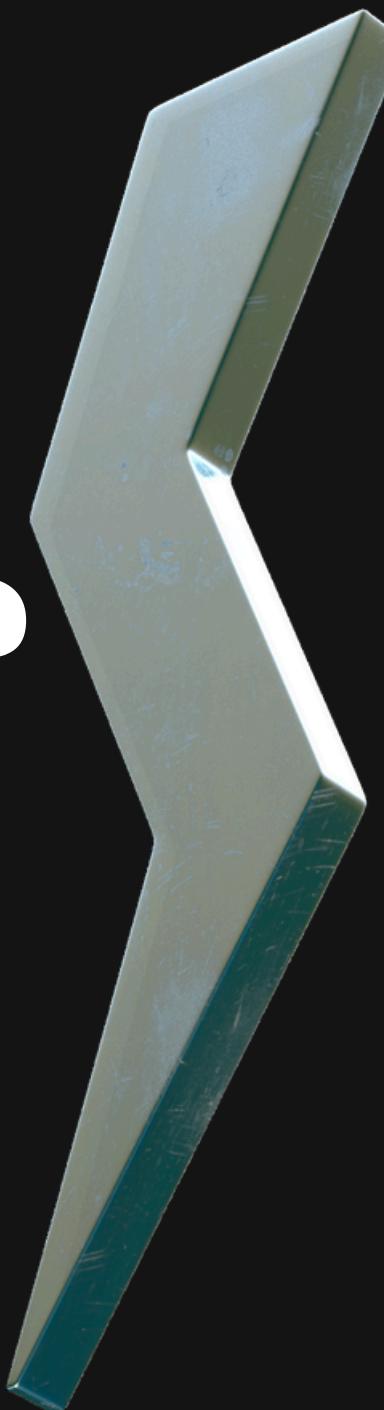


Wzorzec projektowy:

Obserwator



Czym jest wzorzec Obserwator?



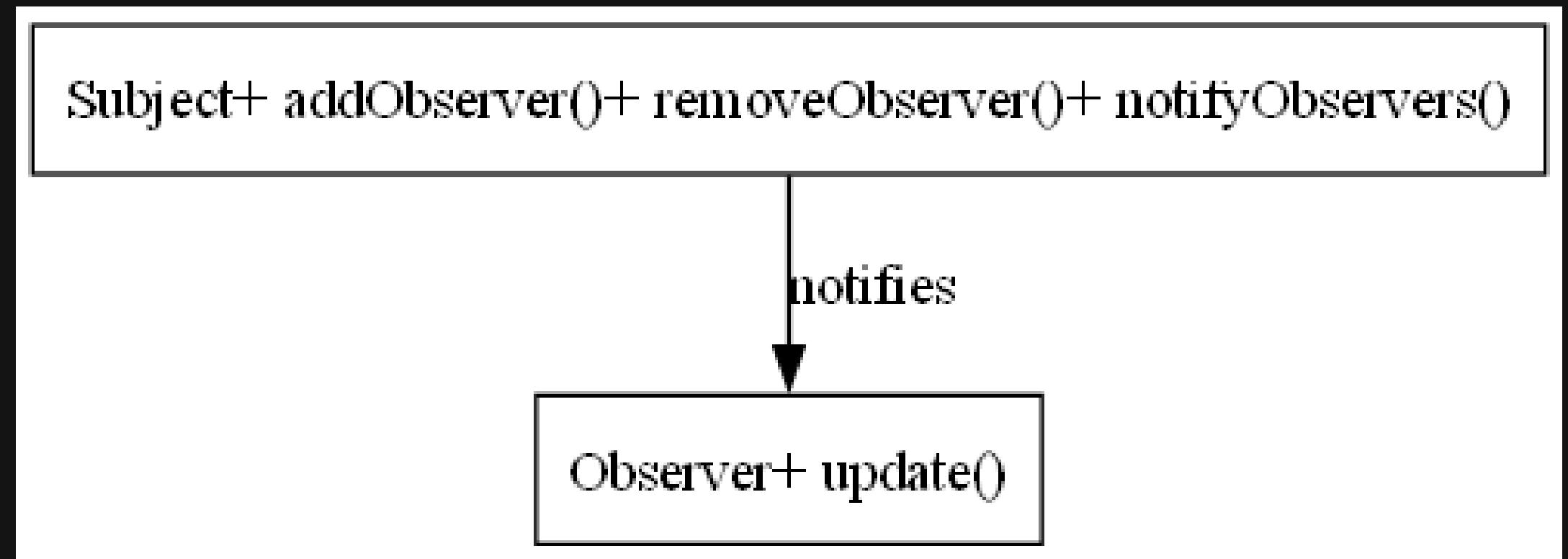
Wzorzec Obserwator umożliwia reagowanie wielu obiektów na zmiany w stanie jednego obiektu

