P05a Transkript predavanja

(Slajd 1)
Mi smo počeli sa ovom temom objektno-orijentisanog projektovanja i mislim da smo prošli put stigli do ovog dodatka klasičnoj priči o objektno-orijentisanom projektovanju, a to je kako u ovim modernim uslovima, kada za generisanje koda koristimo AI asistente, odnosno velike jezičke modele koji znaju puno toga, kako to utiče na celu priču o razvoju softvera, konkretno projektovanju i implementaciji.

(Slajd 2)

Najvažnija promena tu je ova nova ključna oblast — Prompt Engineering, odnosno oblikovanje upita za te AI asistente. Način kako vi njima postavite neki zahtev utiče na kvalitet koda ili bilo čega drugog što tražite, koji ćete dobiti.

(Slajd 3)

Tako da se praktično taj pojam odnosi na proces oblikovanja promptova u cilju generisanja koda koji ima neki profesionalni kvalitet.
To, u suštini, znači da treba da se pridržavamo svih dizajnerskih klasičnih ograničenja i stvari koje smo definisali, nije to nešto sasvim novo, AI generiše kod dosta slično čoveku i može da generiše i loš i dobar kod. Uopšteno, umesto da kažemo najkraće šta želimo, treba definisati kontekst, programski jezik, biblioteke i razna specifična ograničenja. Kroz niz primera u nastavku videćemo neke preporuke, to je isto stvar koja je iskustvena, menjaju se i modeli, ljudi stiču sve više iskustva s njima.

Sami modeli su počeli tako što ste mogli samo da ih pitate kroz tekstualne promptove, i kod koji oni daju koristite dalje. Novi agenti mogu čak i da pokrenu razne alate i tako dalje sami. U principu, to nije samo jedan upit, to je iterativan proces, kroz dodatna pitanja dolazite do onoga što želite. I to nije potpuno neograničeno, jer su najbolji modeli tipično komercijalni i uvode ograničenja u korišćenju. Kada želite nešto ozbiljnije da radite, to košta, broji se broj tokena koji razmenite s njima.

Na tom mestu skoro je jedan diplomac kod mene radio, zadao sam mu temu iz programskih prevodilaca, predmet koji imate na četvrtoj godini. Postoji obavezan projekat pravljenja mini kompajlera. Dobijate neke biblioteke, ali dosta koda morate sami da povežete i prilagodite zahtevima. To se uvek radilo individualno i na klasičan način. Ja sam njemu kroz diplomski zadatak dao da, koristeći ChatGPT 5 i Claude 4.5, koji su tada bili aktuelni, pokuša da kompletno ne piše ništa samostalno, nego kroz Prompt Engineering i interaktivno postavljanje upita izvede ceo projekat. To je, u suštini, kruna tog predmeta, često je teže uraditi projekat nego položiti ispit.

On je imao zadatak da to bude po svim našim pravilima. Postoje jedinični testovi koje projekat mora da prođe, a deo testova se radi i na usmenoj odbrani. Postoje zahtevi oko interfejsa i strukture. On je uspeo, nije to dugo trajalo, par nedelja. Projekat ima tri nivoa — A, B i C. A je osnovni, B srednji, C uključuje kompletne mehanizme nasleđivanja i polimorfizma i predstavlja ozbiljan mali kompajler.

On je stigao do nivoa B, oko 2500 linija koda. Rekao je da u to vreme ChatGPT ne bi mogao dalje od nivoa A, a sa Claude-om je uradio sve za nivo B, ali je tu stao jer mu je istekao budžet, potrošio je oko 100 dolara. To ilustruje da korišćenje tih alata košta. Imao je veliki broj iteracija, milioni tokena su prošli. Moguće je da je to moglo efikasnije, jer su to bila prva iskustva sa prompt inženjeringom. U interesu nam je da u što manje iteracija dođemo do rešenja.

(Slajd 4)

Obradili smo CRC kartice i SOLID principe, pa da vidimo kako se to uklapa u rad sa AI agentima.

Tehnika CRC kartica je i dalje aktuelna, jer može da posluži za strukturiranje prompta. Ona daje tačan spisak saradnika, kontekst, i kroz odgovornosti definiše šta određeni deo koda treba da radi, a šta ne treba. Na taj način se postiže i strukturiranost koda, jer vi unapred definišete klase.

Tvrdi se da dobro strukturiran prompt može da da i do 90% koda iz prvog pokušaja.

(slajd 5)

Primer je bankomat — ima odgovornosti autentifikacije korisnika, prikaza opcija, isplate gotovine, i saradnike klijent i bankarski server. Ideja je da prvo sami razmislite o problemu, uradite dekompoziciju kroz CRC kartice, pa tek onda pristupite AI-u.

Možete, na primer, reći: ti si softverski inženjer koji implementira klasu bankomat prema CRC specifikaciji, koristi interfejse za saradnike, i generiši Python/Django kod koji striktno prati odgovornosti.
Možete probati i sami. Ideja je da dođete do optimalnog načina interakcije sa AI servisima. Trenutno, u besplatnom režimu, deluje da Gemini daje više mogućnosti. Ako imate jaču mašinu, možete koristiti lokalni LLM, korišćenjem alata ollama i sličnih.

(Slajd 6)

Sad dolazimo do onih SOLID principa, i malo da pogledamo kako bi oni mogli da utiču na ovaj prompt engineering.

(Slajd 7)
Primer lošeg prompta: napiši Django view koji prima tekst posta, poziva AI za sažetak i čuva u bazu. Time se krši princip jedne odgovornosti i dependency inversion.

