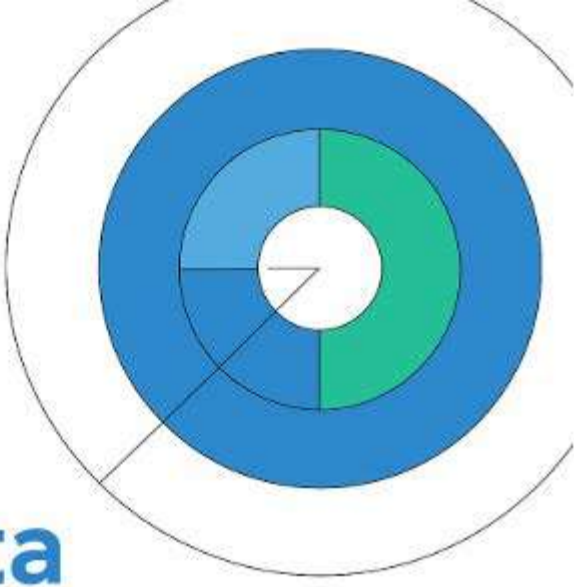
‹packt›



Čista arhitektura u programskom jeziku Python

Razvoj proširivih i održivih aplikacija uz primenu proverenih arhitektonskih principa

SAM KIN

python

Sam Kin

Čista arhitektura
u programskom jeziku Python

Razvoj proširivih i održivih aplikacija uz primenu proverenih arhitektonskih principa

kompjuter biblioteka

Packt>

Izdavač:

kompjuter biblioteka

Obalskih radnika 4a
Beograd, Srbija
Tel: 011/2520272
e-pošta: kombib@gmail.com
veb-sajt: www.kombib.rs

Za izdavača:
Mihailo J. Šolajić, urednik

Autor: Sam Kin

Prevod: Nemanja Lukić
Recezent: Miroslav Ristić
Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Apollo plus“,
Beograd
Tiraž: 500
Godina izdanja: 2025.

Broj knjige: 589
Izdanje: Prvo
ISBN: 978-86-7310-613-7

Naslov originala:

Clean Architecture with Python
ISBN 978-1-83664-289-3

Packt Publishing Ltd.
Birmingham, UK, packt.com

Java programerski problemi

Autorizovani prevod sa engleskog jezika.

SARADNICI

O AUTORU

Sam Kin je vodeći stručnjak u oblasti softverskog inženjeringa sa više od 25 godina iskustva. On je programer koji poznaje više programskih jezika i koji koristi programski jezik Python u različitim kontekstima, od malih startapova do industrijskih giganata kao što su AWS, Lululemon i Nike. Njegova stručnost obuhvata arhitekturu zasnovanu na oblaku, neprekidnu isporuku i izgradnju skalabilnih sistema. U kompaniji Lululemon, Sam je bio pionir prvog tima za razvoj aplikacija zasnovanih na oblaku, postavljajući standarde za distribuiranu arhitekturu u oblaku unutar same kompanije. Trenutno, Sam koristi programski jezik Python za projektovanje i implementaciju internih platformskih inženjerskih rešenja u okviru AWS platforme, sa fokusom na principe čiste arhitekture i održiv kod. Sam živi na severozapadu Sjedinjenih Američkih Država sa svojom voljenom suprugom i dve veoma razmažene mačke.

Želim da izrazim zahvalnost timu izdavačke kuće Packt na njihovom vođstvu i podršci tokom čitavog procesa pisanja. Posebnu zahvalnost dugujem urednicima i recenzentima, čije su povratne informacije bile neprocenjive u usavršavanju koncepata predstavljenih u ovoj knjizi. Takođe sam zahvalan zajednici otvorenog koda, čiji alati i biblioteke čine razvoj u programskom jeziku Python pravim zadovoljstvom. Na kraju, hvala mojoj porodici na nepokolebljivoj podršci tokom dugih sati provedenih u pisanju i programiranju.

O RECENZENTU

Aindrila Ghorai je glavna arhitektinja sa više od 17 godina iskustva u razvoju softvera, specijalizovana za PEGA tehnologiju, veštačku inteligenciju i rešenja u oblasti zdravstvene zaštite. Vodila je strateške inicijative za upravljanje poslovnim procesima (BPM) na nivou preduzeća i doprinosila operativnoj izvrsnosti kroz Agile-Scrum i iterativne metodologije. Aindrila je autorka više radova objavljenih u stručnim časopisima i bila je član žirija za prestižna priznanja, uključujući Globee Awards, NCWIT i Technovation Girls. Strastvena u pogledu inovacija i mentorstva, posvećena je isporuci transformativnih i održivih rešenja kroz primenu novih tehnologija i najboljih praksi.

O RECENZENTU ZA SRPSKO IZDANJE

Miroslav Ristić je redovni profesor na Prirodno-matematičkom fakultetu Univerziteta u Nišu, sa preko 25 godina iskustva u razvoju statističkog softvera. Posebno se ističe njegov rad na razvoju grafičkog korisničkog interfejsa R Commander za programski jezik R. Dugi niz godina recenzirao je značajan broj knjiga za izdavačku kuću Springer i časopis Journal of Applied Statistics. Od 2023. godine aktivno recenzira najaktuelnija izdanja izdavačke kuće "Kompjuter biblioteka". Nakon prevođenja, svako izdanje prolazi kroz njegovo stručno vrednovanje i recenziju prevoda, sa ciljem da se osigura da prevodi budu ne samo jasni, precizni i prilagođeni čitaocima, već i da održe visok kvalitet i stručnu relevantnost knjiga.

Predgovor

Čista arhitektura postala je sve značajnija u savremenom razvoju softvera, naročito kako aplikacije postaju sve složenije, a timovi moraju dugoročno da ih održavaju. Iako se arhitektonski principi često razmatraju u apstraktnim terminima, ova knjiga oživljava čistu arhitekturu kroz praktične primene u programskom jeziku Python i pokazuje kako ti koncepti mogu da promene vaš pristup razvoju.

Svestranost programskog jezika Python čini ga izuzetno pogodnim za primenu principa čiste arhitekture. Njegova dinamična priroda i bogat ekosistem omogućavaju brz razvoj, ali te iste prednosti mogu dovesti do složenih i teško održivih kodnih baza kako se aplikacije razvijaju. Čista arhitektura pruža okvir koji omogućava da se uskladi fleksibilnost programskog jezika Python sa strukturiranim i održivim dizajnom.

U ovoj knjizi prikazaćemo kako se obrasci čiste arhitekture mogu primeniti u Python projektima i kako se grade sistemi koji nisu samo funkcionalni, već i testabilni, održivi i prilagodljivi. Na primeru aplikacije za upravljanje zadacima izgradićemo kompletan sistem od samog početka i pokazati kako pravilno postavljene arhitektonske granice stvaraju softver koji se vremenom može razvijati stabilno i postepeno.

Bilo da gradite nove sisteme ili održavate postojeće, principi i prakse opisani u ovoj knjizi pomoći će vam da kreirate robusnije i fleksibilnije aplikacije u programskom jeziku Python. Prikazaćemo kako se osnovna poslovna logika odvaja od spoljnih zavisnosti, kako se grade jasni interfejsi između komponenti sistema i kako se implementiraju obrasci koji omogućavaju softveru da se prilagođava promenljivim zahtevima.

U ovoj knjizi obrađujemo sledeće glavne teme:

- koncept čiste arhitekture i primena SOLID principa u Python aplikacijama,
- obogaćivanje programskog jezika Python napomenama o tipovima radi jačanja arhitektonskih granica i interfejsa,
- izgradnja robusnih domenskih modela i sloja aplikacije koji obuhvataju poslovnu logiku, nezavisno od spoljnih uticaja,
- kreiranje jasnih interfejsa između arhitektonskih slojeva pomoću kontrolera, prezentera i adaptera,

- integracija sa okvirima i spoljnim sistemima uz očuvanje arhitektonske celovitosti,
- primena čiste arhitekture u praktičnim situacijama: testiranje, veb interfejsi, uvid u stanje sistema i transformacija nasleđenih sistema.

Zajedno, ove teme čine celovit pristup razvoju Python aplikacija koje mogu da izdrže test vremena. Do kraja ove knjige imaćete i teorijsku osnovu i praktične veštine da primenite čistu arhitekturu u sopstvenim projektima i da gradite sisteme koji su održiviji, lakši za testiranje i prilagodljivi promenama.

Kome je namenjena ova knjiga

Ova knjiga je namenjena programerima u programskom jeziku Python koji žele da stvaraju aplikacije koje je lakše održavati, testirati i prilagođavati promenama. Najviše će odgovarati programerima sa srednjim ili višim nivoom znanja, koji imaju iskustvo u radu sa programskim jezikom Python i žele da unaprede svoje arhitektonske veštine. Ako ste nailazili na kodne baze koje se teško menjaju, borili se sa zapetljanim zavisnostima ili jednostavno želite da pišete bolji kod u programskom jeziku Python, ova knjiga ponudiće vam praktične strategije i obrasce za prevazilaženje tih izazova.

