattern, different content. The channel is the subject; subscribers are observers.
- **Email newsletters.** Signing up subscribes you; unsubscribing is the legally-mandated `removeObserver`.
- **Group chat.** The chat room is the subject; every connected client is an observer; every message is a notification.

Live updates

- **Cricket/sports score apps.** The match is the subject; your phone app is an observer that updates the score widget on every notification.
- **Weather alerts.** Exactly our running example, except in production, the subject is on a server somewhere, and notifications cross the network.
- **Stock tickers.** Each stock symbol is a subject; trading platforms, portfolio trackers, and alert systems are observers. We will design a full one of these in the practice problem.
- **Auction bidding.** The auction lot is the subject; every bidder's UI is an observer. When a new bid arrives, every observer is told.

Devices and the IoT

- **Mobile push.** Already mentioned, but worth saying again: it is the largest deployment of the Observer Pattern in human history.
- **Android OS updates.** Your phone monitors Google's release subject and is notified when there is something to install.
- **Smart home.** Your thermostat is a subject; the heating system, your phone app, and the energy-tracking dashboard are observers. The motion sensor is a subject; the security camera, the lights, and the alarm are observers.

- **IoT state changes.** Generally: any sensor in a smart device is a subject, any actuator or display is an observer. The whole architecture of a modern smart-anything is observer chains.

E-commerce

- **"Notify me when in stock".** The product is a subject; your registration is an observer attached to its inventory state.
- **Price drop alerts.** The product price is a subject; the price-watcher service is an observer; you are an observer of the watcher (an observer chain).
- **Order status.** The order is a subject; the website, the mobile app, and the warehouse-management system are all observers, watching state transitions like "packed → shipped → out for delivery".

Software architecture

- **Model-View-Controller (MVC).** The Model is a subject; one or more Views are observers. When the Model changes, each View is notified and re-renders. This is one of the oldest applications of the pattern; Smalltalk-80 used Observer for exactly this in the late 1970s.
- **Game event systems.** Game state is a network of subjects; UI, AI, audio, and gameplay systems are observers. "Player took damage" is a notification fired to whichever subsystems care.
- **Logging and monitoring.** Every event bus, every metrics dashboard, every audit log is an observer attached to some subject inside the application.
- **Spreadsheets.** Each cell is a subject. Other cells that reference it (e.g., `=A1+B1`) are observers. When A1 changes, every dependent cell re-computes.

The shared pattern across all examples: one entity changes; an unknown number of other entities must *react*. Whenever you find that shape in your design, you have found a place for an Observer.

Summary

The Observer Pattern defines a one-to-many dependency between objects so that when one object, the **Subject**, changes state, all of its dependents, the **Observers**, are notified and updated automatically.

It is built from four roles arranged in two parallel interface hierarchies:

- **Subject** (interface): `registerObserver`, `removeObserver`, `notifyObservers`.
- **Observer** (interface): `update`.
- **ConcreteSubject**: a real class implementing Subject. Holds a list of Observer pointers and the application state.
- **ConcreteObserver**: a real class implementing Observer. Typically holds a back-pointer to the ConcreteSubject so it can pull state when notified.

The pattern's value comes from **loose coupling**:

- Subject doesn't know observer concrete classes; only the Observer interface.
- Observers can be added or removed at any time, including at runtime.
- Subject doesn't change when new observer types are introduced.
- Subject and observers can be reused independently of each other.
- Either side can change internally without breaking the other.

Things to remember when you implement it in C++:

- There is no `std::observable`, write the 25 lines yourself.
- Prefer the *pull* style: empty `update()`, observers reach back for what they need.
- Use `std::vector<Observer*>` plus polymorphism, or smart pointers if observer lifetimes are non-trivial.
- Don't rely on notification order; don't mutate the observer list during notification; think about threads before shipping.

Things to remember about the pattern's role in software:

- It is one of the most heavily used patterns in production, push notifications, social media feeds, MVC, spreadsheets, IoT, and games.
- Wherever "one thing changes, many things need to react", Observer is almost certainly the right starting point.
- It is loose coupling made concrete. Turn back to the previous chapter and re-read the design principles after working through this one. You will see them in a different light.

Practice Problems

Ten problems, escalating in difficulty. Problems 1-4 are warm-ups: identify subjects and observers, read code that uses the pattern, and walk through notification semantics. Problems 5-8 ask you to write or modify implementation code. Problems 9-10 are stretch problems, full design exercises with multiple parts. Stretch problems are marked with ★.