W programowaniu jest to bardzo przydatne, gdy potrzebujemy informować różne części systemu o zmianach w jednym miejscu.

Przykłady z życia: newslettery (informowanie subskrybentów o nowościach), powiadomienia w aplikacjach, systemy monitorujące.

Bez tego wzorca wiele aplikacji musiałoby działać w sposób mniej modularny, co utrudniałoby rozwój i utrzymanie systemów.

Opis wzorca Obserwator

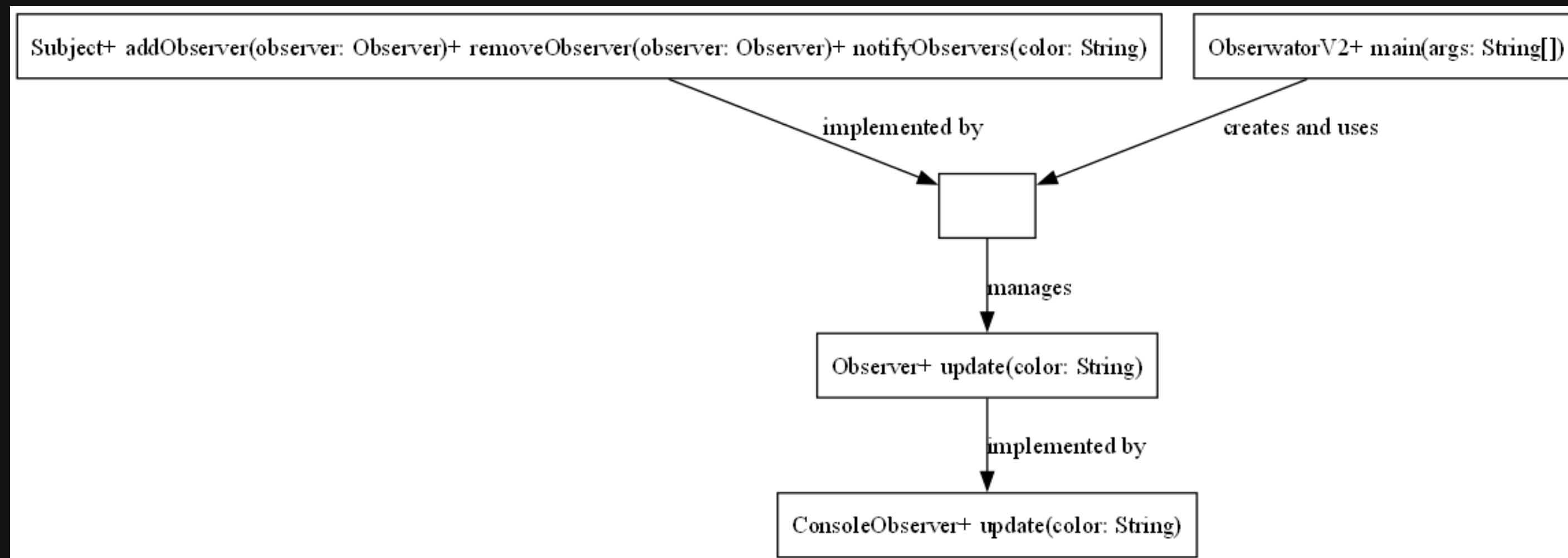


- Wzorzec Obserwator pozwala na ustanowienie zależności "jeden do wielu" między obiekty.
- Gdy stan jednego obiektu (Subject) ulega zmianie, automatycznie powiadomiani są wszyscy zainteresowani obserwatorzy (Observers).
- Kluczowe elementy wzorca:
 - Subject: Obiekt zarządzający listą obserwatorów i powiadamiający ich o zmianach.
 - Observer: Obiekt, który reaguje na zmiany w obiekcie obserwowanym.