Korisnici koji će naročito imati koristi od ovog materijala:

- **arhitekte softverskih sistema** koje žele da primene čiste i održive dizajne sistema u Python projektima,
- **tehnički rukovodioci** odgovorni za vođenje razvojnih timova i uspostavljanje standarda programiranja,
- **programeri koji rade na serverskoj strani sistema** i razvijaju složene aplikacije koje moraju da se razvijaju i menjaju tokom vremena,
- **DevOps inženjeri** koji žele da grade testabilnije i bolje uvidljivije Python servise.

Iako i početnici mogu imati koristi od predstavljenih koncepata, određeno poznavanje programskog jezika Python i principa objektno-orijentisanog programiranja pomoći će da se iz ovog materijala izvuče maksimum. Tehnički rukovodioci, arhitekte i iskusni programeri pronaći će dragocene uvide za primenu čiste arhitekture u timskom okruženju i arhitektonskom odlučivanju.

Šta ova knjiga obuhvata

Poglavlje 1: Osnove čiste arhitekture – transformacija razvoja u programskom jeziku Python predstavlja osnovne koncepte čiste arhitekture i objašnjava zašto su ti principi važni za programere koji programiraju u programskom jeziku Python. U ovom poglavlju uspostavljaju se ključni arhitektonski slojevi i prikazuje se kako čista arhitektura može da unapredi prakse razvoja u programskom jeziku Python.

Poglavlje 2: Osnove SOLID principa – izgradnja robusnih aplikacija u programskom jeziku Python pokazuje kako SOLID principi grade osnove čiste arhitekture. Na praktičnim primerima u programskom jeziku Python prikazano je kako se primenjuju princip jedinstvene odgovornosti (SRP), princip otvorenosti–zatvorenosti (OCP), Liskovin princip zamene (LSP), princip razdvajanja interfejsa (ISP) i princip obrnute zavisnosti (DIP).

Poglavlje 3: Programski jezik Python obogaćen tipovima – jačanje čiste arhitekture objašnjava kako napomene o tipovima u programskom jeziku Python mogu da ojačaju arhitektonske granice. Prikazuje se kako tipizacija unapređuje definisanje interfejsa, podržava obrnute zavisnosti i omogućava bolju alatnu podršku za proveru arhitekture.

Poglavlje 4: DDD – oblikovanje osnovne poslovne logike usmereno je na izgradnju robusnih domenskih modela. Objašnjeno je kako da se identifikuju i modeluju entiteti, vrednosni objekti i domenski servisi, uz očuvanje njihove nezavisnosti od spoljnih uticaja.

Poglavlje 5: Sloj aplikacije – orkestriranje slučajeva upotrebe obrađuje implementaciju slučajeva upotrebe koji povezuju objekte domena kako bi izvršili određene zadatke. Pokazuje se kako da se kreiraju čisti interfejsi između domena i spoljašnjih slojeva, uz očuvanje pravilnog razdvajanja odgovornosti.

Poglavlje 6: Sloj adaptera interfejsa – kontroleri i prezenteri prikazuje kako da se naprave kontroleri koji prevode spoljašnje zahteve i prezenteri koji oblikuju podatke domena. Posebno se naglašava kako se grade čiste granice između jezgra aplikacije i mehanizama isporuke.

Poglavlje 7: Sloj okvira i pokretača – spoljašnji interfejsi pokazuje kako da se integrišu spoljašnji okviri i infrastruktura, a da osnovna poslovna logika ostane nezavisna. Objašnjava se implementacija adaptera za baze podataka, veb okvire i spoljne servise, uz poštovanje granica čiste arhitekture.

Poglavlje 8: Primena obrazaca testiranja u čistoj arhitekturi daje strategije za sveobuhvatno testiranje preko arhitektonskih granica. Prikazuje se kako da se pišu jedinčni testovi za objekte domena, integracioni testovi za slučajeve upotrebe i testovi s kraja na kraj koji potvrđuju ponašanje sistema.

Poglavlje 9: Dodavanje veb korisničkog interfejsa – fleksibilnost interfejsa u čistoj arhitekturi prikazuje kako da se implementira veb interfejs za aplikaciju zasnovanu na čistoj arhitekturi. Na primeru okvira Flask objašnjava se kako čista arhitektura omogućava dodavanje novih interfejsa bez narušavanja postojećih funkcionalnosti.

Poglavlje 10: Implementacija uvidljivosti – nadgledanje i provera obrađuje strategije za dodavanje zapisa, praćenja i arhitektonske provere uz očuvanje granica čiste arhitekture. Pokazuje se kako da se implementiraju aspekti koji se provlače kroz više slojeva bez ugrožavanja arhitektonske celovitosti.

Poglavlje 11: Od nasleđenog koda do čiste arhitekture – refaktorisanje u programskom jeziku Python radi održivosti nudi praktične pristupe za postepenu transformaciju postojećih Python aplikacija. Objašnjene su tehnike inkrementalnog refaktorisanja koje unapređuju arhitekturu, a pri tome zadržavaju stabilnost sistema.

Poglavlje 12: Putovanje kroz čistu arhitekturu – naredni koraci pokazuje kako da se čista arhitektura primeni u različitim vrstama sistema i organizacionim kontekstima. Prikazane su strategije za arhitektonsko liderstvo, izgradnju zajednice i usklađivanje pragmatizma sa arhitektonskim principima.

Kako izvući maksimum iz ove knjige

Podrazumeva se osnovno poznavanje programiranja u programskom jeziku Python, uključujući znanje o objektno-orijentisanim konceptima kao što su klase, nasleđivanje i kompozicija. Iskustvo sa konceptima veb razvoja može biti korisno za kasnija poglavlja, ali nije obavezno.

Softver i hardver obuhvaćeni u knjizi	**Operativni sistem**
Python 3.13 ili novija verzija	Windows, macOS ili Linux

Ako koristite digitalno izdanje ove knjige, savetujemo vam da sami prekucate kod ili da mu pristupite preko GitHub spremišta knjige. Na taj način izbeći ćete moguće greške koje se mogu javiti prilikom kopiranja i umetanja koda.

Preuzimanje datoteka sa primerima koda

Kompletan kod za sve primere dostupan je u GitHub spremištu knjige na adresi: https://github.com/PacktPublishing/Clean-Architecture-with-Python. Pored toga, u odgovarajućim poglavljima pronaći ćete funkcionalnu implementaciju naše aplikacije za upravljanje zadacima u stanju koje odgovara tom poglavlju. To vam omogućava da pokrećete, testirate i istražujete radnu verziju aplikacije dok se ona postepeno razvija. Svako poglavlje se nadovezuje na prethodno, pa je preporučeno da knjigu pratite redosledom kojim je napisana, iako iskusniji programeri mogu odlučiti da se usredsrede samo na određena poglavlja koja odgovaraju njihovim trenutnim izazovima.

Takođe, na adresi https://github.com/PacktPublishing/ dostupni su i drugi paketi koda iz bogatog kataloga knjiga i video materijala. Vredi ih pogledati!

Preuzimanje slika u boji

Obezbeđujemo i PDF datoteku koja sadrži slike u boji svih snimaka ekrana i dijagrama korišćenih u ovoj knjizi. Možete je preuzeti ovde: https://packt.link/gbp/9781836642893.

Korišćene konvencije

Kroz knjigu se primenjuje nekoliko tekstualnih konvencija.

`Kod u tekstu`: označava delove koda u tekstu, imena tabela u bazi podataka, imena direktorijuma, nazive datoteka, ekstenzije, putanje, primer URL adresa, korisničke unose i oznake na X/Twitter platformi. Na primer: „Parametar `project_id` dolazi direktno iz URL adrese (`/projects/<project_id>/tasks/new`), dok polja forme sadrže detalje o zadatku.“

Blok koda prikazan je ovako:

```
# cli_main.py
def main() -> int:
    """Glavna ulazna tačka za aplikaciju komandne linije."""
    app = create_application(
        notification_service=NotificationRecorder(),
        task_presenter=CliTaskPresenter(),
        project_presenter=CliProjectPresenter(),
    )
    cli = ClickCli(app)
    return cli.run()
```

Ulaz i izlaz komandne linije prikazani su ovako:

```
> python web_main.py
 * Serving Flask app 'todo_app.infrastructure.web.app'
 * Debug mode: on
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 954-447-204
127.0.0.1 - - [05/Feb/2025 13:58:57] "GET / HTTP/1.1" 200 -
```

Podebljan tekst: označava novi pojam, važnu reč ili element koji vidite na ekranu. Na primer, stavke u menijima ili dijaloškim okvirima prikazane su tako. Na primer: „**Ograničeni konteksti** predstavljaju konceptualne granice koje definišu gde se primenjuju određeni domenski modeli.“

> **Upozorenja** ili **važna obaveštenja**
> pojavljuju se ovako.

> **Saveti i** trikovi
> pojavljuju se ovako.

Kontakt

Povratne informacije čitalaca uvek su dobrodošle.

Opšte povratne informacije: pošaljite imejl na `feedback@packtpub.com` i u predmetu poruke navedite naslov knjige. Ako imate pitanja u vezi sa bilo kojim delom ove knjige, pišite na `questions@packtpub.com`.

Ispravke: iako smo uložili veliki trud da obezbedimo tačnost sadržaja, greške se ipak mogu potkrasti. Ako pronađete neku grešku u ovoj knjizi, bićemo zahvalni ako nam je prijavite. Posetite http://www.packtpub.com/submit-errata, kliknite na **Submit Errata** i popunite obrazac.

