Schematy blokowe – wprowadzenie do algorytmów

Krzysztof Gębicz

Cel lekcji

- zrozumie, czym są schematy blokowe,
- pozna ich znaczenie w programowaniu,
- nauczy się rozpoznawać podstawowe symbole,
- pozna zasady budowy schematu krok po kroku,
- będzie potrafił czytać proste schematy i analizować ich logikę.

Co to jest schemat blokowy?

- Schemat blokowy to graficzne przedstawienie algorytmu lub procesu.
- Algorytm = zbiór kroków prowadzących do rozwiązania problemu.
- Zamiast długiego opisu słownego używamy symboli graficznych i strzałek.
- Dzięki temu przedstawiamy algorytm w sposób jasny, uporządkowany i łatwy do odczytania.

Dlaczego schematy blokowe są ważne?

- Ułatwiają zrozumienie problemu – widzimy cały proces w jednym miejscu.
- Porządkują myślenie – zmuszają do uporządkowania kroków w logicznej kolejności.
- Pomagają w komunikacji – różne osoby (uczniowie, programiści, menedżerowie) szybciej zrozumieją schemat niż opis słowny.
- Redukują ryzyko błędów – łatwiej dostrzec brakujący krok lub błędny warunek.

Dlaczego schematy blokowe są ważne?

- Przygotowują do pisania kodu – schemat blokowy to pierwszy krok przed zapisaniem algorytmu w języku programowania.
- Uniwersalność – stosowane w informatyce, biznesie, inżynierii, edukacji.
- Przystępność – zrozumiały nawet dla osób, które nie znają programowania.

Zastosowania w praktyce

- Informatyka – planowanie programów, wizualizacja algorytmów.
- Inżynieria – opisywanie procesów produkcyjnych i technologicznych.
- Biznes – przedstawianie procedur, np. proces obsługi klienta.
- Administracja – wizualizacja dokumentów i obiegu informacji.
- Edukacja – nauka logicznego myślenia i analizowania problemów.

Symbol 1 – Owal (Start/Stop)

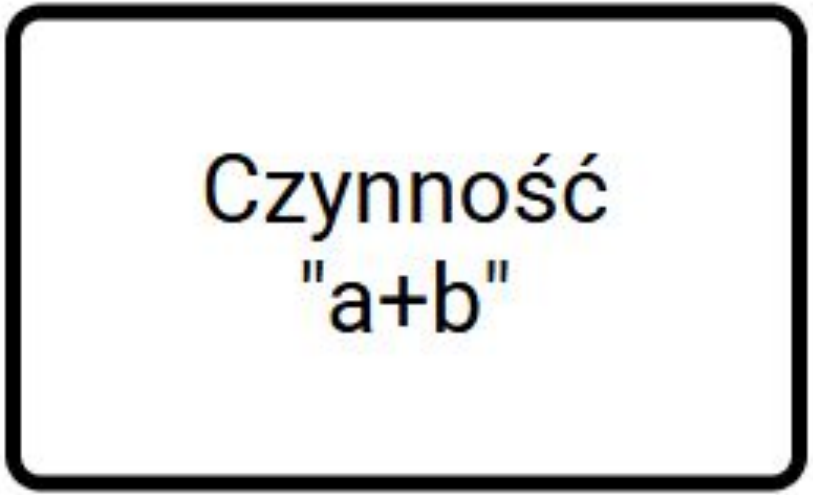
- Wygląd: elipsa lub owal.
- Znaczenie: początek lub koniec algorytmu.
- Ważne zasady:
  - Każdy schemat musi mieć dokładnie jeden punkt Start i przynajmniej jeden punkt Stop.
  - Owal nie może pojawiać się w środku schematu jako „czynność” – ma wyłącznie sygnalizować początek lub zakończenie.

Symbol 1 – Owal (Start/Stop)

Symbol 2 – Prostokąt (Czynność)

- Wygląd: prostokąt.
- Znaczenie: krok, w którym wykonywana jest czynność, obliczenie lub operacja.
- Wskazówki:
  - Opisy powinny być krótkie i jednoznaczne.
  - Prostokąty są najczęściej używanymi symbolami w schematach blokowych.

