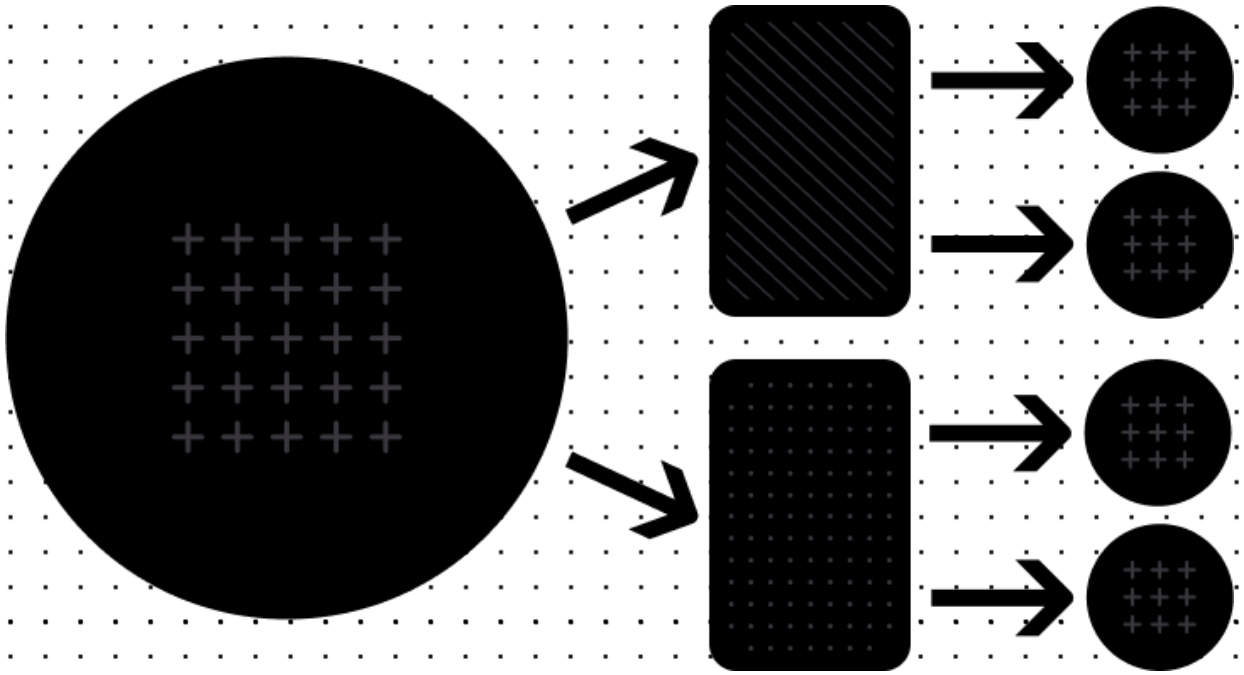
PUB/SUB AND OTHERS DESIGN PATTERN WITH JS

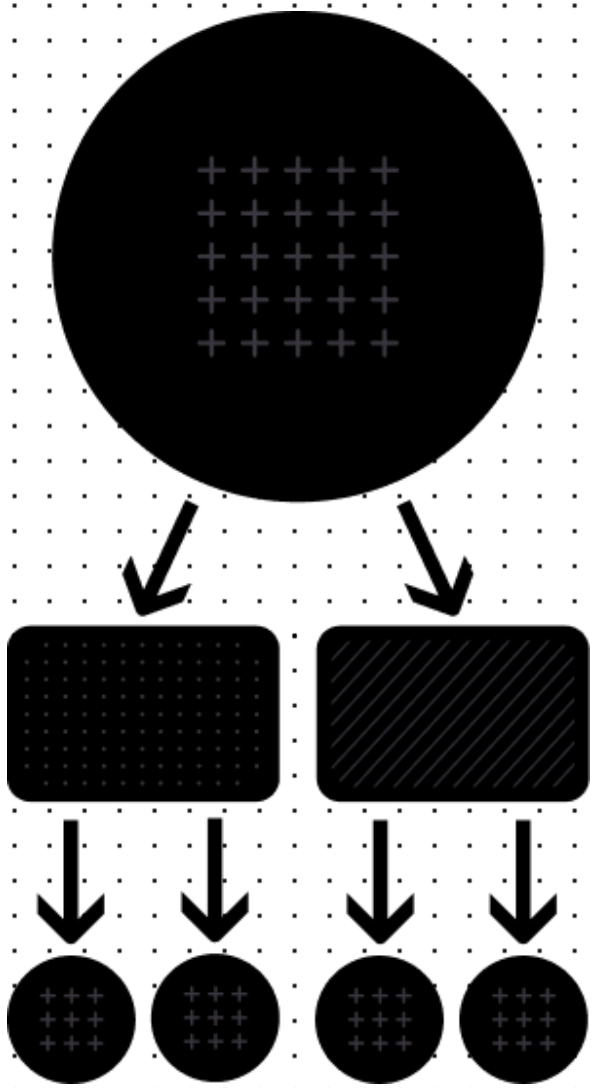
Vítor Norton @vt_norton

Dev Advocate @ SuperViz



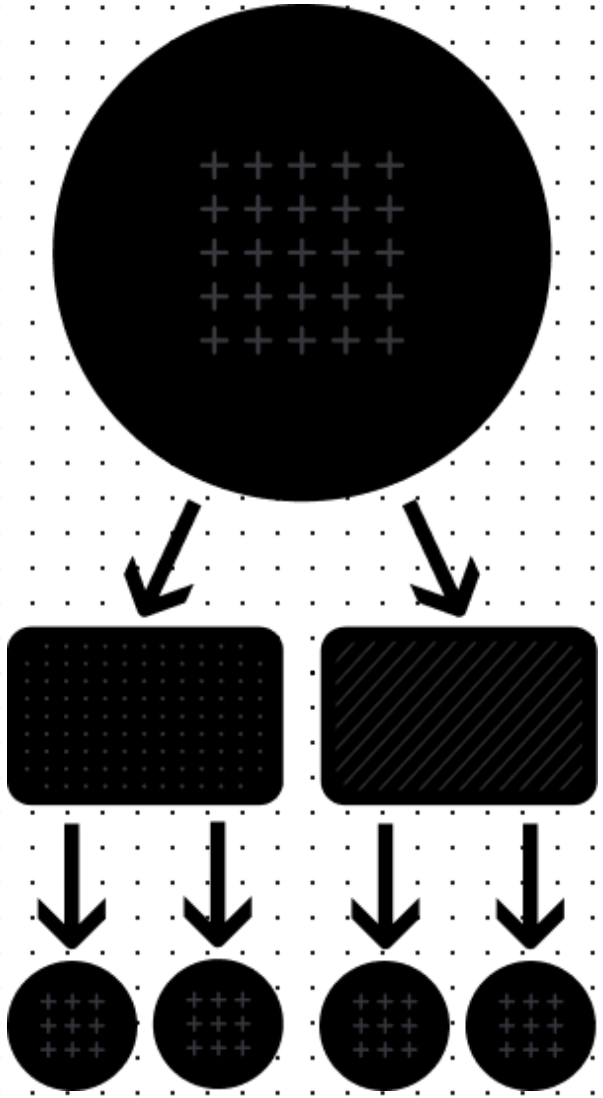
ABOUT ME

- Strong .NET background
- 5 years into TypeScript and React development
- Twitch Streamer (@vt_norton)
- Dev Advocate @ SuperViz
- Love greek mythology
- Passionate about movies and music



