Zagadnienie do egzaminu inżynierskiego 24/25

Semestr V/Semestr VII

Metody Numeryczne

1. Rozwiązywanie równań nieliniowych

Równania nieliniowe to równania, w których niewiadoma występuje w funkcji nieliniowej, np. f(x)=0, gdzie f(x) może zawierać potęgi, funkcje trygonometryczne, wykładnicze itp. W praktyce analityczne rozwiązanie takich równań często nie jest możliwe, dlatego stosuje się metody numeryczne.

Metody numeryczne rozwiązywania równań nieliniowych:

1. **Metoda bisekcji**:
   - Polega na podziale przedziału, w którym występuje pierwiastek, na pół.
   - Działa, gdy funkcja zmienia znak (f(a)*f(b)<0).
   - Zbieżność liniowa, ale gwarantuje znalezienie pierwiastka.
2. **Metoda Newtona-Raphsona**:
   - Iteracyjna metoda oparta na liniowej aproksymacji f(x).
   - Formuła: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.
   - Zbieżność kwadratowa, wymaga znajomości pochodnej f ’(x).
3. **Metoda siecznych**:
   - Modyfikacja metody Newtona, gdzie zamiast pochodnej stosuje się różnicę skończoną.
   - Formuła: $x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$
   - Brak potrzeby obliczania pochodnej.
4. **Metoda stycznych i regula falsi**:
   - Wyznacza pierwiastek iteracyjnie, zastępując funkcję linią prostą.
   - Bardziej stabilna niż metoda Newtona dla niektórych funkcji.

2. Wielomiany interpolacyjne

Interpolacja to proces znajdowania wielomianu P(x), który przechodzi przez zadane punkty (x_i,y_i). Interpolacja jest kluczowa w przybliżaniu funkcji i danych.

Metody interpolacji:

1. **Interpolacja Lagrange’a**:
   - Wielomian postaci: $P(x) = \sum_{i=0}^{n} y_i L_i(x)$, $L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$
   - Bezpośrednia, ale podatna na niestabilności numeryczne przy dużej liczbie węzłów.

2. **Interpolacja Newtona**:
    - Wielomian w postaci: $P(x)=a_0+a_1(x-x_0)+a_2(x-x_0)(x-x_1)+\ldots$
    - Współczynniki a_i obliczane za pomocą różnic dzielonych.
    - Lepsza stabilność niż metoda Lagrange'a.
3. **Sześcienne splajny**:
    - Wielomian stopnia 3 dla każdego przedziału między węzłami.
    - Gładkie przejście między przedziałami (ciągłość pochodnych).
    - Popularne w grafice komputerowej i modelowaniu.

3. Całkowanie numeryczne

Całkowanie numeryczne polega na aproksymacji całki oznaczonej $\int_a^b f(x)\,dx$, gdy nie można znaleźć rozwiązania analitycznego.

Podstawowe metody:

1. **Metoda prostokątów**:
    - Dzieli przedział [a,b] na n podprzedziałów.
    - Formuła: $\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(x_i)\Delta x$
    - Mało dokładna, zależy od wyboru punktu x_i
2. **Metoda trapezów**:
    - Przybliża obszar pod wykresem funkcji trapezami.
    - Formuła:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{2}\left[f(a) + \sum_{i=1}^{n-1} f(x_i) + f(b)\right]$$

3. **Kwadratura Simpsona**:
    - Przybliża funkcję wielomianami kwadratowymi.
    - Formuła:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left[f(a) + 4\sum_{odd\ i} f(x_i) + 2\sum_{even\ i} f(x_i) + f(\mathrm{b})\right]$$

    - Wysoka dokładność dla funkcji gładkich.
4. **Metody adaptacyjne**:
    - Dostosowują podział przedziału [a, b] do lokalnych zmian funkcji f(x), zwiększając dokładność.

4. Aproksymacja średniokwadratowa

Aproksymacja średniokwadratowa polega na znalezieniu funkcji f(x), która najlepiej przybliża dane w sensie minimalizacji sumy kwadratów odchyleń:

Błąd średniokwadratowy: $S = \sum_{i=1}^{n} [y_i - f(x_i)]^2$

Proces aproksymacji:

1. **Wybór modelu aproksymacji**:
   - Wielomian, funkcja trygonometryczna, eksponencjalna itp.
2. **Minimalizacja błędu S**:
   - Wyznaczanie współczynników funkcji przez rozwiązanie układów równań normalnych.
   - W przypadku liniowego modelu y=a+bx:
     a,b są wyznaczane przez metody regresji liniowej.
3. **Zastosowania**:
   - Analiza danych, przewidywanie trendów, przetwarzanie sygnałów.

5. Numeryczne rozwiązywanie równań różniczkowych

Równania różniczkowe opisują zmiany wielkości w czasie lub przestrzeni. Ich numeryczne rozwiązywanie jest kluczowe w naukach technicznych i przyrodniczych.

Podstawowe metody:

1. **Metoda Eulera**:
   - Najprostsza metoda iteracyjna.
   - Przybliżenie: $y_{n+1}=y_n+h\cdot f(x_n,y_n)$, gdzie h to krok czasowy.
2. **Metoda Rungego-Kutty**:
   - Zwiększa dokładność poprzez obliczanie dodatkowych punktów w kroku: $y_{n+1}=y_n+\frac{n}{6}(k_1+2k_2+2k_3+k_4)$, gdzie k_1,k_2,k_3,k_4 to pomocnicze wartości obliczane na podstawie funkcji f(x,y).
3. **Metody wielokrokowe**:
   - Wykorzystują wartości obliczone w poprzednich krokach.
   - Przykład: metoda Adamsa-Bashfortha.
4. **Metody różnic skończonych**:
   - Stosowane do równań różniczkowych cząstkowych (PDE).
   - Przybliżają pochodne za pomocą różnic skończonych, np.:

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1} - u_u}{\Delta x}$$

1. Wymienić i opisać różne typy błędów.
2. Omówić podstawowe metody rozwiązywania równań nieliniowych.
3. Przedstawić wady i zalety interpolacji wielomianowej Newtona i Lagrange'a.
4. Interpolacja wielomianowa a numeryczne obliczanie całek oznaczonych.
5. Opisać aproksymację średniokwadratową i porównać z interpolacją.
6. Omówić metody rozwiązywania równań różniczkowych Eulera i Rungego-Kutty oraz krótko opisać błędy w obu przypadkach.

1. Wymienić i opisać różne typy błędów

Błędy w obliczeniach numerycznych można podzielić na kilka głównych typów:

1. **Błędy zaokrągleń**:
   - Powstają z powodu ograniczonej liczby cyfr w reprezentacji liczb w komputerze.
   - Występują szczególnie w obliczeniach z liczbami zmiennoprzecinkowymi.
2. **Błędy obcięcia**:
   - Wynikają z przybliżenia rzeczywistego rozwiązania przez metodę numeryczną, np. skrócenia szeregu nieskończonego do skończonej liczby wyrazów.
3. **Błędy modelowania**:
   - Powstają, gdy rzeczywisty problem jest upraszczany w celu stworzenia modelu matematycznego.
4. **Błędy metody**:
   - Są związane z niedoskonałością zastosowanej metody numerycznej (np. ograniczoną dokładnością aproksymacji).
5. **Błędy danych wejściowych**:
   - Wynikają z niedokładności wprowadzanych danych.
6. **Błędy propagacji**:
   - Powstają na skutek kumulacji błędów w trakcie wieloetapowych obliczeń.
   -

7. Błąd bezwzględny

Błąd bezwzględny mierzy różnicę między wartością rzeczywistą (x_{true}) a wartością przybliżoną (x_{approx}):

Błąd bezwzględny=$|x_{true}-x_{approx}|$

Charakterystyka:

- Wartość absolutna eliminuje ujemne wyniki, co ułatwia interpretację.
- Wyraża różnicę w tych samych jednostkach, co mierzone wartości.
- Jest przydatny, gdy interesuje nas konkretna wielkość odchylenia.

Przykład:

Jeśli wartość rzeczywista to 10, a przybliżenie wynosi 9.8, to:

Błąd bezwzględny = $|10 - 9.8| = 0.2$

2. Błąd względny

Błąd względny mierzy wielkość błędu w stosunku do wartości rzeczywistej, co pozwala ocenić proporcjonalność odchylenia:

Błąd względny=$(|x_{true}-x_{approx}|)/|x_{true}|$

Charakterystyka:

- Wyrażany zwykle jako ułamek dziesiętny lub procent.
- Pozwala porównywać dokładność przybliżeń różnych wielkości.
- Przydatny, gdy ważne jest względne odchylenie, np. w przypadku dużych różnic w wielkościach danych.

Przykład:

Jeśli wartość rzeczywista to 10, a przybliżenie wynosi 9.8, to:

7. Błąd względny=|10−9.8| / 10 = 0.02 (2%)

2. Omówić podstawowe metody rozwiązywania równań nieliniowych

1. **Metoda bisekcji**:
   - Przybliżeniowa metoda dzielenia przedziału na pół, gdzie funkcja zmienia znak.
   - Zalety: pewność znalezienia pierwiastka.
   - Wady: powolna zbieżność.
2. **Metoda Newtona-Raphsona**:
   - Używa pochodnych funkcji do znajdowania pierwiastka.
   - Zalety: szybka zbieżność dla dobrze wybranych początkowych wartości.
   - Wady: wymaga znajomości pochodnej i odpowiedniego punktu startowego.
3. **Metoda siecznych**:
   - Nie wymaga pochodnych, opiera się na różnicach skończonych.
   - Zalety: uniwersalność.
   - Wady: wolniejsza niż Newtona.
4. **Metoda regulacji falsi**:
   - Kombinacja bisekcji i metod opartych na stycznych.
   - Zalety: pewność znalezienia pierwiastka w ograniczonym przedziale.
   - Wady: wolniejsza niż metoda Newtona.

3. Przedstawić wady i zalety interpolacji wielomianowej Newtona i Lagrange’a

1. **Interpolacja Lagrange’a**:
   - Zalety:
     - Łatwa do zaimplementowania w postaci zamkniętej.
     - Niezależność od kolejności punktów.
   - Wady:
     - Trudności w modyfikacji (np. dodanie nowego punktu wymaga ponownego obliczenia całego wielomianu).
     - Problemy z niestabilnością numeryczną dla dużej liczby węzłów (efekt Rungego).
2. **Interpolacja Newtona**:
   - Zalety:
     - Łatwość dodawania nowych punktów (wykorzystanie różnic dzielonych).
     - Stabilność numeryczna przy odpowiednim rozmieszczeniu punktów.
   - Wady:
     - Trudniejsza interpretacja niż Lagrange’a.
     - Wymaga znajomości kolejności punktów.

4. Interpolacja wielomianowa a numeryczne obliczanie całek oznaczonych

- **Interpolacja wielomianowa**:
  - Tworzy wielomian, który przechodzi przez dane punkty.
  - Dobra dla aproksymacji gładkich funkcji, ale podatna na oscylacje dla nierównomiernych węzłów.
- **Numeryczne obliczanie całek**:
  - Wykorzystuje aproksymację funkcji podcałkowej przez wielomiany w ramach metod, takich jak:
    - **Metoda trapezów** (przybliża obszar prostymi).
    - **Metoda Simpsona** (przybliża obszar wielomianami kwadratowymi).
  - Interpolacja jest wykorzystywana do lepszej reprezentacji funkcji podcałkowej.

5. Opisać aproksymację średniokwadratową i porównać z interpolacją

- **Aproksymacja średniokwadratowa**:
  - Minimalizuje sumę kwadratów odchyleń między wartościami funkcji a danymi.
  - Nie przechodzi przez wszystkie punkty danych, lecz dąży do najlepszego przybliżenia w sensie średniokwadratowym.
  - Stosowana w regresji, modelowaniu trendów.
- **Interpolacja**:
  - Przechodzi dokładnie przez wszystkie punkty danych.
  - Często powoduje nadmierne oscylacje przy nierównomiernych węzłach.

Porównanie:

- Aproksymacja średniokwadratowa jest bardziej odporna na błędy w danych (szum).
- Interpolacja wymaga dokładnych danych wejściowych i jest podatna na oscylacje przy dużej liczbie punktów.

6. Omówić metody rozwiązywania równań różniczkowych Eulera i Rungego-Kutty oraz krótko opisać błędy w obu przypadkach

1. **Metoda Eulera**:
   - Przybliża rozwiązanie iteracyjnie: $y_{n+1}=y_n+h\cdot f(x_n,y_n)$
   - Zalety:
     - Łatwa implementacja.
   - Wady:
     - Niska dokładność (błąd globalny liniowy: O(h)).
     - Niestabilność dla dużych kroków h.
2. **Metoda Rungego-Kutty (RK4)**:
   - Znacznie bardziej dokładna $y_{n+1}=y_n+\frac{n}{6}(k_1+2k_2+2k_3+k_4)$, gdzie k_1,k_2,k_3,k_4 to pomocnicze wartości obliczane w każdym kroku.
   - Zalety:
     - Wysoka dokładność (błąd globalny: O(h4).
     - Stabilność przy umiarkowanych krokach.
   - Wady:
     - Większe obciążenie obliczeniowe niż metoda Eulera.

Bazy bazy danych w aplikacjach internetowych

1. Zapytania zagnieżdżone

Zapytania zagnieżdżone to zapytania SQL zawierające inne zapytania wewnątrz swojego ciała. Mogą być wykorzystywane w klauzulach takich jak SELECT, FROM, czy WHERE.

- **Rodzaje**:
  - **Podzapytań w klauzuli WHERE**:

    ```
    SELECT * FROM employees
    WHERE department_id = (
      SELECT department_id
      FROM departments
      WHERE department_name = 'IT'
    );
    ```

    W tym przykładzie podzapytanie wybiera identyfikator działu dla działu IT, a wynik główny zwraca pracowników z tego działu.

  - **Podzapytania w klauzuli FROM** (zwane *inline views*):

    ```
    SELECT department_name, AVG(salary)
    FROM (
      SELECT department_id, salary
      FROM employees
    ) AS salary_data
    JOIN departments ON salary_data.department_id = departments.department_id
    GROUP BY department_name;
    ```

  - **Podzapytania skorelowane**: Podzapytania, które odwołują się do wiersza z zapytania nadrzędnego:

    ```
    SELECT employee_id, salary
    FROM employees emp1
    WHERE salary > (
      SELECT AVG(salary)
      FROM employees emp2
      WHERE emp1.department_id = emp2.department_id
    );
    ```

2. Indeksy i transakcje

Indeksy

Indeksy to struktury danych, które przyspieszają wyszukiwanie danych w tabelach baz danych.

- **Rodzaje indeksów**:
  - **Indeksy jednowymiarowe** (na pojedynczej kolumnie).
  - **Indeksy złożone** (na wielu kolumnach).
  - **Indeksy pełnotekstowe** (dla wyszukiwania w tekstach).
  - **Indeksy unikalne** (gwarantujące unikalność danych w kolumnach).
- **Zalety**:
  - Szybsze wyszukiwanie i sortowanie danych.
  - Redukcja czasu wykonywania zapytań.
- **Wady**:
  - Zajmowanie dodatkowego miejsca w pamięci.
  - Spowolnienie operacji INSERT, UPDATE, DELETE (przez konieczność aktualizacji indeksów).

Transakcje

Transakcja to jednostka pracy, która składa się z jednego lub więcej zapytań SQL, wykonywanych jako całość.

- **ACID**:
  - **A**tomomicity – wszystkie operacje w transakcji są wykonane albo w całości, albo wcale.
  - **C**onsistency – dane muszą pozostać spójne po wykonaniu transakcji.
  - **I**solation – równoczesne transakcje nie zakłócają siebie nawzajem.
  - **D**urability – zmiany dokonane przez transakcję są trwałe.
- **Komendy transakcji**:
  - BEGIN TRANSACTION – rozpoczęcie transakcji.
  - COMMIT – zatwierdzenie transakcji.
  - ROLLBACK – cofnięcie transakcji.

3. Procedury składowane, funkcje użytkownika, wyzwalacze i widoki

Procedury składowane

Procedury to zestawy instrukcji SQL przechowywane w bazie danych i wykonywane na żądanie.

- **Zastosowania**: Automatyzacja operacji, weryfikacja danych, kompleksowe przetwarzanie.
- **Przykład**:

```
CREATE PROCEDURE UpdateSalary (emp_id INT, new_salary DECIMAL)
BEGIN
  UPDATE employees
  SET salary = new_salary
  WHERE employee_id = emp_id;
END;
```

Funkcje użytkownika

Funkcje to programowalne elementy SQL, które zwracają pojedynczą wartość i są używane w zapytaniach.

- **Przykład**:

```
CREATE FUNCTION GetDepartmentBudget(department_id INT)
RETURNS DECIMAL
BEGIN
  RETURN (
    SELECT SUM(salary)
    FROM employees
    WHERE department_id = department_id
  );
END;
```

Wyzwalacze (Triggers)

Wyzwalacze to automatyczne mechanizmy wywoływane w odpowiedzi na zdarzenia w bazie danych (np. INSERT, UPDATE, DELETE).

- **Przykład**:

```
CREATE TRIGGER BeforeInsertEmployee
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
  SET NEW.hire_date = CURRENT_DATE;
END;
```

Widoki (Views)

Widoki to zapytania SQL zapisane jako obiekty w bazie danych, które działają jak wirtualne tabele.

- **Zastosowania**: Abstrakcja danych, ochrona danych, zmniejszenie złożoności zapytań.
- **Przykład**:

```
CREATE VIEW ActiveEmployees AS
SELECT * FROM employees WHERE status = 'active';
```

4. Projektowanie baz danych dla aplikacji webowych

- **Zasady projektowania**:
  - Używanie modelu ERD do planowania struktur danych.
  - Zapewnienie normalizacji danych (np. eliminacja redundancji).
  - Projektowanie pod kątem wydajności – stosowanie indeksów, partycjonowanie.
  - Uwzględnienie skalowalności i wydajności (np. podział danych na bazy mikroserwisów).
- **Dobre praktyki**:
  - Używanie kluczy głównych i obcych.
  - Projektowanie logicznych i intuicyjnych nazw tabel/kolumn.
  - Tworzenie procedur składowanych dla skomplikowanych operacji.

5. Bezpieczeństwo baz danych w aplikacjach webowych

- **Zagrożenia**:
  - SQL Injection: Wstrzykiwanie złośliwych zapytań SQL.
  - Nieautoryzowany dostęp.
  - Kradzież lub wyciek danych.
- **Środki bezpieczeństwa**:
  - Używanie parametrów zapytań zamiast konkatenacji stringów.
  - Nadawanie użytkownikom minimalnych wymaganych uprawnień.
  - Szyfrowanie danych przechowywanych i przesyłanych.
  - Regularne aktualizacje oprogramowania bazodanowego.

6. Wzorzec projektowy MVC

- **Opis**:
  - MVC (Model-View-Controller) to wzorzec projektowy używany w aplikacjach webowych do separacji logiki aplikacji od interfejsu użytkownika.
- **Elementy MVC**:
  - **Model**: Reprezentuje dane i logikę biznesową (np. operacje na bazie danych).
  - **View**: Odpowiada za prezentację danych (np. generowanie stron HTML).
  - **Controller**: Zarządza interakcją użytkownika i przetwarza żądania, aktualizując model i widok.
- **Korzyści**:
  - Łatwiejsze testowanie i debugowanie.
  - Lepsza organizacja kodu.
  - Możliwość niezależnego rozwijania komponentów.
- **Przykład użycia**: W aplikacjach webowych takich jak Django, Spring lub Ruby on Rails, każda warstwa MVC jest oddzielnym modułem umożliwiającym łatwiejsze zarządzanie i rozwój.

1. Omów skrót ACID.
2. Co to jest transakcja?
3. Do czego służy kursor?
4. W jakim celu tworzy się filtr widoku dzierżawców?
5. Opisz zasady działania i jak wykonać migrację na bazie danych.
6. W jakim celu wykorzystujemy Moduł Identity w ASP.NET?
7. Opisz wzorzec MVC na przykładzie projektu aplikacji w ASP.NET.
8. Omów zasadę działania wyzwalaczy w bazie danych i po co się je stosuje.

1. Skrót ACID

ACID to zestaw właściwości gwarantujących niezawodność transakcji w systemach bazodanowych:

- ***A****tomicity* (atomowość): Transakcja jest niepodzielna – albo wszystkie jej operacje zostaną wykonane, albo żadna.

- **C**onsistency (*Spójność*): Transakcja przeprowadza bazę danych ze stanu jednego spójnego do innego spójnego, bez naruszania ograniczeń.
- **I**solation (*Izolacja*): Transakcje są wykonywane niezależnie od siebie, jakby były wykonywane sekwencyjnie.
- **D**urability (*Trwałość*): Po zakończeniu transakcji jej efekty są trwałe, nawet w przypadku awarii systemu.

2. Co to jest transakcja?

Transakcja to zestaw operacji na bazie danych wykonywanych jako jednostka logiczna. Wszystkie operacje muszą się zakończyć sukcesem, aby zmiany zostały zatwierdzone. Jeśli którakolwiek z operacji się nie powiedzie, transakcja zostaje cofnięta (ROLLBACK).
Przykład w SQL:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT; -- Zatwierdzenie transakcji
```

3. Do czego służy kursor?

Kursor w bazie danych jest wskaźnikiem umożliwiającym iteracyjne przetwarzanie wyników zapytania.

- **Zastosowania**:
  - Operacje wymagające przetwarzania wiersz po wierszu.
  - Zastosowanie w skomplikowanych algorytmach, gdzie proste zapytanie SQL jest niewystarczające.
- **Przykład**:

```
DECLARE my_cursor CURSOR FOR
SELECT name FROM employees;

OPEN my_cursor;
FETCH NEXT FROM my_cursor INTO @name;

WHILE @@FETCH_STATUS = 0
BEGIN
   PRINT @name;
   FETCH NEXT FROM my_cursor INTO @name;
END;

CLOSE my_cursor;
DEALLOCATE my_cursor;
```

4. W jakim celu tworzy się filtr widoku dzierżawców?

Filtr widoku dzierżawców (*Tenant View Filter*) jest używany w systemach wielodzierżawczych (*multi-tenant*), aby ograniczyć dostęp do danych tylko dla wybranego dzierżawcy.

- **Cel**:
  - Ochrona danych przed nieautoryzowanym dostępem.
  - Logiczne rozdzielenie danych dla różnych klientów w ramach jednej bazy danych.
  - Poprawa wydajności zapytań.
- **Przykład implementacji**:

```
SELECT * FROM orders
WHERE tenant_id = CURRENT_TENANT_ID();
```

5. Zasady działania i migracja na bazie danych

Migracja w bazie danych polega na zarządzaniu zmianami w strukturze bazy danych (np. dodanie kolumn, zmiana typów danych). Jest kluczowa w systemach, gdzie zmiany są wprowadzane stopniowo.

Kroki migracji:

1. **Zdefiniowanie zmian**: Tworzenie migracji przy użyciu narzędzi takich jak Entity Framework.

   ```
   dotnet ef migrations add AddNewColumn
   ```

2. **Zastosowanie zmian**: Wprowadzenie zmian do bazy danych.

   ```
   dotnet ef database update
   ```

3. **Testowanie**: Sprawdzenie poprawności migracji i działania aplikacji.

Zasady działania migracji:

- Każda migracja jest zapisana jako plik zawierający instrukcje do wykonania w SQL.
- Migracje są wersjonowane, co pozwala na cofnięcie zmian w razie potrzeby (`dotnet ef database update PreviousMigration`).

6. W jakim celu wykorzystujemy moduł Identity w ASP.NET?

Moduł Identity w ASP.NET zapewnia funkcjonalność zarządzania tożsamością użytkowników, w tym:

- **Rejestrację i logowanie**.
- **Zarządzanie rolami i uprawnieniami**.
- **Szyfrowane przechowywanie haseł**.
- **Integrację z zewnętrznymi systemami uwierzytelniania** (Google, Facebook, Microsoft).

Przykładowe zastosowanie:

- Rejestracja użytkownika z walidacją:

```
var result = await _userManager.CreateAsync(user, password);
if (result.Succeeded)
{
```

```
    await _signInManager.SignInAsync(user, isPersistent: false);
}
```

7. Wzorzec MVC na przykładzie projektu aplikacji w ASP.NET

Opis wzorca MVC:

- **Model**: Odpowiada za dane i logikę aplikacji.
- **View**: Odpowiada za prezentację danych użytkownikowi.
- **Controller**: Zarządza przepływem danych między Modelem a View.

Przykład w ASP.NET:

- **Model**:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

- **Controller**:

```
public class ProductController : Controller
{
    private readonly ApplicationDbContext _context;

    public ProductController(ApplicationDbContext context)
    {
        _context = context;
    }

    public IActionResult Index()
    {
        var products = _context.Products.ToList();
        return View(products);
    }
}
```

- **View** (Index.cshtml):

```
@model IEnumerable<Product>

<h1>Product List</h1>
<ul>
    @foreach (var product in Model)
    {
        <li>@product.Name - @product.Price</li>
    }
</ul>
```

8. Wyzwalacze w bazie danych

Wyzwalacze (*Triggers*) to automatyczne mechanizmy w bazie danych, które uruchamiają określone akcje w odpowiedzi na zdarzenia (np. INSERT, UPDATE, DELETE).

Zasady działania:

- Wyzwalacz działa przed lub po wykonaniu określonej operacji.
- Może być definiowany dla pojedynczego wiersza (FOR EACH ROW) lub całej tabeli.

Przykład:

- Wyzwalacz rejestrujący zmiany w tabeli logów:

```
CREATE TRIGGER LogEmployeeUpdates
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO employee_logs (employee_id, old_salary, new_salary, change_date)
  VALUES (OLD.employee_id, OLD.salary, NEW.salary, CURRENT_TIMESTAMP);
END;
```

Zastosowania:

- Automatyczne rejestrowanie zmian w danych.
- Weryfikacja danych przed wstawieniem/aktualizacją.
- Wykonywanie operacji powiązanych w innych tabelach.

Inżynieria oprogramowania

1. Procesy wytwarzania oprogramowania i ich modele

Procesy wytwarzania oprogramowania

Proces wytwarzania oprogramowania obejmuje wszystkie działania związane z tworzeniem, rozwijaniem i utrzymaniem systemu oprogramowania. Składa się z następujących etapów:

- **Analiza wymagań**: Zbieranie i dokumentowanie wymagań użytkownika.
- **Projektowanie**: Opracowanie architektury i szczegółowych planów technicznych.
- **Implementacja**: Programowanie i tworzenie funkcjonalności.
- **Testowanie**: Weryfikacja i walidacja systemu pod kątem zgodności z wymaganiami.
- **Utrzymanie**: Poprawianie błędów, dodawanie nowych funkcji i zapewnienie stabilności.

Modele procesów wytwarzania

- **Model kaskadowy**: Każdy etap procesu jest realizowany sekwencyjnie, od analizy do wdrożenia.
  - Zalety: Jasna struktura, prosta implementacja.
  - Wady: Brak elastyczności, trudności w wprowadzaniu zmian.
- **Model iteracyjny**: Rozwój oprogramowania odbywa się w cyklach, co pozwala na iteracyjne ulepszanie produktu.

    - Zalety: Możliwość wczesnego dostarczenia działających funkcji.
    - Wady: Może prowadzić do trudności w zarządzaniu zmianami.
- **Model zwinny (Agile)**: Skupia się na elastyczności, iteracyjnych dostawach i współpracy z klientem.
    - Zalety: Szybka reakcja na zmiany, lepsza komunikacja z interesariuszami.
    - Wady: Potrzebna wysoka dyscyplina zespołu.
- **Model DevOps**: Łączy rozwój (Dev) i operacje (Ops), aby zautomatyzować i zoptymalizować proces dostarczania oprogramowania.
    - Zalety: Szybkie wdrożenia, większa niezawodność systemu.
    - Wady: Wymaga zaawansowanego zaplecza technicznego.

2. Inżynieria wymagań dla systemów oprogramowania

Pojęcie inżynierii wymagań

Inżynieria wymagań to proces identyfikowania, analizowania, dokumentowania i zarządzania wymaganiami użytkownika i systemu.

Kluczowe kroki inżynierii wymagań

1. **Elicytacja wymagań**: Zbieranie wymagań od interesariuszy (wywiady, warsztaty, prototypy).
2. **Analiza wymagań**: Uszczegółowienie, weryfikacja i eliminacja konfliktów w wymaganiach.
3. **Specyfikacja wymagań**: Tworzenie dokumentu specyfikacji wymagań (SRS).
4. **Walidacja wymagań**: Weryfikacja, czy wymagania są kompletne, spójne i możliwe do spełnienia.
5. **Zarządzanie wymaganiami**: Monitorowanie zmian i kontrolowanie ich wpływu na projekt.

Rodzaje wymagań

- **Funkcjonalne**: Co system ma robić (np. „System pozwala użytkownikowi wysłać e-mail”).
- **Niefunkcjonalne**: Ograniczenia dotyczące jakości (np. wydajność, bezpieczeństwo).
- **Biznesowe**: Powody powstania systemu i jego cel biznesowy.

3. Metody analizy i modelowania oprogramowania

Techniki analizy i modelowania

- **Diagramy UML (Unified Modeling Language)**: Standardowy język wizualizacji projektów.
    - Diagramy przypadków użycia: Opisują interakcje użytkowników z systemem.
    - Diagramy klas: Pokazują strukturę obiektów i ich zależności.
    - Diagramy sekwencji: Opisują przepływ komunikacji między obiektami w czasie.
- **Modelowanie procesów biznesowych (BPMN)**: Przedstawienie procesów biznesowych za pomocą diagramów.
- **Analiza funkcjonalna**: Rozbijanie wymagań funkcjonalnych na szczegółowe komponenty systemu.

4. Projektowanie architektoniczne systemów oprogramowania

Cel projektowania architektonicznego

Projektowanie architektoniczne definiuje strukturę systemu, jego moduły i interakcje między nimi.

Kluczowe kroki projektowania architektonicznego

1. **Podział systemu na moduły**: Identyfikacja kluczowych komponentów.
2. **Określenie interfejsów**: Definicja sposobów komunikacji między komponentami.
3. **Uwzględnienie wymagań niefunkcjonalnych**: Skalowalność, wydajność, bezpieczeństwo.
4. **Dokumentacja architektury**: Tworzenie diagramów i opisów.

5. Wzorce architektoniczne i ich zastosowania, ogólne architektury aplikacji

Wzorce architektoniczne

- **Monolit**: Cały system jako jeden zestaw aplikacji.
  - Zalety: Prosta implementacja, łatwe debugowanie.
  - Wady: Trudna skalowalność.
- **Mikroserwisy**: System podzielony na niezależne moduły komunikujące się za pomocą API.
  - Zalety: Skalowalność, możliwość szybkiego wdrożenia zmian.
  - Wady: Złożoność w zarządzaniu.
- **Architektura warstwowa**: Podział na warstwy (prezentacja, logika biznesowa, dane).
  - Zalety: Łatwa wymiana warstwy, modularność.
  - Wady: Spowolnienie komunikacji między warstwami.

6. Walidacja i testowanie oprogramowania

Walidacja i testowanie

- **Walidacja**: Czy system spełnia wymagania użytkownika?
- **Weryfikacja**: Czy system działa zgodnie ze specyfikacją?

Rodzaje testów

- **Jednostkowe (Unit Tests)**: Testowanie pojedynczych komponentów.
- **Integracyjne**: Sprawdzanie interakcji między komponentami.
- **Systemowe**: Testowanie całego systemu jako całości.
- **Akceptacyjne**: Weryfikacja zgodności z wymaganiami użytkownika.

Proces testowania

1. **Planowanie**: Opracowanie strategii i harmonogramu testów.
2. **Projektowanie przypadków testowych**: Tworzenie scenariuszy testowych.
3. **Wykonanie testów**: Testowanie funkcji i rejestrowanie wyników.
4. **Analiza wyników**: Identyfikacja i naprawa błędów.

1. Scharakteryzuj podstawowe czynności inżynierii oprogramowania.

2. Wyjaśnij różnice między wymaganiami użytkownika, a wymaganiami systemowymi i uzasadnij, dlaczego ważne jest ich rozróżnianie w procesie inżynierii wymagań.

3. Omów cechy charakterystyczne systemów o architekturze warstwowej.

4. Omów zastosowanie wybranego diagramu UML w procesie projektowania oprogramowania.

5. Jakie czynniki należy uwzględnić przy podejmowaniu decyzji o wyborze platformy dla komponentów systemu rozproszonego.

1. Scharakteryzuj podstawowe czynności inżynierii oprogramowania

Inżynieria oprogramowania obejmuje zestaw działań i procesów mających na celu stworzenie, rozwijanie i utrzymanie oprogramowania. Podstawowe czynności to:

- **Analiza wymagań**: Zbieranie i definiowanie potrzeb użytkowników oraz wymagań systemowych.
- **Projektowanie**: Tworzenie architektury i szczegółowego projektu systemu, uwzględniającego funkcjonalność, interfejsy oraz strukturę danych.
- **Implementacja**: Pisanie kodu zgodnie z wymaganiami i projektem.
- **Testowanie**: Weryfikacja poprawności działania systemu oraz spełnienia wymagań.
- **Wdrożenie**: Udostępnienie gotowego oprogramowania użytkownikom końcowym.
- **Utrzymanie**: Poprawianie błędów, aktualizacje i dodawanie nowych funkcjonalności w odpowiedzi na zmieniające się potrzeby użytkowników.

2. Wyjaśnij różnice między wymaganiami użytkownika a wymaganiami systemowymi i uzasadnij, dlaczego ważne jest ich rozróżnianie w procesie inżynierii wymagań

- **Wymagania użytkownika**:
  - Są opisem funkcjonalności i możliwości systemu z perspektywy użytkownika końcowego.
  - Często mają formę ogólnych opisów w języku naturalnym, np. „System ma umożliwiać wyszukiwanie produktów po nazwie”.
  - Skupiają się na tym, „co” system ma robić, a nie „jak” to osiągnąć.
- **Wymagania systemowe**:
  - Są szczegółową specyfikacją techniczną tego, co system ma realizować, w tym ograniczenia techniczne.
  - Obejmują aspekty implementacyjne, np. „System wyszukuje produkty w czasie krótszym niż 1 sekunda, korzystając z indeksu bazy danych”.

Znaczenie rozróżniania:

- Pozwala unikać nieporozumień między zespołem technicznym a użytkownikami.
- Umożliwia precyzyjne określenie, które funkcjonalności muszą zostać wdrożone.
- Ułatwia walidację systemu – sprawdzanie, czy spełnia wymagania użytkowników, oraz weryfikację techniczną.

3. Omów cechy charakterystyczne systemów o architekturze warstwowej

Architektura warstwowa dzieli system na logiczne warstwy, które odpowiadają za różne aspekty jego działania. Cechy charakterystyczne:

- **Podział na warstwy**: Najczęściej stosowany podział obejmuje:
    1. Warstwa prezentacji (UI) – interfejs użytkownika.
    2. Warstwa logiki biznesowej – implementacja reguł działania systemu.
    3. Warstwa dostępu do danych – zarządzanie operacjami na bazie danych.
- **Niezależność warstw**: Każda warstwa działa jako niezależny moduł, co ułatwia rozwój i utrzymanie systemu.
- **Modularność**: Zmiany w jednej warstwie nie wpływają bezpośrednio na inne warstwy.
- **Łatwość testowania**: Można testować każdą warstwę niezależnie.
- **Zastosowanie**: Wykorzystywana w aplikacjach webowych i systemach złożonych.

4. Omów zastosowanie wybranego diagramu UML w procesie projektowania oprogramowania

Diagram przypadków użycia (Use Case Diagram):
Służy do modelowania interakcji między użytkownikami a systemem.

- **Elementy diagramu**:
    - **Aktorzy**: Użytkownicy (ludzie, systemy) wchodzący w interakcję z systemem.
    - **Przypadki użycia**: Opisują funkcjonalności oferowane przez system.
    - **Relacje**: Połączenia między aktorami a przypadkami użycia.
- **Zastosowanie w projektowaniu**:
    - Pomaga zrozumieć wymagania użytkowników.
    - Identyfikuje funkcje systemu i ich priorytety.
    - Ułatwia komunikację między zespołem technicznym a interesariuszami.

Przykład: W systemie e-commerce przypadki użycia mogą obejmować „Logowanie”, „Przeglądanie produktów” i „Zakup produktu”.

5. Jakie czynniki należy uwzględnić przy podejmowaniu decyzji o wyborze platformy dla komponentów systemu rozproszonego

Podczas wyboru platformy dla systemu rozproszonego należy uwzględnić następujące czynniki:

- **Wymagania niefunkcjonalne**:
    - Skalowalność: Czy platforma wspiera łatwe dodawanie nowych komponentów?
    - Wydajność: Jak platforma radzi sobie z dużą ilością danych i żądań?
    - Niezawodność: Czy zapewnia mechanizmy redundancji i tolerancji błędów?
- **Kompatybilność**:
    - Czy platforma wspiera wymagane języki programowania i technologie?
    - Czy może współpracować z innymi systemami?
- **Bezpieczeństwo**:
    - Czy platforma posiada mechanizmy szyfrowania, uwierzytelniania i zarządzania uprawnieniami?

- **Koszty**:
  - Opłaty licencyjne, koszty infrastruktury i utrzymania.
- **Wsparcie i społeczność**:
  - Dostępność dokumentacji, wsparcia technicznego i społeczności użytkowników.
- **Dostępność narzędzi deweloperskich**:
  - Czy platforma oferuje narzędzia ułatwiające programowanie, debugowanie i monitorowanie?

Przykład: Wybór między chmurą AWS a Microsoft Azure może zależeć od wymagań konkretnego projektu i kompetencji zespołu.

Zastosowanie diagramu BPMN w procesie projektowania oprogramowania

BPMN (Business Process Model and Notation) to standard do modelowania procesów biznesowych. Diagramy BPMN są wykorzystywane do wizualnego przedstawienia przepływów procesów biznesowych, co ułatwia zrozumienie i komunikację między zespołami biznesowymi i technicznymi.

Elementy diagramu BPMN

1. **Obiekty przepływu**:
   - **Zdarzenia (Events)**: Punkty w procesie, które inicjują, przerywają lub kończą działanie.
   - **Działania (Activities)**: Kroki lub zadania do wykonania.
   - **Bramki (Gateways)**: Decyzje i rozgałęzienia w przepływie procesu.
2. **Obiekty łączące**:
   - **Strzałki (Sequence Flows)**: Określają kolejność wykonywania zadań.
   - **Przepływy wiadomości (Message Flows)**: Reprezentują wymianę komunikatów między uczestnikami procesu.
3. **Baseny i tory (Pools and Lanes)**:
   - **Baseny (Pools)**: Reprezentują uczestników procesu, np. organizacje lub systemy.
   - **Tory (Lanes)**: Podział w obrębie basenu, np. na zespoły lub role.
4. **Artefakty**:
   - **Adnotacje**: Opisują dodatkowe szczegóły procesów.
   - **Dane wejściowe/wyjściowe**: Ilustrują informacje przetwarzane w procesie.

Zastosowanie w procesie projektowania oprogramowania

1. **Modelowanie procesów biznesowych**:
   - BPMN pozwala przeanalizować i zrozumieć obecne procesy biznesowe, które oprogramowanie ma wspierać lub automatyzować.
   - Ułatwia wizualizację zadań, decyzji i zależności między działaniami w procesie.
2. **Definiowanie wymagań funkcjonalnych**:
   - Diagramy BPMN mogą być używane do uzgodnienia wymagań funkcjonalnych między zespołami biznesowymi i technicznymi.
   - Każdy krok w procesie można mapować na funkcję w oprogramowaniu.
3. **Identyfikacja interakcji między systemami**:
   - Baseny i przepływy wiadomości pomagają określić, jak różne systemy i moduły aplikacji komunikują się ze sobą.

4. **Weryfikacja i optymalizacja procesów**:
    - BPMN pozwala zidentyfikować nieefektywności w istniejących procesach i zaprojektować nowe, bardziej optymalne przepływy.
5. **Prototypowanie i implementacja**:
    - Diagram BPMN może być używany jako podstawa do tworzenia prototypów aplikacji.
    - Może być też bezpośrednio wykorzystany w systemach BPM (Business Process Management), które wspierają automatyzację procesów.

Przykład zastosowania

W przypadku projektowania systemu zarządzania zamówieniami (Order Management System):

1. Diagram BPMN ilustruje proces realizacji zamówienia od jego złożenia przez klienta, przez płatność, aż po dostawę.
2. W basenach reprezentuje dział sprzedaży (odpowiedzialny za przyjęcie zamówienia) i dział logistyki (odpowiedzialny za wysyłkę).
3. Bramki wskazują na decyzje, takie jak weryfikacja płatności lub dostępność produktu.
4. Zadania można przyporządkować modułom systemu, np. „Sprawdź płatność" jako funkcja modułu księgowego.

Korzyści z wykorzystania BPMN

- **Zrozumiałość**: Diagramy BPMN są czytelne zarówno dla zespołów technicznych, jak i biznesowych.
- **Standaryzacja**: Zapewnia jednolite podejście do opisu procesów.
- **Komunikacja**: Ułatwia komunikację między interesariuszami.
- **Automatyzacja**: Diagramy BPMN mogą być bezpośrednio implementowane w narzędziach BPM, co przyspiesza wdrażanie procesów w systemie.

Systemy czasu rzeczywistego

1. Rodzaje, klasy i przykłady systemów czasu rzeczywistego (SCR)

Rodzaje systemów czasu rzeczywistego:

1. **Twarde systemy czasu rzeczywistego (Hard Real-Time Systems)**:
    - Muszą zawsze spełniać swoje ograniczenia czasowe (deadlines). Naruszenie czasowe prowadzi do katastrofalnych skutków.
    - Przykłady: systemy sterowania lotami, medyczne urządzenia podtrzymujące życie, systemy hamulcowe ABS.
2. **Miękkie systemy czasu rzeczywistego (Soft Real-Time Systems)**:
    - Ograniczenia czasowe mogą być okazjonalnie naruszane, ale skutki są mniej poważne.
    - Przykłady: systemy transmisji strumieniowej, gry komputerowe.
3. **Firmowe systemy czasu rzeczywistego (Firm Real-Time Systems)**:
    - Naruszenie ograniczeń czasowych nie jest krytyczne dla bezpieczeństwa, ale powoduje utratę wartości wyniku.
    - Przykłady: systemy transakcji w bankomatach, systemy rezerwacji.

Klasy systemów czasu rzeczywistego:

- **Jednozadaniowe (Single-Tasking)**: Wykonują jedno zadanie w danym czasie.
- **Wielozadaniowe (Multitasking)**: Obsługują wiele zadań równocześnie.
- **Systemy wbudowane (Embedded Systems)**: Specjalizowane SCR zintegrowane z urządzeniami fizycznymi.

2. Opis jakości i skuteczności działania SCR za pomocą funkcji zysku

Funkcja zysku (Utility Function):

- W systemach czasu rzeczywistego funkcja zysku opisuje efektywność działania SCR w zależności od spełnienia ograniczeń czasowych.

Typowe funkcje zysku:

1. **Funkcja prostokątna (Hard Real-Time)**:
    - Wartość maksymalna, gdy zadanie jest wykonane w terminie.
    - Wartość zerowa lub ujemna, jeśli zadanie nie jest ukończone na czas.
2. **Funkcja liniowa opadająca (Soft Real-Time)**:
    - Wartość zysku zmniejsza się wraz z upływem czasu po przekroczeniu terminu.
3. **Funkcja opadająca wykładniczo**:
    - Skuteczność spada szybko w miarę narastania opóźnienia.

Funkcje zysku są używane do oceny jakości i priorytetyzacji zadań w SCR.

3. Priorytety statyczne i dynamiczne w SCR

Priorytety statyczne:

- Określane w momencie projektowania systemu.
- Priorytet zadania nie zmienia się podczas jego wykonania.
- **Zalety**: prostota implementacji, deterministyczność.
- **Wady**: brak elastyczności w przypadku dynamicznie zmieniających się warunków.

Priorytety dynamiczne:

- Priorytety zadań są ustalane w trakcie działania systemu w oparciu o aktualne warunki, np. zbliżający się deadline.
- **Zalety**: elastyczność, lepsze wykorzystanie zasobów.
- **Wady**: większe obciążenie systemu, większa złożoność algorytmów.

4. Algorytmy szeregowania zadań z wywłaszczeniem stosowane w SCR

Algorytmy szeregowania:

1. **Rate Monotonic Scheduling (RMS)**:
    - Zadania o krótszych okresach wykonania mają wyższy priorytet.
    - Używane w systemach z priorytetami statycznymi.

2. **Earliest Deadline First (EDF)**:
    - Zadania o najbliższym terminie ukończenia mają najwyższy priorytet.
    - Priorytet dynamiczny, wydajny w systemach miękkich i twardych.
3. **Least Laxity First (LLF)**:
    - Priorytet zależy od różnicy między czasem pozostałym do terminu a czasem potrzebnym na wykonanie zadania.

Wywłaszczenie:

- Mechanizm przerwania zadania o niższym priorytecie przez zadanie o wyższym priorytecie.

5. Zjawisko inwersji priorytetów i sposoby zapobiegania w SCR

Inwersja priorytetów:

- Sytuacja, w której zadanie o wysokim priorytecie jest blokowane przez zadanie o niskim priorytecie, które z kolei jest blokowane przez inne zadania średniego priorytetu.

Przykład:

1. Zadanie AAA (wysoki priorytet) czeka na zasób używany przez zadanie BBB (niski priorytet).
2. Zadanie CCC (średni priorytet) blokuje BBB, uniemożliwiając zakończenie AAA.

Sposoby zapobiegania:

1. **Priorytet dziedziczony (Priority Inheritance)**:
    - Zadanie niskiego priorytetu dziedziczy priorytet zadania wysokiego, dopóki nie zwolni zasobu.
2. **Priorytet odwrócony (Priority Ceiling)**:
    - Zadania blokujące mają przypisany priorytet równy najwyższemu priorytetowi zadania, które może używać zasobu.
3. **Zarządzanie czasem blokady (Time-Locking)**:
    - Ustalony limit czasu blokady zasobów.

Kolejka w systemach czasu rzeczywistego (SCR) jest strukturą danych służącą do zarządzania zadaniami (procesami) lub zasobami w czasie rzeczywistym, które muszą być obsługiwane w określonym czasie. Systemy czasu rzeczywistego muszą działać zgodnie z wymaganiami czasowymi, aby zapewnić, że zadania są realizowane w odpowiednim czasie.

Właściwości kolejek:

- **FIFO (First In, First Out)**: Kolejka w systemach czasu rzeczywistego zazwyczaj działa na zasadzie FIFO, co oznacza, że zadanie, które jako pierwsze trafi do kolejki, jest również jako pierwsze obsługiwane. Jest to ważne w kontekście zadań, które mają określony czas wykonania i wymagają sekwencyjnego przetwarzania.
- **Zarządzanie zadaniami w czasie rzeczywistym**: Kolejki w systemach SCR umożliwiają przechowywanie zadań, które czekają na zasoby systemowe, a także umożliwiają zarządzanie kolejnością wykonania zadań. Kolejki są używane w harmonogramowaniu zadań i sterowaniu dostępem do zasobów.

- **Przestrzeganie ograniczeń czasowych**: Zadania w systemach SCR mają określone ograniczenia czasowe, które muszą być spełnione. Kolejki mogą być wykorzystywane do monitorowania i kontrolowania czasu oczekiwania oraz realizacji zadań, aby zapewnić, że wszystkie zadania są realizowane w wymaganych ramach czasowych.

Zastosowania koleje:

- **Zarządzanie procesami w czasie rzeczywistym**: Kolejki są używane w systemach operacyjnych czasu rzeczywistego, aby zapewnić, że zadania o określonym czasie wykonania są obsługiwane w odpowiedniej kolejności, a czas oczekiwania na CPU jest minimalny.
- **Zarządzanie zasobami systemowymi**: W systemach SCR, które zarządzają zadaniami w czasie rzeczywistym, kolejki są wykorzystywane do przydzielania i kontrolowania dostępu do zasobów, takich jak CPU, pamięć i urządzenia wejścia/wyjścia.

Algorytm Round Robin

Round Robin (RR) jest algorytmem planowania, który może być stosowany w systemach czasu rzeczywistego, aby zapewnić sprawiedliwy i cykliczny dostęp do zasobów procesora dla zadań. Choć w systemach czasu rzeczywistego wykorzystuje się różne algorytmy planowania (np. EDF, RMS), Round Robin może być również stosowany w zadaniach o stałych priorytetach, gdy celem jest przydzielanie zasobów w sposób równy i sprawiedliwy dla wszystkich procesów.

Zasada działania Round Robin:

- Każdemu zadaniu w systemie czasu rzeczywistego przypisywany jest **kwant czasu** – jednostkowy czas procesora, przez który zadanie będzie wykonywane.
- Procesy są przechowywane w **kolejce** i cyklicznie otrzymują CPU na czas równy kwantowi. Jeśli proces zakończy swoje zadanie przed upływem kwantu, jest usuwany z kolejki. Jeśli proces nie zakończy zadania w trakcie kwantu, wraca na koniec kolejki, a CPU zostaje przydzielone kolejnemu procesowi.
- Celem Round Robin jest zapewnienie, że wszystkie zadania mają równy dostęp do zasobów CPU, co jest ważne w systemach, w których równoczesna obsługa wielu zadań jest niezbędna.

Zalety Round Robin:

- **Sprawiedliwość**: Każdemu zadaniu jest przydzielany równy czas CPU, co zapewnia sprawiedliwy dostęp do zasobów w systemie, co jest kluczowe w systemach o ograniczonych zasobach.
- **Prostota**: Algorytm Round Robin jest łatwy do implementacji i zrozumienia, co sprawia, że jest wykorzystywany w wielu systemach operacyjnych, w tym w systemach czasu rzeczywistego.
- **Możliwość zarządzania zadaniami o stałych priorytetach**: W systemach SCR, w których zadania mają priorytety, Round Robin może być wykorzystywany do planowania zadań o równych priorytetach, co może pomóc w optymalizacji zasobów.

Wady Round Robin:

- **Brak uwzględnienia czasów krytycznych**: Jeśli zadania mają różne wymagania czasowe, Round Robin może być mniej efektywny w obsłudze zadań, które mają wyższe priorytety lub muszą być realizowane w określonym czasie.
- **Wydajność**: Jeśli kwant czasu jest zbyt mały, algorytm może powodować częste przełączanie kontekstów, co zwiększa obciążenie systemu i wydłuża czas realizacji zadań.
- **Długi czas oczekiwania**: Długie kwanty czasu mogą powodować, że zadania muszą czekać długo na wykonanie, co jest niepożądane w systemach czasu rzeczywistego, w których czas wykonania jest krytyczny.

Zastosowanie Round Robin :

- **Zarządzanie zadaniami o równych priorytetach**: Round Robin jest wykorzystywany do planowania zadań, które mają takie same priorytety w systemach czasu rzeczywistego. Pomaga to w równym rozdzieleniu zasobów między różne zadania, zapewniając sprawiedliwość.
- **Proste systemy czasu rzeczywistego**: Algorytm Round Robin jest przydatny w systemach, w których priorytety zadań są stałe, a celem jest zapewnienie podstawowej sprawiedliwości w dostępie do CPU, bez skomplikowanego zarządzania zadaniami.

1. Tabela zawiera warunki wykonywania się kilku zadań w systemie czasu rzeczywistego, które mają być obsłużone przez jeden procesor.

Zadanie	**Czas wykonania [ms]**	**t_{min} [ms]**	**t_{max} [ms]**
A	10	0	40
B	10	0	30
C	30	30	100
D	40	50	200
E	10	70	90

a) Dokonaj analizy działania programu z zastosowaniem algorytmu EDF (ang. Earlier Deadline First) wykorzystując w tym celu diagram Gantta.

b) Oceń, czy taki program spełnia warunek szeregowalności (tak lub nie – należy to wyjaśnić).

c) Oblicz średni czas oczekiwania na przetwarzanie przez procesor dla tej metody.

2. W systemie czasu rzeczywistego wykonuje się kilka zadań okresowych.

Zadanie	**Czas wykonania C [ms]**	**Okres powtarzania T [ms]**
A	20	100
B	40	150
C	100	350

Na podstawie parametrów podanych w tabeli należy wykonać następujące czynności:

a) obliczyć całkowite wykorzystanie procesora przez zadania: A, B i C

b) wyznaczyć górną granicę szeregowalności za pomocą metody planowania monotonicznego tempa RMS (ang. Rate Monotonic Scheduling) wg wzoru:

granica szeregowalności RMS n=n(21n−1)

c) wykazać, czy wszystkie zadania (A, B, C) można pomyślnie zaplanować stosując algorytm RMS (ang. Rate Monotonic Scheduling).

3. Wyjaśnij różnicę pomiędzy rygorystycznym, a łagodnym traktowaniem zadań czasu rzeczywistego oraz narysuj odpowiednie funkcje zysku.
4. Projektowanie i główne elementy systemu czasu rzeczywistego.
5. Zasady, mechanizmy i języki programowania aplikacji czasu rzeczywistego.
6. Rola priorytetów w systemach czasu rzeczywistego.
7. Mechanizmy redukujące zjawisko inwersji priorytetów.
8. Typy systemów czasu rzeczywistego oraz obszary ich zastosowań.

1. Algorytm Earlier Deadline First (EDF)

a) Diagram Gantta dla EDF

- Zadania uszeregowane według najbliższego terminu końcowego (tmax).
- Tabela zadań:
  - A: t_{max}=40
  - B: t_{max}=30
  - C: t_{max}=100
  - D: t_{max}=200
  - E: t_{max}=90

Kolejność realizacji zadań: B→A→C→E→D

Diagram Gantta:

0 10 20 50 60 100 140

| B | A | C | E | D |

b) Warunek szeregowalności

Warunek szeregowalności dla EDF:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

- Przy założeniu, że okres powtarzania T jest równy t_{max}:
    - U=10/30+10/40+30/100+40/200+10/90=0.333+0.25+0.3+0.2+0.111=1.194U

Ponieważ U > 1, **program nie spełnia warunku szeregowalności**.

c) Średni czas oczekiwania

Czasy oczekiwania:

- B: 0 ms (rozpoczyna natychmiast),
- A: 10 ms,
- C: 20 ms,
- E: 50 ms,
- D: 60 ms.

Średni czas oczekiwania:

średni czas oczekiwania=(0+10+20+50+60)/5=28 ms.

2. Zadania okresowe i algorytm RMS

a) Całkowite wykorzystanie procesora /

$U=C_A/T_A+C_B/T_B+C_C/T_C = 0.2 + 0.267 + 0.286 = 0.753$

b) Górna granica szeregowalności

Granica RMS dla n=3:

$U_{max}=n\cdot(2^{1/n}-1)=3\cdot(2^{1/3}-1)\approx 0.779$

c) Sprawdzenie warunku szeregowalności

$U=0.753 \leq 0.779$

Wszystkie zadania mogą być zaplanowane przy zastosowaniu RMS.

3. Różnica między rygorystycznym a łagodnym traktowaniem zadań SCR

Rygorystyczne traktowanie

- Każde zadanie musi spełniać ograniczenie czasowe.
- Naruszenie terminu powoduje katastrofalne skutki.
- **Funkcja zysku:** Prostokątna.

Łagodne traktowanie

- Zadania mogą przekroczyć termin, ale powoduje to tylko zmniejszenie jakości.
- **Funkcja zysku:** Liniowo lub wykładniczo malejąca.

4. Projektowanie i główne elementy systemu czasu rzeczywistego

1. **Planowanie czasu rzeczywistego**: Algorytmy RMS, EDF.
2. **Priorytety zadań**: Statyczne i dynamiczne.
3. **Mechanizmy synchronizacji**: Semafory, muteksy.
4. **Interfejs sprzętowy**: Obsługa przerwań.
5. **Systemy operacyjne czasu rzeczywistego (RTOS)**: Obsługa planowania i zarządzania zasobami.

5. Zasady, mechanizmy i języki programowania aplikacji czasu rzeczywistego

- **Języki**: Ada, C, C++, Rust.
- **Mechanizmy**:
    - Synchronizacja zadań (semafory, bariery).
    - Planowanie (statyczne, dynamiczne).
    - Obsługa przerwań.
- **Zasady**:
    - Deterministyczność.
    - Minimalizacja opóźnień.
    - Priorytetyzacja krytycznych zadań.

6. Rola priorytetów w systemach czasu rzeczywistego

- **Cel**: Priorytety zapewniają obsługę zadań o kluczowym znaczeniu przed mniej istotnymi.
- **Rodzaje**: Statyczne (ustalone na etapie projektowania) i dynamiczne (zmienne w trakcie działania).

7. Mechanizmy redukujące inwersję priorytetów

1. **Priority Inheritance**: Zadanie niskiego priorytetu dziedziczy wyższy priorytet.
2. **Priority Ceiling**: Zadanie blokujące zasób ma najwyższy priorytet wymagany przez inne zadania.

8. Typy systemów czasu rzeczywistego i obszary zastosowań

- **Typy**:
    - Twarde SCR: Kontrola lotów, medycyna.
    - Miękkie SCR: Multimedia, gry.
- **Zastosowania**:
    - Automatyka przemysłowa.
    - Systemy wbudowane.
    - Telekomunikacja.

Sztuczna inteligencja

1. Test Turinga i komunikacja Człowiek-Komputer (ELIZA)

Test Turinga:

Test Turinga to test zaproponowany przez Alana Turinga w 1950 roku, mający na celu ocenę, czy maszyna potrafi przejawiać inteligencję porównywalną z ludzką. Turing zaproponował, aby rozmowa między człowiekiem a komputerem była prowadzona w sposób, który uniemożliwiałby rozróżnienie, czy rozmówca to człowiek, czy maszyna. Rozmowa ta odbywała się za pomocą tekstu (np. komunikator), bez wizualnego kontaktu. Jeśli rozmówca nie byłby w stanie stwierdzić, czy rozmawia z maszyną, czy człowiekiem, to maszyna przechodzi test.

Komunikacja Człowiek-Komputer (ELIZA):

ELIZA to jeden z pierwszych programów komputerowych, który symulował rozmowę z człowiekiem. Stworzony w 1966 roku przez Josepha Weizenbauma na MIT, ELIZA naśladowała rozmowę psychoterapeuty z pacjentem. Program stosował bardzo prostą metodę, znaną jako "parafrazowanie", polegającą na zamianie fraz w pytaniach, które miały wyglądać jak odpowiedzi terapeuty. Znakomita część sukcesu ELIZA wynikała z efektu "Cygana", który polegał na tym, że użytkownicy łatwo zaczynali traktować komputer jako inteligentnego rozmówcę, mimo że program nie miał żadnej rzeczywistej "wiedzy". To proste podejście do rozumienia naturalnego języka, mimo że ograniczone, zainspirowało rozwój w dziedzinie przetwarzania języka naturalnego (NLP).

2. Algorytmy wyszukiwania i rozwiązywania problemów w praktyce

Algorytmy wyszukiwania:

Wyszukiwanie to proces odnajdywania odpowiedzi w przestrzeni stanów w celu rozwiązania problemu. W kontekście sztucznej inteligencji, algorytmy wyszukiwania pomagają w rozwiązaniu takich problemów jak łamigłówki, gry czy optymalizacja. W praktyce istnieje wiele różnych algorytmów wyszukiwania, które mogą być stosowane w zależności od charakterystyki problemu:

- **Algorytmy przeszukiwania wszerz (BFS - Breadth-First Search):** Przeszukują wszystkie węzły na jednym poziomie przed przejściem do kolejnego. Są użyteczne, gdy wszystkie stany mają tę samą wagę.
- **Algorytmy przeszukiwania w głąb (DFS - Depth-First Search):** Wybierają węzeł głęboko w drzewie przed przejściem na inne ścieżki. Mogą prowadzić do rozwiązania szybciej, ale nie zawsze są optymalne.
- *Algorytm A (A-star):** Jest to jeden z najpopularniejszych algorytmów wyszukiwania wykorzystywanych do znajdowania najkrótszej drogi, który uwzględnia zarówno koszt dotarcia do danego punktu, jak i prognozowany koszt dotarcia do celu.

- **Algorytmy genetyczne:** Używają zasady doboru naturalnego, w których "populacja" rozwiązań jest iteracyjnie modyfikowana za pomocą operacji takich jak krzyżowanie, mutacja i selekcja.

Algorytmy rozwiązywania problemów:

Algorytmy rozwiązywania problemów obejmują zarówno algorytmy wyszukiwania, jak i inne techniki, takie jak:

- **Programowanie dynamiczne:** Pomaga w rozwiązywaniu problemów, które można podzielić na mniejsze podproblemy. Przykładem jest algorytm znajdowania najkrótszej ścieżki w grafie.
- **Metoda rozgałęzień i ograniczeń:** Wykorzystywana do rozwiązywania problemów optymalizacyjnych, gdzie poszukiwany jest najlepszy sposób rozwiązania problemu spośród wielu możliwych ścieżek.

3. Przetwarzanie języka naturalnego (NLP) i metody przygotowania tekstu do NLP

Przetwarzanie języka naturalnego (NLP - Natural Language Processing) to dziedzina sztucznej inteligencji zajmująca się interakcją między komputerami a ludzkim językiem. Celem jest umożliwienie komputerom rozumienia, interpretacji i generowania języka naturalnego w sposób, który jest użyteczny i sensowny dla ludzi.

Etapy przygotowania tekstu do NLP:

- **Tokenizacja:** Dzielenie tekstu na słowa lub frazy (tokeny).
- **Usuwanie stop-words:** Usuwanie słów, które nie mają znaczenia w analizie, takich jak "i", "oraz", "na".
- **Lematyzacja i stemming:** Proces redukcji słów do ich podstawowej formy (np. "biegając" do "bieg").
- **Analiza składniowa:** Określanie struktury gramatycznej zdań.
- **Wydobywanie cech:** Wyodrębnienie cech istotnych z tekstu, takich jak słowa kluczowe, frazy itp.

Modele NLP:

- **Word2Vec:** Model, który przekształca słowa w wektory liczbowych reprezentacji, które mogą uchwycić semantyczne zależności między słowami.
- **BERT (Bidirectional Encoder Representations from Transformers):** Nowoczesny model, który bierze pod uwagę kontekst obu stron słowa (lewy i prawy kontekst), dzięki czemu jest bardziej dokładny w rozumieniu języka.

4. Uczenie maszynowe - zadania klasyfikacyjne, drzewa decyzyjne, las losowy

Zadania klasyfikacyjne:

Uczenie maszynowe w zadaniach klasyfikacyjnych polega na przypisaniu etykiety do obiektu na podstawie danych wejściowych. Przykładem może być rozpoznawanie, czy wiadomość email jest spamem, czy nie.

- **Drzewa decyzyjne:** Jest to algorytm, który dzieli dane wejściowe na różne grupy (klasy), tworząc drzewo, w którym każdy węzeł odpowiada decyzji, a liście to klasy wynikowe. Drzewa decyzyjne są łatwe do zrozumienia i interpretacji.
- **Las losowy (Random Forest):** Jest to zespół drzew decyzyjnych, w którym każdy z drzew jest tworzony na podstawie losowego podzbioru danych. Wynik lasu losowego jest średnią wyników wszystkich drzew, co zwiększa dokładność klasyfikacji.

5. Uczenie maszynowe - zadania regresyjne, regresja liniowa

Zadania regresyjne:

Zadania regresyjne polegają na przewidywaniu ciągłych wartości na podstawie danych wejściowych. Przykładem może być przewidywanie ceny akcji na podstawie różnych czynników ekonomicznych.

- **Regresja liniowa:** Jest jednym z najprostszych algorytmów stosowanych w zadaniach regresyjnych. Zakłada, że zależność między zmienną niezależną (np. wiek) a zmienną zależną (np. zarobki) jest liniowa. Regresja liniowa znajduje najlepszą prostą, która minimalizuje błąd przewidywania.
- **Regresja wielomianowa:** Jest to rozszerzenie regresji liniowej, które używa funkcji wielomianowych do modelowania bardziej złożonych zależności między zmiennymi.

Modele regresji:

- **Regresja logistyczna:** Choć nazywa się regresją, jest to model klasyfikacyjny, który używa funkcji logistycznej do klasyfikowania obiektów w jednej z dwóch klas.

Metody statystyczne w projektach inżynierskich

1. Miary/statystyki opisowe

Miary opisowe to podstawowe narzędzia w statystyce, które pomagają w przedstawieniu danych w sposób zrozumiały. Dzięki nim można szybko zrozumieć ogólne cechy zbioru danych. Oto najważniejsze miary opisowe:

- **Średnia arytmetyczna:** Jest to suma wszystkich wartości podzielona przez ich liczbę. Jest to najczęściej stosowana miara tendencji centralnej.

$$\text{Średnia} = \frac{\sum_{i=1}^{n} x_i}{n}$$

- **Mediana:** Jest to wartość, która dzieli zbiór danych na dwie równe części. Mediana jest szczególnie przydatna, gdy dane zawierają wartości skrajne, ponieważ nie jest na nie wrażliwa, w przeciwieństwie do średniej.
- **Moda:** Jest to wartość, która występuje najczęściej w zbiorze danych. Może być użyteczna, zwłaszcza w przypadku danych nominalnych lub kategorycznych.
- **Rozstęp (Range):** Jest to różnica między największą a najmniejszą wartością w zbiorze danych. Informuje o szerokości rozkładu danych.
- **Wariancja i odchylenie standardowe:** Wariancja mierzy rozproszenie danych wokół średniej, a odchylenie standardowe jest pierwiastkiem z wariancji. Obie miary pokazują, jak zmienne są dane w danym zbiorze.

  Wariancja = $\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$

  Odchylenie standardowe $\sqrt{wariancja}$

  Kurtosis: Mierzy "spiczastość" rozkładu danych. Wartość większa niż 3 oznacza, że rozkład ma bardziej spiczasty charakter niż rozkład normalny.

- **Skala:** Mierzy, jak symetryczny jest rozkład danych. Jeśli skala wynosi 0, rozkład jest symetryczny; dodatnia skala oznacza, że dane są przesunięte na lewo, a ujemna na prawo.

2. Testowanie hipotez

Testowanie hipotez to procedura statystyczna, która pozwala na weryfikację przypuszczeń dotyczących populacji na podstawie próby. W procesie testowania hipotez formułujemy dwie hipotezy:

- **Hipoteza zerowa (H_0):** Hipoteza, która sugeruje brak efektu, różnicy lub związku między badanymi zmiennymi. Jest to hipoteza, którą próbujemy obalić.
- **Hipoteza alternatywna (H_1):** Hipoteza, która sugeruje istnienie efektu, różnicy lub związku między badanymi zmiennymi.

Etapy testowania hipotez:

1. **Formułowanie hipotez:** Określenie hipotezy zerowej i alternatywnej.
2. **Wybór poziomu istotności (α):** Określamy, na jakim poziomie ryzyka jesteśmy gotowi odrzucić hipotezę zerową (np. α = 0,05).
3. **Obliczenie statystyki testowej:** Na podstawie danych próbki obliczamy statystykę testową, np. t-statystykę, z-test lub chi-kwadrat.
4. **Podejmowanie decyzji:** Na podstawie obliczonej statystyki i wartości p (p-value) decydujemy, czy odrzucić hipotezę zerową. Jeśli p < α, odrzucamy hipotezę zerową.

Przykłady testów:

- **Test t-Studenta:** Używany do porównania średnich dwóch grup.
- **Test chi-kwadrat:** Stosowany do analizy zależności między zmiennymi kategorycznymi.
- **Test ANOVA:** Służy do porównania średnich więcej niż dwóch grup.

3. Analiza danych

Analiza danych obejmuje różne techniki statystyczne, które umożliwiają wyciąganie wniosków z danych. Obejmuje to m.in.:

a) Analiza szeregów czasowych

Jest to analiza danych, które zostały zebrane w sposób chronologiczny, w celu uchwycenia wzorców, trendów, cykli oraz sezonowych wahań. Analiza ta jest wykorzystywana w prognozowaniu i identyfikacji zjawisk zależnych od czasu, jak np. ceny akcji czy temperatura powietrza.

- **Model ARIMA (AutoRegressive Integrated Moving Average):** Popularny model do prognozowania szeregów czasowych, który łączy autoregresję, różnicowanie oraz średnią ruchomą.

b) Analiza wariancji (ANOVA)

ANOVA jest metodą statystyczną służącą do porównania średnich między więcej niż dwoma grupami, by sprawdzić, czy różnice między nimi są statystycznie istotne.

- **Jednoczynnikowa ANOVA:** Sprawdza, czy średnie w kilku grupach różnią się od siebie, gdy porównuje się jeden czynnik (np. różne metody leczenia).
- **Wieloczynnikowa ANOVA:** Używana do analizy sytuacji, w których istnieje więcej niż jeden czynnik wpływający na wynik.

c) Analiza regresji

Regresja jest techniką statystyczną służącą do modelowania zależności między zmiennymi. Główne rodzaje to:

- **Regresja liniowa:** Modeluje liniowy związek między zmiennymi niezależnymi a zależną.

  $$Y=\beta_0+\beta_1X_1+\beta_2X_2+\cdots+\beta_nX_n$$

- **Regresja logistyczna:** Stosowana w przypadku zmiennej zależnej o charakterze binarnym (np. tak/nie).

4. Oprogramowanie statystyczne

Oprogramowanie statystyczne to narzędzia wspierające analizę danych i testowanie hipotez. Do najczęściej używanych należą:

- **R:** Jest to jeden z najpotężniejszych języków do analizy statystycznej, z bogatą biblioteką narzędzi do analizy danych, w tym analiz szeregów czasowych, regresji, testów hipotez itp.

- **SPSS:** Popularne oprogramowanie do analizy danych, które jest bardziej przystępne dla osób bez zaawansowanej wiedzy statystycznej. Używane głównie w naukach społecznych.
- **Python (biblioteki takie jak Pandas, NumPy, SciPy, StatsModels):** Używane do analizy danych w formie kodu, często stosowane w połączeniu z Machine Learning i dużymi danymi.
- **MATLAB:** Oprogramowanie do obliczeń numerycznych i analiz, popularne w inżynierii i naukach przyrodniczych.

5. Eksploracja danych i interpretacja wyników

Eksploracja danych (Data Exploration) to proces wstępnej analizy danych w celu znalezienia ukrytych wzorców i zależności. W tym etapie nie używamy jeszcze formalnych testów statystycznych, ale próbujemy zrozumieć dane. Wykorzystuje się takie techniki jak:

- **Wizualizacja danych:** Tworzenie wykresów (np. histogramy, wykresy pudełkowe, wykresy rozrzutu) w celu lepszego zrozumienia struktury danych.
- **Analiza współzależności:** Sprawdzanie, czy zmienne są ze sobą skorelowane, przy użyciu współczynnika korelacji.
- **Wykrywanie anomalii:** Identyfikowanie punktów odstających, które mogą być błędami w danych lub interesującymi przypadkami do dalszej analizy.

Systemy wbudowane

1. Budowa i zasada działania mikrokontrolera

Mikrokontroler to niewielki komputer wbudowany w jedno układ scalony, który zawiera wszystkie podstawowe elementy niezbędne do przetwarzania danych i sterowania różnymi urządzeniami. Mikrokontrolery są wykorzystywane w szerokim zakresie zastosowań, od prostych urządzeń elektronicznych po bardziej złożone systemy wbudowane. Oto podstawowe elementy mikrokontrolera:

- **Procesor (CPU):** To jednostka centralna mikrokontrolera, która wykonuje instrukcje zawarte w programie. Procesor składa się z jednostki arytmetyczno-logicznej (ALU), rejestrów, jednostki sterującej i pamięci cache.
- **Pamięć (RAM, ROM, EEPROM):**
    - **RAM (Random Access Memory):** Jest to pamięć operacyjna, która przechowuje dane tymczasowe wykorzystywane przez procesor w trakcie pracy.
    - **ROM (Read-Only Memory):** Pamięć tylko do odczytu, która przechowuje program startowy (bootloader) i inne stałe dane.
    - **EEPROM (Electrically Erasable Programmable Read-Only Memory):** Pamięć, która może być programowana i kasowana elektrycznie. Jest wykorzystywana do przechowywania danych, które muszą być zachowane nawet po wyłączeniu zasilania (np. konfiguracja urządzenia).
- **Wejścia/Wyjścia (I/O):** Mikrokontroler posiada piny wejść/wyjść, które pozwalają na komunikację z zewnętrznymi urządzeniami. Te piny mogą pełnić funkcje cyfrowe (np. włączanie diody LED) lub analogowe (np. odczytanie wartości z czujnika).

- **Timer i liczniki:** Mikrokontroler często zawiera wbudowane timery, które pozwalają na precyzyjne mierzenie czasu, generowanie przerwań lub realizację operacji w regularnych odstępach czasu.
- **Magistrale komunikacyjne:** Mikrokontroler może zawierać różne magistrale, takie jak SPI, I2C, UART, które umożliwiają komunikację z innymi urządzeniami.

Zasada działania mikrokontrolera polega na wykonaniu programu, który jest zapisany w pamięci ROM. Program steruje różnymi układami peryferyjnymi (czujnikami, silnikami, wyświetlaczami itp.) poprzez odpowiednie operacje na pinach I/O. Mikrokontroler może pracować w różnych trybach: ciągłym (tzw. polling), przetwarzania przerwań lub w trybie uśpienia, oszczędzając energię.

2. Systemy wbudowane sterowane mikrokontrolerami

Systemy wbudowane to złożone urządzenia elektroniczne, w których mikrokontroler pełni rolę centralnego elementu sterującego. Te systemy wykonują określone zadania, a ich działanie jest zazwyczaj w pełni zautomatyzowane i kontrolowane przez oprogramowanie w mikrokontrolerze. Przykłady systemów wbudowanych sterowanych mikrokontrolerami:

- **Urządzenia IoT (Internet of Things):** Mikrokontrolery w urządzeniach IoT służą do zbierania danych z czujników (np. temperatura, wilgotność), przetwarzania ich i wysyłania informacji do innych urządzeń lub chmurowych serwerów za pomocą technologii komunikacyjnych (np. Wi-Fi, Bluetooth).
- **Systemy automotive (motoryzacyjne):** Mikrokontrolery są powszechnie stosowane w samochodach, gdzie sterują systemami takich jak ABS, kontrola trakcji, airbag, a także w systemach infotainment czy nawigacyjnych.
- **Sprzęt AGD:** Mikrokontrolery sterują pracą urządzeń gospodarstwa domowego, takich jak pralki, lodówki, kuchenki mikrofalowe, umożliwiając precyzyjne zarządzanie procesami, takimi jak temperatura, czas czy automatyczne programy.
- **Robotyka:** W robotyce mikrokontrolery są używane do sterowania robotami mobilnymi, manipulatorami, a także do obsługi czujników i aktuatorów.
- **Systemy medyczne:** Mikrokontrolery w systemach medycznych kontrolują urządzenia takie jak defibrylatory, monitory EKG, pompy infuzyjne, umożliwiając zbieranie danych i kontrolowanie procesów terapeutycznych.

3. Programowanie mikrokontrolerów

Programowanie mikrokontrolerów polega na pisaniu oprogramowania, które steruje pracą mikrokontrolera i umożliwia realizację zamierzonych funkcji. Programowanie mikrokontrolerów zwykle odbywa się przy użyciu języków takich jak C, C++ lub asemblera, chociaż nowoczesne mikrokontrolery wspierają również wyższe poziomy języków (np. Python).

- **Język C:** Jest najbardziej popularnym językiem do programowania mikrokontrolerów. Zapewnia bezpośredni dostęp do sprzętu, umożliwiając tworzenie wydajnych i zoptymalizowanych aplikacji.

- **Język asemblera:** Programowanie w asemblerze pozwala na bardzo precyzyjne kontrolowanie mikrokontrolera, ale jest bardziej czasochłonne i trudniejsze do zrozumienia.
- **IDE (Integrated Development Environment):** Do programowania mikrokontrolerów używa się specjalnych środowisk programistycznych, takich jak Arduino IDE, MPLAB X IDE (dla mikrokontrolerów PIC), STM32CubeIDE (dla mikrokontrolerów STM32) oraz Atmel Studio.

Programowanie mikrokontrolerów wymaga znajomości takich zagadnień jak: konfiguracja wejść/wyjść, obsługa przerwań, komunikacja z urządzeniami peryferyjnymi, korzystanie z timerów, a także zarządzanie pamięcią.

4. Pamięci w systemach wbudowanych

Pamięci w systemach wbudowanych pełnią kluczową rolę w przechowywaniu danych oraz programu. W systemach wbudowanych najczęściej używane są następujące typy pamięci:

- **ROM (Read-Only Memory):** Pamięć tylko do odczytu, która przechowuje program startowy (bootloader), jak również podstawowe oprogramowanie systemu wbudowanego. Zwykle jest to pamięć nieulotna, co oznacza, że jej zawartość nie jest tracona po wyłączeniu zasilania.
- **RAM (Random Access Memory):** Pamięć operacyjna, która jest używana do przechowywania danych tymczasowych, takich jak zmienne i bufory. Pamięć ta jest ulotna, tzn. dane są tracone po wyłączeniu zasilania.
- **EEPROM (Electrically Erasable Programmable Read-Only Memory):** Pamięć, którą można elektrycznie zapisać i kasować, ale z ograniczoną liczbą cykli zapisu. Jest używana do przechowywania danych, które muszą być zachowane po wyłączeniu zasilania (np. ustawienia użytkownika, konfiguracja systemu).
- **Flash Memory:** Pamięć flash to rodzaj pamięci nieulotnej, która jest wykorzystywana do przechowywania dużych ilości danych, w tym oprogramowania. Jest szybsza niż EEPROM i ma większą pojemność.

5. Układy peryferyjne i magistrale transmisji danych

Układy peryferyjne to zewnętrzne urządzenia, które współpracują z mikrokontrolerem, umożliwiając realizację różnych funkcji systemu wbudowanego. Peryferia mogą obejmować:

- **Wejścia/wyjścia cyfrowe (GPIO):** Służą do sterowania urządzeniami cyfrowymi, np. diodami LED, przyciskami, przełącznikami.
- **Układy analogowe:** Mikrokontrolery często zawierają przetworniki analogowo-cyfrowe (ADC) do odczytu sygnałów analogowych oraz przetworniki cyfrowo-analogowe (DAC) do generowania sygnałów analogowych.
- **Interfejsy komunikacyjne:** Mikrokontrolery obsługują różne interfejsy komunikacyjne, takie jak:
    - **I2C (Inter-Integrated Circuit):** Stosowany do komunikacji z urządzeniami o niskiej prędkości.
    - **SPI (Serial Peripheral Interface):** Używany do szybszej komunikacji z urządzeniami peryferyjnymi.

    - **UART (Universal Asynchronous Receiver-Transmitter):** Interfejs szeregowy do komunikacji z komputerem lub innymi urządzeniami.
    - **USB:** Używany do komunikacji z komputerami i urządzeniami peryferyjnymi.
- **Magistrale transmisji danych:** Mikrokontrolery mogą używać różnych magistrali do komunikacji z innymi układami lub urządzeniami. Do najczęściej stosowanych magistrali należą I2C, SPI i CAN (Controller Area Network) w systemach motoryzacyjnych.

Układy peryferyjne i magistrale transmisji danych umożliwiają mikrokontrolerowi współpracę z zewnętrznymi urządzeniami, takimi jak czujniki, silniki, ekrany LCD, moduły komunikacyjne (Wi-Fi, Bluetooth), co czyni systemy wbudowane bardziej funkcjonalnymi.

1. Rodzaje pamięci wewnętrznych oraz zewnętrznych i ich zastosowania.
2. Mikrokontroler w układach sterowania urządzeń elektronicznych.
3. Proszę określić sposób połączenia pinów mikrokontrolera AVR z wyprowadzeniami alfanumerycznego wyświetlacza LCD.
4. Proszę uzupełnić fragment kodu, którego zadaniem jest wyświetlenie tekstu „Akku 1” na środku pierwszej linii wyświetlacza LCD o organizacji 16x2. W drugiej linii wyświetlacza należy umieścić tekst „Voltage V=” z wyświetleniem zmiennej ‘Vol’ za znakiem równości. Proszę zadeklarować zmienną ‘Vol’, której wartości należą do przedziału (-32 768 – +32 767). Narzędzie programowania – kompilator BASCOM.