Symbol 2 – Prostokąt (Czynność)



Symbol 3 – Równoległobok (Wejście/Wyjście)

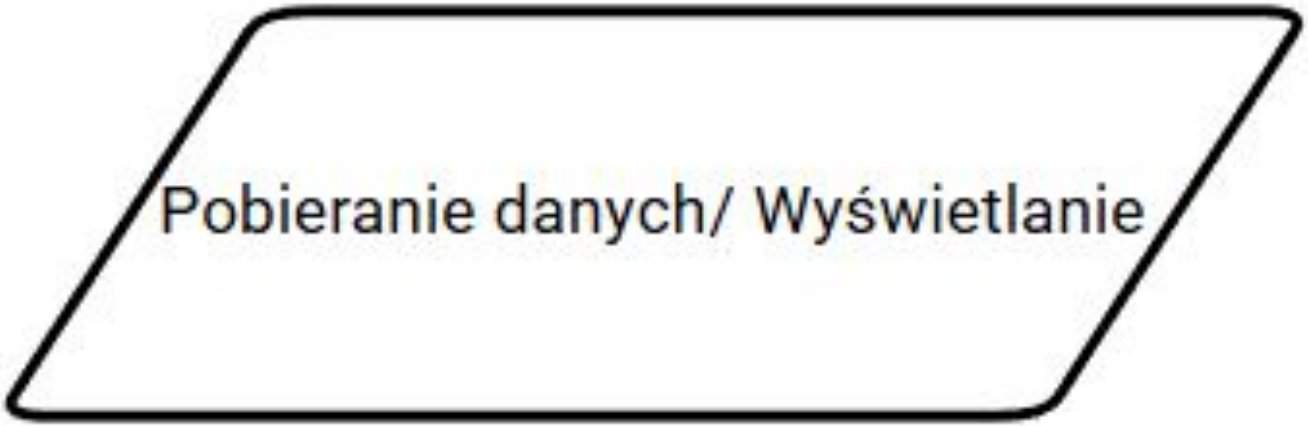
Wygląd: równoległobok, często lekko pochylony.

Znaczenie: interakcja programu z użytkownikiem – pobieranie danych lub wyświetlanie wyników.

Przykłady użycia:

- „Podaj liczbę n” (wejście)
- „Wyświetl wynik” (wyjście)

Symbol 3 – Równoległobok (Wejście/Wyjście)



Symbol 4 – Romb (Warunek/Decyzja)

- Wygląd: romb, z którego wychodzą dwie lub więcej strzałki.
- Znaczenie: miejsce podjęcia decyzji – pytanie, które daje odpowiedź TAK/NIE lub podobną.
- Przykłady:
  - „Czy liczba > 0?”
  - „Czy masz wodę?”
- Uwagi: każda odpowiedź prowadzi do innej ścieżki algorytmu.

Symbol 4 – Romb (Warunek/Decyzja)

Symbol 5 – Strzałki (Kolejność działań)

- Wygląd: linie z grotem.
- Znaczenie: określają kierunek przepływu algorytmu.
- Zasady:
  - Schemat czytamy od góry do dołu, od lewej do prawej.
  - Strzałki nie powinny się krzyżować – w dużych schematach używa się łączników.
  - Strzałka zawsze musi prowadzić do kolejnego symbolu.

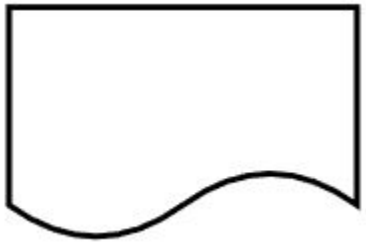
Symbol 5 – Strzałki (Kolejność działań)

Dodatkowe symbole

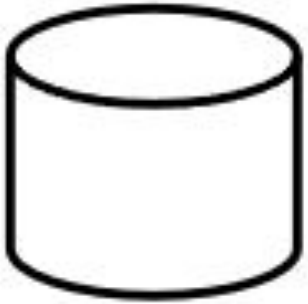
1. Dokument – prostokąt z falowaną dolną linią, symbolizuje wytworzenie dokumentu lub zapis informacji.
2. Baza danych – cylinder, oznacza zapis/odczyt danych w pamięci lub w bazie.