Piraterija: ako nađete nelegalne kopije naših izdanja na internetu, bićemo vam zahvalni ako nam pošaljete adresu ili naziv veb sajta. Kontaktirajte nas na `copyright@packtpub.com` i dostavite vezu do materijala.

Ako želite da postanete autor: ukoliko postoji tema u kojoj ste stručni i zainteresovani ste da pišete ili doprinesete izradi knjige, posetite `http://authors.packtpub.com/`.

Podelite svoje utiske

Kada pročitate *Čista arhitektura u programskom jeziku Python*, bilo bi nam drago da znamo vaše utiske! `Kliknite ovde da biste otišli direktno na Amazon stranicu za recenzije` ove knjige i podelili svoje mišljenje.

Vaša recenzija je važna i nama i tehnološkoj zajednici i pomoći će nam da obezbedimo sadržaj vrhunskog kvaliteta.

Postanite član Kompjuter biblioteke

Kupovinom jedne naše knjige stekli ste pravo da postanete član Kompjuter biblioteke. Kao član možete da kupujete knjige u pretplati sa 40% popusta i učestvujete u akcijama kada ostvarujete popuste na sva naša izdanja. Potrebno je samo da se prijavite preko formulara na našem sajtu.
Link za prijavu: kombib.rs/kblista.php

Skenirajte QR kod
registrujte knjigu
i osvojite nagradu

DEO 1

Osnove čiste arhitekture u programskom jeziku Python

Ovaj deo postavlja osnovne principe i obrasce čiste arhitekture u kontekstu programskog jezika Python. Steći ćete sveobuhvatno znanje o ključnim arhitektonskim konceptima, SOLID principima i načinima na koji sistem tipova u programskom jeziku Python može da unapredi arhitektonske implementacije. Ova poglavlja pružaju neophodno znanje na kojem se zasniva ostatak knjige i pripremaju vas za uspešnu primenu čiste arhitekture u sopstvenim projektima u programskom jeziku Python.

Ovaj deo knjige obuhvata sledeća poglavlja:

- *Poglavlje 1: Osnove čiste arhitekture – transformacija razvoja u programskom jeziku Python*
- *Poglavlje 2: Osnove SOLID principa – izgradnja robusnih aplikacija u programskom jeziku Python*
- *Poglavlje 3: Programski jezik Python obogaćen tipovima – jačanje čiste arhitekture*

1

Osnove čiste arhitekture – transformacija razvoja u programskom jeziku Python

Kao Python programeri, primenjujemo najbolje prakse poput pisanja čistih funkcija, korišćenja deskriptivnih imena promenljivih i težnje ka modularnosti. Ipak, kako aplikacije rastu, često se borimo da očuvamo jasnoću i prilagodljivost u većem obimu. Jednostavnost i svestranost programskog jezika Python čine ga popularnim za projekte od veb razvoja do nauke o podacima, ali te iste prednosti lako postaju izazovi kada aplikacije postanu složenije. Tada shvatamo da nam nedostaje glavni plan — sveobuhvatna arhitektura koja usmerava naše odluke i čini projekte održivim kako se razvijaju. Upravo tu nastupa **čista arhitektura**, nudeći strukturiran pristup izgradnji aplikacija u programskom jeziku Python koje usklađuju planiranje i agilnost, dajući arhitektonsko usmerenje neophodno za održiv razvoj u velikom obimu.

Čistu arhitekturu je 2012. godine predstavio Robert C. Martin (`https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`), objedinjavanjem decenija iskustva u dizajnu softvera u jedinstven skup principa. Ona se bavi stalnim izazovima u razvoju softvera, kao što su upravljanje složenošću i prilagođavanje promenama. Primena principa čiste arhitekture u Python projektima omogućava programerima da grade sisteme koji nisu samo funkcionalni, već i održivi, testabilni i prilagodljivi tokom vremena.

U ovom poglavlju istražićemo suštinu čiste arhitekture i njen značaj za Python razvoj. Razmotrićemo kako se principi čiste arhitekture uklapaju u filozofiju programskog jezika Python zasnovanu na jednostavnosti i čitljivosti, stvarajući prirodnu sinergiju koja dodatno naglašava njegove prednosti. Videćete kako čista arhitektura može da pomogne u izgradnji Python aplikacija koje su lakše za razumevanje, izmenu i proširivanje, čak i kada postanu složenije.

Na kraju ovog poglavlja imaćete jasan pregled principa čiste arhitekture i njihovih potencijalnih koristi za Python razvoj. Biće prikazano kako ovaj pristup može da odgovori na česte izazove razvoja softvera, posebno kada projekti u programskom jeziku Python rastu u obimu i složenosti. Ovo temeljno razumevanje čiste arhitekture biće ključno kada budemo detaljnije ulazili u njenu primenu i najbolje prakse u programskom jeziku Python kroz ostatak knjige.

U ovom poglavlju bavićemo se sledećim glavnim temama:

- Zašto čista arhitektura u programskom jeziku Python – koristi od usklađivanja planiranja i agilnosti
- Šta je čista arhitektura?
- Čista arhitektura i programski jezik Python – prirodna usklađenost

Tehnički zahtevi

Primeri koda u ovom poglavlju služe samo za demonstraciju i prikazuju primenu pojedinih tema i praksi koje se ovde obrađuju. U narednim poglavljima biće predstavljeni složeniji primeri koda, uz jasno naznačene specifične zahteve gde je to potrebno. Sav kod iz svih poglavlja nalazi se u pratećem GitHub spremištu knjige: `https://github.com/PacktPublishing/Clean-Architecture-with-Python`.

Zašto čista arhitektura u programskom jeziku Python – koristi od usklađivanja planiranja i agilnosti

U ovom odeljku razmotrićemo ključnu ravnotežu između planiranja i agilnosti u Python razvoju i način na koji čista arhitektura može da pomogne da se to postigne. Analiziraćemo izazove koje donosi rastuća složenost savremenih Python aplikacija i potrebu agilnosti u današnjem dinamičnom poslovnom okruženju. Zatim ćemo se osvrnuti na odnos između planiranja i fleksibilnosti i pokazati kako arhitektonsko razmišljanje pruža okvir za upravljanje tim kompromisima. Na kraju, razmotrićemo ulogu arhitekture u savladavanju složenosti i postavljanju temelja za dugoročni uspeh. Kroz ovu raspravu steći ćete jasniju predstavu o tome zašto je čista arhitektura posebno vredna Python programerima koji žele da razvijaju održive, prilagodljive i efikasne aplikacije.

Započnimo razmatranjem izazova sa kojima se suočava savremeni razvoj u programskom jeziku Python.

Izazov složenosti savremenog razvoja u programskom jeziku Python

Kako raste popularnost programskog jezika Python, tako rastu i obim i složenost aplikacija koje se u njemu razvijaju. Od veb servisa do tokova obrade podataka, Python projekti postaju sve veći i zamršeniji.

Ovakav rast donosi ozbiljne izazove sa kojima se svaki Python programer neminovno suočava. Što su sistemi složeniji, to ih je teže razumeti, menjati i održavati. Takva složenost može ozbiljno ograničiti mogućnost dodavanja novih funkcionalnosti ili prilagođavanja promenljivim zahtevima. Teret održavanja složenih sistema u programskom jeziku Python lako može da preoptereti razvojne timove, usporavajući napredak i kočeći inovacije. Čak i najmanje izmene u velikim i složenim sistemima mogu imati dalekosežne posledice, zbog čega promene postaju skupe i rizične.

Uzmimo primer izmišljenog velikog e-trgovinskog sajta razvijenog u programskom jeziku Python: PyShop. Poslovanje odlučuje da uvede naizgled jednostavnu funkcionalnost – dodavanje ukrasnog pakovanja uz porudžbine. Međutim, ovaj jednostavan zahtev ubrzo prerasta u složen projekat:

- Modul za obradu porudžbina mora se ažurirati kako bi uključivao opcije za umotavanje pakovanja
- Sistem zaliha mora se prilagoditi da bi pratio potrošni materijal za pakovanje
- Mehanizam za obračun cena mora se izmeniti radi računanja dodatnih troškova
- Korisnički interfejs (UI) mora se ažurirati kako bi prikazao nove opcije
- Sistem realizacije porudžbina mora se izmeniti kako bi obuhvatio instrukcije za pakovanje

Zadatak koji je prvobitno procenjen na dve nedelje prerasta u projekat koji traje mesecima. Svaka izmena utiče na druge delove sistema: prilagođavanja u obradi porudžbina utiču na izveštavanje, izmene u zalihama utiču na upravljanje lancem snabdevanja, a izmene korisničkog interfejsa zahtevaju opsežno testiranje korisničkog iskustva.

Ovaj primer jasno pokazuje kako međusobno povezani moduli u složenom sistemu mogu pretvoriti jednostavnu funkcionalnost u veliki poduhvat, naglašavajući potrebu za arhitekturom koja omogućava izolovane promene i jednostavnije testiranje.