Taka struktura pozwala na elastyczne zarządzanie powiadomieniami w aplikacjach, redukując konieczność ręcznego odpytywania obiektów o zmiany.

Struktura wzorca Obserwator

- Diagram UML przedstawia, jak poszczególne klasy i interfejsy są ze sobą połączone.
- Klasa Subject zarządza listą Observer i powiadamia je o zmianach za pomocą metody notifyObservers.
- Klasa Observer implementowana jest przez różne konkretne klasy, np. ConsoleObserver.
- W moim przykładowym projekcie ColorChangingLabel implementuje Subject, powiadamiając obserwatora o zmianie koloru.



Przykład: Zmiana koloru etykiety w GUI

- W projekcie użytkownik ma możliwość zmiany koloru etykiety w aplikacji GUI za pomocą przycisków.
- Każda zmiana koloru etykiety powoduje powiadomienie obserwatora (konsoli), który informuje o nowym kolorze.
- Taka struktura pozwala na łatwe dodawanie nowych obserwatorów, które mogą reagować na zmiany koloru (np. zapis do pliku lub aktualizacja innego widoku).



```
// redButton.addActionListener(e -> label.changeColor(Color.RED));

// public void update(String color) {
//     System.out.println("Kolor etykiety został zmieniony na: " + color);
// }
```

Zalety wzorca Obserwator



MODULARNOŚĆ:

Oddzielenie logiki zarządzania powiadomieniami od logiki reagowania na zmiany.

ELASTYCZNOŚĆ

Możliwość łatwego dodawania nowych obserwatorów bez modyfikacji kodu obiektu obserwowanego.

PRZYDATNOŚĆ W RZECZYWISTOŚCI:

Wzorzec jest używany w wielu aplikacjach, np. w systemach subskrypcji, aplikacjach do zarządzania danymi w czasie rzeczywistym, oraz w aplikacjach GUI.

Dzięki wzorcowi aplikacje stają się bardziej skalowalne i łatwiejsze w utrzymaniu.

Wady wzorca Obserwator



TRUDNOŚCI W DEBUGOWANIU:

Jeśli lista obserwatorów jest długa lub złożona, może być trudno śledzić, które obserwatory są powiadamiane.

PROBLEMY Z WYDAJNOŚCIĄ:

Gdy liczba obserwatorów rośnie, powiadamianie wszystkich może powodować opóźnienia.