Dodatkowe symbole

1. Dokument



2. Baza danych



Zasady budowania schematów

- Zawsze rozpoczynamy od Start i kończymy na Stop.
- Każdy symbol ma jedno wejście i jedno wyjście (z wyjątkiem rombu).
- Warunki muszą prowadzić do dwóch dróg: TAK i NIE.
- Algorytm powinien być kompletny – każda możliwa ścieżka prowadzi do końca.

Zasady budowania schematów

- Opisy w symbolach mają być krótkie, precyzyjne i jednoznaczne.
- Strzałki nie powinny się plątać ani krzyżować.
- Schemat ma być czytelny – układ od góry do dołu i od lewej do prawej.
- Nie mieszać różnych poziomów szczegółowości – jeśli proces jest zbyt duży, dzielimy go na mniejsze schematy.

Jak powstaje schemat?

1. Zdefiniuj problem i wypisz algorytm słownie.
2. Podziel czynności na: operacje, wejścia/wyjścia, warunki.
3. Zapisz je w odpowiednich symbolach.
4. Połącz symbole strzałkami w logicznej kolejności.
5. Sprawdź, czy każda ścieżka ma początek i koniec.

Przykład teoretyczny 1 – Dodawanie dwóch liczb

Algorytm krok po kroku:

1. START
2. Wprowadź dwie liczby
3. Oblicz sumę
4. Wyświetl wynik
5. STOP

Schemat: prosty układ 5 symboli – owal, równoległobok, prostokąt, równoległobok, owal.

Przykład teoretyczny 2 – Sprawdzenie parzystości

Algorytm krok po kroku:

1. START
2. Wprowadź liczbę
3. Sprawdź: czy liczba mod 2 = 0?
4. Jeśli TAK → „Liczba parzysta”
5. Jeśli NIE → „Liczba nieparzysta”
6. STOP

Schemat: romb jako warunek, dwie ścieżki TAK/NIE prowadzące do różnych wyników

Przykład teoretyczny 3 – Proces codzienny

Algorytm „Robienie herbaty”:

1. START
2. Czy mam wodę? Jeśli NIE → STOP.
3. Zagotuj wodę.
4. Włóż herbatę do kubka.
5. Zalej wrzątkiem.
6. Czy chcę cukier? Jeśli TAK → dodaj cukier.
7. STOP

Schemat: przykład nienumeryczny, łatwy do wyobrażenia i zrozumienia.

Podsumowanie

1. Schemat blokowy = graficzne przedstawienie algorytmu.
2. Najważniejsze symbole: owal (Start/Stop), prostokąt (Czynność), równoległobok (Wejście/Wyjście), romb (Warunek), strzałki (Przepływ).
3. Dodatkowe symbole: łącznik, dokument, baza danych, komentarz.
4. Zasady: prostota, czytelność, poprawna logika, jednoznaczne opisy.

Pierwsze kroki java

Krzysztof Gębicz

Deklaracja i inicjalizacja zmiennej

```
int liczba;        // deklaracja

liczba = 10;     // inicjalizacja


int wiek = 16;    //deklaracja +
inicjalizacja
```

Deklaracja = utworzenie zmiennej

Inicjalizacja = nadanie jej wartości

Przykład kodu

```
public class ZmienneDemo {

    public static void main(String[] args)
{

        int liczba; // Deklaracja zmiennej
(powstaje zmienna, ale jeszcze nie ma
wartości)

        liczba = 10; // Inicjalizacja
(nadanie pierwszej wartości)

        int wiek = 16; // Deklaracja i
inicjalizacja w jednym kroku

        wiek = 18; // Zmienna może później
przyjąć inną wartość

    }

}
```