(Slajd 8)

AI generiše view klasu koja sve to radi — prima request, uzima tekst, poziva OpenAI API, čuva u bazu i vraća template. Kod direktno zavisi od OpenAI API-ja.
Objašnjenje Django modela: view povezuje model i template. Prima HTTP request, uzima podatke iz request-a i prosleđuje ih dalje.
Poziv AI servisa uključuje sistemski i korisnički prompt. Nakon odgovora, rezultat se čuva u bazu pomoću ORM-a.
ORM pravi objekat u memoriji, a zatim ga upisuje u bazu kroz transakciju.

(Slajd 9)
Problem tog koda je što je direktno vezan za OpenAI, teško se testira i nema fallback.

(Slajd 10)
Bolji pristup je da se napravi posebna servisna klasa OpenAISummarizer koja implementira interfejs ITextProcessor i koristi dependency injection.

(Slajd 11)
View tada koristi samo interfejs ITextProcessor i ne zna detalje implementacije.
Dependency injection se radi spolja, preko routera. To omogućava zamenu implementacije i lakše testiranje.
Router (urls.py) mapira URL na view i tu se instanciraju objekti i ubacuju zavisnosti.

(Slajd 12)
Drugi primer ilustruje kršenje Liskov principa. Zamislite da se od AI-a traži da projektuje sistem za slanje SMS obaveštenja. AI će često predložiti nešto što na prvi pogled deluje logično, baznu klasu Notification koja služi za slanje imejl obaveštenja i iz nje izvedenu klasu SMSNotification za slanje SMS obaveštenja.

(Slajdovi 13 i 14)
Problem je što bazna klasa Notification u metodu send koristi email, a izvedena SMSNotification koristi send sa formalno istim (string) a suštinski drugačijim parametrom phone_number, pa nisu zamenljive, narušen je princip Liskove. To bi dovelo do “pucanja” koda koji pokušava svim korisnicima da pošalje SMS obaveštenje, jer imejl korisnika ne može da se koristi kao njegov broj telefona.

(Slajd 15)
Rešenje je da od AI tražimo da poštuje SOLID i da koristi apstraktnu klasu bez zavisnosti od kanala komunikacije.

(Slajd 16)
Time se dobija ispravan dizajn koji poštuje SOLID principe. Metod Send u baznoj klasi Notification ima samo parameter message, od njega je sakriven kanal komunikacije. To je sada atribut parametra message i u konkretnim klasama ovaj podatak (imejl ili broj telefona) injektuje se kroz konstruktor. Konkretne klase preklapaju metod send svaka za svoje potrebe.

(Slajd 17)
Sada postoji niz konkretnih preporuka vezanih za svaki od principa koje smo pomenuli, kako oni utiču na formulisanje prompta — šta ne treba raditi, a šta treba raditi. AI treba tretirati kao junior programera. On će pokušati da uradi ono što je traženo, ali bez šireg razumevanja arhitekture. Vi preuzimate ulogu seniora ili arhitekte — vi postavljate ograničenja i definišete dizajn.

(Slajd 18)

Što se tiče CRC strukture, ne treba samo uopšteno reći: napravi mi sistem za plaćanje. Potrebno je unapred definisati koje klase postoje, koje su njihove odgovornosti i ko su saradnici. To treba osmisliti pre nego što se traži generisanje koda.

(Slajd 19)

Što se tiče principa jedne odgovornosti, preporuka je da se prompt drži malim i konkretnim. Ako u promptu imate nešto tipa: napravi funkciju koja obrađuje sliku, šalje mejl i čuva podatke u bazu, time se krši Single Responsibility Principle. Pravilo je da se traži jedna po jedna klasa ili funkcija. Kroz iteraciju možete dodavati dalje zahteve. Tako se dobija kod sa manje grešaka i lakši za testiranje.

Ne treba očekivati da jedan prompt reši sve. Treba postepeno graditi rešenje.

(Slajd 20)

Što se tiče dependency inversion principa, ne treba u promptu insistirati na konkretnim bibliotekama unutar poslovne logike. Loš primer je: napiši kod koji koristi OpenAI za analizu teksta. Bolji pristup je: napiši klasu koja implementira interfejs ITextAnalyzer i koristi dependency injection, tako da se kasnije može zameniti konkretna implementacija.

(Slajd 21)

Što se tiče Liskov principa, treba proveravati ugovore — da li je interfejs pravilno implementiran u svim izvedenim klasama. Kada AI predloži nasleđivanje, treba postaviti pitanje: da li mogu da zamenim objekat bazne klase objektom izvedene klase, a da se program ne pokvari? Ako ne mogu, treba tražiti drugačiji dizajn, na primer kompoziciju umesto nasleđivanja.

To je nešto o čemu treba razmišljati i pre formulisanja prompta.

(Slajd 22)

Peta preporuka je opšta — nakon što AI generiše kod, treba ga proveriti kroz tri kontrolne tačke:

Prvo, hardkodovanje — da li su neke vrednosti direktno ugrađene u kod, kao što su API ključevi ili putanje. U ozbiljnim aplikacijama konfiguracija treba da bude izdvojena.

Drugo, evolutivnost — da li je kod zatvoren za izmene, a otvoren za proširenja (Open-Closed princip). Treba zamisliti novi zahtev i videti da li bi njegova implementacija zahtevala izmene postojećeg koda ili samo dodavanje novog.

Treće, testabilnost — da li je moguće napisati unit test za taj kod bez pristupa internetu ili bazi podataka. To znači da li su zavisnosti pravilno izdvojene i da li se mogu zameniti mock objektima.

Postoji i termin „vibe coding“ — situacija kada se samo zada prompt, dobije kod i odmah koristi bez dubljeg razumevanja. Ideja ovde je suprotna — kod ozbiljnih aplikacija ne treba prihvatati generisan kod bez analize.