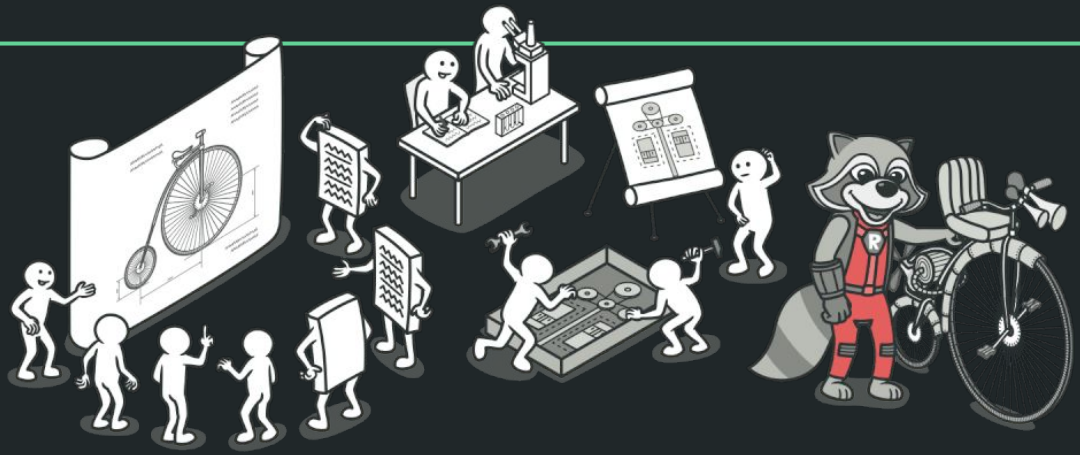
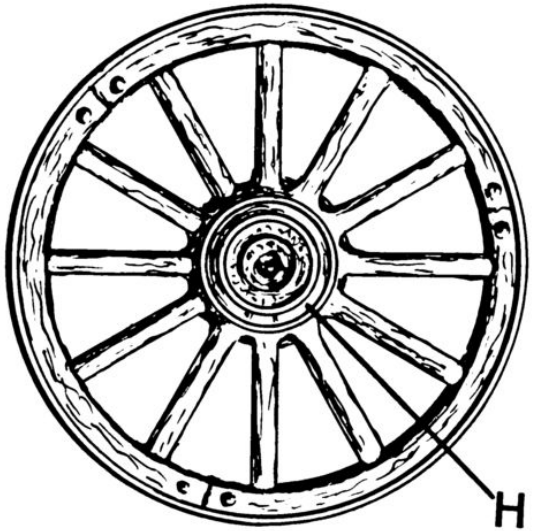
Patrones de diseño



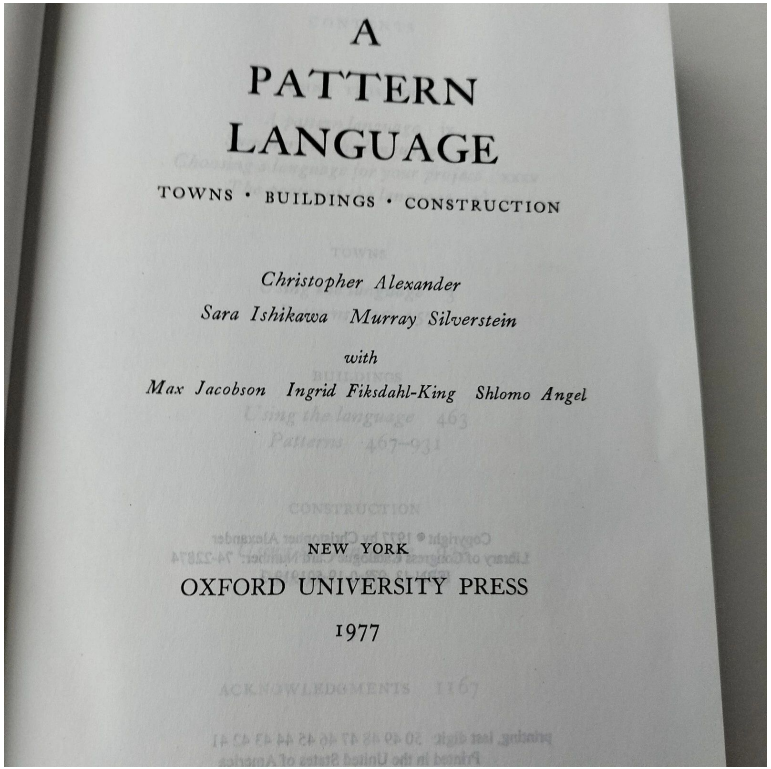
Reinventando la rueda

Reinventar la rueda es un intento de duplicar -lo más probablemente con resultados inferiores- un método básico que ya ha sido previamente creado u optimizado por otros.



Como la rueda ya trabaja como es esperado y no necesita más mejoras, reinventarla es probablemente una pérdida de tiempo y esfuerzo

Patrones de diseño: orígenes



- Lenguaje que surge de la arquitectura, para permitir a una persona hacer un diseño de cualquier clase de edificio.
- Los patrones son respuestas a problemas de diseño comunes (Cuán alta debe ser una ventana? Cuántos ambientes debe tener una casa?)
- **Cada patrón consiste de una descripción del problema, una discusión del problema con una ilustración, y una solución**
- Los patrones son arquetipos, de la esencia de la naturaleza del problema, que sirven hace 500 años, como hoy

Patrón de diseño

En vez de reutilizar *código*, con los patrones queremos reutilizar ***experiencia***.

La mejor forma de usar los patrones es *cargar nuestro cerebro* con ellos, y entonces *reconocer lugares* en nuestros diseños y aplicaciones donde podamos *aplicarlos*.

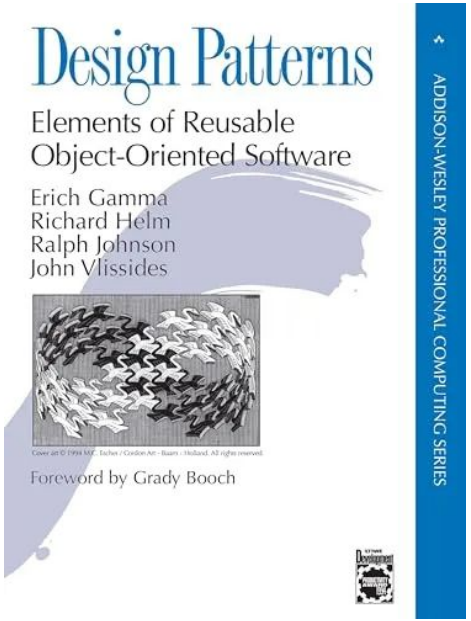
Los patrones son también un *lenguaje compartido* de comunicación en la industria de desarrollo de software.

Design Patterns

En el ámbito de la ingeniería de software, los patrones de diseño se presentan como soluciones probadas y reutilizables a problemas recurrentes que emergen durante el diseño de sistemas de software.

“Design Patterns: Elements of Reusable Object-Oriented Software” (1994) es el libro que da origen a la utilización de patrones en el desarrollo de software orientado a objetos. Los autores del libro son conocidos popularmente como "Gang of Four" (GoF): Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.

En su libro proporcionan un vocabulario común y plantillas para abordar desafíos específicos de diseño de software.