Typy zmiennych w Javie

```
int wiek = 16;      // liczby całkowite

double temperatura = 36.6;     // liczby
zmiennoprzecinkowe

String tekst= "tekst";         // tekst

final double PI = 3.14159;     // stała
(niezmienna)
```

- `int` – liczby całkowite (np. -3, 0, 42).
- `double` – liczby zmiennoprzecinkowe (np. 3.14, 36.6).
- `String` – tekst (napisy w `" "`).
- `final` – tworzy stałą (wartość nie może się zmienić)

Przykład kodu

```
public class TypyZmiennych {

    public static void main(String[] args) {

        int wiek = 16; // liczba całkowita (int)


        double temperatura = 36.6; // liczba zmiennoprzecinkowa (double)


        String tekst= "tekst";          // tekst (String)


      final double PI = 3.14159;     // stała (final)


    }

}
```

Wyświetlanie

```
System.out.println("Witaj w
Javie!");  // wyświetla i
przechodzi do nowej linii

System.out.print("Hello ");
// wyświetla, bez przechodzenia
do nowej linii

System.out.printf("Liczba: %d\n",
10); // wyświetlanie formatowane
```

`System.out.println()` – używamy do większości wyświetleń, np. komunikatów do użytkownika.

`System.out.print()` – przydatne, gdy chcemy kontynuować w tym samym wierszu.

`System.out.printf()` – gdy potrzebujemy formatować tekst, np. liczby z określoną liczbą miejsc po przecinku.

Przykład kodu

```
public class PokazWyświetlania {

    public static void main(String[] args) {

        // 1. Wyświetlanie z przejściem do nowej linii

        System.out.println("Witaj w Javie!");

        System.out.println("To jest kolejna linia.");


        // 2. Wyświetlanie w tym samym wierszu

        System.out.print("Hello ");

        System.out.print("Java ");

        System.out.print("World!\n");  // \n dodaje nową linię ręcznie


            }

}
```

Przykład kodu

```
public class PokazWyświetlania {
    public static void main(String[] args) {
3. Wyświetlanie formatowane
        int liczba = 42;
        double wynik = 3.14159;

        // %d - liczba całkowita, %f - liczba zmiennoprzecinkowa, %.2f - 2
miejsca po przecinku
        System.out.printf("Liczba całkowita: %d\n", liczba);
        System.out.printf("Liczba zmiennoprzecinkowa: %.2f\n", wynik);

        // Możemy łączyć tekst i zmienne w jednej linijce
        System.out.printf("Witaj,twoja liczba to”+liczba+”a wynik to %.2f\n");
    }
}
```

Operacje na zmiennych

```
int a = 5;

int b = 2;

int suma = a + b;

System.out.println("Suma = " + suma);
```

- Deklaracja + inicjalizacja – np. `int a = 5;` (Tworzymy zmienną i od razu nadajemy jej wartość.)
- Użycie operatora `+` – dodawanie dwóch wartości (`a + b`).

Przykład kodu

```java
public class OperacjeNaZmiennych {

    public static void main(String[] args) {

        // 1. Deklaracja zmiennych i inicjalizacja

        int a = 5;    // Tworzymy zmienną całkowitą 'a' i nadajemy jej wartość 5

        int b = 2;    // Tworzymy zmienną całkowitą 'b' i nadajemy jej wartość 2

        // 2. Operacje na zmiennych

        int suma = a + b;          // Dodawanie: wynik dodawania 'a' i 'b' zapisujemy w zmiennej 'suma'

        // 3. Wyświetlanie wyników

        System.out.println("Wartość a = " + a);

        System.out.println("Wartość b = " + b);

        System.out.println("Suma = " + suma);

        // 4. Możesz też pokazać łączenie operacji w jednej linii

        System.out.println("a + b * 2 = " + (a + b * 2));

    }

}
```

Przykład kodu

```
public class Main {

    public static void main(String[] args) {

        int a = 5;

        int b = 3;


        System.out.println("Wartość a = " + a + ", wartość b = " + b + ", suma = " + (a + b));

    }

}
```

Łączenie tekstu i zmiennych – używamy operatora `+`.

