Principiile SOLID și automatizarea dezvoltării software

Conf. dr. Cristian KEVORCHIAN

Facultatea de Matematică și Informatică

Principiile SOLID

Principiile SOLID reprezintă un set de cinci principii fundamentale pentru proiectarea și dezvoltarea software-ului orientat pe obiecte, introduse de Robert C. Martin. Acestea au scopul de a îmbunătăți calitatea codului și de a facilita întreținerea și extinderea aplicațiilor.

1. **Single Responsibility Principle (SRP)** - Fiecare clasă ar trebui să aibă o singură responsabilitate sau un singur motiv pentru a fi modificată. Acest principiu promovează idea separării logicii în module mai mici și independente.
2. **Open/Closed Principle (OCP)** - Clasele sau modulele ar trebui să fie deschise pentru extindere, dar închise pentru modificare. Asta înseamnă că putem adăuga funcționalități noi fără a afecta codul existent.
3. **Liskov Substitution Principle (LSP)** - Obiectele unei clase derivate ar trebui să poată fi utilizate oriunde sunt utilizate obiecte ale clasei de bază, fără a afecta corectitudinea programului. Acest lucru susține utilizarea interfețelor și moștenirii corecte.
4. **Interface Segregation Principle (ISP)** - Clasele nu ar trebui să fie obligate să implementeze interfețe pe care nu le utilizează. Este mai bine să avem mai multe interfețe specifice decât una generală și prea complexă.
5. **Dependency Inversion Principle (DIP)** - Modulul de nivel superior nu ar trebui să depindă de modulul de nivel inferior. Ambele ar trebui să depindă de abstracții. Abstracțiile nu ar trebui să depindă de detalii, ci invers.

Principii de Ghidare Semantica și Sintactică în Programare

	Ghidare Semantică	Ghidare Sintactică
Rol	Conceptual, abstract	Tehnic, detaliat
Obiectiv	Structurează problema în termenii gândirii algoritmice	Optimizează implementarea codului
Durată de impact	Pe tot ciclul de viață al rezolvării	Tactic (pe termen scurt)
Exemple	SRP (Single Responsibility Principle), DIP (Dependency Inversion Principle)	OCP (Open/Closed Principle), LSP (Liskov Substitution Principle), ISP (Interface Segregation Principl

SRS - Principiu de ghidare semantică

- **SRP (Single Responsibility Principle)**:
  - Este esențial pentru claritatea conceptuală și separarea logicii de business de implementare. Fiecare clasă având o singură responsabilitate înseamnă că programatorul trebuie să înțeleagă rolul fiecărei părți a sistemului într-un mod logic și ordonat.
  - Este o reflectare a actului de creație deoarece influențează modul în care sunt concepute entitățile și relațiile dintre ele.
- **În practică:**
  - Această separare nu doar ghidează gândirea programatorului, ci și facilitează adaptarea soluției la noi cerințe sau schimbări tehnologice. Reprezintă o filozofie centrală în dezvoltarea unor arhitecturi precum MVC (Model-View-Controller) dar și în folosirea abstractizărilor și interfețelor.

DIP – Principiu de ghidare semantică

- **DIP (Dependency Inversion Principle)** este un principiu care se bazează pe **gândirea abstractă** a programatorului, permițându-i să conceptualizeze relațiile dintre componentele unui sistem într-un mod **decuplat** și **flexibil**. În esență, DIP încurajează utilizarea **abstractizării** pentru a crea o arhitectură software robustă și extensibilă.
  - Decuplarea nivelurilor de abstractizare: În loc ca modulele de nivel superior să depindă direct de modulele de nivel inferior, ambele depind de abstractizări (interfețe sau clase abstracte).Acest lucru previne cuplajul strâns și facilitează schimbarea sau înlocuirea implementării fără a afecta logica principală.
  - Abstracții, nu detalii: DIP pune accentul pe utilizarea interfețelor și claselor abstracte pentru definirea relațiilor dintre componente, eliminând dependența de implementări concrete.