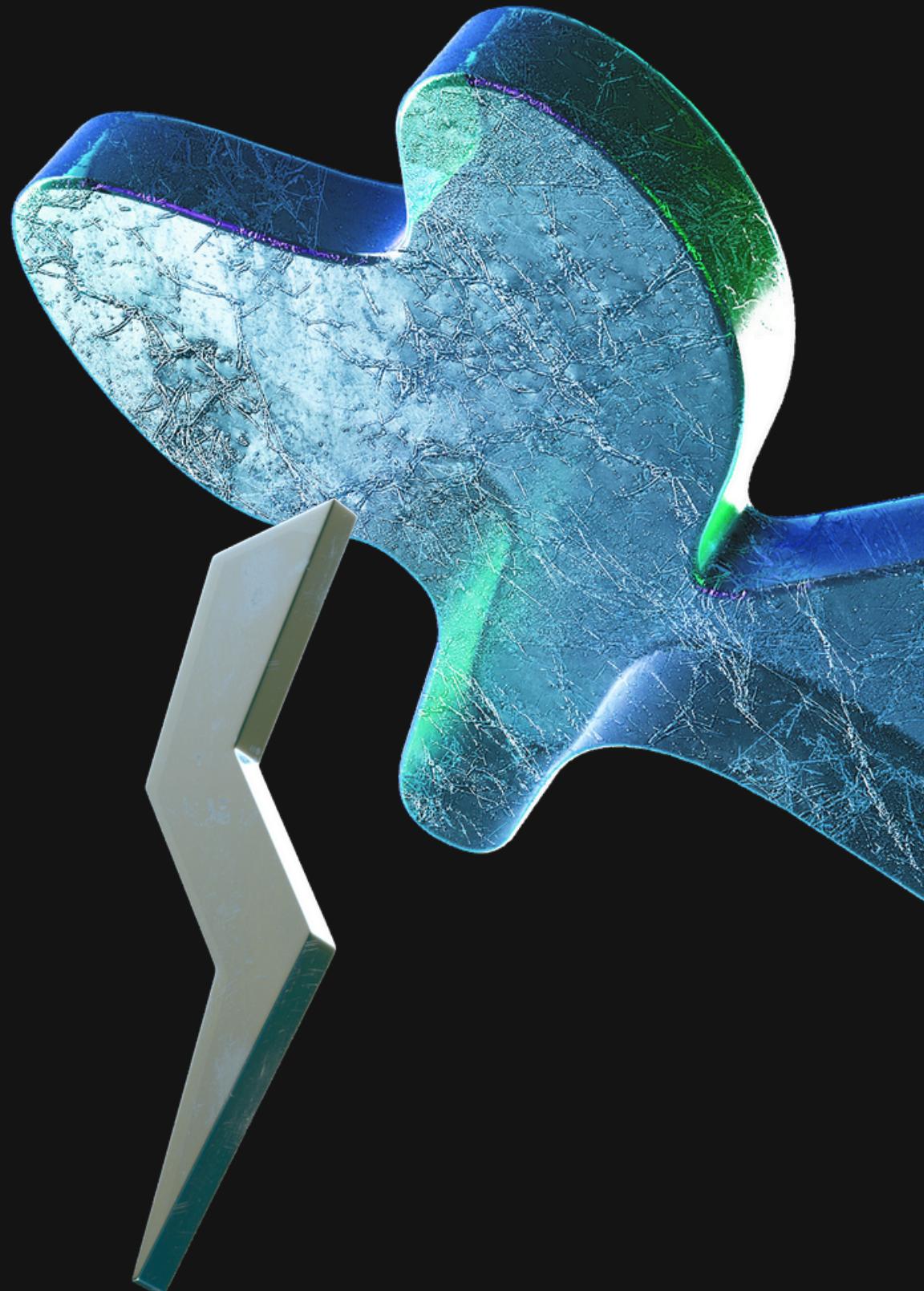
POTENCJALNE PROBLEMY Z ZARZĄDZANIEM PAMIĘCIĄ:

Jeśli obserwatorzy nie zostaną poprawnie usunięci z listy, może dojść do wycieków pamięci.

Wzorzec może być niepotrzebnie skomplikowany w przypadku prostych aplikacji.