Kako Python projekti postaju sve obimniji, programeri se sve češće suočavaju sa problemima apstrakcije – pitanjem koje čista arhitektura pomaže da se reši. Bez jasnih smernica, kodne baze često skliznu u krajnosti: ili postaju zamršena mreža hijerarhija klasa koju je teško razumeti i menjati, ili se svode na monolitne klase bez stvarne apstrakcije. U prvom slučaju, težnja za ponovnim korišćenjem koda vodi do previše složenih struktura nasleđivanja, što stvara krhke sisteme u kojima

promena na jednom mestu izaziva nepredviđene posledice drugde. U drugom slučaju, odsustvo apstrakcije dovodi do masivnih i nezgrapnih klasa i do nekontrolisanog dupliranja koda, što gotovo onemogućava očuvanje konzistentnosti ili sprovođenje sistemskih promena. Oba scenarija rezultiraju kodnim bazama koje je teško razumeti, održavati i razvijati – upravo ono što dobro planirana arhitektura sprečava.

Pored toga, u današnjem dinamičnom tehnološkom okruženju složeni i čvrsto povezani sistemi teško mogu da iskoriste prednosti novih tehnologija. Ovakvo ograničenje može ozbiljno da ugrozi vašu sposobnost da ostanete konkurentni u oblasti u kojoj je tehnološka agilnost presudna.

Nužna agilnost

U današnjem ubrzanom poslovnom okruženju agilnost nije samo prednost – ona je neophodnost. Kako gotovo svaka kompanija postaje i tehnološka kompanija, pritisak da se rešenja isporučuju brzo nikada nije bio veći. Jednostavnost programskog jezika Python i njegovo bogato okruženje čine ga prirodnim izborom za brzi razvoj.

Ipak, održiva agilnost podrazumeva mnogo više od same početne brzine – ona zahteva arhitektonske odluke koje omogućavaju stalni razvoj sistema. To se može uporediti sa trkačkim automobilom visokih performansi: bez pravih projektnih osnova, početno impresivno ubrzanje ubrzo se pretvara u probleme sa upravljanjem i izazove održavanja.

U brzo razvijajućim Python aplikacijama ovo postaje naročito vidljivo. Bez jasne arhitekture, brzo dodavanje funkcionalnosti lako prerasta u zapetljanu mrežu zavisnosti. Kodna baza koja je u početku bila okretna može već posle nekoliko meseci postati kruta i podložna greškama. Programeri tada više vremena provode pokušavajući da razumeju postojeći kod nego razvijajući nove funkcionalnosti. Kada nije jasno gde tačno treba dodati novi kod niti kako ga povezati sa postojećim komponentama, rad pod pritiskom vodi do ishitrenih odluka, neoptimalnih rešenja i grešaka. Takva brza rešenja dodatno komplikuju kodnu bazu i otežavaju svaku narednu izmenu. Početni tempo razvoja postaje neodrživ, ne zbog brzine same po sebi, već zbog nedostatka stabilne arhitektonske osnove koja bi omogućila brze promene i obezbedila jasne putanje za integraciju novih funkcionalnosti.

Zahtevi se stalno menjaju – i to često nepredvidivo. Zato Python projekti moraju biti organizovani tako da omogućavaju lako prilagođavanje tim promenama. Upravo ta prilagodljivost predstavlja ključ dugoročnog uspeha u razvoju softvera.

Postizanje ravnoteže: odnos planiranja i agilnosti

U Python razvoju od presudnog je značaja pronaći pravu ravnotežu između planiranja i agilnosti. Kako je mudro primetio Dejv Tomas: „Veliko projektovanje unapred je glupost. Ne raditi nikakvo projektovanje unapred još je veća glupost." Suština je u pronalaženju sredine koja obezbeđuje i potrebnu strukturu i dovoljnu fleksibilnost.

Dobra arhitektura omogućava odlaganje odluka. Ona omogućava slobodu da se odluke pomere u kasniju fazu, kada postoji dovoljno informacija da se izabere najbolje rešenje. Ovaj pristup posebno je koristan u Python razvoju, gde fleksibilnost jezika ponekad može da dovede do paralize u odlučivanju.

Uvođenje arhitektonskog razmišljanja u Python razvoj podrazumeva razmatranje dugoročne strukture projekta od samog početka, ali bez preteranog komplikovanja. Cilj je oblikovati okvir koji usmerava razvoj, a pritom ostaje dovoljno prilagodljiv promenama.

Uloga arhitekture u upravljanju složenošću

Efikasna arhitektura predstavlja najvažniji alat za savladavanje složenosti u Python sistemima. Dobra arhitektura pojednostavljuje složene sisteme tako što obezbeđuje jasnu strukturu i **razdvajanje odgovornosti** (**SoC**). Jedan od prvih koraka pri projektovanju novog sistema jeste odluka o tome kako ga podeliti: elemente koji se menjaju iz istog razloga treba grupisati zajedno, a one koji se menjaju iz različitih razloga razdvojiti.

Zamislimo dva **sistema za upravljanje sadržajem** (**CMS**) razvijena u programskom jeziku Python za medijske kompanije, oba sa zadatkom da uvedu novu funkcionalnost – automatsko označavanje sadržaja pomoću veštačke inteligencije. U dobro projektovanom sistemu, ova funkcionalnost se uvodi kao poseban modul sa jasno definisanim interfejsima. On se nesmetano povezuje sa postojećim modulima za kreiranje sadržaja i pretragu putem jasno određenih API interfejsa. Programeri mogu da grade i testiraju servis za označavanje pomoću veštačke inteligencije nezavisno, a zatim ga povežu sa bazom sadržaja i korisničkim interfejsom uz minimalne prekide. U loše strukturiranom sistemu, međutim, dodavanje iste funkcionalnosti zahteva izmene kroz celu arhitektonsku slojevitu strukturu – od šema baze podataka do koda na strani klijenta – što često dovodi do neočekivanih grešaka i problema sa performansama. Zadatak koji u dobro projektovanom sistemu traje jedan razvojni ciklus, u loše organizovanom prerasta u projekat refaktorisanja koji traje mesecima, što jasno pokazuje koliko promišljena početna arhitektura može da poveća efikasnost razvoja i prilagodljivost sistema.

Arhitektonske odluke donete u ranoj fazi razvoja imaju presudan uticaj na dugoročne troškove razvoja i fleksibilnost projekata u programskom jeziku Python. Dobro osmišljena arhitektura može značajno da smanji cenu promena tokom vremena, omogućavajući timu da mnogo brže reaguje na nove zahteve i tehnološke promene.

Priprema za čistu arhitekturu

Kada govorimo o čistoj arhitekturi, ključno je razumeti da ona predstavlja sistematski pristup usklađivanju planiranja i agilnosti u Python projektima. Arhitektonski principi pružaju moćne mehanizme za upravljanje složenošću i njeno smanjivanje u softverskim sistemima.

Suština čiste arhitekture jeste strateško razdvajanje odgovornosti (SoC) u Python aplikacijama. Zalaže se za strukturu u kojoj je osnovna poslovna logika izolovana od spoljašnjih faktora kao što su korisnički interfejsi, baze podataka i integracije sa spoljnim sistemima. Takvo razdvajanje postavlja jasne granice između različitih delova aplikacije, pri čemu svaki deo ima sopstvenu odgovornost. Na ovaj način osnovna poslovna pravila ostaju „čista" i nezavisna od detalja implementacije mehanizama za **ulaz/izlaz** (**I/O**) ili **sistema za upravljanje podacima** (**DMS**).

Razumevanjem ovih izazova i principa bićete spremniji da sagledate prednosti koje čista arhitektura donosi Python projektima. U narednim odeljcima videćemo šta zapravo predstavlja čista arhitektura i kako se konkretno primenjuje u Python razvoju, nudeći vam alate za smanjivanje složenosti i smanjenje troškova promena u softverskim sistemima.

Šta je čista arhitektura?

Nakon što smo istražili izazove upravljanja složenošću u Python razvoju i potrebu da se planiranje uskladi sa agilnošću, cilj ovog odeljka jeste da omogući pregled čiste arhitekture na visokom nivou. Predstavićemo nekoliko ključnih koncepata i principa u kratkom vremenskom periodu, kako biste stekli osnovno razumevanje. Nema potrebe da odmah zapamtite sve detalje – ovo je samo uvod u putovanje koje sledi. Svaka od ovih tema biće kasnije razrađena u narednim poglavljima, kroz praktične primere u programskom jeziku Python i situacije iz stvarnog sveta.

Čista arhitektura objedinjuje iskustva prethodnih arhitektonskih stilova, ali se zasniva na jednom osnovnom principu: razdvajanju softverskih elemenata u koncentrične slojeve, uz strogo pravilo da zavisnosti u kodu mogu biti usmerene samo ka unutra – od spolja ka jezgru sistema. Ovo pravilo, poznato kao **pravilo zavisnosti**, jedan je od ključnih aspekata čiste arhitekture. Njegova suština je da zavisnosti u izvornom kodu moraju biti usmerene ka unutrašnjim slojevima, ka apstraktnijim pravilima i politikama. Unutrašnji slojevi ne smeju da zavise od

spoljašnjih, dok spoljašnji slojevi moraju da zavise od unutrašnjih i da se njima prilagođavaju. Na ovaj način promene spoljašnjih elemenata (poput baza podataka, korisničkih interfejsa ili radnih okvira) ne utiču na osnovnu poslovnu logiku. Cilj je izgraditi softverske sisteme koji su ne samo funkcionalni, već i dugoročno održivi i prilagodljivi. Ilustrujmo ovo jednostavnim primerom Python aplikacije za upravljanje bibliotekom:

1. U jezgru se nalazi klasa `Book`, koja predstavlja osnovnu podatkovnu strukturu.
2. U sloju iznad nalazi se klasa `BookInventory`, koja upravlja poslovnim operacijama nad knjigama.
3. U spoljašnjem sloju nalazi se klasa `BookInterface`, koja obrađuje interakciju korisnika sa knjigama.