Problem 1.

For each of the following systems, identify the `Subject`, the `Observer(s)`, and what "update" would mean in concrete terms.

(a) A bank balance, plus a mobile app that shows the balance and a fraud-detection service that watches for anomalies.

(b) A YouTube channel and its 12 million subscribers.

(c) A spreadsheet cell `=A1+B1` with respect to cell `A1`.

(d) A button widget in a GUI library, with three handlers attached to its "clicked" event.

(e) A traffic light at an intersection, with a self-driving car's perception module watching it.

Problem 2.

Consider the following snippet, which builds an Observer-based system from scratch:

```
class Listener {
public:
    virtual ~Listener() = default;
    virtual void onChange() = 0;
};

class Counter {
    int value = 0;
    std::vector<Listener*> listeners;
public:
    void subscribe(Listener* l)   { listeners.push_back(l); }
    void unsubscribe(Listener* l) {
        listeners.erase(std::remove(listeners.begin(), listeners.end(), l),
                        listeners.end());
    }
    void increment() {
        ++value;
        for (auto* l : listeners) l->onChange();
    }
    int  get() const { return value; }
};

class Printer : public Listener {
    Counter* c;
public:
    explicit Printer(Counter* c) : c(c) { c->subscribe(this); }
    void onChange() override { std::cout << "value = " << c->get() << "\n"; }
};
```

(a) Map the actors in this code to the four roles in the Observer Pattern (Subject, Observer, ConcreteSubject, ConcreteObserver).

(b) Is this push or pull style? Justify your answer in one sentence.

(c) Identify two pitfalls in this code (compared to the version in this chapter). Hint: look at `Listener` and at `Printer`'s destructor.

(d) Trace the output of:

```
Counter c;
Printer p1{&c};
Printer p2{&c};
c.increment();
c.increment();
```

Problem 3.

The Weather Station in this chapter uses pull. Suppose we ship version 2 of WeatherData with a fourth measurement: wind speed.

(a) In the pull design, list every file that must be modified to add wind speed.

(b) In an alternative push design where `update(t,h,p)` carries the three measurements as parameters, list every file that must be modified.

(c) If wind speed is only relevant to one out of five observers, which design wastes more parameter-passing work?

(d) Is there any scenario where the push design wins? Think about (i) notifications crossing a network or thread boundary, and (ii) observers whose interface must be stable across versions.

Problem 4.

In the Weather Station, the StatisticsDisplay accumulates a history of temperatures. Suppose we add a fifth observer, MaxAlertDisplay, that prints a warning if the current temperature is more than two standard deviations above the historical mean, using `StatisticsDisplay`'s own state to compute the threshold.

(a) Why would this design be subtly broken if MaxAlertDisplay's `update()` runs before StatisticsDisplay's?

(b) Why is "just register StatisticsDisplay first" not a robust fix?

(c) Propose two structural fixes. (Hint: one of them involves NOT making MaxAlertDisplay observe WeatherData directly.)

Problem 5.

Starting from the Weather Station code in Section 7, implement a new observer `HeatIndexDisplay` that shows the heat index, a function of temperature and humidity that approximates how hot it feels:

```
heatIndex = 16.923 + 1.85212e-1 * t + 5.37941 * rh
          - 1.00254e-1 * t * rh + 9.41695e-3 * t * t
          + 7.28898e-3 * rh * rh + 3.45372e-4 * t * t * rh
          - 8.14971e-4 * t * rh * rh + 1.02102e-5 * t * t * rh * rh
          - 3.8646e-5 * t * t * t * t + 2.91583e-5 * rh * rh * rh
          + 1.42721e-6 * t * t * t * t * rh + 1.97483e-7 * t * rh * rh * rh
          - 2.18429e-8 * t * t * t * t * rh * rh
          + 8.43296e-10 * t * t * t * rh * rh * rh
          - 4.81975e-11 * t * t * t * t * rh * rh * rh;
// t in F, rh in % humidity
```

(a) Write the header file. It should implement both Observer and DisplayElement.

(b) Write the implementation file. Use the pull style.

(c) List the files in the original Weather Station that you had to modify in order to add this observer. (The list should be empty.)

(d) Add a line to `main.cpp` to construct a `HeatIndexDisplay` attached to the existing `weatherData`, and predict the new console output.

Problem 6.

Rewrite the `WeatherData::notifyObservers` method so that it is safe even if an observer calls `removeObserver` on the subject (including removing itself) during notification.

