Arhitektura testabilnog koda i napredne tehnike izolacije (Mocking)

Uvod i problem čvrsto spregnutog koda (Slajdovi 1–5)

[Slajd 1]

Zdravo svima i dobrodošli u drugu temu našeg kursa. Danas se bavimo jednim od najvažnijih temelja profesionalnog razvoja softvera — arhitekturom testabilnog koda i naprednim tehnikama izolacije. Naučićemo kako da napišemo kod koji se uopšte može lako testirati, upoznaćemo pet vrsta testnih dvojnika, a videćemo i gde se krije zamka preteranog mock-ovanja.

[Slajd 2]

Naša agenda za danas prati logičan tok: prvo ćemo identifikovati problem čvrsto spregnutog koda. Zatim ćemo ga rešiti kroz principe dizajna za testabilnost, kao što su SOLID i ubrizgavanje zavisnosti. Nakon toga upoznajemo svih pet vrsta dvojnika — od Dummy-ja do Fake-a — primenjujemo ih u Javi sa JUnit 5 i Javalin kontrolerom, i na kraju pričamo o rizicima krhkih testova.

[Slajd 3]

Krenimo od samog problema. Šta je zapravo čvrsto spregnut kod? To je situacija kada jedna komponenta u našem sistemu direktno zavisi od konkretne implementacije druge komponente. Zamislite da naš `KorisnikServis`, koji treba da sadrži čistu poslovnu logiku, direktno kreira instancu konkretne baze podataka. Time smo poslovnu logiku zalepili za infrastrukturu i više je ne možemo testirati u izolaciji.

[Slajd 4]

Pogledajte ovaj primer loše prakse u Javi. Unutar klase `KorisnikServis` stoji `new UkrstenaBazaPodataka()`. Ovde postoje dva velika problema: servis direktno poznaje konkretnu implementaciju baze, i što je još gore, nema nikakvog ulaza spolja gde bismo u testu mogli da podmetnemo lažnu komponentu.

[Slajd 5]

Zašto je ovakav kod noćna mora za testiranje? Prvo, gubimo izolaciju — ako baza nije podignuta, naš unit test pada iako je biznis logika savršena. Drugo, brzina — komunikacija sa pravom bazom usporava testove sa milisekundi na sekunde ili minute. Treće, nuspojave — testovi mogu da upisuju i menjaju stvarne podatke, pa jedan test počne da ruši drugi.

Dizajn za testabilnost — DfT (Slajdovi 6–10)

[Slajd 6]

Rešenje je koncept koji nazivamo **Dizajn za testabilnost**. To znači da mogućnost lakog testiranja ne posmatramo kao nešto što dolazi na kraju, već kao primarni arhitektonski zahtev od prvog dana. Ovde važi zlatno pravilo softverskog inženjerstva: ***kod koji se lako***

testira je gotovo uvek i dobro dizajniran kod. Do njega stižemo kroz SOLID principe i inverziju kontrole.

[Slajd 7]

Ako pogledamo SOLID principe kroz prizmu testiranja, tri se posebno izdvajaju. Princip jedne odgovornosti znači da klasa radi samo jednu stvar, pa imamo mnogo manje scenarija za proveru. Open/Closed princip nam čuva postojeće testove netaknutim kada dodajemo nove funkcionalnosti. A zvezda današnje priče je slovo D — ***Dependency Inversion Principle***, koji nam direktno omogućava da pravu zavisnost u testu zamenimo dvojnikom.

[Slajd 8]

Šta kaže Dependency Inversion Principle? Moduli višeg nivoa, poput naše poslovne logike, ne smeju da zavise od modula nižeg nivoa, kao što je baza podataka. Oba treba da zavise od apstrakcija. Dakle, `KorisnikServis` više ne zna za `UkrstenuBazuPodataka`, već komunicira isključivo sa interfejsom `KorisnikRepozitorijum`.

[Slajd 9]

Da bismo to sproveli u delo, koristimo **Inverziju kontrole** i **Dependency Injection**. Klasi oduzimamo pravo da sama kreira svoje zavisnosti pomoću ključne reči `new`. Umesto toga, zavisnosti joj ubrizgavamo spolja — najčešće kroz konstruktor. U produkciji ćemo kroz taj konstruktor proslediti pravu bazu, a u testu ćemo na isto to mesto ubaciti dvojnika.

[Slajd 10]

Evo kako izgleda naš refaktorisani kod. Prvo definišemo čist interfejs `KorisnikRepozitorijum`. Zatim u klasi `KorisnikServis` zavisnost primamo kroz konstruktor. Pogledajte koliko je ova klasa sada čista: ona ne zna da li iza interfejsa stoji Oracle baza, fajl na disku ili obična mapa u memoriji. Upravo smo otvorili vrata za testiranje u potpunoj izolaciji.