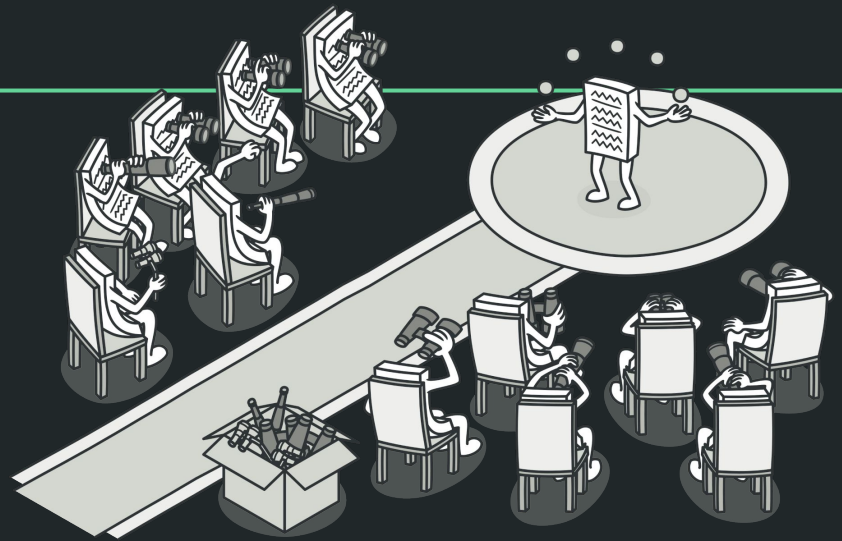


Clasificación de los patrones de diseño de software

GoF proponen en su libro 23 patrones de diseño. Se clasifican como:

- *Patrones creacionales*: Corresponden a patrones de diseño de software que solucionan problemas de creación de instancias. Nos ayudan a encapsular y abstraer dicha creación. En la materia vamos a ver el *Singleton* y el *Factory*.
- *Patrones estructurales*: Son los patrones de diseño de software que solucionan problemas de composición (agregación) de clases y objetos. En la materia vamos a ver el *Decorator*.
- *Patrones de comportamiento*: Se definen como patrones de diseño de software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan. En la materia vamos a ver el *Observer* y el *Iterator*.

El Patrón de Diseño Observer



Patrones de Diseño y Patrones de Comportamiento

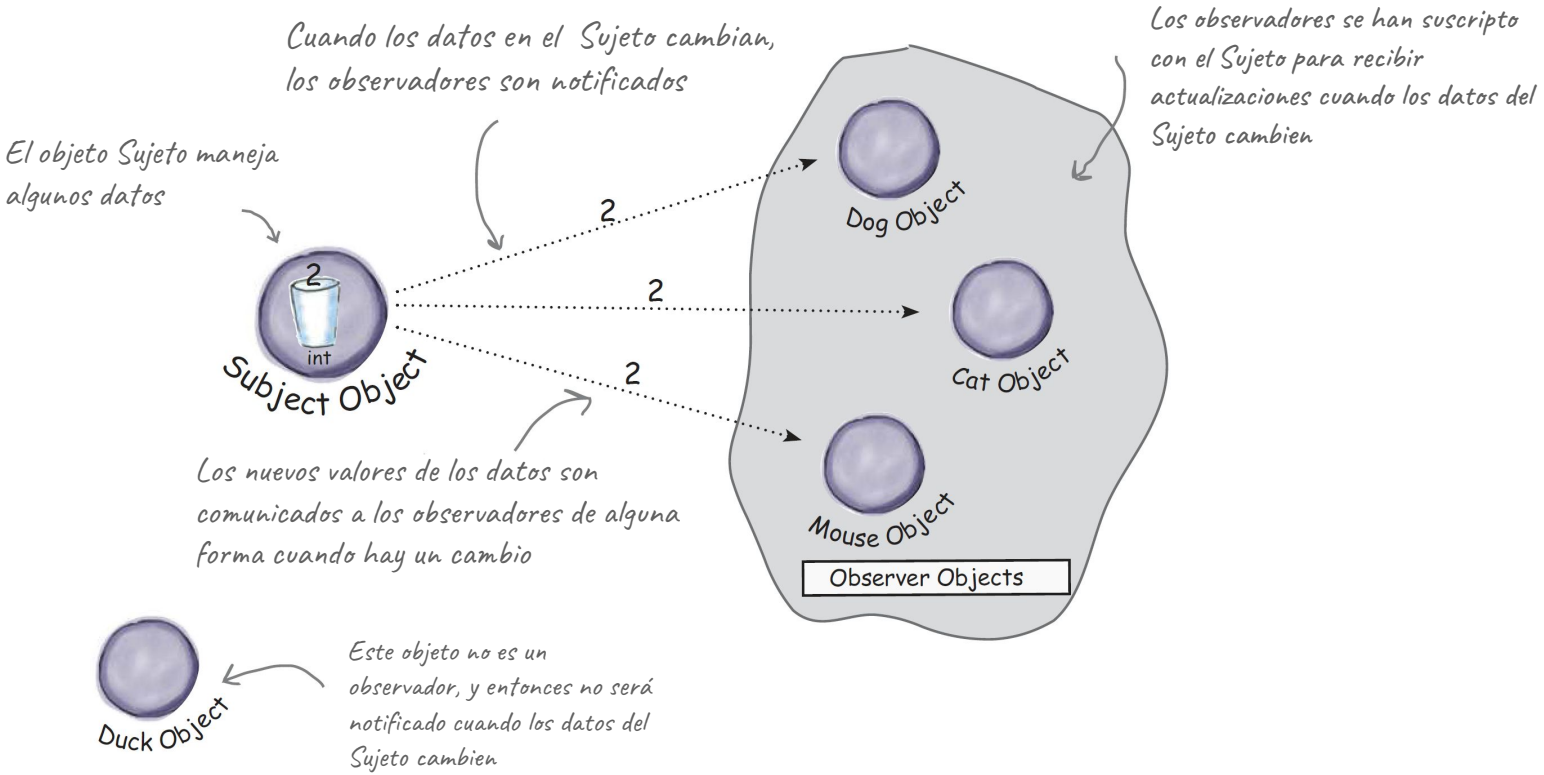
- El patrón *Observer* es un patrón de Comportamiento: estos patrones se centran en los algoritmos y la asignación de responsabilidades entre objetos, así como en la manera en que estos objetos interactúan y se comunican.
- Se busca gestionar las interacciones y responsabilidades de comunicación entre objetos, en lugar de enfocarse en la creación de objetos (patrones creacionales) o en la composición de clases o objetos (patrones estructurales).
- Observer se relaciona con cómo los objetos se comunican y colaboran dinámicamente.

Idea Central: La Analogía del Modelo de "Suscripción"

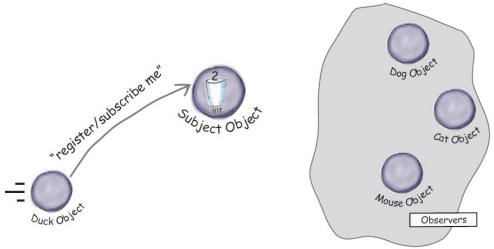
- **Suscripción a Periódicos/Revistas:** Un individuo se suscribe a una publicación (el "sujeto" o "publicador"). Cuando se publica un nuevo número, todos los suscriptores (observadores) reciben automáticamente su copia, sin que la editorial necesite conocer personalmente a cada uno de ellos o gestionar envíos individuales y específicos para cada persona.
- **Notificaciones y Feeds en Redes Sociales:** Los usuarios (observadores) eligen seguir o suscribirse a las actualizaciones de ciertas cuentas o perfiles (sujetos). Cuando estas cuentas publican nuevo contenido, los usuarios suscritos reciben notificaciones o ven las actualizaciones en sus feeds de forma automática, sin una interacción directa y explícita iniciada por el sujeto para cada seguidor en cada momento.
- **Emisión de Radio:** Una torre de radio (sujeto) emite una señal. Múltiples oyentes (observadores) pueden sintonizar esa frecuencia para recibir la transmisión. La torre no conoce la identidad ni la cantidad de oyentes; simplemente transmite, y quienes estén interesados y sintonizados reciben la señal.

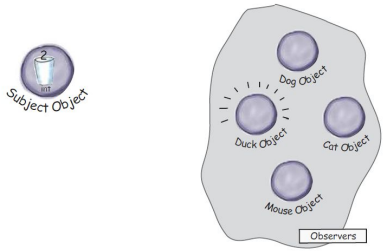
Es importante el concepto de una dependencia "uno a muchos". Un único sujeto puede tener múltiples observadores que dependen de su estado. Fundamentalmente, estas analogías ilustran el desacoplamiento inherente al patrón: el publicador o sujeto no necesita un conocimiento íntimo o referencias directas a las implementaciones concretas de sus suscriptores u observadores. Esta falta de conocimiento directo es una piedra angular del patrón, permitiendo flexibilidad y extensibilidad.