OCP-Principiu de ghidare sintactică

- **OCP(Open/Close Principle)** este un principiu mai apropiat de categoria **"best practices"** sau sintactică, decât de nivelul fundamental și conceptual pe care îl întâlnim la SRP și DIP. Este o regulă tehnică aplicată pentru a organiza codul în mod extensibil și sigur, ceea ce îl face mai mult o indicație practică decât o bază pentru gândirea abstractă.
- **În practică principiul este dedicat extensibilității**:

  OCP sugerează că **clasele trebuie să fie deschise pentru extensie, dar închise pentru modificare**, ceea ce înseamnă că este conservată logica programului existent, asigurând o structură modulară și extensibilă, fără să necesite modificări directe ale codului existent. De aceea, este considerat mai mult aparținând unor tehnici de ”**best practice”** decât operand la nivel conceptual, în comparație cu SRP sau DIP.

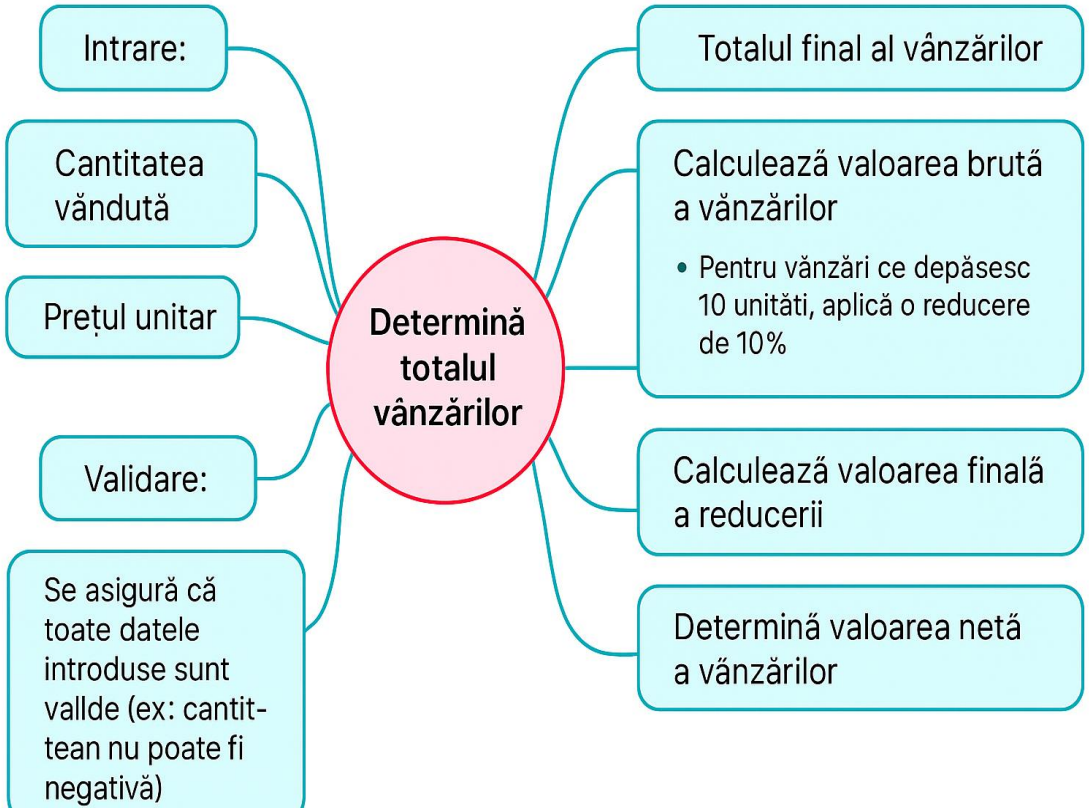
LSP-Principiu de ghidare sintactică

- **LSP(Lisko Substitution Principle)** este un principiu care poate fi privit, ca fiind mai degrabă un principiu **sintactic** decât unul conceptual, asemănător cu **Open/Closed Principle (OCP)**. LSP se concentrează pe specificația tehnică a relației dintre clasele bază și clasele derivate, ceea ce îl face mai orientat spre aplicarea practică și mai puțin axat pe gândirea abstractă a programatorului.
- LSP este o regulă specifică pentru moștenire, iar aplicarea sa este mai tehnică, axată pe cum să scriem codul corect și să evităm erorile.Spre deosebire de principiile conceptuale precum DIP sau SRP, LSP nu afectează modul în care dezvoltatorul își conceptualizează relațiile dintre componente, ci asigură doar că aceste relații sunt implementate conform unui standard tehnic.