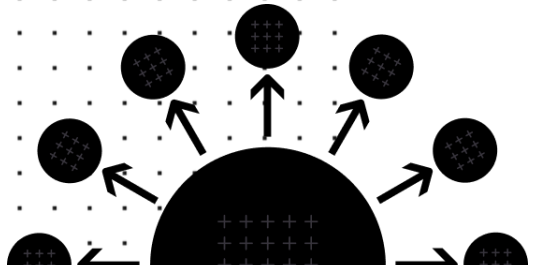
DESIGN PATTERNS

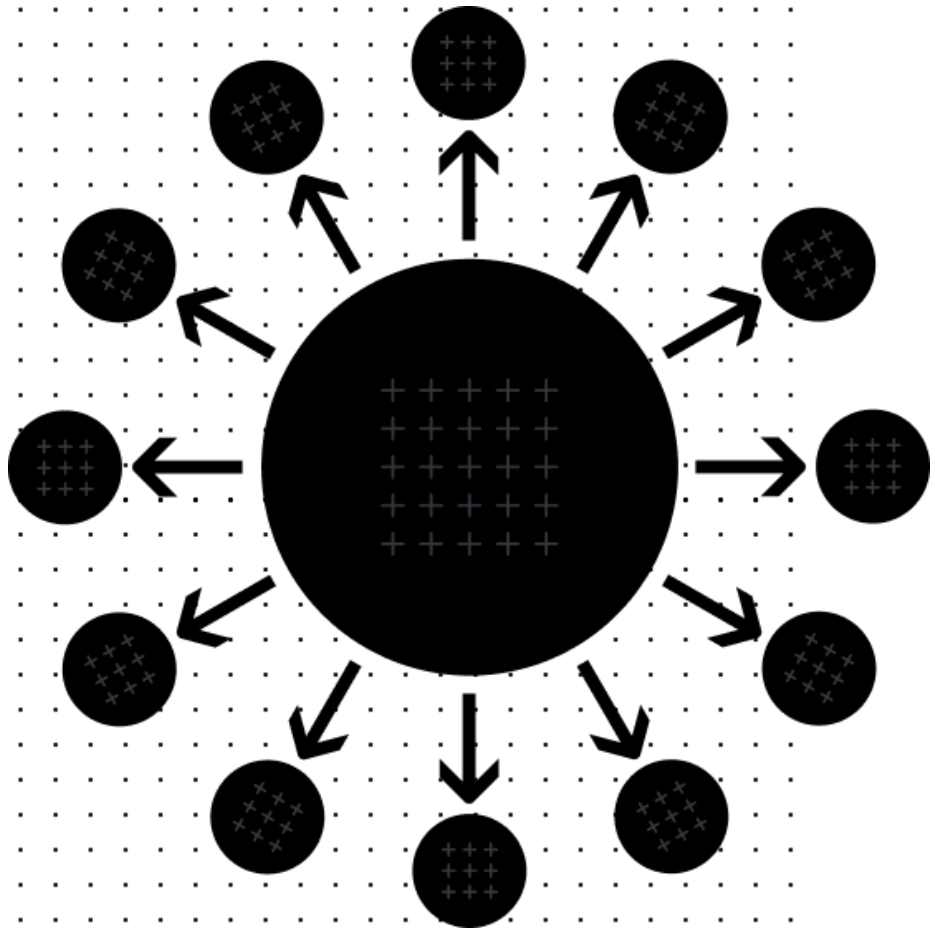
- Standardizes solutions for your team
- Guide for solving known and commons problems
- Make it easier to write reusable and maintainable code
- Today I will show two behavior patterns



WHY LEARN THEM?

- Boost career prospects
- Facilitate communication
- Improve your code quality
- Expand what you can do

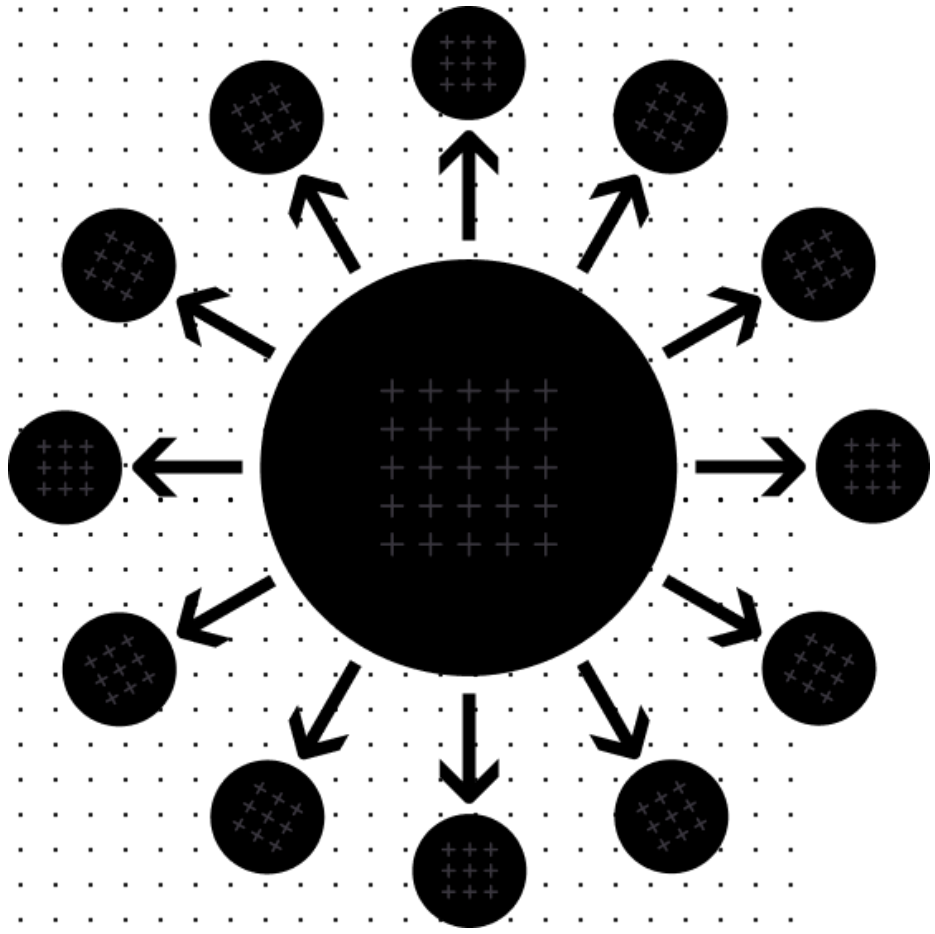




OBSERVER PATTERN

Simple and widely used messaging design, it contains two entities: the **subject** and the **observer**, and this is their behavior:

- The subject maintains a list of its dependents, called observer
- The subject notifies observers of any state changes automatically



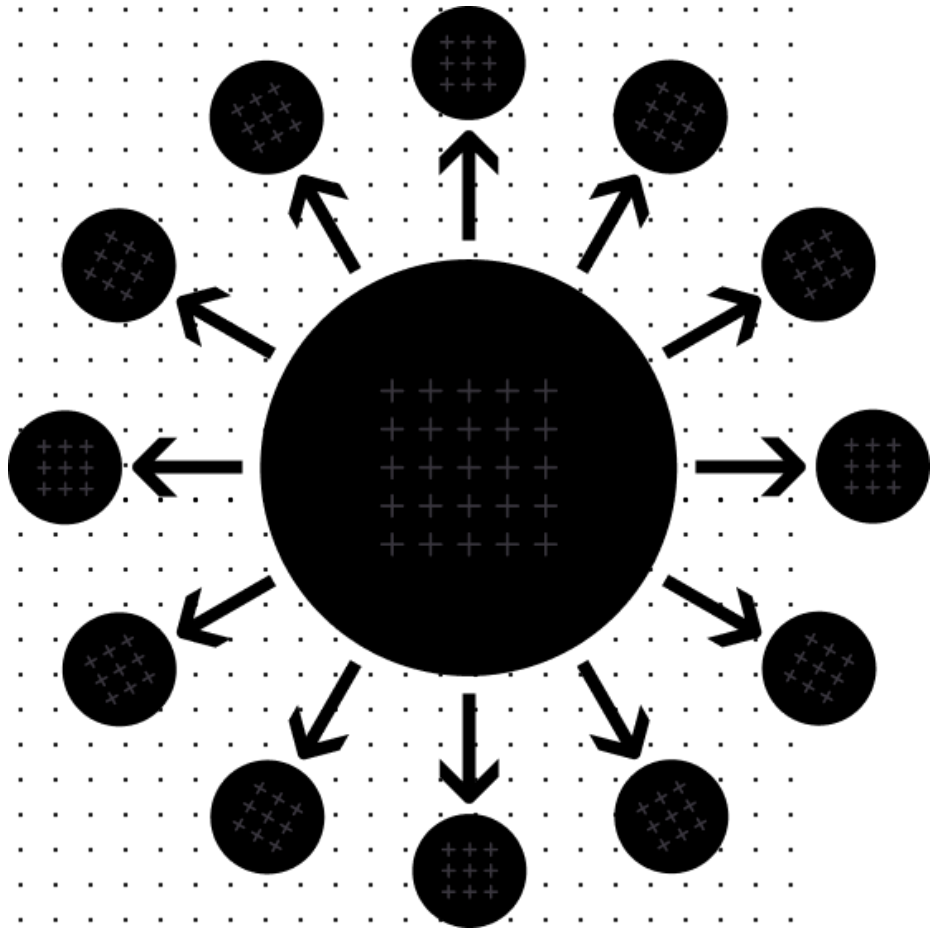
Real case scenario

OBSERVER PATTERN

Robert wants a new iPhone

The store!