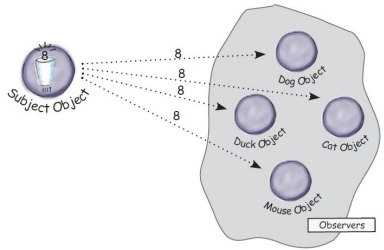
Publicadores + Suscriptores = Patrón Observer

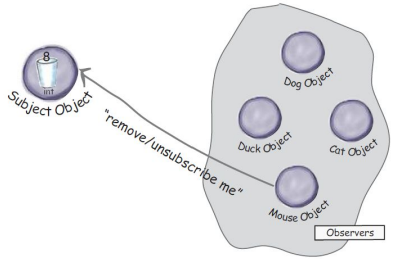


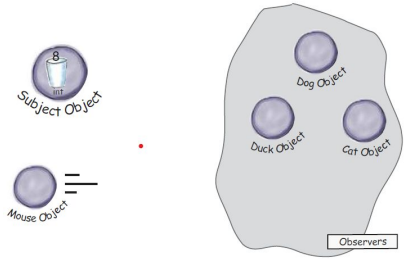
Un día en la vida del patrón Observer

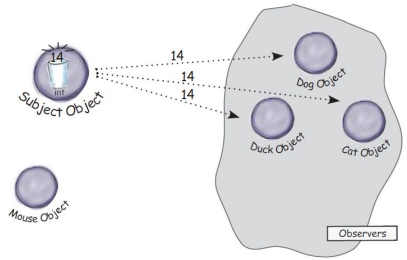












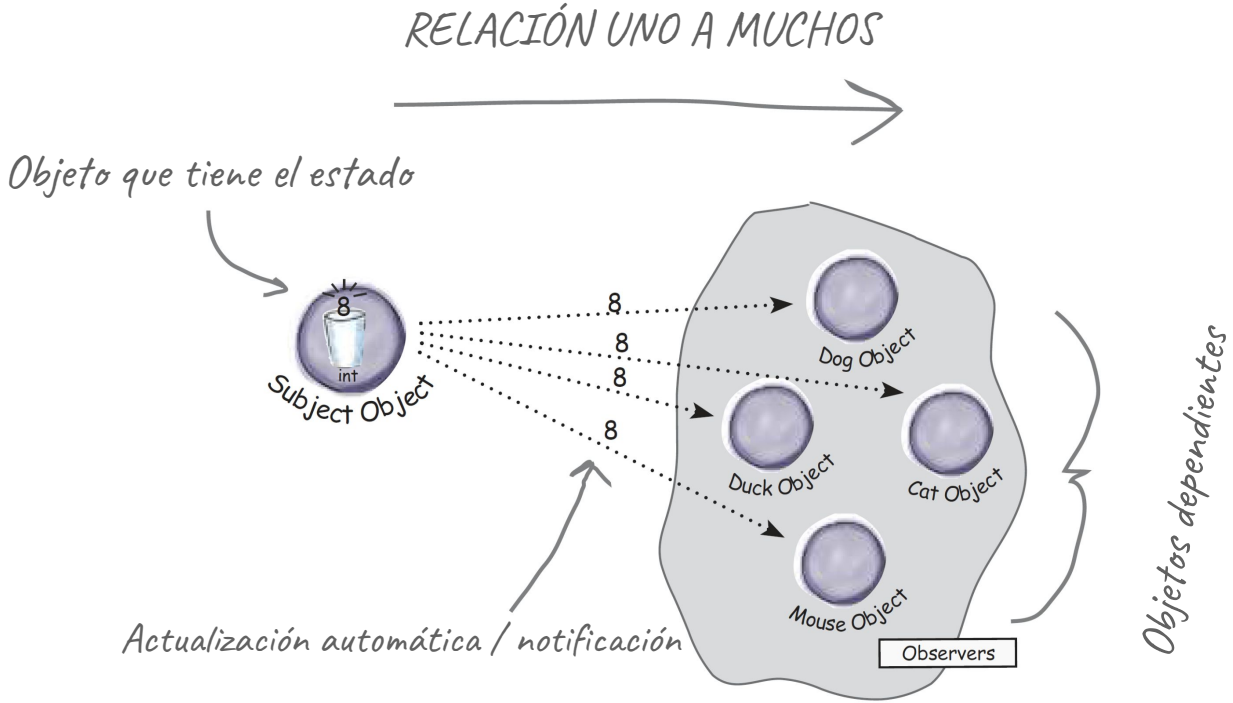
Definición Formal del Patrón Observer (GoF)

La definición canónica del patrón Observer, según el Gang of Four, establece su intención como: **"El Patrón Observer define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente"**.

Desglosando los términos clave de esta definición:

- **Dependencia uno a muchos:** Un único objeto (el sujeto) es observado por múltiples objetos (los observadores).
- **Cuando un objeto cambia de estado:** El sujeto experimenta una alteración en sus datos internos que es relevante para otros.
- **Todos sus dependientes son notificados:** Los observadores son informados del cambio.
- **Actualizados automáticamente:** Los observadores reaccionan al cambio, típicamente actualizando su propio estado o realizando alguna acción.

El Patrón Observer define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.



Planteo del Problema: Gestión de Dependencias de Estado

Cómo puedo mantener la consistencia del estado entre objetos relacionados sin introducir un acoplamiento estrecho entre ellos? La idea es considerar una alternativa donde un objeto "sujeto" (cuyo estado cambia) tuviera que conocer y llamar directamente a métodos de actualización en objetos dependientes específicos y codificados de forma fija. Este enfoque resulta inflexible por varias razones:

- Añadir o eliminar objetos dependientes requeriría modificar el código del sujeto.
- Cambios en la interfaz o implementación de los objetos dependientes podrían forzar cambios en el sujeto.
- El sujeto estaría fuertemente acoplado a las clases concretas de sus dependientes, limitando la reutilización y la evolución independiente de los componentes.

El patrón Observer ofrece una solución elegante donde el sujeto no necesita conocer las clases concretas de sus dependientes. En su lugar, interactúa con ellos a través de una interfaz abstracta, permitiendo que cualquier objeto que implemente dicha interfaz pueda ser notificado.