U ovakvoj strukturi, klasa `Book` nema nikakvo znanje o klasama `BookInventory` ili `BookInterface`. Klasa `BookInventory` koristi klasu `Book`, ali ne zna ništa o interfejsu. Na taj način osnovna logika ostaje nepromenjena i nezavisna od spoljašnjih komponenti.

Ključna prednost ovakve organizacije jeste mogućnost menjanja ili čak potpune zamene spoljašnjih slojeva bez uticaja na jezgro sistema. Na primer, korisnički interfejs možemo prebaciti sa **interfejsa komandne linije** (**CLI**) na veb interfejs tako što menjamo samo klasu `BookInterface`, bez potrebe za izmenama klasa `Book` ili `BookInventory`. Upravo ta fleksibilnost predstavlja jednu od glavnih prednosti čiste arhitekture.

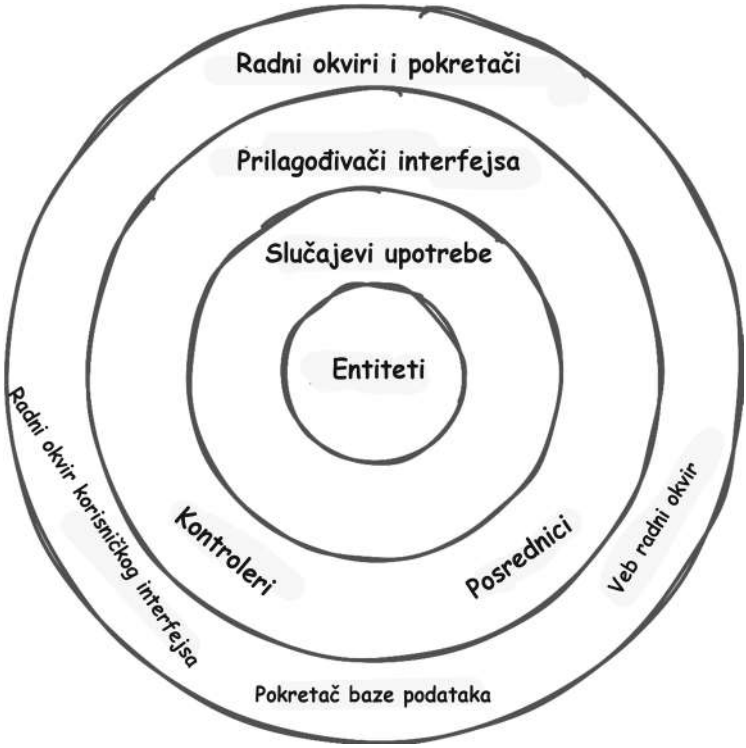
Ova struktura ima za cilj da obezbedi sisteme koji u sebi objedinjuju ključne principe pomenute ranije:

- razdvajanje odgovornosti
- nezavisnost od spoljašnjih detalja
- testabilnost i održivost

U nastavku ćemo detaljnije istražiti kako čista arhitektura postiže ove ciljeve i koje koristi donosi razvoju softvera.

Koncept slojevite arhitekture

Da bismo bolje razumeli ranije pomenute prstenaste nivoe, prikažimo ih kroz dodatni sloj detalja i svrhu svakog od njih. Čista arhitektura se često vizuelno prikazuje kao niz koncentričnih krugova – poput preseka glavice luka. Svaki krug predstavlja jedan sloj softvera, dok pravilo zavisnosti obezbeđuje da zavisnosti uvek teku samo ka unutra, ka jezgru sistema. Unutrašnji slojevi čuvaju poslovnu logiku (entitete), dok spoljašnji obuhvataju interfejse i detalje implementacije (*videti sliku 1.1*).



Slika 1.1: Čista arhitektura – niz koncentričnih slojeva

Slika 1.1 prikazuje kako se unutrašnja poslovna logika jezgra širi ka spoljnim interfejsima:

- **Entiteti:** U centru se nalaze entiteti, nosioci poslovnih pravila na nivou celog sistema. Oni predstavljaju osnovne pojmove vašeg proizvoda – ključne poslovne objekte koji bi postojali i bez softvera. Na primer, u sistemu elektronske trgovine entiteti su *Kupac*, *Proizvod* i *Narudžbina*. U aplikaciji za upravljanje zadacima to mogu biti *Korisnik*, *Zadatak* i *Projekat*. Entiteti sadrže osnovna, univerzalna pravila o ponašanju i međusobnim odnosima tih objekata.
- **Slučajevi upotrebe:** Sledeći sloj obuhvata slučajeve upotrebe, koji upravljaju tokom podataka ka i od entiteta. Slučaj upotrebe je konkretan način korišćenja sistema, opis ponašanja u određenoj situaciji. U aplikaciji za upravljanje zadacima to mogu biti *Kreiraj novi zadatak*, *Završi zadatak* ili *Dodeli zadatak*. Slučajevi upotrebe sadrže pravila poslovanja specifična za aplikaciju i kontrolišu kako i kada se entiteti koriste da bi se ostvarili ciljevi aplikacije.

- **Prilagođavači interfejsa:** Dalje ka spolja smešteni su prilagođavači interfejsa, koji prevode podatke između slučajeva upotrebe i spoljašnjih sistema. Oni služe kao prevodioci između unutrašnjih slojeva (entiteta i slučajeva upotrebe) i spoljašnjeg okruženja. Ovaj sloj obuhvata kontrolere koji obrađuju HTTP zahteve, prezentere interfejsa koji oblikuju podatke za prikaz i posrednike koji transformišu podatke za trajno čuvanje. U Python veb aplikaciji to mogu biti funkcije prikaza ili klase za usmeravanje i obradu zahteva. Suština ovog sloja jeste da sistem učini nezavisnim od konkretnih okvira.
- **Okviri i pokretači:** Spoljni sloj obuhvata okvire i pokretače – spoljne alate i mehanizme pomoću kojih sistem funkcioniše, ali koji nisu deo poslovne logike. U kontekstu programskog jezika Python, primeri su:
  - veb radni okviri kao što su Django ili Flask
  - pokretači baza podataka, npr. `psycopg2` za PostgreSQL ili `pymongo` za MongoDB
  - spoljne biblioteke za zadatke, poput slanja imejlova (`smtplib`) ili obrade plaćanja
  - okviri za korisnički interfejs, ako se razvija desktop ili mobilna aplikacija (npr. `PyQt`)
  - sistemske alatke za zapisivanje ili upravljanje konfiguracijom

Ovaj spoljni sloj je najpodložniji promenama jer predstavlja tačku kontakta sa spoljnim svetom, u kojem se tehnologije najbrže menjaju. Time što ga odvajamo od jezgra, omogućavamo da se alati i tehnologije menjaju bez narušavanja osnovne poslovne logike.

Ovakva slojevita struktura čiste arhitekture podstiče razdvajanje odgovornosti i uspostavlja jasan organizacioni okvir za softverske sisteme. Sada kada imamo jasnu predstavu o osnovnoj strukturi čiste arhitekture, možemo preći na razmatranje njenih širih prednosti.

Prednosti čiste arhitekture

Jedna od glavnih prednosti čiste arhitekture jeste njen naglasak na zaštiti i izolovanju osnovne poslovne logike, domen objekata koji čine temelj poslovanja. Dok se spoljašnji elementi, poput veb okvira i sistema za trajno čuvanje podataka, stalno smenjuju, prava vrednost leži u vremenu uloženom u osmišljavanje i izgradnju ovih ključnih objekata domena. Čista arhitektura to prepoznaje i nudi strukturu koja štiti te važne komponente od nestabilnosti spoljašnjih tehnologija.

Takav arhitektonski pristup čuva vašu investiciju u poslovnu logiku od pritiska da menjate okvir ili tehnologiju. Na primer, ako okvir koji koristite pređe sa otvorenog koda na vlasnički model, čista arhitektura vam omogućava da ga zamenite bez ponovnog pisanja osnovne poslovne logike. Ovo razdvajanje značajno smanjuje cenu i rizik promena tokom vremena, omogućavajući sistemu da se lakše prilagođava novim zahtevima ili tehnologijama uz manje poteškoća. Drugim rečima, čista arhitektura obezbeđuje da najstabilniji i najvredniji deo aplikacije – poslovna logika – ostane netaknut uprkos nestabilnom svetu okvira i tehnologija.

Druga važna prednost jeste unapređena testabilnost na svim slojevima aplikacije. Nezavisnost osnovne poslovne logike od spoljašnjih detalja čini pisanje sveobuhvatnih jedinica testova mnogo jednostavnijim. Pravila poslovanja možete proveravati u izolaciji, bez potrebe da pokrećete bazu podataka, veb server ili pravite nezgrapne imitacije. Rezultat je temeljnije testiranje i, posledično, pouzdaniji softver. Istovremeno, jednostavnost procesa podstiče programere da pišu više testova.