Two ways to solve this without the observer pattern

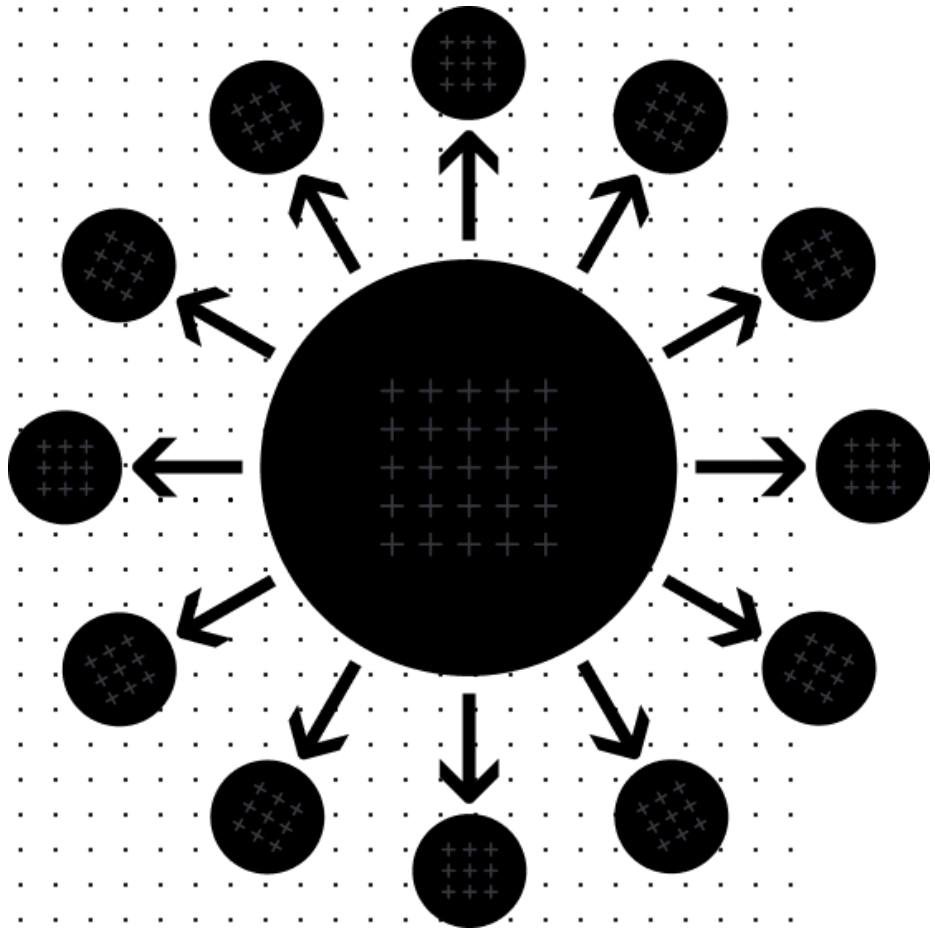


Without Observer Pattern

FIRST WAY TO DO IT

He goes to the store to check if it arrived

The stores doesn't have it *yet*



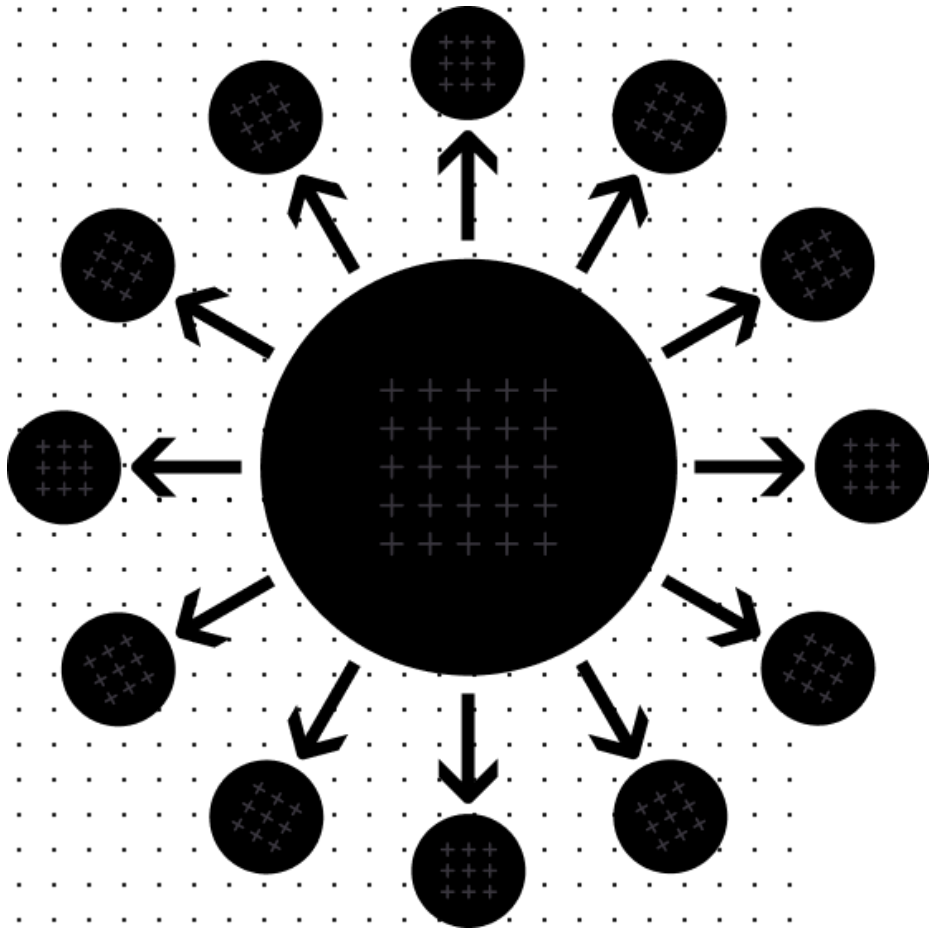
Without Observer Pattern

FIRST WAY TO DO IT

He goes to the store to check if it arrived

The stores doesn't have it *yet*

Repeat every day to check it it arrived



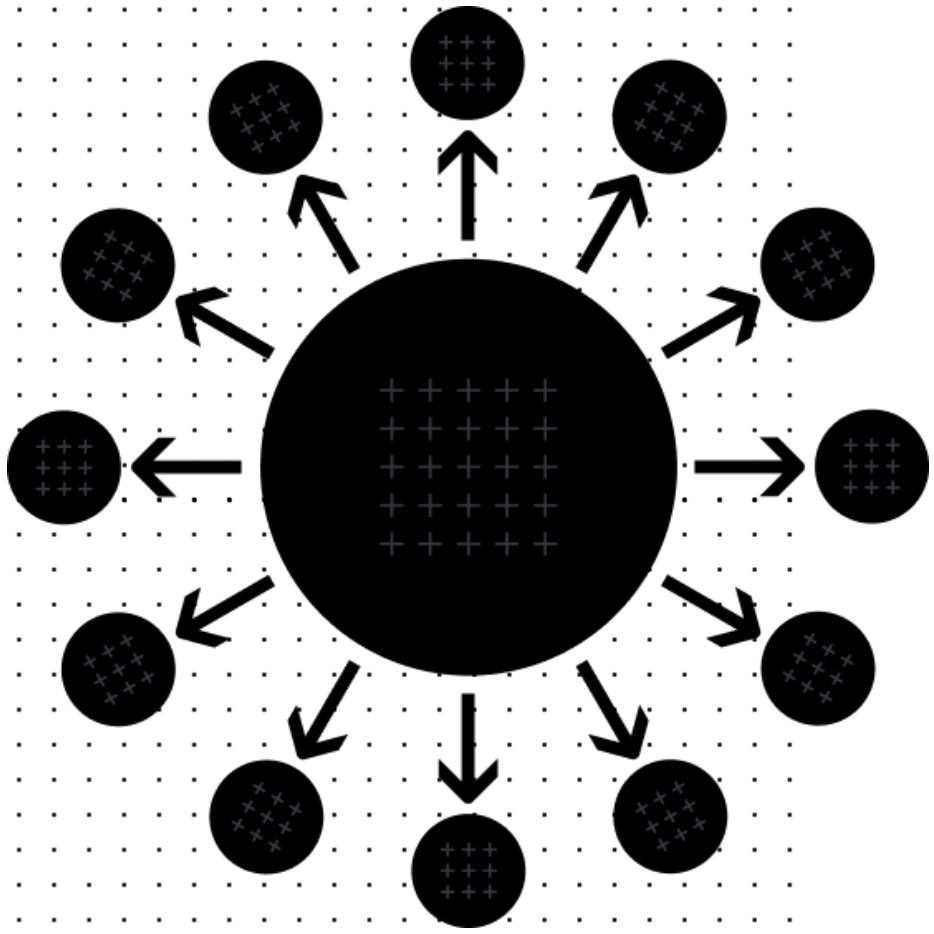
Without Observer Pattern

SECOND WAY TO DO IT

When the product arrive at the store

Sends notification to everyone

Robert doesn’t need to go to the store every day, he is notified



Without Observer Pattern

SECOND WAY TO DO IT

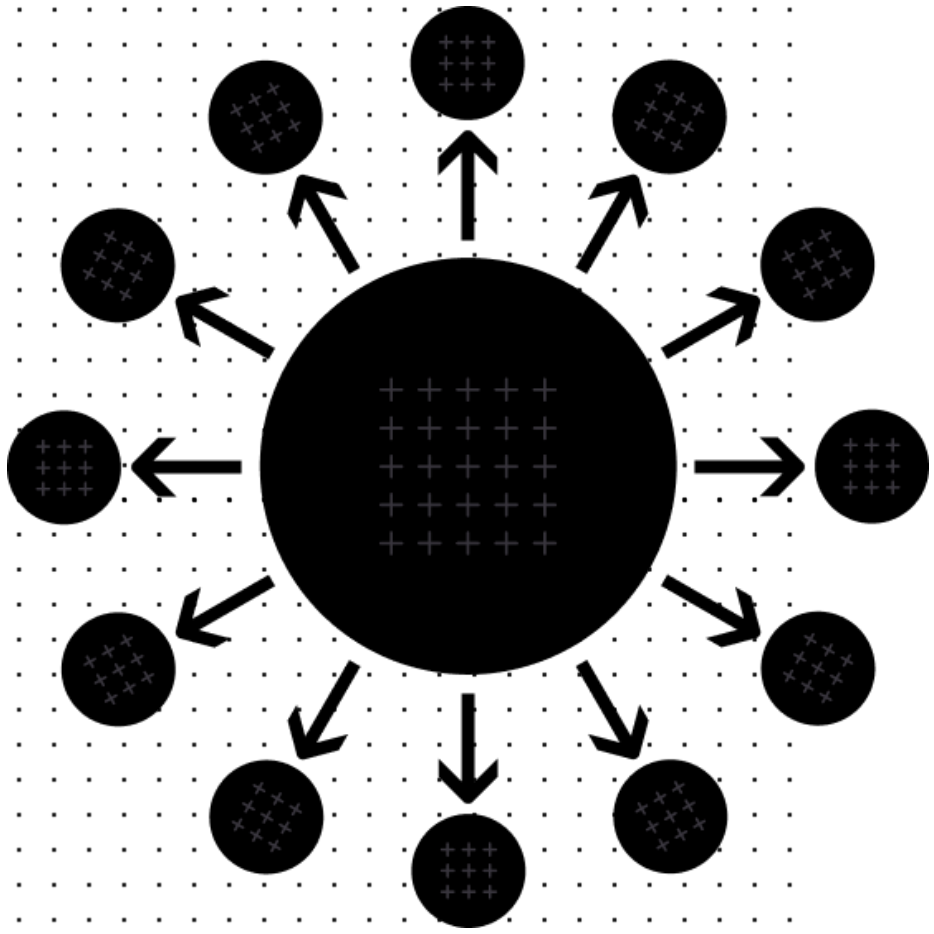
When the product arrive at the store

Sends notification to everyone

Robert doesn’t need to go to the store every day, he is notified

Sarah didn’t want to know

OMG! Another spam!