El poder del acoplamiento débil

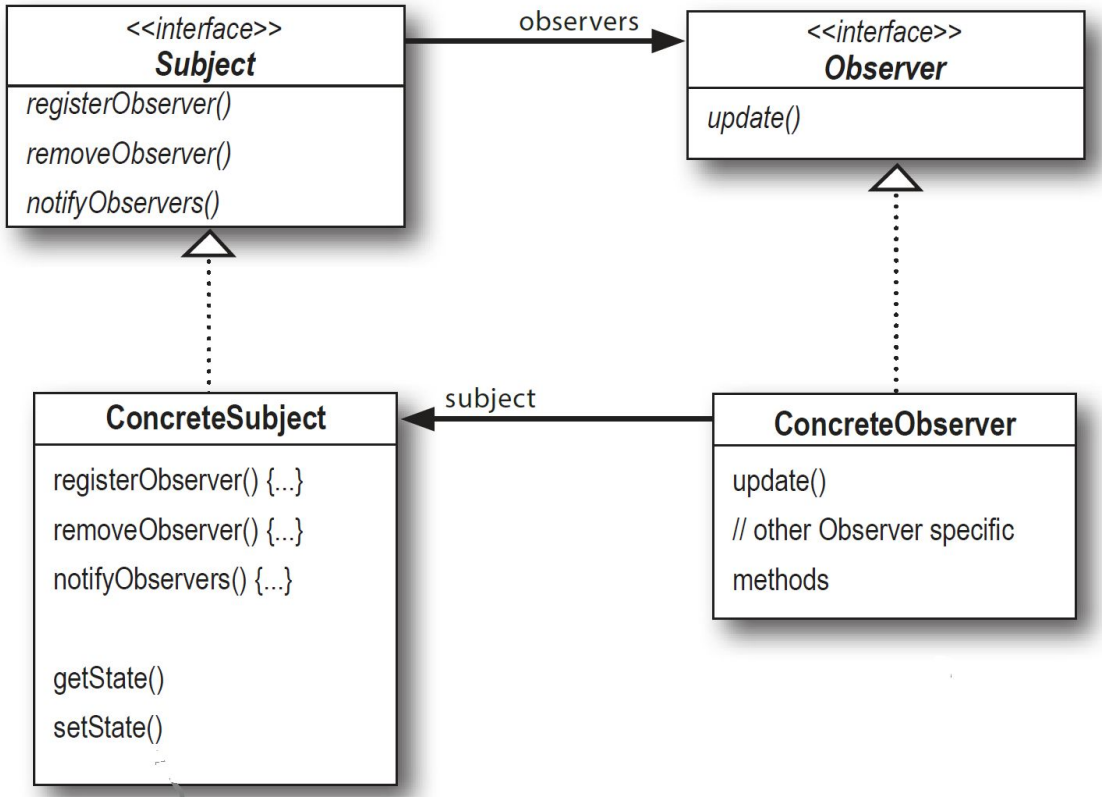
Cuando dos objetos están *débilmente acoplados*, ellos pueden interactuar, aún teniendo muy poco conocimiento del otro objeto.

Los diseños débilmente acoplados nos permiten construir sistemas flexibles ante el cambio ya que minimizan la interdependencia entre los objetos.

El patrón Observer provee un diseño donde los sujetos y los observadores están débilmente acoplados. Ventajas:

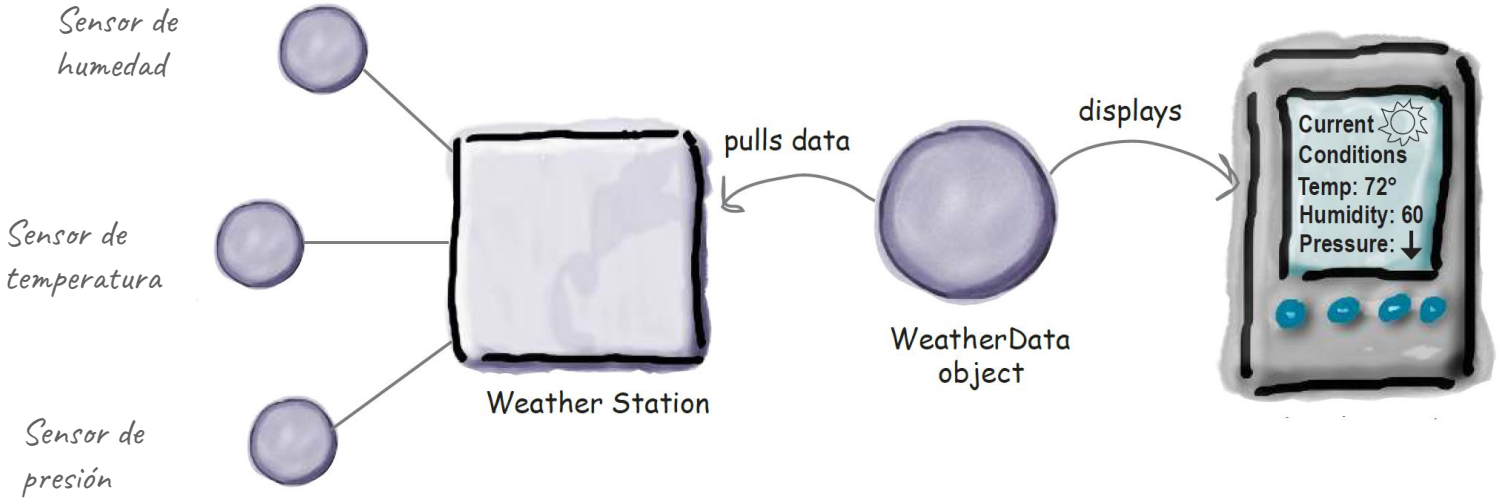
- La única cosa que el sujeto necesita saber del observador es que implementa una cierta interface (la interface Observer).
- Podemos agregar nuevos observadores en cualquier momento.
- No necesitamos modificar el sujeto para agregar nuevos tipos de observadores.
- Podemos reutilizar los sujetos o los observadores independientemente.
- Los cambios en el sujeto o algún observador no influyen al otro.