- LSP este esențial pentru utilizarea corectă a polimorfismului și moștenirii, dar se încadrează mai mult în zona **"best practices"** și **"sintaxă"** a design-ului software decât în gândirea abstractă. Este un instrument pentru garantarea consistenței comportamentelor, mai degrabă decât o bază pentru structurarea arhitecturii conceptuale.

ISP-Principiu de ghidare semantică

- **ISP(Interface Segregation Principle):** clasele nu ar trebui să implementeze metode pe care nu le utilizează ci să creeze interfețe specifice minimale în locul unor interfețe generale de mare complexitate.
- **ISP vizează sintaxa codului, asigurând:**
  - Interfețele sunt simplificate și utilizabile fără a include metode irelevante.
  - Clasele rămân responsabile doar pentru ceea ce le este strict necesar.
- **ISP este sintactic:**
  - ISP nu influențează direct **arhitectura conceptuală** a unui sistem; în schimb, se concentrează pe **detaliile de implementare tehnică**.
  - Acest principiu ghidează programatorul în scrierea corectă și eficientă a **interfețelor**, ceea ce face ca aplicarea sa să fie o formă de **best practice**.

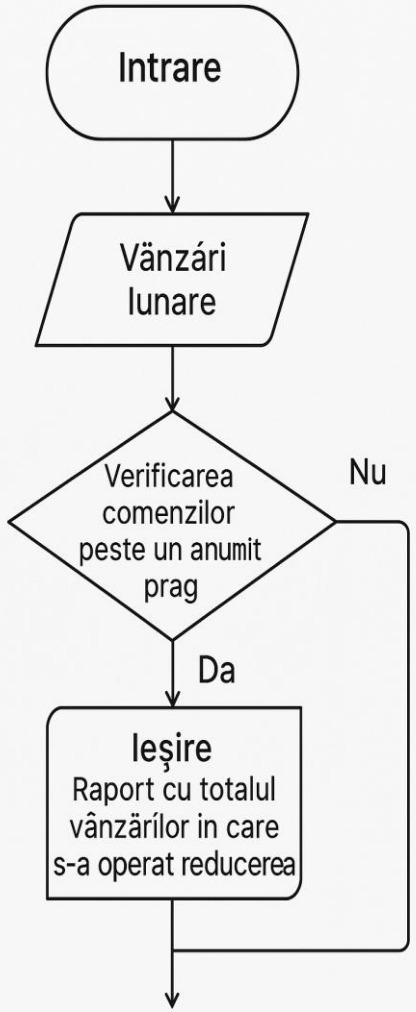
Programerea o chestiune de limbaj...



1. Formularea problemei în limbaj natural

- **Limbajul natural:** Problemele sunt descrise într-un mod accesibil și intuitiv, folosind limbajul natural. Acest pas este crucial pentru definirea clară a cerințelor și a obiectivelor problemei enunțate.
    - Exemplu: "Calculați totalul vânzărilor pentru ultima lună și aplicați o reducere de 10% pentru comenzile peste 100 de unități monetare."
- **Analiză conceptuală:** Analiza conceptuală este, în esență, o metodă logică de organizare a ideilor și relațiilor într-un mod abstract, iar utilizarea unui **mind map** este un instrument excelent pentru a structura și înțelege problema de rezolvat. Poate fi folosită pentru orice tip de problemă, fie că este legată de programare, design sau logică de business.