- `"tekst" + zmienna + "więcej tekstu"` → scala wszystko w jeden ciąg znaków.

Wyniki działań matematycznych w środku tekstu – trzeba użyć nawiasów: `(a + b)`

- Bez nawiasów byłoby: `"a + b"` traktowane jako tekst.

System.out.println(...) – wyświetla wszystko i przechodzi do nowej linii.

Zadania Utrwalające

Zadanie 1

Zadeklaruj zmienną `int wiek` i przypisz jej wartość.

Zadeklaruj zmienną `String imie` i przypisz swoje imię.

Wypisz w konsoli: `"Cześć, mam XX lat i nazywam się YYY."`

Podpowiedź: użyj `System.out.println(...)` i operatora `+` do łączenia tekstu z wartościami zmiennych.

Zadanie 2

1. Zadeklaruj dwie zmienne `int a = 7` i `int b = 4`.

2. Wypisz w konsoli w jednej linii:

   a = 7, b = 4, suma = 11, różnica = 3

3. Użyj operatorów `+` i nawiasów, żeby obliczyć sumę i różnicę w środku tekstu.

Zadanie 3

1. Zadeklaruj zmienną `double temperatura` i przypisz jej wartość np. 23.5.

2. Zadeklaruj zmienną `String dzien` i przypisz np. `"Środa"`.

3. Wypisz w konsoli: `"Dziś jest PONIEDZIAŁEK, a temperatura wynosi 23.5 stopni."`

4. Podpowiedź: użyj `System.out.println(...)` i operatora `+` do łączenia tekstu z wartościami zmiennych.

Zadanie 4

1. Zadeklaruj dwie zmienne `int x = 10` i `int y = 3`.
2. Oblicz w osobnych zmiennych:
   - `iloczyn = x * y`
   - `iloraz = x / y`
   - `reszta = x % y`
3. Wypisz w konsoli w jednej linii:
   `"x = 10, y = 3, iloczyn = 30, iloraz = 3, reszta = 1"`
4. Podpowiedź: użyj operatorów `+` i nawiasów, aby wstawiać wyniki obliczeń w tekst.

Sortowanie

Co to jest sortowanie?

Sortowanie to proces układania danych w określonym porządku – rosnącym lub malejącym.
Przykład: uporządkowanie listy ocen, nazwisk, numerów.

Dlaczego sortowanie jest ważne

- Ułatwia przeszukiwanie tablic,
- Porządkuje dane do analizy,
- W Javie – często używane w listach i tablicach.

Typy sortowania

1. Wewnętrzne – sortowanie w pamięci (tablice, ArrayList),

2. Zewnętrzne – duże zbiory danych (pliki, bazy danych).

Podejścia do sortowania

- Porównawcze – Bubble, Selection, Quick, Merge, Insertion
- Nieporównawcze – Counting Sort, Radix Sort

Sortowanie bąbelkowe (Bubble Sort)

Nazwa pochodzi od „bąbelków” – największe elementy wypływają na górę listy.

Idea

Porównuj sąsiednie elementy:

- jeśli są w złej kolejności – zamień je miejscami,
- powtarzaj aż do końca listy.

Przykład działania

Dane: [5, 3, 8, 4, 2]

1. przebieg: [3, 5, 8, 4, 2]
2. przebieg: [3, 5, 4, 8, 2]
3. przebieg: [3, 4, 5, 2, 8]
4. przebieg: [3, 4, 2, 5, 8]
5. przebieg: [3, 2, 4, 5, 8]
6. przebieg: [2, 3, 4, 5, 8]

Zalety i wady

Zalety:

- Prosty do zrozumienia i napisania,
- Dobry do nauki.

Wady:

- Bardzo wolny przy dużych zbiorach danych.

Przykład

Selection Sort

Sortowanie przez wybór. W każdym przebiegu znajdowany jest najmniejszy element i przenoszony na początek tablicy.

