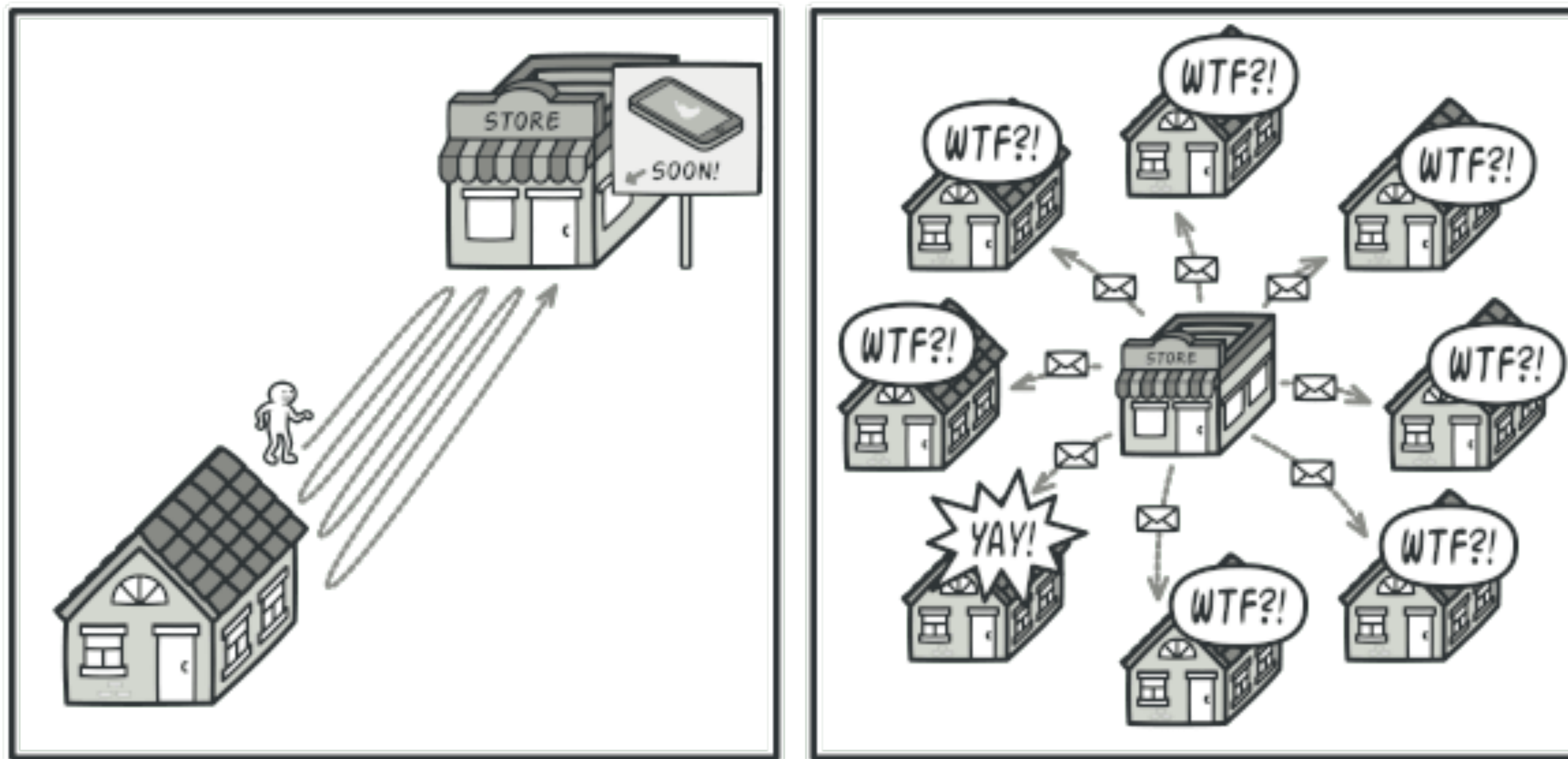


Observer Pattern

Вибрані питання програмної інженерії

Яременко Петро, ФІ БП КН-4, 2024