Programerea o chestiune de limbaj...




```
def calculate_discount(sales, discount=0.1):
    if not sales:  # Verificare pentru listă goală
        return 0, 0  # Returnează 0 pentru total și reduceri
    total = 0
    total_discount = 0
    for sale in sales:
        if sale > 100:
            reduced_price = sale * (1 - discount)
            total_discount += sale * discount
            total += reduced_price
        else:
            total += sale
    return total, total_discount
```

2. Procesarea în gândirea algoritmică

- **Gândirea algoritmică:** instrumentul fundamental care ne permite să organizăm și să sintetizăm informația pentru a crea harta mentală a problemei. Practic, această gândire te ajută să structurezi problema pas cu pas, identificând componentele cheie, relațiile dintre ele și fluxul logic necesar pentru a ajunge la o soluție.

  Problemele descrise sunt formalizate în pași clari, ordonați și logici. Aceasta implică organizarea datelor, structurarea condițiilor și identificarea soluțiilor.
  - Exemplu de algoritm:
    - *Input:* Vânzări lunare.
    - *Decizie:* Verificarea comenzilor peste un prag.
    - *Output:* Total cu reducere.
- Acesta este momentul în care limbajul uman este transformat într-o reprezentare abstractă implementabilă.

Sisteme IA generative de tip întrebare-răspung cu controlul contextului

Un **sistem de inteligență artificială generativ de tip întrebare-răspuns cu control al contextului** este un model de calcul avansat care combină tehnici de procesare a limbajului natural (NLP) cu mecanisme de gestionare a contextului conversațional pentru a produce răspunsuri relevante și coerente. Caracteristicile cheie ale unui astfel de sistem includ:

1. **Capacitate Generativă:** Potențialul de a crea răspunsuri noi și creative pe baza întrebărilor utilizatorului, fără a se limita la răspunsuri predefinite.
2. **Întreținerea contextului:** Sistemul păstrează istoricul interacțiunilor (de o manieră contextuală) pentru a înțelege mai bine intențiile utilizatorului și pentru a oferi răspunsuri adecvate și consecvente.
3. **Algoritmi de învățare profundă:** De obicei, utilizează rețele neuronale de tip transformer (ex. GPT, BERT) pentru a procesa și genera text, având o arhitectură optimizată pentru secvențe de limbaj natural.
4. **Adaptabilitate:** Este capabil să răspundă diverselor tipuri de cerințe, inclusiv întrebări simple, sarcini complexe sau generare de conținut personalizat.
5. **Control al contextului:** Sistemul își ajustează răspunsurile în funcție de scopul și direcția conversației, utilizând tehnici precum atenția contextuală (attention mechanisms).

Tockenizarea

1. **Fragmentarea textului brut**:
   - Textul inițial este împărțit în unități discrete, numite **tokeni**.
   - Permite procesarea precisă a textului.
2. **Crearea unei structuri numerice**:
   - Tokenii sunt asociați unor valori numerice (ID-uri) pentru a putea fi procesați de rețeaua neuronală.
3. **Gestionarea vocabularului**:
   - Este definit un vocabular optimizat, care include tokeni și sub-tokeni frecvent utilizați. Astfel, se minimizează problemele legate de cuvinte necunoscute (**OOV** - out-of-vocabulary).
4. **Captarea relațiilor semantice**:
   - Tokenizarea permite înțelegerea componentelor lingvistice sau tehnice, transformând codul într-o formă procesabilă de către model.
5. **Pregătirea embedding-urilor**:
   - Tokenii sunt transformați în reprezentări vectoriale (embedding-uri), care sunt utilizate pentru a construi contextul semnificativ al codului.

Embedding

- Embedding-ul reprezintă etapa în care fiecare token rezultat din procesul de tokenizare este transformat într-un **vector numeric dens** ce capturează semantica acelui token. Este esențial pentru ca modele precum transformerele să proceseze textul sau codul în mod eficient.

1. Matricea de embedding:

- Fiecare token este asociat unui vector numeric utilizând o matrice predefinită de embedding. Aceasta este antrenată pentru a învăța relațiile semantice și contextuale dintre tokeni. Token-ul „funcție” este reprezentat ca un vector [0.45, -0.18, 0.34, ..., 0.72].