Patrón *Observer*: Diagrama de clases

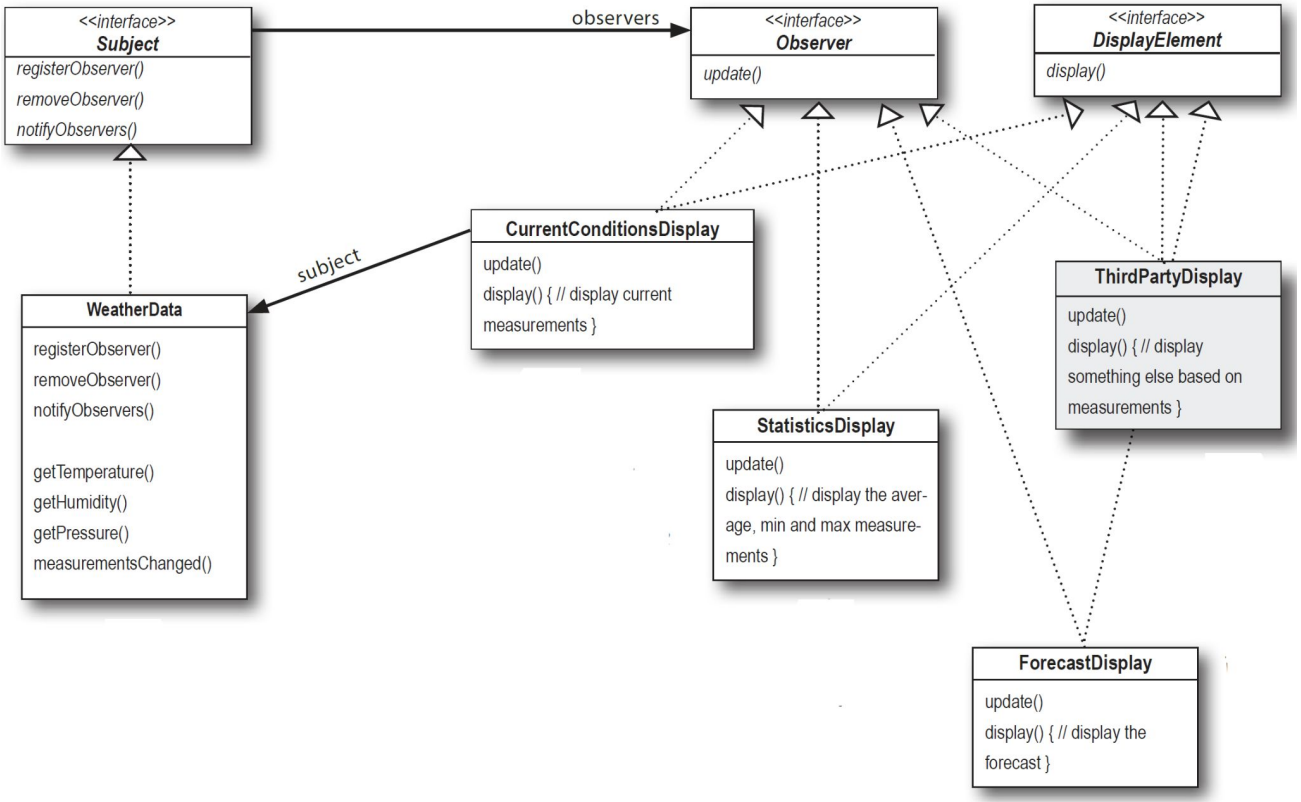


Problema de la estación meteorológica

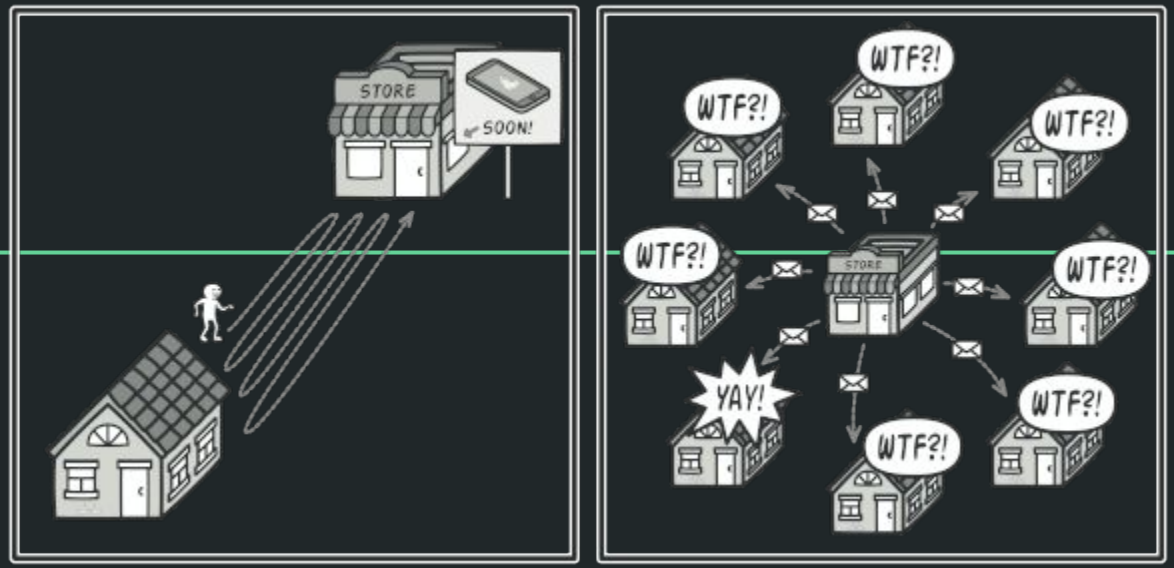
Una empresa que fabrica sensores para meteorología nos contrata para crear una app que use sus sensores y provea 3 displays distintos: condiciones de tiempo actual, pronósitoco del tiempo y estadísticas del tiempo. Además, nos piden hacer una API para que puedan vender el servicio de que los programadores puedan crear sus propios displays usando los datos de sus sensores.



Patrón Observer en estación meteorológica



Anatomía del Patrón Observer



Participantes Clave y Responsabilidades (parte 1)

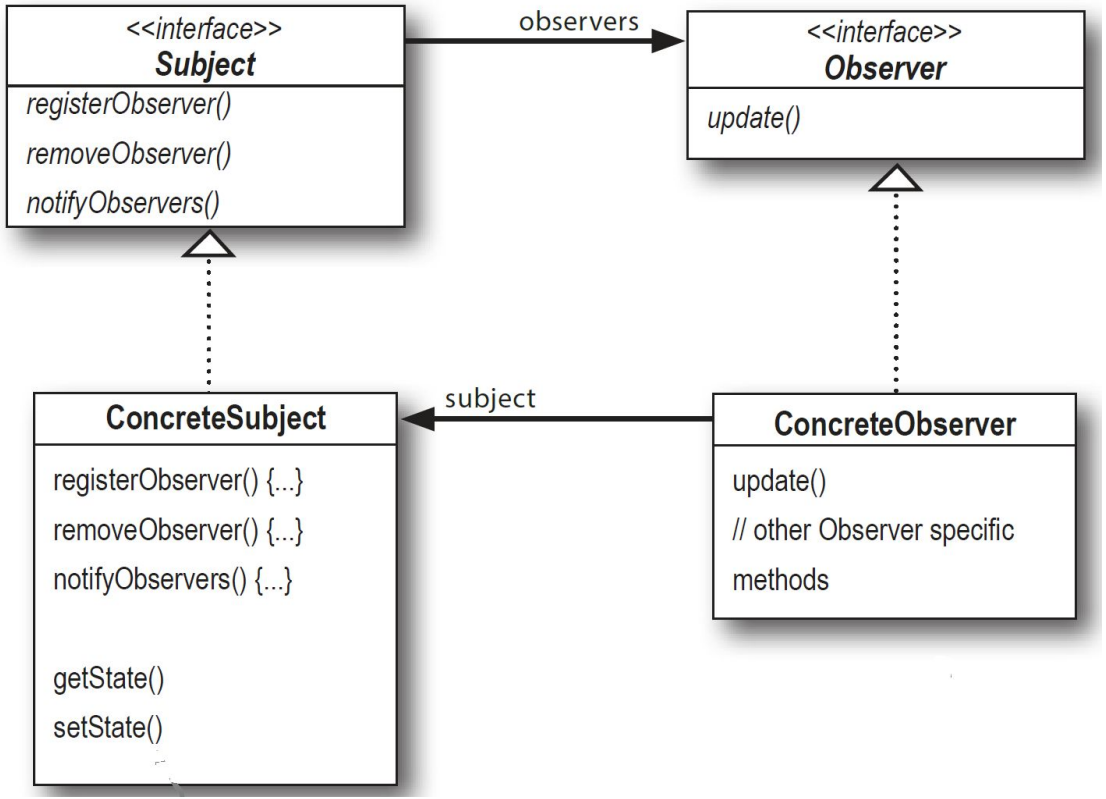
- **Interfaz *Subject* (Sujeto o Observable):**
  - Proporciona una interfaz para que los observadores se registren (`registerObserver`, `addObserver`, `attach`), se den de baja (`removeObserver`, `deleteObserver`, `detach`) y para notificarlos de las actualizaciones de estado.
  - No necesita conocer las clases concretas de sus observadores, sólo necesita que implementen la interfaz Observer.
- **Interfaz *Observer* (Observador o Suscriptor/Oyente):**
  - Define una interfaz de actualización (`update`, `notify`) para los objetos que deben ser notificados de los cambios en el estado de un sujeto.
  - Todos los observadores concretos implementan esta interfaz.
- **ConcreteSubject (Sujeto Concreto):**
  - Implementa la interfaz `Subject`.
  - Almacena el estado que es de interés para los objetos `ConcreteObserver`. Este es el estado cuyos cambios desencadenan notificaciones.
  - Envía una notificación a sus observadores cuando su estado cambia. Esto se logra típicamente llamando a un método (por ejemplo, `notifyObservers()`) que itera sobre la lista de observadores registrados y llama a su método `update()`.

Participantes Clave y Responsabilidades (parte 2)

- **ConcreteObserver (Observador Concreto):**
  i. Implementa la interfaz `Observer`.
  ii. Mantiene una referencia a un objeto `ConcreteSubject` (esta referencia es opcional pero a menudo necesaria, especialmente en el modelo de extracción o "pull", para que el observador pueda obtener el estado actualizado del sujeto).
  iii. Almacena el estado que debe mantenerse consistente con el estado del sujeto.
  iv. Implementa el método `update()` para reaccionar a las notificaciones del `ConcreteSubject`. La lógica dentro de `update()` define cómo el observador responde al cambio de estado del sujeto.

La separación de responsabilidades en interfaces (`Subject`, `Observer`) y sus implementaciones concretas (`ConcreteSubject`, `ConcreteObserver`) es un aspecto fundamental. Esta distinción es lo que permite la flexibilidad y el bajo acoplamiento que caracterizan al patrón. Las interfaces definen el *contrato* para la interacción: el `Subject` define cómo gestionar y notificar a los observadores, y el `Observer` define cómo recibir actualizaciones. Las clases concretas, por otro lado, proporcionan el *comportamiento específico* y el estado. Esta separación permite que diferentes sujetos concretos y observadores concretos puedan combinarse y reutilizarse de forma independiente, siempre que se adhieran a las interfaces definidas.