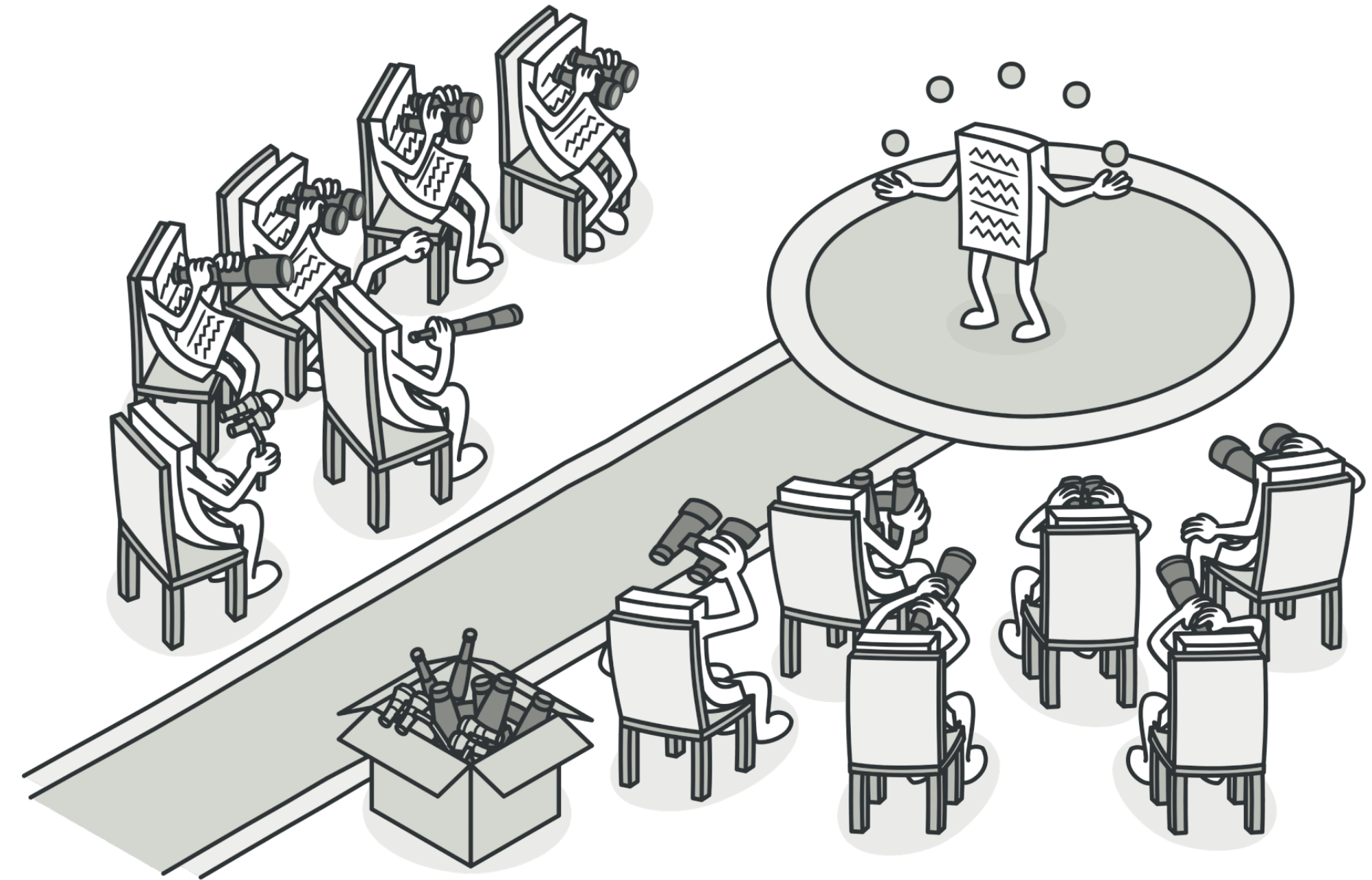
Проблема



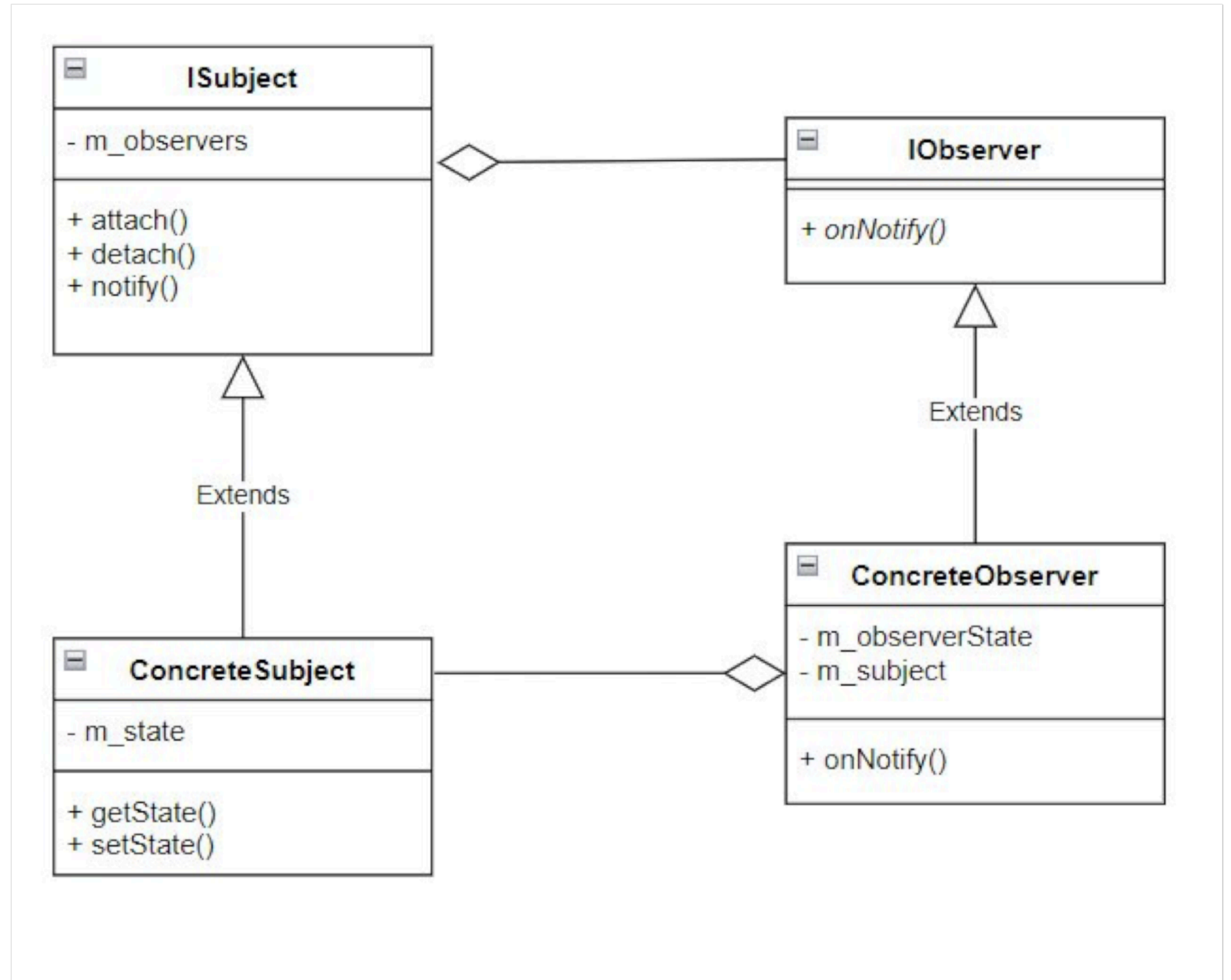
Або клієнт витрачає час на перевірку наявності продукту, або магазин витрачає ресурси на сповіщення неправильних клієнтів.