Čista arhitektura takođe pruža i fleksibilnost u izboru tehnologija. Pošto jezgro aplikacije ne zavisi od spoljašnjih okvira i alata, slobodni ste da ih zamenite kada je to potrebno. To je posebno dragoceno u svetu tehnologije koja se brzo menja, gde današnji popularni okvir sutra može postati zastareo. Tako, na primer, možete započeti sa interfejsom komandne linije za internu upotrebu, a kasnije dodati veb interfejs za širi krug korisnika – bez ikakvih izmena u kodu poslovnih pravila. Poslovna logika ostaje stabilna, dok se u spoljnim slojevima može uvoditi nova tehnologija. Konačno, čista arhitektura podstiče dugoročnu agilnost i vodi ka onome što Robert C. Martin naziva „*vrišteća arhitektura*“ (`https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html`). Razdvajanje odgovornosti i kontrola zavisnosti stvaraju kodnu bazu koju je lakše razumeti i menjati. Ideja „vrišteće arhitekture“ podrazumeva da struktura sistema treba jasno da pokazuje svoju svrhu i slučajeve upotrebe, a ne da ističe alate i okvire. Drugim rečima, arhitektura treba da poručuje „internet knjižara“, a ne „Django aplikacija“. Takva, svrhom usmerena struktura pomaže novim članovima tima da brzo razumeju nameru sistema i doprinesu njegovom razvoju. Arhitektura sama po sebi postaje oblik dokumentacije, jer odmah otkriva osnovnu svrhu i funkcionalnost sistema. Ova jasnoća i prilagodljivost pretvaraju se u veću brzinu razvoja na duži rok, čak i kada sistem postaje složeniji. Takođe obezbeđuje da sistem ostane usmeren na poslovnu logiku, a ne na specifične tehničke implementacije.

Čista arhitektura u kontekstu

Da bismo zaista sagledali vrednost čiste arhitekture, potrebno je razumeti njeno mesto u širem kontekstu razvojnih praksi i metodologija.

Čista arhitektura predstavlja korak napred u odnosu na tradicionalne slojevite arhitekture. Iako se oslanja na ideju slojeva, snažnije naglašava razdvajanje odgovornosti i strože sprovodi pravilo zavisnosti. Dok u tradicionalnim slojevitim arhitekturama donji slojevi često zavise od baze podataka ili infrastrukture, čista arhitektura održava unutrašnje slojeve čistim i usmerenim na poslovnu logiku. Ovakav pomak u fokusu donosi veću fleksibilnost i otpornost na promene.

Čista arhitektura dopunjuje savremene razvojne prakse kao što su agilni pristupi i DevOps. Podržava agilne metodologije time što omogućava **neprekidnu isporuku** (**CD**) i olakšava prilagođavanje promenama. Jasno razdvajanje odgovornosti pogoduje iterativnom razvoju i olakšava izmene ili proširenja funkcionalnosti kada to zahtevaju nove okolnosti. U domenu DevOps prakse, čista arhitektura olakšava **neprekidnu integraciju i isporuku** (**CI/CD**) tako što čini sisteme testabilnijim i modularnijim. Jasno postavljene granice između komponenti omogućavaju timovima da paralelno rade na različitim delovima sistema bez međusobnog ometanja.

U celini, čista arhitektura nudi snažan pristup izgradnji softverskih sistema koji su skalabilni, održivi i prilagodljivi promenama. Naglašavanjem razdvajanja odgovornosti i kontrolom zavisnosti, pruža strukturu koja odoleva vremenu i pritiscima stalnog razvoja tehnologije i poslovnih potreba. U nastavku ćemo videti kako se ovi principi naročito dobro uklapaju u razvojne Python prakse.

Čista arhitektura i programski jezik Python – prirodan spoj

Nakon što smo istražili principe i prednosti čiste arhitekture, prirodno se nameće pitanje koliko se ti koncepti uklapaju u razvoj u programskom jeziku Python. Ovaj odeljak pokazuje da čista arhitektura i programski jezik Python imaju prirodnu povezanost, što Python čini izuzetno pogodnim za implementaciju njenih principa.

Filozofija programskog jezika Python, izražena u *Zen of Python* (`https://peps.python.org/pep-0020/`), u velikoj meri se podudara sa principima čiste arhitekture. I jedno i drugo stavljaju naglasak na jednostavnost, čitljivost i značaj dobro organizovanog koda. Usmerenost Pythona na jasan, održiv i prilagodljiv kod pruža snažnu osnovu za primenu čiste arhitekture. U nastavku ćemo videti kako se osobine ovog jezika mogu iskoristiti za izgradnju robusnih i dugoročno održivih sistema zasnovanih na njenim principima.

Primena čiste arhitekture u programskom jeziku Python

Dinamična priroda programskog jezika Python, uz snažnu podršku i za **objektno-orijentisano programiranje** (**OOP**) i za funkcionalne paradigme, omogućava programerima da primene principe čiste arhitekture uz manje šablonskog koda i sa većom jasnoćom nego u mnogim drugim jezicima.

> **Napomena o primerima koda**
>
> U ovoj knjizi koristićemo napomene o tipovima u primerima koda (npr. `def function(parameter: type) -> return_type`). One unapređuju čitljivost i pomažu u jasnom sprovođenju granica čiste arhitekture. O ovoj mogućnosti detaljno ćemo govoriti u trećem poglavlju.

Jedan od ključnih principa čiste arhitekture jeste oslanjanje na apstrakcije umesto na konkretne implementacije. Time se direktno podržava pravilo zavisnosti: zavisnosti treba uvek da idu ka unutrašnjim slojevima. Pogledajmo kako ovo izgleda u praksi uz pomoć **apstraktnih osnovnih klasa** (**ABCs**) u programskom jeziku Python.

Kao primer, koristićemo model jednostavnog sistema obaveštavanja:

```
from abc import ABC, abstractmethod


class Notifier(ABC):
    @abstractmethod
    def send_notification(self, message: str) -> None:
        pass

class EmailNotifier(Notifier):
    def send_notification(self, message: str) -> None:
        print(f"Sending email: {message}")

class SMSNotifier(Notifier):
    def send_notification(self, message: str) -> None:
        print(f"Šalje SMS: {message}")

class NotificationService:
    def __init__(self, notifier: Notifier):
        self.notifier = notifier
    def notify(self, message: str) -> None:
        self.notifier.send_notification(message)
```

```
# Korišćenje
email_notifier = EmailNotifier()
email_service = NotificationService(email_notifier)
email_service.notify("Zdravo putem imejla")
```

Primer ilustruje ključne pojmove čiste arhitekture kroz Python apstraktne osnovne klase:

1. **Apstraktna osnovna klasa:** Klasa `Notifier` definiše interfejs koji sve klase obaveštavača moraju da slede. Ona predstavlja unutrašnji sloj u strukturi čiste arhitekture.
2. **Apstraktan metod:** Metod `send_notification` u klasi `Notifier` označen je dekoratorom `@abstractmethod`, što znači da ga potklase moraju implementirati.
3. **Konkretne implementacije:** `EmailNotifier` i `SMSNotifier` su spoljašnje klase koje nasleđuju `Notifier` i pružaju konkretne implementacije.
4. **Obrtanje zavisnosti:** `NotificationService` zavisi od apstraktne klase `Notifier`, a ne od konkretnih implementacija. Ovo je u skladu sa pravilom zavisnosti, jer unutrašnji sloj (`Notifier`) ne zavisi od spoljašnjih implementacija.

Inverziju zavisnosti detaljnije ćemo razraditi u sledećem poglavlju.

Ova struktura oslikava principe čiste arhitekture o kojima smo govorili:

- **Poštuje pravilo zavisnosti**: apstraktna klasa Notifier (unutrašnji sloj) ne zna ništa o konkretnim obaveštavačima niti o klasi NotificationService (spoljašnji slojevi).
- **Omogućava lako proširenje**: nove vrste obaveštavača (na primer PushNotifier) mogu se dodati bez ikakvih izmena u klasi NotificationService.
- **Pruža fleksibilnost i održivost**: osnovna poslovna logika (slanje obaveštenja) jasno je razdvojena od načina implementacije (kako se obaveštenje zapravo šalje).

Na ovaj način kod postaje ne samo fleksibilan i dugoročno održiv, već i potpuno usklađen sa osnovnim principima čiste arhitekture. Apstraktna klasa `Notifier` predstavlja osnovna poslovna pravila, dok konkretni obaveštavači i klasa `NotificationService` pripadaju spoljnim slojevima sklonim promenama. Takvo razdvajanje omogućava jednostavno dodavanje ili zamenu metoda obaveštavanja bez uticaja na osnovnu logiku aplikacije.