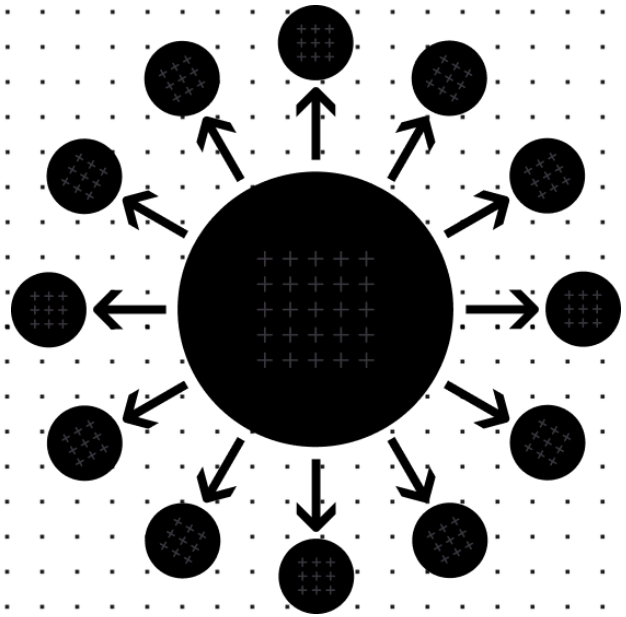


THE OBSERVER PATTERN

Robert tells the store he wants a new iPhone

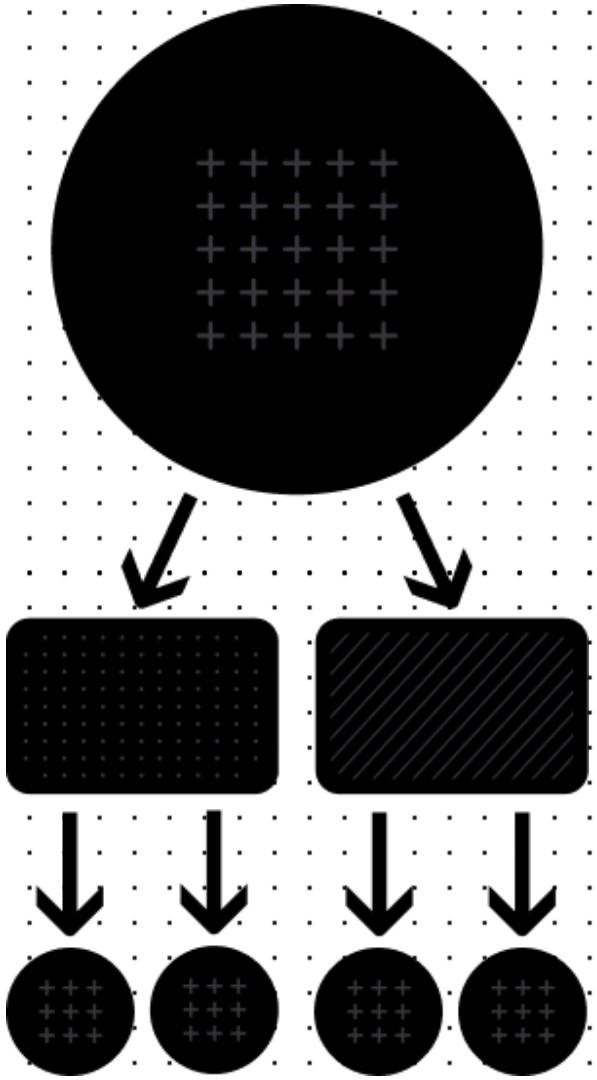
A new iPhone arrive

Sends notification to the subscribers



HANDS-ON

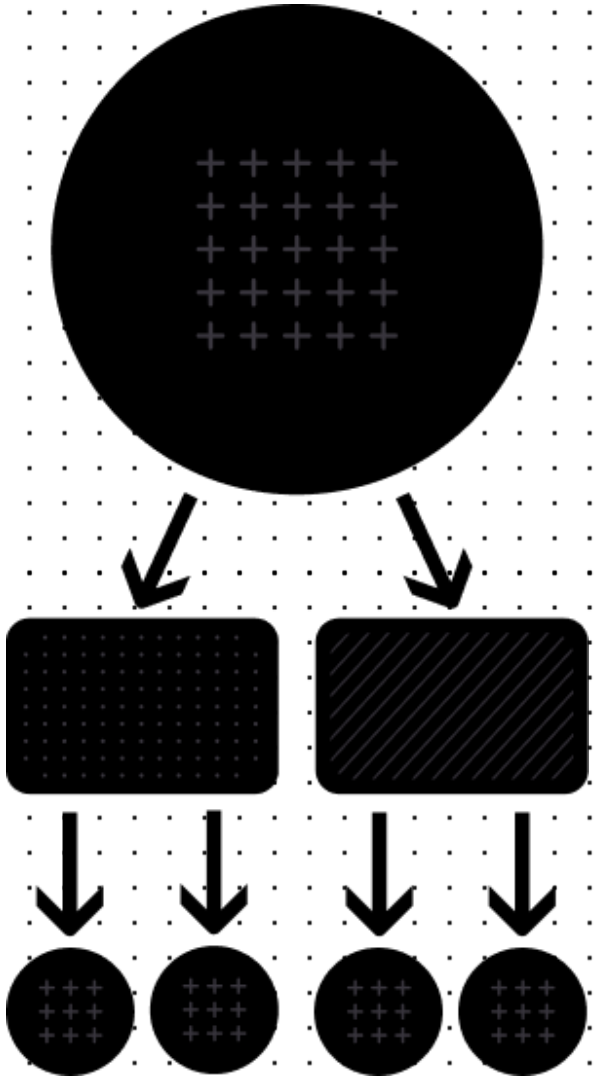
```
// Define a class for the Provider
class Store {
  constructor() {
    this.observers = [];
  }

  // Add an observer to the list
  add(observer) {
    this.observers.push(observer);
  }

  // Notify all observers about new product
  notify(product) {
    this.observers.forEach(observer => observer.update(product));
  }
}

// Define a class for the Observer
class Customer {
  update(product) {
    console.log(`New product added: ${product}`);
  }
}

// Usage
const store = new Store();
const customer = new Customer();
store.add(customer); // Add a customer to the store's observers
store.notify('iPhone 15'); // Notify all observers about a new product
```