Рішення

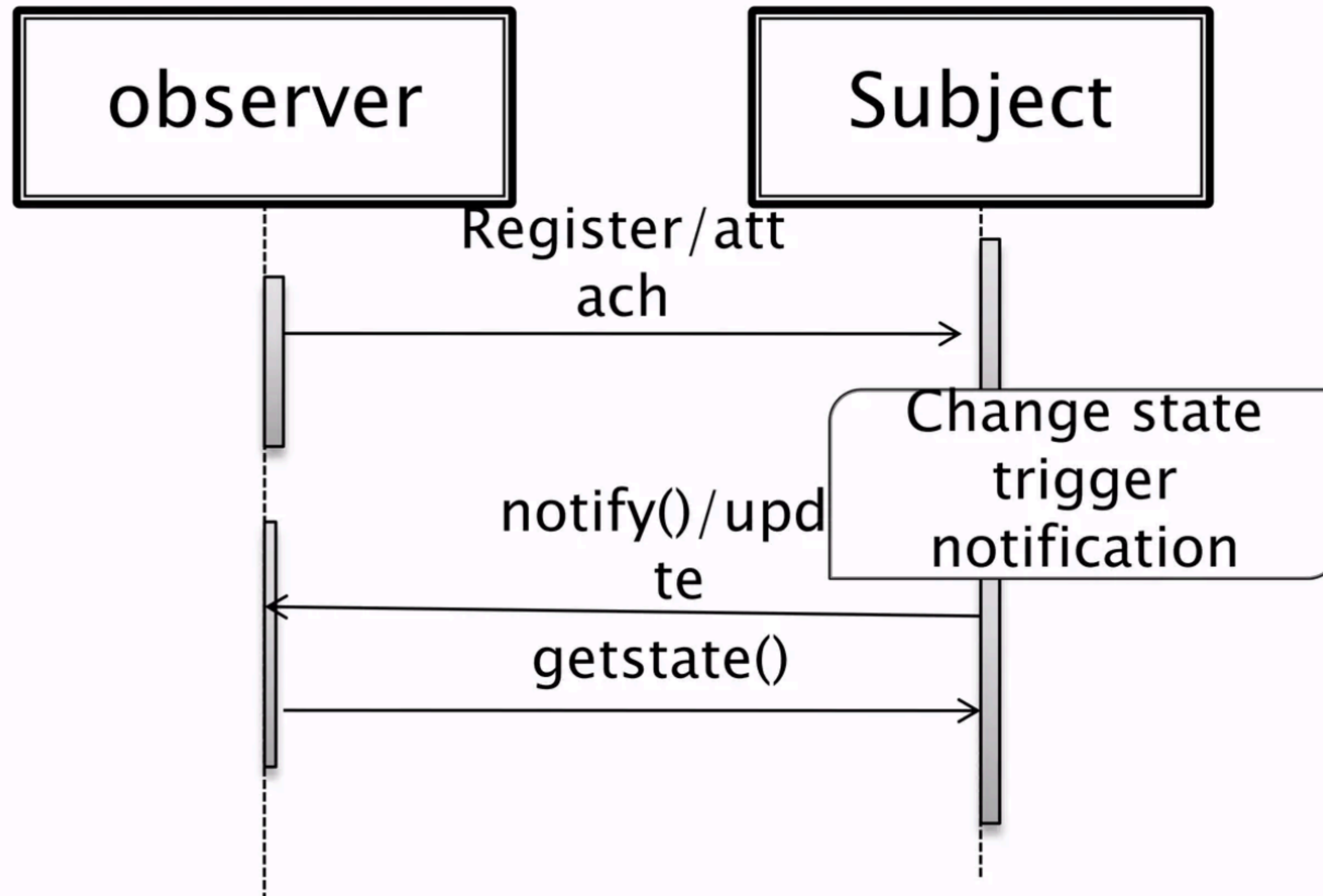
Паттерн програмування **"Observer"** — це паттерн проектування, який використовується для створення механізму subscription, щоб сповіщати об'єкти про будь-які зміни стану у **observable** об'єкті. Цей паттерн входить до категорії поведінкових паттернів, оскільки він визначає спосіб комунікації та взаємодії об'єктів.