Do sada smo videli jednostavan primer apstraktne osnovne klase, ali upravo ovde programski jezik Python pokazuje svoju punu snagu. Iste principe čiste arhitekture moguće je primeniti i bez hijerarhije klasa, oslanjajući se na podršku programskog jezika Python za tipiziranje na osnovu ponašanja (https://en.wikipedia.org/wiki/Duck_typing). Ta fleksibilnost je jedno od najvećih bogatstava ovog jezika, jer programerima omogućava slobodu da izaberu pristup koji najbolje odgovara potrebama projekta, a da se i dalje ostane u okvirima principa čiste arhitekture.

Tipiziranje na osnovu ponašanja je koncept u kojem se podobnost objekta određuje time da li poseduje tražene metode ili svojstva, a ne time kojem tipu pripada. Naziv potiče od izreke: „Ako hoda kao patka i kvače kao patka, onda je to patka." Kod *tipiziranja na osnovu ponašanja* tip objekta je sporedan; bitno je samo da li obavlja ono što se od njega zahteva.

Takav pristup se prirodno uklapa u naglasak čiste arhitekture na apstrakcijama i interfejsima. Ako želite da izbegnete krute hijerarhije klasa, funkcionalnost *Protocol*, predstavljena u verziji Python 3.8 (https://peps.python.org/pep-0544/), nudi najbolje iz oba sveta: slobodu tipiziranja na osnovu ponašanja uz prednost napomena o tipovima. Slede primeri koji pokazuju kako se isti sistem obaveštavanja može implementirati pomoću protokola:

```
from typing import Protocol
class Notifier(Protocol):
    def send_notification(self, message: str) -> None:
        ...

class EmailNotifier: # Note: no explicit inheritance
    def send_notification(self, message: str) -> None:
        print(f"Šaljem imejl: {message}")

class SMSNotifier: # Primedba: nema eksplicitnog nasleđivanja
    def send_notification(self, message: str) -> None:
        print(f"Šaljem SMS: {message}")

class NotificationService:
    # Sposobno da pogodi tip
    def __init__(self, notifier: Notifier):
        self.notifier = notifier

    def notify(self, message: str) -> None:
        self.notifier.send_notification(message)
```

```
# Korišćenje
sms_notifier = SMSNotifier()
sms_service = NotificationService(sms_notifier)
sms_service.notify("Zdravo putem SMS")
```

Ovaj primer prikazuje isti sistem obaveštavanja kao i ranije, ali sada uz funkcionalnost `Protocol` u programskom jeziku Python, umesto apstraktnih osnovnih klasa. Objasnimo detaljno ključne razlike i šta znače za čistu arhitekturu:

- **Protocol naspram apstraktneosnovne klase:** Klasa `Notifier` je sada `Protocol`, a ne *apstaktna osnovna* klasa. Ona uvodi interfejs zasnovan na strukturalnoj podtipizaciji, umesto da nameće eksplicitno nasleđivanje.
- **Implicitna usklađenost:** Klase `EmailNotifier` i `SMSNotifier` ne nasleđuju eksplicitno `Notifier`, ali ispunjavaju njegov interfejs time što implementiraju metod `send_notification`.
- **Tipiziranje na osnovu ponašanja sa napomenama o tipovima:** Ovaj pristup spaja fleksibilnost tipiziranja na osnovu ponašanja programskog jezika Python sa koristima statičke provere tipova, u skladu sa težnjom čiste arhitekture ka slaboj sprezi.
- **Konkretne implementacije:** Klasa `NotificationService` i dalje zavisi od apstraktnog protokola `Notifier`, a ne od konkretnih klasa, pa se principi čiste arhitekture dosledno poštuju.

Pristup zasnovan na protokolima pruža fleksibilnu implementaciju u duhu programskog jezika Python, postižući ravnotežu između bezbednosti tipova i manje krutih hijerarhija klasa. Time se pokazuje kako uskladiti principe čiste arhitekture sa filozofijom programskog jezika Python i podstaknuti kod koji je prilagodljiv i održiv.

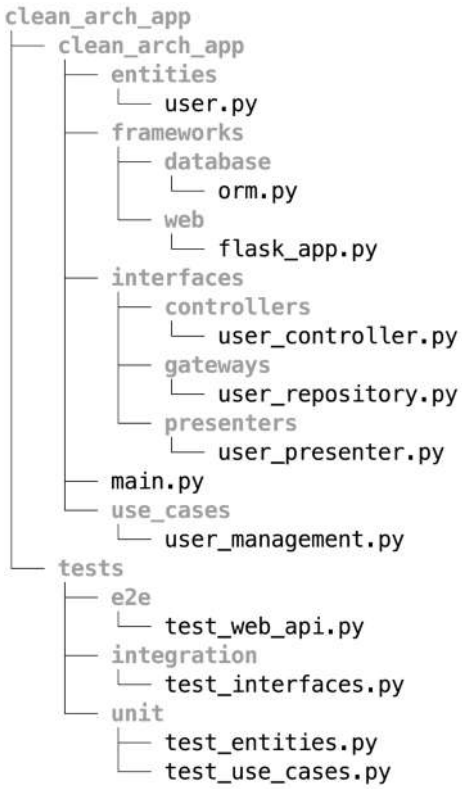
Preporučujemo napomene o tipovima, bilo preko apstraktnih osnovnih klasa ili protokola, pri primeni čiste arhitekture. Takav pristup, za razliku od čistih implicitnih interfejsa bez napomena o tipovima, donosi važne prednosti:

- Jasniji i čitljiviji kod
- Jača podrška u integrisanim razvojnim okruženjima i ranije uočavanje grešaka
- Bolja usklađenost sa ciljevima čiste arhitekture

U nastavku knjige uglavnom ćemo koristiti apstraktne osnovne klase u primerima, jer su češće prisutne u postojećim Python kodnim bazama. Ipak, principi su podjednako primenljivi i na rešenja zasnovana na protokolima, pa čitaoci mogu po potrebi prilagoditi primere da koriste protokole.

Praktičan primer: pogled na čistu arhitekturu u Python projektu

Da bismo ilustrovali koncepte o kojima smo govorili, hajde da ispitamo osnovnu strukturu Python projekta zasnovanog na čistoj arhitekturi. Ova struktura sadrži principe koje smo obradili i pokazuje kako se oni prevode u praktičnu organizaciju datoteka. Ovde ćemo ostati na visokom nivou; u narednim poglavljima detaljno ćemo obrađivati konkretne primere.



Slika 1.2: Mogući raspored za Python veb aplikaciju zasnovanu na čistoj arhitekturi

Ova struktura datoteka na slici 1.2 ilustruje principe čiste arhitekture o kojima smo govorili:

1. **Razdvajanje odgovornosti:** Svaki direktorijum predstavlja poseban sloj aplikacije, u skladu sa koncentričnim krugovima prikazanim na *slici 1.1*.
2. **Pravilo zavisnosti:** Ova struktura sprovodi pravilo zavisnosti o kojem je ranije bilo reči. Kada bismo analizirali unutrašnje slojeve (entitete i slučajeve upotrebe), ne bismo naišli ni na jedan uvoz iz spoljašnjih slojeva.

3. **Sloj entiteta:** Direktorijum `entities` sadrži osnovne poslovne objekte, kao što je `user.py`. Oni se nalaze u samom središtu dijagrama čiste arhitekture i potpuno su nezavisni od spoljašnjih slojeva.
4. **Sloj slučajeva upotrebe:** Direktorijum `use_cases` sadrži pravila poslovne logike specifična za aplikaciju. On se oslanja na entitete, ali je nezavisan od spoljašnjih slojeva.
5. **Sloj prilagođivača interfejsa:** Direktorijum `interfaces` obuhvata kontrolere, prezentere i posrednike. Njihova uloga je da prilagođavaju podatke između slučajeva upotrebe i spoljašnjih sistema (na primer, veb okvira ili baza podataka).
6. **Sloj okvira:** Spoljašnji direktorijum `frameworks` sadrži implementacije spoljašnjih interfejsa, kao što su **objektno-relaciona mapiranja** (**ORM**) za baze podataka ili veb okviri.
7. **Jednostavno testiranje:** Struktura direktorijuma `tests` odgovara strukturi aplikacije, što omogućava potpunu pokrivenost testiranjem na svim nivoima.

Ovakva organizacija donosi ključne prednosti čiste arhitekture o kojima smo ranije govorili:

- **Održivost:** Promene na spoljnim komponentama (u direktorijumu `frameworks`) ne utiču na osnovnu poslovnu logiku u entitetima i slučajevima upotrebe.
- **Fleksibilnost:** Bazu podataka ili veb okvir u direktorijumu `frameworks` možemo zameniti bez diranja poslovne logike.
- **Testabilnost:** Jasno razgraničeni slojevi omogućavaju jednostavno jedinično testiranje osnovnih komponenti i integraciono testiranje interfejsa.

Sećate se razgovora o apstrakcijama? Direktorijum `interfaces` upravo je mesto gde bismo implementirali apstraktne osnovne klase ili protokole o kojima smo ranije govorili. Na primer, datoteka `user_repository.py` može da definiše apstraktnu klasu `UserRepository`, koja se zatim konkretno implementira u datoteci `frameworks/database/orm.py`.

Ovakva struktura ujedno podržava i *glavni plan* o kom je ranije bilo reči. Ona pruža jasan putokaz gde treba dodati novi kod, pomažući programerima da donose dosledne odluke i u situacijama kada projekat raste i postaje složeniji.