(a) Implement the "snapshot the list, iterate the snapshot" technique. Test it with a small main: construct two observers whose `update()` methods both call `the subject's removeObserver(this) method`.

(b) Without the fix, what is the observable misbehaviour of the original code in that scenario? Trace through the iterator arithmetic.

(c) Is the snapshot fix sufficient if an observer's `update()` *adds* a new observer (calls `registerObserver`) during notification? What happens, the new observer fires once, twice, or not at all? Justify.

Problem 7.

Rewrite the Subject/Observer interfaces and the WeatherData implementation to use `std::weak_ptr<Observer>` in place of `Observer*`, with observers held by their callers as `std::shared_ptr<Observer>`.

(a) What does the signature of `registerObserver` become?

(b) Modify `notifyObservers` to lock each weak_ptr; if the lock returns null (observer destroyed), skip it.

(c) As a bonus, prune dead weak_ptrs from the list inside `notifyObservers` while you're at it. Use `std::erase_if` (C++20) or `erase-remove` (older).

(d) What problem does this solve that the raw-pointer version had? What problem does this NOT solve?

Problem 8.

For each of the following code skeletons, decide whether it is an instance of the Observer Pattern (yes/no) and justify in one sentence:

(a)

```
class Document {
    std::vector<Listener*> listeners;
public:
    void addListener(Listener*);
    void textInserted(const std::string&);  // fires listener.onTextInserted
    void textDeleted(int from, int to);     // fires listener.onTextDeleted
};
```

(b)

```
class FileReader {
    std::ifstream file;
public:
    explicit FileReader(const std::string& path);
    std::string readLine();           // blocks until next line is available
    bool eof() const;
};
```

(c)

```
class GameLoop {
    Player player;
    Enemy enemy;
public:
    void tick() {
        player.update();
        enemy.update();
        renderer.draw(player);
        renderer.draw(enemy);
    }
};
```

(d)

```
class Button {
    using Handler = std::function<void()>;
    std::vector<Handler> onClick;
public:
    void addClickHandler(Handler h) { onClick.push_back(std::move(h)); }
    void click() { for (auto& h : onClick) h(); }
};
```

(e)

```
class Logger {
public:
    void debug(const std::string& msg);
    void info(const std::string& msg);
    void error(const std::string& msg);
};
```

Problem 9.

Suppose Weather-O-Rama's product manager comes back with a new requirement: each observer should be able to subscribe to a *subset* of the measurements. Some observers care only about temperature; some only about pressure; some about all three. The current `notifyObservers()` fires every observer on every measurement update, which is wasteful and forces every observer to filter what it cares about.

(a) Redesign the Subject interface to support per-topic subscription. Sketch the new signatures of `registerObserver`, `removeObserver`, and `notifyObservers`. Use an enum or string for "topic".

(b) Redesign WeatherData's internal data structure. The `std::vector<Observer*>` probably becomes a `std::map<Topic, std::vector<Observer*>>`. Discuss the trade-offs of that choice versus "one vector with a topic field on each entry".

(c) Implement `setMeasurements` so that it fires only the topics that actually changed. (Hint: compare new values to old.)

(d) Discuss: is this still the Observer Pattern? Has it become a different pattern (e.g., Publish-Subscribe), and if so, what is the difference?

Problem 10. ★

You're developing a Stock Trading Platform where multiple investors and trading bots need to react to real-time stock price changes. A `StockExchange` tracks prices for multiple stocks (AAPL, GOOGL, TSLA, etc.). Different types of observers need to react when stock prices update:

- `PortfolioTracker` monitors total portfolio value across multiple stocks.
- `PriceAlert` notifies users when a stock hits their target buy/sell price.
- `TradingBot` automatically executes trades based on price thresholds.
- `MarketDashboard` displays real-time price updates on a UI.

Requirements:

- Observers should only receive updates for stocks they're interested in (not all stocks).
- Multiple observers can watch the same stock.
- Observers can start/stop watching specific stocks dynamically.
- When a stock price changes, only observers watching that specific stock should be notified.
- The system should support adding new observer types (like Tax Calculator, News Feed) without modifying StockExchange.

(a) Draw a complete UML class diagram. Include the Subject interface, the Observer interface, StockExchange (the ConcreteSubject), Stock (a data class), and the four concrete observers. Use multiplicities, role names, and the open-triangle generalisation notation.