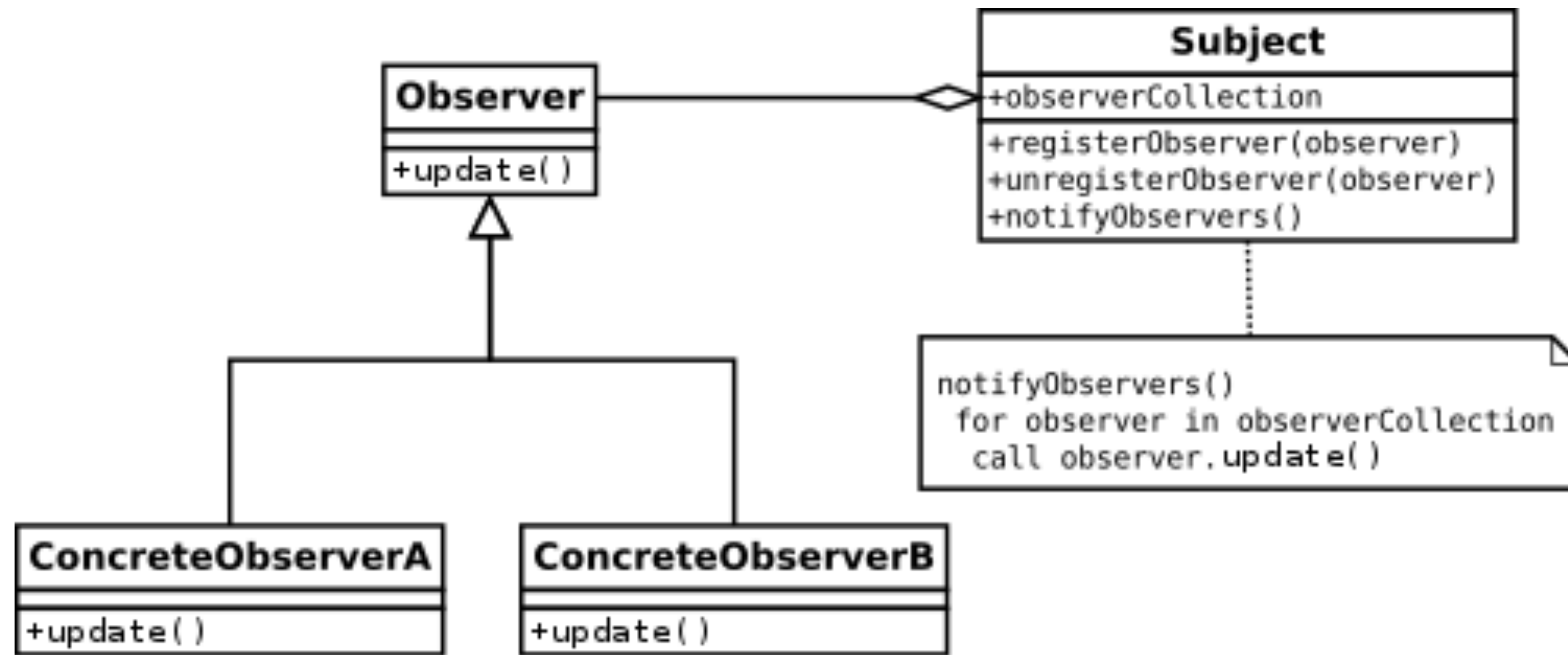
Observer GoF



Sequence diagram



Основні ролі



- **Subject (Суб'єкт)**: Це об'єкт, який має стан і може сповіщати своїх спостерігачів про будь-які зміни свого стану.
- **Observer (Спостерігач)**: Це об'єкт, який очікує на зміни у суб'єкта і реагує на них.