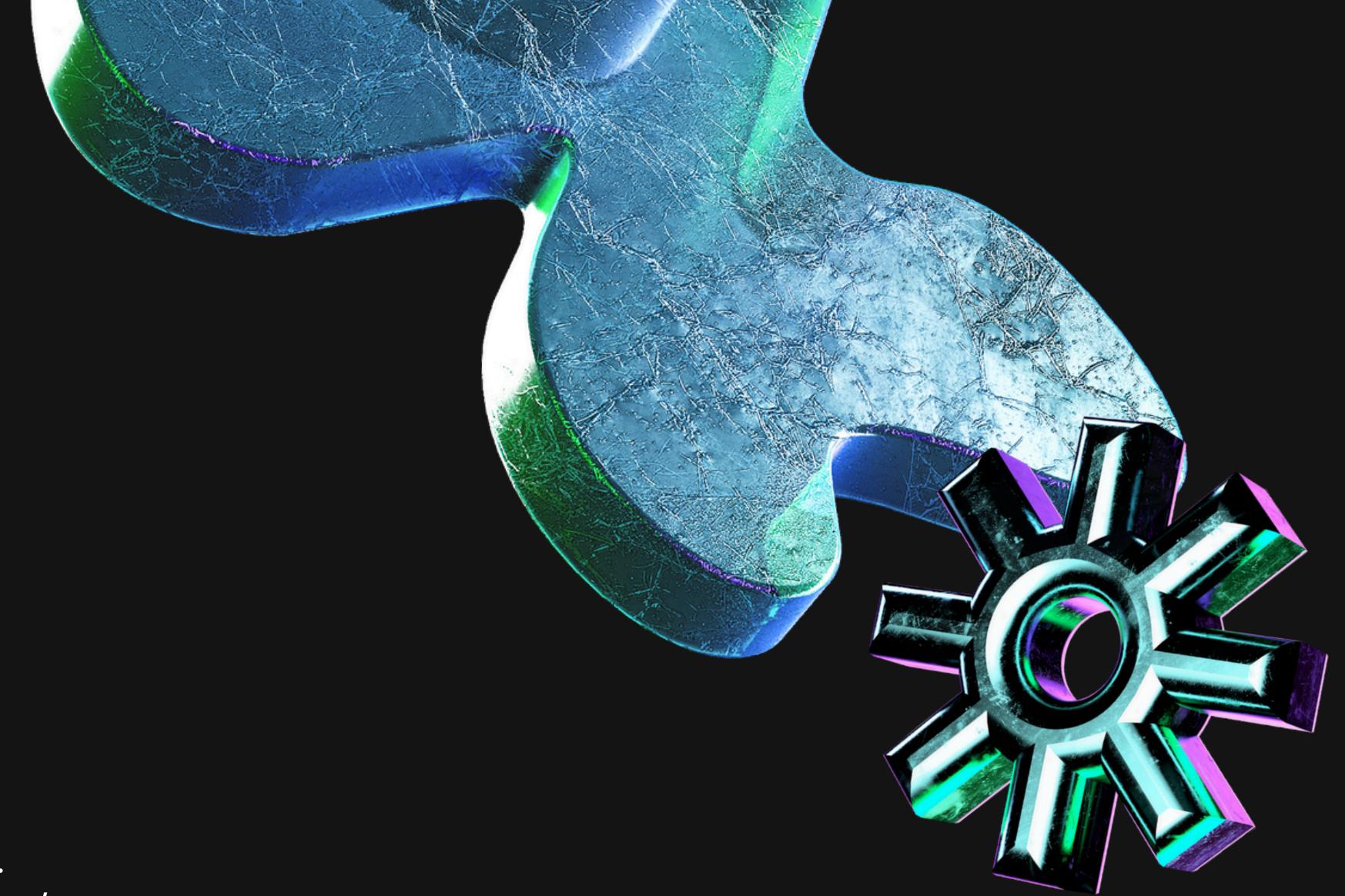
Podsumowanie

- Wzorzec Obserwator jest kluczowym rozwiązaniem, gdy chcemy zrealizować zależność "jeden do wielu" w aplikacjach.
- Pozwala na reakcję wielu obiektów na zmiany w jednym obiekcie, co poprawia modularność i elastyczność systemów.
- Jest to uniwersalne rozwiązanie, które znajduje zastosowanie w różnych dziedzinach, od aplikacji desktopowych po systemy rozproszone.
- Warto stosować ten wzorzec w sytuacjach, gdy różne części aplikacji muszą być informowane o zmianach stanu.



Źródła

- Dokumentacja Java: Obsługa zdarzeń i wzorce projektowe.
- <https://refactoring.guru/pl/design-patterns/observer/java/example>
- <https://www.baeldung.com/java-observer-pattern>
- <https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/>



Dziękuję za
uwagę