```
……..?
Cls
……..?
LCD „Accu 1”
……..?
……..?
END
```

1. Rodzaje pamięci wewnętrznych oraz zewnętrznych i ich zastosowania

Pamięci wewnętrzne w systemach mikrokontrolerów są pamięciami bezpośrednio dostępnymi dla mikrokontrolera, zawartymi w tym samym układzie scalonym. Do najczęściej używanych pamięci wewnętrznych należą:

- **Pamięć ROM (Read-Only Memory):** Jest to pamięć tylko do odczytu, która przechowuje stałe dane, takie jak programy startowe lub oprogramowanie układowe (firmware). W systemach wbudowanych wykorzystywana do przechowywania kodu, który nie zmienia się w trakcie pracy urządzenia.
- **Pamięć Flash:** Jest to rodzaj pamięci nieulotnej, która może przechowywać programy i dane, ale w przeciwieństwie do ROM-u, może być zapisywana i kasowana elektrycznie. Pamięć flash jest bardzo popularna w nowoczesnych mikrokontrolerach i jest wykorzystywana do przechowywania kodu programu.
- **Pamięć RAM (Random Access Memory):** Pamięć operacyjna, która przechowuje dane tymczasowe, zmienne, bufory. Jest to pamięć ulotna, co oznacza, że dane w niej zapisane są tracone po wyłączeniu zasilania.

Pamięci zewnętrzne to pamięci, które nie są częścią mikrokontrolera, lecz są podłączane do niego zewnętrznie. Do najbardziej powszechnych pamięci zewnętrznych należą:

- **Pamięć EEPROM (Electrically Erasable Programmable Read-Only Memory):** Jest to pamięć, którą można kasować i programować elektrycznie. Służy do przechowywania danych, które muszą być zachowane po wyłączeniu urządzenia (np. konfiguracja).
- **Karty SD/CF (Secure Digital/Compact Flash):** Są to zewnętrzne nośniki pamięci, które mogą przechowywać duże ilości danych, takie jak pliki, obrazy, dane użytkownika. W mikrokontrolerach z reguły używane do przechowywania dużych ilości danych (np. w systemach IoT).
- **Pamięci USB:** Mikrokontrolery mogą być połączone z pamięciami USB, które służą do wymiany danych między urządzeniami.

2. Mikrokontroler w układach sterowania urządzeń elektronicznych

Mikrokontrolery pełnią kluczową rolę w systemach sterowania, ponieważ zapewniają elastyczność, kontrolę nad procesami i łatwość integracji z urządzeniami zewnętrznymi. W układach sterowania mikrokontroler może:

- **Monitorować dane wejściowe:** Mikrokontroler odczytuje dane z czujników, takich jak temperatury, wilgotności, czy poziomu ciśnienia. Na podstawie tych danych podejmuje decyzje o dalszym działaniu.
- **Kontrolować urządzenia wyjściowe:** Mikrokontroler może sterować elementami wykonawczymi, takimi jak silniki, serwomechanizmy, diody LED, wyświetlacze, itp.
- **Komunikować się z innymi urządzeniami:** Dzięki różnym interfejsom komunikacyjnym (UART, SPI, I2C, USB) mikrokontroler może współpracować z innymi układami i urządzeniami, tworząc kompleksowe systemy.
- **Zarządzać procesami:** W systemach wbudowanych mikrokontroler pełni rolę jednostki sterującej, która zarządza pracą innych układów peryferyjnych, takich jak przetworniki A/C, przetworniki D/A, regulatory napięcia, itp.

Przykłady zastosowań mikrokontrolerów w układach sterowania to m.in. sterowanie pracą urządzeń AGD (pralki, lodówki), systemy sterowania oświetleniem, robotyka, systemy alarmowe czy automatyka przemysłowa.

3. Połączenie pinów mikrokontrolera AVR z wyprowadzeniami alfanumerycznego wyświetlacza LCD

Połączenie mikrokontrolera AVR (np. ATmega) z wyświetlaczem LCD (np. 16x2) jest jednym z podstawowych zastosowań mikrokontrolerów. Wyświetlacze LCD z interfejsem alfanumerycznym mają dwa tryby komunikacji: równoległy i szeregowy. Zwykle stosuje się połączenie równoległe, gdzie mikrokontroler wykorzystuje kilka pinów do komunikacji z wyświetlaczem.

Przykładowe połączenie pinów mikrokontrolera AVR (np. ATmega328P) z wyświetlaczem LCD 16x2:

- **RS (Register Select):** Pin służący do wyboru trybu pracy wyświetlacza (komenda lub dane). Pin ten jest podłączony do dowolnego pinu mikrokontrolera (np. PB0).

- **RW (Read/Write):** Pin do określenia kierunku komunikacji. Dla trybu zapisu można ustawić stan niski. Pin podłączony do mikrokontrolera (np. PB1).
- **E (Enable):** Pin włączający komunikację z wyświetlaczem. Pin ten jest podłączony do mikrokontrolera (np. PB2).
- **D0-D7 (Data):** Piny danych, które przesyłają dane do wyświetlacza. W przypadku 8-bitowego połączenia podłączamy 8 pinów mikrokontrolera do wyświetlacza LCD (np. PD0–PD7).
- **VSS (Ground):** Zasilanie GND, podłączone do masy.
- **VDD (Power):** Zasilanie +5V, podłączone do źródła zasilania.
- **V0 (Contrast):** Pin do regulacji kontrastu. Zwykle podłączony do potencjometru.
- **BL (Backlight):** Pin do sterowania podświetleniem (jeśli jest dostępne).

4. Fragment kodu do wyświetlania tekstu na wyświetlaczu LCD w BASCOM

Zadanie: Wyświetlanie tekstu na wyświetlaczu LCD 16x2, z tekstem „Akku 1” na środku pierwszej linii i zmienną 'Vol' w drugiej linii.

Kod w BASCOM:

```
$regfile = "m8def.dat"       ' Definicja mikrokontrolera (ATmega8)
$crystal = 8000000           ' Częstotliwość zegara 8 MHz

Dim Vol As Integer           ' Deklaracja zmiennej 'Vol' (typ całkowity)

' Inicjalizacja LCD
Config Lcd = 16 * 2          ' Konfiguracja LCD 16x2
Config Lcdpin = Pinb.0 , Pinb.1 , Pinb.2 , Pinb.3 , Pinb.4 , Pinb.5 , Pinb.6 , Pinb.7

Cls                          ' Czyści ekran

' Wyświetlenie "Akku 1" na środku pierwszej linii
Lcd "   Akku 1   "           ' Wyświetla tekst w pierwszej linii

' Wyświetlenie tekstu "Voltage V=" i wartości zmiennej 'Vol' w drugiej linii
' Załóżmy, że zmienna 'Vol' ma wartość 5000
Vol = 5000                   ' Przykładowa wartość zmiennej 'Vol'
Lcdcr()                      ' Przechodzi do drugiej linii
Lcd "Voltage V="; Vol        ' Wyświetla tekst i wartość zmiennej