Pet vrsta Test Dvojnika (Slajdovi 11–20)

[Slajd 11]

Sada dolazimo do drugog dela predavanja — **Test Doubles**, odnosno testnih dvojnika. Zamislite ih kao kaskadere na filmu: kada je scena previše opasna ili skupa za glavnog glumca, uskače kaskader. Isto tako, u testovima zamenjujemo spore ili skupe produkcione komponente dvojnicima. Prema čuvenoj podeli Gerarda Meszarosa, razlikujemo pet tipova dvojnika.

[Slajd 12]

Evo brzog pregleda svih pet pre nego što uđemo u detalje: **Dummy** samo popunjava parametar i nikada se ne poziva. **Stub** vraća unapred spremljene odgovore. **Spy** prati i beleži kako je pozvan. **Mock** ima unapred programirana očekivanja i sam proverava da li su ispunjena. I konačno, **Fake** je pojednostavljena, ali stvarna funkcionalna implementacija.

[Slajd 13]

Krenimo od prvog — **Dummy** objekta. Njega koristimo kada konstruktor ili metoda zahtevaju neki parametar, ali znamo da se taj objekat uopšte neće koristiti u konkretnom test scenariju.

Pogledajte primer: testiramo da li registracija baca izuzetak kada je ime korisnika prazno. Validacija imena puca na samom početku, pre nego što servis ikada pokuša da snimi korisnika ili pošalje mejl. Zato za repozitorijum prosleđujemo `null`, a za email servis lambda izraz koji pada ako se slučajno pozove.

[Slajd 14]

Drugi tip je **Stub**. Njegov jedini zadatak je da nam vrati unapred definisan odgovor kako bi naš kod pod testom skrenuo u određenu granu izvršavanja. Stub ne pamti koliko puta je pozvan. U ovom primeru pravimo `KorisnikRepozitorijumStub` čija metoda `postojiPoEmailu` uvek vraća `true`.

[Slajd 15]

A evo kako taj Stub koristimo u JUnit testu. Želimo da proverimo da li registracija propada ako email adresa već postoji u sistemu. Ubacimo naš Stub koji uvek tvrdi da email postoji, pozovemo `registrujKorisnika` i pomoću `assertThrows` potvrdimo da servis baca `IllegalStateException`. Jednostavno, brzo i bez ikakve baze.

[Slajd 16]

Treći tip je **Spy**, odnosno špijun. Za razliku od Stub-a, Spy pamti istoriju poziva: koje metode su pozvane, koliko puta i sa kojim tačno argumentima. U našem primeru `EmailServisSpy` ima brojač `brojPoziva` i promenljivu `poslednjiEmail` u koju zapisuje adresu na koju je mejl poslat.

[Slajd 17]

U testu zatim proveravamo te zabeležene podatke. Nakon što uspešno registrujemo korisnika Marka, pomoću `assertEquals` pitamo našeg špijuna: da li je `brojPoziva` jednak jedan, i da li je `poslednjiEmail` zaista `"marko@email.com"`?

[Slajd 18]

Četvrti tip je **Mock**. Studenti često mešaju Spy i Mock, ali razlika je suštinska. Kod Spy-ja test na kraju sam ručno proverava zabeleženo stanje. Kod Mock-a mi pre izvršavanja testa programiramo očekivanja u samom objektu, i na kraju pozivamo metodu za verifikaciju na tom Mock-u.

[Slajd 19]

Pogledajte ručnu implementaciju Mock-a u Javi. Klasa `KorisnikRepozitorijumMock` beleži da li je pozvana metoda `sacuvaj`, ali sadrži i metodu `verifikuj()` koja sama baca `AssertionError` ako očekivanje nije ispunjeno. U praksi ovo za nas rade biblioteke poput Mockito-a, ali je važno da razumete mehanizam ispod haube.

[Slajd 20]

Peti i poslednji dvojnik je **Fake**. Fake nije samo prazna ljuštura — on ima stvarnu, funkcionalnu poslovnu logiku, ali koristi prečicu koja nije za produkciju. Najbolji primer je `FakeKorisnikRepozitorijum` koji umesto prave SQL baze koristi običnu `HashMap`-u u memoriji. On zaista čuva, pronalazi i broji korisnike tokom testa, ali se podaci brišu čim se test završi.

Verifikacija stanja i ponašanja (Slajdovi 21–23)

[Slajd 21]

Ova podela dvojnika nas dovodi do dva fundamentalna stila pisanja testova. Prvi je **State Verification**, ili verifikacija stanja. Ovde nas zanima ***ŠTA*** je krajnji rezultat operacije. Posle izvršene metode proveravamo povratnu vrednost ili stanje u našoj `Fake` bazi — na primer, da li se broj korisnika povećao na jedan.