Idea

- Dzielimy tablicę na część posortowaną i nieposortowaną.
- W nieposortowanej części szukamy najmniejszego elementu.
- Zamieniamy go z pierwszym elementem nieposortowanej części.
- Powtarzamy dla pozostałej części tablicy.

Przykład działania

Tablica: [5, 3, 8, 4, 2]

1. Znajdź min w [5,3,8,4,2] → 2 → zamień z 5 → [2,3,8,4,5]
2. Znajdź min w [3,8,4,5] → 3 → już na miejscu
3. Znajdź min w [8,4,5] → 4 → zamień z 8 → [2,3,4,8,5]
4. Znajdź min w [8,5] → 5 → zamień → [2,3,4,5,8]

Zalety i wady

Zalety:

- prosty
- przewidywalny czas działania
- niewiele zamian.

Wady:

- powolny dla dużych zbiorów ($O(n^2)$)
- nieefektywny w porównaniu do szybszych algorytmów

Przykład

Insertion Sort

Sortowanie przez wstawianie. Wstawia każdy element w odpowiednie miejsce w już posortowanej części tablicy.

Idea

- Pierwszy element traktujemy jako posortowany.
- Kolejne elementy wstawiamy w odpowiednie miejsce w posortowanej części.
- Przesuwamy większe elementy w prawo, aby zrobić miejsce dla nowego elementu.

Przykład działania

Tablica: [5, 3, 8, 4, 2]

1. Posortowana [5], wstaw 3 → [3,5]
2. Wstaw 8 → [3,5,8]
3. Wstaw 4 → [3,4,5,8]
4. Wstaw 2 → [2,3,4,5,8]

Zalety i wady

Zalety:

- szybki dla prawie posortowanych danych
- prosty do implementacji.

Wady:

- $O(n^2)$ w najgorszym przypadku
- nieefektywny dla dużych zbiorów.

Przykład

Quick Sort

Sortowanie szybkie (Quick Sort). Dzieli tablicę na mniejsze i większe elementy względem pivotu.

Idea

- Wybierz pivot (np. środkowy element).
- Podziel tablicę na dwie części: elementy mniejsze od pivot i większe od pivot.
- Rekurencyjnie sortuj obie części.
- Połącz wyniki.

Przykład działania

Tablica: [5,3,8,4,2], pivot = 5

- Mniejsze: [3,4,2]
- Większe: [8]
- Rekurencyjnie sortujemy [3,4,2] → [2,3,4]
- Łączymy

Zalety i wady

Zalety:

- bardzo szybki O(n log n) w średnim przypadku
- bardzo wydajny.

Wady:

- $O(n^2)$ w najgorszym przypadku (np. tablica już posortowana i złe pivoty).

Przykład

Merge Sort

Sortowanie przez scalanie. Dzieli tablicę na pół, sortuje rekurencyjnie i scala w jedną posortowaną tablicę.

Idea

- Dziel tablicę na mniejsze części aż do jednoelementowych.
- Scalaj po dwie części, porównując elementy i tworząc posortowaną tablicę.

Przykład działania

Tablica: [5,3,8,4,2]

- Dzielimy: [5,3,8] i [4,2]
- Dzielimy dalej: [5,3] i [8] → [5],[3]
- Scalanie [3,5] + [8] → [3,5,8]
- [4,2] → [2,4]
- Scalamy [3,5,8] + [2,4] → [2,3,4,5,8]

Zalety i wady

Zalety:

- stabilny
- zawsze O(n log n)
- dobrze działa dla dużych zbiorów.

Wady:

- wymaga dodatkowej pamięci (tablice pomocnicze).

Przykład

Algorytmy

Krzysztof Gębicz

Lista kroków (sekwencja czynności)

Lista kroków to liniowy, uporządkowany opis algorytmu jako ciągu pojedynczych, dobrze zdefiniowanych operacji, wykonywanych jedna po drugiej.

