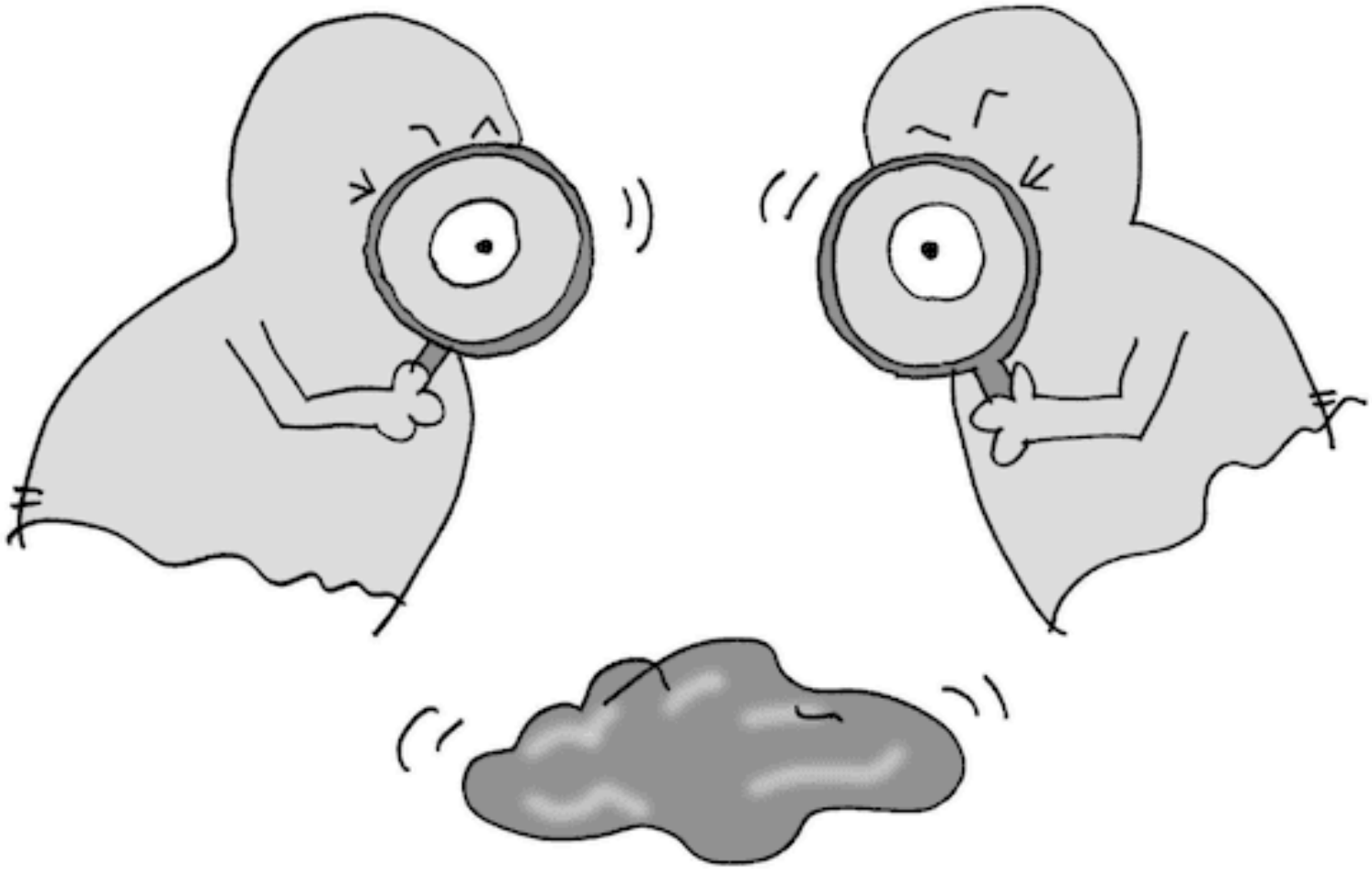
Observer Pattern

Define One-to-Many Dependencies



Observer Pattern

Think of **newspaper subscription**:

- **Newspaper publisher** maintains list of **subscribers**
- **Subscribers** register their interest in receiving newspapers
- When **new edition** is published, all **subscribers** automatically get notified
- **Subscribers** can cancel their subscription anytime

The **publisher** doesn't need to know who subscribers are - just that they want updates.