Subject

```
// Define a class to manage subjects and their observers.  
1 implementation  
pub struct Subject {  
    | observers: Vec<Arc<dyn Observer + Send + Sync>>,  
}  

```

```
// Implementation for Subject.  
impl Subject {  
    // Constructor for Subject, initializing an empty list of observers.  
    pub fn new() -> Self {  
        Self {  
            | observers: Vec::new(),  
        }  
    }  
  
    // Method to add observers to the Subject.  
    pub fn add_observer(&mut self, observer: Arc<dyn Observer + Send + Sync>) {  
        | self.observers.push(observer);  
    }  
  
    // Notify all observers about an order.  
    pub async fn notify_observers(&self, order: &Order) {  
        | let mut handles: Vec<JoinHandle<()>> = vec![];  
        | for observer: &Arc<dyn Observer + Send + Sync> in &self.observers {  
        | | handles.push(observer.notify(order));  
        | }  
  
        | for handle: JoinHandle<()> in handles {  
        | | handle.await.unwrap();  
        | }  
    }  
  
    pub async fn place_order(&mut self, order: Order) {  
        | self.notify_observers(&order).await;  
    }  
}  
} impl Subject
```


Observer

```
// Implementations
pub trait Observer {
    // Notify method which takes an order reference and returns a join handle.
    fn notify(&self, order: &Order) -> tokio::task::JoinHandle<()>;
}
```

```
// Implement the Observer trait for PaymentGateway.
impl Observer for PaymentGateway {
    // Method to handle notification logic for PaymentGateway.
    fn notify(&self, order: &Order) -> tokio::task::JoinHandle<()> {
        let id: u32 = order.order_id;
        let product: String = order.product_name.clone();
        let quantity: u32 = order.quantity;
        let identifier: String = self.identifier.clone();
        tokio::spawn(future: async move {
            sleep(Duration::from_secs(1)).await; // Simulate a 1-second verification time
            println!("{}", verified payment details for Order {}: {} x {}",
                identifier, id, product, quantity);
        })
    }
}
```

```
// Implement the Observer trait for EmailNotifier.
impl Observer for EmailNotifier {
    // Method to handle notification logic for EmailNotifier.
    fn notify(&self, order: &Order) -> tokio::task::JoinHandle<()> {
        let id: u32 = order.order_id;
        let product: String = order.product_name.clone();
        let identifier: String = self.identifier.clone();
        tokio::spawn(future: async move {
            sleep(Duration::from_secs(1)).await; // Simulate email sending delay
            println!("{}", sent email confirmation for Order {}: {}", identifier, id, product);
        })
    }
}
```

```
// Implement the Observer trait for LogisticsManager.
impl Observer for LogisticsManager {
    // Method to handle notification logic for LogisticsManager.
    fn notify(&self, order: &Order) -> tokio::task::JoinHandle<()> {
        let id: u32 = order.order_id;
        let product: String = order.product_name.clone();
        let quantity: u32 = order.quantity;
        let identifier: String = self.identifier.clone();
        tokio::spawn(future: async move {
            sleep(Duration::from_secs(2)).await; // Simulate logistics planning delay
            println!("{}", arranged shipping for Order {}: {} x {}", identifier, id, product, quantity);
        })
    }
}
```