Patrón *Observer*: Diagrama de clases



Estructura: Diagrama de Clases UML

- Una interfaz o clase abstracta `Subject` con métodos como `attach(Observer o)`, `detach(Observer o)` y `notify()`.
- Una interfaz o clase abstracta `Observer` con un método `update()`.
- Una clase `ConcreteSubject` que hereda o implementa `Subject`. Esta clase tendría atributos para almacenar su estado y una colección (por ejemplo, una lista) de objetos `Observer`.
- Una o más clases `ConcreteObserver` que heredan o implementan `Observer`. Cada `ConcreteObserver` podría tener una referencia al `ConcreteSubject` que observa.

Las relaciones clave visualizadas son:

- **Generalización/Implementación:** `ConcreteSubject` implementa `Subject`, y `ConcreteObserver` implementa `Observer`.
- **Asociación:** El `Subject` (o `ConcreteSubject`) tiene una asociación de multiplicidad "uno a muchos" con la interfaz `Observer`. Esto significa que un sujeto puede estar asociado con cero o más observadores. El sujeto mantiene una colección de referencias a objetos `Observer`.
- **Dependencia (opcional):** Un `ConcreteObserver` puede tener una dependencia de `ConcreteSubject` si necesita solicitarle datos (modelo pull).

Este diagrama de clases UML refuerza visualmente el concepto de desacoplamiento. El `Subject` interactúa con sus observadores únicamente a través de la interfaz `Observer`, sin conocer las clases concretas de estos. La colección de observadores en el `Subject` es de tipo `Observer`, no de tipos específicos como `ConcreteObserverA` o `ConcreteObserverB`. Esta representación visual hace más tangible el principio de "programar para una interfaz, no para una implementación", que es central para lograr el bajo acoplamiento.

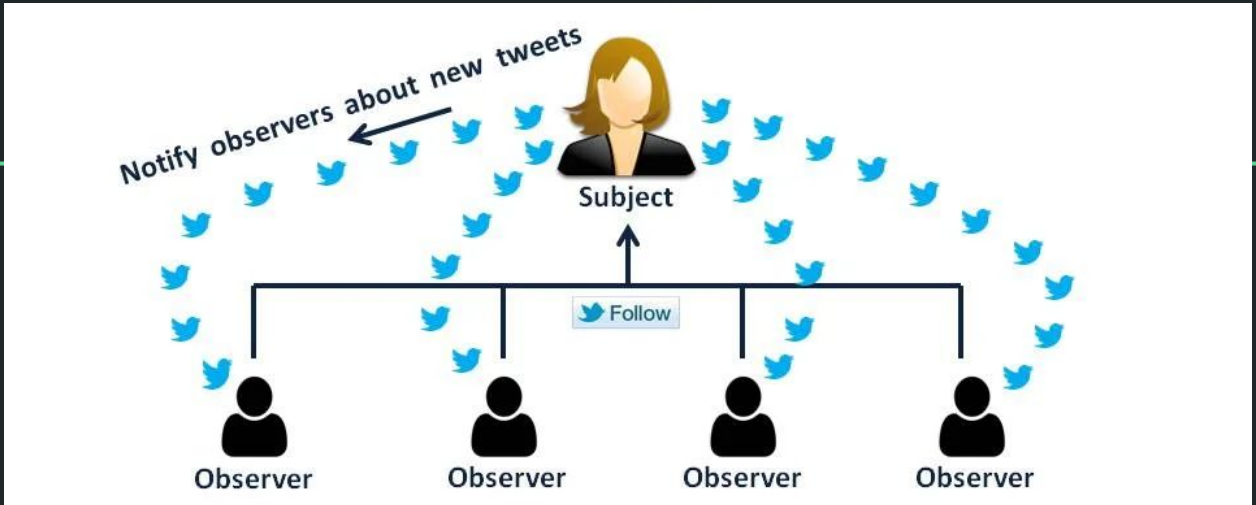
Principio Fundamental Profundizado: Logro del Bajo Acoplamiento

El bajo acoplamiento es el beneficio estratégico principal que ofrece el patrón Observer. Se logra a través de varios mecanismos:

- **Interacción a través de Interfaces:** El Sujeto conoce a sus observadores sólo a través de la interfaz `Observer`. No depende de las clases concretas de los observadores.
- **Adición/Eliminación Dinámica:** Los observadores pueden ser añadidos o eliminados en tiempo de ejecución sin necesidad de modificar el código del sujeto. Esto es posible porque el sujeto simplemente gestiona una lista de objetos que implementan la interfaz `Observer`.
- **Evolución Independiente:** Los cambios en la implementación de los observadores o del sujeto pueden realizarse de forma independiente, siempre y cuando se respeten las interfaces definidas. Un nuevo tipo de observador puede introducirse sin afectar al sujeto ni a otros observadores.
- **Capas de Abstracción:** El patrón permite que el sujeto y los observadores pertenezcan a diferentes capas de abstracción dentro de un sistema. Un sujeto de bajo nivel puede comunicar e informar a un observador de alto nivel, manteniendo intacta la estructuración en capas del sistema.

Este bajo acoplamiento no es meramente un detalle técnico; es una ventaja arquitectónica fundamental que se traduce directamente en una mayor mantenibilidad, reutilización y escalabilidad del software. Cuando los componentes están débilmente acoplados, los cambios en una parte del sistema tienen menos probabilidades de propagarse y causar efectos no deseados en otras partes. Esto simplifica el mantenimiento, ya que las modificaciones tienden a estar localizadas. Los componentes se vuelven más reutilizables porque no tienen dependencias estrictas de otros módulos específicos. Además, el sistema es más escalable, ya que se pueden agregar nuevos observadores con un impacto mínimo en el código existente. Esta conexión entre el mecanismo técnico (el uso de interfaces y el protocolo de notificación) y los atributos de calidad del software de alto nivel (mantenibilidad, reutilización) subraya la importancia estratégica del patrón.

Cómo Funciona el Patrón Observer: Mecanismo



El Flujo de Interacción (Secuencia) (parte 1)

La interacción típica entre los participantes del patrón Observer sigue una secuencia bien definida, que permite la comunicación desacoplada ante cambios de estado:

1. **Registro del Observador:** Un objeto `ConcreteObserver` se registra con un objeto `ConcreteSubject` invocando un método de suscripción (por ejemplo, `attach(this)` o `addObserver(this)`) proporcionado por la interfaz `Subject`. El `ConcreteSubject` añade este observador a su lista interna de observadores.
2. **Cambio de Estado del Sujeto:** El estado del `ConcreteSubject` cambia. Esto puede ocurrir debido a una operación interna, una interacción del usuario o cualquier otro evento que modifique los datos que el sujeto gestiona y que son de interés para los observadores. A menudo, esto se realiza a través de un método específico, como `setState(newState)`.
3. **Notificación a los Observadores por parte del Sujeto:** Inmediatamente después de que su estado cambia (o en un momento apropiado según la lógica de la aplicación), el `ConcreteSubject` invoca su propio método de notificación (por ejemplo, `notify()` o `notifyObservers()`).
4. **Iteración y Llamada a `update()`:** El método de notificación del `ConcreteSubject` itera sobre su lista de observadores registrados. Por cada `Observer` en la lista, llama a su método `update()`.

El Flujo de Interacción (Secuencia) (parte 2)

5. **Actualización del Observador:** El método `update()` de cada `ConcreteObserver` se ejecuta. Dentro de este método, el observador implementa la lógica para reaccionar al cambio de estado del sujeto. Esta lógica puede implicar solicitar más datos al sujeto (modelo pull), utilizar los datos pasados como argumentos (modelo push), actualizar su propio estado interno, refrescar una interfaz de usuario, etc.
6. **Desregistro del Observador (Opcional):** En algún momento, un `ConcreteObserver` puede decidir que ya no está interesado en recibir notificaciones del `ConcreteSubject`. En tal caso, puede darse de baja llamando a un método de desuscripción (por ejemplo, `detach(this)` o `removeObserver(this)`) en el `ConcreteSubject`. El sujeto entonces lo elimina de su lista de observadores.