End
```

Wyjaśnienie:

- **$regfile = "m8def.dat"** - Określa używany mikrokontroler (ATmega8).
- **$crystal = 8000000** - Określa częstotliwość zegara mikrokontrolera (8 MHz).
- **Dim Vol As Integer** - Deklaracja zmiennej 'Vol', która będzie przechowywać wartość liczbową w przedziale od -32 768 do 32 767.
- **Config Lcd = 16 * 2** - Konfiguracja wyświetlacza LCD 16x2.
- **Lcd " Akku 1 "** - Wyświetlanie tekstu „Akku 1” na środku pierwszej linii.
- **Lcd "Voltage V="; Vol** - Wyświetlanie tekstu „Voltage V=” oraz wartości zmiennej 'Vol' w drugiej linii.

Programowanie robotów

1. Podstawy robotyki: definicja robota i robotyki; rodzaje robotów (mobilne, stacjonarne, przemysłowe, humanoidalne, serwisowe); podstawowe elementy budowy robota

Definicja robota: Robot to urządzenie mechaniczne, które jest zaprogramowane do wykonywania określonych zadań autonomicznie lub półautonomicznie. Roboty mogą być wykorzystywane w różnych dziedzinach, takich jak przemysł, medycyna, edukacja, a także w eksploracji kosmicznej.

Definicja robotyki: Robotyka to dziedzina inżynierii i technologii, która zajmuje się projektowaniem, budowaniem, programowaniem i użytkowaniem robotów. Robotyka łączy elementy mechaniki, elektroniki, informatyki i sztucznej inteligencji.

Rodzaje robotów:

- **Roboty mobilne**: Roboty, które mogą się poruszać w przestrzeni, np. roboty mobilne wykorzystywane w eksploracji terenów nieznanych (np. roboty marsjańskie). Zwykle wyposażone są w koła, gąsienice lub nogi, a ich zadaniem jest poruszanie się po danym terenie i zbieranie danych.
- **Roboty stacjonarne**: Roboty, które są zamocowane w jednym miejscu i wykonują operacje w określonym obszarze roboczym. Przykładem są roboty chirurgiczne lub roboty wykorzystywane w magazynach do sortowania.
- **Roboty przemysłowe**: To roboty wykorzystywane głównie w przemyśle, np. w fabrykach, do zautomatyzowanego wykonywania powtarzalnych zadań, takich jak montaż, malowanie, spawanie czy pakowanie. Są one zazwyczaj robotami stacjonarnymi, ale mogą być również mobilne w przypadku transportu materiałów.
- **Roboty humanoidalne**: Roboty zaprojektowane na wzór człowieka, zarówno pod względem wyglądu, jak i ruchów. Przykładem może być robot ASIMO firmy Honda, który potrafi chodzić, rozpoznać twarze, a także rozmawiać z ludźmi.
- **Roboty serwisowe**: To roboty, które wykonują różnorodne usługi, takie jak pomoc w domu (roboty sprzątające, roboty do cięcia trawy), roboty medyczne, roboty do transportu w magazynach i lotniskach, czy roboty dostawcze w miastach.

Podstawowe elementy budowy robota:

- **Czujniki (sensory)**: Służą do zbierania informacji o otoczeniu robota. Sensory mogą obejmować czujniki dotykowe, odległości, kamery, sensory temperatury, wilgotności, ciśnienia itp.
- **Aktuatory**: Elementy, które wykonują fizyczne zadania, takie jak ruch czy manipulacja obiektami. Aktuatory mogą być elektryczne (silniki), hydrauliczne lub pneumatyczne.
- **Mikrokontrolery i systemy komputerowe**: To „mózg” robota, który przetwarza dane z sensorów i steruje aktuatorami na podstawie programu. Mogą być to mikroprocesory lub układy FPGA.

- **Zasilanie**: Zasilanie elektryczne, które zasila roboty, w tym akumulatory, ogniwa słoneczne, czy systemy zasilania kablowego.
- **Struktura mechaniczna**: Szkielet robota, który zapewnia stabilność i pozwala na przechowywanie komponentów, takich jak silniki, czujniki i procesory.

2. Algorytmy unikania przeszkód i poruszania się w przestrzeni

Algorytmy unikania przeszkód są niezbędne do umożliwienia robotom poruszania się w dynamicznych środowiskach, gdzie napotykają na różne przeszkody. Do najpopularniejszych algorytmów unikania przeszkód należą:

- **Algorytm sztucznego potencjału**: Robot porusza się w kierunku minimalizacji potencjalnej energii w przestrzeni roboczej. Przeszkody generują potencjalną „odpychającą" siłę, a cel generuje „przyciągającą" siłę.
- **Algorytmy lokalizacji i mapowania (SLAM)**: Robot buduje mapę otoczenia i jednocześnie lokalizuje swoje położenie w tej mapie, unikając przeszkód w dynamicznym otoczeniu.
- **Algorytmy sztucznej inteligencji**: Wykorzystanie AI w robotach umożliwia im adaptację do zmieniających się warunków, np. algorytmy uczenia maszynowego pozwalają robotowi uczyć się z doświadczenia i unikać przeszkód w bardziej skuteczny sposób.

3. *Algorytmy planowania ścieżki (A, Dijkstra, BFS, DFS)**

Algorytmy planowania ścieżki są stosowane w robotyce, aby określić najlepszą trasę od punktu startowego do celu. Wybór odpowiedniego algorytmu zależy od rodzaju środowiska (czy jest ono dynamiczne, statyczne, itp.).

- **A* (A-star)**: Jest to najbardziej popularny algorytm dla problemu planowania ścieżki. Łączy on algorytm Dijkstry z heurystyką, co pozwala na bardziej efektywne znajdowanie najkrótszej drogi. Wykorzystuje funkcję kosztu f(n) = g(n) + h(n), gdzie g(n) to koszt dojścia do węzła, a h(n) to heurystyka (szacowany koszt dotarcia do celu).
- **Dijkstra**: Jest to algorytm, który znajduje najkrótszą ścieżkę w grafie o dodatnich wagach krawędzi. Działa w sposób podobny do A*, ale bez użycia heurystyki, przez co jest mniej efektywny w przypadkach złożonych.
- **BFS (Breadth-First Search)**: Algorytm przeszukiwania wszerz, który znajduje najkrótszą ścieżkę w grafie, ale działa wolniej niż A* w środowiskach złożonych, ponieważ eksploruje wszystkie węzły na tym samym poziomie przed przejściem na wyższy poziom.
- **DFS (Depth-First Search)**: Algorytm przeszukiwania w głąb, który eksploruje możliwe ścieżki do końca przed przejściem do innych opcji. Jest bardziej kosztowny czasowo w porównaniu do BFS i A*, ponieważ może prowadzić do ślepych zaułków.

4. Rodzaje sensorów: sensory kontaktowe, ultradźwiękowe, LIDAR, kamery, GPS, IMU

- **Sensory kontaktowe**: Czujniki, które wykrywają fizyczny kontakt z obiektem. Przykładem są czujniki siły, dotyku, nacisku, wykorzystywane w robotach do oceny siły nacisku lub wykrywania obecności obiektu.
- **Sensory ultradźwiękowe**: Wykorzystują fale dźwiękowe o wysokiej częstotliwości do pomiaru odległości od obiektów. Są powszechnie wykorzystywane w robotyce do unikania przeszkód i nawigacji.
- **LIDAR (Light Detection and Ranging)**: To technologia wykorzystywana do mapowania przestrzeni i wykrywania przeszkód. LIDAR emituje wiązkę światła i mierzy czas, w jakim powraca ona do czujnika, umożliwiając tworzenie map 3D.
- **Kamery**: Kamery RGB lub kamery głębi (np. kamery 3D) służą do przechwytywania obrazów otoczenia robota, które mogą być następnie analizowane przez algorytmy rozpoznawania obrazów i służyć do podejmowania decyzji o nawigacji.
- **GPS (Global Positioning System)**: System satelitarny wykorzystywany do określania położenia robota w przestrzeni. Stosowany głównie w robotach mobilnych, które poruszają się na dużych obszarach na zewnątrz.
- **IMU (Inertial Measurement Unit)**: Zestaw czujników, takich jak akcelerometry, żyroskopy i magnetometry, służący do określania orientacji robota w przestrzeni. Jest szeroko stosowany w robotach mobilnych, dronach, systemach nawigacji i sterowania.

5. Sensory do wykrywania przeszkód i nawigacji oraz zasady ich działania (np. sonar, czujniki podczerwieni)

- **Sonar (ultradźwiękowy)**: Sensory sonaru wysyłają fale dźwiękowe i mierzą czas ich powrotu, co pozwala określić odległość od przeszkód w najbliższym otoczeniu. Są powszechnie używane w robotach mobilnych do nawigacji i unikania przeszkód.
- **Czujniki podczerwieni (IR)**: Czujniki podczerwieni wykrywają zmiany w poziomie promieniowania IR odbitego od przeszkód w otoczeniu robota. Są szeroko stosowane w robotach do wykrywania bliskich przeszkód, monitorowania otoczenia i komunikacji.

1. **Dokładność i Precyzja**

dokładność odnosi się do stopnia, w jakim robot wykonuje zadanie lub precyzyjnie trafia w zaplanowane miejsce, porównując to do wartości referencyjnej lub docelowej. Jest to miara "bliskości" pomiędzy rzeczywistą lokalizacją robota (lub jego efektu działania) a pożądaną lokalizacją (np. pozycją, do której ma dotrzeć).

Precyzja odnosi się do zdolności robota do powtarzania tych samych działań lub pomiarów w tych samych warunkach, osiągając podobne wyniki. Jest to miara, która mówi o tym, jak spójne są wyniki działania robota przy wielokrotnym powtórzeniu tego samego zadania.

2. Robot mobilny

- **Definicja:** Robot, który może poruszać się w przestrzeni roboczej, zmieniając swoją lokalizację, np. roboty przemysłowe, które poruszają się po fabrykach lub roboty autonomiczne poruszające się w przestrzeni publicznej (np. roboty sprzątające).

3. Robot manipulator

- **Definicja:** Robot przeznaczony do manipulowania obiektami w określonym obszarze roboczym. Zwykle składa się z ramienia robota z kilkoma przegubami, które umożliwiają mu poruszanie i manipulowanie obiektami.

4. Przegub (ang. Joint)

- **Definicja:** Element mechaniczny robota, który łączy dwa segmenty robota i umożliwia im wzajemny ruch. W robotach manipulatorach przeguby pozwalają na ruchy w różnych kierunkach (np. obroty, przesunięcia).

5. Stopień swobody (DOF - Degrees of Freedom)

- **Definicja:** Liczba niezależnych ruchów, które robot może wykonać. Na przykład, w przypadku robota manipulatora, stopnie swobody mogą oznaczać możliwość obracania się w różnych osiach (np. 3 stopnie swobody to możliwość ruchu wzdłuż 3 osi: X, Y, Z).

6. Czujniki (Sensor)

- **Definicja:** Urządzenia, które pozwalają robotowi zbierać informacje o swoim otoczeniu, jak odległość do przeszkód, temperatura, siła nacisku itp. Czujniki mogą obejmować kamery, czujniki odległości (LIDAR), czujniki siły, czy czujniki dotyku.

7. Siłowniki (Actuators)

- **Definicja:** Urządzenia, które wykonują fizyczne ruchy robota na podstawie sygnałów sterujących. Siłowniki to np. silniki elektryczne, hydrauliczne siłowniki czy pneumatyczne siłowniki, które powodują ruch w przegubach robota.

8. Sterowanie (Control)

- **Definicja:** Proces, w którym robot jest kierowany do wykonania określonych zadań. W robotyce sterowanie może być realizowane za pomocą algorytmów, które analizują dane z czujników i przekształcają je na działania wykonywane przez siłowniki.

9. Autonomiczność

- **Definicja:** Zdolność robota do wykonywania zadań bez ciągłego nadzoru człowieka. Autonomiczne roboty mogą samodzielnie podejmować decyzje, na podstawie analizy swojego otoczenia i zapisanego programu.

10. Algorytmy śledzenia trajektorii

- **Definicja:** Algorytmy, które pozwalają robotowi śledzić określoną ścieżkę lub trajektorię. Są używane do precyzyjnego poruszania robota w przestrzeni roboczej, uwzględniając zmiany w otoczeniu.

11. Sztuczna inteligencja w robotyce

- **Definicja:** Wykorzystanie algorytmów AI do umożliwienia robotom podejmowania decyzji, rozwiązywania problemów lub uczenia się na podstawie doświadczeń. Może to obejmować rozpoznawanie obrazów, analizowanie danych z czujników, czy podejmowanie decyzji w zmieniających się warunkach.

12. Robotyka współpracująca (Collaborative Robotics)

- **Definicja:** Zastosowanie robotów, które współpracują z ludźmi w bezpośrednim sąsiedztwie. W przeciwieństwie do tradycyjnych robotów przemysłowych, które działają w odizolowanych obszarach, roboty współpracujące mogą dzielić przestrzeń roboczą z ludźmi, pomagając w wykonywaniu zadań.

13. Robotyka przemysłowa

- **Definicja:** Zastosowanie robotów w środowisku przemysłowym, takim jak montaż, pakowanie, malowanie czy spawanie. Roboty przemysłowe są wykorzystywane w wielu gałęziach przemysłu, w tym w motoryzacji, elektronice, a także w logistyce.

14. Robotyka mobilna

- **Definicja:** Rodzaj robotów, które mogą poruszać się w przestrzeni. Roboty mobilne mogą poruszać się na różnych powierzchniach, w tym na kołach, gąsienicach, a także chodzić lub latać.

15. Roboty humanoidalne

- **Definicja:** Roboty zaprojektowane w celu naśladowania ludzkiego wyglądu i zachowań. Mogą mieć ludzką postać (głowa, ręce, nogi) i przeznaczone są do interakcji z ludźmi w różnych dziedzinach, takich jak edukacja, opieka czy rozrywka.

16. Roboty autonomiczne

- **Definicja:** Roboty, które podejmują decyzje i wykonują zadania bez ludzkiego nadzoru. Działają na podstawie wcześniej zaprogramowanych algorytmów i reakcji na zmieniające się warunki otoczenia.

17. Roboty mobilne autonomiczne (AMR - Autonomous Mobile Robots)

- **Definicja:** Roboty, które potrafią poruszać się autonomicznie w przestrzeni, na przykład w magazynach, fabrykach czy przestrzeniach publicznych. Korzystają z czujników i mapowania w czasie rzeczywistym (np. LIDAR) do orientacji w otoczeniu.

18. ROS (Robot Operating System)

- **Definicja:** Otwarte, oparte na Linuxie oprogramowanie, które dostarcza zestaw narzędzi i bibliotek do budowania robotów. ROS ułatwia programowanie i integrację różnych komponentów robota, takich jak sensory, siłowniki czy algorytmy sterowania.

1. Podaj definicje i kryteria podziałów robotów
2. Struktury kinematyczne robotów. Napędy robotów mobilnych. Wady, zalety.
3. Omów zasadę pracy silnika krokowego.
4. Omów różnice między mechanizacją, automatyzacją i robotyzacją
5. Omów zasadę działania enkodera. Podaj rodzaje i przykłady zastosowania enkoderów.
6. Omów budowę mikrokontrolera i wyjaśnij sposób jego programowania

1. Definicje i kryteria podziałów robotów

Definicja robota:
Robot to urządzenie automatyczne, które może wykonywać zaprogramowane lub autonomiczne czynności w rzeczywistym świecie. Może współpracować z ludźmi lub wykonywać zadania w środowiskach, w których niezbędna jest precyzja, powtarzalność i szybkość.

Kryteria podziału robotów:

1. **Zdolność do poruszania się**:
   - **Roboty mobilne**: Mogą poruszać się w przestrzeni, np. roboty autonomiczne, drony.
   - **Roboty stacjonarne**: Nie mają zdolności do poruszania się, np. roboty przemysłowe montujące, spawające.
2. **Zastosowanie**:
   - **Roboty przemysłowe**: Wykorzystywane głównie w produkcji, np. do montażu, malowania, spawania.
   - **Roboty humanoidalne**: Zbudowane na wzór człowieka, np. roboty ASIMO, roboty pomocnicze.
   - **Roboty serwisowe**: Służą do wykonywania zadań w gospodarstwach domowych, np. roboty sprzątające.
   - **Roboty mobilne autonomiczne (AMR)**: Roboty używane w logistyce, eksploracji lub do pracy w trudnym terenie, np. roboty marsjańskie.
3. **Budowa i napęd**:
   - **Roboty oparte na kołach**: Wykorzystują koła lub gąsienice do poruszania się (np. roboty mobilne).
   - **Roboty oparte na nogach (bipedalne, czworonożne)**: Wykorzystują nogi do poruszania się (np. robota Boston Dynamics – Spot).
   - **Roboty hybrydowe**: Łączą różne mechanizmy napędowe.
4. **Sposób sterowania**:
   - **Roboty autonomiczne**: Samodzielnie podejmują decyzje na podstawie przetwarzanych danych (np. roboty używane w eksploracji).
   - **Roboty zdalnie sterowane**: Sterowane przez operatora zdalnie, np. w przypadku robotów wojskowych.

2. Struktury kinematyczne robotów. Napędy robotów mobilnych. Wady, zalety.

Struktury kinematyczne robotów: Roboty mogą mieć różne układy kinematyczne, zależnie od liczby osi i sposobu poruszania się:

- **Układ kinematyczny o jednej osi (1D)**: Ruch wzdłuż jednej osi, np. suwnica.
- **Układ kinematyczny o dwóch osiach (2D)**: Ruch w płaszczyźnie, np. ramię robota, które może poruszać się w dwóch kierunkach.
- **Układ kinematyczny o trzech osiach (3D)**: Roboty przemysłowe, np. manipulatory.
- **Roboty mobilne (6D)**: Mobilne roboty, które mają pełną swobodę ruchu w trzech wymiarach i mogą obracać się wokół osi, np. roboty mobilne wyposażone w koła.

Napędy robotów mobilnych:

- **Napęd na koła**: Jest najczęściej wykorzystywany w robotach mobilnych, ponieważ pozwala na dużą prędkość. Ma jednak ograniczoną zdolność poruszania się po nierównych powierzchniach.
    - **Zalety**: Duża prędkość, łatwa kontrola, mniejsze zużycie energii.
    - **Wady**: Trudności w poruszaniu się po nierównych powierzchniach, ograniczenia w manewrowaniu.
- **Napęd na gąsienice**: Używany w robotach mobilnych, które muszą poruszać się po trudnym terenie.
    - **Zalety**: Dobre poruszanie się po nierównych powierzchniach, stabilność.
    - **Wady**: Większe zużycie energii, mniejsza prędkość.
- **Napęd na nogi (roboty bipedalne i czworonożne)**: Używany w robotach humanoidalnych i robotach do eksploracji trudnych terenów.
    - **Zalety**: Duża manewrowość, możliwość pokonywania przeszkód.
    - **Wady**: Złożona konstrukcja, duże zużycie energii, większa podatność na awarie.

3. Omów zasadę pracy silnika krokowego

Silnik krokowy to rodzaj silnika elektrycznego, który wykonuje precyzyjne ruchy obrotowe w określonych krokach (ang. steps). Jego zasada działania opiera się na wykorzystaniu elektromagnesów, które przyciągają rotor do określonych pozycji. Zależnie od sterowania, silnik krokowy może wykonywać bardzo dokładne ruchy, co sprawia, że jest szeroko stosowany w robotyce, w urządzeniach wymagających precyzyjnego sterowania.

- **Rodzaje silników krokowych**:
    - **Silnik krokowy unipolarny**: Wymaga prostszego sterowania i jest łatwiejszy w implementacji.
    - **Silnik krokowy bipolarny**: Charakteryzuje się wyższą sprawnością i momentem obrotowym w porównaniu do silnika unipolarnego, ale wymaga bardziej skomplikowanego sterowania.

Zalety silników krokowych:

- Precyzyjne sterowanie pozycją.
- Wysoka niezawodność i długi okres użytkowania.

Wady:

- Wysokie zużycie energii.
- Potrzebują sterowników do dokładnego kontrolowania każdego kroku.

4. Omów różnice między mechanizacją, automatyzacją i robotyzacją

- **Mechanizacja** to proces zastępowania ręcznej pracy maszynami, które wykonują fizyczne zadania, ale wymagają ciągłego nadzoru operatora. Przykładem może być maszyna do cięcia, która wymaga ręcznego podawania materiału.
- **Automatyzacja** to wprowadzenie maszyn i urządzeń do wykonywania procesów, które wcześniej były wykonywane ręcznie, jednak w procesie automatyzacji maszyny wykonują zadania samodzielnie, bez potrzeby ingerencji operatora. Przykład: automatyczna linia produkcyjna, która wykonuje całą produkcję bez interwencji ludzi.
- **Robotyzacja** to zaawansowana forma automatyzacji, która polega na wprowadzeniu robotów do procesów produkcyjnych, które wykonują zadania precyzyjnie, autonomicznie, a w niektórych przypadkach mogą także adaptować się do zmieniających się warunków. Przykład: roboty przemysłowe montujące samochody.

5. Omów zasadę działania enkodera. Podaj rodzaje i przykłady zastosowania enkoderów

Zasada działania enkodera: Enkoder to urządzenie, które przekształca ruch mechaniczny (np. obrót wału) na sygnał elektryczny, który jest następnie używany do pomiaru położenia, prędkości lub kierunku ruchu. Enkoder składa się z wirującego elementu (np. tarczy) i czujników, które wykrywają zmiany w położeniu.

Rodzaje enkoderów:

- **Enkoder inkrementalny**: Rejestruje zmiany w pozycji obiektu, generując impulsy przy każdym kroku. Pozwala na określenie przyrostu pozycji, ale nie daje informacji o absolutnym położeniu.
- **Enkoder absolutny**: Określa absolutną pozycję obiektu w każdym momencie. Może generować unikalny kod dla każdej pozycji.

Zastosowania enkoderów:

- **Robotyka**: Enkodery wykorzystywane są do dokładnego śledzenia pozycji ramion robota.
- **Maszyny CNC**: Enkodery pozwalają na precyzyjne pozycjonowanie narzędzi.
- **Windy, taśmy transportowe**: Enkodery pomagają w ścisłym monitorowaniu pozycji w systemach przemysłowych.

6. Omów budowę mikrokontrolera i wyjaśnij sposób jego programowania

Budowa mikrokontrolera: Mikrokontroler to mały komputer na chipie, zawierający procesor, pamięć oraz zestaw układów wejścia/wyjścia. Składa się z:

- **CPU (Central Processing Unit)**: Jednostka przetwarzania, która wykonuje instrukcje.

- **Pamięć RAM**: Pamięć robocza, w której przechowywane są dane tymczasowe.
- **Pamięć ROM (lub Flash)**: Pamięć trwała, w której zapisany jest program do wykonania.
- **Porty I/O**: Interfejsy do komunikacji z zewnętrznymi urządzeniami (np. czujnikami, silnikami).
- **Timer/liczniki**: Układy do odliczania czasu lub liczenia zdarzeń.
- **A/D (Analog-to-Digital) i D/A (Digital-to-Analog)**: Przetworniki analogowo-cyfrowe i cyfrowo-analogowe, służące do konwersji sygnałów.

Programowanie mikrokontrolera: Mikrokontrolery programuje się w językach wysokiego poziomu (np. C, C++) lub w językach dedykowanych (np. BASCOM, Arduino). Programowanie mikrokontrolera odbywa się poprzez:

- **Tworzenie programu**: Pisanie kodu źródłowego, który wykonuje pożądane zadania.
- **Kompilacja**: Przekształcenie kodu źródłowego w instrukcje maszynowe, które mikrokontroler jest w stanie zrozumieć.
- **Wgrywanie programu do pamięci mikrokontrolera**: Program jest wgrywany do pamięci Flash mikrokontrolera, często za pomocą programatora (np. JTAG, ISP).

Programowanie sieciowe

1. Wielowątkowość w Javie (Runnable i Callable)

Wielowątkowość to technika pozwalająca na równoczesne wykonywanie kilku wątków w obrębie jednego programu. Jest to kluczowy mechanizm do wykonywania równoległych operacji, który poprawia wydajność aplikacji, szczególnie w przypadku zadań o dużym stopniu niezależności.

W Javie można implementować wielowątkowość na dwa główne sposoby:

a) Runnable

Interfejs `Runnable` jest najprostszym sposobem na tworzenie wątków w Javie. Klasa implementująca `Runnable` definiuje metodę `run()`, która zawiera kod do wykonania w osobnym wątku.

```
class MyRunnable implements Runnable {
   public void run() {
      System.out.println("Wątek uruchomiony!");
   }
}

public class Main {
   public static void main(String[] args) {
      Runnable myRunnable = new MyRunnable();
      Thread thread = new Thread(myRunnable);
      thread.start(); // Uruchamia wątek
   }
}
```

Zalety używania Runnable:

- Bardzo prosty sposób na uruchomienie wielu wątków.
- Umożliwia współdzielenie zasobów wątków (np. danych) przez różne wątki.

b) Callable

Interfejs Callable jest bardziej zaawansowaną wersją Runnable. W przeciwieństwie do Runnable, Callable pozwala na zwracanie wartości lub rzucanie wyjątków w trakcie wykonywania wątku. Metoda call() w interfejsie Callable zwraca wynik wykonania zadania (np. obiekt).

```
import java.util.concurrent.*;

class MyCallable implements Callable<String> {
   public String call() throws Exception {
      return "Wątek z wynikiem!";
   }
}

public class Main {
   public static void main(String[] args) throws InterruptedException, ExecutionException {
      ExecutorService executor = Executors.newSingleThreadExecutor();
      Callable<String> task = new MyCallable();
      Future<String> result = executor.submit(task); // Uruchomienie zadania
      System.out.println(result.get()); // Czekanie na wynik
      executor.shutdown();
   }
}
```

Zalety używania Callable:

- Możliwość zwracania wyników (np. danych), co jest przydatne w przypadku zadań, które muszą przetwarzać dane i zwracać wynik do głównego wątku.
- Obsługa wyjątków.

Różnice między Runnable a Callable:

- Runnable nie zwraca wyniku, podczas gdy Callable może zwrócić wynik.
- Runnable nie obsługuje wyjątków, ale Callable może je zgłaszać.

2. Aplikacje klient-serwer

Aplikacje klient-serwer to architektura, w której aplikacja jest podzielona na dwie części:

- **Klient**: Komponent, który inicjuje żądania i oczekuje odpowiedzi od serwera.
- **Serwer**: Komponent, który przetwarza żądania od klientów, wykonuje odpowiednie operacje i zwraca wyniki.

Rodzaje aplikacji klient-serwer:

- **Model jedno-serwerowy**: Jeden serwer obsługujący wielu klientów.
- **Model wielo-serwerowy**: Rozdzielenie obciążenia na wiele serwerów, np. w chmurze.
- **Serwer aplikacji**: Serwer wykonujący operacje biznesowe, np. serwery HTTP lub serwery baz danych.

Przykład w Javie: Klient w Javie wysyła żądanie do serwera, który je przetwarza i zwraca odpowiedź.

- **Serwer**:

```
import java.io.*;
import java.net.*;

public class Server {
   public static void main(String[] args) throws IOException {
      ServerSocket serverSocket = new ServerSocket(12345);
      while (true) {
         Socket socket = serverSocket.accept();
         BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
         PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
         String request = in.readLine();
         out.println("Odpowiedź serwera: " + request);
      }
   }
}
```

- **Klient**:

```
import java.io.*;
import java.net.*;

public class Client {
   public static void main(String[] args) throws IOException {
      Socket socket = new Socket("localhost", 12345);
      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

      out.println("Hello, Server!");
      String response = in.readLine();
      System.out.println(response);
   }
}
```

Zalety aplikacji klient-serwer:

- Centralizacja danych i zarządzania (serwer przechowuje dane, a klienci mogą je przetwarzać).
- Zwiększenie bezpieczeństwa poprzez oddzielność klienta od serwera (serwer może implementować mechanizmy ochrony danych).
- Łatwiejsza skalowalność, bo serwer może obsługiwać wielu klientów.

3. Wybrane mikro usługi i sposoby ich działania

Mikro usługi to podejście do tworzenia aplikacji, które dzieli system na małe, niezależne usługi, z których każda pełni określoną funkcję i może być rozwijana, wdrażana i skalowana niezależnie.

Przykłady mikro usług:

- **Usługa autentykacji**: Mikro usługa odpowiedzialna za logowanie i autentykację użytkowników.
- **Usługa płatności**: Mikro usługa obsługująca płatności i transakcje finansowe.
- **Usługa zarządzania produktami**: Mikro usługa zajmująca się zarządzaniem produktami w sklepie internetowym.
- **Usługa powiadomień**: Mikro usługa odpowiedzialna za wysyłanie e-maili lub powiadomień push.

Zasada działania mikro usług:

1. **Modularność**: Każda mikro usługa to osobny, samodzielny komponent, który komunikuje się z innymi przez API (często RESTful lub gRPC).
2. **Autonomia**: Każda mikro usługa jest odpowiedzialna za konkretną funkcjonalność i może być rozwijana i skalowana niezależnie.
3. **Zarządzanie danymi**: Mikro usługi zazwyczaj posiadają własne bazy danych, co zapewnia izolację danych.
4. **Komunikacja między usługami**: Komunikacja często odbywa się za pomocą protokołów HTTP (REST API) lub gRPC, umożliwiając asynchroniczne przetwarzanie danych.

Zalety mikro usług:

- **Skalowalność**: Możliwość skalowania poszczególnych usług w zależności od ich obciążenia.
- **Elastyczność**: Mikro usługi mogą być rozwijane w różnych technologiach.
- **Izolacja błędów**: Błąd w jednej mikro usłudze nie wpływa bezpośrednio na pozostałe.

Wyzwania mikro usług:

- **Złożoność zarządzania**: Potrzebna jest odpowiednia infrastruktura do monitorowania, wdrażania i skalowania usług.
- **Złożoność komunikacji**: Usługi muszą efektywnie komunikować się między sobą, co może prowadzić do opóźnień i błędów.

Przykład w Javie: Mikro usługi mogą być realizowane z wykorzystaniem frameworków takich jak Spring Boot lub Quarkus, które ułatwiają budowanie, wdrażanie i zarządzanie mikro usługami.

```
@RestController
public class ProductController {

   @GetMapping("/products")
   public List<Product> getAllProducts() {
      return productService.getAllProducts();
   }
}
```

W tym przykładzie mamy mikro usługę odpowiadającą za obsługę zapytań dotyczących produktów. Jest to część większego systemu, w którym każda mikro usługa pełni specyficzną rolę.

Podstawy programowania współbieżnego

1. Abstrakcja współbieżności

Współbieżność to zdolność systemu do wykonywania wielu operacji w tym samym czasie. W kontekście komputerów oznacza to możliwość uruchamiania wielu procesów lub wątków, które mogą działać równolegle lub przeplatać się, co daje efekt równoczesnego wykonywania wielu zadań.

Abstrakcja współbieżności to koncepcja, która pozwala programistom na modelowanie, zarządzanie i koordynowanie współbieżnych procesów i wątków w sposób uproszczony. W systemach operacyjnych współbieżność jest realizowana przy pomocy mechanizmów takich jak wątki, procesy, czy różne narzędzia synchronizacji (np. semafory, blokady).

W programowaniu współbieżnym można wyróżnić dwie główne techniki:

- **Wielowątkowość**: Tworzenie wielu wątków w ramach jednego procesu, gdzie każdy wątek może wykonywać inną część zadania.
- **Wieloprocesowość**: Tworzenie wielu procesów, gdzie każdy proces ma swoją przestrzeń adresową i działa niezależnie.

Przykład w Javie:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Wątek uruchomiony!");
   }
}

public class Main {
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();  // Uruchomienie nowego wątku
   }
}
```

2. Klasyczne problemy synchronizacji procesów

W systemach współbieżnych jednym z głównych wyzwań jest zapewnienie, że procesy/wątki współdzielące zasoby nie spowodują błędów, takich jak **wyścigi** lub **deadlocki**. Do klasycznych problemów synchronizacji procesów należą:

a) Problem producenta i konsumenta

Jest to klasyczny problem, w którym jeden proces (producent) wytwarza dane, a inny proces (konsument) je przetwarza. Problemy pojawiają się, gdy konsument próbuje pobrać dane, które nie zostały jeszcze wytworzone przez producenta, lub producent próbuje dodać dane do pełnego bufora.

b) Problem filozofów jedzących

Pięciu filozofów siedzi przy okrągłym stole, każdy z nich ma talerz i widelec. Filozofowie mogą jeść tylko wtedy, gdy mają oba widelce. Problem polega na synchronizacji dostępu do tych samych zasobów (widełek), aby uniknąć **deadlocków** (zablokowania) i zapewnienia sprawiedliwego dostępu.

c) Problem wyścigu

Problem wyścigu występuje, gdy dwa lub więcej procesów/ wątków próbuje jednocześnie zmieniać wspólny zasób. Może to prowadzić do nieprzewidywalnych wyników, jeśli nie zastosuje się odpowiednich mechanizmów synchronizacji.

3. Narzędzia synchronizacji procesów

Do synchronizacji procesów i wątków w systemach operacyjnych służą różne mechanizmy:

a) Semafory

Semafor to zmienna lub struktura danych, która jest używana do kontrolowania dostępu do wspólnych zasobów w systemie. Istnieją dwa rodzaje semaforów:

- **Semafory binarne** (dwa stany: 0 i 1, np. dla blokady)
- **Semafory liczniki** (można przydzielać różne liczby dostępnych zasobów)

Semafor może mieć dwie podstawowe operacje:

- wait(): Zmniejsza wartość semafora czekaj jeżeli semafor ma wartość 0 .
- signal(): Zwiększa wartość semafora co skutkuje obudzeniem wątków.

b) Muteksy (mutex)

Muteks (z ang. mutual exclusion) to mechanizm używany do zapewnienia, że tylko jeden proces (lub wątek) może uzyskać dostęp do wspólnego zasobu w danym momencie. Jest to bardziej specyficzna forma semafora, która zapewnia wzajemne wykluczenie.

c) Zmienna warunkowa

Zmienna warunkowa to mechanizm synchronizacji, który pozwala wątkom czekać na spełnienie określonego warunku. Pozwala to na np. "obudzenie" wątku, gdy zasób stanie się dostępny.

d) Blokady (Locks)

Blokada jest używana do synchronizacji dostępu do zasobu. Może być w różnych formach, np. **reentrant lock**, **rwlock** (odczyt-zapis) i inne.

e) Bariera synchronizacji

Bariera synchronizacji to mechanizm, który synchronizuje wątki, zmuszając je do oczekiwania na siebie nawzajem do momentu, aż wszystkie osiągną określony punkt w programie.

4. Procesy ciężkie i procesy lekkie w systemie Linux

- **Procesy ciężkie (ang. Heavyweight Processes - HWP)** to procesy, które mają swoją oddzielną przestrzeń adresową i są pełnoprawnymi jednostkami wykonawczymi. Każdy proces w systemie Linux jest procesem ciężkim. Procesy ciężkie są całkowicie izolowane od siebie.
- **Procesy lekkie (ang. Lightweight Processes - LWP)**, to wątki, które dzielą tę samą przestrzeń adresową i są zarządzane przez główny proces. W systemie Linux procesy lekkie są realizowane za pomocą mechanizmu wątków, które współdzielą zasoby procesu, ale mogą wykonywać różne zadania.

Zalety procesów lekkich:

- Mniejsze zużycie zasobów w porównaniu do procesów ciężkich.
- Możliwość wykonywania zadań równolegle w ramach jednego procesu.

Wady:

- Potrzebują one synchronizacji, aby uniknąć błędów związanych z dostępem do współdzielonych zasobów.