The Problem

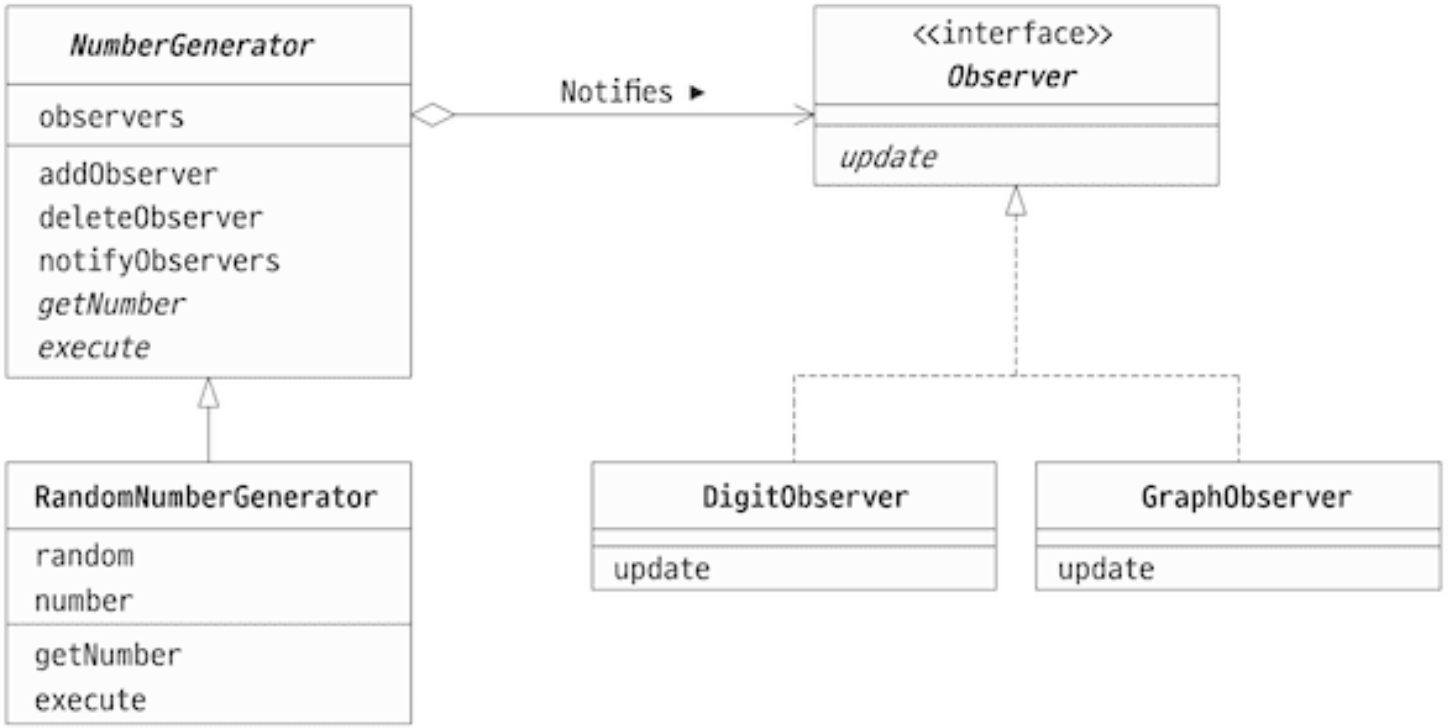
- We have **random number generator** that produces values.
- Multiple **display components** need to show numbers in different ways: digits, graphs, statistics.
- We want **loose coupling** - the generator shouldn't know about specific displays.

Challenge: How to notify multiple objects when state changes?

The *Observer* as the Solution

- We create **subject** that maintains list of *observers*.
- **Observers** register themselves with the subject.
- When **subject** changes, it *notifies* all registered observers automatically.

The Solution (Design)



Step 1: Understand the Players

In this design, we have players:

- *Subject* (NumberGenerator)
    - **ConcreteSubject** (RandomNumberGenerator)
- *Observer* (Observer)
    - **ConcreteObserver** (DigitObserver, GraphObserver)

Step 2: Notification Mechanism

- **Subject** maintains (owns) a list of observers and notifies them when the state changes.
- **Observers** implement the update interface to receive notifications.