Organizovanjem Python projekta na ovaj način, obezbeđujemo temelje za dugoročni uspeh, stvarajući kodnu bazu koja je ne samo funkcionalna, već i održiva, fleksibilna i u skladu sa principima čiste arhitekture.

Specifičnosti programskog jezika Python i potencijalne zamke

Iako su čista arhitektura i Python veoma usklađeni, ipak postoje važna pitanja na koja treba obratiti pažnju kada se ovi principi primenjuju u Python projektima. Tokom čitave knjige pokazaćemo vam kako da umanjite te izazove, uz praktična rešenja i preporučene postupke.

Usklađivanje Python koda sa arhitektonskim principima

Filozofija „priključenih baterija" i bogata standardna biblioteka programskog jezika Python ponekad mogu prevariti programere da preskoče arhitektonske granice radi pogodnosti. Ipak, očuvanje čiste arhitekture često zahteva uvođenje apstrakcija čak i oko funkcija standardne biblioteke, kako bi se zadržalo razdvajanje odgovornosti. Na primer, umesto da u slučajevima upotrebe direktno primenjujete biblioteku `smtplib`, bolje je kreirati sloj apstrakcije za slanje obaveštenja.

Uz ovu knjigu, uverićete se da se uloženi napor u pravljenje apstrakcija isplati kroz održivost, prilagodljivost i mogućnost testiranja. Pokazaćemo da početna ulaganja u principe čiste arhitekture donose značajne dugoročne prednosti.

Jednostavan način na koji Python omogućava uvoz modula ponekad može stvoriti haotične strukture zavisnosti, jer su svi paketi u suštini javni. Pokazaćemo vam kako da dosledno primenjujete pravilo zavisnosti, tako da unutrašnji slojevi ne zavise od spoljašnjih. U *poglavlju 2* predstavićemo vam tehnike i alate koji pomažu u očuvanju uredne strukture zavisnosti u Python projektima.

Skaliranje čiste arhitekture u Python projektima

Primena principa čiste arhitekture treba da bude usklađena sa veličinom i složenošću Python projekta.

U malim projektima ili kod brzih prototipova sasvim je prihvatljivo imati jednostavnu, monolitnu arhitekturu. Ipak, čak i tada, smislena modularna izgradnja može stvoriti temelj za budući razvoj. Početna struktura može izgledati ovako:

```
small_project
├── main.py
├── models.py
├── utils.py
└── views.py
```

Slika 1.3: Kratak prototip Python projekta

Čak i u ovako malom projektu moguće je primeniti principe čiste arhitekture kroz:

- razdvajanje poslovne logike u `models.py` od logike prikaza `u views.py`,
- **ubrizgavanje zavisnosti** (**DI**) radi modularnosti i lakšeg testiranja komponenti,
- jasno definisane interfejse između modula.

Kako projekat raste, arhitektura se može postepeno razvijati u potpuniji oblik. Taj razvoj obuhvata:

1. izdvajanje osnovne poslovne logike (entiteta i slučajeva upotrebe) u posebne module,
2. uvođenje interfejsa za apstrahovanje koda vezanog za konkretan okvir,
3. organizovanje testova u skladu sa arhitektonskim slojevima.

Ova knjiga nudi praktičan pristup, počevši od jednostavne aplikacije i pragmatične primene principa čiste arhitekture. Kako budemo napredovali, primeri će biti složeniji, pa ćete shvatiti kako se pristup čiste arhitekture razvija zajedno sa kodnom bazom.

Čista arhitektura nije „sve ili ništa“, već širok spektar mogućnosti. Obrasci koje ćemo proučavati predstavljaju potpunu implementaciju osmišljenu da pokaže sve mogućnosti ovog pristupa, ali u stvarnosti možete primeniti samo one koji donose pravu vrednost u vašem kontekstu. Mala API aplikacija, na primer, može imati koristi od čistih obrazaca kontrolera bez potrebe za potpunim apstrakcijama sloja prikaza, dok skript za obradu podataka može koristiti entitete oblasti i ne koristiti prilagođivače interfejsa. Naučićete da te principe primenjujete smisleno — da izbegavate nepotrebno komplikovanje u manjim projektima, a da u većim sistemima koristite pun potencijal čiste arhitekture. Suština je da razumete šta svaki obrazac donosi, kako biste mogli da donesete pravu odluku o tome koje arhitektonske granice su za vaš projekat najvažnije.

Pravilno iskorišćavanje dinamične prirode programskog jezika Python

Dinamična priroda programskog jezika Python veoma je moćna, ali može izazvati probleme ako se ne koristi pažljivo. *Poglavlje 3* bavi se ovim osobinama jezika, kao što su tipiziranje na osnovu ponašanja, upotreba napomena o tipovima i savremenije mogućnosti, poput protokola. Do kraja poglavlja imaćete jasnu osnovu kako da najbolje primenite te osobine jezika u službi čiste arhitekture, kombinujući fleksibilnost programskog jezika Python sa potrebnom arhitektonskom disciplinom.

Razmatranja o testiranju

Kao i sama čista arhitektura, ova knjiga snažno podstiče upotrebu testova. Testovi se posmatraju kao ravnopravni klijenti aplikacionog koda — koriste kodnu bazu i proveravaju rezultate. Ista arhitektonska pravila koja važe za glavni kod odnose se i na Python testove.

Uputićemo vas kako da pišete testove koji poštuju arhitektonske granice. Naučićete da prepoznate kada testovi signaliziraju moguće probleme u arhitekturi — na primer, kada zahtevaju previše pripreme ili prekomerno oslanjanje na imitacije. U testnim primerima svakog poglavlja, a posebno u osmom poglavlju, detaljno ćemo obraditi te koncepte, pokazujući kako testovi mogu poslužiti za proveru ispravnosti, ali i kao sredstvo za očuvanje i unapređenje arhitekture.

Ako ste svesni ovih pitanja i potencijalnih zamki i sledite smernice iz knjige, moći ćete da gradite Python sisteme koji su istovremeno čisti i praktični, pomoću prednosti i čiste arhitekture i samog jezika. Suština je da principe primenjujete smisleno, uvek sa ciljem stvaranja održivog, testabilnog i fleksibilnog Python koda.

Rezime

U ovom poglavlju uveli smo čistu arhitekturu i govorili o njenom značaju za razvoj u programskom jeziku Python. Prikazali smo širi kontekst kroz evoluciju softverske arhitekture — od Vodopad modela do Agilnog pristupa — naglašavajući stalne izazove: upravljanje složenošću, prilagođavanje promenama i očuvanje dugoročne produktivnosti.

Predstavili smo ključne principe čiste arhitekture:

- razdvajanje odgovornosti (SoC)
- nezavisnost od spoljašnjih detalja
- testabilnost i održivost

Obradili smo osnovnu strukturu čiste arhitekture — od centralnih entiteta i slučajeva upotrebe do spoljašnjih slojeva prilagođavača interfejsa, radnih okvira i pokretača — naglašavajući kako ta organizacija doprinosi održivosti i fleksibilnosti. Istakli smo prednosti čiste arhitekture, poput bolje prilagodljivosti, veće testabilnosti i dugoročne agilnosti razvoja, kao i njenu povezanost sa savremenim praksama kao što su Agilni pristup i DevOps.

Zatim smo pokazali prirodno slaganje čiste arhitekture sa programskim jezikom Python i razjasnili kako se njegove osobine mogu iskoristiti za uspešnu primenu ovog pristupa. Takođe smo ukazali na specifična pitanja i moguće zamke, te naglasili potrebu da se održava ravnoteža između Python koda i arhitektonskih principa, uz prilagođavanje veličini i složenosti projekta.

U ovom poglavlju postavili smo glavne ciljeve čiste arhitekture i pokazali njeno prirodno uklapanje u Python razvoj. Videli smo kako se principi, poput apstraktnih osnovnih klasa i protokola, mogu primeniti kroz mogućnosti programskog jezika Python, što pruža osnove za stvaranje održivih i fleksibilnih sistema.

U narednom poglavlju nadovezaćemo se na ovu osnovu i detaljno obraditi SOLID principe. Oni predstavljaju osnovu čiste arhitekture i biće objašnjeni kroz praktične primere u programskom jeziku Python, pokazujući kako doprinose stvaranju robusnog i proširivog projektovanja aplikacija.

Dodatna literatura

Za više informacija o o temi obrađenoj u ovom poglavlju pogledajte sledeće materijale:

- Clean Architecture: A Craftsman's Guide to Software Structure and Design autora Roberta C. Martina. Ova knjiga predstavlja sveobuhvatan pregled čiste arhitekture od svog tvorca.
- Domain-Driven Design: Tackling Complexity in the Heart of Software autora Erika Evansa. Iako nije specijalno o čistoj arhitekturi, ova knjiga daje vredne uvide u projektovanje softvera u poslovnim domenima.
- Agile Software Development, Principles, Patterns, and Practices autora Roberta C. Martina. Ova knjiga pokriva mnoge principe čiste arhitekture u kontekstu Agilnog razvoja.
- The Pragmatic Programmer: Your Journey to Mastery autora Endrju Hanta i Dejvida Tomasa. Ova knjiga daje praktične savete o projektovanju softvera i razvoju koje dobro pokrivaju principe čiste arhitekture.