Cechy i warunki poprawności

- Porządek wykonania ma znaczenie — zmiana kolejności może zmienić wynik.
- Kroki powinny być atomowe — każdy krok to pojedyncza, jasna operacja.
- Brak rozgałęzień — w klasycznej liście nie ma warunków ani pętli (choć można je zapisać opisowo).
- Kompletność — lista od Start do Stop bez luk logicznych.

Zalety

- Prostota i przejrzystość: łatwe do zrozumienia dla początkujących.
- Szybkie planowanie: dobre na wczesnym etapie projektowania — łatwo wypisać kroki „co robić”.
- Uniwersalność: można zapisać w formie numerowanej, notatek albo prostych schematów blokowych.
- Niska bariera wejścia: nie wymaga znajomości składni ani struktur programistycznych.

Wady

- Ograniczona ekspresyjność: słabo nadaje się do opisu decyzji i powtarzających się operacji.
- Trudność skalowania: przy większym problemie lista staje się długa i mało czytelna.
- Brak formalności: trudniej z niej wygenerować kod czy wykonać analizę złożoności.

Typowe błędy i pułapki

- Zawieranie w jednym kroku zbyt wielu działań (brak atomowości).
- Pomijanie warunków brzegowych (np. co zrobić gdy wejście jest puste).
- Mieszanie poziomów abstrakcji (czasem krok jest zbyt szczegółowy, innym razem zbyt ogólny).

Kiedy stosować

- Do prostych procedur (np. instrukcje obsługi, checklisty).
- Na początek projektowania algorytmu przed dodaniem warunków i pętli.

Przykład

Ćwiczenia

Drzewa decyzyjne — Definicja i idea

Drzewo decyzyjne modeluje proces podejmowania decyzji: węzły to warunki, gałęzie — wyniki warunków, liście — decyzje lub akcje końcowe.

Struktura drzewa

- Korzeń: pierwszy warunek/pytanie.
- Węzły wewnętrzne: kolejne warunki (mogą być binarne lub wielowartościowe).
- Gałęzie: alternatywy wynikające z warunku.
- Liście: końcowe rezultaty lub działania.

Zalety

- Czytelne modelowanie decyzji: łatwo zobaczyć, które warunki prowadzą do danej decyzji.
- Naturalne odwzorowanie instrukcji warunkowych (if/else, switch) w kodzie.
- Analiza ścieżek: można łatwo policzyć możliwe przypadki i przewidzieć skutki różnych warunków.
- Modularność: poszczególne poddrzewa można rozpatrywać oddzielnie.

Wady

- Eksplozja kombinacji: liczba liści rośnie szybko ze wzrostem warunków (może być wykładnicza), co utrudnia czytelność i analizę.
- Trudność optymalizacji: nie zawsze łatwo skrócić drzewo bez utraty ważnych przypadków.
- Specyficzność do decyzji: mniej przydatne do opisu skomplikowanych obliczeń lub algorytmów iteracyjnych.

Dobre praktyki konstrukcji

- Gałęzie wyczerpujące i wzajemnie wykluczające — z jednego węzła pokryć wszystkie możliwe przypadki bez nakładania się.
- Ograniczanie głębokości — rozbijaj głębokie poddrzewa na osobne moduły lub funkcje.
- Używanie etykiet i komentarzy — opisuj kryteria decyzyjne, aby ułatwić interpretację.
- Normalizacja warunków — porządkowanie warunków według priorytetu lub prawdopodobieństwa.

Typy zastosowań

- Systemy ekspertowe i klasyfikatory decyzyjne (w analityce / ML — chociaż tam drzewa są dopasowywane statystycznie i mają dodatkowe własności).

- Procedury obsługi klienta, reguły biznesowe, diagnostyka.

Rozważania wydajnościowe i złożoność

- Złożoność odwzorowania: liczba możliwych ścieżek = iloczyn rozgałęzień na kolejnych poziomach (dla prostych binarnych warunków — do 2^d, gdzie d to głębokość).

- Analiza przypadków brzegowych: konieczność upewnienia się, że każda możliwa kombinacja prowadzi do liścia lub dalszego przetwarzania.

Przykład

Ćwiczenia