PUB SUB PATTERN

The Publish/Subscriber Pattern, or PutSub for close friends, also widely used for messagin, it contains two entities: the **publisher** and the **subscriber**. This is their behavior:

- The subscriber connects to a topic
- The publisher sends messages to a topic
- The subscriber receives messages from that topic

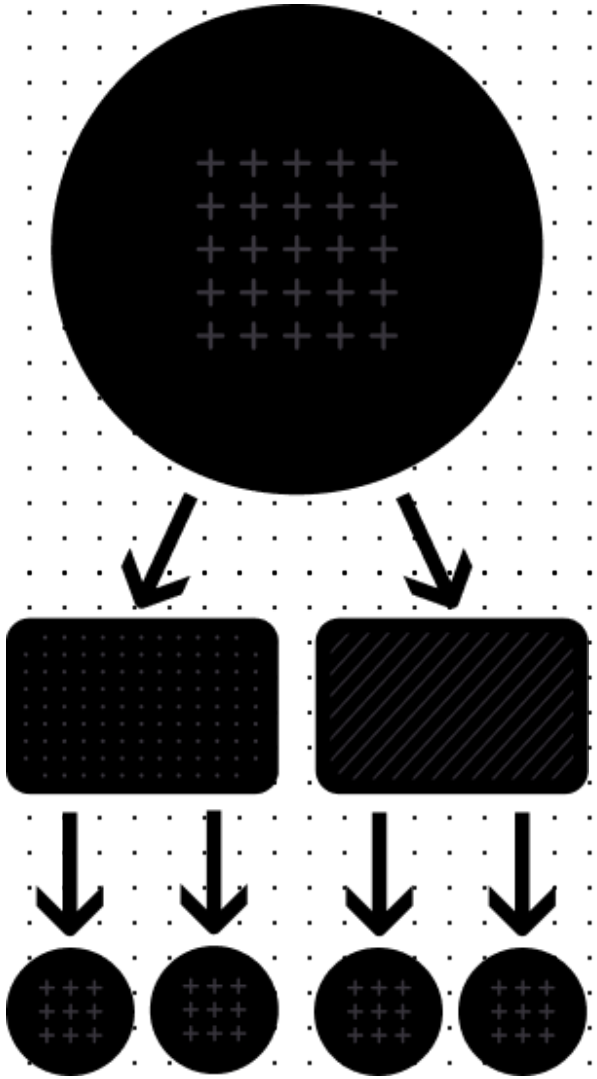


Real case scenario

PUB SUB PATTERN

Robert wants a new iPhone

The store!



Real case scenario

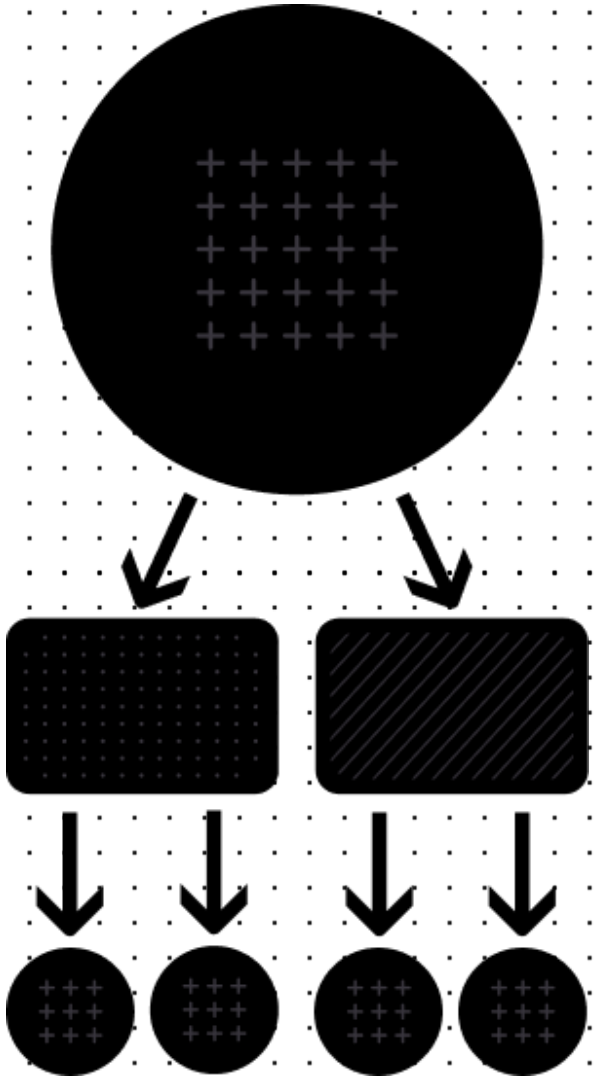
PUB SUB PATTERN

Robert wants a new iPhone

A market place

(I wanted a better icon, but this is the closest emojie available)

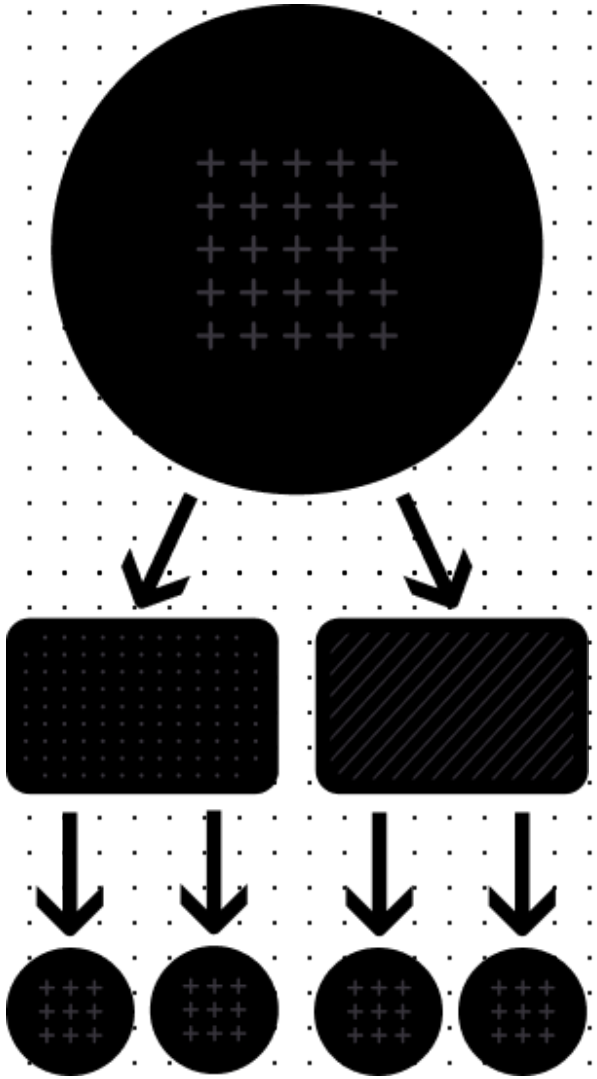
The store!