Este flujo asegura que los observadores sean informados de manera consistente y automática sobre los cambios relevantes en el sujeto, sin que el sujeto necesite conocer los detalles específicos de cada observador. Un diagrama de secuencia simple podría visualizar este proceso, mostrando las llamadas a métodos entre las instancias de `ConcreteObserver`, `ConcreteSubject`, y las interfaces `Observer` y `Subject`.

Modelos de Notificación: Push vs. Pull (parte 1)

Existen dos modelos principales mediante los cuales un sujeto puede transmitir información a sus observadores durante la notificación. La elección entre estos modelos representa una decisión de diseño importante con implicaciones en el acoplamiento, la eficiencia y la complejidad.

- **Modelo de Empuje (Push):**
    - En este modelo, el sujeto envía información detallada sobre el cambio (a menudo, el nuevo estado completo o los datos específicos que cambiaron) al observador como parte de la llamada al método `update()`. Por ejemplo, el método podría ser `update(Object newState)` o `update(String propertyName, Object newValue)`.
    - **Ventajas:**
        - Los observadores tienen los datos necesarios inmediatamente, sin necesidad de realizar consultas adicionales al sujeto. Esto puede ser más eficiente si la mayoría de los observadores necesitan la misma información.
        - Puede simplificar la lógica del observador, ya que no necesita gestionar una referencia al sujeto para obtener datos.
    - **Desventajas:**
        - El sujeto podría enviar datos innecesarios a observadores que solo están interesados en una parte del estado o en un aspecto diferente del cambio. Esto puede ser ineficiente y consumir más ancho de banda o recursos de los necesarios.
        - El sujeto necesita tener una idea de qué información podrían necesitar los observadores, lo que podría acoplarlo más a las expectativas de estos. Si los observadores tienen necesidades muy diversas, el sujeto podría volverse complejo al intentar satisfacer a todos.

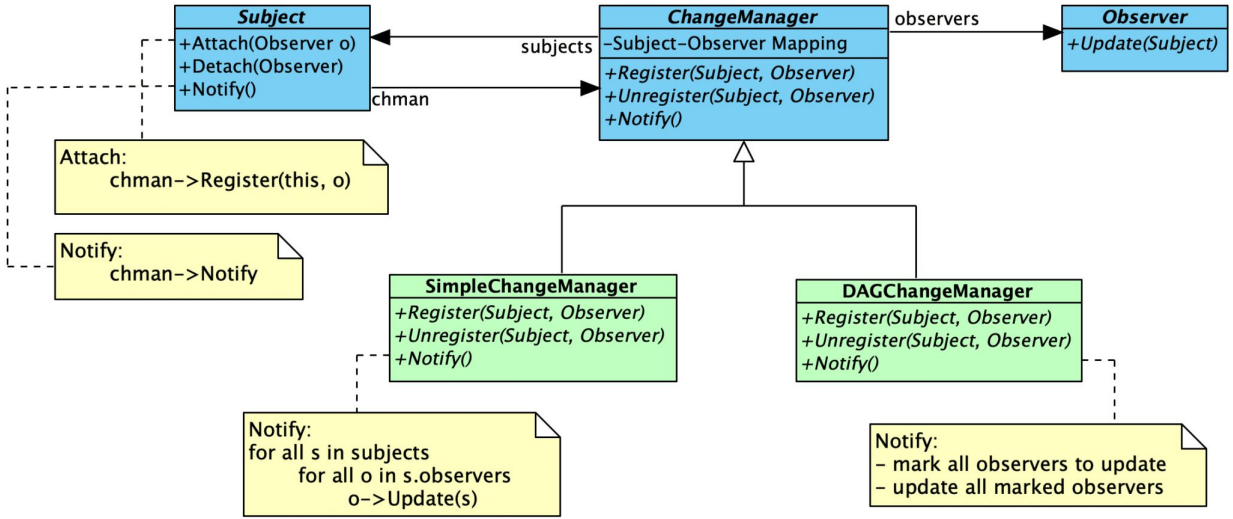
Modelos de Notificación: Push vs. Pull (parte 2)

- **Modelo de Extracción (Pull):**
  - En este modelo, el sujeto simplemente notifica al observador que *algo* ha cambiado, sin enviar detalles específicos del cambio en la llamada a `update()`. La firma del método `update()` podría ser tan simple como `update()` o `update(Subject changedSubject)`.
  - Una vez notificado, el observador es responsable de "extraer" (solicitar) los datos que necesita del sujeto. Para ello, el observador generalmente necesita mantener una referencia al sujeto y llamar a sus métodos `getter` (por ejemplo, `subject.getState()` o `subject.getPropertyValue("someProperty")`).
  - **Ventajas:**
    - Los observadores sólo obtienen los datos que específicamente necesitan, evitando la transferencia de información superflua.
    - El sujeto no necesita conocer las necesidades de datos de cada observador, lo que puede llevar a un diseño de sujeto más simple y genérico.
    - Mayor flexibilidad para los observadores, ya que pueden decidir qué información obtener y cuándo.
  - **Desventajas:**
    - Puede ser menos eficiente si muchos observadores necesitan realizar múltiples llamadas al sujeto para obtener la información requerida, generando una comunicación más "conversacional" (chatty).
    - El observador necesita mantener una referencia al sujeto y conocer su interfaz de consulta (métodos `getter`), lo que puede introducir un ligero acoplamiento si no se gestiona cuidadosamente.
    - Si el sujeto cambia significativamente entre la notificación y el momento en que el observador extrae los datos (especialmente en entornos concurrentes), el observador podría obtener un estado inconsistente o desactualizado, aunque esto es más un problema de concurrencia que del modelo pull en sí.

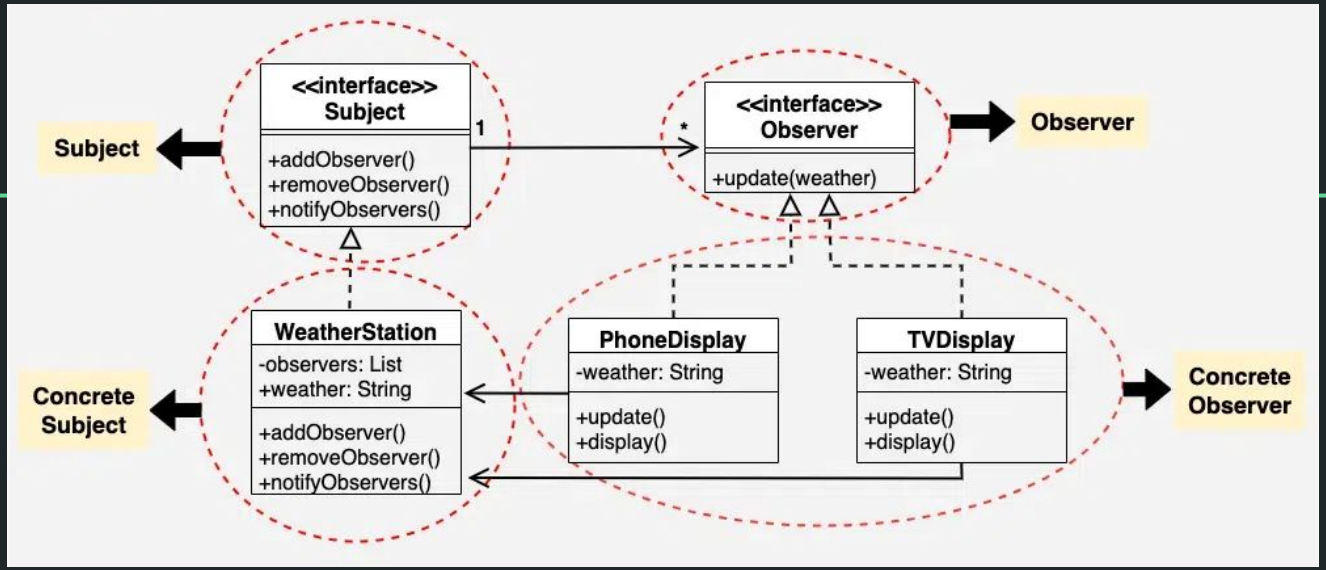
También existen **enfoques híbridos**, donde el sujeto podría "empujar" una cantidad mínima de información esencial (por ejemplo, el tipo de cambio ocurrido) y permitir que los observadores "extraigan" detalles adicionales si los necesitan. La elección entre push y pull depende de factores como la cantidad de estado que cambia, la diversidad de las necesidades de los observadores, las consideraciones de rendimiento y el nivel deseado de acoplamiento. No es una elección trivial, sino una compensación de diseño significativa.