5. Komunikacja międzyprocesowa (IPC)

IPC (Interprocess Communication) to zbiór metod pozwalających na wymianę danych pomiędzy różnymi procesami. W systemach współczesnych procesy są oddzielnymi jednostkami, posiadającymi własną przestrzeń adresową, dlatego muszą komunikować się za pomocą różnych metod.

a) Pamięć dzielona (Shared Memory)

Jest to technika IPC, która pozwala procesom na dostęp do wspólnej przestrzeni pamięci. Jednym z procesów może być odpowiedzialny za zapis, a drugi za odczyt z tej pamięci. Pamięć dzielona jest najszybszą metodą IPC, ale wymaga odpowiedniej synchronizacji, aby uniknąć konfliktów.

b) Kanały (Pipes)

Pipes to mechanizm umożliwiający przesyłanie danych między procesami w formie strumienia. W systemie Linux występują dwa główne typy:

- **Pipe anonimowe**: komunikacja między procesami, które są ze sobą powiązane (np. rodzic i dziecko).
- **Pipe nazwane (FIFOs)**: umożliwiają komunikację między dowolnymi procesami.

c) Sygnały (Signals)

Sygnały to mechanizm komunikacji, który pozwala procesowi wysłać powiadomienie lub przerwanie do innego procesu. Sygnały mogą być używane do przerywania procesów lub informowania o różnych zdarzeniach (np. SIGINT dla przerwania procesu).

d) Kolejki komunikatów (Message Queues)

Kolejki komunikatów to struktury danych, które umożliwiają przesyłanie komunikatów między procesami. Procesy mogą wysyłać komunikaty do kolejki, a odbiorca może je odczytać w dogodnym czasie. Kolejki są przydatne w systemach, w których potrzebne jest przesyłanie dużych ilości danych lub komunikatów.

e) Gniazda (Sockets)

Gniazda to mechanizm komunikacji międzyprocesowej, który pozwala na przesyłanie danych między procesami, nawet jeśli znajdują się one na różnych maszynach. Są szeroko stosowane w aplikacjach sieciowych.

Podsumowanie

- **Abstrakcja współbieżności** umożliwia zarządzanie wieloma procesami i wątkami w systemie komputerowym.
- **Problemy synchronizacji procesów** obejmują takie wyzwania jak wyścigi czy deadlocki, które mogą wystąpić, gdy procesy współdzielą zasoby.
- **Narzędzia synchronizacji procesów** to semafory, mutexy, zmienne warunkowe i inne mechanizmy, które pomagają unikać problemów związanych z dostępem do zasobów.
- **Procesy ciężkie** to procesy z własną przestrzenią adresową, a **procesy lekkie** to wątki, które współdzielą przestrzeń z głównym procesem.
- **Komunikacja międzyprocesowa** (IPC) to zbiór technik umożliwiających wymianę danych między procesami, takich jak pamięć dzielona, sygnały, czy gniazda.

Programowanie aplikacji internetowych

1. Elementy niezbędne do wdrożenia i uruchomienia aplikacji stworzonej w technologii Java Enterprise Edition (Java EE)

Aby wdrożyć i uruchomić aplikację stworzoną w technologii Java EE, wymagane są następujące elementy:

a) Serwer aplikacji

Java EE jest platformą przeznaczoną do tworzenia aplikacji serwerowych. Aplikacje Java EE są uruchamiane w środowisku serwera aplikacji, który obsługuje wszystkie komponenty i funkcje platformy, takie jak servlety, JSP, EJB (Enterprise JavaBeans), czy JPA. Popularne serwery aplikacji to:

- **Apache Tomcat** (obsługuje servlet i JSP, ale nie obsługuje pełnej specyfikacji Java EE)
- **WildFly** (dawniej JBoss)
- **GlassFish**

- **WebLogic** (Oracle)
- **WebSphere** (IBM)

b) Baza danych

Większość aplikacji Java EE wymaga połączenia z bazą danych w celu przechowywania danych. Może to być dowolna baza danych (np. MySQL, PostgreSQL, Oracle), którą aplikacja łączy się za pomocą JDBC lub JPA.

c) JDK (Java Development Kit)

Aby rozwijać aplikacje w Java EE, konieczne jest posiadanie zainstalowanego JDK, które zawiera wszystkie narzędzia do kompilacji i uruchamiania aplikacji Java.

d) Środowisko IDE

Do efektywnego programowania w Java EE potrzebne jest odpowiednie środowisko programistyczne, które oferuje wsparcie dla Java EE, jak np.:

- **Eclipse IDE** (z dodatkiem do Java EE)
- **NetBeans IDE**
- **IntelliJ IDEA**

e) Biblioteki i frameworki

Java EE obejmuje zestaw bibliotek i API, które wspierają różne funkcjonalności aplikacji webowych, takie jak zarządzanie sesjami, interakcję z bazą danych (JPA, JDBC), tworzenie interfejsów użytkownika (JSF, JSP), czy implementację logiki biznesowej (EJB).

f) Konfiguracja aplikacji

Konfiguracja aplikacji Java EE odbywa się w plikach takich jak:

- **web.xml** (plik konfiguracyjny dla aplikacji webowych, w którym definiujemy serwlety, filtry, oraz mapowanie URL)
- **persistence.xml** (plik konfiguracyjny JPA)

2. Środowiska programistyczne wspierające programowanie w Java EE

Aby efektywnie programować w Java EE, programista może skorzystać z dedykowanych środowisk IDE, które wspierają rozwój aplikacji w tej technologii. Najpopularniejsze środowiska to:

a) Eclipse IDE

Eclipse to jedno z najpopularniejszych darmowych środowisk IDE, które oferuje wiele wtyczek umożliwiających rozwój aplikacji Java EE. Dla tego środowiska dostępny jest zestaw wtyczek **Eclim** i **JSDT**, które wspierają rozwój aplikacji w Java EE, jak i integrację z serwerami aplikacji (np. Tomcat, WildFly, GlassFish).

b) NetBeans

NetBeans to kolejne środowisko, które oferuje szeroką obsługę Java EE. Oferuje takie funkcjonalności jak edytor kodu, automatyczne uzupełnianie kodu, narzędzia do pracy z bazami danych i serwerami aplikacji.

c) IntelliJ IDEA

IntelliJ IDEA to zaawansowane środowisko IDE, które oferuje pełną obsługę Java EE, w tym wsparcie dla serwletów, JSP, JPA, EJB i innych technologii. Posiada funkcje automatycznego refaktoryzowania kodu oraz łatwą konfigurację serwerów aplikacji.

3. Serwlet: budowa i zasada działania

Serwlet to specjalny komponent w aplikacjach webowych Java EE, który służy do obsługi żądań HTTP i generowania odpowiedzi. Jest to klasa Java, która implementuje interfejs javax.servlet.Servlet. Działa w środowisku serwera aplikacji i odpowiada na żądania przychodzące z przeglądarek internetowych.

Budowa:

- **Klasa serwletu** implementuje interfejs javax.servlet.Servlet lub rozszerza klasę HttpServlet (dla aplikacji HTTP).
- **Metody**:
    - doGet(): Obsługuje żądania HTTP GET (pobieranie danych).
    - doPost(): Obsługuje żądania HTTP POST (wysyłanie danych).
    - init(): Inicjalizuje serwlet.
    - destroy(): Zamyka serwlet po zakończeniu jego pracy.

Zasada działania:

1. Przeglądarka wysyła żądanie HTTP do serwera aplikacji.
2. Serwer aplikacji mapuje żądanie do odpowiedniego serwletu.
3. Serwlet przetwarza żądanie i generuje odpowiedź (np. HTML, JSON, XML).
4. Odpowiedź jest zwracana do przeglądarki.

Przykład prostego serwletu:

```
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<h1>Hello, world!</h1>");
   }
}
```

4. Sposoby tworzenia i obsługi sesji na przykładzie aplikacji internetowej Java EE

W aplikacjach Java EE sesje użytkowników są używane do przechowywania danych między żądaniami HTTP. Sesja może być tworzona na poziomie serwletu lub aplikacji webowej i może zawierać dane, które będą dostępne dla użytkownika podczas całej jego wizyty na stronie.

a) Zarządzanie sesjami

W Java EE sesje są realizowane za pomocą obiektu HttpSession, który jest przypisany do każdego użytkownika odwiedzającego aplikację webową. Możemy przechowywać w nim dane, takie jak dane logowania, koszyk zakupowy, preferencje użytkownika.

b) Tworzenie sesji

Sesję tworzymy za pomocą:

```
HttpSession session = request.getSession(true);
```

c) Dodawanie danych do sesji

```
session.setAttribute("username", "john_doe");
```

d) Odczyt danych z sesji

```
String username = (String) session.getAttribute("username");
```

e) Usuwanie danych z sesji

```
session.removeAttribute("username");
```

f) Zamykanie sesji

```
session.invalidate();
```

5. Baza danych w aplikacji Java EE: porównanie Java Database Connectivity (JDBC) i Java Persistence API (JPA)

a) JDBC (Java Database Connectivity)

JDBC to interfejs API, który umożliwia aplikacjom Java komunikację z bazą danych za pomocą zapytań SQL. Jest to podejście bardziej "niskopoziomowe", wymagające ręcznego zarządzania połączeniami i wynikami zapytań.

- **Zalety**:
  - Większa kontrola nad zapytaniami i połączeniami.
  - Elastyczność w budowaniu zapytań SQL.
- **Wady**:
  - Konieczność ręcznego mapowania danych między bazą a aplikacją.
  - Trudniejsze zarządzanie transakcjami.

b) JPA (Java Persistence API)

JPA to API w Java EE, które umożliwia mapowanie obiektów Java na tabele bazy danych (ORM - Object-Relational Mapping). JPA upraszcza pracę z bazą danych, automatyzując proces mapowania obiektów na rekordy w bazie i vice versa.

- **Zalety**:
  - Mniejsze obciążenie związane z mapowaniem danych.
  - Łatwiejsze zarządzanie transakcjami.
- **Wady**:
  - Mniejsza kontrola nad zapytaniami w porównaniu do JDBC.
  - Potrzebna jest dodatkowa konfiguracja.

6. Podstawy funkcjonowania i możliwości stron tworzonych przy użyciu frameworków: Java Server Pages (JSP), Java Server Faces (JSF)

a) Java Server Pages (JSP)

JSP to technologia Java służąca do tworzenia dynamicznych stron webowych. Strony JSP są w zasadzie plikami HTML, które mogą zawierać fragmenty kodu Java, który jest uruchamiany po stronie serwera.

- **Zalety**:
  - Rozdzielenie logiki biznesowej od prezentacji.
  - Łatwiejsze tworzenie dynamicznych treści.
- **Wady**:
  - Trudniejsze do utrzymania w przypadku rozbudowanych aplikacji.

b) Java Server Faces (JSF)

JSF to framework Java EE do tworzenia aplikacji webowych, który wspiera komponowanie UI w postaci komponentów. JSF oferuje m.in. mechanizmy do zarządzania zdarzeniami i stanem aplikacji, walidację danych, integrację z innymi technologiami Java EE.

- **Zalety**:
  - Ułatwia tworzenie i zarządzanie interfejsami użytkownika.
  - Integracja z innymi komponentami Java EE.
- **Wady**:
  - Może być bardziej złożony w porównaniu do innych technologii Java EE.

Obie technologie (JSP i JSF) mają swoje miejsce w aplikacjach Java EE, zależnie od skomplikowania projektu i wymagań dotyczących interfejsu użytkownika.

Administracja serwerami www

1. Algorytm nawiązywania połączenia HTTPS

Połączenie HTTPS (Hypertext Transfer Protocol Secure) jest rozszerzeniem protokołu HTTP, który zapewnia bezpieczną komunikację przez Internet, korzystając z SSL/TLS (Secure Socket Layer/Transport Layer Security). Proces nawiązywania połączenia HTTPS, znany jako "TLS handshake", jest kluczowy dla zapewnienia poufności i integralności danych wymienianych między klientem a serwerem. Algorytm ten składa się z kilku kroków:

a) Inicjacja połączenia

1. **ClientHello**: Klient (np. przeglądarka) rozpoczyna komunikację, wysyłając wiadomość ClientHello, w której zawarte są:
    - Protokół TLS, który będzie używany (np. TLS 1.2, TLS 1.3).
    - Lista obsługiwanych algorytmów szyfrowania (cipher suites).
    - Wygenerowany przez klienta losowy ciąg (client random), który będzie używany w późniejszych etapach do generowania kluczy sesyjnych.
2. **ServerHello**: Serwer odpowiada na wiadomość klienta, wysyłając wiadomość ServerHello, w której zawarte są:
    - Wybór algorytmu szyfrowania spośród dostępnych w ofercie klienta.
    - Losowy ciąg wygenerowany przez serwer (server random).
    - Certyfikat serwera (zawierający klucz publiczny) oraz ewentualne dodatkowe dane w zależności od wymagań protokołu.

b) Wymiana kluczy

3. **Wymiana kluczy publicznych**: Po ustaleniu algorytmu szyfrowania, klient generuje tzw. **pre-master secret** (tajny klucz wstępny) i szyfruje go za pomocą klucza publicznego serwera (otrzymanego z certyfikatu). Szyfrowany pre-master secret jest wysyłany do serwera.
4. **Obliczenie klucza sesyjnego**: Zarówno klient, jak i serwer obliczają klucz sesyjny (session key) na podstawie pre-master secret oraz obu losowych ciągów (client random i server random). Klucz sesyjny jest używany do szyfrowania danych wymienianych w ramach sesji.

c) Potwierdzenie

5. **Potwierdzenie klienta**: Klient wysyła wiadomość Finished, która jest szyfrowana za pomocą klucza sesyjnego i zawiera potwierdzenie, że klient pomyślnie przeprowadził wszystkie kroki w procesie ustalania bezpiecznego połączenia.
6. **Potwierdzenie serwera**: Serwer odpowiada wiadomością Finished, potwierdzając, że połączenie zostało nawiązane i że komunikacja będzie szyfrowana za pomocą ustalonego klucza sesyjnego.

Po tym procesie, klient i serwer mogą wymieniać dane bezpiecznie, a komunikacja jest szyfrowana za pomocą wspólnie ustalonego klucza sesyjnego.

2. Generowanie certyfikatów

Certyfikaty SSL/TLS są używane do zapewnienia autentyczności i bezpieczeństwa połączenia HTTPS. Proces generowania certyfikatu SSL obejmuje kilka etapów:

1. **Tworzenie klucza prywatnego**: Klucz prywatny jest generowany w pierwszej kolejności. Jest on trzymany w tajemnicy na serwerze i wykorzystywany do szyfrowania oraz podpisywania danych.
2. **Tworzenie żądania certyfikatu (CSR - Certificate Signing Request)**: Na podstawie klucza prywatnego tworzony jest plik CSR, który zawiera:
    - Publiczny klucz (część certyfikatu).
    - Dane organizacyjne (nazwa, adres, kraj, domena).
    - Inne informacje, które zostaną uwzględnione w certyfikacie.
3. **Wysłanie CSR do Centrum Certyfikacji (CA)**: CSR jest wysyłane do zaufanego centrum certyfikacji (CA), które zweryfikuje informacje zawarte w żądaniu.
4. **Weryfikacja przez CA**: Centrum Certyfikacji weryfikuje tożsamość właściciela domeny oraz inne dane, a następnie wydaje certyfikat SSL/TLS, który zawiera:
    - Klucz publiczny.
    - Informacje o właścicielu certyfikatu.
    - Podpis cyfrowy CA, który potwierdza autentyczność certyfikatu.
5. **Instalacja certyfikatu na serwerze**: Certyfikat wydany przez CA jest instalowany na serwerze. Teraz serwer jest w stanie uwierzytelnić swoją tożsamość w ramach połączenia HTTPS, a klient może zweryfikować ważność certyfikatu poprzez jego podpis cyfrowy.

3. Główne typy serwerów proxy

Serwer proxy to pośrednik pomiędzy klientem a serwerem docelowym. Proxy może pełnić różne funkcje, w tym poprawę wydajności, zapewnienie anonimowości i kontrolę nad dostępem do zasobów.

a) Forward Proxy:

Forward proxy działa pomiędzy klientem a Internetem. Klient wysyła żądania do proxy, które przekazuje je do docelowego serwera. Serwer proxy może również manipulować danymi, np. blokować dostęp do określonych stron, ukrywać adres IP użytkownika lub zapewniać cache.

b) Reverse Proxy:

Reverse proxy działa pomiędzy Internetem a serwerami aplikacyjnymi. Użytkownicy łączą się z serwerem proxy, który następnie przekazuje żądania do odpowiednich serwerów backendowych. Reverse proxy jest używane w celu rozkładania obciążenia, zapewnienia bezpieczeństwa i ukrywania struktury serwerów backendowych.

c) Transparent Proxy:

Transparent proxy działa w tle, nie ingerując w żądania użytkowników. Jego obecność nie jest zazwyczaj zauważalna przez użytkowników, ale może być używane w celu przechwytywania lub modyfikowania ruchu (np. w celu filtrowania treści w sieciach firmowych).

d) Caching Proxy:

Caching proxy przechowuje kopie wcześniej pobranych zasobów. Gdy klient ponownie wysyła żądanie o ten sam zasób, proxy dostarcza go z pamięci podręcznej, co znacząco zwiększa wydajność i zmniejsza obciążenie serwera źródłowego.

e) SOCKS Proxy:

SOCKS proxy obsługuje różne protokoły aplikacji (np. HTTP, FTP, SMTP) i zapewnia możliwość pełnej anonimowości użytkownika w sieci. Zamiast działać tylko na warstwie aplikacji, jak HTTP proxy, SOCKS działa na poziomie wyższym w stosie TCP/IP.

4. Funkcje wirtualnych hostów na serwerze HTTP/HTTPS

Wirtualne hosty pozwalają serwerowi WWW (np. Apache, Nginx) obsługiwać wiele domen na jednym serwerze fizycznym, umożliwiając zarządzanie różnymi stronami internetowymi za pomocą tego samego serwera.

a) Funkcje wirtualnych hostów

- **Izolacja aplikacji**: Każdy wirtualny host może być skonfigurowany z innymi ustawieniami, w tym innymi certyfikatami SSL, plikami logów, ścieżkami do katalogów oraz konfiguracjami aplikacji, co pozwala na uruchomienie wielu niezależnych stron na tym samym serwerze.
- **Rozdzielanie zasobów**: Wirtualne hosty umożliwiają przypisanie określonych zasobów (np. pamięci, pasma) dla różnych domen.
- **Zarządzanie certyfikatami SSL**: Serwer HTTP/HTTPS może obsługiwać różne certyfikaty SSL na różnych wirtualnych hostach, co pozwala na uruchomienie wielu bezpiecznych połączeń HTTPS na tym samym serwerze.
- **Zarządzanie ruchem**: Wirtualne hosty mogą kierować ruch do różnych aplikacji backendowych na podstawie domeny, co ułatwia organizowanie skomplikowanych architektur aplikacyjnych.
- **Równoważenie obciążenia**: Można skonfigurować wirtualne hosty do równoważenia obciążenia pomiędzy różnymi serwerami aplikacyjnymi, poprawiając dostępność i wydajność usług.

1. Omów zadania i możliwości oprogramowania serwerowego WWW.
2. Wymień podstawowe zagrożenia napotykane podczas prowadzenia serwisu WWW.
3. Opisz podstawowe zadania administratora serwera WWW

1. Omów możliwości współczesnych profesjonalnych urządzeń sieciowych

Współczesne profesjonalne urządzenia sieciowe, takie jak **routery**, **przełączniki** (switche), **firewalle**, **punkty dostępowe** (access points), **load balancery** oraz **bramy VPN**, oferują szereg zaawansowanych funkcji, które pozwalają na zarządzanie, zabezpieczanie i optymalizowanie infrastruktury sieciowej w firmach i organizacjach. Oto niektóre z kluczowych możliwości tych urządzeń:

a) Routery

- **Routowanie dynamiczne i statyczne**: Profesjonalne routery obsługują zarówno routowanie statyczne (gdzie administrator sam definiuje trasy), jak i dynamiczne (gdzie routery wymieniają informacje o trasach, np. z użyciem protokołów takich jak RIP, OSPF, EIGRP).
- **Zabezpieczenia i filtracja**: Nowoczesne routery wyposażone są w funkcje filtrowania ruchu (firewall), VPN oraz możliwość inspekcji pakietów w celu detekcji ataków.
- **QoS (Quality of Service)**: Umożliwiają zarządzanie pasmem i priorytetyzację ruchu, co pozwala na zapewnienie odpowiedniej jakości usług (np. dla VoIP, strumieni wideo).
- **Wirtualizacja sieci**: Niektóre routery wspierają technologie takie jak SD-WAN (Software-Defined WAN), umożliwiające elastyczne zarządzanie rozległymi sieciami.

b) Przełączniki (Switche)

- **Zarządzanie VLAN**: Nowoczesne switche umożliwiają tworzenie wirtualnych sieci lokalnych (VLAN), co pozwala na segmentację sieci i poprawę bezpieczeństwa oraz efektywności.
- **PoE (Power over Ethernet)**: Switche mogą dostarczać zasilanie do urządzeń końcowych (np. telefonów IP, kamer CCTV) za pomocą tego samego kabla Ethernet.
- **Zarządzanie pasmem**: Możliwość konfigurowania przepustowości i priorytetu ruchu w ramach różnych portów switcha.
- **Zarządzanie ruchami multicastowymi**: Obsługa protokołów takich jak IGMP snooping dla zarządzania multicastowym ruchem w sieci.

c) Firewalle

- **Inspekcja głębokiego pakietu (DPI)**: Współczesne firewalle oferują inspekcję całego ruchu wchodzącego i wychodzącego w sieci, w celu wykrywania i blokowania złośliwego oprogramowania, prób ataków, czy nieautoryzowanego dostępu.
- **Zarządzanie aplikacjami i użytkownikami**: Firewalle mogą klasyfikować i kontrolować aplikacje w sieci, umożliwiając blokowanie specyficznych aplikacji lub usług (np. BitTorrent).
- **VPN i IPSec**: Możliwość tworzenia i zarządzania połączeniami VPN, oferująca bezpieczne połączenia między oddzielnymi sieciami.

d) Punkty dostępowe (Access Points)

- **Wi-Fi 6 (802.11ax)**: Najnowszy standard Wi-Fi zapewnia większą przepustowość, lepsze zarządzanie ruchem i mniejsze opóźnienia, co jest istotne w zatłoczonych środowiskach.
- **Zarządzanie centralne**: Nowoczesne punkty dostępowe często oferują centralne zarządzanie, które pozwala na konfigurację, monitorowanie oraz diagnostykę wielu punktów dostępowych z jednego miejsca.
- **WPA3**: Najnowszy standard zabezpieczeń Wi-Fi, który oferuje silniejsze mechanizmy ochrony przed atakami typu brute force i poprawia prywatność użytkowników.

e) Load Balancery

- **Równoważenie obciążenia**: Profesjonalne urządzenia do równoważenia obciążenia mogą dynamicznie przydzielać ruch do różnych serwerów w zależności od ich obciążenia, zapewniając równomierne rozłożenie zasobów i wyższą dostępność usług.
- **SSL Offloading**: Rozszerzenie możliwości load balancerów, które mogą przejmować proces deszyfrowania ruchu SSL/TLS, odciążając serwery aplikacji od tego zadania.

f) Bramy VPN

- **Zarządzanie połączeniami VPN**: Bramki VPN zapewniają bezpieczne połączenia między różnymi sieciami (np. oddziałami firmy), umożliwiając pracę zdalną lub łączenie różnych oddzielnych sieci.
- **Obsługa różnych protokołów VPN**: Obsługują różne protokoły takie jak PPTP, L2TP, IPsec, SSL, a także nowoczesne rozwiązania jak MPLS dla bardziej skomplikowanych scenariuszy.

2. Omów tablicę routingu i protokoły routingu dynamicznego

a) Tablica routingu

Tablica routingu jest strukturą danych w routerze, która przechowuje informacje na temat tras dostępnych w sieci. Zawiera ona wpisy o dostępnych ścieżkach, a także o tym, jak je osiągnąć. Każdy wpis w tablicy routingu zawiera:

- **Adres docelowy**: Adres sieci, do którego ruch ma być kierowany.
- **Maska podsieci**: Określa zakres adresów w danej sieci.
- **Brama**: Adres routera, do którego należy wysłać pakiety, aby dotarły do odpowiedniej sieci.
- **Interfejs**: Fizyczny lub wirtualny interfejs sieciowy, przez który ruch powinien być wysyłany.
- **Metryka**: Miara jakości trasy, np. liczba hopów, opóźnienie, przepustowość.

b) Protokoły routingu dynamicznego

Protokoły routingu dynamicznego umożliwiają routerom automatyczne aktualizowanie swoich tablic routingu w odpowiedzi na zmiany w topologii sieci. Oto niektóre z najczęściej używanych protokołów:

- **RIP (Routing Information Protocol)**: Prosty protokół oparty na liczbie hopów. Stosowany głównie w mniejszych sieciach, ma ograniczenie do 15 hopów.
- **OSPF (Open Shortest Path First)**: Protokół link-state, który wykorzystuje algorytm Dijkstry do obliczania najkrótszej trasy. Jest skalowalny i wykorzystywany w dużych sieciach.
- **EIGRP (Enhanced Interior Gateway Routing Protocol)**: Protokół hybrydowy (łączący cechy protokołów link-state i distance-vector). Obsługuje zarówno routowanie w obrębie sieci, jak i w jej obrzeżach.
- **BGP (Border Gateway Protocol)**: Protokoł dynamicznego routingu stosowany do wymiany informacji o trasach między różnymi systemami autonomicznymi (AS), czyli dużymi sieciami (np. między dostawcami Internetu).

3. Omów zasadę działania VPN

VPN (Virtual Private Network) to technologia pozwalająca na stworzenie prywatnej, zaszyfrowanej sieci pomiędzy dwoma lub więcej punktami (np. urządzeniami, serwerami) przez publiczną sieć (np. Internet). VPN umożliwia bezpieczne połączenia i przesyłanie danych, chroniąc przed podsłuchaniem i atakami.

a) Zasada działania VPN

1. **Tunelowanie**: VPN tworzy tzw. "tunel" pomiędzy dwoma punktami, w którym cały ruch sieciowy jest szyfrowany. Tunel ten może być oparty na różnych protokołach (np. IPsec, SSL/TLS, L2TP).

2. **Szyfrowanie**: Dane przesyłane przez VPN są szyfrowane przy użyciu silnych algorytmów kryptograficznych (np. AES). Dzięki temu, nawet jeśli ktoś przechwyci transmisję, nie będzie w stanie jej odczytać.
3. **Autentykacja**: Przed nawiązaniem połączenia VPN odbywa się proces uwierzytelniania użytkownika, zazwyczaj za pomocą loginu i hasła, certyfikatów cyfrowych lub tokenów.
4. **Protokoły VPN**:
    - **IPsec**: Służy do zabezpieczania połączeń między urządzeniami w sieci, zapewniając integrację danych i poufność.
    - **SSL/TLS**: Protokoły, które zabezpieczają połączenie i są często używane w rozwiązaniach VPN typu client-to-site, zapewniając bezpieczeństwo przesyłanych danych.
    - **L2TP (Layer 2 Tunneling Protocol)**: Zwykle wykorzystywany z IPsec do zapewnienia szyfrowania i bezpieczeństwa połączenia.
5. **Zastosowanie VPN**: VPN jest powszechnie stosowane w firmach do zapewnienia bezpiecznego dostępu zdalnego do zasobów firmowych. Pozwala także na omijanie blokad geograficznych i anonimowe przeglądanie Internetu.

Kryptografia

1. Podstawowe elementy kryptografii

Kryptografia to dziedzina matematyki, która zajmuje się zapewnieniem bezpieczeństwa informacji przez ich szyfrowanie, uwierzytelnianie, integrację i podpisywanie. Podstawowe elementy kryptografii obejmują:

1. Szyfrowanie

a) Cel szyfrowania:

Szyfrowanie polega na przekształcaniu danych w taki sposób, aby stały się one niezrozumiałe dla nieuprawnionych odbiorców. Jest stosowane w celu zapewnienia poufności danych.

b) Rodzaje szyfrowania:

- **Szyfrowanie symetryczne**:
    - W tym modelu ten sam klucz jest używany zarówno do szyfrowania, jak i odszyfrowywania danych.
    - **Zalety**: Duża szybkość i efektywność, odpowiednia do szyfrowania dużych ilości danych.
    - **Wady**: Problem bezpiecznego przesyłania klucza.
    - **Przykłady algorytmów**: AES (Advanced Encryption Standard), DES (Data Encryption Standard), RC4.
- **Szyfrowanie asymetryczne**:
    - Wykorzystuje parę kluczy: klucz publiczny (do szyfrowania) i klucz prywatny (do odszyfrowywania).
    - **Zalety**: Bezpieczne przesyłanie klucza, łatwa wymiana danych między stronami.
    - **Wady**: Mniejsza wydajność w porównaniu do szyfrowania symetrycznego.

- **Przykłady algorytmów**: RSA, ECC (Elliptic Curve Cryptography).

2. Funkcje Skrótu (Hashing)

a) Definicja:

Funkcje skrótu przekształcają dane wejściowe o dowolnej długości w wynik o stałej długości (tzw. hash). Funkcje te są nieodwracalne – nie można odzyskać oryginalnych danych z ich skrótu.

b) Właściwości funkcji skrótu:

- **Deterministyczność**: Te same dane wejściowe zawsze generują ten sam wynik.
- **Efektywność**: Funkcja działa szybko i jest łatwa do obliczenia.
- **Odporność na kolizje**: Trudno znaleźć dwa różne zestawy danych generujące ten sam wynik.
- **Odporność na preimage**: Trudno odtworzyć dane wejściowe na podstawie skrótu.

c) Przykłady funkcji skrótu:

- **MD5 (Message Digest 5)**: Funkcja skrótu generująca 128-bitowe hashe. Uznana za przestarzałą z powodu podatności na kolizje.
- **SHA-1 (Secure Hash Algorithm 1)**: Funkcja generująca 160-bitowe hashe. Nie jest już uważana za bezpieczną.
- **SHA-256**: Popularna funkcja z rodziny SHA-2, generująca 256-bitowe hashe.
- **BLAKE2**: Nowoczesna funkcja skrótu, która jest szybsza i bardziej bezpieczna od SHA-2.

d) Zastosowania:

- Sprawdzanie integralności danych.
- Przechowywanie haseł.
- Podpisy cyfrowe.

3. Podpisy Cyfrowe

a) Definicja:

Podpis cyfrowy to kryptograficzny odpowiednik podpisu odręcznego, służący do potwierdzania tożsamości nadawcy i integralności danych.

b) Mechanizm działania:

- Nadawca generuje skrót wiadomości za pomocą funkcji skrótu.
- Skrót jest szyfrowany kluczem prywatnym nadawcy, tworząc podpis cyfrowy.
- Odbiorca odszyfrowuje podpis kluczem publicznym nadawcy i porównuje wynikowy skrót z własnym skrótem wiadomości.

c) Zastosowania:

- Uwierzytelnianie dokumentów.