2. Spatial vectorial:

- Vectorii embedding sunt formează o structură de spațiu n-dimensional în care tokenii similari (din punct de vedere semantic) se află mai aproape unii de ceilalți.

3. Dimensiunea embedding-urilor:

Dimensiunea (de exemplu, 256 sau 768) definește complexitatea și capacitatea de a extrage relațiile semantice dintre tokeni.

4. Optimizarea embedding-urilor:

În timpul antrenării, embedding-urile sunt ajustate pentru a surprinde mai exact semantica preluată din datele de intrare.

5. Rolul embedding-ului:Semantica tokenilor:

Prin embedding se asigură că fiecare token este reprezentat astfel încât relațiile semantice să fie . Embedding-ul devine input principal pentru mecanismele de atenție și procesare ulterioară.

Retele neuronale ce genereaza embedding-uri

Rețele responsabile pentru crearea reprezentărilor vectoriale ale specificațiilor, codului sau documentație.
Word2Vec, GloVe → Utilizează modele neuronale simple, cum ar fi rețele feedforward.

- **LSTM, RNN** → Sunt folosite uneori pentru embedding-uri ce captează relații secvențiale.
- **BERT, GPT** → Transformerele moderne generează embedding-uri **contextuale**.

Aceste rețele învață **relațiile semantice** dintre cuvinte pentru a le reprezenta eficient în spațiul vectorial.

Transformer pentru Generarea de Cod

Un transformer, precum GPT sau alte modele asemănătoare, poate fi privit ca un **predictor de tokeni** care utilizează mecanisme de **self-attention** sau **multi-head attention** pentru a analiza relațiile dintre tokeni și pentru a genera secvențe coerente.

1. Predictor de tokeni:

- Transformer-ul prezice fiecare token (caracter, cuvânt sau unitate semantică) din cod pe baza contextului dat, procesând secvențial intrarea.
- Acesta generează cod token cu token, folosind **contextul anterior** pentru a alege următorul token cel mai probabil.

2. Self-Attention:

- Permite modelului să înțeleagă relațiile dintre diferiti tokeni dintr-o secvență de intrare dată.

3. Multi-Head Attention:

- Utilizează mai multe "capete de atenție" pentru a examina relațiile între tokeni pe diferite reprezentări. Un cap poate analiza structura sintactică, altul se poate concentra pe semnificația logică.
- Prin combinarea rezultatelor din capetele de atenție, modelul poate genera cod care nu este doar sintactic corect, ci și semantic relevant.

4. Construcția codului token cu token:

- În timpul generării, modelul utilizează softmax pentru a evalua probabilitățile următorului token și alege tokenul cel mai potrivit.Procesul continuă până când se atinge un token de oprire, cum ar fi punctul ; sau o anumită lungime a secvenței.

Rețele pentru mecanismul de atenție

Mecanismul de atenție permite modelelor să determine **care părți ale inputului sunt relevante** și câtă importanță trebuie să acorde fiecărei părți, in scopul optimizării fluxului de informatii facilitand alegerea modelului intre ce să rețină și ce să ignore.

- **Multi-Head Self-Attention** → Fiecare "head" analizează diferite aspecte ale textului.
- **Query, Key, Value** → Aceste structuri permit modelului să compare și să prioritizeze informația.
- **Transformerele (ex. BERT, GPT)** → Utilizează mecanisme de atenție pentru a ajusta modul în care interpretarea se face pe baza contextului.

Concluzie

Mecanismul de atenție acționează ca un sistem de acord fin pentru embedding-urile codului, ajutând modelul să determine **ce părți sunt relevante** și **cum să genereze cod inteligent**. Softmax normalizează aceste informații pentru ca distribuția importanței fiecărei componente să fie optimă. În aceste condiții, codul generat de modelele bazate pe transformere nu este doar un **șir de caractere**, ci **o soluție inteligentă**, bazată pe înțelegerea profundă a semnificației și logicii de dezvoltare a codului.