Real case scenario

PUB SUB PATTERN

Robert subscribes to the market place on the event ‘new-phone’

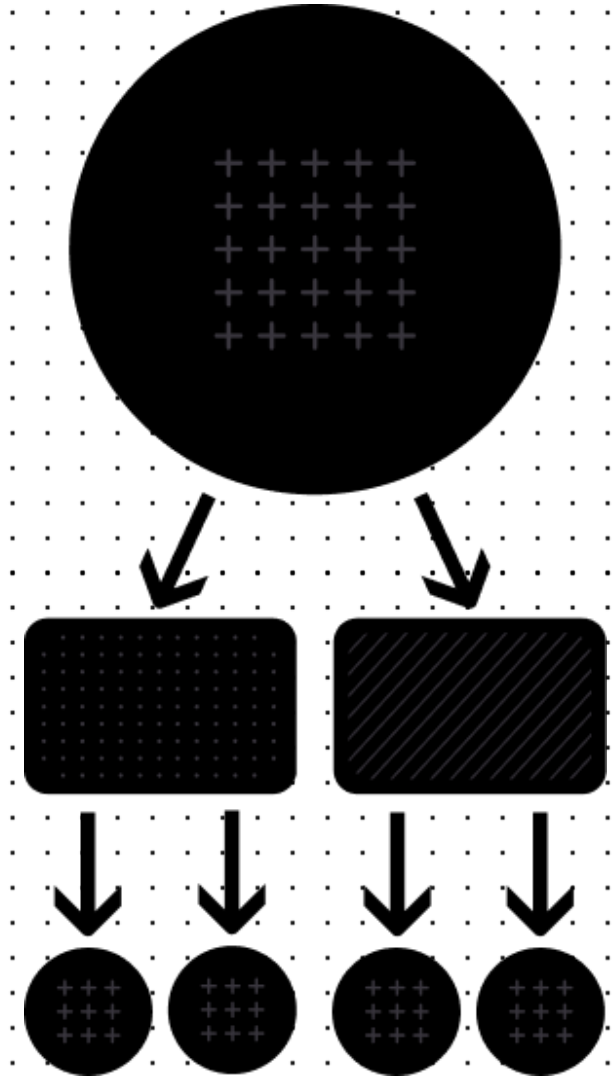


Real case scenario

PUB SUB PATTERN

Robert subscribes to the market place on the event ‘new-phone’

The store publish the arrival of the new iPhone



Real case scenario

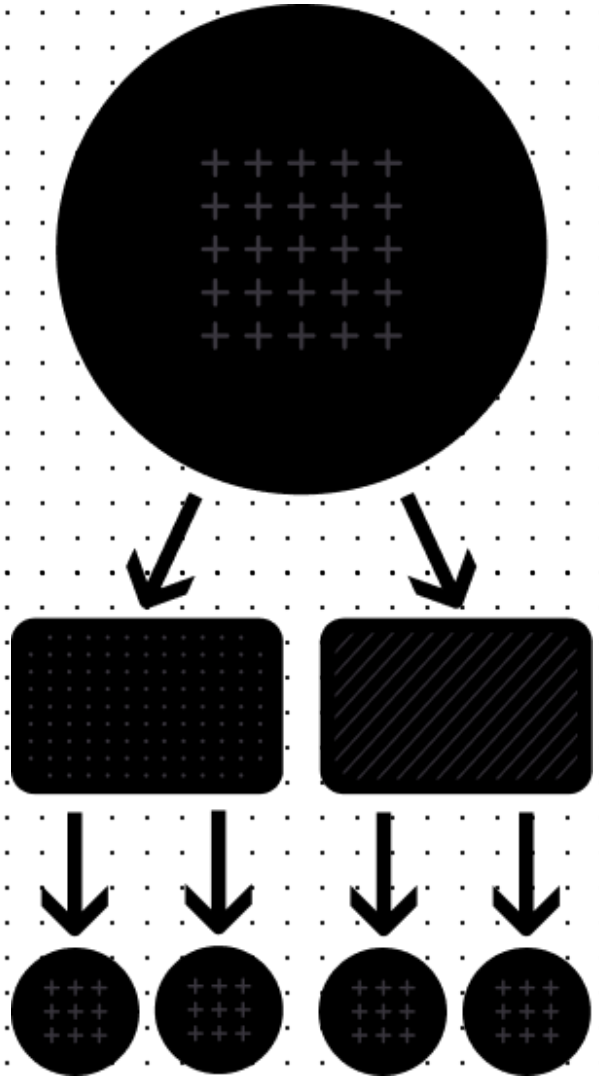
PUB SUB PATTERN



Robert subscribes to the market place on the event ‘new-phone’


The store publish the arrival of the new iPhone

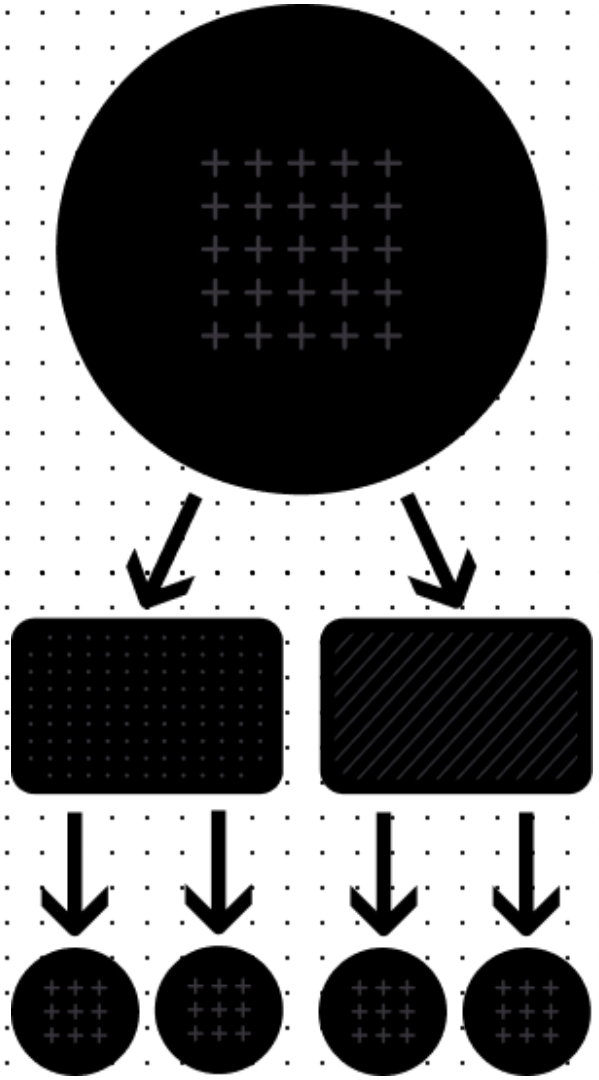
The marketplace tells Robert it arrived!