- Weryfikacja transakcji w kryptowalutach.
- Podpisywanie oprogramowania.

d) Przykłady algorytmów:

- **RSA**: Podpis cyfrowy oparty na kryptografii asymetrycznej.
- **ECDSA (Elliptic Curve Digital Signature Algorithm)**: Podpisy cyfrowe na bazie krzywych eliptycznych, bardziej wydajne od RSA.

4. Certyfikaty Cyfrowe

a) Definicja:

Certyfikat cyfrowy to dokument elektroniczny, który potwierdza tożsamość podmiotu. Jest wydawany przez zaufane instytucje, zwane urzędami certyfikacji (Certificate Authority - CA).

b) Elementy certyfikatu:

- Klucz publiczny właściciela.
- Dane identyfikacyjne właściciela (np. nazwa, adres e-mail).
- Podpis cyfrowy urzędu certyfikacji.
- Data ważności certyfikatu.

c) Zastosowania:

- Bezpieczne połączenia HTTPS.
- Weryfikacja tożsamości w systemach komputerowych.

5. Uwierzytelnianie

Uwierzytelnianie polega na weryfikacji tożsamości użytkownika, urządzenia lub aplikacji. Jest kluczowym elementem bezpieczeństwa systemów informatycznych.

a) Rodzaje uwierzytelniania:

- **Jednoczynnikowe (SFA)**: Oparte na jednym czynniku, np. hasło.
- **Dwuczynnikowe (2FA)**: Łączy dwa różne czynniki, np. hasło i kod SMS.
- **Wieloczynnikowe (MFA)**: Wykorzystuje trzy lub więcej czynników, np. hasło, odcisk palca i token sprzętowy.

b) Metody uwierzytelniania:

- **Hasła i PIN-y**.
- **Certyfikaty cyfrowe**.
- **Biometria** (np. rozpoznawanie twarzy, odcisk palca).

6. Integralność

Integralność danych zapewnia, że informacje nie zostały zmienione w trakcie przesyłania lub przechowywania. Jest to osiągane za pomocą funkcji skrótu i podpisów cyfrowych.

7. Zarządzanie Kluczami Kryptograficznymi

Klucze kryptograficzne to fundament kryptografii. Zarządzanie nimi obejmuje:

- Generowanie kluczy.
- Bezpieczne przechowywanie (np. w modułach HSM - Hardware Security Modules).
- Dystrybucję kluczy (np. za pomocą infrastruktury PKI).
- Rotację kluczy w celu zwiększenia bezpieczeństwa.

8. Zastosowania Kryptografii

Kryptografia znajduje zastosowanie w wielu dziedzinach, takich jak:

- **Transmisja danych**: Zapewnienie poufności w komunikacji internetowej (HTTPS, VPN).
- **Bankowość elektroniczna**: Szyfrowanie transakcji i podpisywanie cyfrowe.
- **Systemy operacyjne**: Uwierzytelnianie użytkowników i szyfrowanie plików.
- **Blockchain**: Szyfrowanie, podpisy cyfrowe i zapewnienie integralności w kryptowalutach.

2. Schematy algorytmów szyfrowania symetrycznego

Szyfrowanie symetryczne polega na tym, że zarówno nadawca, jak i odbiorca używają tego samego klucza do szyfrowania i odszyfrowywania danych. Algorytmy szyfrowania symetrycznego charakteryzują się dużą efektywnością, ale wymaga to bezpiecznego przesyłania i przechowywania klucza.

a) AES (Advanced Encryption Standard)

AES jest jednym z najbardziej popularnych algorytmów szyfrowania symetrycznego. Stosuje on różne długości kluczy (128, 192, 256 bitów) i jest uważany za bezpieczny oraz efektywny. AES działa na blokach danych o rozmiarze 128 bitów i przechodzi przez różne rundy obróbki (szyfrowanie, podstawianie, permutacje).

b) DES (Data Encryption Standard)

DES jest starszym algorytmem szyfrowania symetrycznego, który używa 56-bitowego klucza do szyfrowania danych w blokach 64-bitowych. DES jest uważany za niebezpieczny w dzisiejszych czasach z powodu jego małej długości klucza, która jest podatna na ataki brute-force.

c) Triple DES (3DES)

Triple DES to wariant DES, który stosuje trzy rundy szyfrowania, każdą z innym kluczem (klucz 168 bitów). Jest bardziej bezpieczny niż standardowy DES, ale jest mniej wydajny niż AES.

d) RC4 (Rivest Cipher 4)

RC4 jest jednym z najstarszych i najprostszych algorytmów szyfrowania strumieniowego. Używa jednego klucza, który jest dynamicznie generowany i stosowany do przesyłanych danych. Pomimo jego szybkości, RC4 ma pewne luki bezpieczeństwa, dlatego nie jest zalecany w nowoczesnych systemach.

3. Szyfrowanie i schematy podpisów cyfrowych

a) Szyfrowanie asymetryczne (RSA)

Szyfrowanie asymetryczne jest mechanizmem, w którym stosowane są dwa różne klucze: publiczny (do szyfrowania) i prywatny (do odszyfrowywania). Jeden z najpopularniejszych algorytmów asymetrycznych to RSA. W RSA stosuje się operacje na dużych liczbach pierwszych, zapewniając silne bezpieczeństwo.

b) Podpisy cyfrowe RSA

Podpis cyfrowy RSA jest realizowany przez podpisanie wiadomości przy użyciu klucza prywatnego nadawcy, a odbiorca może sprawdzić podpis za pomocą klucza publicznego nadawcy. Proces podpisywania polega na obliczeniu skrótu wiadomości, a następnie zaszyfrowaniu go kluczem prywatnym.

c) ECDSA (Elliptic Curve Digital Signature Algorithm)

ECDSA to algorytm podpisu cyfrowego, który wykorzystuje kryptografię na krzywych eliptycznych. Jest bardziej efektywny niż RSA, oferując ten sam poziom bezpieczeństwa przy krótszych kluczach.

4. Kryptograficzne funkcje skrótu (Hash Functions)

Funkcje skrótu to algorytmy, które przyjmują dane o dowolnej długości i generują z nich wynik o stałej długości, nazywany hashem. Hash jest unikalnym identyfikatorem danych wejściowych. Funkcje skrótu są wykorzystywane m.in. do przechowywania haseł, weryfikacji integralności danych i tworzenia podpisów cyfrowych.

a) MD5 (Message Digest Algorithm 5)

MD5 jest jedną z najstarszych funkcji skrótu, która generuje 128-bitowy hash. Ze względu na podatność na kolizje (dwa różne zestawy danych generujące ten sam hash), MD5 nie jest już zalecany do stosowania w nowych systemach.

b) SHA-1 (Secure Hash Algorithm 1)

SHA-1 generuje 160-bitowy hash. Chociaż był stosowany w wielu systemach, podobnie jak MD5, jest uznawany za przestarzały, ponieważ odkryto w nim możliwość generowania kolizji.

c) SHA-256 (Secure Hash Algorithm 256-bit)

SHA-256 to jedna z funkcji skrótu należących do rodziny SHA-2, która generuje 256-bitowy hash. Jest szeroko stosowana i uważana za bezpieczną.

d) BLAKE2

BLAKE2 to nowoczesna funkcja skrótu, która zapewnia wysoką wydajność i bezpieczeństwo. Jest szybsza niż SHA-2 i jest stosowana w wielu nowoczesnych aplikacjach.

5. Schematy identyfikacji i uwierzytelniania

a) Hasła (Passwords)

Najprostszą formą identyfikacji jest używanie hasła. Użytkownik podaje hasło, które jest weryfikowane na podstawie przechowywanego haszowanego hasła w bazie danych.

b) Certyfikaty cyfrowe

Certyfikaty cyfrowe wykorzystują kryptografię asymetryczną do identyfikacji użytkowników. Każdy użytkownik ma parę kluczy (publiczny i prywatny), a certyfikat jest podpisany przez zaufane centrum certyfikacji (CA).

c) 2FA (Two-Factor Authentication)

2FA to proces uwierzytelniania, w którym użytkownik musi podać dwa elementy uwierzytelnienia, na przykład hasło oraz kod przesłany na urządzenie mobilne.

d) Biometria

Biometria opiera się na unikalnych cechach fizycznych lub behawioralnych użytkownika, takich jak odciski palców, rozpoznawanie twarzy czy analiza głosu.

6. Kryptografia na krzywych eliptycznych (Elliptic Curve Cryptography - ECC)

Kryptografia na krzywych eliptycznych to rodzaj kryptografii asymetrycznej, która opiera się na problemie rozwiązywania równań eliptycznych. Dzięki mniejszym wymaganiom co do rozmiaru klucza, kryptografia eliptyczna oferuje wysoki poziom bezpieczeństwa przy mniejszych zasobach obliczeniowych.

a) Zalety ECC

- **Wydajność**: Krótsze klucze w porównaniu do RSA o równym poziomie bezpieczeństwa, co skutkuje mniejszymi wymaganiami obliczeniowymi.
- **Bezpieczeństwo**: Krótsze klucze oznaczają mniejszy rozmiar pamięci i szybsze operacje, ale są równie trudne do złamania jak długie klucze RSA.
- **Przykłady zastosowań**: ECC jest stosowane w protokołach takich jak TLS, w podpisach cyfrowych (ECDSA) oraz w wymianie kluczy (ECDH).

Kryptografia jest fundamentem nowoczesnego bezpieczeństwa w Internecie i różnych systemach komputerowych, zapewniając poufność, integralność oraz uwierzytelnienie danych.

Podstawy modelowania i symulacji

1. Etapy Tworzenia Modelu Matematycznego i Kategorie Modelu

Model matematyczny to abstrakcyjna reprezentacja rzeczywistego układu, procesu lub zjawiska w postaci równań matematycznych. Tworzenie modelu obejmuje kilka kluczowych etapów:

Etapy tworzenia modelu matematycznego:

Definicja problemu (Sformułowanie problemu):

- Określenie celu modelowania, np. przewidywanie, optymalizacja, analiza zjawiska.
- Identyfikacja zmiennych wejściowych i wyjściowych oraz opisanie ich wzajemnych zależności.
- Sformułowanie założeń upraszczających, takich jak liniowość czy brak zakłóceń.

Zbieranie danych i założeń (Baza wiedzy):

- Analiza dostępnych danych empirycznych oraz przegląd literatury naukowej w danym obszarze.
- Upraszczanie rzeczywistości poprzez przyjęcie założeń (np. stałość parametrów, jednorodność układu).

Sformułowanie modelu matematycznego (Kategorie modelu):

- **Wybór typu modelu:** Określenie narzędzi matematycznych, takich jak równania różniczkowe, algebra liniowa, czy symulacje komputerowe.
- **Reprezentacja układu:** Przedstawienie systemu w postaci równań, funkcji, lub schematów.
- **Kategorie modeli matematycznych:**
    - **Dyskretne i ciągłe:**
        - *Dyskretne*: Zmienne przyjmują wartości całkowite (np. liczba osobników w populacji).
        - *Ciągłe*: Zmienne mogą przyjmować dowolne wartości w określonym zakresie (np. temperatura, koncentracja substancji).
    - **Deterministyczne i probabilistyczne:**
        - *Deterministyczne*: Wynik określony jednoznacznie na podstawie danych wejściowych (np. równania ruchu).
        - *Probabilistyczne*: Modele zawierające elementy losowości lub niepewności (np. modele populacji z fluktuacjami).
    - **Liniowe i nieliniowe:**

        - *Liniowe*: Relacje między zmiennymi są proporcjonalne i opisane równaniami liniowymi (np. obwody RC).
        - *Nieliniowe*: Relacje bardziej złożone, np. chaos deterministyczny.
    - **Statyczne i dynamiczne:**
        - *Statyczne*: Nie zależą od czasu, np. analiza obciążenia budynku.
        - *Dynamiczne*: Zmieniają się w czasie, np. układy równań różniczkowych opisujące dynamikę populacji.

Redukcja modelu:

- Upraszczanie modelu w celu ułatwienia obliczeń i interpretacji wyników.
- Walidacja modelu przez porównanie z danymi eksperymentalnymi lub rzeczywistością.

Wprowadzenie zmiennych:

- Identyfikacja i zdefiniowanie wszystkich zmiennych kluczowych dla modelu.

Napisać równanie modelu:

- Matematyczne przedstawienie związku między zmiennymi wejściowymi i wyjściowymi.

Próba rozwiązania:

- *Rozwiązanie analityczne*: Dokładne rozwiązania przy pomocy metod matematycznych.
- *Rozwiązanie numeryczne*: Przybliżone rozwiązania za pomocą algorytmów komputerowych.

Weryfikacja:

- Sprawdzenie poprawności modelu przez jego wykorzystanie w przewidywaniu, sterowaniu lub podejmowaniu decyzji.

Animacja i symulacja:

- Graficzna wizualizacja modelu w celu lepszego zrozumienia działania systemu i jego wyników.

3. Układy Pierwszego Rzędu (np. Układ RC)

Równanie $\frac{dx}{dt} = \alpha \cdot x(t)$

Rozwiązanie $x(t) = C * e^{\alpha \cdot t}$ **C- Stałą ulotną wylicza się za pomocą warunków początkowych**

A<0

Dq/dt =a*q(t) obwód RC

Dh/dt = a* h(t) wylewanie wody z wiaderka

dN/dt = a*N(t)

A > 0

Model rozwoju populacji Malthusa

dN/dt = a*N(T)

Układy pierwszego rzędu to systemy opisane równaniami różniczkowymi pierwszego rzędu. Są stosunkowo proste, ale znajdują szerokie zastosowanie w fizyce, biologii i inżynierii.

Przykład: Układ RC (rezystor-kondensator)

1. **Budowa:**
   - Układ RC składa się z rezystora (R) i kondensatora (C) połączonych szeregowo lub równolegle.
2. **Równanie różniczkowe:**
   - Równanie obwodu RC (szeregowego): $v(t) = R \cdot I(t) + \frac{1}{C}\int I(t)\,dt$
   - Po przekształceniu: $\frac{dv(t)}{\mathrm{dt}} + \frac{1}{RC}v(t) = 0$
3. **Charakterystyka:**
   - Odpowiedź czasowa ma postać wykładniczą: $v(t) = v_0 e^{-\frac{t}{Rc}}$
   - Stała czasowa τ=RC: określa szybkość zaniku napięcia.
4. **Zastosowanie:**
   - Filtry dolnoprzepustowe w elektronice.
   - Modelowanie układów ładowania i rozładowania w bateriach.

4. Układy Drugiego Rzędu (np. Ruch w Polu Centralnym, Drgania Sprzężone)

$$\ddot{\boldsymbol{X}}(\boldsymbol{t}) + \mathbf{2}\boldsymbol{\delta}\dot{\boldsymbol{x}}(\boldsymbol{t}) + w_0^2\boldsymbol{x}(\boldsymbol{t}) = u(\boldsymbol{t})$$

$2\delta -$ współczynnik odp. Za tłumienie

$w_0^2 = k/m$

$u(\boldsymbol{t}) - f.wymuszonej$

Przykład 1: Ruch w polu centralnym

- Opisuje ruch ciała w polu sił centralnych (np. grawitacyjnych).
- Równanie różniczkowe: $m\frac{d^2r}{dt^2} = -\frac{\mathrm{GMm}}{r^2}$

- Rozwiązanie: Elipsa, parabola lub hiperbola w zależności od energii układu.

Przykład 2: Drgania sprzężone

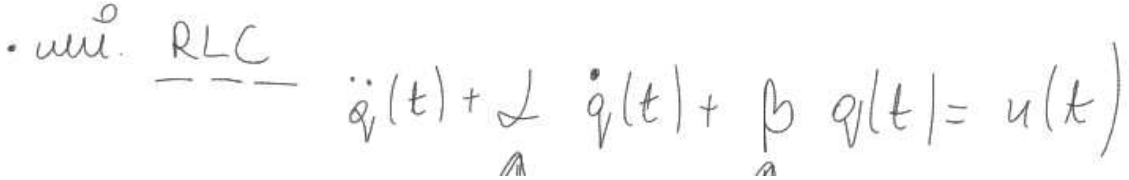
- Układ mas i sprężyn, gdzie drgania jednej masy wpływają na drugą.
- Równania:

$$m_1 \frac{d^2x_1}{dt^2} + k_1 x_1 - k_2(x_2 - x_1) = 0$$

$$m_2 \frac{d^2x_2}{dt^2} + k_2(x_2 - x_1) = 0$$

- Zastosowania:
  - Drgania mostów i budynków.
  - Akustyka i instrumenty muzyczne.

Układ RLC – opis i charakterystyka



Układy drugiego rzędu opisane są równaniami różniczkowymi drugiego rzędu i mają bardziej złożone zachowanie

Układ **RLC** to klasyczny obwód elektryczny, który składa się z trzech elementów:

- **R** – rezystor, który reprezentuje opór elektryczny w obwodzie,
- **L** – cewka indukcyjna, która wprowadza indukcyjność,
- **C** – kondensator, który wprowadza pojemność elektryczną.

Układ ten może być połączony w różne konfiguracje, takie jak szeregowy lub równoległy, a jego zachowanie opisują równania różniczkowe.

Rodzaje układów RLC

1. **Układ szeregowy RLC**
   - W tym układzie wszystkie elementy (R, L, C) są połączone szeregowo.
   - Całkowite napięcie w obwodzie jest sumą napięć na każdym z elementów.

- Równanie różniczkowe dla napięcia w układzie:

$$V(t) = R \cdot I(t) + L\frac{dI(t)}{dt} + \frac{1}{C}\int I(t)dt$$

- Charakterystyka częstotliwościowa (impedancja):

$$Z = R + j\left(\omega L - \frac{1}{\omega C}\right)$$

2. **Układ równoległy RLC**
   - W tym układzie elementy są połączone równolegle, a prąd płynie przez każdy z nich niezależnie.
   - Całkowity prąd w układzie jest sumą prądów płynących przez rezystor, cewkę i kondensator.
   - Równanie różniczkowe dla prądu w układzie:

$$I(t) = I_R(t) + I_L(t) + I_C(t)$$

4. Modelowanie Liczebności Populacji i Modele Epidemii

Równanie Malthusa to klasyczny model wzrostu populacji, który opisuje nieograniczony wzrost liczby osobników w idealnych warunkach, gdy zasoby są nieskończone, a czynniki ograniczające, takie jak przestrzeń czy pożywienie, nie mają wpływu. Można je zapisać w postaci:

$$\frac{dN}{dt} = a \cdot N(t)$$

Wyjaśnienie elementów równania:

1. **N(t):** Wielkość populacji w czasie ttt.
2. **dN/dt :** Tempo wzrostu populacji w czasie.
3. **a:** Współczynnik wzrostu, czyli stopa wzrostu populacji. Jest dodatni dla wzrostu i ujemny dla spadku liczebności.

Interpretacja:

- Równanie mówi, że tempo zmian populacji (dN/dt) jest proporcjonalne do aktualnej liczby osobników N(t).
- Wzrost ten jest wykładniczy, co oznacza, że populacja rośnie coraz szybciej w miarę upływu czasu, o ile a>0.

Rozwiązanie równania:

Rozwiązując równanie różniczkowe, otrzymujemy:

$N(t)=N_0 \cdot e^{a.t}$

- N_0: Populacja początkowa w czasie t=0.
- e: Podstawa logarytmu naturalnego (około 2,718).

Wnioski z modelu Malthusa:

1. **Wykładniczy wzrost:** Gdy a>0, populacja rośnie wykładniczo, co prowadzi do szybkiego wzrostu liczby osobników.
2. **Brak ograniczeń:** Model zakłada, że środowisko ma nieskończone zasoby, co jest mało realistyczne. W rzeczywistości wzrost populacji jest ograniczany przez czynniki środowiskowe.
3. **Zastosowanie:** Równanie to dobrze opisuje początkowe fazy wzrostu populacji, zanim pojawią się ograniczenia zasobów.

Ograniczenia modelu:

- Nie uwzględnia ograniczeń środowiskowych (np. pojemności środowiska K).
- Nie przewiduje efektów spowolnienia wzrostu populacji, które są uwzględniane w modelu logistycznym (model Verhulsta).

Przykłady zastosowania:

- Prognozowanie wzrostu populacji w krótkim okresie.
- Analiza procesów wzrostu wykładniczego w biologii (np. namnażanie się bakterii w idealnych warunkach).
- Ekonomia: wzrost kapitału przy stałej stopie procentowej.

Model populacji Verhulsta (model logistyczny)

Opis:

Model Verhulsta, zwany również modelem logistycznym, opisuje wzrost populacji, który jest ograniczony przez zasoby środowiska. Jest to bardziej realistyczne podejście niż model wzrostu wykładniczego, ponieważ uwzględnia ograniczenia środowiskowe, takie jak dostępność jedzenia, przestrzeni czy innych zasobów.

Równanie logistyczne:

$$\frac{dP}{dt} = r \cdot P \cdot \left(1 - \frac{P}{K}\right)$$

- P(t): populacja w czasie t,
- r: współczynnik wzrostu populacji,
- K: pojemność środowiska, maksymalna liczba osobników, którą środowisko może utrzymać.

Interpretacja:

1. Gdy populacja PPP jest mała ($P \ll K$), wzrost jest wykładniczy ($dP/dt \approx r \cdot P$).
2. W miarę wzrostu populacji ($P \to K$), tempo wzrostu maleje.
3. Gdy $P=K$, populacja osiąga równowagę i przestaje rosnąć ($dP/dt=0$).

Zastosowanie:

- Prognozowanie wzrostu populacji zwierząt lub ludzi.
- Modelowanie zasobów w ograniczonych środowiskach (np. produkcji w fabryce).

Model SIR (Susceptible-Infectious-Recovered)

Opis:

Model SIR jest matematycznym modelem używanym do opisu rozprzestrzeniania się chorób zakaźnych w populacji. Zakłada, że populacja jest podzielona na trzy grupy:

1. **S (Susceptible)** – osoby podatne na infekcję.
2. **I (Infectious)** – osoby zakażone, które mogą przekazywać chorobę.
3. **R (Recovered)** – osoby, które wyzdrowiały i nabyły odporność.

Równania modelu:

$$\frac{dS}{dt} = -\beta \cdot S \cdot I$$

$$\frac{dI}{dt} = \beta \cdot S \cdot I - \gamma \cdot I$$

$$\frac{dR}{dt} = \gamma \cdot I$$

- β: współczynnik transmisji (prawdopodobieństwo zarażenia),
- γ: współczynnik wyzdrowień (odwrotność czasu trwania choroby).

Dynamika:

1. Gdy S jest duże, liczba infekcji rośnie wykładniczo.
2. W miarę wzrostu I i R, liczba podatnych S maleje, co spowalnia dalsze rozprzestrzenianie.
3. Choroba wygasa, gdy liczba zakażonych spada do zera (I=0).

Zastosowanie:

- Modelowanie epidemii i pandemii.
- Prognozowanie efektów interwencji, takich jak szczepienia czy kwarantanna.

Model Lotki-Volterry (model drapieżnik-ofiara)

Opis:

Model Lotki-Volterry opisuje interakcję dwóch gatunków: drapieżnika i jego ofiary. Zakłada, że wzrost populacji jednego gatunku zależy od liczebności drugiego.

Równania modelu:

$$\frac{dN}{dt} = r \cdot N - a \cdot N \cdot P$$

$$\frac{dP}{dt} = b \cdot N \cdot P - d \cdot P$$

- N(t): liczba ofiar,
- P(t): liczba drapieżników,
- r: współczynnik wzrostu ofiar w przypadku braku drapieżników,
- a: współczynnik "ataków" drapieżnika,
- b: efektywność drapieżników w przekształcaniu zjedzonych ofiar na potomstwo,
- d: współczynnik śmiertelności drapieżników.

Dynamika:

1. Gdy liczba ofiar (N) rośnie, drapieżniki (P) mają więcej pożywienia, co zwiększa ich populację.
2. Wzrost liczby drapieżników zmniejsza liczbę ofiar, co prowadzi do spadku liczby drapieżników.
3. Cykl ten powtarza się, tworząc oscylacje populacji obu gatunków.

Zastosowanie:

- Ekologia: analiza interakcji między gatunkami.
- Zarządzanie populacjami w rolnictwie (np. szkodniki i ich naturalni wrogowie).

Model Verhulsta opisuje ograniczony wzrost populacji.
Model SIR analizuje dynamikę rozprzestrzeniania chorób zakaźnych.
Model Lotki-Volterry bada relacje drapieżnik-ofiara w ekosystemach.

1. Wymień etapy tworzenia modelu matematycznego.
2. Wymień i opisz rodzaje modelu matematycznego.
3. Omów równanie rozpadu promieniotwórczego i jego analogie.
4. Podaj równanie oscylatora harmonicznego i omów je.

5. Wymień i omów modele rozwoju populacji.
6. Omów układ RLC.

1. Wymień etapy tworzenia modelu matematycznego.
2. Wymień i opisz rodzaje modelu matematycznego

Definicja problemu (Sformułowanie problemu):

- Określenie celu modelowania, np. przewidywanie, optymalizacja, analiza zjawiska.
- Identyfikacja zmiennych wejściowych i wyjściowych oraz opisanie ich wzajemnych zależności.
- Sformułowanie założeń upraszczających, takich jak liniowość czy brak zakłóceń.

Zbieranie danych i założeń (Baza wiedzy):

- Analiza dostępnych danych empirycznych oraz przegląd literatury naukowej w danym obszarze.
- Upraszczanie rzeczywistości poprzez przyjęcie założeń (np. stałość parametrów, jednorodność układu).

Sformułowanie modelu matematycznego (Kategorie modelu):

- **Wybór typu modelu:** Określenie narzędzi matematycznych, takich jak równania różniczkowe, algebra liniowa, czy symulacje komputerowe.
- **Reprezentacja układu:** Przedstawienie systemu w postaci równań, funkcji, lub schematów.
- **Kategorie modeli matematycznych:**
    - **Dyskretne i ciągłe:**
        - *Dyskretne*: Zmienne przyjmują wartości całkowite (np. liczba osobników w populacji).
        - *Ciągłe*: Zmienne mogą przyjmować dowolne wartości w określonym zakresie (np. temperatura, koncentracja substancji).
    - **Deterministyczne i probabilistyczne:**
        - *Deterministyczne*: Wynik określony jednoznacznie na podstawie danych wejściowych (np. równania ruchu).
        - *Probabilistyczne*: Modele zawierające elementy losowości lub niepewności (np. modele populacji z fluktuacjami).
    - **Liniowe i nieliniowe:**
        - *Liniowe*: Relacje między zmiennymi są proporcjonalne i opisane równaniami liniowymi (np. obwody RC).
        - *Nieliniowe*: Relacje bardziej złożone, np. chaos deterministyczny.
    - **Statyczne i dynamiczne:**
        - *Statyczne*: Nie zależą od czasu, np. analiza obciążenia budynku.
        - *Dynamiczne*: Zmieniają się w czasie, np. układy równań różniczkowych opisujące dynamikę populacji.

Redukcja modelu:

- Upraszczanie modelu w celu ułatwienia obliczeń i interpretacji wyników.
- Walidacja modelu przez porównanie z danymi eksperymentalnymi lub rzeczywistością.

Wprowadzenie zmiennych:

- Identyfikacja i zdefiniowanie wszystkich zmiennych kluczowych dla modelu.

Napisać równanie modelu:

- Matematyczne przedstawienie związku między zmiennymi wejściowymi i wyjściowymi.

Próba rozwiązania:

- *Rozwiązanie analityczne*: Dokładne rozwiązania przy pomocy metod matematycznych.
- *Rozwiązanie numeryczne*: Przybliżone rozwiązania za pomocą algorytmów komputerowych.

Weryfikacja:

- Sprawdzenie poprawności modelu przez jego wykorzystanie w przewidywaniu, sterowaniu lub podejmowaniu decyzji.

Animacja i symulacja:

- Graficzna wizualizacja modelu w celu lepszego zrozumienia działania systemu i jego wyników.

3. Równanie Rozpadu Promieniotwórczego i Jego Analogie

Równanie:

$$\frac{dN}{dt} = -\lambda N$$

Gdzie:

- N: liczba jąder atomowych,
- λ: stała rozpadu promieniotwórczego.

Rozwiązanie:

$$N(t) = N_0 e^{-\lambda t}$$

Gdzie N_0 to początkowa liczba jąder atomowych.

Analogie:

- **Procesy chemiczne:** Reakcje pierwszego rzędu, gdzie tempo reakcji zależy od stężenia substratu.
- **Procesy elektryczne:** Rozładowanie kondensatora w obwodzie RC.
- **Zanik energii mechanicznej:** Tłumione drgania harmoniczne.

4. Równanie Oscylatora Harmonicznego i Omówienie

Równanie:

$$m\frac{d^2x}{dt^2} + b\frac{dx}{dt} + kx = 0$$

Gdzie:

- m: masa,
- b: współczynnik tłumienia,
- k: współczynnik sprężystości,
- x: wychylenie z położenia równowagi.

Omówienie:

- **Oscylator swobodny:** Dla b=0, układ wykonuje drgania harmoniczne: x(t)=Acos(ωt+φ) gdzie $\omega = \sqrt{\frac{k}{m}}$.
- **Oscylator tłumiony:** Dla b>0, drgania wygasają z czasem.
- **Oscylator wymuszony:** Układ reaguje na siłę zewnętrzną, co może prowadzić do rezonansu.

Zastosowania:

- Układy mechaniczne (np. zawieszenia samochodów).
- Układy elektryczne (np. obwody RLC).

5. Modele Rozwoju Populacji

Model Malthusa:

- Równanie: Dp/dt=rP gdzie r to tempo wzrostu.
- Rozwiązanie: $P(t)=P_0^{er}$

Model logistyczny:

- Uwzględnia ograniczenia środowiskowe: dP/dt=rP(1−PK) gdzie K to maksymalna pojemność środowiska.
- Rozwiązanie: $$P(t) = \frac{K}{1 + \left(\frac{K}{P_0} - 1\right) e^{-rt}}$$

Zastosowanie:

- Populacja ludzi, zwierząt.
- Dynamika rozwoju bakterii.

6. Układ RLC

Budowa:

- Układ składa się z rezystora (R), induktora (L) i kondensatora (C), połączonych szeregowo lub równolegle.

Równanie różniczkowe:

$$L\frac{d^2q}{dt^2} + R\frac{dq}{dt} + \frac{q}{C} = V(t)$$

Gdzie:

- q: ładunek,
- V(t): napięcie zewnętrzne.

Charakterystyka:

- **Drgania swobodne:** Dla V(t)=0 i R=0, układ wykonuje drgania harmoniczne.
- **Drgania tłumione:** Dla R>0, układ traci energię z czasem.

- **Rezonans:** Przy określonej częstotliwości wymuszania, amplituda napięcia lub prądu osiąga maksimum.

Zastosowanie:

- Filtry elektryczne.
- Obwody w radiotechnice.
- Stabilizatory napięcia.