Step 3: Understand abstractions (Subject-Observer)

- We have a *Subject* that `maintains` and `notifies` a list of *observers*.
    - *Subject* (NumberGenerator) - manages observers and state
    - *Observer* - defines update interface
- *Subject* calls `update()` on all observers when state changes.
    - Think of the `update()` method as a phone line between the Subject and Observer.

- Notice that **Subject** has methods to `add`, `remove`, and `notify` observers.
    - It maintains a list of observers and iterates through them.
- Notice that **Observer** has `update()` method.
    - This method is called when the subject's state changes.

Step 4: Understand concretion (Subject-Observer)

- We have **RandomNumberGenerator** (subject) and **DigitObserver**, **GraphObserver** (observers).
    - **RandomNumberGenerator**: Generates random numbers and notifies observers
    - **DigitObserver**: Displays numbers as digits
    - **GraphObserver**: Displays numbers as graphs

Code

- Main Method
- Subject Classes
- Observer Classes

Main Method

```
from random_number_generator import RandomNumberGenerator
from digit_observer import DigitObserver
from graph_observer import GraphObserver

def main():
    print("=== Observer Pattern Example ===\n")

    # Create subject
    generator = RandomNumberGenerator()

    # Create and register observers
    digit_obs = DigitObserver()
    graph_obs = GraphObserver()

    generator.add_observer(digit_obs)
    generator.add_observer(graph_obs)

    # Execute - all observers notified automatically
    generator.execute(5)
```

Step 1: Create subject

```
generator = RandomNumberGenerator()
```

- **RandomNumberGenerator** is the **subject** that will notify observers.
- It inherits observer management functionality from *NumberGenerator*.

Step 2: Create and register observers

```
digit_obs = DigitObserver()
graph_obs = GraphObserver()

generator.add_observer(digit_obs)
generator.add_observer(graph_obs)
```

- **Observers** register themselves with the subject.
- **Subject** maintains a list of all registered observers.

Step 3: Automatic notification

```
generator.execute(5)  # Generate 5 random numbers

# What happens internally:
# 1. generator.execute() changes internal state
# 2. generator calls notify_observers()
# 3. notify_observers() calls update() on each observer
# 4. Each observer receives the updated state
```

- **All observers** automatically notified when subject state changes.
- **No direct coupling** between subject and specific observer types.

Subject Classes

```python
# number_generator.py (Abstract Subject)
from abc import ABC, abstractmethod

class NumberGenerator(ABC):
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def delete_observer(self, observer):
        if observer in self._observers:
            self._observers.remove(observer)

    def notify_observers(self):
        for observer in self._observers:
            observer.update(self)

    @abstractmethod
    def get_number(self):
        pass

    @abstractmethod
    def execute(self):
        pass
```

Concrete Subject

```python
# random_number_generator.py
import random

class RandomNumberGenerator(NumberGenerator):
    def __init__(self, seed=None):
        super().__init__()
        self._random = random.Random(seed)
        self._number = 0

    def get_number(self):
        return self._number

    def execute(self, count=10):
        print("Starting random number generation...")

        for i in range(count):
            self._number = self._random.randint(0, 49)
            self.notify_observers()  # Notify all observers

        print("Random number generation completed.")
```

Observer Classes

```python
# observer.py (Abstract Observer)
from abc import ABC, abstractmethod

class Observer(ABC):
    @abstractmethod
    def update(self, generator):
        pass

# digit_observer.py (Concrete Observer)
class DigitObserver(Observer):
    def __init__(self, name="DigitObserver"):
        self._name = name

    def update(self, generator):
        current_number = generator.get_number()
        print(f"{self._name}: {current_number}")
```

Discussion

Push vs Pull Model

Push Model (what we've shown):

```
observer.update(self.get_number())  # pass data
```

- Subject pushes concrete data to the observer.
- Observer just receives what the subject sends.
- Pros: Simple for observers.
- Cons: Subject decides what to send (may be too much or too little).

Pull Model

```
observer.update(self)  # pass subject
```

- Subject pushes a minimal notification (“something changed”).
- Observer pulls the data it needs by querying the subject.
- Pros: Flexible for observers.
- Cons: Observers need to know how to query subject.

Observer Registration

```
# Dynamic observer management
generator.add_observer(digit_observer)     # Add observer
generator.add_observer(graph_observer)     # Add another
generator.delete_observer(digit_observer)  # Remove observer

# Observers can register/unregister at runtime
if need_graph_display:
    generator.add_observer(GraphObserver())
```

- **Observers** can be added/removed dynamically during execution.
- **Subject** doesn't need to know about specific observer types.