HANDS-ON

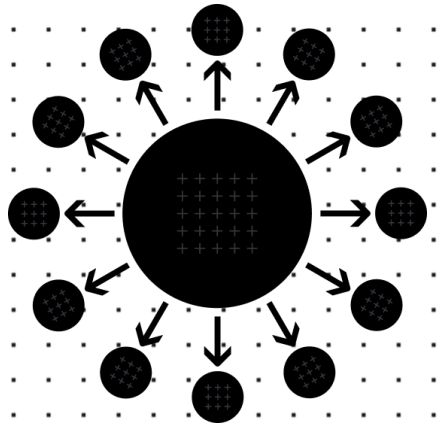
```
pubsub.subscribe("new-phone", callbackFunction);

function callbackFunction(payload) {
    // do something
}
```

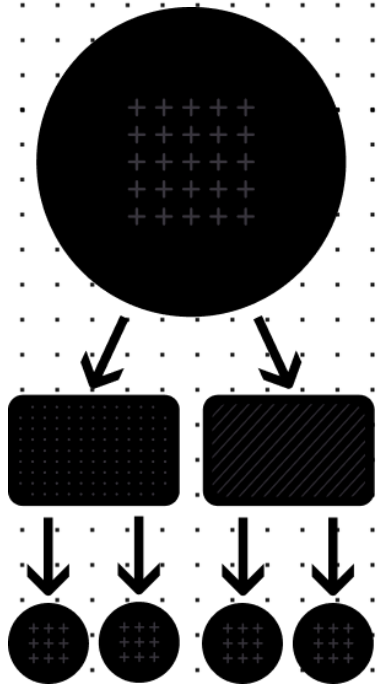
```
const phone = {
    model: 'iPhone 16',
    brand: 'Apple'
    price: 999
};

pubsub.publish("new-phone", phone);
```



HANDS-ON

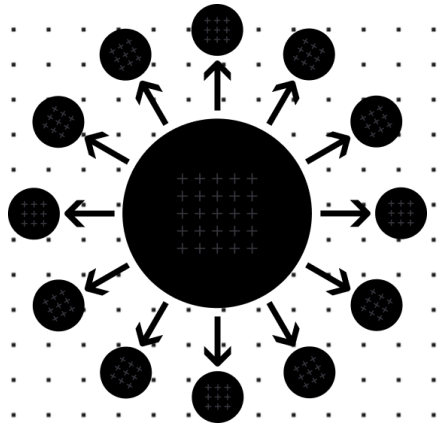
```
class PubSub {
  static events = {}; // It has the an empty list of events

  // The publish method takes an event name and data
  publish(event, data) {
    // Check if the event doesn't exist
    if (!this.events[event])
      return;

    // For each subscriber of this event,
    // call the callback function with the provided data
    this.events[event].forEach((callback) => {
      callback(data);
    });
  }

  // The subscribe method takes an event name and a callback function
  subscribe(eventName, callback) {
    // If the event doesn't exist yet, initialize it as an empty array
    if (!this.events[eventName])
      this.events[eventName] = [];

    // Push the callback function into the array of
    // callbacks for the given event
    this.events[eventName].push(callback);
  }


}
```

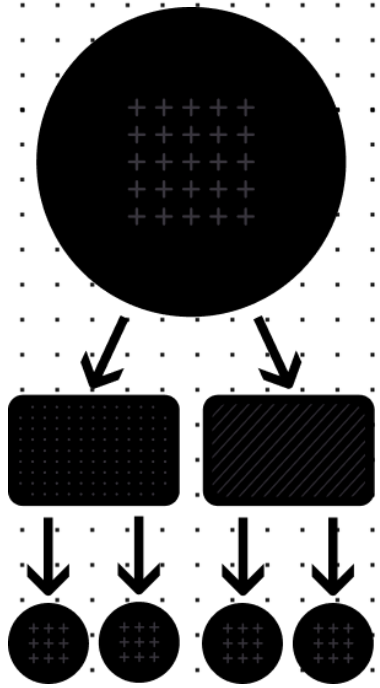




A BIT MORE COMPLEX, RIGHT?

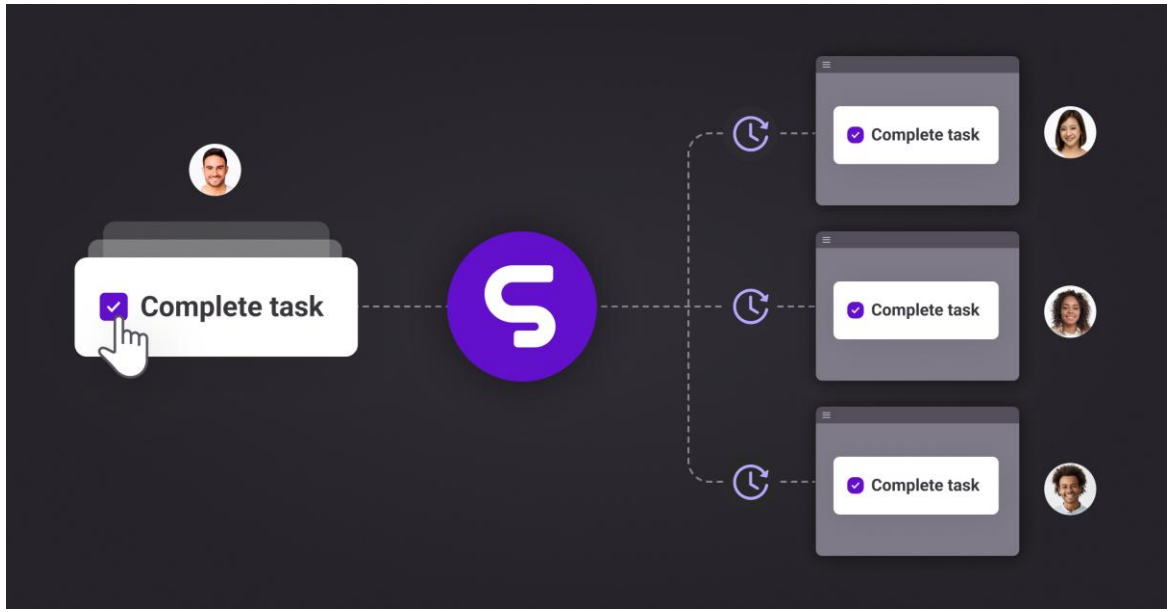
I got you covered! 😉





I got you covered! 😉

FRAMEWORKS



SUPERVIZ SDK - REAL-TIME COMMUNICATION

It uses a concept of a room, everyone in the room, independent of devices or network, will receive the message.

superviz.com

RESOURCES

Design Pattern #3 - Observer Pattern

DEV Community

Design Pattern #4 - Pub/Sub Pattern

DEV Community

Understanding and implementing Event-Driven Communication in Front-End Development

DEV Community