Testowanie oprogramowania

1. Typy i poziomy testowania

Typy testowania

1. **Funkcjonalne:**
   - Sprawdza, czy system działa zgodnie z wymaganiami funkcjonalnymi.
   - Przykłady: testy jednostkowe, testy integracyjne, testy akceptacyjne.
   - Oparte na specyfikacji, np. wymaganiach biznesowych.
2. **Niefunkcjonalne:**
   - Sprawdza aspekty takie jak wydajność, niezawodność, użyteczność, kompatybilność.
   - Przykłady: testy wydajnościowe, testy obciążeniowe, testy bezpieczeństwa.
3. **Testy regresji:**
   - Upewniają się, że nowe zmiany w kodzie nie wprowadziły błędów w istniejącym systemie.
4. **Testy eksploracyjne:**
   - Wykonywane bez szczegółowego planu testów, pozwalając testerowi eksplorować system w poszukiwaniu błędów.
5. **Testy zgodności:**
   - Sprawdzają, czy system spełnia standardy i regulacje.

Poziomy testowania

1. **Testy jednostkowe:**
   - Testowanie pojedynczych modułów lub funkcji w izolacji.
   - Zazwyczaj realizowane przez programistów.
2. **Testy integracyjne:**
   - Sprawdzenie współpracy między modułami.
   - Typy integracji: top-down, bottom-up, sandwich.
3. **Testy systemowe:**
   - Weryfikacja całego systemu jako całości w rzeczywistym środowisku.
4. **Testy akceptacyjne:**
   - Weryfikacja, czy system spełnia wymagania użytkownika.
   - Wykonywane przez klientów lub użytkowników końcowych.

2. Metody testowania

1. **Testowanie czarnoskrzynkowe (Black-box testing):**

- Sprawdza funkcjonalność systemu bez wiedzy o wewnętrznej strukturze kodu.
- Skupia się na wejściach i wyjściach.
- Przykłady: testy funkcjonalne, testy regresji.

2. **Testowanie białoskrzynkowe (White-box testing):**
   - Analizuje wewnętrzną strukturę i implementację kodu.
   - Przykłady: pokrycie instrukcji, pokrycie ścieżek.
3. **Testowanie szarej skrzynki (Gray-box testing):**
   - Łączy podejście black-box i white-box.
   - Tester ma częściową wiedzę o wewnętrznej strukturze systemu.
4. **Testowanie ręczne:**
   - Wykonywane przez testerów bez użycia narzędzi automatyzujących.
   - Wymaga intuicji i doświadczenia testera.
5. **Testowanie automatyczne:**
   - Realizowane za pomocą narzędzi, które wykonują testy według zdefiniowanych skryptów.
   - Przykłady narzędzi: Selenium, JUnit, TestNG.

3. Projektowanie testów

1. **Analiza wymagań:**
   - Dokładne przestudiowanie dokumentacji wymagań w celu zrozumienia funkcjonalności do przetestowania.
2. **Tworzenie przypadków testowych:**
   - Definiowanie scenariuszy testowych, które pokrywają różne aspekty funkcjonalności.
   - Elementy przypadku testowego: cel testu, kroki testowe, oczekiwany wynik.
3. **Zarządzanie danymi testowymi:**
   - Przygotowanie zestawu danych wejściowych potrzebnych do przeprowadzenia testów.
4. **Tworzenie macierzy śledzenia (Traceability Matrix):**
   - Powiązanie wymagań z przypadkami testowymi w celu zapewnienia pełnego pokrycia.

4. Automatyzacja testowania

1. **Korzyści automatyzacji:**
   - Oszczędność czasu i kosztów przy dużej liczbie testów.
   - Większa dokładność i powtarzalność.
   - Ułatwienie testów regresji i wydajnościowych.
2. **Rodzaje testów automatycznych:**
   - Testy funkcjonalne: np. Selenium dla aplikacji webowych.
   - Testy jednostkowe: np. JUnit, NUnit.
   - Testy wydajnościowe: np. JMeter, LoadRunner.
3. **Proces automatyzacji:**
   - Wybór odpowiednich narzędzi.
   - Tworzenie skryptów testowych.
   - Utrzymanie i aktualizacja skryptów w miarę zmian systemu.

5. Zarządzanie testowaniem

1. **Planowanie testów:**

- Opracowanie strategii testowej, określenie zakresu testów, harmonogramu i zasobów.
- Dokument: Plan testów.

2. **Zarządzanie zespołem testowym:**
   - Podział ról i obowiązków.
   - Monitorowanie postępów testów.
3. **Monitorowanie i raportowanie:**
   - Śledzenie błędów i statusu testów.
   - Generowanie raportów z wynikami testów.
4. **Testowanie regresyjne:**
   - Planowanie i zarządzanie testami w odpowiedzi na zmiany w kodzie.

6. Dokumentacja testowa

1. **Plan testów:**
   - Dokument opisujący strategię, zakres, zasoby i harmonogram testów.
2. **Przypadki testowe:**
   - Szczegółowe opisy scenariuszy testowych, które mają być wykonane.
3. **Raporty z testów:**
   - Wyniki testów, napotkane błędy, wnioski i rekomendacje.
4. **Rejestr błędów (Bug report):**
   - Dokumentowanie wykrytych problemów, ich priorytetów i statusu.
5. **Macierz śledzenia wymagań (Traceability Matrix):**
   - Śledzenie, które przypadki testowe pokrywają konkretne wymagania.

7. Narzędzia i środowiska testowe

1. **Narzędzia do testów ręcznych:**
   - TestRail, Zephyr – do zarządzania przypadkami testowymi.
2. **Narzędzia do testów automatycznych:**
   - Selenium – automatyzacja testów przeglądarkowych.
   - JUnit/TestNG – testy jednostkowe.
3. **Narzędzia do testów wydajnościowych:**
   - JMeter – testy obciążeniowe aplikacji webowych.
   - LoadRunner – testy wydajnościowe systemów.
4. **Środowiska wirtualne:**
   - Docker – tworzenie izolowanych środowisk testowych.
   - VirtualBox – symulacja różnych platform.
5. **Narzędzia do raportowania błędów:**
   - Jira, Bugzilla – śledzenie i zarządzanie błędami.
6. **Narzędzia do CI/CD:**
   - Jenkins, GitLab CI – integracja automatyzacji testów z cyklem wydawniczym oprogramowania.

1. Wymień 7 Zasad Testowania Oprogramowania.
2. Opisz V-model wytwarzania oprogramowania.
3. Scharakteryzuj technikę TDD (wymień 3 charakterystyczne etapy tego procesu).
4. Wyjaśnij na czym polegają i do czego służą testy jednostkowe.
5. Wyjaśnij co to są testy regresyjne i do czego służą.

6. Opisz proces tworzenia testów aplikacji webowych (np. z wykorzystaniem frameworka Selenium w języku C# lub Java).

1. Wymień 7 Zasad Testowania Oprogramowania

Testowanie ujawnia błędy, ale nie może zagwarantować ich braku

Testowanie pozwala na wykrywanie defektów w oprogramowaniu, co umożliwia ich naprawę przed wdrożeniem produkcyjnym i zwiększa pewność, że system działa zgodnie z oczekiwaniami. Niemniej jednak, proces testowania nie jest w stanie udowodnić, że oprogramowanie jest całkowicie wolne od błędów. Choć testy pomagają wykryć wiele problemów, nie gwarantują, że wszystkie zostały znalezione, dlatego zawsze należy mieć świadomość, że oprogramowanie może zawierać nieodkryte usterki.

Pełne testowanie jest niemożliwe

Nie da się przetestować każdego możliwego scenariusza, w tym wszystkich kombinacji danych wejściowych i stanów wstępnych w systemie. Próba przetestowania wszystkich możliwych wariantów nie jest ani wykonalna, ani efektywna. W związku z tym, testy muszą być starannie zaplanowane, skupiając się na najbardziej krytycznych i istotnych częściach systemu, co pozwala na zachowanie wysokiej jakości testów, nawet bez konieczności sprawdzania każdego szczegółu kodu.

Wczesne testowanie oszczędza czas i pieniądze

Wczesne rozpoczęcie testowania w cyklu życia oprogramowania jest kluczowe. Testowanie założeń projektowych, dokumentacji i kodu na wczesnym etapie pozwala na szybsze wykrycie błędów, które można łatwiej i taniej naprawić. Ta zasada jest również zgodna z podejściem Agile, w którym testowanie jest integralną częścią całego procesu wytwarzania, a nie tylko końcową fazą.

Defekty się kumulują

W złożonych systemach często określone komponenty lub moduły zawierają największą liczbę defektów lub są odpowiedzialne za większość problemów. Dlatego testowanie powinno koncentrować się na tych obszarach, które są bardziej podatne na błędy. Wiedza o tym, które elementy systemu są szczególnie podatne na awarie, pozwala na bardziej efektywne testowanie i odkrywanie potencjalnych problemów.

Paradoks pestycydów

Jeśli przez cały czas wykonujemy te same testy, przestaną one wykrywać nowe problemy. Choć takie testy mogą potwierdzić, że system działa zgodnie z oczekiwaniami w danym momencie, nie znajdą nowych defektów. Należy regularnie aktualizować przypadki testowe, aby odkrywać nowe rodzaje błędów i poprawić jakość oprogramowania.

Testowanie zależy od kontekstu

Podejście do testowania oprogramowania zależy od kontekstu, w jakim system funkcjonuje. Różne rodzaje aplikacji – takie jak e-commerce, aplikacje mobilne czy interfejsy API – wymagają różnych metod testowania. Specyfika testów powinna być dostosowana do wymagań systemu oraz jego przeznaczenia, aby testy były adekwatne i skuteczne.

Przekonanie o braku błędów jest błędem

Brak błędów w oprogramowaniu nie gwarantuje jego użyteczności. Nawet jeśli system jest pozbawiony technicznych defektów, nie spełniając wymagań użytkowników, nie nadaje się do wdrożenia. Testowanie nie powinno ograniczać się jedynie do wykrywania błędów, ale także do zapewnienia, że oprogramowanie jest użyteczne i spełnia oczekiwania klientów. Warto przeprowadzać testy użyteczności, które pozwolą na zebranie cennych informacji zwrotnych i dostosowanie produktu do potrzeb użytkowników.

2. Opisz V-model wytwarzania oprogramowania

V-model to model wytwarzania oprogramowania, który ilustruje proces testowania w odniesieniu do etapów tworzenia oprogramowania. Jest to rozwinięcie klasycznego modelu kaskadowego, w którym etapy testowania są ściśle powiązane z odpowiednimi etapami rozwoju.

- **Etapy modelu V:**
    1. **Wymagania**: Określenie wymagań użytkownika.
    2. **Projektowanie systemu (High-level design)**: Określenie struktury systemu, podsystemów i interfejsów.
    3. **Projektowanie szczegółowe (Low-level design)**: Szczegółowy projekt komponentów i modułów.
    4. **Implementacja**: Kodowanie systemu.
- **Po prawej stronie "V" znajdują się testy:**
    1. **Testy akceptacyjne**: Testowanie zgodności z wymaganiami użytkownika.
    2. **Testy systemowe**: Testowanie pełnej funkcjonalności systemu.
    3. **Testy integracyjne**: Testowanie interakcji między modułami.
    4. **Testy jednostkowe**: Testowanie poszczególnych komponentów aplikacji.

W modelu V każde testowanie odpowiada etapowi projektowania. Model ten podkreśla, że testowanie powinno być planowane i przeprowadzane równolegle z rozwojem oprogramowania.

3. Scharakteryzuj technikę TDD (Test Driven Development) - 3 charakterystyczne etapy tego procesu

TDD (Test Driven Development) to technika rozwoju oprogramowania, w której testy są pisane przed kodem. Celem jest zapewnienie wysokiej jakości kodu i ciągłej weryfikacji, że kod spełnia wymagania.

3 główne etapy TDD:

1. **Pisanie testu (Red):**
    - Pierwszym krokiem jest napisanie testu, który ma na celu przetestowanie nowej funkcjonalności. Test jest pisany, zanim powstanie jakikolwiek kod. Zwykle na tym etapie test kończy się błędem (czerwony stan), ponieważ funkcjonalność nie została jeszcze zaimplementowana.
2. **Pisanie minimalnego kodu (Green):**
    - Następnie piszemy minimalny kod, który pozwala testowi przejść (kolor zielony). Celem jest jak najszybsze spełnienie wymagań testu przy minimalnej ilości kodu.

3. **Refaktoryzacja (Refactor):**
    - Po uzyskaniu pozytywnego wyniku testu (zielony), kod jest refaktoryzowany w celu poprawy jakości, struktury i wydajności, bez zmiany jego funkcjonalności. Testy nadal są uruchamiane, aby upewnić się, że kod działa zgodnie z oczekiwaniami.

TDD pomaga utrzymać kod czysty, modularny i łatwy do zrozumienia, ponieważ każda zmiana jest dokładnie przetestowana.

4. Wyjaśnij na czym polegają i do czego służą testy jednostkowe

Testy jednostkowe to rodzaj testów, które sprawdzają poprawność pojedynczych funkcji lub metod w obrębie systemu. Testy te są zazwyczaj wykonywane przez programistów i mają na celu zapewnienie, że poszczególne części kodu (tzw. jednostki) działają zgodnie z wymaganiami.

- **Cel testów jednostkowych:**
    - Weryfikacja, czy pojedyncze funkcje lub metody działają poprawnie w izolacji.
    - Szybkie wykrywanie błędów w początkowych fazach programowania.
    - Ułatwienie refaktoryzacji kodu poprzez zapewnienie, że wprowadzone zmiany nie wprowadzają nowych błędów.
- **Przykład:** Jeśli mamy funkcję `add(a, b)`, test jednostkowy mógłby sprawdzić, czy `add(2, 3)` rzeczywiście zwraca wartość 5.

5. Wyjaśnij co to są testy regresyjne i do czego służą

Testy regresyjne to testy, które sprawdzają, czy nowo wprowadzone zmiany w systemie (np. nowe funkcje, poprawki błędów) nie spowodowały uszkodzenia istniejącej funkcjonalności. Celem jest upewnienie się, że poprzednie funkcje aplikacji nadal działają prawidłowo po wprowadzeniu zmian.

- **Służą do:**
    - Zapewnienia, że poprawki lub nowe funkcje nie wprowadziły błędów w innych częściach systemu.
    - Weryfikacji, że system działa tak samo jak przed zmianami.
- **Przykład:** Po dodaniu nowej funkcji logowania w aplikacji, testy regresyjne mogłyby sprawdzić, czy już istniejąca funkcjonalność użytkowników (np. rejestracja) nadal działa poprawnie.

6. Opisz proces tworzenia testów aplikacji webowych (np. z wykorzystaniem frameworka Selenium w języku C# lub Java)

Selenium to popularne narzędzie do automatyzacji testów aplikacji webowych. Umożliwia symulowanie działań użytkownika, takich jak kliknięcia, wprowadzanie tekstu, nawigacja, a następnie weryfikowanie wyników.

Proces tworzenia testów:

1. **Instalacja Selenium:**

    - Zainstalowanie odpowiedniego frameworka w zależności od używanego języka programowania, np. Selenium WebDriver dla C# lub Java.
2. **Tworzenie klasy testowej:**
    - Należy stworzyć klasę, która będzie zawierała metody testowe. Każda metoda testowa to jeden scenariusz, który ma na celu przetestowanie konkretnej funkcji aplikacji.
3. **Konfiguracja środowiska testowego:**
    - Zdefiniowanie lokalizacji aplikacji (URL), którą będziemy testować. Konfiguracja środowiska może obejmować również konfigurację przeglądarki (np. ChromeDriver dla Google Chrome).
4. **Pisanie testów:**
    - Pisanie scenariuszy testowych, które mogą obejmować:
        - Nawigację po stronach: `driver.get("http://example.com")`
        - Interakcje z elementami na stronie (np. kliknięcie przycisku): `driver.findElement(By.id("loginButton")).click()`
        - Wprowadzanie danych do formularza: `driver.findElement(By.id("username")).sendKeys("user")`
5. **Assercje:**
    - Sprawdzanie, czy wyniki testu są zgodne z oczekiwaniami. Na przykład: `assertTrue(driver.findElement(By.id("welcomeMessage")).isDisplayed())`.
6. **Uruchomienie testów i raportowanie:**
    - Uruchamianie testów i generowanie raportów o wynikach testów, np. przy użyciu JUnit w Javie lub NUnit w C#.
7. **Utrzymanie testów:**
    - Regularne aktualizowanie skryptów testowych w miarę rozwoju aplikacji webowej, aby testy pozostawały aktualne i skuteczne.

Wzorce projektowe

1. Ogólne określenie wzorca wg Christophera Alexandra

Christopher Alexander to jeden z pionierów w dziedzinie architektury, który zapoczątkował ideę **wzorców projektowych** w kontekście urbanistyki i projektowania przestrzennego. Jego podejście do wzorców projektowych zostało szeroko zaadoptowane przez inne dziedziny, w tym projektowanie oprogramowania.

Wzorzec wg Alexandra to rozwiązanie powtarzającego się problemu projektowego w określonym kontekście. Alexander w swoich pracach, zwłaszcza w książce *"A Pattern Language: Towns, Buildings, Construction"* (1977), definiuje wzorzec jako spójne, uniwersalne rozwiązanie, które jest wykorzystywane w określonych okolicznościach. Każdy wzorzec składa się z:

- **Nazwy** – identyfikuje wzorzec i pozwala na odnalezienie go w literaturze.
- **Problemu** – wyjaśnia trudności, które rozwiązuje dany wzorzec.
- **Rozwiązania** – proponuje ogólne rozwiązanie, które pomaga w pokonaniu tego problemu.
- **Kontekstu** – opisuje, w jakich okolicznościach stosowanie wzorca jest wskazane.

Wzorce są używane do rozwiązywania problemów projektowych, które występują wielokrotnie, jednak wymagają indywidualnego podejścia w każdym przypadku. Wzorce umożliwiają tworzenie bardziej spójnych, efektywnych i przemyślanych rozwiązań, które łatwiej dostosować do różnorodnych potrzeb.

2. Ogólne cechy wzorca projektowego

Wzorzec projektowy to sprawdzone, powtarzalne rozwiązanie dla powszechnych problemów projektowych. Oto kluczowe cechy wzorca projektowego:

- **Reużywalność** – wzorzec jest zaprojektowany tak, aby można go było wielokrotnie stosować w różnych kontekstach i projektach.
- **Abstrakcja** – wzorzec nie stanowi dokładnego, szczegółowego rozwiązania, lecz ogólną koncepcję, którą można dostosować do specyficznych potrzeb.
- **Dokumentacja** – każdy wzorzec posiada dobrze zdefiniowaną dokumentację, która opisuje problem, rozwiązanie, kontekst oraz wskazówki implementacyjne.
- **Elastyczność** – wzorzec jest elastyczny i może być dostosowany do różnych warunków oraz potrzeb użytkownika.
- **Spójność** – wzorzec powinien być konsekwentny, dostarczać rozwiązanie, które jest spójne w różnych zastosowaniach.
- **Modularność** – wzorce projektowe są zazwyczaj podzielone na mniejsze, łatwiejsze do zarządzania komponenty, które mogą być stosowane niezależnie lub w połączeniu.

3. Wybrane wzorce projektowe

Wzorce projektowe zostały sklasyfikowane na trzy główne grupy: **behawioralne**, **kreacyjne** i **strukturalne**. Każda z tych kategorii odpowiada na inne potrzeby w procesie projektowania i implementacji systemów. Poniżej przedstawiono kilka wybranych wzorców z każdej kategorii, wraz z ich zastosowaniem.

a. Wzorce kreacyjne

Wzorce kreacyjne (konstrukcyjne) to kategoria wzorców projektowych, które koncentrują się na sposobach tworzenia obiektów w programowaniu. Głównym celem tych wzorców jest oddzielenie procesu tworzenia obiektów od ich używania, co zwiększa elastyczność i skalowalność kodu.

1. Wzorzec Budowniczy (Builder)

- **Aspekt, który możemy zmieniać**: Sposób tworzenia obiektu złożonego.

Opis: Wzorzec Budowniczy jest stosowany w sytuacji, gdy obiekt, który tworzymy, jest złożony i składa się z wielu komponentów. Umożliwia on oddzielenie procesu konstrukcji od reprezentacji obiektu. Dzięki temu proces tworzenia złożonego obiektu staje się bardziej elastyczny, co pozwala na tworzenie różnych reprezentacji tego obiektu.

- **Przykład użycia**: Budowanie skomplikowanych obiektów takich jak dokumenty (np. raporty, pliki XML), konfiguracje aplikacji lub systemów operacyjnych.
- **Zaleta**: Pozwala na stopniowe konstruowanie obiektu, a nie tworzenie go za jednym razem. Może być używany, gdy obiekt ma wiele wariantów lub ma różne zestawy ustawień.

Elementy wzorca:

- **Builder**: Interfejs do konstrukcji obiektów.
- **ConcreteBuilder**: Implementacja interfejsu budowniczego, tworząca obiekt.
- **Director**: Obiekt, który kieruje procesem tworzenia obiektu.

2. Wzorzec Fabryka Abstrakcyjna (Abstract Factory)

- **Aspekt, który możemy zmieniać**: Rodziny obiektów-produktów.

Opis: Wzorzec Fabryka Abstrakcyjna dostarcza interfejs do tworzenia rodziny powiązanych ze sobą obiektów bez wskazywania ich konkretnych klas. Fabryka abstrakcyjna tworzy obiekty, które są zgodne z jakąś rodziną (np. zestaw produktów), ale nie wiemy dokładnie, który obiekt zostanie utworzony, ponieważ zależy to od implementacji fabryki.

- **Przykład użycia**: Tworzenie interfejsów użytkownika w różnych systemach operacyjnych (np. Windows, macOS, Linux), gdzie dla każdego systemu potrzebujemy różnych klas, ale fabryka zajmuje się tworzeniem odpowiednich komponentów UI (przyciski, okna itp.).
- **Zaleta**: Umożliwia łatwą wymianę całych rodzin obiektów w zależności od konfiguracji systemu.

Elementy wzorca:

- **AbstractFactory**: Interfejs lub klasa abstrakcyjna tworząca produkty.
- **ConcreteFactory**: Klasy implementujące metody tworzenia produktów.
- **AbstractProduct**: Interfejs lub klasa abstrakcyjna dla produktów.
- **ConcreteProduct**: Konkretny typ produktu.

3. Wzorzec Metoda Wytwórcza (Factory Method)

- **Aspekt, który możemy zmieniać**: Podklasa tworzonego obiektu.

Opis: Wzorzec Metoda Wytwórcza pozwala na tworzenie obiektów bez wskazywania dokładnej klasy tworzonych obiektów. Zamiast tworzyć obiekty bezpośrednio, wykorzystujemy metodę, która jest nadpisywana w klasach dziedziczących, umożliwiając tworzenie różnych typów obiektów w zależności od potrzeb.

- **Przykład użycia**: Tworzenie różnych typów dokumentów (np. PDF, Word, Excel) w aplikacji biurowej, gdzie każda z klas dokumentów ma swoją metodę wytwórczą.
- **Zaleta**: Pozwala na tworzenie obiektów w zależności od potrzeb, unikając bezpośredniego wywołania konstruktora obiektu w kodzie.

Elementy wzorca:

- **Product**: Abstrakcyjna klasa lub interfejs dla tworzonego obiektu.
- **ConcreteProduct**: Konkretna implementacja produktu.
- **Creator**: Klasa lub interfejs, który definiuje metodę tworzenia produktów.
- **ConcreteCreator**: Klasa dziedzicząca, która implementuje metodę tworzenia obiektu.

4. Wzorzec Prototyp (Prototype)

- **Aspekt, który możemy zmieniać**: Klasa tworzonego obiektu.

Opis: Wzorzec Prototyp polega na tworzeniu nowych obiektów przez klonowanie istniejących obiektów. Zamiast tworzyć nowy obiekt od podstaw, wzorzec ten wykorzystuje obiekt prototypowy, który jest kopiowany, a następnie dostosowywany do potrzeb.

- **Przykład użycia**: Gra komputerowa, gdzie wiele obiektów (np. przeciwników, broni, samochodów) ma podobną strukturę i różnią się tylko kilkoma właściwościami. Zamiast tworzyć każdy obiekt od podstaw, możemy je klonować.
- **Zaleta**: Oszczędza zasoby, ponieważ unika konieczności ponownego tworzenia obiektów i ich inicjalizacji.

Elementy wzorca:

- **Prototype**: Abstrakcyjny interfejs, który deklaruje metodę klonowania obiektów.
- **ConcretePrototype**: Klasa, która implementuje metodę klonowania.

5. Wzorzec Singleton

- **Aspekt, który możemy zmieniać**: Jedyny egzemplarz klasy.

Opis: Wzorzec Singleton zapewnia, że dany obiekt jest instancją tylko raz w aplikacji. Zatem klasa ma tylko jedną instancję, która jest dostępna dla wszystkich wątków lub procesów. Wzorzec ten gwarantuje, że nie ma wielu obiektów tej samej klasy, co jest przydatne w przypadku zasobów, które muszą być unikalne (np. baza danych, połączenia sieciowe).

- **Przykład użycia**: Baza danych, gdzie aplikacja potrzebuje tylko jednej instancji połączenia z bazą, aby uniknąć niepotrzebnych zasobów i błędów.
- **Zaleta**: Oszczędność zasobów, unikanie wielokrotnego tworzenia obiektów, a także łatwiejsze zarządzanie globalnym stanem aplikacji.

Elementy wzorca:

- **Singleton**: Klasa, która zapewnia, że istnieje tylko jedna instancja, oraz dostęp do tej instancji.
- **Private Constructor**: Konstruktor klasy jest prywatny, aby zapobiec tworzeniu nowych instancji.
- **Static Method**: Metoda, która zwraca jedyną instancję klasy (lub ją tworzy, jeśli nie istnieje).

Wzorce behawioralne (operacyjne) to wzorce projektowe, które koncentrują się na sposobie, w jaki obiekty współdziałają ze sobą i jakie mają między sobą interakcje. Wzorce te pomagają w organizacji zachowań obiektów i komunikacji pomiędzy nimi, a także umożliwiają elastyczne dostosowanie algorytmów oraz interakcji w aplikacji. Oto szczegółowy opis najpopularniejszych wzorców behawioralnych:

1. Wzorzec Interpreter (Interpreter)

- **Aspekt, który możemy zmieniać**: Gramatyka i interpretacja języka.

Opis: Wzorzec Interpreter umożliwia interpretację wyrażeń w języku formalnym, takim jak języki zapytań, matematyczne wyrażenia lub jakikolwiek inny zdefiniowany język. Gramatyka tego języka jest reprezentowana przez drzewa składniowe, a wzorzec zapewnia mechanizm interpretacji lub przetwarzania tych wyrażeń.

- **Przykład użycia**: Kompilatory, interpretery języków programowania, systemy analizy zapytań w bazach danych.
- **Zaleta**: Umożliwia łatwe rozszerzanie gramatyki lub dodawanie nowych reguł języka.

Elementy wzorca:

- **AbstractExpression**: Interfejs definiujący operację interpretacyjną.
- **TerminalExpression**: Konkretny wyraz w języku.
- **NonTerminalExpression**: Wyraz w języku, który jest złożony z innych wyrazów.

2. Wzorzec Iterator (Iterator)

- **Aspekt, który możemy zmieniać**: Sposób dostępu do elementu i przechodzenia po nich.

Opis: Wzorzec Iterator zapewnia sposób sekwencyjnego dostępu do elementów kolekcji bez ujawniania szczegółów implementacji tej kolekcji. Umożliwia to łatwe przechodzenie po zbiorach danych (np. listach, tablicach) bez konieczności znajomości struktury danych.

- **Przykład użycia**: Przechodzenie po elementach kolekcji w językach programowania (np. pętle foreach, for).
- **Zaleta**: Oddziela logikę przechodzenia przez kolekcję od samej kolekcji, co sprawia, że kod jest bardziej elastyczny i łatwiejszy do zmodyfikowania.

Elementy wzorca:

- **Iterator**: Interfejs umożliwiający przechodzenie po kolekcji.
- **ConcreteIterator**: Implementacja interfejsu Iterator, przechodzi przez elementy kolekcji.
- **Aggregate**: Interfejs kolekcji, który dostarcza iterator.
- **ConcreteAggregate**: Implementacja kolekcji.

3. Wzorzec Łańcuch Zobowiązań (Chain of Responsibility)

- **Aspekt, który możemy zmieniać**: Obiekt potrafiący obsłużyć żądanie.

Opis: Wzorzec Łańcuch Zobowiązań pozwala na tworzenie łańcucha obiektów, z których każdy może obsłużyć dane żądanie. Żądanie jest przekazywane od jednego obiektu do drugiego w łańcuchu, aż któryś obiekt zdecyduje się je obsłużyć.

- **Przykład użycia**: Systemy obsługi żądań, jak np. obsługa błędów w aplikacjach, gdzie każdy komponent może odpowiedzieć na specyficzny typ błędu.
- **Zaleta**: Elastyczność w rozwiązywaniu problemów, ponieważ pozwala na dynamiczną zmianę łańcucha obiektów obsługujących żądania.

Elementy wzorca:

- **Handler**: Abstrakcyjna klasa, która definiuje metodę obsługi żądania.
- **ConcreteHandler**: Konkretny obiekt, który może obsłużyć żądanie.
- **Client**: Obiekt, który inicjuje żądanie.

4. Wzorzec Mediator (Mediator)

- **Aspekt, który możemy zmieniać**: Sposób, w jaki dane obiekty wchodzą ze sobą w interakcje.

Opis: Wzorzec Mediator centralizuje komunikację pomiędzy obiektami, eliminując potrzebę bezpośrednich interakcji między nimi. Mediator przyjmuje rolę pośrednika, który kontroluje, jak obiekty się komunikują.

- **Przykład użycia**: Wzorzec ten jest powszechnie stosowany w GUI, gdzie różne komponenty (np. przyciski, pola tekstowe) muszą współpracować, ale nie powinny bezpośrednio się komunikować.
- **Zaleta**: Umożliwia kontrolowanie przepływu komunikacji między obiektami i zmniejsza ich zależność od siebie nawzajem.

Elementy wzorca:

- **Mediator**: Interfejs mediatora, który definiuje sposób komunikacji.
- **ConcreteMediator**: Konkretna implementacja mediatora.
- **Colleague**: Obiekty, które komunikują się za pomocą mediatora.

5. Wzorzec Metoda Szablonowa (Template Method)

- **Aspekt, który możemy zmieniać**: Kroki algorytmu.

Opis: Wzorzec Metoda Szablonowa definiuje szablon algorytmu, w którym niektóre kroki mogą być zmieniane przez klasy dziedziczące. Część algorytmu jest ustalana w klasie bazowej, a konkretne kroki mogą być nadpisywane przez klasy pochodne.