Key Benefits

1. **Loose coupling**: Subject and observers can vary independently
2. **Dynamic relationships**: Add/remove observers at runtime
3. **Broadcast communication**: Subject notifies all interested parties
4. **Open/Closed**: Add new observer types without changing subject

Key Drawbacks

1. **Unexpected updates**: Hard to track what triggers updates
2. **Update overhead**: All observers notified, even if not interested in the change
3. **Cascading updates**: Observer updates can trigger more updates
4. **Memory leaks**: Observers not properly removed can cause memory issues

When to Use Observer

- When **change to one object** requires changing multiple others
- When object should **notify others** without knowing who they are
- When you need **broadcast communication** mechanism
- When **abstraction has two aspects**, one dependent on the other

When NOT to Use Observer

- When you have **simple one-to-one** relationships
- When **performance is critical** (notification overhead)
- When **update cascades** become too complex to manage
- When observers need **specific update information** (consider specialized interfaces)

Real-World Examples

- **GUI frameworks**: Button clicks notify multiple event handlers
- **MVVM architecture**: ViewModel notifies Views when Model changes
- **Event systems**: Game events notify multiple subsystems
- **Social media**: User posts notify all followers

Related Patterns

- **Mediator**: Observer distributes communication, Mediator centralizes it
- **Singleton**: Subject often implemented as a Singleton

Observer vs Mediator

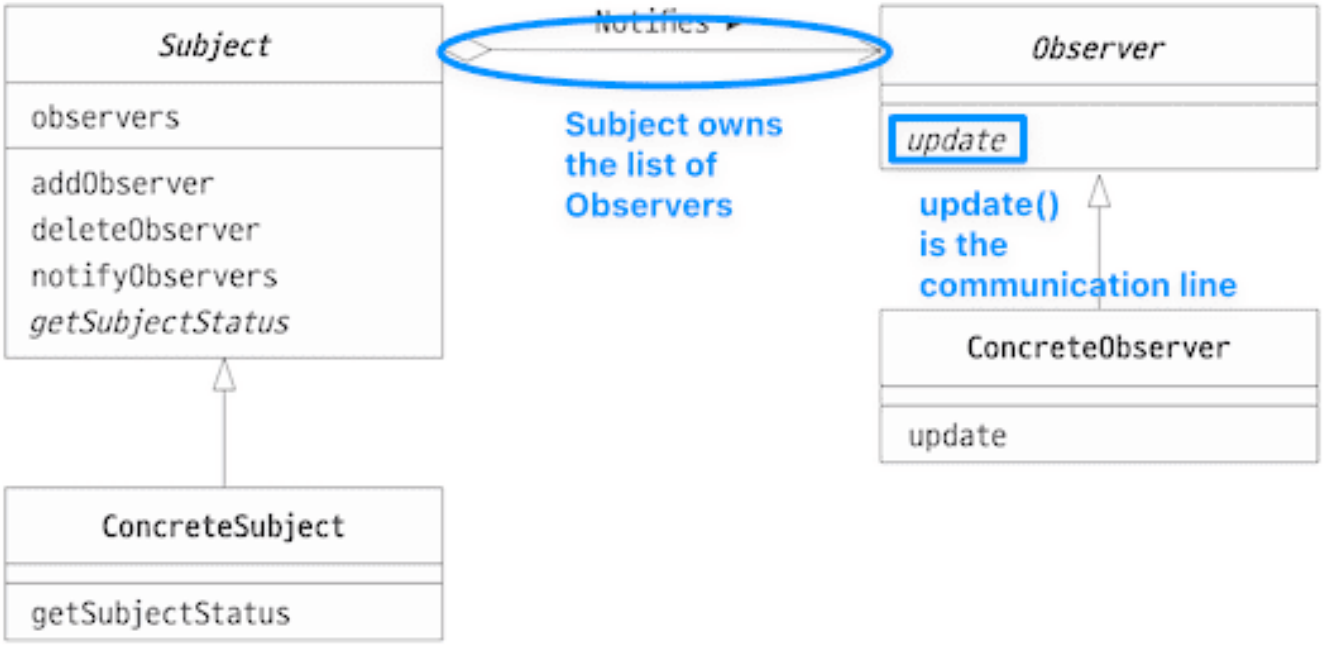
Observer:

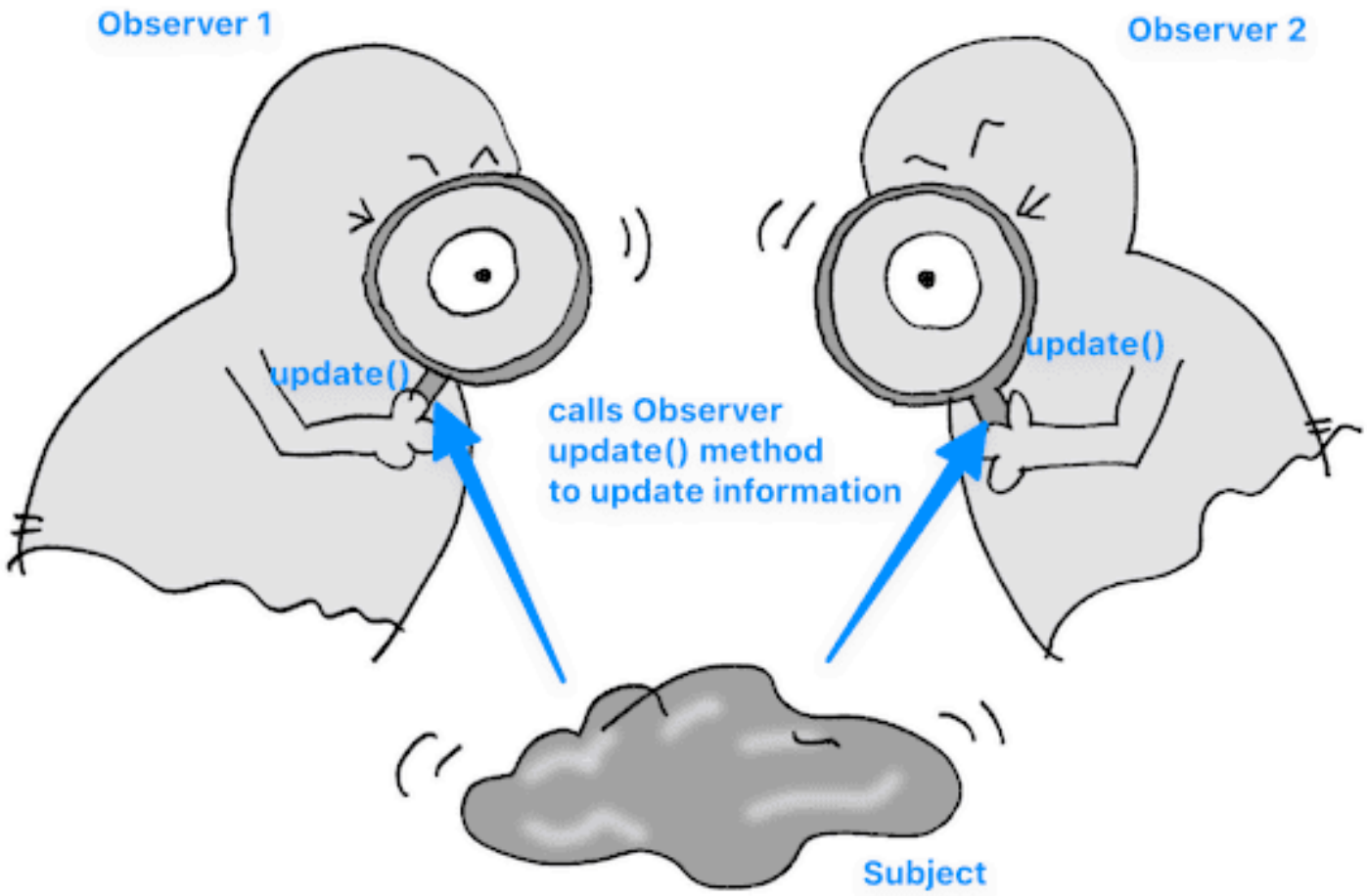
- **One-to-many** communication
- **Subject** broadcasts to all observers
- **Distributed** notification system

Mediator:

- **Many-to-many** communication
- **Centralized** communication hub
- **Complex** interaction coordination

UML


Subject
observers
addObserver
deleteObserver
notifyObservers
getSubjectStatus
Notifies
Subject owns the list of Observers
Observer
update
update() is the communication line
ConcreteObserver
update
ConcreteSubject
getSubjectStatus


Observer 1
Observer 2
update()
update()
calls Observer
update() method
to update information
Subject