Tests

```
#[tokio::main]
▶ Run | Debug
async fn main() {
    // Create a new Subject instance to manage order notifications.
    let mut subject: Subject = Subject::new();

    // Create thread-safe shared instances of order processing components using Arc.
    let payment_processor: Arc<PaymentProcessor> = Arc::new(data: PaymentProcessor::new(identifier: "PaymentProcessor"));
    let payment_gateway: Arc<PaymentGateway> = Arc::new(data: PaymentGateway::new(identifier: "PaymentGateway"));
    let inventory_manager: Arc<InventoryManager> = Arc::new(data: InventoryManager::new(identifier: "InventoryManager"));
    let email_notifier: Arc<EmailNotifier> = Arc::new(data: EmailNotifier::new(identifier: "EmailNotifier"));
    let logistics_manager: Arc<LogisticsManager> = Arc::new(data: LogisticsManager::new(identifier: "LogisticsManager"));

    // Register all observers with the Subject for order notifications.
    subject.add_observer(payment_processor.clone()); // Clone Arc to avoid ownership issues
    subject.add_observer(payment_gateway.clone());
    subject.add_observer(inventory_manager.clone());
    subject.add_observer(email_notifier.clone());
    subject.add_observer(logistics_manager.clone());

    // Simulate multiple orders
    let orders: Vec<Order> = vec![
        Order::new(101, "Super Widget", 10),
        Order::new(102, "Ultra Gadget", 5),
        Order::new(103, "Mega Toolset", 3),
    ];

    for order: Order in orders {
        // Place each order asynchronously using await within the loop.
        subject.place_order(order.clone()).await;
        println!("Completed processing for Order {}", order.order_id);
    }
} fn main
```

Нюанси

- У суб'єкті зберігається лише список спостерігачів
- Не потрібно вказувати, звідки повідомлення
- Спостерігач сам вирішує на кого підписуватись та як реагувати
- Комунікація між різними рівнями
- Ланцюг оновлень після зміни стану

Переваги

- Мінімальний зв'язок між Суб'єктом та Спостерігачем:
- Можна повторно використовувати суб'єкт без повторного використання їх спостерігачів і навпаки.
- Спостерігачі можуть бути додані без зміни суб'єкта.
- Все, що знає суб'єкт, це його список спостерігачів, отже, відбувається декаплінг.
- Суб'єкту не потрібно знати конкретний клас спостерігача.

Недоліки та складнощі

- Можливість витоку пам'яті
- Навантаження на публікацію подій
- Взаємозв'язок між видавцем та підписниками
- Необхідність оновлення всіх підписників
- Синхронізація при багатопотоковості
- Підвищена складність при великій кількості подій та підписників

Приклади

- Stock price updates
- Social media feeds
- Email notifications
- News aggregators
- Game score updates
- Online auction updates
- Task management updates

Порівняння з іншими паттернами

Порівняння з паттерном "Callback"

У паттерні "Callback" функція передається як аргумент іншій функції для виклику у відповідь на певну подію. Паттерн "Observer" використовує об'єкти, які взаємодіють один з одним, замість функцій.

Використання об'єктів може бути більш гнучкою або структурованою, особливо для більш складних взаємодій.

Порівняння з паттерном "Mediator"

У паттерні "Mediator" об'єкт, відомий як посередник, виступає посередником між різними компонентами програми, сприяючи їх взаємодії. У порівнянні з "Observer", де об'єкти підписуються один на одного, у "Mediator" всі взаємодії відбуваються через посередника, що робить систему менш зв'язаною.

Порівняння з паттерном "State" (стан)

У паттерні "State" об'єкт змінює свій стан, і його поведінка змінюється відповідно до цього стану. Паттерн "Observer" може використовуватися для сповіщення про зміну стану об'єкта і оновлення відповідних підписників. Обидва паттерни можуть співпрацювати для організації складних систем з внутрішніми станами та поведінкою.

Висновки

Паттерн "Observer" є потужним і корисним інструментом для організації взаємодії між об'єктами в програмах. Він дозволяє створювати реактивні системи, де підписники автоматично реагують на зміни стану видавця, без необхідності явних запитів або опитувань.

У певних випадках може бути важко визначити, коли і де краще використовувати паттерн "Observer", але він є важливим інструментом при створенні реактивних, модульних та гнучких систем. При правильному використанні паттерн "Observer" допомагає створювати добре організовані та підтримувані програмні системи.

Джерела

- Design Patterns: Elements of Reusable Object-Oriented Software" (The "Gang of Four" book) - Еріх Гамма, Річард Хелм, Ральф Джонсон, Джон Вліссідес.
- "Head First Design Patterns" - Ерік Фрімен, Елізабет Фрімен, Кеті С'єра, Берт Бейтс.
- "Patterns of Enterprise Application Architecture" - Мартін Фаулер.