- **Przykład użycia**: Algorytmy wstępnej konfiguracji, gdzie ogólny proces jest ustalony, ale szczegóły (np. sposób konfiguracji) mogą być dostosowywane przez klasy dziedziczące.
- **Zaleta**: Pozwala na ponowne wykorzystanie kodu i wymusza jednolitą strukturę algorytmów.

Elementy wzorca:

- **AbstractClass**: Klasa bazowa, która definiuje metodę szablonową.
- **ConcreteClass**: Klasa dziedzicząca, która nadpisuje kroki algorytmu.

6. Wzorzec Obserwator (Observer)

- **Aspekt, który możemy zmieniać**: Liczba obiektów zależnych od innego obiektu; sposób aktualizacji obiektów zależnych.

Opis: Wzorzec Obserwator definiuje zależność "jeden do wielu", gdzie zmiana stanu obiektu (podmiotu) powoduje automatyczną aktualizację wszystkich zależnych obiektów (obserwatorów). Jest użyteczny w aplikacjach, gdzie stan obiektu wpływa na wiele innych obiektów.

- **Przykład użycia**: Systemy powiadomień (np. aplikacje emailowe lub społecznościowe), GUI (np. widżety reagujące na zmiany w modelu danych).
- **Zaleta**: Umożliwia łatwe śledzenie i aktualizowanie zależnych obiektów.

Elementy wzorca:

- **Subject**: Obiekt, którego stan jest monitorowany.
- **Observer**: Obiekt, który nasłuchuje zmian w obiekcie Subject.
- **ConcreteObserver**: Implementacja obserwatora, który reaguje na zmiany w Subject.

7. Wzorzec Odwiedzający (Visitor)

- **Aspekt, który możemy zmieniać**: Operacje, które można zastosować do obiektów bez zmiany ich klas.

Opis: Wzorzec Odwiedzający umożliwia dodanie nowych operacji do obiektów bez zmiany ich klas. Operacje są przechowywane w obiektach odwiedzających, które mogą wykonywać różne operacje na elementach kolekcji.

- **Przykład użycia**: Przetwarzanie różnych typów elementów w strukturach danych, takich jak drzewa lub listy (np. obliczenia na drzewach wyrażeń matematycznych).
- **Zaleta**: Rozdziela logikę operacji od struktury obiektów.

Elementy wzorca:

- **Visitor**: Interfejs odwiedzającego, który definiuje operacje na elementach.
- **ConcreteVisitor**: Konkretny odwiedzający, który implementuje operacje.

- **Element**: Obiekt, który akceptuje odwiedzającego.
- **ConcreteElement**: Konkretna implementacja elementu.

8. Wzorzec Pamiątka (Memento)

- **Aspekt, który możemy zmieniać**: Które informacje prywatne są przechowywane poza obiektem i kiedy.

Opis: Wzorzec Pamiątka umożliwia zapisanie stanu obiektu w taki sposób, by mógł on zostać przywrócony w późniejszym czasie. Jest używany do implementacji funkcji "cofnij", gdzie stan obiektu może być przechowywany i przywracany.

- **Przykład użycia**: Systemy do cofaniących się edycji tekstów, gier komputerowych, które pozwalają na cofnąć zmiany w grze.
- **Zaleta**: Umożliwia przechowywanie stanu obiektów w sposób nienaruszający zasad enkapsulacji.

Elementy wzorca:

- **Memento**: Obiekt przechowujący stan obiektu.
- **Originator**: Obiekt, którego stan jest zapisywany.
- **Caretaker**: Obiekt odpowiedzialny za przechowywanie i przywracanie stanów.

9. Wzorzec Polecenie (Command)

- **Aspekt, który możemy zmieniać**: Warunki i sposób obsługi żądania.

Opis: Wzorzec Polecenie zamienia żądanie na obiekt, co pozwala na oddzielenie wywołania od realizacji żądania. Żądanie może być przekazywane i obsługiwane w różny sposób, co pozwala na łatwe dodawanie nowych operacji.

- **Przykład użycia**: Obsługa przycisków w GUI, gdzie kliknięcie na przycisk generuje polecenie, które jest realizowane przez odpowiednią metodę.
- **Zaleta**: Eliminuje bezpośrednią zależność pomiędzy obiektem, który wywołuje operację, a obiektem, który ją wykonuje.

Elementy wzorca:

- **Command**: Interfejs polecenia, który definiuje metodę wykonania.
- **ConcreteCommand**: Implementacja polecenia.
- **Invoker**: Obiekt, który przechowuje polecenia.
- **Receiver**: Obiekt, który wykonuje faktyczną operację.

10. Wzorzec Stan (State)

- **Aspekt, który możemy zmieniać**: Stany obiektu.

Opis: Wzorzec Stan pozwala na zmianę zachowań obiektu w zależności od jego stanu. Zamiast mieć skomplikowane instrukcje warunkowe w metodach, obiekt zmienia swoje zachowanie, przechodząc pomiędzy różnymi stanami.

- **Przykład użycia**: Maszyna do sprzedaży napojów, gdzie zachowanie zmienia się w zależności od tego, czy maszyna jest w stanie "gotowości", "oczekiwania na monetę" czy "wydawania napoju".
- **Zaleta**: Ułatwia zarządzanie stanem obiektu, zmniejszając złożoność kodu.

Elementy wzorca:

- **Context**: Obiekt, którego stan się zmienia.
- **State**: Interfejs dla stanów.
- **ConcreteState**: Konkretne stany, które realizują różne zachowania obiektu.

11. Wzorzec Strategia (Strategy)

- **Aspekt, który możemy zmieniać**: Algorytm.

Opis: Wzorzec Strategia pozwala na wybór algorytmu w czasie działania programu. Algorytmy są zdefiniowane w różnych klasach, a klasa kontekstowa wybiera odpowiednią strategię w zależności od warunków.

- **Przykład użycia**: Algorytmy sortowania, gdzie wybór algorytmu (np. QuickSort, MergeSort, BubbleSort) jest uzależniony od wielkości danych.
- **Zaleta**: Umożliwia wymianę algorytmów w czasie działania programu bez potrzeby zmiany kodu klasu korzystającej z algorytmu.

Elementy wzorca:

- **Context**: Klasa, która korzysta ze strategii.
- **Strategy**: Interfejs definiujący algorytmy.
- **ConcreteStrategy**: Implementacje różnych algorytmów.

Wzorce strukturalne koncentrują się na organizacji i organizowaniu klas oraz obiektów, aby uzyskać elastyczną i efektywną strukturę w systemie. Zajmują się one tworzeniem złożonych struktur z prostszych obiektów, umożliwiając łatwiejsze zarządzanie, modyfikowanie i rozszerzanie systemu. Oto szczegółowy opis popularnych wzorców strukturalnych:

1. Wzorzec Adapter (Adapter)

- **Aspekt, który możemy zmieniać**: Interfejs obiektu.

Opis: Wzorzec Adapter pozwala na dostosowanie interfejsu jednego obiektu do interfejsu oczekiwanego przez inny obiekt. Działa jak tłumacz, przekształcając jedną formę komunikacji na inną, umożliwiając współpracę obiektów o niekompatybilnych interfejsach.

- **Przykład użycia**: Jeśli mamy bibliotekę z funkcjami oczekującymi innego formatu danych, możemy stworzyć adapter, który zamienia dane w odpowiedni format.
- **Zaleta**: Umożliwia łatwą integrację różnych systemów z różnymi interfejsami bez konieczności ich modyfikowania.

Elementy wzorca:

- **Client**: Obiekt korzystający z obiektów o różnych interfejsach.
- **Target**: Interfejs, do którego muszą być dostosowane obiekty.
- **Adapter**: Obiekt, który przekształca jeden interfejs w inny.
- **Adaptee**: Obiekt, który ma interfejs niekompatybilny z wymaganym.

2. Wzorzec Dekorator (Decorator)

- **Aspekt, który możemy zmieniać**: Zadania obiektu (bez tworzenia podklas).

Opis: Wzorzec Dekorator umożliwia dodawanie nowych funkcjonalności do obiektów w sposób dynamiczny, bez konieczności modyfikowania ich klas. Zamiast tworzyć podklasy, nową funkcjonalność dodaje się poprzez opakowanie obiektu w dekoratory, które implementują dodatkowe zadania.

- **Przykład użycia**: Można użyć dekoratora do dodania funkcjonalności logowania lub walidacji do istniejącego obiektu bez zmiany jego kodu źródłowego.
- **Zaleta**: Umożliwia elastyczne rozszerzanie funkcji obiektów w czasie działania programu bez ingerencji w kod obiektów.

Elementy wzorca:

- **Component**: Interfejs definiujący funkcjonalność obiektu.
- **ConcreteComponent**: Klasa implementująca Component.
- **Decorator**: Klasa, która opakowuje obiekt i rozszerza jego funkcjonalności.
- **ConcreteDecorator**: Konkretne dekoratory, które rozszerzają funkcjonalności obiektów.

3. Wzorzec Fasada (Facade)

- **Aspekt, który możemy zmieniać**: Interfejs podsystemu.

Opis: Wzorzec Fasada upraszcza interakcje z bardziej złożonymi systemami, tworząc prostszy, jednorodny interfejs do komunikacji z nimi. Fasada ukrywa złożoność systemu i zapewnia łatwiejszy dostęp do jego funkcji.

- **Przykład użycia**: W systemie z wieloma modułami, jak np. aplikacja bankowa, można stworzyć fasadę, która ukryje skomplikowane operacje systemowe za prostym interfejsem.
- **Zaleta**: Ułatwia korzystanie z złożonych systemów, jednocześnie redukując ich skomplikowanie.

Elementy wzorca:

- **Facade**: Obiekt, który oferuje uproszczony interfejs.
- **Subsystem**: Systemy, które fasada upraszcza.

4. Wzorzec Kompozyt (Composite)

- **Aspekt, który możemy zmieniać**: Struktura i skład obiektu.

Opis: Wzorzec Kompozyt pozwala na traktowanie pojedynczych obiektów i ich zbiorów (drzew) w sposób jednolity. Umożliwia to tworzenie struktur hierarchicznych, gdzie obiekty mogą zawierać inne obiekty, a interakcja z nimi jest jednolita.

- **Przykład użycia**: W aplikacjach graficznych, w których różne elementy (np. prostokąty, okręgi, linie) mogą być częścią złożonego obiektu, takiego jak scena lub rysunek.
- **Zaleta**: Umożliwia łatwe zarządzanie złożonymi strukturami obiektów, traktując zarówno pojedyncze elementy, jak i ich zbiory w taki sam sposób.

Elementy wzorca:

- **Component**: Interfejs dla pojedynczych obiektów i ich zbiorów.
- **Leaf**: Pojedynczy obiekt w strukturze.
- **Composite**: Obiekt, który zawiera inne obiekty (może być samodzielny lub zawierać kolejne Composite).

5. Wzorzec Most (Bridge)

- **Aspekt, który możemy zmieniać**: Implementacja obiektu.

Opis: Wzorzec Most pozwala na oddzielenie abstrakcji od jej implementacji, umożliwiając niezależne zmiany w obu tych aspektach. Umożliwia to rozdzielenie warstwy abstrakcyjnej i warstwy implementacyjnej, co pozwala na większą elastyczność w zmianach.

- **Przykład użycia**: Systemy graficzne, w których różne platformy mogą wykorzystywać tę samą abstrakcję (np. rysowanie prostokąta), ale różne implementacje mogą być zależne od platformy (np. Windows, macOS).
- **Zaleta**: Umożliwia niezależny rozwój różnych implementacji i ich późniejsze łączenie z abstrakcjami.

Elementy wzorca:

- **Abstraction**: Abstrakcyjna warstwa, która używa implementacji.
- **RefinedAbstraction**: Konkretna wersja Abstraction.
- **Implementor**: Interfejs, który definiuje metody implementacji.
- **ConcreteImplementor**: Konkretna implementacja interfejsu.

6. Wzorzec Pełnomocnik (Proxy)

- **Aspekt, który możemy zmieniać**: Sposób dostępu do obiektu i jego lokalizacja.

Opis: Wzorzec Pełnomocnik (Proxy) tworzy pośrednika, który kontroluje dostęp do innego obiektu. Może służyć jako przedstawiciel obiektu, który może być kosztowny w utworzeniu lub przechowywaniu, a pełnomocnik decyduje, kiedy ten obiekt jest rzeczywiście wykorzystywany.

- **Przykład użycia**: Pełnomocnik może być używany do zarządzania dostępem do obiektów w systemie (np. zabezpieczenia lub kontrola dostępu).
- **Zaleta**: Umożliwia leniwe ładowanie lub kontrolowanie dostępu do zasobów.

Elementy wzorca:

- **Subject**: Interfejs, który reprezentuje obiekt, do którego dostęp jest kontrolowany.
- **RealSubject**: Obiekt, który jest rzeczywiście wykorzystywany.
- **Proxy**: Obiekt pełnomocnika, który zarządza dostępem do RealSubject.

7. Wzorzec Pyłek (Flyweight)

- **Aspekt, który możemy zmieniać**: Koszty przechowywania obiektów.

Opis: Wzorzec Pyłek służy do efektywnego zarządzania dużą liczbą obiektów, poprzez współdzielenie tych, które mają wspólne dane (np. identyczne właściwości). Zamiast tworzyć wiele kopii obiektów, wzorzec pozwala na ich współdzielenie, zmniejszając zużycie pamięci.

- **Przykład użycia**: Przechowywanie dużej liczby podobnych obiektów, jak np. litery w edytorze tekstu, które mogą być przechowywane jako jedno "wspólne" obiekt.
- **Zaleta**: Oszczędność pamięci i zwiększenie wydajności przy dużej liczbie podobnych obiektów.

Elementy wzorca:

- **Flyweight**: Interfejs, który określa metody, które powinny być współdzielone przez obiekty.
- **ConcreteFlyweight**: Konkretne implementacje, które przechowują współdzielone dane.
- **FlyweightFactory**: Fabryka, która tworzy i zarządza instancjami Flyweight.

1. Klasyfikacja wzorców projektowych. Krótkie omówienie przykładowego wzorca z każdej grupy.
2. Wzorzec projektowy Strategia (Strategy) – omówienie (klasyfikacja, intencja/przeznaczenie, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).
3. Wzorzec projektowy Most (Bridge) – omówienie (klasyfikacja, intencja/przeznaczenie, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).
4. Wzorzec projektowy Metoda Szablonowa (Template Method) – omówienie (klasyfikacja, intencja/przeznaczenie, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).
5. Wzorzec projektowy Dekorator (Decorator) – omówienie (klasyfikacja, intencja/przeznaczenie, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).

6. Wzorzec projektowy Kompozyt (Composite) – omówienie (klasyfikacja, intencja/przeznaczenie, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).
7. Wzorzec projektowy Fabryka Abstrakcyjna (Abstract Factory) – omówienie (klasyfikacja, intencja/przeznaczenie, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).
8. Wzorzec projektowy Obserwator (Observer) – omówienie (klasyfikacja, intencja/przeznaczenie, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).
9. Wzorzec projektowy Model-Widok-Kontroler (Model-View-Controller) – omówienie (klasyfikacja, intencja/cel, warunki stosowania, struktura/elementy/współdziałanie, konsekwencje).

1. Klasyfikacja wzorców projektowych

Wzorce projektowe są klasyfikowane w trzy główne grupy, w zależności od ich celu i charakterystyki:

1. **Wzorce kreacyjne** – dotyczą procesu tworzenia obiektów w programie, pomagają w zarządzaniu obiektami w sposób elastyczny i decydują o tym, w jaki sposób obiekty są tworzone.
    - Przykład: **Singleton**, **Factory Method**, **Abstract Factory**.
2. **Wzorce strukturalne** – koncentrują się na organizowaniu klas i obiektów w większe struktury, umożliwiające ich lepszą organizację, rozdzielność oraz łatwość współdziałania.
    - Przykład: **Adapter**, **Bridge**, **Decorator**, **Composite**.
3. **Wzorce behawioralne** – dotyczą komunikacji między obiektami oraz sposobu, w jaki obiekty współdziałają, definiują interakcje pomiędzy nimi, np. w kontekście obsługi zdarzeń czy decyzji.
    - Przykład: **Observer**, **Strategy**, **Command**, **Template Method**.

2. Wzorzec projektowy Strategia (Strategy)

- **Klasyfikacja:** Behawioralny
- **Intencja/przeznaczenie:** Wzorzec ten pozwala na zdefiniowanie rodziny algorytmów, które można wymieniać w czasie działania programu. Umożliwia wybór algorytmu w zależności od kontekstu lub sytuacji.
- **Warunki stosowania:** Używany, gdy:
    - Istnieje potrzeba zmiany algorytmu w czasie działania programu.
    - Chcemy unikać zbyt rozbudowanego kodu warunkowego (np. „if-else” lub „switch”).
    - Algorytmy są często zmieniane lub rozszerzane.
- **Struktura/elementy/współdziałanie:**
    - **Context** – obiekt, który przechowuje referencję do obiektu strategii i deleguje wykonanie algorytmu.
    - **Strategy** – interfejs wspólny dla wszystkich klas strategii, który deklaruje metodę wykonywania algorytmu.
    - **ConcreteStrategy** – klasy implementujące różne algorytmy.
- **Konsekwencje:**
    - Eliminuje długie instrukcje warunkowe w kodzie.
    - Umożliwia łatwe dodawanie nowych algorytmów bez zmiany kodu klienta.
    - Wymaga stworzenia wielu klas, co może zwiększyć złożoność systemu.

3. Wzorzec projektowy Most (Bridge)

- **Klasyfikacja:** Strukturalny

- **Intencja/przeznaczenie:** Oddziela abstrakcję od implementacji, umożliwiając ich niezależny rozwój. Pomaga w przypadku, gdy chcemy zmieniać zarówno abstrakcję, jak i implementację bez wzajemnego wpływu.
- **Warunki stosowania:** Używany, gdy:
    - Abstrakcja i implementacja w systemie mogą się rozwijać niezależnie.
    - System ma wiele różnych wariantów implementacji.
- **Struktura/elementy/współdziałanie:**
    - **Abstraction** – definiuje interfejs dla klienta, deleguje zadania do implementacji.
    - **RefinedAbstraction** – rozszerza abstrakcję o dodatkową funkcjonalność.
    - **Implementor** – interfejs, który definiuje operacje, które będą realizowane przez implementacje.
    - **ConcreteImplementor** – klasy, które implementują interfejs implementora.
- **Konsekwencje:**
    - Umożliwia niezależny rozwój abstrakcji i implementacji.
    - Pozwala na łatwiejsze dodawanie nowych implementacji.
    - Może zwiększyć liczbę klas w systemie.

4. Wzorzec projektowy Metoda Szablonowa (Template Method)

- **Klasyfikacja:** Behawioralny
- **Intencja/przeznaczenie:** Zdefiniowanie szkieletu algorytmu w metodzie, pozostawiając implementację niektórych kroków w podklasach. Pozwala to na ponowne wykorzystanie ogólnej struktury algorytmu przy dostosowaniu szczegółów w klasach potomnych.
- **Warunki stosowania:** Używany, gdy:
    - Chcemy zdefiniować wspólną część algorytmu, ale niektóre kroki mają różne implementacje w różnych klasach.
    - Chcemy zminimalizować duplikowanie kodu.
- **Struktura/elementy/współdziałanie:**
    - **AbstractClass** – zawiera szablon metody, która wywołuje kroki algorytmu (częściowo zaimplementowane lub puste metody).
    - **ConcreteClass** – klasy dziedziczące, które implementują szczegóły metod.
- **Konsekwencje:**
    - Zmniejsza duplikację kodu.
    - Zmienia kontrolę przepływu algorytmu (przeciążanie metod w podklasach).
    - Może ograniczyć elastyczność, gdy metoda szablonowa jest zbyt ściśle związana z konkretną implementacją.

5. Wzorzec projektowy Dekorator (Decorator)

- **Klasyfikacja:** Strukturalny
- **Intencja/przeznaczenie:** Pozwala na dynamiczne dodawanie nowych funkcji do obiektu bez zmiany jego struktury. Dzięki temu możliwe jest rozszerzenie klasy o dodatkowe funkcjonalności, nie zmieniając jej kodu źródłowego.
- **Warunki stosowania:** Używany, gdy:
    - Chcemy rozszerzyć funkcjonalność obiektu w sposób elastyczny.
    - Klasa bazowa jest niezmienna lub zbyt duża, aby mogła zostać rozszerzona przez dziedziczenie.
- **Struktura/elementy/współdziałanie:**
    - **Component** – interfejs lub klasa abstrakcyjna, która definiuje funkcje, które będą dekorowane.
    - **ConcreteComponent** – klasa, której funkcjonalność będzie rozszerzana.

    - **Decorator** – klasa, która dodaje dodatkową funkcjonalność, przekazując wywołania do obiektu komponentu.
    - **ConcreteDecorator** – klasy dekoratorów, które dodają konkretne funkcjonalności.
- **Konsekwencje:**
    - Elastyczność w dodawaniu funkcji obiektów.
    - Wzrost liczby klas i obiektów w systemie.
    - Możliwość "nakładania" wielu dekoratorów na jeden obiekt.

6. Wzorzec projektowy Kompozyt (Composite)

- **Klasyfikacja:** Strukturalny
- **Intencja/przeznaczenie:** Umożliwia traktowanie pojedynczych obiektów i ich zbiorów w jednolity sposób, tworząc hierarchiczną strukturę. Ułatwia pracę z kolekcjami obiektów.
- **Warunki stosowania:** Używany, gdy:
    - Chcemy operować na pojedynczych obiektach i ich grupach w jednolity sposób.
    - Potrzebujemy reprezentować hierarchię obiektów.
- **Struktura/elementy/współdziałanie:**
    - **Component** – interfejs dla zarówno pojedynczych obiektów, jak i ich zbiorów.
    - **Leaf** – pojedyncze obiekty, które nie zawierają innych obiektów.
    - **Composite** – obiekt, który może zawierać inne komponenty (zarówno liście, jak i inne kompozyty).
- **Konsekwencje:**
    - Ułatwia operowanie na strukturach drzewiastych.
    - Umożliwia rekursywne przetwarzanie komponentów.
    - Może prowadzić do nadmiaru klas w przypadku bardziej złożonych hierarchii.

7. Wzorzec projektowy Fabryka Abstrakcyjna (Abstract Factory)

- **Klasyfikacja:** Kreacyjny
- **Intencja/przeznaczenie:** Umożliwia tworzenie rodzin powiązanych obiektów bez wskazywania ich konkretnych klas. Zapewnia interfejs do tworzenia obiektów w rodzinie.
- **Warunki stosowania:** Używany, gdy:
    - Istnieje potrzeba tworzenia obiektów w ramach rodzin produktów.
    - Chcemy oddzielić kod tworzący obiekty od reszty aplikacji.
- **Struktura/elementy/współdziałanie:**
    - **AbstractFactory** – interfejs, który deklaruje metody tworzenia produktów.
    - **ConcreteFactory** – klasy implementujące tworzenie produktów.
    - **AbstractProduct** – interfejs dla produktów.
    - **ConcreteProduct** – konkretne implementacje produktów.
- **Konsekwencje:**
    - Umożliwia łatwą wymianę rodzin produktów.
    - Zwiększa liczbę klas w systemie.
    - Zwiększa zależności między fabrykami i produktami.

8. Wzorzec projektowy Obserwator (Observer)

- **Klasyfikacja:** Behawioralny
- **Intencja/przeznaczenie:** Umożliwia powiadamianie jednego lub wielu obiektów o zmianie stanu obiektu, którym są zainteresowane.
- **Warunki stosowania:** Używany, gdy:
    - Jeden obiekt musi powiadomić inne o zmianie swojego stanu.

    - Klientom zależy na niezależnym reagowaniu na zmiany w obiektach.
- **Struktura/elementy/współdziałanie:**
    - **Subject** – obiekt, którego stan jest monitorowany.
    - **Observer** – obiekty, które są informowane o zmianach stanu obiektu subject.
    - **ConcreteSubject** – klasa implementująca obiekt, którego stan jest obserwowany.
    - **ConcreteObserver** – klasy, które reagują na zmiany stanu obiektu.
- **Konsekwencje:**
    - Zmniejsza sprzężenie między obiektami.
    - Pozwala na elastyczne dodawanie nowych obiektów do powiadomień.
    - Może prowadzić do nadmiaru powiadomień w przypadku dużych systemów.

9. Wzorzec projektowy Model-Widok-Kontroler (Model-View-Controller)

- **Klasyfikacja:** Behawioralny
- **Intencja/przeznaczenie:** Oddzielenie logiki aplikacji (model) od interfejsu użytkownika (widok), z kontrolerem pośredniczącym pomiędzy nimi. Dzięki temu można zmieniać interfejs użytkownika bez wpływu na logikę aplikacji.
- **Warunki stosowania:** Używany w aplikacjach, które mają złożony interfejs użytkownika, a także w aplikacjach webowych.
- **Struktura/elementy/współdziałanie:**
    - **Model** – zawiera dane i logikę aplikacji.
    - **View** – wyświetla dane użytkownikowi.
    - **Controller** – pośredniczy między modelem a widokiem.
- **Konsekwencje:**
    - Rozdziela odpowiedzialności, co poprawia organizację kodu.
    - Umożliwia niezależną zmianę logiki i interfejsu.
    - Może prowadzić do większej złożoności w projektowaniu aplikacji.

Inżynieria i analiza danych

1. Podstawowe operacje na tablicach NumPy oraz praca z obiektami DataFrame i Series w Pandas

NumPy - Podstawowe operacje

- **Tworzenie tablic**:

```
import numpy as np
array_1d = np.array([1, 2, 3, 4])
array_2d = np.array([[1, 2], [3, 4]])
```

- **Podstawowe operacje na tablicach**:
    - Suma elementów:

```
np.sum(array_1d)
```

    - Średnia, mediana, wariancja:

```
np.mean(array_1d), np.median(array_1d), np.var(array_1d)
```

- Operacje matematyczne:

```
array_1d + 5  # dodanie liczby
array_1d * 2  # mnożenie przez liczbę
np.dot(array_1d, array_1d)  # iloczyn skalarny
```

Pandas - DataFrame i Series

- **Tworzenie obiektów**:

```
import pandas as pd
series = pd.Series([1, 2, 3, 4])
data = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
df = pd.DataFrame(data)
```

- **Podstawowe operacje**:
  - Dostęp do kolumn i wierszy:

```
df['col1']  # dostęp do kolumny
df.iloc[0]  # dostęp do pierwszego wiersza
```

  - Statystyki:

```
df.describe()  # podstawowe statystyki opisowe
df.mean()  # średnia
```

  - Zmiana wartości:

```
df['col1'] = df['col1'] * 2
```

2. Podstawy statystyki z wykorzystaniem Pythona

Pomiar tendencji centralnej i dyspersji

- **Średnia**:

```
np.mean(data)
```

- **Mediana**:

```
np.median(data)
```

- **Wariancja i odchylenie standardowe**:

```
np.var(data), np.std(data)
```

Współczynnik korelacji i kowariancja

- **Korelacja**:

```
np.corrcoef(data1, data2)
```

- **Kowariancja**:

  ```
  np.cov(data1, data2)
  ```

Centralne twierdzenie graniczne

- **Zastosowanie**: W przypadku dużych prób rozkład średnich z próby dąży do rozkładu normalnego.

  ```
  sample_means = [np.mean(np.random.choice(data, size=30)) for _ in range(1000)]
  ```

Testy statystyczne

- **Test t-studenta**:

  ```
  from scipy import stats
  stats.ttest_ind(data1, data2)
  ```

- **Test chi-kwadrat**:

  ```
  stats.chisquare(f_obs=data1, f_exp=data2)
  ```

Rozkłady prawdopodobieństwa

- **Rozkład normalny**:

  ```
  np.random.normal(loc=0, scale=1, size=1000)
  ```

3. Techniki czyszczenia i przygotowywania danych do analizy

- **Usuwanie brakujących wartości**:

  ```
  df.dropna()  # usuwa wiersze z brakującymi danymi
  df.fillna(value=0)  # uzupełnia brakujące wartości
  ```

- **Zmiana typu danych**:

  ```
  df['col1'] = df['col1'].astype(int)
  ```

- **Usuwanie duplikatów**:

  ```
  df.drop_duplicates()
  ```

4. Techniki odczytywania i zapisywania danych

- **Odczyt danych z plików CSV**:

  ```
  df = pd.read_csv('data.csv')
  ```

- **Zapis danych do pliku CSV**:

```
df.to_csv('output.csv', index=False)
```

- **Odczyt i zapis danych binarnych (np. z pliku Excel)**:

```
df = pd.read_excel('data.xlsx')
df.to_excel('output.xlsx', index=False)
```

- **Interfejs sieciowy** (np. odczyt z API):

```
import requests
response = requests.get('https://api.example.com/data')
data = response.json()
```

5. Wizualizacja danych w języku Python

- **Rodzaje wykresów**:
  - **Wykres liniowy**:

```
import matplotlib.pyplot as plt
plt.plot(x, y)
plt.show()
```

  - **Wykres słupkowy**:

```
plt.bar(x, y)
plt.show()
```

  - **Wykres punktowy**:

```
plt.scatter(x, y)
plt.show()
```

- **Wizualizacja zależności (np. wykres korelacji)**:

```
import seaborn as sns
sns.heatmap(df.corr(), annot=True)
plt.show()
```

6. Analiza regresyjna

- **Regresja liniowa**:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

  - **Ocena skuteczności modelu**:

```
from sklearn.metrics import mean_squared_error, r2_score
mse = mean_squared_error(y_test, predictions)
```

```
r2 = r2_score(y_test, predictions)
```

- **Regresja wielomianowa**:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X_train)
model.fit(X_poly, y_train)
```

- **Regresja logistyczna**:

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_train, y_train)
```

7. Klasyfikacja

- **Naiwny klasyfikator Bayesa**:

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

- **Drzewa decyzyjne**:

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

- **Maszyny wektorów nośnych (SVM)**:

```
from sklearn.svm import SVC
model = SVC()
model.fit(X_train, y_train)
```

- **Podział danych na zestaw uczący i testowy**:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2
```

- **Ocena jakości modelu**:

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, predictions)
```

8. Redukcja wymiarowości i analiza skupień

- **PCA (Principal Component Analysis)**:

```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)
```

- **Analiza skupień - K-means**:

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

- **Analiza skupień - DBSCAN**:

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN()
dbscan.fit(X)
```

9. Przetwarzanie i analiza dużych zbiorów danych w Pythonie

- **Praca z dużymi zbiorami danych (np. Pandas z Dask)**:

```
import dask.dataframe as dd
df = dd.read_csv('large_file.csv')
df.compute()  # do obliczenia wyników
```

- **Przetwarzanie równoległe i użycie Big Data** (np. PySpark):

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("BigDataAnalysis").getOrCreate()
df = spark.read.csv('large_file.csv', header=True, inferSchema=True)
```

By Hubertus Bubertus