[Slajd 22]

Drugi stil je **Behavior Verification**, ili verifikacija ponašanja. Ovde nas ne zanima krajnje stanje, već ***KAKO*** je rezultat postignut — odnosno, da li je naš kod ispravno komunicirao sa svojim spoljnim zavisnostima. Tipičan primer je provera da li je `EmailServis` pozvan tačno jednom sa odgovarajućom adresom.

[Slajd 23]

Na ovom slajdu imate jasan rezime ove dve filozofije. Zapamtite: **State Verification** proverava krajnji rezultat i ide ruku pod ruku sa Stub-ovima i Fake-ovima. **Behavior Verification** proverava interakciju tokom operacije i koristi Mock-ove i Spy-eve.

Praktična realizacija u Javalinu (Slajdovi 24–28)

[Slajd 24]

Hajde sada da sve ovo smestimo u realan Java projekat. Koristimo Maven sa jasnom podelom paketa na model, repozitorijum, servis i kontroler, uz Java 17, Javalin framework i JUnit 5 za pokretanje testova.

[Slajd 25]

Pogledajte arhitekturu našeg web sloja. HTTP zahtev stiže u `KorisnikKontroler`. Uloga kontrolera je strogo ograničena: on treba samo da pročita zahtev, delegira posao `KorisnikServis`-u i upakuje HTTP odgovor. Kontroler nikada ne sme da sadrži poslovnu logiku niti da direktno priča sa bazom.

[Slajd 26]

Evo kako izgleda dobro dizajniran Javalin kontroler. Ponovo primenjujemo Constructor Injection — `KorisnikServis` ubrizgavamo spolja. Metoda `registracijaRuta` parsira JSON telo zahteva, poziva servis i vraća status `201 Created`. Ako servis baci grešku validacije, kontroler je hvata i pretvara u HTTP `400`, a ako email već postoji, vraća status `409 Conflict`.

[Slajd 27]

Sada se postavlja pitanje: kako da testiramo ovaj kontroler u izolaciji, a da ne moramo da podižemo ceo Javalin web server na portu 8080? Rešenje je da izolujemo Javalin-ov `Context` objekat. Pošto kontroler kroz `Context` čita zahtev i upisuje status odgovora, mi možemo napraviti sopstvenog dvojnika za `Context`.

[Slajd 28]

Pogledajte ovaj test. Napravimo servis sa `Fake` repozitorijumom, ubacimo ga u kontroler, a zatim kreiramo naš ručni `ContextMock`, ubacimo mu lažni JSON string kao telo zahteva i pozovemo metodu kontrolera. Na kraju samo proverimo da li je status postavljen na 201. Celo testiranje web sloja izvršava se u deliću milisekunde, bez otvaranja mrežnih portova.

Over-mocking i krhki testovi (Slajdovi 29–33)

[Slajd 29]
Za sam kraj, moramo da razgovaramo o tamnoj strani dvojnika, a to je **Over-mocking antipatern**. Kada programeri otkriju Mock-ove, često počnu da mock-uju apsolutno svaku zavisnost i pišu prekomerno detaljna očekivanja. Time pravimo fatalnu grešku: testiramo ***kako*** je kod napisan iznutra, umesto ***šta*** on zaista radi.
[Slajd 30]
Posledice preteranog mock-ovanja su veoma opasne. Može vam se desiti da imate 100% pokrivenost koda i da svi unit testovi zelene, a da aplikacija padne čim je pokrenete jer ste mock-ovali saradnju između komponenata na pogrešan način. Visoka pokrivenost u tom slučaju daje samo lažan osećaj sigurnosti.
[Slajd 31]
Direktan rezultat over-mocking-a su takozvani **Fragile Tests**, odnosno krhki testovi. Pogledajte primer na slajdu: programer odluči da refaktoriše kod i preimenuje internu metodu `baza.nadji()` u `baza.pretraziPoId()`. Iako aplikacija radi identično kao i pre, naš test pada jer je bio čvrsto vezan za stari naziv metode. Umesto da nam testovi pomognu prilikom refaktorisanja, oni nam postaju teret jer moramo stalno da ih prepravljamo.
[Slajd 32]
Kako da izbegnemo ovu zamku? Pridržavajte se tri jednostavna pravila. Prvo: kad god možete, dajte prednost verifikaciji stanja nad verifikacijom ponašanja. Drugo: mock-ujte samo prave granice sistema — eksterne API-je, slanje mejlova ili fajl sistem. Treće: nikada nemojte mock-ovati čiste podatke kao što su entiteti i Value objekti.
[Slajd 33]
Da zaključimo današnje predavanje. Dizajn za testabilnost je preduslov za održiv softver. Testni dvojnici su moćan alat, ali zahtevaju disciplinu: koristite Stub i Fake za stabilne testove zasnovane na podacima, a Mock i Spy čuvajte samo za proveru ključnih interakcija. Zapamtite najvažniju poruku danas: ***vaš test treba da bude živa dokumentacija ponašanja sistema, a ne čuvar njegove unutrašnje strukture***. Hvala na pažnji!