(b) Implement the system in C++. Use `std::unordered_map<std::string, std::vector<Observer*>>` inside StockExchange, keyed by stock symbol, so that `notifyObservers("AAPL")` notifies only AAPL's observers.

(c) Use the *push* variant for `update`: `update(const std::string& symbol, double newPrice, double oldPrice)`. Justify why push is appropriate *here*, and not in the Weather Station, by referencing the three arguments specifically.

(d) Draw a sequence diagram for: "PortfolioTracker attaches to AAPL; then someone calls `setStockPrice("AAPL", 150)`." Show the messages between PortfolioTracker, StockExchange, and the AAPL Stock object.

(e) Below is one possible UML solution (Figure 12.1) and one possible sequence diagram (Figure 12.2). Don't peek before you draft your own, but use these as a comparison after.

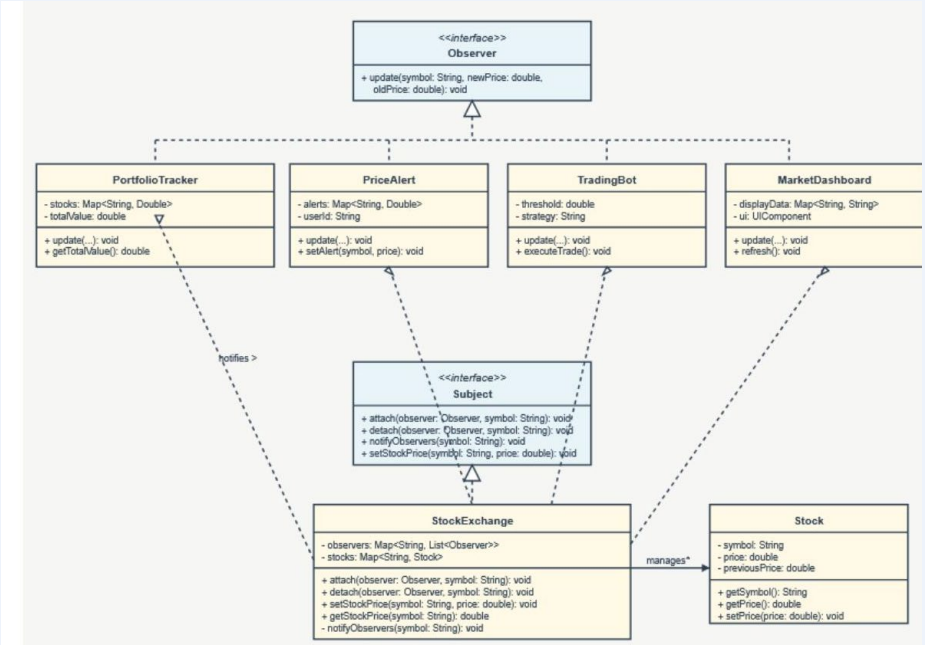

Figure 12.1: One possible UML solution for the Stock Trading Platform. Note the per-topic subscription structure inside StockExchange: observers are stored as a Map<String, List<Observer>> rather than a flat list.

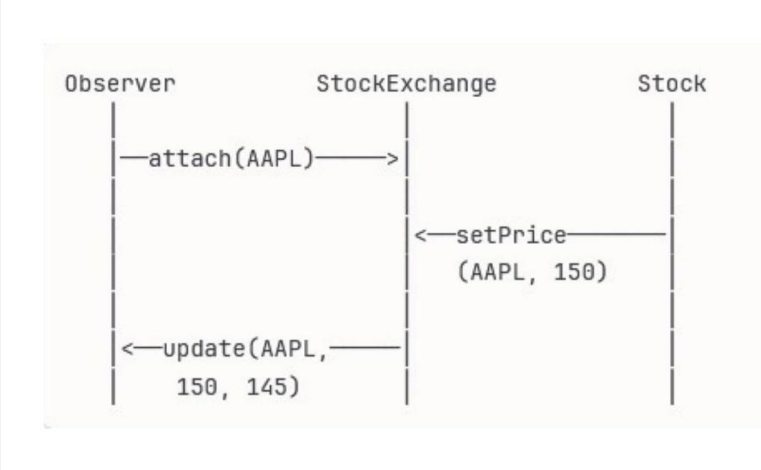

Figure 12.2: One possible interaction-flow sketch. Observer attaches to AAPL; later, someone calls setPrice on the Stock object; StockExchange picks that up and calls update on the registered observer with the new and old prices.