Variaciones del Patrón Observer

Una variación común implica el uso de un `ChangeManager` o un objeto mediador centralizado. En esta variante, tanto los sujetos como los observadores se registran con este gestor central. Los sujetos notifican al gestor de los cambios, y el gestor es responsable de notificar a los observadores apropiados. Esta aproximación puede ayudar a desacoplar aún más a los sujetos de los observadores y puede ser útil en sistemas más complejos donde la gestión directa de las dependencias por parte de cada sujeto podría volverse engorrosa. Esta variante comienza a difuminar las líneas con el patrón Publish-Subscribe, que se discutirá más adelante, ya que el `ChangeManager` actúa de manera similar a un "broker" de eventos.



Perspectivas de Implementación y Ejemplo de Código



Repositorio de Ejemplo en Python

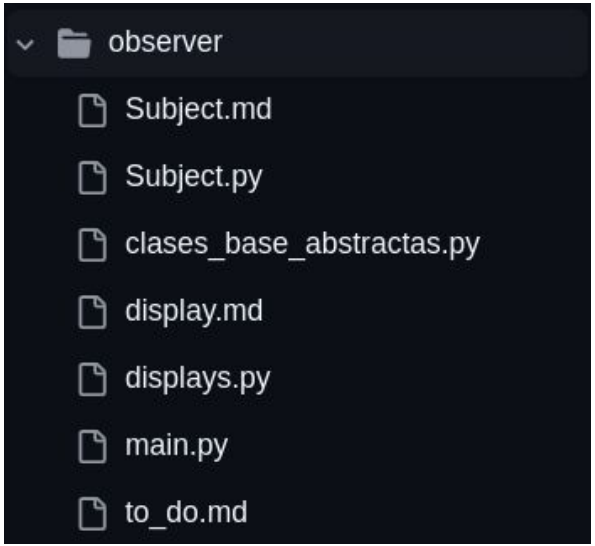
Hemos preparado un repositorio con una implementación del Patrón Observer en Python, similar al ejemplo de la Estación Meteorológica de 'Head First Design Patterns'

Repositorio

Estructura del Repositorio (lista simple):

- `clases_base_abstractas.py`: Definición de `Subject` y `Observer`.
- `Subject.py`: Clase `WeatherData` (Sujeto Concreto).
- `displays.py`: Clases `CurrentConditionsDisplay`, `StatisticsDisplay`, `ForecastDisplay` (Observadores Concretos).
- `main.py`: Script para ejecutar el ejemplo.
- `TO_DO.md`: Contiene un ejercicio práctico.

El código actual implementa el **modelo Push**.



Clases Base Abstractas

Definiendo Contratos: `Subject` y `Observer`
Archivo a revisar en el repositorio: **`clases_base_abstractas.py`**
Puntos clave a observar en el archivo:

- Uso de `ABC` y `@abstractmethod` para definir `Subject` y `Observer`.
- Métodos abstractos en `Subject`: `register_observer`, `remove_observer`, `notify_observers`.
- Método abstracto en `Observer`: `update(self, temperature: float, humidity: float, pressure: float)`.

Pregunta para los alumnos: *¿Por qué la firma actual de `Observer.update()` es característica del modelo Push?*

Sujeto Concreto

El Corazón del Sistema: `WeatherData`
Archivo a revisar en el repositorio: **`Subject.py`**
Puntos clave a observar en la clase `WeatherData`:

- Lista `_observers` para almacenar los observadores.
- Atributos para `_temperature`, `_humidity`, `_pressure`.
- Implementación de `register_observer`, `remove_observer`.
- Funcionamiento de `notify_observers` (iterando y llamando a `observer.update()` con los datos).
- Métodos `set_measurements` y `measurements_changed`.
- (Opcional: `get_temperature`, `get_humidity`, `get_pressure` para el futuro modelo Pull).

Discutir: *¿Cómo `WeatherData` gestiona su estado y notifica a los observadores en el modelo Push actual?* (Referenciar `Subject.md` si se desea).

Observadores Concretos

Reaccionando a los Cambios: Los Displays
Archivo a revisar en el repositorio: **`displays.py`**
Puntos clave a observar (ejemplificar con `CurrentConditionsDisplay` y mencionar que los otros son similares):

- Herencia de `Observer` y `DisplayElement`.
- Registro en el `__init__` con `weather_data.register_observer(self)`.
- Referencia `_weather_data` almacenada.
- Método `update` que recibe los datos y luego llama a `self.display()`.

Discutir: *¿Qué responsabilidad tiene cada observador en el método `update` en el modelo Push?*
(Referenciar `display.md` si se desea).

Ejecución del Ejemplo

Poniendo Todo en Marcha
Archivo a revisar/ejecutar: **`main.py`**
Explicar brevemente la secuencia en `main()`:

1. Creación de `WeatherData`.
2. Creación e instanciación de los displays (que se auto-registran).
3. Llamadas a `weather_data.set_measurements()` para simular cambios.

Mostrar la salida esperada (como la tienes en el `TO_DO.md`).
Animar a los alumnos a ejecutarlo ellos mismos.

¡Ejercicio Práctico! Refactorización al Modelo Pull

🚀 ¡A Programar! Ejercicio: Modelo Pull

Modificar la implementación actual (modelo Push) para convertirla en un modelo Pull, donde los observadores solicitan activamente los datos que necesitan del sujeto.
Debemos mencionar las 3 tareas principales descritas en `TO_DO.md` (modificar `Observer.update`, `WeatherData.notify_observers`, y los `update` de los ConcreteObservers).
Indicar que las instrucciones detalladas y los fragmentos de código modificados están en `TO_DO.md`.

Es recomendable recordar visualmente los cambios conceptuales (quizás con un diagrama simple o iconos):

- `Observer.update(self)`: Ya no recibe datos.
- `WeatherData.notify_observers()`: Llama a `update()` sin datos.
- Observadores Concretos: Usan `self._weather_data.get_...()` dentro de su `update()`.

Discusión y siguientes pasos

Remitir a las preguntas de reflexión del `TO_DO.md`.

- Ventajas/desventajas Push vs. Pull.
- Escenarios de uso para cada modelo.
- Acoplamiento y referencia al sujeto en modelo Pull.
- Eficiencia selectiva de datos.

Se puede continuar completando el ejercicio del `TO_DO.md`, y experimentando:

- Añadiendo un nuevo display (Observador Concreto).
- Modificando qué datos "jala" un display existente.

Objetivos de Aprendizaje de la Clase

Al finalizar el estudio de este patrón, se espera que los alumnos sean capaces de:

- Definir el patrón Observer y su intención fundamental.
- Identificar los participantes clave del patrón y describir sus roles y responsabilidades.
- Explicar el mecanismo de notificación, incluyendo las variantes de push y pull.
- Analizar las ventajas y desventajas inherentes al uso del patrón Observer.
- Reconocer casos de uso comunes y escenarios donde la aplicación del patrón es apropiada.